

University of Pisa



Cloud Computing

**Project of
Bloom Filters in MapReduce**

Student

Davide Vigna

Summary

Abstract.....	3
Requirements.....	3
Analysis and Design.....	4
Implementation	11
Results	15
References	17

Abstract

The aim of this project is to design and realizing a set of bloom filters used to determining the presence of a rating related to the IMDb dataset.

The design phase consists of choosing proper parameters to build a series of bloom filters and to design a cloud computing parallel solution to manage large amount of data for the construction of the set of filters. Subsequently, to design a solution to compute the exact number of false positive for each rating, for each filter in the set.

The implementation phase consists of developing a MapReduce solution for the construction of the set of bloom filters using Hadoop framework and to compute the FPR.

Finally, a short report of the experimental result obtained is produced to evaluate the design assumptions.

Requirements

Functional Requirements

The application should round the average rating value to the closest integer value and assign the corresponding rating to the proper bloom filter. For each rating value (1-10) will be construct and populate a different bloom filter. The same round operation will be used to test the false positive rates, in the test phase.

Non-functional Requirements

A proper usage of memory is required in order to obtain good performances in term of time. It's necessary to evaluate if dividing the dataset in splits can bring benefits to avoid to concentrate all the computation on few mappers and at the same time avoid too much network overhead. The number of splits to assign to the mappers and the number of reducers to use determine how faster the outcome is computed.

In the Hadoop implementation, it's mandatory to use the following classes:

- `org.apache.hadoop.mapreduce.lib.input.NLineInputFormat`: splits N lines of input as one split
- `org.apache.hadoop.util.hash.Hash.MURMUR_HASH`: the hash function family to use.

It is not possible to use the Bloom Filter data structure in the `org.apache.hadoop.util.bloom` package, so a different implementation must be provided.

Analysis and Design

The dataset used in this project consists of 1248408 records and it's was extracted from [IMDb](#) site on the 01/06/22. It has a size of 20.5 MB. Each record is composed of three information separated by a 'tab' character in the following order:

- a unique identifier representing film's title (string)
- a number representing the average rating assigned to a film (float)
- a number representing how many votes are given to that film (integer)

This dataset is splitted randomly in two parts.

The first part is made of 1000000 of records (above the 75% of the global dataset) selected randomly and it is used to build the set of bloom filters. The size of this new dataset it is now 17,4 MB and it will be call with the name ***CostructionDataset*** from now to on.

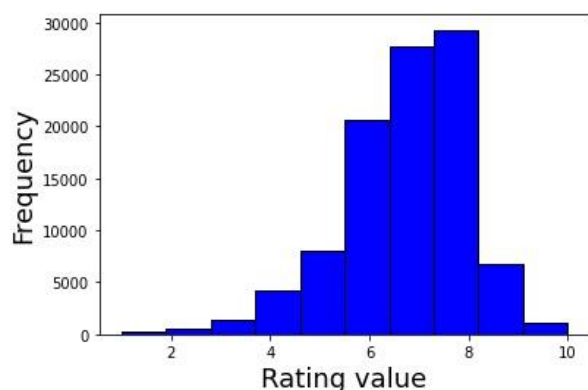
The second part is made of the remaining 248408 records (approximately the 25% of the original dataset). This is used in the final section to test the set of bloom filters with a dataset that is not used during construction phase. The size of this partition is a file of 4,26MB.

Considering that a rating can assume a vote in the range (1 to 10), it's necessary to create 10 bloom filters, each one with a different size because not all the votes have the same probability distribution. For the sake of simplicity, it would be possible to use the same size for all the filters but it would be a waste of memory for low frequency ratings.

Taking into account a sample of 10% of records of the entire *CostructionDataset* (selected randomly), it is evident that the most of the ratings have an evaluation in the range of (6-8) and instead at the extremes there is low number of ratings.

The next pictures contain a frequency representation of a random sample extracted from *CostructionDataset*.

Rating_value	Frequency
1	189
2	610
3	1335
4	4174
5	8186
6	20418
7	27725
8	29373
9	6909
10	1081



The first thing to analyse is the value of parameter n (number of elements). For each filter the number of elements is estimated considering the proportional data to be managed (on the overall *CostructionDataset*). Some approximations are done in order to have a total number of 1000000 elements.

The choice of p (false positive rate) follows this consideration:

if a filter has a low number of elements, it will have a lower probability to be accessed in the future so it's more tolerable to have a higher false positive rate in this case respect to filters whose elements number are higher. Instead, in opposite situation, it is more useful to spend more memory and use more hash function in order to save time avoiding to read more often the entire dataset and reducing the false positive rate.

Following this strategy:

- for rates with a probability lower than 4%, the value chosen for p is equal to 0.1
- for rates with a probability greater than 4% and lower than 20%, the value chosen for p is equal to 0.01
- for rates with probability greater than 20%, the value chosen for p is equal to 0.001.

To determine the optimal values of m (number of bits) and k (number of hash functions) the following formulas are used:

$$m = -\frac{n \ln(p)}{(\ln 2)^2}$$

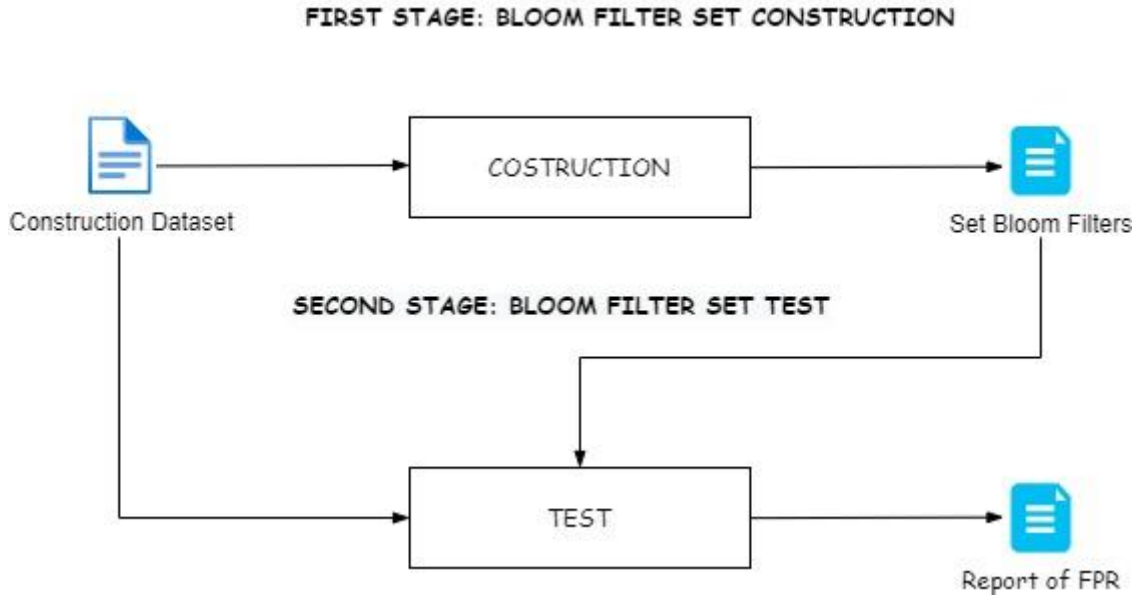
$$k = \frac{m}{n} \ln 2$$

The following table summarize all the parameters and highlights the memory occupation.

Filter (Rating Value)	Freq/(Sample size) %	n	p	m (bits)	k	m(Byte)
1	0,189	2000	0.1	9586	3	$\approx 1,20 \text{ KB}$
2	0,610	6100	0.1	29235	3	$\approx 3,65 \text{ KB}$
3	1,335	13400	0.1	64220	3	$\approx 8,03 \text{ KB}$
4	4,174	41800	0.01	400656	7	$\approx 50,08 \text{ KB}$
5	8,186	82000	0.01	785975	7	$\approx 98,25 \text{ KB}$
6	20,418	204200	0.001	2935904	10	$\approx 366,98 \text{ KB}$
7	27,725	277300	0.001	3986906	10	$\approx 498,36 \text{ KB}$
8	29,373	294000	0.001	4227011	10	$\approx 528,38 \text{ KB}$
9	6,909	69100	0.01	662328	7	$\approx 82,79 \text{ KB}$
10	1,081	10100	0.1	48405	3	$\approx 6,05 \text{ KB}$
		1000000				$\approx 1.64 \text{ MB}$

Considering the size of the *CostructionDataset* (17,4 MB), the set of all bloom filters permits to save the memory about 10 times. In addition, choosing a different value of p for each specific case, allow to save computational time (needed to set/read bits, computing hash functions). Filters with few elements and high false positive rate will be faster respect to the other because they have a lower number of hash function to compute.

In this part it is shown how the parallel algorithm should work in map reduce framework. The entire work can be divided in two main stages: the first stage consists in the construction of the set of bloom filters; the second stage uses the produced set to test the exact number of false positive for each rating in each filter.



The following lines summarize in pseudo code what mapper and reduce respectively do in the first stage. In the next page there are shown the core functions needed to initialize the data structures and to use a bloom filter.

Algorithm 1 Class Mapper

```

1: procedure SETUP(Context ctx)
2:    $\{n1, \dots, n10, p1, \dots, p10\} \leftarrow ctx.getConfiguration();$ 
3:    $BloomFilter1 \leftarrow initializeBloomFilter1(n1, p1);$ 
4:   ...
5:    $BloomFilter10 \leftarrow initializeBloomFilter10(n10, p10);$ 
6:    $BloomFiltersSet \leftarrow \{BloomFilter1, \dots, BloomFilter10\}$ 

1: procedure MAP(Key k, Rating r)
2:    $Vote \leftarrow round(r.avgRate);$   $\triangleright$  take closest integer
3:    $BloomFilterVote \leftarrow selectFilterFromSet(Vote);$ 
4:    $BloomFilterVote.insertKey(r.idRating);$ 

1: procedure CLEANUP(Context ctx)
2:   for all BF in BloomFilerSet do
3:     EMIT ( $BF.index, BF$ );

```

Algorithm 2 Class Reducer

```
1: procedure SETUP(Context ctx)
2:    $\{n1, \dots, n10, p1, \dots, p10\} \leftarrow ctx.getConfiguration();$ 

1: procedure REDUCE(Key idxFilter, BloomFilters[BF1, BF2, ...] )
2:    $BF_{new} \leftarrow initEmptyBloomFilter(idxFilter);$ 
3:   for all BF in BloomFilters do
4:      $BF_{new} \leftarrow BF_{new}.or(BF);$ 
5:   EMIT (idxFilter, BFnew);
```

Algorithm 3 Class BloomFilter

```
1: procedure INITIALIZEBLOOMFILTER(Integer n, Double p)
2:    $m \leftarrow -\frac{n \ln p}{(\ln 2)^2};$  ▷ number of bits
3:    $k \leftarrow \frac{m}{n}(\ln 2);$  ▷ number of hashes
4:    $bits \leftarrow BitVector(m);$  ▷ storage structure

1: procedure INSERTKEY(String idRating)
2:   for i=0 to k do
3:      $bitIndex \leftarrow MurmurHash(i, idRating);$ 
4:      $bits[bitIndex] \leftarrow 1;$ 
5:      $i \leftarrow i + 1;$ 

1: procedure CONTAINSKEY(String idRating) ▷ to test FPR
2:   for i=0 to k do
3:      $bitIndex \leftarrow MurmurHash(i, idRating);$ 
4:     if  $bits[bitIndex] \neq 1$  then
5:       return FALSE
6:      $i \leftarrow i + 1;$ 
7:   return TRUE

8:
9: procedure OR(BloomFilter bf) ▷ Or bit a bit
10:  for i=0 to m do
11:     $bits[i] \leftarrow bits[i] \mid bf.bits[i];$ 
12:     $i \leftarrow i + 1;$ 
```

The mapper class is designed to compute the insertion of keys inside each bloom filter in a parallel way. At the very begin it initializes 10 bloom filter data structure according to the design assumption. For each rating received in input it will decide what filter is responsible for managing the information according to the rating value, then it inserts the key into the filter, computing the designed hash functions specific for that filter. When the input split is processed, the mapper sends each filter inside the set to a reducer as key-value pair, indicating as key the rating value managed by the filter and as a value the bloom filter itself.

The reducer class is designed to take as input all the bloom filters, related to a specific key, and compute the or operation (bit a bit) on each filter. The result of this operation is a final filter used in the next phase to test the exact number of false positive rate.

To make pseudocode more readable, operations for managing the real input received (like splitting) are omitted and some of the functions are not specified in detail but summarised with a human friendly word.

An In-Mapper combiner is used to reduce the network traffic. Each mapper stores internally a set of 10 bloom filters. In this way it's avoided to send duplicate data in the network having a complete local aggregation. The size of the entire set, designed in the previous section, is acceptable for each mapper memory and improve the performance of the program.

Two approaches are evaluated to send data to the network:

- the first idea (initially) was to use the **pairs** approach in which for each filter, the mapper should send to reducers the bit position in the bloom filter set to 1. Generating in the worst-case $O(10*m)$ data in $O(1)$ space. Of course, not all the bits in a bloom filter are set to 1 but generally the most of them do, so this solution is not the best one and it's not considered;
- an alternative idea (effectively used) is to use the **stripes** approach. Instead of sending each bit position, the mapper emits at most 10 key-value pairs, where vales are the entire set of bits of a filter. Generating $O(10)$ data in $O(m)$ space, where m is the maximum bits dimension of a bloom filter in the set. In this solution even the bits to 0 are sent but they are lower than the other set to 1 and in this scenario the network traffic is drastically reduced. This solution is shown in the previous pseudo code.

Another aspect to take in consideration is the input split size. This determines implicitly how many map tasks Hadoop should create. Looking at pseudocode it's easy to understand that lower this number is, lower is the number of map tasks too but each mapper has to do more computation. Instead, if this number is high, the computation will be faster in each single mapper but also the number of key-value pairs produced will be high, so the reduce computation increases a bit and even the network overhead.

If the number of splits is equal to g , each mapper handles n/g records producing in output 10 key value-pairs (as before), however each reducer has an incrementation of the elements in the list associated with a single intermediate key. A reducer handles now $10 * g$ elements.

The choice of the value of g depends a lot on the dimension of the dataset and on memory limitation of the machine where each mapper executes its tasks.

The next part is about the second stage of the work and consists in reloading the set of bloom filters from HDFS created in the previous stage and for each filter calculating the false positive rate using the construction dataset.

The false positive rate is computed using the following formula:

$$FPR = \frac{FP}{FP + TN}$$

where:

- FP (false positive) is the number of ratings that are evaluated as present checking a bloom filter but it is sure apriori that it is not.
Example: if a rating with a rating value of 5 is checked in a bloom filter, who contains information about ratings of value 1 and returns present this is a case of false positive for filter number 1.
- TN (true negative) is the number of ratings that are not present in that filter (a priori).
Example: a rating with a rating value of 5 is checked in a bloom filter, who contains information about ratings of value 1. This is an obvious case of false negative.

In order to check if the filters construction has been done well, two other values are calculated and their values should be equal to:

$$TP = TOT_RECORD - TN$$

$$FN = 0$$

where:

- TP (true positive) is the number of ratings that are present a priori in that filter.
Example: a rating with a rating value of 5 is checked in a bloom filter, who contains information about ratings of value 5.
- FN (false negative) is the number of ratings that are evaluated as not present but instead do
This case should never be present for the property of a bloom filter.

The following lines summarize in pseudo code what mapper and reduce respectively do in the second stage.

Algorithm 1 Class Mapper

```

1: procedure SETUP(Context ctx)
2:   BloomFilerSet  $\leftarrow$  loadBloomFiltersFromHDFS();
3:   CounterListTP  $\leftarrow$  {0, 0, ..., 0};
4:   CounterListFP  $\leftarrow$  {0, 0, ..., 0};
5:   CounterListTN  $\leftarrow$  {0, 0, ..., 0};
6:   CounterListFN  $\leftarrow$  {0, 0, ..., 0};

1: procedure MAP(Key k, Rating r)
2:   Vote  $\leftarrow$  round(r.avgRate); ▷ take closest integer
3:   for i=0 to BloomFilerSet.size do
4:     BF  $\leftarrow$  BloomFilerSet[i];
5:     if i = (Vote - 1) then ▷ rating inserted
6:       CounterListTP[i]  $\leftarrow$  CounterListTP[i] + 1;
7:       if !BF.containsKey(r.idRating) then
8:         CounterListFN[i]  $\leftarrow$  CounterListFN[i] + 1;
9:       else ▷ rating not inserted
10:        CounterListTN[i]  $\leftarrow$  CounterListTN[i] + 1;
11:        if BF.containsKey(r.idRating) then
12:          CounterListFP[i]  $\leftarrow$  CounterListFP[i] + 1;
13:      i  $\leftarrow$  i + 1;

1: procedure CLEANUP(Context ctx)
2:   for i=0 to BloomFilerSet.size do
3:     localStat  $\leftarrow$  concatValuesCounterList(i);
4:     EMIT (i + 1, localStat);
5:     i  $\leftarrow$  i + 1;

1: procedure CONCATVALUESCOUNTERLIST(Integer i)
   return CounterListTP[i] + " - " + CounterListFP[i] +
   " - " + CounterListTN[i] + " - " + CounterListFN[i];

```

```

1: procedure REDUCE(Key idxFilter, Counters[locStat1, locStat2, ...] )
2:   CounterTP ← 0;    ▷ Check eq. to max elems if filter ok
3:   CounterFP ← 0;
4:   CounterTN ← 0;
5:   CounterFN ← 0;    ▷ Check eq. to 0, if filter is ok
6:   for all stat in Counters do
7:     CounterTP ← CounterTP + stat.TP;
8:     CounterFP ← CounterFP + stat.FP;
9:     CounterTN ← CounterTN + stat.TN;
10:    CounterFN ← CounterFN + stat.FN;
11:     $FPR \leftarrow \frac{Counter_{FP}}{Counter_{FP} + Counter_{TN}}$ ;
12:    EMIT (idxFilter, FPR);

```

Stage nr. 2 has to do less computations respect to stage nr. 1.

For each element in the split, the mapper class obtain the rounded rating value of the current rating. In this way it is able to determining if that rating has been inserted in the filter or not before. Then for each filter update a specific count depending on the case where that rating is. At the end it emits a unique string per filter containing all the statistic information calculated.

The reducer class receives as input from the mapper all the statistics and for each filter summarize them to compute its relative FPR, then it emits the outcome.

Implementation

The project is implemented installing Hadoop and HDFS in a fully-distributed mode in order to simulate as much as possible the behaviour of the framework in a small real cluster of machines. In particular it has been used 2 physical machines connected together in the same LAN. On each machine has been installed [Virtual Box](#) software and 3 virtual machines, hosting Ubuntu operating system, have been configured in this way: on the first physical machine (a desktop computer), 2 virtual machines have been created and 4.5 GB of RAM memory have been assigned; on the second physical machine (a laptop computer) 1 virtual machine has been created and 4 GB of RAM has been assigned.

The cluster architecture can be summarised as:

- VM n.1 is the *nameNode* and it is identified by the local ip address 192.168.1.48
- VM n.2 is a *dataNode* and it is identified by the local ip address 192.168.1.103
- VM n.3 is a *dataNode* and it is identified by the local ip address 192.168.1.104

In this part some of the most interesting implementation details are explained:

- Bloom filter structure

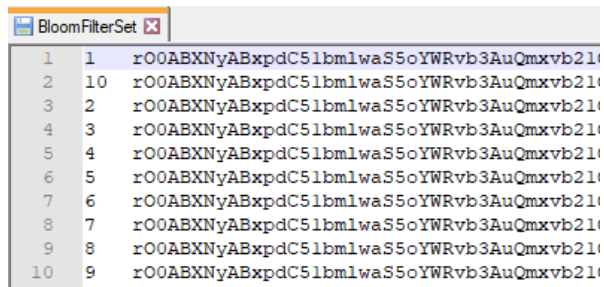
In the first stage a customized data structure named BloomFilter is implemented to exchange data between mappers and reducer. This class extends a writable interface and so it uses the Hadoop's serialization format for interprocess communication, decreasing the data size overhead. This data structure contains a bitSet element to represent a vector of bits that can grow as needed, however the dimension is decided at the begin in the constructor method. The aim of this project is not to create a dynamic bloom filter but a filter with fixed positions decided in the design phase. Each time an operation involves to manage a particular index of the bitSet, a proper bit mask is used to work specifically at bit level and not with the entire representative byte.

- Murmur Hash Function

This kind of hashing function is used internally by the BloomFilter in 2 methods *addKey* and *containsKey* to compute respectively the index of the bit to set to one or read if a bit is set or not. On the BloomFilter constructor method, an instance of this class is implemented and it is called different times with seeds in the range of 1 to the number of hash functions to use (k). For each different seed it generates a different index. It's very simple to implement and it has fast performances with good results.

- Serialization of BloomFilters set on HDFS

The first stage produces in output a file containing the set of all bloom filters computed in the various mapper and summarized in the reducer. This file is then reloaded by the second stage to compute the FPR test. This file includes for each filter, each bit vector value. However, if each value is written in plain in output, the dimension of the output file becomes greater than the input given to Hadoop, this because each character (used to represent a bit value) is not a bit value but a representation of a character. To solve this problem a new class is introduced: *BloomFilterSerializable*. This class is equivalent to the BloomFilter containing information of the data to store, implementing static methods for encoding and decoding in base64. An example of the file produced in output and stored in HDFS in the first stage is shown in the next picture. This situation is obtained used only 1 reducer, alternative it would have n of this file with n equal to the number of reducers.



Line Number	String
1	1 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
2	10 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
3	2 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
4	3 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
5	4 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
6	5 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
7	6 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
8	7 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
9	8 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210
10	9 r00ABXNyABxpdC51bmlwaS5oYWRvb3AuQmxvb210

- Number of lines to consider in a single split and performance

This number is fundamental because allow to partition the dataset in several splits and consequently determine the number of map task to instantiate. It is possible to assign this value in the driver class during the configuration settings:

```
conf.setInt(NLineInputFormat.LINES_PER_MAP, SplitSize);
```

The following table summarize the experimental results obtained assigning different values to this number in the first stage:

Split size	Number of map task	Total time spent by all maps in occupied slots
1000000	1	5100 ms
500000	2	7063 ms
250000	4	12936 ms
125000	8	64901 ms
100000	10	91009 ms

This result is surprising because it suggests to avoid splitting the dataset and using just one only mapper however it's quite normal because it doesn't take in consideration the initial dataset size. The minimum amount of data that HDFS can read is about 64MB and the initial dataset is 17.4MB so it doesn't take full advantage of Hadoop's capability.

- Number of reducers to consider and performance

It is possible to fix the number of reduce task in the driver class during the configuration settings:

```
job.setNumReduceTasks(numberOfReducers);
```

The next table shows the performances of the reducers in the first stage depending on the number chosen:

Number of reduce task	Total time spent by all reducers in occupied slots
1	3850 ms
2	5091 ms
4	18216 ms
8	39700 ms
10	76227 ms

Like for the mappers, increasing the number of reducers make increase the overall computational time. This introduces more I/O operations because the data need to be split across the reducers are more and it requires time. Being the dataset too small, it's not possible to observe the benefits that can be obtained from multiple reduces.

In this case having multiple reducers in the first stage imply also that the set of bloom filter is written on different HDFS files, so in the next stage it requires more read operations to join all the filters together in one only set.

For these reasons the number of reducers is fixed to one.

Once installed and package the application in HDFS, the command to run to execute the entire job is:

```
hadoop jar BloomFilterProject-1.0.jar it.unipi.hadoop.Driver data.tsv output2 500000
```

where:

data.tsv is the name of the *costructionDataset*

output2 is the name of the folder of the final stage output

500000 is the split size

A possible output for the execution is this:

```
hadoop@hadoop-namenode:~$ hadoop jar BloomFilterProject-1.0.jar it.unipi.hadoop.Driver data.tsv output2 500000
Old output-BloomFilterSet folder deleted!
Old output-output2 folder deleted!
2022-07-01 22:12:38,853 INFO client.RMProxy: Connecting to ResourceManager at hadoop-namenode/192.168.1.48:8032
2022-07-01 22:12:39,053 WARN mapreduce.job.ResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2022-07-01 22:12:39,077 INFO mapreduce.job.ResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/hadoop/.staging/job_1656706245056_0003
2022-07-01 22:12:39,165 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhostTrusted = false, remoteHostTrusted = false
2022-07-01 22:12:39,318 INFO input.FileInputFormat: Total input files to process : 1
2022-07-01 22:12:39,339 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhostTrusted = false, remoteHostTrusted = false
2022-07-01 22:12:39,471 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhostTrusted = false, remoteHostTrusted = false
2022-07-01 22:12:39,558 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhostTrusted = false, remoteHostTrusted = false
2022-07-01 22:12:39,585 INFO mapreduce.job.Submitter: number of splits:2
2022-07-01 22:12:39,692 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localhostTrusted = false, remoteHostTrusted = false
2022-07-01 22:12:39,750 INFO mapreduce.job.Submitter: Submitting tokens for job: job_1656706245056_0003
2022-07-01 22:12:39,751 INFO mapreduce.job.Submitter: Executing with tokens: []
2022-07-01 22:12:39,882 INFO conf.Configuration: resource-types.xml not found
2022-07-01 22:12:39,882 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2022-07-01 22:12:39,932 INFO impl.YarnClientImpl: Submitted application application_1656706245056_0003
2022-07-01 22:12:39,965 INFO mapreduce.job: The url to track the job: http://hadoop-namenode:8088/proxy/application_1656706245056_0003/
2022-07-01 22:12:39,965 INFO mapreduce.job: Running job: job_1656706245056_0003
2022-07-01 22:12:47,356 INFO mapreduce.job: Job job_1656706245056_0003 running in uber mode : false
2022-07-01 22:12:47,357 INFO mapreduce.job: map 0% reduce 0%
2022-07-01 22:12:51,623 INFO mapreduce.job: map 50% reduce 0%
2022-07-01 22:12:53,660 INFO mapreduce.job: map 100% reduce 0%
2022-07-01 22:12:57,728 INFO mapreduce.job: map 100% reduce 100%
2022-07-01 22:12:58,779 INFO mapreduce.job: Job job_1656706245056_0003 completed successfully
2022-07-01 22:12:58,908 INFO mapreduce.job: Counters: 53
```

File System Counters

```
FILE: Number of bytes read=3288102
FILE: Number of bytes written=7239915
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=18298242
HDFS: Number of bytes written=2194547
HDFS: Number of read operations=11
HDFS: Number of large read operations=0
HDFS: Number of write operations=2
```

Job Counters

```
Launched map tasks=2
Launched reduce tasks=1
Other local map tasks=2
Total time spent by all maps in occupied slots (ms)=7293
Total time spent by all reduces in occupied slots (ms)=3905
Total time spent by all map tasks (ms)=7293
Total time spent by all reduce tasks (ms)=3905
Total vcore-milliseconds taken by all map tasks=7293
Total vcore-milliseconds taken by all reduce tasks=3905
Total megabyte-milliseconds taken by all map tasks=7468032
Total megabyte-milliseconds taken by all reduce tasks=3998720
```

Map-Reduce Framework

```
Map input records=1000000
Map output records=20
Map output bytes=3288006
Map output materialized bytes=3288108
Input split bytes=226
Combine input records=0
Combine output records=0
Reduce input groups=10
Reduce shuffle bytes=3288108
Reduce input records=20
Reduce output records=10
Spilled Records=40
Shuffled Maps =2
Failed Shuffles=0
Merged Map outputs=2
```

```
File Input Format Counters
  Bytes Read=18298016
File Output Format Counters
  Bytes Written=2194547
Filter construction terminated!
```

The second stage output for the mapper phase is similar to the previous one. It is quite different in performances because the operation to be done are less expensive and even the output is lighter.

```
Job Counters
  Launched map tasks=2
  Launched reduce tasks=1
  Other local map tasks=2
  Total time spent by all maps in occupied slots (ms)=9844
  Total time spent by all reduces in occupied slots (ms)=1818
  Total time spent by all map tasks (ms)=9844
  Total time spent by all reduce tasks (ms)=1818
  Total vcore-milliseconds taken by all map tasks=9844
  Total vcore-milliseconds taken by all reduce tasks=1818
  Total megabyte-milliseconds taken by all map tasks=10080256
  Total megabyte-milliseconds taken by all reduce tasks=1861632
```

```
File Input Format Counters
  Bytes Read=18298016
File Output Format Counters
  Bytes Written=703
Test stage terminated!
```


Results

In this part it's discussed the obtained FPR in each bloom filter making use of the same input used during bloom filter set construction. The output of the final reducer is voluntarily produced in this form to have a complete idea of all the information aggregate for each filter.

The output file obtained from the second stage is the following:

output2						
1	1	TP : 1835	FP : 84408	TN: 998165	FN: 0	FPR: 0.07796979972713157
2	10	TP : 11230	FP : 125441	TN: 988770	FN: 0	FPR: 0.11258280523168412
3	2	TP : 5275	FP : 73698	TN: 994725	FN: 0	FPR: 0.06897829792132891
4	3	TP : 14504	FP : 117822	TN: 985496	FN: 0	FPR: 0.10678879525213945
5	4	TP : 36255	FP : 4969	TN: 963745	FN: 0	FPR: 0.005129480940711087
6	5	TP : 85749	FP : 11365	TN: 914251	FN: 0	FPR: 0.01227830979585487
7	6	TP : 182106	FP : 326	TN: 817894	FN: 0	FPR: 3.9842585123805335E-4
8	7	TP : 301121	FP : 1241	TN: 698879	FN: 0	FPR: 0.0017725532765811576
9	8	TP : 278675	FP : 458	TN: 721325	FN: 0	FPR: 6.345397439396606E-4
10	9	TP : 83250	FP : 21577	TN: 916750	FN: 0	FPR: 0.0229951818502505

As expected, no false negative (FN) ratings are found. This means that the set of bloom filters works well and do not recognize records that are present for sure as not present.

In this result is also possible to observe how many true positives have been recognized and so what is the exact number of keys that have been added to each filter.

The following table highlights the information more relevant to interpret the result:

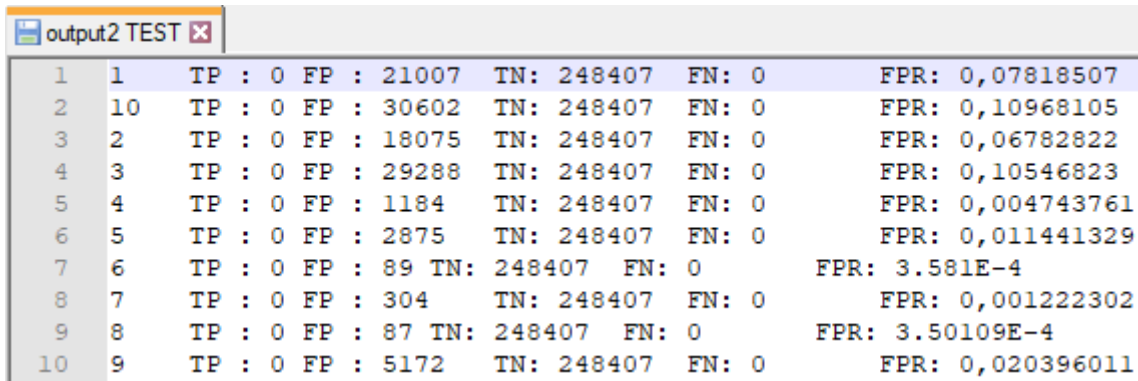
Nr. Filter	FP	TN	FPR	p (designed)	TP	n (designed)	Result
1	84408	998165	0,0779698	0,1	1835	2000	OK
2	73698	994725	0,0689782	0,1	5275	6100	OK
3	117822	985496	0,1067887	0,1	14504	13400	UNDERSTIMATE
4	4969	963745	0,0051294	0,01	36255	41800	OK
5	11365	914251	0,0122783	0,01	85749	82000	UNDERSTIMATE
6	326	817894	0,0003984	0,001	182106	204200	OK
7	1241	698879	0,0017725	0,001	301121	277300	UNDERSTIMATE
8	458	721325	0,0006345	0,001	278675	294000	OK
9	21577	916750	0,0229951	0,01	83250	69100	UNDERSTIMATE
10	125441	988770	0,1125828	0,1	11230	10100	UNDERSTIMATE

Not in all the filters the effective FPR is lower or equal to the value designed at the begin, because in that phase was not possible to know the exact number of true positive in the entire dataset. Theoretically, the dataset size would be very large, so in that phase an estimation of the values to be inserted was done.

In the table, in all the cases where the number of effective TP is bigger than n (number of elements) designed, the false positive value is a bit higher but not completely different. This because a bloom filter guaranties such probability for a given number of keys, of course if this number is bigger, this guarantee is not valid anymore.

In order to execute a different test, the second dataset (25 % about of the original dataset) splitted at the begin is now used as input for the second stage, to see the exact FPR on a dataset not used for the construction of the filters. A little change in the code of the second stage is applied to take in consideration of how computing the values of TP, FN and TN. This part is present commented in the source code of test mapper.

The new output file obtained from the second stage is the following:



Nr. Filter	Nr. Dataset	TP	FP	TN	FN	FPR
1	1	0	21007	248407	0	0,07818507
2	10	0	30602	248407	0	0,10968105
3	2	0	18075	248407	0	0,06782822
4	3	0	29288	248407	0	0,10546823
5	4	0	1184	248407	0	0,004743761
6	5	0	2875	248407	0	0,011441329
7	6	0	89	248407	0	3.581E-4
8	7	0	304	248407	0	0,001222302
9	8	0	87	248407	0	3.50109E-4
10	9	0	5172	248407	0	0,020396011

In this situation the output is quite different. The true negative in all the cases is equal to the total number of records, because none record was inserted before, opposite for true positive equal to 0.

As before the more interesting values are highlighted in the following table:

Nr. Filter	FP	TN	FPR	p (designed)
1	21069	248407	0,0781850	0,1
2	18075	248407	0,0678282	0,1
3	29288	248407	0,1054682	0,1
4	1184	248407	0,0047437	0,01
5	2875	248407	0,0114413	0,01
6	89	248407	0,0003581	0,001
7	304	248407	0,0012223	0,001
8	87	248407	0,0003501	0,001
9	5172	248407	0,0203960	0,01
10	30602	248407	0,1096810	0,1

FPR (TEST1)	FPR (TEST2)
0,0779698	0,0781850
0,0689782	0,0678282
0,1067887	0,1054682
0,0051294	0,0047437
0,0122783	0,0114413
0,0003984	0,0003581
0,0017725	0,0012223
0,0006345	0,0003501
0,0229951	0,0203960
0,1125828	0,1096810

The result obtained make evident that even the number of true negatives is less than in previous test, even the number of records considered as false positive is coherent proportionally to that number.

The FPR obtained in the two tests are quite similar and confirm that in case of filters with underestimated number of keys the false positive rate is higher than designed.

References

The source code is available on the GitHub repository at the link:

<https://github.com/AlterVigna/BloomFiltersInMapReduce>