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
# T
# T
# o

distRDDList = sc.parallelize(inputList)
```

No computation occurs when sc.parallelize() 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 distRDDList or on one of its "descendant" RDDs)

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

```python
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)
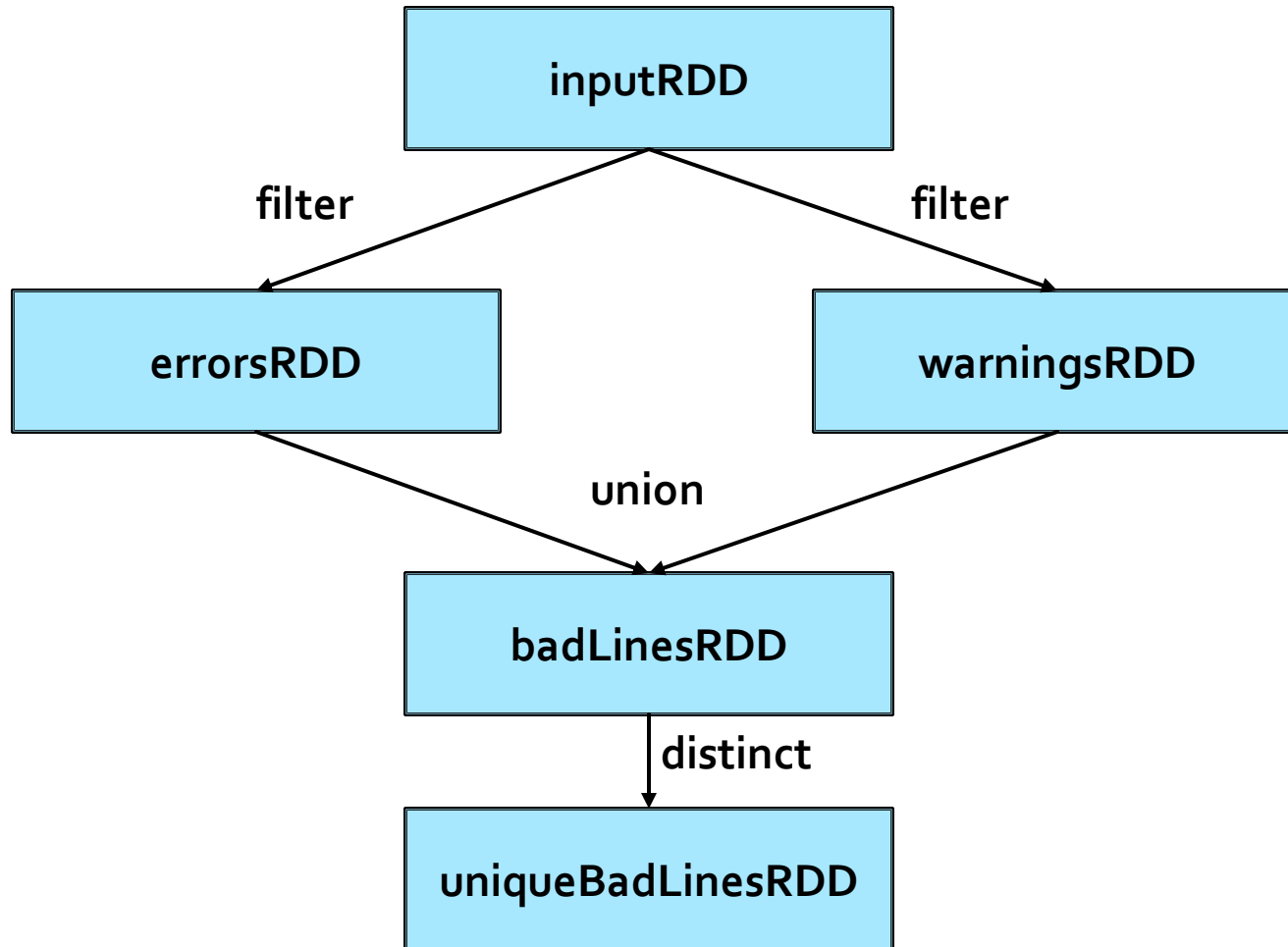
Example of lineage graph (DAG)

- 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

...

in...

> 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')>=o: 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(), substract(), 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. 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 – 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 ...
inputList = [1, 2, 3, 3]
inputRDD = sc.parallelize(inputList)

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

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

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
lenghtsRDD = 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
lenghtsRDD = 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
lenghtsRDD = 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(' '))

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)
```
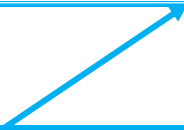
SortBy transformation: Example 1

# 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))

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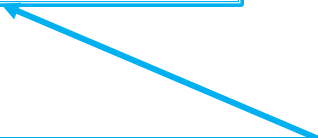
Each element of the returned RDD is a pair (tuple) of integer elements

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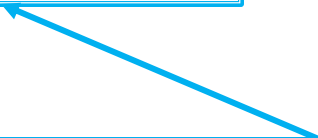
Each element of the returned RDD is a pair (tuple) containing an integer and string

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)	Apply a function to each element in the RDD and return an RDD of the result. The applied function return a set of elements (from 0 to many) for each element of the "input" RDD. The "input" RDD and the new RDD can have a different data type.	flatMap(lambda x: list(range(x,4))) For each input element x, the set of elements with values from x to 3 are returned	[1,2,3,2, 3,3,3]

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)	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(). Duplicate values are not removed. All the RDDs have the same data type.	inputRDD1.union (inputRDD2)	[1, 2, 2, 3, 3, 3, 4, 5]
intersection(other)	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(). All the RDDs have the same data type.	inputRDD1.intersection (inputRDD2)	[3]

Basic transformations based on two RDDs: Summary

Transformation	Purpose	Example	Result
subtract(other)	Return a new RDD the elements appearing only in the "input"" RDD and not in the one passed as parameter to subtract(). All the RDDs have the same data type.	inputRDD1.subtract (inputRDD2)	[1, 2, 2]
cartesian(other)	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(). All the RDDs have the same data type.	inputRDD1.cartesian(in putRDD2)	[(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 occurrencies of each name
namesOccurrences = namesRDD.countByValue()

CountByValue action: Example 1

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

# Compute the number of occurrencies 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 takeSampleaction 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)
```
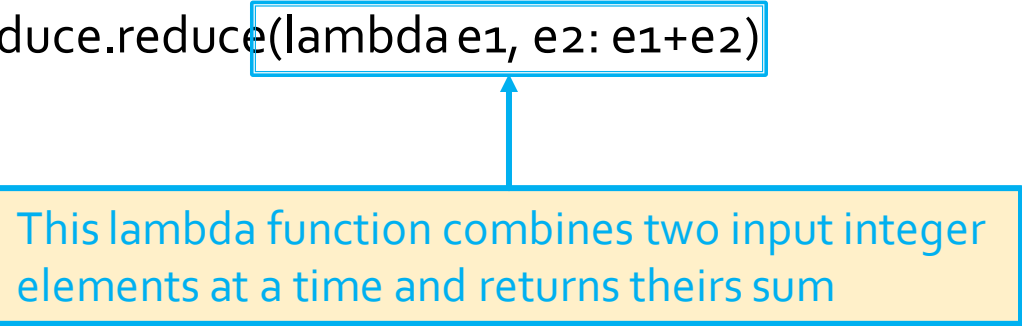
Reduce action: Example 1

.....
# 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 theirs 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

- 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

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)
```

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!=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 woks

- 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 $\{L_1, .., L_n\}$
- 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= $\{p_1, .., p_n\}$**

  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 woks

- 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]))
```

Instantiate the zero value

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]

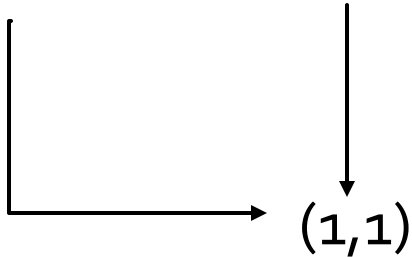# Aggregate action: Simulation

Partition #1

[1, 2]   acc=(0,0)

Partition #2

[3, 3]   acc=(0,0)

Aggregate action: Simulation

Partition #1

[1, 2]   acc=(0,0)

(1,1)

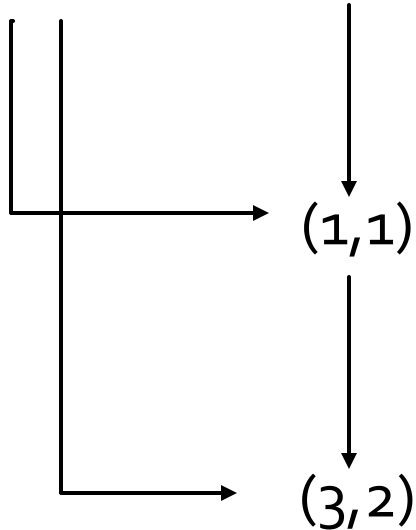Partition #2
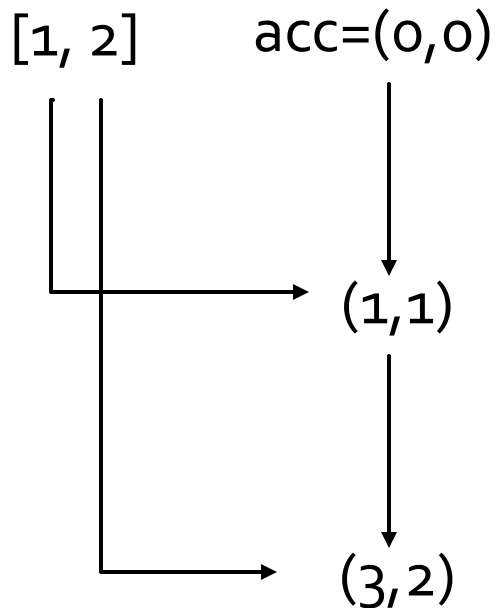
[3, 3]   acc=(0,0)

Aggregate action: Simulation

Partition #1

[1, 2]   acc=(0,0)

(1,1)

(3,2)

Partition #2

[3, 3]   acc=(0,0)

Aggregate action: Simulation

Partition #1

[1, 2] acc=(0,0)

(1,1)

(3,2)

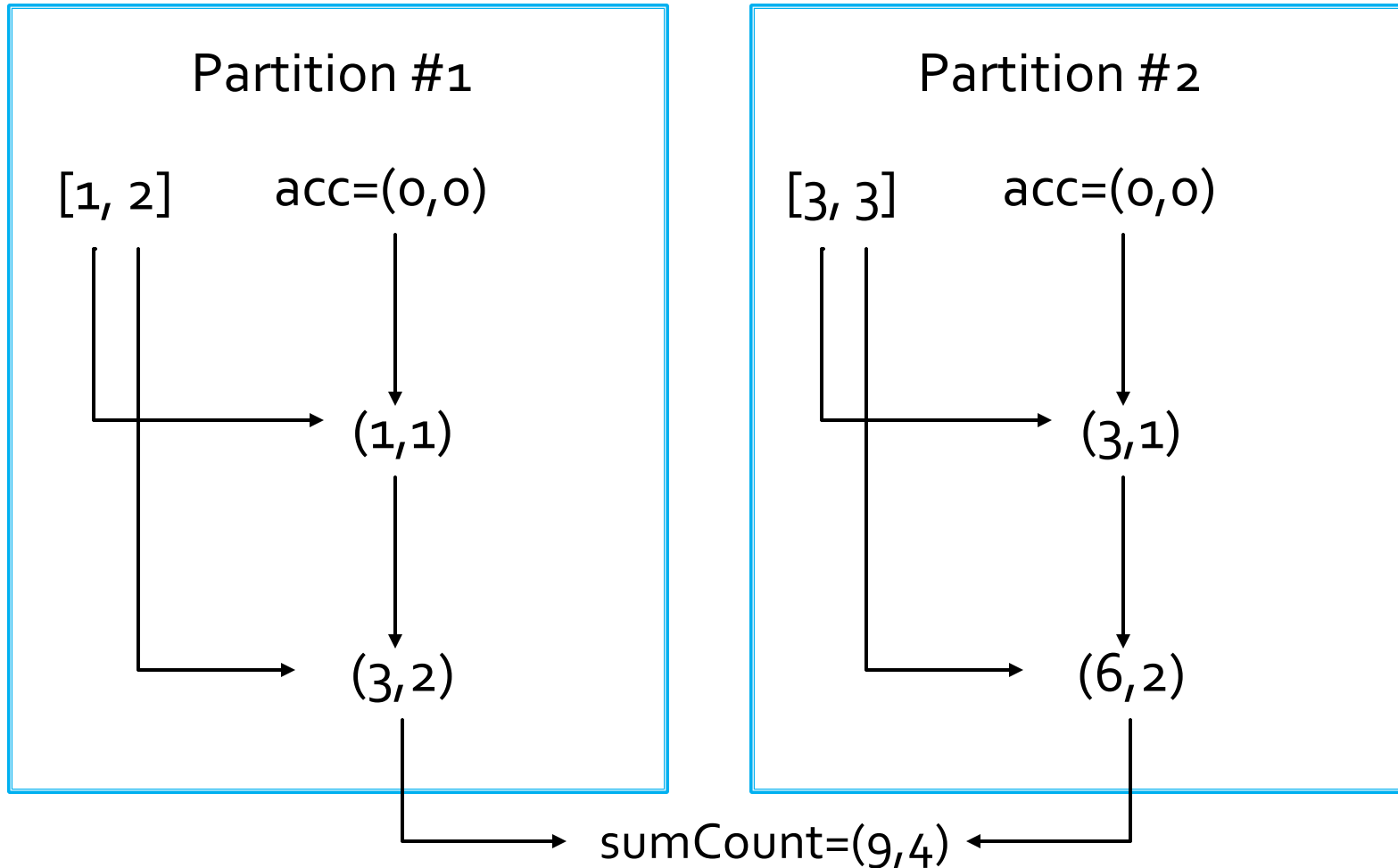Partition #2

[3, 3] acc=(0,0)

(3,1)

(6,2)

Aggregate action: Simulation

Partition #1

[1, 2]   acc=(0,0)

(1,1)

(3,2)

Partition #2

[3, 3]   acc=(0,0)

(3,1)

(6,2)

sumCount=(9,4)
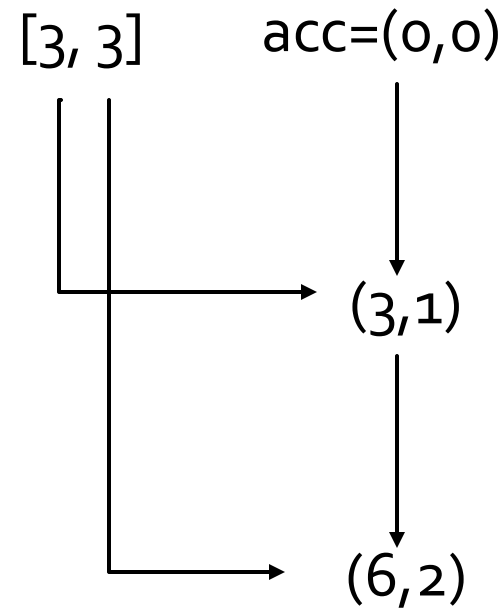
Basic actions: Summary

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()	Return a python list containing all the elements of the RDD on which it is applied. The objects of the RDD and objects of the returned list are objects of the same class.	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)	Return a Python list containing the first num elements of the RDD. The objects of the RDD and objects of the returned list are objects of the same class.	inputRDD.take(2)	[1,2]
first()	Return the first element of the RDD	first()	1
top(num)	Return a Python list containing the top num elements of the RDD based on the default sort order/comparator of the objects. The objects of the RDD and objects of the returned list are objects of the same class.	inputRDD.top(2)	[3,3]

Basic actions: Summary

Action	Purpose	Example	Result
takeSample(withReplacement, num) takeSample(withReplacement, num, seed)	Return a (Python) List containing a random sample of size n of the RDD. The objects of the RDD and objects of the returned list are objects of the same class.	inputRDD.takeSample (False, 1)	Nondeterministic
reduce(f)	Return a single Python object obtained by combining the values of the objects of the RDD by using a user provide "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(0, lambda v1, v2: v1+v2) The passed "function" is the sum and the passed zeroValue is 0	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)