CprE 419 Lab 2: Text Analysis using Hadoop MapReduce

Department of Electrical and Computer Engineering
Iowa State University
Spring 2016

Purpose

The goal is to introduce you to the MapReduce programming model under Hadoop. MapReduce is a significant abstraction for processing large data sets and has been applied successfully to a number of tasks in the past, including web search. At the end of the lab, you will be able to:

- Write and Execute a program using Hadoop MapReduce
- Write algorithms for data processing using MapReduce
- Apply these skills in analyzing basic statistics of a large text corpus

Submission

Create a zip (or tar) archive with the following and hand it in through blackboard**.**

- A write-up answering questions for each experiment in the lab.
- Output for each experiment in a .txt file (example: exp1_output.txt, exp2_output.txt)
- Commented Code for your program. Include all source files needed for compilation.
- Executable JAR that produced your output.
- Due on 01/31/2017 Tuesday 11:59pm

MapReduce

MapReduce is a programming model for parallel programming on a cluster. We will be working with the Hadoop implementation of the MapReduce paradigm.

A Program based on the MapReduce model can have several rounds. Each round must have a *map* and a *reduce* method. The input to the program is processed by the map method and it emits a sequence of key and value pairs <k, v>. The system groups all values corresponding to a given key and sorts them. Thus for each key $k \subset K$, the system generates a list of values corresponding to k, and this is then passed on to the reduce method. The output from the reduce methods can then be used as the input to the next round, or can be output to the distributed file system.

A key point is that two mappers cannot communicate with each other, and neither can reducers communicate with each other. Although this restricts what a program can do, it allows for the system to easily parallelize the actions of the mappers and the reducers. The only communication occurs when the output of a mapper is sent to the reducers.

Resources

Make sure you have access to Cystorm. Take a look at the Example Hadoop Program found on Piazza **– "WordCount.java"**. A MapReduce program typically has at least 3 classes as shown in the example code: A **Driver Class** (in our case, WordCount)**,** a **Map Class** and a **Reduce Class.**

You can write your Java program on your local system (using an IDE such as Eclipse). There are two ways you can compile a Hadoop Map-Reduce program.

1. You can compile the program using Eclipse, while linking the Hadoop libraries which you can find on Piazza. Note that Hadoop only needs your jar file to run a MapReduce job, so you can generate the jar file locally and upload it to CyStorm. Find further instructions for developing in Eclipse (and other helpful development tips) in the Piazza document titled "Development Environment Tips" under General Resources.

2. You can also compile and generate the jar files on CyStorm as follows. You have to first upload the source files to CyStorm. Then do the following:

   Compile the Java program on the namenode as follows.
   $ mkdir class
   $ javac -cp "the path to hadoop jar files" -d class/ WordCount.java
   $ jar -cvf WordCount.jar -C class/ . (note the period at the end is part of the command)

   Or use Eclipse to help you compile and export (preferred). Don't forget to choose jre 1.7.

Notes

- The MapReduce programs read input from the HDFS and write their output to HDFS. However, you don't necessarily need to use the HDFS API while programming with MapReduce.

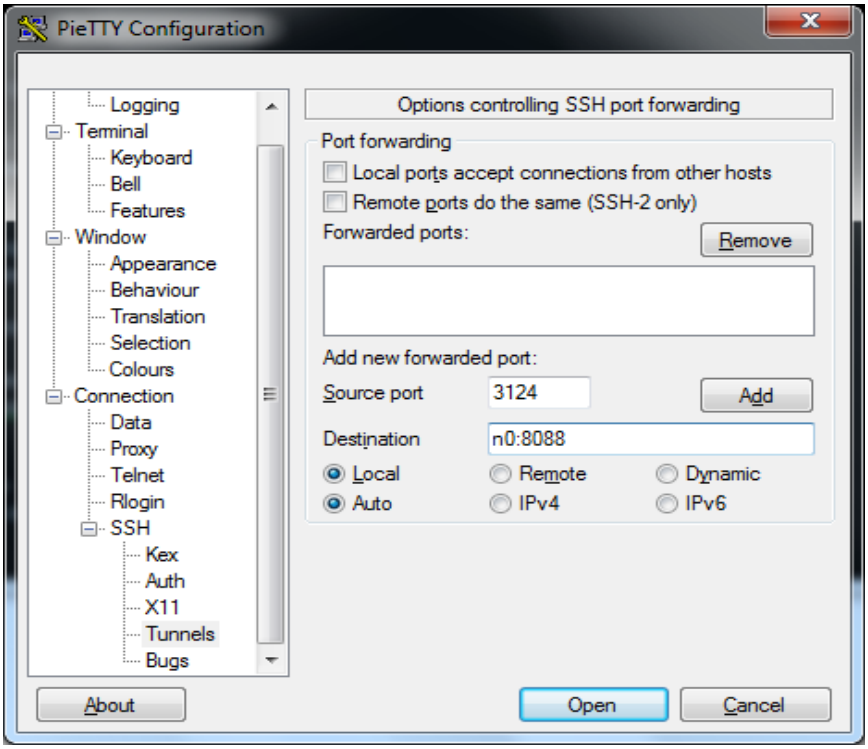
- Two Hadoop programs running simultaneously cannot have the same output path, although they can share the same input path. Thus, make your output path unique, as otherwise your job may have a conflict with the output of other jobs, and hence fail. Finally make sure that the output directory is empty by deleting its contents. You must do this every time you re-run the program or it will fail.

- Note that each MapReduce round can have multiple input paths, but only one output path assigned to it. If given an input path that is a directory, MapReduce reads all files within the input directory and processes them.

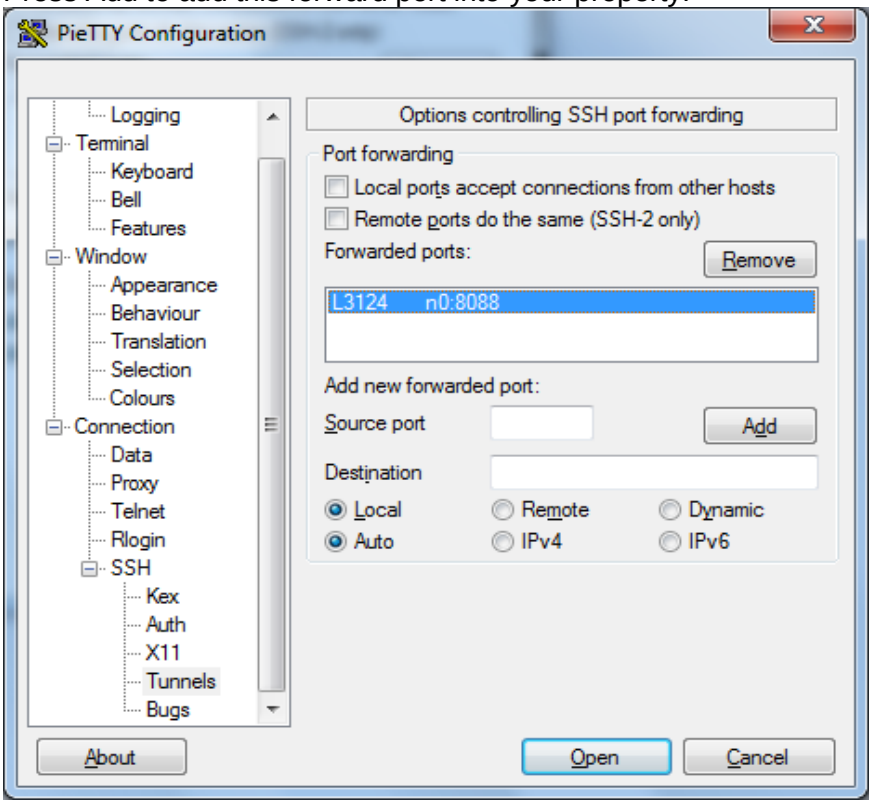
- You can explore the Hadoop MapReduce API in the link: http://hadoop.apache.org/docs/r2.4.1/api/

- For viewing your job status on your web browser, you need to set up your putty properties.

1. Open putty, and click connection->ssh->tunnels. Enter any number to be your port number (3124 in this pic), and n0:8088 as the destination.

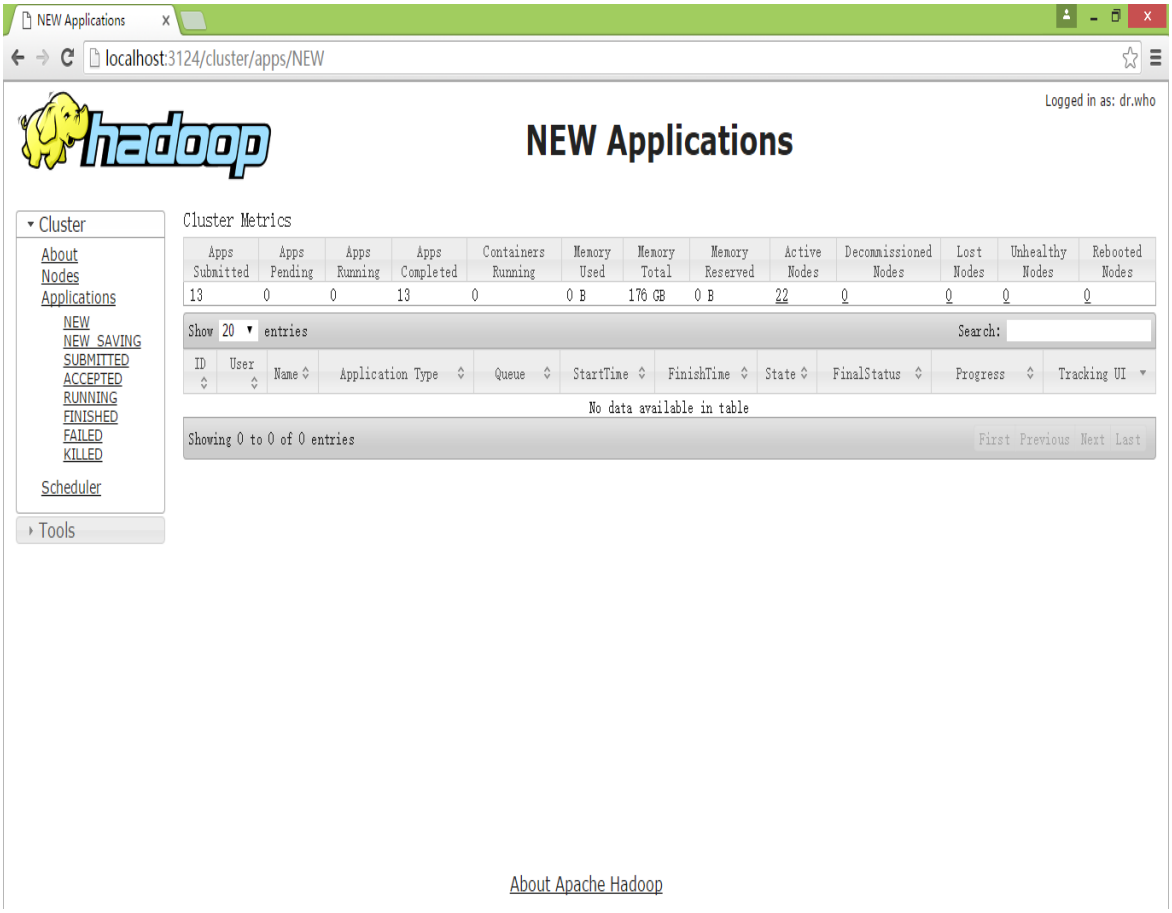


2. Press Add to add this forward port into your property.



3. Open your web browser, and put localhost:<your port number>, (localhost:3124 in the example), and you can see the job queue in your browser. Once you submit a Hadoop job,

you can monitor it through this web page.



Experiments

1. The WordCount program counts the number of occurrences of every distinct word in a document. Download the file WordCount.java and compile the program as described above. Make sure the jar is located in your home directory on CyStorm. Then run the program as follows:

To fairly share resources with other users of CyStorm, you will submit your jobs to a work queue as instructed below. This is different than Lab1, so please read carefully:

```
Step 1:
---------
Create your Hadoop Jar and transfer it to CyStorm (done above!)

Step 2:
---------
Create a file named job.file with the following contents:
hadoop fs -rmr <Output HDFS Path>
hadoop jar <Full path to Jar File> <Class Name>
```

```
For example:
hadoop fs -rmr /scr/<your id>/lab2/exp1/output
Hadoop fs -rmr /scr/<your id>/lab2/temp
hadoop jar /home/<your id>/WordCount.jar WordCount

You may also use this file to specify arguments. For example, WordCount.java
takes an argument for the input and the output. You can specify that in the
job.file as follows:
hadoop jar /home/<your id>/WordCount.jar WordCount
/class/s17419/lab2/shakespeare /scr/<your id>/lab2/exp2/output/

Step 3:
---------
Submit the job in Cystorm (NOT namenode) as follows:
$qsub -q studenthadoop -j oe -o <>.txt -N <> -v job=<> job.file

Example:
$qsub -q studenthadoop -j oe -o output_file.txt -N WordCount_yourID -v
job=WordCount_yourID job.file

In this example, output_file.txt is the name of the log file that will be
saved in your home directory and WordCount_yourID is the label of the job. It
is important to include your net-id to distinguish from the other students
jobs.



Step 4:
--------
Check your job as:
$qstat studenthadoop

Other information:
------------------
You can also delete a job using $qdel <Job Name>
```

Once again, make sure that the output location "/scr/<your id>/lab2/exp1/output" and your temp directory defined at the top of your .java file "/scr/<your id>/lab2/temp" are empty before running as the job will fail otherwise. It is sufficient to include the remove commands in the job.file script as shown above. There will be one output file generated in the output location for every reducer. Show the output files to the lab instructor. (15 points)

2. A bigram is a contiguous sequence of two words within the same sentence. For instance, the text "This is a car. This car is fast." Has the following bigrams: "this is", "is a", "a car", "this car", "car is", "is fast". Note that "car this" is not a bigram since these words are not a part of the same sentence. Also note that we have converted all upper case letters to lower case. (50 points)

Task: Write a Hadoop program to identify the single most frequently occurring bigrams beginning with each letter in an input text corpus, along with their frequencies of occurrence. For example, "ancient history" and "another lambchop" are both bigrams that begin with "a" while "stay shiny" and "serenity firefly" are both bigrams that begin with s. If "stay shiny" occurs 10 times and "serenity firefly" only occurs once, you would output "stay shiny, 10". (Don't worry about the formatting, this is just an example.)

You can assume that each call to the Map method receives one line of the file as its input. Note that it is possible for a bigram to span two lines of input but they are in two mapper input splits; you can ignore such cases and you do not have to count such bigrams. Also remove any punctuation marks such as ",", ".", "?", "!" etc from the input text (this can be done in the mapper), and convert all letters to lowercase.

Question: Think about how you might be able to get around the fact that bigrams might span lines of input. Briefly describe how you might deal with that situation? (5 points)

Hint: In this task, you might need multiple rounds of MapReduce.

Think about the following in designing your algorithm:

- The number of rounds of map reduce
- The communication within each round
- What is the load on a single reducer

Make sure that the output path of each MapReduce job is a unique directory that does not conflict with the output path of another job. Use the following path to avoid conflict:
"/scr/<your id>/lab2/exp2/output".

Skeleton code (Driver.java) is provided to help you to start with the exercise. You can find it on Piazza in the labs section. The code is well commented so as to help you understand it. Read the comments in detail and try to understand the code. Then you can build on this code to write your program.

You can use the Shakespeare corpus as the testing input to your program. It is uploaded on the hdfs at the following location: /class/s17419/lab2/shakespeare
Once your program runs successfully on this input, you can try to run your program on larger input files. You will find a much larger dataset in the location: /class/s17419/lab2/gutenberg

Hint: For long running jobs call context.progress() in your code to avoid timeout dump. For example, in a reduce step you typically iterate over values. You can start each of those iterations with a call to context.progress().

Please submit the source code along with the outputs for both the datasets: Shakespeare and Gutenberg. Your output must include the most frequent bigrams for each letter and their frequencies of occurrence.