UNIVERSITY OF PISA

Master's Degree in Artificial Intelligence and Data
Cloud Computing Project

Bloom Filter Construction in Hadoop and Spark frameworks

Candidates:
Fabio BUCHIGNANI
Massimo Valentino CAROTI
Lorenzo MASSAGLI
Simone LANDI

Academic Year 2021/2022

INTRODUCTION

Bloom Filter is a space-efficient probabilistic data structure that is used for membership testing. Given multiple sets of elements, a standard Bloom Filter is not sufficient to look for what set the element belongs to. For this reason we've used a Bloom Filter for each rate. There is the possibility of having false positives. But there can never be false negatives. A false positive occurs when the lookup operation returns a positive result to the presence of the element in a set, while the element is not present.

A bloom filter is a bit-vector with m elements. It uses k hash functions to map n keys to the m elements of the bit-vector. Given a key id_{i}, every hash function h_1,…,h_{k} computes the corresponding output positions, and sets the corresponding bit in that position to 1, if it is equal to 0.

To build a Bloom filter, we need 4 parameters:

- m: number of bits in the bit-vector,
- k: number of hash functions,
- n: number of keys added for membership testing,
- p: false positive rate (probability between 0 and 1).

The relations between these values can be expressed as:

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

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

$$p \approx (1 - e^{-\frac{kn}{m}})^k$$

DATASET

We've built the bloom filter over the ratings of movies listed in the IMDb datasets.
The dataset has about 1 million items and for each item there are movie id, average rating and number of votes. For the construction of the bloom filter only the film id and the average rating are needed for each movie.
The average ratings are rounded to the closest integer value, and we've computed a bloom filter for each rating value.
From the distribution of films based on ratings, we understood that building each filter of the same size would waste a lot of resources. thanks to knowledge of the distribution of the ratings to reduce the space needed, while maintaining the same FP rate.

We create ten bloom filters and for each: given a false positive rate and the number of movies associated to that rating, we calculate the parameters k and m. By doing this, we obtain an equal false positive rate on all bloom filters without wasting resources.

Distribution of the data in ratings

Rating	N
1	2.540
2	6.635
3	17.816
4	43.515
5	102.399
6	219.436
7	370.902
8	353.805
9	113.237
10	16.021
	1.246.306

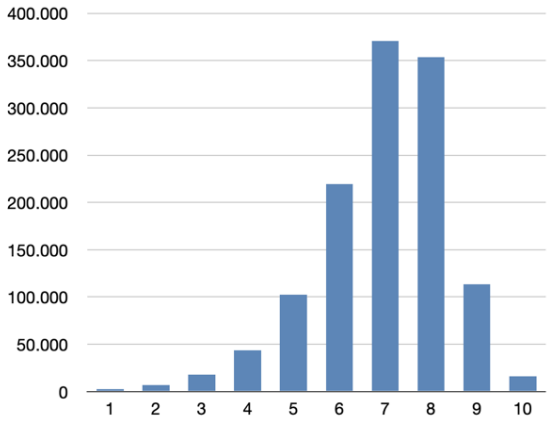

Figure 1: Rating's instances

DESIGN

The Bloom Filter construction has been divided in two Map-Reduce stages:

1. Parameters computation: In this stage has been computed the parameters associated to each Bloom Filter {m_i, k_i}, using the number of instances for each rating as specified in the formula in the introduction chapter.
2. Bloom Filter construction: In this stage has been constructed the Bloom Filters, one for each rating.

Parameters Computation Stage

This stage computes the parameters of each Bloom Filter based on the number of instances associated with the corresponding rating.

1) Mapper
   a) Each mapper defines in the SETUP method an array of ten positions, that represents the counter for each rating, to count the instances.
   b) Once received a film, the mapper increments the corresponding rating's counter by one.
   c) Before terminating, with the CLEANUP method, the mapper will emit the number of instances associated to each rating.

   The pseudocode can be seen in the figure below.

Algorithm 1 ComputeParameters Mapper

```
1:  procedure SETUP
2:      FilmCounter ← new intArray[10]
3:  end procedure
4:  procedure MAP(Film film)
5:      rating ← film.rating
6:      FilmCounter[rating] ← FilmCounter[rating] + 1
7:  end procedure
8:  procedure CLEANUP
9:      for i = 1 to 10 do
10:         EMIT(i, FilmCounter[i])
11:     end for
12: end procedure
```

Figure 2: Compute Parameters Mapper Pseudocode

2) Reducer

   The reducer receives the list of counters associated to a rating, sum all the counters to get the total number of instances and computes the parameters m and k of the Bloom Filter associated to that rating.

   The reducer emits the parameters associated to each rating.

   The pseudocode can be seen in the figure below.

Algorithm 2 ComputeParameters Reducer

```
1: procedure REDUCE(Rating rating, CountersList counters_list)
2:     sum ← 0
3:     for all counter in counters_list do
4:         sum ← sum + counter
5:     end for
6:     m ← −(sum∗ln(p)) / ln(2)²
7:     k ← m/sum ∗ ln(2)
8:     EMIT(rating, (m,k))
9: end procedure
```

Figure 3: Compute Parameters Reducer Pseudocode

Bloom Filter Construction Stage

This stage constructs the Bloom Filters, one for each rating, based on the films and the parameters computed previously.

To add an element inside the Bloom Filters, a set of k hash functions, where k is a parameter, will be defined and used to determine the corresponding indexes of the bloom filter to be set to 1.

The mapper and reducer have been designed as following:

1) Mapper
   a) Each mapper defines in the SETUP method the ten different Bloom Filters, one for each rating.
   b) Once received a film, the mapper computes, for each Hash function and based on the film's ID, the index to be set to 1 of the Bloom Filter corresponding to the film's rating.
   c) Before terminating, with the CLEANUP method, the mapper will emit all the bloom filters, associated with each rating.

   The pseudocode can be seen in the figure below.

Algorithm 3 BloomFilter Construction Mapper

```
1: procedure SETUP
2:     for i = 1 to 10 do
3:         BloomFilters[i] ← newAssociativeArray
4:     end for
5: end procedure
6: procedure MAP(Film film)
7:     bf ← BloomFilters[film.rating]
8:     for all h in [h_1, h_2, ..., h_k] do
9:         index ← h(film.ID)
10:        bf[index] ← 1
11:    end for
12: end procedure
13: procedure CLEANUP
14:    for i = 1 to 10 do
15:        EMIT(i, BloomFilters[i])
16:    end for
17: end procedure
```

Figure 4: Bloom Filter Construction Mapper

2) Reducer
Each Reducer receives all the Bloom Filters associated to a rating and makes the **or** bitwise operation between them.
The reducer emits the final Bloom Filters associated to each rating.
The pseudocode can be seen in the figure below.

Algorithm 4 BloomFilter Construction Reducer

```
1: procedure REDUCE(Rating rating, BloomFilterList bfs_list)
2:     bf_final ← newAssociativeArray
3:     for all bf in bfs_list do
4:         bf_final ← bf_final ∨ bf
5:     end for
6:     EMIT(rating, bf_final)
7: end procedure
```

Figure 5: Bloom Filter Construction Reducer

IMPLEMENTATION

We implemented the designed solution to compute the parameters, construct the bloom filter and test them both in the hadoop framework and in Spark.

Hadoop

The Hadoop implementation takes as inputs, among the others, the name of the file in which the dataset is stored, the desired false positive rate, the number of records in the dataset and the desired number of mappers.

```
hadoop@hadoop-namenode:~$ hadoop jar bloomfilter_map_reduce-1.0-SNAPSHOT.jar
it.unipi.hadoop.bloomfilter.BloomFilter film_ratings.tsv outputDirectory 0.01
 1246306 8
```

In details, the number of records in the dataset (obtainable using the command reported below) were used to compute the exact number of lines for each input split to obtain the desired number of mappers, using the NlineInputFormat option.

```
hadoop@hadoop-namenode:~$ hadoop fs -cat film_ratings.tsv | wc -l
```

According to our design choices, two stages of MapReduce were required to build the BloomFilters: the first stage for computing the parameters, and the second stage for building the actual bloomfilters.

- In the first stage, each map invocation gets in input a line of the dataset (a record containing id, number of votes and average rating of the film), discards the number of votes and the id, rounds the rating to the nearest integer, emitting in principle a pair (rating,1). Using in-mapper combining we were able to reduce the overall amount of data in output from mappers, this was achieved instantiating a single counter for each rating for each mapper. At the end of the map phase, ten pairs (rating, counter) were emitted. The reduce phase consists of summing up all the counters coming from different mappers for the same rating. Once we have the overall value, we are able to compute the number of positions in the bloomfilter and the number of hash functions required, in accordance with the formula provided above.

  These parameters were saved to HDFS, together with the rating associated. To do so we defined a new Java class Parameters that implemented the Writable interface.

- In the second stage, the driver first retrieves the values of the parameters and sets the corresponding resource in the Configuration object. In each mapper the setup phase is needed to instantiate the ten bloomfilters with the parameters provided by the driver. The map function takes as input a record of the dataset and processes it. More specifically it rounds the rating at first, then in accordance with the parameters for that rating, and using the MurmurHash suite of hash functions, computes the results of the hash functions using as input the id of the film. It processes them in order to obtain values in the range from 0 to the size of the bloomfilter for that

rating and use those values to set the relative positions in the relative bloomfilter. In the cleanup phase the Mapper emits the ten bloomfilters, one for each rating.

The reducer takes for each rating the list of bloomfilters and merges them with the OR logical operation. The bloomfilters are then saved on the HDFS together with the relative rating.

Spark

The Spark implementation is very similar. To run it on our cluster we decided to use a python virtual environment to spread the module dependencies to all the workers. We first needed to install the module to create virtual environments.

```
hadoop@hadoop-namenode:~$ sudo apt-get install python3-venv
```

And then create the environment, add the dependencies and zip it into an archive.

```
hadoop@hadoop-namenode:~$ python -m venv pyspark_venv
```

```
hadoop@hadoop-namenode:~$ source pyspark_venv/bin/activate
```

```
hadoop@hadoop-namenode:~$ pip3 install pyspark mmh3 venv-pack
```

```
hadoop@hadoop-namenode:~$ venv-pack -o pyspark_ven.tar.gz
```

Again, the command to launch the application needs to indicate, among the others, the file containing the dataset, the desired false positive rate and the desired number of partitions and the archive file to pass the environment to the application.

```
hadoop@hadoop-namenode:~$ spark-submit --archives pyspark_venv.tar.gz#environ
ment driver.py film_ratings.tsv 0.01 4
```

The master first loads the file as an RDD using the desired number of partitions, moreover it broadcasts the variable containing the false positive rate. This variable is a read-only variable that will be needed to all the workers to perform computation later on, so we need to make it available to them. About the storage level, we decided to keep the default level, MEMORY_ONLY.

In a very similar manner to what we have done with Hadoop we first compute the parameters. All the computations are made through transformations from an RDD to another, and at the end we save the parameters in a text file on HDFS.

Differently from Hadoop we don't need to get them again from file to start the bloomfilter construction, but we can directly use the relative RDD: at the driver, we used the CollectAsMap action to get them as Python tuples and broadcasted them to be available to all the workers.

The bloom filter construction stage required some more modifications: in Hadoop implementation, adapting the algorithm to the MapReduce framework, we decided to use the In-mapper combining pattern.
The Spark framework is different, so, starting from the initial RDD containing the original dataset:

- We first got the rounded ratings for each film.
- We computed in accordance with the parameters the indexes in the relative bloomfilter. To do so we used the module mmh3, that provides an implementation of Hash functions equivalent to the ones provided with Hadoop MurmurHash.
- At this point we had an RDD containing for each film the rating of the film and the set of indexes. We used the ReduceByKey transformation to merge together all the indexes relative to the same rating and to obtain an RDD with 10 lists of indexes, one for each rating.
- We built the bloomfilter for each rating: creating the array using the parameters and iterating over the relative list to set the bits was sufficient to obtain the final bloomfilter.
- The ten bloomfilters were again saved to HDFS.

EXPERIMENTAL RESULTS

The experimental results have been analyzed using a test MAP-REDUCE algorithm, which, given the same dataset in input, checks the false positive rate of each Bloom Filter and compares them with the p given in input for the Bloom Filters construction.
In the test algorithm, for each film we test the presence of it, computing the same hash functions used in the Bloom Filters construction, in all the Bloom Filters that has a different rating from the film's rating.
If all the hash function returns an index of the bloom filters where it is true, then the false positive counter increases by one.
The false positive rate, for each rating, can be computed as following:

$$False\ Positive\ Rate = \frac{false\ positives}{false\ positives + true\ negatives}$$

The results that we obtained can be seen in the following tables.

Rating	P = 0.01	P = 0.05	P = 0.1	P = 0.2
1	0.01047	0.05008	0.01042	0.20288
2	0.01040	0.05011	0.09935	0.20516
3	0.01010	0.05028	0.10076	0.20336
4	0.01017	0.05023	0.10131	0.20209
5	0.01022	0.05033	0.10076	0.20277
6	0.01024	0.05026	0.10093	0.20123
7	0.01010	0.05020	0.10057	0.20289
8	0.01011	0.05003	0.10030	0.20182
9	0.01001	0.05025	0.10014	0.20243
10	0.01047	0.05083	0.09953	0.20248

Table 1: Hadoop false positive rates

Rating	P = 0.01	P = 0.05	P = 0.1	P = 0.2
1	0.01023	0.05114	0.09961	0.20034
2	0.01012	0.05109	0.10327	0.19851
3	0.00998	0.05030	0.10168	0.20228
4	0.01015	0.04963	0.09996	0.20139
5	0.01018	0.05024	0.10073	0.20181
6	0.01005	0.05080	0.10053	0.20193
7	0.01020	0.05093	0.10105	0.20227
8	0.01009	0.05066	0.10076	0.20263
9	0.01015	0.05045	0.10026	0.20166
10	0.01030	0.05007	0.10000	0.20188

Table 2: Spark false positive rates

As we can see from the tables, the Bloom Filters have been constructed correctly considering the false positive rate in input.

We have also performed some experimental analysis on the application time respect to the used number of mappers, the results can be seen in the following figures.

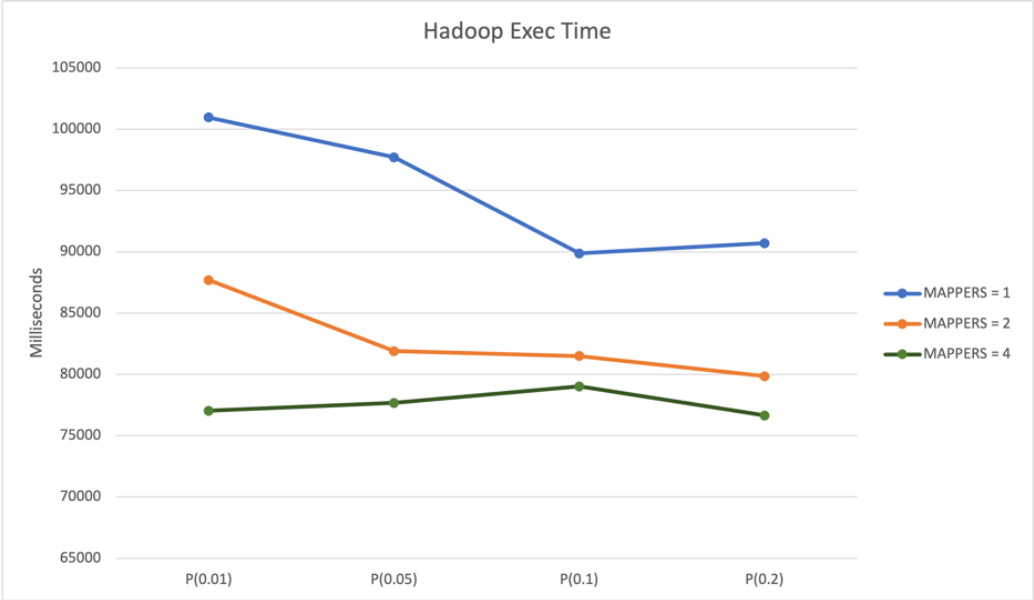

Figure 6: Hadoop Execution Time

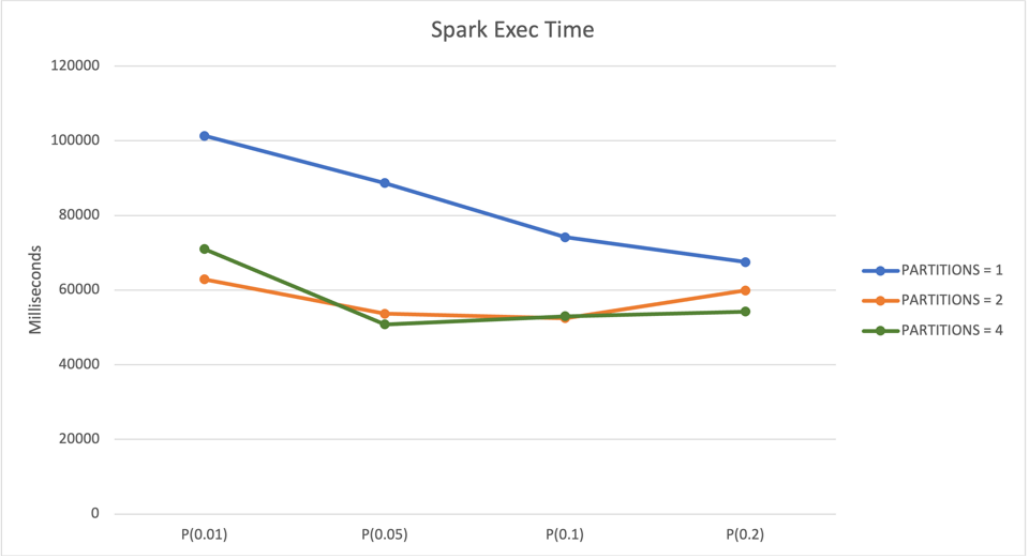

Figure 7: Spark Execution Time

As we can see from the figure 7, when the application uses 4 partitions it takes sometimes more time then 2 partitions and sometimes less.
We noticed that Spark uses always only 2 workers by default, and this is why we obtained those results.
We then tried to force Spark to use 4 executors (all the executors that we had available) and, as we can see in the figure 8, we obtained that the application takes less execution time parallelizing more the computation.

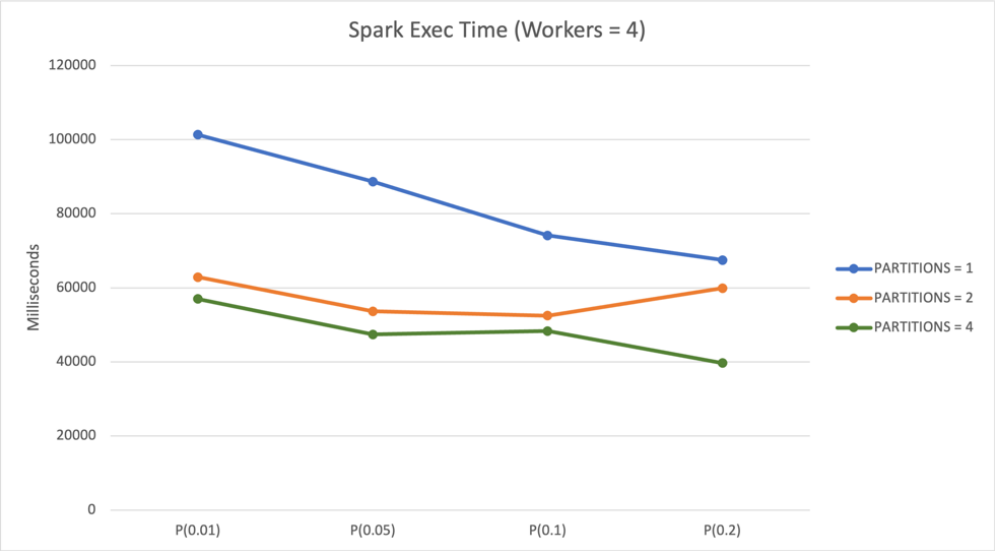

Figure 8: Spark Execution Time (Workers = 4)

CONCLUSION

In conclusion, we obtained that the bloom filter construction is correctly performed and increasing the number of mappers, the distributed computing application takes less time as we expected from the parallel computing paradigm.