Big data: architectures and data analytics

RDD-based programming

RDDs and key-value pairs

RDDs of key-value pairs

- Spark supports also RDDs of key-value pairs
  - They are called pair RDDs
- Pair RDDs are characterized by specific operations
  - reduceByKey(), join, etc.
- Obviously, pair RDDs are characterized also by the operations available for the "standard" RDDs
  - filter(), map(), reduce(), etc.

4

RDDs of key-value pairs

- Many applications are based on pair RDDs
- Pair RDDs allow
  - "grouping" data by key
  - performing computation by key (i.e., group)
- The basic idea is similar to the one of the MapReduce-based programs
  - But there are more operations already available

5

Creating Pair RDDs

Creating Pair RDDs

- Pair RDDs can be built
  - From "regular" RDDs by applying the mapToPair() transformation on the "regular" RDDs
  - From other pair RDDs by applying specific transformations
  - From a Java in-memory collection by using the parallelizePairs() method of the SparkContext class

7

Creating Pair RDDs

- Pairs (i.e., objects of pair RDDs) are represented as tuples composed of two elements
  - Key
  - Value
- Java does not have a built-in tuple data type
- Hence, Java exploits the **scala.Tuple2<K, V>** class to represent tuples

8

Creating Pair RDDs

- **new Tuple2(key, value)** can be used to instance a new object of type Tuple2 in Java
- The (two) elements of a Tuple2 object can be retrieved by using the methods
  - ._1()
    - Retrieves the value of the first element of the tuple
      - The key of the pair in our context
  - ._2()
    - Retrieves the value of the second element of the tuple
      - The value of the pair in our context

9

MapToPair transformation

mapToPair transformation

- Goal
  - The mapToPair transformation is used to create a new PairRDD by applying a function on each element of the "regular" input RDD
  - The new RDD contains one tuple **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**)

11

mapToPair transformation

- Method
  - The mapToPair transformation is based on the **JavaPairRDD<K,V> mapToPair(PairFunction<T,K,V> function)** method of the **JavaRDD<T>** class
  - An object of a class implementing the PairFunction<T,K,V> interface is passed to the mapToPair method
    - The **public Tuple2<K,V> call(T element)** method of the PairFunction<T,K,V> interface must be implemented
      - It contains the code that is applied on each element of the "input" RDD to create the tuples of the returned PairRDD

12

MapToPair transformation: Example

- Create an RDD from a textual file containing the first names of a list of users
  - Each line of the file contains one first name
- Create a PairRDD containing a list of pairs (first name, 1)

13

MapToPair transformation: Example

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

// Create the JavaPairRDD
JavaPairRDD<String, Integer> nameOneRDD =
      namesRDD.mapToPair(name ->
                                  new Tuple2<String, Integer>(name, 1));
```

14

MapToPair transformation: Example

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

// Create the JavaPairRDD
JavaPairRDD<String, Integer> nameOneRDD =
        namesRDD.mapToPair(name ->
                                new Tuple2<String, Integer>(name, 1));
```

The created PairRDD contains pairs (i.e., tuples) of type (String, Integer)

15

parallelizePairs method

parallelizePairs method

- Goal
  - The parallelizePairs method is used to create a new PairRDD from a local Java in-memory collection
- Method
  - The parallelizePairs method is based on the **JavaPairRDD<K,V> parallelizePairs(java.util.List<scala.Tuple2<K,V>> list)** method of the **SparkContext** class
  - Each element (tuple) of the local collection becomes a pair of the returned JavaPairRDD

17

parallelizePairs method: Example

- Create a JavaPairRDD from a local Java list containing the pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 35)

18

parallelizePairs method: Example

```
// Create the local Java collection
ArrayList<Tuple2<String, Integer>> nameAge =
                                new ArrayList<Tuple2<String, Integer>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Paolo", 40);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

19

parallelizePairs method: Example

```
// Create the local Java collection
ArrayList<Tuple2<String, Integer>> nameAge =
                                new ArrayList<Tuple2<String, Integer>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

Create a local in-memory Java list of key-value pairs/tuples. key is a String and value and Integer

20

parallelizePairs method: Example

```
// Create the local Java collection
ArrayList<Tup                                          eger>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Paolo", 40);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

Load pairs/tuples in the in-memory Java list

parallelizePairs method: Example

```
// Create the local Java collection
ArrayList<Tuple2<String, Integer>> nameAge =
                                new ArrayList<Tuple2<String, Integer>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Paolo", 40);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(
localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

Create a JavaPairRDD based on the content of the local in-memory Java list

Transformations on Pair RDDs

Transformations on Pair RDDs

- All the "standard" transformations can be applied
  - Where the specified "functions" operate on tuples
- Specific transformations are available
  - E.g., reduceByKey(), groupyKey(), mapValues(), join(), …

Syntax

- In the following, the following syntax is used
  - <K,V> = Type of the tuples of the PairRDD on which the transformation is applied
    - K = data type of the key
    - V = data type of the value
  - The PairRDD on which the action is applied in referred as "input" PairRDD

25

ReduceByKey transformation

ReduceByKey transformation

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

27

ReduceByKey transformation

- Method
  - The reduceByKey transformation is based on the **JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> f)** method of the **JavaPairRDD<K,V>** class
  - An object of a class implementing the Function2<V, V, V> interface is passed to the reduceByKey method
    - The **public V call(V element1, V element2)** method of the Function2<V, V, V> interface must be implemented
      - It contains the code that is applied to combine the values of the pairs of the input PairRDD and return a new value (same data type)

28

ReduceByKey transformation

- The reduceByKey() transformation is similar to the reduce() action
- However, reduceByKey() is executed on PairRDDs and returns a set of key-value pairs, whereas reduce() returns one single value
- And reduceByKey() **is a transformation** whereas reduce() is an action
  - It is executed lazily and is not stored in the local variables of the driver

29

ReduceByKey transformation: Example

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

30

ReduceByKey transformation: Example

```
// Create the local Java collection
ArrayList<Tuple2<String, Integer>> nameAge =
                        new ArrayList<Tuple2<String, Integer>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Paolo", 40);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

31

ReduceByKey transformation: Example

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

32

ReduceByKey transformation: Example

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

The returned JavaPair contains one pair for each distinct input key

33

FoldByKey transformation

FoldByKey transformation

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

35

FoldByKey transformation

- Method
  - The foldByKey transformation is based on the **JavaPairRDD<K,V> foldByKey(V zeroValue, Function2<V,V,V> f)** method of the **JavaPairRDD<K,V>** class
  - An object of a class implementing the Function2<V, V, V> interface is passed to the foldByKey method
    - The **public V call(V element1, V element2)** method of the Function2<V, V, V> interface must be implemented
      - It contains the code that is applied to combine the values of the pairs of the input PairRDD

36

CombineByKey transformation

CombineByKey transformation

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

38

CombineByKey transformation

- Method
  - The combineByKey transformation is based on the **JavaPairRDD<K,U> combineByKey(Function<V,U> createCombiner, Function2<U,V,U> mergeValue, Function2<U,U,U> mergeCombiner)** method of the **JavaPairRDD<K,V>** class
    - The values of the input PairRDD are of type **V**
    - The values of the returned PairRDD are of type **U**
    - The type of the keys is **K** for both PairRDDs

39

CombineByKey transformation

- The **public U call(V inputElement)** method of the **Function<V,U>** interface must be implemented
  - It contains the code that is used to transform a single value of the input PairRDD (type V) into a value of the data type of the output PairRDD (type U)
  - It is used by a partition containing one single value to return a value of type U

40

CombineByKey transformation

- The **public U call(U intermediateElement, V inputElement)** method of the **Function2<U,V,U>** interface must be implemented
  - It contains the code that is used to combine one value of type U with one value of type V
  - It is used in each partition to combine the initial values (type V) with the intermediate ones (type U)

41

CombineByKey transformation

- The **public U call(U intermediateElement1, U intermediateElement 2)** method of the **Function2<U,U,U>** interface must be implemented
  - It contains the code that is used to combine two values of type U
  - It is used to combine intermediate values

42

CombineByKey transformation

- combineByKey is more general than reduceByKey and foldByKey because the data type of the input and the new pairRDD can be different
  - For this reason, more functions/interfaces must be implemented in this case

43

CombineByKey transformation: Example

- Create a JavaPairRDD from a local Java list containing the pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 35)
  - The key is the first name of a user and the value is his/her age
- Create an output file containing one line for each name followed by the average age of the users with that name

44

CombineByKey transformation: Example

```
// This class is used to store a total sum of values and the number of
// summed values. It is used to compute the average
public class AvgCount implements Serializable {
    public int total;
    public int numValues;

    public AvgCount(int tot, int num) {
        total=tot;
        numValues=num;
    }

    public double average() {
        return (double)total/(double)numValues;
    }

    public String toString() {
        return new String(""+this.average());
    }
}
```

45

CombineByKey transformation: Example

```
// Create the local Java collection
ArrayList<Tuple2<String, Integer>> nameAge =
                                new ArrayList<Tuple2<String, Integer>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Paolo", 40);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

46

CombineByKey transformation: Example

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

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

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

47

CombineByKey transformation: Example

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

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

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

Given an Integer, it returns an AvgCount object

48

CombineByKey transformation: Example

```
JavaPairRDD<String, AvgCount>
        inputElement -> new A
```

Given an Integer and an AvgCount object, it combines them and returns an AvgCount object

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

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

49

CombineByKey transformation: Example

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

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

Given two AvgCount objects, it combines them and returns an AvgCount object

```
        },

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

50

CombineByKey transformation: Example

```
avgAgePerNamePairRDD.saveAsTextFile(outputPath);
```


GroupByKey transformation

GroupByKey transformation

- Goal
  - Create a new PairRDD where there is one pair for each distinct key **k** of the input PairRDD. The value associated with the key **k** in the new PairRDD is the list of values associated with **k** in the input PairRDD
- Method
  - The groupByKey transformation is based on the **JavaPairRDD<K,Iterable<V>> groupByKey()** method of the **JavaPairRDD<K,V>** class

53

GroupByKey transformation: Example

- Create a JavaPairRDD from a local Java list containing the pairs
  - ("Paolo", 40)
  - ("Giorgio", 22)
  - ("Paolo", 35)
  - The key is the first name of a user and the value is his/her age
- Create an output file containing one line for each name followed by the ages of all the users with that name

54

GroupByKey transformation: Example

```
// Create the local Java collection
ArrayList<Tuple2<String, Integer>> nameAge =
                                new ArrayList<Tuple2<String, Integer>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Paolo", 40);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

55

GroupByKey transformation: Example

```
// Create one group for each name with the associated ages
JavaPairRDD<String, Iterable<Integer>> agesPerNamePairRDD =
                                            nameAgeRDD.groupByKey();

// Store the result in a file
agesPerNamePairRDD.saveAsTextFile(outputPath);
```

56

GroupByKey transformation: Example

```
// Create one group for each name with the associated ages
JavaPairRDD<String, Iterable<Integer>> agesPerNamePairRDD =
                                       nameAgeRDD.groupByKey();
```

In the new PairRDD each pair/tuple is composed of
- a string (key of the pair)
- a list of integers (the value of the pair)

ath);

57

MapValues transformation

MapValues transformation

- Goal
  - Apply a user-defined function over the value of each pair of an input PairRDD and return a new PairRDD.
  - One pair is created in the returned PairRDD for each input pair
    - The key of the created pair is equal to the key of the input pair
    - The value of the created pair is obtained by applying the user-defined function on the value of the input pair
  - The data type of the values of the new PairRDD can be different from the data type of the values of the "input" PairRDD
  - The data type of the key is the same

59

MapValues transformation

- Method
  - The mapValues transformation is based on the **JavaPairRDD<K,U> mapValues(Function<V, U> f)** method of the **JavaPairRDD<K,V>** class
  - An object of a class implementing the Function<V, U> interface is passed to the mapValues method
    - The **public U call(V element)** method of the Function<V, U> interface must be implemented
      - It contains the code that is applied to transform the input value into the new value of the new PairRDD

60

MapValues transformation: Example

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

61

MapValues transformation: Example

```
// Create the local Java collection
ArrayList<Tuple2<String, Integer>> nameAge =
                                new ArrayList<Tuple2<String, Integer>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Paolo", 40);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

62

MapValues transformation: Example

```
// Increment age of all users
JavaPairRDD<String, Integer> nameAgePlusOneRDD =
      nameAgeRDD.mapValues(age -> new Integer(age+1));

// Save the result on disk
nameAgePlusOneRDD.saveAsTextFile(outputPath);
```

63

FlatMapValues transformation

FlatMapValues transformation

- Goal
  - Apply a user-defined function over the value of each pair of an input PairRDD and return a new PairRDD
  - A list of pairs is created in the returned PairRDD for each input pair
    - The key of the created pairs is equal to the key of the input pair
    - The values of the created pairs are obtained by applying the user-defined function on the value of the input pair
  - The data type of values of the new PairRDD can be different from the data type of the values of the "input" PairRDD
  - The data type of the key is the same

65

FlatMapValues transformation

- Method
  - The flatMapValues transformation is based on the **JavaPairRDD<K,U> flatMapValues(Function<V, Iterable<U>> f)** method of the **JavaPairRDD<K,V>** class
  - An object of a class implementing the Function<V, Iterable<U>> interface is passed to the flatMapValues method
    - The **public Iterable<U> call(V element)** method of the Function<V, Iterable<U>> interface must be implemented
      - It contains the code that is applied to transform the input value into the set of new values of the new PairRDD

66

Keys transformation

Keys transformation

- Goal
  - Return the list of keys of the input PairRDD
    - The returned RDD is not a PairRDD
    - Duplicates keys are not removed
- Method
  - The keys transformation is based on the **JavaRDD<K> keys()** method of the **JavaPairRDD<K,V>** class

68

Values transformation

Values transformation

- Goal
  - Return the list of values of the input PairRDD
    - The returned RDD is not a PairRDD
    - Duplicates values are not removed
- Method
  - The values transformation is based on the **JavaRDD<V> values()** method of the **JavaPairRDD<K,V>** class

70

SortByKey transformation

SortByKey transformation

- Goal
  - Return a new PairRDD obtained by sorting, in ascending order, the pairs of the input PairRDD by key
    - Note that the data type of the keys (i.e., K) must be a class implementing the Ordered class
  - The data type of the input PairRDD and the new PairRDD is the same

72

SortByKey transformation

- Method
  - The sortByKey transformation is based on the **JavaPairRDD<K,V> sortByKey()** method of the **JavaPairRDD<K,V>** class
  - The **JavaPairRDD<K,V> sortByKey(boolean ascending)** method of the **JavaPairRDD<K,V>** class is also available
    - This method allows specifying if the sort order is ascending or descending

73

SortByKey transformation: Example

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

74

SortByKey transformation: Example

```
// Create the local Java collection
ArrayList<Tuple2<String, Integer>> nameAge =
                                new ArrayList<Tuple2<String, Integer>>();

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Paolo", 40);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Giorgio", 22);
nameAge.add(localPair);

localPair = new Tuple2<String, Integer>("Paolo", 35);
nameAge.add(localPair);

// Create the JavaPairRDD from the local collection
JavaPairRDD<String, Integer> nameAgeRDD = sc.parallelizePairs(nameAge);
```

75

SortByKey transformation: Example

```
// Sort by name
JavaPairRDD<String, Integer> sortedNameAgeRDD =
                                nameAgeRDD.sortByKey();

// Save the result on disk
sortedNameAgeRDD.saveAsTextFile(outputPath);
```

76

Transformations on Pair RDDs: Summary

Transformations on Pair RDDs: Summary

- All the examples reported in the following tables are applied on a PairRDD containing the following tuples (pairs)
  - {("k1", 2), ("k3", 4), ("k3", 6)}
    - The key of each tuple is a String
    - The value of each tuple is an Integer

78

Transformations on Pair RDDs: Summary

Transformation	Purpose	Example of applied function	Result
JavaPairRDD<K,V> reduceByKey(Function2<V,V,V>)	Return a PairRDD<K,V> containing one pair for each key of the "input" PairRDD. The value of each pair of the new PairRDD is obtained by combining the values of the input PairRDD with the same key. The "input" PairRDD and the new PairRDD have the same data type.	sum per key	{("k1", 2), ("k3", 10)}
JavaPairRDD<K,V> foldByKey(V, Function2<V,V,V>)	Similar to the reduceByKey() transformation. However, foldByKey() is characterized also by a zero value	sum per key with zero value = 0	{("k1", 2), ("k3", 10)}

79

Transformations on Pair RDDs: Summary

Transformation	Purpose	Example of applied function	Result
JavaPairRDD<K,U> combineByKey(Function<V,U>, Function2<U,V,U>, Function2<U,U,U>)	Return a PairRDD<K,U> containing one pair for each key of the "input" PairRDD. The value of each pair of the new PairRDD is obtained by combining the values of the input PairRDD with the same key. The "input" PairRDD and the new PairRDD can be different.	average value per key	{("k1", 2), ("k3", 5)}

80

Transformations on Pair RDDs: Summary

Transformation	Purpose	Example of applied function	Result
JavaPairRDD<K,Iterable<V>> groupByKey()	Return a PairRDD<K,Iterable<V> containing one pair for each key of the "input" PairRDD. The value of each pair of the new PairRDD is a "list" containing the values of the input PairRDD with the same key.	-	{("k1", [2]), ("k3", [4, 6])}

81

Transformations on Pair RDDs: Summary

Transformation	Purpose	Example of applied function	Result
JavaPairRDD<K,U> mapValues(Function<V,U>)	Apply a function over each pair of a PairRDD and return a new PairRDD. The applied function returns one pair for each pair of the "input" PairRDD. The function is applied only to the value without changing the key. The "input" PairRDD and the new PairRDD can have a different data type.	v -> v+1 (i.e., for each input pair (k,v), the pair (k,v+1) is included in the new PairRDD)	{("k1", 3), ("k3", 5), ("k3", 7)}

82

Transformations on Pair RDDs: Summary

Transformation	Purpose	Example of applied function	Result
JavaPairRDD<K,U> flatMapValues(Function<V, Iterable<U>>)	Apply a function over each pair in the input PairRDD and return a new RDD of the result. The applied function returns a set of pairs (from 0 to many) for each pair of the "input" RDD. The function is applied only to the value without changing the key. The "input" RDD and the new RDD can have a different data type.	v -> v.to(5) (i.e., for each input pair (k,v), the set of pairs (k,u) with values of u from v to 5 are returned and included in the new PairRDD)	{("k1", 2), ("k1", 3), ("k1", 4), ("k1", 5), ("k3", 4), ("k3", 5)}

83

Transformations on Pair RDDs: Summary

Transformation	Purpose	Example of applied function	Result
JavaRDD<K> keys()	Return an RDD containing the keys of the input pairRDD	-	{"k1", "k3", "k3"}
JavaRDD<V> values()	Return an RDD containing the values of the input pairRDD	-	{2, 4, 6}
JavaPairRDD<K,V> sortByKey()	Return a PairRDD sorted by key. The "input" PairRDD and the new PairRDD have the same data type.	-	{("k1", 2), ("k3", 3), ("k3", 6)}

84