**MapReduceBasicProject**

**package** it.polito.bigdata.hadoop;

**/\*Basic MapReduce Project -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));

}

}

}

**package** it.polito.bigdata.hadoop;

**/\*Basic MapReduce Project -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 1: Word count problem**

♣ Input: (unstructured) textual file

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

package it.polito.bigdata.hadoop.**exercise1**;

**/\*\* \* 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 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 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

**package** it.polito.bigdata.hadoop.**exercise2**;

**/\*\* \* Exercise 2 - 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));

}

}

}

package it.polito.bigdata.hadoop.**exercise2**;

**/\*\* \* Exercise 2 - 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

▪ Supposed to set threshold = 50 μg/m3

▪ Select only the sensors that are associated at least one time with a PM10 above the threshold

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)

**package** it.polito.bigdata.hadoop.**exercise3;**

**/\*\* \* Exercise 3 - Mapper \*/**

class MapperBigData extends Mapper<

Text, // Input key type

Text, // Input value type

Text, // Output key type

ntWritable>{// 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));

} }}

**package** it.polito.bigdata.hadoop.**exercise3**;

**/\*\* \* 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 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

package it.polito.bigdata.hadoop.**exercise5**;

/\*\* \* 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));

    }

}

**\*\*\*\*\*\*\*exercise 5 with Combiner**

package it.polito.bigdata.hadoop.exercise5withcombiner;

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);

    }

}

/\*\* \* 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);

    }

}

/\*\* \* 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);

    }

}

**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

package it.polito.bigdata.hadoop.**exercise6**;

/\*\* \* 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])));

    }

}

package it.polito.bigdata.hadoop.exercise6;

/\*\* \* 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));

    }

}

**\*\*\*\*\*\*\*exercise 6 with 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]));

    }

}

/\*\* \* Combiner \*/

class CombinerBigData 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]);

            }

        }

        // Emit the pair (sensor\_id, max reading value\_min reading value)

        // value is composed of two parts: max and min.

        context.write(new Text(key), new Text(max + "\_" + min));

    }

}

/\*\* \* 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));

    }

}

**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*

package it.polito.bigdata.hadoop.**exercise9**;

***/\*\* \* 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()));

        }

    }

}

package it.polito.bigdata.hadoop.**exercise9**;

***/\*\* \* 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));

    }

}

**Exercise 11: 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

♣ Suppose the number of sensors is equal to 2 and their ids are s1 and s2 (تفاوت با تمرین5)

package it.polito.bigdata.hadoop.**exercise11**;

/\*\* \* Average Mapper \*/

class MapperBigData extends

        Mapper<LongWritable, // Input key type

                Text, // Input value type

                Text, // Output key type

                SumCount> {// Output value type

    HashMap<String, SumCount> statistics;

    protected void setup(Context context) {

        statistics = new HashMap<String, SumCount>();

    }

    protected void map(LongWritable key, // Input key type

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        SumCount currentStat;

        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.split(",");

        String sensorId = fields[0];

        float measure = Float.parseFloat(fields[2]);

        currentStat = statistics.get(sensorId);

        if (currentStat == null) {

            currentStat = new SumCount();

            currentStat.setCount(1);

            currentStat.setSum(measure);

            statistics.put(new String(sensorId), currentStat);

        } else {

            currentStat.setCount(currentStat.getCount() + 1);

            currentStat.setSum(currentStat.getSum() + measure);

            statistics.put(new String(sensorId), currentStat);

        }    }

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

        for (Entry<String, SumCount> pair : statistics.entrySet()) {

            context.write(new Text(pair.getKey()), pair.getValue());

        }

    }

}

/\*\* \* WordCount Reducer \*/

class ReducerBigData extends Reducer<

                Text,           // Input key type

                SumCount,  // Input value type

                Text,           // Output key type

                SumCount> {  // Output value type

    @Override

    protected void reduce(

        Text key, // Input key type

        Iterable<SumCount> values, // Input value type

        Context context) throws IOException, InterruptedException {

        int count=0;

        float sum=0;

                // Iterate over the set of values and sum them.

        // Count also the number of values

        for (SumCount value : values) {

            sum=sum+value.getSum();

            count=count+value.getCount();

        }

        SumCount sumCountPerSensor= new SumCount();

        sumCountPerSensor.setCount(count);

        sumCountPerSensor.setSum(sum);

        // Emits pair (sensor\_id, sum-count = average)

        context.write(new Text(key), sumCountPerSensor);

    }

}

public class SumCount 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;

    }

}

**Exercise 10: Total count**

♣ 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: total number of records

package it.polito.bigdata.hadoop.exercise10;

/\*\* \* Ex. 10 Mapper \*/

class MapperBigData extends

        Mapper<LongWritable, // Input key type

                Text, // Input value type

                NullWritable, // Output key type

                NullWritable> {// Output value type

    protected void map(LongWritable key, // Input key type

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        context.getCounter(MY\_COUNTERS.TOTAL\_RECORDS).increment(1);

    } }

**Exercise 12: Select outliers**

♣ 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)

package it.polito.bigdata.hadoop.exercise12;

/\*\* \* Mapper \*/

class MapperBigData extends

        Mapper<Text, // Input key type

                Text, // Input value type

                Text, // Output key type

                FloatWritable> {// Output value type

    float threshold;

    protected void setup(Context context) {

        // I retrieve the value of the threshold only one time for each mapper

        threshold =

            Float.parseFloat(

                    context.getConfiguration().get("maxThreshold"));

    }

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

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        float measure;

        measure = Float.parseFloat(value.toString());

        // Filter the reading based on the value of threshold

        if (measure < threshold) {

            context.write(new Text(key),

                    new FloatWritable(measure));

        }

    }

}

**Exercise 8:Total Income**

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 - 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())));

    }

}

package it.polito.bigdata.hadoop.exercise8;

/\*\* \* 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())));

    }

}

package it.polito.bigdata.hadoop.exercise8;

/\*\* \* 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));

    }

}

package it.polito.bigdata.hadoop.exercise8;

/\*\* \* 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));

    }

}

\*\*\*\*\*\* **Exercise 8: single job**

/\*\* \* 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));

    }

}

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 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

▪ Supposed to set threshold = 50 μg/m3

▪ Report only the zones with at least one date with PM10 above the threshold

\* 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 13: Top 1 most profitable date**

♣ 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 \*/

class MapperBigData extends Mapper<Text, // Input key type

        Text, // Input value type

        NullWritable, // Output key type

        DateIncome> {// Output value type

    private DateIncome top1;

    protected void setup(Context context) {

        // for each mapper, top1 is used to store the information about the top1

        // date-income of the subset of lines analyzed by the mapper

        top1 = new DateIncome();

        top1.setIncome(Float.MIN\_VALUE);

        top1.setDate(null);

    }

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

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        String date = new String(key.toString());

        float dailyIncome = Float.parseFloat(value.toString());

        // Check if the current income is the top 1 income among the ones

        // analyzed by this

        // mapper. In case of tie (same income value) the earliest date is

        // considered.

        if ( dailyIncome > top1.getIncome() ||

            (dailyIncome == top1.getIncome() && date.compareTo(top1.getDate()) < 0)) {

            // The current line is the current top 1 income value. Store date

            // and income in top1

            top1 = new DateIncome();

            top1.setDate(date);

            top1.setIncome(dailyIncome);

        }

/\*\* \* Reducer \*/

class ReducerBigData extends Reducer<NullWritable, // Input key type

        DateIncome, // Input value type

        Text, // Output key type

        FloatWritable> { // Output value type

    // The reduce method is called only once in this approach

    // All the key-value pairs emitted by the mappers have the

    // same key (NullWritable.get())

    @Override

    protected void reduce(NullWritable key, // Input key type

            Iterable<DateIncome> values, // Input value type

            Context context) throws IOException, InterruptedException {

        String date;

        float dailyIncome;

        DateIncome globalTop1 = new DateIncome();

        globalTop1.setIncome(Float.MIN\_VALUE);

        globalTop1.setDate(null);

        // Iterate over the set of values and select the top 1 income and

        // the related date

        for (DateIncome value : values) {

            date = value.getDate();

            dailyIncome = value.getIncome();

            if ( dailyIncome > globalTop1.getIncome() ||

                (dailyIncome == globalTop1.getIncome() && date.compareTo(globalTop1.getDate()) < 0)) {

                // The current line is the current top 1 income value. Store

                // date and income in globalTop1

                globalTop1 = new DateIncome();

                globalTop1.setDate(date);

                globalTop1.setIncome(dailyIncome);

            }

        }

        // Emit pair (date, income) associated with top 1 income

        context.write(new Text(globalTop1.getDate()), new FloatWritable(globalTop1.getIncome()));

    }

}

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

    private String date;

    private float income;

    public String getDate() { مهم نیست برای امتحان

        return date;

    }

    public void setDate(String dateValue) { مهم نیست برای امتحان

        date = dateValue;

    }

    public float getIncome() { مهم نیست برای امتحان

        return income;

    }

    public void setIncome(float incomeValue) { مهم نیست برای امتحان

        income = incomeValue;

    }

    @Override

    public void readFields(DataInput in) throws IOException {

        income = in.readFloat();

        date = in.readUTF();

    } مهم نیست برای امتحان

    @Override

    public void write(DataOutput out) throws IOException {

        out.writeFloat(income);

        out.writeUTF(date);

    } مهم نیست برای امتحان

    public String toString() { این قسمت براش مهمه که تو امتحان نوشته بشه

        String formattedString = new String("date:" + date + " income:" + income);

        return formattedString;

    }

}

**Exercise 13 Bis: Top 2 most profitable dates**

♣ 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 2 most profitable dates ▪ In case of tie, select the first 2 dates among the ones associated with the highest income

public class DateValue {

    String date;

    float value;

}

/\*\* \* Mapper \*/

class MapperBigData extends

        Mapper<Text, // Input key type

                Text, // Input value type

                NullWritable, // Output key type

                Text> {// Output value type

    DateValue top1;

    DateValue top2;

protected void setup(Context context) {

        top1 = null;

        top2 = null;

    }

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

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        String date = new String(key.toString());

        float dailyIncome = Float.parseFloat(value.toString());

        if (top1 == null || top1.value < dailyIncome || (top1.value == dailyIncome && date.compareTo(top1.date) < 0)) {

            top2 = top1;

            top1 = new DateValue();

            top1.date = date;

            top1.value = dailyIncome;

        } else {

            if (top2 == null || top2.value < dailyIncome

                    || (top2.value == dailyIncome && date.compareTo(top2.date) < 0)) {

                top2 = new DateValue();

                top2.date = date;

                top2.value = dailyIncome;

            }

        }

    }

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

        context.write(NullWritable.get(), new Text(top1.date + "\_" + top1.value));

        context.write(NullWritable.get(), new Text(top2.date + "\_" + top2.value));

    }

}

/\*\* \* Reducer \*/

class ReducerBigData extends Reducer<

                NullWritable,           // Input key type

                Text,  // Input value type

                Text,           // Output key type

                FloatWritable> {  // Output value type

    // The reduce method is called only once in this approach

    // All the key-value pairs emitted by the mappers as the

    // same key (NullWritable.get())

    @Override

    protected void reduce(

        NullWritable key, // Input key type

        Iterable<Text> values, // Input value type

        Context context) throws IOException, InterruptedException {

        float dailyIncome;

        String date;

        DateValue top1;

        DateValue top2;

        top1=null;

        top2=null;

        // Iterate over the set of values and select the top 2

        for (Text value : values) {

            String[] record=value.toString().split("\_");

            date=record[0];

            dailyIncome=Float.parseFloat(record[1]);

            if (top1==null || top1.value<dailyIncome)

            {

                top2=top1;

                top1=new DateValue();

                top1.date=new String(date);

                top1.value=dailyIncome;

            }

            else

                {

                    if (top2==null || top2.value<dailyIncome)

                    {

                        top2=new DateValue();

                        top2.date=new String(key.toString());

                        top2.value=dailyIncome;

                    }

                }

        }

        // Emit pair (date, value) top1

        // Emit pair (date, value) top2

        context.write(new Text(top1.date), new FloatWritable(top1.value));

        context.write(new Text(top2.date), new FloatWritable(top2.value));

    }

}

**Exercise 14: Dictionary**

♣ Input: a collection of news (textual files)

♣ Output: ▪ List of distinct words occurring in the collection

/\*\* \* Exercise 14 - Combiner \*/

class CombinerBigData extends Reducer<Text, // Input key type

        NullWritable, // Input value type

        Text, // Output key type

        NullWritable> { // Output value type

    @Override

    protected void reduce(Text key, // Input key type

            Iterable<NullWritable> values, // Input value type

            Context context) throws IOException, InterruptedException {

        context.write(key, NullWritable.get());

    }

}

/\*\* \* Exercise 14 \*/

class MapperBigData extends Mapper<

                    LongWritable, // Input key type

                    Text,         // Input value type

                    Text,         // Output key type

                    NullWritable> {// 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, null)

                context.write(new Text(cleanedWord), NullWritable.get());

            }

    }}

/\*\* \* Exercise 14 - Reducer \*/

class ReducerBigData extends Reducer<Text, // Input key type

        NullWritable, // Input value type

        Text, // Output key type

        NullWritable> { // Output value type

    @Override

    protected void reduce(Text key, // Input key type

            Iterable<NullWritable> values, // Input value type

            Context context) throws IOException, InterruptedException {

        context.write(key, NullWritable.get());

    }

}

**Exercise 15: Dictionary– Mapping word - integer**

♣ 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)

/\*\*  \* Exercise 15 - Combiner \*/

class CombinerBigData extends Reducer<

                Text,           // Input key type

                NullWritable,    // Input value type

                Text,           // Output key type

                NullWritable> {  // Output value type

    @Override

    protected void reduce(

        Text key, // Input key type

        Iterable<NullWritable> values, // Input value type

        Context context) throws IOException, InterruptedException {

        context.write(key, NullWritable.get());

    }

}

/\*\* \* Exercise 15 \*/

class MapperBigData extends Mapper<

                    LongWritable, // Input key type

                    Text,         // Input value type

                    Text,         // Output key type

                    NullWritable> {// 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, null)

                context.write(new Text(cleanedWord), NullWritable.get());

            }

    }

}

/\*\* \* Exercise 15 - Reducer \*/

class ReducerBigData extends

        Reducer<Text, // Input key type

                NullWritable, // Input value type

                Text, // Output key type

                IntWritable> { // Output value type

    int wordId;

    protected void setup(Context context) {

        // Initialize the variable that is used to remember how many words have

        // been already mapped to an integer (i.e., it stores also the last

        // integer value mapped with a word)

        wordId = 0;

    }

    @Override

    protected void reduce(Text key, // Input key type

            Iterable<NullWritable> values, // Input value type

            Context context) throws IOException, InterruptedException {

        // Emit the current word associated with the next available integer

        wordId = wordId + 1;

        context.write(key, new IntWritable(wordId));

    }

}

**Exercise 21: Stopword elimination problem**

♣ Input: ▪ A large textual file containing one sentence per line

▪ A small file containing a set of stopwords ▪ 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

/\*\*  \* Exercise 21 - Mapper \*/

class MapperBigData extends Mapper<

                    LongWritable, // Input key type

                    Text,         // Input value type

                    NullWritable,         // Output key type

                    Text> {// Output value type

    private ArrayList<String> stopWords;

    protected void setup(Context context) throws IOException, InterruptedException

    {

        String nextLine;

        stopWords=new ArrayList<String>();

        // Open the stopword file (that is shared by means of the distributed

        // cache mechanism)

        URI[] urisCachedFiles = context.getCacheFiles();

        // This application has one single single cached file.

        // Its path is stored in urisCachedFiles[0]

        BufferedReader fileStopWords = new BufferedReader(new

                FileReader(new File(urisCachedFiles[0].getPath())));

        // Each line of the file contains one stopword

        // The stopwords are stored in the stopWords list

        while ((nextLine = fileStopWords.readLine()) != null) {

            stopWords.add(nextLine);

        }

            fileStopWords.close();

    }

      protected void map(

            LongWritable key,   // Input key type

            Text value,         // Input value type

            Context context) throws IOException, InterruptedException {

            boolean stopword;

            // 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+");

            // Remove stopwords from the current sentence

            String sentenceWithoutStopwords=new String("");

            // Iterate over the set of words

            for(String word : words) {

                // if the current word is in the stopWords list it means it is a stopword

                if (stopWords.contains(word)==true)

                    stopword=true;

                else

                    stopword=false;

                // If the current word is a stopword do not consider it

                // Otherwise attach it at the end of sentenceWithoutStopwords

                if (stopword==false)

                {

                    sentenceWithoutStopwords=sentenceWithoutStopwords.concat(word+" ");

                }

            }

                // emit the pair (null, sentenceWithoutStopwords)

            context.write(NullWritable.get(),

                    new Text(sentenceWithoutStopwords));

    }}

**Exercise 17: Select maximum temperature for each date**

♣ 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)

/\*\* \* Mapper first data format \*/

class MapperType1BigData 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]= sensor id

        // fields[1]= date

        // fields[2]= hour:minute

        // fields[3]= temperature

        String[] fields = record.split(",");

        String date = fields[1];

        float temperature = Float.parseFloat(fields[3]);

        context.write(new Text(date), new FloatWritable(temperature));

    }

}

/\*\* \* Mapper second data format \*/

class MapperType2BigData 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]= date

        // fields[1]= hour:minute

        // fields[2]= temperature

        // fields[3]= sensor id

        String[] fields = record.split(",");

        String date = fields[0];

        float temperature = Float.parseFloat(fields[2]);

        context.write(new Text(date), new FloatWritable(temperature));

    }

}

/\*\* \* 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 {

        float max=Float.MIN\_VALUE;

        // Iterate over the set of values and find the maximum value

        for (FloatWritable value : values) {

            if (value.get()>max)

                max=value.get();

        }

        // Emit pair (date, maximum temperature)

        context.write(new Text(key), new FloatWritable(max));

    }

}

**Exercise 20:**

Split the readings of a set of sensors based on the value of the measurement

♣ 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

▪ 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

/\*\* \* Mapper of a map-only job \*  \*/

class MapperBigData extends Mapper<

                    LongWritable, // Input key type

                    Text,         // Input value type

                    Text,         // Output key type

                    NullWritable> {// Output value type

    // Define a MultiOutputs object

    private MultipleOutputs<Text, NullWritable> mos = null;

    protected void setup(Context context)

    {

        // Create a new MultiOuputs using the context object

        mos = new MultipleOutputs<Text, NullWritable>(context);

    }

    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]= sensor id

            // fields[1]= date

            // fields[2]= hour:minute

            // fields[3]= temperature

            String[] fields = record.split(",");

            float temperature=Float.parseFloat(fields[3]);

            if (temperature>30.0)

                mos.write("hightemp", value, NullWritable.get());

            else

                mos.write("normaltemp", value, NullWritable.get());

   }

    protected void cleanup(Context context) throws IOException, InterruptedException

    {

        // Close the MultiOutputs

        // If you do not close the MultiOutputs object the content of the output

        // files will not be correct

        mos.close();

    }

}

**Exercise 23:**

Potential friends of a specific user

♣ 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

\*\*

 \* Exercise 23 - Mapper

 \*/

class MapperBigData extends Mapper<LongWritable, // Input key type

        Text, // Input value type

        Text, // Output key type

        Text> {// Output value type

    String specifiedUser;

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

        // Retrieve the information about the user of interest

        specifiedUser = context.getConfiguration().get("username");

    }

    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)

        // Do not emit pair with key=user of interest. It is not useful

        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]));

    }

}

/\*\* \* Exercise 23 - Reducer \*/

class ReducerBigData extends Reducer<Text, // Input key type

        Text, // Input value type

        Text, // Output key type

        NullWritable> { // Output value type

    HashSet<String> finalListPotentialFriends;

    String specifiedUser;

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

        // Retrieve the information about the user of interest

        specifiedUser = context.getConfiguration().get("username");

        // Instantiate the local variable that is used to store the complete

        // list of potential friends

        finalListPotentialFriends = new HashSet<String>();

    }

    @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>();

        // Key contains one username.

        // If values contains the specified user it means that the specified

        // user and the other users in values have user "key" in common.

        // Hence, the users in values are potential friends

        containsSpecifiedUser = false;

        for (Text value : values) {

            if (specifiedUser.compareTo(value.toString()) == 0)

                containsSpecifiedUser = true;

            else {

                // Store the list of users for a potential "second iteration"

                partialListOfPotentialFriends.add(value.toString());

            }

        }

        // If containsSpecifiedUser is true it means that

        // partialListOfPotentialFriends

        // contains potential friends of the specified user

        // It is useful if and only if partialListOfPotentialFriends is not

        // empty (i.e., if values

        // contains the selected user and also another one)

        if (containsSpecifiedUser == true && partialListOfPotentialFriends.size() > 0) {

            // Extract the list of potential users for

            // partialListOfPotentialFriends

            for (String user : partialListOfPotentialFriends) {

                // If the user is new then it is inserted in the set

                // Otherwise, if it is already in the set, it is ignored

                finalListPotentialFriends.add(user);

            }

        }

    }

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

        // Concatenate the users stored in finalListPotentialFriends

        // to generate the final result

        String globalPotFriends = new String("");

        for (String potFriend : finalListPotentialFriends) {

            globalPotFriends = globalPotFriends.concat(potFriend + " ");

        }

        context.write(new Text(globalPotFriends), NullWritable.get());

    }

}

**Exercise 23:**

/\*\* \* Exercise 22 - Mapper \*/

class MapperBigData extends Mapper<LongWritable, // Input key type

        Text, // Input value type

        NullWritable, // Output key type

        Text> {// Output value type

    String specifiedUser;

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

        // Retrieve the information about the user of interest

        specifiedUser = context.getConfiguration().get("username");

    }

    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 users is specifiedUser

        if (specifiedUser.compareTo(users[0]) == 0) {

            // emit the pair (null, users[1])

            context.write(NullWritable.get(), new Text(users[1]));

        }

        if (specifiedUser.compareTo(users[1]) == 0) {

            // emit the pair (null, users[0])

            context.write(NullWritable.get(), new Text(users[0]));

        }

    }

}

/\*\*

 \* Exercise 23 - Mapper Job 2

 \*/

class MapperBigData2 extends Mapper<LongWritable, // Input key type

        Text, // Input value type

        NullWritable, // Output key type

        Text> {// Output value type

    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("username");

        // Store in the ArraList friends the list of friends available in the

        // shared file

        friends = new ArrayList<String>();

        URI[] CachedFiles = context.getCacheFiles();

        // This application has one single cached file.

        // new Path(CachedFiles[0].getPath()).getName() is the name of the

        // shared file (i.e., part-r-00000 in this application).

        BufferedReader fileFriends =

                new BufferedReader(new FileReader(new File(new Path(CachedFiles[0].getPath()).getName())));

        /\* We can also simply use the following line of code instead of the previous one since we know

          apriori the name of the shared 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])

            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])

            context.write(NullWritable.get(), new Text(users[0]));

        }

    }

}

/\*\*

 \* Exercise 22 - Reducer

 \*/

class ReducerBigData extends Reducer<NullWritable, // Input key type

        Text, // Input value type

        Text, // Output key type

        NullWritable> { // Output value type

    @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());

        }

    }

}

/\*\*

 \* Exercise 22 - Reducer Job 2

 \*/

class ReducerBigData2 extends Reducer<NullWritable, // Input key type

        Text, // Input value type

        Text, // Output key type

        NullWritable> { // Output value type

    @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 + " ");

        }

        // Emit the list of potential friends (in one single line)

        context.write(new Text(listOfPotFriends), NullWritable.get());

    }

}

**Exercise 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= and YearOfBirth= -> 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

/\*\* \* Exercise 27 - Mapper \*/

class MapperBigData extends

        Mapper<LongWritable, // Input key type

                Text, // Input value type

                NullWritable, // Output key type

                Text> {// Output value type

    private ArrayList<String> rules;

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

        String nextLine;

        rules = new ArrayList<String>();

        // Open the business rules file (that is shared by means of the

        // distributed

        // cache mechanism)

        URI[] CachedFiles = context.getCacheFiles();

        // This application has one single single cached file.

        // Its path is URIsCachedFiles[0]

        BufferedReader rulesFile =

                new BufferedReader(

                    new FileReader(new File(CachedFiles[0].getPath())));

        // Each line of the file contains one rule

        while ((nextLine = rulesFile.readLine()) != null) {

            rules.add(nextLine);

        }

        rulesFile.close();

    }

    private String applyBusinessRule(String gender, String year) {

        String category = new String("Unknown");

        // Iterate over the rules

        for (String rule : rules) {

            // Gender=<value> and DateOfBirth=<value> -> Category

            String[] ruleParts = rule.split(" ");

            // ruleParts[0] = Gender=<value>

            // ruleParts[2] = DateOfBirth=<value>

            // ruleParts[4] = category

            // Check if the current rule is satisfied by the current user

            if (ruleParts[0].compareTo("Gender=" + gender) == 0 && ruleParts[2].compareTo("YearOfBirth=" + year) == 0)

                category = ruleParts[4];

        }

        return category;

    }

    protected void map(LongWritable key, // Input key type

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        String category;

        // Split each record in fields

        // UserId,Name,Surname,Gender,YearOfBirth,City,Education

        String[] fields = value.toString().split(",");

        category = applyBusinessRule(fields[3], fields[4]);

        // emit the pair (null, record+category)

        context.write(NullWritable.get(),

                new Text(value.toString() + "," + category));

    }

}

**Exercise 28: Mapping Question-Answer(s)**

♣ 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

/\*\* \* Mapper first data format \*/

class MapperType1BigData 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 {

            // Record format

            // QuestionId,Timestamp,TextOfTheQuestion

            String[] fields=value.toString().split(",");

            String questionId=fields[0];

            String questionText=fields[2];

            // Key = questionId

            // Value = Q:+questionId,questionText

            // Q: is used to specify that this pair has been emitted by

            // analyzing a question

            context.write(new Text(questionId), new Text("Q:"+questionId+","+questionText));

    }

}

/\*\*

 \* Mapper second data format

 \*/

class MapperType2BigData 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 {

            // Record format

            // AnswerId,QuestionId,Timestamp,TextOfTheAnswer

            String[] fields=value.toString().split(",");

            String answerId=fields[0];

            String answerText=fields[3];

            String questionId=fields[1];

            // Key = questionId

            // Value = A:+answerId,answerText

            // A: is used to specify that this pair has been emitted by

            // analyzing an answer

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

    }

/\*\*

 \* WordCount Reducer

 \*/

class ReducerBigData extends Reducer<

                Text,           // Input key type

                Text,  // Input value type

                NullWritable,           // 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 record;

        ArrayList<String> answers=new ArrayList<String>();

        String question=null;

        // Iterate over the set of values and store the answer records in

        // answers and the question record in question

        for (Text value : values) {

            String table\_record=value.toString();

            if (table\_record.startsWith("Q:")==true)

            {   // This is the question record

                record=table\_record.replaceFirst("Q:", "");

                question=record;

            }

            else

            {   // This is an answer record

                record=table\_record.replaceFirst("A:", "");

                answers.add(record);

            }

        }

        // Emit one pair (question, answer) for each answer

        for (String answer:answers)

        {

            context.write(NullWritable.get(), new Text(question+","+answer));

        }

    }

}

**Exercise 29: User selection**

♣ 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 68

♣ Output:

▪ One record for each user that likes both Commedia and Adventure movies ▪ Each output record contains only Gender and YearOfBirth of a selected user

▪ Gender,YearOfBirth ▪ Duplicate pairs must not be removed

/\*\* \* Mapper first data format \*/

class MapperType1BigData 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 {

            // Record format - Users table

            // UserId,Name,Surname,Gender,YearOfBirth,City,Education

            String[] fields=value.toString().split(",");

            String userId=fields[0];

            String gender=fields[3];

            String yearOfBirth=fields[4];

            // Key = userId

            // Value = U:+gender,yearOfBirth

            // U: is used to specify that this pair has been emitted by

            // analyzing the Users table

            context.write(

                    new Text(userId),

                    new Text("U:"+gender+","+yearOfBirth));

    }

  }

\*\* \* Mapper second data format \*/

class MapperType2BigData 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 {

            // Record format - Likes table

            // UserId,MovieGenre

            String[] fields=value.toString().split(",");

            String userId=fields[0];

            String genre=fields[1];

            // Key = userId

            // Value = L

            // L: is used to specify that this pair has been emitted by

            // analyzing the likes file

            // Emit the pair if and only if the genre is Commedia or Adventure

            if (genre.compareTo("Commedia")==0

                    || genre.compareTo("Adventure")==0)

            {

                context.write(new Text(userId), new Text("L"));

            }

    }

}

/\*\* \* WordCount Reducer \*/

class ReducerBigData extends

        Reducer<Text, // Input key type

                Text, // Input value type

                NullWritable, // 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 {

        int numElements;

        String userData = null;

        // Iterate over the set of values and check if

        // 1) there are three elements (one related do the Users table and two

        // related to the Likes table

        // 2) store the information about the "profile/user" element

        numElements = 0;

        for (Text value : values) {

            String table\_record = value.toString();

            numElements++;

            if (table\_record.startsWith("U") == true) {

                // This is the user data record

                userData = table\_record.replaceFirst("U:", "");

            }

        }

      // Emit a pair (null,user data) if the number of elements is equal to 3

        // (2 likes and 1 user data record)

        if (numElements == 3) {

            context.write(NullWritable.get(), new Text(userData));

        }

    }

}

**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”/\*\*

 \* 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 24: Compute the list of friends for each user**

♣ 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

/\*\* \* Exercise 24 – 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 {

            // 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]));

    }

}

/\*\* \* Exercise 24 - 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 listOfFriends=new String("");

        // Key contains one userame.

        // Iterate over the set of values and concatenate them to build the

        // list of friend of the username specified in key.

        for (Text value : values) {

            listOfFriends=listOfFriends.concat(value.toString()+" ");

        }

        context.write(new Text(key+":"), new Text(listOfFriends));

    }

}

**Exercise 25: Compute the list of potential friends for each user**

♣ 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

/\*\* \* Exercise 25 - 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 {

            // 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]));

    }

}

/\*\*

 \* Exercise 25 - Mapper

 \*/

class MapperBigDataFilter 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 {

            // 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));

            }

    }

}

\*\*

 \* Exercise 25 - 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 {

        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));

        }

    }

}

/\*\* \* Exercise 25 - Reducer \*/

class ReducerBigDataFilter 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 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));

    }

}

**Exercise 26: Word (string) to integer conversion**

♣ 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

/\*\*  \* Exercise 26 – Mapper  \*/

class MapperBigData extends

        Mapper<LongWritable, // Input key type

                Text, // Input value type

                NullWritable, // Output key type

                Text> {// Output value type

    private HashMap<String, Integer> dictionary;

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

        String line;

        String word;

        Integer intValue;

        dictionary = new HashMap<String, Integer>();

        // Open the dictionary file (that is shared by means of the distributed

        // cache mechanism)

        URI[] CachedFiles = context.getCacheFiles();

        // This application has one single single cached file.

        // Its path is URIsCachedFiles[0]

        BufferedReader fileStopWords = new BufferedReader(new FileReader(new File(CachedFiles[0].getPath())));

        // Each line of the file contains one mapping

        // word integer

        // The mapping is stored in the dictionary HashMap variable

        while ((line = fileStopWords.readLine()) != null) {

            // record[0] = integer value associated with the word

            // record[1] = word

            String[] record = line.split("\t");

            intValue = Integer.parseInt(record[0]);

            word = record[1];

            dictionary.put(word, intValue);

        }

        fileStopWords.close();

    }

    protected void map(LongWritable key, // Input key type

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        String convertedString;

        Integer intValue;

        // 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+");

        // Convert words to integers

        convertedString = new String("");

        // Iterate over the set of words

        for (String word : words) {

            // Retrieve the integer associated with the current word

            intValue = dictionary.get(word.toUpperCase());

            convertedString = convertedString.concat(intValue + " ");

        }

        // emit the pair (null, sentenceWithoutStopwords)

        context.write(NullWritable.get(), new Text(convertedString));

    }

}

**Exercise 22:Friends of a specific user**

♣ 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

/\*\*

/ \* Exercise 22 - Mapper \*/

class MapperBigData extends Mapper<

                    LongWritable,         // Input key type

                    Text,         // Input value type

                    NullWritable,         // Output key type

                    Text> {// Output value type

    String specifiedUser;

    protected void setup(Context context) {

        // Retrieve the value of the user of interest

        specifiedUser=context.getConfiguration().get("username");

    }

    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 users is specifiedUser

            if (specifiedUser.compareTo(users[0])==0)

            {

                // emit the pair (null, users[1])

                context.write(NullWritable.get(), new Text(users[1]));

            }

            if (specifiedUser.compareTo(users[1])==0)

            {

                // emit the pair (null, users[0])

                context.write(NullWritable.get(), new Text(users[0]));

            }

     }

}

/\*\* \* Exercise 22 - Reducer \*/

class ReducerBigData extends Reducer<

                NullWritable,           // Input key type

                Text,    // Input value type

                Text,           // Output key type

                NullWritable> {  // Output value type

    @Override

    protected void reduce(

        NullWritable key,       // Input key type

        Iterable<Text> values,  // Input value type

        Context context) throws IOException, InterruptedException {

        String listOfFriends=new String("");

        // Iterate over the set of values and concatenate them

        for (Text value : values) {

            listOfFriends=listOfFriends.concat(value.toString()+" ");

        }

        context.write(new Text(listOfFriends), NullWritable.get());

    }

}

**SolutionExamexample1**

**Exercise1:**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured

implements Tool {

  @Override

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

    Path inputPath;

    Path outputDir;

    int numReducers;

  int exitCode;

  // Parse the parameters

    inputPath = new Path(args[0]);

    outputDir = new Path(args[1]);

    numReducers = Integer.parseInt(args[2]);

    Configuration conf = this.getConf();

    // Define a new job

    Job job = Job.getInstance(conf);

    // Assign a name to the job

    job.setJobName("Exercise #1 - Exam 2019/02/15");

    // 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(TextInputFormat.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(IntWritable.class);

    // Set reduce class

    job.setReducerClass(ReducerBigData.class);

    // Set reduce output key and value classes

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(NullWritable.class);

    // The number of instances of the reducer can be any value >=1

    job.setNumReduceTasks(numReducers);

    // Execute the job and wait for completion

    if (job.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 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 record

        // POI\_ID,latitude,longitude,city,country,category,subcategory

        // Example: P101,45.0621644,7.578633,Turin,Italy,shop,shoes

        String[] fields = value.toString().split(",");

        String city = fields[3];

        String country = fields[4];

        String category = fields[5];

        String subcategory = fields[6];

        // Select only POIs of the Italian cities

        // and only "tourism" POIs

        if (country.equals("Italy") == true && category.equals("tourism")) {

            // Check the subcategory

            if (subcategory.contentEquals("museum")) {

                // emit pair (city, 1)

                // -- One new tourism POI

                // -- One new museum POI

                context.write(new Text(city), new IntWritable(1));

            } else {

                // emit pair (city, 0))

                // -- One new tourism POI

                // -- This POI is not a museum POI

                context.write(new Text(city), new IntWritable(0));

            }

        }

    }

}

/\*\* \* Exercise 1 - Reducer \*/

class ReducerBigData extends Reducer<Text, // Input key type

        IntWritable, // Input value type

        Text, // Output key type

        NullWritable> { // Output value type

    @Override

    protected void reduce(Text key, // Input key type

            Iterable<IntWritable> values, // Input value type

            Context context) throws IOException, InterruptedException {

        // Iterate over the set of values and compute

        // - total number of "turism" POIs = number of input values

        // - total number of "museum" POIs = number of ones = sum of values

        int numTurismPOIs = 0;

        int numMuseumPOIs = 0;

        for (IntWritable value : values) {

            numTurismPOIs++;

            numMuseumPOIs = numMuseumPOIs + value.get();

        }

        // Emit the city only if

        // - total number of "turism" POIs > 1000

        // - total number of "museum" POIs >= 20

        if (numTurismPOIs > 1000 && numMuseumPOIs >= 20)

            context.write(new Text(key), NullWritable.get());

    }

}

**SolutionExamExample2**

**Exercise1**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured

implements Tool {

  @Override

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

    Path inputPath;

    Path outputDir;

    int numberOfReducers;

  int exitCode;

  // Parse the parameters

    numberOfReducers = Integer.parseInt(args[0]);

    inputPath = new Path(args[1]);

    outputDir = new Path(args[2]);

    Configuration conf = this.getConf();

    // Define a new job

    Job job = Job.getInstance(conf);

    // Assign a name to the job

    job.setJobName("Exercise #1 - Exam 2017/07/14");

    // 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(TextInputFormat.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(Text.class);

    // Set reduce class

    job.setReducerClass(ReducerBigData.class);

    // Set reduce output key and value classes

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(NullWritable.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;

    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 1 - 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 {

        // Split data

        // date\_reading,city,country,max\_temperature,min\_temperature

        // 2016/07/20,Turin,Italy,32.5,26.0

        String[] fields = value.toString().split(",");

        String city = fields[1];

        Double maxTemp = Double.parseDouble(fields[3]);

        Double minTemp = Double.parseDouble(fields[4]);

        if (maxTemp.doubleValue() > 35) {

            // emit the pair (city, "g") -> g = greater than 35

            context.write(new Text(city), new Text("g"));

        }

        if (minTemp.doubleValue() < -20) {

            // emit the pair (city, "l") -> l = less than -20

            context.write(new Text(city), new Text("l"));

        }

    }

}

/\*\* \* Exercise 1 - Reducer \*/

class ReducerBigData extends

        Reducer<Text, // Input key type

                Text, // Input value type

                Text, // Output key type

                NullWritable> { // Output value type

    @Override

    protected void reduce(Text key, // Input key type

            Iterable<Text> values, // Input value type

            Context context) throws IOException, InterruptedException {

        boolean greaterThan35 = false;

        boolean lessThanMinus20 = false;

        // Iterate over the set of values and check if there is at least

        // one "g" and at least on "l"

        for (Text flags : values) {

            if (flags.toString().compareTo("g") == 0) {

                greaterThan35 = true;

            }

            if (flags.toString().compareTo("l") == 0) {

                lessThanMinus20 = true;

            }

        }

        if (greaterThan35 == true && lessThanMinus20 == true) {

            context.write(key, NullWritable.get());

        }

    }

}

**SolutionExamExample3**

**Exercise1**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured

implements Tool {

  @Override

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

    Path inputPath;

    Path outputDir;

    int numberOfReducers;

  int exitCode;

  // Parse the parameters

    numberOfReducers = Integer.parseInt(args[0]);

    inputPath = new Path(args[1]);

    outputDir = new Path(args[2]);

    Configuration conf = this.getConf();

    // Define a new job

    Job job = Job.getInstance(conf);

    // Assign a name to the job

    job.setJobName("Exercise #1 - Exam 2018/06/26");

    // 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(TextInputFormat.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(IntWritable.class);

    // Set reduce class

    job.setReducerClass(ReducerBigData.class);

    // Set reduce output key and value classes

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(NullWritable.class);

    // Set number of reducers. Any value >=1

    job.setNumReduceTasks(numberOfReducers);

    // Execute the job and wait for completion

    if (job.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 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 data

        // Timestamp,VSID,CPUUtilization%,RAMUtilization%

        // Timestamp format:yyyy/mm/dd\_hh:mm

        // Example: 2018/05/01,15:40,VS1,10.5,0.5

        String[] fields = value.toString().split(",");

        String date = fields[0];

        String time=fields[1];

        String vsid = fields[2];

        Double cpuUtil = Double.parseDouble(fields[3]);

        String[] yyyymmdd = date.split("/");

        String year = yyyymmdd[0];

        String month = yyyymmdd[1];

        int hour = Integer.parseInt(time.split(":")[0]);

        // Select only May 2018 from 9:00 to 17:59 and the lines with

        // CPUUtilization%>99.8

        if (year.compareTo("2018") == 0 && month.compareTo("05") == 0 && hour >= 9 && hour <= 17

                && cpuUtil > 99.8) {

            // emit the pair (vsid, 1)

            context.write(new Text(vsid), new IntWritable(1));

        }

    }

}

/\*\* \* Exercise 1 - Reducer \*/

class ReducerBigData extends

        Reducer<Text, // Input key type

                IntWritable, // Input value type

                Text, // Output key type

                NullWritable> { // Output value type

    @Override

    protected void reduce(Text key, // Input key type

            Iterable<IntWritable> values, // Input value type

            Context context) throws IOException, InterruptedException {

        int sum = 0;

        // Iterate over the set of values and sum them

        for (IntWritable one : values) {

            sum = sum + one.get();

        }

        if (sum >= 10000) {

            context.write(key, NullWritable.get());

        }

    }

}

**SolutionExamExample4**

**Exercise1**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured

implements Tool {

  @Override

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

    Path inputPath;

    Path outputDir;

  int exitCode;

  // Parse the parameters

    inputPath = new Path(args[0]);

    outputDir = new Path(args[1]);

    Configuration conf = this.getConf();

    // Define a new job

    Job job = Job.getInstance(conf);

    // Assign a name to the job

    job.setJobName("Exercise #1 - Exam 2019/09/03");

    // 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(TextInputFormat.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(NullWritable.class);

    job.setMapOutputValueClass(Text.class);

    // Set reduce class

    job.setReducerClass(ReducerBigData.class);

    // Set reduce output key and value classes

    job.setOutputKeyClass(DoubleWritable.class);

    job.setOutputValueClass(Text.class);

    // The number of instances of the reducer must be 1

    job.setNumReduceTasks(1);

    // Execute the job and wait for completion

    if (job.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 1 - Mapper \*/

class MapperBigData extends Mapper<LongWritable, // Input key type

        Text, // Input value type

        NullWritable, // Output key type

        Text> {// Output value type

    double highestPrice;

    String firstTimestampHighestPrice;

    protected void setup(Context context) {

        highestPrice = Double.MIN\_VALUE;

        firstTimestampHighestPrice = null;

    }

    protected void map(LongWritable key, // Input key type

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        // Split value

        // stockId,date,hour:minute,price

        // Example: GOOG,2015/05/21,15:05,45.32

        String[] fields = value.toString().split(",");

        String stockId = fields[0];

        String date = fields[1];

        String hourAndMinute = fields[2];

        double price = Double.parseDouble(fields[3]);

        String currentTimestamp = new String(date + "\_" + hourAndMinute);

        // Select only stockId GOOG and year 2017

        if (date.startsWith("2017") == true && stockId.equals("GOOG")) {

            if (firstTimestampHighestPrice == null || price > highestPrice) {

                // Update highestPrice

                highestPrice = price;

                // Update the date associated with the current highest price

                firstTimestampHighestPrice = currentTimestamp;

            } else if (price == highestPrice) {

                // Check if also the timestamp must be updated

                if (currentTimestamp.compareTo(firstTimestampHighestPrice) < 0)

                    firstTimestampHighestPrice = currentTimestamp;

            }

        }

    }

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

        // emit the local top 1 price and the associated date

        if (firstTimestampHighestPrice != null)

            context.write(NullWritable.get(), new Text(highestPrice + "," + firstTimestampHighestPrice));

    }

}

/\*\* \* Exercise 1 - Reducer \*/

class ReducerBigData extends Reducer<NullWritable, // Input key type

        Text, // Input value type

        DoubleWritable, // Output key type

        Text> { // Output value type

    @Override

    protected void reduce(NullWritable key, // Input key type

            Iterable<Text> values, // Input value type

            Context context) throws IOException, InterruptedException {

        double highestPrice = Double.MIN\_VALUE;

        String firstTimestampHighestPrice = null;

        // Iterate over the set of values and compute the global

        // maximum price and the first date it occurs

        for (Text value : values) {

            // Split value

            // localmaximumprice,timestamp

            String[] fields = value.toString().split(",");

            double price = Double.parseDouble(fields[0]);

            String timestamp = fields[1];

            if (firstTimestampHighestPrice == null || price > highestPrice) {

                // Update highestPrice

                highestPrice = price;

                // Update the date associated with the current highest price

                firstTimestampHighestPrice = timestamp;

            } else if (price == highestPrice) {

                // Check if also the timestamp must be updated

                if (timestamp.compareTo(firstTimestampHighestPrice) < 0)

                    firstTimestampHighestPrice = timestamp;

            }

        }

        // Emit the result

        context.write(new DoubleWritable(highestPrice), new Text(firstTimestampHighestPrice));

    }

}

**SolutionExamExample5**

**Exercise1**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured

implements Tool {

  @Override

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

    Path inputPath;

    Path outputDir;

    int numberOfReducers;

  int exitCode;

  // Parse the parameters

    numberOfReducers = Integer.parseInt(args[0]);

    inputPath = new Path(args[1]);

    outputDir = new Path(args[2]);

    Configuration conf = this.getConf();

    // Define a new job

    Job job = Job.getInstance(conf);

    // Assign a name to the job

    job.setJobName("Exercise #1 - Exam 5");

    // 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(TextInputFormat.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(Text.class);

    // Set reduce class

    job.setReducerClass(ReducerBigData.class);

    // Set reduce output key and value classes

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(NullWritable.class);

    // Set number of reducers. Any value >=1

    job.setNumReduceTasks(numberOfReducers);

    // Execute the job and wait for completion

    if (job.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 1 - 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 {

        // Split data

        // SID13,PentiumV,DC10,Turin,Italy

        String[] fields = value.toString().split(",");

        String CPUversion = fields[1];

        String datacenter = fields[2];

        String country = fields[4];

        // Select only Italian cities

        if (country.compareTo("Spain") == 0) {

            // emit the pair (DataCenter,CPUVersion)

            context.write(new Text(datacenter), new Text(CPUversion));

        }

    }

}

/\*\* \* Exercise 1 - Reducer \*/

class ReducerBigData extends Reducer<Text, // Input key type

        Text, // Input value type

        Text, // Output key type

        NullWritable> { // Output value type

    @Override

    protected void reduce(Text key, // Input key type

            Iterable<Text> values, // Input value type

            Context context) throws IOException, InterruptedException {

        Boolean allEqual = true;

        String CPUversion = null;

        // Iterate over the set of values and check if they are all associated with the

        // same CPU version

        for (Text currentCPUversion : values) {

            if (CPUversion != null && CPUversion.compareTo(currentCPUversion.toString()) != 0) {

                // There are at least two different CPU versions for the current data center. It must

                // be discarded.

                allEqual = false;

            }

            CPUversion = currentCPUversion.toString();

        }

        if (allEqual == true) {

            // All the values are associated with the same CPU version for this data center. The

            // data center is selected.

            context.write(new Text(key.toString() + "," + CPUversion), NullWritable.get());

        }

    }

}

**Previous Exam**

**DBD\_Exam20200627Sol**

**Exercise1**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured

implements Tool {

  @Override

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

    Path inputPath;

    Path outputDir;

    int numReducers;

  int exitCode;

  // Parse the parameters

    inputPath = new Path("exam\_ex1\_data/CarsFailures.txt");

    outputDir = new Path("resultEx1");

    numReducers = Integer.parseInt(args[0]);

    Configuration conf = this.getConf();

    // Define a new job

    Job job = Job.getInstance(conf);

    // Assign a name to the job

    job.setJobName("Exam 2020/06/27 - Exercise #1");

    // 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(TextInputFormat.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(Text.class);

    // Set reduce class

    job.setReducerClass(ReducerBigData.class);

    // Set reduce output key and value classes

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(IntWritable.class);

    // Set number of reducers. Any value >=1 in this case

    job.setNumReduceTasks(numReducers);

    // Execute the job and wait for completion

    if (job.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 1 - 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 {

        // Split record

        // Example: 2015/01/05,08:45,Car15,Engine

        String[] fields = value.toString().split(",");

        String date = fields[0];

        String carID = fields[2];

        String failureType = fields[3];

        // Select only failures of year 2018

        if (date.startsWith("2018")==true) {

            // Emit (CarID,FailureType)

            context.write(new Text(carID), new Text(failureType) );

        }

    }

}

/\*\* \* Exercise 1 - Reducer \*/

class ReducerBigData extends Reducer<Text, // Input key type

        Text, // Input value type

        Text, // Output key type

        IntWritable> { // Output value type

    protected void reduce(Text key, // Input key type

            Iterable<Text> values, // Input value type

            Context context) throws IOException, InterruptedException {

        // Iterate over the set of values

        // Count the number of input values and

        // check if there are at least two different failure types

        int numFailures = 0;

        String previousFailureType = null;

        Boolean atLeastTwoFailureTypes = false;

        for (Text value : values) {

            numFailures++;

            // Check if the current failureType is different from the previous one

            // If it is true there are at least two different failure types

            if( previousFailureType!=null && previousFailureType.equals(value.toString())==false) {

                atLeastTwoFailureTypes = true;

            }

            previousFailureType=value.toString();

        }

        // Emit the CarID and the number of failures only if

        // - number of failures >=5

        // - there are at least two failure types for this car

        if (numFailures>=5 && atLeastTwoFailureTypes==true)

            context.write(new Text(key), new IntWritable(numFailures));

    }

}

**DBD\_Exam20200720Sol**

**Exercise1**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured

implements Tool {

  @Override

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

    Path inputPath;

    Path outputDir;

    int numReducers;

  int exitCode;

  // Parse the parameters

    inputPath = new Path("exam\_ex1\_data/SmartphoneModels.txt");

    outputDir = new Path("resultEx1");

    numReducers = Integer.parseInt(args[0]);

    Configuration conf = this.getConf();

    // Define a new job

    Job job = Job.getInstance(conf);

    // Assign a name to the job

    job.setJobName("Exam 2020/07/20 - Exercise #1");

    // 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(TextInputFormat.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(Text.class);

    // Set reduce class

    job.setReducerClass(ReducerBigData.class);

    // Set reduce output key and value classes

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(Text.class);

    // Set number of reducers. Any value >=1 in this case

    job.setNumReduceTasks(numReducers);

    // Execute the job and wait for completion

    if (job.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 1 - 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 {

        // Split record

        // MID,ModelName,Brand

        String[] fields = value.toString().split(",");

        String mid = fields[0];

        String brand = fields[2];

        // Emit (brand,mida)

        context.write(new Text(brand), new Text(mid));

    }

}

/\*\* \* Exercise 1 - Reducer \*/

class ReducerBigData extends Reducer<Text, // Input key type

        Text, // Input value type

        Text, // Output key type

        Text> { // Output value type

    protected void reduce(Text key, // Input key type

            Iterable<Text> values, // Input value type

            Context context) throws IOException, InterruptedException {

        // Iterate over the set of values

        // Check if there is only one model type

        String mid = null;

        Boolean onlyOneModel = true;

        for (Text value : values) {

            if (mid == null)

                mid = value.toString();

            else if (mid.equals(value.toString()) == false)

                onlyOneModel = false;

        }

        // Emit brand,mid if the brand is associated to one single model

        if (onlyOneModel == true)

            context.write(new Text(key), new Text(mid));

    }

}

**DBD\_Exam20200914Sol**

**Exercise1**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured implements Tool {

    @Override

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

        Path inputPath;

        Path outputDir;

        int exitCode;

        // Parse the parameters

        inputPath = new Path("exam\_ex1\_data/Users.txt");

        outputDir = new Path("resultEx1/");

        Configuration conf = this.getConf();

        // Define a new job

        Job job = Job.getInstance(conf);

        // Assign a name to the job

        job.setJobName("Exam 2020/09/14 - Exercise #1");

        // 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(TextInputFormat.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(NullWritable.class);

        job.setMapOutputValueClass(YearOccurrences.class);

        // Set reduce class

        job.setReducerClass(ReducerBigData.class);

        // Set reduce output key and value classes

        job.setOutputKeyClass(IntWritable.class);

        job.setOutputValueClass(IntWritable.class);

        // Set number of reducers. Exactly 1

        job.setNumReduceTasks(1);

        // Execute the job and wait for completion

        if (job.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 1 - Mapper \*/

class MapperBigData extends Mapper<LongWritable, // Input key type

        Text, // Input value type

        NullWritable, // Output key type

        YearOccurrences> {// Output value type

    YearOccurrences localYearOccurences;

    protected void setup(Context context) {

        localYearOccurences = new YearOccurrences();

        localYearOccurences.year = Integer.MIN\_VALUE;

        localYearOccurences.numOccurrences = 0;

    }

    protected void map(LongWritable key, // Input key type

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        // Split record

        // UID,Name,Surname,Gender,YearOfBirth

        String[] fields = value.toString().split(",");

        int yearOfBirth = Integer.parseInt(fields[4]);

        String gender = fields[3];

        if (gender.compareTo("Female") == 0) {

            // Check if this is the youngest female user (max year of birth)

            if (yearOfBirth > localYearOccurences.year) {

                // New local max year of birth

                localYearOccurences.year = yearOfBirth;

                localYearOccurences.numOccurrences = 1;

            } else if (yearOfBirth == localYearOccurences.year)

                localYearOccurences.numOccurrences++;

        }

    }

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

        // Emit (NullWritbale, local max year of birth and local number of occurrences)

        context.write(NullWritable.get(), localYearOccurences);

    }

}

/\*\* \* Exercise 1 - Reducer \*/

class ReducerBigData extends Reducer<NullWritable, // Input key type

        YearOccurrences, // Input value type

        IntWritable, // Output key type

        IntWritable> { // Output value type

    protected void reduce(NullWritable key, // Input key type

            Iterable<YearOccurrences> values, // Input value type

            Context context) throws IOException, InterruptedException {

        int maxYear = Integer.MIN\_VALUE;

        int numOccurrences = 0;

        // Iterate over the set of values

        // Select the maximum value (year of birth of the youngest user)

        // and count its number of occurrences

        for (YearOccurrences value : values) {

            int year = value.year;

            int localOccurences = value.numOccurrences;

            if (year > maxYear) {

                maxYear = year;

                numOccurrences = localOccurences;

            } else if (year == maxYear)

                numOccurrences = numOccurrences + localOccurences;

        }

        // Emit Year of birth youngest female, number of occurrences

        context.write(new IntWritable(maxYear), new IntWritable(numOccurrences));

    }

}

public class YearOccurrences implements Writable {

    public int year;

    public int numOccurrences;

    @Override

    public void write(DataOutput out) throws IOException {

        out.writeInt(year);

        out.writeInt(numOccurrences);

    }

    @Override

    public void readFields(DataInput in) throws IOException {

        year = in.readInt();

        numOccurrences = in.readInt();

    }

}

**DBD\_Exam** **20210122Sol**

**Exercise1**

/\*\*

 \* MapReduce program

 \*/

public class DriverBigData extends Configured implements Tool {

    @Override

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

        Path inputPath;

        Path outputDir;

        int exitCode;

        // Parse the parameters

        inputPath = new Path("exam\_ex1\_data/Failures.txt");

        outputDir = new Path("resultEx1/");

        Configuration conf = this.getConf();

        // Define a new job

        Job job = Job.getInstance(conf);

        // Assign a name to the job

        job.setJobName("Exam 2021/01/22 - Exercise #1");

        // 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(TextInputFormat.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(NullWritable.class);

        // Set reduce class

        job.setReducerClass(ReducerBigData.class);

        // Set reduce output key and value classes

        job.setOutputKeyClass(IntWritable.class);

        job.setOutputValueClass(NullWritable.class);

        // Set number of reducers. Any value >=1 in this case

        job.setNumReduceTasks(1);

        // Execute the job and wait for completion

        if (job.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 1 - Mapper

 \*/

class MapperBigData extends Mapper<LongWritable, // Input key type

        Text, // Input value type

        Text, // Output key type

        NullWritable> {// Output value type

    protected void map(LongWritable key, // Input key type

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        // Split record

        // R15,FCode122,2018/01/01,06:40:21

        String[] fields = value.toString().split(",");

        String failureType = fields[1];

        String date = fields[2];

        // Select only data related to year 2018

        if (date.startsWith("2018")) {

            // Emit (failureType, NullWritable)

            context.write(new Text(failureType), NullWritable.get());

        }

    }

}

/\*\*

 \* Exercise 1 - Reducer

 \*/

class ReducerBigData extends Reducer<Text, // Input key type

        NullWritable, // Input value type

        IntWritable, // Output key type

        NullWritable> { // Output value type

    int numDistinctFailures;

    protected void setup(Context context) {

        numDistinctFailures = 0;

    }

    protected void reduce(Text key, // Input key type

            Iterable<NullWritable> values, // Input value type

            Context context) throws IOException, InterruptedException {

        // Increment the number of distinct failure types

        numDistinctFailures++;

    }

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

        // Emit (numDistinctFailures, NullWritbale)

        context.write(new IntWritable(numDistinctFailures), NullWritable.get());

    }

}

**DBD\_Exam** **20210621Sol**

**Exercise1**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured

implements Tool {

  @Override

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

    Path inputPath;

    Path outputDir;

    int numberOfReducers;

  int exitCode;

  // Parse the parameters

  // Number of instances of the reducer class

    numberOfReducers = 2;

    // Folder containing the input data

    inputPath = new Path(args[0]);

    // Output folder

    outputDir = new Path(args[1]);

    Configuration conf = this.getConf();

    // Define a new job

    Job job = Job.getInstance(conf);

    // Assign a name to the job

    job.setJobName("Exam DBD 2021/06/21");

    // 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(TextInputFormat.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(IntWritable.class);

    // Set reduce class

    job.setReducerClass(ReducerBigData.class);

    // Set reduce output key and value classes

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(IntWritable.class);

    // Set number of reducers

    // Any numbner >=1

    job.setNumReduceTasks(numberOfReducers);

    // Execute the job and wait for completion

    if (job.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);

  }

}

/\*\*

 \* Basic MapReduce Project - 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[] fields = value.toString().split(",");

            String carModelID = fields[2];

            String date = fields[3];

            String country = fields[4];

            int year = Integer.parseInt(date.split("/")[0]);

            if(country.toLowerCase().equals("Italy") && (year == 2020 || year == 2019)) {

                if (year == 2020 )

                    context.write(new Text(carModelID), new IntWritable(+1));

                else

                    context.write(new Text(carModelID), new IntWritable(-1));

            }

    }

}

/\*\*

 \* Basic MapReduce Project - 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 diffCount2020\_2019 = 0;

        // Iterate over the set of values and sum them

        for (IntWritable value : values) {

            diffCount2020\_2019 = diffCount2020\_2019 + value.get();

        }

        if (diffCount2020\_2019 > 0)

            context.write(key, new IntWritable(diffCount2020\_2019));

    }

}

**DBD\_Exam** **20210705Sol**

**Exercise1**

public class DriverBigData extends Configured implements Tool {

    @Override

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

        int exitCode;

        Path inputPath;

        Path outputPath;

        inputPath = new Path(args[0]);

        outputPath = new Path(args[1]);

        // Any value >=1. The reduce phase can be parallelized.

        int nReducers = 2;

        Configuration conf = this.getConf();

        Job job = Job.getInstance(conf);

        job.setJobName("Exam20210705 - Hadoop - MapReduce");

        FileInputFormat.addInputPath(job, inputPath);

        FileOutputFormat.setOutputPath(job, outputPath);

        job.setJarByClass(DriverBigData.class);

        job.setMapperClass(MapperBigData.class);

        job.setReducerClass(ReducerBigData.class);

        job.setInputFormatClass(TextInputFormat.class);

        job.setOutputFormatClass(TextOutputFormat.class);

        job.setMapOutputKeyClass(Text.class);

        job.setMapOutputValueClass(IntWritable.class);

        job.setOutputKeyClass(Text.class);

        job.setOutputValueClass(DoubleWritable.class);

        job.setNumReduceTasks(nReducers);

        if(job.waitForCompletion(true))

            exitCode = 0;

        else

            exitCode = 1;

        return exitCode;

    }

    public static void main(String[] args) throws Exception {

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

        System.exit(res);

    }

}

public class MapperBigData extends Mapper<LongWritable, Text, Text, IntWritable> {

    @Override

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

        String[] fields = value.toString().split(",");

        String timestamp = fields[0];

        String itemId = fields[2];

        String purchased = fields[3];

        String year = timestamp.split("/")[0];

        int val;

        if (purchased.equals("true"))

            val = 1;

        else

            val = 0;

        if (year.equals("2020"))

            context.write(new Text(itemId), new IntWritable(val));

    }

}

public class ReducerBigData extends Reducer<Text, IntWritable, Text, DoubleWritable> {

    @Override

    protected void reduce(Text key, Iterable<IntWritable> values, Context context)

            throws IOException, InterruptedException {

        int numerator = 0;

        int denominator = 0;

        for (IntWritable value : values) {

            denominator++;

            numerator += value.get();

        }

        double convRate = (double) numerator / denominator;

        if (convRate > 0.001)

            context.write(key, new DoubleWritable(convRate));

    }

}

**LAB**

**LAB1**

/\*\*

 \* MapReduce program

 \*/

public class DriverBigData extends Configured implements Tool {

    @Override

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

        Path inputPath;

        Path outputDir;

        int numberOfReducers;

        int exitCode;

        // Parse the parameters

        // Number of instances of the reducer class

        numberOfReducers = Integer.parseInt(args[0]);

        // Folder containing the input data

        inputPath = new Path(args[1]);

        // Output folder

        outputDir = new Path(args[2]);

        Configuration conf = this.getConf();

        // Define a new job

        Job job = Job.getInstance(conf);

        // Assign a name to the job

        job.setJobName("Basic MapReduce Project - WordCount example");

        // 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(TextInputFormat.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(IntWritable.class);

        // Set reduce class

        job.setReducerClass(ReducerBigData.class);

        // Set reduce output key and value classes

        job.setOutputKeyClass(Text.class);

        job.setOutputValueClass(IntWritable.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;

        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);

    }

}

/\*\*

 \* Basic MapReduce Project - 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));

        }

    }

}

/\*\*

 \* Basic MapReduce Project - 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));

    }

}

**LAB2**

/\*\* \* Driver class. \*/

public class DriverBigData extends Configured implements Tool {

    public static enum COUNTERS {

        SELECTED\_WORDS,

        DISCARDED\_WORDS

    }

    @Override

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

        Path inputPath;

        Path outputDir;

        int exitCode;

        // Parse the parameters

        inputPath = new Path(args[0]);

        outputDir = new Path(args[1]);

        Configuration conf = this.getConf();

        // Get the prefix from the argument list and set it in the configuration.

        conf.set("prefix", args[2]);

        // Define a new job

        Job job = Job.getInstance(conf);

        // Assign a name to the job

        job.setJobName("Exercise 12");

        // 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 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(Text.class);

        // Set number of reducers

        job.setNumReduceTasks(0);

        // Execute the job and wait for completion

        if (job.waitForCompletion(true) == true) {

            exitCode = 0;

            Counter selectedWords = job.getCounters().findCounter(COUNTERS.SELECTED\_WORDS);

            Counter discardedWords = job.getCounters().findCounter(COUNTERS.DISCARDED\_WORDS);

            System.out.println("Selected: "+selectedWords.getValue());

            System.out.println("Discarded: "+discardedWords.getValue());

        }

        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);

    }

}

/\*\* \* Mapper \*/

class MapperBigData extends Mapper<Text, // Input key type

        Text, // Input value type

        Text, // Output key type

        Text> {// Output value type

    String prefix;

    protected void setup(Context context) {

        prefix = context.getConfiguration().get("prefix").toString();

    }

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

            Text value, // Input value type

            Context context) throws IOException, InterruptedException {

        // Check whether the word starts with the specified prefix

        if (key.toString().startsWith(prefix)) {

            context.write(key, value);

            context.getCounter(COUNTERS.SELECTED\_WORDS).increment(1);

        } else {

            context.getCounter(COUNTERS.DISCARDED\_WORDS).increment(1);

        }

    }

}

**LAB3**

/\*\* \* MapReduce program \*/

public class DriverBigData extends Configured implements Tool {

    @Override

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

        int exitCode;

        Configuration conf = this.getConf();

        // Define a new job

        Job job = Job.getInstance(conf);

        // Assign a name to the job

        job.setJobName("Lab#3 - Ex.1 - step 1");

        /\*

         \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

         \* Fill out the missing parts/update the content of this method

         \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

        \*/

        Path inputPath;

        Path outputDir;

        int numberOfReducersJob1;

        // Parse the parameters for the set up of the first job

        numberOfReducersJob1 = Integer.parseInt(args[0]);

        inputPath = new Path(args[1]);

        outputDir = new Path(args[2]);

        // Set the path of the input file/folder for this first job

        FileInputFormat.addInputPath(job, inputPath);

        // Set the 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(...);

        // Set job output format

        job.setOutputFormatClass(...);

        // Set map class

        job.setMapperClass(MapperBigData1.class);

        // Set map output key and value classes

        job.setMapOutputKeyClass(...);

        job.setMapOutputValueClass(...);

        // Set reduce class

        job.setReducerClass(ReducerBigData1.class);

        // Set reduce output key and value classes

        job.setOutputKeyClass(...);

        job.setOutputValueClass(...);

        // Set number of reducers

        job.setNumReduceTasks(numberOfReducersJob1);

        // Execute the first job and wait for completion

        if (job.waitForCompletion(true) == true) {

            // Set up the second job

            Job job2 = Job.getInstance(conf);

            // Assign a name to the second job

            job2.setJobName("Lab#3 - Ex.1 - step 2");

            /\* \*/

            // Change the following part of the code

            Path outputDir2;

            int numberOfReducersJob2;

            outputDir2 = new Path(args[3]);

            // Set path of the input file/folder for this second job

            // The output of the first job is the input of this second job

            FileInputFormat.addInputPath(job2, outputDir);

            // Set path of the output folder for this job

            FileOutputFormat.setOutputPath(job2, outputDir2);

            // Specify the class of the Driver for this job

            job2.setJarByClass(DriverBigData.class);

            // Set job input format

            job2.setInputFormatClass(...);

            // Set job output format

            job2.setOutputFormatClass(...);

            // Set map class

            job2.setMapperClass(MapperBigData2.class);

            // Set map output key and value classes

            job2.setMapOutputKeyClass(...);

            job2.setMapOutputValueClass(...);

            // Set reduce class

            job2.setReducerClass(ReducerBigData2.class);

            // Set reduce output key and value classes

            job2.setOutputKeyClass(...);

            job2.setOutputValueClass(...);

            // Set the number of reducers of the second job

            numberOfReducersJob2 = ..;

            job2.setNumReduceTasks(numberOfReducersJob2);

            // Execute the second job and wait for completion

            if (job2.waitForCompletion(true) == true)

                exitCode = 0;

            else

                exitCode = 1;

        } 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);

    }

}

/\*\*

 \* Lab  - Mapper

 \*/

/\* Set the proper data types for the (key,value) pairs \*/

class MapperBigData1 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 {

            /\* Implement the map method \*/

    }

}

/\*\*

 \* Lab - Reducer

 \*/

/\* Set the proper data types for the (key,value) pairs \*/

class ReducerBigData1 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 {

        /\* Implement the reduce method \*/

    }

}

import java.util.Vector;

/\* This class is used to store the top-k elements of a set of objects of type T.

 \* T is a class implementing the Comparable interface \*/

public class TopKVector<T extends Comparable<T>> {

    private Vector<T> localTopK;

    private Integer k;

    // It is used to create an empty TopKVector object.

    // k = number of top-k objects to store in this TopKVector object

    public TopKVector(int k) {

        this.localTopK = new Vector<T>();

        this.k = k;

    }

    public int getK() {

        return this.k;

    }

    // It is used to retrieve the vector containing the top-k objects among the

    // inserted ones

    public Vector<T> getLocalTopK() {

        return this.localTopK;

    }

    /\*

     \* It is used to insert a new element in the current top-k vector. The new

     \* element is inserted in the this.localTopK vector if and only if it is in

     \* the top-k objects.

     \*/

    public void updateWithNewElement(T currentElement) {

        if (localTopK.size() < k) { // There are less than k objects in

                                    // localTopK. Add the current element at the

                                    // end of localTopK

            localTopK.addElement(currentElement);

            // Sort the objects in localTopk

            sortAfterInsertNewElement();

        } else {

            // There are already k elements

            // Check if the current one is better than the least one

            if (currentElement.compareTo(localTopK.elementAt(k - 1)) > 0) {

                // The current element is better than the least object in

                // localTopK

                // Substitute the last object of localTopK with the current

                // object

                localTopK.setElementAt(currentElement, k - 1);

                // Sort the objects in localTopk

                sortAfterInsertNewElement();

            }

        }

    }

    private void sortAfterInsertNewElement() {

        // The last object is the only one that is potentially not in the right

        // position

        T swap;

        for (int pos = localTopK.size() - 1; pos > 0

                && localTopK.elementAt(pos).compareTo(localTopK.elementAt(pos - 1)) > 0; pos--) {

            swap = localTopK.elementAt(pos);

            localTopK.setElementAt(localTopK.elementAt(pos - 1), pos);

            localTopK.setElementAt(swap, pos - 1);

        }

    }

}

import org.apache.hadoop.io.Writable;

/\* This class is used to store a "word" of type String and a count of type Integer \*/

public class WordCountWritable implements Comparable<WordCountWritable>, Writable {

    private String word; // Contains a "word"

    private Integer count; // number of occurrences of "word"

    public WordCountWritable(String word, Integer count) {

        this.word = word;

        this.count = count;

    }

    public WordCountWritable(WordCountWritable other) {

        this.word = new String(other.getWord());

        this.count = Integer.valueOf(other.getCount());

    }

    public WordCountWritable() {

    }

    public String getWord() {

        return word;

    }

    public void setWord(String pair) {

        this.word = pair;

    }

    public Integer getCount() {

        return count;

    }

    public void setCount(Integer count) {

        this.count = count;

    }

    @Override

    public int compareTo(WordCountWritable other) {

        if (this.count.compareTo(other.getCount()) != 0) {

            return this.count.compareTo(other.getCount());

        } else { // if the count values of the two words are equal, the

                    // lexicographical order is considered

            return this.word.compareTo(other.getWord());

        }

    }

    @Override

    public void readFields(DataInput in) throws IOException {

        word = in.readUTF();

        count = in.readInt();

    }

    @Override

    public void write(DataOutput out) throws IOException {

        out.writeUTF(word);

        out.writeInt(count);

    }

    public String toString() {

        return new String(word + "," + count);

    }

}

/\*\*

 \* Lab  - Mapper

 \*/

/\* Set the proper data types for the (key,value) pairs \*/

class MapperBigData2 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 {

            /\* Implement the map method \*/

    }

}

/\*\*

 \* Lab - Reducer

 \*/

/\* Set the proper data types for the (key,value) pairs \*/

class ReducerBigData2 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 {

        /\* Implement the reduce method \*/

    }

}