Word Count

Counting the number of occurances of words in a text is one of the most popular first eercises when learning Map-Reduce Programming. It is the equivalent to Hello World! in regular programming.

We will do it two way, a simpler way where sorting is done after the RDD is collected, and a more sparky way, where the sorting is also done using an RDD.

Read text into an RDD

Download data file from S3

In [2]:

```
%%time
import urllib
data_dir='../../Data'
filename='Moby-Dick.txt'
f = urllib.urlretrieve ("https://mas-dse-open.s3.amazonaws.com/"+filename, data_dir+'/'+f
ilename)

# First, check that the text file is where we expect it to be
!ls -l $data_dir/$filename
```

```
-rw-r--r--  1 yoavfreund  staff  1257260 Apr 10 21:33 ../../Data/Moby-Dick.txt
CPU times: user 37.2 ms, sys: 35.2 ms, total: 72.4 ms
Wall time: 3.5 s
```

Define an RDD that will read the file

Note that, as execution is Lazy, this does not necessarily mean that actual reading of the file content has occured.

In [3]:

```
%%time
text_file = sc.textFile(data_dir+'/'+filename)
type(text_file)
```

```
CPU times: user 1.41 ms, sys: 1.47 ms, total: 2.88 ms
Wall time: 422 ms
```

Counting the words

- split line by spaces.
- map word to (word,1)
- count the number of occurances of each word.

In [4]:

```
%%time
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .filter(lambda x: x!='')\
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
type(counts)
```

```
CPU times: user 9.68 ms, sys: 3.99 ms, total: 13.7 ms
Wall time: 168 ms
```

Have a look a the execution plan

Note that the earliest node in the dependency graph is the file ../../Data/Moby-Dick.txt.

In [5]:

```
print counts.toDebugString()
```

```
(2) PythonRDD[6] at RDD at PythonRDD.scala:43 []
 |  MapPartitionsRDD[5] at mapPartitions at PythonRDD.scala:374 []
 |  ShuffledRDD[4] at partitionBy at NativeMethodAccessorImpl.java:-2 []
 +-(2) PairwiseRDD[3] at reduceByKey at <timed exec>:1 []
    |  PythonRDD[2] at reduceByKey at <timed exec>:1 []
    |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorIm
pl.java:-2 []
    |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.jav
a:-2 []
```

Count!

Finally we count the number of times each word has occured. Now, finally, the Lazy execution model finally performs some actual work, which takes a significant amount of time.

In [6]:

```
%%time
Count=counts.count()
Sum=counts.map(lambda (w,i): i).reduce(lambda x,y:x+y)
print 'Count=%f, sum=%f, mean=%f'%(Count,Sum,float(Sum)/Count)
```

```
Count=33782.000000, sum=215133.000000, mean=6.368273
CPU times: user 10.2 ms, sys: 4.53 ms, total: 14.7 ms
Wall time: 1.35 s
```

Finding the most common words

- counts: RDD with 33301 pairs of the form (word,count).
- Find the 2 most frequent words.
- **Method1:** collect and sort on head node.
- **Method2:** Pure Spark, collect only at the end.

Method1: **collect** and sort on head node

Collect the RDD into the driver node

- Collect can take significant time.

In [7]:

```
%%time
C=counts.collect()
print type(C)
```

```
<type 'list'>
CPU times: user 43.9 ms, sys: 7.95 ms, total: 51.9 ms
Wall time: 129 ms
```

Sort

- RDD collected into list in driver node.
- No longer using spark parallelism.
- Sort in python
- will not scale to very large documents.

In [8]:

```
C.sort(key=lambda x:x[1])
print 'most common words\n','\n'.join(['%s:\t%d'%c for c in C[-5:]])
print '\nLeast common words\n','\n'.join(['%s:\t%d'%c for c in C[:5]])
```

```
most common words
to:	4510
a:	4533
and:	5951
of:	6587
the:	13766

Least common words
funereal:	1
unscientific:	1
lime-stone,:	1
shouted,:	1
pitch-pot,:	1
```

Method2: Pure Spark, **collect** only at the end.

- Collect into the head node only the more frquent words.
- Requires multiple **stages**

Step 1 split, clean and map to **(word,1)**

In [10]:

```
%%time
RDD=text_file.flatMap(lambda x: x.split(' '))\
    .filter(lambda x: x!='')\
    .map(lambda word: (word,1))
```

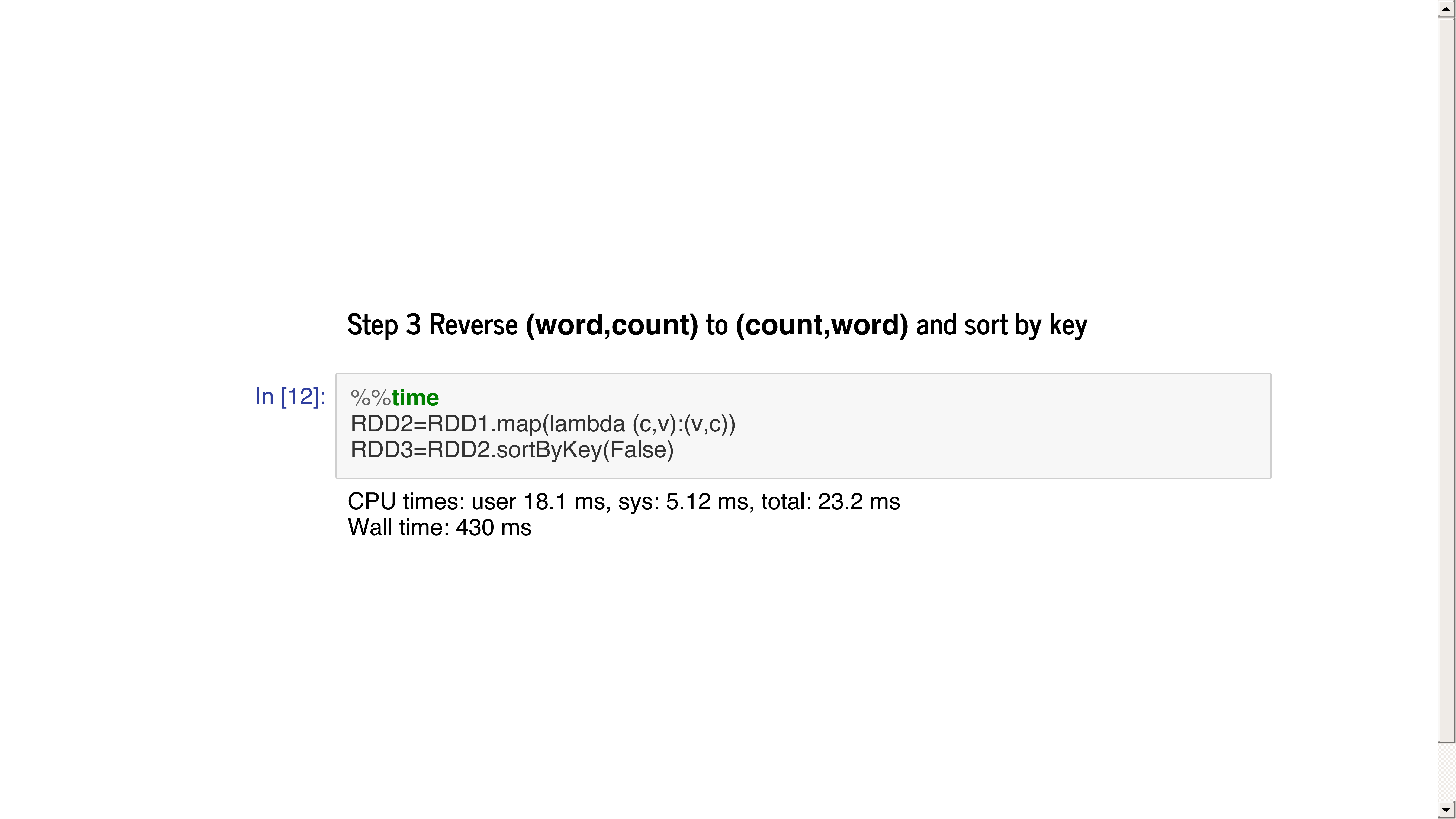
```
CPU times: user 43 µs, sys: 13 µs, total: 56 µs
Wall time: 51 µs
```

Step 2 Count occurances of each word.

In [11]:

```
%%time
RDD1=RDD.reduceByKey(lambda x,y:x+y)
```

```
CPU times: user 8.67 ms, sys: 2.94 ms, total: 11.6 ms
Wall time: 20.5 ms
```



Step 3 Reverse **(word,count)** to **(count,word)** and sort by key

In [12]:

```
%%time
RDD2=RDD1.map(lambda (c,v):(v,c))
RDD3=RDD2.sortByKey(False)
```

```
CPU times: user 18.1 ms, sys: 5.12 ms, total: 23.2 ms
Wall time: 430 ms
```

Full execution plan

We now have a complete plan to compute the most common words in the text. Nothing has been executed yet! Not even one one bye has been read from the file Moby-Dick.txt !

For more on execution plans and lineage see jace Klaskowski's blog

In [13]:

```
print 'RDD3:'
print RDD3.toDebugString()
```

```
RDD3:
(2) PythonRDD[19] at RDD at PythonRDD.scala:43 []
 |  MapPartitionsRDD[18] at mapPartitions at PythonRDD.scala:374 []
 |  ShuffledRDD[17] at partitionBy at NativeMethodAccessorImpl.java:-2 []
 +-(2) PairwiseRDD[16] at sortByKey at <timed exec>:2 []
    |  PythonRDD[15] at sortByKey at <timed exec>:2 []
    |  MapPartitionsRDD[12] at mapPartitions at PythonRDD.scala:374 []
    |  ShuffledRDD[11] at partitionBy at NativeMethodAccessorImpl.java:-2 []
    +-(2) PairwiseRDD[10] at reduceByKey at <timed exec>:1 []
       |  PythonRDD[9] at reduceByKey at <timed exec>:1 []
       |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorI
mpl.java:-2 []
       |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.ja
va:-2 []
```

Step 4 Take the top 5 words. only now the computer executes the plan!

In [14]:

```
%%time
C=RDD3.take(5)
print 'most common words\n','\n'.join(['%d:\t%s'%c for c in C])
```

```
most common words
13766:	the
6587:	of
5951:	and
4533:	a
4510:	to
CPU times: user 11.7 ms, sys: 3.73 ms, total: 15.5 ms
Wall time: 171 ms
```