Massive Data Processing Assignment 1

Samou Timothée

February 16, 2017

Abstract

This assignment is a basic application of Hadoop framework in JAVA based on several text corpus. We are going to implement word count techniques, inverted index and a relative frequencies calculator.
Github : https://github.com/timothee001/HadoopSAMOUHM

Contents

1 Setup and configuration

1.1 Hardware and Virtual Machine

I used Eclipse on a Cloudera Virtual Machine with a Macbook Air late 2013 with I7 processor and 8GO of RAM

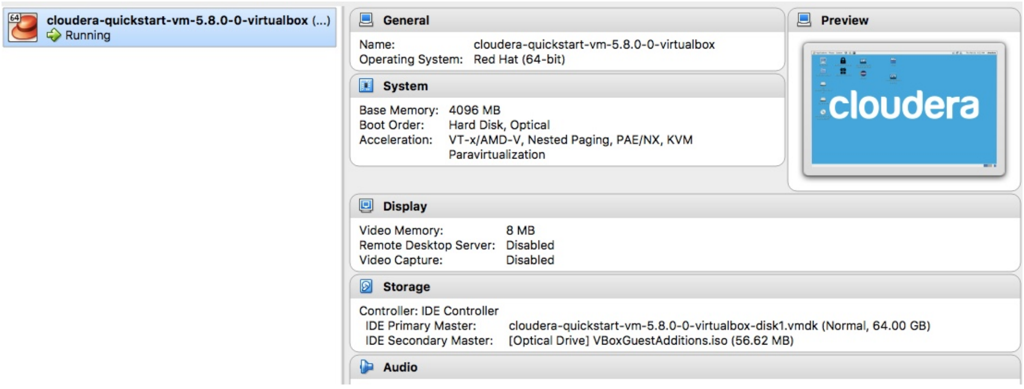


Figure 1: Virtual Machine Configuration

I used 4GO of dedicated RAM because otherwise it would have been to slow

Figure 2: Hadoop Version

1.2 Ressource Manager

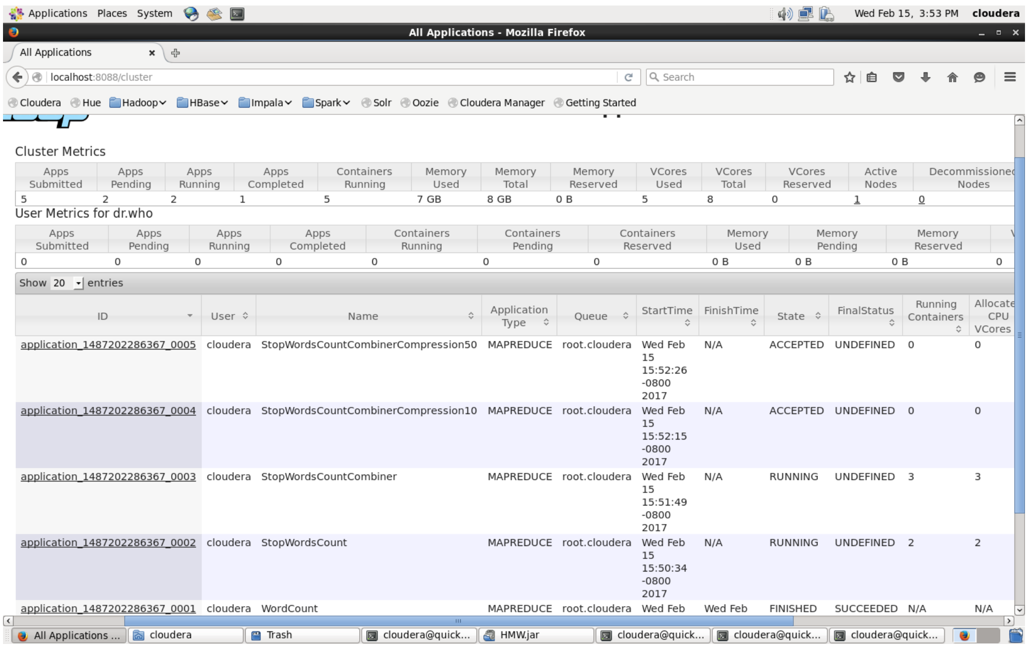


Figure 3: Ressource Manager

By typing localhost:8088 on the webbrowser we can access to the **Resource Manager** to see all jobs status. In order to make the job appear on this manager, I had to compile the project as a JAR file and run command like **hadoop jar HMW.jar stopwords.StopWordsCountCombinerCompression50** to target a specific java file of my project. Otherwise the job will launch locally on eclipse with no way for tracking it.

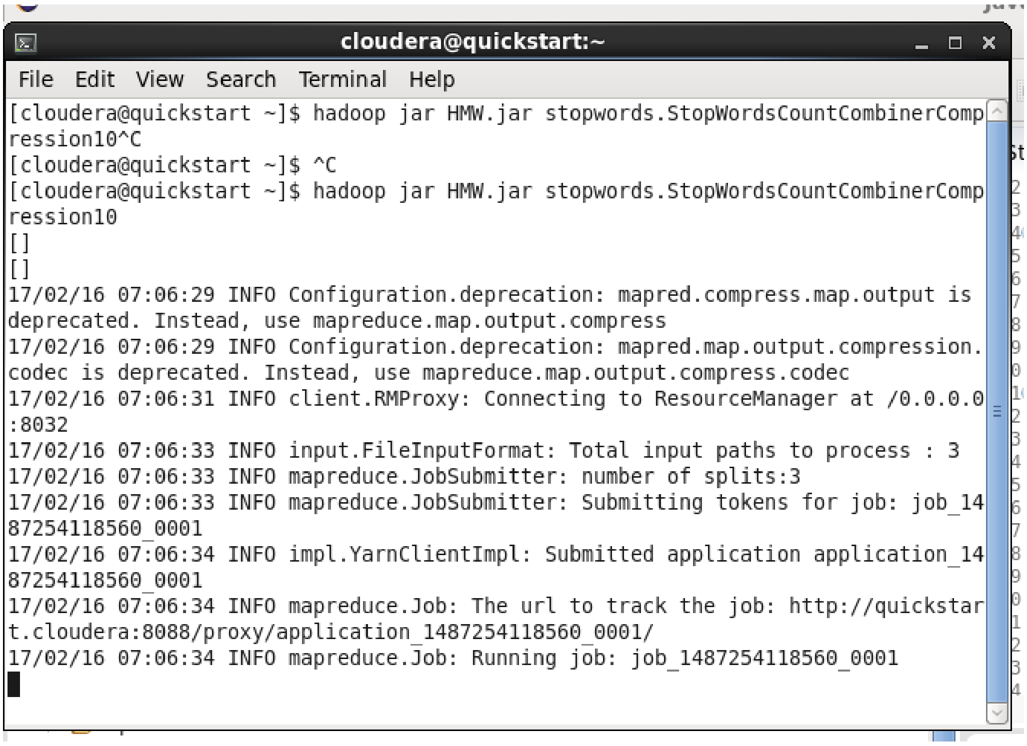


Figure 4: Launching a job

1.3 Job Configuration

Listing 1: Job config

```java
public static void main(String[] args) throws Exception {
      //System.out.println(Arrays.toString(args));
      int res = ToolRunner.run(new Configuration(), new StopWordsCount(), args);

      System.exit(res);
   }

   @Override
   public int run(String[] args) throws Exception {
      System.out.println(Arrays.toString(args));
      Configuration conf = getConf();
      conf.set("mapred.textoutputformat.separatorText", ",");
      Job job = new Job(conf, "StopWordsCount");
      job.setNumReduceTasks(1);
      job.setJarByClass(StopWordsCount.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("input"));
    Path outputPath = new Path("outputStopWordsCount");
    FileOutputFormat.setOutputPath(job, outputPath);
    FileSystem hdfs = FileSystem.get(getConf());
   if (hdfs.exists(outputPath))
      hdfs.delete(outputPath, true);

    job.waitForCompletion(true);

    return 0;
 }
```

For every job, we specified the type of every input and ouput, the number of reducers but also the Paths. Inputs are the same (the three texts corpus : pg100.txt, pg3200.txt, pg31100.txt). I decided to specify a different folder for each job output to have a better structure. We also make sure that every time a job is runned the previous folder is erased.

2 What helped me for this homework

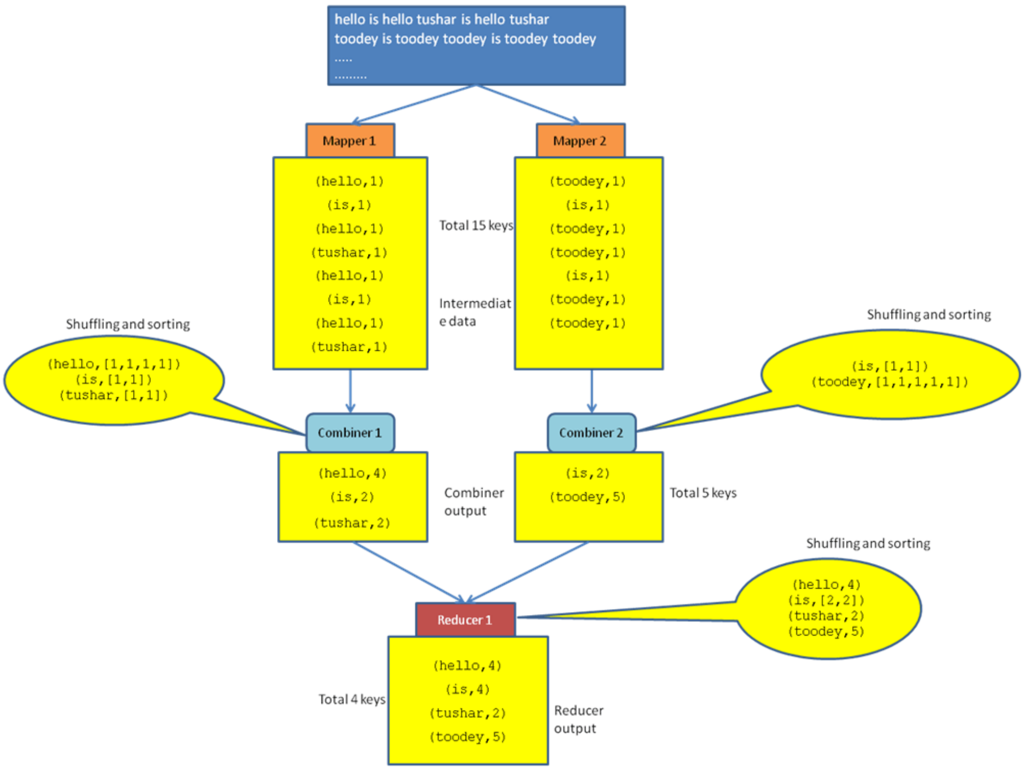


Figure 5: How works Hadoop

In order to do every part of this assignment, I used this comprehensive figure. My methodology was to start from the expected result and building the Hadoop code by back-propagating to the input.

3 Stop Words identification

3.1 Mapper and Text preprocessing

We made the assumption that a word doesn't contains special characters, single or multiple space and uppercase letters, so I used regex function to process it :

Listing 2: Preprocessing text

```
for (String token: value.toString().split("\\s+")) {
        token = token.replaceAll("[^a-zA-Z ]", "").toLowerCase();
        word.set(token);
        context.write(word, ONE);
}
```

This mapper is same for all MapReduce class for the stopword counting part, every time we have a simple word with no special characters and in lower case we emit a Key,Value pair (word,1) of type (Text,LongWritable)

Listing 3: Mapper

```
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+")) {
           token = token.replaceAll("[^a-zA-Z ]", "").toLowerCase();
              word.set(token);
              context.write(word, ONE);
           }
        }
   }
```

The Mapper has a Line and a Line number as input, that's why we split it.

3.2 10 reducers and no combiner

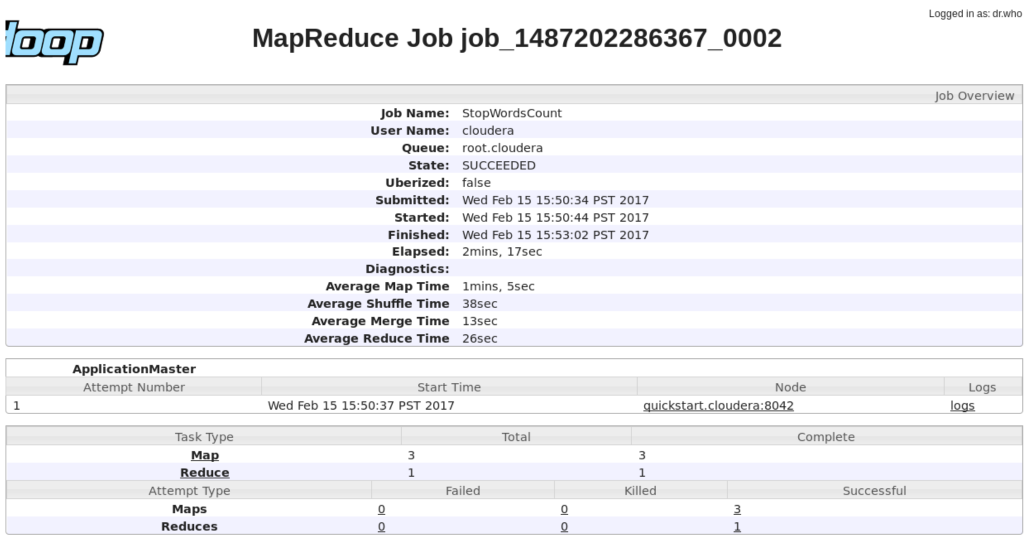


Figure 6: Word Count with 10 Reducers and no Combiner

The time is 2min 17secs

In the reducer, we have handle a Key linked with list ofvalue, but in this case value are all same equal to ONE, we just have to emit the key if the size of the list is greater than 4000.

Listing 4: Reduce function

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        int sum = 0;
```

```
    for (IntWritable val : values) {
       sum += 1;
    }
    if(sum>4000)
     context.write(key, new IntWritable(sum));
  }
```

3.3 With a combiner

Listing 5: Combiner

```
public static class Combine 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 += 1;
       }

       context.write(key, new IntWritable(sum));
          //System.out.println("combine text : " + key + " value : "+ sum);
    }
  }
```

The combiner role is just to do an intermediate processing for every mapper so that the Values are already summed before they are send to the reducer. For instance, instead of having several instance of [Hello,1], [Hello,2] we directly have [Hello,3] and this is key is then unique in the combiner. The reducer still has too sum those Hello at the end. But the list size are then smaller but with bigger values. That is why this time we don't compute the size of the list but the sum of all values of the list.

Listing 6: Compute sum in the reducer

```
    int sum = 0;
      for (IntWritable val : values) {
         sum += val.get();
      }
```

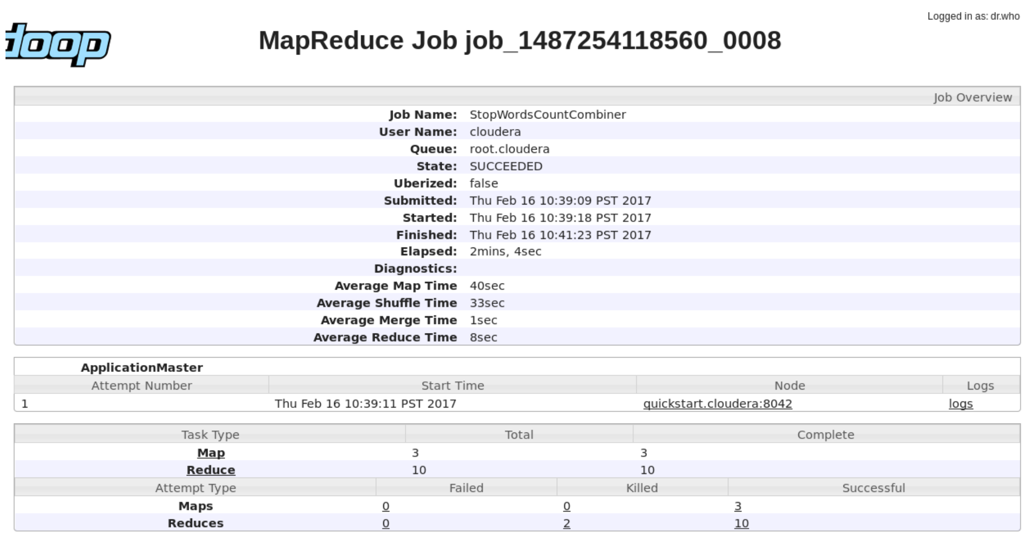


Figure 7: Word Count with 10 Reducers and combiner

The time is 2min 4sec, this is faster than without a combiner because there are way less <key,list<value» to process after the mapper phase. That means that the network is less overload because the shuffle and sort phase is faster since we have less data to process.

3.4 Compressing Intermediate result

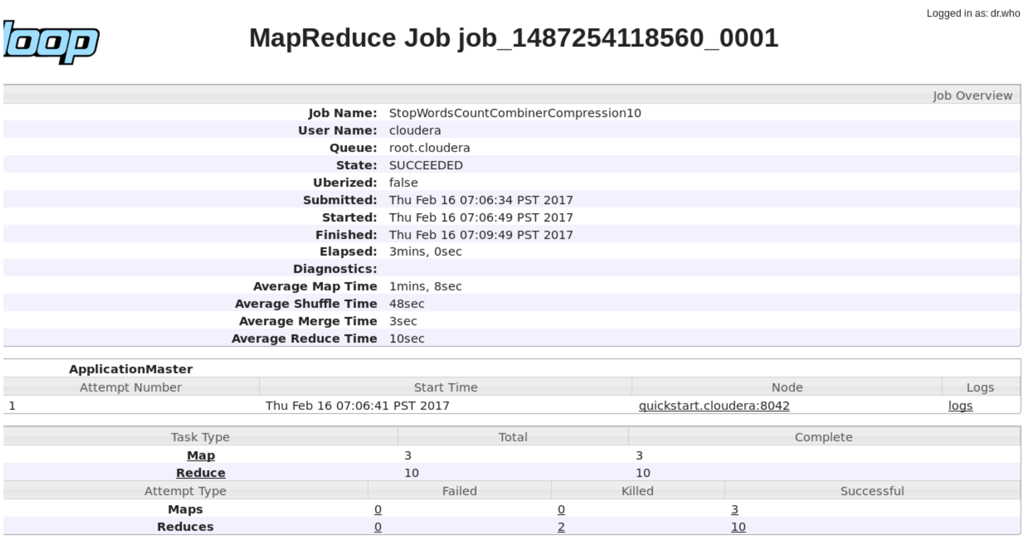
We used the BZip2Codec for this part http://hadoop.apache.org/docs/current/api/org/apache/hadoop/io/compress/BZip2Codec.html.

We also modified the configuration to take in account this codec :

Listing 7: Add Codec config to the job

```
Job job = new Job(getConf(),"StopWordsCountCombinerCompression10");
   job.setNumReduceTasks(10);
   job.getConfiguration().setBoolean("mapred.compress.map.output", true);
   job.getConfiguration().setClass("mapred.map.output.compression.codec",
BZip2Codec.class, CompressionCodec.class);
```

3.4.1 10 Reducers



Logged in as: dr.who

MapReduce Job job_1487254118560_0001

	Job Overview
Job Name:	StopWordsCountCombinerCompression10
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Thu Feb 16 07:06:34 PST 2017
Started:	Thu Feb 16 07:06:49 PST 2017
Finished:	Thu Feb 16 07:09:49 PST 2017
Elapsed:	3mins, 0sec
Diagnostics:	
Average Map Time	1mins, 8sec
Average Shuffle Time	48sec
Average Merge Time	3sec
Average Reduce Time	10sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Thu Feb 16 07:06:41 PST 2017	quickstart.cloudera:8042	logs

Task Type	Total	Complete
Map	3	3
Reduce	10	10

Attempt Type	Failed	Killed	Successful
Maps	0	0	3
Reduces	0	2	10

Figure 8: Word Count with 10 Reducers and combiner using compression

By using a compressor we obtains 3 mins, which is worser than without compression. This is because the time of compression and decompression doesn't compensate the time gained by the processing time of smaller output of the B2Zip codec. We can think that it would have been more efficient if the virtual machine had a better allocated processor and memory to compress and uncompress the result faster.

3.4.2 50 Reducers

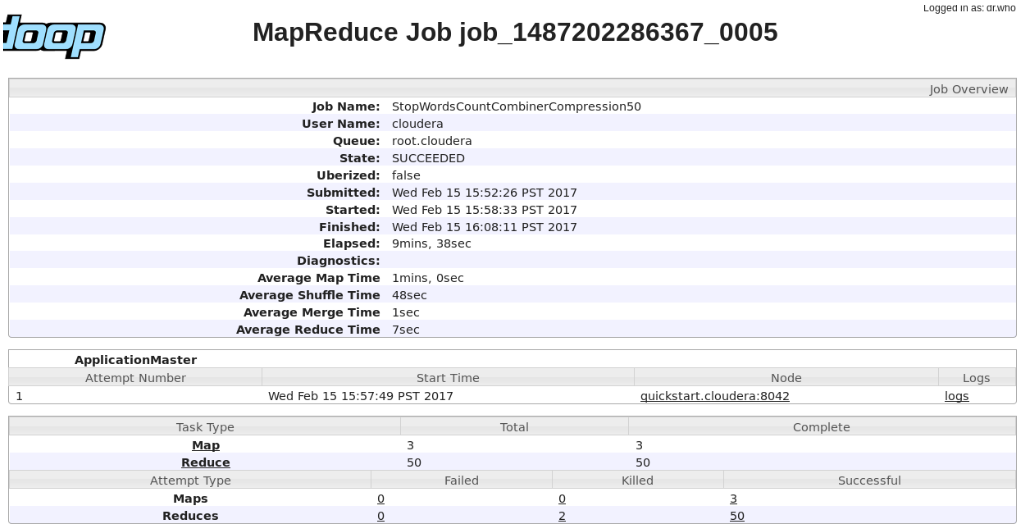


Figure 9: Word Count with 50 Reducers and combiner using compression

With 50 reducers, it get even worse because we have way more output to compress and uncompress. However we can notice that average map, shuffle, merge and reduce times are pretty same that the previous execution with 10 reducers. It means that the job compress and uncompress one by one the output like a queue, it it not doing this in parallel at the same time over several cluster. Hence, we got a worser result.

However compression is a useful tool in case we have a limited bandwidth that may occurs additional fees for instance.

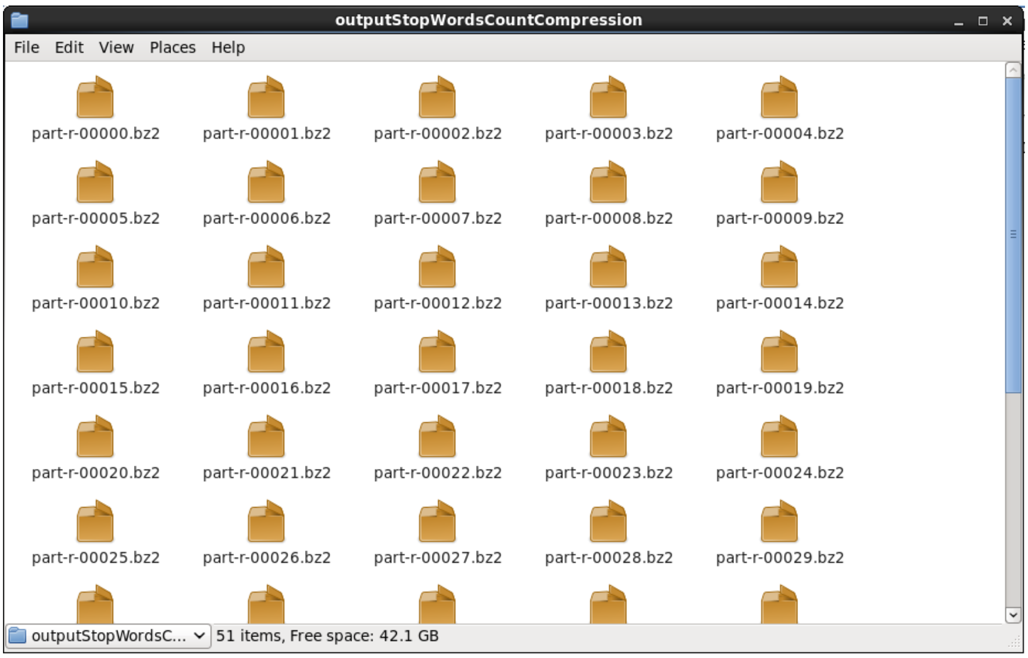
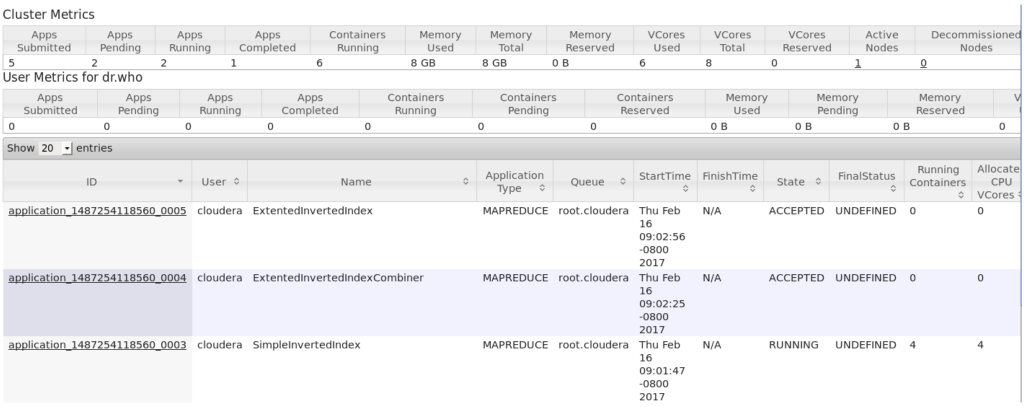


Figure 10: Compression Result with the BZip2 Codec

Use the table and tabular commands for basic tables — see Table **??**, for example.

4 Inverted Index



Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes
5	2	2	1	6	8 GB	8 GB	0 B	6	8	0	1	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	V
0	0	0	0	0	0	0	0 B	0 B	0 B	0

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocate CPU VCores
application_1487254118560_0005	cloudera	ExtentedInvertedIndex	MAPREDUCE	root.cloudera	Thu Feb 16 09:02:56 -0800 2017	N/A	ACCEPTED	UNDEFINED	0	0
application_1487254118560_0004	cloudera	ExtentedInvertedIndexCombiner	MAPREDUCE	root.cloudera	Thu Feb 16 09:02:25 -0800 2017	N/A	ACCEPTED	UNDEFINED	0	0
application_1487254118560_0003	cloudera	SimpleInvertedIndex	MAPREDUCE	root.cloudera	Thu Feb 16 09:01:47 -0800 2017	N/A	RUNNING	UNDEFINED	4	4

Figure 11: Inverted Index Jobs running

4.1 Handling CSV

From the previous part, we have generated a CSV with all stopwords that we are going to use.

We created a static method that return an ArrayList of string of those words.

Listing 8: ReadCSV class

```java
public class ReadCSV {

public static ArrayList<String> getStopWords(){
   ArrayList<String> stopwords= new ArrayList<String>();

   String csvFile = "stopwords.csv";
     String line = "";
     String cvsSplitBy = ",";

     try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {

         while ((line = br.readLine()) != null) {

             // use comma as separator
             String[] words = line.split(cvsSplitBy);


             stopwords.add(words[0].toLowerCase());
         }

     } catch (IOException e) {
         e.printStackTrace();
     }
   //System.out.println(stopwords);
   return stopwords;
}
```

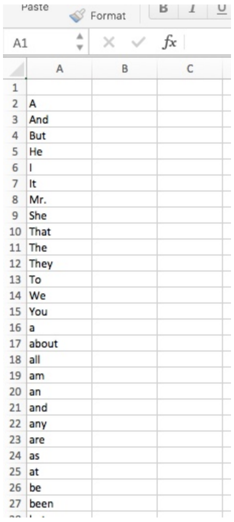


Figure 12: CSV File content

To skip word of this CSV file we just added a small condition

Listing 9: Skipping condition

```
ArrayList<String> allstopwords = ReadCSV.getStopWords();
if(!allstopwords.contains(token)){
...
}
```

4.2 Simple inverted Index and Counter

To implement this index, the approach is to have for every word the name of the original file it cames from and emit it with [word,docname] in the mapper

In the reducer we just check distinct docnames and we write it as the output at figure 14.

```
public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        Path filePath = ((FileSplit) context.getInputSplit()).getPath();
        String filename = ((FileSplit)
            context.getInputSplit()).getPath().getName().toString();

         for (String token: value.toString().split("\\s+")) {
          //System.out.println(token);
          token = token.replaceAll("[^a-zA-Z ]", "").toLowerCase();
         if(!allstopwords.contains(token)){
            word.set(token);
                context.write(word, new Text(filename));
         }
         }
      }
```

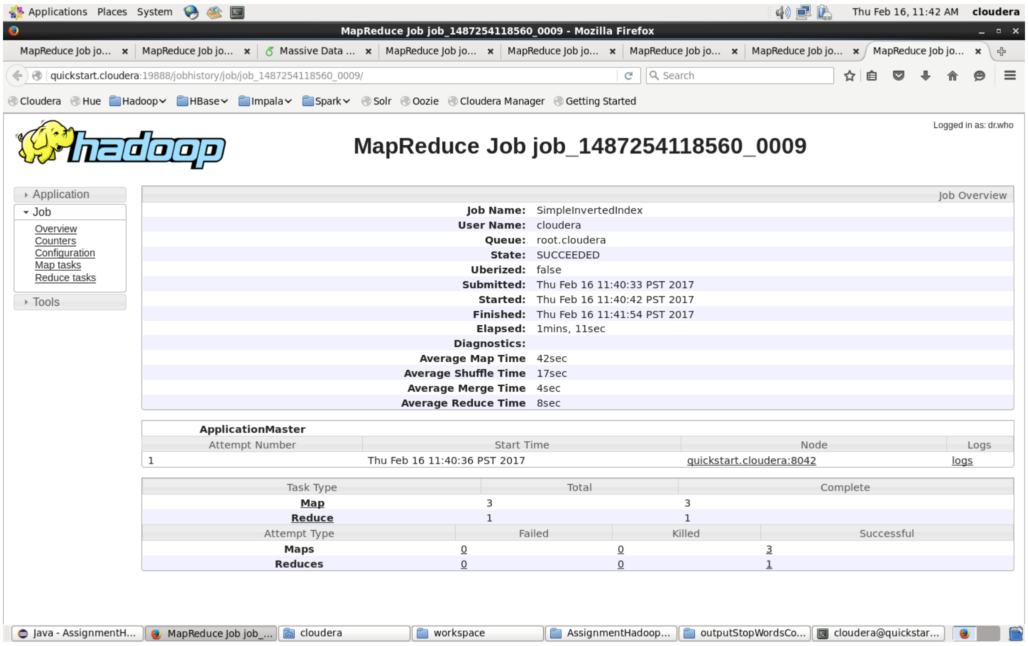


Figure 13: Simple Inverted Index Job

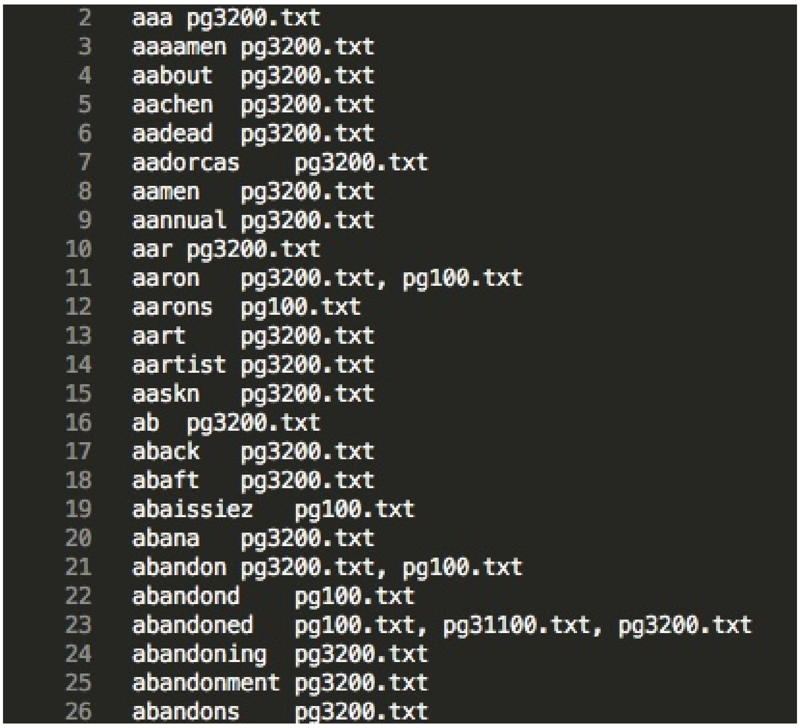


Figure 14: Simple Inverted Index Output

To know the count of unique words, we just count the number of line, and to know the number of word appearing in a single document we just check if the listsize of docnames is equal to one. We have used setup and cleanup function to do this

Listing 10: Counting unique words using setup and cleanup function

```
public static class Reduce extends Reducer<Text, Text, Text, Text> {

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

```java
protected void setup(Context ctxt) throws IOException, InterruptedException {

     counts.put("uniqueWord", 0);
     counts.put("wordSingleDocument", 0);

 }

@Override
public void reduce(Text key, Iterable<Text> values, Context context)
       throws IOException, InterruptedException {

 Set<String> docnames = new LinkedHashSet<String>();
 counts.put("uniqueWord", counts.get("uniqueWord") + 1);

   for (Text val : values) {
    docnames.add(val.toString());
   }
   if(docnames.size()==1){
    counts.put("wordSingleDocument", counts.get("wordSingleDocument") + 1);
 }


   //System.out.println(docnames.toString());
   context.write(key, new Text(docnames.toString().replace("]", "").replace("[",
       "")));



}
protected void cleanup(Context ctxt) throws IOException, InterruptedException {

   System.out.println("Unique words : " + counts.get("uniqueWord"));
   System.out.println("Words in single document " +
       counts.get("wordSingleDocument"));

   try{
     Path pt=new Path("outputSimpleInvertedIndexResult");
     FileSystem fs = FileSystem.get(new Configuration());
     BufferedWriter br=new BufferedWriter(new
         OutputStreamWriter(fs.create(pt,true)));
                              // TO append data to a file, use fs.append(Path f)
     String line;
     line="Unique words : " + counts.get("uniqueWord") + "\n";
     br.write(line);
     line="Words in single document " + counts.get("wordSingleDocument");
     br.write(line);
     br.close();
}catch(Exception e){
       System.out.println("File not found");
}
}
}
```

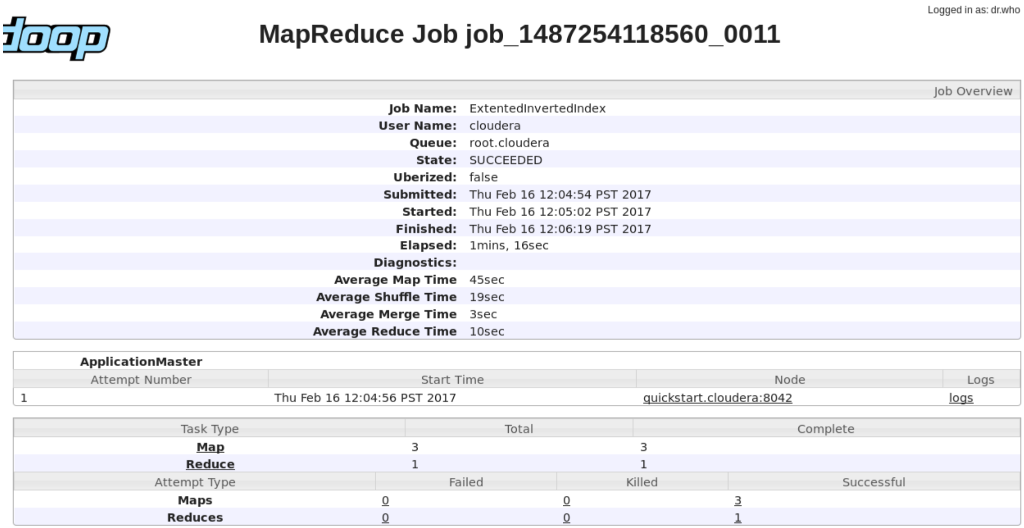
The setup function is call at the beginning of the reduce phase, it create an hashmap with uniqueword and wordSingleDocument key initialized with 0 values.

In the reducer we have to things to do, each time the reducer is called, it means we have a new key then a new unique word, so we increment uniqueword value. Moreover if docnames size is equal to one, it means that this word that is unique only appear in a single doc, we can then increment wordSingleDocument value.

To write those two counters in a separate file we use the cleanup function at the end who just created a file and write the hashmap key value in it

4.3 Extended inverted Index with combiner

A first version of this extended index has been implemented without a combiner at first. We can compare the two runtimes.


Logged in as: dr.who

MapReduce Job job_1487254118560_0011

Job Overview	
Job Name:	ExtentedInvertedIndex
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Thu Feb 16 12:04:54 PST 2017
Started:	Thu Feb 16 12:05:02 PST 2017
Finished:	Thu Feb 16 12:06:19 PST 2017
Elapsed:	1mins, 16sec
Diagnostics:	
Average Map Time	45sec
Average Shuffle Time	19sec
Average Merge Time	3sec
Average Reduce Time	10sec

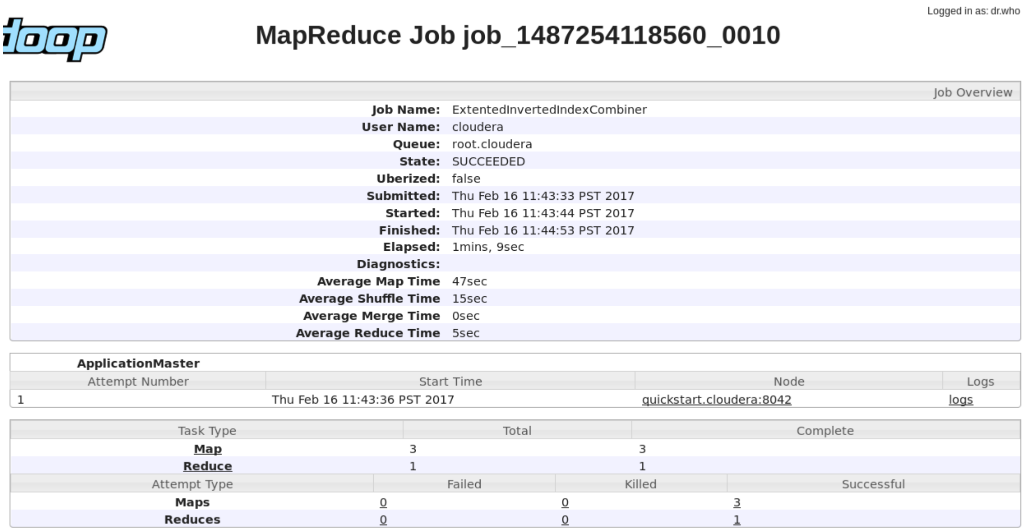
ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Thu Feb 16 12:04:56 PST 2017	quickstart.cloudera:8042	logs

Task Type	Total	Complete
Map	3	3
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	3
Reduces	0	0	1

Figure 15: Extended inverted with no combiner Index job


Logged in as: dr.who

MapReduce Job job_1487254118560_0010

Job Overview	
Job Name:	ExtentedInvertedIndexCombiner
User Name:	cloudera
Queue:	root.cloudera
State:	SUCCEEDED
Uberized:	false
Submitted:	Thu Feb 16 11:43:33 PST 2017
Started:	Thu Feb 16 11:43:44 PST 2017
Finished:	Thu Feb 16 11:44:53 PST 2017
Elapsed:	1mins, 9sec
Diagnostics:	
Average Map Time	47sec
Average Shuffle Time	15sec
Average Merge Time	0sec
Average Reduce Time	5sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Thu Feb 16 11:43:36 PST 2017	quickstart.cloudera:8042	logs

Task Type	Total	Complete
Map	3	3
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	3
Reduces	0	0	1

Figure 16: Extended inverted with combiner Index job

```
aaa pg3200.txt#1
aaaamen pg3200.txt#1
aabout  pg3200.txt#1
aachen  pg3200.txt#1
aadead  pg3200.txt#1
aadorcas    pg3200.txt#1
aamen   pg3200.txt#2
aannual pg3200.txt#1
aar pg3200.txt#2
aaron   pg3200.txt#3, pg100.txt#96
aarons  pg100.txt#1
aart    pg3200.txt#1
aartist pg3200.txt#1
aaskn   pg3200.txt#1
ab  pg3200.txt#5
aback   pg3200.txt#1
abaft   pg3200.txt#7
abaissiez   pg100.txt#1
abana   pg3200.txt#6
abandon pg3200.txt#15, pg100.txt#4
abandond    pg100.txt#6
abandoned   pg100.txt#2, pg31100.txt#5, pg3200.txt#49
abandoning  pg3200.txt#1
abandonment pg3200.txt#3
abandons    pg3200.txt#1
abarking    pg3200.txt#1
abase   pg100.txt#2
abasement   pg3200.txt#1
abash   pg3200.txt#1
```

Figure 17: Extended inverted with combiner Index output

As expected, the one with the combiner is faster because there are less data to process.

5 Relative Frequencies

The goal of this section is to get the top 100 word pairs sorted by decreasing order of relative frequency.

We implemented the two approach with combiners. As usual we kept only words with no special characters and digits and transform them into lowercases.

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocate CPU VCores
application_1487254118560_0007	cloudera	RelativeFrequencyStripes	MAPREDUCE	root.cloudera	Thu Feb 16 09:57:04 -0800 2017	N/A	ACCEPTED	UNDEFINED	1	1
application_1487254118560_0006	cloudera	RelativeFrequencyPair	MAPREDUCE	root.cloudera	Thu Feb 16 09:55:05 -0800 2017	N/A	RUNNING	UNDEFINED	5	5

Figure 18: Strip and Pair jobs

That means that each pair has a frequency that need to be compared with other pairs. So we created a pair class that implements Comparable interface

Listing 11: Pair class

```java
public class Pair implements Comparable<Pair> {
    double relativeFrequency;
    private String word;
    private String neighbor;


    @Override
    public int compareTo(Pair pair) {
```

```
        if (this.relativeFrequency > pair.relativeFrequency) {
            return 1;
        } else {
            return -1;
        }
    }
```

Then, in both approach we used a TreeSet, it allows us to sort them by order of frequency and then update at everystep the state of the 100 top frequency pair.

Finally a cleanup method that is part of the reducer to solve this problematic, it is called only at the end and write on a file the context of the TreeSet object :

Listing 12: Same code for both approach

```
topWordsPair.add(new Pair(entry.getValue() / total, keyStr, entry.getKey()));

                if (topWordsPair.size() > 100) {
                    topWordsPair.pollFirst();
                }
            }
        }

        protected void cleanup(Context ctxt) throws IOException, InterruptedException {
            while (!topWordsPair.isEmpty()) {
                Pair pair = topWordsPair.pollLast();
                ctxt.write(new Text(pair.getWord()), new Text(pair.getNeighbor()));
            }
        }
```

Finally, in both parts we have skipped special characters and numerics that are not considered as a word thanks to regex patterns :

Listing 13: Filtering uncorrect words

```
if (word.matches("^\\w+$") && !word.matches("-?\\d+(\\.\\d+)?"))
```

5.1 Stripes approach

class MAPPER
  method MAP(docid a, doc d)
    for all term $w \in$ doc d **do**
      $H \leftarrow$ new ASSOCIATIVEARRAY
      for all term $u \in$ NEIGHBORS(w) **do**
        $H\{u\} \leftarrow H\{u\} + 1$ ▷ Tally words co-occurring with w
      EMIT(Term w, Stripe H)

class REDUCER
  method REDUCE(term w, stripes $[H_1, H_2, H_3, \ldots]$)
    $H_f \leftarrow$ new ASSOCIATIVEARRAY
    for all stripe $H \in$ stripes $[H_1, H_2, H_3, \ldots]$ **do**
      SUM(H_f, H) ▷ Element-wise sum
    EMIT(term w, stripe H_f)

Figure 3.9: Pseudo-code for the "stripes" approach for computing word co-occurrence matrices from large corpora.

Figure 19: Stripes Algorithm

Listing 14: Implementation of the Stripes

```java
public void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {
         String[] words = value.toString().split(" ");

         for (String word : words) {
          word = word.toLowerCase();
            if (word.matches("^\\w+$") && !word.matches("-?\\d+(\\.\\d+)?")) {
               HashMap<String, Integer> stripe = new HashMap<>();

                for (String term : words) {
                  term = term.toLowerCase();
                    if (term.matches("^\\w+$") && !term.equals(word)) {
                        Integer count = stripe.get(term);
                        if(count==null){
                           stripe.put(term,1);
                        }else{
                           stripe.put(term,count+1);
                        }
                    }
                }

                StringBuilder stripeStr = new StringBuilder();
                for (Entry entry : stripe.entrySet()) {
                    stripeStr.append(entry.getKey()).append(":").append(entry.getValue()).append(",");
                }

                if (!stripe.isEmpty()) {
                    context.write(new Text(word), new Text(stripeStr.toString()));
                    //System.out.println("emiting in mapper :" + word + ",totalcount" +
                         " value : "+ stripeStr.toString());
                }
            }
         }
    }


    public void reduce(Text key, Iterable<Text> values, Context context)
            throws IOException, InterruptedException {
        java.util.Map<String, Integer> stripe = new HashMap<>();
        double total = 0;
        String keyStr = key.toString();

        for (Text value : values) {
            String[] stripes = value.toString().split(",");

            for (String termCountStr : stripes) {
                String[] termCount = termCountStr.split(":");
                String term = termCount[0];
                int count = Integer.parseInt(termCount[1]);

                Integer countSum = stripe.get(term);

                if(countSum == null){
                  stripe.put(term,count);
                }else{
                  stripe.put(term,countSum+count);
                }
                total += count;
            }
        }

        for (Entry<String, Integer> entry : stripe.entrySet()) {
```

```
            topWordsPair.add(new Pair(entry.getValue() / total, keyStr,
                entry.getKey()));

            if (topWordsPair.size() > 100) {
                topWordsPair.pollFirst();
            }
        }
    }
```

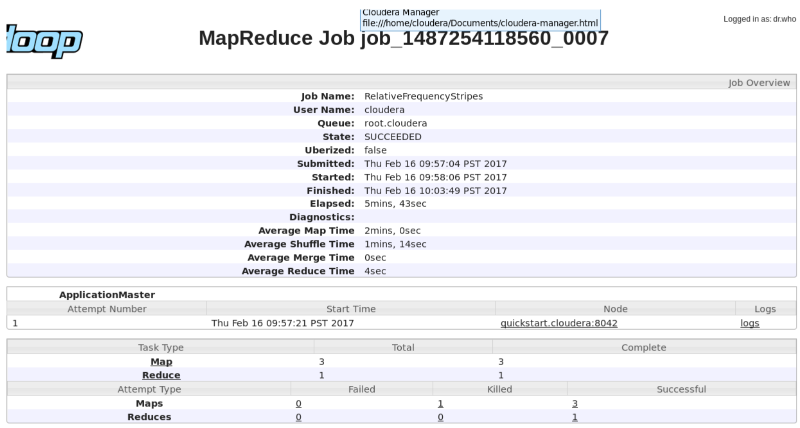
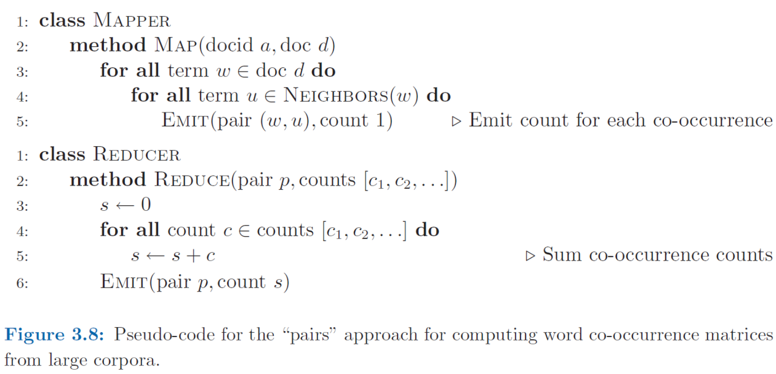


Figure 20: Stripes job

5.2 Pairs approach

We taked into account double pair that have twice the same word.



class MAPPER
  method MAP(docid a, doc d)
    for all term $w \in$ doc d **do**
      for all term $u \in$ NEIGHBORS(w) **do**
        EMIT(pair (w, u), count 1) ▷ Emit count for each co-occurrence

class REDUCER
  method REDUCE(pair p, counts $[c_1, c_2, \ldots]$)
    $s \leftarrow 0$
    for all count $c \in$ counts $[c_1, c_2, \ldots]$ **do**
      $s \leftarrow s + c$ ▷ Sum co-occurrence counts
    EMIT(pair p, count s)

Figure 3.8: Pseudo-code for the "pairs" approach for computing word co-occurrence matrices from large corpora.

Figure 21: Pair Algorithm

Listing 15: Implementation of the Pair

```
public void map(LongWritable key, Text value, Context context) throws IOException,
    InterruptedException {
        String[] words = value.toString().split(" ");

        for (String word : words) {
         word = word.toLowerCase();
            if (word.matches("^\\w+$") && !word.matches("-?\\d+(\\.\\d+)?")) { // check
                if it's a word and not a number
                int count = 0;
                for (String term : words) {
                  term = term.toLowerCase();
                    if (term.matches("^\\w+$") && !term.equals(word)) {
                        context.write(new Text(word + "," + term), new Text("1"));
```

```
                    //System.out.println("emiting in mapper : "+ word + "," + term +
                        " value : 1");
                    count++;
                }
            }
            //System.out.println(word);
            context.write(new Text(word + ",totalcount"), new
                Text(String.valueOf(count)));
            //System.out.println("emiting in mapper :" + word + ",totalcount" + "
                value : "+ String.valueOf(count));
        }
    }
}


public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
    String keyStr = key.toString();
    int count = 0;

    for (Text value : values) {
        count += Integer.parseInt(value.toString());
    }

    if (keyStr.endsWith(",totalcount")) {
        total = count;
    } else {
        String[] pair = keyStr.split(",");
        topWordsPair.add(new Pair(count / total, pair[0], pair[1]));

        if (topWordsPair.size() > 100) {
            topWordsPair.pollFirst();

            //we delete the first pair with the lowest frequency
        }
    }
}
```

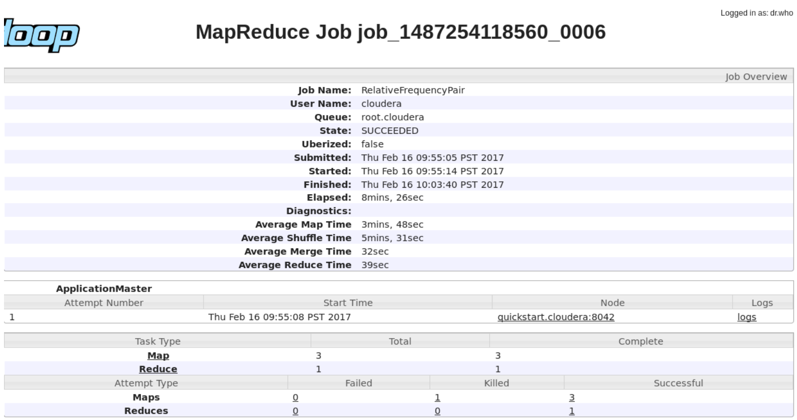


Figure 22: Pair job

5.3 Comparison

The strip approach seems to be much better if we compare the processing time (5 mins 43 sec for Stripes and 8 mins 26sec for Pair).

It is because the Pair approach send all the Pair combination with one value in the mapper. There are many more <Key,List<Value> combinaisons to process unlike to Stripes approach where we sum the value in the mapper before sending it to the combiner.

That means that the stripes approach doesn't overload the network because the suffle and short phase is way more efficient.

References

[1] Tushar Sarde. Hadoop interview questions and answers - what is combiner in mapreduce framework ? *http://toodey.com/*, 2005.

[2] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce.* 2010.

[3] Tiwaryc. Hadoop mapreduce design patterns pairs and stripes. *https://chandramanitiwary.wordpress.com/*, 2102.

[4] Bill Bejeck. Calculating a co-occurrence matrix with hadoop. *http://codingjunkie.net/cooccurrence/*, 2012.