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
  - MLlib
  - GraphX
  - ...more

But using Hadoop as the backing store is a common and sensible option.
Same Idea

Driver
- Python
- Scala
- Java

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?"*
Simple Example

```python
>>> lines = sc.textFile("nasa_19950801.tsv")
>>> stanfordLines = lines.filter(lambda line: "stanford" in line)
>>> stanfordLines.count()
47
>>> stanfordLines.first()
'u'glim.stanford.edu\t-\t807258357\tGET\t/shuttle/missions/61-c/61-c-patch-small.gif\t'
```

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 HDFS filesystem, to start. Your standalone programs will have to specify one:

```python
conf = SparkConf().setMaster("local").setAppName("Test App")
sc = SparkContext(conf = conf)
```
# Common Transformations

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

Full list at [http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD)
## Common Actions

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

Full list at http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD
We said one of the advantages of Spark is that we can control things for better performance. Some of the most effective ways of doing that are:

- Persistence
- Pair RDDs
- Partitioning
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 = sc.textFile("nasa_19950801.tsv")

>>> stanfordLines = lines.filter(lambda line: "stanford" in line)

>>> stanfordLines.persist(StorageLevel.MEMORY_AND_DISK)

>>> stanfordLines.count()
47

>>> stanfordLines.first(1)
[u.glim.stanford.edu\t-\t807258394\tGET\t/shuttle/.../orbiters-logo.gif\t200\t1932\t]\n
.
.
.

>>> stanfordLines.unpersist()
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.
### Pair RDD Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reduceByKey(func)</code></td>
<td>Reduce values using <em>func</em>, but on a key by key basis. That is, combine values with the same key.</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>Combine values with same key. Each key ends up with a list.</td>
</tr>
<tr>
<td><code>sortByKey()</code></td>
<td>Return an RDD sorted by key.</td>
</tr>
<tr>
<td><code>mapValues(func)</code></td>
<td>Use <em>func</em> to change values, but not key.</td>
</tr>
<tr>
<td><code>keys()</code></td>
<td>Return an RDD of only keys.</td>
</tr>
<tr>
<td><code>values()</code></td>
<td>Return an RDD of only values.</td>
</tr>
</tbody>
</table>

Note that all of the regular transformations are available as well.
## Two Pair RDD Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>subtractByKey(otherRDD)</td>
<td>Remove elements with a key present in other RDD.</td>
</tr>
<tr>
<td>join(otherRDD)</td>
<td>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, (v_1, v_2))) tuple, where ((k, v_1)) is in self and ((k, v_2)) is in other.</td>
</tr>
<tr>
<td>leftOuterJoin(otherRDD)</td>
<td>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).</td>
</tr>
<tr>
<td>rightOuterJoin(otherRDD)</td>
<td>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).</td>
</tr>
<tr>
<td>cogroup(otherRDD)</td>
<td>Group data from both RDDs by key.</td>
</tr>
</tbody>
</table>

Full list at [http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD)
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> z = x.join(y)

>>> z.collect()
['a', (1, 2), ('a', (1, 3))]

Simple Example
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.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>countByKey()</td>
<td>Count the number of elements for each key.</td>
</tr>
<tr>
<td>lookup(key)</td>
<td>Return all the values for this key.</td>
</tr>
</tbody>
</table>

Full list at [http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD](http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD)
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.
Spark has two parallel programming features that make it easier and more efficient to do operations in parallel in a more explicit way.

- Accumulators
- Broadcast variables
log = sc.textFile("logs")
blanks = sc.accumulator(0)

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

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

Accumulators

Accumulators are variables that allow many copies of a variable to exist on the separate worker nodes. At some point we can aggregate (reduce is the parallel programming term) these results back at the driver. An easy to understand an common case is to count some kind of exceptions or errors in the data:
Broadcast Variables

It is also common to have some kind of 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:

```python
log = sc.textFile("log.txt")
IPtable = sc.broadcast(loadIPTable())

def countryFilter(IPentry, IPtable):
    return (IPentry.prefix() in IPTable)  # simple-minded, but simple

USentries = log.filter(countryFilter)
```
Spark has a fair, and growing, list of input/output formats it supports. Some important ones:

- **Text**
- **CSV**
- **JSON**
- **Hadoop Interface**
  - Sequence files (key/value)
  - Old and new Hadoop API
  - Compression (gzip...)
  - Database
    - HBase
    - MongoDB
- **Protocol Buffers (Google thing)**
SparkSQL is a Spark component that enables SQL (or similar Hive Query Language) queries. It uses *DataFrames*, which is a *Dataset* that has been organized into names columns. It can allow for some very convenient manipulation:

```python
# spark is an existing SparkSession, which precedes all this usage
df = spark.sql("SELECT * FROM table")

# Create a DataFrame from the content of a JSON file
df = spark.read.json("phonebook.json")

# Print the schema in a tree format
df.printSchema()
```
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 failures or timeouts
- Load balancing
- Integration with static data and interactive queries
- Integration with other components (SQL, Machine Learning)
We have given 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.
MLib rolls in a lot of classic machine learning algorithms. We barely have time to touch upon this interesting topic today, but they include:

- Useful data types
- Basic Statistics
- Classification (including SVMs, Random Forests)
- Regression
- Algorithms (SGD,...)
- Clustering...
One of the reasons we use spark is for easy access to powerful data analysis tools. The MLlib library gives us a machine learning library that is easy to use and utilizes the scalability of the Spark system.

It has supported APIs for Python (with NumPy), R, Java and Scala.

We will use the Python version in a generic manner that looks very similar to any of the above implementations.

There are good example documents for the clustering routine we are using here:

   http://spark.apache.org/docs/latest/mllib-clustering.html

And an excellent API reference document here:

   http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.clustering.KMeans

I suggest you use these pages for all your Spark work.
Clustering is a very common operation for finding grouping in data and has countless applications. You might have a huge customer database (think Amazon) and you want to cluster into customer types to target. Group #1 might like mustache wax and fixed gear bikes, while another group bundles UGGs and pumpkin spice. Clustering allows you to find these groups in a sea of highly dimensional data.

You might think this is trivial in lower dimensional spaces.

But it can get tricky even there.

We will start with 5000 2D points. We want to figure out how many clusters there are, and their centers. Let’s fire up pyspark and get to it...
Finding Clusters

Using Python version 2.7.5 (default, Nov 20 2015 02:00:19)
SparkContext available as sc, HiveContext available as sqlContext.

```python
>>> rdd1 = sc.textFile("5000_points.txt")
>>> rdd2 = rdd1.map(lambda x: x.split() )
>>> rdd3 = rdd2.map(lambda x: [int(x[0]),int(x[1])] )
>>> rdd3.persist(StorageLevel.MEMORY_ONLY)
```

Read into RDD

Transform to words and integers

Let’s keep this around

*RDD map() takes a function to apply to the elements. We can certainly create our own separate function, but lambdas are a way many languages allow us to define trivial functions “in place”.*
>>> rdd1 = sc.textFile("5000_points.txt")
>>> rdd1.count()
5000
>>> rdd1.take(4)
[u'664159 550946', u'665845 557965', u'597173 575538', u'618600 551446']
>>> rdd2 = rdd1.map(lambda x:x.split())
>>> rdd2.take(4)
[[u'664159', u'550946'], [u'665845', u'557965'], [u'597173', u'575538'], [u'618600', u'551446']]
>>> rdd3 = rdd2.map(lambda x: [int(x[0]), int(x[1])])
>>> rdd3.take(4)
[[664159, 550946], [665845, 557965], [597173, 575538], [618600, 551446]]
Finding Clusters

Using Python version 2.7.5 (default, Nov 20 2015 02:00:19)
SparkContext available as sc, HiveContext available as sqlContext.
>>> rdd1 = sc.textFile("5000_points.txt")
>>> rdd2 = rdd1.map(lambda x: x.split())
>>> rdd3 = rdd2.map(lambda x: [int(x[0]), int(x[1])])
>>> rdd3.persist(StorageLevel.MEMORY_ONLY)
>>> from pyspark.mllib.clustering import KMeans

```python
from pyspark.mllib.clustering import KMeans

class KMeans(KMeansModel):
    def __init__(self, clusters, centroids, model_history, run_history):
        """New in version 0.9.0.
        ""

    @staticmethod
    def train(rdd, k, maxIterations=100, runs=1, initializationMode="k-means\]
    (default: None, initializationSteps=5, epsilon=0.0001, initModel=None)"
```
Finding Clusters
Finding Clusters

Let’s see results for 1-30 cluster tries

<table>
<thead>
<tr>
<th>Clusters</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.76807041184e+14</td>
</tr>
<tr>
<td>2</td>
<td>3.43183673951e+14</td>
</tr>
<tr>
<td>3</td>
<td>2.23097486536e+14</td>
</tr>
<tr>
<td>4</td>
<td>1.64792608443e+14</td>
</tr>
<tr>
<td>5</td>
<td>1.19410028576e+14</td>
</tr>
<tr>
<td>6</td>
<td>7.97690151116e+13</td>
</tr>
<tr>
<td>7</td>
<td>7.16451594344e+13</td>
</tr>
<tr>
<td>8</td>
<td>4.81469246295e+13</td>
</tr>
<tr>
<td>9</td>
<td>4.23762700793e+13</td>
</tr>
<tr>
<td>10</td>
<td>3.65230706654e+13</td>
</tr>
<tr>
<td>11</td>
<td>3.16991867996e+13</td>
</tr>
<tr>
<td>12</td>
<td>2.94369408304e+13</td>
</tr>
<tr>
<td>13</td>
<td>2.04031903147e+13</td>
</tr>
<tr>
<td>14</td>
<td>1.3701893034e+13</td>
</tr>
<tr>
<td>15</td>
<td>8.91761561687e+12</td>
</tr>
<tr>
<td>16</td>
<td>3.1833652006e+13</td>
</tr>
<tr>
<td>17</td>
<td>1.39019717893e+13</td>
</tr>
<tr>
<td>18</td>
<td>8.22806178508e+12</td>
</tr>
<tr>
<td>19</td>
<td>8.22513516563e+12</td>
</tr>
<tr>
<td>20</td>
<td>7.79359299283e+12</td>
</tr>
<tr>
<td>21</td>
<td>7.7965059172e+12</td>
</tr>
<tr>
<td>22</td>
<td>7.7001662709e+12</td>
</tr>
<tr>
<td>23</td>
<td>7.24231610447e+12</td>
</tr>
<tr>
<td>24</td>
<td>7.21990743993e+12</td>
</tr>
<tr>
<td>25</td>
<td>7.0935133944e+12</td>
</tr>
<tr>
<td>26</td>
<td>6.9257789424e+12</td>
</tr>
<tr>
<td>27</td>
<td>6.53939015776e+12</td>
</tr>
<tr>
<td>28</td>
<td>6.37782690833e+12</td>
</tr>
<tr>
<td>29</td>
<td>6.37192522244e+12</td>
</tr>
</tbody>
</table>
for trials in range(10):
    ... print
    ... for clusters in range(12, 18):
        ... model = KMeans.train(rdd3, clusters)
        ... print clusters, model.computeCost(rdd3)
for trials in range(10):
    for clusters in range(12, 16):
        model = KMeans.train(rdd3, clusters)
        cost = model.computeCost(rdd3)
        centers = model.clusterCenters
        if cost<1e+13:
            print clusters, cost
            for coords in centers:
                print int(coords[0]), int(coords[1])
            break

Find the Centers

15 8.91761561687e+12
852058 157685
606574 574455
320602 161521
139395 558143
858947 546259
337264 562123
244654 847642
398870 404924
670929 862765
823421 731145
507818 175610
801616 321123
617926 399415
417799 787001
167856 347812
15 8.91765957989e+12
670929 862765
139395 558143
244654 847642
852058 157685
617601 399504
801616 321123
507818 175610
337264 562123
858947 546259
244654 847642
398870 404924
670929 862765
823421 731145
507818 175610
801616 321123
617926 399415
417799 787001
167856 347812
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), it is amazing how much 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.

Make sure it is in your HDFS filesystem. Start “pyspark” and load the data in the usual manner:

```python
>>> 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

3) Count unique words

4) Count the occurrence of each word

5) Show the top 5 most frequent words
Some Simple Answers

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

>>> words = rdd.flatMap(lambda x: x.split())
key_value = words.map(lambda x: (x,1))

>>> key_value.take(5)
[(u'The', 1), (u'Project', 1), (u'Gutenberg', 1), (u'EBook', 1), (u'of', 1)]

>>> word_counts = key_value.reduceByKey(lambda x,y: x+y)

>>> word_counts.take(5)
[(u'fawn', 11), (u'considered-', 1), (u'Fame,', 3), (u'mustachio', 1), (u'protested,', 1)]

>>> flipped = word_counts.map(lambda x: (x[1],x[0]))

>>> flipped.take(5)
[(11, u'fawn'), (1, u'considered-'), (3, u'Fame,'), (1, u'mustachio'), (1, u'protested,')]

>>> results = flipped.sortByKey(False)

>>> results.take(5)
[(23407, u'the'), (19540, u'I'), (18358, u'and'), (15682, u'to'), (15649, u'of')]

>>> results = 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)
```

- **Turn these into k/v pairs**
- **Reduce words**
- **Flip keys and values so we can sort on wordcount instead of words.**
A Homework Problem

We might want to do some serious analysis. Perhaps we want to correlate word frequency between various works. Some words are globally frequent, but uninteresting (“the”), others may be concentrated in meaningful places.

As a first step, you probably want to separate the works. A quick survey of the file reveals that the «<<Copyright>>» text separates sections. But that is also inserted between acts.

Can you separate the works properly? Do you want them in separate RDDs, or with separate keys?