CMU MSP 36602: Spark, Part 1

H. Seltman, March 18-20 2019

1) **Apache Spark** is an alternative to MapReduce that has a wider range of capabilities, is often faster, can use either Hadoop or some other distributed storage system, but uses its own processing system in the form of **parallel in-memory cluster computing**. It requires large amounts of (expensive) RAM.
2) ***Links***:
   a) Official Website: http://spark.apache.org/
   b) Tutorial: http://www.tutorialspoint.com/apache_spark/
   c) Cheatsheet: https://d1jnx9ba8s6j9r.cloudfront.net/blog/wp-content/uploads/2018/10/PySpark_CheatSheet_Edureka.pdf
3) ***Spark is already loaded*** on your virtual machine.
4) ***Features***
   a) The in-memory computing model reduces disk I/O, speeding up tasks, often dramatically.
   b) Spark supports Java, Python, and Scala programming languages.
   c) In addition to "map" and "reduce", Spark supports SQL queries, Streaming data, Machine learning, and Graph algorithms.
   d) Spark can sit directly on the Hadoop File System or on top of Hadoop Yarn or Hadoop MapReduce (both of which sit directly on HDFS).
   e) The ***fundamental data structure*** in Spark is **RDD (Resilient Distributed Datasets)**. It is an immutable structure partitioned across nodes of a computing cluster that is fault tolerant and allows parallel computing.
   f) The main components of Spark are:
      i) **Apache Spark Core**, which provides distributed task dispatching, scheduling, and basic I/O functionalities.
      ii) various top-level components of Spark which sit on the core:
         (1) **Spark SQL:** provides the data abstraction "SchemaRDD" to allow data analysis via SQL-like commands
         (2) **Spark Streaming:** uses a distributed processing system to read data in batches and perform RDD transformations on those batches
         (3) **MLib:** performs distributed machine learning
         (4) **GraphX**: models user defined graphs

g) Spark addresses the ***fundamental speed weakness of MapReduce***: separate jobs in a workflow are stored on the disks of the HDFS. There is no other direct way to connect jobs. Similarly, multiple queries on the same dataset each go back to disk storage to get the data. Spark uses distributed memory (RAM) instead of a distributed file system whenever possible.

5) **Aside on Scala:** Scala is a programming language designed to address deficiencies in Java, but which produces Java bytecode that is run on a Java Virtual Machine.

6) Spark can be ***run in local mode*** for learning and testing. To ***run on a cluster*** you need to prepare a list of Java ARchive (JAR) files to be deployed to each cluster node. These files must be prepared using a "build" system such as Apache Maven or "sbt".

7) The basic Scala ***interactive version*** of Spark uses the **Spark Shell**.

   a) Invoke with `spark-shell` and quit with `:quit`.

   b) The prompt tells you that your language is scala. The `sc` "spark context" variable is pre-defined.

   c) A simple scala program in spark:

```
scala> val inputfile = sc.textFile("input.txt")
scala> inputfile.getClass
res1: Class[_ <: org.apache.spark.rdd.RDD[String]] = class
      org.apache.spark.rdd.MapPartitionsRDD
scala> val words = inputfile.flatMap(line => line.split(" "))
scala> words.collect()
res7: Array[String] = Array(now, is, the, time, for, all, good, men,
   to, come, to, the, aid, of, their, country., Are, you, going, to,
   be, a, good, man?)
scala> val countmap = words.map(word => (word, 1))
scala> countmap.take(6)
res2: Array[(String, Int)] = Array((now,1), (is,1), (the,1), (time,1),
  (for,1), (all,1))
scala> val counts = countmap.reduceByKey(_+_)
scala> counts.take(5)
res3: Array[(String, Int)] = Array((country.,1), (men,1), (is,1),
  (you,1), (to,3))
/* next line keeps the RDD in memory for future operations */
scala> counts.saveAsTextFile("myDir")
:quit
```

8) **pyspark** lets you work in Python. The basic data type is a parallelized list of anything. In context, a tuple of length two may be interpreted as a key / value pair.

   a) Links

      i) http://spark.apache.org/docs/2.1.0/api/python/pyspark.html

      ii) http://www.mccarroll.net/blog/pyspark2/

   b) These command are your in "~/.bashrc" to ***make Spark will use python 3*** instead of 2 (which is deprecated).

```
export PYSPARK_DRIVER_PYTHON=python3.5
export PYSPARK_PYTHON=python3.5
```

c) At the Unix prompt, ***identify the "Filesystem" name to access Hadoop*** within Spark:

```
$ hdfs dfs -df  # "disk free"
Filesystem                     Size        Used   Available  Use%
hdfs://localhost:54310  21523689472  1860243456  9230733312    9%
```

d) ***Start pyspark*** from the Unix prompt with `$pyspark`. This is Python, but with access to a library that does Spark tasks.

e) `dir()` shows that the `sc` object already exists. This is the **spark context** class object that provides the Spark connection. There can be only one spark context in a given session.

f) The spark context has a `.parallelize()` method that **takes a regular python data object and returns an RDD object**. It also has a `.textFile()` method that **converts a file on the Linux or Hadoop system to an RDD object**.

To access a file on the Hadoop side, using *8c* above, the filename starts "hdfs://localhost:54310/user/student/".

g) ***Quick pyspark word count example***:

```
$ echo I am the walrus > walrus.txt
$ echo You are the walrus >> walrus.txt
$ pyspark
>>> walrus = sc.textFile("walrus.txt")
>>> walrus.collect()  # careful -- use only on small data
['I am the walrus', 'You are the walrus']
>>> walrus.map(lambda line: line.split()).collect()
[['I', 'am', 'the', 'walrus'], ['You', 'are', 'the', 'walrus']]
>>> walrus.flatMap(lambda line: line.split()).collect()
['I', 'am', 'the', 'walrus', 'You', 'are', 'the', 'walrus']
>>> words = walrus.flatMap(lambda line: line.split())
>>> words.take(5)
['I', 'am', 'the', 'walrus', 'You']
>>> words = words.map(lambda w: (w, 1))
>>> words.sample(withReplacement=False, fraction=0.5).collect()
[('You', 1), ('are', 1), ('the', 1)]
>>> counts = words.reduceByKey(lambda x, y: x + y)
>>> counts.collect()
[('I', 1), ('the', 2), ('You', 1), ('are', 1), ('am', 1), ('walrus', 2)]
>>>
>>> quit()
$ ls walrus
part-00000  _SUCCESS
$ cat walrus/part-00000
```

h) Running as a batch file

Make a .py file starting with

```
from pyspark import SparkContext
sc = SparkContext("local", "myAppName", pyFiles=[])
```

and then include any pyspark code.
At the unix prompt run:

```
spark-submit myPySparkFile.py
```

i) ***We will divide spark context methods into three groups:*** i) those that do not use a key, ii) those that work with keys, and iii) those that tune the partitions. In this handout, we look at the first of these.

j) Spark context methods that do not use a key

i) Note that the default `print()` result for an RDD shows some information but no contents.

ii) **`.count()`** returns the number of elements in an RDD.

iii) **`.collect()`** converts ***all*** of an RDD to a list, usually for debugging purposes

iv) Except for small data sets, we just ***peek at what is happening*** with **`.take(num)`** which makes a ***list*** (not RDD) of the first "num" elements. To ***get a random subset***, use **`.takeSample(withReplacement=myBoolean, num)`**, which also returns a list and respects the value of "withReplacement". You can also use ***`.first()`***, which is the same as `.take(1)[0]`.

v) To make an ***RDD*** that is a ***random sample of an RDD***, use **`.sample(withReplacement=myBoolean, fraction=myFloat)`**. Then `.collect()` or `.take()` can be used to examine the random sample, if needed.

vi) If the data in the RDD are numeric then **`.sum()`**, **`.mean()`**, **`.min()`**, **`.max()`**, **`.sampleStdev()`**, and **`.sampleVariance()`** return the appropriate sample ***statistic*** as a float.

The .stats() method returns a StatCounter class object, which has methods count(), sum(), etc. You can merge the StatCounter results from two StatCounters using combinedStats = `myStatCounter1.merge(myStatCounter2)`. You can also convert it to a dictionary with `.asDict()`.

vii) You can ***count data in bins*** using **`.histogram(buckets)`**. E.g.,

```
>>> c = sc.parallelize([0, 2, 3, 4, 1.2, 4, 2, 4.5, 24])
>>> c.histogram(3)
([0, 8, 16, 24], [8, 0, 1])
>>> c.histogram(list(range(-1, 30, 10)))
([-1, 9, 19, 29], [8, 0, 1])
>>> c.histogram(list(range(-1, 20, 10)))
([-1, 9, 19], [8, 0])
```

viii) ***Set operations*** are carried out with **`.distinct()`**, **`.intersection()`**, **`.union()`** and **`.subtract().`**

```
>>> A = sc.parallelize([1.5, 1.5, 2.2, 0.6])
>>> setA = A.distinct()
>>> setA.collect()
[0.6, 1.5, 2.2]
>>> setB = sc.parallelize([1.5, 1.7, 1.9])
>>> setA.union(setB).collect()
[1.5, 2.2, 0.6, 1.5, 1.7, 1.9]
>>> setA.intersection(setB).collect()
[1.5]
>>> setA.subtract(setB).collect()
[0.6, 2.2]
>>> setB.subtract(setA).collect()
[1.7, 1.9]
```

ix) The **`.filter(f)`** method ***returns an RDD with only selected elements***. The "`f`" argument is a function that takes one value and returns a Boolean. Elements with a True return value end up in the new RDD.

```
>>> import random
>>> r = sc.parallelize([chr(ord('A')+random.randrange(26)) for __ in
range(10)])
>>> ram = r.filter(lambda x: x <= 'M')
>>> ram.collect()
['A', 'I', 'M', 'G', 'M']
>>> data = sc.parallelize([(1, 4, 2), (2, 2, 4), (3, 5, 8), (3, 5, 9)])
>>> def sums(triple):
...     return triple[0] + triple[1] == triple[2]
>>> data.filter(sums).collect()
[(2, 2, 4), (3, 5, 8)]
```

x) The most commonly used method is **`.map(f)`**, which ***applies function `f()` to every element of the RDD*** and returns a new RDD which is a list of values whose type is the type of `f()`'s return value. The function is commonly a lambda function with a single argument, returning any value. E.g., if the contents of RDD "three" are tuples of length 3, then `three.map(lambda x: (x[1], (x[0] + x[2])/2))` will create an RDD of the same count as the original, but with values that are tuples of length 2 (the original middle value and the mean of the other values).

It is also OK to use a function defined separately using the usual `def f(args):` syntax.

xi) The **`.foreach()`** method is similar to `.map()`, but there is no return value. Therefore it is only useful for its side effects, e.g., printing or inserting specific elements into a database.

xii) The **`.groupBy(f)`** method applies single argument function `f()` to each element of the RDD and ***returns an RDD with a tuple for each unique value of the return value***. The first element of the return tuple is the "key" (return value) and the second is a generator containing the original data matching the key.

```
>>> three = sc.parallelize([(1,2,3), (2,2,2), (2,5,3)])
>>> threeG = three.groupBy(lambda x: x[0])
>>> threeG.collect()
[(1, <pyspark.resultiterable.ResultIterable object at
   0x7f40840b65c0>), (2, <pyspark.resultiterable.ResultIterable
   object at 0x7f40840b6c18>)]
>>> threeG.map(lambda kv: (kv[0], tuple(kv[1]))).collect()
[(1, ((1, 2, 3),)), (2, ((2, 2, 2), (2, 5, 3)))]
```

xiii) The **`.cartesian(other)`** method returns an RDD with ***all possible tuples*** made by taking one value from the "self" RDD and one from the "other" RDD.

```
>>> letters = sc.parallelize(['a', 's', 'd', 'f'])
>>> nums = sc.parallelize([2, 10, 6])
>>> letters.cartesian(nums).collect()
[('a', 2), ('a', 10), ('a', 6), ('s', 2), ('s', 10), ('s', 6), ('d',
2), ('d', 10), ('d', 6), ('f', 2), ('f', 10), ('f', 6)]
```

xiv) The **`.flatMap()`** method makes a larger RDD by applying a function that returns a tuple to every element of the original RDD and treats the result as separate elements.

```
>>> x = sc.parallelize([2, 4, 6])
>>> x.map(lambda x: (x, x**2)).collect()
[(2, 4), (4, 16), (6, 36)]
>>> x.flatMap(lambda x: (x, x**2)).collect()
[2, 4, 4, 16, 6, 36]
```

xv) The **`.fold(zero, op)`** method ***combines values from all elements of the RDD***. The function `op()`, which may be a lambda function, always takes two arguments and the first argument matches `zero`'s type while the second argument matches the RDD type. The return type matches `zero`, not (necessarily) the RDD data type.

E.g., if each value is an "n", a "sum", and an "id", then we might want to add the "n" and "sum" values separately:

```
>>> nv = sc.parallelize([(3, 22, 'a'), (2, 17, 'f'), (1, 5, 'z')])
>>> sums = nv.fold((0,0), lambda x, y: (x[0]+y[0], x[1]+y[1]))
>>> sums
(6, 44)
>>> print("mean =", sums[1]/sums[0])
mean = 7.33333333333333
```

```
>>> def special(a, b):
...     out1 = a[0] + b[0]
...     out2 = a[1] + b[1]
...     out3 = max(a[2], b[2])
...     return (out1, out2, out3)
...
>>> nv.fold((0,0,'a'), special)
(6, 44, 'z')
```

xvi) The **`.reduce(f)`** method is similar to the `.fold()` method, but there is no `zero` argument, so the function return type must be the same as the data in the RDD.

`.fold(zero, op)` is generally safer because it works on an empy RDD.

```
>>> x = sc.parallelize([(1,4), (2, 5), (3, 6)])
>>> x.reduce(lambda x, y: (min(x[0], y[0]), x[1]+y[1]))
(1, 15)
```

xvii) The **`.randomSplit()`** method ***splits an RDD into two or more random subsets***.

```
>>> R = sc.parallelize(range(50))
>>> train, test = R.randomSplit([0.7, 0.3])
>>> test.take(10)
[4, 8, 19, 22, 24, 25, 27, 30, 36, 38]
```

xviii) The **`.saveAsTextFile(path)`** method ***saves the RDD to a file***. Similar functions save in different formats or to the Hadoop rather than Linux side.

xix) The **`.sortBy(keyfunc, ascending=True)`** method returns ***an RDD with the values sorted*** based on the result of `keyfunc()` applied to each element of the RDD.

```
>>> three.collect()
[(1, 2, 3), (2, 2, 2), (2, 5, 3)]
>>> three.sortBy(lambda tup: tup[2]).collect()
[(2, 2, 2), (1, 2, 3), (2, 5, 3)]
```

xx) The **.zip()** method ***creates one RDD from two of the same size*** (count) by constructing tuples.

```
>>> R1 = sc.parallelize(['A', 'E', 'F'])
>>> R2 = sc.parallelize([(1, 4), (2,7), (3,1)])
>>> R1.zip(R2).collect()
[('A', (1, 4)), ('E', (2, 7)), ('F', (3, 1))]
```

xxi) The **`.glom()`** method flattens the data into one giant list (per partition; see next handout).

```
>>> text = sc.parallelize(['This is sentence 1.  And', 'this is',
    'sentence 2.'])
>>> text.glom().collect()
[['This is sentence 1.  And', 'this is', 'sentence 2.']]
>>> text.glom().map(lambda x: " ".join(x)).collect()
['This is sentence 1.  And this is sentence 2.']
```

k) ***pyspark exercise #1***: Read in space-separated files **cust.da**t and **purchase.dat**. Each has a header line. For customers we know a unique numeric id, an indicator for female, and "origin" which is 1 or 2 indicating the original company before two companies merged. For purchase, we have one record per purchase containing id, store number, dollar amount, and an indicator for use of a debit card. The current goal is to find the customer(s) with the single highest purchase amount and return the total purchases, gender, stores used, and percent debit for each of those customers. Use python variables as the intermediary between the two data sets, since we have not learned any "join" type operations yet.