

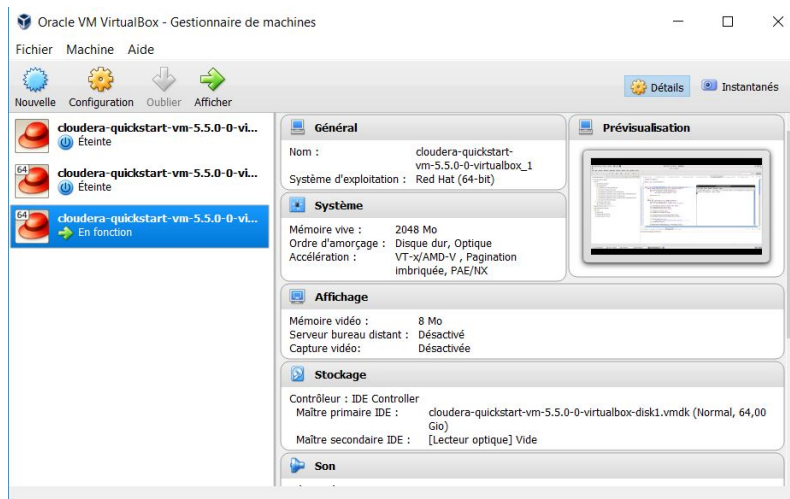
Assignement 1

Hadoop MapReduce

BDPA

Zayd EL KRYECH

The characteristics of virtual machine used for this assignment can be resumed in the following :



1 Question a

We begin by creating a new package Inverted_Index and input and output directories in the Inverted_Index workspace. We can then download the text files and put them in the input directory of our Inverted_Index. we put those files in the HDFS : `hadoop fs -mkdir input` `hadoop fs -put workspace/Inverted_Index/input/pg100.txt input` `hadoop fs -put workspace/Inverted_Index/input/pg3200.txt input` `hadoop fs -put workspace/Inverted_Index/output/pg31100.txt input`

```
[cloudera@quickstart ~]$ hadoop fs -ls input
Found 3 items
-rw-r--r-- 1 cloudera cloudera 5589886 2017-02-16 17:55 input/pg100.txt
-rw-r--r-- 1 cloudera cloudera 5589886 2017-02-16 17:55 input/pg31100.txt
-rw-r--r-- 1 cloudera cloudera 5589886 2017-02-16 17:55 input/pg3200.txt
[cloudera@quickstart ~]$
```

We can now write our mapreduce program. Finding the stopwords in our corpus is the same as doing a wordcount, the difference is adding a condition on the number of times words appear in the reduce task, in order to "select" the stopwords.

I have modified the code found on <http://snap.stanford.edu/class/cs246-data-2014/WordCount.java> taken from the CS246: Mining Massive Datasets Hadoop tutorial.

In order to remove duplicates the method `toLowerCase()` was added:

```
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.toLowerCase());
            context.write(word, ONE);
        }
    }
}
```

An if statement was added to the reduce function in order to put a condition on the occurrences of words.

```
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();
        }
    }
}
```

```

        if (sum > 4000) {
            context.write(key, new IntWritable(sum));
        }
    }
}

```

When the code is done we can export the jar file and execute it using the command in the terminal:

```
hadoop jar Inverted_Index.jar Assignment1.stopwords_10reducers_no_combiner
input output
```

i - Use 10 reducers and do not use a combiner. Report the execution time. In order to define the number of hadoop reducers we use the command :

```
job.setNumReduceTasks(10)
```

We can check now the resulting output stopwords_10reducers_no_combiner.csv

```

about ,7350
be ,27239
before ,4219
by ,19509
her ,24277
mr. ,4742
much ,4712
old ,4092
up ,8608
...

```

We can check the log execution time in the hadoop YARN ResourceManager:



Cluster	
About	
Nodes	
Applications	
NEW	
NEW_SAVING	
SUBMITTED	
ACCEPTED	
RUNNING	
FINISHED	
FAILED	
KILLED	
Scheduler	


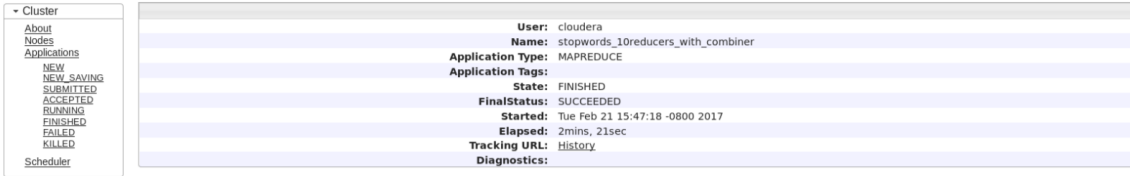
User:	cloudera
Name:	stopwords_10reducers_no_combiner
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Tue Feb 21 15:43:59 -0800 2017
Elapsed:	3mins, 9sec
Tracking URL:	History
Diagnostics:	

Execution time: 3min, 9sec.

ii - 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 code doesn't change much from the previous one we only add the following command:

```
job.setCombinerClass(Reduce.class);
```

We can check the log execution time in the hadoop YARN ResourceManager again:

The screenshot shows the Hadoop YARN ResourceManager web interface. On the left is a navigation menu with options: Cluster, About, Nodes, Applications, NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED, and Scheduler. The main panel displays details for a specific application:

User:	cloudera
Name:	stopwords_10reducers_with_combiner
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Tue Feb 21 15:47:18 -0800 2017
Elapsed:	2mins, 21sec
Tracking URL:	History
Diagnostics:	

Execution time: 2min, 21sec.

iii - 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? The same code as previously can be executed, but this time we add the following code lines:

```
FileOutputFormat.setCompressOutput(job, true);  
FileOutputFormat.setOutputCompressorClass(job, org  
.apache.hadoop.io.compress.SnappyCodec.class);
```

Execution time: 2min, 10sec.

iv - 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 same code as previously can be used. We add the following command to the code:

```
job.setNumReduceTasks(50)
```

Execution time: 8min, 53sec.

The following table summarizes the execution time:

Hadoop Job	Execution Time
stopwords_10reducers_no_combine °	3min, 9sec
stopwords_10reducers_with_combiner	2min, 21sec
stopwords_10reducers_no_combiner_with_comression	2min, 10sec
stopwords_50reducers_no_combiner_with_compression	8min, 53sec

Question b)

The MapReduce needs to be restarted from scratch for this part. For each word found in the documents, the program needs to output a (key, value) pair of the form (word, collection of filenames). As a consequence, we can already set in the Hadoop driver the output format of both keys and values as Text:

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

The map function needs remove the stopwords and get the filename for each word where it exists. We run the code stopwords.java, after we get the outcome in the format stopwords.csv which we then transform to the format stopwords.txt

```
public static class Map extends Mapper<LongWritable, Text, Text, Text> {
    private Text word = new Text();
    private Text filename = new Text();

    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        HashSet<String> stopwords = new HashSet<String>();
        BufferedReader Reader = new BufferedReader(
            new FileReader(
                new File(
                    "/home/cloudera/workspace/InvertedIndex/output/StopWo
```

```

String pattern;
while ((pattern = Reader.readLine()) != null) {
    stopwords.add(pattern.toLowerCase());
}

String filenameStr = ((FileSplit) context.getInputSplit())
    .getPath().getName();
filename = new Text(filenameStr);

for (String token : value.toString().split("\\s+")) {
    if (!stopwords.contains(token.toLowerCase())) {
        word.set(token.toLowerCase());
    }
}

context.write(word, filename);
}
}

```

It gives a (key, value) pair output of the form (word, filename):

```

word1, doc1.txt
word1, doc1.txt
word1, doc1.txt
word1, doc2.txt
word2, doc1.txt
word2, doc2.txt
word2, doc2.txt
word2, doc3.txt
word3, doc1.txt
...

```

Our reducer stores all the filenames for each word in a `HashSet` (a collection that cannot accept duplicates).

```

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

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

        HashSet<String> set = new HashSet<String>();

        for (Text value : values) {
            set.add(value.toString());
        }

        StringBuilder builder = new StringBuilder();

        String prefix = "";
        for (String value : set) {
            builder.append(prefix);
            prefix = ", ";
            builder.append(value);
        }

        context.write(key, new Text(builder.toString()));
    }
}

```

It yields a (key, value) pair output of the form (word, collection of filenames):

```

word1 -> doc1.txt , doc2.txt
word2 -> doc1.txt , doc2.txt , doc3.txt
word3 -> doc1.txt
...

```

Question c)

A custom driver is added to our driver

```

public static enum CUSTOM_COUNTER {

```

```

        UNIQUE_WORDS,
    };

```

In this case, the map function is the same as before. But the reduce function is now different. an if statement is added in order to conditionally select only the unique words, that is to say the words for which the collection of filenames is of length 1. We also add within the reduce function, our UNIQUE_WORDS counter:

```

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

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

        HashSet<String> set = new HashSet<String>();

        for (Text value : values) {
            set.add(value.toString());
        }

        if (set.size() == 1) {

            context.getCounter(CUSTOM_COUNTER.UNIQUE_WORDS).increment(1);

            StringBuilder builder = new StringBuilder();

            String prefix = "";
            for (String value : set) {
                builder.append(prefix);
                prefix = ", ";
                builder.append(value);
            }

            context.write(key, new Text(builder.toString()));

        }
    }
}

```


}

Then, after running the hadoop job, we can see our counter value in the output, representing the number of unique words:

```
centralemdp.invertedindex.InvertedIndex_unique$CUSTOM_COUNTER  
    UNIQUE_WORDS=68476
```

Question d)

Extend the inverted index of (b), in order to keep the frequency of each word for each document. The new output should be of the form:

this	filename and frequency
this	doc1.txt#1, doc2.txt#1c
is	doc1.txt#2, doc2.txt#1, doc3.txt#1
a	doc1.txt#1
program	doc1.txt#1, doc2.txt#1

This means that the word frequency must follow a single '#' character, which should follow the filename, for each file that contains this word. You are required to use a Combiner.

The previous code could be expanded of question b) that we explained before. In this case, the map function is the same as the two maps functions used before. The intuition here is that the reducer should be able store the filenames in a collection but this time without removing the duplicates filenames, in order to get something like this:

```
word1 -> doc1.txt , doc1.txt , doc1.txt , doc2.txt  
word2 -> doc1.txt , doc2.txt , doc2.txt , doc3.txt  
word3 -> doc1.txt  
...
```

We want to keep the duplicates because then we are able with the reducer to count the occurrences of each filename in the value part of the mapper output, with the `Collections.frequency(array, object)` method in Java.

By implementing this new method, our new reduce function becomes

```

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

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

        ArrayList<String> list = new ArrayList<String>();

        for (Text value : values) {
            list.add(value.toString());
        }

        HashSet<String> set = new HashSet<String>(list);
        StringBuilder builder = new StringBuilder();

        String prefix = "";
        for (String value : set) {
            builder.append(prefix);
            prefix = ", ";
            builder.append(value + "#" + Collections.frequency(list, value));
        }

        context.write(key, new Text(builder.toString()));
    }
}

```

It gives a (key, value) pair output of the form (word, collection of filenames with frequency):

```

word1 -> doc1.txt#3, doc2.txt#1
word2 -> doc1.txt#1, doc2.txt#2, doc3.txt#1
word3 -> doc1.txt#1
...

```