Exercise 5

Get started with Apache Spark and Python

Prior Knowledge
Unix Command Line Shell
Simple Python

Learning Objectives
Understand the Spark system
Use the Spark Python shell to interactively work with data
Submit Spark jobs locally and using YARN
Write SparkSQL code in Python
WordCount!

Software Requirements
(see separate document for installation of these)

- Apache Spark 2.0.0
- Python 2.7.12
- Nano text editor, Sublime, PyCharms or other text editor

Part A. Spark Python Shell (pySpark)

1. We are going to do a wordcount against a set of books downloaded from Project Gutenberg. Wordcount is the definitive Big Data program (sort of Hello World for Big Data) and it is frankly embarrassing that we haven't done one yet.

2. Apache Spark has a useful Python shell, which we can use to interactively test and run code. Since we have our data in HDFS, *we need to ensure HDFS is running.* (Follow the instructions from the Hadoop lab).

3. Let's load some books into HDFS. In a terminal window (Ctrl-Alt-T)

   ```
   hadoop fs -mkdir -p /user/oxclo/books
   hadoop fs -put ~/datafiles/books/* /user/oxclo/books/
   ```

4. Now, change to the Spark directory:
   ```
   cd ~/spark
   ```

5. Now start the Spark Python command line tool – pyspark
   ```
   bin/pyspark
   ```

a. You should see a lot of log come up, ending in something like:

```
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-
defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
16/09/08 09:24:49 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
16/09/08 09:24:49 WARN Utils: Your hostname, oxclo resolves to a loopback
address: 127.0.1.1; using 172.16.64.199 instead (on interface ens33)
16/09/08 09:24:49 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to
another address
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.0.0
      /_/

Using Python version 2.7.12 (default, Jul  1 2016 15:12:24)
SparkSession available as 'spark'.
>>>
```

6. Now let's load some data. We already have a SparkContext object defined in the shell (in a program you need to define one, which we will see later)

7. Unfortunately some of the input is handled as Unicode by Python and we want to get rid of that. So let's start:

   ```
   import unicodedata
   ```

8. Then type (on one line):

   ```
   def u2a(u): return str(unicodedata.normalize('NFKD',u).
   encode('ascii','ignore'))
   ```

9. We also want to remove any non-alphanumeric characters:
   ```
   def strip(s): return ''.join(filter(str.isalpha, s))
   ```

10. Now we would like to load the books from HDFS. Now let's load some data. We already have a SparkContext object defined in the shell (in a program you need to define one, which we will see later)

    ```
    books = sc.textFile("hdfs://localhost:54310/user/oxclo/books/*")
    ```

11. Let's split the lines into words:

    ```
    split = books.flatMap(lambda line: line.split())
    ```

12. Now let's transform from Unicode to ascii
    ```
    asc = split.map(u2a)
    ```

13. And remove non-alpha characters

    ```
    stripped = asc.map(strip)
    ```

14. And we should put everything to lower case while we are cleaning it up (you should be able to figure this out!)

15. Finally we are ready to do the classic "WordCount" Map Reduce. We first create a simple <K,V> pair of <word, count>. In the map phase, the count is always 1, since we haven't yet reduced this.

```
numbered = lower.map(lambda word: (word, 1))
```

16. Next we reduce by adding the counts together for the same words:

```
wordcount = numbered.reduceByKey(lambda a,b: a+b)
```

17. Finally, we need to collect the results and print them. In Spark, they may be distributed across different RDD partitions on different machines, so the collect() method brings them together.

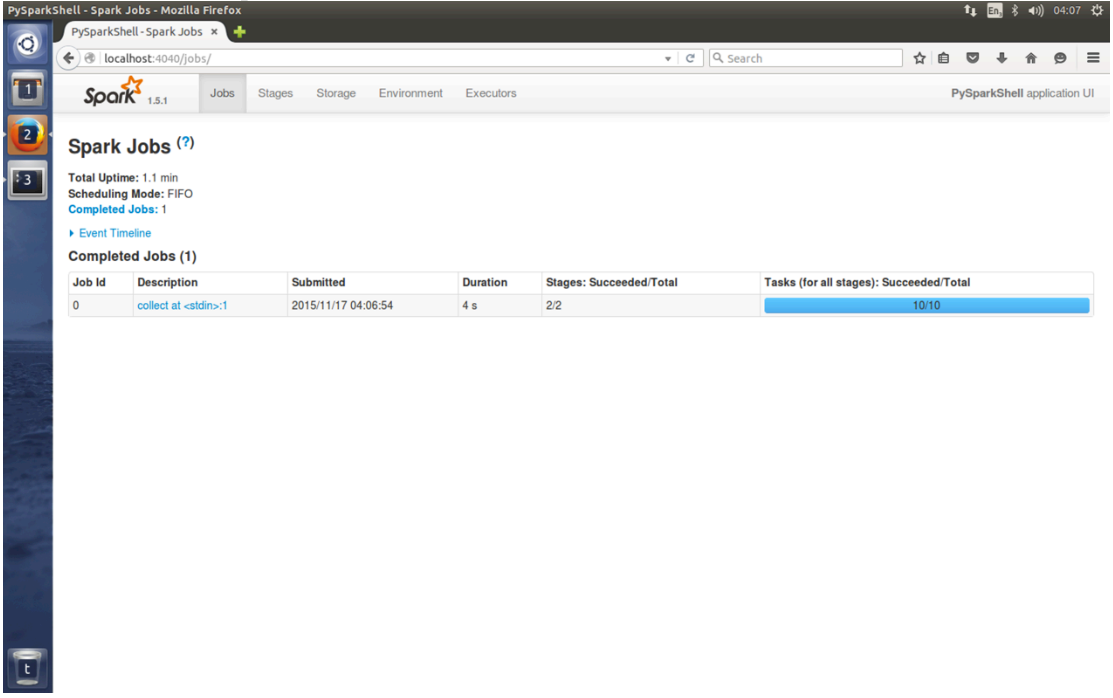
```
for k,v in wordcount.collect(): print k,v
```

18. You should see a lot of word counts go flying past.

19. Congratulations!

20. While the pyspark is still running browse to http://localhost:4040

21. You will see the Spark web console:



PySparkShell - Spark Jobs - Mozilla Firefox
04:07
PySparkShell - Spark Jobs
localhost:4040/jobs/
Search
Spark 1.5.1
Jobs
Stages
Storage
Environment
Executors
PySparkShell application UI

Spark Jobs (?)

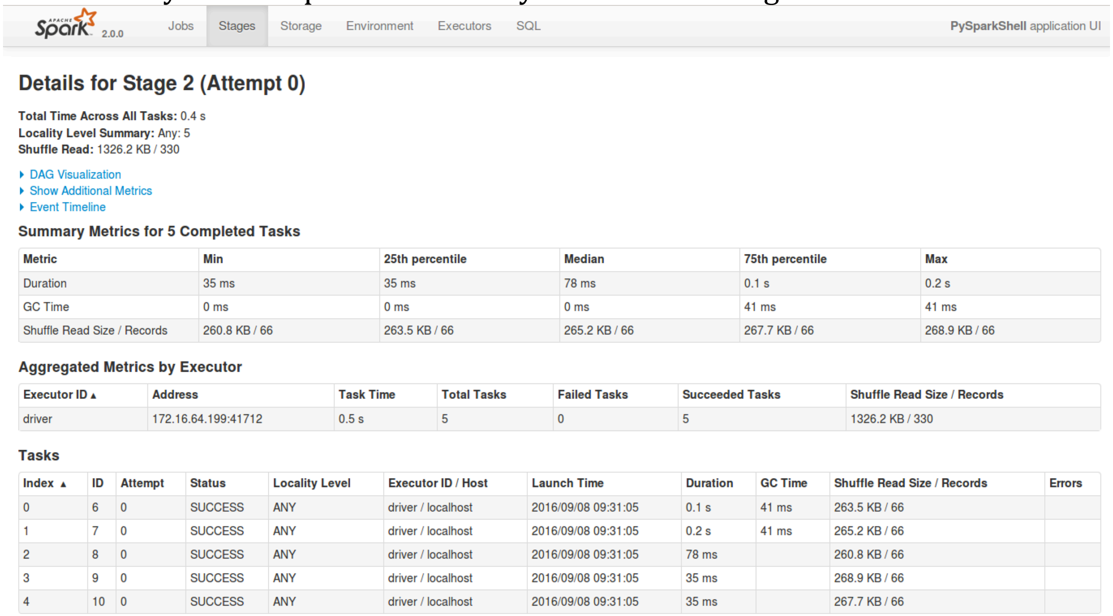
Total Uptime: 1.1 min
Scheduling Mode: FIFO
Completed Jobs: 1

▸ Event Timeline

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <stdin>:1	2015/11/17 04:06:54	4 s	2/2	10/10

22. Click on the blue link “collect at stdin”
    This shows you how Spark converted your code into stages:



Details for Stage 2 (Attempt 0)

Total Time Across All Tasks: 0.4 s
Locality Level Summary: Any: 5
Shuffle Read: 1326.2 KB / 330

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 5 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	35 ms	35 ms	78 ms	0.1 s	0.2 s
GC Time	0 ms	0 ms	0 ms	41 ms	41 ms
Shuffle Read Size / Records	260.8 KB / 66	263.5 KB / 66	265.2 KB / 66	267.7 KB / 66	268.9 KB / 66

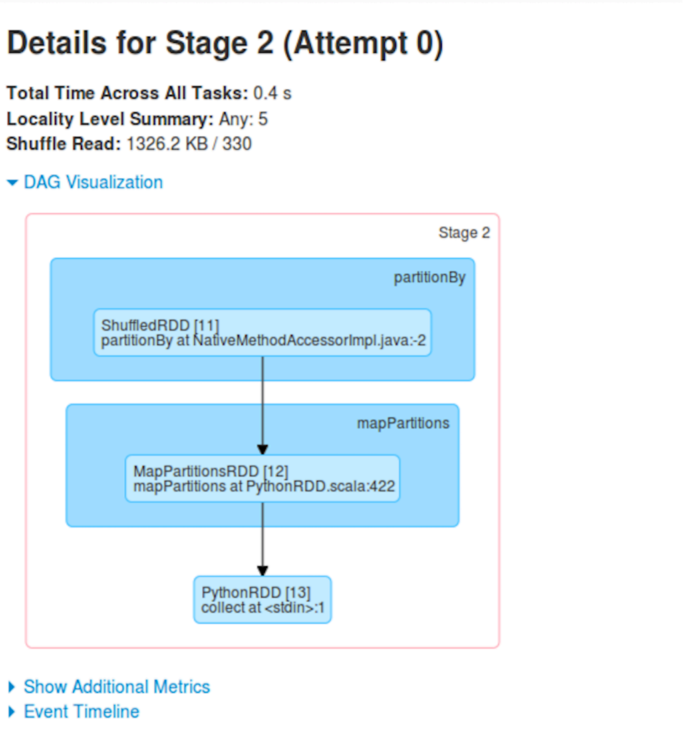
Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
driver	172.16.64.199:41712	0.5 s	5	0	5	1326.2 KB / 330

Tasks

Index ▴	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	6	0	SUCCESS	ANY	driver / localhost	2016/09/08 09:31:05	0.1 s	41 ms	263.5 KB / 66	
1	7	0	SUCCESS	ANY	driver / localhost	2016/09/08 09:31:05	0.2 s	41 ms	265.2 KB / 66	
2	8	0	SUCCESS	ANY	driver / localhost	2016/09/08 09:31:05	78 ms		260.8 KB / 66	
3	9	0	SUCCESS	ANY	driver / localhost	2016/09/08 09:31:05	35 ms		268.9 KB / 66	
4	10	0	SUCCESS	ANY	driver / localhost	2016/09/08 09:31:05	35 ms		267.7 KB / 66	

23. Expand the DAG visualization:

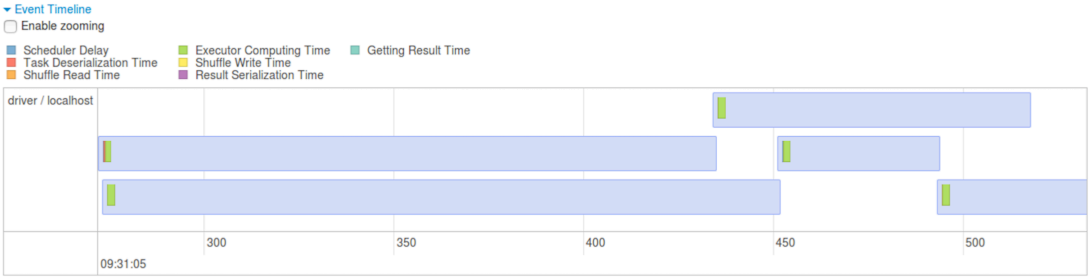


Details for Stage 2 (Attempt 0)

Total Time Across All Tasks: 0.4 s
Locality Level Summary: Any: 5
Shuffle Read: 1326.2 KB / 330

- DAG Visualization
- Show Additional Metrics
- Event Timeline

24. And the Event Timeline:



25. Quit the pyspark shell by typing

    ```
    quit()
    ```

26. Now let's run the same code as a "job" instead of interactively.

27. Make a directory for your spark python code:

    ```
    mkdir ~/pysp
    cd ~/pysp
    ```

28. From http://freo.me/oxclo-wc-py copy the code into a file wc.py

29. You will notice that there is a bunch of "setup" code that we didn't need in the pyspark command line tool. That is because pyspark assumes you want all this and does it for you.

30. Now configure the correct setup so Spark can find the Yarn system:

    ```
    export HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop/
    ```

31. We run jobs locally on a single node directly on Spark:
    The local[*] indicates to use as many threads as you have cores on your system:

    ```
    ~/spark/bin/spark-submit --master local[*] wc.py
    "hdfs://localhost:54310/user/oxclo/books/*"
    ```

32. You could also use Spark's own cluster manager, YARN or Apache Mesos as other options in a larger setup.

    In general, unless you are running mixed workloads with other Hadoop or Mesos workloads, I would always use Spark's cluster manager.

33. Congratulations, the lab is complete!