Key-Value Pairs, Pair RDDs, Two Pair RDDs

Nagiza F. Samatova, samatova@csc.ncsu.edu
Professor, Department of Computer Science
North Carolina State University

Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory

Outline

- **Key/value RDDs and Pair RDDs**
- **Transformations on Pair RDDs**
  - Aggregation: reduceByKey(), foldByKey(), combineByKey()
  - Filtering on values: mapValues(), flatMapValues()
  - Sorting with sortByKey()
- **Actions on Pair RDDs**
  - countByKey(), countByValue()
- **Two Pair RDDs: Transformations**
  - join(), leftOuterJoin(), rightOuterJoin(), subtractByKey()

Working with Key/Value Pairs

- **Key/value RDDs are commonly used to perform *aggregations***
- **Often some initial ETL (Extract, Transform, and Load) gets data into a key/value format**
- **Key/value RDDs expose new operations**
  - counting up reviews for each product
  - grouping together data with the same key
  - grouping together two different RDDs

Pair RDDs

- **Spark provides special operations on RDDs containing key/value pairs.**
- **These RDDs are called pair RDDs.**
- **Pair RDDs have a reduceByKey() method that can aggregate data separately for each key,**
- **join() method that can merge two RDDs together by grouping elements with the same key.**
- **It is common to extract fields from an RDD and use those fields as keys in pair RDD operations.**
  - representing, for instance, an event time, customer ID, or other identifier

Creating Pair RDDs

- **There are a number of ways to get pair RDDs in Spark.**
  - Many formats will directly return pair RDDs for their key/value data.
  - A regular RDD can be turned into a pair RDD via the map() function that returns key/value pairs.
- **Example:**
  - start with an RDD of lines of text and key the data by the first word in each line:

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

- **To create a pair RDD from an in-memory collection, call SparkContext.parallelize() on a collection of pairs.**

```
>>> pairs1 = sc.parallelize([(1,2), (3,4), (3,6)])
>>> pairs1.collect()
[(1, 2), (3, 4), (3, 6)]
```

Pair RDDs are still RDDs

- **Pair RDDs are still RDDs (Python tuples),**
  - and thus support the same functions as RDDs
  - Ex: take pair RDD and filter out lines longer than 20 characters

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

Example: Pair RDDs as "ordinary RDDs"

```
>>> pairs1 = sc.parallelize([(1,2), (3,4), (3,6)])
>>> pairs1.filter(lambda kv: kv[1] > 3 ).collect()
[(3, 4), (3, 6)]
```

Outline

- **Key/value RDDs and Pair RDDs**
- **Transformations on Pair RDDs**
  - Aggregation: reduceByKey(), foldByKey(), combineByKey()
  - Filtering on values: mapValues(), flatMapValues()
  - Sorting with sortByKey()
- **Actions on Pair RDDs**
  - countByKey(), countByValue()
- **Two Pair RDDs: Transformations**
  - join(), leftOuterJoin(), rightOuterJoin(), subtractByKey()

Transformations on Pair RDDs

Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements.

rdd = { (1,2), (3,4), (3,6) }

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.	See Examples 4-12 through 4-14.	

Transformations on Pair RDDs

rdd = { (1,2), (3,4), (3,6) }

Function name	Purpose	Example	Result
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`	`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`

Transformations on Pair RDDs

rdd = { (1,2), (3,4), (3,6) }

Function name	Purpose	Example	Result
`values()`	Return an RDD of just the values.	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`

Examples: Transformations on Pair RDDs

```
>>> pairs1 = sc.parallelize( [(1,2), (3,4), (3,6)] )
>>> a = pairs1.reduceByKey(lambda x,y: x+y).collect()
>>> a
[(1, 2), (3, 10)]
>>> c = pairs1.mapValues(lambda x: x + 1).collect()
>>> c
[(1, 3), (3, 5), (3, 7)]
>>> d = pairs1.keys().collect()
>>> d
[1, 3, 3]
>>> pairs1.values().collect()
[2, 4, 6]
>>> pairs1.sortByKey().collect()
[(1, 2), (3, 4), (3, 6)]
>>> sc.parallelize([(7,2), (3,4), (1,6)] ).sortByKey().collect()
[(1, 6), (3, 4), (7, 2)]
```

Examples: Transformations on Pair RDDs

```
>>> range(4)
[0, 1, 2, 3]
>>> e = pairs1.flatMapValues(lambda x: range(x)
).collect()
>>> e
[(1, 0), (1, 1), (3, 0), (3, 1), (3, 2), (3, 3), (3, 0), (3, 1),
(3, 2), (3, 3), (3, 4), (3, 5)]
```

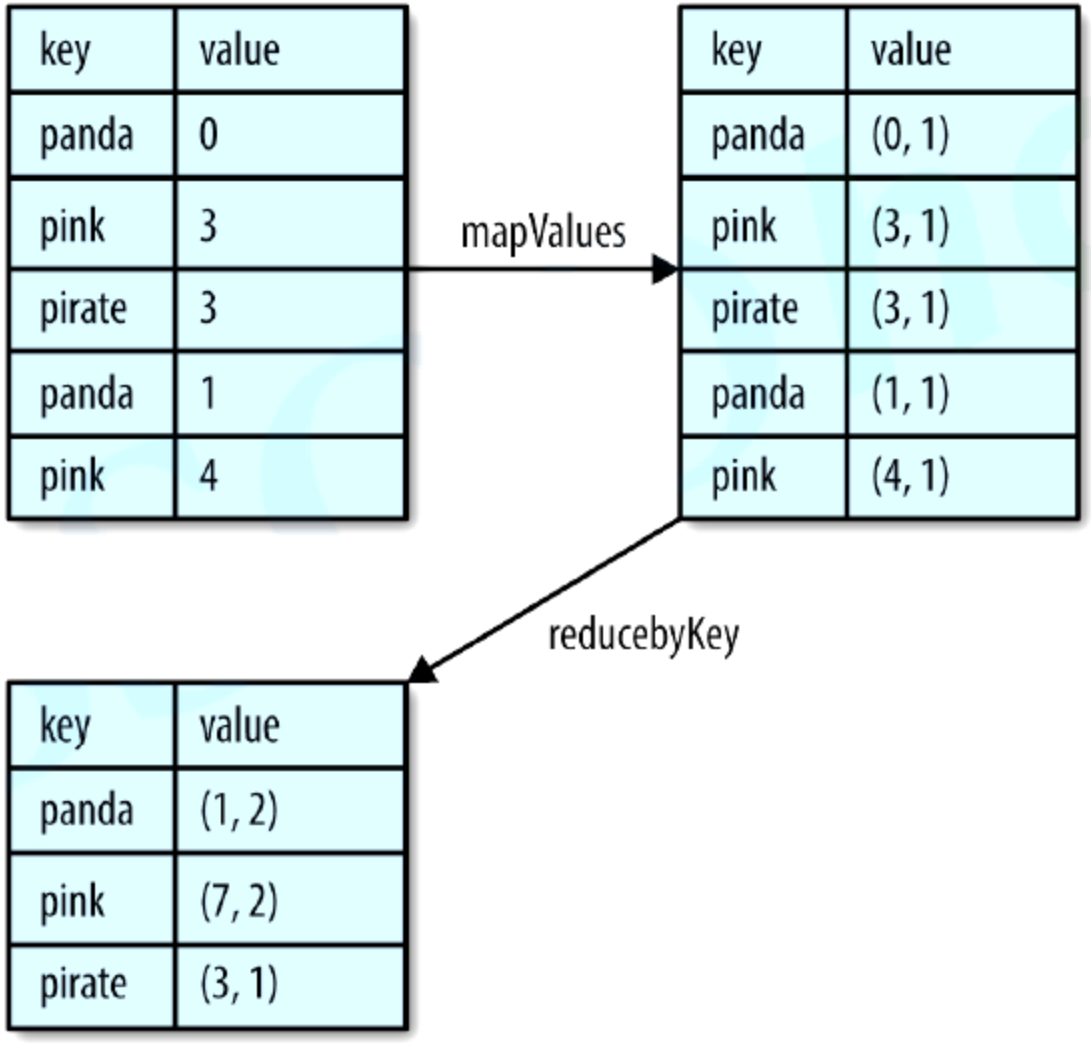
Aggregations: reduceByKey()

- **When datasets are described in terms of key/value pairs, it is common to aggregate statistics across all elements with the same key.**
- **Spark has a set of operations that combines values that have the same key.**
- **These operations return RDDs and thus are transformations rather than actions.**
- **reduceByKey() is quite similar to reduce();**
  - both take a function and use it to combine values.
  - reduceByKey() runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key.
  - Because datasets can have very large numbers of keys, reduceByKey() is not implemented as an action that returns a value to the user program. Instead, it returns a new RDD consisting of each key and the reduced value for that key.

Example: Per-key average w/ reduceByKey()

```
rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

Use **reduceByKey()** along with **mapValues()** to compute the per-key average in a very similar manner to how fold() and map() can be used to compute the entire RDD average



Aggregations: foldByKey()

- **foldByKey() is quite similar to fold();**
  - both use a zero value of the same type of the data in our RDD and combination function.
  - As with fold(), the provided zero value for foldByKey() should have no impact when added with your combination function to another element.

Combing Behavior w/ combineByKey()

- Calling reduceByKey() and foldByKey() will automatically perform combining locally on each machine before computing global totals for each key.
- The user does not need to specify a combiner.
- The more general **combineByKey()** interface allows you to customize combining behavior.

Aggregation with combineByKey()

- combineByKey() is the most general of the per-key aggregation functions. Most of the other per-key combiners are implemented using it.
- Like aggregate(), combineByKey() allows the user to return values that are not the same type as input data.

- As combineByKey() goes through the elements in a partition, each element either has a key it hasn't seen before or has the same key as a previous element.
- Example: computing the average value for each key

```
sumCount = nums.combineByKey((lambda x: (x,1)),
                             (lambda x, y: (x[0] + y, x[1] + 1)),
                             (lambda x, y: (x[0] + y[0], x[1] + y[1])))
sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
```

combineByKey() sample data flow

Partition 1

coffee	1
coffee	2
panda	3

Partition 2

coffee	9

```
def createCombiner(value):
  (value, 1)

def mergeValue(acc, value):
  (acc[0] + value, acc[1] +1)

def mergeCombiners(acc1, acc2):
  (acc1[0] + acc2[0], acc1[1] + acc2[1])
```

Partition 1 trace:
(coffee, 1) -> new key
accumulators[coffee] = createCombiner(1)
(coffee, 2) -> existing key
accumulators[coffee] = merge Value(accumulators[coffee], 2)
(panda, 3) -> new key
accumulators[panda] = createCombiner(3)

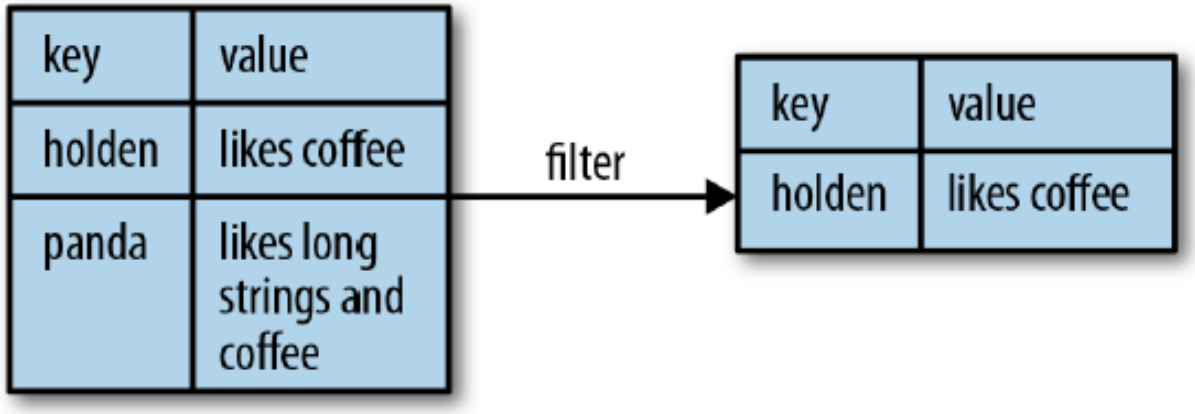
Partition 2 trace:
(coffee, 9) -> new key
accumulators[coffee] = createCombiner(9)

Merge Partitions:
mergeCombiners(partition1.accumulators[coffee],
partition2.accumulators[coffee])

Filter on Values with mapValues()

- Sometimes working with pairs can be awkward if we want to access only the value part of the pair RDD.
- Since this is a common pattern, Spark provides the `mapValues(func)` function, which is the same as

```
map { case (x, y): (x, func(y)) }
```



Examples: Filter on Values

```
>>> pairs = sc.parallelize([("panda", 0), ("pink",3), ("pirate",3), ("panda", 1), ("pink",4)])
>>> pairs.collect()
[('panda', 0), ('pink', 3), ('pirate', 3), ('panda', 1), ('pink', 4)]
>>> pairs.mapValues(lambda x: (x,1)
).reduceByKey(lambda x, y : (x[0] + y[0], x[1] + y[1]
)).collect()
[('pink', (7, 2)), ('panda', (1, 2)), ('pirate', (3, 1))]
```

- From which we can compute average per key

Sorting Data with sortByKey()

- **We can sort an RDD with key/value pairs provided that there is an ordering defined on the key:**
  - Once we have sorted our data, any subsequent call on the sorted data to collect() or save() will result in ordered data.
- **Since we often want our RDDs in the reverse order, the sortByKey() function takes a parameter called ascending indicating whether we want it in ascending order (it defaults to true).**
- **For a different sort order, provide a comparison function.**
  - Example: Custom sort order, sorting integers as if strings

```
rdd.sortByKey(ascending=True, numPartitions=None, keyfunc = lambda x: str(x))
```

Outline

- **Key/value RDDs and Pair RDDs**
- **Transformations on Pair RDDs**
  - Aggregation: reduceByKey(), foldByKey(), combineByKey()
  - Filtering on values: mapValues(), flatMapValues()
  - Sorting with sortByKey()
- **Actions on Pair RDDs**
  - countByKey(), countByValue()
- **Two Pair RDDs: Transformations**
  - join(), leftOuterJoin(), rightOuterJoin(), subtractByKey()

Actions on Pair RDDs

example { (1,2), (3,4), (3,6)}

Function	Description	Example	Result
`countByKey()`	Count the number of elements for each key.	`rdd.countByKey()`	`{(1, 1), (3, 2)}`
`collectAsMap()`	Collect the result as a map to provide easy lookup.	`rdd.collectAsMap()`	`Map{(1, 2), (3, 4), (3, 6)}`
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	`[4, 6]`

Actions on Pair RDDs: countByKey()

- countByKey()

```
>>> sc.parallelize([(1,2), (3,4), (3,6)]).countByKey()
defaultdict(<type 'int'>, {1: 1, 3: 2})
>>> sc.parallelize([(1,2), (3,4), (3,6)]).collectAsMap()
{1: 2, 3: 6}
>>> sc.parallelize([(1,2),(1,900), (3,4), (3,6), (3,7), (3,2)]).lookup(3)
[4, 6, 7, 2]
```

collectAsMap() produces a function, so it keeps the last pair in which a key is mentioned..

Ex: Distributed Word Count w/ countByValue()

- We can use a similar approach to also implement the classic distributed word count problem.
- We will use flatMap() so that we can produce a pair RDD of words and the number 1 and then sum together all of the words using reduceByKey()

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

- We can actually implement word count even faster by using the countByValue() function on the first RDD:
  - `input.flatMap(x => x.split(" ")).countByValue()`

Outline

- **Key/value RDDs and Pair RDDs**
- **Transformations on Pair RDDs**
  - Aggregation: reduceByKey(), foldByKey(), combineByKey()
  - Filtering on values: mapValues(), flatMapValues()
  - Sorting with sortByKey()
- **Actions on Pair RDDs**
  - countByKey(), countByValue()
- **Two Pair RDDs: Transformations**
  - join(), leftOuterJoin(), rightOuterJoin(), subtractByKey()

Transformations on Two Pair RDDs

rdd = { (1,2), (3,4), (3,6) } and other = { (3,9) }

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	`{(1, 2)}`
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.rightOuterJoin(other)`	`{(3,(Some(4),9)), (3,(Some(6),9))}`
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.leftOuterJoin(other)`	`{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,([2],[])), (3, ([4, 6],[9]))}`

Example: Two Pair RDDs: subtractByKey()

```
>>> pairs1.collect()
[(1, 2), (3, 4), (3, 6)]
>>> other.collect()
[(3, 9)]
>>> pairs1.subtractByKey(other).collect()
[(1, 2)]
>>> other.subtractByKey(pairs1).collect()
[]
```

Joins on Two Pair RDDs

- **The simple join() operator is an *inner* join. Only keys that are present in both pair RDDs are output.**

```
rdd = sc.parallelize([("red",20),("red",30),("blue", 100)])
rdd2 = sc.parallelize([("red",40),("red",50),("yellow", 10000)])
rdd.join(rdd2).collect()
# Gives [('red', (20, 40)), ('red', (20, 50)), ('red', (30, 40)), ('red', (30, 50))]
```

Example: Two Pair RDDs: join()

```
>>> pairs1.collect()
[(1, 2), (3, 4), (3, 6)]
>>> other.collect()
[(3, 9)]
>>> pairs1.join(other).collect()
[(3, (4, 9)), (3, (6, 9))]
>>> other.join(pairs1).collect()
[(3, (9, 4)), (3, (9, 6))]
```

Outer Join on Two Pair RDDs

- **Sometimes we don't need the key to be present in both RDDs to want it in our result.**
  - For example, if we were joining customer information with recommendations we might not want to drop customers if there were not any recommendations yet.
- **leftOuterJoin(other) and rightOuterJoin(other) both join pair RDDs by key, where one of the pair RDDs can be missing the key.**
  - With leftOuterJoin() the resulting pair RDD has entries for each key in the source RDD
  - As with join(), we can have multiple entries for each key; this way we get the Cartesian product between the two lists of values

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> sorted(x.leftOuterJoin(y).collect())
[('a', (1, 2)), ('b', (4, None))]
```

Example: Two Pair RDDs: rightOuterJoin()

```
>>> pairs1.collect()
[(1, 2), (3, 4), (3, 6)]
>>> other.collect()
[(3, 9)]
>>> pairs1.rightOuterJoin(other).collect()
[(3, (4, 9)), (3, (6, 9))]
>>> other.rightOuterJoin(pairs1).collect()
[(1, (None, 2)), (3, (9, 4)), (3, (9, 6))]
>>> pairs1.leftOuterJoin(other).collect()
[(1, (2, None)), (3, (4, 9)), (3, (6, 9))]
>>> g = pairs1.cogroup(other).collect()
>>> printStruct1(g)
1 : [ 2 ], [ ]
3 : [ 4 , 6 ], [ 9 ]
```

In rightOuterJoin the key in the result must be present in the "other" (right) operand.

printStruct1 in Next slide

Summary: Key/Value Pair RDDs

- **Key/value RDDs and Pair RDDs**
- **Transformations on Pair RDDs**
  - Aggregation: reduceByKey(), foldByKey(), combineByKey()
  - Filtering on values: mapValues(), flatMapValues()
  - Sorting with sortByKey()
- **Actions on Pair RDDs**
  - countByKey(), countByValue()
- **Two Pair RDDs: Transformations**
  - join(), leftOuterJoin(), rightOuterJoin(), subtractByKey()

Acknowledgements

- Anatoli Melechko, NCSU
- http://spark.apache.org/docs/latest/api/python/pyspark.html
- Learning Spark by Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia (http://shop.oreilly.com/product/0636920028512.do)