5.Word Count

October 11, 2018

1 Word Count

Counting the number of occurances of words in a text is a popular first exercise using map-reduce.

1.1 The Task

Input: A text file consisisting 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.

```
In [1]: #start the SparkContext
        from pyspark import SparkContext
        sc=SparkContext(master="local[4]")

        # set import path
        import sys
        sys.path.append('../../Utils/')
        #from plan2table import plan2table
```

```
In [34]: def pretty_print_plan(rdd):
             for x in rdd.toDebugString().decode().split('\n'):
                 print(x)
```

1.2 Download data file from S3

```
In [11]: %%time

         ##If this cell fails, download the file from https://mas-dse-open.s3.amazonaws.com/Mo
         # and put it in the '../../Data/ directory
         import requests
         data_dir='../../Data'
         filename='Moby-Dick.txt'
         url = "https://mas-dse-open.s3.amazonaws.com/"+filename
         local_path = data_dir+'/'+filename
         !mkdir -p {data_dir}
         # Copy URL content to local_path
         r = requests.get(url, allow_redirects=True)
         open(local_path, 'wb').write(r.content)
```

```
 # check that the text file is where we expect it to be
 !ls -l $local_path


---------------------------------------------------------------------------

gaierror                                  Traceback (most recent call last)

/opt/conda/lib/python3.6/site-packages/urllib3/connection.py in _new_conn(self)
140             conn = connection.create_connection(
--> 141                 (self.host, self.port), self.timeout, **extra_kw)
142


/opt/conda/lib/python3.6/site-packages/urllib3/util/connection.py in create_connection
 59
---> 60     for res in socket.getaddrinfo(host, port, family, socket.SOCK_STREAM):
 61         af, socktype, proto, canonname, sa = res


/opt/conda/lib/python3.6/socket.py in getaddrinfo(host, port, family, type, proto, flag
744     addrlist = []
--> 745     for res in _socket.getaddrinfo(host, port, family, type, proto, flags):
746         af, socktype, proto, canonname, sa = res


gaierror: [Errno -3] Temporary failure in name resolution


During handling of the above exception, another exception occurred:


NewConnectionError                        Traceback (most recent call last)

/opt/conda/lib/python3.6/site-packages/urllib3/connectionpool.py in urlopen(self, metho
600                                                       body=body, headers=headers,
--> 601                                                   chunked=chunked)
602


/opt/conda/lib/python3.6/site-packages/urllib3/connectionpool.py in _make_request(self
345         try:
--> 346             self._validate_conn(conn)
347         except (SocketTimeout, BaseSSLError) as e:


/opt/conda/lib/python3.6/site-packages/urllib3/connectionpool.py in _validate_conn(sel
```

```
    849         if not getattr(conn, 'sock', None):  # AppEngine might not have  `.sock`
--> 850             conn.connect()
    851
```

```
    /opt/conda/lib/python3.6/site-packages/urllib3/connection.py in connect(self)
    283         # Add certificate verification
--> 284         conn = self._new_conn()
    285
```

```
    /opt/conda/lib/python3.6/site-packages/urllib3/connection.py in _new_conn(self)
    149             raise NewConnectionError(
--> 150                 self, "Failed to establish a new connection: %s" % e)
    151
```

```
    NewConnectionError: <urllib3.connection.VerifiedHTTPSConnection object at 0x7f932d1b9ba
```

During handling of the above exception, another exception occurred:

```
    MaxRetryError                             Traceback (most recent call last)

    /opt/conda/lib/python3.6/site-packages/requests/adapters.py in send(self, request, stre
    439                     retries=self.max_retries,
--> 440                     timeout=timeout
    441                 )
```

```
    /opt/conda/lib/python3.6/site-packages/urllib3/connectionpool.py in urlopen(self, metho
    638             retries = retries.increment(method, url, error=e, _pool=self,
--> 639                                         _stacktrace=sys.exc_info()[2])
    640             retries.sleep()
```

```
    /opt/conda/lib/python3.6/site-packages/urllib3/util/retry.py in increment(self, method
    387         if new_retry.is_exhausted():
--> 388             raise MaxRetryError(_pool, url, error or ResponseError(cause))
    389
```

```
    MaxRetryError: HTTPSConnectionPool(host='mas-dse-open.s3.amazonaws.com', port=443): Max
```

During handling of the above exception, another exception occurred:

```
ConnectionError                           Traceback (most recent call last)

<timed exec> in <module>()


/opt/conda/lib/python3.6/site-packages/requests/api.py in get(url, params, **kwargs)
 70
 71     kwargs.setdefault('allow_redirects', True)
---> 72     return request('get', url, params=params, **kwargs)
 73
 74


/opt/conda/lib/python3.6/site-packages/requests/api.py in request(method, url, **kwargs
 56     # cases, and look like a memory leak in others.
 57     with sessions.Session() as session:
---> 58         return session.request(method=method, url=url, **kwargs)
 59
 60


/opt/conda/lib/python3.6/site-packages/requests/sessions.py in request(self, method, ur
506         }
507         send_kwargs.update(settings)
--> 508         resp = self.send(prep, **send_kwargs)
509
510         return resp


/opt/conda/lib/python3.6/site-packages/requests/sessions.py in send(self, request, **kw
616
617         # Send the request
--> 618         r = adapter.send(request, **kwargs)
619
620         # Total elapsed time of the request (approximately)


/opt/conda/lib/python3.6/site-packages/requests/adapters.py in send(self, request, stre
506                 raise SSLError(e, request=request)
507
--> 508             raise ConnectionError(e, request=request)
509
510         except ClosedPoolError as e:


ConnectionError: HTTPSConnectionPool(host='mas-dse-open.s3.amazonaws.com', port=443): M
```

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.

```
In [3]: %%time
        text_file = sc.textFile(data_dir+'/'+filename)
        type(text_file)

CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 1.6 s
```

1.4 Steps for counting the words

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

```
In [4]: %%time
        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 20 ms, sys: 10 ms, total: 30 ms
Wall time: 318 ms
```

1.4.1 flatMap()

Note the line:

```
words =     text_file.flatMap(lambda line: line.split(" "))
```

Why are we using `flatMap`, rather than `map`?

The reason is that the operation `line.split(" ")` generates a **list** of strings, so had we used `map` the result would be an RDD of lists of words. Not an RDD of words.

The difference between `map` and `flatMap` is that the second expects to get a list as the result from the map and it **concatenates** the lists to form the RDD.

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.
- Lets have a look a 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.

```
In [35]: pretty_print_plan(text_file)

(2) ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0
 |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []
```

```
In [36]: pretty_print_plan(words)

(2) PythonRDD[9] at RDD at PythonRDD.scala:48 []
 |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0
 |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []
```

```
In [37]: pretty_print_plan(not_empty)

(2) PythonRDD[8] at RDD at PythonRDD.scala:48 []
 |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0
 |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []
```

```
In [38]: pretty_print_plan(key_values)

(2) PythonRDD[7] at RDD at PythonRDD.scala:48 []
 |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0
 |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []
```

```
In [39]: pretty_print_plan(counts)

(2) PythonRDD[23] at RDD at PythonRDD.scala:48 []
 |  MapPartitionsRDD[22] at mapPartitions at PythonRDD.scala:122 []
 |  ShuffledRDD[21] at partitionBy at NativeMethodAccessorImpl.java:0 []
 +-(2) PairwiseRDD[20] at reduceByKey at <timed exec>:1 []
    |  PythonRDD[19] at reduceByKey at <timed exec>:1 []
    |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.jav
    |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []
```

Execution plan	RDD	Comments
`(2)_PythonRDD[6] at RDD at PythonRDD.scala:48 []`	**counts**	Final RDD
`_/__MapPartitionsRDD[5] at mapPartitions at PythonRDD.scala:436 []`	—"—	

Execution plan	RDD	Comments
`_/__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

1.6 Execution

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 [11]: %%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) #
```

```
        print('Different words=%5.0f, total words=%6.0f, mean no. occurances per word=%4.2f'%
```

```
Different words=33781, total words=215133, mean no. occurances per word=6.37
CPU times: user 10.5 ms, sys: 7.39 ms, total: 17.9 ms
Wall time: 1.04 s
```

1.6.1 Amortization

When the same commands are performed repeatedly on the same data, the execution time tends to decrease in later executions.

The cells below are identical to the one above, with one exception at `Run #3`

Observe that `Run #2` take much less time that `Run #1`. Even though no `cache()` was explicitly requested. The reason is that Spark caches (or materializes) `key_values`, before executing `reduceByKey()` because performng reduceByKey requires a shuffle, and a shuffle requires that the input RDD is materialized. In other words, sometime caching happens even if the programmer did not ask for it.

```
In [5]: %%time
        ## Run #2
        Count=counts.count()
        Sum=counts.map(lambda x:x[1]).reduce(lambda x,y:x+y)
        print('Different words=%5.0f, total words=%6.0f, mean no. occurances per word=%4.2f'%(
```

```
Different words=33781, total words=215133, mean no. occurances per word=6.37
CPU times: user 20 ms, sys: 10 ms, total: 30 ms
Wall time: 3.9 s
```

Explicit Caching In `Run #3` we explicitly ask for `counts` to be cached. This will reduce the execution time in the following run by a little bit, but not by much.

```
In [6]: %%time
        ## Run #3, cache
        Count=counts.cache().count()
        Sum=counts.map(lambda x:x[1]).reduce(lambda x,y:x+y)
        print('Different words=%5.0f, total words=%6.0f, mean no. occurances per word=%4.2f'%(
```

```
Different words=33781, total words=215133, mean no. occurances per word=6.37
CPU times: user 20 ms, sys: 0 ns, total: 20 ms
Wall time: 597 ms
```

```
In [7]: %%time
        #Run #4
        Count=counts.count()
        Sum=counts.map(lambda x:x[1]).reduce(lambda x,y:x+y)
        print('Different words=%5.0f, total words=%6.0f, mean no. occurances per word=%4.2f'%(
```

```
Different words=33781, total words=215133, mean no. occurances per word=6.37
CPU times: user 20 ms, sys: 10 ms, total: 30 ms
Wall time: 432 ms
```

```
In [8]: %%time
        #Run #5
        Count=counts.count()
        Sum=counts.map(lambda x:x[1]).reduce(lambda x,y:x+y)
        print('Different words=%5.0f, total words=%6.0f, mean no. occurances per word=%4.2f'%(
```

```
Different words=33781, total words=215133, mean no. occurances per word=6.37
CPU times: user 20 ms, sys: 0 ns, total: 20 ms
Wall time: 307 ms
```

1.7 Summary

This was our first real pyspark program, hurray!

Some things you learned:

1) An RDD is a distributed immutable array.
   It is 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 on the head node.

5) After the action is done, you are using just the head node, not the workers.

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 internediate 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.

```
In [9]: %%time
        C=counts.collect()

CPU times: user 50 ms, sys: 0 ns, total: 50 ms
Wall time: 200 ms
```

2.1.2 Sort

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

```
In [10]: %%time
         C.sort(key=lambda x:x[1])
         print('most common words\n'+'\n'.join(['%s:\t%d'%c for c in reversed(C[-5:])]))

most common words
the:        13766
of:        6587
and:        5951
a:        4533
to:        4510
CPU times: user 10 ms, sys: 0 ns, total: 10 ms
Wall time: 18.1 ms
```

2.1.3 Compute the mean number of occurances per word.

```
In [11]: Count2=len(C)
         Sum2=sum([i for w,i in C])
         print('count2=%f, sum2=%f, mean2=%f'%(Count2,Sum2,float(Sum2)/Count2))

count2=33781.000000, sum2=215133.000000, mean2=6.368462
```

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

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

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

```
In [12]: %%time
         word_pairs=text_file.flatMap(lambda x: x.split(' '))\
             .filter(lambda x: x!='')\
             .map(lambda word: (word,1))

CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 66.3 ţs
```

2.2.2 Step 2 Count occurances of each word.

```
In [13]: %%time
         counts=word_pairs.reduceByKey(lambda x,y:x+y)

CPU times: user 10 ms, sys: 0 ns, total: 10 ms
Wall time: 37.5 ms
```

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

```
In [14]: %%time
         reverse_counts=counts.map(lambda x:(x[1],x[0]))   # reverse order of word and count
         sorted_counts=reverse_counts.sortByKey(ascending=False)

CPU times: user 30 ms, sys: 10 ms, total: 40 ms
Wall time: 1.49 s
```

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

```
In [40]: print('word_pairs:')
         pretty_print_plan(word_pairs)
         print('\ncounts:')
         pretty_print_plan(counts)
         print('\nreverse_counts:')
         pretty_print_plan(reverse_counts)
         print('\nsorted_counts:')
         pretty_print_plan(sorted_counts)

word_pairs:
(2) PythonRDD[18] at RDD at PythonRDD.scala:48 []
 |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.java:0
 |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []
```

```
counts:
(2) PythonRDD[23] at RDD at PythonRDD.scala:48 []
 |  MapPartitionsRDD[22] at mapPartitions at PythonRDD.scala:122 []
 |  ShuffledRDD[21] at partitionBy at NativeMethodAccessorImpl.java:0 []
 +-(2) PairwiseRDD[20] at reduceByKey at <timed exec>:1 []
    |  PythonRDD[19] at reduceByKey at <timed exec>:1 []
    |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.jav
    |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []

reverse_counts:
(2) PythonRDD[30] at RDD at PythonRDD.scala:48 []
 |  MapPartitionsRDD[22] at mapPartitions at PythonRDD.scala:122 []
 |  ShuffledRDD[21] at partitionBy at NativeMethodAccessorImpl.java:0 []
 +-(2) PairwiseRDD[20] at reduceByKey at <timed exec>:1 []
    |  PythonRDD[19] at reduceByKey at <timed exec>:1 []
    |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl.jav
    |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0 []

sorted_counts:
(2) PythonRDD[31] at RDD at PythonRDD.scala:48 []
 |  MapPartitionsRDD[29] at mapPartitions at PythonRDD.scala:122 []
 |  ShuffledRDD[28] at partitionBy at NativeMethodAccessorImpl.java:0 []
 +-(2) PairwiseRDD[27] at sortByKey at <timed exec>:2 []
    |  PythonRDD[26] at sortByKey at <timed exec>:2 []
    |  MapPartitionsRDD[22] at mapPartitions at PythonRDD.scala:122 []
    |  ShuffledRDD[21] at partitionBy at NativeMethodAccessorImpl.java:0 []
    +-(2) PairwiseRDD[20] at reduceByKey at <timed exec>:1 []
       |  PythonRDD[19] at reduceByKey at <timed exec>:1 []
       |  ../../Data/Moby-Dick.txt MapPartitionsRDD[1] at textFile at NativeMethodAccessorImpl
       |  ../../Data/Moby-Dick.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.java:0
```

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 []	—"—

Execution plan	RDD
_____/__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 [15]: %%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 10 ms, sys: 0 ns, total: 10 ms
Wall time: 398 ms
```

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 to the end.

See you next time!