Intro To Spark

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Spark Capabilities

(i.e. Hadoop shortcomings)

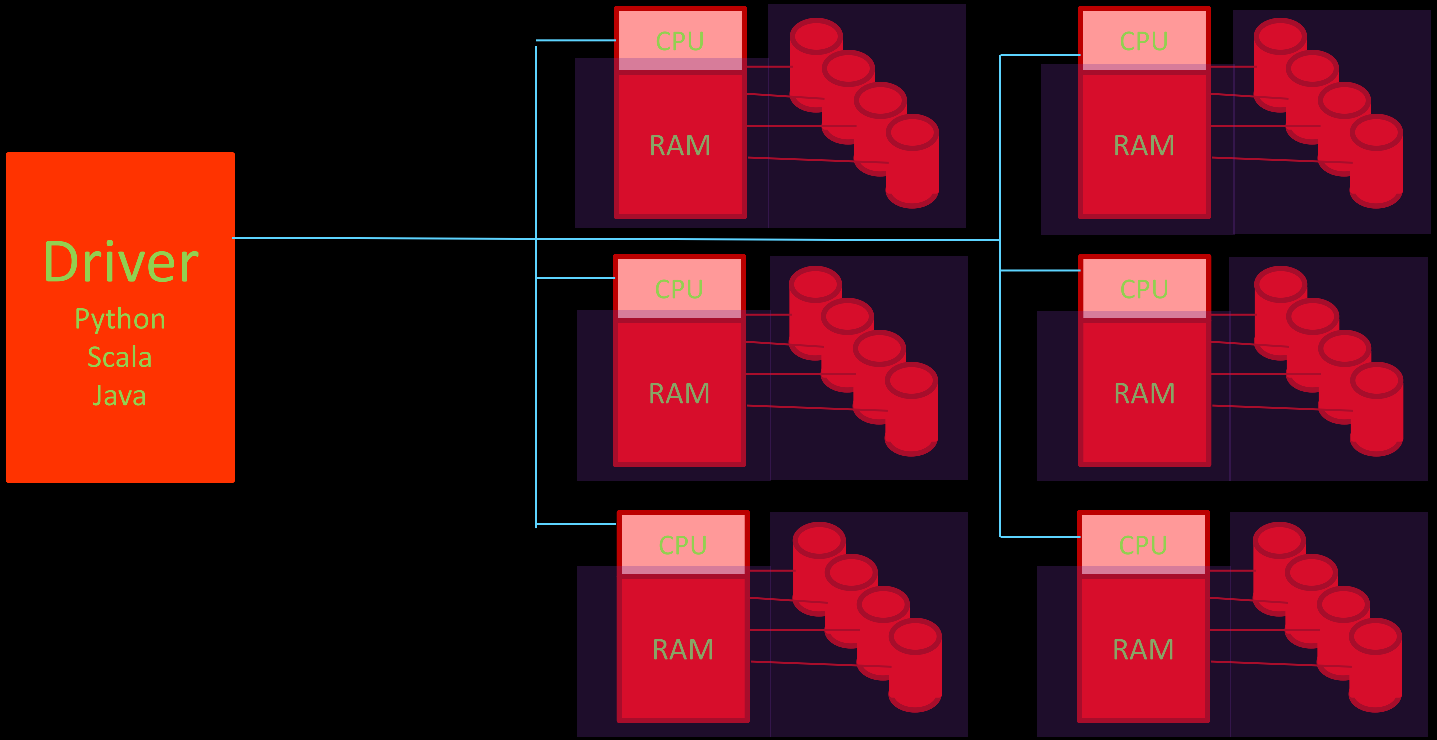
- Performance
  - First, use RAM
  - Also, be smarter

- Ease of Use
  - Python, Scala, Java first class citizens

- New Paradigms
  - SparkSQL
  - Streaming
  - MLib
  - GraphX
  - …more

But using HDFS as the backing store is a common and sensible option.

Same Idea (improved)



RDD

Resilient Distributed Dataset

Spark Formula

1. Create/Load RDD

   Webpage visitor IP address log

2. *Transform* RDD

   "Filter out all non-U.S. IPs"

3. But don't do anything yet!

   Wait until data is actually needed

   Maybe apply more transforms ("Distinct IPs")

4. Perform *Actions* that return data

   Count "How many unique U.S. visitors?"

Let's invite forum visitors to a local conference. How many might there be?

Simple Example

NASA asks: are people viewing our Hubble Space Telescope content?

```
>>> lines_rdd = sc.textFile("nasa_serverlog_202204.csv")
```

Spark Context

The first thing a Spark program requires is a context, which interfaces with some kind of cluster to use. Our pyspark shell provides us with a convenient *sc*, using the local filesystem, to start. Your standalone programs will have to specify one:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("Test_App")
sc = SparkContext(conf = conf)
```

You would typically run these scripts like so:

```
spark-submit Test_App.py
```

Simple Example

```
>>> lines_rdd = sc.textFile("nasa_serverlog_202204.csv")

>>> HubbleLines_rdd = lines_rdd.filter(lambda line: "Hubble" in line)

>>> HubbleLines_rdd.count()
4788

>>> HubbleLines_rdd.first()
'www.nasa.gov\shuttle/missions/Hubble.gif‘
```

Read into RDD

Transform

Actions

<u>Lambdas</u>

We’ll see a lot of these. A lambda is simply a function that is too simple to deserve its own subroutine. Anywhere we have a lambda we could also just name a real subroutine that could go off and do anything.

When all you want to do is see if “*given an input variable line, is “stanford” in there?*”, it isn’t worth the digression.

Most modern languages have adopted this nicety.

Common Transformations

Transformation	Result
map(func)	**Return a new RDD by passing each element through *func*.**
filter(func)	**Return a new RDD by selecting the elements for which *func* returns true.**
flatMap(func)	***func* can return multiple items, and generate a sequence, allowing us to "flatten" nested entries (JSON) into a list.**
distinct()	**Return an RDD with only distinct entries.**
sample(…)	**Various options to create a subset of the RDD.**
union(RDD)	**Return a union of the RDDs.**
intersection(RDD)	**Return an intersection of the RDDs.**
subtract(RDD)	**Remove argument RDD from other.**
cartesian(RDD)	**Cartesian product of the RDDs.**
parallelize(list)	**Create an RDD from this (Python) list (using a spark context).**

Same Size

Fewer Elements

More Elements

Full list at http://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.html#pyspark.RDD

Common Actions

Action	Result
collect()	Return all the elements from the RDD.
count()	Number of elements in RDD.
countByValue()	List of times each value occurs in the RDD.
reduce(func)	Aggregate the elements of the RDD by providing a function which combines any two into one (sum, min, max, …).
first(), take(n)	Return the first, or first n elements.
top(n)	Return the n highest valued elements of the RDDs.
takeSample(…)	Various options to return a subset of the RDD..
saveAsTextFile(path)	Write the elements as a text file.
foreach(func)	Run the *func* on each element. Used for side-effects (updating accumulator variables) or interacting with external systems.

Full list at http://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.html#pyspark.RDD

Transformations vs. Actions

Transformations go from one RDD to another[1].

Actions bring some data back from the RDD.

Transformations are where the Spark machinery can do its magic with lazy evaluation and clever algorithms to minimize communication and parallelize the processing. You want to keep your data in the RDDs as much as possible.

Actions are mostly used either at the end of the analysis when the data has been distilled down (*collect*), or along the way to "peek" at the process (*count*, *take*).

[1] Yes, some of them also create an RDD (parallelize), but you get the idea.

Pair RDDs

- Key/Value organization is a simple, but often very efficient schema, as we mentioned in our NoSQL discussion.

- Spark provides special operations on RDDs that contain key/value pairs. They are similar to the general ones that we have seen.

- On the language (*Python, Scala, Java*) side key/values are simply tuples. If you have an RDD *all* of whose elements happen to be tuples of two items, it is a Pair RDD and you can use the key/value operations that follow.

Pair RDD Transformations

Transformation	Result
reduceByKey(func)	Reduce values using *func*, but on a key by key basis. That is, combine values with the same key.
groupByKey()	Combine values with same key. Each key ends up with a list.
sortByKey()	Return an RDD sorted by key.
mapValues(func)	Use *func* to change values, but not key.
keys()	Return an RDD of only keys.
values()	Return an RDD of only values.

Note that all of the regular transformations are available as well.

Pair RDD Actions

As with transformations, all of the regular actions are available to Pair RDDs, and there are some additional ones that can take advantage of key/value structure.

Action	Result
countByKey()	Count the number of elements for each key.
lookup(key)	Return all the values for this key.

Two Pair RDD Transformations

Transformation	Result
subtractByKey(otherRDD)	Remove elements with a key present in other RDD.
join(otherRDD)	Inner join: Return an RDD containing all pairs of elements with matching keys in self and other. Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in self and (k, v2) is in other.
leftOuterJoin(otherRDD)	For each element (k, v) in self, the resulting RDD will either contain all pairs (k, (v, w)) for w in other, or the pair (k, (v, None)) if no elements in other have key k.
rightOuterJoin(otherRDD)	For each element (k, w) in other, the resulting RDD will either contain all pairs (k, (v, w)) for v in this, or the pair (k, (None, w)) if no elements in self have key k.
cogroup(otherRDD)	Group data from both RDDs by key.

Joins Are Quite Useful

Any database designer can tell you how common joins are. Let's look at a simple example. We have (here we create it) an RD

And an RDD with all of our customers' addr

To create a mailing list of special coupons fo join on the two datasets.

If you are coming from a Pandas DataFrame background, *joins* are congruent with the *Merge* functions. If you've used them, you may have noticed that they can take some time with even small datasets. They do not scale well.

```
>>> best_customers_rdd = sc.parallelize([("Joe", "$103"), ("Alice", "$2000"), ("Bob", "$1200")])

>>> customer_addresses_rdd = sc.parallelize([("Joe", "23 State St."), ("Frank", "555 Timer Lane"), ("Sally", "44
Forest Rd."), ("Alice", "3 Elm Road"), ("Bob", "88 West Oak")])

>>> promotion_mail_rdd = best_customers_rdd.join(customer_addresses_rdd)

>>> promotion_mail_rdd.collect()
[('Bob', ('$1200', '88 West Oak')), ('Joe', ('$103', '23 State St.')), ('Alice', ('$2000', '3 Elm Road'))]
```

Shakespeare, a Data Analytics Favorite

Applying data analytics to the works of Shakespeare has become all the rage. Whether determining the legitimacy of his authorship (it wasn’t Marlowe) or if Othello is actually a comedy (perhaps), or which word makes Macbeth so creepy ("the", yes) it is amazing how much publishable research has sprung from the recent analysis of 400 year old text.

We’re going to do some exercises here using a text file containing all of his works.

Some Simple Problems

We have an input file, Complete _Shakespeare.txt, that you can also find at http://www.gutenberg.org/ebooks/100.
You might find it useful to have http://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.html#pyspark.RDD in a browser window.

If you are starting from scratch on the login node:
1) interact 2) cd BigData/Shakespeare 3) module load spark 4) pyspark

```
...
>>> rdd = sc.textFile("Complete_Shakespeare.txt")
```

Let’s try a few simple exercises.

1) Count the number of lines
2) Count the number of words (hint: Python "split" is a workhorse)
3) Count unique words
4) Count the occurrence of each word
5) Show the top 5 most frequent words

These last two are a bit more challenging. One approach is to think “key/value”. If you go that way, think about which data should be the key and don’t be afraid to swap it about with value. This is a very common manipulation when dealing with key/value organized data.

Some Simple Answers

```
>>> lines_rdd = sc.textFile("Complete_Shakespeare.txt")
>>>
>>> lines_rdd.count()
124787
>>>
>>> words_rdd = lines_rdd.flatMap(lambda x: x.split())
>>> words_rdd.count()
904061
>>>
>>> words_rdd.distinct().count()
67779
>>>
```

Next, I know I'd like to end up with a pair RDD of sorted word/count pairs:

```
(23407, 'the'), (19540,'I'), (15682, 'to'), (15649, 'of') ...
```

Why not just *words_rdd.countByValue()*? It is an *action* that gives us a massive Python unsorted dictionary of results:

```
... 1, 'precious-princely': 1, 'christenings?': 1, 'empire': 11, 'vaunts': 2, 'Lubber's': 1,
'poet.': 2, 'Toad!': 1, 'leaden': 15, 'captains': 1, 'leaf': 9, 'Barnes,': 1, 'lead': 101, 'Hell':
1, 'wheat,': 3, 'lean': 28, 'Toad,': 1, 'trencher!': 2, '1.F.2.': 1, 'leas': 2, 'leap': 17, ...
```

Where to go next? Sort this in Python or try to get back into an RDD? If this is truly *BIG* data, we want to remain as an RDD until we reach our final results. So, no.

Some Harder Answers

```
>>> lines_rdd = sc.textFile("Complete_Shakespeare.txt")
>>>
>>> lines_rdd.count()
124787
>>>
>>> words_rdd = lines_rdd.flatMap(lambda x:
>>> words_rdd.count()
904061
>>>
>>> words_rdd.distinct().count()
67779
>>>
>>> key_value_rdd = words_rdd.map(lambda x: (x,1))
>>>
>>> key_value_rdd.take(5)
[('The', 1), ('Project', 1), ('Gutenberg', 1), ('EBook', 1), ('of', 1)]
>>>
>>> word_counts_rdd = key_value_rdd.reduceByKey(lambda x,y: x+y)
>>> word_counts_rdd.take(5)
[('fawn', 11), ('considered-', 1), ('Fame,', 3), ('mustachio', 1), ('protested,', 1)]
>>>
>>> flipped_rdd = word_counts_rdd.map(lambda x: (x[1],x[0]))
>>> flipped_rdd.take(5)
[(11, 'fawn'), (1, 'considered-'), (3, 'Fame,'), (1, 'mustachio'), (1, 'protested,')]
>>>
>>> results_rdd = flipped_rdd.sortByKey(False)
>>> results_rdd.take(5)
[(23407, 'the'), (19540, 'I'), (18358, 'and'), (15682, 'to'), (15649, 'of')]
>>>
```

Things data scientists do.

Turn these into k/v pairs

Reduce to get words counts

Flip keys and values so we can sort on wordcount instead of words.

```
results_rdd = lines_rdd.flatMap(lambda x: x.split()).map(lambda x: (x,1)).reduceByKey(lambda x,y: x+y).map(lambda x: (x[1],x[0])).sortByKey(False)
```

Spark Anti-Patterns

Here are a couple code clues that you are not working with Spark, but probably against it.

```
for loops, collect in middle of analysis, large data structures
```

```
...

intermediate_results = data_rdd.collect()

python_data = []

for datapoint in intermediate_results:
    python_data.append(modify_datapoint(datapoint))

next_rdd = sc.parallelize(python_data)

...
```

Ask yourself, "would this work with billions of elements?". And likely anything you are doing with a for is something that Spark will gladly parallelize for you, if you let it.

Some Homework Problems

To do research-level text analysis, we generally want to clean up our input. Here are some of the kinds of things you could do to get a more meaningful distinct word count.

1) Remove punctuation. Often punctuation is just noise, and it is here. Do a Map and/or Filter (some punctuation is attached to words, and some is not) to eliminate all punctuation from our Shakespeare data. Note that if you are familiar with regular expressions, Python has a ready method to use those.

2) Remove stop words. Stop words are common words that are also o
remove many obvious stop words with a list of your own, and the *ML*
convenient *StopWordsRemover()* method with default lists for various

3) Stemming. Recognizing that various different words share the same
easy to do simply. Once again, Spark brings powerful libraries into the
Language Tool Kit. You should look at the docs, but you can give it a qu

```
import nltk
from nltk.stem.porter import  *
stemmer = PorterStemmer()
stems_rdd = words_rdd.map( lambda x: stemmer.ste
```

Regular Expressions

This may not be a "Big Data" topic, but this is an incredibly useful capability to have in this field.

These are useful in navigating the command line, building filtering scripts and as integral parts of many programming languages, such as Python, which makes them immediately useful here.

You probably already know some of them, like the * wildcard and can pick up much of the rest in a 10 minute tutorial:

.at *matches any three-character string ending with "at", including "hat", "cat", "bat", "4at", "#at" and " at" (starting with a space).*

[hc]?at *matches "at", "hat", and "cat".*

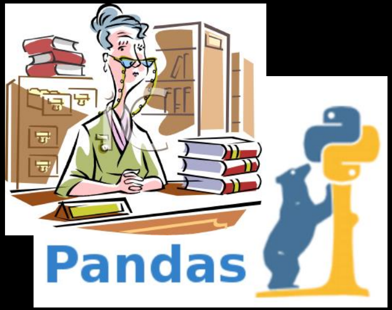
Who needs this Spark stuff?

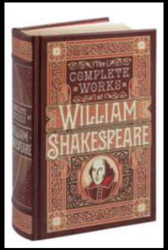
As we do our first Spark exercises, you might think of several ways to accomplish these tasks that you already know. For example, Python *Pandas* is a fine way to do our following problem, and it will probably work on your laptop reasonably well. But they do not scale well*.

However we are learning how to leverage scalable techniques that work on very big data. Shortly, we will encounter problems that are considerable in size, and you will leave this workshop knowing how to harness very large resources.

Searching the *Complete Works of William Shakespeare* for patterns is a lot different from searching the entire Web (perhaps as the 800TB *Common Crawl* dataset).

So everywhere you see an RDD, realize that it is a actually a parallel databank that could scale to PBs.





* See Panda's creator Wes McKinney's "10 Things I Hate About Pandas" at https://wesmckinney.com/blog/apache-arrow-pandas-internals/

Optimizations

We said one of the advantages of Spark is that we can control things for better performance. There are a multitude of optimization, performance, tuning and programmatic features to enable better control. We quickly look at a few of the most important.

- Persistence
- Partitioning
- Parallel Programming Capabilities
- Performance and Debugging Tools

Persistence

- Lazy evaluation implies by default that all the RDD dependencies will be computed when we call an action on that RDD.
- If we intend to use that data multiple times (say we are filtering some log, then dumping the results, but we will analyze it further) we can tell Spark to persist the data.
- We can specify different levels of persistence: *MEMORY_ONLY, MEMORY_ONLY_SER, MEMORY_AND_DISK, MEMORY_AND_DISK_SER, DISK_ONLY*

```
>>> lines_rdd = sc.textFile("nasa_19950801.tsv")
>>> stanfordLines_rdd = lines.filter(lambda line: "stanford" in line)
>>> stanfordLines_rdd.persist(StorageLevel.MEMORY_AND_DISK)
>>> stanfordLines_rdd.count()
47

>>> stanfordLines_rdd.first(1)
['glim.stanford.edu\t-\t807258394\tGET\t/shuttle/…/orbiters-logo.gif\t200\t1932\t\t']
.
.
.
>>> stanfordLines.unpersist()
```

Do before first action.

Actions

Otherwise will just get evicted when out of memory (which is fine).

Partitions

- Spark distributes the data of your RDDs across its resources. It tries to do some obvious things.
- With key/value pairs we can help keep that data grouped efficiently.
- We can create custom partitioners that beat the default (which is probably a hash or maybe range).
- Use persist() if you have partitioned your data in some smart way. Otherwise it will keep getting re-partitioned.

Parallel Programming Features

Spark has several parallel programming features that make it easier and more efficient to do operations in parallel in a more explicit way.

Accumulators are variables that allow many copies of a variable to exist on the separate worker nodes.

It is also possible to have replicated data that we would like all the workers to have access to. Perhaps a lookup table of IP addresses to country codes so that each worker can transform or filter on such information. Maybe we want to exclude all non-US IP entries in our logs. You might think of ways you could do this just by passing variables, but they would likely be expensive in actual operation (usually requiring multiple sends). The solution in Spark is to send an (immutable, read only) broadcast variable

Accumulators

```
log = sc.textFile("logs")
blanks = sc.accumlator(0)

def tokenizeLog(line)
            global blanks      # write-only variable
            if (line =="")
                        blanks += 1
            return line.split(" ")

entries = log.flatMap(tokenizeLog)
entries.saveAsTextFile("parsedlogs.txt")
print "Blank entries: %d" blanks.value
```

Broadcast Variables

```
log = sc.textFile("log.txt")

IPtable = sc.broadcast(loadIPTable())

def countryFilter(IPentry, IPtable)
            return (IPentry.prefix() in IPTable)
USentries = log.filter(countryFilter)
```

Performance & Debugging

We will give unfortunately short shrift to performance and debugging, which are both important. Mostly, this is because they are very configuration and application dependent.

Here are a few things to at least be aware of:

- SparkConf() class. A lot of options can be tweaked here.
- Spark Web UI. A very friendly way to explore all of these issues.

IO Formats

Spark has an impressive, and growing, list of input/output formats it supports. Some important ones:

- Text
- CSV
- SQL type Query/Load
  - JSON (can infer schema)
  - Parquet
  - Hive
  - XML
  - Sequence (Hadoopy key/value)
  - Databases: JDBC, Cassandra, HBase, MongoDB, etc.
- Compression (gzip…)

And it can interface directly with a variety of filesystems: local, HDFS, Lustre, Amazon S3,...

Spark Streaming

Spark addresses the need for streaming processing of data with a API that divides the data into batches, which are then processed as RDDs.

There are features to enable:

- Fast recovery from f
- Load balancing
- Integration with sta
- Integration with oth

15% of the "global datasphere" (quantification of the amount of data created, captured, and replicated across the world) is currently real-time. That number is growing quickly both in absolute terms and as a percentage.

A Few Words About DataFrames

As mentioned earlier, an appreciation for having some defined structure to your data has come back into vogue. For one, because it simply makes sense and naturally emerges in many applications. Often even more importantly, it can greatly aid optimization, especially with the Java VM that Spark uses.

For both of these reasons, you will see that the newest set of APIs to Spark are DataFrame based. This is simply SQL type columns. Very similar to Python pandas DataFrames (but based on RDDs, so not exactly).

We haven't prioritized them here because they aren't necessary, and require a little more code to line up the types properly. But some of the latest features use them.

And while they would just complicate our basic examples, they are often simpler for real research problems. So don't shy away from using them.

Creating DataFrames

It is very pretty intuitive to utilize DataFrames. Your elements just have labeled columns.

A *row RDD* is the basic way to go from RDD to DataFrame, and back, if necessary. A "row" is just a tuple.

```
>>> row_rdd = sc.parallelize([ ("Joe","Pine St.","PA",12543), ("Sally","Fir Dr.","WA",78456),
                               ("Jose","Elm Pl.","ND",45698) ])
>>>
>>> aDataFrameFromRDD = spark.createDataFrame( row_rdd, ["name", "street", "state", "zip"] )
>>> aDataFrameFromRDD.show()
+-----+--------+-----+-----+
| name|  street|state|  zip|
+-----+--------+-----+-----+
|  Joe|Pine St.|   PA|12543|
|Sally| Fir Dr.|   WA|78456|
| Jose| Elm Pl.|   ND|45698|
+-----+--------+-----+-----+
```

Creating DataFrames

You will come across DataFrames created without a schema. They get default column names.

```
>>> noSchemaDataFrame = spark.createDataFrame( row_rdd )
>>> noSchemaDataFrame.show()
+-----+--------+---+-----+
|   _1|      _2| _3|   _4|
+-----+--------+---+-----+
|  Joe|Pine St.| PA|12543|
|Sally| Fir Dr.| WA|78456|
| Jose| Elm Pl.| ND|45698|
+-----+--------+---+-----+
```

Datasets

Spark has added a variation (technically a superset) of *DataFrames* called *Datasets*. For compiled languages with strong typing (Java and Scala) these provide static typing and can detect some errors at compile time.

This is not relevant to Python or R.

And you can create them inline as well.

```
>>> directDataFrame = spark.createDataFrame([ ("Joe","Pine St.","PA",12543), ("Sally","Fir Dr.","WA",78456),
                                              ("Jose","Elm Pl.","ND",45698) ],
                                            ["name", "street", "state", "zip"] )
```

Just Spark DataFrames making life easier…

Data from *https://github.com/spark-examples/pyspark-examples/raw/master/resources/zipcodes.json*

{"RecordNumber":1,"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR","LocationType":"NOT ACCEPTABLE","Lat":17.96,"Long":-66.22,"Xaxis":0.38,"Yaxis":-0.87,"Zaxis":0.3,"WorldRegion":"
{"RecordNumber":2,"Zipcode":704,"ZipCodeType":"STANDARD","City":"PASEO COSTA DEL SUR","State":"PR","LocationType":"NOT ACCEPTABLE","Lat":17.96,"Long":-66.22,"Xaxis":0.38,"Yaxis":-0.87,"Zaxis":0.3,"WorldR
{"RecordNumber":10,"Zipcode":709,"ZipCodeType":"STANDARD","City":"BDA SAN LUIS","State":"PR","LocationType":"NOT ACCEPTABLE","Lat":18.14,"Long":-66.26,"Xaxis":0.38,"Yaxis":-0.86,"Zaxis":0.31,"WorldRegion

```
>>> df = spark.read.json("zipcodes.json")
>>> df.printSchema()
root
 |-- City: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Decommisioned: boolean (nullable = true)
 |-- EstimatedPopulation: long (nullable = true)
 |-- Lat: double (nullable = true)
 |-- Location: string (nullable = true)
 |-- LocationText: string (nullable = true)
 |-- LocationType: string (nullable = true)
 |-- Long: double (nullable = true)
 |-- Notes: string (nullable = true)
 |-- RecordNumber: long (nullable = true)
 |-- State: string (nullable = true)
 |-- TaxReturnsFiled: long (nullable = true)
 |-- TotalWages: long (nullable = true)
 |-- WorldRegion: string (nullable = true)
 |-- Xaxis: double (nullable = true)
 |-- Yaxis: double (nullable = true)
 |-- Zaxis: double (nullable = true)
 |-- ZipCodeType: string (nullable = true)
 |-- Zipcode: long (nullable = true)
```

```
>>> df.show()
+-------------------+-------+-------------+-------------------+-----+--------------------
|               City|Country|Decommisioned|EstimatedPopulation|  Lat|            Location
+-------------------+-------+-------------+-------------------+-----+--------------------
|        PARC PARQUE|     US|        false|               null|17.96|NA-US-PR-PARC PARQUE
|PASEO COSTA DEL SUR|     US|        false|               null|17.96|NA-US-PR-PASEO CO...
|       BDA SAN LUIS|     US|        false|               null|18.14|NA-US-PR-BDA SAN ...
|  CINGULAR WIRELESS|     US|        false|               null|32.72|NA-US-TX-CINGULAR...
|         FORT WORTH|     US|        false|               4053|32.75| NA-US-TX-FORT WORTH
|           FT WORTH|     US|        false|               4053|32.75|   NA-US-TX-FT WORTH
|    URB EUGENE RICE|     US|        false|               null|17.96|NA-US-PR-URB EUGE...
|               MESA|     US|        false|              26883|33.37|       NA-US-AZ-MESA
|               MESA|     US|        false|              25446|33.38|       NA-US-AZ-MESA
|           HILLIARD|     US|        false|               7443|30.69|   NA-US-FL-HILLIARD
|             HOLDER|     US|        false|               null|28.96|     NA-US-FL-HOLDER
|               HOLT|     US|        false|               2190|30.72|       NA-US-FL-HOLT
|          HOMOSASSA|     US|        false|               null|28.78|  NA-US-FL-HOMOSASSA
|       BDA SAN LUIS|     US|        false|               null|18.14|NA-US-PR-BDA SAN ...
|      SECT LANAUSSE|     US|        false|               null|17.96|NA-US-PR-SECT LAN...
|      SPRING GARDEN|     US|        false|               null|33.97|NA-US-AL-SPRING G...
|        SPRINGVILLE|     US|        false|               7845|33.77|NA-US-AL-SPRINGVILLE
|        SPRUCE PINE|     US|        false|               1209|34.37|NA-US-AL-SPRUCE PINE
|           ASH HILL|     US|        false|               1666| 36.4|   NA-US-NC-ASH HILL
|          ASHEBORO|     US|        false|              15228|35.71|   NA-US-NC-ASHEBORO
+-------------------+-------+-------------+-------------------+-----+--------------------
```

And Sometime DataFrames Are Limiting

DataFrames are not as flexible as plain RDDs, and it isn't uncommon to find yourself fighting to do something that would be simple with a map, for example. In that case, don't hesitate to flip back into a plain RDD.

```
>>> row_rdd = sc.parallelize([ ("Joe","Pine St.","PA",12543), ("Sally","Fir Dr.","WA",78456),
                               ("Jose","Elm Pl.","ND",45698) ])

>>> aDataFrameFromRDD = spark.createDataFrame( row_rdd, ["name", "street", "state", "zip"] )

>>> another_row_rdd = aDataFrameFromRDD.rdd
```

Notice that this is not even a method, it is just a property. This is a clue that behind the scenes we are always working with RDDs.

A minor technicality here is that the returned object is actually a "Row" type. You may not care. If you want it be the original tuple type then

```
>>> tuple_rdd = aDataFrameFromRDD.rdd.map(tuple)
```

Note that when our map function is a function that already exists, there is no need for a lambda.

Speaking of pandas, or SciPy, or...

Some of you may have experience with the many Python libraries that accomplish some of these tasks. Immediately relevant to today, *pandas* allows us to sort and query data, and *SciPy* provides some nice clustering algorithms. So why not just use them?

The answer is that Spark does these things in the context of having potentially huge, parallel resources at hand. We don't notice it as Spark is also convenient, but behind every Spark call:

- every RDD could be many TB in size
- every transform could use many thousands of cores and TB of memory
- every algorithm could also use those thousands of cores

So don't think of Spark as just a data analytics library because our exercises are modest. You are learning how to cope with Big Data.

Other Scalable Alternatives: Dask

Of the many alternatives to play with data on your laptop, there are only a few that aspire to scale up to big data. The only one, besides Spark, that seems to have any traction is Dask.

It attempts to retain more of the "laptop feel" of your toy codes, making for an easier port. The tradeoff is that the scalability is a lot more mysterious. If it doesn't work - or someone hasn't scaled the piece you need - your options are limited.

At this time, I'd say it is riskier, but academic projects can often entertain more risk than industry.

Numpy like operations

```
import dask.array as da
a = da.random.random(size=(10000, 10000),
                     chunks=(1000, 1000))
a + a.T - a.mean(axis=0)
```

Dataframes implement Pandas

```
import dask.dataframe as dd
df = dd.read_csv('/.../2020-*-*.csv')
df.groupby(df.account_id).balance.sum()
```

Pieces of Scikit-Learn

```
from dask_ml.linear_model import \
LogisticRegression
lr = LogisticRegression()
lr.fit(train, test)
```

Drill Down?

Run My Programs Or Yours

exec()

```
[urbanic@r001 ~]$ pyspark
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.0-preview2
      /_/

Using Python version 3.7.4 (default, Aug 13 2019 20:35:49)
SparkSession available as 'spark'
In [1]: exec(open("./clustering.py").read())
1 5.76807041184e+14
2 3.73234816206e+14
3 2.13508993715e+14
4 1.38250712993e+14
5 1.2632806251e+14
6 7.97690150116e+13
7 7.14156965883e+13
8 5.7815194802e+13
...
...
...
```

If you have another session window open on bridge's login node, you can edit this file, save it while you remain in the editor, and then run it again in the python shell window with exec(...).

You do *not* need this second session to be on a compute node. Do not start another interactive session.