Cloud Computing: Laborator 3

Honorius Gâlmeanu
galmeanu@unitbv.ro

November 7, 2022

1 Numărul maxim de apariţii

În laboratorul anterior am determinat cu ajutorul programului WordCount numărul maxim de apariţii pentru fiecare cuvânt folosit într-un text. În cele ce urmează ne propunem să determinăm care este cuvântul folosit cel mai des în text. Vom rezolva problema folosind mai întâi Streaming API iar apoi vom arăta cum se pot construi lanţuri de tip map-reduce.

Reamintim că anterior am rulat WordCount folosind un număr de 8 reducers:

Command Line

```
$ hadoop jar /home/user/apps/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.4.jar
  -file mapper.py -mapper mapper.py -file reducer.py -reducer reducer.py
  -input input/ -output output/ -numReduceTasks 8
```

Ca atare, folder-ul de ieşire pentru WordCount are o structură de genul:

Command Line

```
$ hdfs dfs -ls output
Found 9 items
-rw-r--r--   1 user user          0 2018-10-21 19:55 output/_SUCCESS
-rw-r--r--   1 user user      14908 2018-10-21 19:55 output/part-00000
-rw-r--r--   1 user user      16142 2018-10-21 19:55 output/part-00001
-rw-r--r--   1 user user      15089 2018-10-21 19:55 output/part-00002
-rw-r--r--   1 user user      15531 2018-10-21 19:55 output/part-00003
-rw-r--r--   1 user user      15422 2018-10-21 19:55 output/part-00004
-rw-r--r--   1 user user      15321 2018-10-21 19:55 output/part-00005
-rw-r--r--   1 user user      15005 2018-10-21 19:55 output/part-00006
-rw-r--r--   1 user user      15043 2018-10-21 19:55 output/part-00007
```

Fişierele conţin perechi de tip (cuvânt, frecvenţă), unde cuvintele sunt unice:

Command Line

```
$ hdfs dfs -cat output/part-00002
  . . .
executing 4
commute 1
leeves 3
battle 18
displaying 5
assembled 7
sund 1
row 2
apartment 68
baron 45
swam 2
  . . .
```

Folder-ul “output” va fi de fapt sursa de date a job-ului map-reduce pe care îl propunem în continuare.

Pentru a găsi cuvântul cu numărul maxim de apariţii, vom realiza mapper-ul şi reducer-ul. Structura de înlănţuire este cât se poate de simplă:

Info: MAP | REDUCE

Ieşirea mapper-ilor este distribuită reducer-ilor. Avem de-a face cu un simplu piping.

identity.py

```
#!/usr/bin/env python

import sys
import re

# read input from stdin, line by line
for line in sys.stdin:
  words = line.split()
  print ' '.join(words)
```

Mapper-ul “identity.py” nu face nici o prelucrare; intrarea este copiată la ieşire. Reducer-ul este mai interesant:

maximum.py

```
#!/usr/bin/env python

import sys

max = int(sys.argv[1])
maxWord = None

for line in sys.stdin:
  if len(line) == 0:
    continue

  # strip and split considering only the first gap
  word, count = line.strip().split(' ', 1)

  # convert the string to integer
  try:
    count = int(count)
  except ValueError:
    # if the value is not a number, discard line
    continue

  if count > max:
    max = count
    maxWord = word

print '%s %d' % (maxWord, max)
```

Deşi nu are nevoie, am adăugat un parametru pentru reducer, pentru a ilustra ulterior cum se lucrează cu aceştia. Mai departe, reducer-ul va citi liniile de intrare presupuse a fi perechi (cuvânt, frecvenţă), şi va găsi maximul frecvenţei.

Notice: Atenţie, un reducer nu prelucrează decât partea care îi revine; de aceea el nu va determina decât maximul pentru perechile care au acelaşi hash code. Altfel spus, dacă numărul de reduceri nu este 1, veţi obţine mai multe maxime.

Rularea end-to-end folosind Streaming API se face astfel:

Command Line

```
$ hadoop jar /home/user/apps/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.4.jar
  -file identity.py -mapper identity.py
  -file maximum.py -reducer 'maximum.py -1'
  -input output/ -output output2/
  -numReduceTasks 1

$ hdfs dfs -cat output2/part-00000
  the 13835

$
```

2 Mapper şi reducer ce partajează acelaşi cod - maximum

De fapt, mapper-ul "identity.py" este inutil; calculul se face mult mai rapid dacă şi în mapper şi în reducer folosim acelaşi cod:

Command Line

```
$ hadoop jar /home/user/apps/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.4.jar
  -file maximum.py -mapper 'maximum.py -1'
  -file maximum.py -reducer 'maximum.py -1'
  -input output/ -output output4/
  -numReduceTasks 1
```

Profităm de faptul că în fiecare din cele 8 blocuri din "input/", cheile sunt unice. Astfel, fiecare mapper poate determina, iterativ, maximul pe bloc. Mai departe, fiind un singur reducer, acesta procesează toate liniile singulare trimise de mapperi.

Se observă că maximul astfel determinat este identic cu cel găsit prin apelul în linie de comandă al exemplului WordCount în Python din laboratorul anterior şi sortarea rezultatelor:

Command Line

```
$ cat bigbook.txt | python mapper.py | python reducer.py | sort -g -k 2 | tail
he 3216
in 3457
that 3516
i 3851
you 3977
a 4822
and 5939
of 6519
to 6686
the 13835
```

3 Înlănţuirea mapper-ilor şi a reducer-ilor

Odată ce am calculat cu etapele map şi reduce din Wordcount, frecvenţele asociate cuvintelor, putem rula din nou un map şi reduce ce calculează maximele. Problema cu această abordare e că rezultatele intermediare sunt scrise în HDFS, lucru care consumă timp şi resurse.

Până acum, ştim să modelăm job-urile map-reduce ca o succesiune: (map, reduce, map, reduce ...). Putem reprezenta această înlănţuire[1] folosind pseudo-expresii regulate:

```
[ MAP | REDUCE ]+
```

Secvenţa map urmat de reduce se repetă de oricâte ori. Expresia analoagă pentru un ChainMapper şi un ChainReducer însă va fi:

```
MAP+ | REDUCE | MAP*
```

Jobul va rula mai mulţi mappers în secvenţă pentru a pre-procesa datele, iar după ce rulează reduce, poate opţional să ruleze mai mulţi mapperi pe partiţiile astfel create de reducer. Partea utilă este că rularea tuturor paşilor de pre- şi post-procesare nu creează fişiere intermediare şi reduce puternic accesele I/O. Pentru înlănţuire, se cheamă metoda "addMapper()" atât în ChainMapper cât şi în ChainReducer.

Considerăm 4 mapperi (Map1, Map2, Map3 şi Map4) şi un reducer (Reduce), înlănţuiţi într-un singur job în secvenţa:

```
Map1 | Map2 | Reduce | Map3 | Map4
```

Astfel, putem privi Map2 şi Reduce ca şi activitatea de bază a acestui job, cu partiţionarea şi shuffling-ul standard între mapper şi reducer. Map1 este considerat pas de pre-procesare iar Map3 şi Map4 paşi de post-procesare.

Secvenţa de task-uri se specifică cu ajutorul driver-ului job-ului. Este important să ne asigurăm că perechile (cheie, valoare) corespund ca şi tip între ieşirea unui task şi intrarea task-ului următor.

[1] engl. chaining

Maximum.java: driver part

```
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");

    ChainMapper.addMapper(job, TokenizerMapper.class,
              Object.class, Text.class,
              Text.class, IntWritable.class,
              new Configuration(false));

    ChainReducer.setReducer(job, IntSumReducer.class,
              Text.class, IntWritable.class,
              Text.class, IntWritable.class,
              new Configuration(false));

    ChainReducer.addMapper(job, MaximumMapper.class,
              Text.class, IntWritable.class,
              Text.class, IntWritable.class,
              new Configuration(false));

    job.setJarByClass(Maximum.class);

    job.setNumReduceTasks(10);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    if (!job.waitForCompletion(true))
        System.exit(1);

    System.out.println("Jobs completed.");
  }
```

Observaţi cum s-a făcut înlănţuirea. Fiecare task map sau reduce înlănţuit adaugă o nouă configuraţie independentă. Semnătura lui “addMapper()” este:

Hadoop 2.x API

```
static void addMapper(
  Job job,                          // jobul principal
  Class<? extends Mapper> klass, // clasa mapper
  Class<?> inputKeyClass, Class<?> inputValueClass,
  Class<?> outputKeyClass, Class<?> outputValueClass,
  Configuration mapperConf
)
```

Este de remarcat faptul că aceiaşi parametri pe care îi puneam în job sunt acum legaţi de mapper. Semnătura pentru “setReducer()” în cazul reducer-ului este similară.

Notice: Apelul succesiv al metodei “setReducer()” nu face decât să suprascrie ultimul reducer setat, spre deosebire de “addMapper()” care înlănţuie mapperi.

Maximum.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 {
    Pattern pattern = Pattern.compile("[\\w]+");
    Matcher matcher = pattern.matcher(value.toString());

    for(int i=1; matcher.find();i++) {
      word.set(matcher.group());
      context.write(word, one);
    }
  }
}
```

Tokenizer-ul este cel din WordCount, nici o deosebire.

Maximum.java

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

  public void reduce(Text key, Iterable<IntWritable> values, Context context
      ) throws IOException, InterruptedException
  {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

Identic este şi IntSumReducer.

Maximum.java

```
public static class MaximumMapper
    extends Mapper<Text, IntWritable, Text, IntWritable> {

 String word;
 static int maxval = 0;

 public void map(Text key, IntWritable value, Context context
               ) throws IOException, InterruptedException
 {
    if (value.get() > maxval) {
       maxval = value.get();
       word = key.toString();
    }
 }

 public void cleanup(Context context
         ) throws IOException, InterruptedException
 {
    Text text = new Text(word);
    IntWritable result = new IntWritable(maxval);
    context.write(text, result);
 }
}
```

Mapper-ul de post-procesare MaximumMapper conţine o funcţie numită “cleanup()”. Aceasta este este apelată exact înainte de terminarea mapper-ului şi are rolul de a scrie maximul.

Lansarea în execuţie a chain job-ului se face cu un apel de genul:

Command Line

```
$ hadoop jar target/Maximum-1.0-SNAPSHOT.jar net.examples.Maximum input/
  output5/

$ hdfs dfs -cat output5/part-r-0000*
you 3430
is 1881
the 12739
as 1591
of 6427
his 2911
in 3208
from 894
to 6580
```

Notice: De fapt nu am determinat maximul ci primele 10 valori. Setaţi numărul de reducers la 1.

Notice: Băgaţi de seamă că valoarea maximă este diferită de cea calculată cu programul Python. De ce?

4 TF-IDF

Scorul TF-IDF (Term Frequency - Inverse Document Frequency) este folosit pentru a marca importanţa unui cuvânt într-un document. El este format din produsul dintre TF şi IDF.

Cu cât un cuvânt apare mai frecvent într-un document, cu atât scorul său TF este mai mare:

$$TF_{wd} = \frac{wordCount}{wordsPerDocument} \tag{1}$$

Scorul TF este calculat pentru fiecare cuvânt (dintr-un document) în parte, ca raportul dintre numărul de apariţii al cuvântului în document şi numărul total de cuvinte din document.

Pe de altă parte, dacă acelaşi cuvânt apare în toate documentele (sau în majoritatea), relevanţa sa trebuie să scadă. Scorul IDF "ponderează" importanţa cuvântului:

$$IDF_{wd} = \log \frac{noDocuments}{docsPerWord} \tag{2}$$

IDF este egal cu raportul dintre numărul total de documente şi numărul de documente care conţin cuvântul, logaritmat. Dacă un cuvânt apare în toate documentele, el este irelevant, iar acest lucru este indicat de scorul său IDF care devine 0.

Scorul unui cuvânt devine astfel:

$$score_{wd} = TF_{wd} \cdot IDF_{wd} \tag{3}$$

Pentru a putea calcula scorul TF-IDF, va trebui să procesaţi toate documentele şi să emiteţi iniţial perechi de tip (word@document, 1), pe care le consolidaţi în tuple (word@document, wordCount).

Mai departe, veţi consolida aceste tuple introducând şi numărul de cuvinte care apar în fiecare document: (word, document, wordCount, wordsPerDocument).

Penultima etapă înainte de calcularea scorurilor presupune numărarea documentelor care conţin acelaşi cuvânt: (word, document, wordCount, wordsPerDocument, documentsPerWord).

Numărul de documente puteţi să îl hardcodaţi. Încercaţi ca la fiecare etapă de reduce să folosiţi mai mulţi reduceri şi nu doar unul singur.

Implicit, delimitatorul dintre cheie şi valoare folosit de Streaming API este caracterul TAB. Dacă acesta nu e prezent pe o linie, atunci reducer-ul presupune că întreaga linie este cheia. Pentru a seta caracterul delimitator dintre cheie şi valoare folosiţi definiţia stream.map.output.field.separator.

5 Exerciţii propuse

Question 1

Rulaţi codul mapper-ului "identity.py" şi al reducer-ului "maximum.py" după indicaţiile din text. Inspectaţi maximul. Rulaţi apoi folosind un număr de 3 reduceri. Ce observaţi că se scrie în folder-ul de ieşire? Sunt cele 3 valori obţinute, cele mai mari 3 valori?

Question 2

Modificaţi programele Python pentru mapper şi reducer pentru determinarea cuvântului cu frecvenţa maximă. Ne interesează acum primele K cuvinte cu frecvenţele cele mai mari. Rezolvaţi problema pentru un K variabil şi verificaţi, folosind programul WordCount şi "sort" în linie de comandă, că rezultatele sunt cele aşteptate.

Question 3

Modificaţi programul Maximum.java pentru aflarea nu doar a cuvântului cu frecvenţa maximă, ci a primelor K cuvinte cu frecvenţele cele mai mari. Rezolvaţi problema pentru un K variabil şi verificaţi, folosind programul WordCount şi "sort" în linie de comandă, că rezultatele sunt cele aşteptate.

Question 4 *(avansat)*

Consideraţi mai multe cărţi în limba engleză (downloadaţi 5 de la gutenberg.org). Folosind un job înlănţuit de tip map-reduce, calculaţi scorul Term-Frequency / Inverse Document Frequency (TF-IDF) pentru fiecare cuvânt. Rezolvaţi mai întâi problema folosind Python şi Streaming API şi apoi folosind cod Java.

Bibliografie

1. Chuck Lam, “Hadoop in Action”, Manning Publications, 2011
2. Hadoop Streaming, https://hadoop.apache.org/docs/r1.2.1/streaming.html