Programming Assignment 2

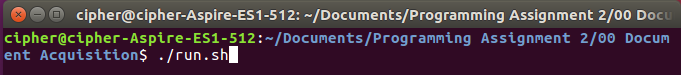
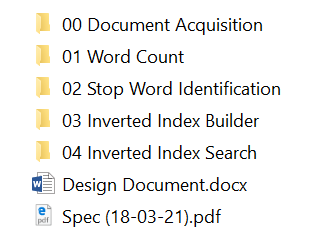
Peiyong Chen and John Till

**Objective**

This project aims to implement a reverse index to map word occurrences in the works of Shakespeare to their corresponding documents and line numbers. The reverse index is built using a MapReduce function in Hadoop. We build a simple linear search function to look up words in the inverted index. We demonstrate how the works of Shakespeare may be parsed to build our reverse index in a CloudLab cluster.

**Project Structure**

The code is logically divided into 5 phases with folders numbered 0-4. Every folder except for #4 has a “run.sh” shell script which executes all necessary code for that phase. Note that for phases 1-3, “hadoop” must be on the path.

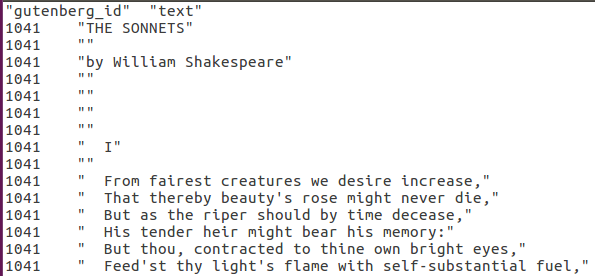


**0 - Obtaining Source Files for Shakespeare’s Work**

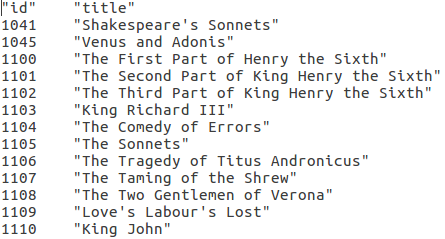
Before building a reverse index, we must obtain Shakespeare’s body of work in a format which can easily be parsed. Project Gutenberg is a volunteer-run organization which hosts public domain works, and we rely on their collection of Shakespeare’s works. The Gutenberg website does not facilitate downloading an author’s entire body of work, but David Robinson has written an excellent R package “gutenbergr” which can download all works by an author [1]. This package also allows the user to omit header/footer information, which will be useful for our reverse index analysis.

It is unnecessary to execute this phase since the output is included with our submission. If you do run this phase, then “Rscript” must be on the path and the “gutenbergr” library must be installed. The library can be installed by using R to call `install.packages(“gutenbergr”)’ and selecting a mirror for the download. Once these preconditions are met, it is possible to use “./run.sh” or simply “Rscript download\_Shakespeare.R” to gather Shakespeare’s body of work.

The resulting text file of Shakespeare’s work looks like:



…continued for 508,657 lines. Although we only consider the document ids for this project, the R script also saves the mapping of document ids to titles which could potentially be useful.

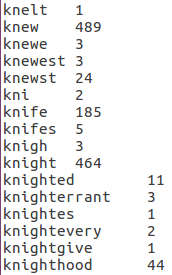


…continued for 96 lines.

**1 - Word Count**

Parsing the works is mainly accomplished using the code from the Hadoop word count tutorial [2] which shows how to count words in a set of documents. The changes we make to this code are 1) to convert all text to lowercase and 2) to discard symbols and numbers. This may result in some loss of meaning with contractions and hyphenated words, but we consider the more complex rules of the English language to be outside the scope of our parser design.

The resulting output looks like



…continued for 53,664 lines. In the next section we will use the word count result to identify stop words before building the inverted index.

**2 - Stop Words**

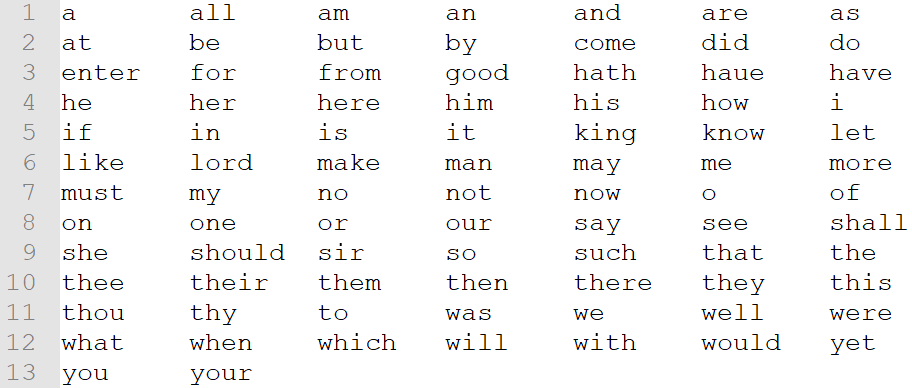
In order to omit common English words from the reverse index, we include a threshold for stop words. We take a sampling of word counts as shown in Table I to decide what threshold to use. By comparing the occurrence of words which are probably significant with the occurrence of noise words, we hope to strike an appropriate balance.

**Table I: Frequency of Selected Words**

|  |  |  |  |
| --- | --- | --- | --- |
| Common Significant Words | Count | Common Noise Words | Count |
| art | 3002 | as | 20740 |
| hamlet | 910 | our | 11049 |
| fool | 834 | so | 17511 |
| life | 2860 | the | 104559 |
| love | 4065 | thee | 10233 |

We want to err on the side of caution- it would be undesirable to remove an important word like “love.” It appears that a threshold of 5000 is appropriate. We modify the word count MapReduce function to make a list of words exceeding the stop word threshold. The reduce function is changed so that only words with more than 5000 occurrences are written to the output as keys, and the value is a “NullWritable” Hadoop object.

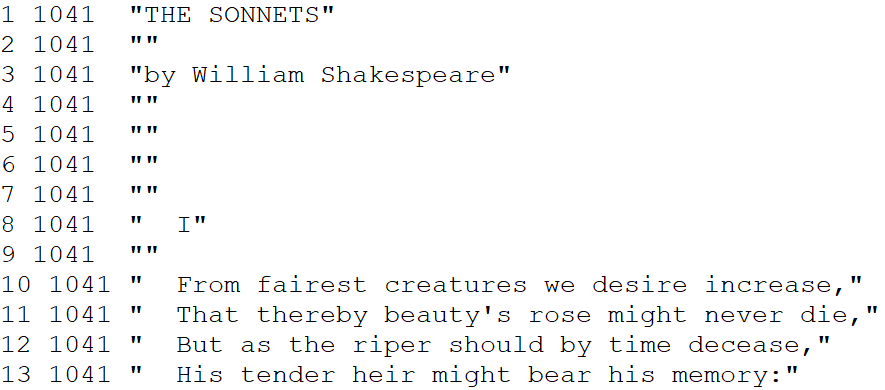
**Table II: Stop Words**



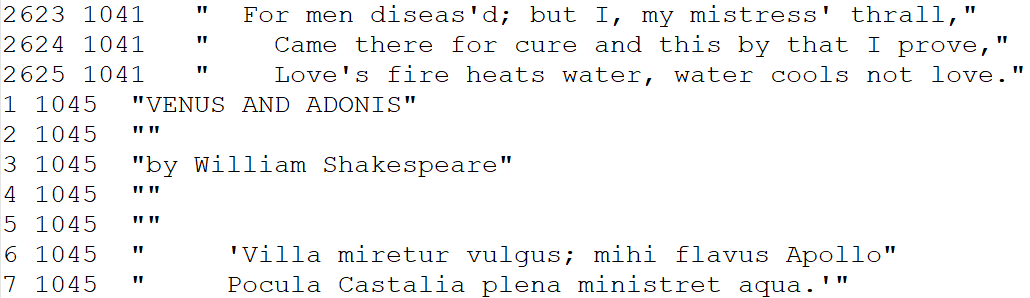
There are 86 words in the stop word list. Words in the stop list are ignored in the inverted index map function presented in the next section.

**3 - Building the Inverted Index**

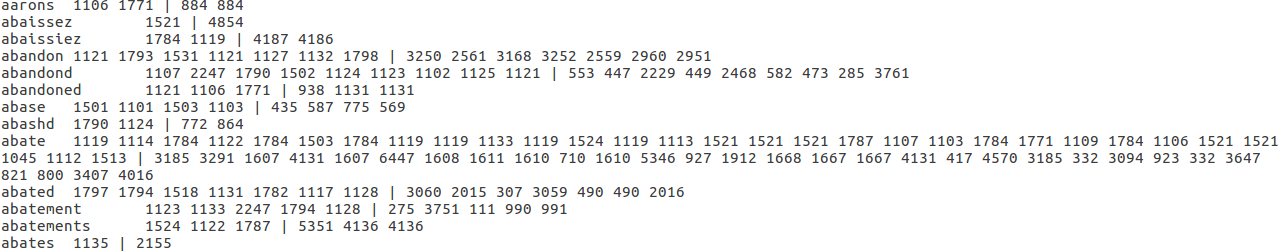
The main additional challenge to consider when building the inverted index is recording line numbers. MapReduce does not record line numbers by default because it splits input files to be processed by different mapper nodes. To account for this, we have a preprocessing stage where we prepend the line number to the start of each line.



This way our MapReduce can parse the line number as part of the line. The line numbers are counted for each document rather than for the entire collection, that is the line number resets to 1 for each new document id. For example, the transition from id 1041 to 1045 is shown:



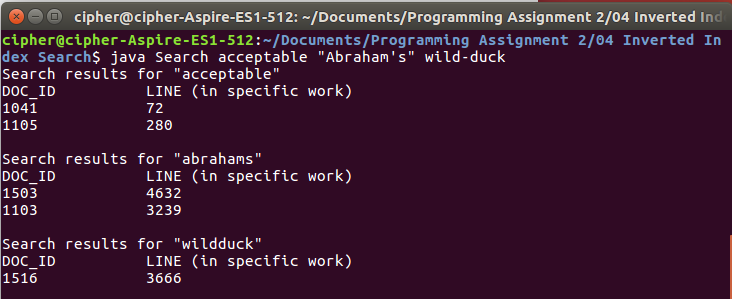
The inverted index is built by a MapReduce function which is like our previous functions. The map function is changed to ignore words in the stop list. Also, we can no longer fully eliminate numbers from the text since we need to parse the document id and line number, but will still ignore any numbers other than the id and line number. The output key for the map is the id and line number as strings, separated by a space. The reducer parses the list of ids and line numbers, then has an output key of all the ids, then a “|” dividing character, then the corresponding line numbers.



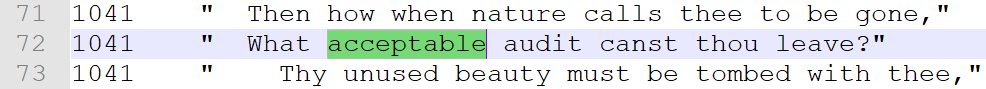
…continued for 33,596 lines. So for example, “abatements” has <id,line#> pairs of <1524,5351>, <1122,4136>, and <1787,4136>. This output format is easily parsed by a search function.

**4 - Querying the Inverted Index**

We implement a simple Java script with searches the inverted index using a linear search. For each searched word, it prints a two-column table with the document id and line numbers where the word occurs. The screenshot below shows an example search:



Note that the search also shows the modified search word, e.g. we can see that “wild-duck” has been converted to “wildduck.” If we look at a work in isolation (strip away the column information and non-related works), we can see that the search results are correct, for example “acceptable” in 1041 (Shakespeare’s Sonnets) occurs in the following context:

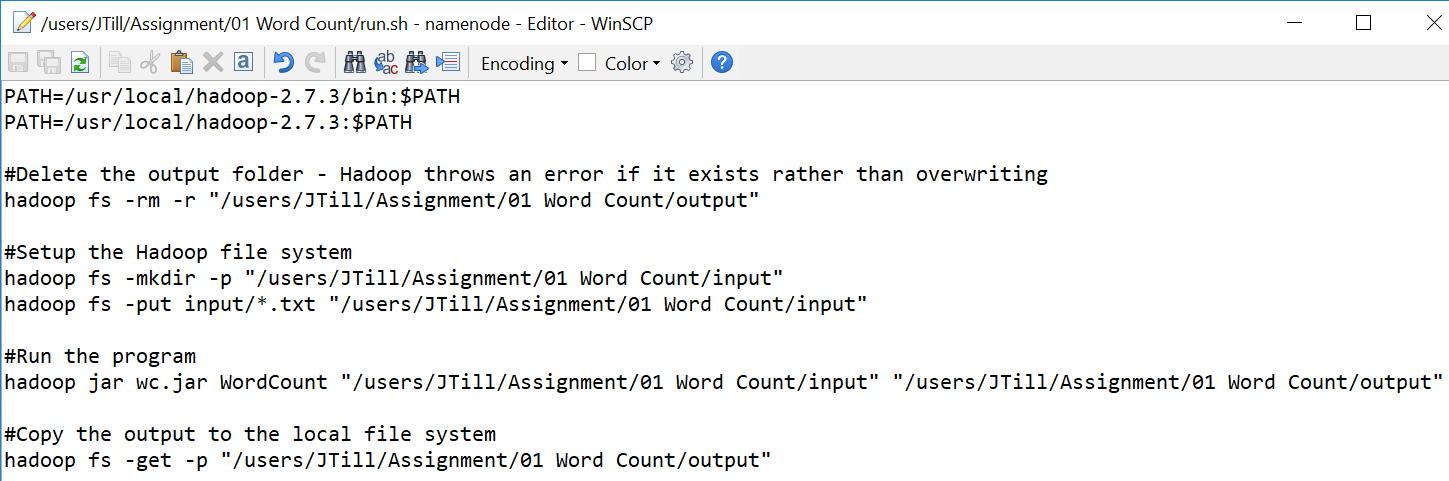


**Queries with Spark**

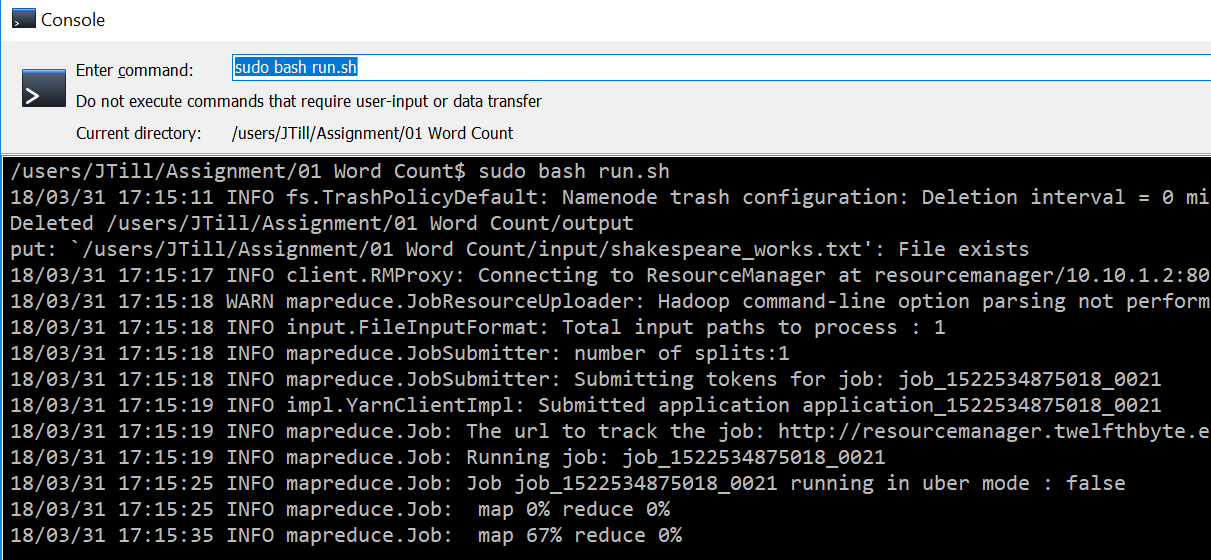
We did not implement any distributed search with Spark.

**Running on CloudLab Cluster**

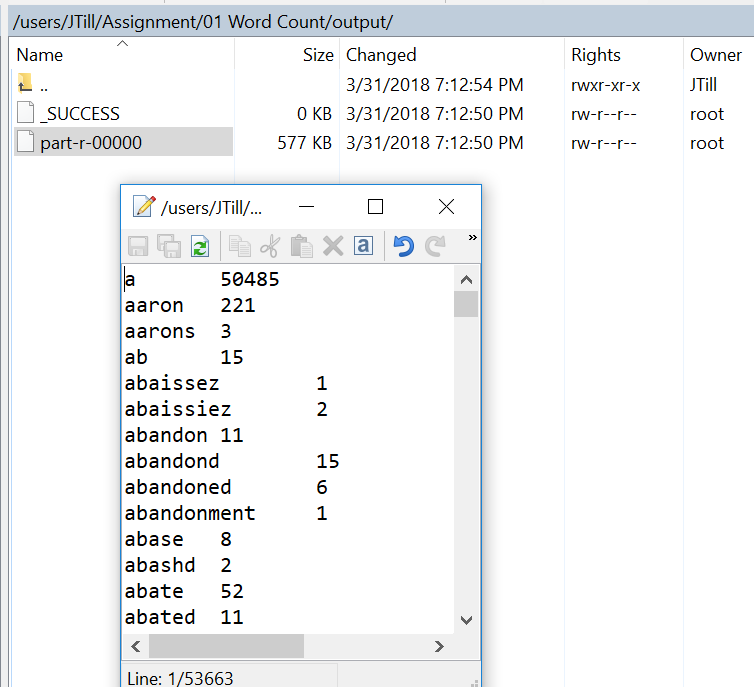
After validating our code locally, we tested it on four CloudLab nodes (one master and three slave nodes). We connect to the cluster using WinSCP. The bash script to run the code needs some modifications. The compilation steps are removed since we can compile the code locally. We also need to add Hadoop to the path depending on how the cluster is setup. The final change compared to running locally is that we several Hadoop file system commands to upload our local files before running MapReduce and to transfer the result into the local file system. The modified code to run the Word Count stage is shown below:



This script is run with elevated privilege using “sudo bash run.sh”. The CloudLab cluster executes our code:



We can see the result in our SSH client:



This matches the results we obtained locally. The process is similar for the other stages using MapReduce for the stop word identification and inverted index.

|  |  |
| --- | --- |
|  | References |

|  |  |
| --- | --- |
| [1] | D. Robinson, "gutenbergr," 26 Jan 2018. [Online]. Available: https://CRAN.R-project.org/package=gutenbergr. |
| [2] | "MapReduce Tutorial," Apache Software Foundation, 18 11 2017. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html. [Accessed 23 03 2018]. |