6.(key,val) RDD operations

October 11, 2018

0.1 Operations on (key,val) RDDs

0.1.1 Types of spark operations

There are Three types of operations on RDDs: Transformations, Actions and Shuffles.

- The most expensive operations are those the require communication between nodes.

Transformations: RDD → RDD. * **Examples** map, filter, sample, More * **No** communication needed.

Actions: RDD → Python-object in head node. * **Examples:** reduce, collect, count, take, More * **Some** communication needed.

Shuffles: RDD → RDD, **shuffle** needed * **Examples:** sort, distinct, repartition, sortByKey, reduceByKey, join More * **A LOT** of communication might be needed.

0.1.2 Key/value pairs

- A python dictionary is a collection of *key/value* pairs.
- The **key** is used to find a set of pairs with the particular key.
- The **value** can be anything.
- Spark has a set of special operations for *(key,value)* RDDs.

Spark provides specific functions to deal with RDDs in which each element is a key/value pair. Key/value RDDs expose new operations (e.g. aggregating and grouping together data with the same key and grouping together two different RDDs.) Such RDDs are also called pair RDDs. **In python, each element of a pair RDD is a pair tuple.**

```
In [1]: #start the SparkContext
        import findspark
        findspark.init()

        from pyspark import SparkContext
        sc = SparkContext(master="local[4]")
```

0.1.3 Creating (key,value) RDDS

Method 1: `parallelize` a list of pairs.

```
In [2]: pair_rdd = sc.parallelize([(1,2), (3,4)])
        print(pair_rdd.collect())
```

```
[(1, 2), (3, 4)]
```

Method 2: `map` a function that maps elements to key/value pairs.

```
In [3]: regular_rdd = sc.parallelize([1, 2, 3, 4, 2, 5, 6])
        pair_rdd = regular_rdd.map( lambda x: (x, x*x) )
        print(pair_rdd.collect())
```

```
[(1, 1), (2, 4), (3, 9), (4, 16), (2, 4), (5, 25), (6, 36)]
```

0.1.4 Transformations on (key,value) rdds

reduceByKey(func) Apply the reduce function on the values with the same key.

```
In [4]: rdd = sc.parallelize([(1,2), (2,4), (2,6)])
        print("Original RDD :", rdd.collect())
        print("After transformation : ", rdd.reduceByKey(lambda a,b: a+b).collect())
```

```
Original RDD : [(1, 2), (2, 4), (2, 6)]
After transformation :  [(1, 2), (2, 10)]
```

Note that although it is similar to the reduce function, it is implemented as a transformation and not as an action because the dataset can have very large number of keys. So, it does not return values to the driver program. Instead, it returns a new RDD.

sortByKey(): Sort RDD by keys in ascending order.

```
In [5]: rdd = sc.parallelize([(2,2), (1,4), (3,6)])
        print("Original RDD :", rdd.collect())
        print("After transformation : ", rdd.sortByKey().collect())
```

```
Original RDD : [(2, 2), (1, 4), (3, 6)]
After transformation :  [(1, 4), (2, 2), (3, 6)]
```

Note: The output of sortByKey() is an RDD. This means that RDDs do have a meaningful order, which extends between partitions.

mapValues(func): Apply func to each value of RDD without changing the key.

```
In [6]: rdd = sc.parallelize([(1,2), (2,4), (2,6)])
        print("Original RDD :", rdd.collect())
        print("After transformation : ", rdd.mapValues(lambda x: x*2).collect())
```

```
Original RDD : [(1, 2), (2, 4), (2, 6)]
After transformation :  [(1, 4), (2, 8), (2, 12)]
```

groupByKey(): Returns a new RDD of `(key,<iterator>)` pairs where the iterator iterates over the values associated with the key.

Iterators are python objects that generate a sequence of values. Writing a loop over `n` elements as

```
for i in range(n):
    ##do something
```

is inefficient because it first allocates a list of `n` elements and then iterates over it. Using the iterator `xrange(n)` achieves the same result without materializing the list. Instead, elements are generated on the fly.

To materialize the list of values returned by an iterator we will use the list comprehension command:

```
[a for a in <iterator>]
```

```
In [7]: rdd = sc.parallelize([(1,2), (2,4), (2,6)])
        print("Original RDD :", rdd.collect())
        print("After transformation : ", rdd.groupByKey().mapValues(lambda x:[a for a in x]).c

Original RDD : [(1, 2), (2, 4), (2, 6)]
After transformation :  [(1, [2]), (2, [4, 6])]
```

flatMapValues(func): Similar to `flatMap()`: creates a separate key/value pair for each element of the list generated by the map operation.

`func` is a function that takes as input a single value and returns an iterator that generates a sequence of values. The application of flatMapValues operates on a key/value RDD. It applies `func` to each value, and gets an list (generated by the iterator) of values. It then combines each of the values with the original key to produce a list of key-value pairs. These lists are concatenated as in `flatMap`

```
In [8]: rdd = sc.parallelize([(1,2), (2,4), (2,6)])
        print("Original RDD :", rdd.collect())
        # the lambda function generates for each number i, an iterator that produces i,i+1
        print("After transformation : ", rdd.flatMapValues(lambda x: list(range(x,x+2))).colle

Original RDD : [(1, 2), (2, 4), (2, 6)]
After transformation :  [(1, 2), (1, 3), (2, 4), (2, 5), (2, 6), (2, 7)]
```

(Advanced) combineByKey(createCombiner, mergeValue, mergeCombiner): Combine values with the same key using a different result type.

This is the most general of the per-key aggregation functions. Most of the other per-key combiners are implemented using it.

The elements of the original RDD are considered here *values*

Values are converted into *combiners* which we will refer to here as "accumulators". An example of such a mapping is the mapping of the value *word* to the accumulator (*word*,1) that is done in WordCount.

accumulators are then combined with values and the other combiner to generate a result for each key.

For example, we can use it to calculate per-activity average durations as follows. Consider an RDD of key/value pairs where keys correspond to different activities and values correspond to duration.

```
In [9]: rdd = sc.parallelize([("Sleep", 7), ("Work",5), ("Play", 3),
                              ("Sleep", 6), ("Work",4), ("Play", 4),
                              ("Sleep", 8), ("Work",5), ("Play", 5)])

        sum_counts = rdd.combineByKey(
            (lambda x: (x, 1)), # createCombiner maps each value into a  combiner (or accumula
            (lambda acc, value: (acc[0]+value, acc[1]+1)),
        #mergeValue defines how to merge a accumulator with a value (saves on mapping each val
            (lambda acc1, acc2: (acc1[0]+acc2[0], acc1[1]+acc2[1])) # combine accumulators
        )

        print(sum_counts.collect())
        duration_means_by_activity = sum_counts.mapValues(lambda value:
                                                          value[0]*1.0/value[1]) \
                                               .collect()
        print(duration_means_by_activity)
```

```
[('Work', (14, 3)), ('Play', (12, 3)), ('Sleep', (21, 3))]
[('Work', 4.666666666666667), ('Play', 4.0), ('Sleep', 7.0)]
```

To understand combineByKey(), it's useful to think of how it handles each element it processes. As combineByKey() traverses 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.

If it's a new key, createCombiner() is called to create the initial value for the accumulator on that key. In the above example, the accumulator is a tuple initialized as (x, 1) where x is a value in original RDD. Note that createCombiner() is called only when a key is seen for the first time in **each partition.**

If it is a key we have seen before while processing that partition, it will instead use the provided function, mergeValue(), with the current value for the accumulator for that key and the new value.

Since each partition is processed independently, we can have multiple accumulators for the same key. When we are merging the results from each partition, if two or more partitions have an accumulator for the same key, we merge the accumulators using the user-supplied mergeCombiners() function. In the above example, we are just adding the 2 accumulators element-wise.

0.1.5 Transformations on two (key-value) RDDs

```
In [10]: rdd1 = sc.parallelize([(1,2),(2,1),(2,2)])
         rdd2 = sc.parallelize([(2,5),(3,1)])
         print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
```

```
rdd1= [(1, 2), (2, 1), (2, 2)]
rdd2= [(2, 5), (3, 1)]
```

1. subtractByKey: Remove from RDD1 all elements whose key is present in RDD2.

```
In [11]: print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
         print("Result:", rdd1.subtractByKey(rdd2).collect())

rdd1= [(1, 2), (2, 1), (2, 2)]
rdd2= [(2, 5), (3, 1)]
Result: [(1, 2)]
```

2. join:

- A fundamental operation in relational databases.
- assumes two tables have a **key** column in common.
- merges rows with the same key.

Suppose we have two `(key,value)` datasets
|**dataset 1**| |.....| **dataset 2** | | |————-|————————————-| |————-|————| | **key=name** | **(gender,occupation,age)** | | **key=name** | **hair color** | | John | (male,cook,21) | | Jill | blond | | Jill | (female,programmer,19) | | Grace | brown | | John | (male, kid, 2) | | John | black | | Kate | (female, wrestler, 54) |

When `Join` is called on datasets of type `(Key, V)` and `(Key, W)`, it returns a dataset of `(Key, (V, W))` pairs with all pairs of elements for each key. Joining the 2 datasets above yields:

key = name	(gender,occupation,age),haircolor
John	((male,cook,21),black)
John	((male, kid, 2),black)
Jill	((female,programmer,19),blond)

```
In [12]: print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
         print("Result:", rdd1.join(rdd2).collect())

rdd1= [(1, 2), (2, 1), (2, 2)]
rdd2= [(2, 5), (3, 1)]
Result: [(2, (1, 5)), (2, (2, 5))]
```

0.1.6 Variants of join.

There are four variants of `join` which differ in how they treat keys that appear in one dataset but not the other. * `join` is an *inner* join which means that keys that appear only in one dataset are eliminated. * `leftOuterJoin` keeps all keys from the left dataset even if they don't appear in

the right dataset. The result of leftOuterJoin in our example will contain the keys `John, Jill, Kate` * `rightOuterJoin` keeps all keys from the right dataset even if they don't appear in the left dataset. The result of leftOuterJoin in our example will contain the keys `Jill, Grace, John` * `FullOuterJoin` keeps all keys from both datasets. The result of leftOuterJoin in our example will contain the keys `Jill, Grace, John, Kate`

In outer joins, if the element appears only in one dataset, the element in `(K,(V,W))` that does not appear in the dataset is represented bye `None`

3. rightOuterJoin: Perform a right join between two RDDs. Every key in the right/second RDD will be present at least once.

```
In [13]: print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
         print("Result:", rdd1.rightOuterJoin(rdd2).collect())

rdd1= [(1, 2), (2, 1), (2, 2)]
rdd2= [(2, 5), (3, 1)]
Result: [(2, (1, 5)), (2, (2, 5)), (3, (None, 1))]
```

4. leftOuterJoin: Perform a left join between two RDDs. Every key in the left RDD will be present at least once.

```
In [14]: print('rdd1=',rdd1.collect())
         print('rdd2=',rdd2.collect())
         print("Result:", rdd1.leftOuterJoin(rdd2).collect())

rdd1= [(1, 2), (2, 1), (2, 2)]
rdd2= [(2, 5), (3, 1)]
Result: [(1, (2, None)), (2, (1, 5)), (2, (2, 5))]
```

0.1.7 Actions on (key,val) RDDs

```
In [15]: rdd = sc.parallelize([(1,2), (2,4), (2,6)])
```

1. countByKey(): Count the number of elements for each key. Returns a dictionary for easy access to keys.

```
In [16]: print("RDD: ", rdd.collect())
         result = rdd.countByKey()
         print("Result:", result)

RDD:  [(1, 2), (2, 4), (2, 6)]
Result: defaultdict(<class 'int'>, {1: 1, 2: 2})
```

2. collectAsMap(): Collect the result as a dictionary to provide easy lookup.

```
In [17]: print("RDD: ", rdd.collect())
         result = rdd.collectAsMap()
         print("Result:", result)

RDD:  [(1, 2), (2, 4), (2, 6)]
Result: {1: 2, 2: 6}
```

3. lookup(key): Return all values associated with the provided key.

```
In [18]: print("RDD: ", rdd.collect())
         result = rdd.lookup(2 )
         print("Result:", result)

RDD:  [(1, 2), (2, 4), (2, 6)]
Result: [4, 6]
```

0.2 Summary

- We saw some more of the operations on Pair RDDs
- For more, see the spark RDD programming guide
- Next **DataFrames** and Spark-SQL