

Learning Big Data Computing with Hadoop

Homework 2 for CS 6220 Big Data Systems & Analytics



Problem 3. Learning Big Data Computing with Hadoop and/or Spark MapReduce

For this homework, I chose to complete Option 1 of Problem 3.

Requirements:

Installed software required to run the programs in this repository:

- Java 14.0.2
- JDK 14.0.2
- Hadoop 3.3.0
- Git 2.28.0 (optional)

My system description:

Specifications of the machine I used to run the programs in this repository:

- macOS Catalina (10.15.6)
- 2 GHz Quad-Core Intel Core i5 (10th Generation)
- 16 GB RAM
- 500 GB SSD
- Hadoop running on Pseudo-Distributed mode

Repository Folder Structure:

- **input:**
 - All text inputs used for the MapReduce jobs.
- **WordCount:**
 - Java MapReduce program to count the occurrence of each word in a document or body of documents.
- **Top100Words:**
 - Java MapReduce program to find the top 100 most common words across a body of documents.
- **output:**
 - All outputs from the MapReduce jobs.
- **import_books:**
 - Contains the jupyter notebook used to import books from the Gutenberg Project.
- **images:**
 - Contains all images used in the Readme file.

1. HDFS Installation:

The first step of this homework was to setup HDFS in my local machine. In order to do so, I installed Java and Hadoop and edited the necessary configuration files.

- **Hadoop version:** 3.3.0
- **Hadoop mode:** Pseudo-Distributed (1 on the hdfs-site.xml configuration file)
- **hadoop command** is available globally (hadoop binary files were added to the path)
- **Configuration File Edits:**
 - **hadoop-env.sh:** Make sure to set export **JAVA_HOME** to the the Java home location in your machine.
 - **core-site.xml:**

```
<configuration>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/usr/local/Cellar/hadoop/hdfs/tmp</value>
    <description>A base for other temporary directories</description>
  </property>
```

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:8020</value>
</property>
</configuration>
```

- - **mapred-site.xml:**

```
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:8021</value>
  </property>
</configuration>
```

- - **hdfs-site.xml:**

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

Since, my computer was running macOS I found the following installation tutorial very helpful: <https://medium.com/beeranddiapers/installing-hadoop-on-mac-a9a3649dbc4d>

If you are running Windows you can follow this tutorial: <https://towardsdatascience.com/installing-hadoop-3-2-1-single-node-cluster-on-windows-10-ac258dd48aef>

If you running Linux, you can follow the official Apache Hadoop documentation: <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html>

After installing HDFS, start all services by running the start-all.sh script on the/sbin folder inside the hadoop folder.

- **Resource Manager Screenshot:**

The screenshot shows the Hadoop Namenode information page. The browser address bar is localhost:9870/. The page has a green navigation bar with links: Hadoop, Overview, Datanodes, Datanode Volume Failures, Snapshot, Startup Progress, and Utilities. The main heading is 'Overview 'localhost:8020' (✓active)'. Below this is a table with the following data:

Started:	Tue Sep 15 16:58:05 -0400 2020
Version:	3.3.0, raa96f1871bfd858f9bac59cf2a81ec470da649af
Compiled:	Mon Jul 06 14:44:00 -0400 2020 by brahma from branch-3.3.0
Cluster ID:	CID-c89c691f-248e-45ac-bdaf-fbfe2329cec8
Block Pool ID:	BP-1349213800-192.168.1.192-1599934795513

Below the table is a 'Summary' section. It states: 'Security is off.', 'Safemode is off.', '416 files and directories, 407 blocks (407 replicated blocks, 0 erasure coded block groups) = 823 total filesystem object(s).', 'Heap Memory used 94.09 MB of 147 MB Heap Memory. Max Heap Memory is 4 GB.', and 'Non Heap Memory used 80.17 MB of 84.69 MB Committed Non Heap Memory. Max Non Heap Memory is <unbounded>.' Below this is another table:

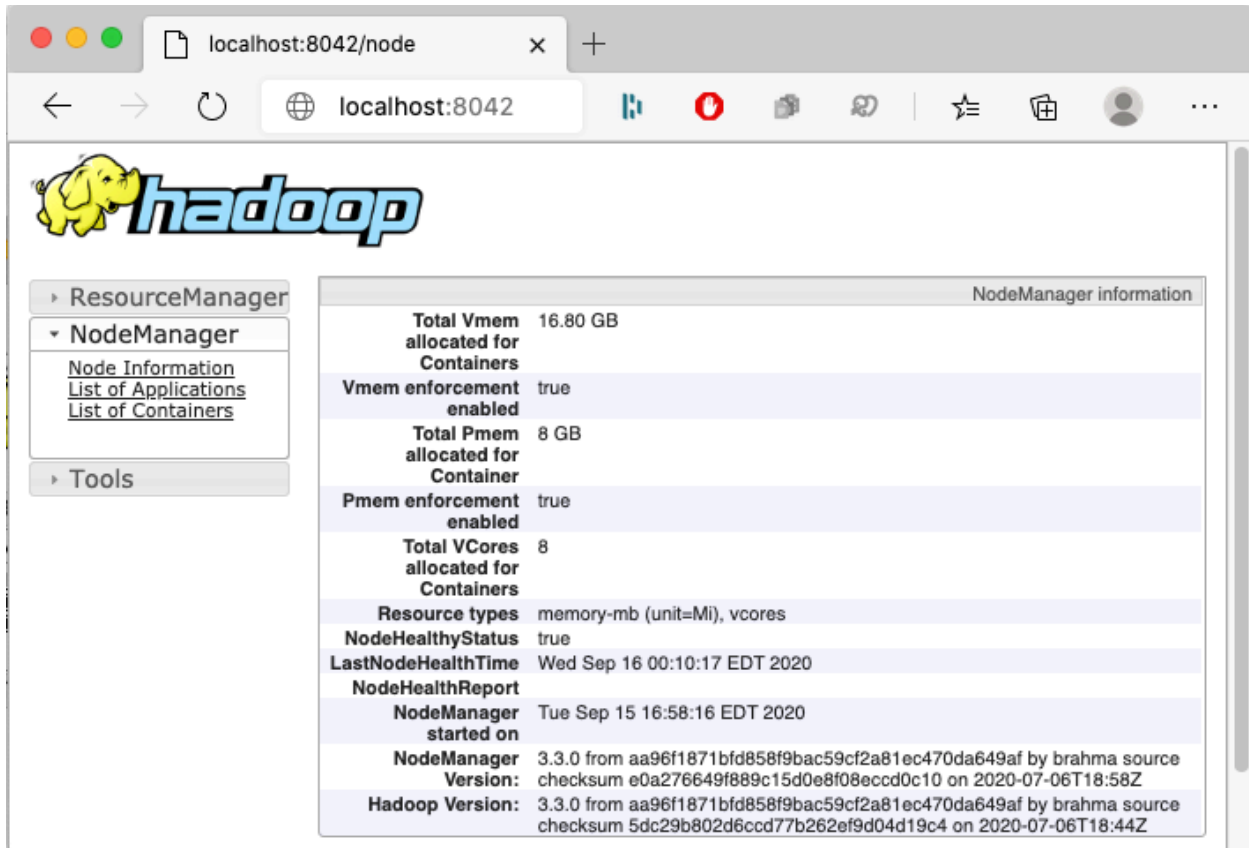
Configured Capacity:	465.63 GB
Configured Remote Capacity:	0 B
DFS Used:	3.84 MB (0%)

- **JobTracker Screenshot:**

The screenshot shows the Hadoop JobTracker 'All Applications' page. The browser address bar is localhost:8088/cluster. The page has a green navigation bar with links: Cluster, About Nodes, Node Labels, Applications, NEW, SCHEDULING, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED, Scheduler, and Tools. The main heading is 'All Applications'. Below this is a table with the following data:

Cluster Metrics	Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved
Cluster Nodes Metrics	0	0	0	0	0	0 B	8 GB	0 B	0	8	0
Scheduler Metrics	1	0	0	0	0	0	0	0	0	0	0
Capacity Scheduler	Showing 0 to 0 of 0 entries										

- **Node Manager Screenshot:**



NodeManager information	
Total Vmem allocated for Containers	16.80 GB
Vmem enforcement enabled	true
Total Pmem allocated for Container	8 GB
Pmem enforcement enabled	true
Total VCores allocated for Containers	8
Resource types	memory-mb (unit=Mb), vcores
NodeHealthyStatus	true
LastNodeHealthTime	Wed Sep 16 00:10:17 EDT 2020
NodeHealthReport	
NodeManager started on	Tue Sep 15 16:58:16 EDT 2020
NodeManager Version:	3.3.0 from aa96f1871bfd858f9bac59cf2a81ec470da649af by brahma source checksum e0a276649f889c15d0e8f08eccd0c10 on 2020-07-06T18:58Z
Hadoop Version:	3.3.0 from aa96f1871bfd858f9bac59cf2a81ec470da649af by brahma source checksum 5dc29b802d6ccd77b262ef9d04d19c4 on 2020-07-06T18:44Z

2. Data:

A number of different text files were used as input data for this homework. They are all saved in the input folder.

- american_pie.txt:
 - lyrics to the song American Pie by Don McLean, obtained from <https://www.lettras.com/don-mclean/25411/>
- hamlet.txt:
 - William Shakespeare's famous tragedy Hamlet, obtained from <https://gist.github.com/provpup/2fc41686eab7400b796b>
- charles_dikens:
 - this folder contains the 20 books published by the famous English author Charles Dickens.

- These files were obtained from *Project Gutenberg* using the Gutenberg python library to fetch the data.
- You can run the Jupyter notebook `/import_books/import_charles_dickens_books.ipynb` to understand the process.
- `bbc_tech_news`:
 - This folder contains 401 news articles by BBC on the topic of Technology.
 - The data was obtained from <http://mlg.ucd.ie/datasets/bbc.html>
- `song_lyrics`:
 - This data set contains the lyrics of songs by a number of different artists. For each artist, all of his or her lyrics were saved as a single txt file.
 - The data was obtained from <https://www.kaggle.com/paultimothymooney/poetry>
 - I manually subdivided the artists into four main genres:
 - Pop: 11 artists
 - Rock: 13 artists
 - Folk/Country: 6 artists
 - Hip-Hop: 11 artists
- Saving the data on HDFS:
 - To save the input data on HDFS, you just need to run the `export_inputs.sh` shell script in the root directory of the project by running the following command:

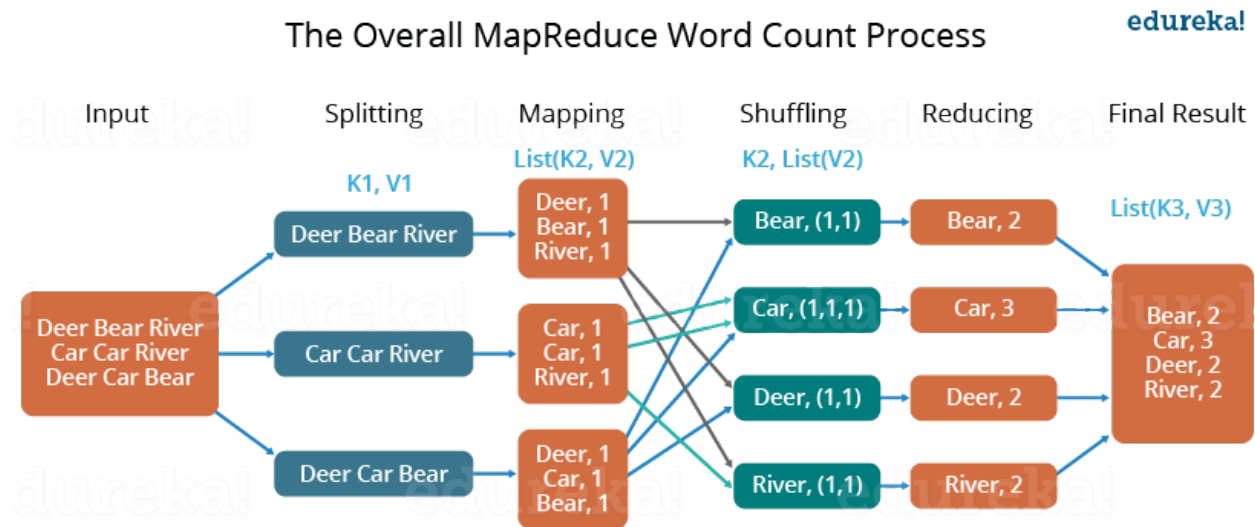
```
sh export_inputs.sh
```

In my machine, the script took ~25 seconds to run.

3. MapReduce WordCount:

This is a classic MapReduce problem where you take a document (or group of documents) as input and output all the different words sorted alphabetically along with the number of times each one occurred in the document or corpus.

- Job Diagram:



This image was obtained from <https://www.edureka.co/blog/mapreduce-tutorial/>

- As a starting point, I used the MapReduce tutorial on the official Apache Hadoop website <https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>.
- After compiling and running the program, I noticed that it was not filtering out punctuation or unusual characters after splitting the text based on whitespace " ". This led to strings like [first,], [who and tears- being considered distinct words.
- Steps to solve this problem:
 - Eliminating all characters besides letters, hyphens (-) and single apostrophes (')
 - After the initial filtering, keep only hyphens or apostrophes that appeared inside a word (e.g. Bye-bye or Nature's)
 - Standardizing the letter case by capitalizing the first letter in each word (e.g. Apple instead of apple, APPLE or aPpLe)

```
while (itr.hasMoreTokens()) {
    word = itr.nextToken();
    String pattern = "[^\\- 'A-Za-z]+";
    word = word.replaceAll(pattern, "");
    if (word.length() > 0) {
        int start = 0;
        while (start < word.length()){
            if (!Character.isLetter(word.charAt(start))) ++start;
            else break;
        }
    }
}
```

```
        int end = Character.isLetter(word.charAt(word.length()-1)) ?
word.length() : word.length() - 1;
        if (end < start) end = start;
        word = word.substring(start,end);
        if (word.length() > 0) {
            word = word.substring(0, 1).toUpperCase() +
word.substring(1).toLowerCase();
            clean_word.set(word);
            context.write(clean_word, one);
        }
    }
```

- **Running the program:**

- To run the MapReduce job, you need to change directories to WordCount/src and run the word_count.sh shell script by running the following commands from the root directory of the repository:

```
cd WordCount/src
sh word_count.sh
```

In my machine, the script took **~40 seconds** to run (including the time it takes to print all the progress reports to the terminal).

- Portion of a MapReduce successful job feedback message:

```
src — -bash — 94x46
2020-09-19 06:21:30,231 INFO mapred.LocalJobRunner: reduce task executor complete.
2020-09-19 06:21:31,038 INFO mapreduce.Job: map 100% reduce 100%
2020-09-19 06:21:31,039 INFO mapreduce.Job: Job job_local321370700_0001 completed successfully
2020-09-19 06:21:31,056 INFO mapreduce.Job: Counters: 36
  File System Counters
    FILE: Number of bytes read=6143696
    FILE: Number of bytes written=65075420
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=156625670
    HDFS: Number of bytes written=278740
    HDFS: Number of read operations=1935
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=44
    HDFS: Number of bytes read erasure-coded=0
  Map-Reduce Framework
    Map input records=150033
    Map output records=1121134
    Map output bytes=9936100
    Map output materialized bytes=1410248
    Input split bytes=5318
    Combine input records=1121134
    Combine output records=109634
    Reduce input groups=27254
    Reduce shuffle bytes=1410248
    Reduce input records=109634
    Reduce output records=27254
    Spilled Records=219268
    Shuffled Maps =41
    Failed Shuffles=0
    Merged Map outputs=41
    GC time elapsed (ms)=62
    Total committed heap usage (bytes)=18051235840
  Shuffle Errors
    BAD_ID=0
    CONNECTION=0
    IO_ERROR=0
    WRONG_LENGTH=0
    WRONG_MAP=0
    WRONG_REDUCE=0
  File Input Format Counters
    Bytes Read=5625626
  File Output Format Counters
    Bytes Written=278740
(base) Pedros-MacBook-Pro:src pedropinto$
```

- Portion of a MapReduce WordCount job output:

A	18	
About	1	
Above	1	
Adjourned	1	1
Admire	1	
Again	1	
Ago	1	
Air	1	
All	3	

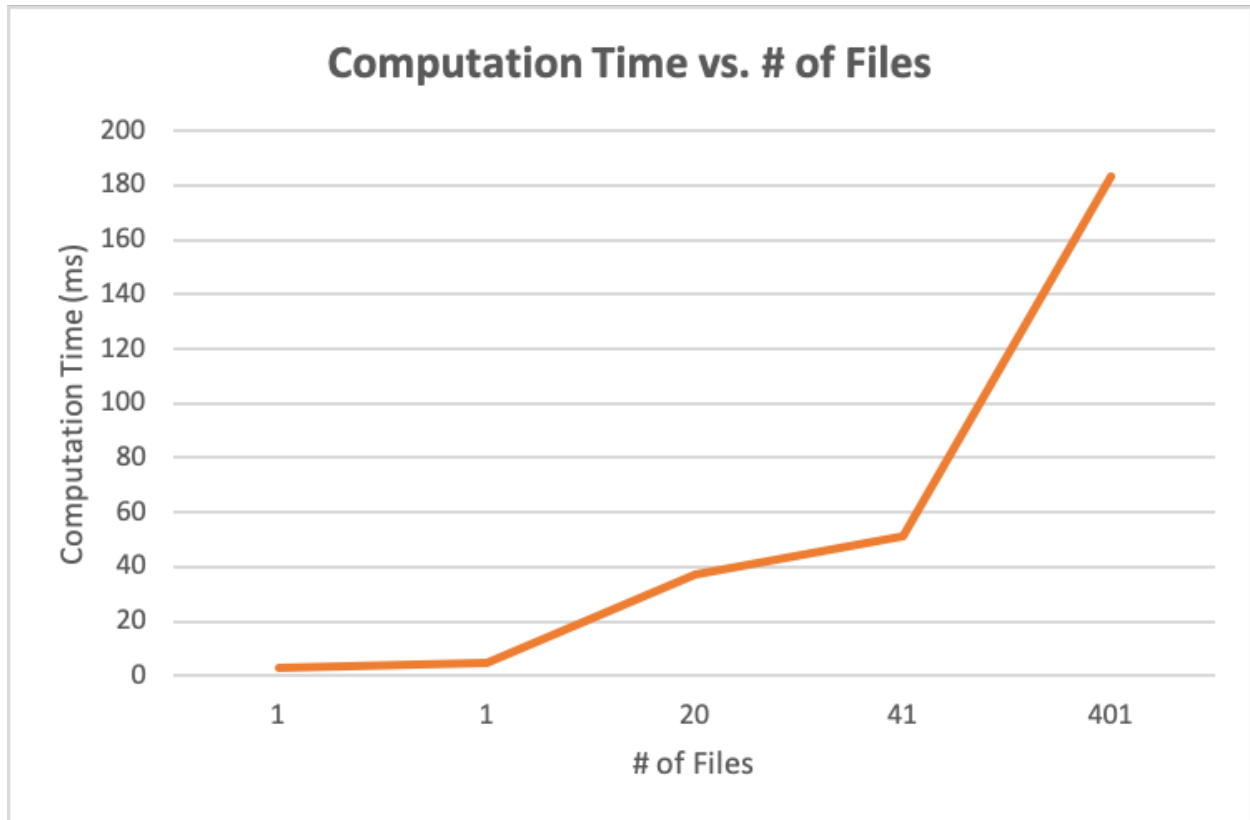
American	7
And	37
Angel	1
As	2
Asked	1
Away	1
Bad	1
Band	1
Be	17
Been	1

- **Performance Analysis:**

Dataset	Size	# of Files	Time Elapsed	# of Unique Words
American Pie	4 KB	1	3 ms	313
Hamlet	192 KB	1	5 ms	4835
Charles Dickens	6.4 MB	20	37 ms	45,331
BBC Tech News	1.2 MB	401	183 ms	12,673
Song Lyrics	5.6 MB	41	51 ms	27,254

As can be seen from the table above, while the size of the data set does influence in the computation time, the total number of files has a way more significant impact. That makes perfect sense since given that writing to and reading from disk are time-consuming tasks (if my machine had a traditional hard drive instead of a solid state drive this discrepancy would probably be even higher).

- **Computation Time vs # of Files Graph:**



It would be very interesting to run this program on an actual Hadoop cluster with multiple nodes and compare its performance to my single machine running Hadoop on pseudo-distributed mode. Hadoop excels in the world of Big Data by providing horizontal expansion capabilities with commodity hardware, parallel computation performed where the data resides and robust fault-tolerance.

4. MapReduce Top100Words:

This is a more complicated MapReduce problem where you take group of documents (corpus) as input and output **Top 100 words ranked in 3 different ways**:

- First, by the number of files in the corpus where that word appears.
- Second, by the total number of times that word appears in the corpus (in case multiple words appear on the same number of files).
- Third, sort the words alphabetically (as a final tie-breaker if necessary).

The final output will contain **100 words** displaying the number of documents where it appears and total number of occurrences in the corpus separated by tabs (e.g.: **World 33 215**)

In order to achieve this, I made several changes to the WordCount program:

- Used a **HashMap to count the total occurrences of each word per document** during the Mapper stage instead of writing it to the context every iteration:

```
Map<String, Integer> word_list = new HashMap<String, Integer>();

if (word.length() > 0) {
    word = word.substring(0, 1).toUpperCase() + word.substring(1).toLowerCase();

    Integer count = word_list.get(word);

    if (count == null) word_list.put(word, 1);
    else word_list.put(word, count + 1);
}
```

- **Override the Mapper's cleanup method** to write compounded sums to the context only once. This way, each word will be written to context once for every document that contains that word (containing the number of times it appears in that document):

```
@Override
protected void cleanup(Context context) throws IOException,
InterruptedException {
    for (String key : word_list.keySet()) {
        word_count.set(word_list.get(key));
        clean_word.set(key);
        context.write(clean_word, word_count);
    }
}
```

- Created the **WordTuple** class that implements the **Comparable** class inside the **Reducer** class to make the "triple sorting" process easier.

```
public class WordTuple implements Comparable<WordTuple> {

    private int id = 1;
    private String word;
    private int document_count;
    private int total_count;

    WordTuple(String word, int document_count, int total_count) {
        this.word = word;
        this.document_count = document_count;
        this.total_count = total_count;
    }

    public int getDocCount() {
        return this.document_count;
    }
}
```

```
public int getTotalCount() {
    return this.total_count;
}

public String getWord() {
    return this.word;
}

@Override
public int compareTo(WordTuple o) {
    return this.id - o.id;
}
}
```

- **Override the Reducers's cleanup method** to receive an **ArrayList of WordTuple** objects, perform the "triple sort" and write top 100 words, the number of documents they appear in and the total number of occurrences all documents separated by tabs:

```
ArrayList<WordTuple> master_list = new ArrayList<>();

@Override
protected void cleanup(Context context) throws IOException,
InterruptedException {

    Comparator<WordTuple> comparison = Comparator
        .comparing(WordTuple::getDocCount)
        .thenComparing(WordTuple::getTotalCount)
        .thenComparing(WordTuple::getWord);

    master_list.sort(comparison.reversed());

    int counter = 0;

    for (int i = 0; i < master_list.size(); i++) {
        if (counter++ > 99) break;
        String word = master_list.get(i).getWord();
        int occ = master_list.get(i).getDocCount();
        int tot = master_list.get(i).getTotalCount();
        final_result.set(Integer.toString(occ) + "\t" +
Integer.toString(tot));
        context.write(new Text(word), final_result);
    }
}
```

Additionally, I also decided to include a look-up HashSet of the 150 most common words in the English language (found at <http://shabanali.com/upload/1000words.pdf>) and exclude them from the output. Since words like "The", "Be" and "Of" are so commonly used, that they would not add much value when comparing different groups of documents.

- I achieved this by always checking if the word was **contained in the HashSet** before adding any object to the WordTuple Arraylist.

```
String[] common_words = {"The", "Of", "And", "A", "To", "In", "Is", "You",  
"That", "It", "He", "Was", "For",  
"On", "Are", "As", "With", "His", "They", "I", "At", "Be",  
"This", "Have", "From", "Or", "One",  
"Had", "By", "Word", "But", "Not", "What", "All", "Were", "We",  
"When", "Your", "Can", "Said",  
"There", "Use", "An", "Each", "Which", "She", "Do", "How",  
"Their", "If", "Will", "Up", "Other",  
"About", "Out", "Many", "Then", "Them", "These", "So", "Some",  
"Her", "Would", "Make", "Like",  
"Him", "Into", "Time", "Has", "Look", "Two", "More", "Write",  
"Go", "See", "Number", "No", "Way",  
"Could", "People", "My", "Than", "First", "Water", "Been",  
"Call", "Who", "Oil", "Its", "Now",  
"Find", "Long", "Down", "Day", "Did", "Get", "Come", "Made",  
"May", "Part", "Over", "New", "Sound",  
"Take", "Only", "Little", "Work", "Know", "Place", "Year",  
"Live", "Me", "Back", "Give", "Most",  
"Very", "After", "Thing", "Our", "Just", "Name", "Good",  
"Sentence", "Man", "Think", "Say", "Great",  
"Where", "Help", "Through", "Much", "Before", "Line", "Right",  
"Too", "Mean", "Old", "Any", "Same",  
"Tell", "Boy", "Follow", "Came", "Want", "Show", "Also",  
"Around", "Form", "Three", "Small"};  
  
Set<String> common_words_set = new  
HashSet<>(Arrays.asList(common_words));  
  
if (!common_words_set.contains(key.toString())) master_list.add(new  
WordTuple(key.toString(), doc_count, total_count))
```

- **Running the program:**
 - To run the MapReduce job, you need to change directories to Top100Words/src and run the top_100_words.sh shell script by running the following commands from the root directory of the repository:

```
cd Top100Words/src  
sh top_100_words.sh
```

In my machine, the script took **~45 seconds** to run (including the time it takes to print all the progress reports to the terminal).

- Portion of a MapReduce Top100Words job output:

Such 201	377	
Technology	196	504
Us 178	366	
Mr 175	509	
Users	173	407

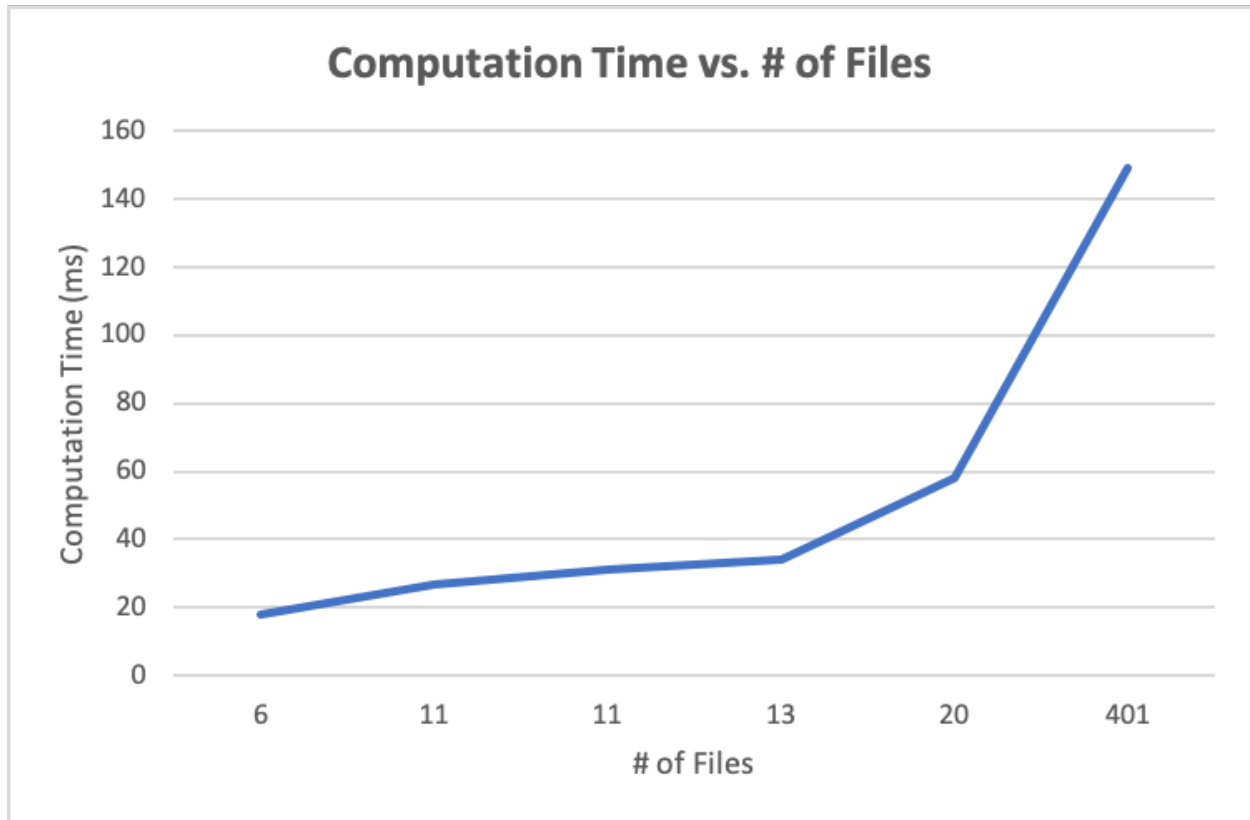
Being	169	279
Because	159	246
Used	156	281
Using	152	238
Those	149	251
Digital	139	373
Computer	136	299
Last	135	186
Million	134	253
Online	133	307

- **Performance Analysis:**

Dataset	Size	# of Files	Time Elapsed
Charles Dickens	6.4 MB	20	58 ms
BBC Tech News	1.2 MB	401	149 ms
Pop Lyrics	1.4 MB	11	27 ms
Rock Lyrics	1.4 MB	13	34 ms
Folk/Country Lyrics	672 KB	6	18 ms
Hip-Hop Lyrics	2.2 MB	11	31 ms

Once again, it can be confirmed how the total number of documents is most important factor when predicting computational time. It would also be very interesting to see how much faster this program would run in a larger scale Hadoop cluster.

- **Computation Time vs # of Files Graph:**



- **Output Analysis:**

In order to illustrate the outputs of this MapReduce job, I created a few *Word Clouds* using the website <https://worditout.com/word-cloud/>:

- **Charles Dickens' Books:**



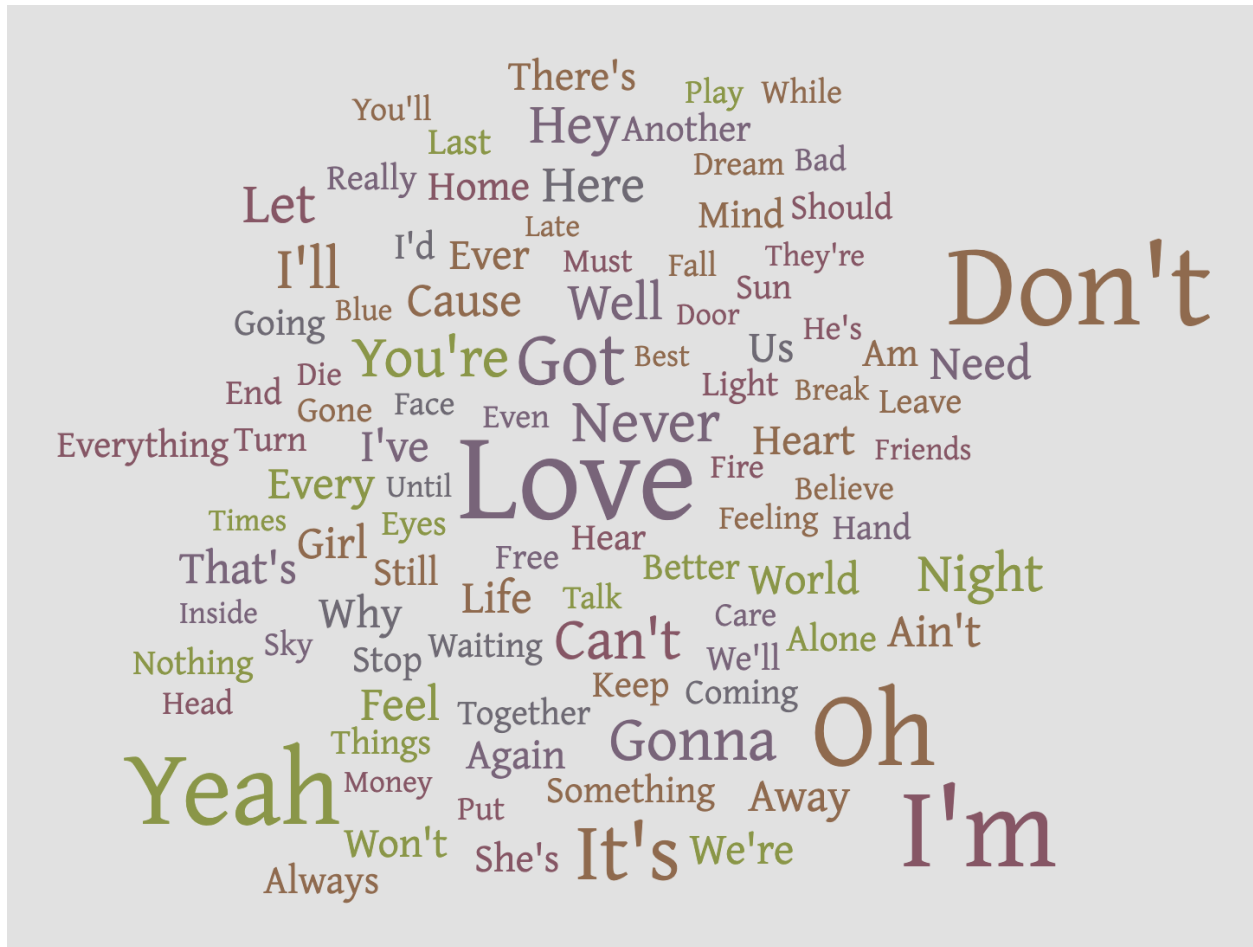
- BBC Technology News:



- Pop Song Lyrics:

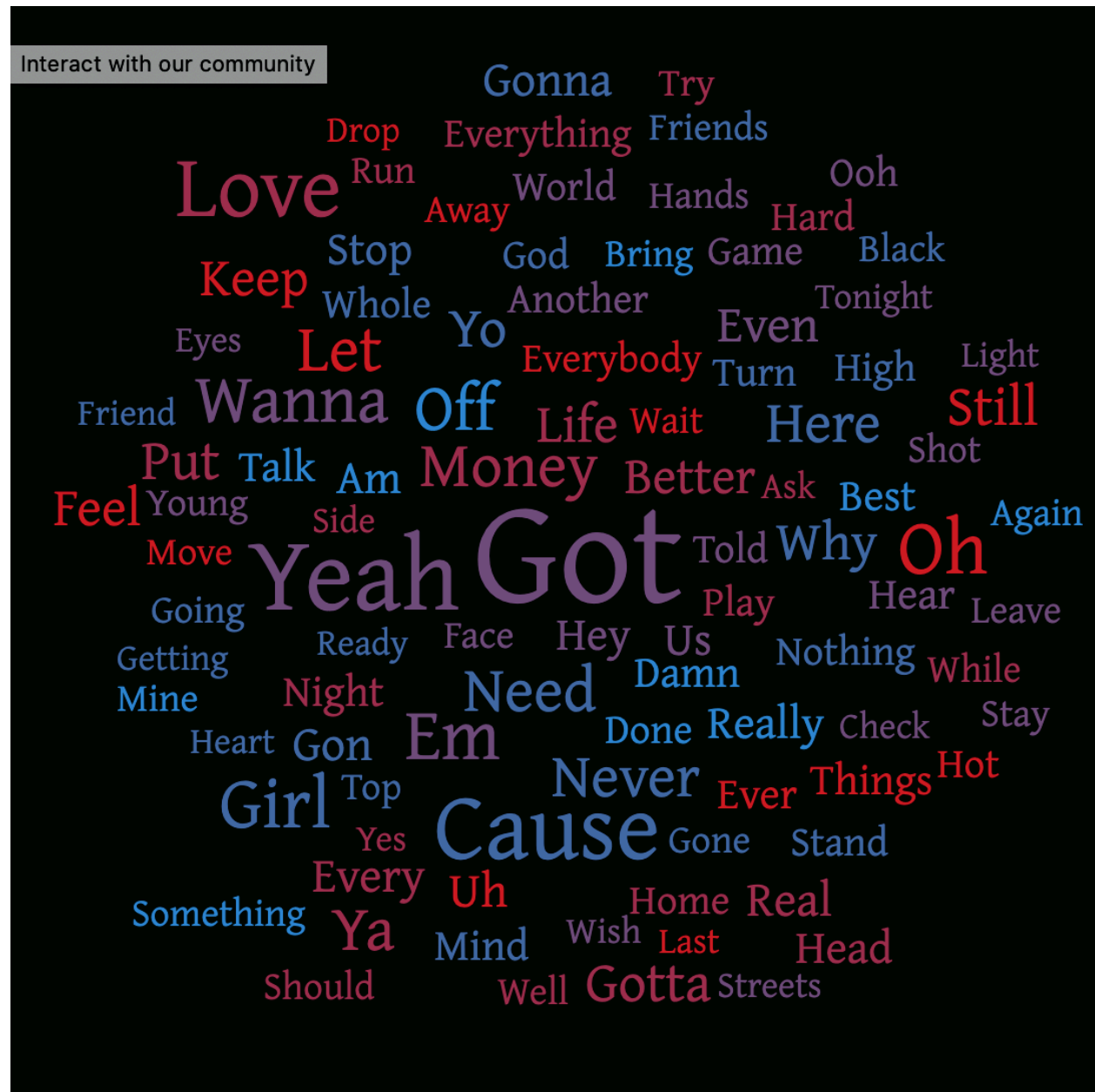


- **Rock Song Lyrics:**



- Folk/Country Song Lyrics:





- Hip-Hop Song Lyrics:























Final View of my HDFS Dashboard:

Hadoop Overview Datanodes Datanode Volume Failures Snapshot Startup Progress Utilities ▾

Browse Directory

Show 25 entries Search:

<input type="checkbox"/>	 Permission	 Owner	 Group	 Size	 Last Modified	 Replication	 Block Size	 Name	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 08:16	0	0 B	top_100_bbc_tech_news	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 08:05	0	0 B	top_100_charles_dickens	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 08:13	0	0 B	top_100_folk-country	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 08:14	0	0 B	top_100_hip-hop	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 08:13	0	0 B	top_100_pop	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 08:12	0	0 B	top_100_rock	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 06:36	0	0 B	word_count_american_pie	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 06:37	0	0 B	word_count_bbc_tech_news	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 06:37	0	0 B	word_count_charles_dickens	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 06:36	0	0 B	word_count_hamlet	
<input type="checkbox"/>	drwxr-xr-x	pedropinto	supergroup	0 B	Sep 19 06:38	0	0 B	word_count_song_lyrics	

Showing 1 to 11 of 11 entries Previous **1** Next

Hadoop, 2020.

Conclusion:

This Homework was an amazing hands-on experience with big data processing in Hadoop. For the first time, I fully installed HDFS from scratch in my local machine, developed some really interesting MapReduce programs and tested them out with a variety of textual data sets. It was clear that, while the size of the files does influence performance, the number of files being processed influences way more due to the long time it takes to read/write to disk.

I'm looking forward to building on top of this knowledge in future homeworks by building more intricate MapReduce programs, comparing its performance with other tools like Spark and start performing more in depth analytics using libraries such as Apache Mahout and Spark ML lib.