

Big data for internet applications

RDD-based programming

Basic Actions

Basic RDD actions

- The Spark's actions can
 - Return the content of the RDD and “store” it in a local python variable of the Driver program
 - **Pay attention to the size of the returned value**
 - Or store the content of an RDD in an output file or database table
- The basic actions returning (python) objects to the Driver program 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()
```

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)
```

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

.....

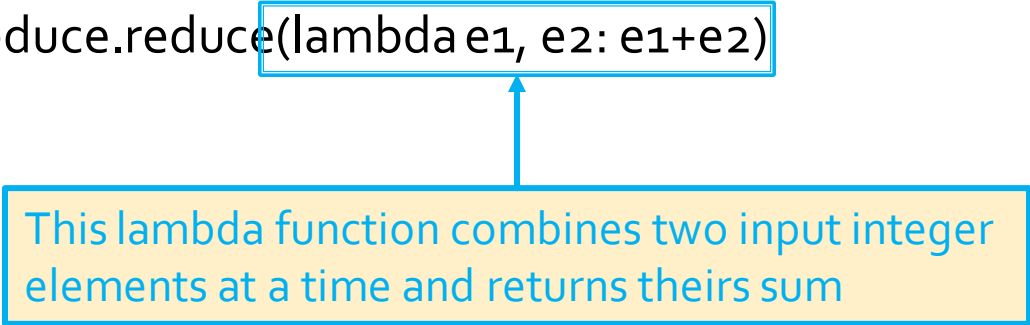
Create an RDD of integers. Load the values 1, 2, 3, 3 in this RDD

```
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 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

- 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 strings containing the values ['This ', 'is ', 'a ', 'test']  
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 ($T \neq U$)
 - 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

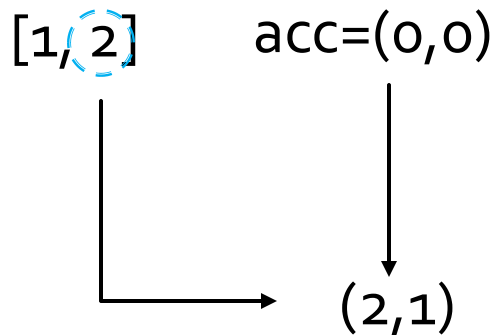
[1, 2] $\text{acc}=(0,0)$

Partition #2

[3, 3] $\text{acc}=(0,0)$

Aggregate action: Simulation

Partition #1



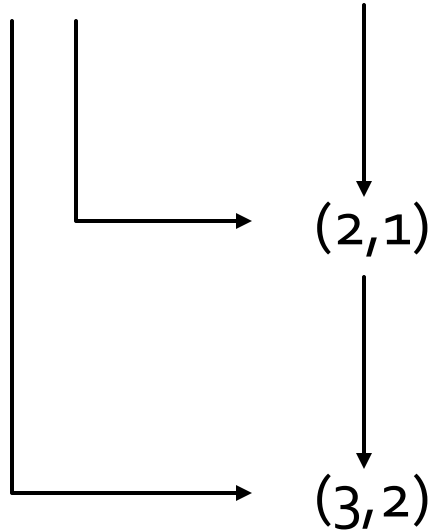
Partition #2

$[3, 3]$ $\text{acc}=(0,0)$

Aggregate action: Simulation

Partition #1

$[1, 2]$ $\text{acc}=(0,0)$

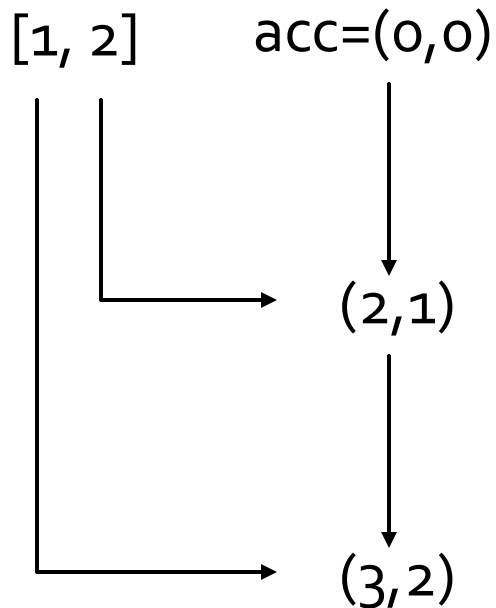


Partition #2

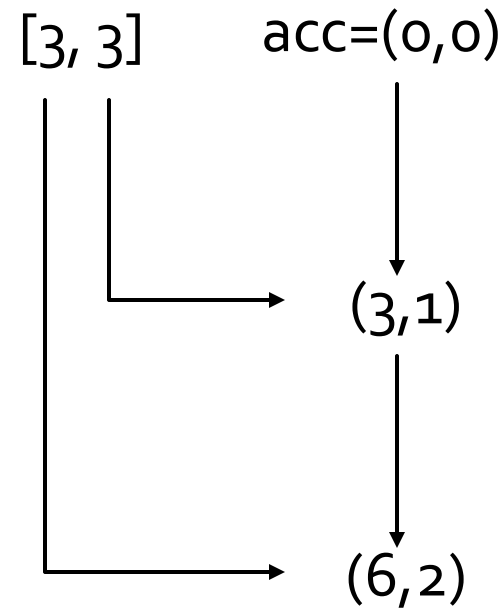
$[3, 3]$ $\text{acc}=(0,0)$

Aggregate action: Simulation

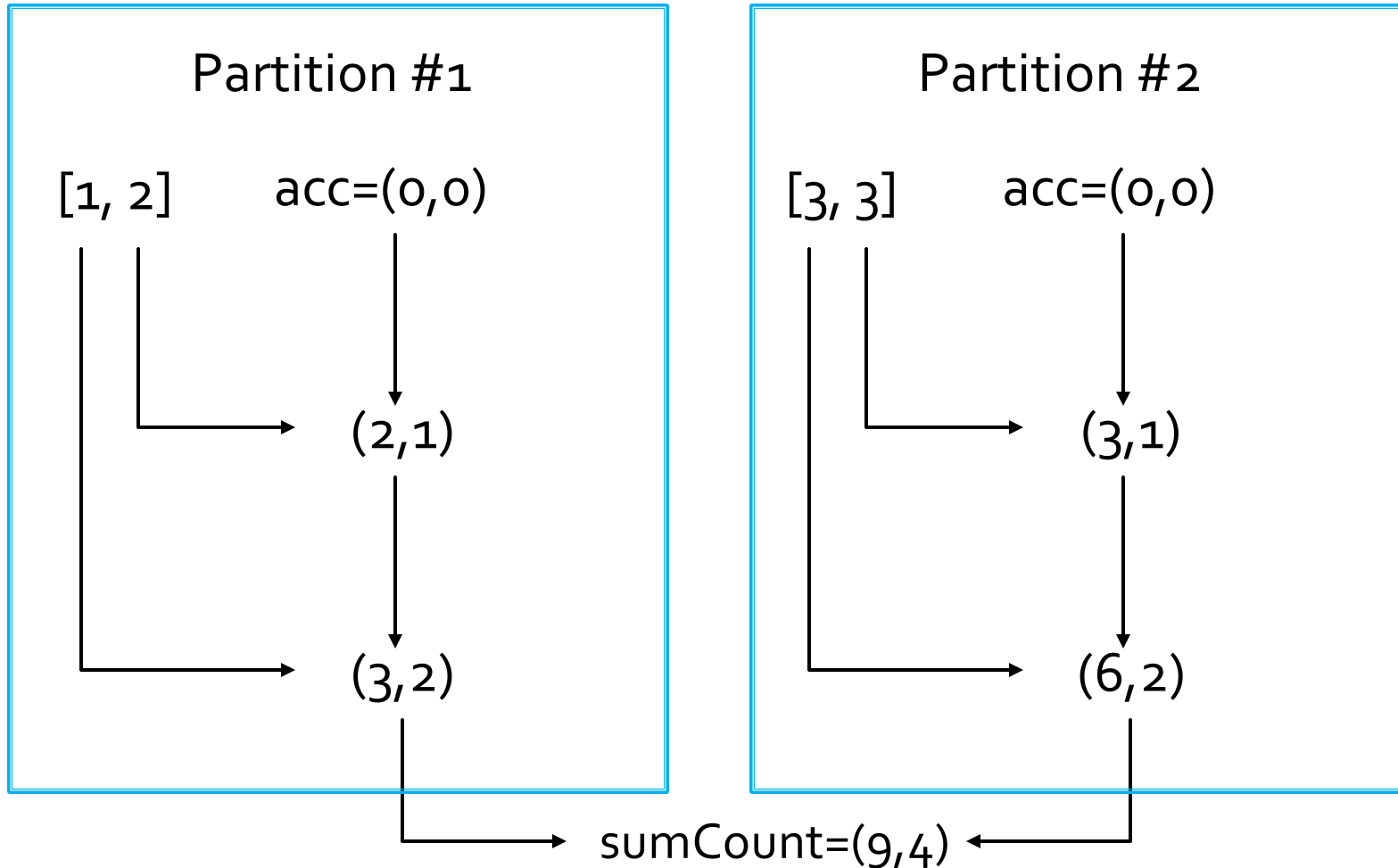
Partition #1



Partition #2



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
<code>collect()</code>	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.	<code>inputRDD.collect()</code>	<code>[1,2,3,3]</code>
<code>count()</code>	Return the number of elements of the RDD	<code>inputRDD.count()</code>	4
<code>countByKey()</code>	Return a Map object containing the information about the number of times each element occurs in the RDD.	<code>inputRDD.countByKey()</code>	<code>[(1, 1), (2, 1), (3, 2)]</code>

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
<code>takeSample(withReplacement, num)</code> <code>takeSample(withReplacement, num, seed)</code>	Return a (Java) 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.	<code>inputRDD.takeSample(False, 1)</code>	Nondeterministic
<code>reduce(f)</code>	Return a single Java 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.	<code>inputRDD.reduce(lambda e1, e2: e1+e2)</code> The passed "function" is the sum	9

Basic actions: Summary

Action	Purpose	Example	Result
<code>fold(zeroValue, op)</code>	Same as <code>reduce</code> but with the provided zero value.	<code>inputRDD.fold(o, lambda v1, v2: v1+v2)</code> The passed "function" is the sum and the passed <code>zeroValue</code> is <code>o</code>	9
<code>Aggregate(zeroValue, seqOp, combOp)</code>	Similar to <code>reduce()</code> but used to return a different type.	<code>inputRDD.aggregate(zeroValue, lambda acc, e: (acc[0]+e, acc[1]+1), lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))</code> 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)