Table of contents

Spark supports also RDDs of key-value pairs. Key-value pairs in python are represented by means of python tuples:

- The first value is the key part of the pair
- The second value is the value part of the pair

RDDs of key-value pairs are sometimes called “pair RDDs”.

RDDs of key-value pairs are characterized by

- specific operations (e.g., `reduceByKey()`, `join()`), which analyze the content of one group (key) at a time;
- operations available for the standard RDDs (e.g., `filter()`, `map()`, `reduce()`).

Many applications are based on RDDs of key-value pairs: the operations available for RDDs of key-value pairs allow grouping data by key and performing computation by key (i.e., by group). The basic idea is similar to the one of the MapReduce-based programs in Hadoop, but there are more operations already available.

1 Creating RDDs of key-value pairs

RDDs of key-value pairs can be built

- From other RDDs by applying the `map()` or the `flatMap()` transformation on other RDDs;
- From a Python in-memory collection of tuple (key-value pairs) by using the `parallelize()` method of the `SparkContext` class.

Key-value pairs are represented as standard built-in Python tuples composed of two elements

- Key
- Value

2 RDDs of key-value pairs by using the Map transformation

The goal is to define an RDD of key-value pairs by using the map transformation: apply a function `f` on each element of the input RDD that returns one tuple for each input element. The new RDD of key-value pairs contains one tuple `y` for each element `x` of the input RDD.

The standard `map(f)` transformation is used, and the new RDD of key-value pairs contains one tuple `y` for each input element `x` of the input RDD ($y = f(x)$).

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 an RDD of key-value pairs containing a list of pairs `(first name, 1)`.

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

# Create an RDD of key-value pairs
nameOnePairRDD = namesRDD.map(lambda name: (name, 1))
```

`nameOnePairRDD`	It contains key-value pairs (i.e., tuples) of type (string, integer)

3 RDDs of key-value pairs by using the flatMap transformation

Define an RDD of key-value pairs by using the flatMap transformation: apply a function `f` on each element of the input RDD that returns a list of tuples for each input element. The new PairRDD contains all the pairs obtained by applying `f` on each element `x` of the input RDD.

The standard `flatMap(f)` transformation is used, and the new RDD of key-value pairs contains the tuples returned by the execution of `f` on each element `x` of the input RDD.

$$[y] = f(x)$$

- Given a element x of the input RDD, f applied on x returns a list of pairs $[y]$;
- The new RDD is a list of pairs contains all the pairs of the returned list of pairs. It is not an RDD of lists.

$[y]$ can be the empty list.

Example

1. Create an RDD from a textual file; each line of the file contains a set of words;
2. Create a `PairRDD` containing a list of pairs `(word, 1)`: one pair for each word occurring in the input document (with repetitions).

Version 1

```python
# Define the function associated with the flatMap transformation
def wordsOnes(line):
    pairs = []
    for word in line.split(' '):
        pairs.append( (word, 1))
    return pairs

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

# Create an RDD of key-value pairs based on the input document
# One pair (word,1) for each input word
wordOnePairRDD = linesRDD.flatMap(wordsOnes)
```

Version 2

```python
# Read the content of the input textual file
linesRDD = sc.textFile("document.txt")

# Create an RDD of key-value pairs based on the input document
# One pair (word,1) for each input word
wordOnePairRDD = linesRDD.flatMap(
    lambda line: map(lambda w: (w, 1), line.split(' '))
)
```

`map(lambda w: (w, 1), line.split(' '))`	This is the map of python. It is not the Spark's map transformation.

4 RDDs of key-value pairs by using parallelize

Use the parallelize method to create an RDD of key-value pairs from a local python in-memory collection of tuples.

It is based on the standard `parallelize(c)` method of the `SparkContext` class: each element (tuple) of the local python collection becomes a key-vaue pair of the returned RDD.

Example

Create an RDD from a local python list containing the following key-value pairs

- `("Paolo", 40)`
- `("Giorgio", 22)`
- `("Paolo", 35)`

```python
# Create the local python list
nameAge = [
    ("Paolo",40),
    ("Giorgio",22),
    ("Paolo",35)
]

# Create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)
```

`nameAge`	This is a local in-memory python list of key-value pairs (tuples), that is stored in the main memory of the Driver.
`nameAgePairRDD`	This is an RDD or key-value pairs based on the content of the local in-memory python list. The RDD is stored in the "distributed" main memory of the cluster servers

5 Transformations on RDDs of key-value pairs

All the standard transformations can be applied, where the specified functions operate on tuples, but also specific transformations are available (e.g., `reduceByKey()`, `groupyKey()`, `mapValues()`, `join()`).

5.1 ReduceByKey transformation

The goal is to create a new RDD of key-value pairs where there is one pair for each distinct key `k` of the input RDD of key-value pairs:

- The value associated with key `k` in the new RDD of key-value pairs is computed by applying a function `f` on the values associated with `k` in the input RDD of key-value pairs; the function `f` must be associative and commutative, otherwise the result depends on how data are partitioned and analyzed;
- The data type of the new RDD of key-value pairs is the same of the input RDD of key-value pairs.

The reduceByKey transformation is based on the `reduceByKey(f)` method of the `RDD` class. A function `f` is passed to the reduceByKey method

- Given the values of two input pairs, `f` is used to combine them in one single value;

- `f` is recursively invoked over the values of the pairs associated with one key at a time until the input values associated with one key are reduced to one single value.

The retuned RDD contains a number of key-value pairs equal to the number of distinct keys in the input key-value pair RDD.

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, while `reduce()` is executed on an RDD and returns one single value (stored in a local python variable). Moreover, `reduceByKey()` is a transformation, and so it is executed lazily and its result is stored in another RDD, whereas `reduce()` is an action.

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.

Example

1. Create an RDD from a local python list containing the pairs, where the key is the first name of a user and the value is his/her age
   - `("Paolo", 40)`
   - `("Giorgio", 22)`
   - `("Paolo", 35)`
2. Create a new RDD of key-value pairs containing one pair for each name. In the returned RDD, associate each name with the age of the youngest user with that name.

```python
# Create the local python list
nameAge = [
    ("Paolo",40),
    ("Giorgio",22),
    ("Paolo",35)
]

# Create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# Select for each name the lowest age value
youngestPairRDD= nameAgePairRDD.reduceByKey(lambda age1, age2: min(age1, age2))
```

`youngestPairRDD`	The returned RDD of key-value pairs contains one pair for each distinct input key (i.e., for each distinct name in this example)

5.2 FoldByKey transformation

The `foldByKey()` has the same goal of the `reduceBykey()` transformation, however

- It is characterized also by a “zero” value
- Functions must be associative but are not required to be commutative

The foldByKey transformation is based on the `foldByKey(zeroValue, op)` method of the `RDD` class. A function `op` is passed to the fold method:

- Given values of two input pairs, `op` is used to combine them in one single value
- `op` is also used to combine input values with the “zero” value
- `op` is recursively invoked over the values of the pairs associated with one key at a time until the input values are reduced to one single value

The “zero” value is the neutral value for the used function `op` (i.e., “zero” combined with any value v by using `op` is equal to v).

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.

Example

1. Create an RDD from a local python list containing the pairs, where the key is the first name of a user and the value is a message published by him/her
   - `("Paolo", "Message1")`
   - `("Giorgio", "Message2")`
   - `("Paolo", "Message3")`
2. Create a new RDD of key-value pairs containing one pair for each name. In the returned RDD, associate each name the concatenation of its messages (preserving the order of the messages in the input RDD).

```python
# Create the local python list
nameMess = [
    ("Paolo","Message1"),
    ("Giorgio","Message2"),
    ("Paolo","Message3")
]

# Create the RDD of pairs from the local collection
nameMessPairRDD = sc.parallelize(nameMess)

# Concatenate the messages of each user
concatPairRDD= nameMessPairRDD.foldByKey('', lambda m1, m2: m1+m2)
```

5.3 CombineByKey transformation

The goal is to create a new RDD of key-value pairs where there is one pair for each distinct key k of the input RDD of key-value pairs. The value associated with the key k in the new RDD of key-value pairs is computed by applying user-provided functions on the values associated with k in the input RDD of key-value pairs: the user-provided function must be associative, otherwise the result depends how data are partitioned and analyzed.

The data type of the new RDD of key-value pairs can be different with respect to the data type of the input RDD of key-value pairs.

The combineByKey transformation is based on the `combineByKey(createCombiner, mergeValue, mergeCombiner)` method of the `RDD` class

- The values of the input RDD of pairs are of type V
- The values of the returned RDD of pairs are of type U
- The type of the keys is K for both RDDs of pairs

The `createCombiner` function contains the code that is used to transform a single value (type V) of the input RDD of key-value pairs into a value of the data type (type U) of the output RDD of key-value pairs. It is used to transform the first value of each key in each partition to a value of type U.

The `mergeValue` function 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.

The `mergeCombiner` function 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.

The `combineByKey` function 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 must be implemented in this case.

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.

Example

1. Create an RDD from a local python list containing the the following pairs: the key is the first name of a user and the value is his/her age
   - `("Paolo", 40)`
   - `("Giorgio", 22)`
   - `("Paolo", 35)`
2. Store the results in an output HDFS folder. The output contains one line for each name followed by the average age of the users with that name

```python
# Create the local python list
nameAge = [
    ("Paolo",40),
    ("Giorgio",22),
    ("Paolo",35)
]

# Create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# Compute the sum of ages and
# the number of input pairs for each name (key)
sumNumPerNamePairRDD = nameAgePairRDD.combineByKey(
    lambda inputElem: (inputElem, 1),
    lambda intermediateElem, inputElem: (
        intermediateElem[0]+inputElem,
        intermediateElem[1]+1
    ),
    lambda intermediateElem1, intermediateElem2: (
        intermediateElem1[0]+intermediateElem2[0],
        intermediateElem1[1]+intermediateElem2[1]
    )
)

# Compute the average for each name
avgPerNamePairRDD = sumNumPerNamePairRDD.map(
    lambda pair: (pair[0], pair[1][0]/pair[1][1])
)

# Store the result in an output folder
avgPerNamePairRDD.saveAsTextFile(outputPath)
```

`lambda inputElem:`	Given an input value (an age), it returns a tuple containing (age,1)
`lambda intermediateElem, inputElem:`	Given an input value (an age) and an intermediate value (*sum ages*, *num represented values*), it combines them and returns a new updated tuple (*sum ages*, *num represented values*)
`lambda intermediateElem1, intermediateElem2:`	Given two intermediate result tuples (*sum ages*, *num represented values*), it combines them and returns a new updated tuple (*sum ages*, *num represented values*)
`lambda pair:`	Compute the average age for each key (i.e., for each name) by combining *sum ages* and *num represented values*. Each input pair is characterized by a value that is a tuple containing (*sum ages*, *num represented values*).

5.4 GroupByKey transformation

The goal is to create a new RDD of key-value pairs where there is one pair for each distinct key k of the input RDD of key-value pairs: the value associated with key k in the new RDD of key-value pairs is the list of values associated with k in the input RDD of key-value pairs.

The groupByKey transformation is based on the `groupByKey()` method of the `RDD` class.

If 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 an aggregation or compute a function that is not associative must be applied.

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.

Example

1. Create an RDD from a local python list containing the following pairs: the key is the first name of a user and the value is his/her age.
   - `("Paolo", 40)`
   - `("Giorgio", 22)`
   - `("Paolo", 35)`
2. Store the results in an output HDFS folder. The output contains one line for each name followed by the ages of all the users with that name.

```
# Create the local python list
nameAge = [
    ("Paolo",40),
    ("Giorgio",22),
    ("Paolo",35)
]

# Create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# Create one group for each name with the list of associated ages
agesPerNamePairRDD = nameAgePairRDD.groupByKey()

# Store the result in an output folder
agesPerNamePairRDD\
    .mapValues(lambda listValues: list(listValues))\
    .saveAsTextFile(outputPath)
```

`agesPerNamePairRDD`	In this RDD of key-value pairs each tuple is composed of a string (key of the pair) and a collection of integers (the value of the pair - a `ResultIterable` object)
`.mapValues(lambda listValues: list(listValues))`	This part is used to format the content of the value part of each pair before storing the result in the output folder: this transforms a `ResultIterable` object to a Python list. Without this map the output will contain the pointers to `ResultIterable` objects instead of a readable list of integer values

5.5 MapValues transformation

The goal is to apply a function f over the value of each pair of an input RDD or key-value pairs and return a new RDD of key-value pairs: one pair is created in the returned RDD 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 function f on the value of the input pair

The data type of the values of the new RDD of key-value pairs can be different from the data type of the values of the input RDD of key-value pairs. The data type of the key is the same.

The mapValues transformation is based on the `mapValues(f)` method of the `RDD` class: a function `f` is passed to the mapValues method, where `f` contains the code that is applied to transform each input value into the a new value that is stored in the RDD of key-value pairs. The retuned RDD of pairs contains a number of key-value pairs equal to the number of key-value pairs of the input RDD of pairs (the key part is not changed).

Example

1. Create an RDD from a local python list containing the following pairs: the key is the first name of a user and the value is his/her age.
   - `("Paolo", 40)`
   - `("Giorgio", 22)`
   - `("Paolo", 35)`
2. Increase the age of each user (+1 year) and store the result in the HDFS file system, one output line per user.

```python
# Create the local python list
nameAge = [
    ("Paolo",40),
    ("Giorgio",22),
    ("Paolo",35)
]

# Create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# Increment age of all users
plusOnePairRDD = nameAgePairRDD.mapValues(lambda age: age+1)

# Save the result on disk
plusOnePairRDD.saveAsTextFile(outputPath)
```

5.6 FlatMapValues transformation

The goal is to apply a function f over the value of each pair of an input RDD or key-value pairs and return a new RDD of key-value pairs (f returns a list of values for each input value). A list of pairs is inserted in the returned RDD 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 function f on the value of the input pair

The data type of the values of the new RDD of key-value pairs can be different from the data type of the values of the input RDD of key-value pairs. The data type of the key is the same.

The flatMapValues transformation is based on `flatMapValues(f)` method of the `RDD` class: a function `f` is passed to the `mapValues` method, where `f` contains the code that is applied to transform each input value into a set of new values that are stored in the new RDD of key-value pairs. The keys of the input pairs are not changed.

Example

1. Create an RDD from a local python list containing the pairs
    - `("Sentence#1", "Sentence test")`
    - `("Sentence#2", "Sentence test number 2")`
    - `("Sentence#3", "Sentence test number 3")`
2. Select the words of each sentence and store in the HDFS file system one pair (senteceId, word) per line.

```python
# Create the local python list
sentences = [
    ("Sentence#1", "Sentence test"),
    ("Sentence#2", "Sentence test number 2"),
    ("Sentence#3", "Sentence test number 3")
]

# Create the RDD of pairs from the local collection
sentPairRDD = sc.parallelize(sentences)

# "Extract" words from each sentence
sentIdWord = sentPairRDD.flatMapValues(lambda s: s.split(' '))

# Save the result on disk
sentIdWord.saveAsTextFile(outputPath)
```

5.7 Keys transformation

The goal is to return the list of keys of the input RDD of pairs and store them in a new RDD. The returned RDD is not an RDD of key-value pairs, instead it is a standard RDD of single elements, with duplicate keys not removed.

The keys transformation is based on the `keys()` method of the `RDD` class.

Example

1. Create an RDD from a local python list containing the following pairs: the key is the first name of a user and the value is his/her age
   - `("Paolo", 40)`
   - `("Giorgio", 22)`
   - `("Paolo", 35)`
2. Store the names of the input users in an output HDFS folder. The output contains one name per line (duplicate names are removed).

```python
# Create the local python list
nameAge = [
    ("Paolo",40),
    ("Giorgio",22),
    ("Paolo",35)
]

# Create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# Select the key part of the input RDD of key-value pairs
namesRDD = nameAgePairRDD.keys().distinct()

# Store the result in an output folder
namesRDD.saveAsTextFile(outputPath)
```

5.8 Values transformation

The goal is to return the list of values of the input RDD of pairs and store them in a new RDD. The returned RDD is not an RDD of key-value pairs, instead it is a standard RDD of single elements, with duplicate values are not removed.

The values transformation is based on the `values()` method of the `RDD` class.

Example

1. Create an RDD from a local python list containing the pairs: the key is the first name of a user and the value is his/her age
   - `("Paolo", 40)`
   - `("Giorgio", 22)`
   - `("Paolo", 22)`
2. Store the ages of the input users in an output HDFS folder, containing one age per line and duplicate ages/values are not removed.

```python
# Create the local python list
nameAge = [
    ("Paolo",40),
    ("Giorgio",22),
    ("Paolo",22)
]

# Create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# Select the value part of the input RDD of key-value pairs
agesRDD = nameAgePairRDD.values()

# Store the result in an output folder
agesRDD.saveAsTextFile(outputPath)
```

5.9 SortByKey transformation

The goal is to return a new RDD of key-value pairs obtained by sorting, in ascending order, the pairs of the input RDD by key (notice that the final order is related to the default sorting function of the data type of the input keys). The content of the new RDD of key-value pairs is the same of the input RDD but the pairs are sorted by key in the new returned RDD.

The sortByKey transformation is based on the `sortByKey()` method of the `RDD` class (pairs are sorted by key in ascending order). The `sortByKey(ascending)` method of the `RDD` class is also available: this method allows to specify if the sort order is ascending or descending by means of a Boolean parameter (`True` for ascending, `False` for descending).

A shuffle operation is executed for computing the result of the `sortByKey()` transformation, since pairs from different partitions of the input RDD must be compared to sort the input pairs by key.

Note

1. Create an RDD from a local python list containing the pairs: the key is the first name of a user and the value is his/her age
    - `("Paolo", 40)`
    - `("Giorgio", 22)`
    - `("Paolo", 35)`
2. Sort the users by name and store the result in the HDFS file system.

```python
# Create the local python list
nameAge = [
    ("Paolo",40),
    ("Giorgio",22),
    ("Paolo",35)
]

# Create the RDD of pairs from the local collection
nameAgePairRDD = sc.parallelize(nameAge)

# Sort by name the content of the input RDD of key-value pairs
sortedNameAgePairRDD = nameAgePairRDD.sortByKey()

# Store the result in an output folder
sortedNameAgePairRDD.saveAsTextFile(outputPath)
```

5.10 Summary

All the examples reported in the following tables are applied on an RDD of pairs 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

Purposes

Transformation	Purpose
`reduceByKey(f)`	Return an RDD of pairs containing one pair for each key of the input RDD of pairs. The value of each pair of the new RDD of pairs is obtained by combining the values of the input RDD associated with the same key. The input RDD of pairs and the new RDD of pairs have the same data type.
`foldByKey(zeroValue, op)`	Similar to the `reduceByKey()` transformation, however `foldByKey()` is characterized also by a zero value.
`combineByKey( createCombiner, mergeValue, mergeCombiner )`	Return an RDD of key-value pairs containing one pair for each key of the input RDD of pairs. The value of each pair of the new RDD is obtained by combining the values of the input RDD associated with the same key. The values of the input RDD of pairs and the values of the new (returned) RDD of pairs can be characterized by different data types.
`groupByKey()`	Return an RDD of pairs containing one pair for each key of the input RDD of pairs. The value of each pair of the new RDD of pairs is a collection containing all the values of the input RDD associated with one of the input keys.
`mapValues(f)`	Apply a function over each pair of an RDD of pairs and return a new RDD of pairs. The applied function returns one pair for each pair of the input RDD of pairs. The function is applied only on the value part without changing the key. The values of the input RDD and the values of new RDD can have different data types.
`flatMapValues(f)`	Apply a function over each pair of an RDD of pairs and return a new RDD of pairs. The applied function returns a set of pairs (from 0 to many) for each pair of the input RDD of pairs. The function is applied only on the value part without changing the key. The values of the input RDD and the values of new RDD can have different data types.
`keys()`	Return an RDD containing the keys of the input `pairRDD`.
`values()`	Return an RDD containing the values of the input `pairRDD`.
`sortByKey()`	Return a `PairRDD` sorted by key. The input `PairRDD` and the new `PairRDD` have the same data type.

Examples

Transformation	Example	Result
`reduceByKey(f)`	`reduceByKey(lambda v1, v2: v1+v2)` Sum values per key	`[("k1",2),("k3",10)]`
`foldByKey(zeroValue, op)`	`foldByKey(0,lambda v1, v2: v1+v2)` Sum values per key. The zero value is 0	`[("k1",2),("k3",10)]`
`combineByKey( createCombiner, mergeValue, mergeCombiner )`	`combineByKey( lambda e: (e, 1), lambda c, e: (c[0]+e, c[1]+1), lambda c1, c2: (c1[0]+c2[0], c1[1]+c2[1]) )` Sum values by key and count the number of pairs by key in one single step	`[("k1",(2,1)),("k3",(10,2))]`
`groupByKey()`	`groupByKey()`	`[("k1",[2]),("k3",[4,6])]`
`mapValues(f)`	`mapValues(lambda v: v+1)` Increment the value part by 1	`[("k1",3),("k3",5),("k3",7)]`
`flatMapValues(f)`	`flatMapValues(lambda v: list(range(v,6)))`	`[ ("k1",2), ("k1",3), ("k1",4), ("k1",5), ("k3",4), ("k3",5) ]`
`keys()`	`keys()`	`["k1","k3","k3"]`
`values()`	`values()`	`[2, 4, 6]`
`sortByKey()`	`sortByKey()`	`[("k1",2),("k3",4),("k3",6)]`