Cloud Computing

Bloom Filters in Map Reduce

Report

Developed by

Fabio Cognata, Luca Di Giacomo,

Daniele Laporta, Francesco Zingariello

Summary

Introduction

The aim of the project is to build a bloom filter over the ratings of movies listed in the IMDb datasets.

The overall job is divided into 3 main area: Design, Implementation and Results.

Firstly, the design phase involves the creation of the MapReduce algorithm, presented using a pseudocode. These instructions are useful to implement the bloom filters.

Then, it follows the implementation of the already discussed algorithm using both the Hadoop and the Spark framework, separately.

Lastly, a test phase for both the implementations on the IMDb ratings dataset, computing the exact number of false positives for each rating, has been executed on a cluster of 4 virtual machines provided by the University of Pisa.

Repository link: projectbloomfilter (github.com)

Design

Overall implementation idea

As far as looking at first sight at the dataset, the overall implementation idea bases on the creation of a single bloom filter for each rating. So this means having 10 bloom filters associated each one to a specific rating going from 1 to 10.

Prerequisites

In order to develop the MapReduce algorithm is needed the installation of the Virtual Machines provided by the University of Pisa and group them as a cluster, which are the following:

IP	Name on the cluster
172.16.4.179	hadoop-namenode
172.16.4.142	hadoop-datanode-2
172.16.4.197	hadoop-datanode-3
172.16.4.171	hadoop-datanode-4

On each VM is carried out the installation of the latest version of Hadoop and Spark, running through YARN.

For the implementation is used VSCode as principal IDE and Git to take track of the versioning development.

Pseudocode

The pseudocode divided into “Ratings instances counting”, “Bloom filter construction” and “Bloom filter testing” are shown below:

RATINGS INSTANCES COUNTING

```
class MAPPER:
  method MAP(chunkid cid, chunk c):
    foreach line in c:
      int vote <- round(line[1])
      int n <- 1
      EMIT(vote, n)

class REDUCER:
  method REDUCE(int vote, counts [c1, c2 ...]):
    int total <- sum(counts)
    EMIT(vote, total)
```

BLOOM FILTERS CONSTRUCTION

```
class MAPPER:
  method MAP(chunkid cid, chunk c):
    foreach line in c:
      int vote <- round(line[1])
      str title <- line[0]
      EMIT(vote, title)

class REDUCER:
  method REDUCE(int vote, counts [t1, t2 ...]):
    bloomfilter bf <- new bloomfilter(n,p)
    file f <- new file(outputpath+"filter_"+vote)
    foreach el in counts:
      bf.add(el)
    write(f, bf)

class BLOOMFILTER:
  int nbits
  int nhash
  hash[nhash] h
  boolean bitfield[nbits]

  constructor(int num_entries, double pFP):
    m <- %formula_nbits%
    k <- %formula_nhash%
    this.nbits <- m
    this.nhash <- k
    foreach i in [0,k-1]:
      h <- new murmurhash

  add(string word):
    foreach hashf in h:
      bitfield[hashf(word)] <- True

  check(string word):
    foreach hashf in h:
     if bitfield[hashf(word)] == False:
       return False
    return True
```

BLOOM FILTERS TESTING

```
class MAPPER:
  method MAP(chunkid cid, chunk c):
    foreach line in c:
      int vote <- round(line[1])
      str title <- line[0]
      EMIT(vote, title)

class REDUCER:
  method REDUCE(int vote, counts [t1, t2 ...]):
    foreach el in counts:
      foreach i in [1,10]:
        if bloomfilter[i].check(el) == True:
          if i == vote:
            true_positive[i]+=1
          else:
            false_positive[i]+=1
        else:
          if i != vote:
            true_negative[i]+=1
          else:
            false_negative[i]+=1
    write(output_file, results)
```

Implementation

In this section is presented the implementation of the overall project divided in small phases, some common for both Hadoop and Spark.

Hadoop Libraries

For implementing the hashing functions of the Bloom filter class the following libraries are being used:

- org.apache.hadoop.util.bloom.HashFunction;
- org.apache.hadoop.util.bloom.Key;
- org.apache.hadoop.util.hash.Hash;

For handling the HDFS and the inputs formatted as lines, the following libraries are being used:

- org.apache.hadoop.fs.FSDataOutputStream;
- org.apache.hadoop.fs.FSDataInputStream;
- org.apache.hadoop.fs.FileSystem;
- org.apache.hadoop.mapreduce.lib.input.NLineInputFormat;

Spark Libraries

For implementing the hashing function of the Bloom Filter class the ***mmh3*** libraries has been used by hashing the string and passing the vote as the seed of the hash for finally modulating over the total length of the filter.

To implement the spark application the ***PySpark*** library has been used with the support of ***colorama*** library to make the on screen results more readable.

HDFS Structure

For the hadoop user in the namenode machine, a folder “~/project” has been created containing all the necessary scripts and files for running the execution of both spark and hadoop programs, then in HDFS the structure has been splitted for the various jobs:

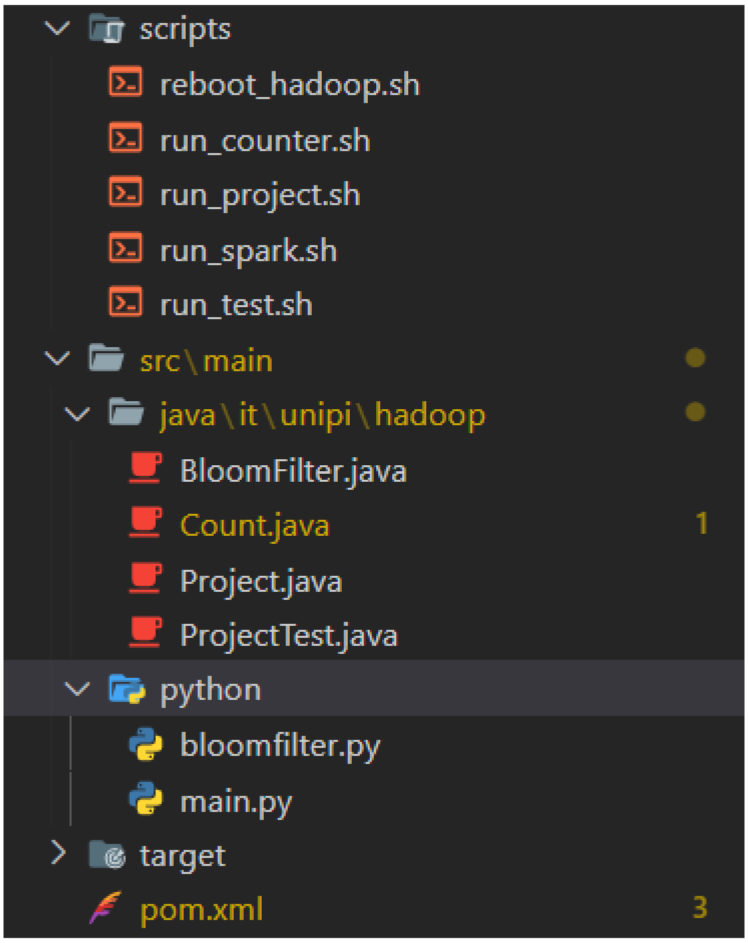
- */project/counters/output* is the folder containing the various counts for each rank’s instances named as *count_i* where i stands for the rank value

- */project/output/filters* is the folder containing the various filters serialized by Hadoop Writable system, saved in files named as *filter_i* where i stands for the rank value
- */project/test* is the folder containing the test results saved by Hadoop context.write method in the format *Rank - #TP - #TN - #FP - #FN*
- /project/spark_output for the spark's pickle file saving and retrieval of the bloom filters for the second python branch of the project.

Application project source organization (packages)

Following is presented the packages organization and the java classes necessary for the application, including some snapshots of the code.

The picture below represents the packages as are presented into VSCode:



```
∨ scripts
    reboot_hadoop.sh
    run_counter.sh
    run_project.sh
    run_spark.sh
    run_test.sh
∨ src\main                         ●
  ∨ java\it\unipi\hadoop           ●
      BloomFilter.java
      Count.java                   1
      Project.java
      ProjectTest.java
  ∨ python
      bloomfilter.py
      main.py
> target
  pom.xml                          3
```

First of all, it is due to say that as package prefix the reverse domain of the University of Pisa has been used and organized in packages by layers.

The implementation of Hadoop is written in Java code. In particular:

- *BloomFilter.java* describes the bloom filter class with the implementation of the specific formulas to calculate the needed parameters.
- *Count.java* describes the implementation of the job dedicated to count each rating instances.
- *Project.java* is the job that creates the bloom filters and saves a file for each bloom filter created containing its serialized instances.
- *ProjectTest.java* is the testing job for the bloom filters. To be technical, it takes the bloom filters saved on the last job and produces an output file containing the characteristics of the test.

The implementation of Spark, instead, is written in Python code and contains:

- *Bloomfilter.py* which contains the bloom filter class, so it is the alter ego of the *Bloomfilter.java.*
- *main.py* contains all the implementation needed such as the creation and testing of the bloom filters.

Main differences between Hadoop (Java) and Spark (Python)

Even though the task is the same for both the frameworks Hadoop and Spark, the implementation is quite different for some aspects, like the followings:

- In java, for each task is needed to assign a job and so a class that contains the mapper and the reducer. Thus jobs are separated and to share resources is necessary to use and recall appropriate files.

  Instead in python there is a context that manages the jobs which are composed by a series of stages. Precisely, there is the memory of the driver that contains all the information that can be easily used among different nodes.

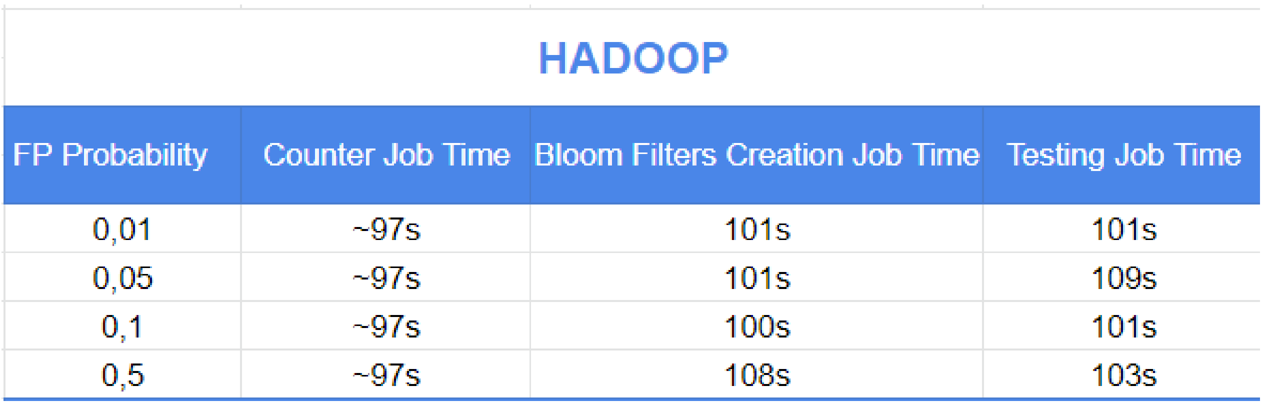
- In Java the shuffle and sort is done entirely by Hadoop. The management of inputs and output has been consequently obtained through the usage of the *cleanup* and *setup* methods.

  Instead in python this phase must be explicated by the programmer using appropriate functions like *sortByKey* or *groupByKey*.

Project utilities

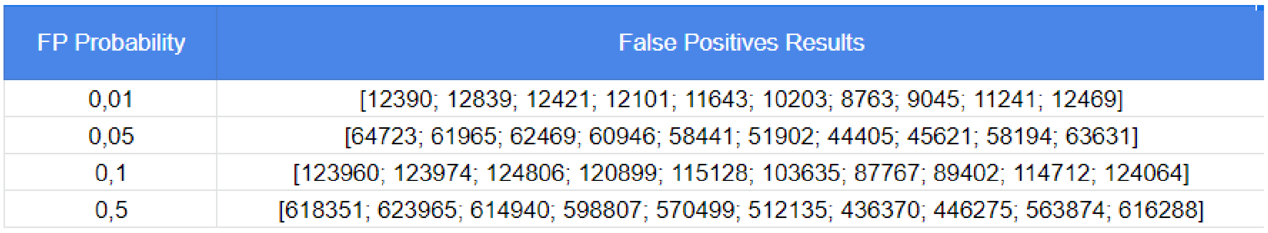
- To speed-up the process of loading the .jar and the .py files, some shell scripts have been written and loaded into the Namenode of Hadoop’s cluster. Those scripts include the command to delete the previous outputs running *hdfs dfs -rm -r -f* command, then the line to copy the needed files into the *~/project* folder and finally the run command for the needed framework.
- To format the Spark output and make it more readable some utility functions have been written to loop over the stages results and print them by selecting the color (using colorama library).
- To perform quicker tests over the two frameworks a shorter version of the original *title_ratings.tsv* file taken from the IMDb datasets has been created and named *tsr_short.tsv* containing a small sample of the original file(about 60 rows)

Results

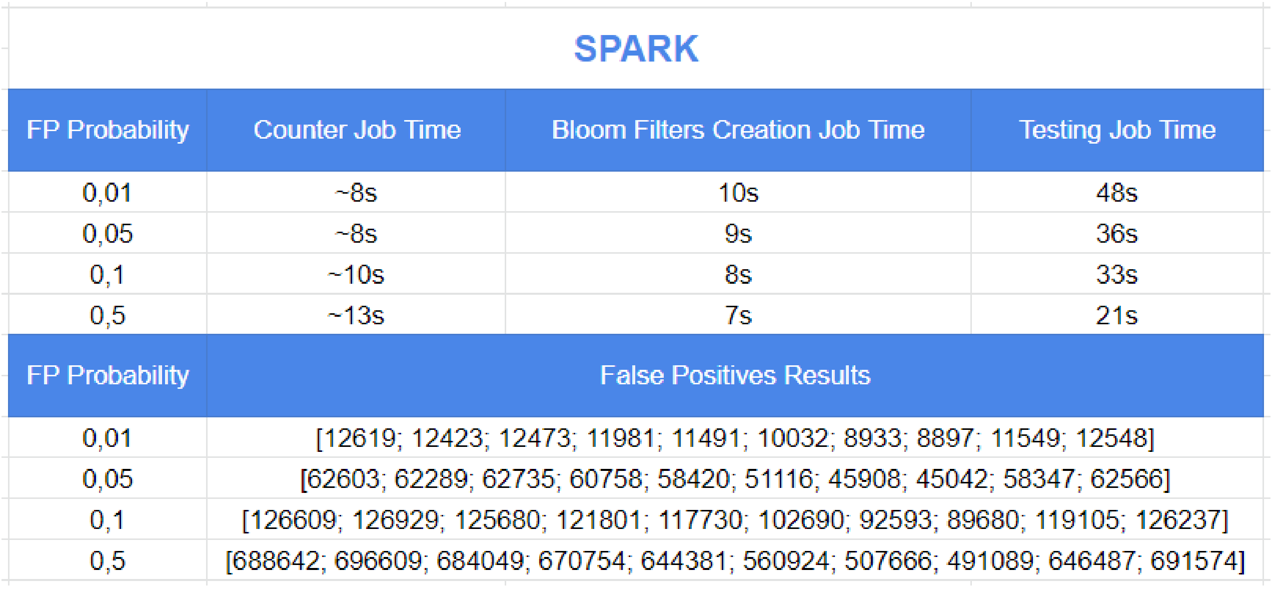


HADOOP

FP Probability	Counter Job Time	Bloom Filters Creation Job Time	Testing Job Time
0,01	~97s	101s	101s
0,05	~97s	101s	109s
0,1	~97s	100s	101s
0,5	~97s	108s	103s



FP Probability	False Positives Results
0,01	[12390; 12839; 12421; 12101; 11643; 10203; 8763; 9045; 11241; 12469]
0,05	[64723; 61965; 62469; 60946; 58441; 51902; 44405; 45621; 58194; 63631]
0,1	[123960; 123974; 124806; 120899; 115128; 103635; 87767; 89402; 114712; 124064]
0,5	[618351; 623965; 614940; 598807; 570499; 512135; 436370; 446275; 563874; 616288]



SPARK

FP Probability	Counter Job Time	Bloom Filters Creation Job Time	Testing Job Time
0,01	~8s	10s	48s
0,05	~8s	9s	36s
0,1	~10s	8s	33s
0,5	~13s	7s	21s

FP Probability	False Positives Results
0,01	[12619; 12423; 12473; 11981; 11491; 10032; 8933; 8897; 11549; 12548]
0,05	[62603; 62289; 62735; 60758; 58420; 51116; 45908; 45042; 58347; 62566]
0,1	[126609; 126929; 125680; 121801; 117730; 102690; 92593; 89680; 119105; 126237]
0,5	[688642; 696609; 684049; 670754; 644381; 560924; 507666; 491089; 646487; 691574]

Conclusions

Has been ascertained that Spark works in much lower time. In all the tasks involved and with all the configurations tested, in fact, Spark has reached performances that exceed the 40% of efficiency compared to hadoop ones.

Spark has also been preferred for the possibility to decide whether to load datas in memory or to save them in files, giving much more flexibility in code's structure and performance management.