Hadoop Introduction

DS 730

In this activity, you will be testing your Hadoop setup on Hortonworks on Azure. You will be running a very simple MapReduce program on your Hadoop setup. You will also be writing solutions to a couple of MapReduce problems. Be careful when copying and pasting commands from this activity. Many of the commands are, unfortunately, quite long and won't fit on 1 line in this document. However, the commands are supposed to be written on 1 line.

Task 1: Test HDFS

1. Start your Azure Hortonworks virtual machine[1]. Note that you need to replace YourAzureIP with your actual IP address of your Hortonworks virtual machine. When it is running, go to http://YourAzureIP:4200. Whenever you go to this website, it will be called the **terminal** or **command line**. Log in using **maria_dev** as the username and password.
2. To ensure HDFS is up and running, type in:

   hdfs dfs -ls /

   It should show a bunch of folders like /app-logs, /apps, /ats and so on.
3. Our goal is to test our HDFS (Hadoop distributed file system) and make sure it is working correctly. We are also going to be stress testing it to see what kind of performance we can get out of our filesystem. You should have already read about HDFS, but if you have not, give this site a read before continuing: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html
4. In order to test our DFS speed, we will use a built-in example to write some random data to a file. To write data to our HDFS, enter the following command. The command may appear to stop on "map 0% reduce 0%" but do not cancel it. It will eventually run and finish.

```
hadoop jar
/usr/hdp/2.6.4.0-91/hadoop-mapreduce/hadoop-mapreduce-client-job
client-tests.jar TestDFSIO -write -nrFiles 10 -fileSize 50
```

Note: The test jar file is built into Hadoop. The **TestDFSIO** is the actual program we want to run. The **-write** tells the program to write the files. The **-nrFiles** is a switch to tell it how many files to write (in our case 10). The **-fileSize** is the number of megabytes per file (in our case 50). A nice

[1] The commands in this activity will not work on your local version of Hortonworks on VirtualBox if you created one. This is because the version of Hadoop is different. To make the commands work locally, navigate to /usr/hdp/ and make a note of the version number. Edit the command such that the version number is correct to make this work locally.

explanation on TestDFSIO can be found here:
https://support.pivotal.io/hc/en-us/articles/200864057-Running-DFSIO-mapreduce-benchmark-test

The program may take a couple of minutes to run depending on the speed of your computer.

5. Once you have run your command, copy the last few lines of the output starting with the following line of code and save that information to an output file called **output.txt** that you will eventually submit with your work:

```
18/07/06 01:26:13 INFO fs.TestDFSIO: ----- TestDFSIO ----- : write
```

6. Let's make sure the TestDFSIO actually worked. In order to check it, type:

```
hdfs dfs -ls /benchmarks/TestDFSIO/io_data
```

- You should see something like this:

```
-rw-r--r--   1 maria_dev hdfs   52428800 2018-07-06 01:26
/benchmarks/TestDFSIO/io_data/test_io_0
-rw-r--r--   1 maria_dev hdfs   52428800 2018-07-06 01:26
/benchmarks/TestDFSIO/io_data/test_io_1

…
```

7. We want to read in our data from the HDFS and make sure that is working properly. Our read test is done similar to the write test. Do not stop the command from running if it appears to lag. It will finish.

```
hadoop jar
/usr/hdp/2.6.4.0-91/hadoop-mapreduce/hadoop-mapreduce-client-job
client-tests.jar TestDFSIO -read -nrFiles 10 -fileSize 50
```

8. As before, take the 10 or so lines that start with the following line and save them to an output file called **output.txt**:

```
18/07/06 01:29:15 INFO fs.TestDFSIO: ----- TestDFSIO ----- : read
```

9. Once our benchmarks have been run, we can safely remove the random data that was created. To clean it up, type:

```
hadoop jar
/usr/hdp/2.6.4.0-91/hadoop-mapreduce/hadoop-mapreduce-client-job
client-tests.jar TestDFSIO -clean
```

10. To make sure our files are gone, type:

```
hdfs dfs -ls /benchmarks/TestDFSIO/io_data
```

and it will say there is no such file or directory.

Task 2: Test Hadoop's Sorting Speed

Now that we know we can write to and read from our HDFS, we want to know how quickly our Hadoop setup can sort a lot of data. First we want to generate a large amount of data. We will use a built-in jar file to do this for us.

1. Now we wish to use mapreduce to generate 2GB of random data and store it in a folder. Here is how we create our data:

```
hadoop jar
/usr/hdp/2.6.4.0-91/hadoop-mapreduce/hadoop-mapreduce-examples.j
ar teragen 20000000 /user/maria_dev/teradata
```

2. Now that our data is generated, we want to sort it. We are sorting 2GB of data using mapreduce. In order to sort them, type:

```
hadoop jar
/usr/hdp/2.6.4.0-91/hadoop-mapreduce/hadoop-mapreduce-examples.j
ar terasort /user/maria_dev/teradata /user/maria_dev/sorted-data
```

Our program runs using mapreduce and sorts all of the data. It will likely take several minutes to run.

3. You should copy everything starting with this line (you'll have to scroll up a bit) and store it in your output file called **output.txt**:

```
18/07/06 01:45:51 INFO mapreduce.Job: Counters: 49
```

4. You likely don't want to keep all of that data, so we will remove it from our HDFS. In order to remove a directory, we use the following to first remove all of the files that were created:

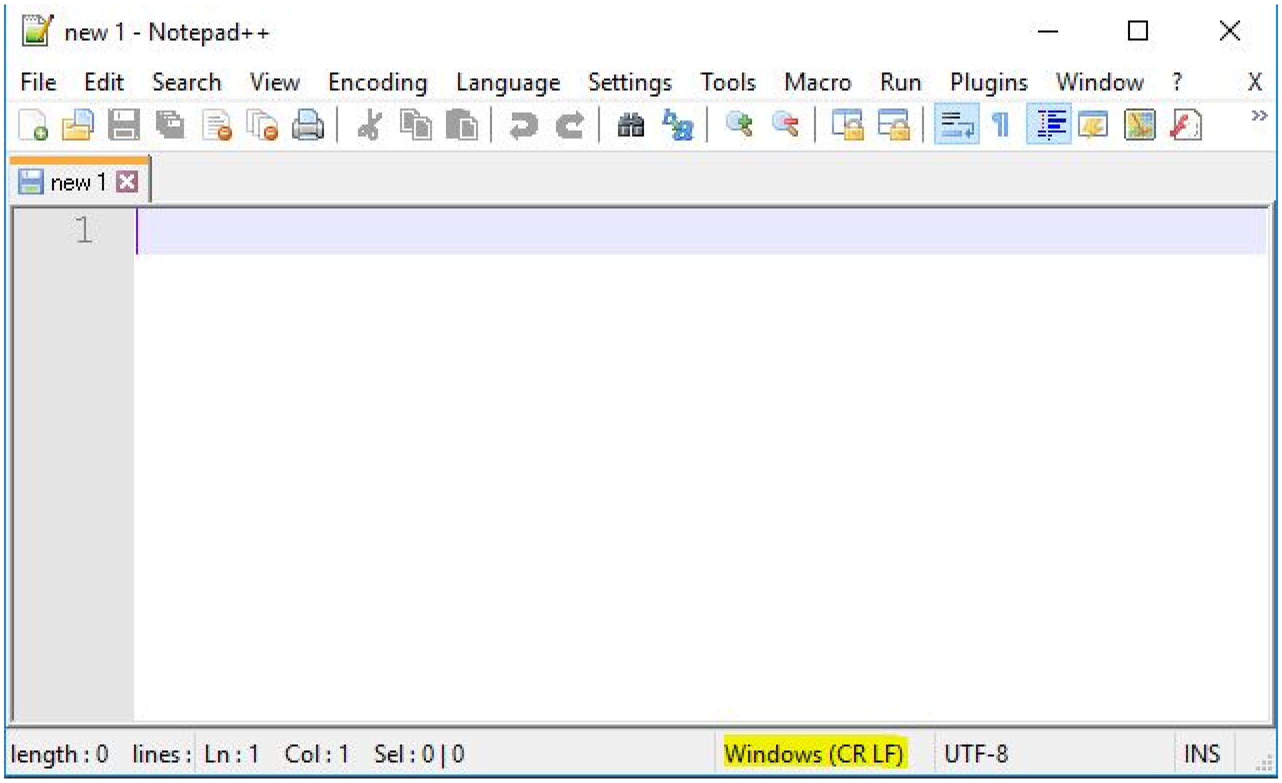
```
hdfs dfs -rm -r /user/maria_dev/teradata

hdfs dfs -rm -r /user/maria_dev/sorted-data
```

Task 3: Send Files Between Filesystem and HDFS

Hortonworks does a good job of abstracting away the different filesystems that exist in the virtual machine. The virtual machine that is running is a Linux machine. On that Linux machine, Hadoop, and more importantly for this activity, HDFS, is running. This means you have a filesystem installed on another filesystem. Understanding how this is done is beyond the scope of this course but you should be aware of what is going on. Log into Ambari by going to http://YourAzureIP:8080 and using **maria_dev** as the username and password. Click on the 9 squares icon in the upper right hand corner and click on **Files View**. When you view the files in Ambari, you are looking at the HDFS.

All of this is important because the files that you create must work in the Linux environment. If you are running a Linux machine or a Mac, you can ignore this task because your files will already be in the correct format. If you are running a Windows machine, you have to be careful about the types of files you upload to your HDFS. If you upload the wrong type of file, your code may not run at all even though the code is 100% correct. To address this issue, you must use a text editor that allows the creation of Unix newlines instead of Windows newlines. A very simple editor that you can use is called Notepad++. The following are instructions on how to install it, write some code and upload it to the HDFS.

1. Go to https://notepad-plus-plus.org/download and download the version most appropriate for your machine. Download that version and install it.
2. Open up Notepad++ and you will see something similar to this:


new 1 - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
new 1
length : 0 lines : Ln : 1 Col : 1 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

3. Note the **Windows (CR LF)** highlighted in the image. Right click on that part of the program and you will be given a couple of options. Select the **Unix (LF)** option. Doing so will only change the current document to use Unix linefeeds (LF). If you want this to be the default for all new documents, go to **Settings** → **Preferences** → **New Document** and select the **Unix (LF)** button. Click Close.
4. To avoid problems with spacing, never use tabs and only use spaces for indentation. To accomplish this automatically, go up to **Settings** → **Preferences** → **Language** and in the **Tab Settings** area, click on the **Replace by space** checkbox. Then click Close.

Task 4: Run a Mapreduce Job

As one might expect, there are many ways to access and process our big data. A goal of these activities is to teach you how to use many of the tools that are common in processing big data. Because these technologies change constantly, some industries will be on the cutting edge of using the newest thing where other industries will rely on the tried and true ways of tools that have worked in the past. If you understand the core principles of each of these tools, you will be able to use them in whatever environment your employer has for you. This task will be to run a mapreduce job using our word count Python code.

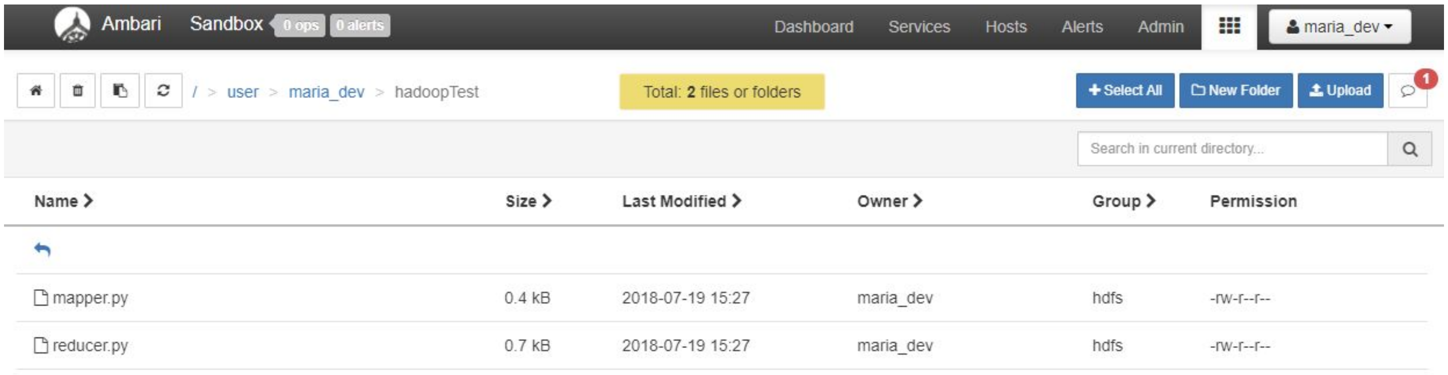
1. Open up Notepad++ and copy and mapper and reducer code from the end of this document. Make sure to copy them exactly as they are written. The indentation is important and your

code will break if even a space is missing. Also ensure that the spacing is consistent. Having one tab is different from having four spaces even though they look the same. Along with the code, create a simple text file called **input.txt** and store the following text in that file: **hello this is a test this is**

Note: Hadoop requires the Python code to be in a main function. The code will look slightly different from what we made during lecture.

Important:
Do *not* create the Python files using something other than Notepad++ unless you know you are using UNIX newlines. The short reason is because Windows uses a different set of characters for a newline than Ubuntu. Hadoop notices these extra characters and fails even though your actual code might be perfect.

2. Open up your Files View in Ambari and click on the **user** folder. Click on the **maria_dev** folder. Create a new folder called **hadoopTest**. Click on the **hadoopTest** folder and then click on the upload button. Upload the mapper.py and reducer.py files you created in the previous step to this folder. You should see something like this:



Name	Size	Last Modified	Owner	Group	Permission
mapper.py	0.4 kB	2018-07-19 15:27	maria_dev	hdfs	-rw-r--r--
reducer.py	0.7 kB	2018-07-19 15:27	maria_dev	hdfs	-rw-r--r--

3. You will notice near the top it shows **/ > user > maria_dev > hadoopTest**. This tells you what folder you are in. Click on the maria_dev link to "go up" a level in the folder hierarchy. Your newly created hadoopTest folder should be there. Create a new folder inside the hadoopTest directory called **wcinput**. Copy over your **input.txt** to that folder like you did in the previous step.
4. In order to see the nature of the filesystems involved here, we will transfer the mapper and reducer from the HDFS to the "local" filesystem on hortonworks. In order to do this, go to

command prompt at http://YourAzureIP:4200. Log in using **maria_dev** as the username and password. Type in the following command:

hdfs dfs -ls /user/maria_dev/hadoopTest

You should see the mapper.py and reducer.py file being listed. Type in **pwd** to see the current folder you are in. It should say /home/maria_dev. Type in **ls** (that's a lowercase L) and you will notice that no files are in your current folder.

5. Copy the files from HDFS to your local filesystem. In order to do this, use the following command:

   hdfs dfs -copyToLocal /user/maria_dev/hadoopTest/* .

   Note there is a space between the * and the period. The command should be pretty self-explanatory. It is saying to copy everything from the HDFS folder of /user/maria_dev/hadoopTest to the current folder (denoted with the period).

6. Now we are ready to run our first mapreduce program using Python code. To do this, we need to use the streaming feature of Hadoop. This lets us send a map function to Hadoop and a reduce function to Hadoop using Python instead of having to write it in Java. Run this command:

```
hadoop jar
/usr/hdp/2.6.4.0-91/hadoop-mapreduce/hadoop-streaming.jar
-files /home/maria_dev/mapper.py,/home/maria_dev/reducer.py
-input /user/maria_dev/hadoopTest/wcinput/*
-output /user/maria_dev/hadoopTest/wcoutput -mapper mapper.py
-reducer reducer.py
```

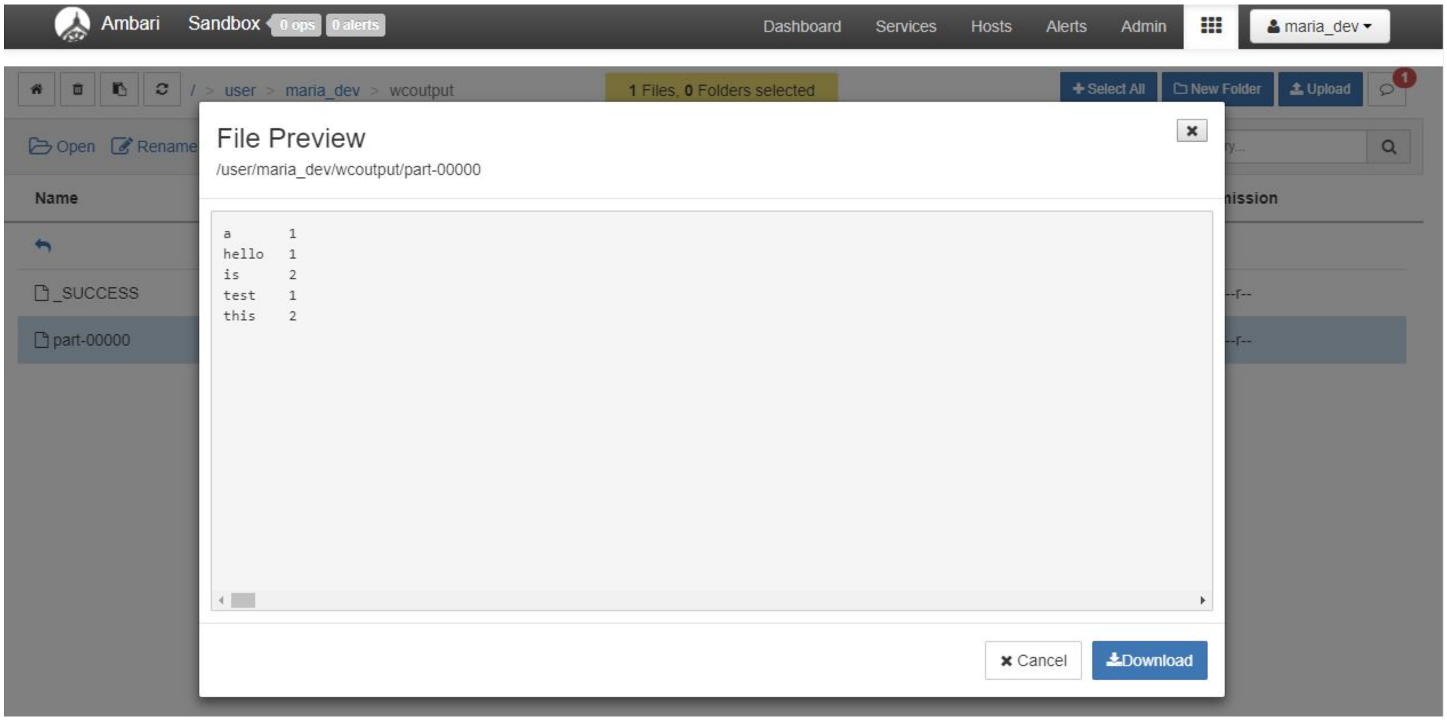
7. The command above is somewhat self-explanatory. The **-input** is telling Hadoop where the input files are located on the HDFS. In our case, there was only 1 file but there could have been many files in that folder. The **-output** is telling Hadoop where to put the output of the program on HDFS. The **-files** is telling Hadoop where the mapper and reducer file are located. In our case, they were located on the local filesystem folder of /home/maria_dev. The **-mapper**

and **-reducer** are simply telling Hadoop which file, that were read in during the -files part, is the mapper and which is the reducer.

8. Assuming everything went well, confirm this is the last line of the output (with different date and time of course):

```
18/07/20 16:39:35 INFO streaming.StreamJob: Output directory:
/user/maria_dev/wcoutput
```

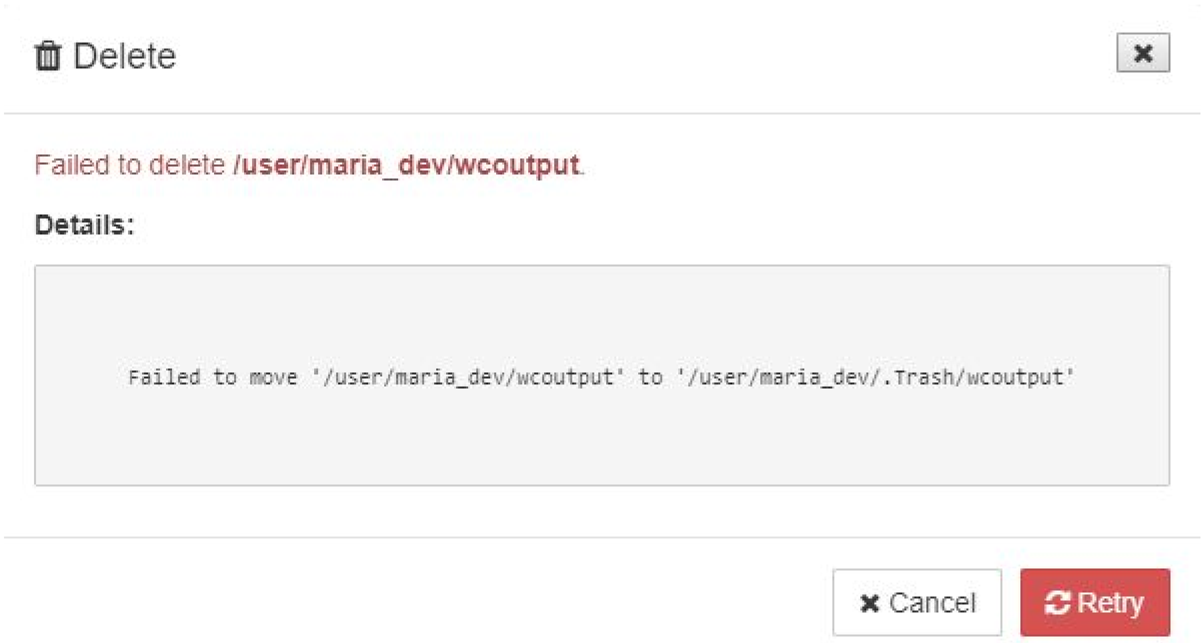
9. Our output is sitting in a folder on the HDFS. Go back to the Ambari File View and go to the /user/maria_dev folder. You will notice that there is a **wcoutput** folder. If you do not see the folder, click on the refresh icon to the left of the forward slash (/). Once you see the wcoutput folder, click on it. You will find two files in that folder, a _SUCCESS file and a part-00000 file. Our answer is stored in the part-00000 file. Click on the part-00000 file and click on Open. You should see something like this:


Ambari
Sandbox 0 ops 0 alerts
Dashboard Services Hosts Alerts Admin
maria_dev
/ > user > maria_dev > wcoutput
1 Files, 0 Folders selected
Select All
New Folder
Upload
Open Rename
Name
_SUCCESS
part-00000
File Preview
/user/maria_dev/wcoutput/part-00000
Cancel
Download

```
a       1
hello   1
is      2
test    1
this    2
```

10. In order to get the files/folder out so you can submit it, go back to the /user/maria_dev folder. Click on **wcoutput** and then click on **Download**. You should not include this file in your submission but this is the process to download your folders/files.
11. Hadoop is picky when it comes to output. If the folder already exists, Hadoop will fail. Therefore, *always be sure the output folder you are using has not yet been created. If it has been created, delete it or use a different folder*. You can delete your folder from the Ambari Files View by clicking on the folder and clicking delete. It will confirm you want to delete the

folder. Simply confirm by clicking on Delete. In some instances, the Ambari File View is not able to delete the folder because something went wrong. You may see something that looks like this when trying to delete a folder:Use the following commands to delete the folder:



Failed to delete **/user/maria_dev/wcoutput**.

Details:

```
Failed to move '/user/maria_dev/wcoutput' to '/user/maria_dev/.Trash/wcoutput'
```

12. If this happens, go back to your command line (http://YourAzureIP:4200) and type in the following commands:

```
hdfs dfs -rm /user/maria_dev/wcoutput/*
hdfs dfs -rmdir /user/maria_dev/wcoutput
```

13. Copying the mapper and reducer from HDFS to the local hortonworks filesystem was not necessary. We can use the mapper and reducer on HDFS. The following command is similar to the command from before but now the mapper and reducer are loaded from HDFS:

```
hadoop jar
/usr/hdp/2.6.4.0-91/hadoop-mapreduce/hadoop-streaming.jar -files
hdfs://sandbox-hdp.hortonworks.com/user/maria_dev/hadoopTest/map
per.py,hdfs://sandbox-hdp.hortonworks.com/user/maria_dev/hadoopT
est/reducer.py -input /user/maria_dev/wcinput/* -output
/user/maria_dev/wcoutput -mapper mapper.py -reducer reducer.py
```

Task 5: Solve MapReduce Problems (Optional, solutions given at end)

1. Your goal for this task is to write code for the MapReduce problems presented during the lecture. The pseudocode solutions have already been given in the lecture. Your goal is to implement the solutions using Python code and then verifying your solutions work using Hadoop.
2. In order to help with your solutions, a template for the mapper and reducer have been posted. You should follow the template and insert code where the comments say to insert code. You can download the mapper and reducer starter code from:
    a. http://www.uwosh.edu/faculty_staff/krohne/ds730/mapperStarter.py
    b. http://www.uwosh.edu/faculty_staff/krohne/ds730/reducerStarter.py

    You are encouraged to use these starter files for the problems on the project as well.
3. These are the problems to solve for this task:
    a. Count the number of times each pair of characters (excluding whitespace) appear consecutively in a file.
    b. For each word length, which character pairing appeared the most times?

4. Running code on the Hadoop system can be a bit challenging. The error messages may not make sense and it can be difficult to figure out what it wrong with the code. There are a couple of ways for you to do a sanity check on your code before running it on Hadoop. Neither of these tests are guaranteed to find all errors but they are tools you can use to check if your code might work. Code that fails these tests will fail on Hadoop. Code that passes these tests ***might pass*** on Hadoop. Here are the things you can try:
    a. Create your mapper.py and reducer.py file and go to http://YourAzureIP:4200 and type in the following commands:
        i. python mapper.py < input.txt > intermediate.txt
        ii. sort -n intermediate.txt > sorted.txt
        iii. python reducer.py < sorted.txt

        The problem with the above commands is that it doesn't ensure your code is stateless. When you run your mapper and reducer in this fashion, you are simulating running them on 1 machine. Therefore, all lines in the input file will end up on the same mapper and all keys will end up on the same reducer. Therefore, if your code is not stateless, the above commands might work and produce the correct output but would fail on Hadoop. Therefore, the above is to be used as an initial sanity check to make sure your code would work if there was only 1 mapper and 1 reducer. If it doesn't work in that scenario, it certainly won't work when there are multiple mappers and multiple reducers.

b. If your code produces the correct output with the previous commands, the next test can check if the code will probably work on Hadoop. I have created a Java program that tests the statelessness of your code. It is not 100% accurate but it is pretty close. Go to http://YourAzureIP:4200 and type in the following commands:
   i. wget http://www.uwosh.edu/faculty_staff/krohne/ds730/TestStateless.java
   ii. javac TestStateless.java
   iii. java TestStateless mapper.py reducer.py inputFile outputFile Y

   The TestStateless code has been updated since the presentation was created so the command to run it is a little bit different than is explained in the presentation. The mapper.py and reducer.py are you mapper and reducer. The TestStateless code only accepts 1 input file and is specified in the inputFile argument. It produces 1 output file and is specified by the outputFile argument. The last argument is either a Y or an N. If you put a Y, the temporary mapper and reducer files will be kept. The mapper temporary files are not that interesting as they are just your inputFile split up into files 1 line at a time. The temporary reducer files store all (key,value) pairs that have the same key. These will likely be of interest to you. All lines in reducerInX.tmp will end up on the same reducer. Lines from a file called reducerInY.tmp may or may not end up on the same reducer as reduceInX.tmp (assuming Y ≠ X). There are several caveats to using this Java program:

   iv. Your input file needs to be relatively small. The program creates a new file for each line of your input. If you have 50,000 lines in your input, the program will create 50,000 files. Make sure your input is small enough.
   v. The number of keys you create must be relatively small. The program creates a new file for each unique key your mapper generates so if you create thousands of keys, the program will create thousands of files.

5. The main goal of this activity is to get you running MapReduce code on Hadoop and using some of the tools available to test your MapReduce code. You are encouraged to write the solutions to these problems without looking at the solutions that are provided. If you need to look at the provided solutions, then not being able to translate an English solution into Python code is a bit of a red flag that you may not be ready for this course. If your solution is wildly different from the ones given below and you are curious if your solution is good, feel free to post it to the message board.
   a. Mapper solution for 3a:
      http://www.uwosh.edu/faculty_staff/krohne/ds730/mapreduce/mapper.py
   b. Reducer solution for 3a is identical to the reducer shown in the MapReduce Example presentation, slide 13.

    c. Mapper solution for 3b:
       http://www.uwosh.edu/faculty_staff/krohne/ds730/mapreduce/mapper2.py
    d. Reducer solution for 3b:
       http://www.uwosh.edu/faculty_staff/krohne/ds730/mapreduce/reducer2.py

Task 6: Gather and Test Your Code

Gather Previous Output

1. Save your output from the following in a file called **output.txt**. Separate the outputs with some delimiter so it is easy to read.
    a. Task 1, Test HDFS, step 5
    b. Task 1, Test HDFS, step 8
    c. Task 2, step 3

Test Your Word Count Code

Run the word count program using Hadoop MapReduce on Leo Tolstoy's novel *War and Peace*.

1. You can download the whole document using this link[2]:
   http://www.uwosh.edu/faculty_staff/krohne/ds730/wap.txt
   There are some headers and other information in the document other than the novel. Do not make any changes to the wap.txt document.
2. Run the word count program and copy everything below this line:

```
18/07/20 17:17:48 INFO mapreduce.Job: Counters: 49
```

3. The next line should be **`File System Counters`**. Do not worry if you have a different number of Counters. Start from there and copy to the end.
4. Store the copied lines into a file called **wapOutput.txt**.
5. Download the **part-00000** file from the **wcoutput** folder. Rename the file to **wapWords.txt**.

Submitting Your Work

When you are finished, zip up the three following files into a single zipped file called `a2.zip`:

- **output.txt**
- **wapOutput.txt**

[2] For those of you who want to do this from the command line. You can download the file by using the following command:
`wget http://www.uwosh.edu/faculty_staff/krohne/ds730/wap.txt`
You can then upload it to your HDFS folder by using the following command:
`hdfs dfs -copyFromLocal wap.txt /user/maria_dev/wcinput/`

- **wapWords.txt**

and upload only the `a2.zip` file to the **Activity 2 dropbox**.

`mapper.py`

```
#!/usr/bin/env python
#Be sure the indentation is identical and also be sure the line above this is
on the first line


import sys

import re


def main(argv):

  line = sys.stdin.readline()

  pattern = re.compile("[a-zA-Z0-9]+")

  while line:

    for word in pattern.findall(line):

      print(word+"\t"+"1")

    line = sys.stdin.readline()

#Note there are two underscores around name and main

if __name__ == "__main__":

  main(sys.argv)
```

reducer.py

```
#!/usr/bin/env python
#Be sure the indentation is correct and also be sure the line above this is on
the first line

import sys

def main(argv):
  current_word = None
  current_count = 0
  word = None
  for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)
    count = int(count)
    if current_word == word:
      current_count += count
    else:
      if current_word:
        print('%s\t%s' % (current_word, current_count))
      current_count = count
      current_word = word
  if current_word == word:
    print('%s\t%s' % (current_word, current_count))

#Note there are two underscores around name and main
if __name__ == "__main__":
  main(sys.argv)
```