Distributed Architectures for Big Data Processing and Analytics

Francesco Giuseppe Gillio

Department of Computer Science
Polytechnic University of Turin

March 26, 2024

Contents

Chapter 1

Apache Hadoop MapReduce

The Apache Hadoop framework coordinates the execution of the MapReduce program over the cluster nodes/servers, i.e. the parallel execution of the map and reduce phases, the execution of the shuffle and sort phase, the subtask scheduling, and the synchronization. Thus, Hadoop enables program developers to focus solely on implementing the `map()` and `reduce()` functions. The MapReduce program branches into three main parts, or, with reference to the appropriate programming language (**Java**), three specific classes:

- **Driver class**: the class containing the method/code that coordinates the configuration of the job and the workflow of the application;
- **Mapper class**: the class implementing the `map()` function;
- **Reducer class**: the class implementing the `reduce()` function.

Regarding the Hadoop framework terminology, the term (Hadoop) **job** refers to the execution of a MapReduce code over a data set; the term **task** refers to the execution of a Mapper (map task) or a Reducer (reduce task) on a slice of data (many task for each job); the term **input split** refers to a piece of the input data with approximately the same size of a HDFS block/chunk.

The Driver, i.e. an instance (object) of the Driver class, implements (1.) the `main()` method (the entry point of the application), which accepts arguments from the command line, and (2.) the `run()` method, which configures the job, submits the job to the Hadoop Cluster and "coordinates" the workflow of the application. The Mapper, i.e. an instance of the Mapper class, implements the **map phase**, i.e. executes the `map()` method individually for each `(key, value)` pair of the input file (read from the HDFS file system) to output a set of intermediate `(key, value)` pairs. The framework stores the map phase results in the local file system of the computing server (i.e. not in the HDFS).

The shuffle and sort procedure aggregates the intermediate results to generate a set of `(key, [list of values])` pairs, with a `(key, [list of values])` pair for each distinct key. The framework stores the shuffle and sort results in the local node (transient). The Reducer, i.e. an instance of the Reducer class, implements the **reduce phase**, i.e. executes the `reduce()` method individually for each `(key, [list of values])` pair coming from the previous phase (shuffle and sort) to output the final set of `(key, value)` pairs. The framework stores the reduce phase results in HDFS (the final result of the MapReduce application). In order to parallelize the job, Hadoop executes a set of tasks in parallel, i.e. instantiates a single Mapper (Map task) for each input split and a user-specifiable number of Reducers. The Driver runs on the client machine while the Mappers and Reducers executions run on the cluster nodes/servers.

1.1 Structure of a MapReduce program in Hadoop

1.1.1 MapReduce Data Types

The Hadoop framework implements some basic data types suitable for network serialization:

- `Java String` → `org.apache.hadoop.io.Text`
- `Java Float` → `org.apache.hadoop.io.FloatWritable`
- `Java Integer` → `org.apache.hadoop.io.IntWritable`
- etc...

In order to enable Hadoop to execute comparison operations (between keys and/or between values) in the sorting and shuffling phase, the basic Hadoop data types implement the `Writable` and `WritableComparable` interfaces of `org.apache.hadoop.io`. To manage complex data types, the framework enables developers to set up appropriate classes with the constraint of implementing the `Writable` and/or `WritableComparable` interfaces (mandatory), e.g.

```
public class MyClass implements org.apache.hadoop.io.Writable {....}
```

1.1.2 InputFormat and OutputFormat

The MapReduce program reads as input an HDFS file (or an HDFS folder). In order to logically transform the input HDFS file into a set of `(key, value)` pairs suitable for the Mapper, the Driver class extends the `org.apache.hadoop.mapreduce.InputFormat` abstract class, which "describes" the input-format specification for a MapReduce application. The abstract class enables the Driver to read the input data, split the input file(s) into logical input splits (reference to a part of the input file for a single Mapper), and provide the record reader implementation suitable

to divide the logical input split in a set of `(key,value)` pairs (records) for the Mapper. Each Mapper of the program reads a single input split, one record at a time, i.e. a single `(key,value)` pair at a time. The Hadoop framework provides a set of basic classes extending the `InputFormat` abstract class for standard input file formats:

- `TextInputFormat`: an `InputFormat` extension for plain text files that breaks files into lines (carriage-return to signal end of line) and returns one `(key, value)` pair for each line of the file, with the position (offset) of the line within the file as key and the content of the line as value.
- `KeyValueTextInputFormat`: another `InputFormat` extension for plain text files, with lines in the format `key<separator>value`, that breaks files into lines (carriage-return to signal end of line) and returns one `(key, value)` pair for each line of the file, with the text preceding the separator as key and the text following the separator as value.
- ...

The Driver class extends the `org.apache.hadoop.mapreduce.OutputFormat` abstract class to write the output of the MapReduce program into the HDFS. Again, the Hadoop framework provides a set of basic classes extending the `OutputFormat` abstract class for standard output file formats:

- `TextOutputFormat`: an `OutputFormat` extension for plain text files that writes one line in the output file for each output `(key, value)` pair.
- `SequenceFileOutputFormat`: an `OutputFormat` extension for sequential/binary files.
- ...

1.1.3 Driver

The Driver class extends the `org.apache.hadoop.conf.Configured` class and implements the `org.apache.hadoop.util.Tool` interface. The developer implements the `main()` method, that accepts arguments from the command line, and the `run()` method, that configures the job, i.e. sets the input and output format specification of the job, the mapper class along with the respective output type, the reducer class along with the respective output type, and the number of reducer.

```
1 /* set package */
2 package it.polito.bigdata.hadoop."package-name";
3
4 /* import libraries */
5 import java.io.IOException;
6 import org.apache.hadoop.mapreduce.Job;
7 import org.apache.hadoop.util.Tool;
8 import org.apache.hadoop.util.ToolRunner;
```

```
9  import org.apache.hadoop.conf.Configuration;
10 import org.apache.hadoop.conf.Configured;
11 import org.apache.hadoop.io.*;
12 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
14
15 /* the driver class */
16 public class Driver extends Configured implements Tool {
17
18   /* run method of the driver class */
19   @Override
20   public int run(String[] args) throws Exception {
21
22     // variables
23     int exitCode;
24     int numberOfReducers;
25     Path inputPath;
26     Path outputDir;
27
28     // parse the parameters
29     // number of instances of the reducer class
30     numberOfReducers = Integer.parseInt(args[0]);
31     // folder containing the input data
32     inputPath = new Path(args[1]);
33     // output folder
34     outputDir = new Path(args[2]);
35
36     // define and configure the job
37     Configuration conf = this.getConf();
38     Job job = Job.getInstance(conf);
39     // assign a name to the job
40     job.setJobName("MapReduce program");
41
42     // set path of the input file/folder for the job
43     // for a folder, the job reads each files in the folder
44     FileInputFormat.addInputPath(job, inputPath);
45     // set path of the output folder for the job
46     FileOutputFormat.setOutputPath(job, outputDir);
47     // specify the class of the Driver for the job
48     job.setJarByClass(Driver.class);
49     // set job input format (TextInputFormat, KeyValueTextInputFormat, ...)
50     job.setInputFormatClass(InputFormat.class);
51     // set job output format (TextOutputFormat, SequenceFileOutputFormat, ...)
52     job.setOutputFormatClass(OutputFormat.class);
53     // set mapper class
54     job.setMapperClass(Mapper.class);
55     // set map output key and value classes (Text, IntWritable, ...)
56     job.setMapOutputKeyClass(MapperOutputKeyType.class);
57     job.setMapOutputValueClass(MapperOutputValueType.class);
58     // set reducer class
59     job.setReducerClass(Reducer.class);
60     // set reduce output key and value classes (Text, IntWritable, ...)
61     job.setOutputKeyClass(ReducerOutputKeyType.class);
62     job.setOutputValueClass(ReducerOutputValueType.class);
63     // set number of reducers
64     job.setNumReduceTasks(numberOfReducers);
65
66     // execute the job and wait for completion
67     if (job.waitForCompletion(true) == true)
68       exitCode = 0;
69     else
70       exitCode = 1;
71     return exitCode;
72   } // end of the run method
73
74   /* main method of the driver class */
75   public static void main(String args[]) throws Exception {
76     // exploit the ToolRunner class to "configure" and run the Hadoop application
```

```
77      int res = ToolRunner.run(new Configuration(), new Driver(), args);
78      System.exit(res);
79    } // end of the main method
80  } // end of public class Driver
```

1.1.4 Mapper

The Mapper class extends the `org.apache.hadoop.mapreduce.Mapper` class, i.e. a generic type/generic class with four type parameters: input key type and input value type (consistent with the `InputFormat` of the Driver), output key type and output value type (consistent with the `MapperOutputKeyType` and the `MapperOutputValueType` of the Driver). The developer implements the `map()` method, which the framework automatically executes for each `(key, value)` pair of the input file. The `map()` method processes the input `(key, value)` pair via standard **Java** code and emits a `(key, value)` pair via the `context.write(key, value)` method.

```
1  /* set package */
2  package it.polito.bigdata.hadoop."package-name";
3
4  /* import libraries */
5  import java.io.IOException;
6  import org.apache.hadoop.mapreduce.Mapper;
7  import org.apache.hadoop.io.*;
8
9  /* the mapper class */
10 class Mapper extends Mapper
11     <MapperInputKeyType, // input key type
12     MapperInputValueType, // input value type
13     MapperOutputKeyType, // output key type
14     MapperOutputValueType> // output value type
15     {
16
17   /* the map method */
18   protected void map(
19       MapperInputKeyType key, // input key type
20       MapperInputValueType value, // input value type
21       Context context) throws IOException, InterruptedException {
22
23       /* processes the input (key, value) pair via standard Java code
24       and emits a (key, value) pair via the context.write(key, value) method; */
25
26   } // end of the map method
27 } // end of class Mapper
```

1.1.5 Reducer

The Reducer class extends the `org.apache.hadoop.mapreduce.Reducer` class, i.e. a generic type/generic class with four type parameters: input key type (consistent with the `MapperOutputKeyType` of the Mapper), input value type (consistent with the `MapperOutputValueType` of the Mapper), output key type and output value type (consistent with the `OutputFormat` class of the Driver). The developer implements the `reduce()` method, which the framework automatically executes for each `(key, [list of values])` pair coming from the shuffle and sort phase(s), i.e. from the aggregation of the Mapper(s) output. The

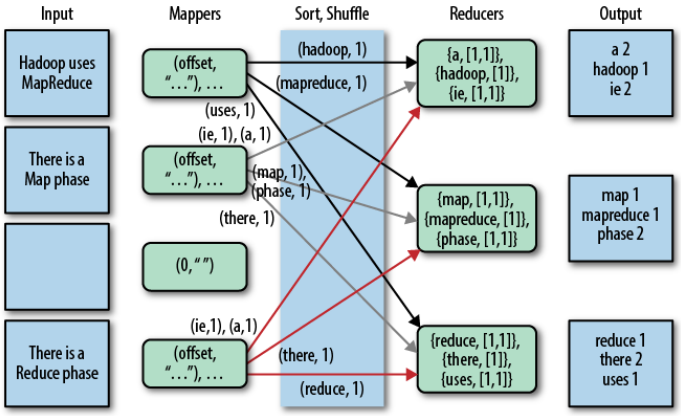

Figure 1.1: Word Count Problem

`reduce()` method processes the input `(key, [list of values])` pair via standard `Java` code and emits a `(key, value)` pair via the `context.write(key, value)` method.

```
1  /* set package */
2  package it.polito.bigdata.hadoop.package-name;
3
4  /* import libraries */
5  import java.io.IOException;
6  import org.apache.hadoop.mapreduce.Reducer;
7  import org.apache.hadoop.io.*;
8
9  /* the reducer class */
10 class Reducer extends Reducer
11     <ReducerInputKeyType, // input key type
12     ReducerInputValueType, // input value type
13     ReducerOutputKeyType, // output key type
14     ReducerOutputValueType> // output value type
15     {
16
17   /* the reduce method */
18   protected void reduce(
19       ReducerInputKeyType key, // input key type
20       Iterable<ReducerInputValueType> values, // input value type
21       Context context) throws IOException, InterruptedException {
22
23       /* processes the input (key, [list of values]) pair via standard Java code
24       and emits a (key, value) pair via the context.write(key, value) method; */
25
26   } // end of the reduce method
27 } // end of class Reducer
```

Word Count Problem

In the word count problem, as shown in Figure 1.1, the MapReduce program reads a textual file (without structure) as input, with each line containing a set of words, and returns as output the number of occurrences of each word appearing in the input file. The program runs after the `main(String args[])`

method reads the arguments of the application from the command line:

```
"args":  ["3", "input-file", "output-folder"]
```

- `args[0]`: number of instances of the Reducer class;
- `args[1]`: path of the input file;
- `args[2]`: path of the output folder.

The Driver class:

```
1  package it.polito.bigdata.hadoop.wordcount;
2  import org.apache.hadoop.conf.Configuration;
3  import org.apache.hadoop.conf.Configured;
4  import org.apache.hadoop.fs.Path;
5  import org.apache.hadoop.io.IntWritable;
6  import org.apache.hadoop.io.Text;
7  import org.apache.hadoop.mapreduce.Job;
8  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
9  import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
10 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
11 import
12 org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
13 import org.apache.hadoop.util.Tool;
14 import org.apache.hadoop.util.ToolRunner;
15 /* the driver class */
16 public class WordCount extends Configured implements Tool {
17   @Override
18   public int run(String[] args) throws Exception {
19     Path inputPath;
20     Path outputDir;
21     int numberOfReducers;
22     int exitCode;
23     numberOfReducers = Integer.parseInt(args[0]);
24     inputPath = new Path(args[1]);
25     outputDir = new Path(args[2]);
26     Configuration conf = this.getConf();
27     Job job = Job.getInstance(conf);
28     job.setJobName(name: "MapReduce program - Word Count");
29     FileInputFormat.addInputPath(job, inputPath);
30     FileOutputFormat.setOutputPath(job, outputDir);
31     job.setInputFormatClass(TextInputFormat.class); // -> text file
32     job.setOutputFormatClass(TextOutputFormat.class); // -> text file
33     job.setJarByClass(WordCount.class);
34     job.setMapperClass(WordCountMapper.class);
35     job.setMapOutputKeyClass(Text.class); // -> word
36     job.setMapOutputValueClass(IntWritable.class); // -> 1
37     job.setReducerClass(WordCountReducer.class);
38     job.setOutputKeyClass(Text.class); // -> word
39     job.setOutputValueClass(IntWritable.class); // -> number of occurrences
40     job.setNumReduceTasks(numberOfReducers);
41     if (job.waitForCompletion(true) == true)
42       exitCode = 0;
43     else
44       exitCode = 1;
45     return exitCode;
46   }
47   /* main method of the driver class */
48   public static void main(String args[]) throws Exception {
49     int res = ToolRunner.run(new Configuration(), new WordCount(), args);
50     System.exit(res);
51   }
52 }
```

The Mapper class:

```java
package it.polito.bigdata.hadoop.wordcount;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
/* the mapper class */
class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
  @Override
  protected void map(LongWritable key, Text value,
  Context context) throws IOException, InterruptedException {
    // (offset, "Hadoop uses MapReduce")
    // key: offset - LongWritable
    // value: "Hadoop uses MapReduce" - Text
    String[] words = value.toString().split("\\s+");
    // "Hadoop uses MapReduce" -> ["Hadoop", "uses", "MapReduce"]
    for(String word : words) {
      String cleanedWord = word.toLowerCase(); // "Hadoop" -> "hadoop"
      context.write(new Text(cleanedWord), new IntWritable(1)); // -> ("hadoop", 1)
    }
  }
}
```

The Reducer class:

```java
package it.polito.bigdata.hadoop.wordcount;
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
/* the reducer class */
class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
  @Override
  protected void reduce(Text key, Iterable<IntWritable> values,
  Context context) throws IOException, InterruptedException {
    // {there, [1,1]}
    // key: there - Text
    // value: [1,1] - Iterable<IntWritable>
    int occurrances = 0;
    for (IntWritable value : values) {
      occurrances = occurrances + value.get();
      // iter 1 -> occurrances = 0 + 1
      // iter 2 -> occurrances = 1 + 1
    }
    context.write(key, new IntWritable(occurrances));
    // -> (there, 2)
  }
}
```

1.1.6 MapReduce applications with Combiners

In standard MapReduce applications, the framework sends the `(key, value)` output pairs of the Mappers to the Reducers over the network. The Combiners (or "mini-reducers") enable "pre-aggregations" on the Mappers output to limit the amount of data sent over the network. For each Mapper, a Combiner, i.e. an instance of the Combiner class, implements the pre-aggregation phase by 1. combining the local output `(key, value)` pairs of the Mapper into `(key, [list of values])` pairs, and 2. invoking the `reduce()` method, individually for each pair coming from the previous "shuffle and sort" phase, to output a set of intermediate `(key, value)` pairs. The framework stores the pre-reduce phase results in the main memory (or on the local disk) of the cluster node

(i.e. not in the HDFS). The process operates in case of **commutative** and **associative** reduce function. The Hadoop framework runs the MapReduce job regardless of Combiners executions (runtime check).

The Combiner class extends the `org.apache.hadoop.mapreduce.Reducer` class. The developer implements the `reduce()` method, which the framework automatically executes for each `(key, [list of values])` pair coming from the aggregation of the local output of a Mapper. The MapReduce program instantiates the Combiner class in the run method of the Driver, i.e., in the job configuration part of the code, via the `job.setCombinerClass()` method.

Word Count Problem with Combiner

As before, in the word count problem the MapReduce program reads a textual file (without structure) as input, with each line containing a set of words, and returns as output the number of occurrences of each word appearing in the input file. The program runs after the `main(String args[])` method reads the arguments of the application from the command line:

```
"args": ["3", "input-file", "output-folder"]
```

- `args[0]`: number of instances of the reducer;
- `args[1]`: path of the input file;
- `args[2]`: path of the output folder.

The Mapper and Reducer classes remain the same as in the example above (without the Combiner). The Driver class differs from the previous code block by adding the instantiation of the Combiner class via the `job.setCombinerClass()` method within the `run()` method of the Driver, i.e. in the job configuration part of the code:

```
1  package it.polito.bigdata.hadoop.wordcount;
2  ....
3  /* the driver class */
4  public class WordCount extends Configured implements Tool {
5    /* run method of the driver class */
6    @Override
7    public int run(String[] args) throws Exception {
8    ....
9    // set combiner class
10   job.setCombinerClass(WordCountCombiner.class);
11   ....
12   }
13   /* main method of the driver class */
14   public static void main(String args[]) throws Exception {
15     int res = ToolRunner.run(new Configuration(), new WordCount(), args);
16     System.exit(res);
17   }
18 }
```

The Combiner class:

```
1  package it.polito.bigdata.hadoop.wordcount;
2  import java.io.IOException;
3  import org.apache.hadoop.io.IntWritable;
4  import org.apache.hadoop.io.Text;
5  import org.apache.hadoop.mapreduce.Reducer;
6  /* the combiner class */
7  class WordCountCombiner extends Reducer<Text, IntWritable, Text, IntWritable> {
8    /* the reduce method */
9    protected void reduce(Text key, Iterable<IntWritable> values,
10   Context context) throws IOException, InterruptedException {
11     int occurrances = 0;
12     for (IntWritable value : values) {
13       occurrances = occurrances + value.get();
14     }
15     context.write(key, new IntWritable(occurrances));
16   }
17 }
```

Note that the Reducer and Combiner classes perform the same computational operation by implementing the `reduce()` method in the same way. A common scenario therefore, instead of implementing a Combiner class, reuses the Reducer class in the `run()` method of the Driver, i.e., in the job configuration part of the code, via the `job.setCombinerClass(WordCountReducer.class)` method. About 99% of Hadoop applications instantiate the same class to implement both Combiner and Reducer.

1.1.7 Personalized Data Types

To handle a `(key, value)` pair with a complex data structure, developers resort to custom data types. Take into account that defining custom data types requires implementing the `org.apache.hadoop.io.Writable` interface (`WritableComparable` in the case of key management) and methods:

- `public void readFields(DataInput in)`
- `public void write(DataOutput out)`

To properly format the output of the job, the developer also "redefines" the method:

- `public String toString()`

Suppose an input file with "complex" values, i.e. with a composition of the type: a counter (`int`) and a sum (`float`). The developer defines an ad-hoc data type to implement such a complex data type in Hadoop.

```
1  package it.polito.bigdata.hadoop.combinerexample;
2  import java.io.DataInput;
3  import java.io.DataOutput;
4  import java.io.IOException;
5
6  public class SumAndCountWritable implements org.apache.hadoop.io.Writable {
7    /* private variables */
8    private float sum = 0;
9    private int count = 0;
10   /* methods to get and set private variables of the class */
11   public float getSum() {
```

```
12      return sum;
13    }
14    public void setSum(float sumValue) {
15      sum = sumValue;
16    }
17    public int getCount() {
18      return count;
19    }
20    public void setCount(int countValue) {
21      count = countValue;
22    }
23    /* methods to serialize and deserialize the contents of the instances of this class */
24    @Override /* serialize the fields of this object to out */
25    public void write(DataOutput out) throws IOException {
26      out.writeFloat(sum);
27      out.writeInt(count);
28    }
29    @Override /* deserialize the fields of this object from in */
30    public void readFields(DataInput in) throws IOException {
31      sum = in.readFloat();
32      count = in.readInt();
33    }
34    /* specify how to convert the contents of the instances of this class to a String
35    * useful to specify how to store/write the content of this class
36    * in a textual file */
37    public String toString() {
38      String formattedString = new String("sum = " + sum + ", count = " + count);
39      return formattedString;
40    }
41  }
```

1.1.8 Sharing parameters among Driver, Mappers, and Reducers

The configuration object operates to share the (basic) configuration of the Hadoop environment across the Driver, the Mappers and the Reducers of the application/job. In the job configuration part of the `run()` method, the Driver retrieves the configuration object via `Configuration conf = this.getConf()`. The Hadoop framework then enables developers to set an appropriate data type via the `set()` method of the configuration object, i.e. via `conf.set("property-name", "value")`. The `conf` instance therefore enables the storage of a list of `("property-name", "value")` pairs, readable but not writable by Mappers and Reducers, which instead retrieve a `String` containing the value of the property via the `get("property-name")` method of `context.getConfiguration()`, i.e.

```
context.getConfiguration().get("property-name")
```

1.1.9 Counters

Hadoop provides a set of basic, built-in, counters to store some statistics about jobs, Mappers, Reducers, e.g., number of input and output records (i.e., pairs), number of bytes sent, etc. Nevertheless, the framework enables developers to define an arbitrary number of ad-hoc counters to compute global "statistics" by means of `Java enum`. For instance, suppose to update the word count application to: 1. filter out words beginning with a command-line specifiable prefix,

and 2. count the number of words that the program filters out. The request 1. concerns sharing parameters among Driver, Mappers and Reducers. In order to configure and archive the value of the prefix parameter (input from the command line) within the application, the developer implements the `set()` method of the configuration object in the job configuration part of the code (`run()` method of the Driver).

```
1  package it.polito.bigdata.hadoop.wordcount;
2  ....
3  /* the driver class */
4  public class WordCount extends Configured implements Tool {
5    /* run method of the driver class */
6    @Override
7    public int run(String[] args) throws Exception {
8      ....
9      // set the filtering prefix
10     conf.set("prefix", new String(args[3]));
11     ....
12   }
13   public static void main(String args[]) throws Exception {
14     ....
15   }
16 }
```

The above code expects to receive the prefix as the third argument (`args[3]`). To enable the Mapper class and/or the Reducer class to retrieve a `String` containing the value of the prefix parameter (to support the filtering operation), the developer implements the `getConfiguration()` method of the context object within the corresponding method (`map()` and/or `reduce()`).

```
1  package it.polito.bigdata.hadoop.wordcount;
2  ....
3  /* the mapper class */
4  class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
5    @Override
6    protected void map(LongWritable key, Text value,
7    Context context) throws IOException, InterruptedException {
8      ....
9      // read the filtering prefix
10     String prefix = context.getConfiguration().get("prefix");
11     ....
12   }
13 }
```

The request 2. concerns counters. The developer implements in the Driver class (outside the other methods) the `enum COUNTERS {IN, OUT}`, which instantiates the `COUNTERS.IN` counter, to count the number of words that the program mantains in, and the `COUNTERS.OUT` counter, to count the number of words that the program filters out. Take into account that `COUNTERS` refers to the group name (the `enum`'s name) and `IN` and `OUT` refer to the counter names (the `enum`'s fields).

```
1  package it.polito.bigdata.hadoop.wordcount;
2  ....
3  /* the driver class*/
4  public class WordCount extends Configured implements Tool {
5    // set counters
6    public enum COUNTERS {IN, OUT}
7    @Override
8    public int run(String[] args) throws Exception {
9      ....
```

```
10    }
11    public static void main(String args[]) throws Exception {
12      ....
13    }
14  }
```

To enable Mappers and/or Reducers to increment the counters, the developer implements the `getCounter().increment(value)` method of the context object within the corresponding method (`map()` and/or `reduce()`).

```
1   package it.polito.bigdata.hadoop.wordcount;
2   ....
3   /* the mapper class */
4   class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
5     @Override
6     protected void map(LongWritable key, Text value,
7     Context context) throws IOException, InterruptedException {
8       ....
9       // read the filtering prefix
10      String prefix = context.getConfiguration().get("prefix");
11      ....
12      String[] words = value.toString().split("\\s+");
13      for(String word : words) {
14        if (word.startsWith(prefix)) {
15          context.write(new Text(word), new IntWritable(1));
16          // increment in counter
17          context.getCounter(WordCount.COUNTERS.IN).increment(1);
18        }
19        else {
20          // increment out counter
21          context.getCounter(WordCount.COUNTERS.OUT).increment(1);
22        }
23      }
24    }
25  }
```

In order to retrieve and print the final value of the counters, the developer implements in the job configuration part of the code (`run()` method of the Driver) the `getCounters()` and `findCounter()` methods.

```
1   package it.polito.bigdata.hadoop.wordcount;
2   ....
3   /* the driver class*/
4   public class WordCount extends Configured implements Tool {
5     // set counters
6     public enum COUNTERS {IN, OUT}
7     @Override
8     public int run(String[] args) throws Exception {
9       ....
10      if (job.waitForCompletion(true)==true) {
11        exitCode = 0;
12        // retrieve the counter values
13        long in = job.getCounters().findCounter(COUNTERS.IN).getValue();
14        long out = job.getCounters().findCounter(COUNTERS.OUT).getValue();
15        // print on the standard output of the Driver
16        System.out.println("IN WORDS: " + in);
17        System.out.println("OUT WORDS: " + out);
18      } else {
19        exitCode = 1;
20      }
21      return exitCode;
22    }
23    public static void main(String args[]) throws Exception {
24    ....
25    }
```

The framework also enables developers to define ad-hoc counters on the fly via the `incrCounter("group name", "counter name", value)` method, useful for unknown counters at design time.

1.1.10 Map-only job

In some operations, e.g. record filtering applications, the Hadoop framework enables the management of the job via Mappers alone. In such operations, the framework disables the shuffle and sort phase, since the map phase archives the output, i.e., the `(key, value)` pairs, directly in the HDFS. The developers implement the `map()` method and instantiate the number of Reducers to zero over the job configuration phase (in the Driver class), i.e. via the `job.setNumReduceTasks(0)` method;

1.1.11 In-Mapper Combiner

The MapReduce framework provides Mapper classes with a `setup()` method and a `cleanup()` method, both empty if not overridden.

- `setup()`: the MapReduce program executes the `setup()` method once for each Mapper, prior to the several execution of the `map()` method. The `setup()` method sets the in-mapper variables, useful to maintain in-mapper statistics and preserve the state (locally for each Mapper) within and across executions of the `map()` method. The program updates the values of the in-mapper variables at each execution of the `map()` method, according to some updating rule. Take into account that each Mapper (each instance of the Mapper class) maintains a private copy of the in-mapper variables.

- `cleanup()`: the MapReduce program executes the `cleanup()` method once for each Mapper, after the several execution of the `map()` method. The `cleanup()` method emits `(key, value)` pairs according to the values of the in-mapper variables/statistics.

The MapReduce framework also provides Reduces classes with a `setup()` method and a `cleanup()` method, both empty if not overridden. As before, the MapReduce program executes the `setup()` method once for each Reducer, prior to the several execution of the `reduce()` method, and the `cleanup()` method once for each Reducer, after the several execution of the `reduce()` method.

In-Mapper Combiners represents a possible improvement over "standard" Combiners. The developers initialize a set of in-mapper variables in the `setup()` method (`@Override`) of the Mapper class (over the instance of the Mapper), set the updating rule for the in-mapper variables/statistics in the `map()` method and the emitting rule for the output `(key, value)` pairs in the `cleanup()` method (`@Override`) of the Mapper. Each Mapper, after analyzing each input record (input `(key, value)` pair) of the relative input-split via the `map()`

method, emits the output `(key, value)` pairs according to the values of the in-mapper variables via the `cleanup()` method. The in-mapper variables operates as internal Combiner in the Mapper, enabling the improvement of the overall performance of the application.

In-Mapper Combiner - Word Count Problem

```
class WordCountMapper {
  @Override
  method setup {
    // initialize the in-mapper variables
    A <- new AssociativeArray
  }
  @Override
  method map(offset key, line l) {
    // set the updating rule for the in-mapper variables
    for each word w in line l {
      A{w} <- A{w} + 1
    }
  }
  @Override
  method cleanup {
    // set the emitting rule for the output (key, value) pairs
    for each word w in A {
      emit(term w, count A{w})
    }
  }
}
```

Exercise 9. Input: a textual file (without structure). Output: report the number of occurrences of each word appearing in the input file via in-mapper combiners.

Input:

```
Test of the word count program
The word program is the Hadoop hello word program
Example document for hadoop word count
```

MapReduce program:

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, Text, IntWritable> {
    // set in-mapper variable
    HashMap<String, Integer> wordcount;
    @Override
    protected void setup(Context context) throws
    IOException, InterruptedException {
        // initialize in-mapper variable
        wordcount = new HashMap<String, Integer>();
    }
    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {
        String[] line = value.toString().split("\\s+");
        // update in-mapper variable
        for (String word : line) {
            if (wordcount.get(word.toLowerCase()) != null) {
                Integer count = wordcount.get(word.toLowerCase());
                wordcount.put(word.toLowerCase(), count + 1);
                }
            else {
                wordcount.put(word.toLowerCase(), 1);
                }
```

```
            }
    }
  @Override
  protected void cleanup(Context context) throws IOException, InterruptedException {
        for (String word : wordcount.keySet()) {
            context.write(new Text(word), new IntWritable(wordcount.get(word)));
            }
        }
}
```

```
/* the reducer class */
class ReducerBigData extends Reducer <Text, IntWritable, Text, IntWritable> {
    protected void reduce(Text key, Iterable<IntWritable> values,
        Context context) throws IOException, InterruptedException {
        int occurrences = 0;
        for (IntWritable value : values) {
            occurrences = occurrences + value.get();
            }
        context.write(key, new IntWritable(occurrences));
    }
}
```

Output:

```
count 2
document 1
example 1
for 1
hadoop 2
hello 1
is 1
of 1
program 3
test 1
the 3
word 4
```

1.2 MapReduce Patterns

MapReduce patterns refers to templates for solving common and general data manipulation problems with MapReduce.

1.2.1 Numerical Summarization

The numerical summarization pattern refers to a template for grouping records/objects by a key field(s) and computing a numerical aggregate (average, max, min, standard deviation, etc.) per group, useful to provide a top-level view over large input data sets. In the numerical summarization pattern, the Mappers output `(key, value)` pairs with:

- `key`: the fields for defining groups (the "group-by" fields);
- `value`: the fields for computing the aggregate statistics;

The Reducers receive a set of numerical values for each "group-by" key and compute the final statistics for each "group". In case of commutative and associative statistics, the framework accelerates performances via Combiners.

Some examples: word count problem, record count (per group) problem, min/-max/count (per group) problem, mean/median/standard deviation (per group) problem.

Exercise 5. Input: a collection of textual (`csv`) files (without structure) containing the daily value of PM10 for a set of sensors. Each line of the files appears in the format: `SID,YYYY-MM-DD,VALUE`. Output: report for each sensor (`SID`: the "group-by" field) the average value of PM10 (`VALUE`'s: the field for computing the aggregate statistics).

Input:

```
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
```

MapReduce program:

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, Text, FloatWritable> {
  protected void map(LongWritable key, Text value,
    Context context) throws IOException, InterruptedException {
    // input: "s1,2016-01-01,20.5"
    // split the line into (key, value) pair
    String[] line = value.toString().split(",");
    // "s1,2016-01-01,20.5" -> ["s1", "2016-01-01", "20.5"]
    String sensor = line[0].toLowerCase();
    // key: the "group-by" field -> "s1"
    float pm = Float.parseFloat(line[2]);
    // value: the field for computing the aggregate statistics -> 20.5
        context.write(new Text(sensor), new FloatWritable(pm));
        // -> ("s1", 20.5)
    // output: ("s1", 20.5)
  }
}
```

```
/* the reducer class */
class ReducerBigData extends Reducer <Text, FloatWritable, Text, FloatWritable> {
  @Override
  protected void reduce(Text key, Iterable<FloatWritable> values,
    Context context) throws IOException, InterruptedException {
    // input: ("s1", [20.5, 60.2, 55.5])
    // compute the average statistics
    float sum = 0;
    float number = 0;
    for (FloatWritable value : values) {
        sum = sum + value.get();
        number = number + 1;
        }
    context.write(key, new FloatWritable((float) sum / number));
    // output: ("s1", 45.4)
  }
}
```

Output:

```
s1  45.4
s2  34.3
```

1.2.2 Inverted Index Summarization

The inverted index summarization pattern refers to a template for building an index from the input data, i.e. mapping terms to a list of identifiers, useful to support faster searches or data enrichment. In the inverted inverted summarization pattern, the Mappers output `(key, value)` pairs with:

- `key`: the set of fields for indexing (keywords) groups;
- `value`: the unique identifier of the objects to associate with each "keyword";

The Reducers receive and concatenate the set of identifiers for each keyword. An example: web search engine problem `(word, [URLs])`.

Exercise 7. Input: a textual (`txt`) file containing a set of sentences. Each line of the file appears in the format: `sentence-id \t sentence \n`. Output: report for each word (`word`: the "keyword") the list of sentence ids of the sentences containing such word (`sentence-id`: the unique identifier of the objects to associate with each "keyword"). Also, filter out the words "and", "or", "not" (by command line).

```
"args":  ["1", "input-file", "output-folder", "and,or,not"]
```

Input:

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

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    // define and configure the job
    Configuration conf = this.getConf();
    // set the filtering prefix/prefixes
    conf.set("prefix", new String(args[3]));
    ...
  }
  /* main method of the driver class */
  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);
  }
}
```

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, Text, Text> {

  Set<String> filters;

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

```
8       filters = new HashSet<String>();
9       // read the filtering prefix/prefixes
10      String[] prefix = context.getConfiguration().get("prefix").toString().split(",");
11      for (String filter : prefix) {
12          filters.add(filter);
13          }
14      }
15
16    /* the map method */
17    protected void map(LongWritable key, Text value,
18        Context context) throws IOException, InterruptedException {
19          // split the line into [sentence id, sentence]
20          String[] line = value.toString().split("\t");
21          String id = line[0].toLowerCase(); // sentence id
22          // split the sentence into [list of words]
23          String[] words = line[1].toLowerCase().toString().split("\\s+");
24          for (String word : words) {
25              if (!filters.contains(word)) {
26                  context.write(new Text(word), new Text(id));
27              }
28          }
29    }
30  }
```

```
1   /* the reducer class */
2   class ReducerBigData extends Reducer <Text, Text, Text, Text> {
3     /* the reduce method */
4     @Override
5     protected void reduce(Text key, Iterable<Text> values,
6           Context context) throws IOException, InterruptedException {
7           Set<String> ids = new HashSet<String>();
8           for (Text id : values) {
9               ids.add(id.toString());
10          }
11          context.write(key, new Text(ids.toString()));
12    }
13  }
```

Output:

```
1   big [sentence#3]
2   data [sentence#3]
3   hadoop [sentence#3, sentence#2, sentence#1]
4   java [sentence#2]
5   spark [sentence#2, sentence#1]
```

1.2.3 Counting with Counters

The counting with counters pattern refers to a template for computing count summarizations of data sets, useful to provide a top-level view over large data sets. In the counting with counters summarization pattern, the Mappers process each input `(key, value)` pair (record) and increment a set of counters. The pattern avoids the implementation of Reducers and Combiners (map-only job). Some examples: records count problem, unique instances count problem, summarizations problem.

1.2.4 Filtering

The filtering pattern refers to a template for filtering out uninteresting input records, useful to provide analysis over the records of interest. In the filtering

pattern, the Mappers output a `(key, value)` pair for each record (value) that matches the filtering rule. The pattern avoids the implementation of Reducers and Combiners (map-only job). Some examples: record filtering problem, tracking events problem, data cleaning problem.

Exercise 12. Input: a collection of textual (`csv`) files containing the daily value of PM10 for a set of sensors. Each line of the files appears in the format: `SID,YYYY-MM-DD \t VALUE \n`. Output: report the records with a PM10 value below a user-specifiable threshold (argument of the program).

```
"args":  ["0", "input-file", "output-folder", 30.5]
```

Input:

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

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    // define and configure the job
    Configuration conf = this.getConf();
    // set the filtering threshold
    conf.set("threshold", new String(args[3])); // -> 30.5
    ...
    // set number of reducers
    job.setNumReduceTasks(0); // -> map-only job
    ...
  }
  /* main method of the driver class */
  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);
  }
}
```

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, Text, FloatWritable> {

  /* the map method */
  protected void map(LongWritable key, Text value,
      Context context) throws IOException, InterruptedException {
            // read the filtering threshold
            String threshold = context.getConfiguration().get("threshold");
            // split the line into [record, PM10]
            String[] line = value.toString().split("\t");
            String record = line[0].toLowerCase();
            float PM10 = Float.parseFloat(line[1]);
            if (PM10 < Float.parseFloat(threshold)) {
                context.write(new Text(record), new FloatWritable(PM10));
    }
  }
}
```

Output:

```
s1,2016-01-01 20.5
s1,2016-01-01 30.1
s2,2016-01-02 20.4
```

1.2.5 Distinct

The distinct pattern refers to a template for discovering a unique set of values/records, useful to provide duplicate data removals. In the distinct pattern, the Mappers emit a `(key, value)` pair for each input record with:

- `key`: the input record;
- `value`: the `null` value;

The Reducers emit a `(key, value)` pair for each input `(key, [list of values])` pair with:

- `key`: the input key, i.e., input record;
- `value`: the `null` value;

Some examples: duplicate data removal, distinct value selection.

Exercise 14. Input: a collection of news (textual files). Output: report a list of distinct words occurring in the collection.

Input:

```
Toy example file for Hadoop.
Hadoop running example.
```

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    // set mapper class
    job.setMapperClass(MapperBigData.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(NullWritable.class); // -> NullWritable
    // set combiner class
    job.setCombinerClass(CombinerBigData.class); // -> optional (optimization)
    // set reducer class
    job.setReducerClass(ReducerBigData.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(NullWritable.class); // -> NullWritable
    ...
  }
  /* main method of the driver class */
  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);
  }
}
```

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, Text, NullWritable> {

  /* the map method */
  protected void map(LongWritable key, Text value,
      Context context) throws IOException, InterruptedException {
        // split the line into [list of words]
        String[] line = value.toString().toLowerCase().split("\\s+");
        for (String word : line) {
            context.write(new Text(word), NullWritable.get());
        }
  }
}
```

```
/* the combiner class */
class CombinerBigData extends Reducer <Text, NullWritable, Text, NullWritable> {
  /* the reduce method */
  @Override
  protected void reduce(Text key, Iterable<NullWritable> values,
      Context context) throws IOException, InterruptedException {
      context.write(key, NullWritable.get());
  }
}
```

```
/* the reducer class */
class ReducerBigData extends Reducer <Text, NullWritable, Text, NullWritable> {
  /* the reduce method */
  @Override
  protected void reduce(Text key, Iterable<NullWritable> values,
      Context context) throws IOException, InterruptedException {
      context.write(key, NullWritable.get());
  }
}
```

Output:

```
example
file
for
hadoop
running
toy
```

1.2.6 Top K

The top K pattern refers to a template for selecting a set of top K records according to a ranking function, useful to output the most important records of the input data set. In the top K pattern, each Mapper initializes an in-mapper (local) top K list via the `setup()` method, updates the current in-mapper top K list at each execution of the `map()` function and, via the `cleanup()` method, emits the K `(key, value)` pairs with:

- `key`: the `null` key;
- `value`: a in-mapper top K record;

The framework stores the current (local) top K records of each Mapper (i.e., instance of the Mapper class) in the main (local) memory. The top K pattern enables the instantiation of a single Reducer (i.e., a single instance of the Reducer class), to provide a single global view over the intermediate results coming

from the Mappers and compute the final top K records. The Reducer computes the final top K list by merging the local lists coming from the Mappers. Since the Mappers output `(key, value)` pairs with the same key (the `null` key), the framework executes the `reduce()` method once. Some examples: outlier analysis problem (according to a ranking function), interesting data selection problem (according to a ranking function).

Exercise 13. Input: a collection of textual (`csv`) files containing the daily income of a company. Each line of the files appears in the format: `date \t daily income \n`. Output: report the date and income of the top K (argument of the program) most profitable dates. In case of tie, report the first K dates among the ones with the highest income.

```
"args":  ["1", "input-file", "output-folder", "5"]
```

Input:

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

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    // define and configure the job
    Configuration conf = this.getConf();
    // set the K value
    conf.set("K", new String(args[3])); // -> 5
    ...
    // set mapper class
    job.setMapperClass(MapperBigData.class);
    job.setMapOutputKeyClass(NullWritable.class); // -> NullWritable
    job.setMapOutputValueClass(Text.class);
    // set reducer class
    job.setReducerClass(ReducerBigData.class);
    job.setOutputKeyClass(NullWritable.class); // -> NullWritable
    job.setOutputValueClass(Text.class);
    job.setNumReduceTasks(tasks:1);
    ...

  }
  /* main method of the driver class */
  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);
  }
}
```

```
public class DateValue {
```

```
  protected String date;
  protected float value;
  public DateValue(String date, float value) {
    this.date = date;
    this.value = value;
  }
  public String getDate() {
    return this.date;
  }
  public float getValue() {
    return this.value;
  }
  @Override
  public String toString() {
      return this.date + "\t" + this.value;
  }
}
```

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, NullWritable, Text> {

  ArrayList<DateValue> top;
  int K;

  @Override
  protected void setup(Context context) throws IOException, InterruptedException {
    top = new ArrayList<DateValue>();
    K = Integer.parseInt(context.getConfiguration().get("K"));
  }

  /* the map method */
  protected void map(LongWritable key, Text value,
      Context context) throws IOException, InterruptedException {
      // split the line into [date, daily income]
      String[] line = value.toString().toLowerCase().split("\t");
      top.add(new DateValue(line[0], Float.parseFloat(line[1])));
  }

  @Override
  protected void cleanup(Context context) throws IOException, InterruptedException {
    top.sort((x, y) -> {
      if (Float.compare(y.getValue(), x.getValue()) == 0) {
        return x.getDate().compareTo(y.getDate());
      } else {
        return Float.compare(y.getValue(), x.getValue());
      }
    });
    for (int i = 0; i < K; i ++) {
      context.write(NullWritable.get(), new Text(top.get(i).toString()));
    }
  }
}
```

```
/* the reducer class */
class ReducerBigData extends Reducer <NullWritable, Text, NullWritable, Text> {

  ArrayList<DateValue> top;
  int K;

  @Override
  protected void setup(Context context) throws IOException, InterruptedException {
    top = new ArrayList<DateValue>();
    K = Integer.parseInt(context.getConfiguration().get("K"));
  }

  /* the reduce method */
  @Override
  protected void reduce(NullWritable key, Iterable<Text> values,
```

```
16        Context context) throws IOException, InterruptedException {
17        for (Text value : values) {
18          String[] line = value.toString().toLowerCase().split("\t");
19          top.add(new DateValue(line[0], Float.parseFloat(line[1])));
20        }
21    }
22
23    @Override
24    protected void cleanup(Context context) throws IOException, InterruptedException {
25      top.sort((x, y) -> {
26        if (Float.compare(y.getValue(), x.getValue()) == 0) {
27          return x.getDate().compareTo(y.getDate());
28        } else {
29          return Float.compare(y.getValue(), x.getValue());
30        }
31      });
32      for (int i = 0; i < K; i ++) {
33        context.write(NullWritable.get(), new Text(top.get(i).toString()));
34      }
35    }
36  }
```

Output:

```
1  2015-11-02 1305.0
2  2016-11-02 1305.0
3  2016-01-02 1145.0
4  2015-11-01 1000.0
5  2015-12-02 750.0
```

1.3 Multiple Inputs, Multiple Outputs, Distributed Cache

1.3.1 Multiple Inputs

The Hadoop framework enables reading data from multiple inputs (multiple datasets) with different formats by setting a different Mapper for each input dataset. In order to maintain coherence in the MapReduce program, developers set each Mapper to output `(key, value)` pairs consistent in terms of data types (relative to the others). Some examples: collection of PM10 measurements from different sensors with different format, such as `SID,YYYY-MM-DD,VALUE` and `YYYY-MM-DD,SID,VALUE`.

For managing multiple inputs, the developers implement in the job configuration part of the code (`run()` method of the Driver) the `addInputPath()` method of the `MultipleInputs` class multiple times to: 1. add a single input path at a time, 2. set the input format class for each input path and 3. set the Mapper class with respect to each input path. The method manages four parameters: the job object, the input path to the respective dataset, the proper `InputFormat` class, the proper Mapper class. For instance, suppose to implement a MapReduce program for managing different textual datasets with different data formats. The developer implements the `addInputPath()` method for each input dataset by setting the respective input path (command line argument, i.e. `args[i]`), the class for reading the data (`TextInputFormat` class for

textual data), and the Mapper class to manage the input `(key, value)` pairs of the respective path, i.e. `Mapper-i`.

```
class Driver {
  method run() {
    MultipleInputs.addInputPath(job,
                            new Path(args[1]),
                            TextInputFormat.class,
                            Mapper1.class);
    MultipleInputs.addInputPath(job,
                            new Path(args[2]),
                            TextInputFormat.class,
                            Mapper2.class);
  }
}
```

Exercise 17. Input: 2 different textual (`txt`) files containing the temperature measurements from different sensors. Each line of the file 1 appears in the format: `SID,YYYY-MM-DD,HOUR,TEMPERATURE`. Each line of the file 2 appears in the format: `YYYY-MM-DD,HOUR,TEMPERATURE,SID`. Output: report the maximum temperature for each date (considering the data of both input files).

Input (1):

```
s1,2016-01-01,14:00,20.5
s2,2016-01-01,14:00,30.2
s1,2016-01-02,14:10,11.5
s2,2016-01-02,14:10,30.2
```

Input (2):

```
2016-01-01,14:00,20.1,s3
2016-01-01,14:00,10.2,s4
2016-01-02,14:15,31.5,s3
2016-01-02,14:15,20.2,s4
```

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    // variables
    int exitCode;
    Path inputPath1;
    Path inputPath2;
    Path outputDir;
    int numberOfReducers;
    // parse the parameters
    numberOfReducers = Integer.parseInt(args[0]);
    // folder containing the input data
    inputPath1 = new Path(args[1]);
    inputPath2 = new Path(args[2]);
    // output folder
    outputDir = new Path(args[3]);
    // define and configure the job
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf);
    job.setJobName("Exercise 17");
    // set multiple inputs
    MultipleInputs.addInputPath(job, inputPath1, TextInputFormat.class, MapperType1BigData.class);
    MultipleInputs.addInputPath(job, inputPath2, TextInputFormat.class, MapperType2BigData.class);
    // set path of the output folder for the job
    FileOutputFormat.setOutputPath(job, outputDir);
```

```
28      // specify the class of the Driver for the job
29      job.setJarByClass(DriverBigData.class);
30      // set job output format (TextOutputFormat)
31      job.setOutputFormatClass(TextOutputFormat.class);
32      // set map output key and value classes (Text, FloatWritable)
33      job.setMapOutputKeyClass(Text.class);
34      job.setMapOutputValueClass(FloatWritable.class);
35      // set reducer class
36      job.setReducerClass(ReducerBigData.class);
37      // set reduce output key and value classes (Text, FloatWritable)
38      job.setOutputKeyClass(Text.class);
39      job.setOutputValueClass(FloatWritable.class);
40      // set number of reducers
41      job.setNumReduceTasks(numberOfReducers);
42      // execute the job and wait for completion
43      if (job.waitForCompletion(true) == true)
44        exitCode = 0;
45      else
46        exitCode = 1;
47      return exitCode;
48    }
49    /* main method of the driver class */
50    public static void main(String args[]) throws Exception {
51      // exploit the ToolRunner class to "configure" and run the Hadoop application
52      int res = ToolRunner.run(new Configuration(), new DriverBigData(), args);
53      System.exit(res);
54    }
55  }
```

```
1   /* the mapper class */
2   class MapperType1BigData extends Mapper <LongWritable, Text, Text, FloatWritable> {
3
4     /* the map method */
5     protected void map(LongWritable key, Text value,
6         Context context) throws IOException, InterruptedException {
7         // split the line
8         String[] line = value.toString().toLowerCase().split(",");
9         String date = line[1];
10        float temperature = Float.parseFloat(line[3]);
11        context.write(new Text(date), new FloatWritable(temperature));
12    }
13  }
```

```
1   /* the mapper class */
2   class MapperType2BigData extends Mapper <LongWritable, Text, Text, FloatWritable> {
3
4     /* the map method */
5     protected void map(LongWritable key, Text value,
6         Context context) throws IOException, InterruptedException {
7         // split the line
8         String[] line = value.toString().toLowerCase().split(",");
9         String date = line[0];
10        float temperature = Float.parseFloat(line[2]);
11        context.write(new Text(date), new FloatWritable(temperature));
12    }
13  }
```

```
1   /* the reducer class */
2   class ReducerBigData extends Reducer <Text, FloatWritable, Text, FloatWritable> {
3     /* the reduce method */
4     @Override
5     protected void reduce(Text key, Iterable<FloatWritable> values,
6         Context context) throws IOException, InterruptedException {
7               float max = 0;
8               for (FloatWritable value : values) {
9                   if (value.get() > max) {
10                      max = value.get();
```

```
11                  }
12              }
13              context.write(key, new FloatWritable(max));
14    }
15  }
```

Output:

```
1  2016-01-01 30.2
2  2016-01-02 31.5
```

1.3.2 Multiple Outputs

The Hadoop framework enables storing the output `(key, value)` pairs of a MapReduce application in different files, useful to support splitting and filtering operations. Each file contains a proper subset of the output `(key, value)` pairs (according to some rules). The framework stores the files (each file with a different developer-specifiable prefix) in a single output directory (multiple output files with different prefixes in a single directory). i.e., with no multiple output directories. Hadoop enables to select a different prefix for each "type" of output file via the `addNamedOutput()` method of the `MultipleOutputs` class (an output file of each type for each Reducer, or for each Mapper for map-only jobs). Hadoop exploits `"part-"` as standard prefix (see the content of the output directory of some of the previous applications).

For managing multiple outputs, the developers implement in the job configuration part of the code (`run()` method of the Driver) the `addNamedOutput()` method of the `MultipleOutputs` class multiple times to set the prefixes of the output files, once for each "output file type". The method manages five parameters: the job object, the "name/prefix" of `MultipleOutputs`, the `OutputFormat` class, the key output data type class and the value output data type class. For instance, suppose to implement a MapReduce program for storing different types of output textual files with different prefixes. The developer implements the `MultipleOutputs.addNamedOutput()` method for each output file by setting the respective prefix, the class for writing the data (`TextOutputFormat` class for textual data), the key output data type class and the value output data type class.

```
1  class Driver {
2    method run() {
3      MultipleOutputs.addNamedOutput(job,
4                                     "prefix-A",
5                                     TextOutputFormat.class,
6                                     Text.class,
7                                     NullWritable.class);
8      MultipleOutputs.addNamedOutput(job,
9                                     "prefix-B",
10                                    TextOutputFormat.class,
11                                    Text.class,
12                                    NullWritable.class);
13   }
14 }
```

Afterwards, the developers implement in the Reducer (or in the Mapper, for map-only job) a private `MultipleOutputs` variable and, in the `setup()` method, an instance of the `MultipleOutputs` class. The framework enables to: 1. write the `(key, value)` pairs in the file of interest via the `write()` method of the `MultipleOutputs` object inside the `reduce()` method (or inside the `map()` method, for map-only job), and 2. close the `MultipleOutputs` object via the `close()` method of the `MultipleOutputs` object inside the `cleanup()` method of the Reducer (or of the Mapper, for map-only job).

```
class Reducer/Mapper {
  private MultipleOutputs<Text, NullWritable> mos = null;
  @Override
  method setup {
    mos = new MultipleOutputs<Text, NullWritable>(context);
  }
  @Override
  method reduce()/map() {
    mos.write("prefix_A", key, value);
    mos.write("prefix-B", key, value);
  }
  @Override
  method cleanup {
    mos.close();
  }
}
```

Exercise 20. Split the readings of a set of sensors according to the value of the measurement. Input: a set of textual files (`txt`) containing the temperature measurements from different sensors. Each line of the files appears in the format: `SID,YYYY-MM-DD,HOUR,TEMPERATURE`. Output: report 1. a set of files with the prefix `hightemp` containing the lines of the input files with a temperature value greater than 30.0, and 2. a set of files with the prefix `normaltemp` containing the lines of the input files with a temperature value less than or equal to 30.0.

Input:

```
s1,2016-01-01,14:00,20.5
s2,2016-01-01,14:00,30.2
s1,2016-01-02,14:10,11.5
s2,2016-01-02,14:10,30.2
```

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    // variables
    int exitCode;
    Path inputPath;
    Path outputDir;
    int numberOfReducers;
    // parse the parameters
    numberOfReducers = Integer.parseInt(args[0]);
    inputPath = new Path(args[1]);
    outputDir = new Path(args[2]);
    // define and configure the job
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf);
    job.setJobName("Exercise 20");
```

```
    FileInputFormat.addInputPath(job, inputPath);
    FileOutputFormat.setOutputPath(job, outputDir);
    // set multiple outputs
    MultipleOutputs.addNamedOutput(job, "hightemp",
                                   TextOutputFormat.class,
                                   Text.class,
                                   NullWritable.class);
    MultipleOutputs.addNamedOutput(job, "normaltemp",
                                   TextOutputFormat.class,
                                   Text.class,
                                   NullWritable.class);
    // specify the class of the Driver for the job
    job.setJarByClass(DriverBigData.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    // set mapper class
    job.setMapperClass(MapperBigData.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(NullWritable.class);
    // set number of reducers
    job.setNumReduceTasks(0); // -> map-only job
    // execute the job and wait for completion
    if (job.waitForCompletion(true) == true)
      exitCode = 0;
    else
      exitCode = 1;
    return exitCode;
  }
  /* main method of the driver class */
  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);
  }
}
```

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, Text, NullWritable> {
  // set a MultiOutputs object
  private MultipleOutputs<Text, NullWritable> mos = null;
  /* the setup method */
  protected void setup(Context context) {
  mos = new MultipleOutputs<Text, NullWritable>(context);
  }
  /* the map method */
  protected void map(LongWritable key, Text value,
      Context context) throws IOException, InterruptedException {
      // split the line into [list of words]
      String[] line = value.toString().toLowerCase().split(",");
      float temperature = Float.parseFloat(line[3]);
      if (temperature > 30.0) {
        mos.write("hightemp", value, NullWritable.get());
      }
      else {
        mos.write("normaltemp", value, NullWritable.get());
      }
  }
  /* the cleanup method */
  protected void cleanup(Context context) throws IOException, InterruptedException {
    // close the MultiOutputs
    mos.close();
  }
}
```

Output (1) - `hightemp`:

```
s2,2016-01-01,14:00,30.2
s2,2016-01-02,14:10,41.5
```

Output (2) - `normaltemp`:

```
s1,2016-01-01,14:00,20.5
s1,2016-01-02,14:10,11.5
```

1.3.3 Distributed Cache

The Hadoop framework enables caching files (e.g., text, archives, jars) from the HDFS to the nodes of the clusters, useful to share and cache (small) read-only files in each node of the cluster to perform efficiently some applications task. The distributed cache facility provide each node with an accessible copy of the HDFS files. In the Driver of the application the developers set the files to cache by means of the `job.addCacheFile(path)` method. At the initialization of the job, the Hadoop framework creates a "local copy" of the files in each nodes (to execute some tasks of the job). The Mappers (or the Reducers) read the files through the `setup()` method. The efficiency of the distributed cache depends on the number of multiple Mappers (or Reducers) running on the same node/server. The framework stores a local copy of the file for each node at the initialization of the job, accessible by each Mapper (Reducer) running on the same node/server. Without the distributed cache, each Mapper (Reducer) read, in the `setup()` method, the HDFS file, requiring more time than reading data from the local file system of the node running the Mappers (Reducers).

The Driver class:

```
public int run(String[] args) throws Exception {
    .....
    // add the HDFS file in the distributed cache
    job.addCacheFile(new Path("hdfs path/filename").toUri());
    ......
}
```

The Mapper/Reducer class:

```
protected void setup(Context context) throws IOException, InterruptedException {
    .........
    String line;
    // retrieve the (original) paths of the files
    URI[] urisCachedFiles = context.getCacheFiles();
    // read the content of the file and process it.
    BufferedReader file = new BufferedReader(
        new FileReader(new File(new Path(urisCachedFiles[0].getPath()).getName()))
        );
    //iterate over the lines of the file
    while ((line = file.readLine()) != null) {
        // process the current line
        .....
        }
    file.close();
}
```

The `.getName()` retrieves the name of the file. The method stores the file in the root of a local temporary folder (one for each server of the running application) of the distributed cache. Take into account that the path of the original folder appears different from the one for storing the local copy of the file.

Exercise 21. Stopword elimination problem. Input: 1. a large textual file containing one sentence per line and 2. a small file containing a set of stopwords (one stopword per line). Output: report a textual file containing the same sentences of the large input file without the words appearing in the small file.

Input (1):

```
This is the first sentence and it contains some stopwords
Second sentence with a stopword here and another here
Third sentence of the stopword example
```

Input (2) - `stopwords.txt`:

```
a
an
and
the
```

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf);
    // add the HDFS file in the distributed cache
    job.addCacheFile(new Path("stopwords.txt").toUri());
    ...
    // set number of reducers
    job.setNumReduceTasks(0); // -> map-only job
    // execute the job and wait for completion
    if (job.waitForCompletion(true) == true)
      exitCode = 0;
    else
      exitCode = 1;
    return exitCode;
  }
  /* main method of the driver class */
  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);
  }
}
```

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, NullWritable, Text> {

  private ArrayList<String> stopwords;

  protected void setup(Context context) throws IOException, InterruptedException {
    String line;
    stopwords = new ArrayList<String>();
    // retrieve the (original) paths of the files
    URI[] urisCachedFiles = context.getCacheFiles();
    // read the content of the file and process it.
    BufferedReader file = new BufferedReader(
        new FileReader(new File(new Path(urisCachedFiles[0].getPath()).getName()))
        );
    //iterate over the lines of the file (each line of the file contains one stopword)
    while ((line = file.readLine()) != null) {
      stopwords.add(line);
```

```
18          }
19        file.close();
20      }
21      /* the map method */
22      protected void map(LongWritable key, Text value,
23          Context context) throws IOException, InterruptedException {
24          // split the line into [list of words]
25          String[] line = value.toString().split("\\s+");
26          String result = new String("");
27          for (String word : line) {
28            if (!stopwords.contains(word)) {
29              result = result.concat(word + " ");
30              }
31            }
32        context.write(NullWritable.get(), new Text(result));
33      }
34  }
```

Output:

```
1  This is first sentence contains some stopwords
2  Second sentence with stopword here another here
3  Third sentence stopword example
```

1.4 Data Organization Patterns

Data organization patterns refer to templates for reorganizing/splitting the input data into subsets (binning, shuffling). The output of an application with an organization pattern usually appears as input of another application(s).

1.4.1 Binning

The binning pattern refers to a template for organizing/moving the input records into categories, useful to partition a large data set into distinct, smaller data sets ("bins") containing similar records (each partition appears usually as the input for a subsequent analysis). In the binning pattern (map-only job), 1. the Driver sets the list of "bins/output files" by means of `MultipleOutputs`, and 2. the Mappers, for each input `(key, value)` record, select the proper output bin/file and emit a `(key, value)` pair in such file, with:

- `key`: key of the input pair;
- `value`: value of the input pair;

The binning pattern do not rely on Combiner or Reducer. Some examples: partition an input data set containing heterogeneous data to focus only on a specific subsets.

Exercise. Split the readings of a sensor according to the year of the measurement. Input: a set of textual files (`txt`) containing the temperature measurements from a sensor over differen years. Each line of the files appears in the format: `YYYY-MM-DD,HOUR,TEMPERATURE`. Output: report 1. a set of files with the prefix `2016` containing the lines of the input files with a year value equal to

2016, and 2. a set of files with the prefix 2017 containing the lines of the input files with a year value equal to 2017.

Input:

```
2016-12-01,14:00,20.5
2016-12-01,14:00,30.2
2016-12-02,14:10,11.5
2016-12-02,14:10,41.5
2017-01-01,14:00,20.5
2017-01-01,14:00,30.2
2017-01-02,14:10,11.5
2017-01-02,14:10,41.5
```

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    // set multiple outputs
    MultipleOutputs.addNamedOutput(job, "2016",
                                   TextOutputFormat.class,
                                   LongWritable.class,
                                   Text.class);
    MultipleOutputs.addNamedOutput(job, "2017",
                                   TextOutputFormat.class,
                                   LongWritable.class,
                                   Text.class);
    ...
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    // set mapper class
    job.setMapperClass(MapperBigData.class);
    job.setMapOutputKeyClass(LongWritable.class);
    job.setMapOutputValueClass(Text.class);
    // set number of reducers
    job.setNumReduceTasks(0); // -> map-only job
    // execute the job and wait for completion
    if (job.waitForCompletion(true) == true)
      exitCode = 0;
    else
      exitCode = 1;
    return exitCode;
  }
  /* main method of the driver class */
  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);
  }
}
```

```
/* the mapper class */
class MapperBigData extends Mapper <LongWritable, Text, LongWritable, Text> {
  // set a MultiOutputs object
  private MultipleOutputs<LongWritable, Text> mos = null;
  /* the setup method */
  protected void setup(Context context) {
  mos = new MultipleOutputs<LongWritable, Text>(context);
  }
  /* the map method */
  protected void map(LongWritable key, Text value,
      Context context) throws IOException, InterruptedException {
      String[] line = value.toString().toLowerCase().split(",");
```

```
13        String date = line[0]; // -> "2016-12-01"
14        String year = date.split("-")[0]; // -> "2016"
15        if (Integer.parseInt(year) == 2016) {
16          mos.write("2016", key, value);
17        }
18        else {
19          mos.write("2017", key, value);
20        }
21    }
22    /* the cleanup method */
23    protected void cleanup(Context context) throws IOException, InterruptedException {
24      // close the MultiOutputs
25      mos.close();
26    }
27  }
```

Output (1) - 2016:

```
1  0   2016-12-01,14:00,20.5
2  22  2016-12-01,14:00,30.2
3  44  2016-12-02,14:10,11.5
4  66  2016-12-02,14:10,41.5
```

Output (2) - 2017:

```
1  88  2017-01-01,14:00,20.5
2  110 2017-01-01,14:00,30.2
3  132 2017-01-02,14:10,11.5
4  154 2017-01-02,14:10,41.5
```

1.4.2 Shuffling

The shuffling pattern refers to a template for randomizing the order of the data (records), useful to randomize the order of the data for anonymization reasons or for selecting a subset of random data (records). In the shuffling pattern, the Mappers emit a single `(key, value)` pair for each input record, with:

- `key`: a random key (i.e., a random number);
- `value`: the input record;

Afterwards, the Reducers emit a single `(key, value)` pair for each value in the `[list of values]` of the input `(key, [list of values])` pair, with:

- `key`: the (random) input record;
- `value`: the null value;

1.5 Metapatterns

Metapatterns refer to templates for organizing the workflow of a complex application executing many jobs.

1.5.1 Job Chaining

The job chaining pattern refers to a template for executing a sequence of jobs (synchronization), useful to manage the workflow of complex applications, i.e. relating to many phases (iterations), with each phase implementable by means of a proper MapReduce job. In the job chaining pattern, the (single) Driver contains the workflow of the application and executes the jobs in the proper order, each with an appropriate Mapper and Reducer.

1.6 Join Patterns

Join patterns refer to templates for implementing the join operators of the relational algebra (i.e., the join operators of traditional relational databases).

1.6.1 Reduce Side Natural Join

The reduce side natural join pattern refers to a template for joining the content of two relations (i.e., relational tables). In the reduce side natural join pattern, the (two) Mappers, one for each table, emit a single `(key, value)` pair for each input record, with:

- `key`: the value of the common attribute(s);
- `value`: the concatenation of the table name of the current record and the content of the current record;

Suppose to join the tables:

- `USERS`: with schema `user id, name, surname`;
- `LIKES`: with schema `user id, movie genre`;

The record `user id = u, name = Paolo, surname = Garza` of the `USERS` table generates the `(user id = u, 'Users: name = Paolo, surname = Garza')` pair, while the record `user id = u, movie genre = horror` of the `LIKES` table generates the `(user id = u 'Likes: genre = horror')` pair. Afterwards, the Reducers iterate over the values of each key (value of the common attribute) and compute the "local natural join" for the current key. For instance, the `(key, [list of values])` pair `(user id = u, ['User: name = Paolo, surname = Garza', 'Likes: genre = horror', 'Likes: genre = adventure']` generates the following output `(key,value)` pairs: `(user id = u, 'name = Paolo, surname = Garza, genre = horror')`, `(user id = u, 'name = Paolo, surname = Garza, genre = adventure')`. For each key, the Reducers output `(key,value)` pairs with:

- `key`: the null value;
- `value`: the "local natural join" for the current key;

Exercise 28. Mapping Question-Answer(s). Input: 1. a large textual file with each line containing a single question in the format `question id, timestamp, text of the question`, and 2. a large textual file with each line containing a single answer in the format `answer id, question id, timestamp, text of the answer`. Output: report a single line for each pair `(question, answer)` with the format `question id, text of the question, answer id, text of the answer`.

Input (1) - `questions`:

```
Q1,2015-01-01,What is ..?
Q2,2015-01-03,Who invented ..
```

Input (2) - `answers`:

```
A1,Q1,2015-01-02,It is ..
A2,Q2,2015-01-03,John Smith
A3,Q1,2015-01-05,I think it is ..
```

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    inputPath1 = new Path(args[1]);
    inputPath2 = new Path(args[2]);
    ...
    MultipleInputs.addInputPath(job, inputPath1, TextInputFormat.class, MapperType1BigData.class);
    MultipleInputs.addInputPath(job, inputPath2, TextInputFormat.class, MapperType2BigData.class);
    ...
    // set map output key and value classes (Text, Text)
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);
    // set reducer class
    job.setReducerClass(ReducerBigData.class);
    // set reduce output key and value classes (NullWritable, Text)
    job.setOutputKeyClass(NullWritable.class);
    job.setOutputValueClass(Text.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 method of the driver class */
  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);
  }
}
```

```
/* the mapper class */
class MapperType1BigData extends Mapper <LongWritable, Text, Text, Text> {

  /* the map method */
  protected void map(LongWritable key, Text value,
      Context context) throws IOException, InterruptedException {
```

```
        String[] line = value.toString().split(",");
        String QID = line[0];
        String question = line[2];
        context.write(new Text(QID), new Text("Q:" + QID + "," + question));
    }
}
```

```
/* the mapper class */
class MapperType2BigData extends Mapper <LongWritable, Text, Text, Text> {

    /* the map method */
    protected void map(LongWritable key, Text value,
        Context context) throws IOException, InterruptedException {
        String[] line = value.toString().split(",");
        String QID = line[1];
        String AID = line[0];
        String answer = line[3];
        context.write(new Text(QID), new Text("A:" + AID + "," + answer));
    }
}
```

```
/* the reducer class */
class ReducerBigData extends Reducer <Text, Text, NullWritable, Text> {
    /* the reduce method */
    @Override
    protected void reduce(Text key, Iterable<Text> values,
        Context context) throws IOException, InterruptedException {
                String record;
                ArrayList<String> answers = new ArrayList<String>();
                String question = null;
                for (Text value : values) {
                    String line = value.toString();
                    if (line.startsWith("Q:") == true) {
                        record = line.replaceFirst("Q:", "");
                        question = record;
                    }
                    else {
                        record = line.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));
                }
    }
}
```

Output:

```
Q1,What is ..?,A3,I think it is ..
Q1,What is ..?,A1,It is ..
Q2,Who invented ..,A2,John Smith
```

1.6.2 Map Side Natural Join

The map side natural join pattern refers to a template for joining the content of two relations (e.g., relational tables) of the type:

- 1: large relation table;
- 2: small relation table (enough to rely on the main memory);

In the map side natural join pattern (map-only job), the Mapper class processes the content of the large table, i.e. receives a single `(key,value)` pair for each input record of the large table, and compute a join operation with the small table. The pattern leverages the distributed cache approach to provide a copy of the small table to each Mapper. Each Mapper performs the "local natural join" between the current record (of the large table) and the records of the small table (in the distributed cache). Each Mapper loads the content of the small table (file) at the execution of the setup method. For each input record, the Mappers output a `(key,value)` pair with:

- `key`: the null value;
- `value`: the "local natural join" between the current record (of the large table) and the records of the small table (in the distributed cache);

Exercise 27. Categorization rules. Input: 1. a large textual file (set of records) with each line containing the information about a single user in the format `user id, name, surname, gender, year of birth, city, education`, and 2. a small file (set of business rules useful to assign each user to a category) with each line containing a business rule in the format `gender = <value> and year of birth = <value> -> category`. Take into account the mutually exclusivity of the rules. Output: report a record for each user containing the original information about the user plus the category of belonging by means of the business rules.

Input (1):

```
User#1,John,Smith,M,1934,New York,Bachelor
User#2,Paul,Jones,M,1956,Dallas,College
User#3,Jenny,Smith,F,1934,Philadelphia,Bachelor
User#4,Laura,White,F,1926,New York,Doctorate
```

Input (2) - `businessrules.txt`:

```
Gender=M and YearOfBirth=1934 -> Category#1
Gender=M and YearOfBirth=1956 -> Category#3
Gender=F and YearOfBirth=1934 -> Category#2
Gender=F and YearOfBirth=1956 -> Category#3
```

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf);
    // add the HDFS file in the distributed cache
    job.addCacheFile(new Path("businessrules.txt").toUri());
    ...
    // set number of reducers
    job.setNumReduceTasks(0); // -> map-only job
    // execute the job and wait for completion
    if (job.waitForCompletion(true) == true)
      exitCode = 0;
```

```
    else
      exitCode = 1;
    return exitCode;
  }
  /* main method of the driver class */
  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);
  }
}
```

```
import java.io.BufferedReader;
...
import org.apache.hadoop.mapreduce.Mapper;

import java.util.regex.Matcher; // -> to perform match operations on a character sequence
import java.util.regex.Pattern; // -> to process regular expressions

/* the mapper class */
public class MapperBigData extends Mapper<LongWritable, Text, NullWritable, Text> {

    private Map<String, String> businessrules;

    @Override
    protected void setup(Context context) throws IOException, InterruptedException {
        businessrules = new HashMap<>();
        URI[] urisCachedFiles = context.getCacheFiles();
        BufferedReader file = new BufferedReader(
            new FileReader(new File(new Path(urisCachedFiles[0].getPath()).getName()))
        );
        Pattern pattern = Pattern.compile("Gender=(\\w) and YearOfBirth=(\\d{4}) -> Category#(\\d)");

        String line;
        while ((line = file.readLine()) != null) {
            Matcher matcher = pattern.matcher(line);
            if (matcher.find()) {
                String gender = matcher.group(1);
                String year = matcher.group(2);
                String category = matcher.group(3);
                String key = gender + year;
                businessrules.put(key, category);
            }
        }
        file.close();
    }

  /* the map method */
  protected void map(LongWritable key, Text value,
      Context context) throws IOException, InterruptedException {
        String[] fields = value.toString().split(",");
        String ruleKey = fields[3] + fields[4];
        String category = businessrules.getOrDefault(ruleKey, "Unknown");

        StringBuilder result = new StringBuilder();
        for (String field : fields) {
            result.append(field).append(",");
        }
        result.append(category);

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

Output:

```
User#1,John,Smith,M,1934,New York,Bachelor,1
User#2,Paul,Jones,M,1956,Dallas,College,3
```

```
User#3,Jenny,Smith,F,1934,Philadelphia,Bachelor,2
User#4,Laura,White,F,1926,New York,Doctorate,Unknown
```

Exercise 29. Input: 1. a large textual file (set of records) with each line containing information about a single user in the format `user id, name, surname, gender, year of birth, city, education`, and 2. a large textual file with `(user id, movie genre)` pairs. Output: report a record for each user who likes both comedy and adventure movies (containing the original information about the user's gender and the user's birth year). Remove duplicate data.

Input (1):

```
User#1,John,Smith,M,1934,New York,Bachelor
User#2,Paul,Jones,M,1956,Dallas,College
User#3,Jenny,Smith,F,1934,Philadelphia,Bachelor
User#4,Laura,White,F,1926,New York,Doctorate
```

Input (2):

```
User#1,Commedia
User#1,Adventure
User#1,Drama
User#2,Commedia
User#2,Crime
User#3,Commedia
User#3,Horror
User#3,Adventure
```

MapReduce program:

```
/* the driver class */
public class DriverBigData extends Configured implements Tool {
  /* run method of the driver class */
  @Override
  public int run(String[] args) throws Exception {
    ...
    numberOfReducers = Integer.parseInt(args[0]);
    inputPath1 = new Path(args[1]);
    inputPath2 = new Path(args[2]);
    outputDir = new Path(args[3]);
    ...
    Configuration conf = this.getConf();
    Job job = Job.getInstance(conf);
    job.setJobName("Exercise 29");
    MultipleInputs.addInputPath(job, inputPath1, TextInputFormat.class, MapperType1BigData.class);
    MultipleInputs.addInputPath(job, inputPath2, TextInputFormat.class, MapperType2BigData.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(NullWritable.class);
    job.setOutputValueClass(Text.class);
    ...
    // execute the job and wait for completion
    if (job.waitForCompletion(true) == true)
      exitCode = 0;
    else
      exitCode = 1;
    return exitCode;
  }
  /* main method of the driver class */
  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);
  }
}
```

```
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
     // user id, name, surname, gender, year of birth, city, education
     String[] fields = value.toString().split(",");
     String user = fields[0];
     String gender = fields[3];
     String year = fields[4];
     // key = user id
     // value = U:gender,year
     // U: to specify the emission of the pair
     // analyzing the users table
     context.write(new Text(user),
                   new Text("U:" + gender + "," + year));
    }
}
```

```
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
     // user id, movie genre
     String[] fields = value.toString().split(",");
     String user = fields[0];
     String genre = fields[1];
     // key = user id
     // value = L
     // L: to specify the emission of the pair
     // analyzing the likes file
     // emit the pair if and only if the genre = Commedia or Adventure
     if (genre.compareTo("Commedia") == 0 || genre.compareTo("Adventure") == 0) {
         context.write(new Text(user), new Text("L"));
         }
    }
}
```

```
class ReducerBigData extends Reducer<
                    Text,         // input key type
                    Text,         // input value type
                    NullWritable, // 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 {

    int counter;
```

```
12      String data = null;
13
14      counter = 0;
15      for (Text value : values) {
16
17        String record = value.toString();
18        counter++;
19        if (record.startsWith("U") == true) {
20          // the user data record
21          data = record.replaceFirst("U:", "");
22        }
23      }
24      // emit a pair (null, user data) if the number of elements == 3
25      // (2 likes and 1 user data record)
26      if (counter == 3) {
27        context.write(NullWritable.get(), new Text(data));
28      }
29    }
30  }
```

Output:

```
1  M,1934
2  F,1934
```

Chapter 2

Apache Spark

The Apache Spark framework revolves around the concept of a resilient distributed dataset (RDD), i.e. a fault-tolerant collection of elements executable in parallel. More in detail, a Spark RDD appears as an immutable, resilient, and distributable collection of objects (records). The framework splits each RDD into partitions to parallelize and execute the code on each RDD partition in isolation. The Spark application represents a driver program executing parallel operations on a cluster.

2.1 Structure of a Spark RDD-based program in Hadoop

The framework provides the connection between the driver and the cluster via the Spark Context object or, with reference to the appropriate programming language (**Python**), the `SparkContext` class. The RDD-based program retrieves the class by passing the configuration object as a parameter to the class constructor:

```
1  # create a configuration object and set the name of the application
2  conf = SparkConf().setAppName("Application name")
3
4  # create a Spark Context object
5  sc = SparkContext(conf = conf)
```

The `SparkContext.getOrCreate(conf)` method represents an alternative approach to retrieve the configuration object. Each Spark application relies on a single `SparkContext` object, so in case the object already exists for the application, the above method returns the current `SparkContext` object, and otherwise, a novel `SparkContext` object.

2.1.1 RDD creation

The developers of the RDD-based program create a single RDD (or many RDDs) by:

- 1. loading an external dataset (e.g., the content of a folder, a single file, a database table, etc.);
- 2. parallelizing a local collection of objects (e.g., a Python collection).

The creation of an RDD from an input textual file (1.) require the `textFile(name)` method of the `SparkContext` class, which returns an RDD of `String` lines, one for each line of the input textual file. By default, for an HDFS-type input file, the framework splits the RDD into partitions equal to the number of HDFS blocks/chunks.

```
# build an RDD of strings from the input textual file myfile.txt
# each element of the RDD represents a line of the input file
inputFile = "myfile.txt"
lines = sc.textFile(inputFile)
```

No computation occurs upon invocation of the `sc.textFile()` method since the Spark framework reads data from the input file only when necessary (i.e. when applying an action on the `lines` RDD, or on some descendant).

The creation of an RDD from a folder containing textual files (1.) require the `textFile(name)` method of the `SparkContext` class, which returns an RDD of `String` lines, one for each line of the input textual files within the folder.

```
# build an RDD of strings from all the files in myfolder
# each element of the RDD represents a line of the input files
inputFolder = "myfolder/"
lines = sc.textFile(inputFolder)
```

The framework enables developers to manually set the (minimum) number of partitions by means of the `textFile(name, minPartitions)` method of the `SparkContext` class, useful to increase the parallelization of the application. Take into account that for HDFS-type files the framework requires a value for the number of partitions, i.e. `minPartitions`, greater than the number of blocks/chunks.

```
# build an RDD of strings from the input textual file myfile.txt
# manually set the number of partitions to 4
# each element of the RDD represents a line of the input file
inputFile = "myfile.txt"
lines = sc.textFile(inputFile, 4)
```

The creation of an RDD from a "local" Python collection/list of local Python objects (2.) require the `parallelize(c)` method of the `SparkContext` class, which returns an RDD of objects of the same type of objects of the input Python collection `c` (one object for each element of the input collection). The Spark frameworks attempts to set the number of partitions according to the cluster's characteristics.

```
# create a local python list
inputList = ['First element', 'Second element', 'Third element']
# build an RDD of Strings from the local list
# the number of partitions appears set automatically by Spark
# one element of the RDD for each element of the local list
distRDDList = sc.parallelize(inputList)
```

Again, no computation occurs upon invocation of the `sc.parallelize()` method since the Spark framework reads data from the input file only when necessary (i.e. when applying an action on the `distRDDList` RDD, or on some descendant). The framework enables developers to manually set the (minimum) number of partitions by means of the `parallelize(c, numSlices)` method of the `SparkContext` class.

```
# create a local python list
inputList = ['First element', 'Second element', 'Third element']
# build an RDD of Strings from the local list.
# set the number of partitions to 3
# one element of the RDD for each element of the local list
distRDDList = sc.parallelize(inputList, 3)
```

2.1.2 RDD storage

The developers of the RDD-based program store the content of a single RDD in:

- 1. textual (HDFS) files;
- 2. a local python variable of the driver.

The storage of an RDD in textual (HDFS) files (1.) require the `saveAsTextFile(path)` method of the RDD class, which stores each object of the RDD in one line of the output files of the output folder (one output file for each partition of the input RDD).

```
# store the content of linesRDD in the output folder
# the method stores each element of the RDD in one line of the textual files of the output folder
outputPath = "risFolder/"
linesRDD.saveAsTextFile(outputPath);
```

Take into account that the `saveAsTextFile()` operation represents an action, hence Spark computes the content of the `linesRDD` RDD upon invocation. Moreover, note that the output folder contains one textual file for each partition of the `linesRDD` RDD (each output file contains the elements of one partition).

The storage of an RDD in a local Python variable of the driver (2.) require the `collect()` method of the RDD class, which returns a local Python list of objects containing the same objects of the RDD (**pay attention to the size of the RDD to not overcome the storage availability of the Python variable**).

```
# retrieve the content of the linesRDD and store it in a local python list
# the local python list contains a copy of each element of linesRDD
contentOfLines = linesRDD.collect();
```

Take into account that the framework allocates the local python variable, i.e. `contentOfLines`, in the main memory of the driver process/task, and the RDD of strings, i.e. `linesRDD`, across the nodes of the cluster.

2.1.3 RDD operations

The RDD architecture supports the subsequent types of operations:

- **Transformations**: applying transformation operations over the elements of one or more input RDD(s) and storing the result on an output RDD. Due to the immutability characteristic of RDDs, transformation operations maintain the contents of the input RDD(s) unalterable. The framework computes/execute the transformations only at the execution of an action operation on the output RDD, or some descendent. Therefore, when executing a transformation, Spark maintains only track of the dependency between the input RDD(s) and the output RDD coming from the transformation, without computing the content of the output RDD.

- **Actions**: applying action operations over the elements of one or more input RDD(s) and 1. storing the action result(s) on one or more local (Python) variable(s) or 2. writing the action result(s) on a storage output file/folder or on a distributable file system (**pay attention to the size of the results to not overcome the storage availability of the driver program**).

For example, consider the following lines of code:

```
1  conf = SparkConf().setAppName("Spark Application")
2  sc = SparkContext(conf = conf)
3  # read the content of a log file
4  inputRDD = sc.textFile("log.txt")
5
6  # select the rows containing the word 'error'
7  errorsRDD = inputRDD.filter(lambda line: line.find('error') >= 0)
8  # RDD -> RDD (transformation)
9
10 # select the rows containing the word 'warning'
11 warningRDD = inputRDD.filter(lambda line: line.find('warning') >= 0)
12 # RDD -> RDD (transformation)
13
14 # union of errorsRDD and warningRDD
15 # the framework associates the result with a novel RDD: badLinesRDD
16 badLinesRDD = errorsRDD.union(warningRDD)
17 # RDDs -> RDD (transformation)
18
19 # remove duplicates lines (i.e., those lines containing both 'error' and 'warning')
20 uniqueBadLinesRDD = badLinesRDD.distinct()
21 # RDD -> RDD (transformation)
22
23 # count the number of bad lines by applying the count() action
24 numBadLines = uniqueBadLinesRDD.count()
25 # RDD -> Python variable (action)
26
27 # print the result on the standard output of the driver
28 print("Lines with problems:", numBadLines)
```

The application reads the input log file only at the invocation of the `count()` action, i.e. the first action of the program. The `filter()`, `union()`, and `distinct()` operations represent transformations, processing the content of one or more input RDD(s) to return the result on an output RDD (the `textFile()` instead operates on the input log file, not an RDD).

2.1.4 Pass functions to Transformations and Actions

Many transformations (and some actions) rely on user-specifiable functions to set the type of "transformation" to apply over the elements of the "input" RDD. For example, the `filter()` transformation selects the elements of the input RDD satisfying a user-specifiable constraint, i.e. a boolean function to apply over each element of the input RDD. Each language holds an appropriate solution to pass functions to Spark's transformations and actions. With reference to the **Python** language:

- **lambda functions/expressions**: simple functions writable as a single expression;
- **local user-definable functions** (local `defs`): for multi-statement functions or statements computing values without return.

For example, take into account the following lines of code:

```
conf = SparkConf().setAppName("Spark Application")
sc = SparkContext(conf = conf)
# read the content of a log file
inputRDD = sc.textFile("log.txt")
# select the rows containing the word 'error'
errorsRDD= inputRDD.filter(lambda line: line.find('error') >= 0)
```

The `filter()` transformation applies the lambda function on each object of the `inputRDD` RDD. At each iteration, the transformation stores the current object in the output `errorsRDD` RDD in case the lambda function returns `True` and discards the object otherwise (`False`). Another way to achieve the same objective comes up via local user-definable functions (local `defs`):

```
def function(line):
    if line.find('error') >= 0:
        return True
    else:
        return False
# read the content of a log file
inputRDD = sc.textFile("log.txt")
# select the rows containing the word 'error'
errorsRDD = inputRDD.filter(function)
```

The framework automatically executes the `function` function over each object of the `inputRDD` RDD.

2.2 Basic Transformations

The framework provides some basic transformations to analyze the content of a single input RDD and return an output RDD, e.g., `filter()`, `map()`, `flatMap()`, `distinct()`, `sample()`. The framework also provides some other basic transformations to analyze the content of two (input) RDDs and return an output RDD, e.g., `union()`, `intersection()`, `substract()`, `cartesian()`, which require two input RDDs: one to execute the method on and the other to pass as a method parameter. Pay attention to the data types, since only the cartesian transformation enables 'mixed' data types.

2.2.1 Filter Transformation

The filter transformation returns an output RDD containing only the elements of the "input" RDD satisfying a user-specifiable condition. The filter transformation relies on the `filter(f)` method of the RDD class, with a function `f` returning a boolean value as parameter of the method. The function `f` contains the code relating to the condition to apply on each element of the "input" RDD. If the current element satisfies the condition, the function returns `True` and the operation saves the element. Otherwise, the function returns `False` and the operation discards the element.

Example. Create an RDD from a log file, and then create a novel RDD containing only the lines of the log file containing the word 'error'.

```
.........
# read the content of a log file
inputRDD = sc.textFile("log.txt")

# select the rows containing the word 'error'
errorsRDD = inputRDD.filter(lambda e: e.find('error') >= 0)
```

Example. Create an RDD of integers containing the values `[1, 2, 3, 3]`, and then create a novel RDD containing only the values greater than 2.

```
.........
# create an RDD of integers and load the values 1, 2, 3, 3 in this RDD
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList);

# select the values greater than 2
greaterRDD = inputRDD.filter(lambda num: num > 2)
```

or, alternatively:

```
.........
# define the function for the filter transformation
def greaterThan2(num):
    return num > 2
# create an RDD of integers and load the values 1, 2, 3, 3 in this RDD
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList);

# select the values greater than 2
greaterRDD = inputRDD.filter(greaterThan2)
```

Exercise 30. Input: a log of a web server (i.e., a textual file), with each line of the file referencing to a URL request. Output: the lines containing the word "google". Store the output in an HDFS folder.

Input:

```
66.249.69.97 - - [24/Sep/2014:22:25:44 +0000] "GET http://www.google.com/bot.html"
66.249.69.97 - - [24/Sep/2014:22:26:44 +0000] "GET http://www.google.com/how.html"
66.249.69.97 - - [24/Sep/2014:22:28:44 +0000] "GET http://dbdmg.polito.it/course.html"
71.19.157.179 - - [24/Sep/2014:22:30:12 +0000] "GET http://www.google.com/faq.html"
66.249.69.97 - - [24/Sep/2014:31:28:44 +0000] "GET http://dbdmg.polito.it/thesis.html"
```

Spark program:

```
from pyspark import SparkConf, SparkContext

```

```
conf = SparkConf().setAppName("Exercise 30")
sc = SparkContext(conf = conf)

import sys

# set input and output paths
inputPath = sys.argv[1]
outputPath = sys.argv[2]

# read the content of the input file
# each element/string of the logRDD corresponds to one line of the input file
logRDD = sc.textFile(inputPath)

# select the elements of the RDD satisfying the filter condition
googleRDD = logRDD.filter(lambda logLine: logLine.lower().find("google") >= 0)

# store the result in the output folder
googleRDD.saveAsTextFile(outputPath)
```

Output:

```
66.249.69.97 - - [24/Sep/2014:22:25:44 +0000] "GET http://www.google.com/bot.html"
66.249.69.97 - - [24/Sep/2014:22:26:44 +0000] "GET http://www.google.com/how.html"
71.19.157.179 - - [24/Sep/2014:22:30:12 +0000] "GET http://www.google.com/faq.html"
```

2.2.2 Map Transformation

The map transformation returns an output RDD by applying a function `f` on each element of the "input" RDD. The output RDD contains exactly one single element `y` for each element `x` of the "input" RDD. The operation computes the value of `y` by applying a user-definable function `f` on `x`, i.e. `y = f(x)` (the data type of `y` may differs from the data type of `x`). The map transformation relies on the `map(f)` method of the RDD class, with a function `f` implementing the transformation as parameter of the method. The function `f` contains the code relating to the mapping to apply on each element of the "input" RDD. The function returns exactly one single novel element for each input element of the "input" RDD.

Example. Create an RDD from a textual file containing the surnames of a list of users (each line of the file contains one surname), and then create a novel RDD containing the length of each surname.

```
.........
# read the content of the input textual file
inputRDD = sc.textFile("usernames.txt")

# compute the lengths of the input surnames
lenghtsRDD = inputRDD.map(lambda line: len(line))
```

2.2.3 FlatMap Transformation

The flat map transformation returns an output RDD by applying a function `f` on each element of the "input" RDD. The output RDD contains a list of elements `[y]` for each element `x` of the "input" RDD. The operation computes the value of the list `[y]` by applying a user-definable function `f` on `x`, i.e. `[y] =`

`f(x)` (the data type of `y` may differs from the data type of `x`). The operation returns as final result the concatenation of the list of values coming from the application of `f` over each element of the "input" RDD (the operation maintains the **duplicates**). The flat map transformation relies on the `flatMap(f)` method of the RDD class, with a function `f` implementing the transformation as parameter of the method. The function `f` contains the code relating to the mapping to apply on each element of the "input" RDD. The function returns exactly one single list of novel elements for each input element of the "input" RDD.

Example. Create an RDD from a textual file containing a generic text (each line of the input file contains many words), and then create a novel RDD containing the list of words, with repetitions, occurring in the input textual document. Each element of the novel RDD represents one of the words occurring in the input textual file. The words occurring multiple times in the input file appear multiple times, as distinct elements, also in the novel RDD.

```
1 .........
2 # read the content of the input textual file
3 inputRDD = sc.textFile("document.txt")
4
5 # compute/identify the list of words occurring in document.txt
6 listOfWordsRDD = inputRDD.flatMap(lambda line: line.split(''))
```

2.2.4 Distinct Transformation

The distinct transformation returns an output RDD containing the list of distinct elements (values) of the "input" RDD. The distinct transformation relies on the `distinct()` method of the RDD class, with no function as parameter of the method. For computing the result of the distinct transformation the `distinct()` method executes a shuffle operation (1.) to remove duplicates data from different input partitions and (2.) to re-partition the input data. The re-partitioning operation associates each repetition of the same input element with the same output partition via a hash function.

Example. Create an RDD from a textual file containing the names of a list of users (each line of the input file contains one name), and then create a novel RDD containing the list of distinct names occurring in the input file.

```
1 .........
2 # read the content of a textual input file
3 inputRDD = sc.textFile("names.txt")
4 # select the distinct names occurring in inputRDD
5 distinctNamesRDD = inputRDD.distinct()
```

Exercise 31. Input: a log of a web server (i.e., a textual file), with each line of the file referencing to a URL request. Output: the list of distinct IP addresses with connections to a google page (i.e., connections to URLs containing the term "www.google.com"). Store the output in an HDFS folder.

Input:

```
66.249.69.97 - - [24/Sep/2014:22:25:44 +0000] "GET http://www.google.com/bot.html"
66.249.69.97 - - [24/Sep/2014:22:26:44 +0000] "GET http://www.google.com/how.html"
66.249.69.97 - - [24/Sep/2014:22:28:44 +0000] "GET http://dbdmg.polito.it/course.html"
71.19.157.179 - - [24/Sep/2014:22:30:12 +0000] "GET http://www.google.com/faq.html"
66.249.69.95 - - [24/Sep/2014:31:28:44 +0000] "GET http://dbdmg.polito.it/thesis.html"
66.249.69.97 - - [24/Sep/2014:56:26:44 +0000] "GET http://www.google.com/how.html"
56.249.69.97 - - [24/Sep/2014:56:26:44 +0000] "GET http://www.google.com/how.html"
```

Spark program with `map()`:

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setAppName("Exercise 31")
sc = SparkContext(conf = conf)

import sys

# set input and output paths
inputPath = sys.argv[1]
outputPath = sys.argv[2]

# read the content of the input file
# each element/string of the logRDD corresponds to one line of the input file
logRDD = sc.textFile(inputPath)

# select the elements of the RDD satisfying the filter condition
googleRDD = logRDD.filter(lambda logLine: logLine.lower().find("google") >= 0)

# use map to select only the IP address (the first field before -)
IPsRDD = googleRDD.map(lambda logLine: logLine.split('-')[0])

# remove duplicates
distinctIPsRDD = IPsRDD.distinct()

# store the result in the output folder
distinctIPsRDD.saveAsTextFile(outputPath)
```

Spark program with `flatMap()`:

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setAppName("Exercise 31")
sc = SparkContext(conf = conf)

import sys

# set input and output paths
inputPath = sys.argv[1]
outputPath = sys.argv[2]

# read the content of the input file
# each element/string of the logRDD corresponds to one line of the input file
logRDD = sc.textFile(inputPath)

def filterAndExtractIP(line):
    # inizialize the list to return
    listIPs = []

    # if line contains www.google.com add the IP of this line in the list
    if line.lower().find("www.google.com") >= 0:
        IP = line.split('-')[0]
        listIPs.append(IP)

    # return listIPs
    return listIPs

# select only the elements of the RDD satisfying the filter
# and return the respective IPs
```

```
# those lines that do not contain "www.google.com" return an empy list
IPsRDD = logRDD.flatMap(filterAndExtractIP)

# remove duplicates
distinctIPsRDD = IPsRDD.distinct()

# store the result in the output folder
distinctIPsRDD.saveAsTextFile(outputPath)
```

Output:

```
66.249.69.97
71.19.157.179
56.249.69.97
```

2.2.5 SortBy Transformation

The sort by transformation returns an output RDD containing the same content of the input RDD in ascending order. The sort by transformation relies on the `sortBy(keyfunc)` method of the RDD class. The method branches into several sub-procedures:

1. `mapping`: the method maps each element of the input RDD into a novel value by applying the specific function (`keyfunc`);
2. `sorting`: the method sorts the input elements with respect to the values coming from the previous procedure, i.e. the invocation of `keyfunc` on the input values.

The `sortBy(keyfunc, ascending)` method of the RDD class enables specifying the sorting order of the values in the output RDD (ascending or descending) via the boolean parameter `ascending`:

- `ascending = True`: ascending
- `ascending = False`: descending

Example. Create an RDD from a textual file containing the names of a list of users (each line of the input file contains one name), and then create a novel RDD containing the list of users in order by name, according to the alphabetic order.

```
.........
# read the content of a textual input file
inputRDD = sc.textFile("names.txt")

# sort the content of the input RDD by name and store the result in a novel RDD
sortedNamesRDD = inputRDD.sortBy(lambda name: name)
```

Example. Create an RDD from a textual file containing the names of a list of users (each line of the input file contains one name), and then create a novel RDD containing the list of users in order by name length, i.e., the sort order relies on `len(name)`.

```
.........
# read the content of a textual input file
inputRDD = sc.textFile("names.txt")

# sort the content of the input RDD by len(name) and store the sorted in a novel RDD
sortedNamesLenRDD = inputRDD.sortBy(lambda name: len(name))
```

2.2.6 Sample Transformation

The sample transformation returns an output RDD containing a random sample over the elements (values) of the "input" RDD. The sample transformation relies on the `sample(withReplacement, fraction)` method of the RDD class, where 1. `withReplacement` sets the replacement paradigm of the random sample (`True` or `False`), and 2. `fraction` sets the predictable size of the random sample as a fraction of the "input" RDD's size (values in the range `[0,1]`).

Example. Create an RDD from a textual file containing a set of sentences (each line of the file contains one sentence), and then create a novel RDD containing a random sample of sentences (set the "without replacement" strategy and the fraction to 0.2, i.e., 20%).

```
.........
# read the content of a textual input file
inputRDD = sc.textFile("sentences.txt")

# create a random sample of sentences
randomSentencesRDD = inputRDD.sample(False, 0.2)
```

2.2.7 Union Transformation

The union transformation returns an output RDD containing the union (**with duplicates**) of the elements coming from two input RDDs. The union transformation relies on the `union(other)` method of the RDD class, where `other` refers to the second input RDD to analyze.

To remove duplicates, the framework enables developers to apply the `distinct()` transformation on the output of the `union()` transformation. Pay attention that `distinct()` represents a computational costly operation (shuffle operation), so the framework warns developers to exploit the `distinct()` transformation only in indispensable situations.

2.2.8 Intersection Transformation

The intersection transformation returns an output RDD containing the intersection (**without duplicates**) of the elements coming from two input RDDs, i.e. the elements occurring in both input RDDs. The intersection transformation relies on the `intersection(other)` method of the RDD class, where `other` refers to the second input RDD to analyze. Take into account that the `intersection(other)` method execute a shuffle operation to compare elements from different input partitions in order to return common elements.

2.2.9 Subtract Transformation

The subtract transformation returns an output RDD containing the elements appearing only in the first (1) input RDD (**with duplicates**). The subtract transformation relies on the `subtract(other)` method of the RDD class, where `other` refers to the second (2) input RDD to analyze. Take into account that the `subtract(other)` method execute a shuffle operation to compare elements from different input partitions in order to discard common elements.

2.2.10 Cartesian Transformation

The cartesian transformation returns an output RDD of pairs (tuples) containing each combination resulting from the composition of an element of the first (1) input RDD and an element of the second (2) input RDD. The cartesian transformation relies on the `cartesian(other)` method of the RDD class, where `other` refers to the second (2) RDD to analyze.

Example. Create two RDDs of integers, with `inputRDD1` containing the values `[1, 2, 2, 3, 3]` and `inputRDD2` containing the values `[3, 4, 5]`. Then, create three novel RDDs: `outputUnionRDD` containing the union of `inputRDD1` and `inputRDD2`, `outputIntersectionRDD` containing the intersection of `inputRDD1` and `inputRDD2`, and `outputSubtractRDD` containing the result of `inputRDD1` / `inputRDD2`.

```
.........
# create two RDD of integers
inputList1 = [1, 2, 2, 3, 3]
inputRDD1 = sc.parallelize(inputList1)
inputList2 = [3, 4, 5]
inputRDD2 = sc.parallelize(inputList2)
# create three novel RDDs by  union, intersection, and subtract
outputUnionRDD = inputRDD1.union(inputRDD2)
outputIntersectionRDD = inputRDD1.intersection(inputRDD2)
outputSubtractRDD = inputRDD1.subtract(inputRDD2)
```

Output:

```
outputUnionRDD: [1, 2, 2, 3, 3, 3, 4, 5]
outputIntersectionRDD: [3]
outputSubtractRDD: [1, 2, 2]
```

Example. Create two RDDs of integers, with `inputRDD1` containing the values `[1, 2, 2, 3, 3]` and `inputRDD2` containing the values `[3, 4, 5]`. Then, create a novel RDD containing the cartesian product of `inputRDD1` and `inputRDD2`.

```
.........
# create two RDD of integers
inputList1 = [1, 2, 2, 3, 3]
inputRDD1 = sc.parallelize(inputList1)
inputList2 = [3, 4, 5]
inputRDD2 = sc.parallelize(inputList2)
# compute the cartesian product
outputCartesianRDD = inputRDD1.cartesian(inputRDD2)
```

Example. Create two RDDs, with `inputRDD1` containing the values `[1, 2, 3]` and `inputRDD2` containing the values `['A', 'B']`. Then, create a novel RDD containing the cartesian product of `inputRDD1` and `inputRDD2`.

```
.........
#create an RDD of integers and an RDD of strings
inputList1 = [1, 2, 3]
inputRDD1 = sc.parallelize(inputList1)
inputList2 = ["A", "B"]
inputRDD2 = sc.parallelize(inputList2)
# compute the cartesian product
outputCartesianRDD = inputRDD1.cartesian(inputRDD2)
```

Output:

```
outputCartesianRDD: [(1, "A"), (1, "B"), (2, "A"), (2, "B"), (3, "A"), (3, "B")]
```

2.3 Basic Actions

The framework provides some basic actions to:

- retrieve and store the content of an RDD (or the result of a function applicable on an RDD) in a local Python variable of the driver program;
- retrieve and store the content of an RDD (or the result of a function applicable on an RDD) in an output folder or database;

e.g., `collect()`, `count()`, `countByValue()`, `take()`, `top()`, `takeSample()`, `reduce()`, `fold()`, `aggregate()`, `foreach()`.

2.3.1 Collect Action

The collect action returns a local Python list of objects containing the same objects of the input RDD (**pay attention to the size of the results to not overcome the storage availability of the driver program**). The collect action relies on the `collect()` method of the RDD class.

Example. Create an RDD of integers containing the values `[1, 2, 3, 3]`. Retrieve and store the values of the RDD in a local Python list.

```
.........
# create an RDD of integers and load the values 1, 2, 3, 3 in the RDD
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList)
# retrieve and store the elements of the inputRDD in a local python list
retrievedValues = inputRDD.collect()
```

Output:

```
retrievedValues = [1, 2, 3, 3]
```

2.3.2 Count Action

The count action returns the number of elements of the input RDD. The collect action relies on the `count()` method of the RDD class.

Example. Consider the textual files `document1.txt` and `document2.txt` and print the name of the file with more lines.

```
.........
# read the content of the two input textual files
inputRDD1 = sc.textFile("document1.txt")
inputRDD2 = sc.textFile("document2.txt")
# count the number of lines of the two files = number of elements of the two RDDs
numLinesDoc1 = inputRDD1.count()
numLinesDoc2 = inputRDD2.count()
if numLinesDoc1 > numLinesDoc2:
    print("document1.txt")
elif numLinesDoc2 > numLinesDoc1:
    print("document2.txt")
else:
    print("same number of lines")
```

2.3.3 CountByValue Action

The count by value action returns a local Python dictionary containing the information about the number of times each element occurs in the RDD, with:

- `keys`: the input elements;
- `values`: the frequencies of the elements.

The count by value action relies on the `countByValue()` method of the RDD class. The amount of necessary memory in the driver comes from the number of distinct elements/keys.

Example. Create an RDD from a textual file containing the first names of a list of users (each line contain one name), and then compute the number of occurrences of each name (store the result in a local variable of the driver).

Input - `names.txt`:

```
Alice
Bob
Charlie
Alice
Alice
Bob
Eve
```

Spark program:

```
.........
# read the content of the input textual file
namesRDD = sc.textFile("names.txt")
# compute the number of occurrencies of each name
namesOccurrences = namesRDD.countByValue()
```

Output:

```
namesOccurrences = {'Alice': 3, 'Bob': 2, 'Charlie': 1, 'Eve': 1}
```

2.3.4 Take Action

The take action returns a local Python list of objects containing the first `num` (parameter of the method) elements of the input RDD. The order of the elements in an RDD relies on the order of the elements in the file or collection of the RDD. The take action relies on the `take(num)` method of the RDD class.

Example. Create an RDD of integers containing the values `[1, 5, 3, 3, 2]`. Retrieve and store the first two values of the RDD in a local variable of the driver.

```
.........
# create an RDD of integers and load the values 1, 5, 3, 3,2 in the RDD
inputList = [1, 5, 3, 3, 2]
inputRDD = sc.parallelize(inputList)
# retrieve and store the first two elements of the inputRDD in a local python list
retrievedValues = inputRDD.take(2)
```

Output:

```
retrievedValues = [1, 5]
```

2.3.5 First Action

The first action returns a local Python object containing the first element of the input RDD. The order of the elements in an RDD relies on the order of the elements in the file or collection of the RDD. The first action relies on the `first()` method of the RDD class. The difference between `first()` and `take(1)` comes by the fact that: `first()` returns a single element (the first element of the RDD), while `take(1)` returns a list of elements containing one single element (the first element of the RDD).

Exercise 32. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: report the maximum value of PM10 and print the result on the standard output.

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 32")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
readingsRDD = sc.textFile(inputPath)

```

```
pm10ValuesRDD = readingsRDD.map(lambda reading: float(reading.split(",")[2]))
# [20.5, 30.1, 60.2, 20.4, 55.5, 52.5]
maxPM10Value = pm10ValuesRDD.sortBy(lambda PMValue: PMValue, ascending = False).first()
# 60.2

print(maxPM10Value)
```

Output:

```
60.2
```

2.3.6 Top Action

The top action returns a local Python list of objects containing the top `num` (parameter of the method) largest elements of the input RDD (descending order). The top action relies on the `top(num)` method of the RDD class.

Example. Create an RDD of integers containing the values `[1, 5, 3, 4, 2]`. Retrieve and store the top-2 greatest values of the RDD in a local Python list of the driver.

```
.........
# create an RDD of integers and load the values 1, 5, 3, 4,2 in the RDD
inputList = [1, 5, 3, 4, 2]
inputRDD = sc.parallelize(inputList)
# retrieve and store the top-2 elements of the inputRDD in a local python list
retrievedValues = inputRDD.top(2)
```

Output:

```
retrievedValues = [5, 4]
```

The top action enables also to order the top `num` largest elements of the input RDD according to a user-specifiable "sorting" function. The ordering relies on the `top(num, key)` method of the RDD class, with:

- `num`: the number of elements to return in the Python list;
- `key`: a function to apply on each input element before the comparison.

Example. Create an RDD of strings containing the values `['Paolo', 'Giovanni', 'Luca']`. Retrieve and store the 2 longest names (longest strings) of the RDD in a local Python list of the driver.

```
.........
# create an RDD of strings and load the values ['Paolo', 'Giovanni', 'Luca'] in the RDD
inputList = ['Paolo', 'Giovanni', 'Luca']
inputRDD = sc.parallelize(inputList)
# retrieve and store the 2 longest names of the inputRDD in a local python list
retrievedValues = inputRDD.top(2, lambda s: len(s))
```

Output:

```
retrievedValues = ['Giovanni', 'Paolo']
```

Exercise 33. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: report the top-3 maximum values of PM10

and print the result on the standard output.

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setAppName("Exercise 33")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
readingsRDD = sc.textFile(inputPath)

pm10ValuesRDD = readingsRDD.map(lambda reading: float(reading.split(",")[2]))
# [20.5, 30.1, 60.2, 20.4, 55.5, 52.5]
topPMsRDD = pm10ValuesRDD.top(3)
# [60.2, 55.5, 52.5]
print(topPMsRDD)
```

Output:

```
[60.2, 55.5, 52.5]
```

2.3.7 TakeOrdered Action

The take ordered action returns a local Python list of objects containing the `num` (parameter of the method) smallest elements of the input RDD (ascending order). The take ordered action relies on the `takeOrdered(num)` method of the RDD class.

Example. Create an RDD of integers containing the values [1, 5, 3, 4, 2]. Retrieve and store the 2 smallest values of the RDD in a local Python list of the driver.

```
.........
# create an RDD of integers and load the values 1, 5, 3, 4,2 in the RDD
inputList = [1, 5, 3, 4, 2]
inputRDD = sc.parallelize(inputList)
# retrieve and store the 2 smallest elements of the inputRDD in a local python list
retrievedValues = inputRDD.takeOrdered(2)
```

The take ordered enables also to order the `num` smallest elements of the input RDD according to a user-specifiable "sorting" function. The ordering relies on the `takeOrdered(num, key)` method of the RDD class, with:

- `num`: the number of elements to return in the Python list;
- `key`: a function to apply on each input element before the comparison.

Example. Create an RDD of strings containing the values `['Paolo', 'Giovanni', 'Luca']`. Retrieve and store the 2 shortest names (shortest strings) of the RDD in a local Python list of the driver.

```
.........
# create an RDD of strings and load the values 'Paolo', 'Giovanni', 'Luca'] in the RDD
inputList = ['Paolo', 'Giovanni', 'Luca']
inputRDD = sc.parallelize(inputList)
# retrieve and store the 2 shortest names of the inputRDD in a local python list
retrievedValues = inputRDD.takeOrdered(2, lambda s: len(s))
```

Output:

```
retrievedValues = ['Luca', 'Paolo']
```

Exercise 32. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: report the maximum value of PM10 and print the result on the standard output.

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 32")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
readingsRDD = sc.textFile(inputPath)

pm10ValuesRDD = readingsRDD.map(lambda PM10Reading: float(PM10Reading.split(',')[2]))
# [20.5, 30.1, 60.2, 20.4, 55.5, 52.5]
maxPM10Value = pm10ValuesRDD.takeOrdered(1, lambda value: - value)[0]
# 60.2

print(maxPM10Value)
```

Output:

```
60.2
```

2.3.8 TakeSample Action

The take sample action returns a local Python list of objects containing the `num` (parameter of the method) random elements of the input RDD, useful to set the seed. The take sample action relies on the `takeSample(withReplacement, num)` method of the RDD class, where the `withReplacement` parameter sets the replacement paradigm of the random sample (`True` or `False`).

Example. Create an RDD of integers containing the values `[1, 5, 3, 3, 2]`. Retrieve and store randomly, without replacement, 2 values from the RDD in a local Python list of the driver.

```
.........
# create an RDD of integers and load the values 1, 5, 3, 3, 2 in the RDD
inputList = [1, 5, 3, 3, 2]
inputRDD = sc.parallelize(inputList)
# retrieve and store randomly two elements of the inputRDD in a local python list
randomValues = inputRDD.takeSample(True, 2)
```

Output:

```
randomValues = [1, 2]
```

2.3.9 Reduce Action

The reduce action returns a single Python object coming from the combination of the input RDD objects via a user-specifiable (**associative** and **commutative**) "function". The reduce action relies on the `reduce(f)` method of the RDD class. In an iterative manner, the method (1.) retrieves two arbitrary elements from the input RDD, (2.) returns the output element (combination of the input elements), (3.) remove the "original" elements and (4.) overwrite/insert the output element. The method then stops when the final output appears with a single element (the combination of the input RDD objects). Take into account that the reduce action returns objects of the same data type/class of the input RDD.

Example. Create an RDD of integers containing the values `[1, 2, 3, 3]`, and compute the sum of the values occurring in the RDD (store the result in a local Python integer variable in the driver).

```
.........
# create an RDD of integers and load the values 1, 2, 3, 3 in the RDD
inputListReduce = [1, 2, 3, 3]
inputRDDReduce = sc.parallelize(inputListReduce)
# compute the sum of the values
sumValues = inputRDDReduce.reduce(lambda e1, e2: e1 + e2)
```

Example. Create an RDD of integers containing the values `[1, 2, 3, 3]`, and compute the maximum value occurring in the RDD (store the result in a local Python integer variable in the driver).

```
.........
# define the function for the reduce action
def computeMax(v1, v2):
    if v1 > v2:
        return v1
    else:
        return v2
# create an RDD of integers and load the values 1, 2, 3, 3 in the RDD
inputListReduce = [1, 2, 3, 3]
inputRDDReduce = sc.parallelize(inputListReduce)
# compute the maximum value
maxValue = inputRDDReduce.reduce(computeMax)
```

or, alternatively:

```
.........
# create an RDD of integers and load the values 1, 2, 3, 3 in the RDD
inputListReduce = [1, 2, 3, 3]
inputRDDReduce = sc.parallelize(inputListReduce)
# compute the maximum value
maxValue = inputRDDReduce.reduce(lambda e1, e2: max(e1, e2))
```

Exercise 32. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: report the maximum value of PM10 and print the result on the standard output.

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 32")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
readingsRDD = sc.textFile(inputPath)

pm10ValuesRDD = readingsRDD.map(lambda PM10Reading: float(PM10Reading.split(',')[2]))
maxPM10Value = pm10ValuesRDD.reduce(lambda value1, value2: max(value1, value2))

print(maxPM10Value)
```

Output:

```
60.2
```

2.3.10 Fold Action

The fold action returns a single Python object coming from the combination of the input RDD objects and a "zero" value via a user-specifiable (**associative**) "function". The fold action relies on the `fold(zeroValue, op)` method of the RDD class. In an iterative manner, the method (1.) retrieves two arbitrary elements from the input RDD, (2.) returns the output element (combination of the input elements via the `op` function), (3.) remove the "original" elements and (4.) overwrite/insert the output element. The method then stops when the final output appears with a single element (the combination of the input RDD objects). The fold action appears suitable to parallelize associative but non-commutative functions, e.g., concatenation of a list of strings.

Example. Create an RDD of strings containing the values `['This ', 'is', 'a ', 'test']`, and compute the concatenation of the values occurring in the RDD, from left to right (store the result in a local Python string variable in the driver).

```
.........
inputListFold = ['This ', 'is ', 'a ', 'test']
inputRDDFold = sc.parallelize(inputListFold)
# concatenate the input strings
finalString = inputRDDFold.fold('', lambda s1, s2: s1 + s2)
```

Output:

```
This is a test
```

2.3.11 Aggregate Action

The aggregate action returns a single Python object coming from the combination of the input RDD objects and an initial "zero" value via a user-specifiable (**associative**) "function". The aggregate action relies on the `aggregate(zeroValue, seqOp, combOp)` method of the RDD class. From the "input" RDD, containing objects of type `T`, the method returns an object of type `U` (with `T != U`), by means of:

- a "function" for merging an element of type `T` with an element of type `U` to return an output element of type `U`, i.e. to merge the elements of the input RDD and the accumulator of each partition: the `seqOp` function contains the code to combine the accumulator value (one accumulator for each partition) with the elements of each partition (the framework computes one "local" result per partition by recursively applying `seqOp`);

- a "function" for merging two elements of type `U` to return an output element of type `U`, i.e. to merge two elements of type `U` (partial results) coming from two different partitions: the `combOp` function contains the code to combine two elements of type U (partial results) coming from two different partitions (the framework computes the global final result by recursively applying `combOp`).

Suppose that L contains the list of elements of the "input" RDD, which appears split in a set of partitions, i.e., a set of lists $\{L_1, ..., L_n\}$. The aggregate action computes a partial result in each partition and then combines/merges the results, as follow:

1. aggregate the partial results in each partition, resulting in a set of partial results (of type `U`) $P = \{p_1, ..., p_n\}$;

2. apply the `combOp` function on a pair of elements p_1 and p_2 in P and return an output element p_{out};

3. remove the "original" elements p_1 and p_2 from P and then insert the element p_{out} in P;

4. if P contains only one value then return it as final result of the aggregate action. Otherwise, return to step 2.

Suppose that L_i represents the list of elements on the i-th partition of the "input" RDD and `zeroValue` represents the initial zero value. To compute the partial result over the elements in L_i the aggregate action operates as follows:

1. set the accumulator to `zeroValue` (accumulator = `zeroValue`);
2. apply the `seqOp` function on the accumulator and an element e_j in L_i to update the accumulator with the value coming from the `seqOp` function;
3. remove the "original" element e_j from L_i;
4. if L_i appears empty return the accumulator as (final) partial result p_i of the i-th partition. Otherwise, return to step 2.

Example. Create an RDD of integers containing the values `[1, 2, 3, 3]` and compute both 1. the sum of the values occurring in the input RDD and 2. the number of elements of the input RDD. Store in a local Python variable of the driver the average over the values of the input RDD.

```
.........
# create an RDD of integers and load the values 1, 2, 3, 3 in the RDD
inputListAggr = [1, 2, 3, 3]
inRDD = sc.parallelize(inputListAggr)
# instantiate the zero value
# use a tuple containing two values: (sum, number of elements)
zeroValue = (0, 0)
# compute the sum of the elements in inputRDDAggr and count them
sumCount = inRDD.aggregate(zeroValue,
                           lambda acc, el: (acc[0] + el, acc[1] + 1),
                           lambda p1, p2: (p1[0] + p2[0], p1[1] + p2[1])
                           )
# compute the average value
myAvg = sumCount[0] / sumCount[1]
# print the average on the standard output of the driver
print('Average:', myAvg)
```

Exercise 36. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: compute the average PM10 value and print the result on the standard output.

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 36")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
readingsRDD = sc.textFile(inputPath)

```

```
9  pm10ValuesRDD = readingsRDD.map(lambda PM10Reading: float(PM10Reading.split(',')[2]))
10
11 zeroValue = (0, 0)
12 sumCount = pm10ValuesRDD.aggregate(zeroValue,
13                                     lambda acc, el: (acc[0] + el, acc[1] + 1),
14                                     lambda p1, p2: (p1[0] + p2[0], p1[1] + p2[1])
15                                     )
16 # compute the average value
17 myAvg = sumCount[0] / sumCount[1]
18
19 print('Average:', myAvg)
```

Output:

```
1  Average: 39.86666666666667
```

2.4 RDDs of key-value pairs

The Spark framework supports also RDDs of key-value pairs, i.e., with reference to the appropriate programming language (**Python**), RDDs of tuples, with the first value as key part of the pair and the second value as value part of the pair. The framework provides RDDs of key-value pairs with specific operations, e.g. `reduceByKey()`, `join()`, etc, to analyze the content of a single group (key) at a time, but also with operations available for the "standard" RDDs, e.g. `filter()`, `map()`, `reduce()`, etc. The most of the applications rely on RDDs of key-value pairs. The operations available for RDDs of key-value pairs enable:

- "grouping" data by key;
- performing computation by key (i.e., by group).

The basic idea appears similar to the one of the MapReduce-based programs in Hadoop, but with more operations already available.

2.4.1 Creating RDD of key-value pairs

The developers create RDDs of key-value pairs from:

- other RDDs, by applying the `map()` or the `flatMap()` transformation;
- a Python in-memory collection of tuple (key-value pairs), by applying the `parallelize()` method of the `SparkContext` class.

2.4.2 Map Transformation

The map transformation enables the creation of RDDs of key-value pairs by applying a function `f` on each element of the input RDD. The output RDD of key-value pairs contains a tuple `y` for each element `x` of the "input" RDD, i.e. `y = f(x)`. The process relies on the standard `map(f)` method of the RDD class.

Example. Create an RDD from a textual file containing the first names of a list of users (each line of the file contains one first name), and then create an RDD of key-value pairs containing a list of pairs `(name, 1)`.

```
.........
# read the content of the input textual file
namesRDD = sc.textFile("names.txt")
# create an RDD of key-value pairs
nameOnePairRDD = namesRDD.map(lambda name: (name, 1))
```

2.4.3 FlatMap Transformation

The flat map transformation enables the creation of RDDs of key-value pairs by applying a function `f` on each element of the input RDD. The output RDD of key-value pairs contains a list of tuples `[y]` for each element `x` of the "input" RDD, i.e. `[y] = f(x)`. The process relies on the standard `flatMap(f)` method of the RDD class.

Example. Create an RDD from a textual file (each line of the file contains a set of words), and then create an RDD of key-value pairs (`PairRDD`) containing a list of pairs `(word, 1)` (one pair for each word occurring in the input document, with repetitions).

Input - `document.txt`:

```
Hello world
Hello Spark
```

Spark program:

```
.........
# read the content of the input textual file
linesRDD = sc.textFile("document.txt")
# ['Hello world', 'Hello Spark']

# create an RDD of key-value pairs according to the input document
# one pair (word,1) for each input word
wordOnePairRDD = linesRDD.flatMap(lambda line: map(lambda w: (w, 1), line.split(' ')))
# [('Hello', 1), ('world', 1), ('Hello', 1), ('Spark', 1)]
```

2.4.4 Parallelize

The parallelize method enables the creation of RDDs of key-value pairs from a local Python in-memory collection of tuples. The process relies on the standard `parallelize(c)` method of the `SparkContext` class, by which each element (tuple) of the local Python collection becomes a key-value pair in the output RDD.

Example. Create an RDD from a local Python list containing the following key-value pairs `("Paolo", 40)`, `("Giorgio", 22)` and `("Paolo", 35)`.

```
.........
# create the local python list
nameAge = [("Paolo",40), ("Giorgio", 22), ("Paolo", 35)]
```

```
4  # create the RDD of pairs from the local collection
5  nameAgePairRDD = sc.parallelize(nameAge)
```

2.4.5 Transformations on RDDs of key-value pairs

The Spark framework enables each of the previous "standard" transformations to process also on RDDs of key-value pairs, with "functions" operating on tuples instead of single elements (such as `int`, `string`, etc.). Moreover, the framework also provides some specific transformation for operating on RDDs of key-value pairs, e.g., `reduceByKey()`, `groupyKey()`, `mapValues()`, `join()`, and others.

2.4.6 ReduceByKey Transformation

The reduce by key transformation enables the creation of RDDs of key-value pairs by applying an (**associative** and **commutative**) function `f` on each `k`-compatible value in the input RDD of key-value pairs. The output RDD of key-value pairs contains one pair for each distinct key `k` of the input RDD of key-value pairs. The process relies on the `reduceByKey(f)` method of the RDD class. The method recursively executes the function `f` over the values of pairs compatible with a single key at a time until the reduction/combination of such values into a single value. The method then returns an RDD containing a number of key-value pairs equal to the number of distinct keys in the input RDD of key-value pairs. Similarly to the `reduce()` action, the `reduceByKey()` transformation aggregates values. However,

- the framework executes `reduceByKey()` (i.e. a **transformation**) on RDDs of key-value pairs and returns a set of key-value pairs;
- the framework executes `reduce()` (i.e. an **action**) on an RDD and returns a single value (in a local Python variable).

Take into account that the `reduceByKey()` method executes a shuffle operation for computing the result of the transformation. The framework stores in different partitions the result/value for each group/key.

Example. Create an RDD from a local Python list containing the pairs `("Paolo", 40)`, `("Giorgio", 22)`, `("Paolo", 35)`, with as key the first name of a user and as value his/her age. Create a novel RDD of key-value pairs containing one pair for each name and associate each name with the age of the youngest user with that name.

```
1  .........
2  # create the local python list
3  nameAge = [("Paolo",40), ("Giorgio", 22), ("Paolo", 35)]
4  # create the RDD of pairs from the local collection
5  nameAgePairRDD = sc.parallelize(nameAge)
6  # select for each name the lowest age value
7  youngestPairRDD = nameAgePairRDD.reduceByKey(lambda age1, age2: min(age1, age2))
```

Exercise 37. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: the maximum value of PM10 for each sensor. Store the result in an HDFS file.

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 37")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
outputPath = sys.argv[2]

readingsRDD = sc.textFile(inputPath)

# create an RDD of key-value pairs
# each pair contains a sensorId (key) and a PM10 value (value)
# the function of the map transformation returns a tuple
sensorsPM10ValuesRDD = readingsRDD.map(lambda PM10Reading: (PM10Reading.split(',')[0],
                                                            float(PM10Reading.split(',')[2])))

# apply the reduceByKey transformation to compute the maximum PM10 value for each sensor
sensorsMaxValuesRDD = sensorsPM10ValuesRDD.reduceByKey(lambda value1, value2: max(value1, value2))

# store the result in the output folder
sensorsMaxValuesRDD.saveAsTextFile(outputPath)
```

Output:

```
(s1,60.2)
(s2,52.5)
```

Exercise 38. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: the sensors with at least 2 readings with a PM10 value greater than the critical threshold 50. Store in an HDFS file the `sensor ids` of the output sensors and also the number of times each of those sensors results with a PM10 value greater than 50.

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 38")
```

```
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
outputPath = sys.argv[2]

readingsRDD = sc.textFile(inputPath)

# apply a filter transformation to select only the lines with PM10>50
readingsHighValueRDD = readingsRDD.filter(lambda PM10Reading: float(PM10Reading.split(',')[2]) > 50)

# create an RDD of key-value pairs
# each pair contains a sensor id (key) and +1 (value)
# the function of the map transformation returns a tuple
sensorsPM10CriticalValuesRDD = readingsHighValueRDD.map(lambda PM10Reading: (PM10Reading.split(',')[0], 1))

# count the number of critical values for each sensor by using the reduceByKey transformation.
sensorsCountsRDD = sensorsPM10CriticalValuesRDD.reduceByKey(lambda value1, value2: value1 + value2)

# select only the pairs with a value (number of critical PM10 values) at least equal to 2
sensorsCountsCriticalRDD = sensorsCountsRDD.filter(lambda sensorCountPair: sensorCountPair[1] >= 2)

# store the result in the output folder
sensorsCountsCriticalRDD.saveAsTextFile(outputPath)
```

Output:

```
(s1,2)
```

Exercise 41. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`), and the value of k (argument of the application). Output: an HDFS file containing the top-k critical sensors. The "criticality" of a sensor comes from the number of days with a PM10 values greater than 50. Each line contains the number of critical days and the `sensor id`.

Input:

```
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

k = 1
```

Spark program - (1):

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 41")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
outputPath = sys.argv[2]
k = argv[3]

readingsRDD = sc.textFile(inputPath)

# apply a filter transformation to select only the lines with PM10>50
readingsHighValueRDD = readingsRDD.filter(lambda PM10Reading: float(PM10Reading.split(',')[2]) > 50)

# create an RDD of key-value pairs
# each pair contains a sensor id (key) and +1 (value)
```

```
# the function of the map transformation returns a tuple
sensorsPM10CriticalValuesRDD = readingsHighValueRDD.map(lambda PM10Reading:
                                                        (PM10Reading.split(',')[0], 1))

# count the number of critical values for each sensor by using the reduceByKey transformation.
sensorsCountsRDD = sensorsPM10CriticalValuesRDD.reduceByKey(lambda value1, value2: value1 + value2)

# use top to select the top k pairs according to the number of critical dates
topKSensorsNumCriticalValues = sensorsCountsRDD.top(1, lambda pair: pair[1])

topKSensorsRDD.saveAsTextFile(outputPath)
```

Spark program - (2):

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 41")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
outputPath = sys.argv[2]
k = argv[3]

readingsRDD = sc.textFile(inputPath)

# apply a filter transformation to select only the lines with PM10>50
readingsHighValueRDD = readingsRDD.filter(lambda PM10Reading: float(PM10Reading.split(',')[2]) > 50)

# create an RDD of key-value pairs
# each pair contains a sensor id (key) and +1 (value)
# the function of the map transformation returns a tuple
sensorsPM10CriticalValuesRDD = readingsHighValueRDD.map(lambda PM10Reading:
                                                        (PM10Reading.split(',')[0], 1))

# count the number of critical values for each sensor by using the reduceByKey transformation.
sensorsCountsRDD = sensorsPM10CriticalValuesRDD.reduceByKey(lambda value1, value2: value1 + value2)

# sort pairs by number of critical values - descending order
sortedNumCriticalValuesSensorRDD = sensorsCountsRDD.sortBy(lambda pair: pair[1], False)

# select the first k elements of sortedNumCriticalValuesSensorRDD.
topKSensorsNumCriticalValues = sortedNumCriticalValuesSensorRDD.take(k)

topKSensorsRDD.saveAsTextFile(outputPath)
```

Output:

```
s1, 2
```

2.4.7 FoldByKey Transformation

The fold by key transformation provides the same features of the `reduceBykey()` transformation. However, `foldByKey()` also provides a "zero" value to enable the utilization of **associative** (but **not commutative**) functions. The process relies on the `foldByKey(zeroValue, op)` method of the RDD class. The method recursively executes the function `op` over the values of pairs compatible with a single key at a time until the reduction/combination of such values into a single value. The method then returns an RDD containing a number of key-value pairs equal to the number of distinct keys in the input key-value pair RDD. Take into account that the `foldByKey()` method executes a shuffle

operation for computing the result of the transformation.

Example. Create an RDD from a local Python list containing the pairs `("Paolo", "Message 1")`, `("Giorgio", "Message 2")`, `("Paolo", "Message 3")`, with as key the first name of a user and as value his/her message. Create a novel RDD of key-value pairs containing one pair for each name and associate each name with the concatenation of his/her messages (preserving the order of the messages in the input RDD).

```
1 .........
2 # create the local python list
3 nameMess = [("Paolo", "Message 1"), ("Giorgio", "Message 2"), ("Paolo", "Message 3")]
4 # create the RDD of pairs from the local collection
5 nameMessPairRDD = sc.parallelize(nameMess)
6 # concatenate the messages of each user
7 concatPairRDD = nameMessPairRDD.foldByKey('', lambda m1, m2: m1 + m2)
```

2.4.8 CombineByKey Transformation

The combine by key transformation enables the creation of RDDs of key-value pairs by applying a (**associative**) function `f` on each `k`-compatible value in the input RDD of key-value pairs. The output RDD of key-value pairs contains one pair for each distinct key of the input RDD of key-value pairs. The data type of the output RDD of key-value pairs may differ with respect to the data type of the "input" RDD of key-value pairs. The process relies on the `combineByKey(createCombiner, mergeValue, mergeCombiner)` method of the RDD class, with:

- the values of the input RDD of pairs of type `V`;
- the values of the output RDD of pairs of type `U`;
- the keys for both RDDs of pairs of type `K`.

The `createCombiner` function contains the code to transform a single value (type `V`) of the input RDD of key-value pairs into a value (type `U`) of the output RDD of key-value pairs, i.e. useful to transform the first value of each key in each partition to a value of type `U`. The `mergeValue` function contains the code to combine a value of type `U` with a value of type `V`, i.e useful in each partition to combine the initial values (type `V`) of each key with the intermediate values (type `U`) of each key. The `mergeCombiner` function contains the code to combine two values of type `U`, i.e. useful to combine intermediate values of each key coming from the analysis of different partitions. The `combineByKey()` appears more general than `reduceByKey()` and `foldByKey()` because the data types of the values of the input and the output RDD of pairs may differ. Take into account that the `combineByKey()` method executes a shuffle operation for computing the result of the transformation.

Example. Create an RDD from a local Python list containing the following key-value pairs `("Paolo", 40)`, `("Giorgio", 22)` and `("Paolo", 35)`. Store

the results in an output HDFS folder. The output contains one line for each name with the average age of the users with such name.

```
.........
# create the local python list
nameAge = [("Paolo", 40), ("Giorgio", 22), ("Paolo", 35)]
# create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# compute the sum of ages and the number of input pairs for each name (key)
sumNumPerNamePairRDD = nameAgePairRDD.combineByKey(
    lambda inputElem: (inputElem, 1),
    # from an input value, i.e. an age
    # -> return a tuple (age, 1)

    lambda intermediateElem, inputElem: (intermediateElem[0] + inputElem, intermediateElem[1] + 1),
    # from an input value, i.e. an age, and an intermediate value, i.e. a (sum ages, num of values) tu
    # -> update the tuple (sum ages, num of values)

    lambda intermediateElem1, intermediateElem2: (intermediateElem1[0] + intermediateElem2[0],
                                                  intermediateElem1[1] + intermediateElem2[1])
    # from two intermediate results, i.e. two (sum ages, num of values) tuples
    # -> combine the tuples into a novel single (sum ages, num of values) tuple
    )

# compute the average for each name
avgPerNamePairRDD = sumNumPerNamePairRDD.map(lambda pair: (pair[0], pair[1][0] / pair[1][1]))
# store the result in an output folder
avgPerNamePairRDD.saveAsTextFile(outputPath)
```

2.4.9 GroupByKey Transformation

The group by key transformation enables the creation of RDDs of key-value pairs by grouping the `k`-compatible values in the input RDD of key-value pairs. The output RDD of key-value pairs contains one pair for each distinct key `k` of the input RDD of key-value pairs. The method returns, for each key, the list of `k`-compatible values in the input RDD of key-value pairs. The process relies on the `groupByKey()` method of the RDD class.

Take into account that `groupByKey` appears not suitable for grouping values by key to perform an aggregation such as `sum` or `avg` on the values of each key. The `reduceByKey`, `aggregateByKey` or `combineByKey` provide better performances for **associative** and **commutative** aggregations. The `groupByKey` appears useful to apply an aggregation by means of a **not-associative** function. Take also into account that the `groupByKey()` method executes a shuffle operation for computing the result of the transformation.

Example. Create an RDD from a local Python list containing the following key-value pairs `("Paolo", 40)`, `("Giorgio", 22)` and `("Paolo", 35)`. Store the results in an output HDFS folder. The output contains one line for each name, along with the age of all users with that name.

```
.........
# create the local python list
nameAge = [("Paolo",40), ("Giorgio",22),("Paolo",35)]
# create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)
```

```

# create one group for each name with the list of ages
agesPerNamePairRDD = nameAgePairRDD.groupByKey()
# store the result in an output folder
agesPerNamePairRDD.mapValues(lambda listValues: list(listValues)).saveAsTextFile(outputPath);
```

Exercise 39.1. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: an HDFS file containing one line for each sensor, each line contains a `sensor id` and the list of dates with a PM10 values greater than 50 for that sensor. Consider only the sensors associated at least one time with a PM10 value greater than 50.

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 39.1")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
outputPath = sys.argv[2]

readingsRDD = sc.textFile(inputPath)

# apply a filter transformation to select only the lines with PM10>50
readingsHighValueRDD = readingsRDD.filter(lambda PM10Reading: float(PM10Reading.split(',')[2]) > 50)

# create an RDD of key-value pairs
# each pair contains a sensor id (key) and a date (value)
# the function of the map transformation returns a tuple
sensorsPM10CriticalValuesRDD = readingsHighValueRDD.map(lambda PM10Reading:
                                                        (PM10Reading.split(',')[0],
                                                         PM10Reading.split(',')[1]) )

# create one pair for each sensor (key) with the list of dates relating to that sensor (value)
# by using the groupByKey transformation
finalSensorCriticalDates = sensorsCriticalDatesRDD.groupByKey()

# exploit the map method to transform the content of the iterable over the values of each key
# into a list (storable in a readable format)
finalSensorCriticalDateStringFormat = finalSensorCriticalDates.mapValues(lambda dates : list(dates))

# store the result in the output folder
finalSensorCriticalDateStringFormat.saveAsTextFile(outputPath)
```

Output:

```
(s1,[2016-01-02,2016-01-03])
(s2, [2016-01-03])
```

Exercise 39.2. Input: a collection of textual `csv` files containing the daily value of PM10 for a set of sensors (each line of the files with the format `sensor id, date, PM10 value`). Output: an HDFS file containing one line for each

sensor, each line contains a `sensor id` and the list of dates with a PM10 values greater than 50 for that sensor. Consider in the result also the sensors not relating to a PM10 values greater than 50 (with an empty set).

Input:

```
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
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 39.2")
sc = SparkContext(conf = conf)

import sys
inputPath = sys.argv[1]
outputPath = sys.argv[2]

readingsRDD = sc.textFile(inputPath)

# apply a filter transformation to select only the lines with PM10>50
readingsHighValueRDD = readingsRDD.filter(lambda PM10Reading: float(PM10Reading.split(',')[2]) > 50)

# create an RDD of key-value pairs
# each pair contains a sensor id (key) and a date (value)
# the function of the map transformation returns a tuple
sensorsPM10CriticalValuesRDD = readingsHighValueRDD.map(lambda PM10Reading:
                                                        (PM10Reading.split(',')[0],
                                                         PM10Reading.split(',')[1]) )

# create one pair for each sensor (key) with the list of dates relating to that sensor (value)
# by using the groupByKey transformation
finalSensorCriticalDates = sensorsCriticalDatesRDD.groupByKey()

# exploit the map method to transform the content of the iterable over the values of each key
# into a list (storable in a readable format)
finalSensorCriticalDateStringFormat = finalSensorCriticalDates.mapValues(lambda dates : list(dates))

# all sensors ID from the complete input file
allSensorsRDD = readingsRDD.map(lambda PM10Reading: PM10Reading.split(',')[0])

# select the identifiers of the sensors not relating to a PM10 values greater than 50
sensorsNeverHighValueRDD = allSensorsRDD.subtract(finalSensorCriticalDates.keys())

# map each sensor not relating to a PM10 values greater than 50
# to a tuple/pair (sensorId, [])
sensorsNeverHighValueRDDEmptyList = sensorsNeverHighValueRDD.map(lambda sensorId: (sensorId, []))

# Compute the final result
resultRDD = finalSensorCriticalDateStringFormat.union(sensorsNeverHighValueRDDEmptyList)

# store the result in the output folder
resultRDD.saveAsTextFile(outputPath)
```

Output:

```
(s1,[2016-01-02,2016-01-03]) (s2, [2016-01-03])
(s3, [])
```

2.4.10 MapValues Transformation

The map values transformation enables the creation of RDDs of key-value pairs by applying a function `f` on the value of each pair of an input RDD of key-value pairs. The output RDD of key-value pairs contains one pair for each input pair of the input RDD of key-value pairs, with:

- the `key` of the output pair equals to the key of the input pair (with the same data type);
- the `value` of the output pair equals to the result coming by applying the function `f` on the value of the input pair (with a different or same data type).

The process relies on the `mapValues(f)` method of the RDD class. The function `f` contains the code to transform each input value into an output value (the framework stores the result into an RDD of key-value pairs). The output RDD of pairs contains a number of key-value pairs equal to the number of key-value pairs of the input RDD of pairs.

Example. Create an RDD from a local Python list containing the pairs `("Paolo", 40)`, `("Giorgio", 22)`, `("Paolo", 35)`. The first name of a user represents the key and his/her age represents the value. Increase the age of each user (`+1` year) and store the result in the HDFS file system (one output line per user).

```
1  .........
2  # create the local python list
3  nameAge = [("Paolo",40), ("Giorgio",22),("Paolo",35)]
4  # create the RDD of pairs from the local collection
5  nameAgePairRDD = sc.parallelize(nameAge)
6
7  # increment the age of each user
8  plusOnePairRDD = nameAgePairRDD.mapValues(lambda age: age + 1)
9  # save the result on disk
10 plusOnePairRDD.saveAsTextFile(outputPath)
```

2.4.11 FlatMapValues Transformation

The flat map values transformation enables the creation of RDDs of key-value pairs by applying a function `f` on the value of each pair of an input RDD of key-value pairs, with `f` returning a list of values for each input value. The output RDD of key-value pairs contains one pair for each input pair of the input RDD of key-value pairs, with:

- the `key` of the output pair equals to the key of the input pair (with the same data type);
- the `[list of values]` of the output pair equals to the result coming by applying the function `f` on the value of the input pair (with a different or same data type).

The process relies on the `flatMapValues(f)` method of the RDD class. The function `f` contains the code to transform each input value into a set of output values (the framework stores the result in the RDD of key-value pairs). The output RDD of pairs contains a number of key-value pairs equal to the number of key-value pairs of the input RDD of pairs.

Example. Create an RDD from a local Python list containing the pairs `('Sentence1', 'Sentence test')`, `('Sentence2', 'Sentence test number 2')`, `('Sentence3', 'Sentence test number 3')`. Select the words of each sentence and store in the HDFS file system one pair `(sentece id, [list of words])` per line.

```
.........
# create the local python list
sentences = [("Sentence1", "Sentence test"),
             ("Sentence2", "Sentence test number 2"),
             ("Sentence3", "Sentence test number 3") ]
# create the RDD of pairs from the local collection
sentPairRDD = sc.parallelize(sentences)

# 'extract' words from each sentence
sentIdWord = sentPairRDD.flatMapValues(lambda s: s.split(' '))
# save the result on disk
sentIdWord.saveAsTextFile(outputPath)
```

2.4.12 Keys Transformation

The keys transformation enables the creation of a (standard) RDD (of "single" elements) containing the list of keys (with **duplicates**) of an input RDD of key-value pairs. The process relies on the `keys()` method of the RDD class.

Example. Create an RDD from a local Python list containing the pairs `("Paolo", 40)`, `("Giorgio", 22)`, `("Paolo", 35)`. The first name of a user represents the key and his/her age represents the value. Store the names of the input users in an output HDFS folder. The output contains one name per line (without duplicates).

```
.........
# create the local python list
nameAge = [("Paolo",40), ("Giorgio",22),("Paolo",35)]
# create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# select the key part of the input RDD of key-value
pairs namesRDD = nameAgePairRDD.keys().distinct()
# store the result in an output folder
namesRDD.saveAsTextFile(outputPath);
```

2.4.13 Values Transformation

The values transformation enables the creation of a (standard) RDD (of "single" elements) containing the list of values (with **duplicates**) of an input RDD of key-value pairs. The process relies on the `values()` method of the RDD class.

Example. Create an RDD from a local Python list containing the pairs `("Paolo", 40)`, `("Giorgio", 22)`, `("Paolo", 35)`. The first name of a user represents the key and his/her age represents the value. Store the ages of the input users in an output HDFS folder. The output contains one age per line (with duplicates).

```
.........
# create the local python list
nameAge = [("Paolo",40), ("Giorgio",22),("Paolo",35)]
# create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# select the value part of the input RDD of key-value pairs
agesRDD = nameAgePairRDD.values()
# store the result in an output folder
agesRDD.saveAsTextFile(outputPath);
```

2.4.14 SortByKey Transformation

The sort by key transformation enables the creation of RDDs of key-value pairs by sorting, in ascending order, the pairs of the input RDD by key. The method relates the final order according to the default sorting function of the input keys data type. The output RDD of key-value pairs contains one pair for each input pair of the input RDD of key-value pairs, in ascending order. The process relies on the `sordByKey(ascending)` method of the RDD class. The method enables to set the sorting criteria via the `ascending` parameter:

- `ascending = True`: ascending order;
- `ascending = False`: descending order.

Take into account that the `sortByKey()` method executes a shuffle operation for computing the result of the transformation.

Example. Create an RDD from a local Python list containing the pairs `("Paolo", 40)`, `("Giorgio", 22)`, `("Paolo", 35)`. The first name of a user represents the key and his/her age represents the value. Sort the users by name and store the result in the HDFS file system.

```
.........
# create the local python list
nameAge = [("Paolo",40), ("Giorgio",22),("Paolo",35)]
# create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# sort by name the content of the input RDD of key-value pairs
sortedNameAgePairRDD = nameAgePairRDD.sortByKey()
# store the result in an output folder
sortedNameAgePairRDD.saveAsTextFile(outputPath);
```

2.4.15 Transformation on two RDDs of key-value pairs

The Spark framework enables also to apply some transformations on two RDDs of key-value pairs at the same time, e.g., `subtractByKey`, `join`, `coGroup`, etc.

2.4.16 SubtractByKey Transformation

The subtract by key transformation enables the creation of an output RDD of key-value pairs containing the k-compatible pairs appearing in the "first input" RDD but not in the "second input" RDD of pairs. The method maintains the data type of the "first input" RDD of pairs. Take into account that the method requires the two input RDDs of pairs to present the same type of keys (the data type of the values may differ). The process relies on the `subtractByKey(other)` method of the RDD class, where `other` refers to the second input RDD to analyze. Take also into account that the `subtractByKey()` method executes a shuffle operation for computing the result of the transformation.

Example. Create two RDDs of key-value pairs from two local Python lists: first list – profiles of the users of a blog (username, age), i.e. `[("Paolo", 40), ("Giorgio", 22), ("Paolo", 35)]`; second list – banned users (username, motivation), i.e. `[("PaoloB", "spam"), ("Giorgio", "Vandalism")]`. Create a novel RDD of pairs containing only the profiles of the non-banned users.

```
.........
# create the first local python list
profiles = [("PaoloG", 40), ("Giorgio", 22), ("PaoloB", 35)]
# create the RDD of pairs from the profiles local list
profilesPairRDD = sc.parallelize(profiles)
# create the second local python list
banned = [("PaoloB", "spam"), ("Giorgio", "Vandalism")]
# create the RDD of pairs from the banned local list
bannedPairRDD = sc.parallelize(banned)

# select the profiles of the 'good' users
selectedProfiles = profilesPairRDD.subtractByKey(bannedPairRDD)
```

2.4.17 Join Transformation

The join transformation enables to join the key-value pairs of two input RDDs of key-value pairs by key. The process returns each possible combination by key between the input RDDs, i.e. combines each pair of the input RDD of pairs with each pair of the other RDD of pairs with the same key. The output RDD of key-value pairs appears with (1.) the same key data type of the "input" RDDs of pairs and (2.) a tuple as value (the join between the pair of values of the two input pairs). Take into account that the method requires the two input RDDs of pairs to present the same type of keys (the data type of the values may differ). The process relies on the `join(other)` method of the RDD class, where `other` refers to the second input RDD to analyze. Take also into account that the `join()` method executes a shuffle operation for computing the result of the transformation.

Example. Create two RDDs of key-value pairs from two local Python lists: first list – list of questions (question id, text of the question), i.e. `[(1, "What is ..  ?"), (2, "Who is ..?")]`; second list – list of answers (question id, text of the answer), i.e. `[(1, "It is a car"), (1, "It is a bike"), (2,`

`"She is Jenny")]`. Create a novel RDD of pairs to associate each question with its answers (one pair for each possible pair question - answer).

```
.........
# create the first local Python list
questions = [(1, "What is .. ?"), (2, "Who is ..?")]
# create the RDD of pairs from the local list
questionsPairRDD = sc.parallelize(questions)
# create the second local python list
answers = [(1, "It is a car"), (1, "It is a bike"), (2, "She is Jenny")]
# create the RDD of pairs from the local list
answersPairRDD = sc.parallelize(answers)

# join questions with answers
joinPairRDD = questionsPairRDD.join(answersPairRDD)
```

Input:

```
questions = [
                (1, "What is .. ?"),
                (2, "Who is ..?")
            ]
answers = [
              (1, "It is a car"),
              (1, "It is a bike"),
              (2, "She is Jenny")
          ]
```

Output:

```
joinPairRDD = [
                  (1, ("What is .. ?", "It is a car")),
                  (1, ("What is .. ?", "It is a bike")),
                  (2, ("Who is ..?", "She is Jenny"))
              ]
```

2.4.18 Cogroup Transformation

The co group transformation enables to associate each key `k` of the two input RDDs of key-value pairs with:

- the list of `k`-compatible values in the first input RDD of pairs;
- the list of `k`-compatible values in the second input RDD of pairs.

The transformation enables the creation of an output RDD of key-value pairs with (1.) the same key data type of the "input" RDDs of pairs and (2.) a tuple as value (the join between the pair of values of the two input pairs). Take into account that the method requires the two input RDDs of pairs to present the same type of keys (the data type of the values may differ). The process relies on the `cogroup(other)` method of the RDD class, where `other` refers to the second input RDD to analyze. Take also into account that the `cogroup(other)` method executes a shuffle operation for computing the result of the transformation.

Example. Create two RDDs of key-value pairs from two local Python lists: first list – list of liked movies (user id, liked movies), i.e. `[(1, "Star Trek"),`

`(1, "Forrest Gump") , (2, "Forrest Gump")]`; second list – list of liked directors (user id, liked director), i.e. `[(1, "Woody Allen"), (2, "Quentin Tarantino") , (2, "Alfred Hitchcock")]`. Create a novel RDD of pairs containing one pair for each user id (key) compatible with (1.) the list of liked movies and (2.) the list of liked directors.

Inputs:

```
[(1, "Star Trek"), (1, "Forrest Gump"), (2, "Forrest Gump")]
[(1, "Woody Allen"), (2, "Quentin Tarantino") , (2, "Alfred Hitchcock")]
```

Spark program:

```
.........
# create the first local python list
movies = [(1, "Star Trek"), (1, "Forrest Gump"), (2, "Forrest Gump")]
# create the RDD of pairs from the first local list
moviesPairRDD = sc.parallelize(movies)
# create the second local python list
directors = [ (1, "Woody Allen"), (2, "Quentin Tarantino"), (2, "Alfred Hitchcock")]
# create the RDD of pairs from the second local list
directorsPairRDD = sc.parallelize(directors)
# cogroup movies and directors per user
cogroupPairRDD = moviesPairRDD.cogroup(directorsPairRDD)
```

Output:

```
(1, (["Star Trek", "Forrest Gump"], ["Woody Allen"]) )
(2, (["Forrest Gump"], ["Quentin Tarantino", "Alfred Hitchcock"]))
```

Exercise 42. Input: (1) a large textual file containing a set of questions (each line contains one question in the format `question id, timestamp, text of the question`), (2) a large textual file containing a set of answers (each line contains one answer in the format `answer id, question id, timestamp, text of the answer`). Output: a file containing one line for each question (each line contains a question and the list of answers to that question, i.e. `question id, text of the question, list of answers`).

Input (1) - questions:

```
Q1,2015-01-01,What is ..?
Q2,2015-01-03,Who invented ..
```

Input (2) - answers:

```
A1,Q1,2015-01-02,It is ..
A2,Q2,2015-01-03,John Smith
A3,Q1,2015-01-05,I think it is ..
```

Spark program:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("Exercise 42")
sc = SparkContext(conf = conf)

import sys
inputPathQuestions  = argv[1]
inputPathAnswers = argv[2]
outputPath = argv[3]

questionsRDD = sc.textFile(inputPathQuestions)
```

```
11
12 # create an RDD of pairs with the question id as key and the question text as value
13 questionsPairRDD = questionsRDD.map(lambda question: (question.split(",")[0],
14                                                      question.split(",")[2]))
15
16 answersRDD = sc.textFile(inputPathAnswers)
17
18 # create an RDD of pairs with the question id as key and the answer text as value
19 answersPairRDD = answersRDD.map(lambda answer: (answer.split(",")[1],
20                                                 answer.split(",")[3]))
21
22 # "cogroup" the two RDDs of pairs
23 questionsAnswersPairRDD = questionsPairRDD.cogroup(answersPairRDD)
24
25 # use map to transform the two iterables of each pair into a list (reformat them)
26 questionsAnswersReformatted = questionsAnswersPairRDD.mapValues(lambda value:
27                                                                 (list(value[0]),
28                                                                  list(value[1])))
29
30 questionsAnswersReformatted.saveAsTextFile(outputPath)
```

Output:

```
1 (Q1,([What is ..?],[It is .., I think it is ..]))
2 (Q2,([Who invented ..],[John Smith]))
```

2.4.19 Actions on RDDs of key-value pairs

The Spark framework supports also some specific actions on RDDs of key-value pairs, e.g., `countByKey`, `collectAsMap`, `lookup`.

2.4.20 CountByKey Action

The count by key action returns a local Python dictionary containing the information about the number of `k`-compatible elements in the input RDD of key-value pairs, i.e., the number of times each key `k` occurs in the input RDD (**pay attention to the number of distinct keys of the input RDD of pairs to not overcome the storage availability of the driver program**). The process relies on the `countByKey()` method of the RDD class. The framework sends the data over the network to compute the final result.

Example. Create an RDD of pairs from the following Python list `[("Forrest Gump", 4), ("Star Trek", 5) , ("Forrest Gump", 3)]` (each pair contains a movie and a rating). Compute the number of ratings for each movie.

```
1 .........
2 # create the local python list
3 movieRating = [("Forrest Gump", 4), ("Star Trek", 5), ("Forrest Gump", 3)]
4 # create the RDD of pairs from the local collection
5 movieRatingRDD = sc.parallelize(movieRating)
6
7 # compute the number of rating for each movie
8 movieNumRatings = movieRatingRDD.countByKey()
9 # print the result on the standard output
10 print(movieNumRatings)
```

Output:

```
1 {'Forrest Gump': 2, 'Star Trek': 1}
```

2.4.21 CollectAsMap Action

The collect as map action returns a local Python dictionary containing the same pairs of the input RDD of pairs (**pay attention to the number of distinct keys of the input RDD of pairs to not overcome the storage availability of the driver program**). The process relies on the `collectAsMap()` method of the RDD class. The framework sends the data over the network to compute the final result. Pay attention that the `collectAsMap()` action returns a dictionary object (i.e. with no duplicates). The method associates each key with at most one value, therefore, for multiple tuples with the same key in the input RDD of pairs, the method returns the last occurrence. Pay attention to leverage `collectAsMap()` only in situation with an input RDD of key-value pairs without duplicate keys. The `collectAsMap()` action returns a local dictionary while `collect()` return a list of key-value pairs (i.e., a list of tuples). The list of pairs from `collect()` may contain more than one pair compatible with the same key.

Example. Create an RDD of pairs from the following Python list `[("User1", "Paolo"), ("User2", "Luca"), ("User3", "Daniele")]` (each pair contains a user id and the name of the user). Retrieve and print the pairs of the input RDD of pairs as a local Python dictionary.

```
.........
# create the local python list
users = [("User1","Paolo"), ("User2","Luca"), ("User3","Daniele")]
# create the RDD of pairs from the local list
usersRDD = sc.parallelize(users)

# retrieve the content of usersRDD and store it in a local python dictionary
retrievedPairs = usersRDD.collectAsMap()
# print the result on the standard output
print(retrievedPairs)
```

Output:

```
{'User1': 'Paolo', 'User2': 'Luca', 'User3': 'Daniele'}
```

2.4.22 Lookup Action

The look up action returns a local Python list containing the values of the `k`-compatible pairs pairs of the input RDD. The process relies on the `lookup(k)` method of the RDD class.

Example. Create an RDD of pairs from the following Python list `[("Forrest Gump", 4), ("Star Trek", 5) , ("Forrest Gump", 3)]` (each pair contains a movie and a rating). Retrieve and print the ratings compatible with the movie "Forrest Gump" as a local Python list.

```
.........
# create the local python list
movieRating = [("Forrest Gump", 4), ("Star Trek", 5), ("Forrest Gump", 3)]
# create the RDD of pairs from the local collection
movieRatingRDD = sc.parallelize(movieRating)
```

```
6
7  # select the ratings compatible with 'Forrest Gump'
8  movieRatings = movieRatingRDD.lookup("Forrest Gump")
9  # print the result on the standard output
10 print(movieRatings)
```

Output:

```
1  [4, 3]
```

2.5 RDDs of Numbers

The framework provides specific actions for RDD containing numerical values (integers or floats). Spark provides the creation of RDDs of numbers by exploiting the standard methods:

- parallelize;
- transformations returning RDD of numbers.

Moreover, the framework provides specific actions for such type of RDDs, e.g., `sum()`, `mean()`, `stdev()`, `variance()`, `max()`, `min()`.

Example. Create an RDD containing the following float values `[1.5, 3.5, 2.0]` and print on the standard output the following statistics: sum, mean, standard deviation, variance, maximum value, and minimum value.

```
1  .........
2  # create an RDD containing a list of float values
3  inputRDD = sc.parallelize([1.5, 3.5, 2.0])
4  # compute the statistics of interest and print them on the standard output
5  print("sum: ", inputRDD.sum())
6  print("mean: ", inputRDD.mean())
7  print("stdev: ", inputRDD.stdev())
8  print("variance: ", inputRDD.variance())
9  print("max: ", inputRDD.max())
10 print("min: ", inputRDD.min())
```

Output:

```
1  sum: 7.0
2  mean: 2.3333333333333335
3  stdev: 0.849836585598797
4  variance: 0.7222222222222222
5  max: 3.5
6  min: 1.5
```

2.6 Cache, Accumulators, Broadcast variables, Custom Partitioners, Broadcast Join

2.6.1 Persistence and Cache

The Spark framework computes the content of an RDD each time the application executes an action on it. For iterative applications, Spark recomputes the RDD

content every time the application executes another action on it, or on any "descendents". Such operation results quite expensive. The framework enables therefore to persist/cache RDDs. Through the persist/cache RDDs operation, the framework enables each node to store the content of RDD partitions in memory and to reuse such partitions in other actions. After the persist/cache operation on an RDD, the first time the framework executes and action on it, i.e. computes the respective content, the framework maintains the content in the main memory of the nodes. For the subsequent actions on the same RDD, the framework reads the respective content from memory, i.e., Spark persists/caches the content of the RDD across operations, enabling future actions to appear much faster (often by more than 10x). Spark supports several storage levels, useful to specify the storage location of the RDD content:

- in the main memory of the nodes;
- on the local disks of the nodes;
- partially in the main memory and partially on the local disk.

The persist operation relies on the `persist(storageLevel)` method of the RDD class, with as parameter a value of the type:

- `pyspark.StorageLevel.MEMORY_ONLY`
- `pyspark.StorageLevel.MEMORY_AND_DISK`
- `pyspark.StorageLevel.DISK_ONLY`
- `pyspark.StorageLevel.NONE`
- `pyspark.StorageLevel.OFF_HEAP`
- `pyspark.StorageLevel.MEMORY_ONLY_2`
- `pyspark.StorageLevel.MEMORY_AND_DISK_2`

The storage level `2` replicate each partition on two cluster nodes to perform the actions without recomputing the content of the RDD in case of failing node. The cache operation relies on the `cache()` method of the RDD class, which corresponds to persist the RDD with the storage level `MEMORY_ONLY`, i.e., equivalent to `inRDD.persist(pyspark.StorageLevel.MEMORY_ONLY)`. Take into account that both persist and cache return an output RDD, because RDDs appear immutable. The utilization of the persist/cache mechanism on an RDD provides an advantage for iterative applications, i.e., for applications applying multiples actions on RDDs (or any RDDs "descendents"). The storage levels that store RDDs on the local disk appear useful if and only if (1.) the "size" of the RDD results significantly smaller than the size of the input dataset, or (2.) the functions computing the content of the RDD appears expensive. Otherwise, recomputing a partition may result as fast as reading such partition from the

disk. Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion. The framework enables to manually remove an RDD from the cache by means of the `unpersist()` method of the RDD class.

Example. Create an RDD from a textual file containing a list of words (one word for each line) and print on the standard output (1.) the number of lines of the input file and (2.) the number of distinct words.

```
.........
# read the content of a textual file and cache the respective RDD
inputRDD = sc.textFile("words.txt").cache()
print("Number of words: ", inputRDD.count())
print("Number of distinct words: ", inputRDD.distinct().count())
```

At the execution of the `count()` action, i.e. the first action, the framework computes the `inputRDD` RDD content by reading the lines of the `words.txt` file and returns the result of the `count()` action. Via the `cache()` method, the framework also stores the content of the `inputRDD` RDD in the main memory of the nodes of the cluster. Hence, the framework provides the computation of `distinct() + count()` by reading the data from the main memory and not from the input (HDFS) file `words.txt`.

2.6.2 Accumulators

In the case where an operation executes a function on a remote cluster node, the framework processes the function on separate copies of the function variables, i.e. Spark copies the function variables into each node of the cluster without propagating the updates of such variables back to the driver program. Spark provides therefore a type of shareable variables, i.e. the **accumulators**, addable through an associative operation. Accumulators appear suitable to compute simple statistics while performing some other actions on the input RDD, i.e. to avoid exploiting actions such as `reduce()` to compute simple statistics (e.g., count the number of lines with some characteristics). The driver defines and initializes the accumulator. The code executable in the processing nodes increases the value of the accumulator, i.e., the code in transformation "functions". The framework returns the final value of the accumulator to the driver node since only the driver accesses the final value of the accumulator (the processing nodes increases the accumulator value without accessing to the accumulator). Spark natively supports numerical accumulators (integers and floats), but enables developers to add different data types. Accumulators represents `pyspark.accumulators.Accumulator` objects, definable and initializable via the `accumulator(value)` method of the `SparkContext` class. Afterwords, the `add(value)` method of the `Accumulator` class enables to increase the value of an accumulator (add "value" to the current value of the accumulator). The `value` of the `Accumulator` class enables instead to retrieve the final value of an accumulator in the driver program.

Example. Create an RDD from a textual file containing a list of email addresses (one email for each line). Select the lines containing a valid email and store them in an HDFS file. In this example, an email containing the `@` symbol results a valid email. Print also, on the standard output, the number of invalid emails.

```
.........
# define an accumulator and initialize to 0
invalidEmails = sc.accumulator(0)

# read the content of the input textual file
emailsRDD = sc.textFile("emails.txt")

# define the filtering function
def validEmailFunc(line):
    if (line.find('@') < 0):
        invalidEmails.add(1) # increase the accumulator value
        return False
    else:
        return True

# select only valid emails and count also the number of invalid emails
validEmailsRDD = emailsRDD.filter(validEmailFunc)
# store valid emails in the output file
validEmailsRDD.saveAsTextFile(outputPath)

# print the number of invalid emails
print("Invalid email addresses: ", invalidEmails.value)
```

Pay attention that without executing the action (`saveAsTextFile`) the value of the accumulator results zero, because the action triggers the execution of the transformation operation that increments the accumulator. The framework enables developers to define accumulators according to different data types (different from integers and floats). To define a different accumulator data type of type `T`, the developer define a class subclassing the `AccumulatorParam` interface, with the methods: `zero` for providing a "zero value" for the data type, and `addInPlace` for adding two values together.

2.6.3 Broadcast variables

The Spark framework supports broadcast variables, i.e. read-only (small/medium) shareable variables (instantiable in the driver). The framework stores the broadcast variable in a local variable of the main memory of the driver. Afterwards, the framework sends the in-memory broadcast variable to each processing node that exploits such variable in one or more Spark operations. The framework also stores the broadcast variable in the main memory of the executors, i.e., sends a copy of each broadcast variable to each executor executing a Spark operation according to such variable. More in detail, the framework sends a broadcast variable a single time to each executor executing at least one Spark operation according to such variable. Hence, the broadcast variable enables to limit the amount of data sent over the network with respect to the "standard" variables. Broadcast variables appear suitable to share (small/medium) lookup-tables (small because the application stores the broadcast variables in the main memory of the driver and also in the main memory of the executors). Broadcast

variables represents objects of type `Broadcast`. A broadcast variable (of type `T`) results definible in the driver by means of the `broadcast(value)` method of the `SparkContext` class. The application retrieves the value of a broadcast variable (of type `T`) by means of the `value` of the Broadcast class (usually in transformations).

Example. Create an RDD from a textual file containing a dictionary of pairs `(word, integer value)`, with one pair for each line. Suppose that the content of this first file results large but archiveable in the main-memory. Create an RDD from a textual file containing a set of words, with a sentence (set of words) for each line. "Transform" the content of the second file mapping each word to an integer according to the dictionary within the first file. Store the result in an HDFS file.

Input file (1) - `dictionary.txt`:

```
java 1
spark 2
test 3
```

Input file (2) - `document.txt`:

```
java spark
spark test java
```

Spark program:

```
# read the content of the dictionary from the first file
# and map each line to a pair (word, integer value)
dictionaryRDD = sc.textFile("dictionary.txt")
                  .map(lambda line: (line.split(" ")[0], line.split(" ")[1]))

# create a broadcast variable according to the content of dictionaryRDD
# pay attention that a broadcast variable results instantiable only
# by passing as parameter a local variable and not an RDD.
# hence, the collectAsMap method attempts to retrieve the content of the
# RDD and store it in the dictionary variable
dictionary = dictionaryRDD.collectAsMap()
# broadcast dictionary
dictionaryBroadcast = sc.broadcast(dictionary)

# read the content of the second file
textRDD = sc.textFile("document.txt")
# define the function to map strings to integers
def myMapFunc(line):
    transformedLine = ""
    for word in line.split(" "):
        intValue = dictionaryBroadcast.value[word]
        transformedLine = transformedLine + intValue + " "
    return transformedLine.strip()
# map words in textRDD to the corresponding integers and concatenate them
mappedTextRDD = textRDD.map(myMapFunc)
# store the result in an HDFS file
mappedTextRDD.saveAsTextFile(outputPath)
```

Output:

```
1 2
2 3 1
```

2.6.4 RDDs and Partitions

In the Spark paradigm each RDD splits the content into partitions. The number of partitions and the content of each partition rely on the definition/creation method of the RDD. The number of partitions impacts on the maximum parallelization degree of the Spark application. Pay attention to the limitation due to the amount of resources (there appears a maximum number of executors and parallel tasks). From too few partitions results (1.) less concurrency/parallelism (i.e. with processing nodes in idle state), (2.) data skewing and improper resource utilization (i.e. with a single partition with many data and many partitions with few data). The framework provides some specific transformations to set the number of partitions of the resulting RDD, e.g. `parallelize()`, `textFile()`, `epartition()`, `coalesce()`. The majority of the Spark transformations maintains unchangeable the number of partitions, i.e. preserve the number of partitions of the input RDD (the resulting RDD appears with the same number of partitions of the input RDD).

- `parallelize(collection)`: the number of partitions of the resulting RDD appears equal to `sc.defaultParallelism`. Sparks tries to balance the number of elements per partition in the resulting RDD.

- `parallelize(collection, numSlices)`: the number of partitions of the resulting RDD appears equal to `numSlices`. Sparks tries to balance the number of elements per partition in the resulting RDD.

- `textFile(pathInputData)`: the number of partitions of the resulting RDD appears equal to the number of input chunks/blocks of the input HDFS data. Each partition contains the content of one of the input blocks.

- `textFile(pathInputData, minPartitions)`: the number of partitions of the resulting RDD appears greater than or equal to the specifiable value `minPartitions`. Each partition contains a part of one input blocks.

- `repartition(numPartitions)`: with `numPartitions` greater or smaller than the number of partitions of the input RDD. The number of partitions of the resulting RDD appears equal to `numPartitions`. Sparks tries to balance the number of elements per partition in the resulting RDD. The process executes a shuffle operation to assign input elements to the partitions of the resulting RDD.

- `coalesce(numPartitions)`: with `numPartitions` < number of partitions of the input RDD. The number of partitions of the resulting RDD appears equal to `numPartitions`. Sparks tries to balance the number of elements per partition in the resulting RDD, usually without the shuffle operation. The `coalesce()` method results more efficient than `repartition()` to reduce the number of partitions.

2.6.5 Partitioning of Pair RDDs

The Spark framework enables specifying how to partition the content of RDDs of key-value pairs, i.e. by grouping the input pair in partitions according to the integer value coming from a function applicable on the key of each input pair, useful to improve the efficiency of the next transformations by reducing the amount of shuffle operations and the amount of data sent over the network in the next steps of the application. The partitioning process relies on the `PartitionBy(numPartitions)` transformation, which groups the input pair in partitions according to the integer value coming from a default hash function applicable on the key of each input pair. The process executes a shuffle operation to assign input elements to the partitions of the resulting RDD. Suppose that:

- the number of partition of the resulting Pair RDD comes out as `numPart`;
- the default partition function result `portable hash`, which returns an integer;
- from an input pair `(key, value)`, the process stores a copy of such pair in the partition number `n` of the resulting RDD, where `n = portable hash(key) % numPart`.

The framework also provides the `partitionBy(numPartitions, partitionFunc)` transformation, which groups the input pair in partitions according to the integer value coming from the user-specifiable `partitionFunc` function. The process executes a shuffle operation to assign input elements to the partitions of the resulting RDD. Suppose that:

- the number of partition of the resulting Pair RDD comes out as `numPart`;
- the default partition function result `partitionFunc`, which returns an integer;
- from an input pair `(key, value)`, the process stores a copy of such pair in the partition number `n` of the resulting RDD, where `n = partitionFunc(key) % numPart`.

Partitioning Pair RDDs via `partitionBy()` results useful only when the application caches and reuses the same partitionable RDD multiple times in the task in time- and network-consuming key-orientable transformations, e.g., for applications processing the same partitionable RDD in many `join()`, `cogroup`, `groupyByKey()` transformations in different paths/branches of the task (different paths/branches of the DAG). Pay attention to the amount of data the application actually sends over the network, since the `partitionBy()` slow down the application instead of speeding it up.

Example. Create an RDD from a textual file containing a list of pairs `(page id, list of linked pages)` and implement the (simplifiable) `PageRank` algorithm to compute the page rank of each input page. Print the result on the standard output.

```
.........
# read the input file with the structure of the web graph
inputData = sc.textFile("links.txt")
# format of each input line
# PageId, LinksToOtherPages - e.g., P3 [P1,P2,P4,P5]
def mapToPairPageIDLinks(line):
    fields = line.split(" ")
    pageID = fields[0]
    links = fields[1].split(",")
    return (pageID, links)
links = inputData.map(mapToPairPageIDLinks)
                 .partitionBy(inputData.getNumPartitions()).cache()
# initialize each page rank to 1.0; since exploiting mapValues,
# the resulting RDD comes with the same partitioner as links
ranks = links.mapValues(lambda v: 1.0)
# function that returns a set of pairs from each input pair
# input pair: (page id, (linked pages, current page rank of page id))
# one output pair for each linked page. Output pairs:
# (page id linked page,
# current page rank of the linking page page id / number of linked pages)
def computeContributions(pageIDLinksPageRank):
    pagesContributions = []
    currentPageRank = pageIDLinksPageRank[1][1]
    linkedPages = pageIDLinksPageRank[1][0]
    numLinkedPages = len(linkedPages)
    contribution = currentPageRank / numLinkedPages
    for pageidLinkedPage in linkedPages:
        pagesContributions.append((pageidLinkedPage, contribution))
    return pagesContributions
# run 30 iterations of PageRank
for x in range(30):
    # retrieve for each page its current pagerank and
    # the list of linked pages by using the join transformation
    pageRankLinks = links.join(ranks)
    # compute contributions from linking pages to linked pages for this iteration
    contributions = pageRankLinks.flatMap(computeContributions)
    # update current pagerank of all pages for this iteration
    ranks = contributions.reduceByKey(lambda contrib1, contrib2: contrib1 + contrib2)
# print the result
ranks.collect()
```

The application executes the `join()` transformation many times on the `links` Pair RDD. The content of `links` remain immutable during the loop iterations. Hence, caching `links` and partitioning the respective content by key appears useful to:

- compute and cache (in the main memory of the executors) the content once;
- shuffle and sent over the network once due to `partitionBy()`.

Chapter 3

Apache Spark SQL and DataFrames

The Spark SQL framework refers to the Spark component for processing tabular data structures, providing a programming abstraction (DataFrame) and acting as a distributable SQL query engine. The Spark SQL framework provides an interface with information about the structure of the data and the computation to perform, enabling the system to query the input data via (1.) ad hoc methods or (2.) a SQL-like language. The framework leverages such additional information to perform additional optimizations according to a "SQL-like" optimizer (Catalyst). DataFrame-based programs result usually faster than standard RDD-based programs.

The DataFrame represents a distributable collection of structurable data, conceptually equivalent to a table in a relational database, definable by reading data from different types of external sources (CSV files, JSON files, RDBM, ...). The Spark SQL's optimizable execution engine benefits from the DataFrame abstraction to leverage the information about the data structure. Each Spark SQL feature relies on an instance of the `pyspark.sql.SparkSession` class.

```
1 from pyspark.sql import SparkSession
2 spark = SparkSession.builder.getOrCreate()
3 .........
4 spark.stop()
```

3.1 DataFrames

DataFrame represents a distributable collection of structurable data, conceptually equivalent to a table in a relational database, i.e., lists of `Row` objects. The framework provides some classes to define/create DataFrames, such as `pyspark.sql.DataFrame` and `pyspark.sql.Row`. The Spark SQL framework enables developers to create DataFrames from different sources, such as from

structurable (textual) data files (e.g. CSV files, JSON files), existing RDDs, hive tables, and external relational databases.

3.1.1 Creating a DataFrame from CSV files

Spark SQL provides an API that enables the creation of DataFrames directly from CSV files.

Example of CSV file:

```
name, age
Andy, 30
Michael,
Justin, 19
```

The example file contains the names and ages of three people, with an unknown age for the second person. The creation of a DataFrame from a CSV file relies on the `load(path)` method of the `pyspark.sql.DataFrameReader` class, with the `path` parameter referencing the path to the input file. The framework enables to get a `DataFrameReader` through the `read()` method of the `SparkSesssion` class, i.e.

- `df = spark.read.load(path, options ...)`

Example. Create a DataFrame from a CSV file (`persons.csv`) containing the profiles of a set of persons (each line of the file contains name and age of a person). The age may assume the `null` value (i.e., missing). The first line contains the header (i.e., the names of the attributes/columns).

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
```

The `header = True` option enables to specify that the first line of the file contains the name of the attributes/columns. The `inferSchema = True` option enables the system to infer the data types of each column. Take into account that by setting `inferSchema = False` the system considers each column of the input CSV file as a string.

3.1.2 Creating a DataFrame from JSON files

Spark SQL provides an API that enables the creation of DataFrames directly from a textual file containing a JSON object for each line. Hence, the input file results not a "standard" JSON file. In order to retrieve a single JSON object (tuple) for each line, the framework formats the input file in compliance with the "JSON Lines text format", also namely `newline-delimited JSON`.

Example of JSON Lines text format file compatible with the Spark format:

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

The example file contains the names and ages of three people, with an unknown age for the second person. The creation of a DataFrame from JSON files relies on the `load(path)` method of the `pyspark.sql.DataFrameReader` class, with the `path` parameter referencing the path to the input file. The framework enables to get a `DataFrameReader` through the `read()` method of the `SparkSesssion` class, i.e.

- `df = spark.read.load(path, format = "json", ...)`
- `df = spark.read.json(path, ...)`

The above API also enables to read "standard" multi-line JSON files by setting the argument `multiLine = True` on the `DataFrameReader`. Pay attention that reading a set of small JSON files from HDFS results very slow.

Example. Create a DataFrame from a JSON text format file (`person.json`) containing the profiles of a set of persons (each line of the file contains a JSON object containing name and age of a person). The age may assume the `null` value (i.e., missing).

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# create a DataFrame from persons.json
df = spark.read.load( "persons.json",
                      format = "json")
```

Example. Create a DataFrame from a folder (`folderJSONFiles/`) containing a set of "standard" multi-line JSON files (each input JSON file contains the profile of one person, with name and age). The age may assume the `null` value (i.e., missing).

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()

# create a DataFrame
df = spark.read.load( "folderJSONFiles/",
                      format = "json",
                      multiLine = True)
```

3.1.3 Creating a DataFrame from other sources

The `DataFrameReader` class also provides other methods for reading standard format (textual) data from external databases, such as Apache parquet files, external relational databases, across a JDBC connection, hive tables, etc.

3.1.4 Creating a DataFrame from RDDs or Python Lists

The Spark SQL framework enables to store the content of an RDD of tuples or the content of a Python list of tuples in a DataFrame by means of the `spark.createDataFrame(data, schema)` method, with:

- the `data` parameter referencing an RDD of tuples or `Row`, a Python list of tuples or `Row`, or a pandas DataFrame;
- the `schema` parameter (optional) referencing a list of string with the names of the columns/attributes.

Take into account that without specifying the `schema` parameter the method sets the column names to `1, 2, ..., n` for input RDDs/lists of tuples.

Example. Create a DataFrame from a Python list containing the following data: `[(19, "Justin"), (30, "Andy"), (None, "Michael")]`. Set the column names to `age` and `name`.

```
1 .........
2 # create a Spark Session object
3 spark = SparkSession.builder.getOrCreate()
4 # create a Python list of tuples
5 profilesList = [(19, "Justin"), (30, "Andy"), (None, "Michael")]
6 # create a DataFrame from the profilesList
7 df = spark.createDataFrame(profilesList, ["age", "name"])
```

3.1.5 From DataFrame to RDD

The `rdd` member of the DataFrame class returns an RDD of `Row` objects containing the content of the DataFrame. Each `Row` object appears as a dictionary containing the values of a record, i.e. with the column names as keys and the column values as values. The `Row` class provides some techniques to access the fields:

- as attributes (`row.key`), where `key` refers to a column name;
- as dictionary values (`row["key"]`), where `key` refers to a column name;
- as Python dictionary (`asDict()`).

Example. Create a DataFrame from a CSV file containing the profiles of a set of persons (each line of the file contains name and age of a person). The first line contains the header, i.e., the name of the attributes/columns. Transform the input DataFrame into an RDD, select only the `name` field/column and store the result in the output folder.

```
1 .........
2 # create a Spark Session object
3 spark = SparkSession.builder.getOrCreate()
4 # create a DataFrame from persons.csv
5 df = spark.read.load( "persons.csv",
6                       format = "csv",
```

```
7                       header = True,
8                       inferSchema = True)
9  # define an RDD according to the content of the DataFrame
10 rddRows = df.rdd
11 # use the map transformation to extract the name field/column
12 rddNames = rddRows.map(lambda row: row.name)
13 # store the result
14 rddNames.saveAsTextFile(outputPath)
```

3.2 DataFrame Operations

The Spark SQL framework provides a set of specific methods for the DataFrame class, e.g., `show()`, `printSchema()`, `count()`, `distinct()`, `select()`, `filter()`, together with the standard `collect()` and `count()` actions.

3.2.1 Show

The `show(n)` method of the DataFrame class prints on the standard output the first `n` objects of the input DataFrame (default value `n = 20`).

Example. Create a DataFrame from a CSV file containing the profiles of a set of persons and print the content of the first 2 person (i.e., the first 2 rows of the DataFrame).

Input - `persons.csv`:

```
1 name, age
2 Andy, 30
3 Michael,
4 Justin, 19
```

Spark application:

```
1 .........
2 # create a Spark Session object
3 spark = SparkSession.builder.getOrCreate()
4 # create a DataFrame from persons.csv
5 df = spark.read.load("persons.csv",
6                       format = "csv",
7                       header = True,
8                       inferSchema = True)
9 df.show(2)
```

3.2.2 PrintSchema

The `printSchema()` method of the DataFrame class prints on the standard output the schema of the DataFrame, i.e., the name of the data attributes within the DataFrame.

3.2.3 Count

The `count()` method of the DataFrame class returns the number of rows in the input DataFrame.

3.2.4 Distinct

The `distinct()` method of the DataFrame class returns an output DataFrame containing only the unique rows of the input DataFrame. Pay attention that the distinct operation results always an heavy operation in terms of data sent over the network due to the shuffle operation.

Example. Create a DataFrame from a CSV file (with the header) containing the names of a set of persons, and then create a novel DataFrame without duplicates.

Input - `names.csv`:

```
name
Andy
Michael
Justin
Michael
```

Spark application:

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from names.csv
df = spark.read.load("names.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
df_distinct = df.distinct()
```

3.2.5 Select

The `select(column 1, ..., column n)` method of the DataFrame class returns an output DataFrame containing only the parameter-specifiable columns of the input DataFrame. The method returns every column by passing the special character `*` as parameter. Pay attention that passing incorrect column names causes the select method to generate errors at runtime.

Example. Create a DataFrame from the `persons2.csv` file that contains the profiles of a set of persons (the first line contains the header and the others lines contain the users' profiles, with one line per person). Each line contains the name, the age, the gender of a person, e.g. `Paul, 40, male`. Create a novel DataFrame containing only name and age of the persons.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons2.csv
df = spark.read.load("persons2.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
dfNamesAges = df.select("name", "age")
```

3.2.6 SelectExpr

The `selectExpr(expression 1, ..., expression n)` method of the DataFrame class results a variant of the `select` method, with `expr` referencing a (possible) SQL expression, e.g. `df.selectExpr("name", "age")` or `df.selectExpr("name", "age + 1 AS novel age")`.

Example. Create a DataFrame from the `persons2.csv` file that contains the profiles of a set of persons (the first line contains the header and the others lines contain the users' profiles, with one line per person). Each line contains the name, the age, the gender of a person, e.g. `Paul, 40, male`. Create a novel DataFrame containing the same columns of the initial dataset plus an additional column, namely `newAge`, containing the value of age `+ 1`.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons2.csv
df = spark.read.load("persons2.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
# create a new DataFrame with four columns:
# name, age, gender, newAge = age +1
dfNewAge = df.selectExpr("name", "age", "gender", "age + 1 as newAge")
```

3.2.7 Filter

The `filter(conditionExpr)` method of the DataFrame class returns an output DataFrame containing only the rows satisfying the parameter-specifiable condition. The condition represents a Boolean SQL expression. Pay attention that passing a filter expression with syntax errors causes the filter method to generate errors at runtime. The parameter of the method appears as a string and the system check the correctness of the expression at runtime (not a compile time).

Example. Create a DataFrame from the `persons.csv` file that contains the profiles of a set of persons, where the first line contains the header and the others lines contain the users' profiles (each line contains name and age of a person). Create a novel DataFrame containing only the persons with age between 20 and 31.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
df_filtered = df.filter("age >= 20 and age <= 31")
```

3.2.8 Where

The `where(expression)` method of the DataFrame class represents an alias of the `filter(conditionExpr)` method.

3.2.9 Join

The `join(right, on, how)` method of the DataFrame class returns an output DataFrame containing the join of the tuples coming from two input DataFrames, according to the `on` join condition. In other words, the `join(right, on, how)` method attempts to join two input DataFrames, with:

- `on`: the join condition (1.) as a string (the join column), (2.) as list of strings (multiple join columns) or (3.) as condition/an expression on the columns, e.g. `joindf = df.join(df2, df.name == df2.name)`
- `how`: the type of join, i.e. `inner, cross, outer, full, full outer, left, left outer, right, right outer, left semi, and left anti` (default: `inner`).

Pay attention that passing a filter expression with syntax errors causes the join method to generate errors at runtime.

Example. Create two DataFrames, one according to the `personsid.csv` file that contains the profiles of a set of persons (with schema: `uid, name, age`), and one acccording to the `likedsports.csv` file that contains the liked sports for each person (with schema: `uid, sportname`). Join the content of the two DataFrames (`uid` represents the join column) and show it on the standard output.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# read personsid.csv and store it in a DataFrame
dfPersons = spark.read.load("personsid.csv",
                            format = "csv",
                            header = True,
                            inferSchema = True)
# read likedsports.csv and store it in a DataFrame
dfUidSports = spark.read.load("likedsports.csv",
                              format = "csv",
                              header = True,
                              inferSchema = True)
# join the two input DataFrames
dfPersonLikes = dfPersons.join(dfUidSports, dfPersons.uid == dfUidSports.uid)
# Print the result on the standard output
dfPersonLikes.show()
```

The `on` join condition of the above code, i.e. `dfPersons.uid == dfUidSports.uid`, specifies the join condition on the `uid` columns.

Example. Create two DataFrames, one according to the `personsid.csv` file that contains the profiles of a set of persons (with schema: `uid, name, age`), and one acccording to the `banned.csv` file that contains the banned users (with

schema: `uid, bannedmotivation`). Join the content of the two DataFrames (`uid` represents the join column) and show it on the standard output.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# read personsid.csv and store it in a DataFrame
dfPersons = spark.read.load("personsid.csv",
                            format = "csv",
                            header = True,
                            inferSchema = True)
# read banned.csv and store it in a DataFrame
dfBannedUsers = spark.read.load("banned.csv",
                              format = "csv",
                              header = True,
                              inferSchema = True)
# apply the Left Anti Join on the two input DataFrames
dfSelectedProfiles = dfPersons.join(dfBannedUsers,
                                    dfPersons.uid == dfBannedUsers.uid,
                                    "left_anti")
# print the result on the standard output
dfSelectedProfiles.show()
```

3.2.10 Aggregate functions

The Spark SQL framework provides some aggregate functions to compute aggregates over the set of column values, e.g. `avg(column)`, `count(column)`, `sum(column)`, `abs(column)`, etc. Each aggregate function returns a single value coming from the computation over the input column values. The `agg(expr)` method of the DataFrame class enables to specify the aggregate functions to apply and the respective input columns. The method returns an output DataFrame containing a single row and column for each of the parameter-specifiable aggregate functions. The name of the resulting column compatible with each input aggregate function results `function-name(column)`, e.g. `avg(age)` or `count(name)`. Pay attention that passing incorrect attribute names or incorrect data types causes the aggregate method to generate errors at runtime.

Example. Create a DataFrame from the `persons.csv` file containing the profiles of a set of persons (with the header). Create a Dataset containing the average value of age.

Input - `persons.csv`:

```
name, age
Andy, 30
Michael, 15
Justin, 19
Andy, 40
```

Spark application:

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format = "csv",
                     header = True,
```

```
                        inferSchema = True)
# compute the average of age
averageAge = df.agg({"age": "avg"})
```

Output:

```
avg(age)
26.0
```

3.2.11 GroupBy

The `groupBy(column 1, ..., column n)` method of the DataFrame class, together with a set of aggregate methods, appears suitable to split the input data into groups and compute aggregate function over each group. Pay attention that passing semantic errors (e.g. incorrect attribute names, incorrect data types) causes the aggregate method to generate errors at runtime. The `groupBy(column 1, ..., column n)` method enables to set as parameters the attributes on which to split the input data into groups. Afterwards, by applying some aggregate functions (e.g. `avg(column)`, `count(column)`, `sum(column)`, `abs(column)`, etc.), the application returns an output DataFrame with the results. The `agg(..)` method enables to apply multiple aggregate functions at the same time over each group. See the static methods of the `pyspark.sql.GroupedData` class for a complete list.

Example. Create a DataFrame from the `persons.csv` file containing the profiles of a set of persons (with the header). Create a DataFrame containing, for each name, the average value of age.

Input - `persons.csv`:

```
name, age
Andy, 30
Michael, 15
Justin, 19
Andy, 40
```

Spark application:

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
# compute the average of age
grouped = df.groupBy("name").avg("age")
```

Output:

```
name, avg(age)
Andy, 35
Michael, 15
Justin, 19
```

Example. Create a DataFrame from the `persons.csv` file containing the profiles of a set of persons (with the header). Create a DataFrame containing, for each name, the average value of age and the number of person with that name.

Input - `persons.csv`:

```
name, age
Andy, 30
Michael, 15
Justin, 19
Andy, 40
```

Spark application:

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
# compute the average of age
grouped = df.groupBy("name").agg({"age": "avg", "name": "count"})
```

Output:

```
name, avg(age), count(name)
Andy, 35, 2
Michael, 15, 1
Justin, 19, 1
```

3.2.12 Sort

The `sort(column 1, ..., column n, ascending = True)` method of the DataFrame class returns an output DataFrame containing the same data of the input DataFrame in ascending order (the `ascending` parameter determines the ascending or descending order of the output data via boolean condition, i.e. `True` or `False`).

3.3 DataFrames and the SQL language

The Spark SQL framework enables querying the content of a DataFrame also via the SQL language. In order to query the content of a DataFrame, the framework requires a "table name" to associate with the DataFrame. The `createOrReplaceTempView(tableName)` method of the DataFrame class enables to assign a "table name" to the DataFrame. Once the application executes the mapping from the DataFrame into "table names", the framework enables to executes SQL-like queries. Note that such SQL-like queries return DataFrame objects. The `sql(query)` method of the `SparkSession` class enables to execute an SQL-like query.

Example. Create a DataFrame from a JSON file containing the profiles of a set of persons (each line of the file contains a JSON object containing name, age, and gender of a person). Create a novel DataFrame containing only the persons with age between 20 and 31 and print them on the standard output. Use the SQL language to perform the operation.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons.csv
df = spark.read.load("persons.json",
                     format = "json")

# assign the "table name" people to the df DataFrame
df.createOrReplaceTempView("people");

# select the persons with age between 20 and 31
# by querying the people table
selectedPersons = spark.sql("SELECT * FROM people WHERE age >= 20 and age <= 31")

# print the result on the standard output
selectedPersons.show()
```

Example. Create two DataFrames, one according to the `personsid.csv` file that contains the profiles of a set of persons (with schema: `uid, name, age`), and one according to the `likedsports.csv` file that contains the liked sports for each person (with schema: `uid, sportname`). Join the content of the two DataFrames and show it on the standard output.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# read personsid.csv and store it in a DataFrame
dfPersons = spark.read.load("personsid.csv",
                            format = "csv",
                            header = True,
                            inferSchema = True)

# assign the "table name" people to the dfPerson
dfPersons.createOrReplaceTempView("people")

# read likedsports.csv and store it in a DataFrame
dfUidSports = spark.read.load("liked_sports.csv",
                              format = "csv",
                              header = True,
                              inferSchema = True)

# assign the "table name" liked to dfUidSports
dfUidSports.createOrReplaceTempView("liked")

# join the two input tables by using the SQL-like syntax
dfPersonLikes = spark.sql("SELECT * from people, liked where people.uid = liked.uid")

# print the result on the standard output
dfPersonLikes.show()
```

Example. Create a DataFrame from the `persons.json` file that contains the profiles of a set of persons (the first line contains the header and the others lines contain the users' profiles). Each line contains name and age of a person. Create a DataFrame containing for each name the average value of age and the number of person with that name. Print its content on the standard output.

Input - `persons.json`:

```
name, age
Andy, 30
Michael, 15
Justin, 19
Andy, 40
```

Spark application:

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons.json
df = spark.read.load("persons.json",
                     format = "json")

# assign the "table name" people to the df DataFrame
df.createOrReplaceTempView("people")

# define groups according to the value of name and
# compute average and number of records for each group
nameAvgAgeCount = spark.sql("SELECT name, avg(age), count(name) FROM people GROUP BY name")

# print the result on the standard output
nameAvgAgeCount.show()
```

Output:

```
name, avg(age), count(name)
Andy, 35, 2
Michael, 15, 1
Justin, 19, 1
```

3.4 Save DataFrames

The Spark SQL framework enables to store the content of DataFrames on disk by means of two different approaches:

- convert DataFrames into traditional RDDs via the `rdd` method of the DataFrame class and store the content via the standard `saveAsTextFile(outputFolder)` method of the RDD class;
- store the content via the `write()` method of the DataFrame class, i.e. by means of a `DatFrameWriter` class instance.

Example. Create a DataFrame from the `persons.csv` file containing the profiles of a set of persons (with the header). Each line contains name and age of a single person. Store the DataFrame in the output folder via the `saveAsTextFile(..)` method.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
# save it
df.rdd.saveAsTextFile(outputPath)
```

Example. Create a DataFrame from the `persons.csv` file containing the profiles of a set of persons (with the header). Each line contains name and age of a single person. Store the DataFrame in the output folder (in CSV format) via the `write()` method.

```
.........
# create a Spark Session object
spark = SparkSession.builder.getOrCreate()
# create a DataFrame from persons.csv
df = spark.read.load("persons.csv",
                     format = "csv",
                     header = True,
                     inferSchema = True)
# save it
df.write.csv(outputPath, header = True)
```

Chapter 4

Apache Spark MLlib

The Spark MLlib (Apache Spark's scalable machine learning library) framework refers to the Spark component for executing machine learning/data mining algorithms, e.g. pre-processing techniques, classification (supervised learning), clustering (unsupervised learning) and itemset mining. The MLlib APIs appear split into two packages:

- `pyspark.mllib`, providing the original APIs built on top of RDDs;
- `pyspark.ml`, providing higher-level API built on top of DataFrames (i.e, Dataset of `Row` objects) for constructing ML pipelines.

4.1 Spark MLlib - Data Types

The Spark MLlib framework relies on a set of basic local and distributable data types, e.g. local vector, local matrix, distributable matrix, and others. DataFrames for ML (machine leraning) applications contain objects relying on such basic data types.

4.1.1 Local Vectors

The local `pyspark.ml.linalg.Vector` objects of the MLlib library enables to store vectors of double values, i.e. the data type suitable to represent the input records/data (with a vector for each input record) for machine learning algorithms. The framework requires to map non double attributes/values to double values before applying MLlib algorithms (MLlib algorithms rely the computation on vectors of double values). The framework also supports dense vector and sparse vector, e.g., the vector of doubles `[1.0, 0.0, 3.0]` in dense format as `[1.0, 0.0, 3.0]`, or in sparse format as `(3, [0, 2], [1.0, 3.0])` (where `3` represents the size of the vector, the array `[0,2]` contains the indexes of the non-zero cells and the array `[1.0, 3.0]` contains the values of the non-zero

cells). The subsequent code shows the creation of dense and sparse vectors in Spark.

```
.........
from pyspark.ml.linalg import Vectors
# create a dense vector [1.0, 0.0, 3.0]
dv = Vectors.dense([1.0, 0.0, 3.0])
# create a sparse vector [1.0, 0.0, 3.0] by specifying
# its indices and values corresponding to non-zero entries
# by means of a dictionary
sv = Vectors.sparse(3, {0: 1.0, 2: 3.0})
```

In the `sv` sparse vector, `3` refers to the vector size, {`0: 1.0, 2: 3.0`} represents the dictionary of `index: value` pairs, i.e. {`index 0: value 1.0, index 1: value 0.0, index 2: value 3.0`}

4.1.2 Local Matrices

The local `pyspark.ml.linalg.Matrix` objects of the MLlib library enables to store matrices of double values. The framework also supports dense matrices and sparse matrices. The column-major order enables to store the content of the matrix in a linear way. The subsequent code shows the creation of dense and sparse matrices in Spark.

```
.........
from pyspark.ml.linalg import Matrices
# create a dense matrix with two rows and three columns
# 3.0 0.0 0.0
# 1.0 1.5 2.0
dm = Matrices.dense(2, 3, [3.0, 1.0, 0.0, 1.5, 0.0, 2.0])
# create a sparse version of the same matrix
sm = Matrices.sparse(2, 3, [0, 2, 3, 4], [0, 1, 1, 1], [3, 1, 1.5, 2])
```

In the `dm` dense matrix, `2` refers to the number of rows, `3` refers to the number of columns, `[3.0, 1.0, 0.0, 1.5, 0.0, 2.0]` represents the values in column-major order. In the `sm` sparse matrix, `2` refers to the number of rows, `3` refers to the number of columns, `[0, 2, 3, 4]` contains an element per column that encodes the offset in the array of non-zero values, i.e. the integer indicating the distance (displacement) between the beginning of the value (the last element represents the number of non-zero values), `[0, 1, 1, 1]` refers to the row index of each non-zero values, `[3, 1, 1.5, 2]` represents the array of non-zero values of the matrix.

4.2 Spark MLlib - Main Concepts

The Spark MLlib framework requires DataFrames as input data, i.e. data in tabular format, to execute ML algorithms. The framework then represents each input data via "tables" before applying the MLlib algorithms. The framework also represents collections of documents in tabular format before applying MLlib algorithms. The MLlib algorithms associates each of the several DataFrames columns with a different role/meaning:

- **label**: target of a classification/regression analysis;

- **features**: a vector containing the values of the attributes/features of the input records/data points;
- **text**: the original text of a document before the transformation in a tabular format;
- **prediction**: the prediction values of a classification/regression analysis.
- ....

4.2.1 Transformer

The transformer represents a ML algorithm/procedure that transforms a DataFrame into another DataFrame via the `transform(inputDataFrame)` method, e.g., a feature transformer reads an input DataFrame, maps an "original" column (e.g., text) into an "additional" column (e.g., feature vectors), and returns an output DataFrame with the original columns plus the column coming from the mapping procedure. For instance, the classification model represents a transformer transforming a DataFrame with only features into a DataFrame with features and predictions (prediction column).

4.2.2 Estimator

The estimator represents a ML algorithm/procedure to fit over an input (training) DataFrame in order to produce a transformer. Each estimator implements the `fit()` method, which reads a DataFrame and outputs a model of type transformer. An Estimator abstracts the concept of a learning algorithm, or any algorithm that fits/trains on an input dataset and returns a model. For instance, the logistic regression classification algorithm represents an estimator executing `fit(input training DataFrame)` to return a logistic regression model, which represents a model/a transformer.

4.2.3 Pipeline

The pipeline chains multiple transformers and estimators together to set a machine learning/data mining workflow, i.e. with the output of a transformer/estimator as input of the subsequent in the pipeline. For instance, a simple text document processing workflow aiming at creating a classification model includes several steps: (1.) split each document into a set of words, (2) convert each set of words into a numerical feature vector and (3.) learn a prediction model leveraging the feature vectors and the respective class labels. The APIs of Spark MLlib recommend the pipeline approach for solving ML problems. The pipeline approach relies on the subsequent steps:

1. instantiate the set of transformers and estimators;
2. implement the pipeline object with the representative sequence of transformers and estimators;

3. execute the pipeline (training of the model);
4. (optional) apply the model on different data.

4.2.4 Parameters

In the Spark MLlib paradigm, transformers and estimators share common APIs for specifying parameter values.

4.3 Data Preprocessing

The framework processes the input data before applying machine learning and data mining algorithms, in order to:

- organize data in a format consistent with algorithms requirements;
- define reasonable (predictive) features;
- remove bias, e.g., normalization;
- remove noise and missing values.

The Spark MLlib library provides a set of transformers suitable to extract, transform and select features from input DataFrames, e.g. feature extractors (`TF-IDF`, `Word2Vec`, ...), feature transformers (`Tokenizer`, `StopWordsRemover`, `StringIndexer`, `IndexToString`, `OneHotEncoderEstimator`, `Normalizer`, ...), feature selectors (`VectorSlicer`, ...).

4.3.1 Feature Transformations

The Spark MLlib framework provides several techniques to transform features, useful to create different columns/features by combining or transforming other features. The developers implement feature transformations and feature creations by means of the standard methods already available for DataFrames and RDDs.

Vector Assembler

The `VectorAssembler` (`pyspark.ml.feature.VectorAssembler`) represents a transformer that combines a list of columns into a single vector column, useful for combining features into a single feature vector before applying ML algorithms. The `VectorAssembler(inputCols, outputCol)` method relies on the (1.) `inputCols` parameter, representing the list of original columns to include in the output column of type `Vector` (of numeric types, boolean type, or vector type), and the (2.) `outputCol` parameter, representing the name of the output column. After the execution on an input DataFrame, the `transform()` method of `VectorAssembler` returns an output DataFrame with an output column (`outputCol`). For each record, the value of the output column results the

"concatenation" of the values of the input columns. The resulting DataFrame also contains the same columns of the input DataFrame.

Example. Consider an input DataFrame with three columns. Create a novel DataFrame with a novel column containing the "concatenation" of `colB` and `colC` in a novel vector column. Set the name of the novel column to `features`.

Input:

colA	colB	colC
1	4.5	True
2	0.6	True
3	1.5	False
4	12.1	True
5	0.0	True

```
.........
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler

# input and output folders
inputPath = "data/exampleDataAssembler.csv"
# create a DataFrame from the input data
inputDF = spark.read.load(inputPath,
                          format = "csv",
                          header = True,
                          inferSchema = True)

# create a VectorAssembler that combines columns colB and colC
# set the novel vetor column name to features
myVectorAssembler = VectorAssembler(inputCols = ['colB', 'colC'], outputCol = 'features')

# apply myVectorAssembler on the input DataFrame
transformedDF = myVectorAssembler.transform(inputDF)
```

Output:

colA	colB	colC	features
1	4.5	True	[4.5, 1.0]
2	0.6	True	[0.6, 1.0]
3	1.5	False	[1.5, 0.0]
4	12.1	True	[12.1, 1.0]
5	0.0	True	[0.0, 1.0]

4.3.2 Data Normalization

The MLlib framework provides a set of normalization algorithms (namely "scalers"), e.g. `StandardScaler`, `MinMaxScaler`, `Normalizer` and `MaxAbsScaler`.

StandardScaler

The `StandardScaler` (`pyspark.ml.feature.StandardScaler`) represents an estimator to return a transformer of type `pyspark.ml.feature.StandardScalerModel`. The `StandardScalerModel` transforms a vector column of an input DataFrame by normalizing each "feature" of the input vector column to appear with unit standard deviation and/or zero mean. The `StandardScaler(inputCol,outputCol)` method relies on the (1.) `inputCol` parameter, representing the name of the input vector column (of doubles) to normalize, and the (2.) `outputCol` parameter, representing the name of the output vector column coming from the normalization. Execute the `fit()` method of `StandardScaler` over the input DataFrame to infer a `StandardScalerModel`, i.e. to return a transformer. Then, execute the `transform()` method of `StandardScalerModel` over the input DataFrame to return an output DataFrame containing the output column (`outputCol`). For each record, the value of the output column results the normalization version of the input vector column. The output DataFrame also contains each column of the input DataFrame.

Example. Consider an input DataFrame with four columns. Create a novel DataFrame with a novel column containing the normalization version of the vector column `features`. Set the name of the novel column to `scaledFeatures`.

Input:

colA	colB	colC
1	4.5	True
2	0.6	True
3	1.5	False
4	12.1	True
5	0.0	True

```
.........
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
# input and output folders
inputPath = "data/exampleDataAssembler.csv"
# create a DataFrame from the input data
inputDF = spark.read.load(inputPath,
                          format = "csv",
                          header = True,
                          inferSchema = True)

# create a VectorAssembler that combines columns colB and colC
# set the novel vetor column name to features
myVectorAssembler = VectorAssembler(inputCols = ['colB', 'colC'], outputCol = 'features')
# apply myVectorAssembler on the input DataFrame
transformedDF = myVectorAssembler.transform(inputDF)

# create a Standard Scaler to scale the content of features
myScaler = StandardScaler(inputCol = "features", outputCol = "scaledFeatures")
# compute summary statistics by fitting the StandardScaler
```

```
22 # before normalizing the content of the data -> compute mean and
23 # standard deviation of the data
24 scalerModel = myScaler.fit(transformedDF)
25 # apply myScaler on the input column features
26 scaledDF = scalerModel.transform(transformedDF)
```

Output:

colA	colB	colC	features	scaledFeatures
1	4.5	True	[4.5, 1.0]	[0.903, 2.236]
2	0.6	True	[0.6, 1.0]	[0.120, 2.236]
3	1.5	False	[1.5, 0.0]	[0.301, 0.0]
4	12.1	True	[12.1, 1.0]	[2.428,2.236]
5	0.0	True	[0.0, 1.0]	[0.0 ,2.236]

4.3.3 Categorical Columns

In most cases ML applications deal with categorical input data, i.e. with categorical attributes (e.g. string columns), for example in classification problems with categorical class labels. However, Spark MLlib's classification and regression algorithms only process numeric values, and therefore the framework requires a pre-processing step to map categorical columns into double values.

StringIndexer

The `StringIndexer` (`pyspark.ml.feature.StringIndexer`) represents an estimator to return a transformer of type `pyspark.ml.feature.StringIndexerModel`. The `StringIndexerModel` encodes a string column of "labels" into a integer column of "label indices", i.e. maps each distinct value of the input string column into an integer value in the range `[0, num. distinct values)`. The `StringIndexer(inputCol, outputCol)` method relies on the (1.) `inputCol` parameter, representing the name of the input string column to map into a set of integers, and the (2.) `outputCol` parameter, representing the name of the output column. Execute the `fit()` method of `StringIndexer` over the input DataFrame to infer a `StringIndexerModel`, i.e. to return a transformer. Then, execute the `transform()` method of `StringIndexerModel` over the input DataFrame to return an output DataFrame containing the output column (`outputCol`). For each record, the value of the output column results the integer (to a double) relating to the value of the input string column.

Example. Consider an input DataFrame with two columns. Create a novel DataFrame with a novel column containing the "integer" version of the string column `category`. Set the name of the novel column to `categoryIndex`.

Input:

id	category
1	a
2	b
3	c
4	c
5	a

```
.........
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import StringIndexer
# input DataFrame
df = spark.createDataFrame([(1, "a"), (2, "b"), (3, "c"), (4, "c"), (5, "a")], ["id", "category"])

# create a StringIndexer to map the content of category to a set of "integers"
indexer = StringIndexer(inputCol = "category", outputCol = "categoryIndex")
# analyze the input data to define the mapping string -> integer
indexerModel = indexer.fit(df)
# apply indexerModel on the input column category
indexedDF = indexerModel.transform(df)
```

Output:

id	category	categoryIndex
1	a	0.0
2	b	2.0
3	c	1.0
4	c	1.0
5	a	0.0

IndexToString

The `IndexToString` (`pyspark.ml.feature.IndexToString`) represents a transformer that maps a numerical column of "label indices" back to a string column of original "labels" (symmetrical to `StringIndexer`). The classification model returns the integer version of the prediction label values, therefore the `IndexToString` transformer enables to remap the integer values to the original categorical classes. The `IndexToString(inputCol, outputCol, labels)` method relies on the (1.) `inputCol` parameter, representing the name of the input numerical column to map to the original set of string "labels", the (2.) `outputCol` parameter, representing the name of the output column, and the (3.) `labels`, representing the list of original "labels"/strings (the mapping with integer values comes from the positions of the strings inside labels). Execute the `transform()` method of `IndexToString` over the input DataFrame to return an output DataFrame containing the output column (`outputCol`). For each record, the value of the output column results the original string relating to the value of the input numerical column.

Example. Consider an input DataFrame with two columns. Create a novel DataFrame with a novel column (`categoryIndex`) containing the "integer" version of the string column `category` and then map it back to the string version in a novel column. Set the name of the novel column to `originalCategory`.

Input:

id	category
1	a
2	b
3	c
4	c
5	a

```
.........
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import StringIndexer
# input DataFrame
df = spark.createDataFrame([(1, "a"), (2, "b"), (3, "c"), (4, "c"), (5, "a")], ["id", "category"])

# create a StringIndexer to map the content of category to a set of     integers
indexer = StringIndexer(inputCol = "category", outputCol = "categoryIndex")
# analyze the input data to define the mapping string -> integer
indexerModel = indexer.fit(df)
# apply indexerModel on the input column category
indexedDF = indexerModel.transform(df)

# create an IndexToString to map the content of numerical attribute categoryIndex
# to the original string value
converter = IndexToString(inputCol = "categoryIndex",
                          outputCol = "originalCategory",
                          labels = indexerModel.labels)
# apply converter on the input column
categoryIndex reconvertedDF = converter.transform(indexedDF)
```

Output:

id	category	categoryIndex	originalCategory
1	a	0.0	a
2	b	2.0	b
3	c	1.0	c
4	c	1.0	c
5	a	0.0	a

4.3.4 SQLTransformer

The `SQLTransformer` (`pyspark.ml.feature.SQLTransformer`) represents a transformer that implements the transformations definiable by SQL queries. The `SQLTransformer` executes an SQL query on the input DataFrame and returns an output DataFrame with the result of the query. The `SQLTransformer(statement)`

method relies on the `statement` parameter, i.e. the SQL query to execute.

Example. Consider an input DataFrame with two columns: `text` and `id`. Create a novel DataFrame with a novel column, namely `numWords`, containing the number of words occurring in column text.

Input:

id	text
1	This is Spark
2	Spark
3	Another sample sentence of words
4	Paolo Rossi
5	Giovanni

```
.........
from pyspark.sql.types import *
from pyspark.ml.feature import SQLTransformer
# local input data
inputList = [(1, "This is Spark"),
             (2, "Spark"),
             (3, "Another sample sentence of words"),
             (4, "Paolo Rossi"),
             (5, "Giovanni")]
# create the initial DataFrame
dfInput = spark.createDataFrame(inputList, ["id", "text"])
# define a UDF function that counts the number of words in an input string
spark.udf.register("countWords", lambda text: len(text.split(" ")), IntegerType())
# define an SQLTranformer to create the columns of interest
sqlTrans = SQLTransformer(statement = """SELECT *, countWords(text) AS numLines FROM __THIS__""")
# create the new DataFrame by invoking the transform method of the SQLTranformer
newDF = sqlTrans.transform(dfInput)
```

Output:

id	text	numWords
1	This is Spark	3
2	Spark	1
3	Another sample sentence of words	5
4	Paolo Rossi	2
5	Giovanni	1

4.4 Classification Algorithms

The Spark MLlib framework provides a set of classification algorithms, e.g. logistic regression (binomial logistic regression and multinomial logistic regression), decision tree classifier, random forest classifier, gradient-boosted tree classifier, multilayer perceptron classifier, linear support vector machine. The

available classification algorithms rely on two phases: (1.) model generation according to a set of training data, and (2.) class label prediction for external data (without labels). The classification algorithms available in Spark operate only on numerical attributes, i.e. the framework maps categorical values to integer values (one distinct value per class) before applying MLlib classification algorithms. The framework trains the Spark classification algorithms on top of an input DataFrame containing (at least) two columns: (1.) **label**, the class label (in integer value format), containing the values of the attribute to predict via classification model, and (2.) **features**, a vector of doubles (`pyspark.ml.linalg.Vector` data type, both dense and sparse) containing the values of the predictive attributes of the input records/data points.

Suppose a classification problem to predict customer goodness (class label: good customer/bad customer) according to monthly income and number of children (predictive attributes: monthly income, number of children).

CustomerType	MonthlyIncome	NumChildren
Good customer	1400.0	2
Bad customer	11105.5	0
Good customer	2150.0	2

In order to train a MLlib classification algorithm, the framework (1.) maps the categorical values (good customer/bad customer) to integer values (`1/0`), and (2.) stores the values of the predictive attributes in vectors of doubles (one single vector for each input record).

label	features
1	[1400.0, 2]
0	[11105.5, 0]
1	[2150.0, 2]

4.4.1 Classification

The classification problem via logistic regression algorithm branches into:

1. analyze the **training data** (i.e. the example records/data points with class label values) to infer a classification model according to the logistic regression algorithm;

2. apply the model over the **test data** to predict missing class labels.

Suppose to store the input training data in a text file (with the header). The file contains a single record/data point per line in a structurable format, i.e. with a fixable number of attributes (four), with the first column of each record containing the boolean value of the class label, i.e. `1/0` (binary classification problem), and the others three columns containing the double values of predictive attributes.

Example input training data file:

label	attr1	attr2	attr3
1.0	0.0	1.1	0.1
0.0	2.0	1.0	-1.0
0.0	2.0	1.3	1.0
1.0	0.0	1.2	-0.5

The first operation consists in transforming the content of the input training file into a DataFrame containing two columns: (1.) the **label** column, with double values (`1/0`) storing the label of each training record, and (2.) the **features** column, with vector of doubles storing the values of the predictive features.

Training DataFrame:

label	features
1.0	[0.0, 1.1, 0.1]
0.0	[2.0, 1.0, -1.0]
0.0	[2.0, 1.3, 1.0]
1.0	[0.0, 1.2, -0.5]

After the training step, the framework applies the model over the test data to predict missing class labels.

Example input test data file:

label	attr1	attr2	attr3
	-1.0	1.5	1.3
	3.0	2.0	-0.1
	0.0	2.2	-1.5

The second operation consists in transforming the content of the input test file into a DataFrame containing two columns: (1.) the **label** column, with `null` values storing the missing label of each training record, and (2.) the **features** column, with vector of doubles storing the values of the predictive features.

Test DataFrame:

label	features
null	[-1.0, 1.5, 1.3]
null	[3.0, 2.0, -0.1]
null	[0.0, 2.2, -1.5]

After the prediction step, the Spark framework returns an output DataFrame containing the same columns as the input DataFrame plus three more output columns: the **prediction** column and the **probability** columns. For each input test record, the "prediction" column contains the prediction of the class label, and the respective probability in the corresponding "probability" columns.

Output DataFrame:

label	features	prediction	rawPrediction	probability
null	[-1.0, 1.5, 1.3]	1.0	...	...
null	[3.0, 2.0, -0.1]	0.0	...	...
null	[0.0, 2.2, -1.5]	1.0	...	...

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
# input and output folders
trainingData = "ex_data/trainingData.csv"
unlabeledData = "ex_data/unlabeledData.csv"
outputPath = "predictionsLR/"

# *************************
# training step
# *************************
# create a DataFrame from trainingData.csv
# training data in raw format
trainingData = spark.read.load(trainingData,
                               format = "csv",
                               header = True,
                               inferSchema = True)

# define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols = ["attr1", "attr2", "attr3"], outputCol = "features")
# apply the assembler to create column features for the training data
trainingDataDF = assembler.transform(trainingData)

# create a LogisticRegression object.
# logisticRegression represents an Estimator, useful to
# create a classification model according to logistic regression.
lr = LogisticRegression()
# set the values of the parameters of the
# Logistic Regression algorithm via the setter methods.
# (one set method for each parameter)
# for example, set the number of maximum iterations to 10
# and the regularization parameter to 0.01
lr.setMaxIter(10)
lr.setRegParam(0.01)
```

```
36
37 # train a logistic regression model on the training data
38 classificationModel = lr.fit(trainingDataDF)
39
40 # ************************
41 # prediction step
42 # ************************
43 # create a DataFrame from unlabeledData.csv
44 # unlabeled data in raw format
45 unlabeledData = spark.read.load(unlabeledData,
46                                 format = "csv",
47                                 header = True,
48                                 inferSchema = True)
49
50 # apply the same assembler as before to create column features
51 unlabeledDataDF = assembler.transform(unlabeledData)
52
53 # make predictions on the unlabled data via the transform() method of the
54 # trained classification model transform
55 # leverage only the content of 'features' to perform the predictions
56 predictionsDF = classificationModel.transform(unlabeledDataDF)
57
58 # the returned DataFrame appears with the following schema (attributes)
59 # - attr1
60 # - attr2
61 # - attr3
62 # - features: vector (values of the attributes)
63 # - label: double (value of the class label)
64 # - rawPrediction: vector (nullable = true)
65 # - probability: vector (the i-th cell contains the probability that the current
66 # record belongs to the i-th class
67 # - prediction: double (the predicted class label)
68 # select only the original features (i.e., the value of the original attributes
69 # attr1, attr2, attr3) and the predicted class for each record
70 predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")
71 # save the result in an HDFS output folder
72 predictions.write.csv(outputPath, header = "true")
```

The above solution applies the same preprocessing steps on both training and test data, i.e. the same assembler on both input data. The framework recommends the "pipeline" functionality to set up common steps once.

```
1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.classification import LogisticRegression
4  from pyspark.ml import Pipeline
5  from pyspark.ml import PipelineModel
6  # input and output folders
7  trainingData = "ex_data/trainingData.csv"
8  unlabeledData = "ex_data/unlabeledData.csv"
9  outputPath = "predictionsLR/"
10
11 # ************************
12 # training step
13 # ************************
14 # create a DataFrame from trainingData.csv
15 # training data in raw format
16 trainingData = spark.read.load(trainingData,
17                                format = "csv",
18                                header = True,
19                                inferSchema = True)
20 # define an assembler to create a column (features) of type Vector
21 # containing the double values associated with columns attr1, attr2, attr3
22 assembler = VectorAssembler(inputCols = ["attr1", "attr2", "attr3"], outputCol = "features")
23 # create a LogisticRegression object.
24 # logisticRegression represents an Estimator, useful to
25 # create a classification model according to logistic regression.
```

```
lr = LogisticRegression()
# set the values of the parameters of the
# Logistic Regression algorithm via the setter methods.
# (one set method for each parameter)
# for example, set the number of maximum iterations to 10
# and the regularization parameter to 0.01
lr.setMaxIter(10)
lr.setRegParam(0.01)

# define a pipeline to create the logistic regression model on the training data.
# the pipeline includes also the preprocessing step
pipeline = Pipeline().setStages([assembler, lr])
# execute the pipeline on the training data to build the classification model
classificationModel = pipeline.fit(trainingData)
# now, the classification model appears suitable to predict the class label of the unlabeled data

# ************************
# prediction step
# ************************
# create a DataFrame from unlabeledData.csv
# unlabeled data in raw format
unlabeledData = spark.read.load(unlabeledData,
                                format = "csv",
                                header = True,
                                inferSchema = True)
# make predictions on the unlabled data via the transform() method of the
# trained classification model transform
# leverage only the content of 'features' to perform the predictions
# the model falls inside pipeline and hence
# the the transform() method executes also the assembler
predictionsDF = classificationModel.transform(unlabeledDataDF)

# the returned DataFrame appears with the following schema (attributes)
# - attr1
# - attr2
# - attr3
# - features: vector (values of the attributes)
# - label: double (value of the class label)
# - rawPrediction: vector (nullable = true)
# - probability: vector (the i-th cell contains the probability that the current
# record belongs to the i-th class
# - prediction: double (the predicted class label)
# select only the original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")
# save the result in an HDFS output folder
predictions.write.csv(outputPath, header = "true")
```

4.4.2 Decision Trees

The classification problem via decision tree algorithm branches into:

1. analyze the **training data** (i.e. the example records/data points with class label values) to infer a classification model according to the decision tree algorithm;

2. apply the model over the **test data** to predict missing class labels.

The classification problem via decision tree algorithm relies on the same main steps of the previous classification problem via logistic regression algorithm. The difference with respect to the previous code block appears in the definition and configuration part of the classification algorithm.

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
# input and output folders
trainingData = "ex_data/trainingData.csv"
unlabeledData = "ex_data/unlabeledData.csv"
outputPath = "predictionsLR/"

# *************************
# training step
# *************************
# create a DataFrame from trainingData.csv
# training data in raw format
trainingData = spark.read.load(trainingData,
                               format = "csv",
                               header = True,
                               inferSchema = True)
# define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols = ["attr1", "attr2", "attr3"], outputCol = "features")
# create a DecisionTreeClassifier object.
# DecisionTreeClassifier represents an Estimator, useful to
# create a classification model according to decision trees.
dt = DecisionTreeClassifier()
# set the values of the parameters of the Decision Tree
# for example, set the "gini index" measure to decide the splitting condition of a node
dt.setImpurity("gini")
# define a pipeline to create the decision tree model on the training data.
# the pipeline includes also the preprocessing step
pipeline = Pipeline().setStages([assembler, dt])
# execute the pipeline on the training data to build the
# classification model
classificationModel = pipeline.fit(trainingData)
# now, the classification model appears suitable to predict the class label of the unlabeled data

# *************************
# prediction step
# *************************
# create a DataFrame from unlabeledData.csv
# unlabeled data in raw format
unlabeledData = spark.read.load(unlabeledData,
                                format = "csv",
                                header = True,
                                inferSchema = True)
# make predictions on the unlabled data via the transform() method of the
# trained classification model transform
# leverage only the content of 'features' to perform the predictions
# the model falls inside pipeline and hence
# the the transform() method executes also the assembler
predictionsDF = classificationModel.transform(unlabeledDataDF)
# the returned DataFrame appears with the following schema (attributes)
# - attr1
# - attr2
# - attr3
# - features: vector (values of the attributes)
# - label: double (value of the class label)
# - rawPrediction: vector (nullable = true)
# - probability: vector (the i-th cell contains the probability that the current
# record belongs to the i-th class
# - prediction: double (the predicted class label)
# select only the original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")
# save the result in an HDFS output folder
predictions.write.csv(outputPath, header = "true")
```

4.4.3 Categorical Class Labels

In most cases ML applications deal with categorical input data, i.e. with categorical attributes (e.g. string columns), for example in classification problems with categorical class labels. However, Spark MLlib's classification and regression algorithms only process numeric values, and therefore the framework requires a pre-processing step to map categorical columns into double values.

The estimator `StringIndexer` and the transformer `IndexToString` support the transformation of categorical class label into numerical one and vice versa:

- `StringIndexer` maps each categorical class label to an integer value (castable to a double).
- `IndexToString` performs the opposite operation.

The Spark MLlib framework recommends to execute some pre-processing steps for classification/regression problems with categorical class labels, in order:

1. exploit `StringIndexer` to extend the input DataFrame with the **label** column, containing the numerical representation of the class label column;
2. create a **features** column of type vector, containing the predictive features;
3. infer a classification model via a classification algorithm (e.g., Decision Tree, Logistic regression), by considering the values of "features" and "label", i.e. without considering the other columns for the prediction model generation;
4. apply the model over the test data to predict numerical class labels;
5. exploit `IndexToString` to convert the numerical predictions (numerical class label values) to the original categorical values.

Suppose to store the input training data in a text file (with the header). The file contains a single record/data point per line in a structurable format, i.e. with a fixable number of attributes (four), with the first column of each record containing the categorical value of the class label, i.e. `positive/negative` (boolean classification problem), and the others three columns containing the double values of predictive attributes.

Example input training data file:

categoricalLabel	attr1	attr2	attr3
positive	0.0	1.1	0.1
negative	2.0	1.0	-1.0
negative	2.0	1.3	1.0

The first operation consists in transforming the content of the input training file into a DataFrame containing three columns: (1.) the original class label column, with categorical values (`positive/negative`) storing the categorical label of each training record, (2.) the **label** column, with the numerical representations (`1/0`) of the original class label column, and (3.) the **features** column, with vector of doubles storing the values of the predictive features.

Training DataFrame:

categoricalLabel	features	label
positive	[0.0, 1.1, 0.1]	1.0
negative	[2.0, 1.0, -1.0]	0.0
negative	[2.0, 1.3, 1.0]	0.0

After the training step, the framework applies the model over the test data to predict the missing class labels.

Example input test data file:

categoricalLabel	attr1	attr2	attr3
	-1.0	1.5	1.3
	3.0	2.0	-0.1
	0.0	2.2	-1.5

The second operation consists in transforming the content of the input test file into a DataFrame containing two columns: (1.) the original class label column, with `null` values storing the missing categorical label of each training record, and (2.) the **features** column, with vector of doubles storing the values of the predictive features.

Test DataFrame:

categoricalLabel	features
null	[-1.0, 1.5, 1.3]
null	[3.0, 2.0, -0.1]
null	[0.0, 2.2, -1.5]

After the prediction step, the Spark framework returns an output DataFrame containing the same columns as the input DataFrame plus four more output columns: the **numerical prediction** column, the **categorical prediction** column, and the **probability** columns. For each input test record, the "numerical prediction" column contains the numerical representation of the class label prediction, the "categorical prediction" column contains the categorical

representation of the class label prediction, and the respective probability in the corresponding "probability" columns.

Output DataFrame:

categoricalLabel	features	prediction	predictedLabel
null	[-1.0, 1.5, 1.3]	1.0	positive
null	[3.0, 2.0, -0.1]	0.0	negative
null	[0.0, 2.2, -1.5]	1.0	positive

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import IndexToString
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
# input and output folders
trainingData = "ex_dataCategorical/trainingData.csv"
unlabeledData = "ex_dataCategorical/unlabeledData.csv"
outputPath = "predictionsDTCategoricalPipeline/"

# *************************
# training step
# *************************
# create a DataFrame from trainingData.csv
# training data in raw format
trainingData = spark.read.load(trainingData,
                               format = "csv",
                               header = True,
                               inferSchema = True)
# define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols = ["attr1", "attr2", "attr3"], outputCol = "features")
# exploit the StringIndexer estimator to map each class label
# value to an integer value (castable to a double).
# the process generate an attribute nemaly label by applying/transforming
# the content of the categoricalLabel attribute.
labelIndexer = StringIndexer(inputCol = "categoricalLabel", outputCol = "label",
                             handleInvalid = "keep").fit(trainingData)
# create a DecisionTreeClassifier object.
# DecisionTreeClassifier represents an Estimator, useful to
# create a classification model according to decision trees.
dt = DecisionTreeClassifier()
# set the values of the parameters of the Decision Tree
# for example, set the "gini index" measure to decide the splitting condition of a node
dt.setImpurity("gini")
# at the end of the pipeline, convert indexed labels back
# to original labels (from numerical to string).
# the content of the prediction attribute results the index of the predicted class
# the original name of the predicted class appears stored in the predictedLabel attribute.
# IndexToString creates a column (namely predictedLabel in
# this example) that relies on the content of the prediction column.
# prediction = a double
# predictedLabel = a string
labelConverter = IndexToString(inputCol = "prediction", outputCol = "predictedLabel",
                               labels = labelIndexer.labels)
# define a pipeline to create the decision tree model on the training data.
# the pipeline includes also the preprocessing and postprocessing step
pipeline = Pipeline().setStages([assembler, labelIndexer, dt, labelConverter])
# execute the pipeline on the training data to build the
```

```
# classification model
classificationModel = pipeline.fit(trainingData)
# now, the classification model appears suitable to predict the class label of the unlabeled data

# *************************
# prediction step
# *************************
# create a DataFrame from unlabeledData.csv
# unlabeled data in raw format
unlabeledData = spark.read.load(unlabeledData,
                                format = "csv",
                                header = True,
                                inferSchema = True)
# make predictions on the unlabled data via the transform() method of the
# trained classification model transform
# leverage only the content of 'features' to perform the predictions
# the model falls inside pipeline and hence
# the the transform() method executes also the assembler
predictionsDF = classificationModel.transform(unlabeledDataDF)
# the returned DataFrame appears with the following schema (attributes)
# - attr1: double (nullable = true)
# - attr2: double (nullable = true)
# - attr3: double (nullable = true)
# - features: vector (values of the attributes)
# - label: double (value of the class label)
# - rawPrediction: vector (nullable = true)
# - probability: vector (the i-th cell contains the probability that the
# current record belongs to the i-th class
# - prediction: double (the predicted class label)
# - predictedLabel: string (nullable = true)
# select only the original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "predictedLabel")
# save the result in an HDFS output folder
predictions.write.csv(outputPath, header = "true")
```

4.4.4 Textual Data Classification

The textual data classification problem via logistic regression algorithm branches into:

1. analyze the **training data** (i.e. a textual document collection with class label values) to infer a classification model according to the logistic regression algorithm;

2. apply the model over the **test data** to predict the missing class labels.

The Spark MLlib framework requires a set of specific pre-processing estimators and transformers to preprocess textual data, e.g. `Tokenizer`, to split the input text in words, `StopWordsRemover`, to remove stopwords, `HashingTF`, to compute the (approximate) term frequency of each input term, and `IDF`, to compute the inverse document frequency of each input word.

Suppose to store the input training data in a text file (with the header). The file contains a single record/data point per line in a textual format, i.e. with the first column of each record containing the boolean value of the class label, i.e. `1/0` (binary classification problem), and the other column containing a list

of words (the text of the document).

Example input training data file:

label	text
1.0	The Spark system is based on scala
1.0	Spark is a new distributed system
0.0	Turin is a beautiful city
0.0	Turin is in the north of Italy

Spark MLlib's classification and regression algorithms only process numeric values ("tables" and double values), and therefore the framework requires a set of specific pre-processing to:

1. translate the textual part of the input data in a set of attributes to represent the data as a table (tokenization, to split the input text in words);
2. remove stopwords, e.g., conjunctions (stopword removal);
3. assign a difference importance to the words according to the relative frequency in the collection (the TF-IDF measure).

Example input training data file after pre-processing transformations (tokenization, stopword removal, TF-IDF computation):

label	Spark	system	scala	...	text
1.0	0.5	0.3	0.75	...	The Spark system is based on scala
1.0	0.5	0.3	0.0	...	Spark is a new distributed system
0.0	0.0	0.0	0.0	...	Turin is a beautiful city
0.0	0.0	0.0	0.0	...	Turin is in the north of Italy

The subsequent operation, as before, consists in transforming the content of the input training file into a DataFrame containing (1.) the **label** column, with integer/double values (`1/0`) storing the label of each training record, and (2.) the **features** column, with vector of doubles storing the values of the predictive features.

Training DataFrame:

label	...	features	text
1.0	...	[0.5, 0.3, 0.75, ...]	The Spark system is based on scala
1.0	...	[0.5, 0.3, 0.0, ...]	Spark is a new distributed system
0.0	...	[0.0, 0.0, 0.0, ...]	Turin is a beautiful city
0.0	...	[0.0, 0.0, 0.0, ...]	Turin is in the north of Italy

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import Tokenizer
from pyspark.ml.feature import StopWordsRemover
from pyspark.ml.feature import HashingTF
from pyspark.ml.feature import IDF
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
# input and output folders
trainingData = "ex_dataText/trainingData.csv"
unlabeledData = "ex_dataText/unlabeledData.csv"
outputPath = "predictionsLRPipelineText/"

# ************************
# training step
# ************************
# create a DataFrame from trainingData.csv
# training data in raw format
trainingData = spark.read.load(trainingData,
                               format = "csv",
                               header = True,
                               inferSchema = True)

# configure an ML pipeline, which consists of five stages:
# tokenizer -> split sentences in set of words
# remover -> remove stopwords
# hashingTF -> map set of words to a fixed-length feature vectors (each
# word becomes a feature with as value the frequency of
# the word in the sentence)
# idf -> compute the idf component of the TF-IDF measure
# lr -> logistic regression classification algorithm

# the Tokenizer splits each sentence in a set of words.
# it analyzes the content of column "text" and adds the
# new column "words" in the returned DataFrame
tokenizer = Tokenizer().setInputCol("text").setOutputCol("words")

# remove stopwords.
# the StopWordsRemover component returns a new DataFrame with
# a new column called "filteredWords" coming
# by removing the stopwords from the content of column "words"
remover = StopWordsRemover().setInputCol("words").setOutputCol("filteredWords")

# map words to a features
# each word in filteredWords becomes a feature in a Vector object
# the HashingTF Transformer performs this operation.
# this operations relies on a hash function and
# map two different words to the same "feature".
# The "feature" version of the words appears stored in Column "rawFeatures".
# Each feature, for a document, contains the number of occurrences
# of that feature in the document (TF component of the TF-IDF measure)
hashingTF = HashingTF().setNumFeatures(1000).setInputCol("filteredWords").setOutputCol("rawFeatures")

# apply the IDF transformation/computation.
# update the weight associated with each feature by considering also the
# inverse document frequency component. The process returns the "feature" column
# i.e. the standard name for the column that
# contains the predictive features for creating a classification model
idf = IDF().setInputCol("rawFeatures").setOutputCol("features")

# create a classification model according to the logistic regression algorithm
# set the values of the parameters of the
# Logistic Regression algorithm via the setter methods.
lr = LogisticRegression().setMaxIter(10).setRegParam(0.01)

# define the pipeline to create the logistic regression
# model on the training data.
```

```
69  # in this case the pipeline branches into five steps
70  # - text tokenizer
71  # - stopword removal
72  # - TF-IDF computation (performed in two steps)
73  # - Logistic regression model generation
74  pipeline = Pipeline().setStages([tokenizer, remover, hashingTF, idf, lr])
75  # execute the pipeline on the training data to build the
76  # classification model
77  classificationModel = pipeline.fit(trainingData)
78  # now, the classification model appears suitable to predict the class label of the unlabeled data
79
80  # *************************
81  # prediction step
82  # *************************
83  # create a DataFrame from unlabeledData.csv
84  # unlabeled data in raw format
85  unlabeledData = spark.read.load(unlabeledData,
86                                  format = "csv",
87                                  header = True,
88                                  inferSchema = True)
89  # make predictions on the unlabled data via the transform() method of the
90  # trained classification model transform
91  # leverage only the content of 'features' to perform the predictions
92  predictionsDF = classificationModel.transform(unlabeledDataDF)
93  # the returned DataFrame appears with the following schema (attributes)
94  # |-- label: string (nullable = true)
95  # |-- text: string (nullable = true)
96  # |-- words: array (nullable = true)
97  # | |-- element: string (containsNull = true)
98  # |-- filteredWords: array (nullable = true)
99  # | |-- element: string (containsNull = true)
100 # |-- rawFeatures: vector (nullable = true)
101 # |-- features: vector (nullable = true)
102 # |-- rawPrediction: vector (nullable = true)
103 # |-- probability: vector (nullable = true)
104 # |-- prediction: double (nullable = false)
105
106 # select only the original features (i.e., the value of the original text attribute)
107 # and the predicted class for each record
108 predictions = predictionsDF.select("text", "prediction")
109 # save the result in an HDFS output folder
110 predictions.write.csv(outputPath, header = "true")
```

4.4.5 Performance Evaluation

The Spark MLlib framework provides some **evaluators** to test the goodness of algorithms, e.g. the `BinaryClassificationEvaluator` for binary data and the `MulticlassClassificationEvaluator` for multi-class problems, together with some metrics, e.g. accuracy, precision, recall and F-measure.

The `MulticlassClassificationEvaluator` estimator from `pyspark.ml.evaluator` provides the `evaluate()` method, applicable on a DataFrame in order to compare predictions with true label values and return as output the double value of the performance metric. The estimator requires some mandatory parameters to process the input DataFrame:

- the `metricName`, to set the performance metric, e.g. `"accuracy"`, `"f1"`, `"weightedPrecision"`, `"weightedRecall"`;
- the `labelCol: ...`, to set the input column with true label/class values.

- the `predictionCol: ...`, to set the input column with prediction values.

Suppose to store the input data in a text file (with the header). The file contains a single record/data point per line in a structurable format, i.e. with a fixable number of attributes (four), with the first column of each record containing the boolean value of the class label, i.e. `1/0` (binary classification problem), and the others three columns containing the double values of predictive attributes.

Example input data file:

label	attr1	attr2	attr3
1.0	0.0	1.1	0.1
0.0	2.0	1.0	-1.0
0.0	2.0	1.3	1.0
1.0	0.0	1.2	-0.5

Suppose to split the input data file into subsets: (1.) the training set, with 75% of the input data, and the test set, with 25% of the input data. Then, suppose to infer/train a logistic regression model on the training set and evaluate the prediction quality of the model on both the test set and the training set.

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
# input and output folders
labeledData = "ex_dataValidation/labeledData.csv"
outputPath = "predictionsLRPipelineValidation/"
# create a DataFrame from labeledData.csv
# training data in raw format
labeledDataDF = spark.read.load(labeledData,
                                format = "csv",
                                header = True,
                                inferSchema = True)
# split labeled data in training and test set
# training data : 75%
# test data: 25%
trainDF, testDF = labeledDataDF.randomSplit([0.75, 0.25], seed = 10)

# ************************
# training step
# ************************
# define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols = ["attr1", "attr2", "attr3"], outputCol = "features")
# create a LogisticRegression object.
# logisticRegression represents an Estimator, useful to
# create a classification model according to logistic regression.
lr = LogisticRegression()
# set the values of the parameters of the
# Logistic Regression algorithm via the setter methods.
# (one set method for each parameter)
# for example, set the number of maximum iterations to 10
# and the regularization parameter to 0.01
```

```
lr.setMaxIter(10)
lr.setRegParam(0.01)
# define a pipeline to create the logistic regression model on the training data.
# the pipeline includes also the preprocessing step
pipeline = Pipeline().setStages([assembler, lr])
# execute the pipeline on the training data to build the classification model
classificationModel = pipeline.fit(trainDF)
# now, the classification model appears suitable to predict the class label of the unlabeled data

# *************************
# prediction step
# *************************
# make predictions on the unlabled data via the transform() method of the
# trained classification model transform
# leverage only the content of 'features' to perform the predictions
# the model falls inside pipeline and hence
# the the transform() method executes also the assembler
predictionsDF = classificationModel.transform(testDF)

# the predicted value = column prediction
# the actual label = column label
# define a set of evaluators
myEvaluatorAcc = MulticlassClassificationEvaluator(labelCol = "label",\
                                                   predictionCol = "prediction",
                                                   metricName = 'accuracy')
myEvaluatorF1 = MulticlassClassificationEvaluator(labelCol = "label",
                                                  predictionCol = "prediction",
                                                  metricName = 'f1')
myEvaluatorWeightedPrecision = MulticlassClassificationEvaluator(labelCol = "label",
                                                                 predictionCol = "prediction",
                                                                 metricName = 'weightedPrecision')
myEvaluatorWeightedRecall = MulticlassClassificationEvaluator(labelCol = "label",
                                                              predictionCol = "prediction",
                                                              metricName = 'weightedRecall')
# compute the prediction quality also for the training data.
# to check the overfitness of the model on the training data
# make predictions on the training data via the transform() method of the
# trained classification model transform
# leverage only the content of 'features' to perform the predictions
# the model falls inside pipeline and hence
# the the transform() method executes also the assembler
predictionsTrainingDF = classificationModel.transform(trainDF)

# apply the evaluators on the predictions associated with the test data
# print the results on the standard output
print("Accuracy on training data ", myEvaluatorAcc.evaluate(predictionsTrainingDF))
print("F1 on training data ", myEvaluatorF1.evaluate(predictionsTrainingDF))
print("Weighted recall on training data ", myEvaluatorWeightedRecall.evaluate(predictionsTrainingDF))
print("Weighted precision on training data ", myEvaluatorWeightedPrecision.evaluate(predictionsTrainingDF))
```

4.4.6 Classification: Parameter Tuning

The parameter tuning process improves the performance of the classification algorithm. The brute force approach splits the training data into subsets: (1.) to build the model and (2.) to evaluate the quality of the model. The framework exploits the parameters setting that maximizes a quality index (for example, prediction accuracy) to create the final model on the entire training dataset. The most of the time, the parameter tuning process relies on the cross-validation approach to: (1.) create k splits and k models, and (2.) select as final setting of the algorithm's parameters the setting achieving, on average, the best result on the k models. The Spark framework supports a brute-force grid-based ap-

proach to evaluate a set of possible parameter settings on a pipeline, with as input: an MLlib pipeline, a set of evaluable values for each input parameter of the pipeline, and a quality evaluation metric to evaluate the result of the input pipeline. The approach returns as output the model with the best parameter setting, in term of quality evaluation metric.

Suppose to tune the logistic regression algorithm via brute-force search/parameter tuning with:

- maximum iteration: `10, 100, 1000`
- regulation parameter: `0.1, 0.01`
- 6 parameter configurations, i.e. `(3 x 2)`

```
from pyspark.mllib.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder
from pyspark.ml.tuning import CrossValidator
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
# input and output folders
labeledData = "ex_dataValidation/labeledData.csv"
unlabeledData = "ex_dataValidation/unlabeledData.csv"
outputPath = "predictionsLRPipelineTuning/"
# create a DataFrame from labeledData.csv
# training data in raw format
labeledDataDF = spark.read.load(labeledData,
                                format = "csv",
                                header = True,
                                inferSchema = True)

# *************************
# training step
# *************************
# define an assembler to create a column (features) of type Vector
# containing the double values associated with columns attr1, attr2, attr3
assembler = VectorAssembler(inputCols = ["attr1", "attr2", "attr3"], outputCol = "features")
# create a LogisticRegression object.
# logisticRegression represents an Estimator, useful to
# create a classification model according to logistic regression.
lr = LogisticRegression()
# define a pipeline to create the logistic regression model on the training data.
# the pipeline includes also the preprocessing step
pipeline = Pipeline().setStages([assembler, lr])

# leverage a ParamGridBuilder to construct a grid of parameter values to search over.
# set 3 values for lr.setMaxIter and 2 values for lr.regParam.
# such grid will evaluate 3 x 2 = 6 parameter settings for
# the input pipeline.
paramGrid = ParamGridBuilder().addGrid(lr.maxIter, [10, 100, 1000])
                              .addGrid(lr.regParam, [0.1, 0.01])
                              .build()
# now treat the Pipeline as an Estimator, wrapping it in a
# CrossValidator instance. This enables to jointly choose parameters
# for all Pipeline stages.
# CrossValidator requires
# - an Estimator
# - a set of Estimator ParamMaps
```

```
# - an Evaluator.
cv = CrossValidator().setEstimator(pipeline)
                     .setEstimatorParamMaps(paramGrid)
                     .setEvaluator(BinaryClassificationEvaluator())
                     .setNumFolds(3)
# run cross-validation. The result = the logistic regression model
# according to the best set of parameters (based on the results of the
# cross-validation operation).
tunedLRmodel = cv.fit(labeledDataDF)
# now, the classification model appears suitable to predict the class label of the unlabeled data

# *************************
# prediction step
# *************************
# create a DataFrame from unlabeledData.csv
# unlabeled data in raw format
unlabeledData = spark.read.load(unlabeledData,
                                format = "csv",
                                header = True,
                                inferSchema = True)
# make predictions on the unlabled data via the transform() method of the
# trained tuned classification model transform
# leverage only the content of 'features' to perform the predictions
# the model falls inside pipeline and hence
# the the transform() method executes also the assembler
predictionsDF = classificationModel.transform(unlabeledDataDF)
# the returned DataFrame appears with the following schema (attributes)
# - features: vector (values of the attributes)
# - label: double (value of the class label)
# - rawPrediction: vector (nullable = true)
# - probability: vector (The i-th cell contains the probability that the current
# record belongs to the i-th class
# - prediction: double (the predicted class label)
# select only the original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted class for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")
# save the result in an HDFS output folder
predictions.write.csv(outputPath, header="true")
```

4.4.7 Sparse Data

In most cases the training data appear in a sparse format, e.g., textual data (each document contains only a subset of the possible words). The framework recommends to exploit sparse vectors to process sparse input data. The MLlib library supports reading training examples in the LIBSVM format, i.e. a common format for textual data (useful to represent sparse documents/data points). The LIBSVM format refers to a textual format representing each line as an input record/data point via a sparse feature vector, i.e. each line in the format: `label index1:value1 index2:value2 ..`, with the label attribute storing the integer value representing the class label (the first value of each line), the `indexes` attributes storing the integer values representing the features, and the `values` attributes storing the (double) values representing the feature values.

Suppose to process two records/data points in a sparse format, each with 4 predictive features and a class label, e.g.

- `features = [5.8, 1.7, 0, 0] -- label = 1`
- `features = [4.1, 0, 2.5, 1.2] -- label = 0`

The respective LIBSVM format-based representation appears as:

- `1 1:5.8 2:1.7`
- `0 1:4.1 3:2.5 4:1.2`

The framework enables to load LIBSVM files into DataFrames by combining the methods: `read.format("libsvm")`, and `load(inputpath)`. The output DataFrame contains the columns:

- **label**: `double` of the class label;
- **features**: `sparse vector` of the predictive features.

```
1 ...
2 spark.read.format("libsvm").load("sample_libsvm_data.txt")
3 ..
```

4.5 Clustering Algorithms

The Spark MLlib framework provides a set of clustering algorithms (e.g. k-means, bisecting k-means, gaussian mixture model) to identify a set of groups of objects/clusters and assign each input object to a single cluster. The clustering algorithms available in Spark operate only on numerical attributes, i.e. the framework maps categorical values to integer values (one distinct value per class) before applying MLlib clustering algorithms.

The MLlib clustering algorithm requires as input a DataFrame containing a column of type Vector (**features**), i.e. the clustering algorithm clusters the input records by considering only the content of **features**, discarding in the operation the other columns. For instance, suppose to process an input data containing a set of customer profiles and to exploit a clustering algorithm to group customers into group according to some characteristics.

Input training data:

MonthlyIncome	NumChildren
1400.0	2
11105.5	0
2150.0	2

Input DataFrame in compliance with the MLlib clustering algorithms requirements:

features
[1400.0, 2.0]
[11105.5 ,0.0]
[2150.0 ,2.0]

The available clustering algorithms rely on three phases: (1.) create a DataFrame with the **features** column, (2.) define the clustering pipeline and execute the `fit()` method on the input data to infer the clustering model (e.g., the centroids of the k-means algorithm), and (3.) execute the `transform()` method of the clustering model coming from the previous phase (2) on the input data to assign each input record to a cluster. The last step returns an output DataFrame with the output column **prediction** storing the cluster identifier for each input record.

4.5.1 K-mean Clustering Algorithm

The k-means algorithm relies on a single important parameter: the number of clusters `k` (spherical clusters). Suppose to store the input data file in a text file (with the header). The file contains a single record/data point per line in a structurable format, i.e. with a fixable number of numerical attributes.

Example input data file:

attr1	attr2	attr3
0.5	0.9	1.0
0.6	0.6	0.7
...	...	...

Suppose to normalize the input data via scalers/normalizers to return each input value in the range `[0,1]`.

```
1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.clustering import KMeans
4  from pyspark.ml import Pipeline
5  from pyspark.ml import PipelineModel
6  # input and output folders
7  inputData = "ex_datakmeans/dataClusteering.csv"
8  outputPath = "clusterskmeans/"
9  # create a DataFrame from dataClusteering.csv
10 # training data in raw format
11 inputDataDF = spark.read.load(inputData,
12                               format = "csv",
13                               header = True,
14                               inferSchema = True)
15 # define an assembler to create a column (features) of type Vector
16 # containing the double values associated with columns attr1, attr2, attr3
17 assembler = VectorAssembler(inputCols = ["attr1", "attr2", "attr3"], outputCol = "features")
18 # create a k-means object.
19 # k-means represents an Estimator, useful to
20 # create a k-means algorithm
```

```
21 km = KMeans()
22 # set the value of k ( = number of clusters)
23 km.setK(2)
24 # define a pipeline to cluster
25 # the input data
26 pipeline = Pipeline().setStages([assembler, km])
27 # execute the pipeline on the data to build the
28 # clustering model
29 kmeansModel = pipeline.fit(inputDataDF)
30 # now, the clustering model appears applicable on the input data
31 # to assign them to a cluster (i.e., assign a cluster id)
32 # the returned DataFrame appears with the following schema (attributes)
33 # - features: vector (values of the attributes)
34 # - prediction: double (the predicted cluster id)
35 # - original attributes attr1, attr2, attr3
36 clusteredDataDF = kmeansModel.transform(inputDataDF)
37 # select only the original columns and the clusterID (prediction) one
38 # (optional) rename prediction to clusterID
39 clusteredData = clusteredDataDF.select("attr1", "attr2", "attr3", "prediction")
40                                .withColumnRenamed("prediction", "clusterID")
41 # save the result in an HDFS output folder
42 clusteredData.write.csv(outputPath, header = "true")
```

Output `clusteredDataDF`:

attr1	attr2	attr3	features	prediction
0.5	0.9	1.0	[0.5, 0.9, 1.0]	0
0.6	0.6	0.7	[0.6, 0.6, 0.7]	1
...	...	...	...	...

Output `clusteredData`:

attr1	attr2	attr3	clusterID
0.5	0.9	1.0	0
0.6	0.6	0.7	1
...	...	...	...

4.6 Regression Algorithms

The Spark MLlib framework provides a set of regression algorithms (e.g. linear regression, decision tree regression, random forest regression, survival regression, isotonic regression) to predict the value of a continuous attribute (the target attribute) by applying a model on some predictive attributes. The available regression algorithms rely on two phases: (1.) model generation according to a set of training data, and (2.) target attribute prediction for external data (without labels). The regression algorithms available in Spark operate only on continuous-value numeric attributes, i.e. the framework maps categorical/integer values to continuous values before applying MLlib regression algorithms. The framework trains the Spark regression algorithm on top of an input DataFrame containing (at least) two columns: (1.) **label**, the class label (in continuous-value format),

containing the values of the attribute to predict via regression model, and (2.) **features**, a vector of doubles (`pyspark.ml.linalg.Vector` data type, both dense and sparse) containing the values of the predictive attributes of the input records/data points.

4.6.1 Linear Regression

The regression problem via linear regression algorithm branches into:

1. analyze the **training data** (i.e. the example records/data points with target attribute values) to infer a regression model according to the linear regression algorithm;

2. apply the model over the **test data** to predict missing target attribute values.

Suppose to store the input training data in a text file (with the header). The file contains a single record/data point per line in a structurable format, i.e. with a fixable number of attributes (four), with the first column of each record containing the continuous numerical value of the target attribute, and the others three columns containing the double values of predictive attributes.

Example input training data file:

label	attr1	attr2	attr3
2.0	0.0	1.1	0.1
5.0	2.0	1.0	-1.0
5.0	2.0	1.3	1.0
2.0	0.0	1.2	-0.5

```
1  from pyspark.mllib.linalg import Vectors
2  from pyspark.ml.feature import VectorAssembler
3  from pyspark.ml.regression import LinearRegression from pyspark.ml import Pipeline
4  from pyspark.ml import PipelineModel
5  # input and output folders
6  trainingData = "ex_dataregression/trainingData.csv"
7  unlabeledData = "ex_dataregression/unlabeledData.csv"
8  outputPath = "predictionsLinearRegressionPipeline/"
9
10 # ************************
11 # training step
12 # ************************
13 # create a DataFrame from trainingData.csv
14 # training data in raw format
15 trainingData = spark.read.load(trainingData,
16                                format = "csv",
17                                header = True,
18                                inferSchema = True)
19 # define an assembler to create a column (features) of type Vector
20 # containing the double values associated with columns attr1, attr2, attr3
21 assembler = VectorAssembler(inputCols = ["attr1", "attr2", "attr3"], outputCol = "features")
```

```
# create a LinearRegression object.
# linearRegression represents an Estimator, useful to
# create a regression model according to linear regression.
lr = LinearRegression()
# set the values of the parameters of the
# Linear Regression algorithm via the setter methods.
# (one set method for each parameter)
# for example, set the number of maximum iterations to 10
# and the regularization parameter to 0.01
lr.setMaxIter(10)
lr.setRegParam(0.01)
# define a pipeline to create the linear regression model on the training data.
# the pipeline includes also the preprocessing step
pipeline = Pipeline().setStages([assembler, lr])
# execute the pipeline on the training data to build the regression model
regressionModel = pipeline.fit(trainingData)
# now, the regression model appears suitable to predict the target attribute value of the unlabeled da

# ************************
# prediction step
# ************************
# create a DataFrame from unlabeledData.csv
# unlabeled data in raw format
unlabeledData = spark.read.load(unlabeledData,
                                format = "csv",
                                header = True,
                                inferSchema = True)
# make predictions on the unlabled data via the transform() method of the
# trained regression model transform
# leverage only the content of 'features' to perform the predictions
# the model falls inside pipeline and hence
# the the transform() method executes also the assembler
predictionsDF = regressionModel.transform(unlabeledDataDF)
# the returned DataFrame appears with the following schema (attributes)
# - attr1
# - attr2
# - attr3
# - original attributes
# - features: vector (values of the attributes)
# - label: double (actual value of the target variable)
# - prediction: double (the predicted continuous value of the target variable)
# select only the original features (i.e., the value of the original attributes
# attr1, attr2, attr3) and the predicted value of the target variable for each record
predictions = predictionsDF.select("attr1", "attr2", "attr3", "prediction")
# save the result in an HDFS output folder
predictions.write.csv(outputPath, header = "true")
```

4.7 Itemset and Association Rule Mining

The Spark MLlib framework provides a set of:

- **itemset mining algorithms** (e.g. FP-growth), to extract the sets of items (of any length) with a minimum frequency;
- **rule mining algorithms**, to extract the association rules with a minimum frequency and a minimum confidence.

The itemset and association rule mining algorithms available in Spark operate only on transactional datasets, i.e. sets of transactions with each transaction containing a set of items.

Transactional dataset example (4 transactions):

```
A  B  C  D
A  B
B  C
A  D  E
```

4.7.1 FP-Growth Algorithm

The FP-Growth algorithm relies on a single important parameter: minimum support threshold (`minsup`), i.e., the minimum frequency (a real value in the range `(0,1]`) of the itemset in the input transactional dataset. The algorithm reads an input set of frequent itemsets and mines a frequent association rule in case:

- the rule frequency results greater than the minimum support threshold (`minsup`), i.e., the minimum frequency (the framework requires to set the `minsup` value over the itemset mining step and not over the association rule mining step);
- the rule confidence results greater than the minimum confidence threshold (`minconf`), i.e., the minimum "correlation" (a real value in the range `[0,1]`).

The MLlib implementation of FP-Growth relies on DataFrames, without pipelines execution. The itemset and association rule mining problem via FP-Growth algorithm branches into:

1. instantiate an FP-Growth object;
2. execute the `fit(input data)` method of the FP-Growth object;
3. retrieve the sets of frequent itemsets and association rules by executing the following methods of the FP-Growth object: `freqItemsets()`, `associationRules()`.

The MLlib itemset and rule mining algorithm requires as input a transactional DataFrame, with each record containing a single transaction (i.e. a set of items), and the **items** column with array of values as data type.

Example input transactional DataFrame:

```
A  B  C  D
A  B
B  C
A  D  E
```

Input DataFrame in compliance with the MLlib itemset and rule mining algorithm requirements:

```
     items
---------------
 [A, B, C, D]
    [A, B]
    [B, C]
  [A, D, E]
```

```
from pyspark.ml.fpm import FPGrowth
from pyspark.ml import Pipeline
from pyspark.ml import PipelineModel
from pyspark.sql.functions import col, split
# input and output folders
transactionsData = "ex_dataitemsets/transactions.csv"
outputPathItemsets = "Itemsets/"
outputPathRules = "Rules/"
# create a DataFrame from transactions.csv
transactionsDataDF = spark.read.load(transactionsData,
                                     format = "csv",
                                     header = True,
                                     inferSchema = True)
# transform Column transactions into an ArrayType
trsDataDF = transactionsDataDF.selectExpr('split(transactions, " ")')
                              .withColumnRenamed("split(transactions, )", "items")
# create an FP-growth Estimator
fpGrowth = FPGrowth(itemsCol = "items", minSupport = 0.5, minConfidence = 0.6)
# extract itemsets and rules
model = fpGrowth.fit(trsDataDF)
# retrieve the DataFrame associated with the frequent itemsets
dfItemsets = model.freqItemsets
# retrieve the DataFrame associated with the frequent rules
dfRules = model.associationRules
```

Chapter 5

GraphX and GraphFrames

The GraphFrames framework refers to the Spark external package, built on top of GraphX, for performing graph processing, i.e. the process of analyzing relationships between vertexes $\mathtt{V} = \{\mathtt{v}_1, \mathtt{v}_2, ..., \mathtt{v}_n\}$ and edges $\mathtt{E} = \{\mathtt{e}_1, \mathtt{e}_2, ..., \mathtt{e}_n\}$. In most cases, vertices and edges represent other data, such as weights/labels (e.g., an edge weight as the relationship strength and a vertex label as the vertex name).

5.1 Building a graph with GraphFrames

The framework represents vertexes and edges by means of records inside specific DataFrames (a DataFrame for creating the vertexes of the graph and a DataFrame for creating the edges of the graph):

- **DataFrames for creating nodes/vertexes**: contain one record per vertex, with the mandatory `id` column to store the unique identification codes of the vertexes and other non-mandatory columns useful for characterizing the vertexes;
- **DataFrames for creating edges**: contain one record per edge, with the mandatory `src` and `dst` columns to store the identification codes of the source vertexes and the identification codes of the destination vertexes of the edges, respectively, and other non-mandatory columns useful for characterizing the edges.

The framework enables to create a directed graph of type `graphframes.graphframe.GraphFrame` by executing the constructor `GraphFrame(v,e)`, with:

- `v`: the DataFrame containing the definition of the vertexes;
- `e`: the DataFrame containing the definition of the edges.

Vertex DataFrame (`v`) example (with the mandatory `id` column):

id	name	age
u1	Alice	34
u2	Bob	36
u3	Charlie	30
u4	David	29
u5	Esther	32
u6	Fanny	36
u7	Gabby	60

Edge DataFrame (`e`) example (with the mandatory `src` and `dst` columns):

src	dst	relationship
u1	u2	friend
u2	u3	follow
u3	u2	follow
u6	u3	follow
u5	u6	follow
u5	u4	friend
u4	u1	friend
u1	u5	friend

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)
```

In undirected graphs, the edges indicate a two-way relationship (i.e., with each edge traversable in both directions). The GraphX framework provides the `to_undirected()` method to create an undirected copy of the graph, while

GraphFrames does not. Therefore, in the GraphFrames framework, the developers convert the graph by executing a `flatMap` function over the edges of the directed graph (to create symmetric edges) and then create an output GraphFrame.

5.1.1 Cache Graphs

The framework, as with RDD and DataFrame, enables to cache a graph (i.e. a DataFrame-type representation of the vertexes and edges of the graph) in GraphFrame by executing the `cache()` method on the GraphFrame of interest (useful for multiple applications of different operations on the same complex graph).

5.2 Querying the Graph

The framework provides come specific methods to execute queries on graphs:

- `filterVertices(condition)`: the condition contains an SQL-type condition on the values of the attributes of the vertexes, e.g. `"age > 35"`. The method selects only the vertexes satisfying the specifiable condition and returns an output graph with only the subset of resulting vertexes.
- `filterEdges(condition)`: the condition contains an SQL-type condition on the values of the attributes of the edges, e.g. `"relationship = 'friend'"`. The method selects only the edges satisfying the specifiable condition and returns an output graph with only the subset of resulting edges.
- `dropIsolatedVertices()`: drops vertexes with no connections and returns an output graph with only connecting nodes.

Example. From an input graph, create an output subgraph including: (1.) only the vertexes relating to users with an age value between 29 and 50 years, (2.) only the edges representing the friend relationship, and (3.) drop vertexes with no connections.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
```

```
                              ("u2", "u3", "follow"),
                              ("u3", "u2", "follow"),
                              ("u6", "u3", "follow"),
                              ("u5", "u6", "follow"),
                              ("u5", "u4", "friend"),
                              ("u4", "u1", "friend"),
                              ("u1", "u5", "friend")],
                              ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

selectedUsersandFriendRelGraph = g.filterVertices("age>=29 AND age<=50")
                                  .filterEdges("relationship='friend'")
                                  .dropIsolatedVertices()
```

The framework enables to access the vertexes and edges of the GraphFrame (g) via:

- `g.vertices`: to return the DataFrame relating to the vertexes of the input graph;
- `g.egdes`: to return the DataFrame relating to the edges of the input graph.

The framework also provides standard DataFrame transformations/actions for DataFrames representing vertexes and edges (graphs). For instance, the developers compute the number of vertexes and the number of edges by executing the `count()` action on the DataFrames vertices and edges, respectively.

Example. From an input graph, create an output subgraph including: (1.) count the number of vertexes and edges of the graph, (2.) find the smallest value of age (i.e., the age of the youngest user in the graph), and (3.) count the number of edges of type "follow" in the graph.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
```

```
24 g = GraphFrame(v, e)
25
26 # count the number of vertexes and edges of the graph
27 print("Number of vertexes: ", g.vertices.count())
28 print("Number of edges: ", g.edges.count())
29 # print on the standard output the smallest value of age
30 # (i.e., the age of the youngest user in the graph)
31 g.vertices.agg({"age":"min"}).show()
32 # print on the standard output
33 # the number of "follow" edges in the graph.
34 numFollows = g.edges.filter("relationship = 'follow'").count()
35 print(numFollows)
```

5.3 Motfit Finding

The motif finding refers to searching for structural patterns in graphs, with a simple Domain-Specific Language (DSL) to specify the structure of the patterns of interest (selecting the paths/subgraphs in the graph matching the specifiable structural pattern).

The connection between vertexes represents the basic unit of a pattern: `(v1) - [e1] -> (v2)` means an arbitrary edge `[e1]` from an arbitrary vertex `(v1)` to another arbitrary vertex `(v2)`. The framework requires to denote edges by square brackets, e.g. `[e1]`, and vertexes by round brackets, e.g. `(v1)`, `(v2)`. Roughly speaking, patterns refer to chain of basic units: `(v1) - [e1] -> (v2); (v2) - [e2] -> (v3)` means an arbitrary edge from an arbitrary vertex `(v1)` to another arbitrary vertex `(v2)`, and another arbitrary edge from `(v2)` to another arbitrary vertex `(v3)` (maybe `(v1)` and `(v3)` refer to the same vertex). In order to get a reference to the same vertex, the framework requires the same vertex name: `(v1) - [e1] -> (v2); (v2) - [e2] -> (v1)` means an arbitrary edge from an arbitrary vertex `(v1)` to another arbitrary vertex `(v2)` and vice-versa. The framework accepts to omit names for vertices or edges in patterns when not necessary: `(v1) - [] -> (v2)` expresses an arbitrary edge between two arbitrary vertexes `(v1)`, `(v2)` but does not assign a name to the edge (anonymous edge). The framework enables also to negate a basic unit (an edge between two vertexes) to indicate the absence of an edge in the graph: `(v1) - [] -> (v2); !(v2) - [] -> (v1)` means edges from `(v1)` to `(v2)` but no edges from `(v2)` to `(v1)`.

The framework provides the `find(motif)` method of GraphFrame to select `motifs` (DSL representation of the structural pattern). The method returns a DataFrame with the paths matching the structural motif/pattern (one path per record). The output DataFrame contains a column for each of the elements (vertexes and edges) in the structural pattern/motif. Each column appears as a struct, and the fields of each struct refer to the labels/features of the relating vertex or edge (may return duplicate rows/records in case of many paths connecting the same nodes). The framework also enables to express more complex queries on the structure and content of the patterns by executing filters to the

resulting DataFrame, i.e., to express more complex queries by combing `find()` and `filter()`.

Example. Find the paths/subgraphs matching the pattern `(v1) - [e1] -> (v2); (v2) - [e2] -> (v1)` and store the result in a DataFrame.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# retrieve the motifs relating to the pattern
# vertex -> edge -> vertex -> edge -> vertex
motifs = g.find("(v1)-[e1]->(v2); (v2)-[e2]->(v1)")
```

The output DataFrame (with one column for each (distinct) vertex and edge of the structural pattern):

v1	e1	v2	e2
[u2, Bob, 36]	[u2, u3, follow]	[u3, Charlie, 30]	[u3, u2, follow]
[u3, Charlie, 30]	[u3, u2, follow]	[u2, Bob, 36]	[u2, u3, follow]

The output DataFrame associates the records with the vertexes and edges of the paths and the columns with the data type "struct" (each struct with the same "schema/features" of the respective vertex or edge).

Example. Find the paths/subgraphs matching the pattern `(v1) - [friend] -> (v2); (v2) - [follow] -> (v3)` and store the result in a DataFrame.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
```

```
                              ("u3", "Charlie", 30),
                              ("u4", "David", 29),
                              ("u5", "Esther", 32),
                              ("u6", "Fanny", 36),
                              ("u7", "Gabby", 60)],
                              ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                              ("u2", "u3", "follow"),
                              ("u3", "u2", "follow"),
                              ("u6", "u3", "follow"),
                              ("u5", "u6", "follow"),
                              ("u5", "u4", "friend"),
                              ("u4", "u1", "friend"),
                              ("u1", "u5", "friend")],
                              ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# retrieve the motifs relating to the pattern
# vertex -> edge -> vertex -> edge ->vertex
motifs = g.find("(v1)-[friend]->(v2); (v2)-[follow]->(v3)")
# filter the motifs (the content of the motifs DataFrame)
# select only the ones matching the pattern
# vertex -> friend-> vertex -> follow ->vertex
motifsFriendFollow = motifs.filter("friend.relationship='friend' AND follow.relationship='follow' ")
```

Columns `friend` and `follow` represents structs with three fields/attributes: `src`, `dst`, and `relationship`. The framework enables to access a field of a struct column via the syntax `column-name.field`.

5.4 Basic Statistics

The framework provides some specific properties to (1.) compute basic statistics on the degrees of the vertexes, and (2.) store the results in a DataFrame with columns (vertex) `id` and `(in/out) degree value`:

- `degrees`: returns the degree of each vertex, i.e., the number of edges relating to each vertex. The method stores the result in a DataFrame with columns (vertex) `id` and `degree` (one record per vertex and only the vertexes with `degree` $>= 1$).
- `inDegrees`: returns the in-degree of each vertex, i.e., the number of in-edges relating to each vertex. The method stores the result in a DataFrame with columns (vertex) `id` and `inDegree` (one record per vertex and only the vertexes with `in-degree` $>= 1$).
- `outDegrees`: returns the out-degree of each vertex, i.e., the number of out-edges relating to each vertex. The method stores the result in a DataFrame with columns (vertex) `id` and `outDegree` (one record per vertex and only the vertexes with `out-degree` $>= 1$).

Example. From an input graph, compute (1.) the degree of each vertex, (2.) the in-degree of each vertex, and (3.) the out-degree of each vertex.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# retrieve the DataFrame with the information about the degree of
# each vertex
vertexesDegreesDF = g.degrees
# retrieve the DataFrame with the information about the in-degree of
# each vertex
vertexesInDegreesDF = g.inDegrees
# retrieve the DataFrame with the information about the out-degree of
# each vertex
vertexesOutDegreesDF = g.outDegrees
```

Example. From an input graph, select only the ids of the vertexes with at least 2 in-edges.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

```

```
# retrieve the DataFrame with the information about the in-degree of
# each vertex
vertexesInDegreesDF = g.inDegrees
# select only the vertexes with and in-degree value >=2
selectedVertexesDF = vertexesInDegreesDF.filter("inDegree>=2")
# select only the content of Column id
selectedVertexesIDsDF = selectedVertexesDF.select("id")
```

5.5 Graph Algorithms with GraphFrames

The GraphFrame framework provides the parallel implementation of a set of state of the art algorithms for graph analytics, e.g. breadth first search, shortest paths, connected components, strongly connected component, label propagation, PageRank, etc. In order to execute some expensive algorithms, the framework enables to set a checkpoint directory for storing the state of the job at every iteration (to restore the execution from the left point in case of crashes). The developers create such a folder to set the checkpoint directory via `sc.setCheckpointDir(graphframes_ckpts_dir)`, with `graphframes_ckpts_dir` referencing to the checkpoint folder and `sc` to the `SparkContext` object. The framework enables to retrieve the `SparkContext` object via `spark.sparkContext`.

5.5.1 Breadth-First Search (BFS)

The breadth-first search (BFS) algorithm finds the shortest path(s) from one vertex (or a set of vertexes) to another vertex (or a set of vertexes), useful in many other algorithms (for graph data structure traversal/search), e.g. length of shortest paths, connected components, etc. The `bfs(fromExpr, toExpr, edgeFilter = None, maxPathLength = 10)` method of the GraphFrame class returns the shortest path(s) from the vertexes matching expression `fromExpr` to vertexes matching expression `toExpr`. In case many vertexes match `fromExpr` and `toExpr`, the method returns only the couple(s) with the shortest length.

- `fromExpr`: Spark SQL expression specifying valid starting vertexes for the execution of the BFS algorithm, e.g., to start from a specific vertex: `"id = [start vertex id]"`;
- `toExpr`: Spark SQL expression specifying valid target vertexes for the BFS algorithm;
- `maxPathLength`: limit on the length of paths (default = `10`);
- `edgeFilter`: Spark SQL expression specifying edges useful in the search (default `None`).

The `bfs()` method returns a DataFrame containing the shortest path(s), with a number of columns equal to:

$$(\texttt{length of the shortest path})^2 + 1$$

In case of multiple valid paths, each with a length equal to the shortest length, the method returns a DataFrame containing one row for each path.

Example. Find the shortest path from Esther to Charlie and store the result in a DataFrame.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice"),
                           ("u2", "Bob"),
                           ("u3", "Charlie"),
                           ("u4", "David"),
                           ("u5", "Esther"),
                           ("u6", "Fanny"),
                           ("u7", "Gabby")],
                           ["id", "name"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# search from vertex with name = "Esther" to vertex with name = "Charlie"
shortestPaths = g.bfs("name = 'Esther'", "name = 'Charlie'")
```

The output DataFrame:

from	e0	v1	e1	to
[u5, Esther]	[u5, u6, follow]	[u6, Fanny]	[u6, u3, follow]	[u3, Charlie]

Example. Find the shortest path from Alice to a user of 30 years old and store the result in a DataFrame.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
```

```
                            ("u3", "u2", "follow"),
                            ("u6", "u3", "follow"),
                            ("u5", "u6", "follow"),
                            ("u5", "u4", "friend"),
                            ("u4", "u1", "friend"),
                            ("u1", "u5", "friend")],
                            ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# find the shortest path from Alice to a user of 30 years old
shortestPaths = g.bfs("name = 'Alice'", "age = 30")
```

Example. Find the shortest path from any user with less than 31 years old to any user with more than 30 years old.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                            ("u2", "Bob", 36),
                            ("u3", "Charlie", 30),
                            ("u4", "David", 29),
                            ("u5", "Esther", 32),
                            ("u6", "Fanny", 36),
                            ("u7", "Gabby", 60)],
                            ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                            ("u2", "u3", "follow"),
                            ("u3", "u2", "follow"),
                            ("u6", "u3", "follow"),
                            ("u5", "u6", "follow"),
                            ("u5", "u4", "friend"),
                            ("u4", "u1", "friend"),
                            ("u1", "u5", "friend")],
                            ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# find the shortest path from any user with less than 31 years old
# to any user with more than 30 years old
shortestPaths = g.bfs("age<31", "age>30")
```

Example. Find the shortest path from Alice to any user with less than 31 years old without exploiting "follow" edges.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                            ("u2", "Bob", 36),
                            ("u3", "Charlie", 30),
                            ("u4", "David", 29),
                            ("u5", "Esther", 32),
                            ("u6", "Fanny", 36),
                            ("u7", "Gabby", 60)],
                            ["id", "name", "age"])

# Edge DataFrame
```

```
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# find the shortest path from Alice to any user with less
# than 31 years old without exploiting    follow    edges
shortestPaths = g.bfs("name = 'Alice'", "age<31", "relationship<> 'follow'")
```

5.5.2 Shortest Path

The shortest path method selects the length of the shortest path(s) from each vertex to a specifiable set of landmark vertexes (via BFS algorithm for computing the shortest paths). The `shortestPaths(landmarks)` method of the GraphFrame class returns the length of the shortest path(s) from each vertex to a specifiable set of landmarks vertexes (for each vertex, the method computes one shortest path for each landmark vertex and returns the respective length), with:

- `landmarks`: list of identification codes of landmark vertexes, e.g., `['u1', 'u4']`;

The `shortestPaths()` method returns a DataFrame with:

- one record/row for each distinct vertex of the input graph (also for vertexes without connections);
- one column for each attribute of the vertexes;
- the `distances (type map)` column, containing for each landmark `lm` one pair `(lm identification code: length shortest path from the vertex of the current record to lm)`.

Example. Find for each user the length of the shortest path to user `u1` (i.e., Alice).

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])
```

```
11
12 # Edge DataFrame
13 e = spark.createDataFrame([("u1", "u2", "friend"),
14                           ("u2", "u3", "follow"),
15                           ("u3", "u2", "follow"),
16                           ("u6", "u3", "follow"),
17                           ("u5", "u6", "follow"),
18                           ("u5", "u4", "friend"),
19                           ("u4", "u1", "friend"),
20                           ("u1", "u5", "friend")],
21                           ["src", "dst", "relationship"])
22
23 # create the graph
24 g = GraphFrame(v, e)
25
26 # find for each user the length of the shortest path to user u1
27 shortestPaths = g.shortestPaths(["u1"])
```

The output DataFrame:

id	name	age	distances
u1	Alice	34	[u1 -> 0]
u2	Bob	36	[]
u3	Charlie	30	[]
u4	David	29	[u1 -> 1]
u5	Esther	32	[u1 -> 2]
u6	Fanny	36	[]
u7	Gabby	60	[]

Example. Find for each user the length of the shortest path to users `u1` (Alice) and `u4` (David).

GraphFrame application:

```
1  from graphframes import GraphFrame
2  # Vertex DataFrame
3  v = spark.createDataFrame([("u1", "Alice", 34),
4                             ("u2", "Bob", 36),
5                             ("u3", "Charlie", 30),
6                             ("u4", "David", 29),
7                             ("u5", "Esther", 32),
8                             ("u6", "Fanny", 36),
9                             ("u7", "Gabby", 60)],
10                            ["id", "name", "age"])
11
12 # Edge DataFrame
13 e = spark.createDataFrame([("u1", "u2", "friend"),
14                           ("u2", "u3", "follow"),
15                           ("u3", "u2", "follow"),
16                           ("u6", "u3", "follow"),
17                           ("u5", "u6", "follow"),
18                           ("u5", "u4", "friend"),
19                           ("u4", "u1", "friend"),
20                           ("u1", "u5", "friend")],
21                           ["src", "dst", "relationship"])
22
23 # create the graph
24 g = GraphFrame(v, e)
25
26 # find for each user the length of the shortest paths to users u1 and u4
```

```
27 shortestPaths = g.shortestPaths(["u1", "u4"])
```

The output DataFrame:

id	name	age	distances
u1	Alice	34	[u1 -> 0, u4 -> 2]
u2	Bob	36	[]
u3	Charlie	30	[]
u4	David	29	[u1 -> 1, u4 -> 0]
u5	Esther	32	[u1 -> 2, u4 -> 1]
u6	Fanny	36	[]
u7	Gabby	60	[]

5.5.3 Connected Components

The connected component of a graph refers to a subgraph `sg` with:

- at least one connection path between any two vertexes in `sg`;
- no connection path between the vertexes in `sg` and the additional vertexes of the original graph.

The `connectedComponents()` method of the GraphFrame class returns the connected components of the input graph (expensive algorithm, requires a Spark checkpoint directory). The method returns a DataFrame with:

- one record/row for each distinct vertex of the input graph;
- one column for each attribute of the vertexes;
- the `component (type long)` column, containing for each vertex the identifier (identification code) of the connected component of belonging (for the current vertex).

Example. Print on the standard output the number of connected components of the graph in Figure 2.1.

GraphFrame application:

```
1  from graphframes import GraphFrame
2  # Vertex DataFrame
3  v = spark.createDataFrame([("u1", "Alice", 34),
4                             ("u2", "Bob", 36),
5                             ("u3", "Charlie", 30),
6                             ("u4", "David", 29),
7                             ("u5", "Esther", 32),
8                             ("u6", "Fanny", 36),
9                             ("u7", "Gabby", 60)],
10                            ["id", "name", "age"])
11
12 # Edge DataFrame
```

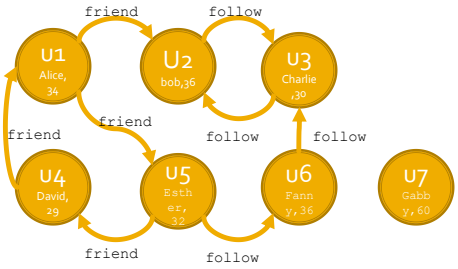

Figure 5.1: Graph

```
13 e = spark.createDataFrame([("u1", "u2", "friend"),
14                            ("u2", "u3", "follow"),
15                            ("u3", "u2", "follow"),
16                            ("u6", "u3", "follow"),
17                            ("u5", "u6", "follow"),
18                            ("u5", "u4", "friend"),
19                            ("u4", "u1", "friend"),
20                            ("u1", "u5", "friend")],
21                            ["src", "dst", "relationship"])
22
23 # create the graph
24 g = GraphFrame(v, e)
25
26 # set checkpoint folder
27 sc.setCheckpointDir("tmp_ckpts")
28 # run the algorithm
29 connComp = g.connectedComponents()
30 # count the number of components
31 nComp = connComp.select("component").distinct().count()
32 print("Number of connected components: ", nComp) # -> 2
```

The output DataFrame:

id	name	age	component
u1	Alice	34	146028888064
u2	Bob	36	146028888064
u3	Charlie	30	146028888064
u4	David	29	146028888064
u5	Esther	32	146028888064
u6	Fanny	36	146028888064
--	-----	--	----------
u7	Gabby	60	1546188226560

5.5.4 Strongly Connected Components

The directed subgraph `sg` appears strongly connected when every vertex in `sg` results reachable from every other vertex in `sg` (Figure 2.2). For undirected graph, connected components = strongly connected components. The `stronglyConnectedComponents()` method of the GraphFrame class returns

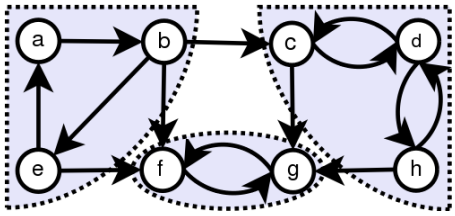

Figure 5.2: Strongly Connected Components

the strongly connected components of the input graph (expensive algorithm, requires a Spark checkpoint directory). The method returns a DataFrame with:

- one record/row for each distinct vertex of the input graph;
- one column for each attribute of the vertexes;
- the `component (type long)` column, containing for each vertex the identifier (identification code) of the strongly connected component of belonging (for the current vertex).

Example. Print on the standard output the number of strongly connected components of the graph in Figure 2.1.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# set checkpoint folder
sc.setCheckpointDir("tmp_ckpts")
# run the algorithm
strongConnComp = g.stronglyConnectedComponents(maxIter = 10)
# count the number of strongly connected components
nComp = strongConnComp.select("component").distinct().count()
print("Number of strongly connected components: ", nComp) # -> 4
```

The output DataFrame:

id	name	age	component
u1	Alice	34	498216206336
u4	David	29	498216206336
u5	Esther	32	498216206336
--	-----	--	----------
u2	Bob	36	146028888064
u3	Charlie	30	146028888064
--	-----	--	----------
u6	Fanny	36	1090921693184
--	-----	--	----------
u7	Gabby	60	1546188226560

5.5.5 Label Propagation

The label propagation algorithm detects communities in graphs, i.e. as clustering but exploiting connectivity (without guaranteeing convergence). Upon instantiation, the algorithm assigns each vertex in the network to the respective community. Afterward, at each step, vertexes send the respective community affiliation to each neighbor and update the state to the mode community affiliation of incoming messages. The `labelPropagation(maxIter)` method of the GraphFrame class executes and returns the result of the label propagation algorithm, with `maxIter`: the number of iterations to execute. The method returns a DataFrame with:

- one record/row for each distinct vertex of the input graph;
- one column for each attribute of the vertexes;
- the `label (type long)` column, containing for each vertex the identifier (identification code) of the community of belonging (for the current vertex).

Example. Split in groups the vertexes of the graph in Figure 2.1 via label propagation algorithm.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])
```

```
11
12 # Edge DataFrame
13 e = spark.createDataFrame([("u1", "u2", "friend"),
14                             ("u2", "u3", "follow"),
15                             ("u3", "u2", "follow"),
16                             ("u6", "u3", "follow"),
17                             ("u5", "u6", "follow"),
18                             ("u5", "u4", "friend"),
19                             ("u4", "u1", "friend"),
20                             ("u1", "u5", "friend")],
21                             ["src", "dst", "relationship"])
22
23 # create the graph
24 g = GraphFrame(v, e)
25
26 # run the label propagation algorithm
27 labelComm = g.labelPropagation(10)
```

The output DataFrame:

id	name	age	component
u1	Alice	34	498216206336
u4	David	29	498216206336
u5	Esther	32	498216206336
--	-----	--	----------
u2	Bob	36	1606317768704
u6	Fanny	36	1606317768704
--	-----	--	----------
u3	Charlie	30	146028888064
--	-----	--	----------
u7	Gabby	60	1546188226560

5.5.6 PageRank

PageRank refers to the famous original Google search engine algorithm for ranking vertexes (web pages) in a graph in order of importance. For the Google search engine, vertexes refer to web pages in the World Wide Web and edges refer to hyperlinks between web pages. The algorithm assigns a numerical weight (importance) to each node.

The algorithm computes the likelihood of arriving at a particular web page by randomly clicking on links. Roughly speaking, the algorithm assigns to each link a vote proportional to the importance of the respective source page `p`. In case the page `p` (with importance `PageRank(p)`) appears with `n` out-links, each out-link gets a vote equal to `PageRank(p)/n`. The page `p` gets an importance equal to the sum of the votes on the respective in-links. The PageRank recursive formulation results:

- initialize each page's rank to `1.0`, i.e. set `PageRank(p)` to `1.0` for each `p` in pages;

- iterate (for a maximum number of iterations): (a.) page `p` sends a contribution `PageRank(p)/numOutLinks(p)` to the respective neighbors (the pages `p` links), (b.) update each page's rank `PageRank(p)` to `sum(contributions)`, and (c.) return to step (a.).

The PageRank algorithm simulates the random walk of a user on the web. At each step of the random walk, the random surfer: (1.) with probability $1 - \alpha$, follow a link at random among the ones in the current page, (2.) with probability α, jump to a random page, i.e.

- initialize each page's rank to `1.0`, i.e. set `PageRank(p)` to `1.0` for each `p` in pages;
- iterate (for a maximum number of iterations): (a.) page `p` sends a contribution `PageRank(p)/numOutLinks(p)` to the respective neighbors (the pages `p` links), (b.) update each page's rank `PageRank(p)` to $\alpha + (1 - \alpha) \cdot$ `sum(contributions)`, and (c.) return to step (a.).

The `pageRank()` method of the GraphFrame class executes the PageRank algorithm on the input graph, with paratemers:

- `resetProbability`: probability of resetting to a random vertex (probability α of random jumps);
- `maxIter`: if set, the framework executes the algorithm for a specifiable number of iterations;
- `Tol`: if set, the framework executes the algorithm until the specifiable tolerance;
- `sourceId (optional)`: the source vertex for a personalizable PageRank.

The `pageRang()` method returns a GraphFrame with:

- the same vertexes and edges of the input graph;
- the `pagerank` attribute for each vertex of the output graph to store the respective PageRank;
- the `weight` attribute for each edge of the output graph to store the respective weight (PageRank contribution);
- `sourceId (optional)`: the source vertex for a personalizable PageRank.

Example. Apply the PageRank algorithm on the graph and select the user with the highest PageRank value.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
                           ("u4", "u1", "friend"),
                           ("u1", "u5", "friend")],
                           ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# run the PageRank algorithm
pageRanks = g.pageRank(maxIter = 30)
# select the maximum value of PageRank
maxPageRank = pageRanks.vertices.agg({"pagerank":"max"}).first()["max(pagerank)"]
# select the user with the maximum PageRank
pageRanks.vertices.filter(pageRanks.vertices.pagerank == maxPageRank).show()
```

5.5.7 Custom Graph Algorithms

The GraphFrames framework provides primitives for developing other custom graph algorithms. The process relies on message passing approach, with two key components: (1.) `aggregateMessages`, to send messages between vertexes and aggregate messages for each vertex, and (2.) `Joins`, to join message aggregates with the original graph.

Example. For each user, compute the sum of the ages of adjacent users.

GraphFrame application:

```
from graphframes import GraphFrame
# Vertex DataFrame
v = spark.createDataFrame([("u1", "Alice", 34),
                           ("u2", "Bob", 36),
                           ("u3", "Charlie", 30),
                           ("u4", "David", 29),
                           ("u5", "Esther", 32),
                           ("u6", "Fanny", 36),
                           ("u7", "Gabby", 60)],
                           ["id", "name", "age"])

# Edge DataFrame
e = spark.createDataFrame([("u1", "u2", "friend"),
                           ("u2", "u3", "follow"),
                           ("u3", "u2", "follow"),
                           ("u6", "u3", "follow"),
                           ("u5", "u6", "follow"),
                           ("u5", "u4", "friend"),
```

```
                            ("u4", "u1", "friend"),
                            ("u1", "u5", "friend")],
                            ["src", "dst", "relationship"])

# create the graph
g = GraphFrame(v, e)

# for each user, sum the ages of the adjacent users
# send the age of each destination of an edge to its source
msgToSrc = AggregateMessages.dst["age"]
# send the age of each source of an edge to its destination
msgToDst = AggregateMessages.src["age"]
# aggregate messages
aggAge = g.aggregateMessages(sum(AggregateMessages.msg), sendToSrc = msgToSrc, sendToDst = msgToDst)
# show result
aggAge.show()
```

Chapter 6

Streaming Data Analytics

Stream processing refers to the act of continuously incorporating input data to compute a result (input data without bounds, series of events arriving at the stream processing system). Some important applications process large streams of live data and provide results in near-real-time. The applications output multiple versions of the results over executions. Several frameworks for processing in real-time or in near real-time data streams propose two main solutions (via cluster of servers to scale horizontally with respect to the amount of data):

- **continuous** computation of data streams: the framework processes the input data upon arrival in order to emit the result as soon as possible (real-time processing);
- **micro-batch** stream processing: the framework collects the input data in micro-batches, each containing data coming from a time window, and processes a single micro-batch at a time when ready (near real-time processing).

6.1 Spark Streaming

Spark Streaming refers to the Spark framework for large scale stream processing (scales to 100s of nodes, second scale latencies, **micro-batch** streaming processing, **exactly-once** guarantees).

6.1.1 Discretized Stream Processing

The Spark Streaming framework executes a streaming computation as a series of very small, deterministic batch jobs, splitting each input stream into "portions" and processes one portion at a time (in incoming order). The framework applies the same computation on each portion (batch) of the stream. Spark streaming:

1. splits the live stream into batches of `x` seconds;

2. treats each batch of data as RDDs, processing accordingly via RDD operations;
3. returns the results of the RDD operations in batches.

Word Count Problem - Spark Streaming Version

In the word count problem (Spark streaming version), the application (1.) takes as input a stream of sentences, (2.) splits the input stream into batches of 10 seconds each, and (3.) prints on the standard output, for each batch, the occurrences of each word appearing in the batch (i.e., executes the word count application one time for each batch of 10 seconds).

Take into account some key concepts:

- **DStream**: sequence of RDDs representing a discretized version of the input stream of data (one RDD for each batch of the input stream);
- **Transformations**: modify data from one DStream to another via standard RDD operations (`map`, `countByValue`, `reduce`, `join`, etc), and window and stateful operations (`window`, `countByValueAndWindow`, etc);
- **Output Operations/Actions**: send data to external entity (`saveAsHadoopFiles`, `saveAsTextFile`, etc).

6.2 Spark Streaming Programs

The Spark framework enables to (1.) define a Spark Streaming Context object, (2.) define the size of the batches (in seconds) relating to the Streaming context, (3.) specify the input stream and define a DStream in compliance, (4.) specify the operations to execute for each batch of data, (5.) exploit transformations and actions similar to the ones available for "standard" RDDs, (6.) execute the start method for processing the input stream, and (7.) wait until the application dies or the timeout specifiable in the application expires (otherwise the application executes forever).

6.2.1 Spark Streaming Context

The framework enables to define the Spark Streaming Context object via the `StreamingContext(SparkConf sparkC, Duration batchDuration)` constructor of the `pyspark.streaming.StreamingContext` class. The `batchDuration` parameter specifies the "size" of the batches in seconds.

```
1 from pyspark.streaming import StreamingContext
2 ssc = StreamingContext(sc, 10)
```

The above code splits the input streams in batches of 10 seconds. After instantiating a context the developers (1.) define the input sources by creating input DStreams and (2.) define the streaming computations by applying transformation and output operations to DStreams.

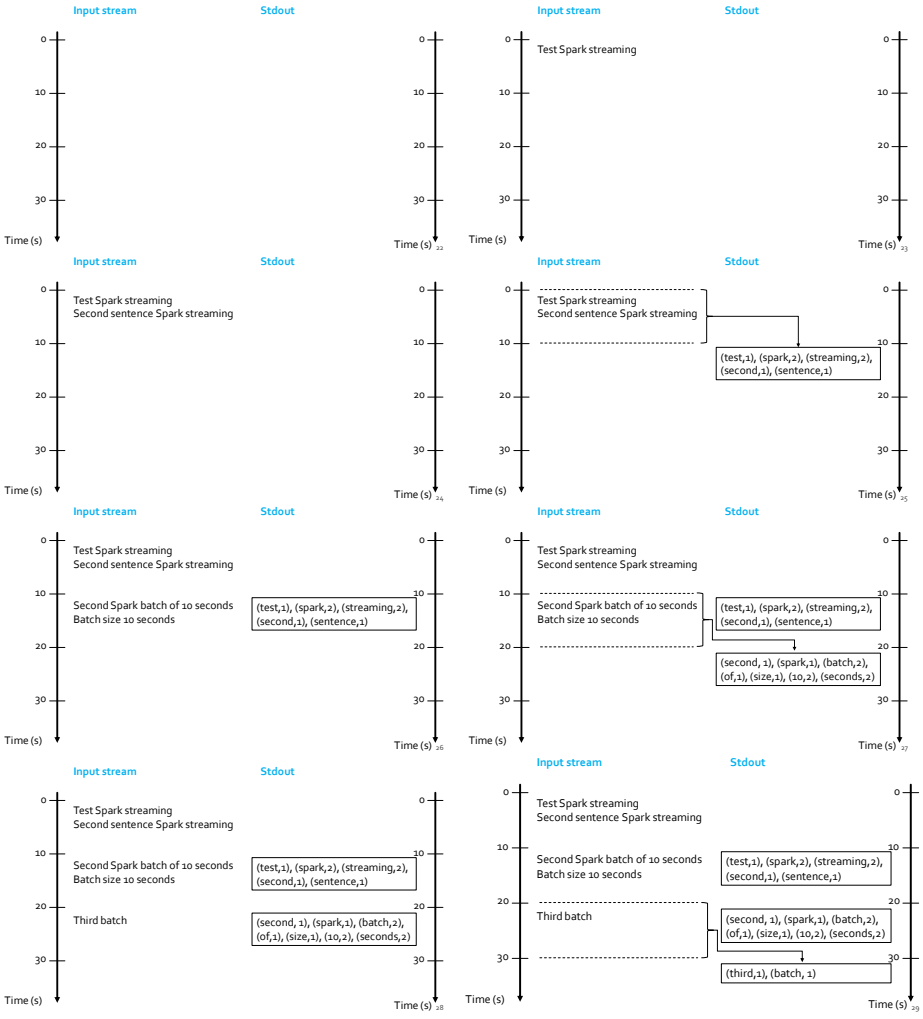

Figure 6.1: Word Count Problem (Spark Streaming Version)

6.2.2 Input Streams

The framework enables to generate input streams from different sources, e.g., TCP socket, Kafka, Flume, Kinesis, Twitter, HDFS folder:

- the `socketTextStream(String hostname, int port_number)` method enables to create a DStream according to the textual content coming from a TPC socket. For instance, `lines = ssc.socketTextStream("localhost", 9999)` "store" the content coming from `localhost:9999` in the lines DStream.
- the `textFileStream(String folder)` method enables to create a DStream according to the content of the input (HDFS) folder. For instance, `lines = textFileStream(inputFolder)` "store" the content of the files in the input folder in the lines Dstream. Every time someone inserts some files in the input folder, the method stores the respective content in the current "batch" of the stream.
- ...

In most cases, the developers define DStream objects on top of streams coming from specific applications emitting real-time streaming data, e.g., Apache Kafka, Apache Flume, Kinesis, Twitter.

6.3 Basic Transformations on DStreams

The framework provides the analogous set of standard RDD transformations to DStreams. Take into account that, as before, the transformations return a different DStream object with respect to the input one. The framework applies the transformation on one batch (RDD) of the input DStream at a time and returns one batch (RDD) of the output DStream, i.e. the framework relates each batch (RDD) of the input DStream to exactly one batch (RDD) of the output DStream.

- `map(func)`: returns an output DStream by passing each element of the source DStream through a function `func`;
- `flatMap(func)`: returns an output DStream, mapping each input item to 0 or more output items;
- `filter(func)`: returns an output DStream by selecting only the records of the source DStream on which `func` returns true;
- `reduce`: returns an output DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream via a (associative and commutative) function `func`;
- `reduceByKey(func)`: executable on a DStream of `(key, value)` pairs, returns an output DStream of `(key, value)` pairs by aggregating the values for each key via the specifiable reduce function `func`;

- `combineByKey(createCombiner, mergeValue, mergeCombiners)`: executable on a DStream of `(key, value)` pairs, returns an output DStream of `(key, value)` pairs by aggregating the values for each key via the specifiable combine functions `createCombiner, mergeValue, mergeCombiners`;
- `groupByKey()`: executable on a DStream of `(key, value)` pairs, returns an output DStream of `(key, Iterable<value>)` pairs by concatenating the values for each key with the other `k`-compatible values (apply `groupByKey` on one batch (one RDD) of the input stream at a time);
- `countByValue()`: executable on a DStream of elements of type `key`, returns an output DStream of `(key, long)` pairs, with as value for each key the relative frequency in each batch of the source Dstream;
- `count()`: returns an output DStream of single-element RDDs by counting the number of elements in each batch (RDD) of the source Dstream, i.e., it counts the number of elements in each input batch (RDD);
- `union(otherStream)`: returns an output DStream that contains the union of the elements in the source DStream and `otherDStream`;
- `join(otherStream)`: executable on two DStreams of `(key, v)` and `(key, w)` pairs, returns an output DStream of `(key, (v, w))` pairs with each pair of elements for each key;
- `cogroup(otherStream)`: executable on two DStreams of `(key, v)` and `(key, w)` pairs, returns an output DStream of `(key, Seq[v], Seq[w])` tuples;
- `pprint()`: prints the first 10 elements of every batch of data in a DStream on the standard output of the driver node running the streaming application (useful for development and debugging);
- `saveAsTextFiles(prefix,[suffix])`: saves the content of the DStream as text files, with one folder for each batch, by generating the folder name at each batch interval according to `prefix`, time of the batch (and `suffix`): `"prefix-TIME_IN_MS[.suffix]"`, e.g. `Counts.saveAsTextFiles(outputPathPrefix, "")`.

6.3.1 Start the computation

The Spark Streaming framework provides (1.) the `streamingContext.start()` method to start the application on the input stream(s), (2.) the `awaitTerminationOrTimeout(long millisecons)` method to specify the time termination criterion, (3.) the `waitTermination()` method to execute the application forever (until the applications dies or the manual stop via `streamingContext.stop()`). Take into account that:

- upon initialization (start) of the context, the application denies to set or add other streaming computations;

- upon termination of the context, the application denies the restore;
- the application activates only one `StreamingContext` at a time;
- `stop()` on `StreamingContext` also stops the `SparkContext` (to stop only the `StreamingContext`, set the optional parameter of `stop()`, i.e. `stopSparkContext`, to `False`).

Example - Word Count Problem. Input: a stream of sentences coming from `localhost:9999`. Split the input stream in batches of 5 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch, i.e., execute the word count problem for each batch of 5 seconds. Store the results also in an HDFS folder.

GraphFrame application:

```
1  from pyspark.streaming import StreamingContext
2
3  # set prefix of the output folders
4  outputPathPrefix = "resSparkStreamingExamples"
5
6  # create a configuration object and
7  # set the name of the application
8  conf = SparkConf().setAppName("Streaming word count")
9
10 # create a SparkContext
11 object sc = SparkContext(conf = conf)
12
13 # sreate a Spark Streaming Context object
14 ssc = StreamingContext(sc, 5)
15
16 # create a (Receiver) DStream that will connect to localhost:9999
17 lines = ssc.socketTextStream("localhost", 9999)
18
19 # apply a chain of transformations to perform the word count task
20 # the output RDDs = DStream RDDs
21 words = lines.flatMap(lambda line: line.split(" "))
22 wordsOnes = words.map(lambda word: (word, 1))
23 wordsCounts = wordsOnes.reduceByKey(lambda v1, v2: v1 + v2)
24
25 # print the result on the standard output
26 wordsCounts.pprint()
27
28 # store the result in HDFS
29 wordsCounts.saveAsTextFiles(outputPathPrefix, "")
30
31 # start the computation
32 ssc.start()
33 # run this application for 90 seconds
34 ssc.awaitTerminationOrTimeout(90)
35
36 ssc.stop(stopSparkContext = False)
```

6.4 Windowed Computation

The Spark Streaming framework also provides windowed computations, enabling developers to apply transformations over a sliding window of data (each window contains a set of batches of the input stream). Every time the window

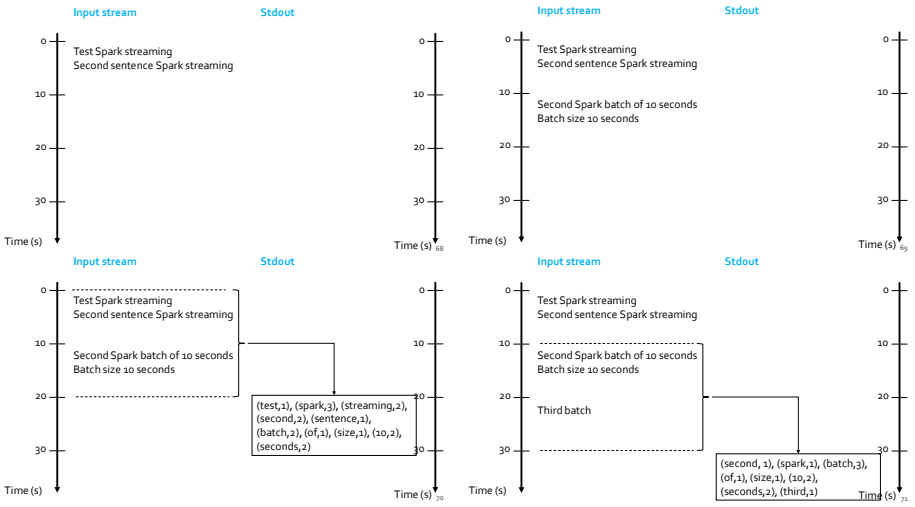

Figure 6.2: Word Count Problem and Window - with batches of 10 seconds, window length of 20 seconds (i.e., 2 batches), sliding interval of 10 seconds (i.e., 1 batch)

.

slides over a source DStream, the application combines and operates the source RDDs falling within the window upon to produce the RDDs of the windowed DStream. The window operation requires to specify two parameters: (1.) the window length, i.e. the duration of the window, and (2.) the sliding interval, i.e. the interval in which the application performs the window operation. The framework requires both parameters as multiples of the batch interval of the source DStream.

6.4.1 Basic Window Transformations

- `window(windowLength, slideInterval)`: returns an output DStream by computing according to the windowed batches of the source DStream;
- `countByWindow(windowLength, slideInterval)`: returns an output single-element stream containing the number of elements of each window (the method returns a Dstream of `long` objects, containing only one value for each window, i.e. the number of elements of the last window);
- `reduceByWindow(reduceFunc, invReduceFunc, windowDuration, slideDuration)`: returns an output single-element stream by aggregating elements in the stream over a sliding interval via a (associative and commutative) function `func` (in case `invReduceFunc` ! = `None`, the method reduces incrementally via the old window's 'reduce' value);
- `countByValueAndWindow(windowDuration, slideDuration)`: executable

on a DStream of elements of type `key`, returns an output DStream of `(key, long)` pairs, with as value for each key the relative frequency in each window of the source Dstream;

- `reduceByKeyAndWindow(func,invFunc, windowDuration, slideDuration = None, numPartitions = None)`: executable on a DStream of `(key, value)` pairs, returns an output DStream of `(key, value)` pairs by aggregating the values for each key via the specifiable reduce function `func over batches in a sliding window` (the `windowDuration` parameter enables to specify the window duration/length): (1.) in case `slideDuration` = `None`, the method exploits the `batchDuration` of the `StreamingContext` object, i.e., `1` batch sliding window, (2.) in case `invFunc` ! = `None`, the method reduces incrementally via the old window's 'reduce' value, i.e., the method exploits `invFunc` to apply an inverse reduce operation by considering the old values that left the window (e.g., subtracting old counts).

6.5 Checkpoints

In most cases, streaming applications operate 24/7 and hence not resilient to system failures, JVM crashes, and other failures external to the application logic. The Spark Streaming framework requires to checkpoint enough information to a fault-tolerant storage system in order to recover from failures. The checkpoints refers therefore to operations for storing the data and metadata necessary to restart the computation in case of failures (necessary even for some window transformations and stateful transformations). The framework enables checkpointing via the `checkpoint(String folder)` method of `SparkStreamingContext`, with as parameter the folder o store temporary data (similar as for processing graphs with GraphFrames library).

Example - Word Count Problem. Input: a stream of sentences coming from `localhost:9999`. Split the input stream in batches of 5 seconds each. Define widows with the following characteristics: window length = 15 seconds (i.e., 3 batches), sliding interval = 5 seconds (i.e., 1 batch). Print on the standard output, for each window, the occurrences of each word appearing in the window, i.e., execute the word count problem for each window. Store the results also in an HDFS folder.

Spark application (`version 1`):

```
1  from pyspark.streaming import StreamingContext
2
3  # set prefix of the output folders
4  outputPathPrefix = "resSparkStreamingExamples"
5
6  # create a configuration object and set the name of the application
7  conf = SparkConf().setAppName("Streaming word count")
8
9  # create a SparkContext object
10 sc = SparkContext(conf = conf)
```

```
11
12 # create a Spark Streaming Context object
13 ssc = StreamingContext(sc, 5)
14
15 # set the checkpoint folder (it necessary for some window transformations)
16 ssc.checkpoint("checkpointfolder")
17
18 # create a (Receiver) DStream that will connect to localhost:9999
19 lines = ssc.socketTextStream("localhost", 9999)
20
21 # apply a chain of transformations to perform the word count task
22 # the output RDDs = DStream RDDs
23 words = lines.flatMap(lambda line: line.split(" "))
24 wordsOnes = words.map(lambda word: (word, 1))
25
26 # leverage reduceByKeyAndWindow instead of reduceByKey
27 # and specify the durantion of the window
28 wordsCounts = wordsOnes.reduceByKeyAndWindow(lambda v1, v2: v1 + v2, None, 15)
29
30 # print the num. of occurrences of each word of the current window
31 # (only 10 of them)
32 wordsCounts.pprint()
33
34 # store the output of the computation in the folders with prefix
35 # outputPathPrefix
36 wordsCounts.saveAsTextFiles(outputPathPrefix, "")
37
38 # start the computation
39 ssc.start()
40
41 ssc.awaitTermination ()
```

Spark application (`version 2`):

```
1  from pyspark.streaming import StreamingContext
2
3  # set prefix of the output folders
4  outputPathPrefix = "resSparkStreamingExamples"
5
6  # create a configuration object and set the name of the application
7  conf = SparkConf().setAppName("Streaming word count")
8
9  # create a SparkContext object
10 sc = SparkContext(conf = conf)
11
12 # create a Spark Streaming Context object
13 ssc = StreamingContext(sc, 5)
14
15 # set the checkpoint folder (it necessary for some window transformations)
16 ssc.checkpoint("checkpointfolder")
17
18 # create a (Receiver) DStream that will connect to localhost:9999
19 lines = ssc.socketTextStream("localhost", 9999)
20
21 # apply a chain of transformations to perform the word count task
22 # the output RDDs = DStream RDDs
23 words = lines.flatMap(lambda line: line.split(" "))
24 wordsOnes = words.map(lambda word: (word, 1))
25
26 # leverage reduceByKeyAndWindow instead of reduceByKey
27 # and specify the durantion of the window
28 wordsCounts = wordsOnes.reduceByKeyAndWindow(lambda v1, v2: v1 + v2,
29                                              lambda vnow, vold: vnow-vold, 15)
30
31 # print the num. of occurrences of each word of the current window
32 # (only 10 of them)
33 wordsCounts.pprint()
34
```

```
35 # store the output of the computation in the folders with prefix
36 # outputPathPrefix
37 wordsCounts.saveAsTextFiles(outputPathPrefix, "")
38
39 # start the computation
40 ssc.start()
41
42 # run the application for 90 seconds
43 ssc.awaitTerminationOrTimeout(90)
44
45 ssc.stop(stopSparkContext = False)
```

6.6 Stateful Computation

The `updateStateByKey` transformation enables to maintain a "state" for each key. The transformation continuously updates the value of the state of each key each time the application parses an input batch. The `updateStateByKey` method relies on two steps:

- define the state: the data type of the state relating to the keys;
- define the state update function: specify with a function how to update the state of a key via the previous state and the output values from an input stream relating to such key.

In every batch, the Spark framework applies the state update function for each existing key, i.e. for each key:

- the application exploits the update function to update the value relating to a key by combining the former value and the output values relating to such key;
- the application executes the call method of the "function" on the list of output values and the former state value and returns the output value for the key of interest by aggregation.

By means of the `UpdateStateByKey`, the application continuously update the number of occurrences of each word, i.e. the transformation computes the number of occurrences over the union of every batches (from the first one to the current one) in order to return and store the results in the DStream. For efficiency reasons, the transformation computes the output value for each key by combining the last value for such key with the values of the current batch for the same key.

Example - Word Count Problem. Input: a stream of sentences coming from `localhost:9999`. Split the input stream in batches of 5 seconds each. Print on the standard output, every 5 seconds, the occurrences of each word appearing in the stream (from time `0` to the current time), i.e., execute the word count problem from the beginning of the stream to current time. Store

the results also in an HDFS folder.

Spark application:

```
from pyspark.streaming import StreamingContext

# set prefix of the output folders
outputPathPrefix = "resSparkStreamingExamples"

# create a configuration object and set the name of the application
conf = SparkConf().setAppName("Streaming word count")

# create a SparkContext object
sc = SparkContext(conf = conf)

# create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# set the checkpoint folder (it necessary for some window transformations)
ssc.checkpoint("checkpointfolder")

# create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# apply a chain of transformations to perform the word count task
# the output RDDs = DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))
wordsOnes = words.map(lambda word: (word, 1))

# define the function that to update the state of a key at a time
def updateFunction(newValues, currentCount):
    if currentCount is None:
        currentCount = 0
    # sum the output values to the previous state for the current key
    return sum(newValues, currentCount)

# DStream made of cumulative counts for each key that get updated in every batch
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)

# print the num. of occurrences of each word of the current window
# (only 10 of them)
totalWordsCounts.pprint()

# store the output of the computation in the folders with prefix
# outputPathPrefix
totalWordsCounts.saveAsTextFiles(outputPathPrefix, "")

# start the computation
ssc.start()

# run the application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext = False)
```

6.7 Transform Transformation

The framework does not provide some types of transformations for DStreams, e.g., `sortBy`, `sortByKey`, `distinct()`. Moreover, sometimes developers require to combine DStreams and RDDs, but the DStream API does not directly expose the functionality of joining every batch in a data stream with another dataset (a "standard" RDD). The framework provides the `transformation()` method

to overcome such situations. The `transform(func)` method refers to a specific transformation of DStreams to return an output DStream by applying an RDD-to- RDD function to every RDD of the source Dstream, useful to apply arbitrary RDD operations on the DStream.

Example - Word Count Problem. Input: a stream of sentences coming from `localhost:9999`. Split the input stream in batches of 5 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch. Sort the pairs by decreasing number of occurrences (per batch). Store the results also in an HDFS folder.

Spark application:

```
from pyspark.streaming import StreamingContext

# set prefix of the output folders
outputPathPrefix = "resSparkStreamingExamples"

# create a configuration object and set the name of the application
conf = SparkConf().setAppName("Streaming word count")

# create a SparkContext object
sc = SparkContext(conf = conf)

# create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# set the checkpoint folder (it necessary for some window transformations)
ssc.checkpoint("checkpointfolder")

# create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# apply a chain of transformations to perform the word count task
# the output RDDs = DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

wordsCounts = wordsOnes.reduceByKey(lambda v1, v2: v1 + v2)

# sort the content/the pairs by decreasing value (# of occurrences)
wordsCountsSortByKey = wordsCounts.transform(lambda batchRDD: batchRDD
                                    .sortBy(lambda pair: -1 * pair[1]))

# print the result on the standard output
wordsCountsSortByKey.pprint()

# store the result in HDFS
wordsCountsSortByKey.saveAsTextFiles(outputPathPrefix, "")

# start the computation
ssc.start()

# run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext = False)
```

6.8 Spark Structured Streaming

The Structured Streaming framework refers to a scalable and fault-tolerant stream processing engine built on top of the Spark SQL engine. The framework represents the input data by means of (streaming) DataFrames and leverages the existing Spark SQL APIs to query data streams, i.e. with the same methods useful for analyzing "static" DataFrames. The engine provides a set of specific methods to define input and output streams and windows.

6.8.1 Input Data Model

The engine models each input data stream as a continuously upgradeable table. For instance, every time an input data `A` arrives, the engine appends `A` at the end of the table, i.e. Structured Streaming considers each data stream as an input table without bounds (input data in the stream = rows to a table without bounds).

6.8.2 Queries

The engine executes queries incrementally on the input tables without bounds. The arrival of an input data triggers the execution of incremental queries, which the engine executes on the data present in the table from the beginning to the current arrival time step of the query-triggering input data. Aggregation queries combine input data with the previous results to optimize the computation of the results. The engine may executes queries as (1.) **micro-batch** queries with a fixable batch interval (standard behavior, exactly-once fault-tolerance guarantees), or as (2.) **continuous** queries (experimental, at-least-once fault-tolerance guarantees).

Take into account some key concepts:

- input sources;
- transformations;
- outputs (external destinations/sinks, output models);
- query run/execution;
- triggers.

6.8.3 Input Sources

File source: (1.) reads files written in a directory as a stream of data, (2.) each line of the input file refers to an input record, (3.) `text`, `csv`, `json`, `orc`, `parquet`, and others available file formats. Kafka source: (1.) reads data from Kafka, (2.) each Kafka message refers to one input record. Socket source (for debugging purposes): (1.) reads UTF-8 text data from a socket connection,

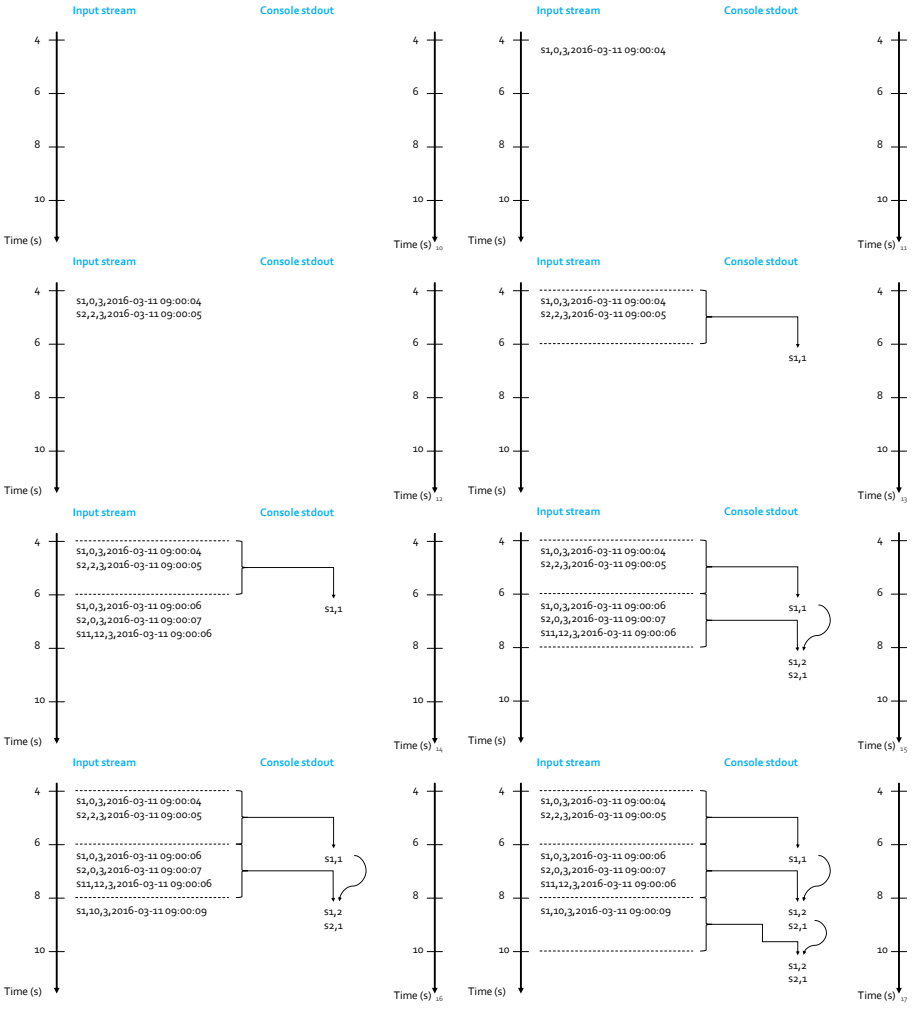

Figure 6.3: Example - Input: a stream of records coming from `localhost:9999`. Each input record refers to a reading about the status of a station of a bike sharing system in a specific timestamp, each in the format `station-id, free-slots, used-slots, timestamp`. For each `station-id`, print on the standard output the total number of input readings with a number of free slots equal to `0`. Print the information whenever an input data arrives via the micro-batch processing mode. Suppose the batch-duration = 2 seconds.

.

(2.) such type of source does not provide end-to-end fault-tolerance guarantees. Rate source (for debugging purposes): (1.) generates data at the specifiable number of rows per second, (2.) each row coming from the generation step contains a timestamp and value of type `long`.

The framework exploits the `readStream` property of the `SparkSession` class to create `DataStreamReaders`. The engine leverages then the `format()` and `option()` methods of the `DataStreamReader` class to specify the input streams (type, location, etc), and the `load()` method of the `DataStreamReader` to return DataFrames relating to the input data streams. For instance, to create the (streaming) DataFrame `recordsDF` and associate such DataFrame with the input stream of type socket (address: `localhost`, input port: `9999`), execute:

```
1 recordsDF = spark.readStream.format("socket").option("host", "localhost")
2                             .option("port", 9999).load()
```

6.8.4 Transformations

Transformations remain the same of DataFrames, but with some restrictions due to some not incrementally executable types of queries/transformations. Operations not available:

- multiple streaming aggregations (i.e. a chain of aggregations on a streaming DataFrame);
- limit and take first `N` rows;
- distinct operations;
- sorting operations on streaming DataFrames before an aggregation and in complete output mode;
- few types of outer joins on streaming DataFrames.

6.8.5 Outputs

Sinks: instances of the `DataStreamWriter` class, useful to specify the external destinations and store the results in the external destinations. File sink: stores the output to a directory (`text`, `csv`, `json`, `orc`, `parquet`, and others available file formats). Kafka sink: stores the output to one or more topics in Kafka. For each sink: executes arbitrary computation on the output records. Console sink (for debugging purposes): prints the output to the console every time the engine analyzes a batch of records, useful or debugging purposes on low data volumes (the engine collects and stores the entire output in the driver's memory after each computation). Memory sink (for debugging purposes): stores the output in memory as an in-memory table, useful for debugging purposes on low data volumes (the engine collects and stores the entire output in the driver's memory

after each computation).

The framework requires developers to signal Spark how to write output data to external destinations (query-type dependent task). Output modes available:

- **append** mode (default mode): the mode outputs the last trigger (computation) and adds only the current input rows to the result, available only for queries that never change rows (the mode guarantees to output each row only once), e.g. queries with only `select`, `filter`, `map`, `flatMap`, `filter`, `join`, etc;
- **complete** mode: the mode outputs the whole result to the sink after trigger (computation), available for aggregation queries;
- **update** mode: the mode outputs the last trigger (computation) and updates only the current rows (since the last trigger) in the result.

The engine exploits the `writeStream` property of the `SparkSession` class to create `DataStreamWriters`. The engine leverages then the `outputMode()`, `format()` and `option()` methods of the `DataStreamWriter` class to specify the output destination (data format, location, output mode, etc). For instance, to create the `DataStreamWriter streamWriterRes` and associate such `streamWriterRes` with the console, the engine set the output mode to append:

```
1 streamWriterRes = stationIdTimestampDF.writeStream.outputMode("append").format("console")
```

6.8.6 Query Run/Execution

The framework requires developer to explicitly execute the `start()` action on the relative sinks (`DataStreamWriter` objects referencing to the external destinations for storing the results) to start executing the respective queries/structured streaming applications (the engine enables to start several queries in the same application). The structured streaming queries run forever, until reaching some developer-specifiable termination criteria.

6.8.7 Triggers

The framework enables to specify whether and when to execute (by the engine) the input data for each Spark structured streaming query (as a micro-batch query with a fixable batch interval or as a continuous processing query (experimental). The framework provides the `trigger()` method of the `DataStreamWriter` class to specify the trigger type for each query:

- no trigger type (default trigger setting): the engine executes the query in micro-batch mode, each generable and processable as soon as the processing of the previous micro-batch ends;

- fixed interval micro-batches: the engine executes the query in micro-batch mode, processable at user-specifiable intervals (the `processingTime` of the `trigger()` method enables to specify the micro-batch size): (1.) in case the previous micro-batch completes within the interval, then the engine waits until the end of the interval before processing the next micro-batch, or (2.) in case the previous micro-batch takes longer than the interval to complete, then the next micro-batch starts as soon as the previous one completes;

- one-time micro-batch: the engine executes the query in micro-batch mode, but only one time on one single micro-batch containing every available data of the input stream (the query stops after the single execution): such trigger type appears useful when developers attempt to periodically spin up a cluster, processing everything available since the last period, and then shutdown the cluster;

- continuous with fixable checkpoint interval (experimental): the engine executes the query in the low-latency, continuous processing mode, with at-least-once fault-tolerance guarantees.

Example (1) - Sparks Structured Streaming. Input: a stream of records coming from `localhost:9999`. Each input record refers to a reading about the status of a station of a bike sharing system in a specific timestamp, each in the format `station-id, free-slots, used-slots, timestamp`. Output: for each input reading with a number of free slots equal to `0` print on the standard output the value of `station-id` and `timestamp`.

Spark application:

```
1  from pyspark.sql.types import *
2  from pyspark.sql.functions import split
3
4  # create a "receiver" DataFrame for connecting to localhost:9999
5  recordsDF = spark.readStream.format("socket").option("host", "localhost")
6                              .option("port", 9999).load()
7
8  # the input records appear with one single column called value
9  # of type string
10 # example of an input record: s1,0,3,2016-03-11 09:00:04
11 # define four more columns by splitting the input column value
12 # new columns:
13 # - stationId
14 # - freeslots
15 # - usedslots
16 # - timestamp
17
18 readingsDF = recordsDF.withColumn("stationId", split(recordsDF.value, ',')[0].cast("string"))
19                       .withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer"))
20                       .withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer"))
21                       .withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))
22
23 # filter data
24 # use the standard filter transformation
25 fullReadingsDF = readingsDF.filter("freeslots = 0")
26
27 # select stationid and timestamp
```

```
# use the standard select transformation
stationIdTimestampDF = fullReadingsDF.select("stationId", "timestamp")

# store/print the result of the structured streaming query on
# the console "sink".
# append output mode
queryFilterStreamWriter = stationIdTimestampDF.writeStream.outputMode("append").format("console")

# start the execution of the query (until explicit stop)
queryFilter = queryFilterStreamWriter.start()
```

Example (2) - Sparks Structured Streaming. Input: a stream of records coming from `localhost:9999`. Each input record refers to a reading about the status of a station of a bike sharing system in a specific timestamp, each in the format `station-id, free-slots, used-slots, timestamp`. Output: for each `station-id`, print on the standard output the total number of input readings with a number of free slots equal to `0`. Print the information whenever an input data arrives via the standard micro-batch processing mode.

Spark application:

```
from pyspark.sql.types import *
from pyspark.sql.functions import split

# create a "receiver" DataFrame for connecting to localhost:9999
recordsDF = spark.readStream.format("socket").option("host", "localhost")
                            .option("port", 9999).load()

# the input records appear with one single column called value
# of type string
# example of an input record: s1,0,3,2016-03-11 09:00:04
# define four more columns by splitting the input column value
# new columns:
# - stationId
# - freeslots
# - usedslots
# - timestamp

readingsDF = recordsDF.withColumn("stationId", split(recordsDF.value, ',')[0].cast("string"))
                      .withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer"))
                      .withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer"))
                      .withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))

# filter data
# use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots = 0")

# count the number of readings with a number of free slots equal to 0
# for each stationId
# use the standard groupBy method
countsDF = fullReadingsDF.groupBy("stationId").agg({"*":"count"})

# store/print the result of the structured streaming query on
# the console "sink".
# complete output mode
# (do not use append output mode for aggregation queries)
queryFilterStreamWriter = stationIdTimestampDF.writeStream.outputMode("complete").format("console")

# start the execution of the query (until explicit stop)
queryFilter = queryFilterStreamWriter.start()
```

6.9 Event Time and Window Operations

In the context of streaming data analytics, input streaming records comes with temporal information, e.g. the data generation time (event time). In most cases, developers attempt to operate by taking into account the event-time and windows containing data referencing to the same event-time range. For instance, compute the number of events coming from each IoT device every minute according to the event-time (the data generation time). For each window referencing to one distinct minute consider only the data with an event-time inside that minute/window and compute the number of events for each IoT device (one computation for each minute/window). In such scenario, developers require the data generation time (i.e., the event time) rather than the data arrival time to the Spark application. Spark enables to define windows according to the time-event input column and then apply aggregation functions over each window. The framework requires developers to specify the name of the time-event column in the input (streaming) DataFrame for each structured streaming query on which to apply a window computation. The (sliding) windows comes with some characteristics, e.g. `windowDuration` and `slideDuration` (in order to get non-overlapping windows, i.e., with a `slideDuration` equal to `windowDuration`, do not set the `slideDuration` parameter). The framework enables to set different window characteristics for each query of the application. The framework provides the `window(timeColumn, windowDuration, slideDuration = None)` function to specify, inside the standard `groupBy()`, the characteristics of the windows (only with queries applying aggregation functions).

Example (3) - Event Time and Window Operations. Input: a stream of records coming from `localhost:9999`. Each input record refers to a reading about the status of a station of a bike sharing system in a specific timestamp, each in the format `station-id, free-slots, used-slots, timestamp`, where `timestamp` refers to the `event-time column`. Output: for each `station-id`, print on the standard output the total number of input readings with a number of free slots equal to `0` in each window (query execution for each window). Set `windowDuration` to `2` seconds and no `slideDuration`, i.e.,non-overlapping windows.

Spark application:

```
1  from pyspark.sql.types import *
2  from pyspark.sql.functions import split
3  from pyspark.sql.functions import window
4
5  # create a "receiver" DataFrame for connecting to localhost:9999
6  recordsDF = spark.readStream.format("socket").option("host", "localhost")
7                              .option("port", 9999).load()
8
9  # the input records appear with one single column called value
10 # of type string
11 # example of an input record: s1,0,3,2016-03-11 09:00:04
12 # define four more columns by splitting the input column value
13 # new columns:
14 # - stationId
```

```
# - freeslots
# - usedslots
# - timestamp

readingsDF = recordsDF.withColumn("stationId", split(recordsDF.value, ',')[0].cast("string"))
                      .withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer"))
                      .withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer"))
                      .withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))

# filter data
# use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots = 0")

# count the number of readings with a number of free slots equal to 0
# for each stationId in each window
# windowDuration = 2 seconds
# no overlapping windows
countsDF = fullReadingsDF.groupBy(window(fullReadingsDF.timestamp, "2 seconds"), "stationId")
                         .agg({"*":"count"})
                         .sort("window")

# store/print the result of the structured streaming query on
# the console "sink".
# complete output mode
# (do not use append output mode for aggregation queries)
queryCountWindowStreamWriter = countsDF.writeStream.outputMode("complete")
                                       .format("console")
                                       .option("truncate", "false")

# start the execution of the query (until explicit stop)
queryCountWindow = queryCountWindowStreamWriter.start()
```

6.9.1 Late Data

The framework handles data arriving later than expected (late data) according to the respective event-time. Spark provides an accurate control over updating old aggregates even in presence of late data. For each time the Spark application processes an input data, the framework computes the result by combining old aggregate values and the input data by considering the event-time column instead of the time Spark receives the data.

Example (4) - Event Time and Window Operations. Input: a stream of records coming from `localhost:9999`. Each input record refers to a reading about the status of a station of a bike sharing system in a specific timestamp, each in the format `station-id, free-slots, used-slots, timestamp`, where `timestamp` refers to the `event-time column`. Output: for each window, print on the standard output the total number of input readings with a number of free slots equal to `0` (query execution for each window). Set `windowDuration` to `2` seconds and no `slideDuration`, i.e.,non-overlapping windows.

Spark application:

```
from pyspark.sql.types import *
from pyspark.sql.functions import split
from pyspark.sql.functions import window

```

```
# create a "receiver" DataFrame for connecting to localhost:9999
recordsDF = spark.readStream.format("socket").option("host", "localhost")
                            .option("port", 9999).load()

# the input records appear with one single column called value
# of type string
# example of an input record: s1,0,3,2016-03-11 09:00:04
# define four more columns by splitting the input column value
# new columns:
# - stationId
# - freeslots
# - usedslots
# - timestamp

readingsDF = recordsDF.withColumn("stationId", split(recordsDF.value, ',')[0].cast("string"))
                      .withColumn("freeslots", split(recordsDF.value, ',')[1].cast("integer"))
                      .withColumn("usedslots", split(recordsDF.value, ',')[2].cast("integer"))
                      .withColumn("timestamp", split(recordsDF.value, ',')[3].cast("timestamp"))

# filter data
# use the standard filter transformation
fullReadingsDF = readingsDF.filter("freeslots = 0")

# count the number of readings with a number of free slots equal to 0
# for each window
# windowDuration = 2 seconds
# no overlapping windows
countsDF = fullReadingsDF.groupBy(window(fullReadingsDF.timestamp, "2 seconds"))
                         .agg({"*":"count"})
                         .sort("window")

# store/print the result of the structured streaming query on
# the console "sink".
# complete output mode
# (do not use append output mode for aggregation queries)
queryCountWindowStreamWriter = countsDF.writeStream.outputMode("complete")
                                       .format("console")
                                       .option("truncate", "false")

# start the execution of the query (until explicit stop)
queryCountWindow = queryCountWindowStreamWriter.start()
```

6.10 Watermarking

Watermarking refers to a Spark feature that enables (1.) the user to specify the threshold of late data and (2.) the engine to accordingly clean up old state. In most real-world scenarios, the streaming application does not require results relating to old event-times. The watermarking therefore enables to drop old state to improve the efficiency of the application,so as to enable the application to consider only recent data whenever processing an input data. Specifically, to execute window-based queries for days, the system requires to bound the amount of accumulate intermediate in-memory state, i.e. the system requires to understand whenever an old aggregate appears droppable from the in-memory state. To enable such feature, Spark 2.1 introduces watermarking. Watermarking lets the Spark Structured Streaming engine automatically track the current event time in the data and attempt to clean up old state accordingly. The feature enables developers to define the watermark of a query by specifying the event time column and the late threshold (threshold for the expected delay for

data in terms of event time). For a specific window ending at time `T`, the engine maintains state and enables late data to update the state/the result until the maximum event time seen by the engine $<$ `T` + late threshold. Roughly speaking, the engine aggregates late data within the threshold, but drops data later than `T` + late threshold.

6.11 Join Operations

Spark Structured Streaming manages also join operations between two streaming DataFrames and/or between a streaming DataFrame and a static DataFrame. The framework generates the result of the streaming join incrementally. In case of join between two streaming DataFrames, the framework buffers/records, for both input streams, past input streaming in order to match every future input record with past input data and accordingly generate joined results. Since storing each input data results expensive in terms of resources, the framework discards old data, i.e. developers define watermark thresholds on both input streams such that the engine understands when to drop old data. The framework provides the `join()` and `withWatermark()` methods to join streaming DataFrames (similar to the one available for static DataFrame).

```
1  from pyspark.sql.functions import expr
2  impressions = spark.readStream. ...
3  clicks = spark.readStream. ...
4
5  # apply watermarks on event-time columns
6  impressionsWithWatermark = impressions.withWatermark("impressionTime", "2 hours")
7  clicksWithWatermark = clicks.withWatermark("clickTime", "3 hours")
8
9  # join with event-time constraints
10 impressionsWithWatermark.join(clicksWithWatermark,
11                               expr("""clickAdId=impressionAdId AND
12                               clickTime>=impressionTime AND
13                               clickTime <= impressionTime + interval 1 hour""") )
```