map(func)

Map transformation returns a new RDD by applying a function to each element of this RDD

```
>>> baby_names = sc.textFile("baby_names.csv")
>>> rows = baby_names.map(lambda line: line.split(","))
```

So, in this transformation example, we're creating a new RDD called "rows" by splitting every row in the baby_names RDD. We accomplish this by mapping over every element in baby_names and passing in a lambda function to split by commas.

From here, we could use Python to access the array

```
>>> for row in rows.take(rows.count()): print(row[1])

First Name
DAVID
JAYDEN
...
```

flatMap(func)

flatMap is similar to map, because it applies a function to all elements in a RDD. But, flatMap flattens the results.

Compare *flatMap* to *map* in the following

```
>>> sc.parallelize([2, 3, 4]).flatMap(lambda x: [x,x,x]).collect()
[2, 2, 2, 3, 3, 3, 4, 4, 4]

>>> sc.parallelize([1,2,3]).map(lambda x: [x,x,x]).collect()
[[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

This is helpful with nested datasets such as found in JSON.

Adding *collect* to *flatMap* and *map* results was shown for clarity. We can focus on Spark aspect (re: the RDD return type) of the example if we don't use collect:

```
>>> sc.parallelize([2, 3, 4]).flatMap(lambda x: [x,x,x])
PythonRDD[36] at RDD at PythonRDD.scala:43
```

filter(func)

Create a new RDD bye returning only the elements that satisfy the search filter. For SQL minded, think where clause.

```
>>> rows.filter(lambda line: "MICHAEL" in line).collect()
Out[36]:
[[u'2013', u'MICHAEL', u'QUEENS', u'M', u'155'],
```

```
 [u'2013', u'MICHAEL', u'KINGS', u'M', u'146'],
 [u'2013', u'MICHAEL', u'SUFFOLK', u'M', u'142']...
```

mapPartitions(*func*, *preservesPartitioning=False*)

Consider mapPartitions a tool for performance optimization if you have the resources available. It won't do much when running examples on your laptop. It's the same as "map", but works with Spark RDD partitions which are distributed. Remember the first D in RDD – Resilient Distributed Datasets.

In examples below that when using *parallelize,* elements of the collection are copied to form a distributed dataset that can be operated on in parallel.

A distributed dataset can be operated on in parallel.

One important parameter for parallel collections is the number of *partitions* to cut the dataset into. Spark will run one task for each partition of the cluster.

```
>>> one_through_9 = range(1,10)
>>> parallel = sc.parallelize(one_through_9, 3)
>>> def f(iterator): yield sum(iterator)
>>> parallel.mapPartitions(f).collect()
[6, 15, 24]

>>> parallel = sc.parallelize(one_through_9)
>>> parallel.mapPartitions(f).collect()
[1, 2, 3, 4, 5, 6, 7, 17]
```

See what's happening? Results [6,15,24] are created because `mapPartitions` loops through 3 partitions which is the second argument to the sc.parallelize call.

Partion 1: 1+2+3 = 6

Partition 2: 4+5+6 = 15

Partition 3: 7+8+9 = 24

The second example produces [1,2,3,4,5,6,7,17] which I'm guessing means the default number of partitions on my laptop is 8.

Partion 1 = 1

Partition 2= 2

Partion 3 = 3

Partition 4 = 4

Partion 5 = 5

Partition 6 = 6

Partion 7 = 7

Partition 8: 8+9 = 17

Typically you want 2-4 partitions for each CPU core in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster or hardware based on standalone environment.
To find the default number of partitions and confirm the guess of 8 above:

```
>>> print sc.defaultParallelism
8
```

mapPartitionsWithIndex(func)

Similar to `mapPartitions`, but also provides a function with an int value to indicate the index position of the partition.

```
>>> parallel = sc.parallelize(range(1,10),4)
>>> def show(index, iterator): yield 'index: '+str(index)+" values: "+
str(list(iterator))
>>> parallel.mapPartitionsWithIndex(show).collect()

['index: 0 values: 1',
 'index: 1 values: 3',
 'index: 2 values: 5',
 'index: 3 values: 7']
```

When learning these APIs on an individual laptop or desktop, it might be helpful to show differences in capabilities and outputs. For example, if we change the above example to use a parallelized list with 3 slices, our output changes significantly:

```
>>> parallel = sc.parallelize(range(1,10),3)
>>> def show(index, iterator): yield 'index: '+str(index)+" values: "+
str(list(iterator))
>>> parallel.mapPartitionsWithIndex(show).collect()

['index: 0 values: [1, 2, 3]',
 'index: 1 values: [4, 5, 6]',
 'index: 2 values: [7, 8, 9]']
```

sample(*withReplacement*,*fraction*, *seed*)

Return a random sample subset RDD of the input RDD

```
>>> parallel = sc.parallelize(range(1,10))
>>> parallel.sample(True,.2).count()
2
```

```
>>> parallel.sample(True,.2).count()
1

>>> parallel.sample(True,.2).count()
2
```

`sample`(*withReplacement*, *fraction*, *seed=None*)

Parameters:

- **withReplacement** – can elements be sampled multiple times (replaced when sampled out)
- **fraction** – expected size of the sample as a fraction of this RDD's size without replacement: probability that each element is chosen; fraction must be [0, 1] with replacement: expected number of times each element is chosen; fraction must be >= 0
- **seed** – seed for the random number generator

union(a different rdd)

Simple. Return the union of two RDDs

```
>>> one = sc.parallelize(range(1,10))
>>> two = sc.parallelize(range(10,21))
>>> one.union(two).collect()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

intersection(a different rdd)

Again, simple. Similar to union but return the intersection of two RDDs

```
>>> one = sc.parallelize(range(1,10))
>>> two = sc.parallelize(range(5,15))
>>> one.intersection(two).collect()
[5, 6, 7, 8, 9]
```

distinct([numTasks])

Another simple one. Return a new RDD with distinct elements within a source RDD

```
>>> parallel = sc.parallelize(range(1,9))
>>> par2 = sc.parallelize(range(5,15))

>>> parallel.union(par2).distinct().collect()
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

Formal API: distinct(): RDD[T]

The Keys

The group of transformation functions (groupByKey, reduceByKey, aggregateByKey, sortByKey, join) all act on key,value pair RDDs.

For the following, we're going to use the baby_names.csv file again which was introduced in a previous post What is Apache Spark?

All the following examples presume the baby_names.csv file has been loaded and split such as:

```
>>> baby_names = sc.textFile("baby_names.csv")
>>> rows = baby_names.map(lambda line: line.split(","))
```

groupByKey([numTasks])

"When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. "

The following groups all names to counties in which they appear over the years.

```
>>> rows = baby_names.map(lambda line: line.split(","))
>>> namesToCounties = rows.map(lambda n: (str(n[1]),str(n[2])
)).groupByKey()
>>> namesToCounties.map(lambda x : {x[0]: list(x[1])}).collect()

[{'GRIFFIN': ['ERIE',
   'ONONDAGA',
   'NEW YORK',
   'ERIE',
   'SUFFOLK',
   'MONROE',
   'NEW YORK',
...
```

reduceByKey(*func*, [*numTasks*])

Operates on key, value pairs again, but the func must be of type (V,V) => V

Let's sum the yearly name counts over the years in the CSV. Notice we need to filter out the header row. Also notice we are going to use the "Count" column value (n[4])

```
>>> filtered_rows = baby_names.filter(lambda line: "Count" not in
line).map(lambda line: line.split(","))
```

```
>>> filtered_rows.map(lambda n:  (str(n[1]), int(n[4]) )
).reduceByKey(lambda v1,v2: v1 + v2).collect()

[('GRIFFIN', 268),
 ('KALEB', 172),
 ('JOHNNY', 219),
 ('SAGE', 5),
 ('MIKE', 40),
 ('NAYELI', 44),
....
```

Formal API: reduceByKey(func: (V, V) ⇒ V): RDD[(K, V)]

aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])

Ok, I admit, this one drives me a bit nuts. Why wouldn't we just use reduceByKey? I don't feel smart enough to know when to use aggregateByKey over reduceByKey. For example, the same results may be produced as reduceByKey:

```
>>> filtered_rows = baby_names.filter(lambda line: "Count" not in
line).map(lambda line: line.split(","))
>>> filtered_rows.map(lambda n:  (str(n[1]), int(n[4]) )
).aggregateByKey(0, lambda k,v: int(v)+k, lambda v,k: k+v).collect()

[('GRIFFIN', 268),
 ('KALEB', 172),
 ('JOHNNY', 219),
 ('SAGE', 5),
...
```

sortByKey(*ascending=True*, *numPartitions=None*, *keyfunc=<function <lambda>>)*

This simply sorts the (K,V) pair by K. Try it out. See examples above on where babyNames originates.

```
>>> filtered_rows.map (lambda n:  (str(n[1]), int(n[4]) )
).sortByKey().collect()
[('AADEN', 18),
 ('AADEN', 11),
 ('AADEN', 10),
 ('AALIYAH', 50),
```

```
 ('AALIYAH', 44),
...

#opposite
>>> filtered_rows.map (lambda n:  (str(n[1]), int(n[4]) )
).sortByKey(False).collect()

[('ZOIE', 5),
 ('ZOEY', 37),
 ('ZOEY', 32),
 ('ZOEY', 30),
...
```

join(*otherDataset*, [*numTasks*])

If you have relational database experience, this will be easy. It's joining of two datasets. Other joins are available as well such as *leftOuterJoin* and *rightOuterJoin*.

```
>>> names1 = sc.parallelize(("abe", "abby", "apple")).map(lambda a: (a,
1))
>>> names2 = sc.parallelize(("apple", "beatty", "beatrice")).map(lambda a:
(a, 1))
>>> names1.join(names2).collect()

[('apple', (1, 1))]
```

leftOuterJoin, *rightOuterJoin*

```
>>> names1.leftOuterJoin(names2).collect()
[('abe', (1, None)), ('apple', (1, 1)), ('abby', (1, None))]

>>> names1.rightOuterJoin(names2).collect()
[('apple', (1, 1)), ('beatrice', (None, 1)), ('beatty', (None, 1))]
```