RDDs and key-value pairs

Pair RDDs

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.

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

Creating Pair RDDs

Creating Pair RDDs

- Pair RDDs can be built
  - From "regular" RDDs by applying the mapToPair() or the flatMapToPair() transformation on "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

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

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

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**)

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

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)

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));
```

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)

flatMapToPair transformation

flatMapToPair transformation

- Goal
  - The flatMapToPair transformation is used to create a new PairRDD by applying a function **f** on each element of the "input" RDD
  - The new PairRDD contains a list of pairs 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 pairs **[y]**
    - **[y]**= **f**(**x**)
    - **[y]** can be the empty list

flatMapToPair transformation

- Method
  - The flatMapToPair transformation is based on the **JavaPairRDD<K,V> flatMapToPair(PairFlatMapFunction<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 Iterator<Tuple2<K, V>> call(T element)** method of the PairFlatMapFunction<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

flatMapToPair transformation: Example

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

flatMapToPair transformation: Example

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

// Create the JavaPairRDD based on the input document
// One pair (word,1) for each input word
JavaPairRDD<String, Integer> wordOneRDD =
	linesRDD.flatMapToPair(line -> {
		List<Tuple2<String, Integer>> pairs = new ArrayList<>();
		String[] words = line.split(" ");
		for (String word : words) {
			pairs.add(new Tuple2<String, Integer>(word, 1));
		}
		return pairs.iterator();
	});
```

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

parallelizePairs method: Example

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

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);
```

parallelizePairs method: Example

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

Tuple2<String, Integer> localPair;
lo
n

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

parallelizePairs method: Example

```
// Create the local Java collection
ArrayList<Tu
```

Load pairs/tuples in the in-memory Java list

```
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);
```

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

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

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

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 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

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)

ReduceByKey transformation

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

ReduceByKey transformation

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

ReduceByKey transformation: Example

- Create 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

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);
```

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;
                }
        );
```

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

FoldByKey transformation

FoldByKey transformation

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

FoldByKey transformation

- Method
  - The foldByKey transformation is based on the **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

FoldByKey transformation

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

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**
      - 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

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

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 to transform the first value of each key in each partition to a value of type U

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) of each key with the intermediate ones (type U) of each key

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 of each key returned by the analysis of different partitions

CombineByKey transformation

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

CombineByKey transformation

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

CombineByKey transformation: Example

- Create 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

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());
    }
}
```

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);
```

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;
        }
);
```

CombineByKey transformation: Example

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

Given an Integer, it returns an AvgCount object

```
                                                    ent, 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;
        }
);
```

CombineByKey transformation: Example

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

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

        (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;
        }
);
```

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;
        }
);
```

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 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

GroupByKey transformation

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

GroupByKey transformation

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

GroupByKey transformation: Example

- Create 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

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);
```

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);
```

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);

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

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

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

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);
```

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);
```

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

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

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

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

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 new PairRDD is the same of the input PairRDD

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

SortByKey transformation

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

SortByKey transformation: Example

- Create 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

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);
```

SortByKey transformation: Example

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

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

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

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)}

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)}

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])}

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)}

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)}

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)}

Transformations on two Pair RDDs

Transformations on pairs of Pair RDDs

- Spark supports also some transformations on two PairRDDs
  - SubtractByKey, join, coGroup, etc.

SubtractByKey transformation

SubtractByKey transformation

- Goal
  - Create a new PairRDD containing only the pairs of the input PairRDD associated with a key that is not appearing as key in the pairs of the other PairRDD
    - The data type of the new PairRDD is the same of the "input" PairRDD
    - The input PairRDD and the other PairRDD must have the same type of keys
      - The data type of the values can be different

SubtractByKey transformation

- Method
  - The subtractByKey transformation is based on the **JavaPairRDD<K,V> subtractByKey(JavaPairRDD<K,U> other)** method of the **JavaPairRDD<K,V>** class

SubtractByKey transformation

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

SubtractByKey transformation: Example

- Create two JavaPairRDDs from two local Java lists
  - First list – Profiles of the users of a blog (username, age)
    - {("PaoloG", 40), ("Giorgio", 22), ("PaoloB", 35)}
  - Second list – Banned users (username, motivation)
    - {("PaoloB", "spam"), ("Giorgio", "Vandalism")}
- Create a new PairRDD containing only the profiles of the non-banned users

SubtractByKey transformation: Example

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

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

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

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

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

SubtractByKey transformation: Example

```
// Create the second local Java collection
ArrayList<Tuple2<String, String>> banned =
                                    new ArrayList<Tuple2<String, String>>();
Tuple2<String, String> localPair2;
localPair2 = new Tuple2<String, String>("PaoloB", "spam");
banned.add(localPair2);

localPair2 = new Tuple2<String, String> ("Giorgio", "Vandalism");
banned.add(localPair2);
// Create the JavaPairRDD from the local collection
JavaPairRDD<String, String> bannedPairRDD = sc.parallelizePairs(banned);

// Select the profiles of the "good" users
JavaPairRDD<String, Integer> selectedUsersPairRDD =
                                    profilesPairRDD.subtractByKey(bannedPairRDD);
```

Join transformation

Join transformation

- Goal
  - Join the key-value pairs of two PairRDDs based on the value of the key of the pairs
    - Each pair of the input PairRDD is combined with all the pairs of the other PairRDD with the same key
    - The new PairRDD
      - Has the same key data type of the "input" PairRDDs
      - Has a tuple as value (the pair of values of the two joined input pairs)
    - The input PairRDD and the other PairRDD
      - Must have the same type of keys
      - But the data types of the values can be different

Join transformation

- Method
  - The join transformation is based on the **JavaPairRDD <K, Tuple2<V,U>> join(JavaPairRDD<K,U>)** method of the **JavaPairRDD<K,V>** class

Join transformation

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

Join transformation: Example

- Create two JavaPairRDD from two local Java lists
  - First list – List of questions (QuestionId, Text of the question)
    - {(1, "What is .. ?"), (2, "Who is ..?")}
  - Second list – List of answers (QuestionId, Text of the answer)
    - {(1, "It is a car"), (1, "It is a byke"), (2, "She is Jenny")}
- Create a new PairRDD associating each question with its answers
  - One pair for each possible pair question - answer

Join transformation: Example

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

Tuple2<Integer, String> localPair;
localPair = new Tuple2<Integer, String>(1, "What is .. ?");
questions.add(localPair);


localPair = new Tuple2<Integer, String> (2, "Who is ..?");
questions.add(localPair);

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

Join transformation: Example

```
// Create the second local Java collection
ArrayList<Tuple2<Integer, String>> answers =
                                    new ArrayList<Tuple2<Integer, String>>();
Tuple2<Integer, String> localPair2;
localPair2 = new Tuple2<Integer, String>(1, "It is a car");
answers.add(localPair2);

localPair2 = new Tuple2<Integer, String>(1, "It is a byke");
answers.add(localPair2);

localPair2 = new Tuple2<Integer, String>(2, "She is Jenny");
answers.add(localPair2);

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

Join transformation: Example

```
// Join questions with answers
JavaPairRDD<Integer, Tuple2<String, String>> joinPairRDD =
                                        questionsPairRDD.join(answersPairRDD);
```

Join transformation: Example

```
// Join questions with answers
JavaPairRDD<Integer, Tuple2<String, String>> joinPairRDD =
                                    questionsPairRDD.join(answersPairRDD);
```

Note that the value part is a Tuple2 element (i.e., also the value is a pair).

CoGroup transformation

Cogroup transformation

- Goal
  - Associated each key **k** of the input PairRDDs with
    - The list of values associated with **k** in the input PairRDD
    - And the list of values associated with **k** in the other PairRDD
  - The new PairRDD
      - Has the same key data type of the "input" PairRDDs
      - Has a tuple as value (the two lists of values of the two input pairs)
    - The input PairRDD and the other PairRDD
      - Must have the same type of keys
      - But the data types of the values can be different

Cogroup transformation

- Method
  - The cogroup transformation is based on the **JavaPairRDD <K, Tuple2<Iterable<V>, Iterable<U>>> cogroup(JavaPairRDD<K,U>)** method of the **JavaPairRDD<K,V>** class

Cogroup transformation

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

Cogroup transformation: Example

- Create two JavaPairRDD from two local Java lists
  - First list – List of liked movies (userId, likedMovies)
    - {(1, "Star Trek"), (1, "Forrest Gump") , (2, "Forrest Gump")}
  - Second list – List of liked directors (userId, likedDirector)
    - {(1, "Woody Allen"), (2, "Quentin Tarantino") , (2, "Alfred Hitchcock")}
- Create a new PairRDD containing one pair for each userId (key) associated with
  - The list of liked movies
  - The list of liked directors

Cogroup transformation: Example

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

Cogroup transformation: Example

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

Tuple2<Integer, String> localPair;
localPair = new Tuple2<Integer, String>(1, "Star Trek");
movies.add(localPair);

localPair = new Tuple2<Integer, String>(1, "Forrest Gump");
movies.add(localPair);

localPair = new Tuple2<Integer, String>(2, "Forrest Gump");
movies.add(localPair);

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

Cogroup transformation: Example

```
// Create the second local Java collection
ArrayList<Tuple2<Integer, String>> directors =
                                    new ArrayList<Tuple2<Integer, String>>();
Tuple2<Integer, String> localPair2;
localPair2 = new Tuple2<Integer, String>(1, "Woody Allen");
directors.add(localPair2);

localPair2 = new Tuple2<Integer, String>(2, "Quentin Tarantino");
directors.add(localPair2);

localPair2 = new Tuple2<Integer, String>(2, "Alfred Hitchcock");
directors.add(localPair2);

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

Cogroup transformation: Example

```
// Cogroup movies and directors per user
JavaPairRDD<Integer, Tuple2<Iterable<String>, Iterable<String>>>
        cogroupPairRDD = moviesPairRDD.cogroup(directorsPairRDD);
```

Cogroup transformation: Example

```
// Cogroup movies and directors per user
JavaPairRDD<Integer, Tuple2<Iterable<String>, Iterable<String>>>
        cogroupPairRDD = moviesPairRDD.cogroup(directorsPairRDD);
```

Note that the value part is a Tuple2 element containing two "lists":
-The first part of the Tuple2 element (._1()) contains the "list" of movies liked by the user
- The second part (._2()) contains the "list" of directors liked by the user

Transformations on two PairRDDs: Summary

Transformations on two Pair RDDs: Summary

- All the examples reported in the following tables are applied on the following two PairRDDs
  - inputRDD1 {("k1", 2), ("k3", 4), ("k3", 6)}
  - inputRDD2 {("k3", 9)}

Transformations on two Pair RDDs: Summary

Transformation	Purpose	Example	Result
JavaPairRDD<K,V> subtractByKey(JavaPairRDD<K,U>)	Return a new PairRDD where the pairs associated with a key appearing only in the "input"" PairRDD and not in the one passed as parameter. The values are not considered to take the decision.	inputRDD1. subtract (inputRDD2)	{("K1",2)}
JavaPairRDD <K, Tuple2<V,U>> join(JavaPairRDD<K,U>)	Return a new PairRDD corresponding to join of the two PairRDDs. The join is based in the value of the key.	inputRDD1.join (inputRDD2)	{("k3", (4,9)), ("k3", (6,9))}

Transformations on two Pair RDDs: Summary

Transformation	Purpose	Example	Result
JavaPairRDD <K, Tuple2<Iterable<V>, Iterable<U>>> cogroup(JavaPairRDD<K,U>)	For each key k in one of the two PairRDDs, return a pair (k, tuple), where tuple contains the list of values of the first PairRDD with key k in the first element of the tuple and the list of values of the second PairRDD with key k in the second element of the tuple.	inputRDD1. cogroup (inputRDD2)	{("K1", ([2], [])), ("K3", ([4, 6], [9]))}

Actions on Pair RDDs

Actions on Pair RDDs

- Spark supports also some specific actions on PairRDDs
  - countByKey, collectAsMap, lookup

CountByKey action

CountByKey action

- Goal
  - The countByKey action returns a local Java Map object containing the information about the number of elements associated with each key in the PairRDD
    - i.e., the number of times each key occurs in the PairRDD
  - **Pay attention to the number of distinct keys of the PairRDD**
  - **If the number of distinct keys is large, the result of the action cannot be stored in a local variable of the Driver**

CountByKey action

- Method
  - The countBykey action is based on the **java.util.Map<K, java.lang.Object> countBykey()** method of the **JavaPairRDD<K,V>** class
    - The values of the returned java.util.Map are returned as "generic" java.lang.Object
    - However, they are java.lang.Long objects

CountByKey action: Example 1

- Create a JavaPairRDD from the following Java list
  - {("Forrest Gump", 4), ("Star Trek", 5) , ("Forrest Gump", 3)}
  - Each pair contains a movie and the rating given by someone to the movie
- Compute the number of ratings for each movie

CountByKey action: Example 1

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

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Forrest Gump", 4);
movieRating.add(localPair);

localPair = new Tuple2<String, Integer>("Star Trek", 5);
movieRating.add(localPair);

localPair = new Tuple2<String, Integer>("Forrest Gump", 3);
movieRating.add(localPair);
```

CountByKey action: Example 1

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


// Compute the number of rating for each movie
 java.util.Map<String, java.lang.Object> movieNumRatings =
                                        movieRatingRDD.countByKey();

// Print the result on the standard output
System.out.println(movieNumRatings);
```

CountByKey action: Example 1

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


// Compute the number of rating for each movie
java.util.Map<String, java.lang.Object> movieNumRatings =
                                    movieRatingRDD.countByKey();

// Pr
Syst
```

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

CollectAsMap action

CollectAsMap action

- Goal
  - The collectAsMap action returns a local Java java.util.Map<K,V> containing the same pairs of the considered PairRDD
  - **Pay attention to the size of the PairRDD**
- Method
  - The collectAsMap action is based on the **java.util.Map<K,V> collectAsMap()** method of the **JavaPairRDD<K,V>** class

CollectAsMap action

- **Pay attention** that the **collectAsMap** action **returns a java.util.Map** object
- **A Map cannot contain duplicate keys**
  - Each key can map to at most one value
  - If the "input" PairRDD contains more than one pair with the same key, only one of those pairs is stored in the returned local Java Map
    - Usually, the last one in the PairRDD
- Use collectAsMap only if you are sure that each key appears only once in the PairRDD

CollectAsMap action: Example 1

- Create a JavaPairRDD from the following Java list
  - {("User1", "Paolo"), ("User2", "Luca"), ("User3", "Daniele")}
  - Each pair contains a userId and the name of the user
- Retrieve the pairs of the created PairRDD and store them in a local Java Map that is instantiated in the Driver

CollectAsMap action: Example 1

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

Tuple2<String , String> localPair;
localPair = new Tuple2<String , String>("User1", "Paolo");
users.add(localPair);

localPair = new Tuple2<String , String>("User2", "Luca");
users.add(localPair);

localPair = new Tuple2<String , String>("User3", "Daniele");
users.add(localPair);
```

CollectAsMap action: Example 1

```
// Create the JavaPairRDD from the local collection
JavaPairRDD< String, String> usersRDD =
                                sc.parallelizePairs(users);


// Retrieve the content of usersRDD and store it in a local Java Map
 java.util.Map<String, String> retrievedPairs = usersRDD.collectAsMap();

// Print the result on the standard output
System.out.println(retrievedPairs);
```

CollectAsMap action: Example 1

```
// Create the JavaPairRDD from the local collection
JavaPairRDD< String, String> usersRDD =
                                sc.parallelizePairs(users);


// Retrieve the content of usersRDD and store it in a local Java Map
java.util.Map<String, String> retrievedPairs = usersRDD.collectAsMap();

// Print the
System.out.println(retrievedPairs);
```

Pay attention to the size of the returned map

Lookup action

Lookup action

- Goal
  - The lookup(**k**) action returns a local Java java.util.List<V> containing the values of the pairs of the PairRDD associated with the key **k** specified as parameter
- Method
  - The lookup action is based on the **java.util.List<V> lookup(K key)** method of the **JavaPairRDD<K,V>** class

Lookup action: Example 1

- Create a JavaPairRDD from the following Java list
  - {("Forrest Gump", 4), ("Star Trek", 5) , ("Forrest Gump", 3)}
  - Each pair contains a movie and the rating given by someone to the movie
- Retrieve the ratings associated with the movie "Forrest Gump" and store them in a local Java list in the Driver

Lookup action: Example 1

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

Tuple2<String, Integer> localPair;
localPair = new Tuple2<String, Integer>("Forrest Gump", 4);
movieRating.add(localPair);

localPair = new Tuple2<String, Integer>("Star Trek", 5);
movieRating.add(localPair);

localPair = new Tuple2<String, Integer>("Forrest Gump", 3);
movieRating.add(localPair);
```

Lookup action: Example 1

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


// Select the ratings associated with “Forrest Gump”
 java.util.List<Integer> movieRatings =
                                movieRatingRDD.lookup("Forrest Gump");

// Print the result on the standard output
System.out.println(movieRatings);
```

Lookup action: Example 1

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


// Select the ratings associated with "Forrest Gump"
java.util.List<Integer> movieRatings =
                          movieRatingRDD.lookup("Forrest Gump");

// Pr
Syst
```

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

Actions on PairRDDs: Summary

Actions on PairRDDs: Summary

- All the examples reported in the following tables are applied on the following PairRDD
  - {("k1", 2), ("k3", 4), ("k3", 6)}

Actions on PairRDDs: Summary

Transformation	Purpose	Example	Result
java.util.Map<K,java.lang.Object> countByKey()	Return a local Java java.util.Map containing the number of elements in the input PairRDD for each key of the input PairRDD.	inputRDD.countByKey()	{("K1",1), ("K3",2)}
java.util.Map<K,V> collectAsMap()	Return a local Java java.util.Map containing the pairs of the input PairRDD	inputRDD.collectAsMap()	{("k1", 2), ("k3", 6)} Or {("k1", 2), ("k3", 4)} Depending on the order of the pairs in the PairRDD

Actions on PairRDDs: Summary

Transformation	Purpose	Example	Result
java.util.List<V> lookup(K key)	Return a local Java java.util.List containing all the values associated with the key specified as parameter	inputRDD. lookup("k3")	{4, 6}