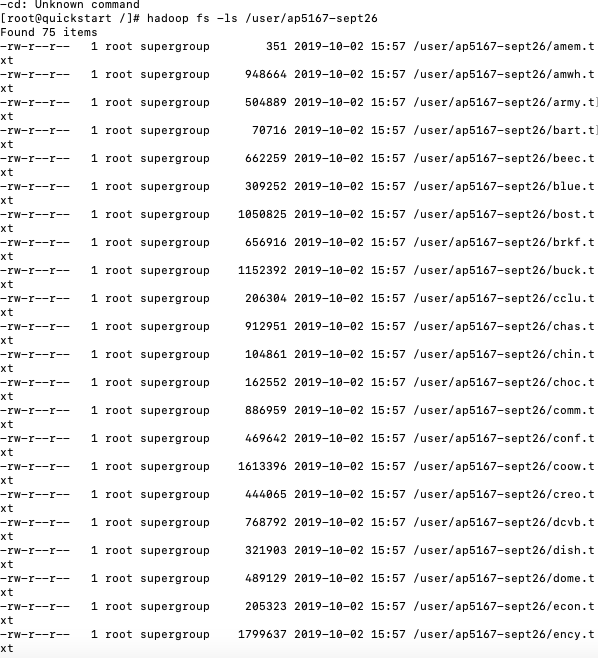
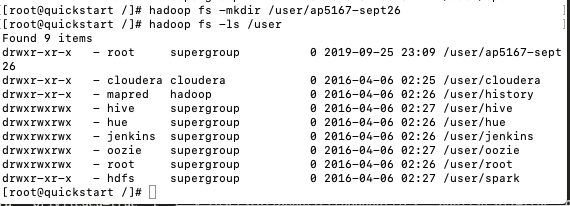
Allan Porter

Ap5167

Part 1: HDFS custom directory(ap5167-sept26) created and files needed for part 2 inserted inside.

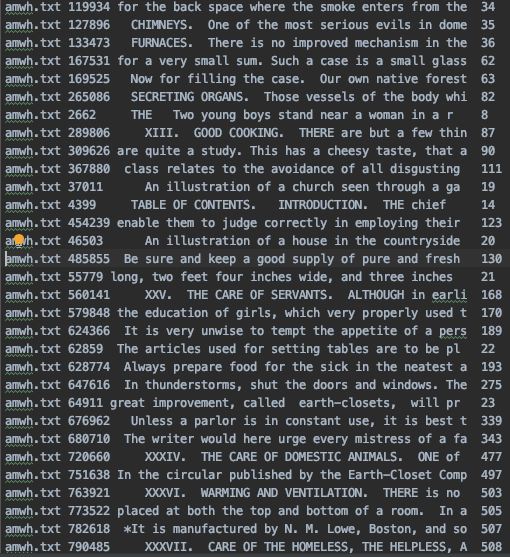
**Note**: In the file “WordCount.java” I used a different WordCount class for part 2 and part 3. By default, the part 3 word count class is active and the part 2 is commented out. To run part 2, part 3 needs to be commented out and part 2 needs to be active.

Configuration for part 2: Use data as input

Part 2: To develop a WordGrep program in Hadoop, I customized the map and reduce functions on top of the template code given in class. In the map function, I created a temporary string entitled *temp* and set it to the (filename, byte offset of line, line content) for each key value pair only if the value (the body of the line of the text file) when converted to completely lowercase contained the word “door”. If the value did not contain the word “door” then I simply did not output. I was also able to get the line number by setting a counter and incrementing it each time the map ran over a key value pair. For the line content, because the lines in the text files were very long and not well spaced, I only output the first 50 characters. Note if the lines contained less than 50 characters I output the entire line content. I passed temp as the key and the line count as the value to the reducer, which merely output those values. I outputted the line # after the line content.

Note: I tested my WordGrep program on another directory I created with only 2 short text files with well defined lines to assure that it ran correctly, since the text files in the directory given were very long. After doing so, I then ran it on the cookbook directory given.

Part 2 output



Part 3:

Configuration for part 3: use cookbook.zip as input

To read in a zip file with Hadoop I had to customize 2 classes- the InputFormat class that initially takes in the data and splits it into blocks, and also the RecordReader class that actually takes the split data and turns it into records of key, value pairs to then give to the mapper.

**Customizing InputFormat**

The issue with zip files is that they start with metadata and can not be partitioned well by the InputFormat class. Therefore, I had to edit the InputFormat class to not split the zip files data at all and instead just take the data in. To do this, in the getSplits function of the Input format class I set isSplittable to return false meaning that the data could not be split.

**Customizing RecordReader**

The first thing to do is define the variables needed in RecordReader.

Firstly, global key and value variables will have to be set to indicate the keys and values that will have to be given to the mapper.

Secondly, an instance of Java ZipInputStream class for reading in zip files should be defined.

FSData input stream for reading input files should also be set.

* Initialization Function

First define the input split as a file split (since it is splitting by file). A file path for the split is then subsequently defined. The configuration is set and the filesystem is set from the path an is then opened. The zip input stream class then is used to read in the split files.

* Next Key Value Function

First make sure that the entry from the zip input stream isn’t null. If it is have nextKeyValue() return false. Then get the name (filename) and set that as the key for the record to be sent to the mapper later. Set a ByteArrayOutputStream along with a max number of bytes to write (here I selected 2000). Have the zip input stream read in the files (2000 per iteration) and assuming there are bytes remaining, have the ByteArrayOutputStream write them. The value will then be set as the Bytes that were written by ByteArrayOutputStream in a ByteArray object.

Then have the getCurrentKey and getCurrentValue functions return key and value (subsequently).

Note: The following file was used as a reference for merely learning how to set the configuration for part 3, <https://gist.github.com/jteso/1868049>.

**Customizing Mapper and Reducer**

**Note: For WordCount I denoted in my file which class should be used for part 2 and which should be used for part 3. I commented out part 3 to run part 2, and then commented out part 2 to run part 3.**

The records are then passed to the WordCount class where the mapper will process them.

Here, the mapper is taking in value type BytesWritable. I used value.getBytes() to turn the data in the Bytes to a string, which I subsequently name “vstring”. I then set a line\_no variable for the line number and an offset variable for the line offset.

I then used a string tokenizer to tokenize the vstring while using endline as the delimiter so it iterates line by line. On each line, I then checked if the line content contained the word “door”. If it did, I checked if the line was over 50 chars and if it was I capped the line at 50 chars (since lines in the file given could be very very long). I then had the mapper write the filename, line#, line content (up to 50 chars) and line offset.

Part 3 output:

