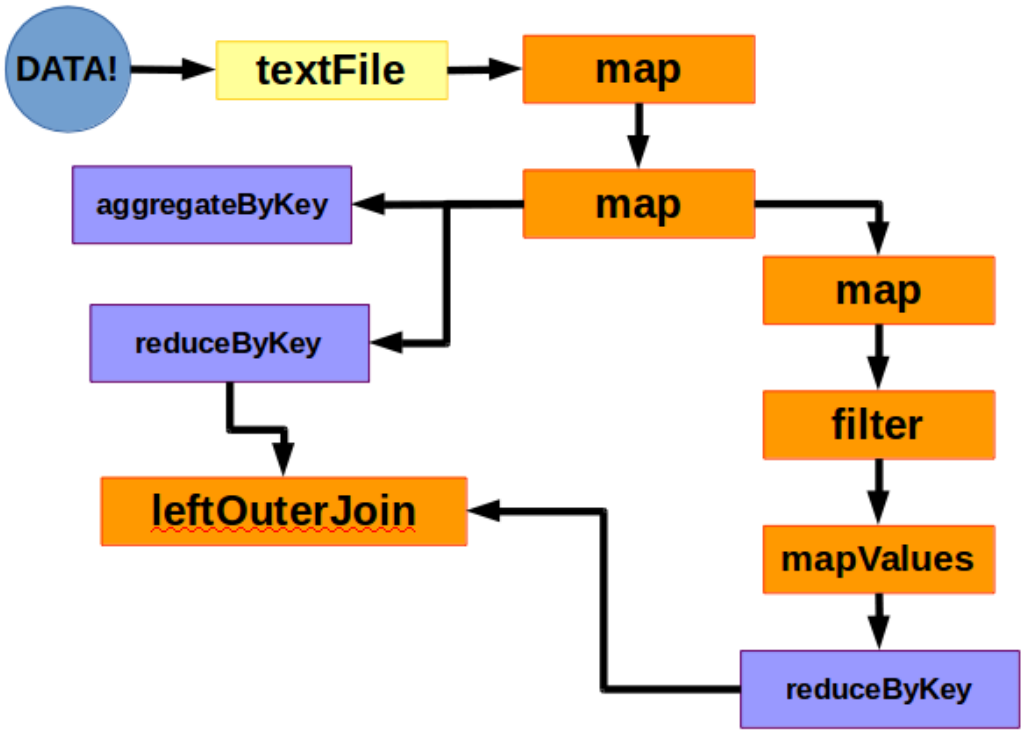
Learn by Marketing

Data Mining + Marketing in Plain English

Working in Pyspark: Basics of Working with Data and RDDs

This entry was posted in Python Spark *on April 23, 2016 by Will*

Summary: Spark (and Pyspark) use map, mapValues, reduce, reduceByKey, aggregateByKey, and join to transform, aggregate, and connect datasets. Each function can be stringed together to do more complex tasks.



Update: Pyspark RDDs are still useful, but the world is moving toward DataFrames. Learn the basics of Pyspark SQL joins as your first foray.

When I first started playing with MapReduce, I was immediately disappointed with how complicated everything was. I'm not a strong Java programmer. I know Python and R fairly well. They're nothing like the complicated Java programs needed for MapReduce.

When I started using Spark, I was enamored. Everything was so simple! With Python (pyspark) on my side, I could start writing programs by combining simple functions. Amazing!

Of course, there is a learning curve. I'd like to share some basic pyspark expressions and idiosyncrasies that will help you explore and work with your data. For this post, I'll be focusing on manipulating Resilient Distributed

Datasets (RDDs) and discuss SQL / Dataframes at a later date.

- Loading Data
- Manipulating Data
- Aggregating Data
- Joining Data

Loading Data

It's as easy as setting...

```
mydata = sc.textFile('file/path/or/file.something')
```

In this line of code, you're creating the "mydata" variable (technically an RDD) and you're pointing to a file (either on your local PC, HDFS, or other data source). Notice how I used the word "pointing"? Spark is *lazy*.

Spark's lazy nature means that it doesn't automatically compile your code. Instead, it waits for some sort of action occurs that requires some calculation.

Manipulating Data

The main way you manipulate data is using the the map() function.

The map (and mapValues) is one of the main workhorses of Spark. Imagine you had a file that was tab delimited and you wanted to rearrange your data to be column1, column3, column2.

I'm working with the MovieLens 100K dataset for those who want to follow along.

```
mycomputer:~$ head u.data
196     242     3       881250949
186     302     3       891717742
22      377     1       878887116
244     51      2       880606923
166     346     1       886397596
298     474     4       884182806
115     265     2       881171488
253     465     5       891628467
305     451     3       886324817
6       86      3       883603013
```

Now we have to first load the data into spark.

```
mydata = sc.textFile("../u.data")
```

Next we have to map a couple functions to our data.

```
mydata.map(lambda x: x.split('\t')).\
    map(lambda y: (y[0], y[2], y[1]))
```

We're doing two things in this one line:

- Using a map to split the data wherever it finds a tab (\t).
- Taking the results of the split and rearranging the results (Python starts its lists / column with zero instead of one).

You'll notice the "lambda x:" inside of the map function. That's called an anonymous function (or a lambda function). The "x" part is a key/value pair of your data. You use "x" after the colon like any other python object – which is why we can split it into a list and later rearrange it.

Here's what the data looks like after these two map functions.

```
(u'196', u'3', u'242'),
(u'186', u'3', u'302'),
(u'22', u'1', u'377'),
(u'244', u'2', u'51'),
(u'166', u'1', u'346'),
(u'298', u'4', u'474'),
(u'115', u'2', u'265'),
(u'253', u'5', u'465'),
(u'305', u'3', u'451'),
(u'6', u'3', u'86')
```

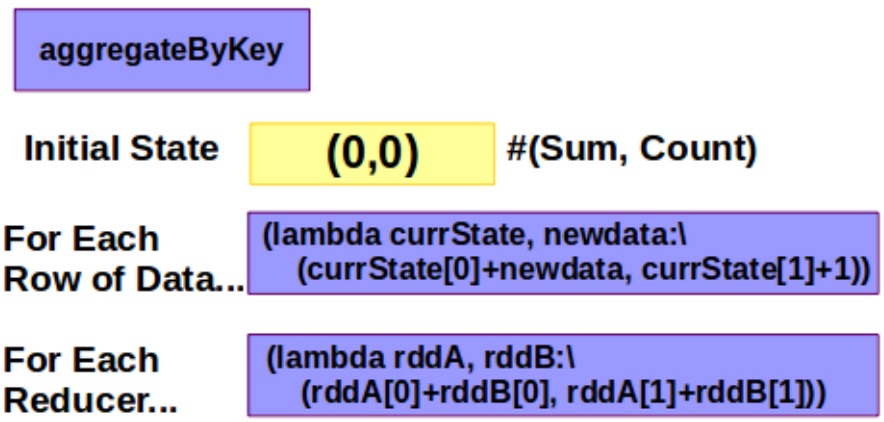
Here's another example of how Spark treats its data. It assumes it's text (especially coming from a textFile load).

So, if we wanted to make those values numeric, we should have written our map as...

```
mydata.map(lambda x: x.split('\t')).\
    map(lambda y: (int(y[0]), float(y[2]), int(y[1])))
```

We now have data that looks like `(196, 3.0, 242)`.

Aggregating Data



The simplest way to add up your values is to use reduce. You'll also frequently use aggregate to do more complicated calculations (like averages)

In comparison to SQL, Spark is much more procedural / functional. If you ask for a grouped count in SQL, the Query Engine takes care of it. In Spark, you need to "teach" the program how to group and count.

Let's assume we saved our cleaned up map work to the variable "clean_data" and we wanted to add up all of the ratings. In order to use the reduce function, we need an RDD of only the numbers we want to add up.

```
clean_data.map(lambda x:(x[1])).\
    reduce(lambda x,y:(x+y))
```


- We re-map the RDD to be only the ratings (the second "column" of our clean_data RDD).
- Then we use the reduce function which needs two parameters
  1. x which is the "previous" value
  2. y which is the "new" value

This is a crucial concept. In functions that aggregate, you're teaching Spark what to do on every row. It's like giving a child a set of instructions: you have to spell out every step.

Now that you have a basic reduce function, you might want to know the average rating for each user. You can do that by using the aggregateByKey function.

First thing, we have to map our data into a pair RDD.

- Essentially we need to have a key in our first column and a single value in the second.
- This is most often done by creating a single tuple containing the multiple values.

So the mapping phase would look like this:

```
user_ratingprod = clean_data.map(lambda x:(x[0],(x[1],x[2])))
```

And the outcome would look like: `(196, (3.0, 242))`. Using this pair RDD, we can take advantage of functions that automatically recognize the key and value components.

```
user_sumcount = user_ratingprod.aggregateByKey((0,0.0),\
    (lambda x, y: (x[0]+y[0],x[1]+1)),\
    (lambda rdd1, rdd2: (rdd1[0]+rdd2[0], rdd1[1]+rdd2[1])))
```

Let's break down what's happening by each line.

1. Call the aggregateByKey function and create a result set "template" with the inital values.
   - We're starting the data out as 0 and 0.0 which will hold our sum of ratings and count of records.
2. For each row of data we're going to do some adding.
   - x is the new template, so x[0] is referring to our "sum" element where x[1] is the "count" element.
   - y is a row's worth of the original data. So you have to pull the right element from the original data. y[0] is the rating.
3. Final step, you're combining RDDs if they were processed on multiple machines.
   - Simply add rdd1 values to rdd2 values based on the template we made.

The data will end up looking like... `(2, (230.0, 62.0))`

Based on the functions we wrote, the first entry contains the sum of the ratings. The second entry contains the count of movies rated.

So in order to get the actual average, we need to mapValues and divide the first

entry by the second entry.

```
user_avgrating = user_sumcount.mapValues(lambda x:(x[0]/x[1]))
#Results in...
#(2, 3.7096774193548385)
```

Finally, you have the average rating by user!

Joining Data

A huge advantage of Spark is its simplicity with joins. Take a trip through history and see what it was like to do a join in MapReduce.

Guess how you do a join in Spark?

```
rdd.join(other_rdd)
```

The only thing you have to be mindful of is the key in your pairRDD.

Just like joining in SQL, you need to make sure you have a common field to connect the two datasets. For Spark, the first element is the key. So you need only two pairRDDs with the same key to do a join.

An important note is that you can also do left (`leftOuterJoin()`)and right joins (`rightOuterJoin()`). In pyspark, when there is a null value on the "other side", it returns a None value.

Let's end with an example:

```
movie_counts = clean_data.map(lambda x: (x[2], 1)).\
    reduceByKey(lambda x,y:x+y)
#(2, 131)
high_rating_movies = clean_data.map(lambda x: (x[2],x[1])).\
    filter(lambda y: y[1] >= 4).\
    mapValues(lambda x: 1).\
    reduceByKey(lambda x,y: x+y)
#(2, 51)
mchr = movie_counts.leftOuterJoin(high_rating_movies)
#(2, (131, 51))
movie_perc_hr = mchr.mapValues(lambda x: x[1]/float(x[0]))
#(2, 0.3893129770992366)
```

Movie Counts: How many ratings did each movie receive?

- Map the clean_data to be movie ID and the number 1.
- Add each row of data together (e.g. 1+1+1+1+1+...1 = 131)

High Rating Movies: How many movies had a higher than average (3) rating?

- Map the data to movie ID and rating.
- Filter the data only for those records with ratings 4 or higher.
- Map the data to movie ID and the number 1.
- Add each row of data together.

mchr: Join the two datasets using a leftOuterJoin (so keey all of movie_counts and return None if not in high_rating_movies).

Movie Perc HR: Calculate the percent of ratings that are higher.

- Using mapValues, take the second element (the high rating count) and divide it by the first element (the total count)

There you have it! You can now load, map, aggregate, and join data in pyspark. It's as simple as stringing together functions that understand and manipulate your data.

Post navigation

← Always Have a Baseline

Building a Recommender System in Spark with ALS →

Back to top