

# Introduction to Spark

---

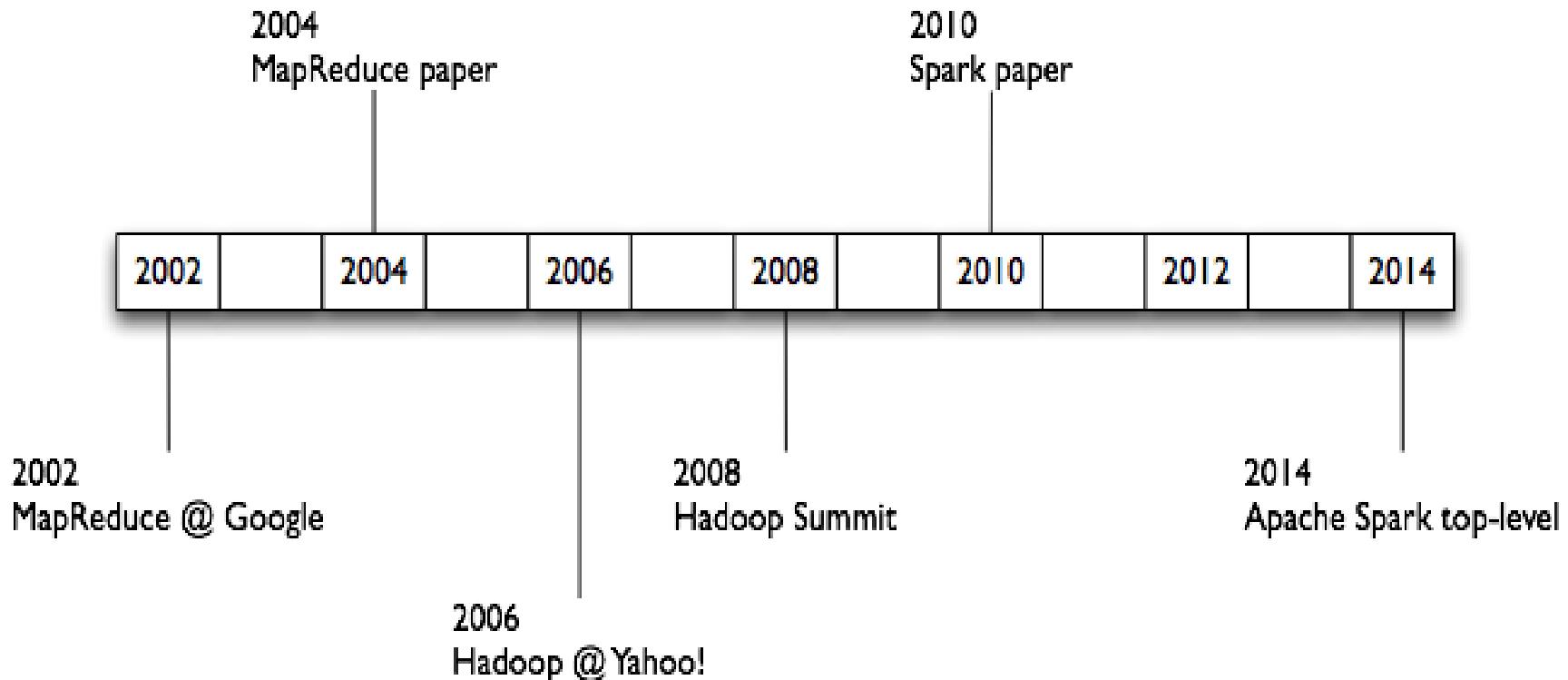
# Spark

---

- Apache Spark™ is a fast and general-purpose engine for large-scale data processing
- Spark aims at achieving the following goals in the Big data context
  - Generality: diverse workloads, operators, job sizes
  - Low latency: sub-second
  - Fault tolerance: faults are the norm, not the exception
  - Simplicity: often comes from generality

# Spark History

- Originally developed at the University of California - Berkeley's AMPLab

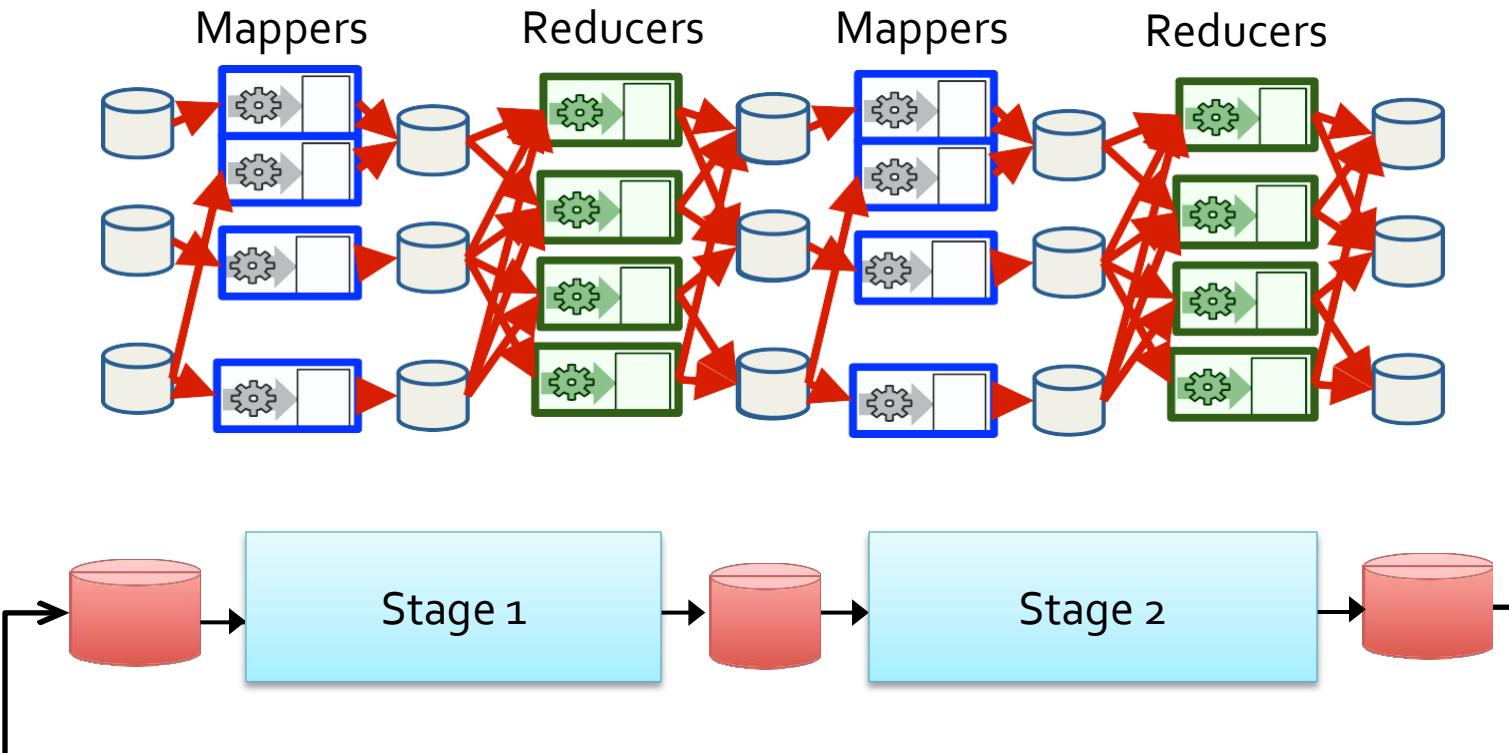


# **Spark: Motivations**

---

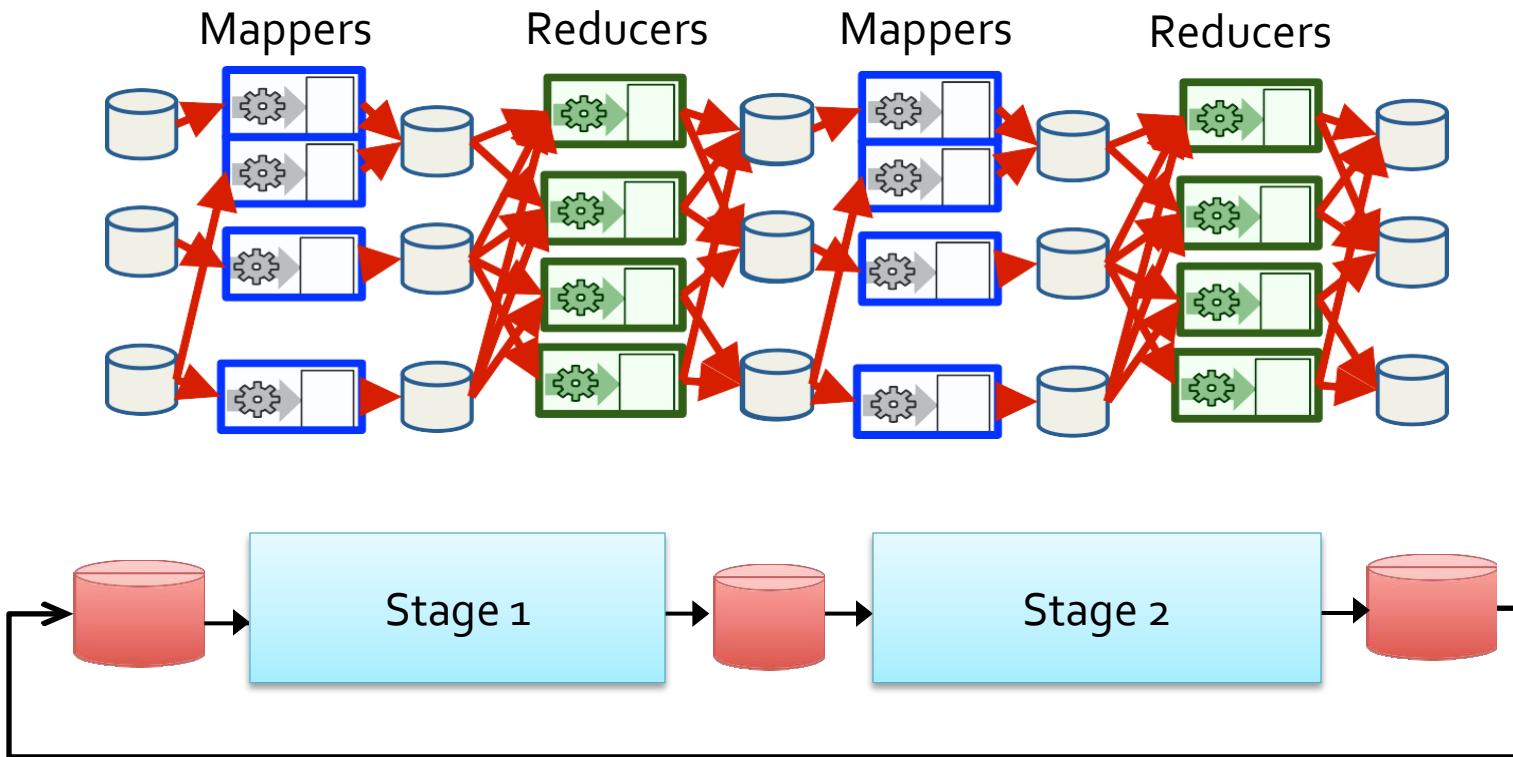
# MapReduce and Iterative Jobs

- Iterative jobs, with MapReduce, involve a lot of disk I/O for each iteration and stage



# MapReduce and Iterative Jobs

- Disk I/O is very slow (even if it is local I/O)



# Apache Spark: Motivation and Opportunity

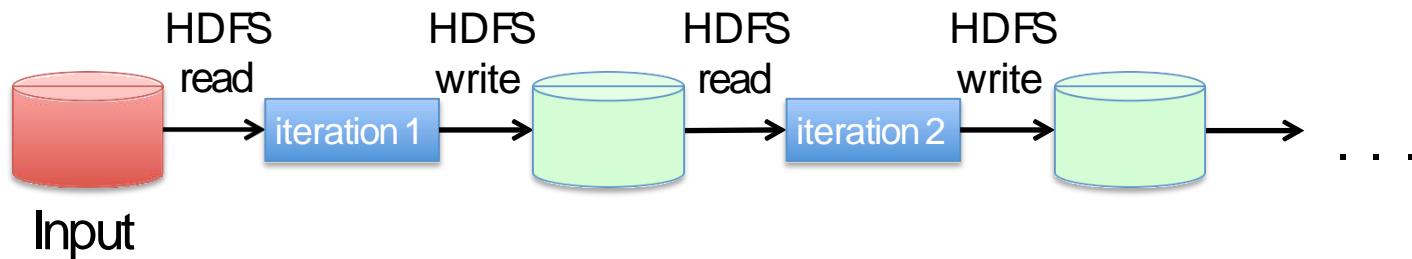
---

- Motivation
  - Using MapReduce for complex **iterative jobs** or **multiple jobs on the same data** involves lots of disk I/O
- Opportunity
  - The **cost of main memory decreased**
    - Hence, large main memories are available in each server
- Solution
  - Keep **more data in main memory**
    - Basic idea of Spark

# From MapReduce to Spark

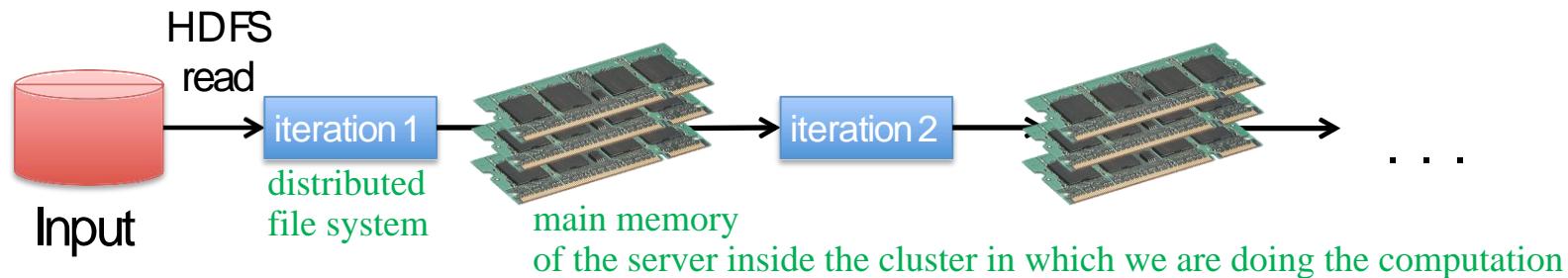
---

- MapReduce: Iterative job



# From MapReduce to Spark

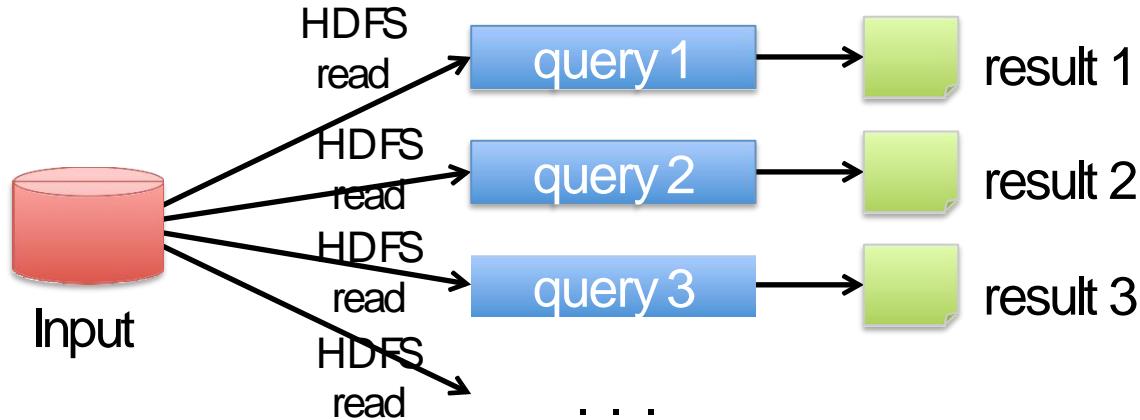
- Spark: Iterative job



- Data are shared between the iterations by using the main memory
  - Or at least part of them
- 10 to 100 times faster than disk

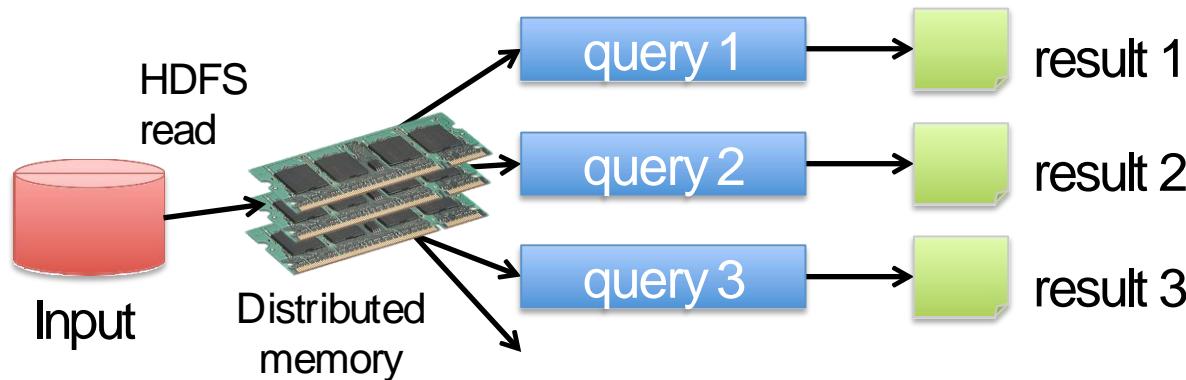
# From MapReduce to Spark

- MapReduce: Multiple analyses of the same data



# From MapReduce to Spark

- Spark: Multiple analyses of the same data



- Data are read only once from HDFS and stored in main memory
  - Split of the data across the main memory of each server

# Spark: Resilient Distributed Data sets (RDDs)

---

- Data are represented as Resilient Distributed Datasets (RDDs)
  - Partitioned/Distributed collections of objects spread across the nodes of a cluster
  - Stored in main memory (when it is possible) or on local disk
- Spark programs are written in terms of operations on resilient distributed data sets

# Spark: Resilient Distributed Data sets (RDDs)

---

- RDDs are built and manipulated through a set of parallel
  - Transformations
    - map, filter, join, ...
  - Actions
    - count, collect, save, ...
- RDDs are automatically rebuilt on machine failure

# Spark Computing Framework

---

- Provides a programming abstraction (based on RDDs) and transparent mechanisms to execute code in parallel on RDDs
  - Hides complexities of fault-tolerance and slow machines
  - Manages scheduling and synchronization of the jobs

# MapReduce vs Spark

	Hadoop Map Reduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc...
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, Python, and R

# MapReduce vs Spark

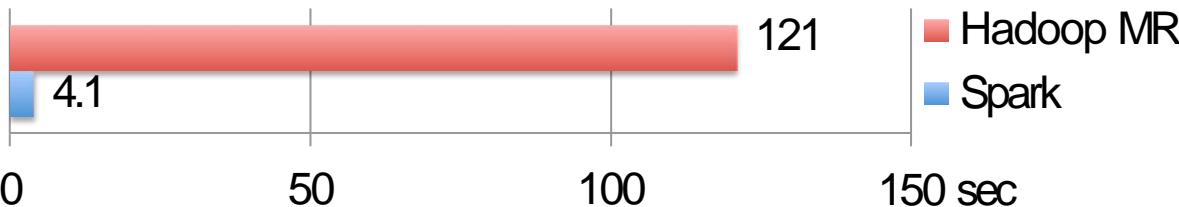
---

- Lower overhead for starting jobs
- Less expensive shuffles

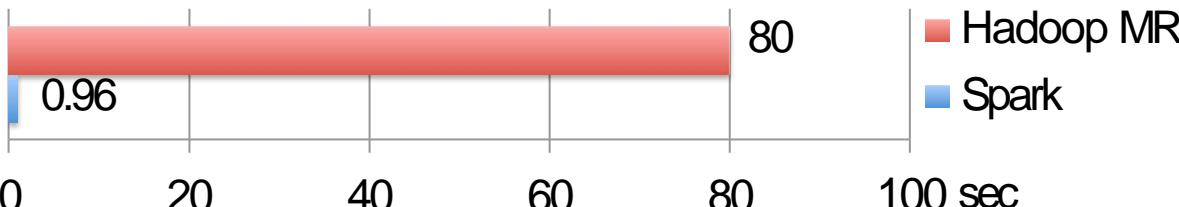
# In-Memory RDDs Can Make a Big Difference

- Two iterative Machine Learning algorithms:

- K-means Clustering



- Logistic Regression



# Petabyte Sort Challenge

	<b>Hadoop MR Record</b>	<b>Spark Record</b>	<b>Spark 1 PB</b>
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
<b>Sort rate</b>	<b>1.42 TB/min</b>	<b>4.27 TB/min</b>	<b>4.27 TB/min</b>
<b>Sort rate/node</b>	<b>0.67 GB/min</b>	<b>20.7 GB/min</b>	<b>22.5 GB/min</b>

[Daytona Gray](#)  
[100 TB](#) sort  
benchmark  
record (tied  
for 1<sup>st</sup> place)

# **Spark: Main components**

---

# Spark Components

---

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 Components

---

- Spark is based on a basic component (the **Spark Core component**) that is exploited by all the high-level data analytics components
  - This solution provides a more uniform and efficient solution with respect to Hadoop where many non-integrated tools are available
- When the efficiency of the core component is increased also the efficiency of the other high-level components increases

# Spark Components

---

- Spark Core
  - Contains the basic functionalities of Spark exploited by all components
    - Task scheduling
    - Memory management
    - Fault recovery
    - ...
  - Provides the APIs that are used to create RDDs and applies transformations and actions on them

# Spark Components

---

- **Spark SQL** structured data
  - This component is used to interact with structured **datasets** by means of the SQL language or specific querying APIs
    - Based on Datasets
  - It supports also
    - Hive Query Language (HQL)
  - It interacts with many data sources
    - Hive Tables, Parquet, Json, ..
  - It exploits a query optimizer engine

# Spark Components

---

- **Spark Streaming real-time**
  - It is used to process **live streams of data** in real-time
  - The APIs of the Streaming real-time components operated on RDDs and are similar to the ones used to process standard RDDs associated with “static” data sources

# Spark Components

---

- **MLlib**
  - It is a machine learning/data mining library
  - It can be used to apply the parallel versions of some machine learning/data mining algorithms
    - Data preprocessing and dimensional reduction
    - Classification algorithms
    - Clustering algorithms
    - Itemset mining
    - ....

# Spark Components

---

- **GraphX**
  - A graph processing library
  - Provides algorithms for manipulating graphs
    - Subgraph searching
    - PageRank
    - ....
  - The Python version is not available
- **GraphFrames**
  - A graph library based on DataFrames and Python

# Spark Schedulers

---

- Spark can exploit many schedulers to execute its applications
  - Hadoop YARN
    - Standard scheduler of Hadoop
  - Mesos cluster
    - Another popular scheduler
  - Standalone Spark Scheduler
    - A simple cluster scheduler included in Spark

# **Spark Basic Concepts**

---

# Resilient Distributed Data sets (RDDs)

---

- RDDs are the primary abstraction in Spark
- RDDs are distributed collections of objects spread across the nodes of a clusters
  - They are split in partitions
  - Each node of the cluster that is running an application contains at least one partition of the RDD(s) that is (are) defined in the application

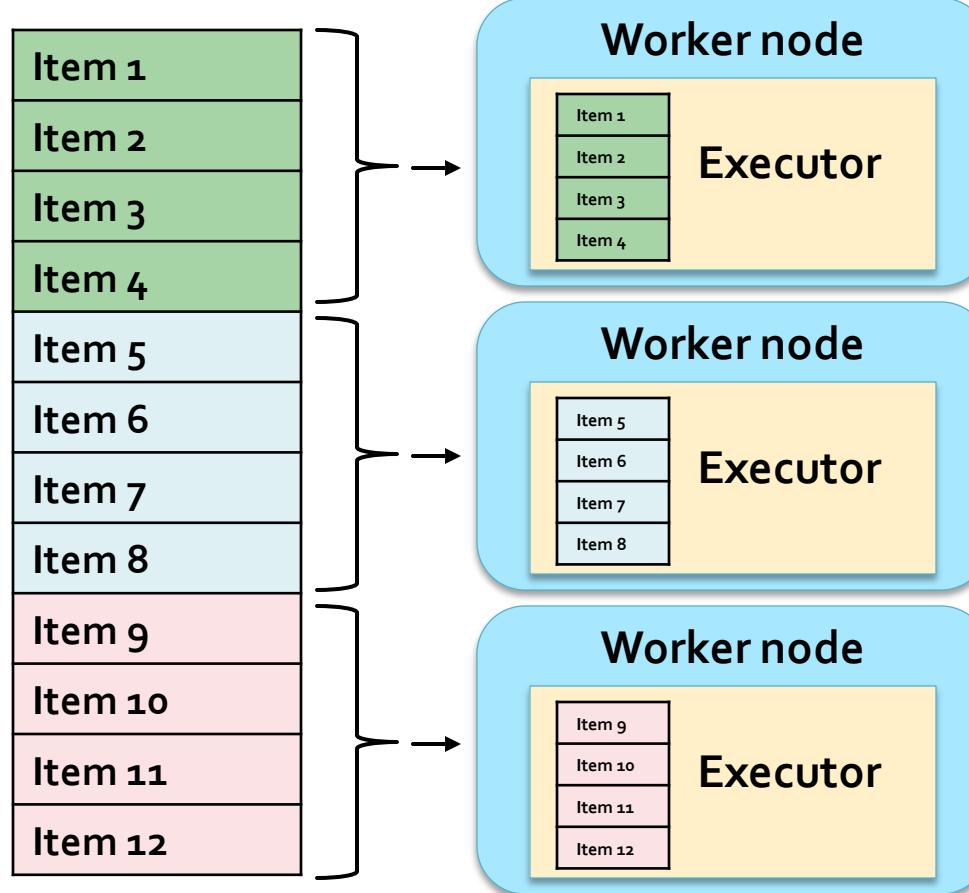
# Resilient Distributed Data sets (RDDs)

---

- RDDs
  - Are stored in the main memory of the executors running in the nodes of the cluster (when it is possible) or in the local disk of the nodes if there is not enough main memory
  - Allow executing in parallel the code invoked on them
    - Each executor of a worker node runs the specified code on its partition of the RDD

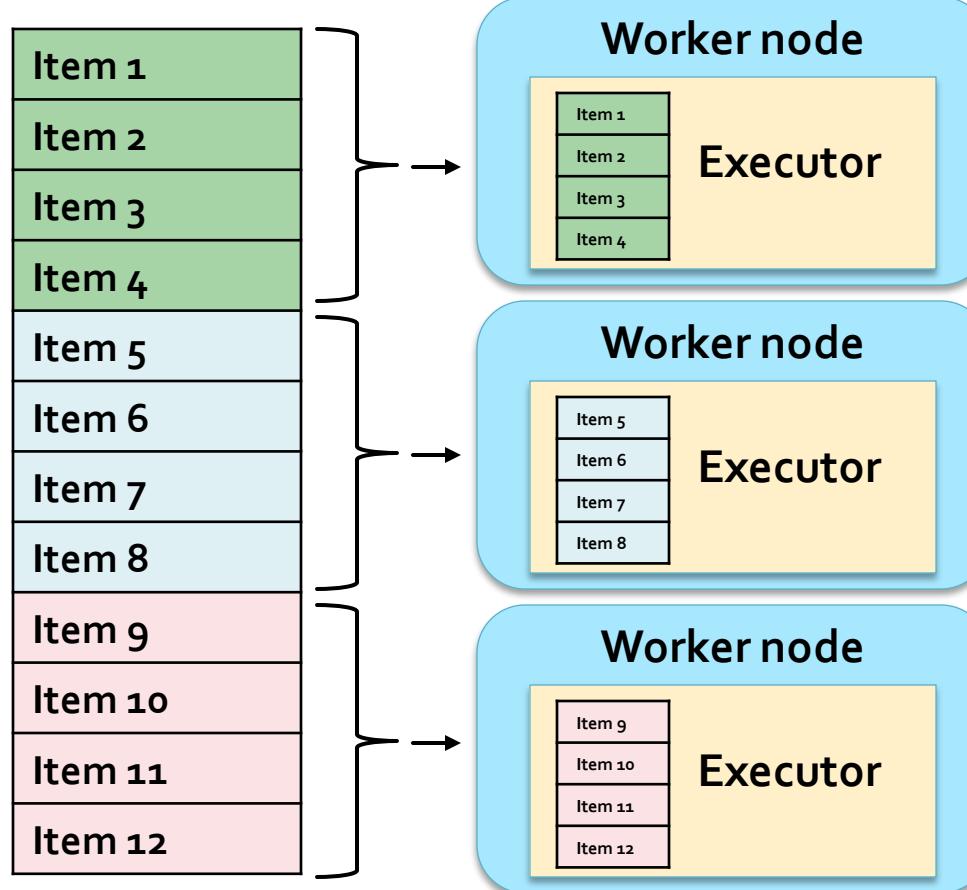
# Resilient Distributed Data sets (RDDs)

- Example of an RDD split in 3 partitions



# Resilient Distributed Data sets (RDDs)

- Example of an RDD split in 3 partitions



more partitions  
=  
more parallelism

# Resilient Distributed Data sets (RDDs)

---

- RDDs
  - Are **immutable** once constructed
    - i.e., the content of an RDD cannot be modified
- Spark tracks lineage information to efficiently recompute lost data (**due to failures of some executors**)
  - i.e., for each RDD, Spark knows how it has been constructed and can rebuilt it if a failure occurs
  - This information is represented by means of a **DAG** (**Direct Acyclic Graph**) connecting input data and RDDs

# Resilient Distributed Data sets (RDDs)

---

- RDDs can be created
  - by parallelizing existing collections of the hosting programming language (e.g., collections and lists of Scala, Java, Python, or R)
    - In this case the number of partition is specified by the user
  - from (large) files stored in HDFS
    - In this case there is one partition per HDFS block
  - from files stored in many traditional file systems or databases
  - by transforming an existing RDDs
    - The number of partitions depends on the type of transformation

# Resilient Distributed Data sets (RDDs)

---

- Spark programs are written in terms of operations on resilient distributed data sets
  - Transformations
    - map, filter, join, ...
  - Actions
    - count, collect, save, ...

# Spark Framework

---

- **Spark**

- Manages scheduling and synchronization of the jobs
- Manages the split of RDDs in partitions and allocates RDDs' partitions in the nodes of the cluster
- Hides complexities of fault-tolerance and slow machines
  - RDDs are automatically rebuilt in case of machine failures

# Spark Programs

---

# Supported languages

---

- Spark supports many programming languages
  - Scala
    - The same language that is used to develop the Spark framework and all its components (Spark Core, Spark SQL, Spark Streaming, MLlib, GraphX)
  - Java
  - Python
  - R

# Supported languages

---

- Spark supports many programming languages
  - Scala
    - The same language that is used to develop the Spark framework and all its components (Spark Core, Sparl SQL, Spark Streaming, MLlib, GraphX)
  - Java
  - Python ← We will use Python
  - R

# Structure of Spark programs

---

- The **Driver** program
  - Contains the main method
  - “Defines” the workflow of the application
  - Accesses Spark through the **SparkContext** object
    - The SparkContext object represents a connection to the cluster
  - Defines Resilient Distributed Datasets (RDDs) that are “allocated” in the nodes of the cluster
  - Invokes parallel operations on RDDs

# Structure of Spark programs

---

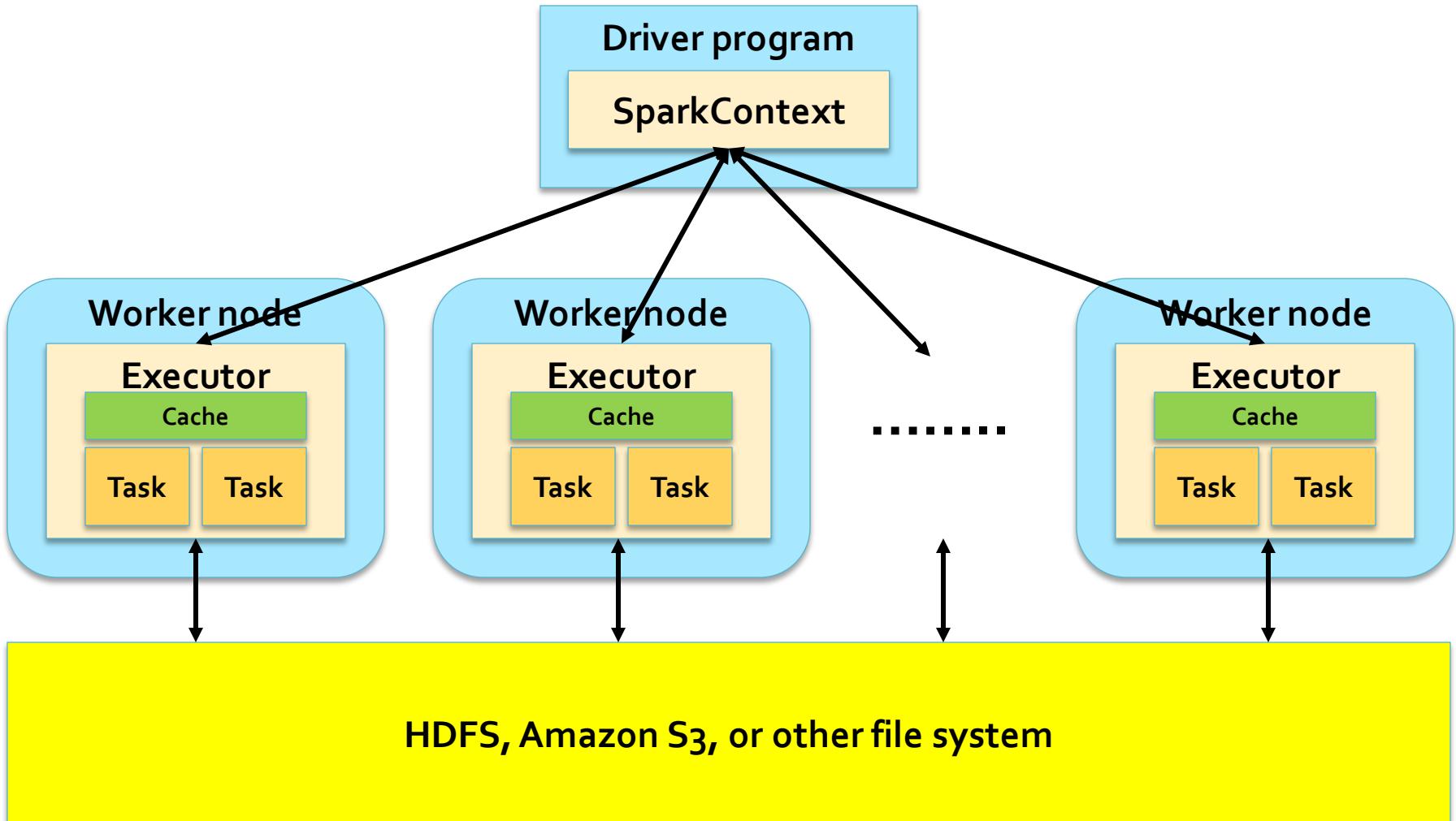
- The **Driver** program **defines**
  - Local variables
    - The standard variables of the Python programs
  - RDDs
    - Distributed “variables” stored in the nodes of the cluster
- The **SparkContext** object allows
  - Creating RDDs
  - “Submitting” executors (processes) that execute in parallel specific operations on RDDs
    - Transformations and Actions

# Structure of Spark programs

---

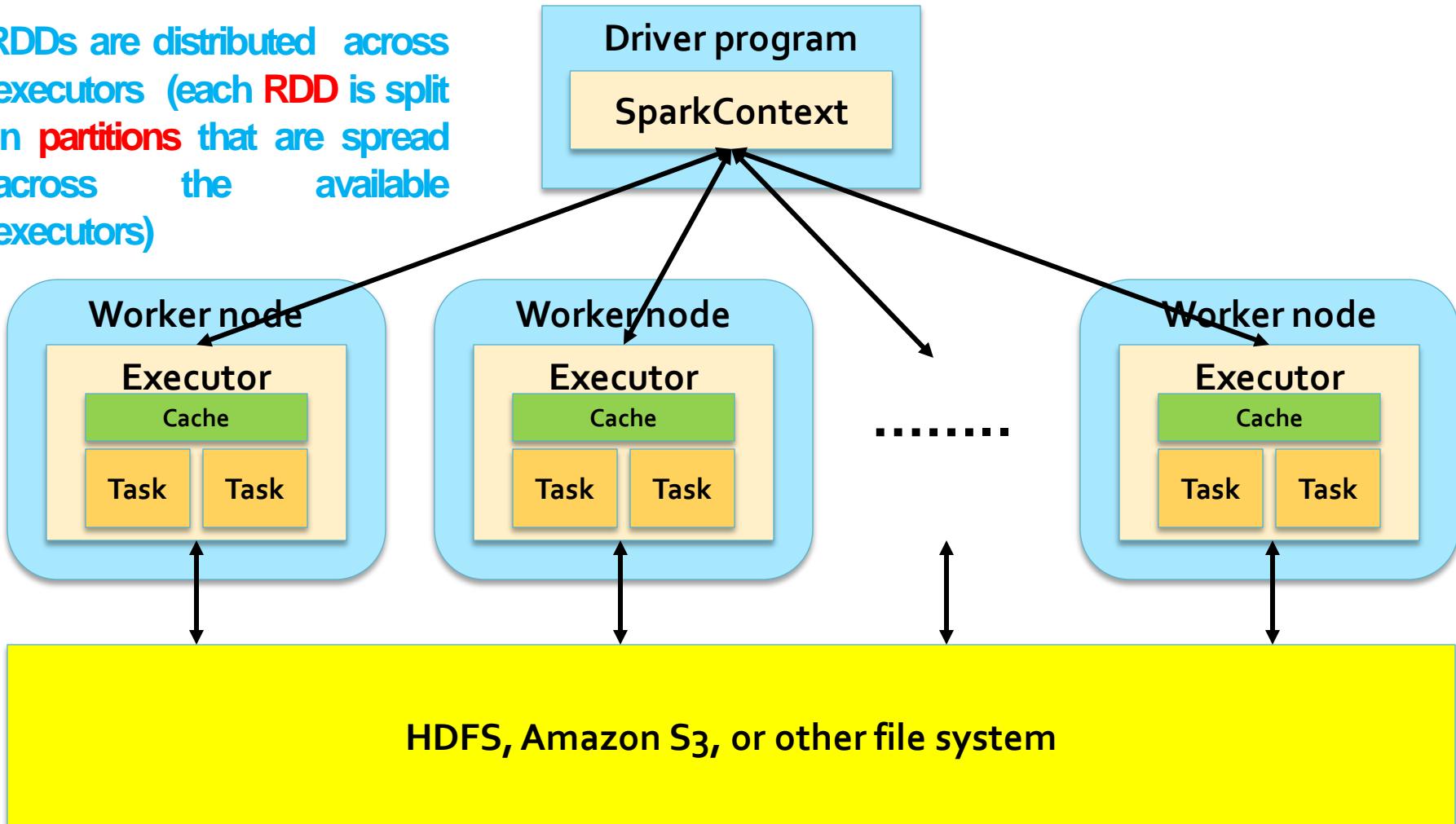
- The **worker nodes** of the cluster are used to run your application by means of **executors**
- Each **executor** runs on its partition of the RDD(s) the **operations** that are specified in the driver

# Distributed execution of Spark



# Distributed execution of Spark

RDDs are distributed across executors (each RDD is split in partitions that are spread across the available executors)

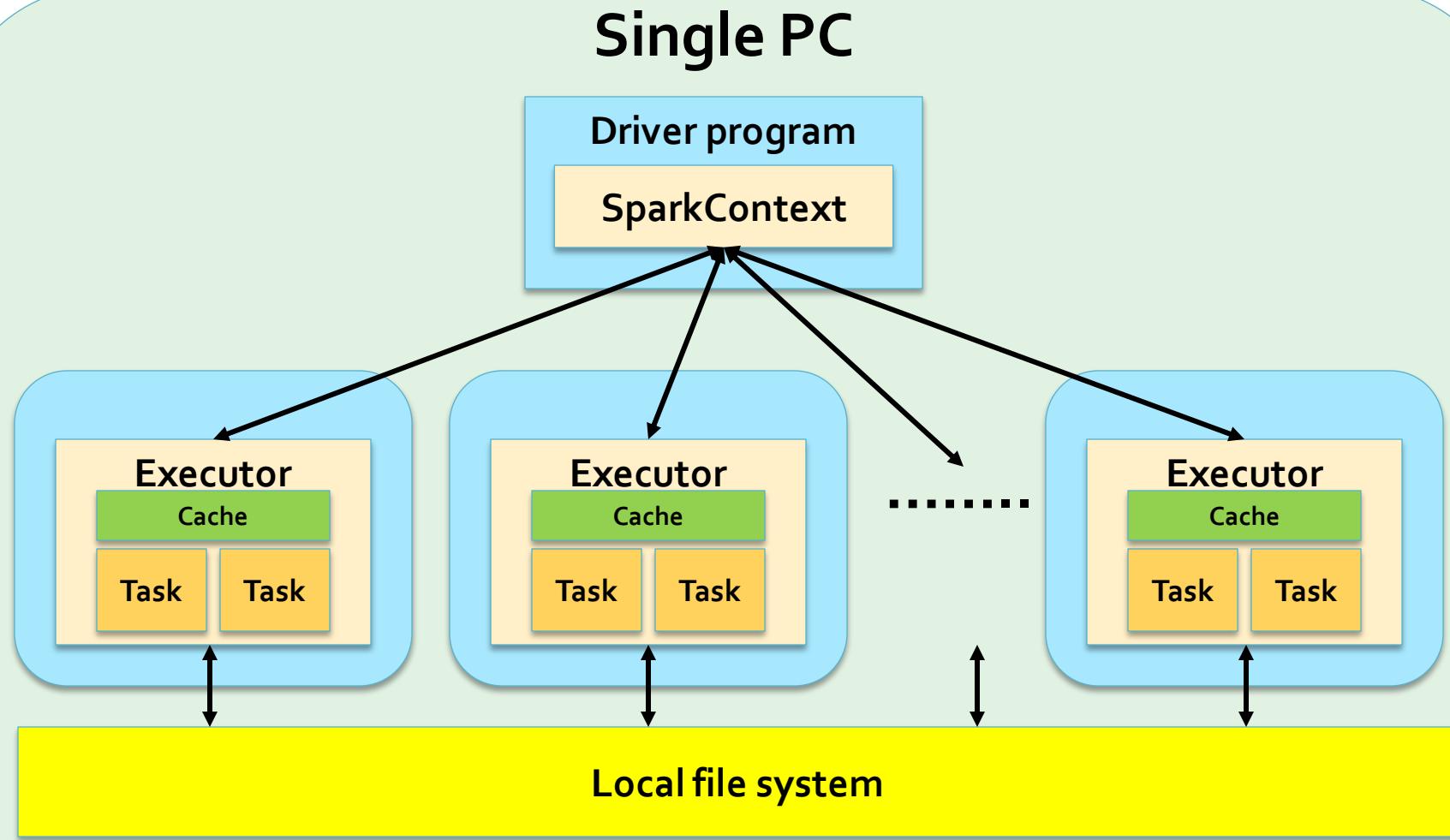


# Local execution of Spark

---

- Spark programs can also be executed locally
  - Local threads are used to parallelize the execution of the application on RDDs on a single PC
    - Local threads can be seen as “pseudo-worker” nodes
  - It is useful to develop and test the applications before deploying them on the cluster
  - A local scheduler is launched to run Spark programs locally

# Local execution of Spark



# Spark official terminology

---

- **Application**
  - User program built on Spark
  - It consists of a driver program and executors on the cluster
- **Driver program**
  - The process running the main() function of the application and creating the SparkContext

Based on <http://spark.apache.org/docs/latest/cluster-overview.html>

# Spark official terminology

---

- **Cluster manager**
  - An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
- **Deploy mode**
  - Distinguishes where the driver process runs
    - In "**cluster**" mode, the framework launches the driver inside of the cluster
    - In "**client**" mode, the submitter launches the driver outside of the cluster
- **Worker node**
  - Any node of the cluster that can run application code in the cluster

# Spark official terminology

---

- **Executor**
  - A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them
  - Each application has its own executors
- **Task**
  - A unit of work that will be sent to one executor
- **Job**
  - A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect)

# Spark official terminology

---

- **Stage**
  - Each **job** gets **divided into** smaller sets of tasks called **stages**
  - The output of one stage is the input of the next stage(s)
    - Except the stages that compute (part of) the final result (i.e., the stages without output edges in the graph representing the workflow of the application)
      - The outputs of those stages is stored in **HDFS** or a **database**
  - The **shuffle operation** is always executed **between** two **stages**
    - Data must be grouped/repartitioned based on a grouping criteria that is different with respect to the one used in the previous stage
    - Similar to the shuffle operation between the map and the reduce phases in MapReduce
    - **Shuffle** is a **heavy operation**

# **Spark Programs: Examples**

---

# Spark program: Count line

---

- Count the number of lines of the input file
  - The name of the file is set to “myfile.txt”
- Print the results on the standard output

# Spark program: Count line

---

```
from pyspark import SparkConf, SparkContext  
  
if __name__ == "__main__":  
  
    #Create a configuration object and  
    #set the name of the application  
    conf = SparkConf().setAppName("Spark Line Count")  
  
    # Create a Spark Context object  
    sc = SparkContext(conf=conf)  
  
    # Store the path of the input file in inputFile  
    inputFile= "myfile.txt"
```

# Spark program: Count line

---

```
# Build an RDD of Strings from the input textual file  
# Each element of the RDD is a line of the input file  
linesRDD = sc.textFile(inputFile)  
  
# Count the number of lines in the input file  
# Store the returned value in the local variable numLines  
numLines = linesRDD.count()  
# Print the output in the standard output  
print("NumLines:", numLines)  
  
# Close the Spark Context object  
sc.stop()
```

# Spark program: Count line

```
from pyspark import SparkConf, SparkContext  
  
if __name__ == "__main__":  
  
    #Create a configuration object and  
    #set the name of the application  
    conf = SparkConf().setAppName("MyApp")  
  
    # Create a Spark Context object  
    sc = SparkContext(conf=conf)  
  
    # Store the path of the input file in inputFile  
    inputFile = "myfile.txt"
```

**Local Python variables.**  
They are allocated in the main memory  
of the same process instancing  
the Driver

# Spark program: Count line

```
# Build an RDD of Strings from the input textual file  
# Each element of the RDD is a line of the input file
```

```
linesRDD = sc.textFile(inputFile)
```

RDD.

It is allocated/stored in the main memory  
or in the local disk of the executors of the  
worker nodes

```
# Count the number of lines
```

```
# Store the returned value in a local Python variable
```

```
numLines = linesRDD.count()
```

```
# Print the output in the standard output
```

```
print("NumLines:", numLines)
```

Local Python variable.

It is allocated in the main memory  
of the same process instancing  
the Driver

```
# Close the Spark Context
```

```
sc.stop()
```

# Spark program: Count line

---

- **Local variables**
  - Can be used to store only “small” objects/data
    - The maximum size is equal to the main memory of the process associated with the Driver
- **RDDs**
  - Are used to store “big/large” collections of objects/data in the nodes of the cluster
    - In the main memory of the worker nodes, when it is possible
    - In the local disks of the worker nodes, when it is necessary

# Spark program: Word Count

---

- Word Count implemented by means of Spark
  - The name of the input file is specified by using a command line parameter (i.e., `argv[1]`)
  - The output of the application (i.e., the pairs (word, num. of occurrences) is stored in an output folder (i.e., `argv[2]`)
- **Note:** Do not worry about details

# Spark program: Word Count

---

```
from pyspark import SparkConf, SparkContext
import sys

if __name__ == "__main__":
    """
    Word count example
    """

    inputFile= sys.argv[1]
    outputPath = sys.argv[2]

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

    # Create a Spark Context object
    sc = SparkContext(conf=conf)
```

# Spark program: Word Count

---

```
# Build an RDD of Strings from the input textual file
# Each element of the RDD is a line of the input file
lines = sc.textFile(inputFile)

# Split/transform the content of lines in a
# list of words and store them in the words RDD
words = lines.flatMap(lambda line: line.split(sep=' '))

#Map/transform each word in the words RDD
#to a pair/tuple (word,1) and store the result in the words_one RDD
words_one = words.map(lambda word: (word, 1))
```

# Spark program: Word Count

---

```
# 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  
counts = words_one.reduceByKey(lambda c1, c2: c1 + c2)  
  
# Store the result in the output folder  
counts.saveAsTextFile(outputPath)  
  
# Close/Stop the Spark Context object  
sc.stop()
```

# **How to submit/execute a Spark application**

---

# Spark-submit

---

- Spark programs are executed (submitted) by using the spark-submit command
  - It is a command line program
  - It is characterized by a set of parameters
    - E.g., the name of the jar file containing all the classes of the Spark application we want to execute
    - The name of the Driver class
    - The parameters of the Spark application
    - etc.

# Spark-submit

---

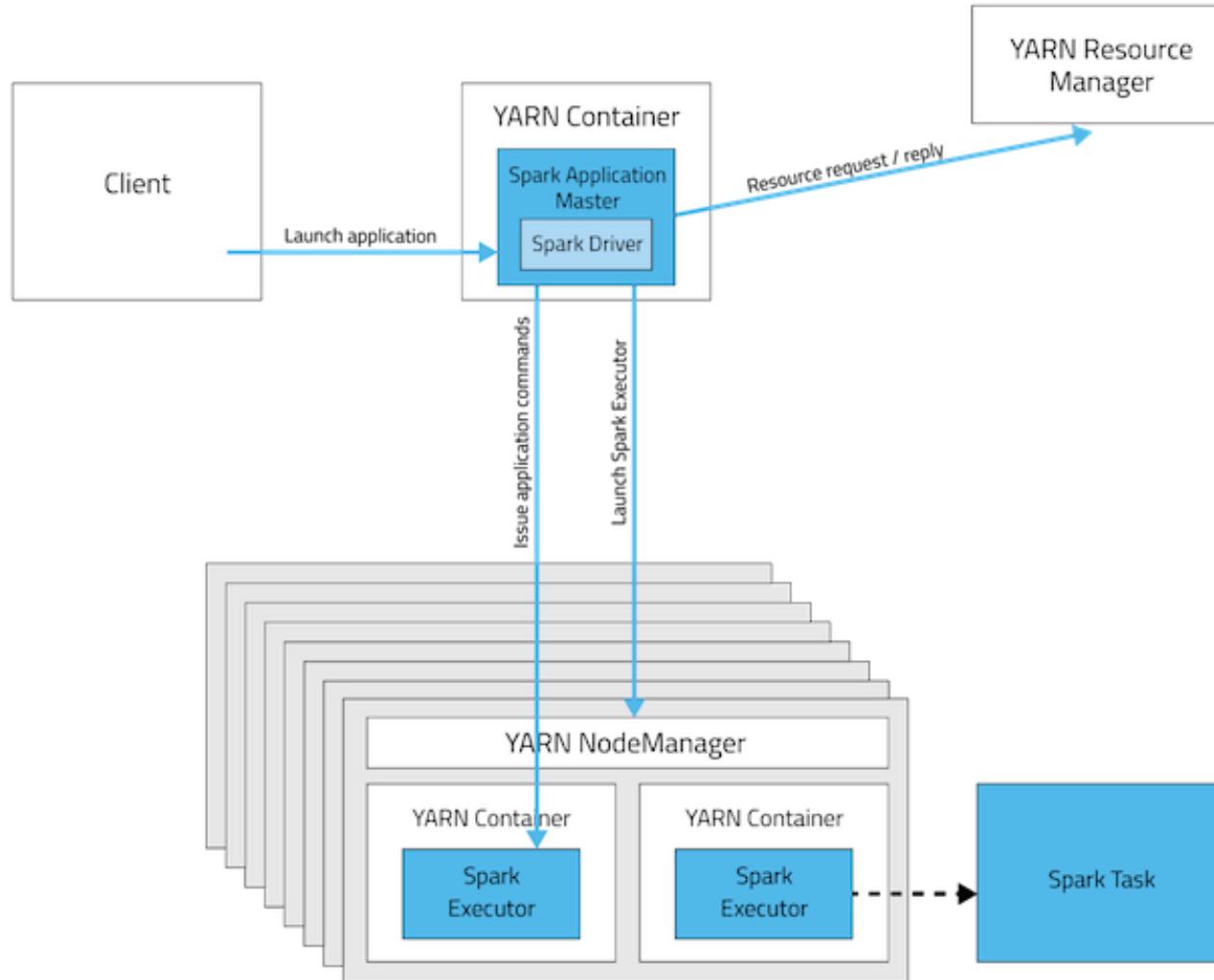
- **spark-submit** has also **two parameters** that are used to specify where the application is executed
  - **--master** option
    - Specify which **environment/scheduler** is used to execute the application
      - spark://host:port              The spark scheduler is used
      - mesos://host:port              The memos scheduler is used
      - yarn                              The YARN scheduler (i.e., the one of Hadoop)
      - local                            The application is executed exclusively on the local PC

# Spark-submit

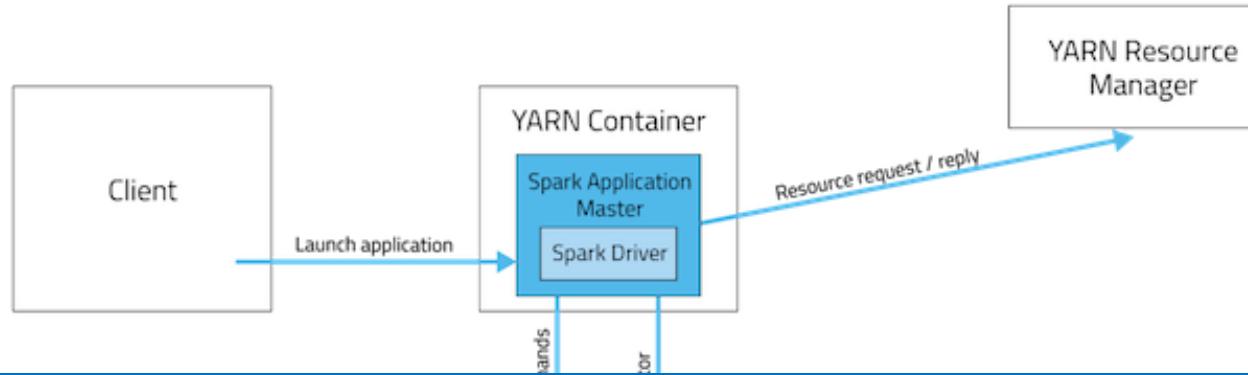
---

- **--deploy-mode** option
  - Specify where the Driver is launched/executed
    - client                          The driver is launched locally (in the “local” PC executing spark-submit)
    - cluster                        The driver is launched on one node of the cluster

# Cluster Deployment Mode

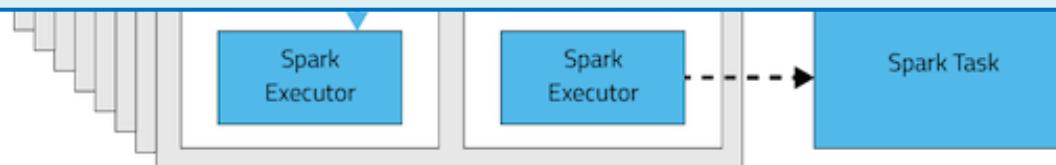


# Cluster Deployment Mode

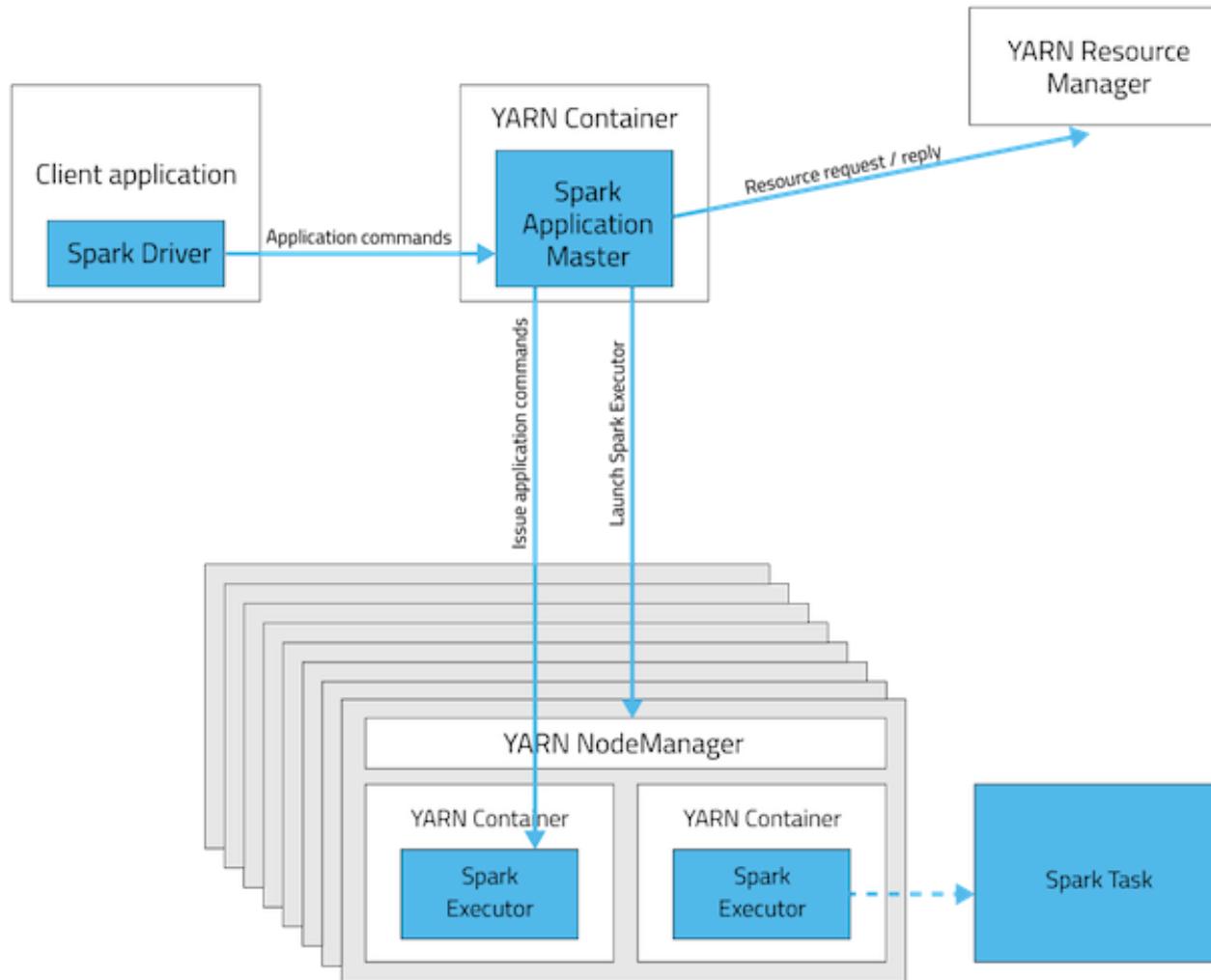


## In cluster mode

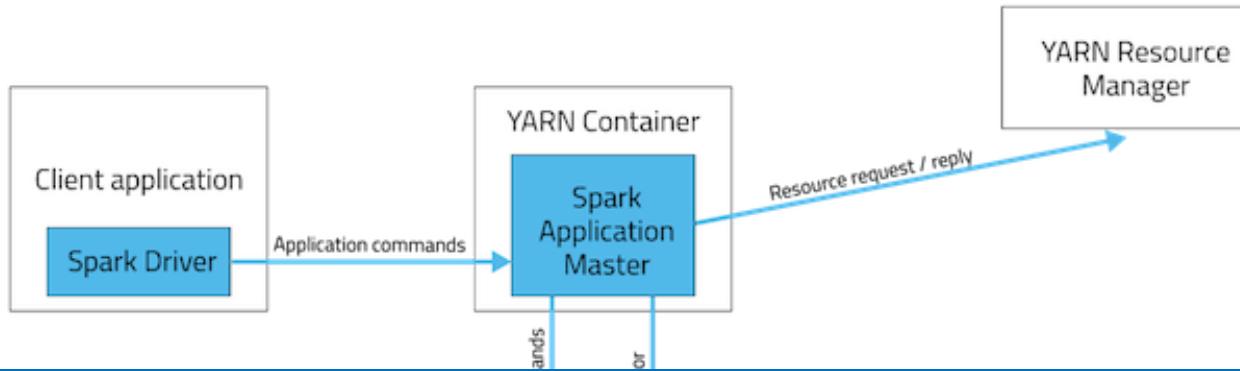
- The Spark driver runs in the ApplicationMaster on a cluster node.
- The cluster nodes are used also to store RDDs and execute transformations and actions on the RDDs
- A single process in a YARN container is responsible for both driving the application and requesting resources from YARN.
- The resources (memory and CPU) of the client that launches the application are not used.



# Client Deployment Mode

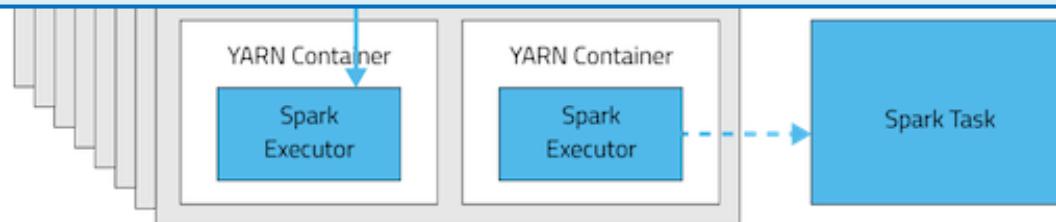


# Client Deployment Mode



## In client mode

- The Spark driver runs on the host where the job is submitted (i.e., the resources of the client are used to execute the Driver)
- The cluster nodes are used to store RDDs and execute transformations and actions on the RDDs
- The ApplicationMaster is responsible only for requesting executor containers from YARN.



# Spark-submit: setting executors

---

- Spark-submit allows specifying
  - The number of executors
    - --num-executors NUM
      - Default value: NUM=2 executors
  - The number of cores per executor
    - --executor-cores NUM
      - Default value: NUM=1 core
  - Main memory per executor
    - --executor-memory MEM
      - Default value: MEM=1GB
- The maximum values of these parameters are limited by the configuration of the cluster

# Spark-submit: setting driver

---

- Spark-submit allows specifying
  - The number of cores for the driver
    - --driver-cores NUM
      - Default value: NUM=1 core
  - Main memory for the driver
    - --driver-memory MEM
      - Default value: MEM=1GB
- Also the maximum values of these parameters are limited by the configuration of the cluster when the deploy-mode is set to cluster

# Spark-submit: Execution on the cluster

---

- The following command submits a Spark application on a Hadoop cluster

```
spark-submit --deploy-mode cluster --master yarn  
MyApplication.py arguments
```

- It executes/submits the application contained in MyApplication.py
- The application is executed on a Hadoop cluster based on the YARN scheduler
  - Also the Driver is executed in a node of cluster

# Spark-submit: Local execution

---

- The following command submits a Spark application on a local PC

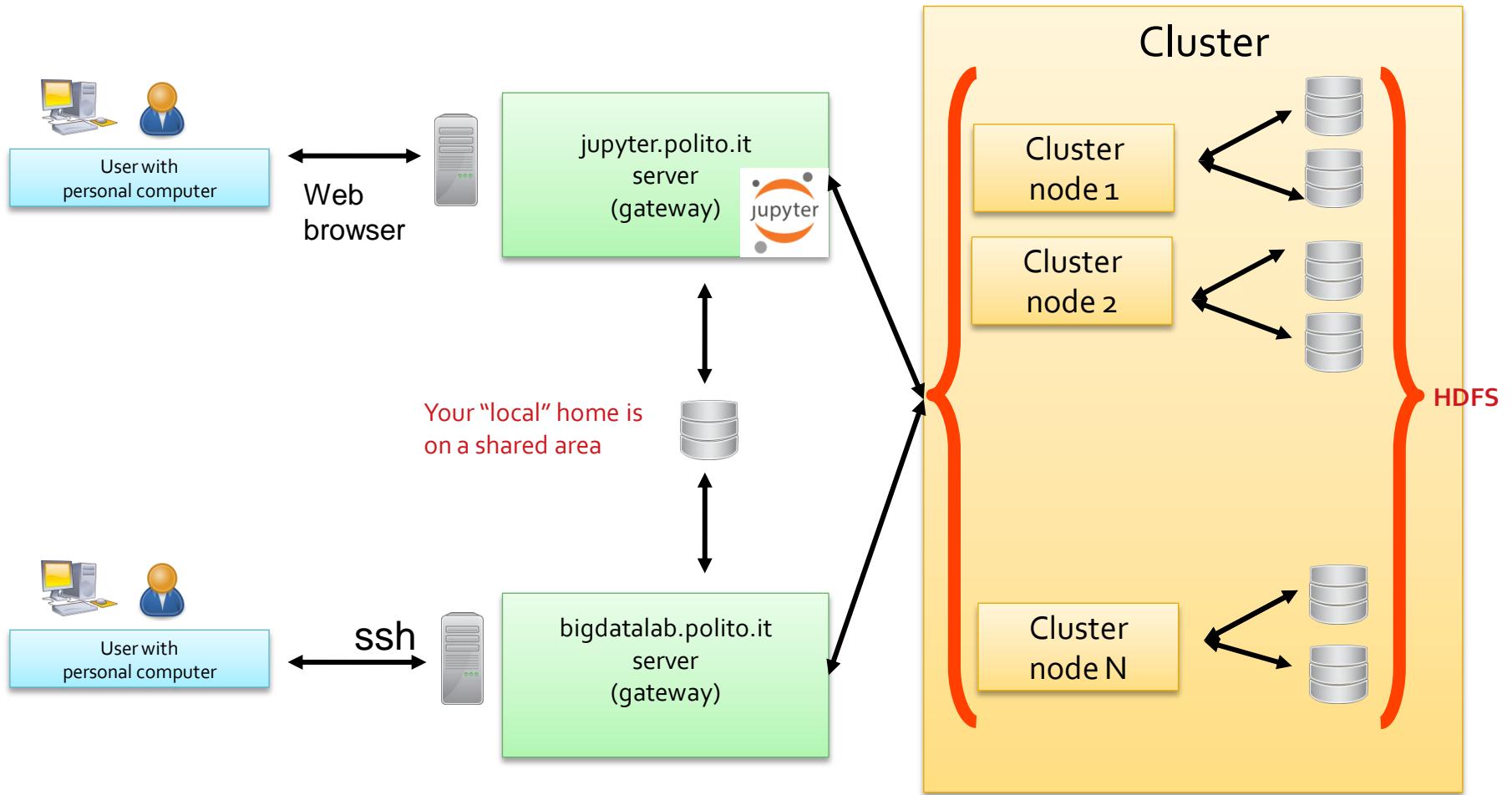
```
spark-submit --deploy-mode client --master local  
MyApplication.py arguments
```

- It executes/submits the application contained in *MyApplication.py*
- The application is completely executed on the local PC
  - Both Driver and Executors
  - Hadoop is not needed in this case
  - You only need the Spark software

# **How to execute Spark applications by using Jupyter notebooks**

---

# The BigData@Polito environment





# Jupyter notebooks

---

- Jupyter notebook
  - Browser-based interactive IDE
- Specific “notebooks” can be used to run Spark applications on the Spark cluster
  - PySpark (Local)
    - Run the application in a container (in a local instance of Spark)
  - PySpark (Yarn)
    - Run the application on the BigData@Polito cluster
  - Both notebooks read/write data from/in HDFS

# Execute an application by using PySpark on a Jupyter notebook

---



- Copy the input data of your application from the local drive of your personal workstation on the HDFS file system of the cluster
- Open an interactive PySpark shell by using a Jupyter notebook
- Write the python/spark code you want to execute and execute it step-by-step by using the PySpark notebook
- The result is stored in the output HDFS folder specified in your application

# Execute an application by using PySpark on a Jupyter notebook



The screenshot shows the Jupyter Notebook interface. On the left is a sidebar with file navigation and a list of files in the current directory (2019\_course). The main area displays a list of notebooks under the heading '2019\_course'. A black oval highlights two icons: 'PySpark (Local)' and 'PySpark (Yarn)'. An arrow points from this oval to a callout box containing the following text:

Create a PySpark Jupyter notebook in the local file system of the machine hosting Jupyter ([jupyter.polito.it](http://jupyter.polito.it))

The main area also includes sections for 'Console' and 'Other' with icons for Python 3, PySpark (Local), Text File, Markdown File, and Contextual Help.

# RDD-based programming

---

# **Spark context**

---

# SparkContext

---

- The “**connection**” of the driver to the cluster is based on the **Spark Context** object
  - In Python the name of the class is **SparkContext**
- The Spark Context is built by means of the constructor of the **SparkContext** class
  - The only parameter is a configuration object

# SparkContext

---

- Example

```
#Create a configuration object and  
#set the name of the application  
conf = SparkConf().setAppName("Application name")
```

```
# Create a Spark Context object  
sc = SparkContext(conf=conf)
```

# SparkContext

---

- The Spark Context object can be obtained also by using the `SparkContext.getOrCreate(conf)` method
  - The only parameter is a configuration object
  - If the SparkContext object already exists for this application the current SparkContext object is returned
  - Otherwise, a new SparkContext object is returned
- There is always **one single SparkContext object for each application**

# SparkContext

---

- Example

```
#Create a configuration object and  
#set the name of the application  
conf = SparkConf().setAppName("Application name")
```

```
# Retrieve the current SparkContext object or  
# create a new one  
sc = SparkContext.getOrCreate(conf=conf)
```

# **RDD basics**

---

# RDD basics

---

- A Spark RDD is an **immutable distributed collection** of objects
- Each RDD is split in **partitions**
  - This choice allows parallelizing the code based on RDDs
    - Code is executed on each partition in isolation
- RDDs can contain any type of Scala, Java, and Python objects
  - Including user-defined classes

# **RDD: create and save**

---

# RDD creation

---

- **RDDs** can be created
  - By **loading** an external dataset (e.g., the content of a folder, a single file, a database table, etc.)
  - By **parallelizing** a local collection of objects created in the Driver (e.g., a Java collection)

# Create RDDs from files

---

- An RDD can be built from an input textual file
  - It is based on the `textFile(name)` method of the `SparkContext` class
    - The returned RDD is an RDD of Strings associated with the content of the *name* textual file
    - Each line of the input file is associated with an object (a string) of the instantiated RDD
    - By default, if the input file is an HDFS file the number of partitions of the created RDD is equal to the number of HDFS blocks used to store the file
      - To support data locality

# Create RDDs from files

---

- Example

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

# Create RDDs from files

---

## ■ Example

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

No computation occurs when `sc.textFile()` is invoked

- Spark only records how to create the RDD
- The data is lazily read from the input file only when the data is needed (i.e., when an action is applied on `lines`, or on one of its “descendant” RDDs)

# Create RDDs from files

---

- An RDD can be built from a **folder** containing textual files
  - It is based on the **textFile(name)** method of the **SparkContext** class
    - If *name* is the path of a folder all files inside that folder are considered
    - The returned RDD contains one string for each line of the files contained on the *name* folder

# Create RDDs from files

---

- Example

```
# Build an RDD of strings from all the files stored in  
# myfolder  
# Each element of the RDD is a line of the input files  
inputFolder = "myfolder/"  
lines = sc.textFile(inputFolder)
```

# Create RDDs from files

---

- Example

```
# Build an RDD of strings from all the files stored in  
# myfolder  
# Each element of the RDD is a line of the input files  
inputFolder = "myfolder/"  
lines = sc.textFile(inputFolder)
```

Pay attention that **all files** inside myfolder **are considered**.  
Also those without suffix or with a suffix different from .txt

# Create RDDs from files

---

- The developer can manually set the (minimum) **number of partitions**
  - In this case the **textFile(name, minPartitions)** method of the **SparkContext** class is used
    - This option can be used to increase the parallelization of the submitted application
    - For the HDFS files, the number of partitions *minPartitions* must be greater than the number of blocks/chunks

# Create RDDs from files

---

- Example

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

# Create RDDs from a local Python collection

---

- An RDD can be built from a “local” Python collection/list of local python objects
  - It is based on the `parallelize(c)` method of the `SparkContext` class
    - The created RDD is an RDD of objects of the same type of objects of the input python collection c
    - In the created RDD, there is one object for each element of the input collection
    - Spark tries to set the number of partitions automatically based on your cluster’s characteristics

# Create RDDs from a local Python collection

---

- Example

```
# 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 is set automatically by Spark  
# There is one element of the RDD for each element  
# of the local list  
distRDDList = sc.parallelize(inputList)
```

# Create RDDs from a local Python collection

## ■ Example

```
# Create a local python list  
inputList = ['First element', 'Second element', 'Third  
element']
```

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

```
distRDDList = sc.parallelize(inputList)
```

# Create RDDs from a local Python collection

---

- When the `parallelize(c)` is invoked
  - Spark tries to set the number of partitions automatically based on your cluster's characteristics
- The developer can set the number of partition by using the method `parallelize(c, numSlices)` of the `SparkContext` class

# Create RDDs from a local Python collection

---

- Example

```
# 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 is set to 3  
# There is one element of the RDD for each element  
# of the local list  
distRDDList = sc.parallelize(inputList, 3)
```

# Save RDDs

---

- An RDD can be easily **stored** in textual (HDFS) files
  - It is based on the **saveAsTextFile(path)** method of the **RDD** class
    - *path* is the path of a folder
    - The method is invoked on the RDD that we want to store in the output folder
    - Each object of the RDD on which the saveAsTextFile method is invoked is stored in one line of the output files stored in the output folder
      - There is one output file for each partition of the input RDD

# Save RDDs

---

- Example

```
# Store the content of linesRDD in the output folder  
# Each element of the RDD is stored in one line  
# of the textual files of the output folder  
outputPath="risFolder/"  
linesRDD.saveAsTextFile(outputPath);
```

# Save RDDs

---

## ■ Example

```
# Store the content of linesRDD in the output folder  
# Each element of the RDD is stored in one line  
# of the textual files of the output folder  
outputPath="risFolder/"  
  
linesRDD.saveAsTextFile(outputPath);
```

saveAsTextFile() is an action.

Hence Spark computes the content associated with linesRDD when saveAsTextFile() is invoked.

Spark computes the content of an RDD only when that content is needed.

# Save RDDs

---

- Example

```
# Store the content of linesRDD in the output folder  
# Each element of the RDD is stored in one line  
# of the textual files of the output folder  
  
outputPath="risFolder/"  
  
linesRDD.saveAsTextFile(outputPath);
```

Note that the output folder contains one textual file for each partition of linesRDD.  
Each output file contains the elements of one partition.

# Retrieve the content of RDDs and “store” it in local python variables

---

- The content of an RDD can be retrieved from the nodes of the cluster and “stored” in a local python variable of the Driver
  - It is based on the **collect()** method of the **RDD** class

# Retrieve the content of RDDs and “store” it in local python variables

---

- The `collect()` method of the `RDD` class
  - Is invoked on the RDD that we want to “retrieve”
  - Returns a `local python list` of objects containing the same objects of the considered RDD
  - **Pay attention to the size of the RDD**
  - **Large RDD cannot be stored in a local variable of the Driver**

# Retrieve the content of RDDs and “store” it in local python variables

---

- Example

```
# 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();
```

# Retrieve the content of RDDs and “store” it in local python variables

- Example

```
# 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();
```

Local python variable.  
It is allocated in the main memory  
of the Driver process/task

RDD of strings.  
It is distributed across  
the nodes of the cluster

# **Transformations and Actions**

---

# RDD operations

---

- RDD support two types of operations
  - Transformations
  - Actions

# RDD operations

---

## ■ Transformations

- Are operations on RDDs that return a new RDD
- Apply a transformation on the elements of the input RDD(s) and the result of the transformation is “stored in/associated with” a new RDD
  - Remember that RDDs are immutable
    - Hence, you cannot change the content of an already existing RDD
    - You can only apply a transformation on the content of an RDD and “store/assign” the result in/to a new RDD

# RDD operations

---

- Transformations
  - Are **computed lazily**
    - i.e., transformations are computed ("executed") only when an action is applied on the RDDs generated by the transformation operations
    - When a transformation is invoked
      - Spark keeps only **track** of the **dependency between** the **input RDD** and the **new RDD** returned by the transformation
      - The content of the new RDD is not computed

# RDD operations

---

- The **graph of dependencies between RDDs** represents the information about which RDDs are used to create a new RDD
  - This is called **lineage graph**
    - It is represented as a **DAG (Directed Acyclic Graph)**
  - It is **needed** to **compute** the content of an RDD the first time an **action is invoked on it**
  - Or to **compute** again the content of an RDD (or some of its partitions) when **failures occur**

# RDD operations

---

- The lineage graph is also useful for optimization purposes
  - When the content of an RDD is needed, Spark can consider the chain of transformations that are applied to compute the content of the needed RDD and potentially decide how to execute the chain of transformations
    - Spark can potentially change the order of some transformations or merge some of them based on its optimization engine

# RDD operations

---

- **Actions**

- Are operations that
  - **Return results** to the **Driver program**
    - i.e., return **local (python) variables**
    - **Pay attention to the size of the returned results** because they must be stored in the main memory of the Driver program
  - Or **write** the result **in the storage** (output file/folder)
    - The size of the result can be large in this case since it is directly stored in the (distributed) file system

# Example of lineage graph (DAG)

---

- Consider the following code

```
from pyspark import SparkConf, SparkContext  
import sys
```

```
if __name__ == "__main__":  
    conf = SparkConf().setAppName("Spark Application")  
    sc = SparkContext(conf=conf)
```

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

# Example of lineage graph (DAG)

---

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

```
# Select the rows containing the word "warning"  
warningRDD = inputRDD.filter(lambda line:  
                           line.find('warning')>=0)
```

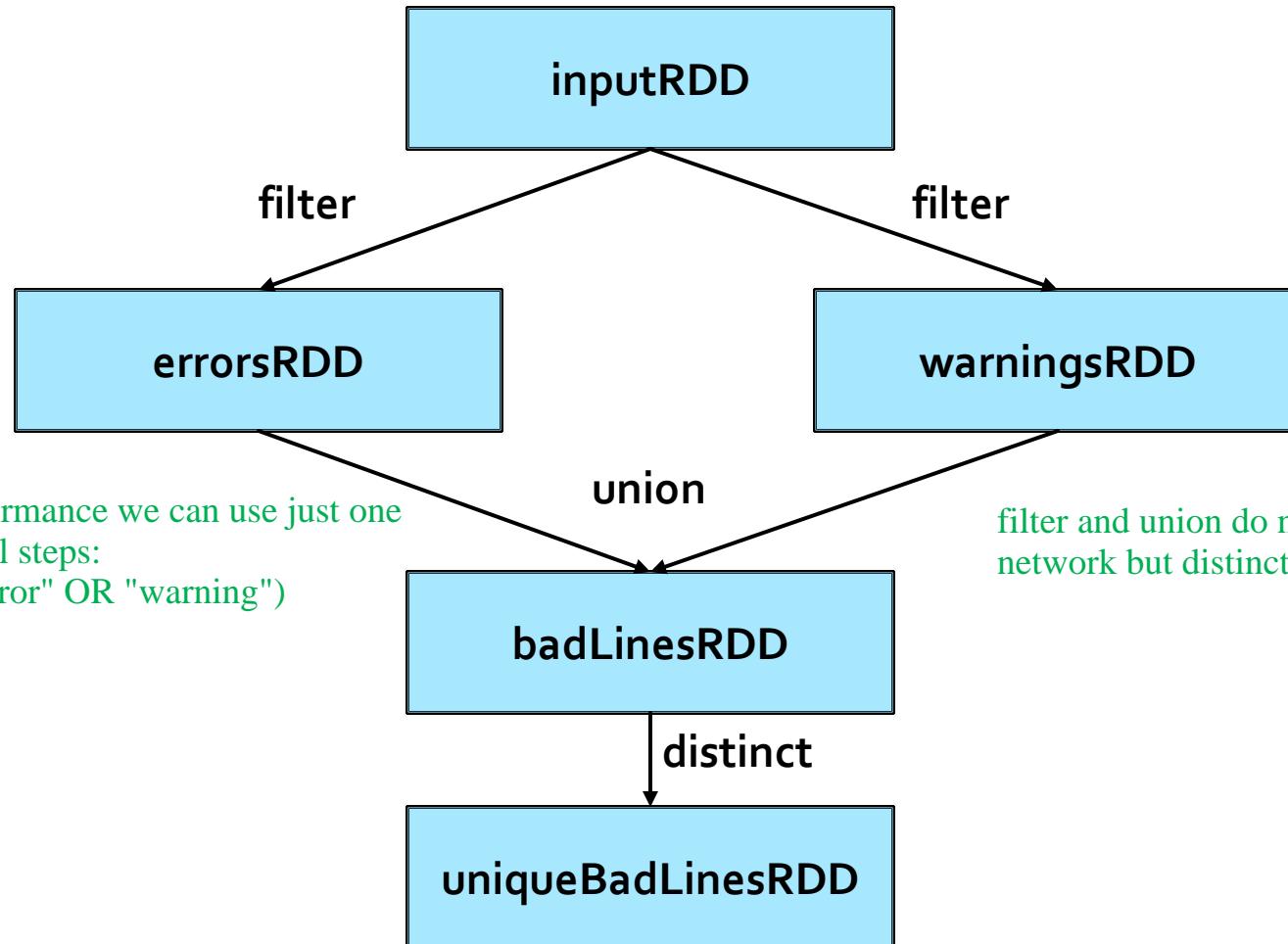
```
# Union of errorsRDD and warningRDD  
# The result is associated with a new RDD: badLinesRDD  
badLinesRDD = errorsRDD.union(warningRDD)
```

# Example of lineage graph (DAG)

---

```
# Remove duplicates lines (i.e., those lines containing  
# both "error" and "warning")  
uniqueBadLinesRDD = badLinesRDD.distinct()  
  
# Count the number of bad lines by applying  
# the count() action  
numBadLines = uniqueBadLinesRDD.count()  
  
# Print the result on the standard output of the driver  
print("Lines with problems:", numBadLines)
```

# Example of lineage graph (DAG)



# Example of lineage graph (DAG)

---

- The application reads the input log file only when the `count()` **action** is invoked
  - It is the first action of the program
- `filter()`, `union()`, and `distinct()` are **transformations**
  - They are computed lazily
- Also `textFile()` is computed lazily
  - However, it is **not a transformation** because it is not applied on an RDD

# Example of lineage graph (DAG)

---

- Spark, similarly to an SQL optimizer, can potentially optimize the “execution” of some transformations
  - For instance, in this case the two filters + union + distinct can be potentially optimized and transformed in one single filter applying the constraint
    - The element contains the string “error” or “warning”
  - This optimization improves the efficiency of the application
    - Spark can performs this kind of optimizations only on particular types of RDDs: Datasets and DataFrames

# **Passing function to Transformations and Actions**

---

# Passing functions to Transformations and Actions

---

- Many transformations (and some actions) are based on user provided functions that specify which “transformation” function must be applied on the elements of the “input” RDD
- For example the filter() transformation selects the elements of an RDD satisfying a user specified constraint
  - The user specified constraint is a Boolean function applied on each element of the “input” RDD

# Passing functions to Transformations and Actions

---

- Each language has its own solution to pass functions to Spark's transformations and actions
- In python, we can use
  - **Lambda** functions/expressions
    - Simple functions that can be written as one single expression
  - **Local user defined functions** (local “defs”)
    - For multi-statement functions or statements that do not return a value

# Example based on the filter transformation

---

- Create an RDD from a log file
- Create a new RDD containing only the lines of the log file containing the word “error”
  - The filter() transformation applies the filter constraint on each element of the input RDD
    - The filter constraint is specified by means of a Boolean function that returns true for the elements satisfying the constraint and false for the others

# Solution based on lambda expressions

---

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

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

# Solution based on lambda expressions

# This part of the code, which is based on a lambda expression, defines on the fly the function that we want to apply. This part of the code is applied on each object of inputRDD. If it returns true then the current object is “stored” in the new errorsRDD RDD. Otherwise the input object is discarded

# Select the rows containing the word “error”

```
errorsRDD = inputRDD.filter(lambda l: l.find('error')>=0)
```

# Solution based on def

---

```
# Define the content of the Boolean function that is applied
# to select the elements of interest
def myFunction(l):
    if l.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(myFunction)
```

# Solution based on def

---

```
# Define the content of the Boolean function that is applied  
# to select the elements of interest
```

```
def myFunction(l):  
    if l.find('error')>=0: return True  
    else: return False
```

When it is invoked, this function analyzes the value of the parameter line and returns true if the string line contains the substring "error". Otherwise, it returns false.

```
# Select the rows containing the word "error"  
errorsRDD = inputRDD.filter(myFunction)
```

# Solution based on def

---

```
# Define the content of the Boolean function that is applied  
# to select the elements of interest  
def myFunction(l):  
    if l.find('error')>=0: return True  
    else: return False
```

Apply the filter() transformation on inputRDD.

The filter transformation selects the elements of inputRDD satisfying the constraint specified in myFunction.

```
# Select the rows containing the word "error"  
errorsRDD = inputRDD.filter(myFunction)
```

# Solution based on def

---

```
# Define the content of the Boolean function that is applied  
# to select the elements of interest  
def myFunction(l):  
    if l.find('error')>=0: return True  
    else: return False
```

For each object o in inputRDD the myFunction function is automatically invoked.  
If myFunction returns true then o is “stored” in the new RDD errorsRDD.  
Otherwise o is discarded

# Select the rows containing the word “error”

```
errorsRDD = inputRDD.filter(myFunction)
```

# Solution based on def - Version 2

---

```
# Define the content of the Boolean function that is applied  
# to select the elements of interest  
def myFunction(l):  
    return l.find('error')>=0
```

This part of the code is the same used in the lambda-based version.

```
# Select the rows containing the word "error"  
errorsRDD = inputRDD.filter(myFunction)
```

# Lambda functions vs local defined functions

---

- The two solutions are more or less equivalent in terms of efficiency
- **Lambda function-based code**
  - More concise
  - More readable
  - But multi-statement functions or statements that do not return a value are not supported
- **Local user defined functions (local “defs”)**
  - Multi-statement functions or statements that do not return a value are supported
  - Code can be reused
    - Some functions are used in several applications

# **Basic Transformations**

---

# Basic RDD transformations

---

- Some basic transformations analyze the content of **one single RDD** and return a new RDD
  - E.g., filter(), map(), flatMap(), distinct(), sample()
- Some other transformations analyze the content of **two (input) RDDs** and return a new RDD
  - E.g., union(), intersection(), subtract(), cartesian()

# **Filter transformation**

---

# Filter transformation

---

- Goal
  - The filter transformation is applied on one single RDD and returns a new RDD containing only the elements of the “input” RDD that satisfy a user specified condition

# Filter transformation

---

- Method
  - The filter transformation is based on the **filter(f)** method of the **RDD** class
  - A function **f** returning a Boolean value is passed to the filter method
    - It contains the code associated with the condition that we want to apply on each element **e** of the “input” RDD
      - If the condition is satisfied then the call method returns true and the input element **e** is selected
      - Otherwise, it returns false and the **e** element is discarded

# Filter transformation: Example 1

---

- Create an RDD from a log file
- Create a new RDD containing only the lines of the log file containing the word “error”

# Filter transformation: Example 1

---

.....

```
# 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)
```

# Filter transformation: Example 1

---

.....

```
# 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)
```

We are working with an input RDD containing strings .  
Hence, the implemented lambda function is applied on one  
string at a time and returns a Boolean value

# Filter transformation: Example 2

---

- Create an RDD of integers containing the values [1, 2, 3, 3]
- Create a new RDD containing only the values greater than 2

# Filter transformation: Example 2

---

.....

```
# Create an RDD of integers. 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)
```

# Filter transformation: Example 2

We are working with an input RDD of integers.

Hence, the implemented lambda function is applied on one integer at a time and returns a Boolean value

```
# Create an RDD of integers. It has the values 1, 2, 3, 3 and RDD  
inputList = [1, 2, 3, 3]  
inputRDD = sc.parallelize(inputList),
```

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

# Filter transformation: Example 2 – Use of def

---

```
.....  
# Define the function to be applied in the filter transformation  
def greaterThan2(num):  
    return num>2  
  
# Create an RDD of integers. 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)
```

# Filter transformation: Example 2 – Use of def

```
.....  
# Define the function to be applied in the filter transformation  
def greaterThan2(num):  
    return num>2
```

```
# Create an RDD of integers. Load the data from a list.  
inputList = [1, 2, 3, 3]  
inputRDD = sc.parallelize(inputList)
```

The function we want to apply is defined by using def and then is passed to the filter transformation

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

# **Map transformation**

---

# Map transformation

---

- Goal
  - The map transformation is used to create a new RDD by applying a function **f** on each element of the “input” RDD
  - The new RDD contains **exactly one** element **y** for each element **x** of the “input” RDD
  - The value of **y** is obtained by applying a user defined function **f** on **x**
    - $y = f(x)$
  - The **data type** of **y** can be **different** from the data type of **x**

# Map transformation

---

- Method
  - The map transformation is based on the **RDD** **map(f)** method of the **RDD** class
  - A function **f** implementing the transformation is passed to the map method
    - It contains the code that is applied over each element of the “input” RDD to create the elements of the returned RDD
      - **For each input element** of the “input” RDD **exactly one single new element is returned** by **f**

# Map transformation: Example 1

---

- Create an RDD from a textual file containing the surnames of a list of users
  - Each line of the file contains one surname
- Create a new RDD containing the length of each surname

# Map transformation: Example 1

---

.....

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

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

# Map transformation: Example 1

The input RDD is an RDD of strings.  
Hence also the input of the lambda function is a String

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

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

# Map transformation: Example 1

The new RDD is an RDD of Integers.

The lambda function returns a new Integer for each input element

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

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

# Map transformation: Example 2

---

- Create an RDD of integers containing the values [1, 2, 3, 3]
- Create a new RDD containing the square of each input element

# Map transformation: Example 2

---

.....

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

# Compute the square of each input element
squaresRDD = inputRDD.map(lambda element: element*element)
```

# **FlatMap transformation**

---

# FlatMap transformation

---

- Goal
  - The flatMap transformation is used to create a new RDD by applying a function **f** on each element of the “input” RDD
  - The new RDD contains a list of elements obtained by applying **f** on each element **x** of the “input” RDD
  - The function **f** applied on an element **x** of the “input” RDD returns a list of values **[y]**
    - $[y] = f(x)$
    - $[y]$  can be the empty list

# FlatMap transformation

---

- The final result is the concatenation of the list of values obtained by applying **f** over all the elements of the “input” RDD
  - i.e., the final RDD contains the “concatenation” of the lists obtained by applying **f** over all the elements of the input RDD
  - **Duplicates are not removed**
- The data type of **y** can be different from the data type of **x**

# FlatMap transformation

---

- Method
  - The flatMap transformation is based on the **flatMap(f)** method of the **RDD** class
  - A function **f** implementing the transformation is passed to the flatMap method
    - It contains the code that is applied on each element of the “input” RDD and returns a list of elements which will be included in the new returned RDD
    - For each element of the “input” RDD a list of new elements is returned by **f**
      - The returned list can be empty

# FlatMap transformation: Example 1

---

- Create an RDD from a textual file containing a generic text
  - Each line of the input file can contain many words
- Create a new RDD containing the list of words, with repetitions, occurring in the input textual document
  - Each element of the returned RDD is 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 returned RDD

# FlatMap transformation: Example 1

---

.....

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

```
# Compute/identify the list of words occurring in document.txt  
listOfWordsRDD = inputRDD.flatMap(lambda l: l.split(' '))
```

a b c	a
d e	b
	c
	d
	e

if we use map we have:

[a, b, c]  
[d, e]

# FlatMap transformation: Example 1

---

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

```
# Compute/identify the list of words occurring in document.txt  
listOfWordsRDD = inputRDD.flatMap(lambda l: l.split(' '))
```

In this case the lambda function returns a “list” of values for each input element

# FlatMap transformation: Example 1

---

```
.....  
  
# Read the content of the input textual file  
inputRDD = sc.textFile("document.txt")  
  
# Compute/identify the list of words occurring in document.txt  
listOfWordsRDD= inputRDD.flatMap(lambda l: l.split(' '))
```

The new RDD contains the “concatenation” of the lists obtained by applying the lambda function over all the elements of inputRDD

# FlatMap transformation: Example 1

---

```
.....  
  
# Read the content of the input textual file  
inputRDD = sc.textFile("document.txt")  
  
# Compute/identify the list of words occurring in document.txt  
listOfWordsRDD= inputRDD.flatMap(lambda l: l.split(' '))
```

The new RDD is an RDD of strings and not an RDD of lists of strings

# **Distinct transformation**

---

# Distinct transformation

---

- Goal
  - The distinct transformation is applied on one single RDD and returns a new RDD containing the list of distinct elements (values) of the “input” RDD
- Method
  - The distinct transformation is based on the **distinct()** method of the **RDD** class
  - No functions are needed in this case

# Distinct transformation

---

- **Shuffle**

- A **shuffle** operation is executed for computing the result of the **distinct** transformation
  - Data from different input partitions must be compared to remove duplicates
- The shuffle operation is used to repartition the input data
  - All the repetitions of the same input element are associated with the same output partition (in which one single copy of the element is stored)
  - A hash function assigns each input element to one of the new partitions

# Distinct transformation: Example 1

---

- Create an RDD from a textual file containing the names of a list of users
  - Each line of the input file contains one name
- Create a new RDD containing the list of distinct names occurring in the input file
  - The type of the new RDD is the same of the “input” RDD

# Distinct transformation: Example 1

---

```
# Read the content of a textual input file  
inputRDD = sc.textFile("names.txt")  
  
# Select the distinct names occurring in inputRDD  
distinctNamesRDD = inputRDD.distinct()
```

# Distinct transformation: Example 2

---

- Create an RDD of integers containing the values [1, 2, 3, 3]
- Create a new RDD containing only the distinct values appearing in the “input” RDD

# Distinct transformation: Example 2

---

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
```

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

```
inputRDD = sc.parallelize(inputList)
```

```
# Compute the set of distinct words occurring in inputRDD
```

```
distinctIntRDD = inputRDD.distinct()
```

# **SortBy transformation**

---

# SortBy transformation

---

- Goal
  - The sortBy transformation is applied on one RDD and returns a new RDD containing the same content of the input RDD sorted in ascending order
- Method
  - The sortBy transformation is based on the **sortBy(keyfunc)** method of the **RDD** class
    - Each element of the input RDD is initially mapped to a new value by applying the specified function **keyfunc**
    - The input elements are sorted by considering the values returned by the invocation of **keyfunc** on the input values

# SortBy transformation

---

- The `sortBy(keyfunc, ascending)` method of the **RDD** class allows specifying if the values in the returned RDD are sorted in ascending or descending order by using the Boolean parameter **ascending**
  - **ascending** set to True = ascending
  - **ascending** set to False = descending

# SortBy transformation: Example 1

---

- Create an RDD from a textual file containing the names of a list of users
  - Each line of the input file contains one name
- Create a new RDD containing the list of users sorted by name (based on the alphabetic order)

# SortBy transformation: Example 1

---

```
# Read the content of a textual input file  
inputRDD = sc.textFile("names.txt")  
  
# Sort the content of the input RDD by name.  
# Store the sorted result in a new RDD  
sortedNamesRDD = inputRDD.sortBy(lambda name: name)
```

# SortBy transformation: Example 1

```
# Read the content of a textual input file  
inputRDD = sc.textFile("names.txt")  
  
# Sort the content of the input RDD by name.  
# Store the sorted result in a new RDD  
sortedNamesRDD = inputRDD.sortBy(lambda name: name)
```

Each input element is a string.  
We are interested in sorting the input names  
(strings) in alphabetic order, which is the standard  
sort order for strings.  
For this reason the lambda function returns the  
input strings without modifying them.

# SortBy transformation: Example 2

---

- Create an RDD from a textual file containing the names of a list of users
  - Each line of the input file contains one name
- Create a new RDD containing the list of users sorted by the length of their name (i.e., the sort order is based on `len(name)`)

# SortBy transformation: Example 2

---

```
# Read the content of a textual input file  
inputRDD = sc.textFile("names.txt")  
  
# Sort the content of the input RDD by name.  
# Store the sorted result in a new RDD  
sortedNamesLenRDD = inputRDD.sortBy(lambda name: len(name))  
in case having same length:  
lambda name: (len(name), name)
```

# SortBy transformation: Example 2

```
# Read the content of a textual input file  
inputRDD = sc.textFile("names.txt")  
  
# Sort the content of the input RDD by name.  
# Store the sorted result in a new RDD  
sortedNamesLenRDD = inputRDD.sortBy(lambda name: len(name))
```

Each input element is a string but we are interested in sorting the input names (strings) by length (integer), which is not the standard sort order for strings.

For this reason the lambda function returns the length of each input string. The sort operation is performed on the returned integer values (the lengths of the input names).

# Sample transformation

---

# Sample transformation

---

- Goal
  - The sample transformation is applied on one single RDD and returns a new RDD containing a random sample of the elements (values) of the “input” RDD
- Method
  - The sample transformation is based on the **sample(withReplacement, fraction)** method of **RDD** class
    - withReplacement specifies if the random sample is with replacement (true) or not (false)
    - fraction specifies the expected size of the sample as a fraction of the “input” RDD's size (values in the range [0, 1])

# Sample transformation: Example 1

---

- Create an RDD from a textual file containing a set of sentences
  - Each line of the file contains one sentence
- Create a new RDD containing a random sample of sentences
  - Use the “without replacement” strategy
  - Set fraction to 0.2 (i.e., 20%)

# Sample transformation: Example 1

---

```
# 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)
```

# Sample transformation: Example 2

---

- Create an RDD of integers containing the values [1, 2, 3, 3]
- Create a new RDD containing a random sample of the input values
  - Use the “replacement” strategy
  - Set fraction to 0.2

# Sample transformation: Example 2

---

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
```

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

```
inputRDD = sc.parallelize(inputList)
```

```
# Create a sample of the inputRDD
```

```
randomSentencesRDD = inputRDD.sample(True,0.2)
```

# **Set transformations**

---

# Set transformations

---

- Spark provides also a set of transformations that operate on two input RDDs and return a new RDD
- Some of them implement standard set transformations
  - Union any single RDD can be partitioned in several chunks and each server can have one or more of these partitions
  - Intersection
  - Subtract
  - Cartesian

# Set transformations

---

- All these transformations have
  - Two input RDDs
    - One is the RDD on which the method is invoked
    - The other RDD is passed as parameter to the method
  - One output RDD
- All the involved RDDs have the same data type when union, intersection, or subtract are used
- “Mixed” data types can be used with the cartesian transformation

# Union transformation

---

- The union transformation is based on the **union(other)** method of the **RDD** class
  - **other** is the second RDD we want to use
  - It returns a new RDD containing the union (with duplicates) of the elements of the two input RDDs
  - **Duplicates elements are not removed**
    - This choice is related to optimization reasons
      - Removing duplicates means having a global view of the whole content of the two input RDDs
      - Since each RDD is split in partitions that are stored in different nodes of the cluster, the contents of all partitions should be “shared” to remove duplicates → Computational costly operation
    - The shuffle operation is not needed in this case

# Union transformation

---

- If you really need to union two RDDs and remove duplicates you can apply the `distinct()` transformation on the output of the `union()` transformation
  - But pay attention that **distinct() is a computational costly operation**
    - It is associated with a shuffle operation
  - Use `distinct()` if and only if duplicate removal is indispensable for your application

# Intersection transformation

---

- The intersection transformation is based on the **intersection(other)** method of the **RDD** class
  - **other** is the second RDD we want to use
  - It returns a new RDD containing the elements (**without duplicates**) of the elements occurring in both input RDDs
  - A **shuffle** operation is executed for computing the result of **intersection**
    - Elements from different input partitions must be compared to find common elements

# Subtract transformation

---

- The subtract transformation is based on the **subtract(other)** method of the **RDD** class
  - **other** is the second RDD we want to use
  - The result contains the elements appearing only in the RDD on which the subtract method is invoked
    - In this transformation the two input RDDs play different roles
  - **Duplicates are not removed**
  - A **shuffle** operation is executed for computing the result of **subtract**
    - Elements from different input partitions must be compared

# Cartesian transformation

---

- The cartesian transformation is based on the **cartesian(other)** method of the **RDD** class
  - The **data types** of the objects of the two “input” RDDs can be **different**
  - The **returned** RDD is an RDD of **pairs** (tuples) containing all the combinations composed of one element of the first input RDD and one element of the second input RDD
    - We will see later what an RDD of pairs is

# Cartesian transformation

---

- A large amount of data is sent on the network
  - Elements from different input partitions must be combined to compute the returned pairs
    - The elements of the two input RDDs are stored in different partitions, which could be in different servers

# Set transformations: Example 1

---

- Create two RDDs of integers
  - `inputRDD1` contains the values [1, 2, 2, 3, 3]
  - `inputRDD2` contains the values [3, 4, 5]
- Create three new RDDs
  - `outputUnionRDD` contains the union of `inputRDD1` and `inputRDD2`
  - `outputIntersectionRDD` contains the intersection of `inputRDD1` and `inputRDD2`
  - `outputSubtractRDD` contains the result of `inputRDD1 \ inputRDD2`

# Set transformations: Example 1

---

```
# 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 new RDDs by using union, intersection, and subtract
outputUnionRDD = inputRDD1.union(inputRDD2)

outputIntersectionRDD = inputRDD1.intersection(inputRDD2)

outputSubtractRDD = inputRDD1.subtract(inputRDD2)
```

# Cartesian transformation: Example 1

---

- Create two RDDs of integers
  - inputRDD1 contains the values [1, 2, 2, 3, 3]
  - inputRDD2 contains the values [3, 4, 5]
- Create a new RDD containing the cartesian product of inputRDD1 and inputRDD2

# Cartesian transformation: Example 1

---

```
# 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)
```

# Cartesian transformation: Example 1

---

```
# 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)
```

Each element of the returned RDD is a pair (tuple) of integer elements

# Cartesian transformation: Example 2

---

- Create two RDDs
  - inputRDD1 contains the Integer values [1, 2, 3]
  - inputRDD2 contains the String values ["A", "B"]
- Create a new RDD containing the cartesian product of inputRDD1 and inputRDD2

# Cartesian transformation: Example 2

---

```
# 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)
```

# Cartesian transformation: Example 2

---

```
# 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)
```

Each element of the returned RDD is a pair (tuple) containing an integer and string

# **Basic transformations: Summary**

---

# Basic transformations based on one single RDD: Summary

---

- All the examples reported in the following tables are applied on an RDD of integers containing the following elements (i.e., values)
  - [1, 2, 3, 3]

# Basic transformations based on one single RDD: Summary

Transformation	Purpose	Example of applied function	Result
filter(f)	Return an RDD consisting only of the elements of the “input” RDD that pass the condition passed to filter(). The “input” RDD and the new RDD have the same data type.	filter(lambda x: x != 1)	[2,3,3]
map(f)	Apply a function to each element in the RDD and return an RDD of the result. The applied function return one element for each element of the “input” RDD. The “input” RDD and the new RDD can have a different data type.	map(lambda x: x+1)  For each input element x, the element with value $x+1$ is included in the new RDD	[2,3,4,4]

# Basic transformations based on one single RDD: Summary

Transformation	Purpose	Example of applied function	Result
flatMap(f)	<p>Apply a function to each element in the RDD and return an RDD of the result.</p> <p>The applied function return a set of elements (from 0 to many) for each element of the “input” RDD.</p> <p>The “input” RDD and the new RDD can have a different data type.</p>	<p>flatMap(lambda x: list(range(x,4)))</p> <p>For each input element <math>x</math>, the set of elements with values from <math>x</math> to 3 are returned</p>	[1,2,3,2,3,3]

# Basic transformations based on one single RDD: Summary

Transformation	Purpose	Example of applied function	Result
distinct()	Remove duplicates	distinct()	[1, 2, 3]
sortBy(keyfunc)	Return a new RDD containing the same values of the input RDD sorted in ascending order	sortBy(lambda v: v)  Sort the input integer values in ascending order by using the standard integer sort order	[1, 2, 3, 3]
sample(withReplacement, 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.	sample(True, 0.2)	Non deterministic

# Basic transformations based on two RDDs: Summary

---

- All the examples reported in the following tables are applied on the following two RDDs of integers
  - inputRDD1 [1, 2, 2, 3, 3]
  - inputRDD2 [3, 4, 5]

# Basic transformations based on two RDDs: Summary

Transformation	Purpose	Example	Result
union(other)	<p>Return a new RDD containing the union of the elements of the “input” RDD and the elements of the one passed as parameter to union().</p> <p>Duplicate values are not removed.</p> <p>All the RDDs have the same data type.</p>	inputRDD1.union (inputRDD2)	[1, 2, 2, 3, 3, 3, 4, 5]
intersection(other)	<p>Return a new RDD containing the intersection of the elements of the “input” RDD and the elements of the one passed as parameter to intersection().</p> <p>All the RDDs have the same data type.</p>	inputRDD1.intersection (inputRDD2)	[3]

# Basic transformations based on two RDDs: Summary

Transformation	Purpose	Example	Result
subtract(other)	<p>Return a new RDD the elements appearing only in the “input” RDD and not in the one passed as parameter to subtract().</p> <p>All the RDDs have the same data type.</p>	inputRDD1.subtract(inputRDD2)	[1, 2, 2]
cartesian(other)	<p>Return a new RDD containing the cartesian product of the elements of the “input” RDD and the elements of the one passed as parameter to cartesian().</p> <p>All the RDDs have the same data type.</p>	inputRDD1.cartesian(inputRDD2)	[(1, 3), (1, 4), ..., (3, 5)]

# **Basic Actions**

---

# Basic RDD actions

---

- Spark actions can **retrieve** the content of an RDD or the result of a function applied on an RDD and
  - “**Store**” it in a **local Python variable** of the **Driver** program
    - **Pay attention to the size of the returned value**
    - **Pay attentions that date are sent on the network from the nodes containing the content of RDDs and the executor running the Driver**
  - Or **store** the content of an RDD in an **output folder** or **database**

# Basic RDD actions

---

- The spark actions that return a result that is stored in **local (Python) variables of the Driver**
  1. Are executed locally on each node containing partitions of the RDD on which the action is invoked
    - Local results are generated in each node
  2. Local results are sent on the network to the Driver that computes the final result and store it in local variables of the Driver
- The **basic actions** returning (Python) objects to the Driver are
  - `collect()`, `count()`, `countByValue()`, `take()`, `top()`,  
`takeSample()`, `reduce()`, `fold()`, `aggregate()`, `foreach()`

# **Collect action**

---

# Collect action

---

- Goal
  - The `collect` action returns a `local Python list` of objects containing the same objects of the considered RDD
  - **Pay attention to the size of the RDD**
  - **Large RDD cannot be memorized in a local variable of the Driver**
- Method
  - The collect action is based on the `collect()` method of the `RDD` class

# Collect action: Example 1

---

- Create an RDD of integers containing the values [1, 2, 3, 3]
- Retrieve the values of the created RDD and store them in a local python list that is instantiated in the Driver

# Collect action: Example 1

---

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

# Retrieve the elements of the inputRDD and store them in
# a local python list
retrievedValues = inputRDD.collect()
```

# Collect action: Example 1

---

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
```

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

```
inputRDD = sc.parallelize(inputList)
```

```
# Retrieve the elements of the inputRDD and store them in
```

```
# a local python list
```

```
retrievedValues = inputRDD.collect()
```

inputRDD is distributed across the nodes of the cluster.  
It can be large and it is stored in the local disks of the nodes  
if it is needed



# Collect action: Example 1

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
```

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

```
inputRDD = sc.parallelize(inputList)
```

```
# Retrieve the elements of the inputRDD and store them in
```

```
# a local python list
```

```
retrievedValues = inputRDD.collect()
```

retrievedValues is a local python variable.

It can only be stored in the main memory of the process/task associated with the Driver.

**Pay attention to the size of the list.**

**Use the collect() action if and only if you are sure that the list is small.**

**Otherwise, store the content of the RDD in a file by using the**

**saveAsTextFile method**

# **Count action**

---

# Count action

---

- Goal
  - Count the number of elements of an RDD
- Method
  - The count action is based on the `count()` method of the **RDD** class
  - It returns the number of elements of the input RDD

# Count action: Example

---

- Consider the textual files “document1.txt” and “document2.txt”
- Print the name of the file with more lines

# Count action: Example

---

```
# 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")
```

# **CountByValue action**

---

# CountByValue action

---

- Goal
  - The countByValue action returns a local python dictionary containing the information about the number of times each element occurs in the RDD
    - The keys of the dictionary are associated with the input elements
    - The values are the frequencies of the elements
- Method
  - The countByValue action is based on the `countByValue()` method of the `RDD` class
  - The amount of used main memory in the Driver is related to the number of distinct elements/keys

# CountByValue action: Example 1

---

- Create an RDD from a textual file containing the first names of a list of users
  - Each line contain one name
- Compute the number of occurrences of each name and “store” this information in a local variable of the Driver

# CountByValue action: Example 1

---

```
# Read the content of the input textual file  
namesRDD = sc.textFile("names.txt")
```

```
# Compute the number of occurrences of each name  
namesOccurrences = namesRDD.countByValue()
```

in order to save this local python variable in local file system, first we need to change it to RDD by parallelize() method and then use saveToText to save in local file system of driver server

# CountByValue action: Example 1

```
# Read the content of the input textual file  
namesRDD = sc.textFile("names.txt")
```

```
# Compute the number of occurrences of each name  
namesOccurrences = namesRDD.countByValue()
```

Also in this case, pay attention to the size of the returned dictionary (that is related to the number of distinct names in this case).

Use the countByValue() action if and only if you are sure that the returned dictionary is small.

Otherwise, use an appropriate chain of Spark's transformations and write the final result in a file by using the saveAsTextFile method.

# **Take action**

---

# Take action

---

- Goal
  - The `take(num)` action returns a local python list of objects containing the first **num** elements of the considered RDD
    - The **order** of the elements in an RDD is **consistent** with the order of the elements in the file or collection that has been used to create the RDD
- Method
  - The take action is based on the **take(num)** method of the **RDD** class

# Take action: Example

---

- Create an RDD of integers containing the values [1, 5, 3, 3, 2]
- Retrieve the first two values of the created RDD and store them in a local python list that is instantiated in the Driver

# Take action: Example

---

```
# Create an RDD of integers. Load the values 1, 5, 3, 3, 2 in this RDD
inputList = [1, 5, 3, 3, 2]
inputRDD = sc.parallelize(inputList)

# Retrieve the first two elements of the inputRDD and store them in
# a local python list
retrievedValues = inputRDD.take(2)
```

# **First action**

---

# First action

---

- Goal
  - The `first()` action returns **a local python object** containing the first element of the considered RDD
    - The order of the elements in an RDD is consistent with the order of the elements in the file or collection that has been used to create the RDD
- Method
  - The first action is based on the `first()` method of the **RDD** class

# First vs Take(1)

---

- The only difference between first() and take(1) is given by the fact that
  - first() returns a **single element**
    - The returned element is the first element of the RDD
  - take(1) returns a **list** of elements **containing one single element**
    - The only element of the returned list is the first element of the RDD

# Top action

---

# Top action

---

- Goal
  - The `top(num)` action returns a **local python list of objects** containing the top **num (largest)** elements of the considered RDD
    - The ordering is the default one of class associated with the objects stored in the RDD
    - The **descending** order is used
- Method
  - The top action is based on the **top(num)** method of the **RDD** class

# Top action: Example

---

- Create an RDD of integers containing the values [1, 5, 3, 4, 2]
- Retrieve the top-2 greatest values of the created RDD and store them in a local python list that is instantiated in the Driver

# Top action: Example

---

```
# Create an RDD of integers. Load the values 1, 5, 3, 4, 2 in this RDD
inputList = [1, 5, 3, 4, 2]
inputRDD = sc.parallelize(inputList)

# Retrieve the top-2 elements of the inputRDD and store them in
# a local python list
retrievedValues = inputRDD.top(2)
```

# Top action: personalized “sorting”

---

- Goal
  - The `top(num, key)` action returns a local python list of objects containing the `num` largest elements of the considered RDD sorted by considering a user specified “sorting” function
- Method
  - The top action is based on the `top(num, key)` method of the `RDD` class
    - `num` is the number of elements to be selected
    - `key` is a function that is applied on each input element before comparing them
      - The comparison between elements is based on the values returned by the invocations of this function

# Top action: Example

---

- Create an RDD of strings containing the values ['Paolo', 'Giovanni', 'Luca']
- Retrieve the 2 longest names (longest strings) of the created RDD and store them in a local python list that is instantiated in the Driver

# Top action: Example

---

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

# **TakeOrdered action**

---

# TakeOrdered action

---

- Goal
  - The `takeOrdered(num)` action returns a local python list of objects containing the **num** smallest elements of the considered RDD
    - The ordering is the default one of class associated with the objects stored in the RDD
    - The **ascending** order is used
- Method
  - The `takeOrdered` action is based on the **takeOrdered (num)** method of the **RDD** class

# TakeOrdered action: Example

---

- Create an RDD of integers containing the values [1, 5, 3, 4, 2]
- Retrieve the 2 smallest values of the created RDD and store them in a local python list that is instantiated in the Driver

# TakeOrdered action: Example

---

```
# Create an RDD of integers. Load the values 1, 5, 3, 4, 2 in this RDD
```

```
inputList = [1, 5, 3, 4, 2]
```

```
inputRDD = sc.parallelize(inputList)
```

```
# Retrieve the 2 smallest elements of the inputRDD and store them in
```

```
# a local python list
```

```
retrievedValues = inputRDD.takeOrdered(2)
```

# TakeOrdered action: personalized “sorting”

---

- Goal
  - The takeOrdered(num, key) action returns a local python list of objects containing the **num** smallest elements of the considered RDD sorted by considering a user specified “sorting” function
- Method
  - The takeOrdered action is based on the **takeOrdered (num, key)** method of the **RDD** class
    - **num** is the number of elements to be selected
    - **key** is a function that is applied on each input element before comparing them
      - The comparison between elements is based on the values returned by the invocations of this function

# TakeOrdered action: Example

---

- Create an RDD of strings containing the values ['Paolo', 'Giovanni', 'Luca']
- Retrieve the 2 shortest names (shortest strings) of the created RDD and store them in a local python list that is instantiated in the Driver

# TakeOrdered action: Example

---

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

# TakeSample action

---

# TakeSample action

---

- Goal
  - The `takeSample(withReplacement, num)` action returns a **local python list of objects** containing **num** random elements of the considered RDD
- Method
  - The `takeSample` action is based on the **`takeSample(withReplacement, num)`** method of the **RDD** class
    - `withReplacement` specifies if the random sample is with replacement (True) or not (False)

# TakeSample action

---

- Method
  - The `takeSample(withReplacement, num, seed)` method of the `RDD` class is used when we want to set the seed

# TakeSample action: Example

---

- Create an RDD of integers containing the values [1, 5, 3, 3, 2]
- Retrieve randomly, without replacement, 2 values from the created RDD and store them in a local python list that is instantiated in the Driver

# TakeSample action: Example

---

```
# Create an RDD of integers. Load the values 1, 5, 3, 3, 2 in this RDD  
inputList = [1, 5, 3, 3, 2]  
inputRDD = sc.parallelize(inputList)
```

```
# Retrieve randomly two elements of the inputRDD and store them in  
# a local python list  
randomValues= inputRDD.takeSample(True, 2)
```

# **Reduce action**

---

# Reduce action

---

- Goal
  - Return **a single python object** obtained by **combining** all the objects of the input RDD by using a user provide “function”
    - The provided “function” must be **associative** and **commutative**
      - otherwise the result depends on the content of the partitions and the order used to analyze the elements of the RDD’s partitions
    - The returned object and the ones of the “input” RDD are **all instances of the same data type/class**

# Reduce action

---

- Method
  - The reduce action is based on the **reduce(f)** method of the **RDD** class
  - A function **f** is passed to the reduce method
    - Given two arbitrary input elements, **f** is used to combine them in one single value
    - **f** is **recursively** invoked over the elements of the input RDD until the input values are “reduced” to one single value

# Reduce action: how it works

---

- Suppose  $L$  contains the list of elements of the “input” RDD
- To compute the final element/value, the reduce action operates as follows
  1. Apply the user specified “function” on a pair of elements  $e_1$  and  $e_2$  occurring in  $L$  and obtain a new element  $e_{new}$
  2. Remove the “original” elements  $e_1$  and  $e_2$  from  $L$  and then insert the element  $e_{new}$  in  $L$
  3. If  $L$  contains only one value then return it as final result of the reduce action.  
Otherwise, return to step 1

# Reduce action: how it works

---

- “Function”  $f$  must be associative and commutative
  - The computation of the reduce action can be performed in parallel without problems

# Reduce action: how it works

---

- “Function”  $f$  must be associative and commutative
  - The computation of the reduce action can be performed in parallel without problems
- Otherwise the result depends on how the input RDD is partitioned
  - i.e., for the functions that are not associative and commutative the output depends on how the RDD is split in partitions and how the content of each partition is analyzed

# Reduce action: Example 1

---

- Create an RDD of integers containing the values [1, 2, 3, 3]
- Compute the sum of the values occurring in the RDD and “store” the result in a local python integer variable in the Driver

# Reduce action: Example 1

---

.....

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
```

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

```
inputRDDReduce = sc.parallelize(inputListReduce)
```

```
# Compute the sum of the values
```

```
sumValues = inputRDDReduce.reduce(lambda e1, e2: e1+e2)
```

# Reduce action: Example 1

---

.....

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD  
inputListReduce = [1, 2, 3, 3]  
inputRDDReduce = sc.parallelize(inputListReduce)
```

```
# Compute the sum of the values
```

```
sumValues = inputRDDReduce.reduce(lambda e1, e2: e1+e2)
```

This lambda function combines two input integer elements at a time and returns their sum

# Reduce action: Example 2

---

- Create an RDD of integers containing the values [1, 2, 3, 3]
- Compute the maximum value occurring in the RDD and “store” the result in a local python integer variable in the Driver

# Reduce action: Example 2

---

.....

```
# Define the function for the reduce action
def computeMax(v1,v2):
    if v1>v2:
        return v1
    else:
        return v2

# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputListReduce = [1, 2, 3, 3]
inputRDDReduce = sc.parallelize(inputListReduce)

# Compute the maximum value
maxValue = inputRDDReduce.reduce(computeMax)
```

# Reduce action: Example 2 - ver. 2

---

.....

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputListReduce = [1, 2, 3, 3]
inputRDDReduce = sc.parallelize(inputListReduce)

# Compute the maximum value
maxValue = inputRDDReduce.reduce(lambda e1, e2: max(e1, e2))
```

# **Fold action**

---

# Fold action

it is based on order in opposite of reduce method using random selection

---

## ■ Goal

- Return **a single python object** obtained by combining all the objects of the input RDD and a **“zero” value** by using a user provide “function”
  - The provided “function”
    - Must be **associative**
      - Otherwise the result depends on how the RDD is partitioned
      - It is **not required to be commutative**
    - An initial neutral “zero” value is also specified

# Fold action

---

- Method
  - The fold action is based on the **fold(zeroValue, op)** method of the **RDD** class
  - A function **op** is passed to the fold method
    - Given two arbitrary input elements, **op** is used to combine them in one single value
    - **op** is also used to combine input elements with the “zero” value
    - **op** is recursively invoked over the elements of the input RDD until the input values are “reduced” to one single value
  - The “**zero**” value is the **neutral value** for the used function **op**
    - i.e., “zero” combined with any value **v** by using **op** is equal to **v**

# Fold action: Example 1

---

- Create an RDD of strings containing the values ['This ', 'is ', 'a ', 'test']
- Compute the concatenation of the values occurring in the RDD (from left to right) and “store” the result in a local python string variable in the Driver

it is associative but not commutative

# Fold action: Example 1

---

.....

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
```

```
inputListFold = ['This', 'is', 'a', 'test']
```

```
inputRDDFold = sc.parallelize(inputListFold)
```

```
# Concatenate the input strings
```

```
finalString = inputRDDFold.fold("", lambda s1, s2: s1+s2)
```

[see video](#)

# Fold vs Reduce

---

- Fold is characterized by the “zero” value
- Fold can be used to parallelize functions that are associative but non-commutative
  - E.g., concatenation of a list of strings

# **Aggregate action**

---

# Aggregate action

---

- Goal
  - Return **a single python object** obtained by combining the objects of the RDD and an initial “zero” value by using two user provide “functions”
    - The provided “functions” must be **associative**
      - Otherwise the result depends on how the RDD is partitioned
    - The **returned objects** and the **ones of the “input” RDD** can be instances of **different classes**
      - This is the main difference with respect to reduce () and fold()

# Aggregate action

---

- Method
  - The aggregate action is based on the **aggregate(zeroValue, seqOp, combOp)** method of the **RDD** class
  - The “input” RDD contains objects of type T while the returned object is of type U ( $T \neq U$ )
    - We need one “function” for **merging** an element of type **T** with an element of type **U** to return a new element of type **U**
      - It is used to merge the elements of the input RDD and the accumulator of each partition
    - We need one “function” for **merging two** elements of type **U** to return a new element of type **U**
      - It is used to merge two elements of type **U** obtained as partial results generated by two different partitions

# Aggregate action

---

- The **seqOp** function contains the code that is applied to combine the accumulator value (one accumulator for each partition) with the elements of each partition
  - One “local” result per partition is computed by recursively applying **seqOp**
- The **combOp** function contains the code that is applied to combine two elements of type U returned as partial results by two different partitions
  - The global final result is computed by recursively applying **combOp**

# Aggregate action: how it works

---

- Suppose that  $L$  contains the list of elements of the “input” RDD and this RDD is split in a set of partitions, i.e., a set of lists  $\{L_1, \dots, L_n\}$
- The aggregate action computes a partial result in each partition and then combines/merges the results.
- It operates as follows
  1. Aggregate the partial results in each partition, obtaining a set of partial results (of type U)  $P = \{p_1, \dots, p_n\}$
  2. Apply the **combOp** function on a pair of elements  $p_1$  and  $p_2$  in  $P$  and obtain a new element  $p_{new}$
  3. Remove the “original” elements  $p_1$  and  $p_2$  from  $P$  and then insert the element  $p_{new}$  in  $P$
  4. If  $P$  contains only one value then return it as final result of the aggregate action. Otherwise, return to step 2

# Aggregate action: how it works

---

- Suppose that
  - $L_i$  is the list of elements on the  $i$ -th partition of the “input” RDD
  - And **zeroValue** is the initial zero value
- To compute the partial result over the elements in  $L_i$ , the aggregate action operates as follows
  1. Set **accumulator** to **zeroValue** (**accumulator**=**zeroValue**)
  2. Apply the **seqOp** function on **accumulator** and an elements  $e_j$  in  $L_i$  and update **accumulator** with the value returned by **seqOp**
  3. Remove the “original” elements  $e_j$  from  $L_i$
  4. If  $L_i$  is empty return **accumulator** as (final) partial result  $p_i$  of the  $i$ -th partition. Otherwise, return to step 2

# Aggregate action: Example 1

---

- Create an RDD of integers containing the values [1, 2, 3, 3]
- Compute both
  - the sum of the values occurring in the input RDD
  - and the number of elements of the input RDD
- Finally, “store” in a local python variable of the Driver the average computed over the values of the input RDD

# Aggregate action: Example 1

---

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputListAggr = [1, 2, 3, 3]
inRDD = sc.parallelize(inputListAggr)

# Instantiate the zero value
# We use a tuple containing two values:
#   (sum, number of represented elements)
zeroValue = (0, 0)

# Compute the sum of the elements in inputRDDAggr and count them
sumCount = inRDD.aggregate(zeroValue, \
                           lambda acc, e: (acc[0]+e, acc[1]+1), \
                           lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))
```

# Aggregate action: Example 1

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD
inputListAggr = [1, 2, 3, 3]
inRDD = sc.parallelize(inputListAggr)

# Instantiate the zero value
# We use a tuple
#   (sum, number of represented elements)
zeroValue = (0, 0)

# Compute the sum of the elements in inputRDDAggr and count them
sumCount = inRDD.aggregate(zeroValue, \
                            lambda acc, e: (acc[0]+e, acc[1]+1), \
                            lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))
```

# Aggregate action: Example 1

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD  
inputListAggr = [1, 2, 3, 3]  
inRDD = sc.parallelize(inputListAggr)
```

```
# Instantiate the zero value
```

Given a partition  $p$  of the input RDD, this is the function that is used to combine the elements of partition  $p$  with the accumulator of partition  $p$ .

- acc is a tuple object (it is initially initialized to the zero value)
- e is an integer

```
# Compute the sum of the elements in inputRDDAggr and count them  
sumCount = inRDD.aggregate(zeroValue, \  
    lambda acc, e: (acc[0]+e, acc[1]+1), \  
    lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))
```

# Aggregate action: Example 1

```
# Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD  
inputListAggr = [1, 2, 3, 3]  
inRDD = sc.parallelize(inputListAggr)
```

```
# Instantiate the zero value
```

```
# We use a tuple containing two values:
```

This is the function that is used to combine the partial results emitted by the RDD's partitions.

- p1 and p2 are tuple objects

```
# Compute the sum of the elements in inputRDDAggr and count them  
sumCount = inRDD.aggregate(zeroValue, \
```

```
    lambda acc, e: (acc[0]+e, acc[1]+1), \
```

```
    lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))
```

# Aggregate action: Example 1

---

```
# Compute the average value
```

```
myAvg = sumCount[0]/sumCount[1]
```

```
# Print the average on the standard output of the driver
```

```
print('Average:', myAvg)
```

# Aggregate action: Simulation

---

- inRDD = [1, 2, 3, 3]
- Suppose inRDD is split in the following two partitions
  - [1, 2] and [3, 3]

# Aggregate action: Simulation

---

Partition #1

[1, 2] acc=(o,o)

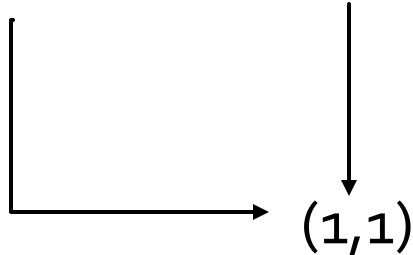
Partition #2

[3, 3] acc=(o,o)

# Aggregate action: Simulation

Partition #1

[1, 2] acc=(o,o)



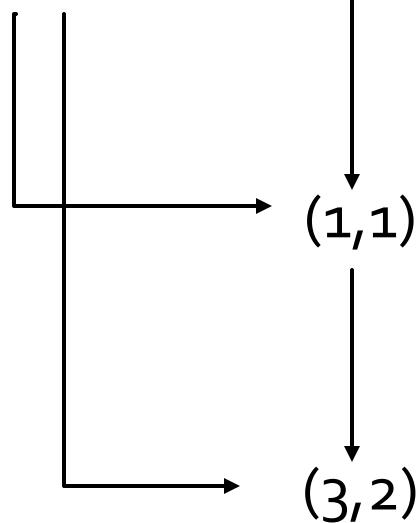
Partition #2

[3, 3] acc=(o,o)

# Aggregate action: Simulation

Partition #1

[1, 2] acc=(o,o)



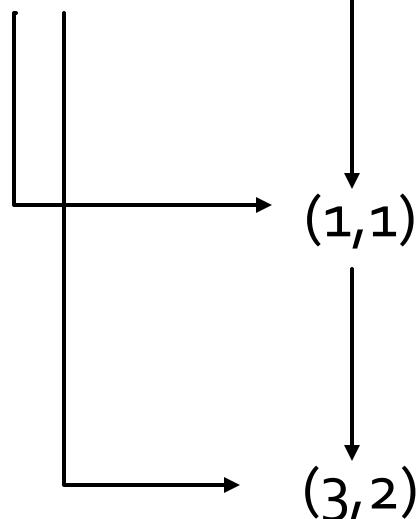
Partition #2

[3, 3] acc=(o,o)

# Aggregate action: Simulation

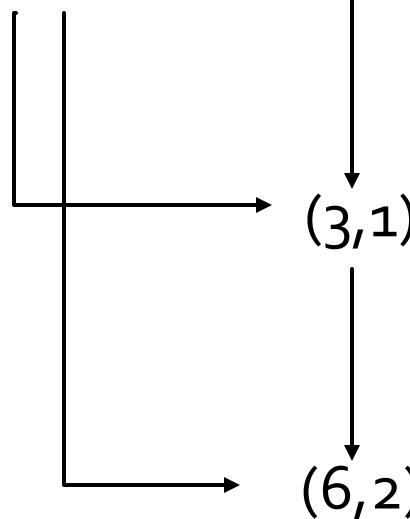
Partition #1

[1, 2] acc=(o,o)

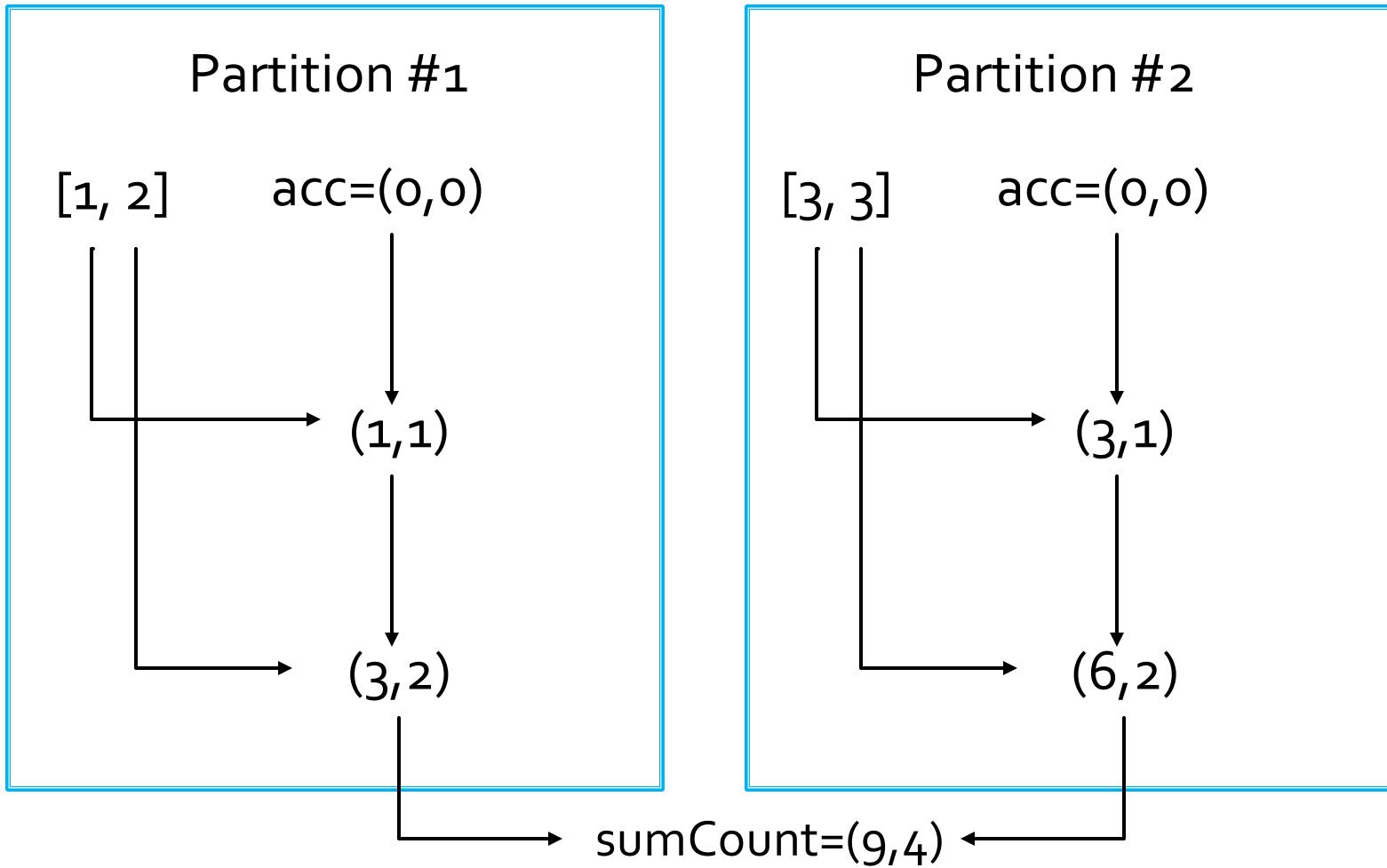


Partition #2

[3, 3] acc=(o,o)



# Aggregate action: Simulation



# **Basic actions: Summary**

---

# Basic actions: Summary

---

- All the examples reported in the following tables are applied on inputRDD that is an RDD of integers containing the following elements (i.e., values)
  - [1, 2, 3, 3]

# Basic actions: Summary

Action	Purpose	Example	Result
collect()	<p>Return a python list containing all the elements of the RDD on which it is applied.</p> <p>The objects of the RDD and objects of the returned list are objects of the same class.</p>	inputRDD.collect()	[1,2,3,3]
count()	Return the number of elements of the RDD	inputRDD.count()	4
countByValue()	Return a Map object containing the information about the number of times each element occurs in the RDD.	inputRDD.countByValue()	[(1, 1), (2, 1), (3, 2)]

# Basic actions: Summary

Action	Purpose	Example	Result
take(num)	<p>Return a Python list containing the first num elements of the RDD.</p> <p>The objects of the RDD and objects of the returned list are objects of the same class.</p>	inputRDD.take(2)	[1,2]
first()	Return the first element of the RDD	first()	1
top(num)	<p>Return a Python list containing the top num elements of the RDD based on the default sort order/comparator of the objects.</p> <p>The objects of the RDD and objects of the returned list are objects of the same class.</p>	inputRDD.top(2)	[3,3]

# Basic actions: Summary

Action	Purpose	Example	Result
takeSample(withReplacement, num)	Return a (Python) List containing a random sample of size n of the RDD.	inputRDD.takeSample(False, 1)	Nondeterministic
takeSample(withReplacement, num, seed)	The objects of the RDD and objects of the returned list are objects of the same class.		
reduce(f)	Return a single Python object obtained by combining the values of the objects of the RDD by using a user provided “function”. The provided “function” must be associative and commutative The object returned by the method and the objects of the RDD belong to the same class.	inputRDD.reduce(lambda e1, e2: e1+e2)  The passed “function” is the sum	9

# Basic actions: Summary

Action	Purpose	Example	Result
fold(zeroValue, op)	Same as reduce but with the provided zero value.	inputRDD. fold(o, lambda v1, v2: v1+v2)  The passed “function” is the sum and the passed zeroValue is o	9
Aggregate(zeroValue, seqOp, combOp)	Similar to reduce() but used to return a different type.	inputRDD.aggregate (zeroValue, lambda acc, e: (acc[0]+e, acc[1]+1), lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))  Compute a pair of integers where the first one is the sum of the values of the RDD and the second the number of elements	(9, 4)

# RDDs and key-value pairs

---

# RDDs of key-value pairs

---

- Spark supports also RDDs of key-value pairs
  - Key-value pairs in python are represented by means of python tuples
    - The first value is the **key** part of the pair
    - The second value is the **value** part of the pair
- RDDs of key-value pairs are sometimes called “**pair RDDs**”

# RDDs of key-value pairs

---

- RDDs of key-value pairs are characterized by specific operations
  - `reduceByKey()`, `join()`, etc.
  - These operations analyze the content of one group (key) at a time
- RDDs of key-value pairs are characterized also by the operations available for the “standard” RDDs
  - `filter()`, `map()`, `reduce()`, etc.

# RDDs of key-value pairs

---

- Many applications are based on RDDs of key-value pairs
- The operations available for RDDs of key-value pairs allow
  - “grouping” data by key
  - performing computation by key (i.e., by group)
- The basic idea is similar to the one of the MapReduce-based programs in Hadoop
  - But there are more operations already available

# **Creating RDDs of key-value pairs**

---

# Creating RDDs of key-value pairs

---

- RDDs of key-value pairs can be built
  - From other RDDs by applying the map() or the flatMap() transformation on other RDDs
  - From a Python in-memory collection of tuple (key-value pairs) by using the parallelize() method of the SparkContext class

# Creating Pair RDDs

---

- Key-value pairs are represented as tuples composed of two elements
  - Key
  - Value
- The standard built-in Python tuples are used

**RDDs of key-value pairs by  
using the Map transformation**

---

# RDDs of key-value pairs by using the map transformation

---

- Goal
  - Define an RDD of key-value pairs by using the map transformation
  - Apply a function **f** on each element of the input RDD that **returns one tuple for each input element**
    - The new RDD of key-value pairs contains one tuple **y** for each element **x** of the “input” RDD

# RDDs of key-value pairs by using the map transformation

---

- Method
  - The standard **map(f)** transformation is used
    - The new RDD of key-value pairs contains **one tuple y** for each input element **x** of the “input” RDD
      - $y = f(x)$

# RDDs of key-value pairs by using the map transformation: 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
- Create an RDD of key-value pairs containing a list of pairs (first name, 1)

# RDDs of key-value pairs by using the map transformation: Example

---

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

# RDDs of key-value pairs by using the map transformation: Example

```
# Read the content of the input textual file  
namesRDD = sc.textFile("first_names.txt")
```

```
# Create an RDD of key-value pairs  
nameOnePairRDD = namesRDD.map(lambda name: (name, 1))
```

nameOnePairRDD contains key-value pairs (i.e., tuples) of type (string, integer)

**RDDs of key-value pairs by using  
the flatMap transformation**

---

# RDDs of key-value pairs by using the flatMap transformation

---

- Goal
  - Define an RDD of key-value pairs by using the flatMap transformation
  - Apply a function **f** on each element of the input RDD that **returns a list of tuples** for each **input element**
    - The new PairRDD contains all the pairs obtained by applying **f** on each element **x** of the “input” RDD

# RDDs of key-value pairs by using the flatMap transformation

---

- Method
  - The standard **flatMap(f)** transformation is used
    - The new RDD of key-value pairs contains the tuples returned by the execution of **f** on each element **x** of the “input” RDD
    - **[y]=f(x)**
      - Given a element **x** of the input RDD, **f** applied on **x** returns a list of pairs **[y]**
      - The new RDD is a “list” of pairs contains all the pairs of the returned list of pairs. It is not an RDD of lists.
      - **[y]** can be the empty list

# RDDs of key-value pairs by using the flatMap transformation: Example

---

- Create an RDD from a textual file
  - Each line of the file contains a set of words
- Create a PairRDD containing a list of pairs (word, 1)
  - One pair for each word occurring in the input document (with repetitions)

# RDDs of key-value pairs by using the flatMap transformation: Example v1

```
# Define the function associated with the flatMap transformation
def wordsOnes(line):
    pairs = []
    for word in line.split(' '):
        pairs.append((word, 1))
    return pairs

# Read the content of the input textual file
linesRDD = sc.textFile("document.txt")

# Create an RDD of key-value pairs based on the input document
# One pair (word,1) for each input word
wordOnePairRDD = linesRDD.flatMap(wordsOnes)
```

# RDDs of key-value pairs by using the flatMap transformation: Example v2

---

```
# Read the content of the input textual file  
linesRDD = sc.textFile("document.txt")  
  
# Create an RDD of key-value pairs based on the input document  
# One pair (word,1) for each input word  
wordOnePairRDD = linesRDD.flatMap(lambda line: \  
                                     map(lambda w: (w, 1), line.split('')))
```

# RDDs of key-value pairs by using the flatMap transformation: Example v2

```
# Read the content of the input textual file  
linesRDD = sc.textFile("document.txt")  
  
# Create an RDD of key-value pairs based on the input document  
# One pair (word,1) for each input word  
wordOnePairRDD = linesRDD.flatMap(lambda line: \  
    map(lambda w: (w, 1), line.split('')))
```

This is the map of python.  
It is not the Spark's map transformation.

**RDDs of key-value pairs by  
using parallelize**

---

# RDDs of key-value pairs by using parallelize

---

- Goal
  - Use the parallelize method to create an RDD of key-value pairs from a local python in-memory collection of tuples
- Method
  - It is based on the standard **parallelize(c)** method of the **SparkContext** class
  - Each element (tuple) of the local python collection becomes a key-value pair of the returned RDD

# RDDs of key-value pairs by using parallelize: Example

---

- Create an RDD from a local python list containing the following key-value pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 35)

# RDDs of key-value pairs by using parallelize: Example

---

```
# 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)
```

# RDDs of key-value pairs by using parallelize: Example

---

```
# Create the local python list
```

```
nameAge = [ ("Paolo", 40), ("Giorgio", 22), ("Paolo", 35)]
```

Create a local in-memory python list of key-value pairs (tuples).

This list is stored in the main memory of the Driver.

# RDDs of key-value pairs by using parallelize: Example

```
# 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 an RDD or key-value pairs based on the content of the local in-memory python list.

The RDD is stored in the “distributed” main memory of the cluster servers

# **Transformations on RDDs of key-value pairs**

---

# Transformations on RDDs of key-value pairs

---

- All the “standard” transformations can be applied
  - Where the specified “functions” operate on tuples
- Specific transformations are available
  - E.g., `reduceByKey()`, `groupByKey()`, `mapValues()`, `join()`, ...

# **ReduceByKey transformation**

---

# ReduceByKey transformation

---

- Goal
  - Create a new RDD of key-value pairs where there is **one pair for each distinct key  $k$**  of the input RDD of key-value pairs
    - The value associated with key  $k$  in the new RDD of key-value pairs is computed by applying a function  $f$  on the values associated with  $k$  in the input RDD of key-value pairs
      - The function  $f$  must be **associative** and **commutative**
        - otherwise the result depends on how data are partitioned and analyzed
    - The **data type** of the new RDD of key-value pairs is the **same** of the “input” RDD of key-value pairs

# ReduceByKey transformation

---

- Method
  - The reduceByKey transformation is based on the **reduceByKey(f)** method of the **RDD** class
  - A function **f** is passed to the reduceByKey method
    - Given the values of two input pairs, **f** is used to combine them in one single value
    - **f** is recursively invoked over the values of the pairs associated with one key at a time until the input values associated with one key are “reduced” to one single value
  - The returned RDD contains a number of key-value pairs equal to the number of distinct keys in the input key-value pair RDD

# ReduceByKey transformation

---

- Similarly to the `reduce()` action, the `reduceByKey()` transformation aggregate values
- However,
  - `reduceByKey()` is executed on RDDs of key-value pairs and returns a set of key-value pairs
  - `reduce()` is executed on an RDD and returns one single value (stored in a local python variable)
- And
  - `reduceByKey()` is a transformation
    - `reduceByKey()` is executed lazily and its result is stored in another RDD
  - Whereas `reduce()` is an action

# ReduceByKey transformation

---

- Shuffle
  - A **shuffle** operation is executed for computing the result of the **reduceByKey()** transformation
    - The result/value for each group/key is computed from data stored in different input partitions

# ReduceByKey transformation: Example

---

- Create an RDD from a local python list containing the pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 35)
  - The key is the first name of a user and the value is his/her age
- Create a new RDD of key-value pairs containing one pair for each name. In the returned RDD, associate each name with the age of the youngest user with that name

# ReduceByKey transformation: Example

---

```
# 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 for each name the lowest age value
youngestPairRDD= nameAgePairRDD.reduceByKey(lambda age1, age2:\n                                              min(age1, age2))
```

# ReduceByKey transformation: Example

```
# 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 for each name the lowest age value
```

```
youngestPairRDD= nameAgePairRDD.reduceByKey(lambda age1, age2:\n                                              min(age1, age2))
```

The returned RDD of key-value pairs contains one pair for each  
distinct input key (i.e., for each distinct name in this example)

# **FoldByKey transformation**

---

# FoldByKey transformation

---

- Goal
  - The foldByKey() has the same goal of the reduceByKey() transformation
  - However, foldByKey()
    - Is characterized also by a “zero” value
    - Functions **must be associative** but are not required to be commutative

# FoldByKey transformation

---

- Method
  - The foldByKey transformation is based on the **foldByKey(zeroValue, op)** method of the **RDD** class
  - A function **op** is passed to the fold method
    - Given values of two input pairs, **op** is used to combine them in one single value
    - **op** is also used to combine input values with the “zero” value
    - **op** is recursively invoked over the values of the pairs associated with one key at a time until the input values are “reduced” to one single value
  - The “zero” value is the neutral value for the used function **op**
    - i.e., “zero” combined with any value **v** by using **op** is equal to **v**

# FoldByKey transformation

---

- Shuffle
  - A **shuffle** operation is executed for computing the result of the **foldByKey()** transformation
    - The result/value for each group/key is computed from data stored in different input partitions

# FoldByKey transformation: Example

---

- Create an RDD from a local python list containing the pairs
  - ("Paolo", "Message1")
  - ("Giorgio", "Message2")
  - ("Paolo", "Message3")
  - The key is the first name of a user and the value is a message published by him/her
- Create a new RDD of key-value pairs containing one pair for each name. In the returned RDD, associate each name the concatenation of its messages (preserving the order of the messages in the input RDD)

# FoldByKey transformation: Example

---

```
# Create the local python list
nameMess = [("Paolo", "Message1"), ("Giorgio", "Message2"), \
            ("Paolo", "Message3")]

# Create the RDD of pairs from the local collection
nameMessPairRDD = sc.parallelize(nameMess)

# Concatenate the messages of each user
concatPairRDD=nameMessPairRDD.foldByKey(", lambda m1, m2:\\
                                         m1+m2)
```

# **CombineByKey transformation**

---

# CombineByKey transformation

---

- Goal
  - Create a new RDD of key-value pairs where there is one pair for each distinct key **k** of the input RDD of key-value pairs
    - The value associated with the key **k** in the new RDD of key-value pairs is computed by applying **user-provided** functions on the values associated with **k** in the input RDD of key-value pairs
      - The user-provided “function” must be **associative**
        - otherwise the result depends how data are partitioned and analyzed
    - The **data type** of the new RDD of key-value pairs **can be different** with respect to the data type of the “input” RDD of key-value pairs

# CombineByKey transformation

---

- Method
  - The combineByKey transformation is based on the **combineByKey(createCombiner, mergeValue, mergeCombiner)** method of the **RDD** class
    - The values of the input RDD of pairs are of type **V**
    - The values of the returned RDD of pairs are of type **U**
    - The type of the keys is **K** for both RDDs of pairs

# CombineByKey transformation

---

- The `createCombiner` function contains the code that is used to `transform` a single value (type `V`) of the input RDD of key-value pairs into a value of the data type (type `U`) of the output RDD of key-value pairs
  - It is used to transform the first value of each key in each partition to a value of type `U`

# CombineByKey transformation

---

- The `mergeValue` function contains the code that is used to `combine` one value of type `U` with one value of type `V`
  - It is used in each partition to combine the initial values (type `V`) of each key with the intermediate ones (type `U`) of each key

# CombineByKey transformation

---

- The `mergeCombiner` function contains the code that is used to **combine two** values of type **U**
  - It is used to combine intermediate values of each key returned by the analysis of different partitions

# CombineByKey transformation

---

- `combineByKey` is more general than `reduceByKey` and `foldByKey` because the **data types of the values of the input and the returned RDD of pairs can be different**
  - For this reason, more functions must be implemented in this case

# CombineByKey transformation

---

- Shuffle
  - A **shuffle** operation is executed for computing the result of the **combineByKey()** transformation
    - The result/value for each group/key is computed from data stored in different input partitions

# CombineByKey transformation: Example

---

- Create an RDD from a local python list containing the pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 35)
  - The key is the first name of a user and the value is his/her age
- Store the results in an output HDFS folder. The output contains one line for each name followed by the average age of the users with that name

# CombineByKey transformation: Example

---

```
# 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), \
    lambda intermediateElem, inputElem: \ 
        (intermediateElem[0]+inputElem, intermediateElem[1]+1), \
    lambda intermediateElem1, intermediateElem2: \ 
        (intermediateElem1[0]+intermediateElem2[0], \
         intermediateElem1[1]+intermediateElem2[1])
)
```

# CombineByKey transformation: Example

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

Given an input value (an age), it returns a tuple containing (age, 1)

```
# Compute the sum of ages and  
# the number of input pairs for each name (key)  
sumNumPerNamePairRDD=nameAgePairRDD.combineByKey(\  
    lambda inputElem: (inputElem, 1), \  
    lambda intermediateElem, inputElem: \  
        (intermediateElem[0]+inputElem, intermediateElem[1]+1),  
    lambda intermediateElem1, intermediateElem2: \  
        (intermediateElem1[0]+intermediateElem2[0], \  
         intermediateElem1[1]+intermediateElem2[1])  
)
```

# CombineByKey transformation: Example

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

Given an input value (an age) and an intermediate value (sum ages, num represented values), it combines them and returns a new updated tuple (sum ages, num represented values)

```
# the number of input pairs for each name (key)  
sumNumPerNamePairRDD=nameAgePairRDD.combineByKey(\  
    lambda inputElem: (inputElem, 1), \  
  
    lambda intermediateElem, inputElem: \  
        (intermediateElem[0]+inputElem, intermediateElem[1]+1), \  
  
    lambda intermediateElem1, intermediateElem2: \  
        (intermediateElem1[0]+intermediateElem2[0], \  
         intermediateElem1[1]+intermediateElem2[1])  
)
```

# CombineByKey transformation: Example

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

Given two intermediate result tuples (sum ages, num represented values), it combines them and returns a new updated tuple (sum ages, num represented values)

```
# the number of input pairs for each name (key)  
sumNumPerNamePairRDD=nameAgePairRDD.combineByKey(\  
    lambda inputElem: (inputElem, 1), \  
  
    lambda intermediateElem, inputElem: \  
        (intermediateElem[0]+inputElem, intermediateElem[1]+1),  
  
    lambda intermediateElem1, intermediateElem2: \  
        (intermediateElem1[0]+intermediateElem2[0], \  
         intermediateElem1[1]+intermediateElem2[1])  
)
```

# CombineByKey transformation: Example

---

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

# CombineByKey transformation: Example

```
# 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('output')
```

Compute the average age for each key (i.e., for each name) by combining “sum ages” and “num represented values”.

Each input pair is characterized by a value that is a tuple containing (sum ages, num represented values).

# **GroupByKey transformation**

---

# GroupByKey transformation

---

- Goal
  - Create a new RDD of key-value pairs where there is **one pair for each distinct key `k`** of the input RDD of key-value pairs
    - The value associated with key `k` in the new RDD of key-value pairs is the **list of values** associated with `k` in the input RDD of key-value pairs
- Method
  - The groupByKey transformation is based on the **groupByKey()** method of the **RDD** class

# GroupByKey transformation

---

- If you are grouping values per key to perform then an aggregation such as sum or average over the values of each key then groupByKey is not the right choice
  - `reduceByKey`, `aggregateByKey` or `combineByKey` provide better performances for associative and commutative aggregations
  - `groupByKey` is useful if you need to apply an aggregation/compute a function that is not associative

# GroupByKey transformation

---

- Shuffle
  - A **shuffle** operation is executed for computing the result of the **groupByKey()** transformation
    - Each group/key is associated with/is composed of values which are stored in different partitions of the input RDD

# GroupByKey transformation: Example

---

- Create an RDD from a local python list containing the pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 35)
  - The key is the first name of a user and the value is his/her age
- Store the results in an output HDFS folder. The output contains one line for each name followed by the ages of all the users with that name

# GroupByKey transformation: Example

---

```
# 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 associated ages
```

```
agesPerNamePairRDD = nameAgePairRDD.groupByKey()
```

```
# Store the result in an output folder
```

```
agesPerNamePairRDD.mapValues(lambda listValues: list(listValues))  
    .saveAsTextFile(outputPath);
```

# GroupByKey transformation: Example

```
# 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 associated ages
```

```
agesPerNamePairRDD = nameAgePairRDD.groupByKey()
```

In this RDD of key-value pairs each tuple is composed of

- a string (key of the pair) (Paolo, Giorgio, ...))

- a “collection” of integers (the value of the pair) – a ResultIterable object

# GroupByKey transformation: Example

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

This part is used to format the content of the value part of each pair before storing the result in the output folder.

- This transforms a ResultIterable object to a Python list
- Without this “map” the output will contains the pointers to ResultIterable objects instead of a readable list of integer values

```
# Store the result in an output folder  
agesPerNamePairRDD.mapValues(lambda listValues: list(listValues))  
    .saveAsTextFile(outputPath);
```

# **MapValues transformation**

---

# MapValues transformation

---

## ■ Goal

- Apply a function **f** over the value of each pair of an input RDD or key-value pairs and return a new RDD of key-value pairs
- One pair is created in the returned RDD for each input pair
  - The **key** of the created pair is equal to the key of the input pair
  - The **value** of the created pair is obtained by applying the function **f** on the value of the input pair
- The **data type** of the values of the new RDD of key-value pairs **can be different** from the data type of the values of the “input” RDD of key-value pairs
- The **data type of the key is the same**

# MapValues transformation

---

- Method
  - The mapValues transformation is based on the **mapValues(f)** method of the **RDD** class
  - A function **f** is passed to the mapValues method
    - **f** contains the code that is applied to transform each input value into the a new value that is stored in the RDD of key-value pairs
  - The **retuned RDD of pairs** contains a **number** of key-value pairs **equal** to the number of key-value pairs of the **input RDD of pairs**
    - The key part is not changed

# MapValues transformation: Example

---

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

# MapValues transformation: Example

---

```
# 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)
```

```
# Increment age of all users
```

```
plusOnePairRDD = nameAgePairRDD.mapValues(lambda age: age+1)
```

```
# Save the result on disk
```

```
plusOnePairRDD.saveAsTextFile(outputPath)
```

# **FlatMapValues transformation**

---

# FlatMapValues transformation

---

- Goal
  - Apply a function **f** over the value of each pair of an input RDD or key-value pairs and return a new RDD of key-value pairs
    - **f** returns a **list of values for each input value**
  - A list of pairs is inserted in the returned RDD for each input pair
    - The key of the created pairs is equal to the key of the input pair
    - The values of the created pairs are obtained by applying the function **f** on the value of the input pair
  - The **data type** of the values of the new RDD of key-value pairs **can be different** from the data type of the values of the “input” RDD of key-value pairs
  - The **data type** of the **key** is the **same**

# FlatMapValues transformation

---

- Method
  - The flatMapValues transformation is based on **flatMapValues(f)** method of the **RDD** class
  - A function **f** is passed to the mapValues method
    - **f** contains the code that is applied to transform each input value into a set of new values that are stored in the new RDD of key-value pairs
  - The keys of the input pairs are not changed

# FlatMapValues transformation: Example

---

- Create an RDD from a local python list containing the pairs
  - ("Sentence#1", "Sentence test")
  - ("Sentence#2", "Sentence test number 2")
  - ("Sentence#3", "Sentence test number 3")
- Select the words of each sentence and store in the HDFS file system one pair (sentenceld, word) per line

# FlatMapValues transformation: Example

---

```
# Create the local python list
sentences = [("Sentence#1", "Sentence test"), \
             ("Sentence#2", "Sentence test number 2"), \
             ("Sentence#3", "Sentence test number 3") ]\n\n# Create the RDD of pairs from the local collection
sentPairRDD = sc.parallelize(sentences)\n\n# "Extract" words from each sentence
sentIdWord = sentPairRDD.flatMapValues(lambda s: s.split(''))\n\n# Save the result on disk
sentIdWord.saveAsTextFile(outputPath)
```

# **Keys transformation**

---

# Keys transformation: Example

---

- Goal
  - Return the **list of keys** of the input RDD of pairs and store them in a new RDD
    - The **returned RDD** is **not** an **RDD of key-value pairs**
    - The **returned RDD** is a “standard” RDD of “single” elements
    - **Duplicates** keys **are not removed**
- Method
  - The keys transformation is based on the **keys()** method of the **RDD** class

# Keys transformation: Example

---

- Create an RDD from a local python list containing the pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 35)
  - The key is the first name of a user and the value is his/her age
- Store the names of the input users in an output HDFS folder. The output contains one name per line (duplicate names are removed)

# Keys transformation: Example

---

```
# 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()
```

if we have distinct.keys, the results are different since the distinct works on the whole tuples, but here first we separate the keys and choose distinct ones

```
# Store the result in an output folder
```

```
namesRDD.saveAsTextFile(outputPath);
```

# Values transformation

---

# Values transformation

---

- Goal
  - Return the **list of values** of the input RDD of pairs and store them in a new RDD
    - The **returned RDD** is **not** an **RDD of key-value pairs**
    - The **returned RDD** is a “standard” RDD of “single” elements
    - **Duplicates** values **are not removed**
- Method
  - The values transformation is based on the **values()** method of the **RDD** class

# Values transformation: Example

---

- Create an RDD from a local python list containing the pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 22)
  - The key is the first name of a user and the value is his/her age
- Store the ages of the input users in an output HDFS folder.
  - The output contains one age per line
  - Duplicate ages/values are not removed

# Values transformation: Example

---

```
# Create the local python list  
nameAge = [ ("Paolo", 40), ("Giorgio", 22), ("Paolo", 22)]  
  
# 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);
```

# **SortByKey transformation**

---

# SortByKey transformation

---

- Goal
  - Return a new RDD of key-value pairs obtained by sorting, in **ascending order**, the pairs of the input RDD by key
    - Note that the final order is related to the default sorting function of the data type of the input keys
  - The content of the new RDD of key-value pairs is the same of the input RDD but the pairs are sorted by key in the new returned RDD

# SortByKey transformation

---

- Method
  - The sortByKey transformation is based on the **sortByKey()** method of the **RDD** class
    - Pairs are sorted by key in ascending order
  - The **sortByKey(ascending)** method of the **RDD** class is also available
    - This method allows specifying if the sort order is ascending or descending by means of a Boolean parameter
      - True = ascending
      - False = descending

# SortByKey transformation

---

- Shuffle
  - A **shuffle** operation is executed for computing the result of the **sortByKey()** transformation
    - Pairs from different partitions of the input RDD must be compared to sort the input pairs by key

# SortByKey transformation: Example

---

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

# SortByKey transformation: Example

---

```
# 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);
```

# **Transformations on RDDs of key-value pairs: Summary**

---

# Transformations on RDDs of key-value pairs: Summary

---

- All the examples reported in the following tables are applied on an RDD of pairs containing the following tuples (pairs)
  - [("k<sub>1</sub>", 2), ("k<sub>3</sub>", 4), ("k<sub>3</sub>", 6)]
    - The key of each tuple is a string
    - The value of each tuple is an integer

# Transformations on RDDs of key-value pairs: Summary

Transformation	Purpose	Example of applied function	Result
reduceByKey(f)	<p>Return an RDD of pairs containing one pair for each key of the “input” RDD of pairs. The value of each pair of the new RDD of pairs is obtained by combining the values of the input RDD associated with the same key.</p> <p>The “input” RDD of pairs and the new RDD of pairs have the same data type.</p>	<p>reduceByKey( lambda v1, v2: v1+v2)</p> <p>Sum values per key</p>	[("k1", 2), ("k3", 10)]

# Transformations on RDDs of key-value pairs: Summary

Transformation	Purpose	Example of applied function	Result
foldByKey(zeroValue, op)	Similar to the reduceByKey() transformation. However, foldByKey() is characterized also by a zero value	foldByKey(o, lambda v1, v2: v1+v2)  Sum values per key. The zero value is o	[("k1", 2), ("k3", 10)]

# Transformations on RDDs of key-value pairs: Summary

Transformation	Purpose	Example of applied function	Result
combineByKey( createCombiner, mergeValue, mergeCombiner )	<p>Return an RDD of key-value pairs containing one pair for each key of the “input” RDD of pairs. The value of each pair of the new RDD is obtained by combining the values of the input RDD associated with the same key.</p> <p>The values of the “input” RDD of pairs and the values of the new (returned) RDD of pairs can be characterized by different data types.</p>	<pre>combineByKey( lambda e: (e, 1), \ lambda c, e: (c[0]+e, c[1]+1), \ lambda c1, c2: (c1[0]+c2[0], c1[1]+c2[1]) )</pre> <p>Sum values by key and count the number of pairs by key in one single step</p>	[("k1", (2,1)), ("k3", (10,2))]

# Transformations on RDDs of key-value pairs: Summary

Transformation	Purpose	Example of applied function	Result
groupByKey()	<p>Return an RDD of pairs containing one pair for each key of the “input” RDD of pairs.</p> <p>The value of each pair of the new RDD of pairs is a “collection” containing all the values of the input RDD associated with one of the input keys.</p>	groupByKey()	[("k1", [2]), ("k3", [4, 6])]

# Transformations on RDDs of key-value pairs: Summary

Transformation	Purpose	Example of applied function	Result
mapValues(f)	<p>Apply a function over each pair of an RDD of pairs and return a new RDD of pairs.</p> <p>The applied function returns one pair for each pair of the “input” RDD of pairs.</p> <p>The function is applied only on the value part without changing the key.</p> <p>The values of the “input” RDD and the values of new RDD can have different data types.</p>	<pre>mapValues( lambda v: v+1)</pre> <p>Increment the value part by 1</p>	[("k1", 3), ("k3", 5), ("k3", 7)]

# Transformations on RDDs of key-value pairs: Summary

Transformation	Purpose	Example of applied function	Result
flatMapValues(f)	<p>Apply a function over each pair of an RDD of pairs and return a new RDD of pairs.</p> <p>The applied function returns a set of pairs (from 0 to many) for each pair of the “input” RDD of pairs.</p> <p>The function is applied only on the value part without changing the key.</p> <p>The values of the “input” RDD and the values of new RDD can have different data types.</p>	<p>flatMapValues(lambda v: list(range(v,6)))</p> <p>for each input pair (k,v), the set of pairs (k,u) with values of u from v to 5 are returned and included in the new RDD</p>	[("k1", 2), ("k1", 3), ("k1", 4), ("k1", 5), ("k3", 4), ("k3", 5)]

# Transformations on RDDs of key-value pairs: Summary

Transformation	Purpose	Example of applied function	Result
keys()	Return an RDD containing the keys of the input pairRDD	keys()	["k1", "k3", "k3"]
values()	Return an RDD containing the values of the input pairRDD	values()	[2, 4, 6]
sortByKey()	Return a PairRDD sorted by key. The “input” PairRDD and the new PairRDD have the same data type.	sortByKey() -	[("k1", 2), ("k3", 4), ("k3", 6)]

# RDD-based programming

---

# **Transformations on two RDDs of key-value pairs**

---

# Transformations on two RDDs of pairs

---

- Spark supports also some transformations that are applied on two RDDs of key-value pairs at the same time
  - `subtractByKey`, `join`, `coGroup`, etc.

# **SubtractByKey transformation**

---

# SubtractByKey transformation

---

- Goal
  - Create a new RDD of key-value pairs containing only the pairs of the first input RDD of pairs associated with a key that is not appearing as key in the pairs of the second input RDD or pairs
    - The data type of the new RDD of pairs is the same of the “first input” RDD of pairs
    - The two input RDD of pairs must have the same type of keys
      - The data type of the values can be different

# SubtractByKey transformation

---

- Method
  - The subtractByKey transformation is based on the **subtractByKey(other)** method of the **RDD** class
  - The two input RDDs of pairs analyzed by subtractByKey are the one on which the method is invoked and the one passed as parameter (i.e., **other**)

# SubtractByKey transformation

---

- Shuffle
  - A **shuffle** operation is executed for computing the result of the **subtractByKey()** transformation
    - Keys from different partitions of the two input RDDs must be compared

# SubtractByKey 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)
    - [("PaoloG", 40), ("Giorgio", 22), ("PaoloB", 35)]
  - Second list – Banned users (username, motivation)
    - [("PaoloB", "spam"), ("Giorgio", "Vandalism")]
- Create a new RDD of pairs containing only the profiles of the non-banned users

# SubtractByKey transformation: Example

---

```
# 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)
```

# Join transformation

---

# Join transformation

---

## ■ Goal

- Join the key-value pairs of two RDDs of key-value pairs based on the value of the key of the pairs
  - Each pair of the input RDD of pairs is combined with all the pairs of the other RDD of pairs with the same key
  - The new RDD of key-value pairs
    - Has the same **key** data type of the “input” RDDs of pairs
    - Has a tuple as **value** (the pair of values of the two joined input pairs)
  - The **two input** RDDs of key-value pairs
    - Must have the **same type of keys**
    - But the **data types of the values** can be **different**

# Join transformation

---

- Method
  - The join transformation is based on the **join(other)** method of the **RDD** class
  - The two input RDDs of pairs analyzed by join are the one on which the method is invoked and the one passed as parameter (i.e., **other**)

# Join transformation

---

- Shuffle
  - A **shuffle** operation is executed for computing the result of the **join()** transformation
    - Keys from different partitions of the two input RDDs must be compared and values from different partitions must be retrieved

# Join transformation: Example

---

- Create two RDDs of key-value pairs from two local python lists
  - First list – List of questions (QuestionId, Text of the question)
    - [(1, "What is .. ?"), (2, "Who is ..?")]
  - Second list – List of answers (QuestionId, Text of the answer)
    - [(1, "It is a car"), (1, "It is a byke"), (2, "She is Jenny")]
- Create a new RDD of pairs to associate each question with its answers
  - One pair for each possible pair question - answer

# Join transformation: Example

---

```
# 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 byke"), (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)
```

# Join transformation: Example

```
# 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)
```

The key part of the returned RDD of pairs is an integer number

The value part of the returned RDD of pairs is tuple containing two values: (question, answer)

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

# **CoGroup transformation**

---

# Cogroup transformation

---

- Goal
  - Associated each key **k** of the two input RDDs of key-value pairs with
    - The **list** of values associated with **k** in the **first** input RDD of pairs
    - And the **list** of values associated with **k** in the **second** input RDD of pairs
  - The new RDD of key-value pairs
    - Has the same key data type of the two “input” RDDs of pairs
    - Has a tuple as value (the two lists of values of the two input RDDs of pairs)
  - The two input RDDs of key-value pairs
    - Must have the same type of keys
    - But the data types of the values can be different

# Cogroup transformation

---

- Method
  - The cogroup transformation is based on the **cogroup(other)** method of the **RDD** class
  - The two input RDDs of pairs analyzed by cogroup are the one on which the method is invoked and the one passed as parameter (i.e., **other**)

# Cogroup transformation

---

- Shuffle
  - A **shuffle** operation is executed for computing the result of the **cogroup()** transformation
    - Keys from different partitions of the two input RDDs must be compared and values from different partitions must be retrieved

# Cogroup transformation: Example

---

- Create two RDDs of key-value pairs from two local python lists
  - First list – List of liked movies (userId, likedMovies)
    - [(1, "Star Trek"), (1, "Forrest Gump"), (2, "Forrest Gump")]]
  - Second list – List of liked directors (userId, likedDirector)
    - [(1, "Woody Allen"), (2, "Quentin Tarantino"), (2, "Alfred Hitchcock")]]

# Cogroup transformation: Example

---

- Create a new RDD of pairs containing one pair for each userId (key) associated with
  - The list of liked movies
  - The list of liked directors

# Cogroup transformation: Example

---

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

# Cogroup transformation: Example

---

```
# 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)
```

# Cogroup transformation: Example

```
# 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")]
```

Note that the value part of the returned tuples is a tuple containing two “lists”:

- The first value contains the “list” of movies (iterable) liked by a user
- The second value contains the “list” of directors (iterable) liked by a user

```
# Cogroup movies and directors per user  
cogroupPairRDD= moviesPairRDD.cogroup(directorsPairRDD)
```

# **Transformations on two RDDs of key-value pairs: Summary**

---

# Transformations on two RDDs of key-value pairs: Summary

---

- All the examples reported in the following tables are applied on the following two RDDs of key-value pairs
  - inputRDD1: [('k1', 2), ('k3', 4), ('k3', 6)]
  - inputRDD2: [('k3', 9)]

# Transformations on two RDDs of key-value pairs: Summary

Transformation	Purpose	Example	Result
subtractByKey(other)	<p>Return a new RDD of key-value pairs.</p> <p>The returned pairs are those of input RDD on which the method is invoked such that the key part does not occur in the keys of the RDD that is passed as parameter.</p> <p>The values are not considered to take the decision.</p>	inputRDD1.subtractByKey(inputRDD2)	[('k1', 2)]
join(other)	<p>Return a new RDD of pairs corresponding to join of the two input RDDs. The join is based on the value of the key.</p>	inputRDD1.join(inputRDD2)	[('k3', (4,9)), ('k3', (6,9))]

# Transformations on two RDDs of key-value pairs: Summary

Transformation	Purpose	Example	Result
cogroup(other)	<p>For each key k in one of the two input RDDs of pairs, return a pair (k, tuple), where tuple contains:</p> <ul style="list-style-type: none"><li>- the list (iterable) of values of the first input RDD associated with key k</li><li>- the list (iterable) of values of the second input RDD associated with key k</li></ul>	inputRDD1.cogroup(inputRDD2)	[('k1', ([2], [])), ('k3', ([4, 6], [9]))]

# **Actions on RDDs of key-value pairs**

---

# Actions on RDDs of key-value pairs

---

- Spark supports also some specific actions on RDDs of key-value pairs
  - `countByKey`, `collectAsMap`, `lookup`

# **CountByKey action**

---

# CountByKey action

---

- Goal
  - The countByKey action returns a local python dictionary containing the information about the number of elements associated with each key in the input RDD of key-value pairs
    - i.e., the number of times each key occurs in the input RDD
  - Pay attention to the number of distinct keys of the input RDD of pairs
  - If the number of distinct keys is large, the result of the action cannot be stored in a local variable of the Driver

# CountByKey action

---

- Method
  - The countByKey action is based on the `countByKey()` method of the `RDD` class
- **Data are sent on the network** to compute the final result

# CountByKey action: Example 1

---

- 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 the rating given by someone to that movie
- Compute the number of ratings for each movie

# CountByKey action: Example 1

---

```
# 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)  
  
# Compute the number of rating for each movie
movieNumRatings = movieRatingRDD.countByKey()  
  
# Print the result on the standard output
print(movieNumRatings)
```

# CountByKey action: Example 1

```
# 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)  
  
# Compute the number of rating for each movie  
movieNumRatings= movieRatingRDD.countByKey()
```

Pay attention to the size of the returned local python dictionary  
(i.e., the number of distinct movies in this case).

# **CollectAsMap action**

---

# CollectAsMap action

---

- Goal
  - The collectAsMap action returns a local dictionary containing the same pairs of the considered input RDD of pairs
  - Pay attention to the size of the returned RDD
  - Data are sent on the network
- Method
  - The collectAsMap action is based on the `collectAsMap()` method of the `RDD` class

# CollectAsMap action

---

- Pay attention that the **collectAsMap** action **returns a dictionary** object
- A **dictionary cannot contain duplicate keys**
  - Each **key** can be **associated** with **at most one value**
  - If the “**input**” RDD of pairs contains **more than one pair with the same key**, **only one of those pairs** is **stored** in the returned local python dictionary
    - Usually, the **last one** occurring in the input RDD of pairs
- Use **collectAsMap** **only if** you are sure that each **key** **appears only once** in the input RDD of key-value pairs

# CollectAsMap vs collect

---

- 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 returned by `collect()` can contain more than one pair associated with the same key

# CollectAsMap action: Example 1

---

- Create an RDD of pairs from the following python list
  - [("User1", "Paolo"), ("User2", "Luca"), ("User3", "Daniele")]
  - Each pair contains a userId and the name of the user
- Retrieve the pairs of the created RDD of pairs and store them in a local python dictionary that is instantiated in the Driver

# CollectAsMap action: Example 1

---

```
# 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)
```

# CollectAsMap action: Example 1

```
# 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()
```

Pay attention to the size of the returned local python dictionary  
(i.e., the number of distinct users in this case).

# **Lookup action**

---

# Lookup action

---

- Goal
  - The lookup(`k`) action returns a local python list containing the values of the pairs of the input RDD associated with the key `k` specified as parameter
- Method
  - The lookup action is based on the `lookup(key)` method of the `RDD` class

# Lookup action: Example 1

---

- 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 the rating given by someone to that movie
- Retrieve the ratings associated with the movie “Forrest Gump” and store them in a local python list in the Driver

# Lookup action: Example 1

---

```
# 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)  
  
# Select the ratings associated with "Forrest Gump"
movieRatings = movieRatingRDD.lookup("Forrest Gump")  
  
# Print the result on the standard output
print(movieRatings)
```

# Lookup action: Example 1

---

```
# 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)
```

```
# Select the ratings associated with "Forrest Gump"  
movieRatings= movieRatingRDD.lookup("Forrest Gump")
```

Pay attention to the size of the returned list (i.e., the number of ratings associated with "Forrest Gump" in this case).

# **Actions on RDDs of key-value pairs: Summary**

---

# Actions on RDDs of key-value pairs: Summary

---

- All the examples reported in the following tables are applied on the following RDD of key-value pairs
  - inputRDD: [('k1', 2), ('k3', 4), ('k3', 6)]

# Actions on RDDs of key-value pairs: Summary

Transformation	Purpose	Example	Result
countByKey()	Return a local python dictionary containing the number of elements in the input RDD for each key of the input RDD of pairs	inputRDD.countByKey()	{('k1',1), ('K3',2)}
collectAsMap()	Return a local python dictionary containing the pairs of the input RDD of pairs	inputRDD.collectAsMap()	{('k1', 2), ('k3', 6)} Or {('k1', 2), ('k3', 4)} Depending on the order of the pairs in the input RDD of pairs
lookup(key)	Return a local python list containing all the values associated with the key specified as parameter	inputRDD.lookup('k3')	[4, 6]

# **RDDs of numbers**

---

# RDDs of numbers

---

- Spark provides specific actions for RDD containing numerical values (integers or floats)
- **RDDs of numbers** can be **created** by using the standard methods
  - `parallelize`
  - `transformations` that return an RDD of numbers
- The following specific actions are also available on this type of RDDs
  - `sum()`, `mean()`, `stdev()`, `variance()`, `max()`, `min()`

# RDDs of numbers: actions

---

- All the examples reported in the following are applied on inputRDD that is an RDD containing the following double values
  - [1.5, 3.5, 2.0]

# RDDs of numbers: Summary

Action	Purpose	Example	Result
sum()	Return the sum over the values of the inputRDD	inputRDD.sum()	7.0
mean()	Return the mean value	inputRDD.mean()	2.3333
stdev()	Return the standard deviation computed over the values of the inputRDD	inputRDD.stdev()	0.8498
variance()	Return the variance computed over the values of the inputRDD	inputRDD.variance()	0.7223
max()	Return the maximum value	inputRDD.max()	3.5
min()	Return the minimum value	inputRDD.min()	1.5

# RDDs of numbers: example

---

- Create an RDD containing the following float values
  - [1.5, 3.5, 2.0]
- Print on the standard output the following statistics
  - sum, mean, standard deviation, variance, maximum value, and minimum value

# DoubleRDD actions: example

---

```
# Create an RDD containing a list of float values  
inputRDD = sc.parallelize([1.5, 3.5, 2.0])
```

```
# Compute the statistics of interest and print them on  
# the standard output  
print("sum:", inputRDD.sum())  
print("mean:", inputRDD.mean())  
print("stdev:", inputRDD.stdev())  
print("variance:", inputRDD.variance())  
print("max:", inputRDD.max())  
print("min:", inputRDD.min())
```

# **Cache, Accumulators, Broadcast variables**

---

# **Persistence and Cache**

---

# Persistence and Cache

---

- Spark computes the content of an RDD each time an action is invoked on it
- If the same RDD is used multiple times in an application, Spark recomputes its content every time an action is invoked on the RDD, or on one of its “descendants”
- This is expensive, especially for iterative applications
- We can ask Spark to persist/cache RDDs

# Persistence and Cache

---

- When you ask Spark to persist/cache an RDD, each node stores the content of its partitions in memory and reuses them in other actions on that RDD/dataset (or RDDs derived from it)
  - The first time the content of a persistent/cached RDD is computed in an action, it will be kept in the main memory of the nodes
  - The next actions on the same RDD will read its content from memory
    - i.e., Spark persists/caches the content of the RDD across operations
    - This allows future actions to be much faster (often by more than 10x)

# Persistence and Cache

---

- Spark supports several storage levels
  - The storage level is used to specify if the content of the RDD is stored
    - In the main memory of the nodes
    - On the local disks of the nodes
    - Partially in the main memory and partially on disk

# Persistence and Cache: Storage levels

Storage Level	Meaning
MEMORY_ONLY	Store RDD as serialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as serialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on (local) disk, and read them from there when they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONL, but store the data in off-heap memory. This requires off-heap memory to be enabled.

# Persistence and Cache

---

- You can mark an RDD to be persisted by using the **persist(storageLevel)** method of the **RDD** class
- The **parameter of persist** can assume the following values
  - `pyspark.StorageLevel.MEMORY_ONLY`
  - `pyspark.StorageLevel.MEMORY_AND_DISK`
  - `pyspark.StorageLevel.DISK_ONLY`
  - `pyspark.StorageLevel.NONE`
  - `pyspark.StorageLevel.OFF_HEAP`

# Persistence and Cache

---

- `pyspark.StorageLevel.MEMORY_ONLY_2`
- `pyspark.StorageLevel.MEMORY_AND_DISK_2`
- The storage level \* \_2 replicate each partition on two cluster nodes
  - If one node fails, the other one can be used to perform the actions on the RDD without recomputing the content of the RDD

# Persistence and Cache

---

- You can cache an RDD by using the `cache()` method of the `RDD` class
  - It corresponds to persist the RDD with the storage level '`MEMORY_ONLY`'
  - i.e., it is equivalent to  
`inRDD.persist(pyspark.StorageLevel.MEMORY_ONLY)`
- Note that both `persist` and `cache` **return** a **new RDD**
  - Because RDDs are immutable

# Persistence and Cache

---

- The use of the persist/cache mechanism on an RDD provides an **advantage** if the same RDD is used multiple times
  - i.e., multiples actions are applied on it or on its descendants

the number of paths is important to have a cache or not, path is related to the actions

# Persistence and Cache

---

- The **storage levels** that store RDDs on **disk** are **useful** if and only if
  - The “**size**” of the RDD is significantly **smaller** than the size of the **input dataset**
  - Or the **functions** that are used to compute the content of the RDD are **expensive**
  - Otherwise, recomputing a partition may be as fast as reading it from disk

when have some transformations and aggregations to get new RDD

# Remove data from cache

---

- Spark automatically monitors cache usage on each node and drops out old data partitions in a least-recently-used (LRU) fashion
- You can manually remove an RDD from the cache by using the `unpersist()` method of the **RDD** class

# Cache: Example

---

- Create an RDD from a textual file containing a list of words
  - One word for each line
- Print on the standard output
  - The number of lines of the input file
  - The number of distinct words

# Cache: Example

---

```
# Read the content of a textual file
# and cache the associated RDD
inputRDD = sc.textFile("words.txt").cache() if the size of the memory is enough to store the
                                                whole data, spark read the data from DFS just
                                                once, otherwise depending on size of data
print("Number of words: ", inputRDD.count())
print("Number of distinct words: ", inputRDD.distinct().count())
                                                there is no need to cache the content of distinct
                                                output, bc it is used just one time by count
```

# Cache: Example

---

```
# Read the content of a textual file  
# and cache the associated RDD  
inputRDD = sc.textFile("words.txt").cache()  
  
print("Number of words: " + inputRDD.count().toString())  
print("Number of distinct words: " + inputRDD.distinct().count().toString())
```

The cache method is invoked.

Hence, inputRDD is a “cached” RDD

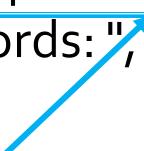
# Cache: Example

---

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

This is the first time an action is invoked on the inputRDD RDD.

The content of the RDD is computed by reading the lines of the words.txt file and the result of the count action is returned. The content of inputRDD is also stored in the main memory of the nodes of the cluster.



# Cache: Example

---

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

The content of inputRDD is in the main memory if the nodes of the cluster. Hence the computation of distinct() + count() is performed by reading the data from the main memory and not from the input (HDFS) file words.txt

# **Accumulators**

---

# Accumulators

---

- When a “function” passed to a Spark operation is executed on a remote cluster node, it works on separate copies of all the variables used in the function
  - These variables are copied to each node of the cluster, and no updates to the variables on the nodes are propagated back to the driver program

# Accumulators

---

- Spark provides a type of shared variables called **accumulators**
- **Accumulators** are **shared variables** that are only “added” to through an **associative operation** and can therefore be efficiently supported in **parallel**
- **They** can be used to implement **counters** or **sums**

# Accumulators

---

- Accumulators are usually used to compute simple statistics while performing some other actions on the input RDD
  - The avoid using actions like reduce() to compute simple statistics (e.g., count the number of lines with some characteristics)

# Accumulators

---

- The driver defines and initializes the accumulator
- The code executed in the worker nodes increases the value of the accumulator
  - i.e., the code in the “functions” associated with the transformations
- The final value of the accumulator is returned to the driver node
  - Only the driver node can access the final value of the accumulator
  - The worker nodes cannot access the value of the accumulator
    - They can only add values to it

# Accumulators

---

- Pay attention that the **value** of the accumulator is **increased** in the **functions** associated **with transformations**
- Since transformations are lazily evaluated, the value of the accumulator is computed only when an action is executed on the RDD on which the transformations increasing the accumulator are applied

# Accumulators

---

- Spark natively supports numerical accumulators
  - Integers and floats
- But programmers can add support for new data types
- Accumulators are  
**pyspark.accumulators.Accumulator** objects

# Accumulators

---

- Accumulators are defined and **initialized** by using the **accumulator(value)** method of the **SparkContext** class
- The value of an accumulator can be “**increased**” by using the **add(value)** method of the **Accumulator** class
  - Add “value” to the current value of the accumulator
- The **final** value of an accumulator can be retrieved in the **driver** program by using **value** of the **Accumulator** class

# Accumulators: 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 is considered a valid email if it contains the @ symbol
- Print also, on the standard output, the number of invalid emails

# Accumulators: Example

---

```
# Define an accumulator. Initialize it 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)
        return False
    else:
        return True

# Select only valid emails
# Count also the number of invalid emails
validEmailsRDD = emailsRDD.filter(validEmailFunc)
```

if we dont use acc. we need to use filter and add two times which means two times of reading RDD

not only the number of actions determines the paths but also the number of aggregation func like join does it too

# Accumulators: Example

---

```
# Define an accumulator. Initialize it to 0
invalidEmails = sc.accumulator(0)

# Read the content of the input text file
emailsRDD = sc.textFile("input.txt")

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

# Select only valid emails
# Count also the number of invalid emails
validEmailsRDD = emailsRDD.filter(validEmailFunc)
```

Definition of an accumulator of type integer

# Accumulators: Example

---

```
# Define an accumulator. Initialize it 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)
        return False
    else:
        return True
```

This function increments the value of the  
invalidEmails accumulator if the email is invalid

```
validEmailsRDD = emailsRDD.filter(validEmailFunc)
```

# Accumulators: Example

---

```
# Store valid emails in the output file  
validEmailsRDD.saveAsTextFile(outputPath)
```

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

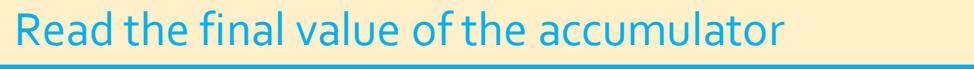
# Accumulators: Example

---

```
# Store valid emails in the output file  
validEmailsRDD.saveAsTextFile(outputPath)
```

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

Read the final value of the accumulator



# Accumulators: Example

---

```
# 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 the value of the accumulator is correct only because an action (saveAsTextFile) has been executed on the validEmailsRDD and its content has been computed (the function validEmailFunc has been executed on each element of emailsRDD)

# Personalized accumulators

---

- Programmers can define accumulators based on new data types (different from integers and floats)
- To define a new accumulator data type of type T, the programmer must define a class subclassing the **AccumulatorParam** interface
  - The AccumulatorParam interface has two methods
    - zero for providing a “zero value” for your data type
    - addInPlace for adding two values together

# **Broadcast variables**

---

# Broadcast variables

---

- Spark supports broadcast variables
- A broadcast variable is a **read-only** (small/medium) **shared variable**
  - That is **instantiated** in the **driver**
    - The **broadcast variable** is **stored** in the **main memory of the driver** in a local variable
  - And it is **sent** to **all worker nodes** that use it in one or more Spark operations
    - The broadcast variable is also stored in the main memory of the executors (which are instantiated in the used worker nodes)

# Broadcast variables

---

- A copy each broadcast variable is sent to all executors that are used to run a task executing a Spark operation based on that variable
  - i.e., the variable is sent “num. executors” times
- A broadcast variable is sent only one time to each executor that uses that variable in at least one Spark operation (i.e., in at least one of its tasks)
  - Each executor can run multiples tasks associated with the same broadcast variable
    - The broadcast variable is sent only one time for each executor
  - Hence, the amount of data sent on the network is limited by using broadcast variables instead of “standard” variables

# Broadcast variables

---

- Broadcast variables are usually used to share (small/medium) **lookup-tables**
  - They are stored in local variables
  - They must be small enough to be stored in the main memory of the driver and also in the main memory of the executors

# Broadcast variables

---

- Broadcast variables are objects of type **Broadcast**
- A broadcast variable (of type T) is defined in the driver by using the **broadcast(value)** method of the **SparkContext** class
- The value of a broadcast variable (of type T) is retrieved (usually in transformations) by using **value** of the **Broadcast** class

# Broadcast variables: Example

---

- Create an RDD from a textual file containing a dictionary of pairs (word, integer value)
  - One pair for each line
  - Suppose the content of this first file is large but can be stored in main-memory
- Create an RDD from a textual file containing a set of words
  - A sentence (set of words) for each line
- “Transform” the content of the second file mapping each word to an integer based on the dictionary contained in the first file
  - Store the result in an HDFS file

# Broadcast variables: Example

---

- First file (dictionary)

- java 1

- spark 2

- test 3

- Second file (the text to transform)

- java spark

- spark test java

- Output file

- 1 2

- 2 3 1

# Broadcast variables: Example

---

```
# 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 based on the content of dictionaryRDD.
# Pay attention that a broadcast variable can be instantiated only
# by passing as parameter a local variable and not an RDD.
# Hence, the collectAsMap method is used to retrieve the content of the
# RDD and store it in the dictionary variable
dictionary = dictionaryRDD.collectAsMap()  
  
# Broadcast dictionary
dictionaryBroadcast = sc.broadcast(dictionary)
```

# Broadcast variables: Example

```
# 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 based on the content of dictionaryRDD.  
# Pay attention that a broadcast variable can be instantiated only  
# by passing as parameter a local variable and not an RDD.  
# Hence, the collectAsMap method is used to retrieve the content of the  
# RDD and store it in the dictionary variable  
dictionary = dictionaryRDD.collectAsMap()
```

```
# Broadcast dictionary  
dictionaryBroadcast = sc.broadcast(dictionary)
```

Define a broadcast variable

# Broadcast variables: Example

---

```
# Read the content of the second file
textRDD = sc.textFile("document.txt")

# Define the function that is used to map strings to integers
def myMapFunc(line):
    transformedLine=""

        here there isn't any advantages of using broadcast since the trns. uses it
    for word in line.split(' '): just once
        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)
```

# Broadcast variables: Example

```
# Read the content of the second file
textRDD = sc.textFile("document.txt")

# Define the function that is used 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)
```

Retrieve the content of the broadcast variable and use it

# Broadcast variables: Example

---

```
# Store the result in an HDFS file  
mappedTextRDD.saveAsTextFile(outputPath)
```

# **RDDs and Partitions**

---

# RDDs and Partitions

---

- The content of each RDD is split in partitions
  - The number of partitions and the content of each partition depend on how RDDs are defined/created
- The number of partitions impacts on the maximum parallelization degree of the Spark application
  - But pay attention that the amount of resources is limited (there is a maximum number of executors and parallel tasks)

# How many Partitions are good ?

---

- **Disadvantages** of too few partitions
  - Less concurrency/parallelism
    - There could be worker nodes that are idle and could be used to speed up the execution of your application
  - Data skewing and improper resource utilization
    - Data might be skewed on one partition
      - One partition with many data
      - Many partitions with few data
    - The worker node that processes that large partition needs more time than the other workers
      - It becomes the bottleneck of your application

# How many Partitions are good ?

---

- Disadvantages of too many partitions
  - Task scheduling may take more time than actual execution time if the amount of data in some partitions is too small

`min partition` must be at least as many as input blocks, otherwise spark will create the input blocks number of partitions

# RDDs and Partitions

---

- Only some specific transformations set the number of partitions of the returned RDD
  - `parallelize()`, `textFile()`, `repartition()`, `coalesce()`
- The majority of the Spark transformations do not change the number of partitions
  - Those transformations preserve the number of partitions of the input RDD
    - i.e., the returned RDD has the same number of partitions of the input RDD

# RDDs and Partitions

---

- **parallelize(collection)**
  - The number of partitions of the returned RDD is equal to `sc.defaultParallelism`
  - Sparks tries to balance the number of elements per partition in the returned RDD
    - Elements are **not** assigned to partitions based on their value
- **parallelize(collection, numSlices)**
  - The number of partitions of the returned RDD is equal to `numSlices`
  - Sparks tries to balance the number of elements per partition in the returned RDD
    - Elements are **not** assigned to partitions based on their value

# RDDs and Partitions

---

- `textFile(pathInputData)`
  - The number of partitions of the returned RDD is 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 user specified number of partitions must be **greater** than the number of input blocks
  - The number of partitions of the returned RDD is greater than or equal to the specified value **minPartitions**
  - Each partition contains **a part of one input blocks**

# RDDs and Partitions

---

- **repartition(numPartitions)**
  - **numPartitions** can be **greater or smaller** than the number of partitions of the input RDD
  - The number of partitions of the returned RDD is equal to **numPartitions**
  - Sparks tries to balance the number of elements per partition in the returned RDD
    - Elements are **not** assigned to partitions based on their value
  - A **shuffle** operation is executed to assign input elements to the partitions of the returned RDD

# RDDs and Partitions

---

- **coalesce(numPartitions)**
  - **numPartitions** < number of partitions of the input RDD
  - The number of partitions of the returned RDD is equal to **numPartitions**
  - Sparks tries to balance the number of elements per partition in the returned RDD
    - Elements are **not** assigned to partitions based on their value
  - Usually **no shuffle operation** is executed to assign input elements to the partitions of the returned RDD
  - **coalesce is more efficient than repartition to reduce the number of partitions**

# Partitioning of Pair RDDs

---

- Spark allows specifying how to partition the content of RDDs of **key-value pairs**
  - The input pairs are grouped in partitions based on the integer value returned by a **function applied on the key** of each input pair
  - This operation can be useful to improve the efficiency of the next transformations by reducing the amount of shuffle operations and the amount of data sent on the network in the next steps of the application
    - Spark can optimize the execution of the transformations if the input RDDs of pairs are properly partitioned

# partitionBy

just for key-value pairs RDD

each partition is associated with the keys, so pay attention to the number of distinct keys and num of partition

---

- Partitioning is based on the **partitionBy()** transformation
- **partitionBy(numPartitions)**
  - The input pairs are grouped in partitions based on the integer value returned by a **default hash function** applied on the key of each input pair
  - A **shuffle** operation is executed to assign input elements to the partitions of the returned RDD

# partitionBy

---

- Suppose that
  - The number of partition of the returned Pair RDD is `numPart`
  - The default partition function is `portable_hash`
  - Given an input pair `(key, value)` a copy of that pair will be stored in the partition number `n` of the returned RDD, where
$$n = \text{portable\_hash}(\text{key}) \% \text{numPart}$$

# partitionBy

---

- Suppose that
  - The number of partition of the returned Pair RDD is `numPart`
  - The default partition function is `portable_hash`
  - Given an input pair `(key, value)` a copy of that pair will be stored in the partition number `n` of the returned RDD, where

`n = portable_hash(key) % numPart`

This function returns an integer

# partitionBy: Custom function

---

- **partitionBy(numPartitions, partitionFunc)**
  - The input pairs are grouped in partitions based on the **integer** value returned by the user provided **partitionFunc** function
  - A **shuffle** operation is executed to assign input elements to the partitions of the returned RDD

# partitionBy: Custom function

---

- Suppose that
  - The number of partition of the returned Pair RDD is `numPart`
  - The partition function is `partitionFunc`
  - Given an input pair `(key, value)` a copy of that pair will be stored in the partition number `n` of the returned RDD, where
$$n = \text{partitionFunc(key)} \% \text{numPart}$$

# partitionBy: Custom function

---

- Suppose that
  - The number of partition of the returned Pair RDD is **numPart**
  - The partition function is **partitionFunc**
  - Given an input pair **(key, value)** a copy of that pair will be stored in the partition number **n** of the returned RDD, where

$n = \text{partitionFunc}(\text{key}) \% \text{numPart}$

Custom partition function

# partitionBy: Use case scenario

---

- Partitioning Pair RDDs by using `partitionBy()` is useful only when the same partitioned RDD is cached and **reused multiple times** in the application **in time and network consuming key-oriented transformations**
  - E.g., the same partitioned RDD is used in many `join()`, `cogroup`, `groupByKey()`, .. transformations in different paths/branches of the application (different paths/branches of the DAG)
- **Pay attention** to the amount of data that is actually sent on the network
  - **partitionBy() can slow down your application** instead of speeding it up

if we have e.g. many `reduceByKey` trans. (meaning that having several paths) it is better to use `partitionBy()` since it puts pairs with the same key in one partition and does it just once by sending data on the network.

# partitionBy: Example

---

- Create an RDD from a textual file containing a list of pairs (pageID, list of linked pages)
- Implement the (simplified) PageRank algorithm and compute the pageRank of each input page
- Print the result on the standard output

# partitionBy: Example

---

```
# 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()
```

# partitionBy: Example

```
# 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(',')  
    ...
```

Note that the returned Pair RDD is partitioned and cached

we have new arrangement of data

```
links = inputData.map(mapToPairPageIDLinks) \  
    .partitionBy(inputData.getNumPartitions()) \  
    .cache()
```

we do this since in the next step we use a loop which uses this data a couple of time and the content of links is constant for this loop

# partitionBy: Example

---

```
# Initialize each page's rank to 1.0; since we use mapValues,  
# the resulting RDD will have the same partitioner as links  
ranks = links.mapValues(lambda v: 1.0)
```

# partitionBy: Example

---

```
# Function that returns a set of pairs from each input pair
# input pair: (pageid, (linked pages, current page rank of pageid) )
# one output pair for each linked page. Output pairs:
# (pageid linked page,
#   current page rank of the linking page pageid / 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
```

# partitionBy: Example

---

```
# 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)
    this is not the same ranks as previous one, just the names are the same but every time we have new
    partition since RDD is immutable
    # Update current pagerank of all pages for this iteration
    ranks = contributions\
        .reduceByKey(lambda contrib1, contrib2: contrib1+contrib2)

    # Print the result
    ranks.collect()
```

# partitionBy: Example

```
# 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)
```

The join transformation is invoked many times on the links Pair RDD.  
The content of links is constant (it does not change during the loop iterations).

Hence, caching it and also partitioning its content by key is useful.

- Its content is computed one time and cached in the main memory of the executors
- It is shuffled and sent on the network only one time because we applied partitionBy on it.

```
# Print the result
ranks.collect()
```

# Default partitioning behavior of the main transformations

Transformation	Number of partitions	Partitioner
sc.parallelize(...)	sc.defaultParallelism	NONE
sc.textFile(...)	sc.defaultParallelism or number of file blocks , whichever is greater	NONE
filter(),map(),flatMap(), distinct()	same as parent RDD	NONE except <b>filter</b> preserve parent RDD's partitioner
rdd.union(otherRDD) <i>no data to network, just adding partitions</i>	rdd.partitions.size + otherRDD. partitions.size	
rdd.intersection(otherRDD) <i>sending data to the network</i>	max(rdd.partitions.size, otherRDD. partitions.size)	
rdd.subtract(otherRDD)	rdd.partitions.size	
rdd.cartesian(otherRDD) <i>sending data to the network</i>	rdd.partitions.size * otherRDD. partitions.size	

# Default partitioning behavior of the main transformations

Transformation	Number of partitions	Partitioner
reduceByKey(), foldByKey(), combineByKey(), groupByKey()	same as parent RDD	HashPartitioner
sortByKey()	same as parent RDD	RangePartitioner
mapValues(), flatMapValues()	same as parent RDD	parent RDD's partitioner
cogroup(), join(), ,leftOuterJoin(), rightOuterJoin()	depends upon input properties of two involved RDDs	HashPartitioner

# Broadcast join

---

# Broadcast join

---

- The join transformation is expensive in terms of execution time and amount of data sent on the network
- If one of the two input RDDs of key-value pairs is small enough to be stored in the main memory when can use a more efficient solution based on a broadcast variable
  - Broadcast hash join (or map-side join)
  - The smaller the small RDD, the higher the speed up

# Broadcast join: Example

---

- Create a large RDD from a textual file containing a list of pairs (userID, post)
  - Each user can be associated to several posts
- Create a small RDD from a textual file containing a list of pairs (userID, (name, surname, age))
  - Each user can be associated to one single line in this second file
- Compute the join between these two files

# Broadcast join: Example

---

```
# Read the first input file
largeRDD = sc.textFile("post.txt")
.map(lambda line: (int(line.split(',')[0]), line.split(',')[1]))  
  
# Read the second input file
smallRDD = sc.textFile("profiles.txt")
.map(lambda line: (int(line.split(',')[0]), line.split(',')[1]))  
  
# Broadcast join version
# Store the "small" RDD in a local python variable in the driver
# and broadcast it
localSmallTable = smallRDD.collectAsMap()
localSmallTableBroadcast = sc.broadcast(localSmallTable)
```

# Broadcast join: Example

---

```
# Function for joining a record of the large RDD with the matching
# record of the small one
def joinRecords(largeTableRecord):
    returnedRecords = []
    key = largeTableRecord[0]
    valueLargeRecord = largeTableRecord[1]

    if key in localSmallTableBroadcast.value:
        returnedRecords.append((key, (valueLargeRecord,\n            localSmallTableBroadcast.value[key])))\n\n    return returnedRecords\n\n# Execute the broadcast join operation by using a flatMap\n# transformation on the "large" RDD\nuserPostProfileRDDBroadcatJoin = largeRDD.flatMap(joinRecords)
```

we use flatmap instead of map in case there is no match of  
big file with small one and returning empty list  
(inconsistency)

# Introduction to PageRank

---

# PageRank

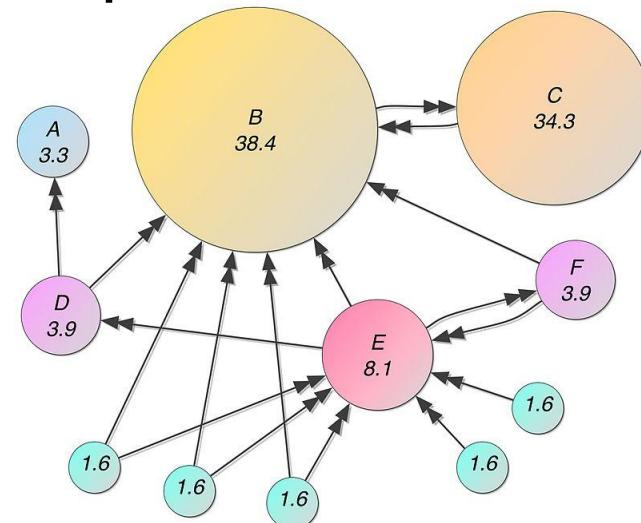
---

- PageRank is the original famous algorithm used by the Google Search engine to rank vertexes (web pages) in a graph by order of importance
  - For the Google search engine
    - Vertexes are web pages in the World Wide Web,
    - Edges are hyperlinks among web pages
  - It assigns a numerical weighting (importance) to each node

# PageRank

---

- It computes a likelihood that a person randomly clicking on links will arrive at any particular web page
- For a high PageRank, it is important to
  - Have many in-links
  - Be liked by relevant pages (pages characterized by a high PageRank)



# PageRank

---

- Basic idea
  - Each link's vote is proportional to the importance of its source page  $p$
  - If page  $p$  with importance  $\text{PageRank}(p)$  has  $n$  out-links, each out-link gets  $\text{PageRank}(p)/n$  votes
  - Page  $p$ 's importance is the sum of the votes on its in-links

# PageRank: Simple recursive formulation

---

1. # Initialize each page's rank to 1.0  
For each  $p$  in pages set  $\text{PageRank}(p)$  to 1.0
2. Iterate for max iterations
  - a. Page  $p$  sends a contribution  $\text{PageRank}(p)/\text{numOutLinks}(p)$  to its neighbors (the pages it links)
  - b. Update each page's rank  $\text{PageRank}(p)$  to  $\text{sum}(\text{received contributions})$
  - c. Go to step 2

# PageRank with Random jumps

---

- The PageRank algorithm simulates the random walk of a user on the web
- At each step of the random walk, the random surfer has two options:
  - With probability  $1-\alpha$ , follow a link at random among the ones in the current page
  - With probability  $\alpha$ , jump to a random page

# PageRank with Random jumps

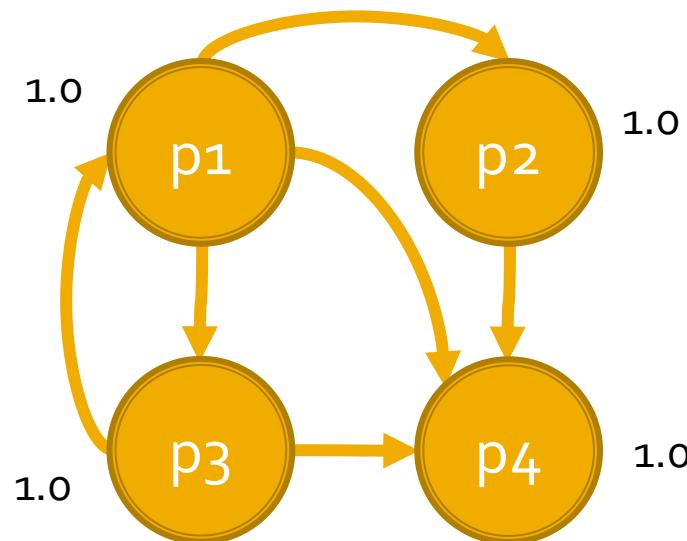
---

1. # Initialize each page's rank to 1.0  
For each  $p$  in pages set  $\text{PageRank}(p)$  to 1.0
2. Iterate for max iterations
  - a. Page  $p$  sends a contribution  $\text{PageRank}(p)/\text{numOutLinks}(p)$  to its neighbors (the pages it links)
  - b. Update each page's rank  $\text{PageRank}(p)$  to  $\alpha + (1 - \alpha) * \text{sum}(\text{received contributions})$
  - c. Go to step 2

# PageRank: Example

---

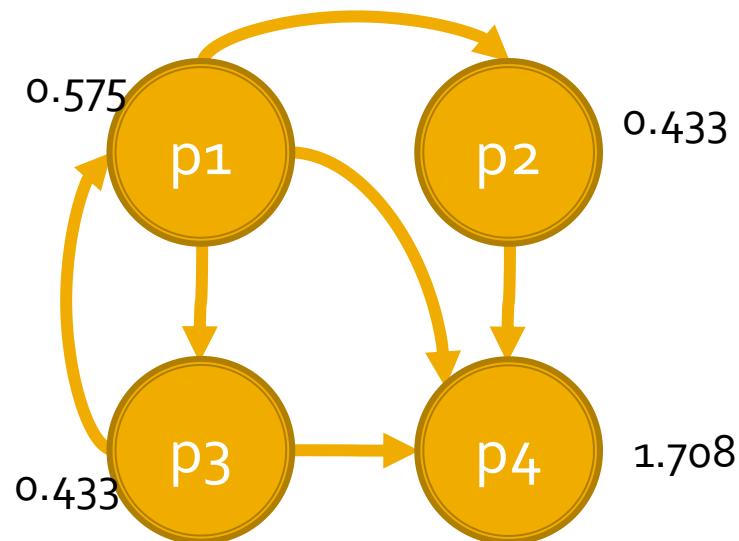
- $\alpha = 0.15$
- Initialization:  $\text{PageRank}(p) = 1.0 \forall p$



# PageRank: Example

---

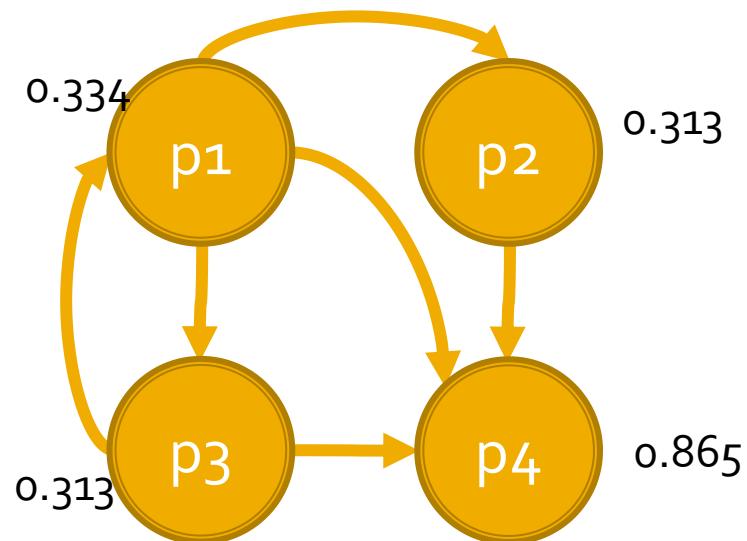
- Iteration #1



# PageRank: Example

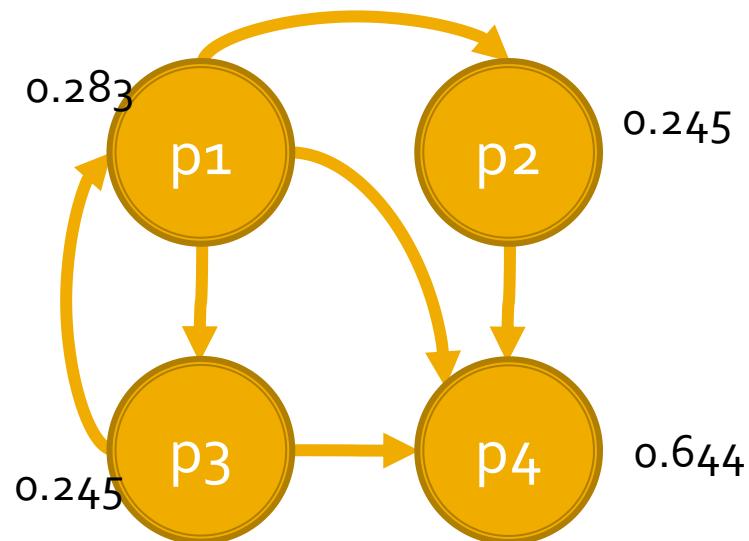
---

- Iteration #2



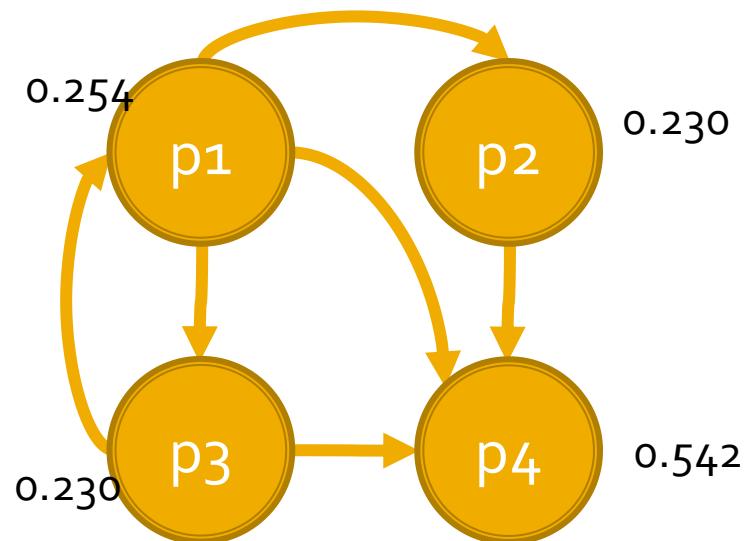
# PageRank: Example

- Iteration #3



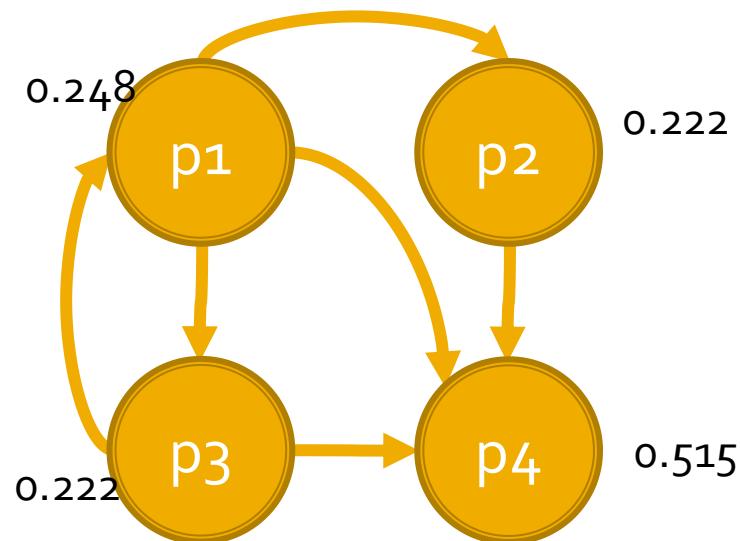
# PageRank: Example

- Iteration #4



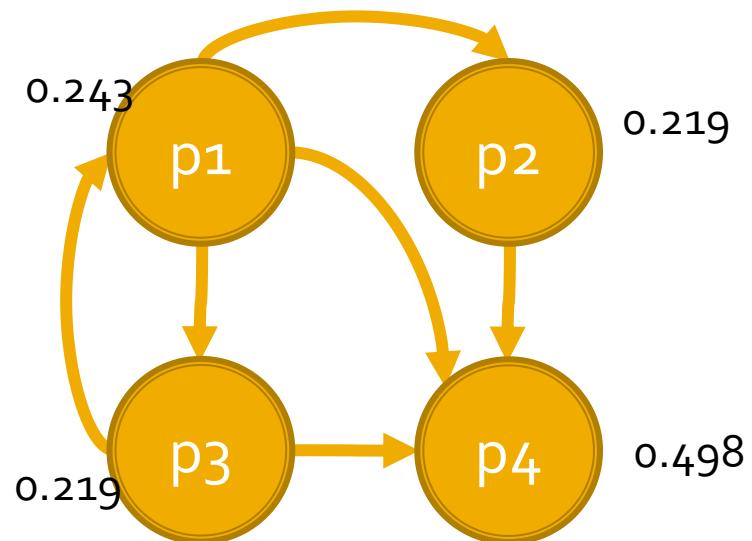
# PageRank: Example

- Iteration #5



# PageRank: Example

- Iteration #50



# **Spark SQL and DataFrames**

---

# Spark SQL

---

- Spark SQL is the Spark component for structured data processing
- It provides a programming abstraction called *Dataframe* and can act as a distributed SQL query engine
  - The input data can be queried by using
    1. Ad-hoc methods
    2. Or an SQL-like language

# Spark SQL vs Spark RDD APIs

---

- The **interfaces** provided by Spark SQL provide more information about the structure of both the data and the computation being performed
- Spark SQL uses this **extra information** to perform extra **optimizations** based on an “**SQL-like**” optimizer called **Catalyst**
  - => Programs based on **Dataframe** are usually **faster than standard RDD-based programs**

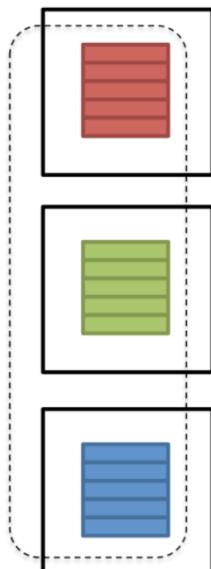
# Spark SQL vs Spark RDD APIs

RDD

Unstructured

vs

DataFrame  
Structured



Distributed list of objects

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	43	B Jones
Chem	61	M Kennedy

~Distributed SQL table

# DataFrames

---

- **DataFrame**
  - Distributed collection of structured data
    - It is **conceptually equivalent to a table** in a relational database
    - It can be **created reading data from different types of external sources** (CSV files, JSON files, RDBMs, ...)
  - Benefits from Spark SQL's optimized execution engine exploiting the information about the data structure

# Spark Session

---

- All the **Spark SQL** functionalities are based on an **instance** of the

`pyspark.sql.SparkSession class`

- Import it in your standalone applications

```
from pyspark.sql import SparkSession
```

- To instance a **SparkSession object**:

```
spark = SparkSession.builder.getOrCreate()
```

# Spark Session

---

- To “close” a Spark Session use the **SparkSession.stop()** method

```
spark.stop()
```

# DataFrames

---

# DataFrames

---

- **DataFrame**
  - It is a distributed collection of data organized into named columns
  - It is equivalent to a relational table
- **DataFrames** are lists of Row objects
- Classes used to define DataFrames
  - **pyspark.sql.DataFrame**
  - **pyspark.sql.Row**

# DataFrames

---

- **DataFrames** can be **created** from different sources
  - Structured (textual) data files
    - E.g., csv files, json files
  - Existing RDDs
  - Hive tables
  - External relational databases

# Creating DataFrames from csv files

---

- Spark SQL provides an API that allows creating DataFrames directly from CSV files
- Example of csv file

name,age

Andy,30

Michael,

Justin,19

- The file contains name and age of three persons
  - The age of the second person is unknown

# Creating DataFrames from csv files

---

- The creation of a DataFrame from a csv file is based the
  - **load(path)** method of the **pyspark.sql.DataFrameReader** class
    - Path is the path of the input file
  - You get a **DataFrameReader** with the **read()** method of the **SparkSession** class.

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

# Creating DataFrames from csv files: 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
    - Age can assume the null value (i.e., it can be missing)
  - The first line contains the header (i.e., the names of the attributes/columns)
  - Example of csv file

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

# Creating DataFrames from csv files: Example

---

```
# 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)
```

# Creating DataFrames from csv files: Example

---

```
# 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)
```

This is used to specify the format of the input file

# Creating DataFrames from csv files: Example

```
# 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)
```

This is used to specify that the first line of the file contains the name of the attributes/columns

# Creating DataFrames from csv files: Example

```
# 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)
```

This method is used to specify that the system must infer the data types of each column. Without this option all columns are considered strings

# Creating DataFrames from JSON files

---

- Spark SQL provides an API that allows creating a DataFrame directly from a textual file where each line contains a JSON object
  - Hence, the input file is not a “standard” JSON file
    - It must be properly formatted in order to have one JSON object (tuple) for each line
  - The format of the input file is complaint with the “JSON Lines text format”, also called newline-delimited JSON

# Creating DataFrames from JSON files

---

- Example of JSON Lines text formatted file compatible with the Spark expected format
  - {"name": "Michael"}
  - {"name": "Andy", "age": 30}
  - {"name": "Justin", "age": 19}
- The example file contains name and age of three persons
  - The age of the first person is unknown

# Creating DataFrames from JSON files

---

- The creation of a DataFrame from JSON files is based on the same method used for reading csv files
  - **load(path)** method of the **pyspark.sql.DataFrameReader** class
    - Path is the path of the input file
  - You get a **DataFrameReader** with the **read()** method of the **SparkSession** class

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

- or

```
df = spark.read.json(path, ...)
```

# Creating DataFrames from JSON files

---

- The same API allows also reading “standard” multiline JSON files
  - Set the multiline option to true by setting the argument **multiLine = True** on the defined DataFrameReader for reading “standard” JSON files
  - This feature is available since Spark 2.2.0
- Pay attention that reading a set of **small JSON files** from HDFS is **very slow**

# Creating DataFrames from JSON files: Example 1

---

- Create a DataFrame from a JSON Lines text formatted file containing the profiles of a set of persons
  - Each line of the file contains a JSON object containing name and age of a person
    - Age can assume the null value

# Creating DataFrames from JSON files: Example 1

---

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

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

# Creating DataFrames from JSON files: Example 1

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

This method is used to specify the format of the input file

# Creating DataFrames from JSON files: Example 2

---

- Create a DataFrame from a folder containing a set of “standard” multiline JSON files
- Each input JSON file contains the profile of one person
  - Name and Age
    - Age can assume the null value

# Creating DataFrames from JSON files: Example 2

---

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

# Create a DataFrame from persons.csv
df = spark.read.load("folder_JSONFiles/",
                      format="json",
                      multiLine=True)
```

# Creating DataFrames from JSON files: Example 2

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

# Create a DataFrame from persons.csv
df = spark.read.load("folder_JSONFiles/",
                      format="json",
                      multiLine=True)
```

This multiline option is set to true to specify that the input files are “standard” multiline JSON files

# Creating DataFrames from other data sources

---

- The `DataFrameReader class` (the same we used for reading a json file and store it in a DataFrame) provides other methods to read many standard (textual) formats and read data from external databases
  - Apache `parquet` files
  - External `relational database`, through a `JDBC` connection
  - `Hive` tables
  - Etc.

# Creating DataFrames from RDDs or Python lists

---

- The content of an **RDD of tuples** or the content of a **Python list of tuples** can be stored in a DataFrame by using the **spark.createDataFrame(data, schema)** method
  - **data**: RDD of tuples or Rows, Python list of tuples or Rows, or pandas DataFrame
  - **schema**: list of string with the names of the columns/attributes
    - **schema** is optional. If it is not specified the column names are set to `_1, _2, ..., _n` for input RDDs/lists of tuples

# Creating DataFrames from RDDs or Python lists: Example

---

- Create a DataFrame from a Python list containing the following data
  - (19, "Justin")
  - (30, "Andy")
  - (None, "Michael")
- The column names must be set to “age” and “name”

# Creating DataFrames from RDDs or Python lists: Example

---

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

# Create a Python list of tuples
profilesList = [(19, "Justin"), (30, "Andy"),
                 (None, "Michael")]

# Create a DataFrame from the profilesList
df = spark.createDataFrame(profilesList, ["age", "name"] )
```

# From DataFrame to RDD

---

- The `rdd` member of the `DataFrame` class returns an RDD of Row objects containing the content of the DataFrame on which it is invoked
- Each `Row` object is like a dictionary containing the values of a record
  - It contains column names in the keys and column values in the values

# From DataFrame to RDD

---

## Usage of the **Row** class

- The fields in it can be accessed:
  - like attributes (`row.key`)
    - where `key` is a column name
  - like dictionary values (`row["key"]`)
    - for `key` in `row` will search through row keys
- `asDict()` method:
  - Returns the Row content as a Python dictionary

# From DataFrame to RDD: 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

# From DataFrame to RDD: Example

---

```
# 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)
```

# From DataFrame to RDD: Example

---

```
# Define an RDD based on the content of
# the DataFrame
rddRows = df.rdd

# Use the map transformation to extract
# the name field/column
rddNames = rddRows.map(lambda row: row.name)

# Store the result
rddNames.saveAsTextFile(outputPath)
```

# **Operations on DataFrames**

---

# DataFrame operations

---

- Now you know how to create DataFrames
- How to manipulate them?
- How to compute statistics?

	Col1	Col2	Col3	....
Row 1				
Row 2				
Row 3	*			
*				

# DataFrame operations

---

- A set of **specific methods** are available for the DataFrame class
  - E.g., `show()`, `printSchema()`, `count()`, `distinct()`,  
`select()`, `filter()`
- Also the **standard** `collect()` and `count()` **actions** are available

# Show

---

- The `show(n)` method of the `DataFrame` class prints on the standard output the first `n` of the input DataFrame
  - Default value of `n`: 20

# Show: Example

---

- Create a DataFrame from a csv file containing the profiles of a set of persons
  - The content of persons.csv is

name	age
Andy	30
Michael	
Justin	19
- Print the content of the first 2 persons (i.e., the first 2 rows of the DataFrame)

# Show: Example

---

```
# 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.show(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 attributes of the data stored in the DataFrame

# Count

---

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

# Distinct

---

- The `distinct()` method of the `DataFrame` class `returns` a new `DataFrame` that `contains` `only` the `unique rows` of the input `DataFrame`
  - Pay attention that the distinct operation is always an heavy operation in terms of data sent on the network
  - A shuffle phase is needed

# Distinct: Example

---

- Create a DataFrame from a csv file containing the names of a set of persons
  - The content of names.csv is

name
Andy
Michael
Justin
Michael
  - The first line is the header
- Create a new DataFrame without duplicates

# Distinct: Example

---

```
# 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()
```

# Select

---

- The `select(col1, ..., coln)` method of the `DataFrame` class returns a new DataFrame that contains **only** the specified columns of the input DataFrame
- Use `'*'` as special column to select all columns
- Pay attention that the select method **can generate errors at runtime** if there are mistakes in the names of the columns

# Select: Example

---

- Create a DataFrame from the persons2.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - One line per person
    - Each line contains name, age, and gender of a person
  - Example

name,age,gender  
Paul,40,male  
John,40,male  
..
- Create a new DataFrame containing only name and age of the persons

# Select: Example

---

```
# 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")
```

# SelectExpr

---

- The `selectExpr(expression1, ..., expressionN)` method of the `DataFrame` class is a variant of the `select` method, where `expr` can be an SQL expression.
- Example:

```
df.selectExpr("name", "age")
```

```
df.selectExpr("name", "age + 1 AS new_age")
```

# SelectExpr: Example

---

- Create a DataFrame from the persons2.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Create a new DataFrame containing the same columns of the initial dataset plus an additional column called newAge containing the value of age incremented by one

# SelectExpr: Example

---

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

# Create a DataFrame from persons.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")
```

# SelectExpr: Example

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

```
# Create a DataFrame from persons.csv  
df = spark.read.load("persons2.csv",  
                      format="csv",  
                      header=True,
```

This part of the expression is used to specify the name of the column associated with the result of the first part of the expression in the returned DataFrame. Without this part of the expression, the name of the returned column will be "age+1"

```
dfNew = df.selectExpr("name", "age", "gender",  
                      "age+1 as newAge")
```

# Filter

---

- The **filter(conditionExpr)** method of the **DataFrame** class returns a new DataFrame that contains only the rows satisfying the specified condition
  - The **condition** is a Boolean **SQL expression**
  - Pay attention that this version of the filter method **can generate errors at runtime** if there are errors in the filter expression
    - The parameter is a string and the system cannot check the correctness of the expression at compile time

# Filter: Example

---

- Create a DataFrame from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name and age of a person
- Create a new DataFrame containing only the persons with age between 20 and 31

# Filter: Example

---

```
# 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")
```

# Where

---

- The `where(expression)` method of the `DataFrame` class is an `alias` of the `filter(conditionExpr)` method

# Join

---

- The `join(right, on, how)` method of the `DataFrame` class is used to join two DataFrames
  - It returns a DataFrame that contains the join of the tuples of the two input DataFrames based on the `on` join condition

# Join

---

- **on**: the join condition
  - It can be:
    - A string: the join column
    - A list of strings: multiple join columns
    - A condition/an expression on the columns.
      - E.g.:
        - `joined_df = df.join(df2, df.name == df2.name)`
  - **how**: the type of join
    - `inner`, `cross`, `outer`, `full`, `full_outer`, `left`, `left_outer`, `right`, `right_outer`, `left_semi`, and `left_anti`
    - Default: `inner`

# Join

---

- Pay attention that this method
  - Can generate errors at runtime if there are errors in the join expression

# Join: Example

---

- Create two DataFrames
  - One based on the persons\_id.csv file that contains the profiles of a set of persons
    - Schema: uid, name, age
  - One based on the liked\_sports.csv file that contains the liked sports for each person
    - Schema: uid, sportname
- Join the content of the two DataFrames (uid is the join column) and show it on the standard output

# Join: Example

---

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

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                             format="csv",
                             header=True,
                             inferSchema=True)

# Read liked_sports.csv and store it in a DataFrame
dfUidSports = spark.read.load("liked_sports.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()
```

# Join: Example

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

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                             format="csv",
                             header=True,
                             inferSchema=True)

# Read liked_sports.csv and store it in a DataFrame
dfUidSports = spark.read.load("liked_sports.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()
```

Specify the join condition on the uid columns

# Join: Example #2

---

- Create two DataFrames
  - One based on the persons\_id.csv file that contains the profiles of a set of persons
    - Schema: uid,name,age
  - One based on the banned.csv file that contains the banned users
    - Schema: uid,bannedmotivation
- Select the profiles of the non-banned users and show them on the standard output

# Join: Example #2

---

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

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.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)

like NOT IN for SQL

# 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()
```

# Join: Example #2

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

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.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 [join condition] on the uid column of the BannedUsers
dfSelectedProfiles = dfPersons.join(dfBannedUsers,
                                      dfPersons.uid == dfBannedUsers.uid,
                                      "left_anti")

# Print the result on the standard output
dfSelectedProfiles.show()
```

Specify the (anti) join condition on the uid columns

# Join: Example #2

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

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.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 between input DataFrames
dfSelectedProfiles = dfPersons.join(dfBannedUsers,
                                      dfPersons.uid == dfBannedUsers.uid,
                                      "left_anti")

# Print the result on the standard output
dfSelectedProfiles.show()
```

Use Left Anti Join

# Aggregates functions

---

- Aggregate functions are provided to compute aggregates over the set of values of columns
  - Some of the provided aggregate functions/methods are:
    - `avg(column)`, `count(column)`, `sum(column)`, `abs(column)`, etc.
  - Each aggregate function returns one value computed by considering all the values of the input column

# Aggregates functions

---

- The `agg(expr)` method of the DataFrame class is used to specify which aggregate function we want to apply on one input column
  - The `result` is a DataFrame containing `one single row` and `one single column`
  - The `name of the return column` is “`function_name(column)`”
- Pay attention that this methods **can generate errors at runtime**
  - E.g., wrong attribute name, wrong data type

# Aggregates functions: Example

---

- Create a DataFrame from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name and age of a person
- Create a Dataset containing the average value of age

# Aggregates functions: Example

---

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

avg(age)

26.0

# Aggregates functions: Example

---

```
# 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"})
```

# groupBy and aggregates functions

---

- The method **groupBy(col1, ..., coln)** method of the **DataFrame** class combined with a set of aggregate methods can be used to split the input data in groups and compute aggregate function over each group
- Pay attention that this methods **can generate errors at runtime** if there are semantic errors
  - E.g., wrong attribute names, wrong data types

# groupBy and aggregates functions

---

- Specify which attributes are used to split the input data in groups by using the **groupBy(col1, ..., coln)** method
- Then, apply the aggregate functions you want to compute by final result
  - The **result is a DataFrame**

# groupBy and aggregates functions

---

- Some of the provided **aggregate functions**/methods are
  - `avg(column)`, `count(column)`, `sum(column)`, `abs(column)`, etc.
  - The `agg(..)` method can be used 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

# groupBy and aggregates functions:

## Example 1

---

- Create a DataFrame from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name and age of a person
- Create a DataFrame containing the for each name the average value of age

# groupBy and aggregates functions: Example 1

---

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

name,avg(age)

Andy,35

Michael,15

Justin,19

# groupBy and aggregates functions:

## Example 1

---

```
# 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)

grouped = df.groupBy("name").avg("age")
```

# groupBy and aggregates functions: Example 2

---

- Create a DataFrame from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name and age of a person
- Create a DataFrame containing the for each name the average value of age and the number of person with that name

# groupBy and aggregates functions: Example 2

---

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

name,avg(age),count(name)

Andy,35,2

Michael,15,1

Justin,19,1

# groupBy and aggregates functions: Example 2

---

```
# 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)

grouped = df.groupBy("name")
          .agg({ "age": "avg", "name": "count" })
```

# Sort

---

- The `sort(col1, ..., coln, ascending=True)` method of the `DataFrame` class returns a new DataFrame that
  - contains the same data of the input one
  - but the content is sorted by `col1, ..., coln`
  - `Ascending` determines ascending vs. descending

# DataFrames and the SQL language

---

# DataFrames and the SQL language

---

- Sparks allows querying the content of a DataFrame also by using the SQL language
  - In order to do this a “table name” must be assigned to a DataFrame
- The `createOrReplaceTempView` (`tableName`) method of the `DataFrame` class can be used to assign a “table name” to the DataFrame on which it is invoked

# DataFrames and the SQL language

---

- Once the DataFrame has been mapped to “table names”, SQL-like queries can be executed
  - The executed queries return DataFrame objects
- The **sql(query)** method of the **SparkSession** class can be used to execute an SQL-like query
  - **query** is an SQL-like query
- Currently some SQL features are not supported
  - E.g., nested subqueries in the WHERE clause are not allowed

# DataFrames and the SQL language: Example 1

---

- 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 new DataFrame containing only the persons with age between 20 and 31 and print them on the standard output
  - Use the SQL language to perform this operation

# DataFrames and the SQL language: Example 1

---

```
# 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()
```

# DataFrames and the SQL language: Example 2

---

- Create two DataFrames
  - One based on the persons\_id.csv file that contains the profiles of a set of persons
    - Schema: uid,name,age
  - One based on the liked\_sports.csv file that contains the liked sports for each person
    - Schema: uid,sportname
- Join the content of the two DataFrames and show it on the standard output

# DataFrames and the SQL language: Example 2

---

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

# Read persons_id.csv and store it in a DataFrame
dfPersons = spark.read.load("persons_id.csv",
                            format="csv",
                            header=True,
                            inferSchema=True)

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

# Read liked_sports.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")
```

# DataFrames and the SQL language: Example 2

---

```
# 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()
```

# DataFrames and the SQL language: Example 3

---

- Create a DataFrame from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - 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

# DataFrames and the SQL language: Example 3

---

- Input file

name,age

Andy,30

Michael,15

Justin,19

Andy,40

- Expected output

name,avg(age),count(name)

Andy,35,2

Michael,15,1

Justin,19,1

# DataFrames and the SQL language: Example 3

---

```
# 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")

# Define groups based on 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()
```

# **Save DataFrames**

---

# Save DataFrames

---

- The content of DataFrames can be stored on disk by using two approaches
  - 1 Convert DataFrames to traditional RDDs by using the `rdd` method of the `DataFrame`  
And then use `saveAsTextFile(outputFolder)`
  - 2 Use the `write()` method of DataFrames, that returns a `DatFrameWriter` class instance

# Save DataFrames: Example 1

---

- Create a DataFrame from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Store the DataFrame in the output folder by using the saveAsTextFile(..) method

# Save DataFrames: Example 1

---

```
# 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)
```

# Save DataFrames: Example 2

---

- Create a DataFrame from the persons.csv file that contains the profiles of a set of persons
  - The first line contains the header
  - The others lines contain the users' profiles
    - Each line contains name, age, and gender of a person
- Store the DataFrame in the output folder by using the write() method
  - Store the result by using the CSV format

# Save DataFrames: Example 2

---

```
# 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)

    this is always a folder and the number of files with the same header in this folder is equal
    to the number of partition of df
```

# **UDFs: User Defined Functions**

---

# UDFs: User Defined Functions

---

- Spark SQL provides a set of system predefined functions
  - `hour(Timestamp)`, `abs(Integer)`, ..
  - Those functions can be used in some transformations (e.g., `selectExpr(..)`, `sort(..)`) but also in the SQL queries
- Users can defined their personalized functions
  - They are called User Defined Functions (UDFs)

# UDFs: User Defined Functions

---

- UDFs are defined/registered by invoking the `udf().register(name, function, datatype)` on the SparkSession
  - `name`: name of the defined UDF
  - `function`: lambda function used to specify how the parameters of the function are used to generate the returned value
    - One or more input parameters
    - One single returned value
  - `datatype`: SQL data type of the returned value

# UDFs: User Defined Functions – Example

---

- Define a UDFs that, given a string, returns the length of the string

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

# Define the UDF
# name: length
# output: integer value
spark.udf.register("length", lambda x: len(x))
```

# UDFs: User Defined Functions – Example

---

- Use of the defined UDF in a `selectExpr` transformation

```
result = inputDF.selectExpr("length(name) as size")
```

- Use of the defined UDF in a SQL query

```
result = spark.sql("SELECT length(name) FROM profiles")
```

# **Data warehouse methods**

---

# cube and rollup

---

- The method **cube(col<sub>1</sub>, .., col<sub>n</sub>)** of the **DataFrame** class can be used to create a multi-dimensional cube for the input DataFrame
  - On top of which aggregate functions can be computed for each “group”
- The method **rollup(col<sub>1</sub>, .., col<sub>n</sub>)** of the **DataFrame** class can be used to create a multi-dimensional rollup for the input DataFrame
  - On top of which aggregate functions can be computed for each “group”

# cube and rollup

---

- Specify which attributes are used to split the input data in “groups” by using `cube(col1, ..., coln)` or `rollup(col1, ..., coln)`, respectively
- Then, apply the aggregate functions you want to compute for each group of the cube/rollup
  - The result is a DataFrame

# **cube and rollup**

---

- The same aggregate functions/methods we already discussed for groupBy can be used also for cube and rollup

# cube and rollup: Example

---

- Create a DataFrame from the purchases.csv file
  - The first line contains the header
  - The others lines contain the quantities of purchased products by users
    - Each line contains userid,productid,quantity
- Create a first DataFrame containing the result of the cube method. Define one group for each pair userid, productid and compute the sum of quantity in each group
- Create a second DataFrame containing the result of the rollup method. Define one group for each pair userid, productid and compute the sum of quantity in each group

# cube and rollup: Example

---

- Input file

userid,productid,quantity

U1,p1,10

U1,p1,20

U1,p2,20

U1,p3,10

U2,p1,20

U2,p3,40

U2,p3,30

# cube and rollup: Example

---

- Expected output - cube

userid,productid,sum(quantity)

null	null	150
null	p1	50
null	p2	20
null	p3	80
u1	null	60
u1	p1	30
u1	p2	20
u1	p3	10
u2	null	90
u2	p1	20
u2	p3	70

# cube and rollup: Example

---

- Expected output - rollup

userid,productid,sum(quantity)

null	null	150
u1	null	60
u1	p1	30
u1	p2	20
u1	p3	10
u2	null	90
u2	p1	20
u2	p3	70

# cube and rollup: Example

---

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

# Read purchases.csv and store it in a DataFrame
dfPurchases = spark.read.load("purchases.csv", \
                                format="csv", \
                                header=True, \
                                inferSchema=True)

dfCube=dfPurchases.\
cube("userid","productid").agg({ "quantity": "sum" })

dfRollup=dfPurchases\
.rollup("userid","productid")\
.agg({ "quantity": "sum" })
```

# **Set methods**

---

# Set transformations

---

- Similarly to RDDs also **DataFrames** can be combined by using set transformations
  - `df1.union(df2)`
  - `df1.intersect(df2)`
  - `df1.subtract(df2)`

# Broadcast join

---

# Broadcast join and DataFrames

---

- Spark SQL automatically implements a broadcast version of the join operation if one of the two input DataFrames is small enough to be stored in the main memory of each executor

# Broadcast join and DataFrames

---

- We can suggest/force it by creating a broadcast version of a DataFrame
- E.g.,

```
dfPersonLikesBroadcast = dfUidSports\  
    .join(broadcast(dfPersons),\  
          dfPersons.uid == dfUidSports.uid)
```

# Broadcast join and DataFrames

---

- We can suggest/force it by creating a broadcast version of a DataFrame
- E.g.,

```
dfPersonLikesBroadcast = dfUidSports\  
    .join(broadcast(dfPersons),\  
          dfPersons.uid == dfUidSports.uid)
```

In this case we specify that dfPersons must be broadcasted and hence Spark will execute the join operation by using a broadcast join

# **Execution plan**

---

# Explain execution plan

---

- The method `explain()` can be invoked on a `DataFrame` to print on the standard output the execution plan of the part of the code that is used to compute the content of the DataFrame on which `explain()` is invoked