Word Count

DSC 232R

1 Word Count

Counting the number of occurrences of words in a text is a popular first exercise using MapReduce.

1.1 The Task

Input: A text file consisting of words separated by spaces.

Output: A list of words and their counts, sorted from the most to the least common.

We will use the book “Moby Dick” as our input.

1.3 Define an RDD that will read the file

- Execution of read is **lazy**.
- File has been opened.
- Reading starts when **stage** is executed.
- What is a stage — explained later

```
In [18]: %%time
         text_file = sc.textFile(data_dir+'/'+filename)
         type(text_file)

         CPU times: user 1.7 ms, sys: 1.47 ms, total: 3.16 ms
         Wall time: 23.3 ms

Out[18]: pyspark.rdd.RDD
```

1.4 Steps for counting the words

- Split line by spaces.
- Map `word` to `(word, 1)`.
- Count the number of occurrences of each word.

```
In [9]: %%time
        ## text_file is an RDD of lines
        words =      text_file.flatMap(lambda line: line.split())
        not_empty = words.filter(lambda x: x!='')
        key_values= not_empty.map(lambda word: (word, 1))
        counts=     key_values.reduceByKey(lambda a, b: a + b)

CPU times: user 6.63 ms, sys: 3.42 ms, total: 10.1 ms
Wall time: 90.3 ms
```

1.5 The execution plan

In the last cell we defined the execution plan, but we have not started to execute it.

- Preparing the plan took ~100ms, which is a non-trivial amount of time,
- But much less than the time it will take to execute it.
- Let’s have a look at the execution plan.

1.5.1 Understanding the details

To see which step in the plan corresponds to which RDD we print out the execution plan for each of the RDDs.

Note that the execution plan for `words`, `not_empty`, and `key_values` are all the same.

Execution plan	RDD	Comments
`(2) PythonRDD[6] at RDD at PythonRDD.scala:48 []`	counts	Final RDD
`_/__MapPartitionsRDD[5] at mapPartitions at PythonRDD.scala:436 []`	---"---	
`_/__ShuffledRDD[4] at partitionBy at NativeMethodAccessorImpl.java:0 [`	---"---	RDD is partitioned by key
`_+-(2) PairwiseRDD[3] at reduceByKey at <timed exec>:4 []`	---"---	Perform mapByKey
`____/__PythonRDD[2] at reduceByKey at <timed exec>:4 []`	words, not_empty, key_values	The result of partitioning into words
		removing empties, and making into (word,1) pairs
`____/__../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at Nat`	text_file	The partitioned text
`____/__../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMeth`	---"---	The text source

Accessible Table of Page 8

Execution plan	RDD	Comments
(2)_PythonRDD[6] at RDD at PythonRDD.scala:48 [ ]	**counts**	Final RDD
_/__MapPartitionsRDD[5] at mapPartitions at PythonRDD.scala:436 [ ]	---"---	
__/___ShuffledRDD[4] at partitionBy at NativeMethodAccessorImpl.java:0 [	---"---	RDD is partitioned by key
__+-(2)_PairwiseRDD[3] at reduceByKey at <timed exec>:4 [ ]	---"---	Perform mapByKey
___/__PythonRDD[2] at reduceByKey at <time exec>:4 [ ]	**words**, **not_empty**, **key_values**	The result of partitioning into words
		Removing empties, and making into (word,1) pairs
___/__../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at Nat	**text_file**	The partitioned text
___/__../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMeth	---"---	The text source

1.6 Execution

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

In [15]:
```
%%time
## Run #1
Count=counts.count()  # Count = the number of different words
Sum=counts.map(lambda x:x[1]).reduce(lambda x,y:x+y) #
```

```
CPU times: user 9.5 ms, sys: 6.93 ms, total: 16.4 ms
Wall time: 795 ms
```

In [16]:
```
print('Different words=%5.0f, total words=%6.0f, mean no. occurances per word=%4.2f'%(Count,Sum,float(Sum)/Count))
```

```
Different words=    9, total words=     9, mean no. occurances per word=1.00
```

1.7 Summary

- This was our first real PySpark program, hurray!

1.7.0.1 Some things you learned:

1. An RDD is a distributed immutable array. The core data structure of Spark is an RDD.
2. You cannot operate on an RDD directly. Only through **Transformations** and **Actions**.
3. **Transformations** transform an RDD into another RDD.
4. **Actions** output their results to your jupyter notebook.
5. Computations done after actions do not use Spark resources, only resources on the host of the jupyter notebook.

1.7.0.1 Lazy Execution

1. RDD operations are added to an **Execution Plan**.
2. The plan is executed when a result is needed.
3. Explicit and implicit caching cause intermediate results to be saved.

Next: Finding the most common words.

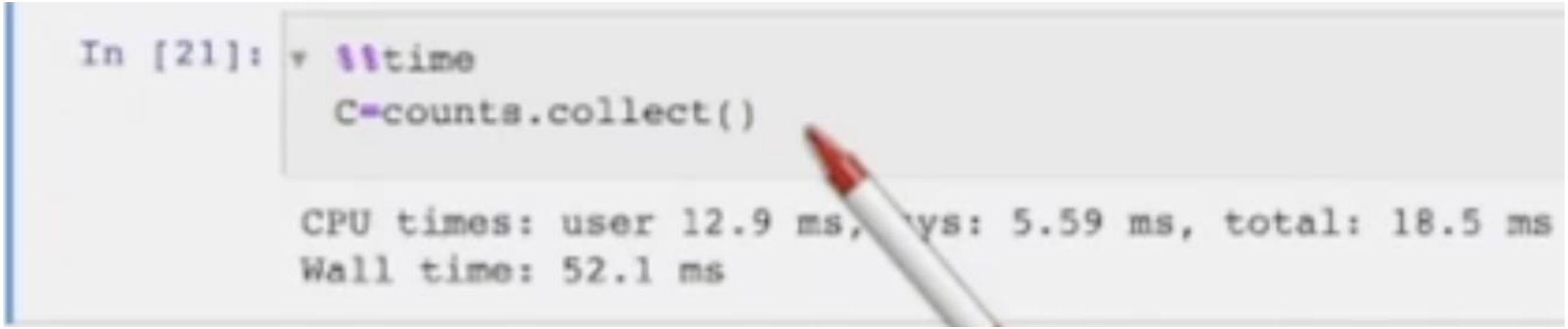
2 Finding the most common words

- `counts`: RDD with 33301 pairs of the form `(word,count)`.
- Find the 5 most frequent words.
- **Method1:** `collect` and sort on head node.
- **Method2:** Pure Spark, `collect` only at the end.

2.1 Method1: `collect` and sort on head node

2.1.1 Collect the RDD into the driver node

- Collect can take significant time.



Note: Pen is covering “sys.”

2.1.2 Sort

- RDD collected into list in driver node.
- No longer using Spark parallelism.
- Sort in Python.
- Will not scale well to very large documents.

2.2 Method2: Pure Spark, `collect` only at the end

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

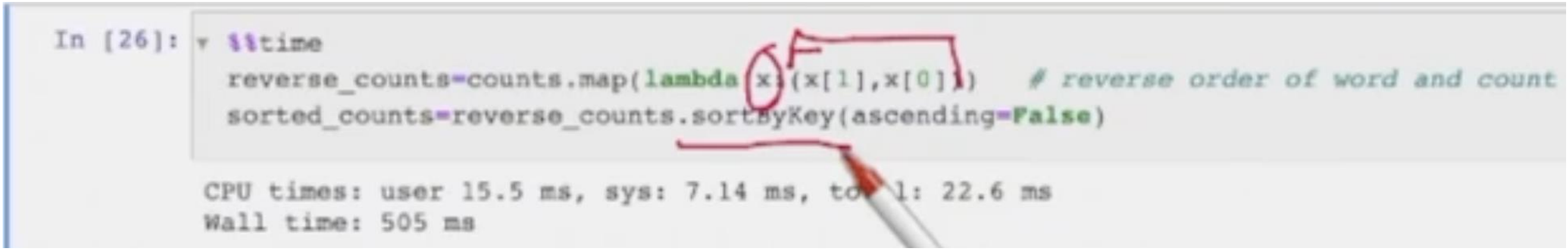
2.2.1 Step 1 split, clean and map to `(word,1)`

```
In [24]: %%time
         word_pairs=text_file.flatMap(lambda x: x.split(' '))\
             .filter(lambda x: x!='')\
             .map(lambda word: (word,1))

         CPU times: user 713 µs, sys: 729 µs, total: 1.44 ms
         Wall time: 900 µs
```

2.2.2 Step 2 Count occurrences of each **word**

```
In [25]: %%time
         counts=word_pairs.reduceByKey(lambda x,y:x+y)

         CPU times: user 5.67 ms, sys: 2.91 ms, total: 8.59 ms
         Wall time: 21.6 ms
```

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



Note: Pen is covering “total.” Text underneath red ink is (lambda x: x[1], x[0])).

2.2.4 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 a single byte has been read from the file `Moby-Dick.txt`!

For more on execution plans and lineage, see Jace Klaskowski's blog.

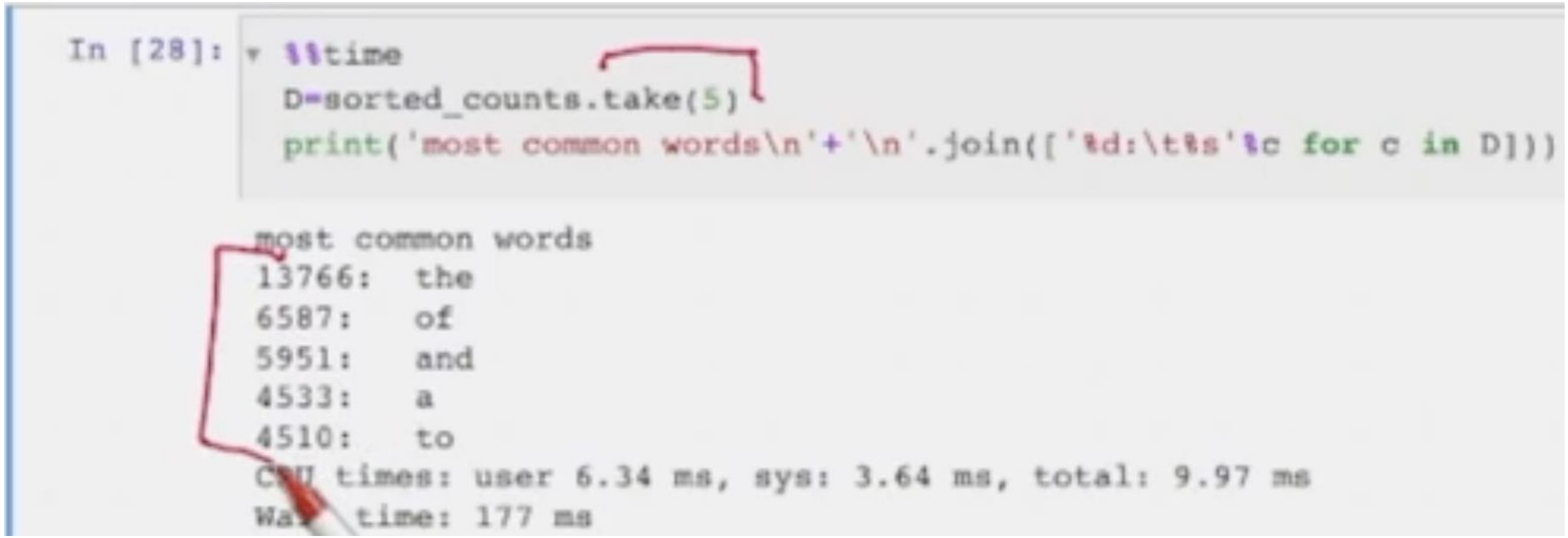
sorted_counts:

Execution plan	RDD
(2) PythonRDD[20] at RDD at PythonRDD.scala:48 []	sorted_counts
/__MapPartitionsRDD[19] at mapPartitions at PythonRDD.scala:436 []	"---
/__ShuffledRDD[18] at partitionBy at NativeMethodAccessorImpl.java:0	"---
+-(2) PairwiseRDD[17] at sortByKey at <timed exec>:2 []	"---
/__PythonRDD[16] at sortByKey at <timed exec>:2 []	"counts, reverse_counts"
/__MapPartitionsRDD[13] at mapPartitions at PythonRDD.scala:436 []	"---
/__ShuffledRDD[12] at partitionBy at NativeMethodAccessorImpl.java	"---
+-(2) PairwiseRDD[11] at reduceByKey at <timed exec>:1 []	"---
/__PythonRDD[10] at reduceByKey at <timed exec>:1 []	word_pairs
/__../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at	"---
/__../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeM	"---

Accessible Table of Page 23

Execution plan	RDD
(2)_PythonRDD[20] at RDD at PythonRDD.scala:48 [ ]	sorted_counts
__/___MapPartitionsRDD[19] at mapPartitions at PythonRDD.scala:436 [ ]	---"---
_/__ShuffledRDD[18] at partitionBy at NativeMethodAccessorImpl.java:0	---"---
_+-(2)_PairwiseRDD[17] at sortByKey at <timed exec>:2 [ ]	---"---
___/__PythonRDD[16] at sortByKey at <timed exec>:2 [ ]	** counts, reverse_counts**
___/__MapPartitionsRDD[13] at MapPartitions at PythonRDD.scala:436 []	---"---
___/__ShuffledRDD[12] at partitionBy at NativeMethodAccessorImpl.java	---"---
___/+-(2)_PairwiseRDD[11] at reduceByKey at <timed exec>:1 [ ]	---"---
_____/__PythonRDD[10] at reduceByKey at <timed exec>:1 [ ]	Word_pairs
_____/__../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at	---"---
_____/__../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeM	---"---

2.2.5 Step 4 Take the top 5 words



```
In [28]: %%time
         D=sorted_counts.take(5)
         print('most common words\n'+'\n'.join(['%d:\t%s'%c for c in D]))

most common words
13766:   the
6587:    of
5951:    and
4533:    a
4510:    to
CPU times: user 6.34 ms, sys: 3.64 ms, total: 9.97 ms
Wall time: 177 ms
```

Note: Pen is covering “CPU” and “Wait.”

2.3 Summary

We showed two ways for finding the most common words:

1. Collecting and sorting at the head node. – Does not scale.
2. Using RDDs for everything, `take(5)` moves result to head node at the end.

See you next time!