Big Data Processing and Analytics: Assignment 1

Louis MARTIN
louis.martin@student.ecp.fr

March 6, 2017

In this assignment we are going to create an inverted index on a corpus including the complete works of William Shakespear, Mark Twain and Jane Austen using Hadoop's MapReduce framework.
The source code is available on Github.

1 Setup

(a) System specifications

- **Operating system**:
  Ubuntu 16.04 (Native)
- **System specifictations**:
  Model: Dell Inspiron 17R 5720
  Processor: i5-3210M
  Cores: 2
  Threads: 4
  Ram: 8 GB
  Storage: 256GB SSD (MLC)
- **Java version**:
  openjdk version "1.8.0_121"
  OpenJDK Runtime Environment (build 1.8.0_121-8u121-b13-0ubuntu1.16.04.2-b13)
  OpenJDK 64-Bit Server VM (build 25.121-b13, mixed mode)
- **Haddop version**:
  Hadoop 2.7.3
  Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r baa91f7c6bc9cb92be5982de4719c1c8af91ccff
  Compiled by root on 2016-08-18T01:41Z
  Compiled with protoc 2.5.0
  From source with checksum 2e4ce5f957ea4db193bce3734ff29ff4
  This command was run using /usr/local/hadoop/share/hadoop/common/hadoop-common-2.7.3.jar

Hadoop was installed using this tutorial and configured using this tutorial.

(b) Configuration

The configuration comes from the official documentation for a single Hadoop node cluster. The following configuration files allows Hadoop and YARN to run in a pseudo-distributed mode.

- **core-site.xml:**

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
```

```
    </property>
</configuration>
```

- **hdfs-site.xml:**

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

- **mapred-site.xml:**

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

- **yarn-site.xml:**

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>

    <!--
    To prevent unhealthy nodes and jobs hanging due to low disk space
    http://stackoverflow.com/questions/29131449/why-does-hadoop-report-
        unhealthy-node-local-dirs-and-log-dirs-are-bad
    -->
    <property>
        <name>yarn.nodemanager.disk-health-checker.max-disk-utilization-per-
            disk-percentage</name>
        <value>98.5</value>
    </property>
</configuration>
```

- **Commands to set up HDFS and YARN:**

```
    # Format the filesystem
    hdfs namenode -format
    # Start HDFS
    start-dfs.sh

    # Create directories to execute MapReduce jobs
    hdfs dfs -mkdir /user
    hdfs dfs -mkdir /user/louis

    # Put the data in HDFS
    hdfs dfs -put ~/dev/bdpa/a1/data data

    # Start YARN Ressource manager
    start-yarn.sh
```

(c) Bugs encountered

I encountered a bug while running the MapReduce jobs where I would be logged out and all my processes would be killed with no warning. After investigation, I found that this is a known bug referenced here and caused by /bin/kill in ubuntu 16.04.

I compiled propcps-3.3.10 from source to solve the problem as indicated in launchpad but to no avail.

```
# (1) download the sourcecode
sudo apt-get source procps

# (2) install dependency
sudo apt-get build-dep procps

# (3) compile procps
cd procps-3.3.10
sudo dpkg-buildpackage
```

I tried another solution from this stackoverflow thread but it didn't work either. The supposed fix was to add

```
[login]
KillUserProcesses=no
```

to /etc/systemd/logind.conf and restart.

As I could find a solution I had to cope with the problem and restart all the services every once in a while.

2 Inverted index

Workflow

In order to facilitate compiling the *.java* files into *.jar* files, I created the *compile.sh* script which takes as input parameter the name of the main class of the *.java* file.
Example: ./compile.sh StopWords.
compile.sh:

```
#!/bin/bash
rm -rf *.class *.jar;
hadoop com.sun.tools.javac.Main $1.java;
jar cf $1.jar $1*.class;
```

Assumptions

After investigating the results of the different mapreduce jobs, I chose to define all of the following characters as words separators in addition to space
Separators: .,?!"'()[]$*-_;:|

Even if I lost some hyphenated words and contractions like "first-born" or "He's", the gain in coherence and clarity was worth the hassle.

(a) Stop words
(30) Run a MapReduce program to identify stop words (words with frequency $>$ 4000) for the given document corpus. Store them in a single csv file on HDFS (stopwords.csv). You can edit the several parts of the reducers' output after the job finishes (with hdfs commands or with a text editor), in order to merge them as a single csv file.

Based on the wordcount example from the official documentation, we implement a MapReduce program that retrieves all the stopwords from a corpus, i.e. it retrieves the words with wordcount greater than 4000.
The file is **StopWords.java**

- **Mapper**:
  This mapper splits a string into words (tokens) and outputs one (key, value) pair for each word with the key being the word and the value equal to 1.

```java
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      // Splits a string to tokens (here words)
      StringTokenizer itr = new StringTokenizer(value.toString(), " .,?!\"'()[]$*-_;:|");
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken().toLowerCase().trim());
        // Write one (key, value) pair to context
        context.write(word, one);
      }
    }
  }
```

- **Reducer**:
  The reducers, simply counts the number of occurences of each key writes it to the output file only if its count is greater than 4000.

```java
  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable(10);

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      // Only write the key value pair if its frequency is high enough
      if (sum > 4000){
        result.set(sum);
        context.write(key, result);
      }
    }
  }
```

- **Job configuration**:
  The MapReduce task is set to write the output as a csv file. We can set the number of reducers, combiner and compression through command line arguments.

```java
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    // Remove output folder if it exists
    Path output = new Path(args[1]);
    FileSystem hdfs = FileSystem.get(conf);
    // delete existing directory
    if (hdfs.exists(output)) {
```

```
            hdfs.delete(output, true);
        }

        // Set separator to write as a csv file
        conf.set("mapred.textoutputformat.separator", ",");
        // Set compression
        if ((args.length >= 5) && (Integer.parseInt(args[4]) == 1)) {
            conf.set("mapreduce.map.output.compress", "true");
        }

        Job job = Job.getInstance(conf, "stop words");
        job.setJarByClass(StopWords.class);

        // Set number of reducers and combiner through cli
        if (args.length >= 3) {
            job.setNumReduceTasks(Integer.parseInt(args[2]));
        }
        if ((args.length >= 4) && (Integer.parseInt(args[3]) == 1)) {
            job.setCombinerClass(IntSumReducer.class);
        }
        job.setMapperClass(TokenizerMapper.class);
        job.setReducerClass(IntSumReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        long startTime = System.nanoTime();
        if (job.waitForCompletion(true)) {
          long endTime = System.nanoTime();
          float duration = (endTime - startTime);
          duration /= 1000000000;
          System.out.println("***** Elapsed: " + duration + "s *****\n");
          System.exit(0);
        }
        else {
          System.exit(1);
        }
    }
}
```

- **Script running the experiments**:
  We run the set of experiments with the **run_stopwords.sh** script. This script executes the StopWords MapReduce task with different parameters and merges all the outputs into a csv file in HDFS.

```
#!/bin/bash
./compile.sh StopWords

# Usage:
# hadoop jar myfile.jar Class input_dir output_dir n_reducers combiner
    compression

# 10 reducers no combiner
hadoop jar StopWords.jar StopWords data/corpus/ out 10 0 0;
hdfs dfs -getmerge out stopwords.csv;

# 10 reducers, combiner
```

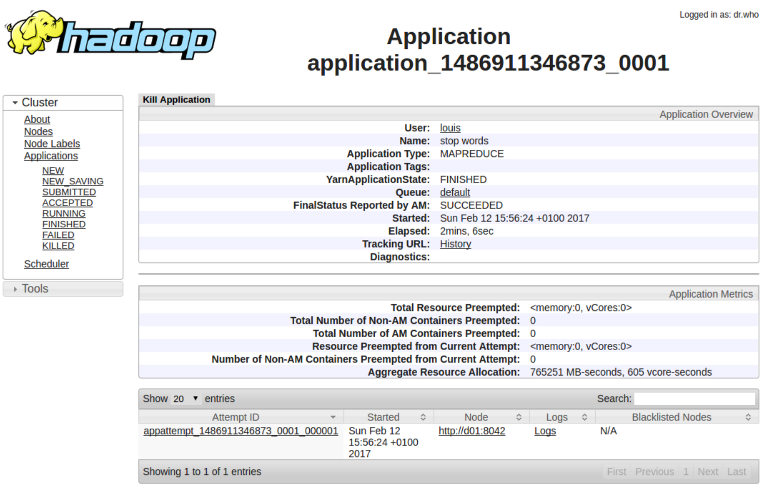
```
hadoop jar StopWords.jar StopWords data/corpus/ out 10 1 0;
hdfs dfs -getmerge out stopwords.csv;

# 10 reducers, combiner, compression
hadoop jar StopWords.jar StopWords data/corpus/ out 10 1 1;
hdfs dfs -getmerge out stopwords.csv;

# 50 reducers, combiner, compression
hadoop jar StopWords.jar StopWords data/corpus/ out 50 1 1;
hdfs dfs -getmerge out stopwords.csv;
```

- **Results**: Here is an extract of the output csv:

```
with, 35591
good, 4632
from, 9677
has, 5190
its, 4569
man, 5267
made, 4046
not, 35143
said, 8434
have, 24625
my, 27231
```

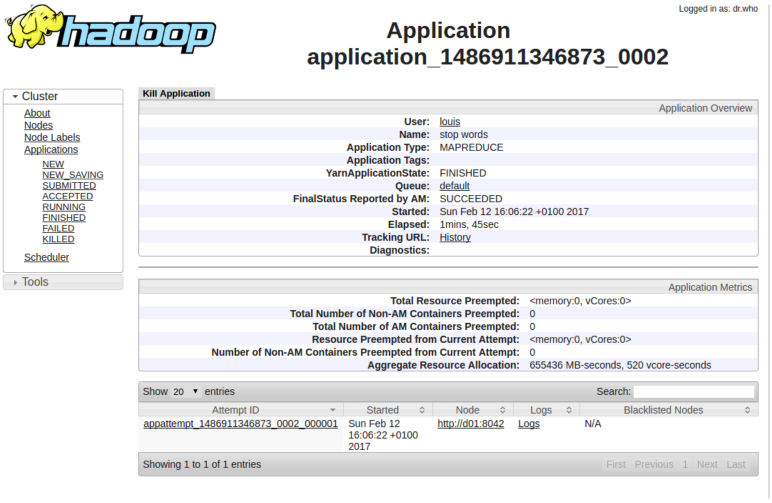
i. (10) Use 10 reducers and do not use a combiner. Report the execution time.

The running time with 10 reducers is **126 seconds**.



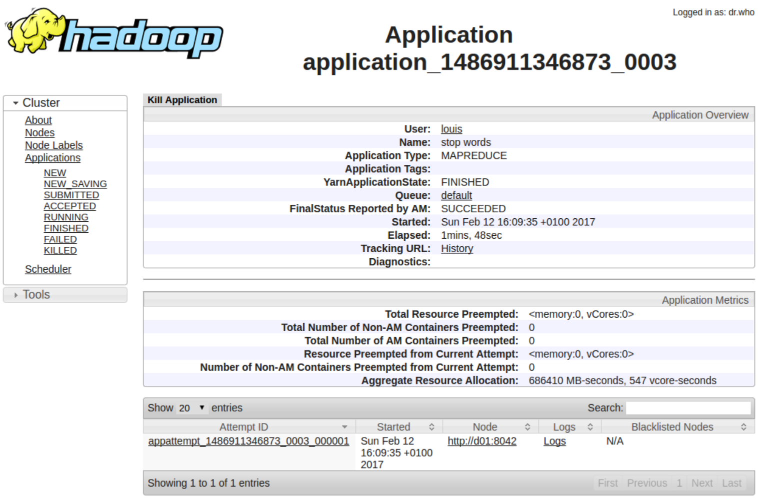
ii. (10) Run the same program again, this time using a Combiner. Report the execution time. Is there any difference in the execution time, compared to the previous execution? Why?

The running time with 10 reducers and combiner is **105 seconds**. The running time is lower with the combiner. Indeed the combiner takes the output of each mapper separately and tries to reduce as many (key, value) pairs coming from this specific mapper as it can. Combining the data like this will lower the load for the reducers, and all the intermediary steps between the mappers and the reducers such as network transfer (does not occur for a pseudo-distributed cluster), sorting all the (key, value) pairs, assigning them to each reducer...



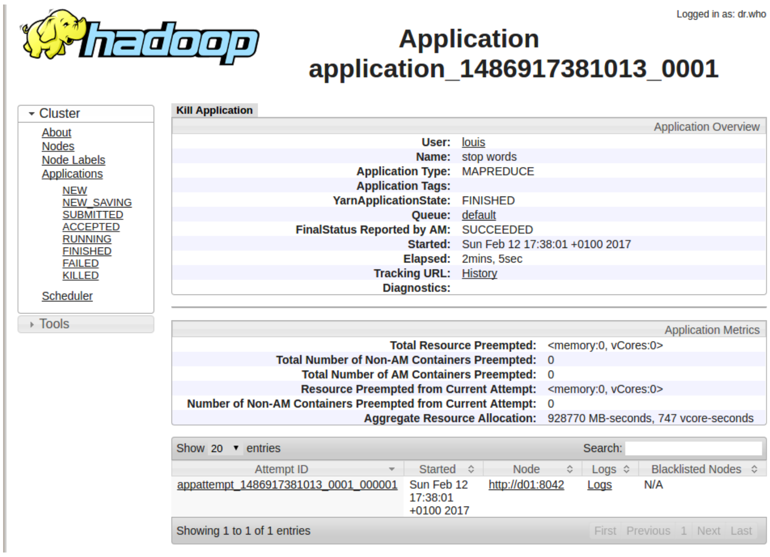
iii. (5) Run the same program again, this time compressing the intermediate results of map (using any codec you wish). Report the execution time. Is there any difference in the execution, time compared to the previous execution? Why?

Compression compresses the data between the mappers and the reducers. The running time with 10 reducers, combiner and compression is **125 seconds**. The running time increased. This can be explained by the fact that compression adds an additional overhead and its benefit is not used for a single node cluster! Indeed compression is often used to reduce network transfer times, which does not occur here as all the mappers and reducers are on the same device.



iv. (5) Run the same program again, this time using 50 reducers. Report the execution time. Is there any difference in the execution time, compared to the previous execution? Why?

The running time with 50 reducers, combiner and compression is **126 seconds**. The running time is about the same as the previous experiment. We did not benefit from the additional number of reducers. This is logical because with a single node cluster, the processing power caps with the limited number of cores. No additional gain is achieved with this extra parallelization.



(b) (30) Implement a simple inverted index for the given document corpus, as shown in the previous Table, skipping the words of stopwords.csv.

The inverted index is implemented in **InvertedIndex.java**.

- **readStopWords function:**
  First we need to read the stopwords from the csv file. We do it with the readStopWords function.

```java
    public static HashSet<String> readStopWords() {
        // Read stopwords into a HashSet for fast membership testing
        // The hashtable underlying structure provides O(1) membership testing
        HashSet<String> stopWords = new HashSet<String>();

        // Read csv file: inspired from https://www.mkyong.com/java/how-to-
            read-and-parse-csv-file-in-java/
        String csvFile = "stopwords.csv";
        BufferedReader br = null;
        String line = "";
        String cvsSplitBy = ",";

        try {
            br = new BufferedReader(new FileReader(csvFile));
            while ((line = br.readLine()) != null) {
                // use comma as separator
                String[] splittedLine = line.split(cvsSplitBy);
                stopWords.add(splittedLine[0]);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
```

```
        }
        return stopWords;
    }
```

- **Mapper:**
  Our mapper here outputs only words which are not in stopwords.csv as keys and the file from which they came from as values (formatted as a posting list). These posting lists are implemented using a custom class of MapWritable in order to have all the ComparableWritable properties that Hadoop's MapReduce needs (more details below). The posting list returned by the mapper is just a map containing one element which is a (key, value) pair of the form (filename, 1).

```
    public static class TokenizerMapper
         extends Mapper<Object, Text, Text, PostingListWritable>{
      private Text word = new Text();
      private Text doc = new Text();
      private IntWritable one = new IntWritable(1);
      private PostingListWritable postingList = new PostingListWritable();
      private HashSet<String> stopWords = InvertedIndex.readStopWords();

      public void map(Object key, Text value, Context context
                      ) throws IOException, InterruptedException {
        FileSplit fileSplit = (FileSplit) context.getInputSplit();
        String filename = fileSplit.getPath().getName();
        doc.set(filename);

        // Splits a string to tokens (here words)
        StringTokenizer itr = new StringTokenizer(value.toString(), " .,?!\"'()[]$*-_;:|");
        String token = new String();
        while (itr.hasMoreTokens()) {
          token = itr.nextToken().toLowerCase().trim();
          if (!stopWords.contains(token)) {
            word.set(token);
            // Output is a PostingListWritable containing the filename and the value 1
            postingList.clear();
            postingList.put(doc, one);
            // Write one (key, value) pair to context
            context.write(word, postingList);
          }
        }
      }
    }
```

- **Reducer:**
  Our reducer reduces all the posting lists it receives and also counts the number of occurences of each word for the following frequency part (more details below).

```
      public void reduce(Text word, Iterable<PostingListWritable> postingLists,
                         Context context
                         ) throws IOException, InterruptedException {
        // We are going to aggregate all posting lists together
        postingList.clear();
        // Iterate through all posting lists comming from the mappers or the combiners
        for (PostingListWritable mw : postingLists) {
```

```
        // Iterate through the entries of one given posting list
        for (PostingListWritable.Entry<Writable, Writable> entry : mw.entrySet()) {
          Text doc = (Text) entry.getKey();
          IntWritable newValue = (IntWritable) entry.getValue();
          if (postingList.containsKey(doc)) {
            IntWritable oldValue = (IntWritable) postingList.get(doc);
            postingList.put(doc, new IntWritable(oldValue.get() + newValue.get()));
          } else {
            postingList.put(doc, new IntWritable(newValue.get()));
          }
        }
      }

      isLastWordInUniqueDoc = (postingList.size() == 1);
      context.write(word, postingList);
    }
  }
```

(c) (10) How many unique words exist in the document corpus (excluding stop words)? Which counter(s) reveal(s) this information? Define your own counter for the number of words appearing in a single document only. What is the value of this counter? Store the final value of this counter on a new file on HDFS.

The counter that counts the unique words in the documents is **TaskCounter.REDUCE_INPUT_GROUPS**. This counter counts the number of keys which is exactly the number of unique words.

In order to count the words appearing in a single document only, we implement the following counter:

```
  public static enum WordCounter {
    WORDS_IN_UNIQUE_DOC
  };
```

Which is incremented in the reducer's reduce function:

```
      if (isLastWordInUniqueDoc) {
        // Increment counter for word appearing in a single document only
        // We are guaranteed that this is the only time that the counter
        // will be incremented for this word because all the values from
        // a given key all go to the same reduce call.
        context.getCounter(WordCounter.WORDS_IN_UNIQUE_DOC).increment(1);
      }
```

The counter gives us a count **36343 words appearing in a single document only** which seems pretty high given the number of total unique words 56491 (excluding about 140 stopwords).

After investigation, a lot of words appear only in pg3200.txt which are the Mark Twain works. This file is roughly 300.000 lines long, which is about three times longer than the other files. This might explained why it contains a lot more vocabulary specific than the rest.

The counter value is stored in a file in HDFS:

```
    // Write the count of unique words to a file in HDFS
    Path filePath = new Path("words_in_unique_file.txt");
    if (hdfs.exists(filePath)) {
        hdfs.delete(filePath, true);
    }
    FSDataOutputStream fin = hdfs.create(filePath);
    fin.writeUTF(message);
```

```
        fin.close();
```

(d) (30) Extend the inverted index of (b), in order to keep the frequency of each word for each document. You are required to use a Combiner.

- **PostingListWritable:**
  As explained before we implemented a custom class extending MapWritable in order to have all the ComparableWritable properties that hadoop needs and to be able to use a combiner along with a reducer without losing the word frequencies during the transfer. The custom class allows pretty printing a MapWritable instance to a file.

```
  public static class PostingListWritable extends MapWritable {
    // Creates a custom class that overrides the toString method to pretty
    // print the posting list
    @Override
    public String toString() {
      String stringRepr = new String();
      for (PostingListWritable.Entry<Writable, Writable> entry : this.entrySet()) {
        stringRepr += entry.getKey() + "#" + entry.getValue() + ", ";
      }
      // Remove last comma and space of string
      stringRepr = stringRepr.substring(0, stringRepr.length()-2);
      return stringRepr;
    }
  }
```

- **Combiner:**
  Different posting lists coming from different mappers or combiners can be reduced into a single one using the reduce function of the combiner.

```
  public static class PostingListCombiner
       extends Reducer<Text, PostingListWritable, Text, PostingListWritable> {
    private PostingListWritable postingList = new PostingListWritable();
    protected boolean isLastWordInUniqueDoc = false;

    public void reduce(Text word, Iterable<PostingListWritable> postingLists,
                       Context context
                       ) throws IOException, InterruptedException {
      // We are going to aggregate all posting lists together
      postingList.clear();
      // Iterate through all posting lists comming from the mappers or the combiners
      for (PostingListWritable mw : postingLists) {
        // Iterate through the entries of one given posting list
        for (PostingListWritable.Entry<Writable, Writable> entry : mw.entrySet()) {
          Text doc = (Text) entry.getKey();
          IntWritable newValue = (IntWritable) entry.getValue();
          if (postingList.containsKey(doc)) {
            IntWritable oldValue = (IntWritable) postingList.get(doc);
            postingList.put(doc, new IntWritable(oldValue.get() + newValue.get()));
          } else {
            postingList.put(doc, new IntWritable(newValue.get()));
```

```
                }
            }
        }

        isLastWordInUniqueDoc = (postingList.size() == 1);
        context.write(word, postingList);
    }
}
```

- **Reducer:**
  The reducer shares the reduce function (it is a child of the combiner) with the combiner with the only exception that it will increment the counter for words in unique documents (which should only occur once per key and therefore only in the reducer).

```
public static class PostingListReducer
        extends PostingListCombiner {
    // The only thing different between the reducer and the combiner is the
    // counter incrementation. It must be incremented only once per key, hence in
    // the reducer.
    @Override
    public void reduce(Text word, Iterable<PostingListWritable> postingLists,
                                Context context
                                ) throws IOException, InterruptedException {
        super.reduce(word, postingLists, context);
        if (isLastWordInUniqueDoc) {
            // Increment counter for word appearing in a single document only
            // We are guaranteed that this is the only time that the counter
            // will be incremented for this word because all the values from
            // a given key all go to the same reduce call.
            context.getCounter(WordCounter.WORDS_IN_UNIQUE_DOC).increment(1);
        }
    }
}
```

- **Results**: Here is an extract of the output inverted index csv:

```
enactment : pg3200.txt#2
enchant : pg100.txt#4, pg3200.txt#2
enchanting : pg31100.txt#1, pg100.txt#6, pg3200.txt#58
encircled : pg100.txt#1, pg3200.txt#5
enclosed : pg31100.txt#9, pg100.txt#5, pg3200.txt#37
enclouded : pg100.txt#1
encroach : pg31100.txt#2, pg3200.txt#2
encyclopaedic : pg3200.txt#2
endeavored : pg3200.txt#12
endeavors : pg31100.txt#1, pg3200.txt#10
endlich : pg3200.txt#1
endue : pg100.txt#2
enforc : pg100.txt#15
enforcing : pg31100.txt#2, pg3200.txt#1
enfreedoming : pg100.txt#1
engineering : pg3200.txt#21
```

3 Conclusion

Inverted indexes are widely used for text retrieval or for search engines for the performance gain they provide. Creating an inverted index can however be a computing power intensive task but it can be highly parallelized. Hadoop's MapReduce framework is a perfect candidate for parallelizing this kind of task on a highly distributed cluster with no centralized data.