

## Big data: architectures and data analytics

## 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 Java 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 (Java) objects to the Driver are
  - collect(), count(), countByValue(), take(), top(), takeSample(), reduce(), fold(), aggregate(), foreach()

4

## Syntax

- In the following, the following syntax is used
  - T = Type of the objects of the RDD on which the transformation is applied
  - The RDD on which the action is applied is referred as "input" RDD

5

## Collect action

## Collect action

- Goal
  - The collect action returns a local Java 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 `List<T> collect()` method of the `JavaRDD<T>` class

7

## 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 Java list that is instantiated in the Driver

8

## Collect action: Example 1

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

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

9

## Collect action: Example 1

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

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

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

10

## Collect action: Example 1

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

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

retrievedValues is a local Java 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

11

## Count action

## Count action

- Goal
  - Count the number of elements of an RDD
- Method
  - The count action is based on the `long count()` method of the `JavaRDD<T>` class

13

## Count action: Example 1

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

14

## Count action: Example 1

```
// Read the content of the two input textual files
JavaRDD<String> inputRDD1 = sc.textFile("document1.txt");
JavaRDD<String> inputRDD2 = sc.textFile("document2.txt");

// Count the number of lines of the two files = number of elements
// of the two RDDs
long numLinesDoc1 = inputRDD1.count();
long numLinesDoc2 = inputRDD2.count();

if (numLinesDoc1 > numLinesDoc2) {
    System.out.println("document1.txt");
}
else {
    if (numLinesDoc2 > numLinesDoc1)
        System.out.println("document2.txt");
    else
        System.out.println("Same number of lines");
}
}
```

15

## CountByValue action

## CountByValue action

- Goal
  - The countByValue action returns a local Java Map object containing the information about the number of times each element occurs in the RDD
- Method
  - The countByValue action is based on the `java.util.Map<T, java.lang.Long> countByValue()` method of the `JavaRDD<T>` class

17

## 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

18

## CountByValue action: Example 1

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

// Compute the number of occurrences of each name
java.util.Map<String, java.lang.Long> namesOccurrences =
namesRDD.countByValue();
```

29

## CountByValue action: Example 1

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

// Compute the number of occurrences of each name
java.util.Map<String, java.lang.Long> namesOccurrences =
namesRDD.countByValue();
```

Also in this case, pay attention to the size of the returned map (i.e., the number of names in this case). Use the `countByValue()` action if and only if you are sure that the returned `java.util.Map` is small. Otherwise, use an appropriate chain of Spark's transformations and write the final result in a file by using the `saveAsTextFile` method.

30

## Take action

## Take action

- Goal
  - The `take(n)` action returns a local Java list of objects containing the first `n` 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 `List<T> take(int n)` method of the `JavaRDD<T>` class

22

## Take action: Example 1

- 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 Java list that is instantiated in the Driver

23

## Take action: Example 1

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

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

24

## First action

## First action

- Goal
  - The first() action returns a local Java 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 `T first()` method of the `JavaRDD<T>` class

26

## First vs Take(1)

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

27

## Top action

## Top action

- Goal
  - The top(n) action returns a local Java list of objects containing the top `n` (largest) elements of the considered RDD
    - The ordering is the default one of class T (the class of the objects of the RDD)
    - The descending order is used
- Method
  - The top action is based on the `List<T> top(int n)` method of the `JavaRDD<T>` class

29

## Top action: Example 1

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

30

## Top action: Example 1

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

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

31

## TakeOrdered action

## TakeOrdered action

- Goal
  - The `takeOrdered(n, comparator<T>)` action returns a local Java list of objects containing the top `n` (smallest) elements of the considered RDD
    - The ordering is specified by the developer by means of a class implementing the `java.util.Comparator<T>` interface
- Method
  - The `takeOrdered` action is based on the `List<T> takeOrdered(int n, java.util.Comparator<T> comp)` method of the `JavaRDD<T>` class

33

## TakeSample action

## TakeSample action

- Goal
  - The `takeSample(withReplacement, n, [seed])` action returns a local Java list of objects containing `n` random elements of the considered RDD
- Method
  - The `takeSample` action is based on the `List<T> takeSample(boolean withReplacement, int n)` method of the `JavaRDD<T>` class
    - `withReplacement` specifies if the random sample is with replacement (true) or not (false)

35

## TakeSample action

- Method
  - The `List<T> takeSample(boolean withReplacement, int n, long seed)` method of the `JavaRDD<T>` class is used when we want to set the seed

36

## TakeSample action: Example 1

- 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 Java list that is instantiated in the Driver

37

## TakeSample action: Example 1

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

// Retrieve randomly two elements of the inputRDD and store them in
// a local Java list
List<Integer> randomValues = inputRDD.takeSample(true, 2);
```

38

## Reduce action

## Reduce action

- Goal
  - Return a single Java object obtained by combining the objects of the 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 class (T)

40

## Reduce action

- Method
  - The reduce action is based on the **T reduce(Function2<T, T, T> f)** method of the **JavaRDD<T>** class
  - A lambda function implementing the call method of the Function2<T, T, T> interface is passed to the reduce method
    - The **public T call(T element1, T element2)** method of the Function2<T, T, T> interface must be implemented
      - It contains the code that is applied to combine the values of the elements of the RDD

41

## Reduce action: how it works

- Suppose  $L$  contains the list of elements of the "input" RDD
- To compute the final element, 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

42

## Reduce action: how it works

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

43

## Reduce action: how it works

- The “function” 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

44

## 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 Java integer variable in the Driver

45

## Reduce action: Example 1

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

// Compute the sum of the values;
Integer sum = inputRDDReduce.reduce(
    (element1, element2) -> element1 + element2);
```

46

## Reduce action: Example 1

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

// Compute the sum of the values;
Integer sum = inputRDDReduce.reduce(
    (element1, element2) -> element1 + element2);
```

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

47

## 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 Java integer variable in the Driver

48



## Reduce action: Example 2

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

// Compute the maximum value
Integer max = inputRDDReduce.reduce(
    (element1, element2) ->
    {
        if (element1 > element2)
            return element1;
        else
            return element2;
    });
```

49

## Fold action

## Fold action

- Goal
  - Return a single Java object obtained by combining the objects of the RDD 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 "zero" value is also specified

51

## Fold action

- Method
  - The fold action is based on the **T fold(T zeroValue, Function2<T, T, T> f)** method of the **JavaRDD<T>** class
  - The "zero" value of type T is passed
  - A lambda function implementing the call method of the **Function2<T, T, T>** interface is passed to the fold method
    - The **public T call(T element1, T element2)** method of the **Function2<T, T, T>** interface must be implemented
      - It contains the code that is applied to combine the values of the elements of the RDD

52

## 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

53

## Aggregate action

## Aggregate action

- Goal
  - Return a single Java 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** and **commutative**
    - 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 ()

55

## Aggregate action

- Method
  - The aggregate action is based on the **U aggregate(U zeroValue, Function2<U,T,U> seqOp, Function2<U,U,U> combOp)** method of the **JavaRDD<T>** class
  - The "input" RDD contains objects of type T while the returned object is of type 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 zero value
    - 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

56

## Aggregate action

- The first "function" is based on a lambda function implementing the call method of the Function2<U, T, U> interface
  - The **public U call(U element1, T element2)** method of the Function2<U, T, U> interface must be implemented
    - It contains the code that is applied to combine the zero value, and the intermediate values, with the elements of the RDD
- The second "function" is based on a lambda function implementing the call method of the Function2<U, U, U> interface
  - The **public U call(U element1, U element2)** method of the Function2<U, U, U> interface must be implemented
    - It contains the code that is applied to combine two elements of type U returned as partial results by two different partitions

57

## 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<sub>1</sub>, ..., L<sub>n</sub>}**
- 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<sub>1</sub>, ..., p<sub>n</sub>}**
  2. Apply the second user specified "function" on a pair of elements **p<sub>1</sub>** and **p<sub>2</sub>** in **P** and obtain a new element **p<sub>new</sub>**
  3. Remove the "original" elements **p<sub>1</sub>** and **p<sub>2</sub>** from **P** and then insert the element **p<sub>new</sub>** in **P**
  4. If **P** contains only one value then return it as final result of the aggregate action. Otherwise, return to step 2

58

## Aggregate action: how it woks

- Suppose that
  - **L<sub>i</sub>** 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<sub>i</sub>** the aggregate action operates as follows
  1. Set **accumulator** to **zeroValue** (accumulator=zeroValue)
  2. Apply the first user specified "function" on **accumulator** and an elements **e<sub>i</sub>** in **L<sub>i</sub>** and update **accumulator** with the value returned by the function
  3. Remove the "original" elements **e<sub>i</sub>** from **L<sub>i</sub>**
  4. If **L<sub>i</sub>** is empty return **accumulator** as (final) partial result of the current partition. Otherwise, return to step 2

59

## 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 and finally "store" in a local Java variable of the Driver the average computed over the values of the input RDD

60

## Aggregate action: Example 1

```
// Define a class to store two integers: sum and numElements
class SumCount implements Serializable {
    public int sum;
    public int numElements;

    public SumCount(int sum, int numElements) {
        this.sum = sum;
        this.numElements = numElements;
    }

    public double avg() {
        return sum / (double) numElements;
    }
}
```

61

## Aggregate action: Example 1

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

62

## Aggregate action: Example 1

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

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

63

## Aggregate action: Example 1

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

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

64

## Aggregate action: Example 1

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

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

Define the code that is used to implement the call method of the `Function2<SumCount, Integer, SumCount>` interface. It is used to combine the elements of the input RDD (Integer objects) with the zero value (SumCount objects) and the intermediate values (SumCount objects)

65

## Aggregate action: Example 1

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

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

Data types:  
- a is a SumCount object  
- e is an Integer

The lambda function returns an updated version of a (SumCount object)

66

### Aggregate action: Example 1

```

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

//Define the code that is used to implement the call method of the
Function2<SumCount, SumCount, SumCount> interface.
It is used to combine the partial results (SumCount objects) emitted by
partitions
(a1, a2) -> {
    a1.sum = a1.sum + a2.sum;
    a1.numElements = a1.numElements + a2.numElements;
    return a1;
};
    
```

67

### Aggregate action: Example 1

```

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

//Compute sum and number over the elements of input RDD Aggr
Data types:
- a1 is a SumCount object
- a2 is a SumCount object
The lambda function returns an updated version of a1 (SumCount object)
(a1, a2) -> {
    a1.sum = a1.sum + a2.sum;
    a1.numElements = a1.numElements + a2.numElements;
    return a1;
};
    
```

68

### Aggregate action: Example 1

```

// Compute the average value
double avg = result.avg();

// Print the average on the standard output of the driver
System.out.println(avg);
    
```

69

### Aggregate action: Simulation

- inputRDDAggr = {1, 2, 3, 3}
- Suppose inputRDDAggr is split in the following two partitions
  - {1, 2} and {3, 3}

70

### Aggregate action: Simulation

|  |  |
|--|--|
| Partition #1<br>{1, 2} zeroValue=(0,0) | Partition #2<br>{3, 3} zeroValue=(0,0) |
|--|--|

71

### Aggregate action: Simulation

|   |  |
|---|--|
| Partition #1<br>{1, 2} zeroValue=(0,0) <div style="margin-top: 20px;"> </div> | Partition #2<br>{3, 3} zeroValue=(0,0) |
|---|--|

72

### Aggregate action: Simulation

Partition #1

{1, 2} zeroValue=(0,0)

```

graph TD
    A["{1, 2} zeroValue=(0,0)"] --> B["(2,1)"]
    A --> C["(3,2)"]
    
```

Partition #2

{3, 3} zeroValue=(0,0)

73

### Aggregate action: Simulation

Partition #1

{1, 2} zeroValue=(0,0)

```

graph TD
    A["{1, 2} zeroValue=(0,0)"] --> B["(2,1)"]
    A --> C["(3,2)"]
    
```

Partition #2

{3, 3} zeroValue=(0,0)

```

graph TD
    D["{3, 3} zeroValue=(0,0)"] --> E["(3,1)"]
    D --> F["(6,2)"]
    
```

74

### Aggregate action: Simulation

Partition #1

{1, 2} zeroValue=(0,0)

```

graph TD
    A["{1, 2} zeroValue=(0,0)"] --> B["(2,1)"]
    A --> C["(3,2)"]
    
```

Partition #2

{3, 3} zeroValue=(0,0)

```

graph TD
    D["{3, 3} zeroValue=(0,0)"] --> E["(3,1)"]
    D --> F["(6,2)"]
    
```

result=(9,4)

75

## 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}

77

### Basic actions: Summary

| Action   | Purpose  | Example                 | Result                         |
|--|--|-------------------------|--------------------------------|
| java.util.List<T> collect()                    | Return a (Java) 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}                      |
| long count()                                   | Return the number of elements of the RDD   | inputRDD.count()        | 4                              |
| java.util.Map<T,java.lang.Long> countByValue() | Return a Map object containing the information about the number of times each element occurs in the RDD.   | inputRDD.countByValue() | {{1, 1},<br>{2, 1},<br>{3, 2}} |

78

## Basic actions: Summary

| Action                        | Purpose   | Example          | Result |
|-------------------------------|---|------------------|--------|
| java.util.List<T> take(int n) | Return a (Java) 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}  |
| T first()                     | Return the first element of the RDD   | first()          | {1}    |
| java.util.List<T> top(int n)  | Return a (Java) 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}  |

79

## Basic actions: Summary

| Action  | Purpose  | Example                                   | Result            |
|---|--|---|-------------------|
| java.util.List<T> takeSample(boolean withReplacement, int n, [long seed]) | 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.   | inputRDD.takeSample(false, 1)             | Non deterministic |
| T reduce(Function2<T, T, T> f)  | 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. | The passed "function" is the sum function | 9                 |

80

## Basic actions: Summary

| Action  | Purpose  | Example  | Result |
|---|--|--|--------|
| T fold(T zeroValue, Function2<T,T,T> f)                                       | Same as reduce but with the provided zero value.         | The passed "function" is the sum function and the passed zeroValue is 0  | 9      |
| <U> U aggregate(U zeroValue, Function2<U,T,U> seqOp, Function2<U,U,U> combOp) | Similar to reduce() but used to return a different type. | 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) |

81