# MASSIVE DATA PROCESSING ( Ecole Centrale )

## ASSIGNMENT 1 - HADOOP

Raunaq Paul

## 1. OBJECTIVE

The objective of the assignment is to get used to working with Haddop Framework and execute wordcount and map reduce jobs on given text files. The following are the questions answered in this assignment:

We are asked to implement an inverted index in MapReduce for the document corpus of pg100.txt (from http://www.gutenberg.org/cache/epub/100/pg100.txt), pg31100.txt (from http://www.gutenberg.org/cache/epub/31100/pg31100.txt) and pg3200.txt (from http://www.gutenberg.org/cache/epub/3200/pg3200.txt). This corpus includes the complete works of William Shakespear, Mark Twain and Jane Austen, respectively.

An inverted index (also referred to as postings file or inverted file) is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents.The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database. The inverted file may be the database file itself, rather than its index. It is the most popular data structure used in document retrieval systems,used on a large scale for example in search engines.

It provides for each distinct word in a document corpus, the filenames that contain this word, along with some other information (e.g., count/position within each document).

For example, assume you are given the following corpus, consisting of doc1.txt, doc2.txt and doc3.txt:

- doc1.txt: “This is a very useful program. It is also quite easy.”
- doc2.txt: “This is my first MapReduce program.”
- doc3.txt: “Its result is an inverted index.”

An inverted index would contain the following data (in random order):

| Word | FileName |
|---|---|
| this | doc1.txt, doc2.txt |
| is | doc1.txt, doc2.txt, doc3.txt |
| a | doc1.txt |
| program | doc1.txt, doc2.txt |

Before the inverted index, we will also be executing wordcount jobs in order to find StopWords or frequently appearing words which do not affect the content of the documents much e.g. articles.

# 2. Initializaiton

First we create a new workspace called MDP_Assignment1 from the Cloudera Terminal. Then, we create an input and output directory in our workspace:

```
cd workspace/InvertedIndex
mkdir input
mkdir output
```

We then download the text files and put the, in the input directory of our MDP_Assignment1 workspace. We can now put the input text files into Hadoop:

```
[cloudera@quickstart ~]$ hadoop fs -mkdir input
[cloudera@quickstart ~]$ hadoop fs -put /home/cloudera/workspace/MDP_Assignment1/input/pg100
[cloudera@quickstart ~]$ hadoop fs -put /home/cloudera/workspace/MDP_Assignment1/input/pg320
[cloudera@quickstart ~]$ hadoop fs -put /home/cloudera/workspace/MDP_Assignment1/input/pg311
[cloudera@quickstart ~]$ hadoop fs -ls input
Found 3 items
-rw-r--r--   1 cloudera cloudera    5589917 2017-02-16 02:51 input/pg100.txt
-rw-r--r--   1 cloudera cloudera    4454075 2017-02-16 02:52 input/pg31100.txt
-rw-r--r--   1 cloudera cloudera   16013958 2017-02-16 02:52 input/pg3200.txt
```

We are now ready to start creating our jobs for the assignment.

## 2.1. Question (a):

**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.**

We will now implement our map reduce program. We will try to find the stopwords in our corpus by finding the wordcounts for the entire corpus first and then separating the words with frequency > 4000 a "stopwords" by adding the condition.

For this, we reused the code from Standford http://snap.stanford.edu/class/cs246-data-2014/WordCount.java given in the CS246: Mining Massive Datasets Hadoop tutorial.

```
package edu.stanford.cs246.wordcount;

import java.io.IOException;
import java.util.Arrays;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {
   public static void main(String[] args) throws Exception {
      System.out.println(Arrays.toString(args));
      int res = ToolRunner.run(new Configuration(), new WordCount(), args);

      System.exit(res);
   }

   @Override
   public int run(String[] args) throws Exception {
      System.out.println(Arrays.toString(args));
      Job job = new Job(getConf(), "WordCount");
      job.setJarByClass(WordCount.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);

      job.setMapperClass(Map.class);
      job.setReducerClass(Reduce.class);

      job.setInputFormatClass(TextInputFormat.class);
      job.setOutputFormatClass(TextOutputFormat.class);
```

```
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);

        return 0;
    }

    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable ONE = new IntWritable(1);
        private Text word = new Text();

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            for (String token: value.toString().split("\\s+")) {
                word.set(token);
                context.write(word, ONE);
            }
        }
    }

    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
}
```

The changes made to the above code are as follows:

- Added the following code to the Main Class ( Driver):

  ```
  job.getConfiguration().set("mapreduce.output.textoutputformat.separator", ",");
  ```

  The above code creates our output in the form of a csv format as needed per the task.

- toLowerCase() method and replaceAll("[^a-zA-Z ]", "") in the words iteration in the Mapper function in order to remove redundancy/duplicates and only have alphabetical outputs:

```
public static class StubMapper extends Mapper<LongWritable, Text, Text, IntWritable>
    {
                private final static IntWritable ONE = new IntWritable(1);
                private Text word = new Text();

            @Override
             public void map(LongWritable key, Text value, Context context)
                 throws IOException, InterruptedException
            {
                                for (String token : value.toString().replaceAll("[^a-zA-Z ]'
                                {
                                   word.set(token.toLowerCase());
                                   context.write(word, ONE);
                                }
            }
       }
```

- if statement in the Reducer class that selects only the stopwords ( words with Frequency>4000, before writing the (key,value) outputs