NUMERO: 2369A

ANNO: 2018

A P P U N T I

STUDENTE: Chiapello Nicolò

MATERIA: Bigdata - Prof. Garza

BIG DATA

THEORY

Polytechnic of Turin

Master of Science in Computer Engineering

Chiapello Nicolò

Content

About this course

Name:	Big Data: architectures and data analytics
Scientific-disciplinary sector:	ING-INF/05
Activities:	characterizing
Credits:	6 ECTS
Professor:	Paolo Garza
Academic degree:	master degree in computer engineering
Period:	first year, second semester
Academic year:	2017 / 2018
Author:	Chiapello Nicolò
Version:	1.0.1

About this work

This work would like to be the transposition of the class explanations, provided by the teacher.
This is the result of the writing done directly during lessons and following corrections and adjustments (especially during exam preparation time). They are not lecturer stenography, but *real-time* notes, with all the inaccuracies that this could lead to.
This work is not checked by the teacher in any way, so it could contain mistakes. If you find one you are invited to signal it to the author, in order to correct it (nicochina.notes@gmail.com).
This notes are *as is*: they are an useful tool to help in studying, but they are not a source of academic notions.

Legend

In this work the following convention is adopted:

- *italic*: concept/entity previously defined
- **bold**: relevant concept, in evidence
- underlined: mini-title to which the following text is referred to
- **yellow bold**: definition of a concept (reported in the final index)
- light blue: topic covered by the paragraph

Course presentation

Course of *Big Data: architectures and data analytics* in the second semester of the first year of the master degree.

Professors

This course is hold by the following teachers:

- lecturer: Paolo Garza
- lab assistant: Daniele Apiletti

Teaching organization

With respect with the previous year, the course spend less time on Hadoop and more about Spark.

There are 10 laboratories:

- 1-4 labs: Hadoop
- 5-10 labs: Spark

We need a second account in LABINF, to access to big data cluster (BigData@Polito).

Topics

This course will deal with the following topics.

- Introduction to Big Data
- Hadoop
  - Architecture
  - MapReduce programming paradigm
- Spark
  - Architecture
  - Spark programs
- Data mining and Machine Learning libraries for Big Data
  - MLlib (Apache Spark's scalable machine learning library)
- Streaming data analysis
  - Spark streaming
- SQL databases for relational big data and NoSQL databases
  - Data models, Design, Querying

Hadoop is a framework used to store and analyse data. We will see MapReduce paradigm, on Hadoop. It is used to solve basic problems (not all kind of problems) just using two functions: Map and Reduce.
Spark is another framework, more complex, with more functionalities. It can also be executed on Hadoop architecture.

Introduction to Big Data

Big Data is, simply, a large amount of data (about petabytes) to be analysed in order to extract information (by machine learning approach).

Internet is a huge source of data (Facebook, Google, etc…). Also sensor generate many data to be analysed all together.

Using Internet is often performing queries. Analysing those queries, we can perform *nowcasting*. While *forecasting* is something that will happen in the future, the **nowcasting** in predict something that is happening right now.

Many companies has many data, but they do not know how to use them. How to use those data is the most challenging task in the Big Data area.
Another main problem is the storage of those data. Locally some aggregation is performed and then only interesting information are send trough Internet to reach the main control site.
Of course the algorithm efficiency is a problem, too.

Big Data source

There are many Big Data sources:

- user generated content (web and mobile): social network and website
- health and scientific computing: research, geographical distribution and medical diagnostic
- log files: web server log files, machine system log files
- IoT (Internet of Things): sensor network, RFID, smart meters

While a log file analysis is often performed offline, a sensor analysis have to be real-time, so an online computing.

"If a service is free, so probably you're the product."
-cit. Garza

Big Data definition

There are many potential definition of Big Data, but there are some standard keyword.

The **Big Data** are data whose scale, diversity and complexity require new architecture, technique, algorithms and analytics to manage it and extract value and hidden knowledge from it.

Big Data is about produce new architecture to solve new problem, but also to solve traditional problems with new algorithms.

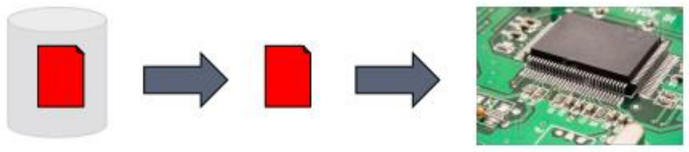
Big data challenges

The Big Data lead to many challenges of different type:

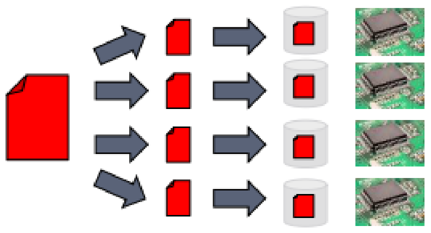
- technology and infrastructure: new architecture, programming paradigm and technique
- data management and analysis: data science related topics

The processors processes data, while hard drives store them. We need to transfer data from the disk to the processor.
This task could became a bottleneck in our process. A possible solution is to split the file in different independent smaller files and process them in a parallel way. Then, only at the end of all the elaboration, local statistics are aggregated together.
Not all algorithm can be parallelised, because all file must be independent from the content of the other files.

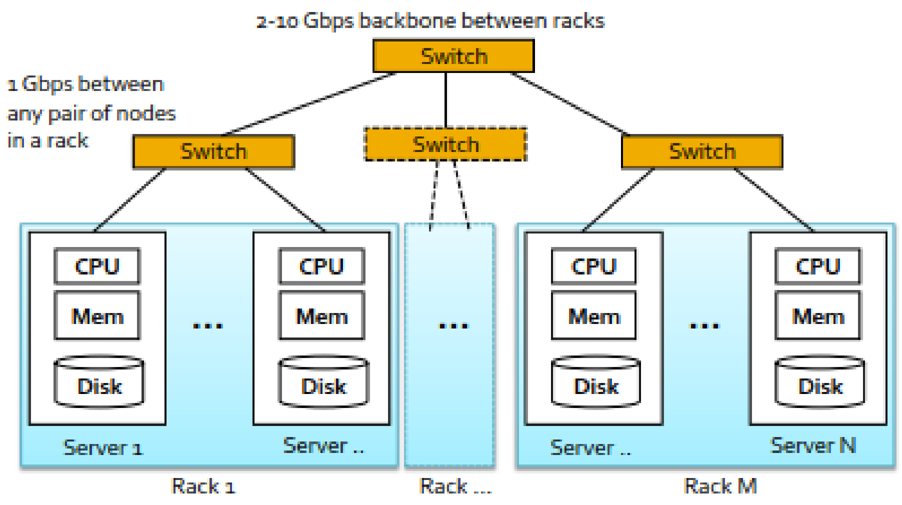
sequential execution



parallel execution



According to *data locality*, each computation should be performed locally, but to put all the **racks** (standardized frame or enclosure for mounting multiple electronic equipment modules) together, a network infrastructure (with switch) is required.
The communication inside a rack is much more faster.



Scalability

Scalability is the ability to grown up the power of the system. System must scale to address the increasing amount of data, number of users and complexity of the problem.

Type of scalability:

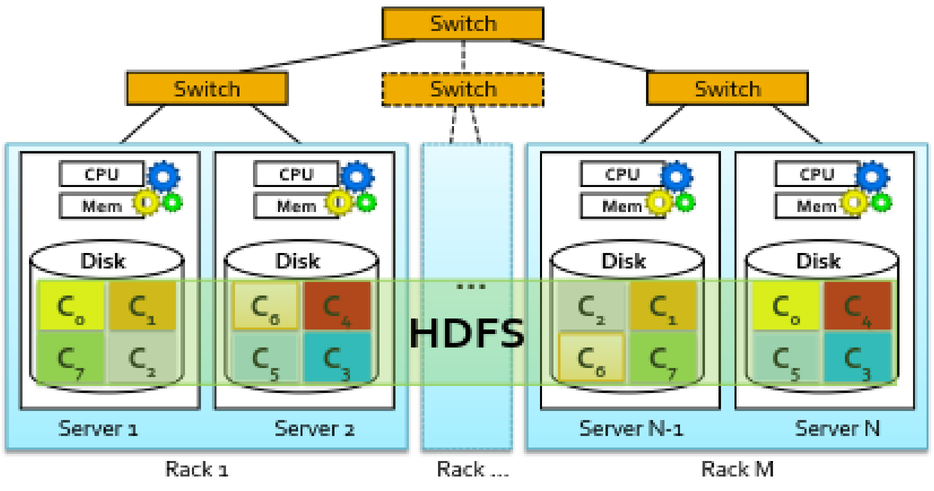
- **vertical scalability** (scale up): increase the power of a single component
- **horizontal scalability** (scale out): increase the number and the parallelism of the whole system

For large amount of data, *scale out* approach is better and less expensive (theoretically).

- single component: price not linear with respect to the amount of memory
- many components: price linear with respect to the amount of memory

With Hadoop framework we do not take care of the schedule or the parallelism management, because the system do it automatically.
Hadoop provides a very high level API to address those tasks:

- parallelization
- distributed storage of data sets
- node failure management
- diverse input format



Example with number of replicas per chunk = 2

1. Distributed Big Data Processing Infrastructure

The **Distributed Big Data Processing Infrastructure** separates the "what" from the "how", so an high level view. Hadoop abstract away the "distributed" part of the problem (scheduling, synchronization, etc…).
The programmer still should take care about the amount of data he send through the network (because of the bottleneck problem).

2. HDFS

The **HDFS (Hadoop Distributed File System)** is the standard File System provided by Hadoop. The programmer can use also some other distributed File System or a Database source for the Hadoop application.
The HDFS is a distributed File System that is fault-tolerant and designed to handle huge files (GB to TB). A single data block store only one file, even though there's also remaining free space. For this reason HDFS is not the best choice for many little files.
Another principle of HDFS (and of MapReduce approach, too) is that collected new data are inserted at the end of the file. There's not random access to data (all data are read sequentially) and old data are not updated.

- insert only at the end
- read all data (no random access)
- no update

In HDFS each file is split in **chunks** (pieces of the file) that are spread across the server. Each chunk is replicated on different server a number of time that is a parameter of the framework (usually 3 times). Each replica cost time and computational power, so a trade-off is requested.
Replicas are stored in different racks (if possible) for persistence and availability matters.
Typically each chunk is $64MB$ or $128MB$.

Principles of Hadoop and MapReduce

We will analyse the *Word Count* problem, that is a simple introduction to the MapReduce paradigm as well as *Hello Word* problem is to the general programming.

Word Count

The **Word Count problem**, given a large textual file of word as input, aims to count the number of occurrences of each word. The output would be the list of the pair $\langle word, repetition \rangle$.
The sequential execution of this task is trivial: scan all the file, one word after the other and increase a counter array at each word occurrence.
The parallel execution is much more interesting in order to improve the speed. The textual file is divided into N pieces and use to feed N machines that provide a **local solution** (word frequency inside the given section).
To avoid a bottleneck in the collection process, a second step is introduced. This second layer is composed by N machines that perform the merge of all local results. In order to parallelize also this task, we can instruct each machine to perform the **global result** of a set of items having similar characteristics (words starting with the same letter).
By using this approach each machines in the first layer have to send data to many machines in the second layer, but the performances are really efficient (thanks to parallelization).
This method is the base concept for the **MapReduce programming paradigm**:

- first step: map function
- second step: reduce function

According to the file size, there are two possible solution:

- entire file fits in main memory: a traditional single node approach is more efficient, because it avoid the overhead of the framework
- file too large to fit in main memory: detect a set of independent sub-tasks

We will analyse only the second possibility.

Implementation

For this implementation, suppose the following situations:

- the cluster has 3 servers
- the content of the input file is this string: “Toy example file for Hadoop. Hadoop running example.”
- the input file is split in two chunks (number of replicas=1)

The *Word Count problem* is parallelized by splitting it into two distinct phases:

1. each server processes its chunk of data and counts the number of times each word appears in its chunk
   - each server can perform it independently $\Longrightarrow$ synchronization is not needed in this phase

MapReduce Programming Paradigm

MapReduce is based on **functional programming** in which everything is a function and those can be sequentially applied to some data in order to retrieve a result. A chain of functions can be applied, each works on the previous function's output. The output is dependent only from the value of the input; it is like a mathematical function, there is no stored state.

The MapReduce implements a subset of functional programming because it uses only two building blocks:

- **Map function**
- **Reduce function**

Being the programming model so limited, the complexity of the problems we can solve is limited as well.

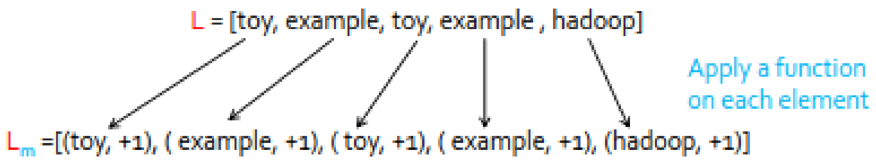
The **Map function** is applied over each element of an input data set and emits a set of $(key, value)$ pairs.
The **Reduce function** is applied over each set of $(key, value)$ pairs (emitted by the map function) with the same key and emits a set of $(key, value)$ pairs, that is usually the final result.

Example: Word Count

Text:
Given the Word Count problem, analyse it using the MapReduce approach.

- input: a textual file (i.e., a list of words)
- problem: count the number of times each distinct word appears in the file
- output: a list of pairs $\langle word, number\ of\ occurrences\ in\ the\ input\ file \rangle$

Solution:
We can see the input file as a list of words L. For each input word, we emit a key-value pair having as key the word itself and as value the fixed number +1.


L = [toy, example, toy, example , hadoop]
Apply a function on each element
L_m =[(toy, +1), (example, +1), (toy, +1), (example, +1), (hadoop, +1)]

Instead of increasing the value of and already met word, we simply create a further key-value pair (e.g. toy and example words), because there is no state about the number of occurrences.
The second step is to group the first set of key-value pair, by considering the key. The generated key-value pairs have the word as key and the list of occurrences (+1) as value.

Phases

In the MapReduce paradigm there are three fixed steps:

- **Map phase**: applying the Map function to each key-value pair, we emit a set of key-value pair
- **Shuffle and Sort phase**: merging the key-value pairs sharing the same key
- **Reduce phase**: merging the local results to compute the final one

The developer should take care only about Map and Reduce phases because the Shuffle and Sort one is always the same. This step is automatically handled by Hadoop.

The mathematical representation: given the input list of words L, the Map function $m(\cdot)$ (applied to each word w) is defined as follows:

$$m(w) = (w, 1)$$

The a new list L_m of key-value pairs is generated. The key-value pairs in L_m are aggregated by key (i.e., by word in our example) generating many G_w groups:

$$G_w = (w, [list\ of\ values])$$

The Reduce function $r(\cdot)$ is applied to each group G_w in order to obtain the final result:

$$r(G_w) = \big(w, \mathrm{sum}(G_w.[list\ of\ values])\big)$$

The Map phase can be viewed as a **transformation** over each element of a data set. This transformation is a function $m(\cdot)$ defined by the designer and each its application happens in isolation (independently to other items), so can be parallelized.
The Reduce phase can be viewed as an **aggregate operation**. This aggregate function is a function $r(\cdot)$ defined by the designer and each group of key-value pairs with the same key can be processed in isolation (so in parallel, but cannot merge different keys in this phase).
The Shuffle and Sort phase is always the same and it does not need to be defined by the designer.

Data structure

MapReduce is based on the **key-value pair** data structure. Everything is a list of key-value pair, also the input (not as supposed in the previous example).
The keys and values can be:

- integer
- float
- string
- user defined structure

The design of MapReduce involves also the definition of the key and value structure for both input and output data sets.

Formal definition

The map and reduce functions are formally defined as follows:

$$map: (k_1, v_1) \rightarrow [(k_2, v_2)]$$
$$reduce: (k_2, [v_2]) \rightarrow [(k_3, v_3)]$$

Hadoop implementation of MapReduce

The designer/developer have to implement only two methods: `map()` and `reduce()`. They stand each in the corresponding classes: `Map` and `Reduce`.
There is no need to manage the distributed execution of the phases. The Hadoop framework will also take care of coordinate the execution:

- parallel execution of the map and reduce phases
- execution of the shuffle and sort phase
- scheduling of subtasks
- synchronization

In this course we will use Java programming language. An Hadoop MapReduce program consists in three main classes:

- **Driver**: contains the `main()` method, the application workflow and all the needed configuration
- **Mapper**: implements the map function
- **Reducer**: implements the reduce function

Each class is characterized by some specific interfaces/abstract classes and each class are instantiated in an object.

Terminology

Just for sake of comprehension, we define some terminology:

- Driver class: configuration of the job and of the workflow
- Mapper class: implements the map function
- Reducer class: implements the reduce function
- Driver: instance of the Driver class
- Mapper: instance of the Mapper class
- Reducer: instance of the Reducer class
- (Hadoop) job: execution of a MapReduce program on a dataset (composed by many tasks)
- task: execution related to each instantiated object (Driver/Mappers/Reducers)
- input split: fixed-size portion of the input data (usually having the same size of a HDFS block/chunk)

Driver

The **Driver** is a part of Hadoop MapReduce implementation. It is characterized by the `main()` method, which accepts arguments from the command line (e.g. entry point of the application).
The Driver performs the following operations:

- configures the job: setting cluster options and defining the job name
- submits the job to the Hadoop Cluster: actually launch the job with `submit()` command
- coordinates the workflow of the application: specifying the Mapper/Reducer classes to use and report the `map()`/`reduce()` functions in a MapReduce chain

Hadoop automatically splits the input file and creates a Mapper task for each *input split*. The number of Mappers is defined by the number of *input splits* (so by the input file size).
The number of Reducer is specified by the developer, it depends on the specific situation.
Number of:

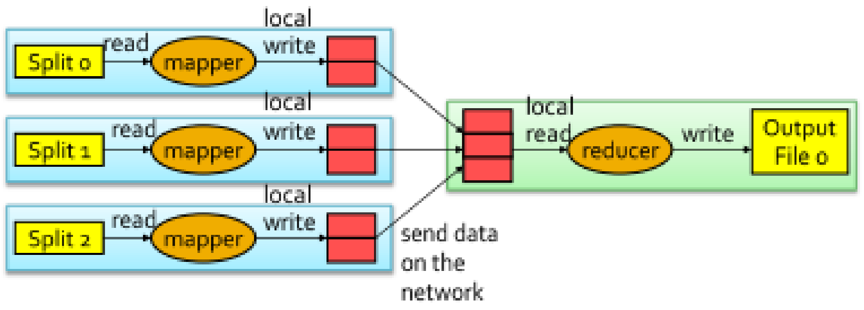
- Mappers: automatically defined by Hadoop
- Reducers: developer defined

Example:

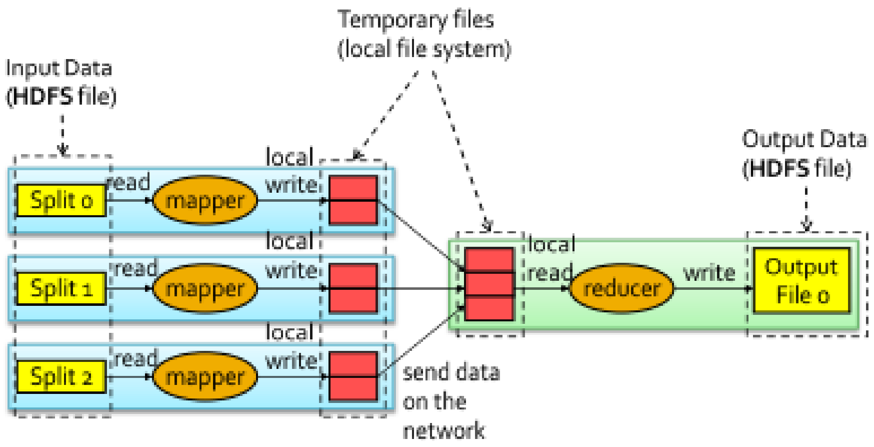
Given three different servers containing only one input split each, the sequential action are defined by the following diagrams.

Single reducer:

Having a single Reducer, Hadoop perform the following actions:



For efficiency reason the intermediate result are not stored in the distributed file system (HDFS), but locally:



Usually the amount of data emitted by the Mapper should be smaller than the size of the input file. If this does not happen, probably there is some issue.

- mapper class
  - name of the class
  - type of its input $(key, value)$ pairs
  - type of its output $(key, value)$ pairs
- reducer class
  - name of the class
  - type of its input $(key, value)$ pairs
  - type of its output $(key, value)$ pairs
- number of reducers

Mapper class

The **Mapper class** is a part of a MapReduce program. Using OO inheritance properties, there are predefined templates, so it should:

- extends `org.apache.hadoop.mapreduce.Mapper` class

that is a generic class (with generic type parameters), so the developer should define the input key type, input value type, output key type and output value type. The generic parameters are indicated using the Java *diamond* operator.

The developer should implement only the following method:

- `map()`: analyses data

This method is automatically called by the framework for each $(key, value)$ pair of the input file. The `map()` method processes its input by using standard Java code and emits $(key, value)$ pairs by using the following method:

```
context.write(key, value);
```

A peculiar case of mapper is the **identity mapper**: the mapper simply emit the incoming $(key, value)$ pair without performing any computation or filtering on them. The reason for their existence is that is possible to create a job without the reducer (map-only job), but it is not possible to create a reduce-only job (the map element always should be present).

Reducer class

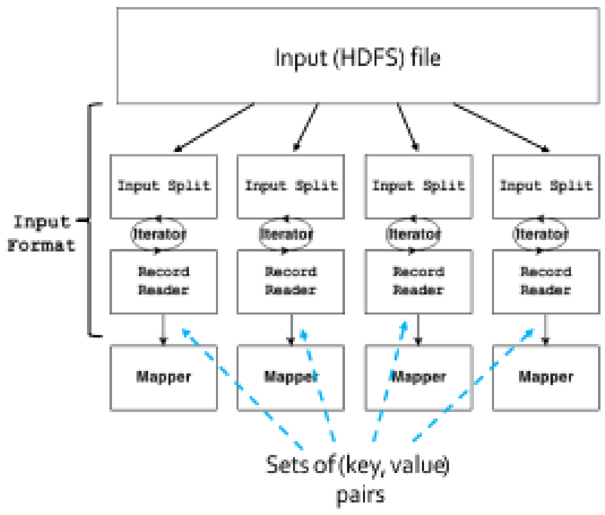
The **Reducer class** is a part of a MapReduce program. Using OO inheritance properties, there are predefined templates, so it should:

- extends `org.apache.hadoop.mapreduce.Reducer` class

that is a generic class (with generic type parameters), so the developer should define the input key type, input value type, output key type and output value type. The generic parameters are indicated using the Java *diamond* operator.

The developer should implement only the following method:

- `reduce()`: group together similar data



There already are standard implementation for plain text files:

> `TextInputFormat`
> `KeyValueTextInputFormat`

and also for sequential/binary files:

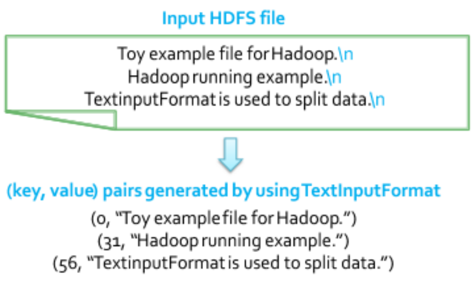
> `SequenceFileInputFormat`

Textual files are split into lines (linefeed or carriage-return separator) and a key-value pair is emitted for each of them:

- <u>`TextInputFormat`</u>: the key is the position (offset) in the file of the first character and the value is the content of the line
- <u>`KeyValueTextInputFormat`</u>: requires to specify a delimiter, the key is the string preceding the delimiter and the value is the following one

<u>Example:</u>

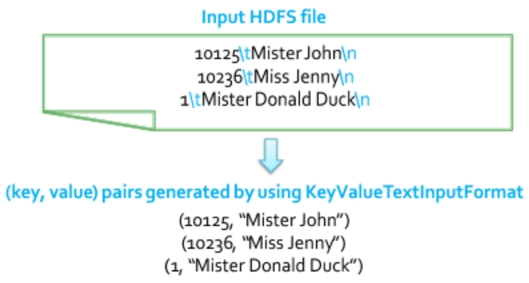
> Given a textual file, split it using standard `InputFormat` classes.
>
> `TextInputFormat`:



```
Toy example file for Hadoop.\n
Hadoop running example.\n
TextinputFormat is used to split data.\n
```

```
(0, "Toy example file for Hadoop.")
(31, "Hadoop running example.")
(56, "TextinputFormat is used to split data.")
```

> `KeyValueTextInputFormat`:



```
10125\tMister John\n
10236\tMiss Jenny\n
1\tMister Donald Duck\n
```

(key, value) pairs generated by using KeyValueTextInputFormat

```
(10125, "Mister John")
(10236, "Miss Jenny")
(1, "Mister Donald Duck")
```

```
        ...

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

        // Define and configure a new job
        Configuration conf = this.getConf();
        Job job = Job.getInstance(conf);

        // Assign a name to the job
        job.setJobName("My first MapReduce program");

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

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

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

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

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

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

```
                ReducerInputKeyType key,
                Iterable<ReducerInputValueType> values,
                Context context) throws IOException, InterruptedException {

            //process the input (key, [list of values]) pairs and emit a set of (key,
            value) pairs
            context.write(new outputkey, new outputvalue);
    }

}
```

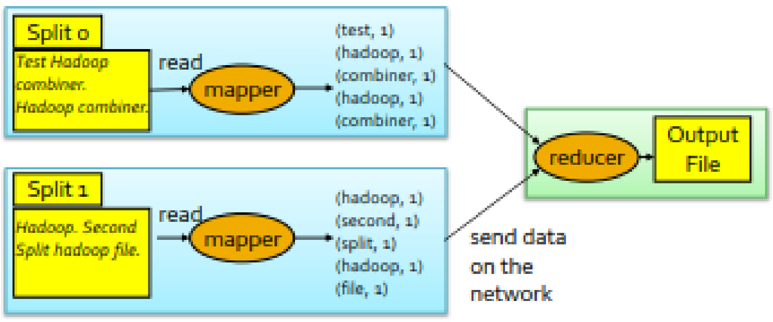
4. Combiner

The **Combiner** is a part of Hadoop MapReduce modelled in a class. Their instance are created in the same node of the Mapper and have the aim to perform some aggregation in order to limit the amount of network data.

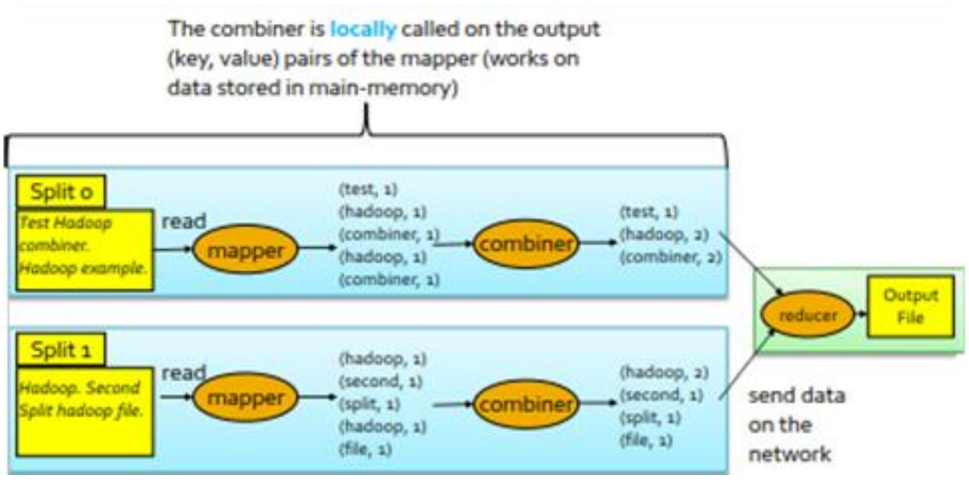
Example:

Consider the Word Count problem having two Input Splits.

Standard solution:



With Combiners:



A Combiner is similar to a Reducer, but it is applied locally on the output of a single Mapper. Frequently the we can exploit the same class both for the Combiner and for the Reducer.

```
public class SumAndCountWritable implements org.apache.io.Writable {
    //private variables
    private float sum = 0;
    private int count = 0;

    //methods to serialize and deserialize the contents of the instances of this
    class
    @Override //serialize the fields of this object to out
    public void write(DataOutput out) throws IOException {
        out.writeFloat(sum);
        out.writeInt(count);
    }

    @Override //deserialize the fields of this object to out
    public void readFields(DataInput in) throws IOException {
        sum = in.readFloat(sum);
        count = in.readInt(count);
    }

    // Specify how to convert the contents of the instances of this class to a
    String
    // Useful to specify how to store/write the content of this class in a textual
    file
    public String toString() {
        String formattedString = new String("sum="+sum+",count="+count);

        return formattedString;
    }
}
```

Keys

Personalized data types can be used also to manage complex keys. In this situation it should be comparable, so it must implement two interfaces:

```
org.apache.hadoop.io.Writable
org.apache.hadoop.io.ComparableWritable
```

Looking at two key, the program should know if one precedes the other or, at least, if they are equals. The implementation is really similar to the value case.

Sharing parameters among Driver, Mappers and Reducers

We can send data among Driver, Mappers and Reducers. This three classes compose the standard MapReduce application. They contain the Map and the Reduce function.
Sometimes the programmer need to share some parameters among them (e.g. a minimum threshold of occurrences). To do so we can exploit the **Configuration** object: it contains a set of predefined properties (name, value). This object is used by the gateway to properly configure the cluster for the submitting of a task, but we can exploit it to set some particular personalized properties.
Those properties should be configured before submit the job. The properties are composed by the pair ⟨*propery_name*; *property_value*⟩.

Example:

> A user defined counter, to store the number input lines not consistent with the format, can be defined, in the Driver, by the following code:
>
> ```
> public static enum COUNTERS {
>         ERROR_COUNT,
>         MISSING_FIELD_RECORD_COUNT
> }
> ```
>
> This enum defines two counters:
>
> - COUNTERS.ERROR_COUNT
> - COUNTERS.MISSING_FIELDS_RECORD_COUNTER
>
> In the Mapper or in the Reducer, we can increment the value of the wanted counter (for instance `COUNTERS.ERROR_COUNT` counter):
>
> ```
> context.getCounter(COUNTERS.ERROR_COUNT).increment(1);
> ```
>
> At the end of the job, in the Driver, to retrieve the final value of the counter, we write the following code (for `COUNTERS.ERROR_COUNT` counter):
>
> ```
> job.getCounters().findCounter(COUNTERS.ERROR_COUNT);
> ```

The counters store only integer values, so we need to implement smart solution to store float value (multiply for 10^x and then to for 10^{-x} at the end of the job) or to get the average of a value (store two information in two different counters).

Be careful because invoking the method:

```
System.out.println(….);
```

The result will be shown inside the cluster (the machine actually performing the job), not in the shell of our own IDE.
In our specific situation, the HUE environment show differ tabs for both: `stdout`, `stderr` and `syslog`.

Map-only job

To just select the lines satisfying a particular constraint and removing the other, we can exploit a *map-only job*.
The Map part, in the emitted key-value pair:

- key: the whole line
- value: nothing

The Reduce do not perform any job: it simply move as output, the given input. For those situation the Reducer is removed.

In the **map-only job** only the Map is implemented (not Reduce is created), so the output of the Mapper is the output of the whole application.
We use a map-only job only when we want a filter on the input file. It's the only possible application for this kind of job.

The *reduce-only job* does not exists: it has no meaning.

Advanced aspects

Multiple Inputs and Multiple Outputs

The input for an Hadoop job could be an entire directory, but all files should have the same format. Even the output can be composed by several files, but all having the same format.
In this chapter we will analyse how to exploits input/output having different format. This happens often when we use different datasets.

1. Multiple Inputs

Is it possible to use a specific part of the Hadoop API to specify a different Mapper for each input file, but the Mapper output should be the same.

A **dataset** is a set of files sharing the same format. For each different dataset is necessary to create a specific Mapper able to handle that specific data format. However the key-value pair emitted by all the Mappers should be consistent (same).

In order to handle multiple input format within the same application, it is possible to use several times the `addInputPath()` method, of the **`MultipleInput` class**, in the Driver to add one input path at a time. The method has the following parameters:

- object of the job
- input path to add
- input format class
- Mapper class associated with the specified input path

It is not necessary to specify several time the format of the Mapper output, because it is always the same (it is consistent).

Example:

> To handle two different input data types, creates two Mapper classes and define a multiple input:
> ```
> MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class,
>                             Mapper1.class);
> MultipleInputs.addInputPath(job, new Path(args[2]), TextInputFormat.class,
>                             Mapper2.class);
> ```

2. Multiple Outputs

Similarly to the `MultipleInput`, sometimes we need to provide multiple outputs (several different format and semantic).
Each file contains a specific subset of the emitted key-value pairs (based on some rules), so this approach is usually used for splitting and filtering operations. There could be only one single folder containing all the multiple output files.
We can specify the file name, by setting a specific prefix in order to distinguish one from another. The standard prefix is "part-" with variant R (for using a reducer) or M (for map-only job).

The efficiency of the distributed cache approach depends on the number of multiple mappers (or reducers) running on the same node because the copy during the initialization costs, so use it for several tasks reduce the overall negative impact.

Code structure

We will analyse the template to implement the Distributed Cache by code.

To define the shared files in the Driver:

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

To exploits the shared files in the Mapper or/and in the Reducer:

```
protected void setup(Context context) throws IOException, InterruptedException {
    ...
    String line;

    // Retrieve the paths of the local copies of the distributed files
    Path[] PathsCachedFiles = context.getLocalCacheFiles();

    //  Read the content of the cached file and process it in this example the
    content of the first shared file is opened
    BufferedReader file = new BufferedReader(
                 new FileReader(
                              new File(PathsCachedFiles[0].toString())
                              )
                 );

    // Iterate over the lines of the file
    while ((line = file.readLine()) != null) {
          // process the current line
          ...
    }

    file.close();

}
```

All the key-value pairs associated with the same key are always processed by the same Reducer (generic MapReduce feature). There is one Reducer for each key.
In this pattern the key-value pairs emitted on the network are often equal to the input amount of data (no filtering is applied).

The *numerical summarizations pattern* is used for the following use cases (situations):

- word count
- record count
- min/max/count
- average/median/standard deviation

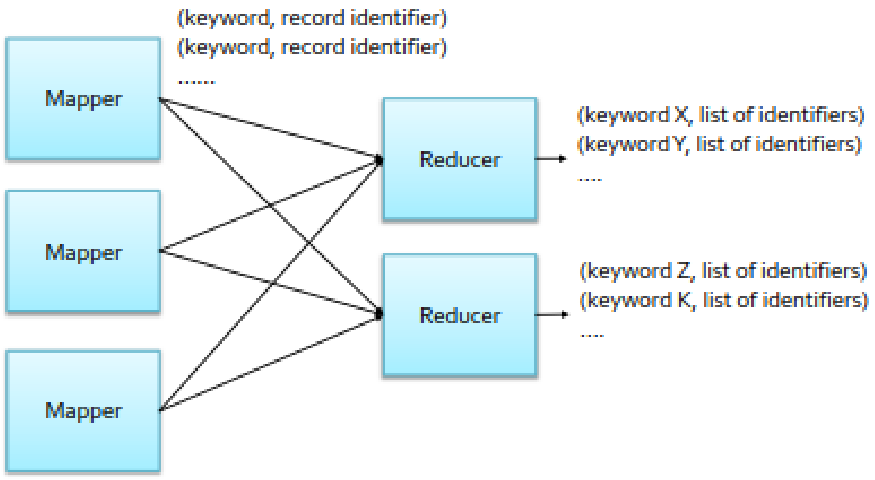
Those applications can also be performed by a DBMS, but using MapReduce they are parallelized and so they can handle a huge amount of data.

b. Inverted index summarizations

The **inverted index summarization pattern** is focused on creating an *inverted index*. An **inverted index** is an index data structure storing a mapping from content to its location in the file. It allows fast searches or data enrichment and it can be easily updated.
To create an inverted index with MapReduce, the final key should be a content (e.g. a specific word) and the value should be the list of the positions of their occurrences.

The pattern is implemented as follows:

- Mappers: the output are $(key, value)$ pairs in which the key is the fields to index (a keyword) and the value is an unique identifier of the objects to associate with each “keyword”
- Reducers: receive a set of identifiers for each keyword and simply concatenate them
- Combiners: usually not used in this pattern because there are no values to aggregate


(keyword, record identifier)
(keyword, record identifier)
......
Mapper
Mapper
Mapper
Reducer
Reducer
(keyword X, list of identifiers)
(keyword Y, list of identifiers)
....
(keyword Z, list of identifiers)
(keyword K, list of identifiers)
....

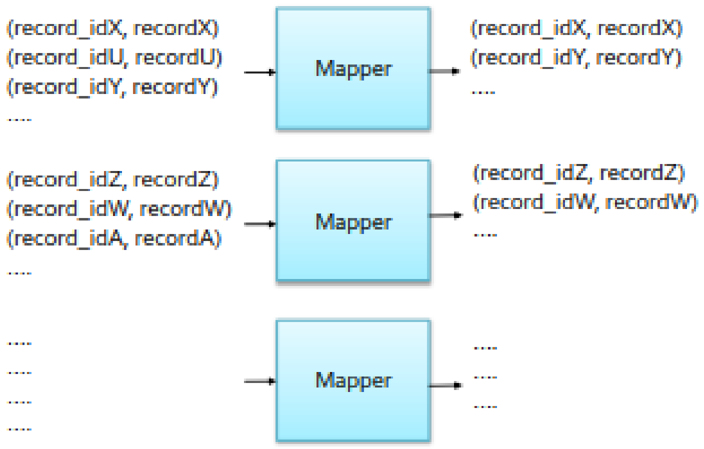
The diagram is similar to the *numerical summarizations patterns*, but it is different because the final output contains a list of items (not single summary).

The *Inverted Index Summarization pattern* most famous use case:

- Web search engine (word ↔ list of URLs)

- Mappers: the output is a $(key, value)$ pair, for each record that satisfy the enforced filtering rule, in which the key is the primary key of the record and the value is the selected record itself

Once filtered, the usage of key and value is not fixed (not only the proposed one allowed): are possible other mapping between the input key-value and the output key-value pairs.



The *filtering pattern* typical use cases:

- record filtering (remove not interesting record)
- tracking events (identify interesting situations)
- distributed grep (find specific words in a text)
- data cleaning (remove errors from the incoming file)

b. Top K

The **top *K* pattern** is focused on selecting the K top records according to a ranking function. After performing a rank often not all record are interesting, but only the most important ones according to the ranking criteria.

The pattern is implemented usually using an in-mapper combiner:

- Mappers:
  - each mapper initializes an in-mapper top k list
    - k is usually small ($k \simeq 10$)
    - the current top k-records of each mapper can be stored in main memory
    - initialization performed in the setup method of the mapper
  - the map function updates the current in-mapper top k list
  - the `cleanup()` method emits the k key-value pairs associated with the in-mapper local top k records
    - key is the `null key`
    - value is a in-mapper top k record
- Reducer:
  - a single reducer is instantiated
  - it computes the final top k list by merging the local lists emitted by the mappers

The *distinct pattern* typical use cases:

- duplicate data removal
- distinct value selection

3. Data organization patterns

The **data organization patterns** are used to organize the data: it allows to split the input data in subsets. They exploit the `MultipleInput` and `MultipleOutput` feature and usually provide the input for another application.
The two main patterns are the following:

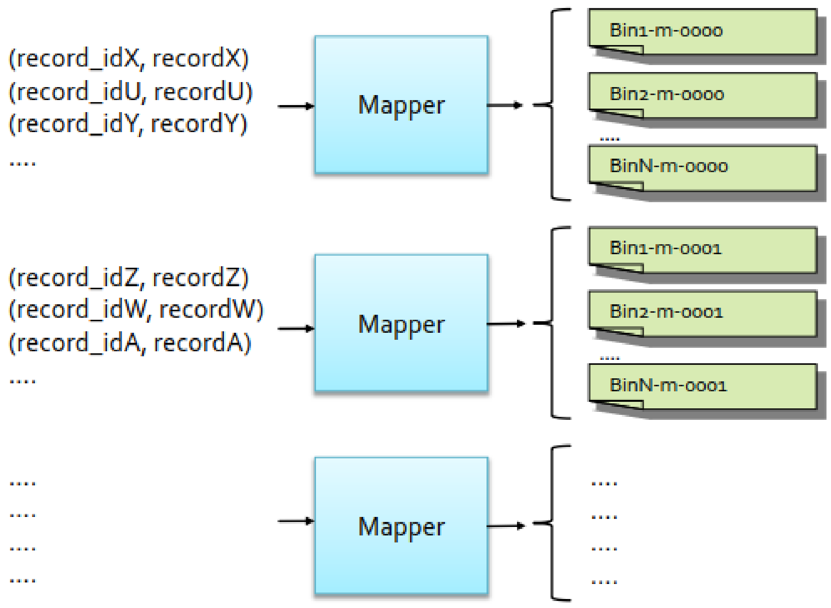
- binning
- shuffling

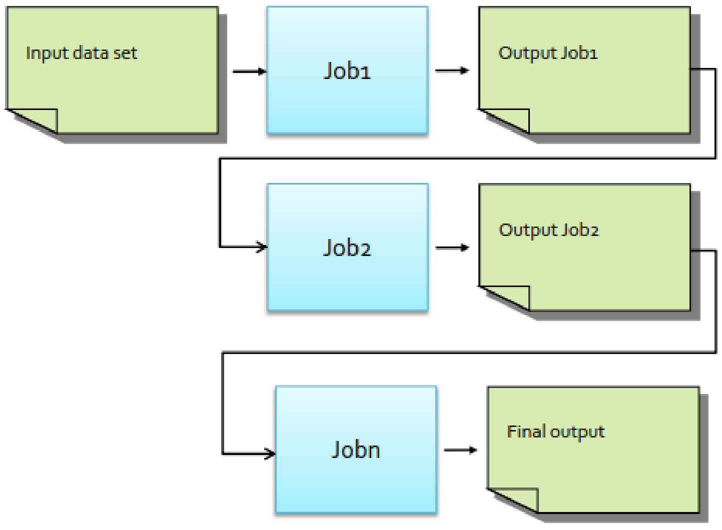
a. Binning

The **binning pattern** is focused on organize the input data into categories. Those groups are called **bins**: smaller dataset containing similar records (according to some specified rules).
To do so with Hadoop it is possible to use `MultipleOutput` prefixes to define different output files (a different prefix for each *bin*). Hence the number of output files is equal to the number of defined *bins*. Often the resulting files are analysed by other applications.
The rules should be specified to the application (using coding constructs) and they should be effective only over one single record (isolation required).

The pattern is implemented as a map-only job:

- Driver: sets an output file prefix to each bin, by means of `MultipleOutput`
- Mappers: according to some given rules, for each input $(key, value)$ pair, they select the proper output bin and emit a $(key, value)$ in that file
  - emitted key: input key
  - emitted value: input value





It is also possible to run jobs in parallel and then merge their results. In the Driver there's a proper command to wait the end of a specified job and it is necessary for the synchronization among different jobs.

5. Join patterns

The **join patterns** are used to implement the SQL-like join operators. The **join operation** is an relational algebra operation that merge together two different tables joining the records that share some value. The resulting schema is the sum of the two single schemas (excluding the overall).
The *join patterns* are the most frequent ones: they are really used to join different input files.
The main related patterns are the following:

- Reduce side join
- Map side join

We will focus on the *natural join*, however the pattern is analogous for the other types of joins:

- *natural join*
- *theta-join*
- *semi-join*
- *outer-join*

It is necessary to change only one function in the entire application.

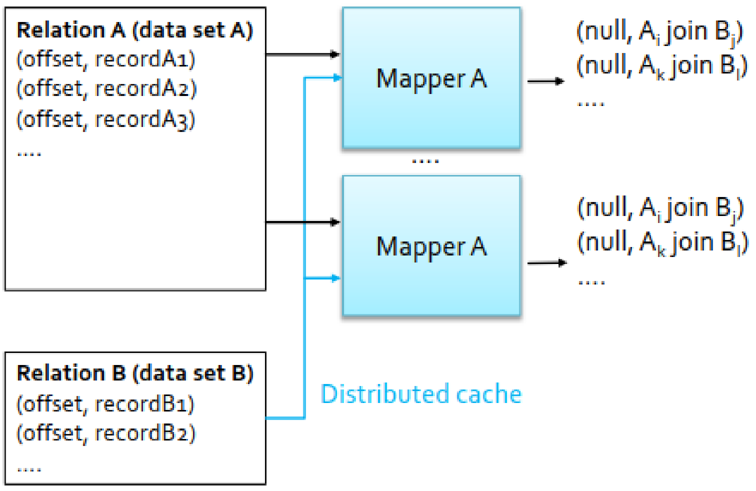
a. Reduce side natural join

The **reduce side natural join** is focused on joining the content of two relations (tables). The **natural join** has no predicates and join records according to the attribute having the same name.
The *reduce side join* is useful for big tables that cannot be stored in main memory. This approach can always be used, but it less efficient than the *map side join*.

The pattern is implemented as a map-only job with a single mapper class that processes the content of the large table. The distributed cache approach is used to provide a copy of the small table to each node performing a map task.

It is needed only one class:

- Mapper: perform the "local natural join" between the current record (of the large table) and the records of the small table (that is in the distributed cache loaded during the `setup()` method)



c. Theta-join, semi-join, outer-join

The SQL language provides many types of joins:

- *natural join*
- *theta-join*
- *semi-join*
- *outer-join*

In order to implement them, the same pattern (shown above for the *natural join*) can be exploited simply by changing the part called "local join".

This step appears:

- reduce side join: in the Reducer
- map side join: in the Mapper

The "local join" should be substituted with the type of join of interest (theta-, semi-, or outer-join).

2. Projection

The **projection**, for each record of table R, keeps only the attributes in S producing a relation with schema S and remove duplicates (if any):

$\pi_S(R)$

In MapReduce the projection is implemented as follows:

- Mappers: analyse one record r in R at a time and for each of them they create a new record r' (according to attribute in S) and emit a key-value pair where $key = r'$ and $value = null$
- Reducers: emit one key-value pair for each input $(key, [list\ of\ values])$ pair with $key = r'$ and $value = null$

3. Union

The **union** of two tables (R and S) having the same schema, produce a relation with the same schema, but with a record t for each record t appearing in R or S (duplicated records are removed)

$R \cup S$

In MapReduce the union is implemented by defining two Mapper classes (one for R and the other for S), but the task assigned to each mapper is the same.

- Mappers: one mapper for R and another for S; they both emit a key-value pair, for each input record t, in which $key = t$ and $value = null$
- Reducers: emit one (key, value) pair for each input $(key, [list\ of\ values])$ pair with $key = t$ and $value = null$

The duplicate are automatically removed by applying this pattern.
The problem of those solutions is the fact that all data are send through the network from the Mappers to the Reducers. There is no filtering and this network hyper use is a problem especially for big tables.

4. Intersection

The **intersection** of two tables (R and S) having the same schema, produce a relation with the same schema, but with a record t in the output if and only if t appears in both relations (R and S)

$R \cap S$

In MapReduce it is implemented using two Mapper classes: each one associated with one table.

- Mappers: one mapper for R and another for S; they both emit a key-value pair, for each input record t, in which $key = t$ and $value = table\_name$
- Reducers: emit one key-value pair with $key = t$ and $value = null$ for each input $(key, [list\ of\ values])$ pair with $[list\ of\ values]$ containing two values (both table names). This happens if and only if both R and S contain t

Hadoop Internals

We will analyse how Hadoop works in its internal mechanism. Unfortunately they are not related to the last version of Hadoop (so it is not a fundamental topic).

Terminology

Just for sake of comprehension, we recap some terminology:

- **job**: execution of a MapReduce application across a data set
- **task**: execution of a Mapper or a Reducer on a split of data
- **task attempt**: attempt to execute a task

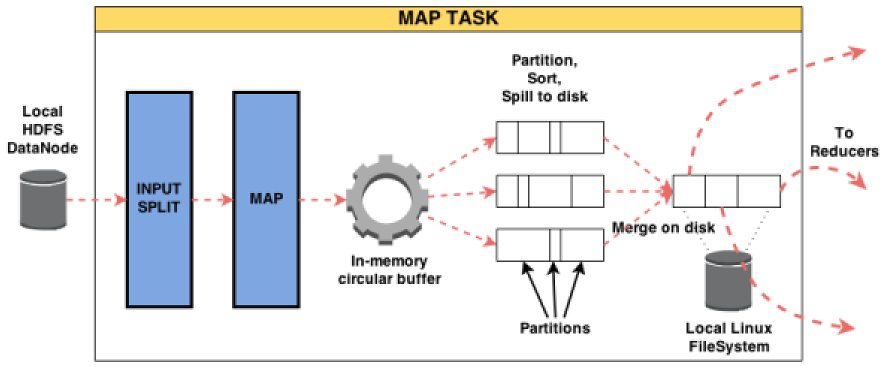
Example:

> Considering the running of a Word Count with 20 different splits. In this situation there are:
>
> - 1 job
> - 20 map tasks (one for each input split)
> - a user specified number of reduce tasks
> - at least 20 mapper tasks + number of reducers tasks attempts will be performed (more if machine crashes)
>
> Each task is attempted (**task attempts**) at least a maximum number of times (the maximum number of attempts per task is a parameter of the cluster configuration, often 5 times). If there is a temporary fault, the execution of each task may initially fail but it succeeds in the following attempts.
>
> Multiple attempts may occur in parallel (a.k.a. **speculative execution**). If there is enough available resources (i.e., there are processors in the idle state and enough main memory to run new tasks) Hadoop can duplicate a task and execute each "copy" of the task in a different node of the cluster (containing the input split).

Anatomy of a MapReduce Job Run

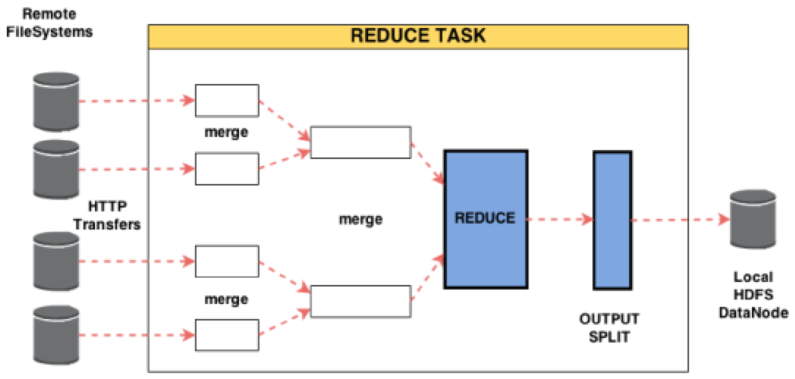
The user works on a **Client machine**; try to submit a job means to send a message to the **JobTracker machine** (only one for all the configuration) that creates the job and decide with node will execute the application. The *JobTracker* requires some job information from the client: the folder containing all input data, and the size in order to decide how many Mappers to instantiate.
The *JobTracker* will create a task for each Mapper and a task for each Reducer by sending a message to the **TaskTracer machine** (there could be many for all the configuration) that will creates one virtual machine for each Mapper/Reduce task.
If everything work properly, at the end each *TaskTracker* machine will send the result to the *JobTracker* machine that forward it to the client. But a task fails, the *TaskTracker* tries to run it again on the same node, and if the problem persist, it will try to execute it on another node.
The drawback of this configuration is that the *JobTracker* is a single point of failure.

The Mapper process the key-value pairs emitting a set of key-value pairs that will be inserted in a circular buffer (in node main memory). Each key-value pair is extracted and sent to different queues according to the key content, sort locally and then merge al together. The temporary result is stored on the local disk and then sent to the Reducers (through the network).

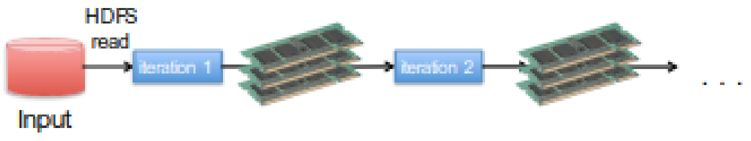


2. Reducer side

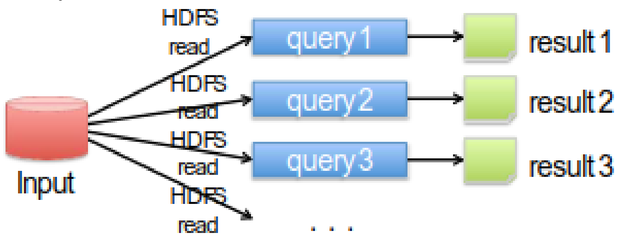
In the Reducer task, the system receives data from many remote file systems (from the node performing Map tasks). A single Reducer is related so many Mappers, so the input of the reduce method is obtained by sequential merging (often in the local disk because of the dimension). Then the reduce method is invoked on each (*key*; [*list of values*]) that generate the final result to be stored in the distributed file system (HDFS). In particular one copy is stored in the same node running the Reducer (to reduce network traffic) but the other two copies will be sent on the network (inside the HDFS).



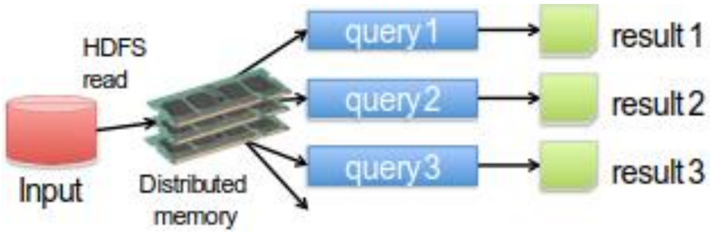
stored in the main memory of several nodes (each chunk in one node). The total amount of main memory in the cluster is bigger enough to store all the result.
Using the main memory the performance increases up to a scale of 10.



Also for multiple analyses of the same data are improved in Spark, with respect to Hadoop.
In Hadoop al the parallel analysis takes data from the HDFS itself:



while Spark performs only one HDFS read and store data in the distributed memory:



Spark is preferred with respect to Hadoop because more performing and each job performed with Hadoop can be done with Spark.

Spark exploits **RDD (Resilient Distributed Dataset)**, a distributed collection of objects spread across the nodes of a cluster, in particular in their main memory.
Spark programs are written in terms of operations on resilient distributed data sets. Datasets are based on RDDs.
The RDDs are built and manipulated through a set of parallel:

- transformation: map, filter, join, …
- actions: count, collect, save, …

If a failure happens, the developer has no to take care of it because the Spark framework will automatically rebuilt the job and perform it again.

The Spark computing framework provides a programming abstraction (based on RDDs) and transparent mechanism to execute in parallel RDDs.

- hides complexities of fault-tolerance and slow machines
- manages scheduling and synchronization of the jobs

The used data types are only RDDs. The Spark companies suggest to represent data in the following ways:

- relational data: use dataset
- unstructured data: use RDD

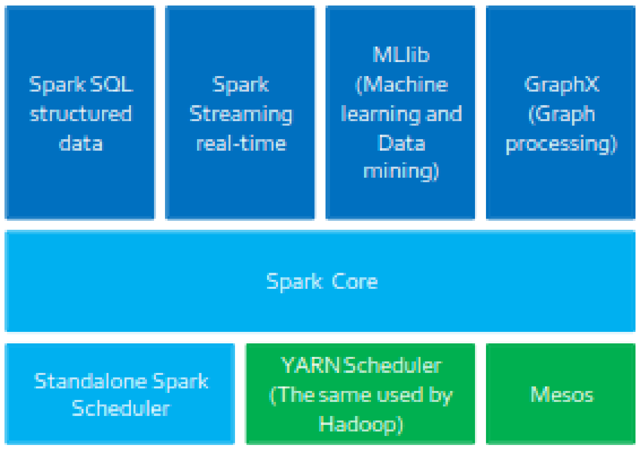
Main components

Spark is based on many components that exploits the same Spark core. All those component provide some specific APIs to perform specific tasks.
The main components are the following:

- **Spark SQL**: used for structured data
- **Spark Streaming**: used for real-time analysis
- **MLib**: used for machine learning and data mining features
- **GraphX**: used to analyse and process graphs

The most important part is the **Spark core** that allows to write complete programs (only not specific for a single environment).
The Spark core is based on the schedulers:

- **standalone spark scheduler**
- **YarN scheduler**
- **Mesos**


Spark SQL structured data
Spark Streaming real-time
MLlib (Machine learning and Data mining)
GraphX (Graph processing)
Spark Core
Standalone Spark Scheduler
YARN Scheduler (The same used by Hadoop)
Mesos

Spark is not characterized by its own distributed file system, but it exploits any (also HDFS).

Spark basic concepts

The RDDs are the primary abstraction in Spark: they are a distributed collection of objects spread across the nodes of clusters. RDD is the main data type, each RDDs is split in partitions and stored in the main memory of the executors running in the nodes.
Having more partitions means having more parallelism.

Example:

> Consider a RDD split in 3 different partitions:

Spark programs

Spark support many programming languages:

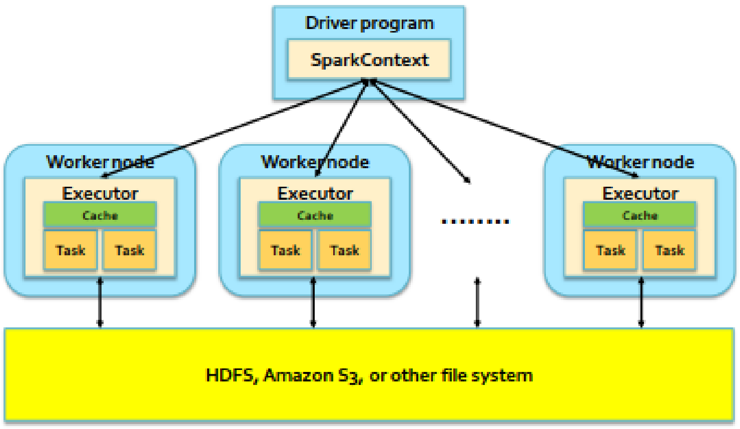
- Scala
- Java
- Phyton
- R

The **Scala programming language** is the same language that is used to develop the Spark framework and all its components (Spark Core, Spark SQL, Spark Streaming, MLlib, GraphX).

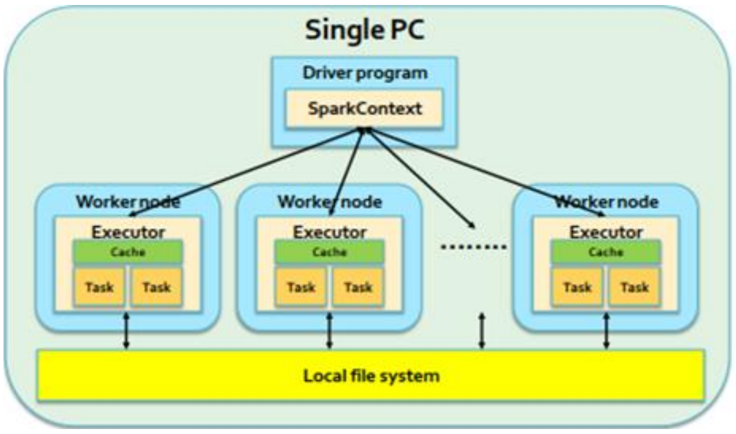
Spark programs consists only in a Driver class that contains the main method and defines the workflow of the application. The Driver accesses Spark through the SparkContext object.
The Driver defines:

- local variables
- RDDs
- SparkContext object

The **worker nodes** of the cluster are used to run the application by means of the **executors**. Each executors runs on its partition of the RDD(s) the operations that are specified in the driver.



Differently from Hadoop, Spark can be executed locally on a simple laptop. Each node is simulated with a different thread that implement the parallelization.
Scala, like Java, runs on a virtual machine, so it is easier to implement a Spark job on a single machine.



```
            JavaRDD<String> lines = sc.textFile(inputFile);

            // Count the number of lines in the input file
            // Store the returned value in the local variable numLines
            numLines = lines.count();

            // Print the output in the standard output (stdout)
            System.out.println("Number of lines="+numLines);

            // Close the Spark Context object
            sc.close();
      }
}
```

An action returns a local variable. There cannot be an `OutOfMemory` problem for an RDD.
In the same script can coexist local variables (local) and RDDs (distributed).

2. Word count

The ***word count* problem** is the simplest task in Big Data environment: consists in simply counting the occurrences of each word in the input file.

In Spark there are two different type of data that the developer can exploits:

- local variables: used to store small objects/data
- RDDs: used to store big/large objects/data

The implementation of the Driver using lambda functions:

```
import java.util.Arrays;
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import scala.Tuple2;

public class SparkWordCount {
    @SuppressWarnings("serial")
    public static void main(String[] args) {

            String inputFile = args[0];
            String outputPath = args[1];

            // Create a configuration object and set the name of the application
            SparkConf conf = new SparkConf().setAppName("Spark Word Count");

            // Create a Spark Context object
            JavaSparkContext sc = new JavaSparkContext(conf);

            //Build an RDD of Strings from the input textual file
            // Each element of the RDD is a line of the input file
            JavaRDD<String> lines = sc.textFile(inputFile);

            // Split/transform the content of lines in a list of words an store in the
            words RDD
            JavaRDD<String> words =
                        lines.flatMap(line ->
                              Arrays.asList(line.split("\\s+")).iterator());
                     }
                 });

     //Map/transform each word in the words RDD to a pair (word,1) an store the
     result in the words_one RDD
     JavaPairRDD<String, Integer> words_one =
                 words.mapToPair(
                         new PairFunction<String, String, Integer>() {
                             @Override
                             public Tuple2<String, Integer> call(
                                 String word) {
                                      return new Tuple2<String, Integer>(
                                              word.toLowerCase(), 1);
                             }
                         });

     //Count the num. of occurrences of each word.
     // Reduce by key the pairs of the words_one RDD and store the result (the
     list of pairs (word, num. of occurrences) in the counts RDD
     JavaPairRDD<String, Integer> counts =
                 words_one.reduceByKey(
                         new Function2<Integer, Integer, Integer {
                             @ Override
                             public Integer call(Integer c1, Integer c2){
                                 return c1 + c2;
                             }
                         });

     // Store the result in the output folder
     counts.saveAsTextFile(outputPath);

     // Close the Spark Context object
     sc.close();
 }
}
```

Before to the Java8.0 version there was not lambda function, so anonymous classes should be exploited. They require way more code, but are easier than the lambda functions.

Unnamed classes are classes created on the fly without specifying their name. It is necessary to specify the abstract interface that it implements. Those interfaces have just one abstract method.

The developer can manually set the number of partitions. It is useful when reading file from the local file system. To do so, the following method (from the `JavaSparkContext` class) is exploited:

```
textFile(String inputPath, int numPartitions);
```

Example:

Create RDDs from files, by specifying as 5 the desired number of partitions:

```
//Build an RDD of Strings from a local input textual file.
// The number of partitions is manually set to 5
// Each element of the RDD is a line of the input file
JavaRDD<String> lines = sc.textFile(inputFile, 5);
```

Parallelize local object collection

Given a local collection/list of local Java object, it is possible to transform them into RDDs by exploiting the following two methods (from the `JavaSparkContext` class):

```
parallelize(List<T> list)
parallelize(List<T> list, int numPartitions)
```

Without specifying manually the number of partitions, the system automatically decide the number of splits.

Example:

Create RDDs from a local Java collection:

```
//Create a local Java list
List<String> inputList = Arrays.asList("First element", "Second element",
                                        "Third element");

// Build an RDD of Strings from the local list.
// The number of partitions is set automatically by Spark
// There is one element of the RDD for each element of the local list
JavaRDD<String> distList = sc.parallelize(inputList);
```

No computation occurs when `sc.parallelize()` is invoked. Spark only records how to create the RDD and then the data is lazily read from the input file only when the data is needed (i.e., when an action is applied on the lines RDD, or on one of its "descendant" RDDs).

In order to create exactly 3 partitions:

```
//Build an RDD of Strings from the local list.
// The number of partitions is set to 3
// There is one element of the RDD for each element of the local list
JavaRDD<String> distList = sc.parallelize(inputList,3 );
```

Save RDDs

Save in the file system

To store the content of the RDDs in the distributed file system the following method (from the `JavaRDD<T>` class) is exploited:

```
saveAsTextFile(String outputPath);
```

It is necessary to specify the wanted path. The system automatically decide if data should be read/write on the local file system or in the distributed one.
Again, in the output file, there is a line for each RDD element.

potentially change the order of some transformations or merge some of them based on its optimization engine.

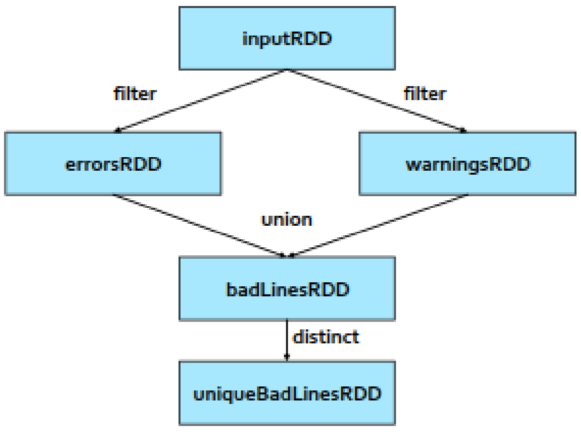
The actions return results to the Driver program (i.e. local variables), or in the storage. It is important to check if the returned result is too large to be stored in the main memory.

Example:

Given the following script.

```
public static void main(String[] args) {
    // Initialization of the application
    ...

    // Read the content of a log file
    JavaRDD<String> inputRDD = sc.textFile("log.txt");

    // Select the rows containing the word "error"
    JavaRDD<String> errorsRDD = inputRDD.filter(line -> line.contains("error"));

    // Select the rows containing the word "warning"
    JavaRDD<String> warningRDD = inputRDD.filter(line ->
                                     line.contains("warning"));

    // Union errorsRDD and warningRDD
    // The result is associated with a new RDD: badLinesRDD
    JavaRDD<String> badLinesRDD = errorsRDD.union(warningRDD);

    // Remove duplicates lines (i.e., those lines containing both "error" and
    "warning")
    JavaRDD<String> uniqueBadLinesRDD = badLinesRDD.distinct();

    // Count the number of bad lines by applying the count() action
    long numBadLines = uniqueBadLinesRDD.count();

    // Print the result on the standard output of the driver
    System.out.println("Lines with problems:" + numBadLines);
    ...
}
```

The related *lineage graph* is the following:



That graph can be improved by removing `distinct()` and `union()` (expensive operations) and performing a filtering instead.

- R: Boolean

There are different solution (three).

Named class:

The more standard solution is to define a class, implementing the given interface.

```
//Define a class implementing the Function interface
class ContainsError implements Function<String, Boolean> {
    // Implement the call method
    public Boolean call(String x) {
        return x.contains("error");
    }
}
...
// Read the content of a log file
JavaRDD<String> inputRDD = sc.textFile("log.txt");

// Select the rows containing the word "error"
JavaRDD<String> errorsRDD = inputRDD.filter(new ContainsError());
```

When `call()` is invoked, this method analyzes the value of the parameter `X` and returns true if the string `x` contains the substring “error”. Otherwise, it returns false.
The `filter()` transformation selects the elements of the `inputRDD` satisfying the constraint specified in the call method of the `ContainsError` class.
An object of the `ContainsError` is instantiated and its call method is applied on each element `x` of `inputRDD`. If the function returns true, then `x` is stored in the new errorsRDD RDD, otherwise `x` is discarded.

Anonymous class:

It is also possible to define inline a class, without specifying its name (this is why its anonymous) that has a limited scope and can be used only where it is defined:

```
//Read the content of a log file
JavaRDD<String> inputRDD = sc.textFile("log.txt");

// Select the rows containing the word “error”
JavaRDD<String> errorsRDD = inputRDD.filter(
        new Function<String, Boolean>() {
                public Boolean call(String x) {
                        return x.contains("error");
                }
        });
```

The `new Function<>(){}` defines on the fly a temporary anonymous class implementing the `Function<String, Boolean>` interface. An object of this class is instantiated and its call method is applied on each element `x` of `inputRDD`. If the call method returns true then `x` is “stored” in the new `errorsRDD` RDD, otherwise `x` is discarded.
The anonymous class itself is equal to the content of the `ContainErrors` class defined in the previous solution based on named classes.

We are working with RDDs of Strings, hence, the data type T of the Function<T, Boolean> interface we are implementing is String and also each element on which the lambda function is applied is a String.

Example:

Create an RDD of integers containing the values {1, 2, 3, 3} and then create a new RDD containing only the values greater than 2:

```
// Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Select the values greater than 2
JavaRDD<Integer> greaterRDD = inputRDD.filter(element -> {
    if (element>2)
        return true;
    else
        return false;
    });
```

We are working with Integers, hence the data type T of the Function<T, Boolean> interface we are implementing is Integer and also the elements we are analysing by means of the lambda functions are Integer values.

b. Map transformation

This method is different form the one exploited in the MapReduce applications. To elaborate a $(key, value)$ pair we should use FlatMap transformation.
The Spark ***map* transformation** is a one-to-one relation: for each input element x there could be only one output element $y = f(x)$. The function $f(\cdot)$ is defined by the user.
The exploited method (from the JavaRDD<T> class) is the following:

```
JavaRDD<R> map(Function<T, R>)
```

Example:

Create an RDD from a textual file containing the surnames of a list of users. Then create a new RDD containing the length of each surname:

```
// Read the content of the input textual file
JavaRDD<String> inputRDD = sc.textFile("usernames.txt");

// Compute the lengths of the input surnames
JavaRDD<Integer> lenghtsRDD = inputRDD.map(element -> new Integer(element.length());
```

The element parameter, of the lambda function, is a String (as the inputRDD type), while the lambda function itself should return an Integer (as the legthsRDD type).

Example:

Create an RDD of integers containing the values {1, 2, 3, 3} and then create a new RDD containing the square of each input element:

```
//Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);
```

```
//Read the content of a textual input file
JavaRDD<String> inputRDD = sc.textFile("names.txt");
// Select the distinct names occurring in inputRDD
JavaRDD<String> distinctNamesRDD = inputRDD.distinct();
```

The distinct transformation is not efficient because it should send all the occurrences through the network, in the shuffle operation. All the copy to the same element should be sent to the same node, in order to discard duplicates. This implies an huge network usage because all input data should be sent on the network.
In general, using the shuffling operation, could decrease the performance of the task. So distinct should be used only if really needed.

e. Sample transformation

The ***sample* transformation** selects only a subset of the data. The developer should specify if it is possible to have duplicates in the final outcome and the fraction of input size we would like to take as sample.
The samples are selected randomly in the input RDD.
The used method (from the `JavaRDD<T>` class) is the following:

```
JavaRDD<T> sample(boolean withReplacement, double fraction);
```

Example:

Create an RDD from a file that contains a sentence for each line and then create a new RDD containing a random sample of sentences (using the “without replacement” strategy and a fraction of 20%):

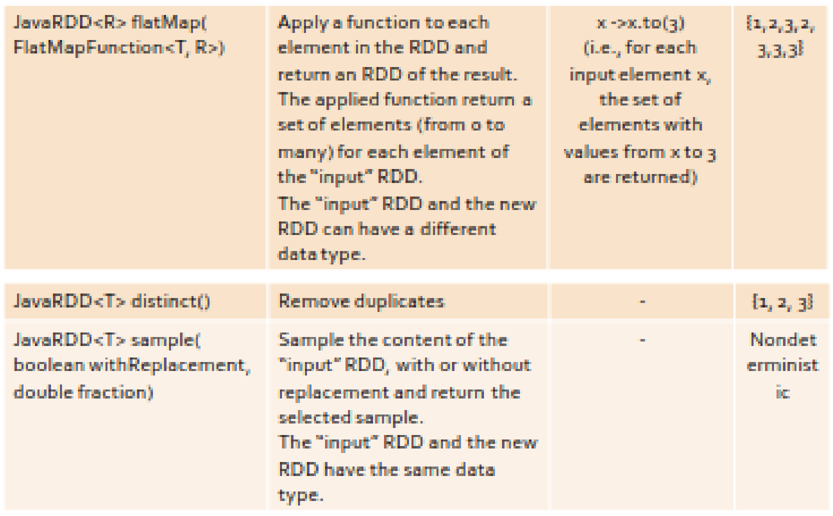
```
//Read the content of a textual input file
JavaRDD<String> inputRDD = sc.textFile("sentences.txt");

// Create a random sample of sentences
JavaRDD<String> randomSentencesRDD = inputRDD.sample(false, 0.2);
```

f. Set transformation

Those transformation are applied on a pair of RDDs. Some of them implement standard **set transformations**:

- union
- intersection
- subtract
- cartesian

All of them have two input RDDs (on one the method is applied, the second is passed as parameter) and only one output RDD.
All those transformation have the same data type in input and in output, except for the Cartesian one that allow mixed data types.



JavaRDD<R> flatMap(FlatMapFunction<T, R>)	Apply a function to each element in the RDD and return an RDD of the result. The applied function return a set of elements (from 0 to many) for each element of the "input" RDD. The "input" RDD and the new RDD can have a different data type.	x ->x.to(3) (i.e., for each input element x, the set of elements with values from x to 3 are returned)	{1, 2, 3, 2, 3, 3, 3}
JavaRDD<T> distinct()	Remove duplicates	-	{1, 2, 3}
JavaRDD<T> sample(boolean withReplacement, double fraction)	Sample the content of the "input" RDD, with or without replacement and return the selected sample. The "input" RDD and the new RDD have the same data type.	-	Nondeterministic

2. Basic actions

The Spark actions can return a local Java variable, or store the RDD content in an output file or a database table.

a. Collect action

The ***collect* action** returns a local Java list of object containing the same objects of the considered RDD.
Be careful about the size of the RDD: large amount of data cannot be memorized in a local variable of the Driver.
The used method (from the `JavaRDD<T>` class) is the following:

```
List<T> collect()
```

Example:

Create an RDD of integers containing the values {1, 2, 3, 3} and retrieve the values of the created RDD and store them in a local Java list that is instantiated in the Driver:

```
//Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputList = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

// Retrieve the elements of the inputRDD and store them in a local Java list
List<Integer> retrievedValues = inputRDD.collect();
```

It is really important to check if the whole RDD content could fit in main memory, because `retrievedValues` is a Java local variable (hence stored in RAM).

b. Count action

The ***count* action** simply count the number of elements of an RDD. There could never happen an out of memory problem because the output data is a single value. There is no shuffling operation, because only the local amount of elements is sent on the network.
The used method (from the `JavaRDD<T>` class) is the following:

```
long count();
```

The used method (from the `JavaRDD<T>` class) is the following:

```
List<T> take(int n);
```

This action could be used to perform the top-*k* pattern, but the list should be sorted before to take the elements.

Example:

Create an RDD of integers containing the values {1, 5, 3, 3, 2} and then retrieve the first two values of the created RDD and store them in a local Java list that is instantiated in the Driver:

```
//Create an RDD of integers. Load the values 1, 5, 3, 3, 2 in this RDD
List<Integer> inputList = Arrays.asList(1, 5, 3, 3, 2);
JavaRDD<Integer> inputRDD = sc.parallelize(inputList);

//Retrieve the first two elements of inputRDD and store them in a local Java list
List<Integer> retrievedValues = inputRDD.take(2);
```

e. First action

The ***first* action** is similar to the `take(1)` action: it returns the first element in the RDD. The differences between take and first actions is that the second one returns a single object, not a collection.

The used method (from the `JavaRDD<T>` class) is the following:

```
T first();
```

First vs take(1) actions: the only difference between `first()` and `take(1)` is given by the fact that

- `first()`: returns a single element of type `T`
- `take(1)`: returns a list of elements containing one single element of type `T`

f. Top action

The ***top* action** returns a local Java list of objects containing the top *n* (largest) elements of the considered RDD. The used ordering criteria is the default one for class `T`, otherwise the descending order is used.

The used method (from the `JavaRDD<T>` class) is the following:

```
List<T> top(int n);
```

g. TakeOrdered action

The ***take ordered* action** returns a local list of objects containing the top *n* (smallest) elements of the considered RDD. The ordering is specified by the developer by means of the class, passed as parameter, implementing the `java.util.Comparator<T>` interface.

The used method (from the `JavaRDD<T>` class) is the following:

```
List<T> takeOrdered (int n, java.util.Comparator<T> comp);
```

The parameter of the `reduce()` method can be exploited directly using lambda expression (inside the parenthesis) to avoid write an whole class.

Example:

> Create an RDD of integers containing the values {1, 2, 3, 3}, then compute the sum of the values occurring in the RDD and “store” the result in a local Java integer variable in the Driver:
> ```
> //Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
> List<Integer> inputListReduce = Arrays.asList(1, 2, 3, 3);
> JavaRDD<Integer> inputRDDReduce = sc.parallelize(inputListReduce);
>
> // Compute the sum of the values
> Integer sum= inputRDDReduce.reduce(
>         (element1, element2) -> element1+element2);
> ```

Example:

> Create an RDD of integers containing the values {1, 2, 3, 3}, then compute the maximum value occurring in the RDD and “store” the result in a local Java integer variable in the Driver:
> ```
> //Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
> List<Integer> inputListReduce = Arrays.asList(1, 2, 3, 3);
> JavaRDD<Integer> inputRDDReduce = sc.parallelize(inputListReduce);
>
> // Compute the maximum value
> Integer max = inputRDDReduce.reduce(
>         (element1, element2) -> {
>                 if (element1>element2)
>                         return element1;
>                 else
>                         return element2;
>         });
> ```

j. Fold action

The ***fold* action** is similar to the *reduce action*, the only difference is that the given function could also be **not commutative (but still associative)**. So the fold action recursively merge together all the elements (using the function specified as parameter) until it retrieve a single object.
An initial value (initial zero) should be provide together with a lambda function implementing the call method of the `Function2<T,T,T>` interface.
The used method (from the `JavaRDD<T>` class) is the following:

> ```
> T fold(T zeroValue, Function2<T, T, T> f);
> ```

Fold vs reduce: differently from the *reduce action*, the *fold action* is characterized by a “zero” value and could be used to parallelize functions that are associative, but non-commutative (e.g. concatenation of a list of strings). The given function should be at least associative, otherwise the result depends on how the RDD is partitioned.

Example:

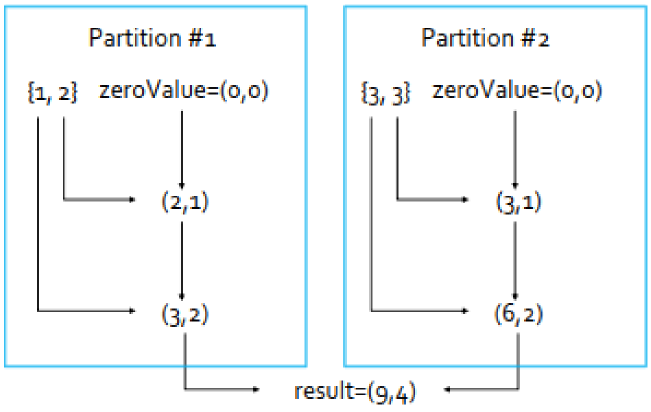
> The concatenation of string has the following properties.
> It is NOT commutative:

```
    public int numElements;

    public SumCount(int sum, int numElements) {
        this.sum = sum;
        this.numElements = numElements;
    }
    public double avg() {
        return sum/ (double) numElements;
    }
}
```

The usage of those class is the following:

```
//Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
List<Integer> inputListAggr = Arrays.asList(1, 2, 3, 3);
JavaRDD<Integer> inputRDDAggr = sc.parallelize(inputListAggr);

//Instantiate the zero value
SumCount zeroValue = new SumCount(0, 0);

// Compute sum and number over the elements of inputRDDAggr
SumCount result = inputRDDAggr.aggregate(zeroValue,
        (a, e) -> {
                a.sum = a.sum + e;
                a.numElements = a.numElements + 1;
                return a;
        },
        (a1, a2) -> {
                a1.sum = a1. sum + a2.sum;
                a1.numElements = a1.numElements + a2.numElements;
                return a1;
        });
```

The graphical simulation of the example is the following:



Key-value pair RDDs

The **pair RDDs** are the Spark support to store key-value pairs. This data format was usually used in MapReduce applications.
Pair RDDs are characterized by specific operations (`reduceByKey()`, `join()`, etc...) but also by the standard operations for RDDs (`filter()`, `map()`, `reduce()`, etc...).

Pair RDDs allow:

- grouping data according to similar keys
- computing computation by key

The basic idea is similar to the MapReduce-based programs, but Spark provide more operations.

Creating Pair RDDs

Similarly to standard RDDs, also Pair RDDs are lazily created and evaluated (when an action occurs).
Pair RDDs can be created from:

- standard RDDs:
  - applying the `mapToPair()` transformation
  - applying other specific transformations
- Java in-memory collection:
  - Using the `parallelizePairs()` method of the `SparkContext` class

In Java there is not a specific class to represent pairs, so it exploits the **`Tuple2<>` Scala class**:

```
scala.Tuple2<K,V>
```

to represent tuples containing two values. In this case the first column is the key, while the second is the actual value of the pair.
To instance a new object, we can invoke the constructor:

```
new Tuple2(key, value)
```

and to retrieve the content, there are two specific methods:

- first value (key): `tuple._1()`
- second value (value): `tuple._2()`

There are not the methods `getKey()` or `getValue()` because Tuple2 was not designed specifically to handle key-value pairs. We semantically map the key to the first value and the value to the second one.

The ***map to pair* transformation** is used to create a new `PairRDD` by applying it on each single element of the input standard RDD.
For each x input element, the $y = f(x)$ tuple is inserted in the new PairRDD (where $f(\cdot)$ is defined by the user).
The signature of the method (of the `JavaRDD<T>` class) is the following:

```
JavaPairRDD<K,V> mapToPair(PairFunction<T,K,V> function)
```

a. ReduceByKey transformation

The ***reduce by key* transformation** is similar to the `reduce()` action, but they have the following differences:

- reduceByKey is a transformation while Reduce is an action
- reduce action returns a single value, while reduceByKey transformation returns a pair for each key

The user provided function must be both associative and commutative, otherwise the result would be partition-dependent.
The data type of the new PairRDD is the same of the input PairRDD, because the reduction is just about merging, not changing.
The used method (of the `JavaPairRDD<K,V>` class) is the following:

```
JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> f)
```

The `f` parameter should implement the `Function2<V, V, V>` interface and so define the following method:

```
public V call(V element1, V element2)
```

Example:

Given a PairRDD containing the pair $(name, age)$, associate each name with the age of the youngest user with that name.

```
// Given the PairRDD composed by (name,age)
JavaPairRDD<String, Integer> nameAgeRDD = ...;


// Select for each name the lowest age value
JavaPairRDD<String, Integer> youngestPairRDD =
    nameAgeRDD.reduceByKey(
                (age1, age2) -> {
                        if (age1<age2)
                                return age1;
                        else
                                return age2;
                });
```

The ReduceByKey transformation is useful also when we want to count the number of element, in a set, associated with a specific key. This transformation is used to solve the WordCount problem.

b. FoldByKey transformation

The ***fold by key* transformation** is similar to the ReduceByKey one, but there are some slight differences:

- foldByKey is characterized by a zero value
- the function can be just associative (not commutative)

The zero value is needed only when a set can be empty, but in BigData application it is difficult to have empty sets, so FoldByKey is rarely used (ReduceByKey is preferred).

The used method (of the `JavaPairRDD<K,V>` class) is the following:

```
JavaPairRDD<K,V> foldByKey(V zeroValue, Function2<V,V,V> f);
```

```
    }
}
```

Afterwards we exploit it as follows:

```
JavaPairRDD<String, AvgCount> avgAgePerNamePairRDD = nameAgeRDD.combineByKey(
        inputElement -> new AvgCount(inputElement, 1),

        (intermediateElement, inputElement) -> {
              AvgCount combine = new AvgCount(inputElement, 1);
              combine.total = combine.total + intermediateElement.total;
              combine.numValues = combine.numValues +
                          intermediateElement.numValues;
              return combine;
        },

        (intermediateElement1, intermediateElement2) -> {
              AvgCount combine = new AvgCount(intermediateElement1.total,
                          intermediateElement1.numValues);
              combine.total = combine.total + intermediateElement2.total;
              combine.numValues = combine.numValues +
                          intermediateElement2.numValues;
              return combine;
        });
avgAgePerNamePairRDD.saveAsTextFile(outputPath);
```

The three main parameters are the following:

- `createCombiner()`: given an `Integer`, it returns an `AvgCount` object
- `mergeValue()`: given an `Integer` and an `AvgCount` object, it combines them and returns an `AvgCount` object
- `mergeCombiner()`: given two `AvgCount` objects, it combines them and returns an `AvgCount` object

d. GroupByKey transformation

The ***group by key* transformation** conceptually returns the list of all the values related to the same key. Actually, for each input tuple, it is created a key-value pair where the value is an `Iterable<>` object over the whole list of values.

The used method (of the `JavaPairRDD<K,V>` class) is the following:

```
JavaPairRDD<K,Iterable<V>> groupByKey();
```

Example:

Given a PairRDD containing the pair $(name, age)$ (as in the previous example), create an output file containing one line for each name followed by the ages of all the users with that name.

```
// Create one group for each name with the associated ages
JavaPairRDD<String, Iterable<Integer>> agesPerNamePairRDD = nameAgeRDD.groupByKey();
// Store the result in a file
agesPerNamePairRDD.saveAsTextFile(outputPath);
```

In the new PairRDD each pair/tuple is composed by:

- key: a string
- value: a list of integers