Assignment 2 – Big Data Processing – Olivier Salaün – 17/03/17.

Settings.

```
[cloudera@quickstart ~]$ hadoop version
Hadoop 2.6.0-cdh5.8.0 Subversion http://github.com/cloudera/hadoop -r
57e7b8556919574d517e874abfb7ebe31a366c2b
Compiled by jenkins on 2016-06-16T19:38Z
Compiled with protoc 2.5.0
From source with checksum 9e99ecd28376acfd5f78c325dd939fed
This command was run using /usr/lib/hadoop/hadoop-common-2.6.0-cdh5.8.0.jar
```

```
[cloudera@quickstart ~]$ cat /proc/cpuinfo
processor	: 0
vendor_id	: GenuineIntel
cpu family	: 6
model		: 58
model name	: Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
stepping	: 9
microcode	: 25
cpu MHz		: 2494.316
cache size	: 3072 KB
physical id	: 0
siblings	: 1
core id		: 0
cpu cores	: 1
apicid		: 0
initial apicid	: 0
fpu		: yes
fpu_exception	: yes
cpuid level	: 13
wp		: yes
flags		: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 syscall nx rdtscp lm constant_tsc up rep_good xtopology
nonstop_tsc unfair_spinlock pni pclmulqdq monitor ssse3 cx16 sse4_1 sse4_2 x2apic
popcnt aes xsave avx rdrand hypervisor lahf_lm
bogomips	: 4988.63
clflush size	: 64
cache_alignment	: 64
address sizes : 36 bits physical, 48 bits virtual
power management:
```

Creation of StopWords and WordCount files.

Before the pre-processing step, we use a stopword file from a previous assignment… :

```
about
be
before
by
```

```
 her
 mr.
 much
 old
 up
 where
 you
```

... and we generate a wordcount file based in the input file that will be used for the frequencies:

```
Armagnac;3
Arme;2
Armed;3
Armenia;2
Armies;1
Armigero;2
Arming;1
Armourer;3
Arms;2
Army;6
Aroint;1
Aroused;1
Arragon;4
Arraign;1
Arraigning;1
Array;1
```

Pre-processing the input.

We set a counter for the total number of lines (documents) of the output file and the result is stored in a text file.

```
long counter =
job.getCounters().findCounter(NUMBER_OF_LINES_COUNTER.TOTAL_NU
MBER_OF_LINES).getValue();
Path outFile = new Path("TOTAL_NUMBER_OF_LINES.txt");
BufferedWriter BufWri = new BufferedWriter(new
OutputStreamWriter(fs.create(outFile, true)));
BufWri.write(String.valueOf(counter));
BufWri.close();
```

In the mapper class, we store in a hashset all the stopwords contained in a text file that was generated in a previous assignment.

```
File stopwords_csv = new
File("/home/cloudera/workspace/preprocess/StopWords.csv");
HashSet<String> stopwords_HS = new HashSet<String>();

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

```
BufferedReader bufread = new BufferedReader(new
FileReader(stopwords_csv));
String string ;
// transfer stopwords from csv file to HashSet
while ((string = bufread.readLine()) != null){
  stopwords_HS.add(string.trim().toLowerCase());
  // need to trim string in order to remove whitespaces
      }
bufread.close();
}
```

For the mapper output, we skip the empty lines of the original text file, remove special characters and split by whitespaces, remove stopwords with the stopwords hashset and us the ArrayList in order not to have duplicates in the values of the mapper output:

```
ArrayList <String> words_AL = new ArrayList <String>();
if (value.toString().length() != 0){
// filter empty lines
  for   (String   token:   value.toString().replaceAll("\\W","
").split("\\s+")){
  // remove all special characters and split by whitespaces
    if (!stopwords_HS.contains(token.toLowerCase())){
    // keep if token is not a stopword
      if (!words_AL.contains(token.toLowerCase())){
      // keep if word not already included in ArrayList
words_AL.add(token.replaceAll("\\W","").toString().toLowerCase
());
word.set(token.toString());
// word value keeps uppercase letters unlike words_AL
context.write(key, word);
}
}
}
}
}
}
```

The mapper output returns a result like the one below:

```
0    EBook
0    Works
0    Project
0    Complete
0    William
0    Gutenberg
0    Shakespeare
78   William
78   Shakespeare
101  use
101  eBook
101  anyone
```

In the reducer, we transfer the content of the wordcount csv file to a hashmap containing both a string and an integer.

```
File wordcount_csv = new
File("/home/cloudera/workspace/preprocess/WordCount.csv");

HashMap<String,  Integer>  wordcount_HM  =  new  HashMap<String,
Integer>();

  protected  void  setup(Context  context)  throws  IOException,
InterruptedException {
    BufferedReader    bufread    =    new    BufferedReader(new
FileReader(wordcount_csv));
    String string;

    while ((string = bufread.readLine()) != null){
      String[] line_content = string.split(";");
      wordcount_HM.put(line_content[0],
Integer.parseInt(line_content[1]));
      // transfer the content of the wordcount csv file to the
hashmap wordcount_HM
     }
     bufread.close();

 }
```

Each word contained in the values in stored in an ArrayList for string and their respective frequencies (retrieved from the wordcount hashmap) are stored in a separated ArrayList for integers. Next, we sort the words of the ArrayList in ascending order with the ArrayList containing the integers:

```
int i = 0;
while(i < word_string_AL.size() + 1){
// until we've spanned the whole ArrayList of strings
  for(int j = 0; j  < word_string_AL.size();  j++){
  // for each word of the ArrayList
    String string = word_string_AL.get(j);
    // pick the j-th woth
      if(word_count_AL.get(j)                              ==
Collections.min(word_count_AL)){
      // if that j-th word has the lowest frequency...
      sorted_word_AL.add(string);
      // ...add it to the new sorted ArrayList of words...
      word_count_AL.set(j,1000000);
      //  ...set  the  count  of  the  j-th  to  1  million  to  make
sure it won't be selected again...
      i++;
      // increment i by 1

     }
   }
```

```
 }
```

Finally, for each line-document-value of the reducer output, words are appended in increasing order.

```
  StringBuilder sorted_value = new StringBuilder();
  // store value of reduce output

  for (int k = 0; k < word_string_AL.size(); k++){
  // spans the whole length of the ArrayList
     sorted_value.append(sorted_word_AL.get(k).toString());
     // append the words from sorted array list in ascending
order for that document-line
     //sorted_value.append("#");

//sorted_value.append(wordcount_HM.get(sorted_word_AL.get(k).t
oString().toLowerCase()));
     sorted_value.append(" ");
   }
```

__Set-similarity joins.__

a)

In the mapper, we can pair keys together without duplicates by ensuring that the id/integer of the first key remains strictly lower than the key of the other document.

We first use the output file of previous step as input file and within the mapper class we store all of its keys in a HashSet id_HS that captures IDs of all documents (lines of the file):

```
HashSet<String> id_HS = new HashSet<String>();
bufread = new BufferedReader(new FileReader(new
File("/home/cloudera/workspace/comparison_A/input/part-r-
00000.csv")));

String pattern;
     while ((pattern = bufread.readLine()) != null) {
          String[] word = pattern.split(";");
          id_HS.add(word[0]);
          // store all keys of input file into the string
hashset
          }
```

Then, we create the key pairs without duplicate. We go over all candidate pairs and we skip those in which both keys are identical or in which the first key is larger (as an integer) than the second one:

```
for (String id1_string : id_HS) {
// for each element of the ID hashset...
```

```
     String id2_string = key.toString();
     // set id1_string as a string copy of keys

     int int1 = Integer.parseInt(id1_string);
     int int2 = Integer.parseInt(id2_string);


     if (int1 > int2 || id1_string.equals(id2_string)) {
          continue;
     // skip candidate pair when the first key is higher than
the second one
     // or when the two keys are identical
     }

     StringBuilder pair_key = new StringBuilder();

     pair_key.append(id1_string + ";" + id2_string);
     context.write(new Text(pair_key.toString()), new
Text(value.toString() ) );
     // create the keys pair

}
```

In the reducer class, we first create a function for computing the similarity score for each documents pair. The set of words of each line is stored in hashsets. These hashsets (doc1_HS and doc2_HS) are used to compute the intersection hashset and the union hashset. The sizes of these latters are used for computing the Jaccard similarity:

```
public double simil(HashSet<String> doc1_HS, HashSet<String>
doc2_HS){

     HashSet<String> intersection_HS = new HashSet<String>
(doc1_HS);
     // copy hashset doc1_HS into intersection_HS
     HashSet<String> union_HS = new HashSet<String> (doc1_HS);
     // copy hashset doc1_HS into union_HS

     intersection_HS.retainAll(doc2_HS);
     // keep elements of intersection_HS that also match the
existing ones in doc2_HS
     union_HS.addAll(doc2_HS);
     // add elements of doc2_HS to those of union_HS without
having duplicates

     int intersection_int = intersection_HS.size();
     // compute elements in common between doc1 and doc 2
     int union_int = union_HS.size();
     // compute number of distinct elements in total with both
doc1 and doc2

     double similarity = ((double) intersection_int) /
((double)union_int);
```

```
     return similarity;
     // compute Jaccard similarity
     }
```

After this function (we will come back to it later), we load the input file again and transfer its content into a HashMap copy_input_HM:

```
HashMap<String,  String>  copy_input_HM  =  new  HashMap<String,
String>();
bufread = new BufferedReader(new FileReader(new
File("/home/cloudera/workspace/comparison_A/input/part-r-
00000.csv")));
String pattern;
while ((pattern = bufread.readLine()) != null) {
     String[] word = pattern.split(";");
     copy_input_HM.put(word[0], word[1]);
     // copy input file to hashmap copy_input_HM
}
```

We load the key-pairs of the mapper output and we split them. We also create two HashSets that will be used as inputs for the function we created previously:

```
String[] double_key_string = key.toString().split(";");
String key1_string = double_key_string[0];
String key2_string = double_key_string[1];
// separate keys from each pair from mapper output

HashSet<String> hashset1 = new HashSet<String>();
HashSet<String> hashset2 = new HashSet<String>();
//  create  two  hashsets  as  input  for  comparison  in  simil
function
```

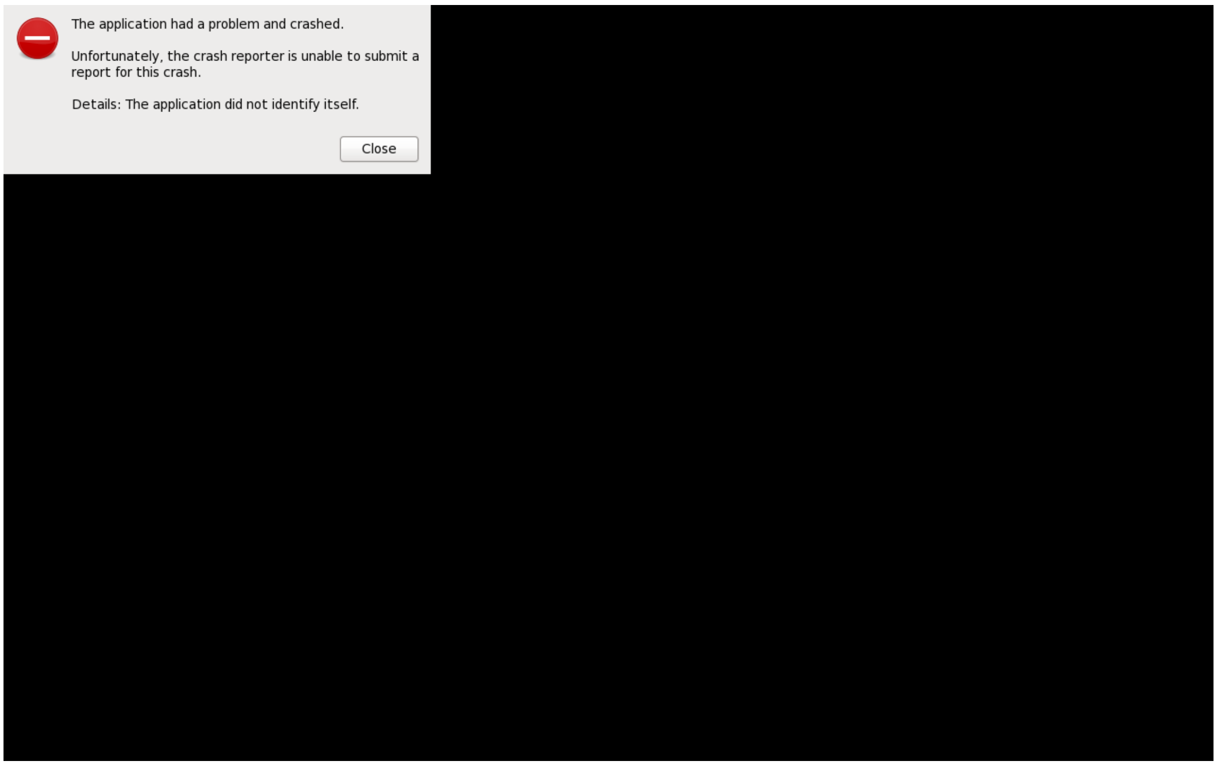
Next, for each key, we copy the content of each line-document from the HashMap copy_input_HM into two strings that will fulfill hashset1 and hashset2:

```
String string1 = copy_input_HM.get(key1_string);
String string2 = copy_input_HM.get(key2_string);
// for each set, copy the value-document-line of copy_input_HM
to a string for the corresponding key

for (String string : string1.split(" ")) {
     hashset1.add(string);
     // transfer the content of the string to a hashset
}

for (String string : string2.split(" ")) {
     hashset2.add(string);
     // transfer the content of the string to a hashset
}
```

We compute the similarity score for each pair of HashSets and we add it as a value to the reducer output if it exceeds 0.8:

```
double similarity = simil(hashset1, hashset2);
// compute similarity score between both input hashsets that
correspond to the content of two documents

context.getCounter(COMPARISONS_COUNTER.number_of_comparisons).
increment(1);
// increment the comparison counter by 1
if (similarity >= 0.8) {
     context.write(new Text( "(" + key.toString()+ ")" ), new
Text(String.valueOf(similarity)) );
     }
```

It was not possible to complete the execution of the programme because of a "Spill failed" error and because it was very time-consuming (below 20% mapping in 2 hours). During the mapping execution, the process was filling up a snapshot within the virtualbox whose size was slowly reaching 50 GB before make both the programme and the virtual machine crash. A compression of intermediate files might be required.


The application had a problem and crashed.

Unfortunately, the crash reporter is unable to submit a report for this crash.

Details: The application did not identify itself.

The total number of comparisons can be computed as the number of edges linking 114,699 nodes since the file contains 114,699 documents-lines. We can consider it as the sum of the number of sides of a 114,699-sided polygon and its diagonals: 114,699 + (114,699 * (114,699 – 3) / 2) = 6,577,872,951 unique pairs or comparisons. The number of comparisons that exceed the threshold is very likely lower than this.

b)

c)

We can expect the inverted index method to have a lower execution runtime than the naïve comparison method since it only focuses on pairs that already have at least a word in common and since it disregards those which have nothing in common and that would not reach the 0.8 threshold anyway.