University of Pisa

Master's Degree in

Artificial Intelligence and Data Engineering

Multi-Set Bloom Filter in Parallel Cloud Computing

Candidates

Cecchetti Jacopo

Del Seppia Matteo

Minniti Federico

Terranova Franco

ACADEMIC YEAR 2021/2022

INDICE

INTRODUCTION

Bloom Filter is a space-efficient probabilistic data structure that allows to check the membership of an element in a set. Given multiple sets of elements, a standard Bloom Filter is not sufficient to look for what set the element belongs to.

In this project, we proposed a solution of the multiple set matching problem, defining a multi-set Bloom Filter.

By definition, a Bloom Filter is a vector of m bit elements, able to answer whether a value belongs to a particular set with a certain probability of false positives.

It uses k hash functions to map a number n of keys to the m elements of the vector.

Given a key id, every hash function $h1$,…, hk will provide a corresponding output position, and the Bloom Filter will set the corresponding bit of that position to 1, if not set already.

The new data structure presented in this project, has associated multiple sets of data and support the construction operation of all the sets through an efficient construction operation.

An implementation based on the MapReduce paradigm is presented, specifically employing the Hadoop and Spark frameworks.

1.1. FALSE POSITIVE RATE

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.

It has been shown that the probability p of false positives in a Bloom Filter of size *m*, with *n* different elements to store, each encoded with *k* hash functions is:

$$p = (1 - e^{-k\frac{n}{m}})^k$$

If the number of elements and the FPR are known, we can derive the value of m and k, as following.

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

$$k = \ln 2\frac{m}{n} = -log_2 p$$

The Figure 1 shows the variation of the FP Rate with respect to m and k. The plot shows that in front of the increase of the value of m, the FP Rate decreases, but then increases again as k becomes larger. This trend is expected, since with the increase of the number of hash functions we have an increase of the number of bits set to 1, then leading to more collisions.

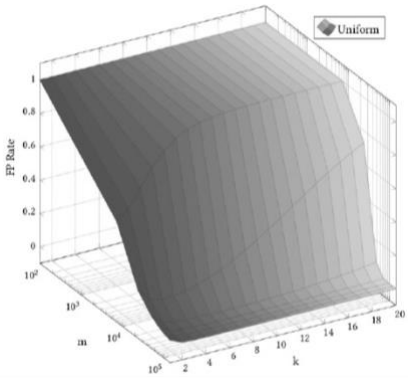


Figure 1. Variation of the FP Rate w.r.t. m and k.

1.2. DATASET

We've built our multi-set Bloom Filter over the ratings of movies listed in the IMDb datasets, with a set for each different rating. The dataset consists of approximately 1M items with associated a rating from 1 to 10.

The rating of each movie is rounded to the closest integer value. After this operation, the resulting distribution of ratings was the one highlighted in Figure 2.

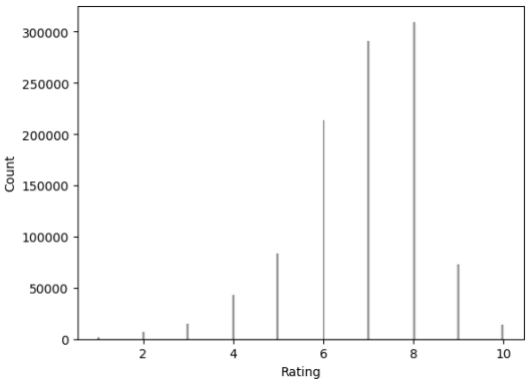


Figure 2. Distribution of the data in rounded ratings

Considering our problem, the number of elements associated with a rating (n_i), can vary and thus, considering all Bloom Filters with the same size, equal to the maximum of these values, can result in a waste of space.

We can use the knowledge of the distribution of the ratings to reduce the space needed, while maintaining the same FP rate.

We can consider a set of Bloom Filters with a number of Bloom Filters equal to the number of possible ratings, and, given an expected FP rate p_i and the number of elements associated to that rating n_i, we can calculate the dimension of each Bloom Filter m_i, using the previous mentioned equations.

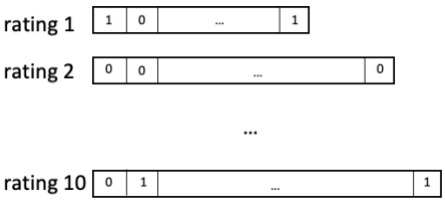


Figure 3. Multi-set Bloom Filter

1.3. MURMURHASH

A set of hash functions should be used to determine the corresponding bits to be set to 1 inside each Bloom Filter. We considered a 32-bit MurmurHash of range [0, 2^32], non-cryptographic hash function suitable for general hash-based lookup, so that the bits to be altered will be the following:

$$MurmurHash(FilmID) \,\%\, mi \quad i = 1, \dots, k$$

1.4. FALSE POSITIVE RATE IN A MULTISET BLOOM FILTER

Considered the overall set of *l* different Bloom Filters, the overall False positive rate will correspond to the average FP Rate, and it can be computed as following.

$$p = \frac{\sum_{i=1}^{l}(1 - e^{-ki\frac{ni}{mi}})^{ki}}{l}$$

where k_i, m_i, and n_i are the number of hash functions, bits and elements of the i-th Bloom Filter, respectively.

2. HADOOP DESIGN

The Bloom Filter construction can be decomposed in two stages, suitable for a MapReduce implementation:

1. *Parameter Calculation* Stage, where the parameters of each Bloom Filter $\{m_i, k_i\}$ are computed, based on the number of elements with a given rating in the dataset.
2. *Construction* Stage, where the Bloom Filters are filled based on the results of the hash functions.

2.1. PARAMETER CALCULATION STAGE

The first stage calculates the parameter of each Bloom Filter based on the number of movies associated with the corresponding rating.

1. Each mapper owns a counter for each rating, represented as an associative array. Once received a movie, the mapper will increment the corresponding rating counter. Before terminating, the mapper will emit the number of films received associated to each rating. In-mapper combining is used to reduce the number of intermediate data emitted.

Algorithm 1 method MAP(Filename, Films)

```
H ← new AssociativeArray
for all film f ∈ Films do
    rating ← round(f.rating)
    H(rating) ← H(rating) + 1
end for
for all rating r ∈ H do
    EMIT(r,H(r))
end for
```

2. The reducer receives the counters associated to a rating emitted by the mappers, sum the counters' values and calculate the parameters of the Bloom Filter associated to that rating.

Algorithm 1 method REDUCE(Rating, counts[c1,c2,...])

```
n ← 0
for all count c ∈ counts[c1, c2, ..] do
    n ← n + c
end for
m ← −n ∗ ln(p)/ln(2)^2
k ← ln(2) ∗ m/n
EMIT(Rating, {m, k})
```

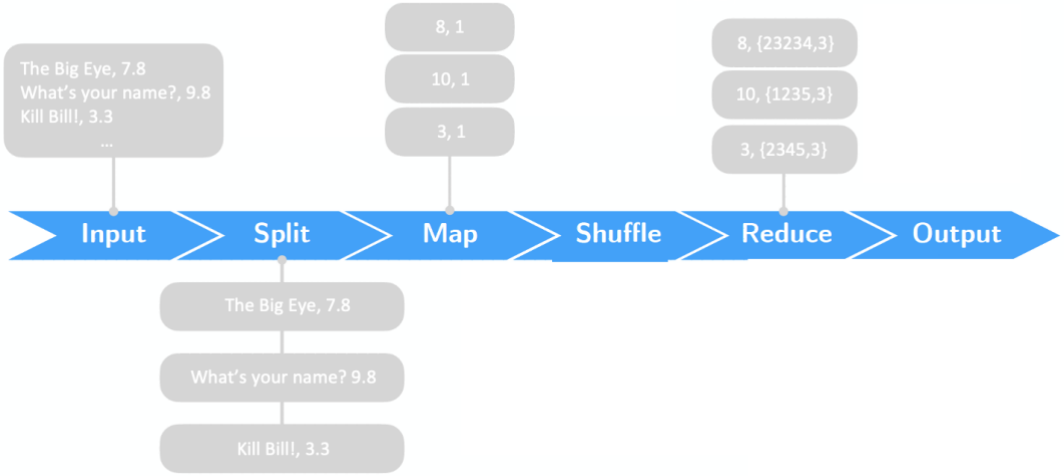


Figure 4. MapReduce Workflow Parameters Calculation Stage

2.2. CONSTRUCTION STAGE

In the construction stage, the Bloom Filters are created.

In order to add an element inside a Bloom Filter, a set of k hash functions will be used to determine the corresponding bits to be set to 1.

We proposed two different solutions to perform the construction stage, then compared to determine which one could be the best solution for this purpose.

In the *first solution*, the following workflow has been conducted.

1. Each mapper calculates the k different hash functions of the films' IDs, bounding the results to the maximum size of the related Bloom Filter, and emit the rating, together with the hash array.

Algorithm 1 method MAP(Filename, Films)

```
1: for all film f ∈ Films do
2:     rating ← round(f.rating)
3:     hash ← new HashSet
4:     for i ∈ 1,...k[rating] do
5:         hash[i] ← abs(MurmurHash(f.ID) % m[rating])
6:     end for
7:     EMIT(rating, hash)
8: end for
```

2. Each reducer receives in input the hash values associated to a rating, populates the Bloom Filter associated to the rating, and writes it to the HDFS.

Algorithm 1 method REDUCE(Rating, hashes[h1,h2,...])

```
1: BloomFilter ← new BloomFilter
2: for all hash h ∈ hashes[h1, h2, ..] do
3:     BloomFilter.setBit(h)
4: end for
5: EMIT(Rating, BloomFilter)
```

Deciding to use VIntWritable instead of IntWritable as wrapper class in this solution, a reduction, on average, on the number of map output bytes has been highlighted.

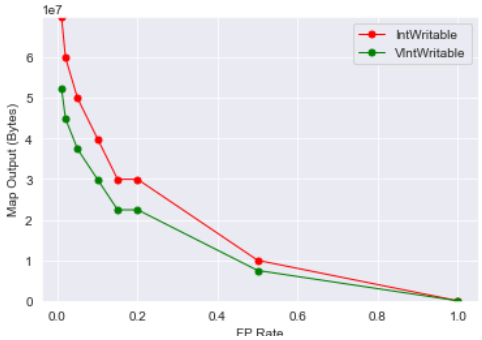


Figure 5. Variation of the Map Output Bytes considering IntWritable and VIntWritable.

We've tried to optimize the amount of traffic sent, by using an in-mapper combiner and emitting only once each repeated hash result. Due to the negligible number of collisions, we though decided to not follow this solution, since the overhead needed to maintain the associative array in memory was much higher than the benefits we could get from this improvement.

In the *second solution*, the subsequent steps have been followed.

1. Each mapper creates a Bloom Filter for each rating and populates each of them until new films arrive. Before terminating, the mapper emits all the Bloom Filters populated. An in-mapper combiner has been then used in this solution. We've defined a class *BloomFilter* to store all the information needed for a Bloom Filter. The class implements the *Writable* Interface.

Algorithm 1 method MAP(Filename, Films)

```
1: BloomFilterArray ← new BloomFilterArray
2: for all film f ∈ Films do
3:     rating ← round(f.rating)
4:     hash ← new HashSet
5:     for i ∈ 1,...k[rating] do
6:         hash[i] ← abs(MurmurHash(f.ID) % m[rating])
7:     end for
8:     BloomFilterArray[rating].setBit(hash)
9: end for
10: for all rating t ∈ {1,...,10} do
11:     EMIT(rating, BloomFilterArray[rating])
12: end for
```

2. Each reducer receives all the Bloom Filters associated to a rating, join the Bloom Filters (performing an or operation bitwise), and writes the final Bloom Filter to the HDFS.

Algorithm 1 method REDUCE(Rating, BloomFilterArray[B1,B2,...])

```
1: BloomFilter ← new BloomFilter
2: for all BloomFilter B ∈ BloomFilterArray[B1, B2, ..] do
3:     BloomFilter.or(B)
4: end for
5: EMIT(Rating, BloomFilter)
```

2.3. SOLUTIONS COMPARISON

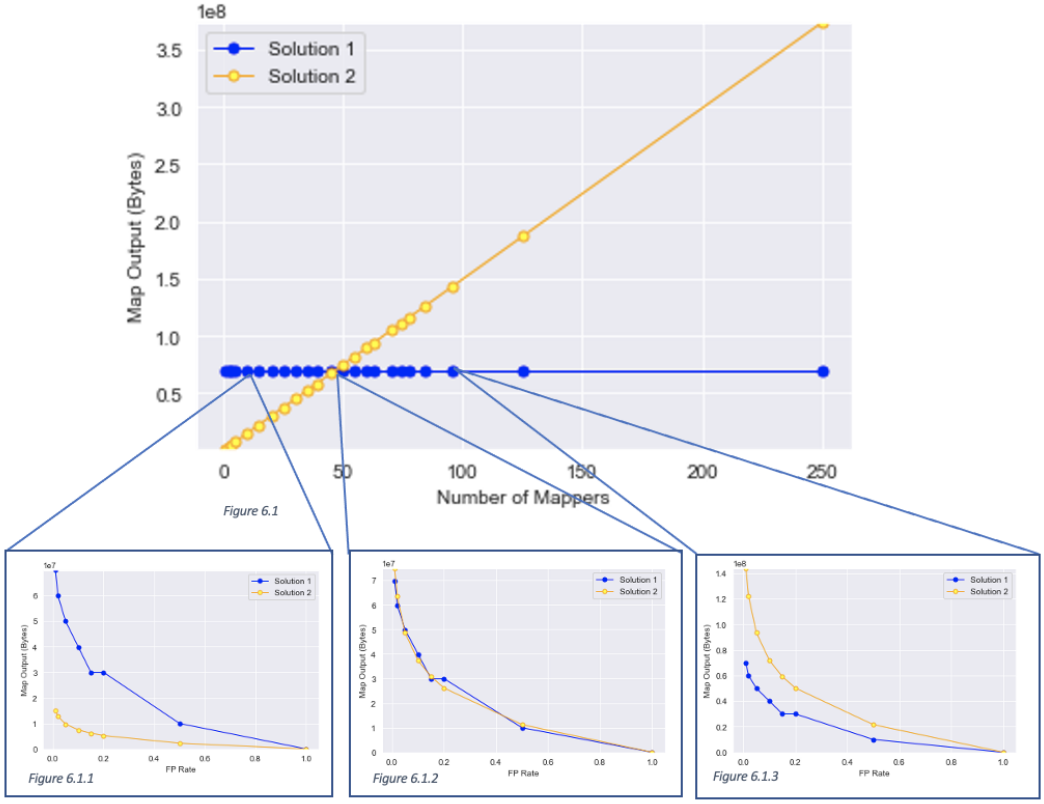


Figure 6. Map Output Bytes Comparison

We've started comparing the *number of bytes emitted in output* in the *map phase* for each solution varying the number of lines for each split, and thus the number of mappers.

In Figure 6.1 we can notice that with the increase of the number of mappers, obtained reducing the number of lines of input, the map output bytes of the first solution remain constant, since regardless of the number of mappers, always k hash values will be emitted for each film.

In the second solution we can notice a linear increase of the number of output bytes, due to the fact that for each mapper a number of Bloom Filters equal to the number of classes will be emitted.
In case we have less than 50 mappers, the second solution seems to perform better; while in case more mappers are present, the first solution is preferable, since it maintains a constant and lower number of output bytes.
Focusing on the increasing of the FP Rate, we expect for both solutions to have a decreasing number of output bytes. In fact, for the first solution a lower amount of hash functions will be calculated, while for the second solution, smaller Bloom Filters will be emitted.
Focusing on the intersection of the two lines in Figure 6.1, with a number of Mappers equal to 50 (Figure 6.1.2) the situation is approximately the same for the two solutions, as expected.
For a smaller number of mappers (Figure 6.1.1), e.g.10 mappers, we can notice that the map output bytes of the first solution are significantly more than the second one.
Moving towards a point with a number of mappers greater than 50 (Figure 6.1.3), e.g. 96, the situation is overturned since we emit a huge number of Bloom Filters, most of which will also be sparse. The situation will then become worse, and the gap becomes larger with the decrease of the FPR, since bigger Bloom Filters will be emitted.

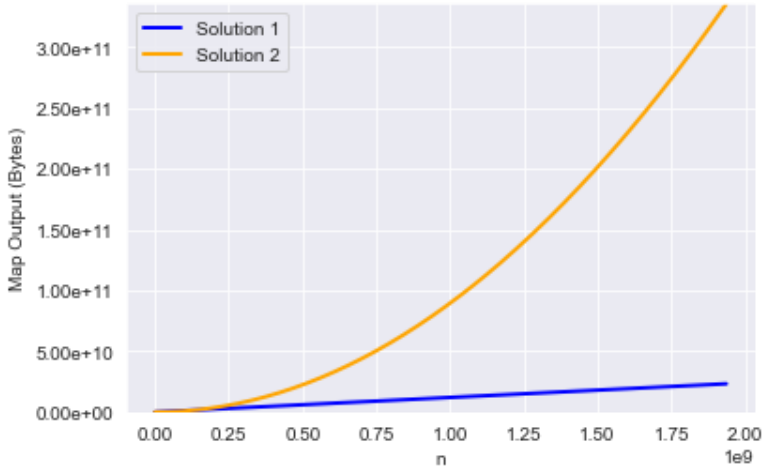


Figure 7. Map Output Bytes Comparison

Figure 7 demonstrates the increase of the number of map output bytes with the increase of the dimension of the dataset, fixed the number of line splits.
With the increase of the dimension of the dataset used to construct the Bloom Filter we can notice a linear increase of the number of output bytes of the first solution w.r.t. a quadratic increase of the number of output bytes of the second solution.

This consideration can also be confirmed by the expressions of the map output bytes in the two solutions as function of n.

$$map\ output\ bytes_1 = k * n * v = -log_2 p * n * v = \alpha * n$$

where v is the number of bytes of the wrapper VIntWritable.

$$map\ output\ bytes_2 = numMapper * 10 * \left(\frac{1}{8} * \sum_i^{10} m_i + additionalData\right)$$

$$= \frac{n}{NLineSplit} * 10 * \left(\frac{1}{8} * \sum_i^{10} -n_i \frac{\ln p}{\ln 2^2} + additionalData\right)$$

$$= -\frac{n^2}{NLineSplit} * \frac{100}{8} * \frac{\ln p}{\ln 2^2} + \frac{n}{NLineSplit} * 10 * additionalData$$

$$= \beta * n^2 + \gamma * n$$

where $additionalData$ represents the additional fields present in the BloomFilter class (in our case 2*4 bytes, since we store 2 integers, m and k, per Bloom Filter).

Figure 8 demonstrates the decreasing of the *average construction time* with the increase of the FP Rate for both solutions.

Considering the CPU time, the second solution achieves better performance than the first one, with a bigger gap for small values of the FP Rate.

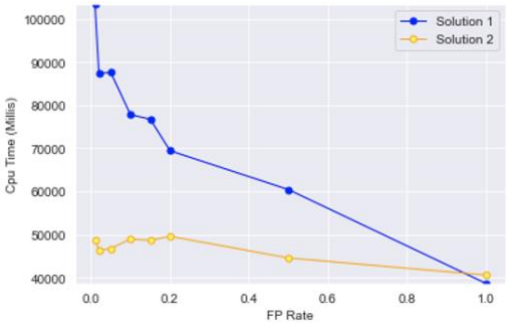


Figure 8. Average Construction Time Comparison

Comparing the two solutions in terms of *memory overhead*, the second solution maintain as class fields the different bloom filters and their parameters.

The first solution does not have this additional burden.

For the second solution, we break the functional paradigm and we may also have potential ordering-dependent bugs. Moreover, we are not tolerant to faults, if we have an error (e.g. the JVM dies), we may lose all the previous computations.

Furthermore, we may encounter memory scalability problems since we store all Bloom Filters in memory.

After comparing the two solutions, considering the metrics previously highlighted, we decided to continue with the first solution, mainly because of the higher amount of the map output bytes emitted and the memory overhead.

When dealing with big data, the gap between the solutions' traffic may become very large, as previously demonstrated.

This situation is real also in our domain of application, considering that the IMDb dataset is updated daily, and for this reason its size tends to increase.

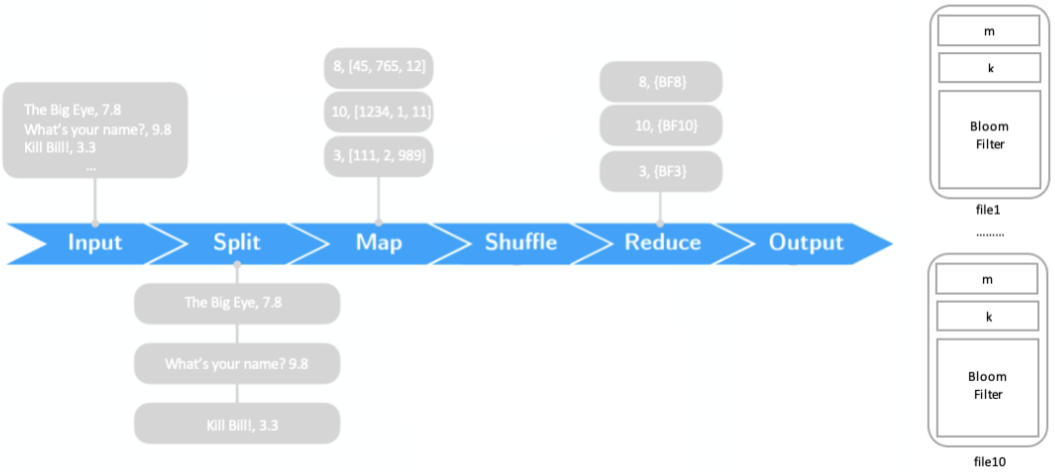


Figure 9. MapReduce Workflow Construction Stage

The solution chosen in the construction stage will store the information related to each Bloom Filter in a different file in the HDFS.

2.4. MAP REDUCE COMPLEXITY

Assuming that the intermediate keys skew is not a problem, we've calculated the following parameters for evaluating the complexity of our MapReduce solution.

$$reducer\ size\ q = k * \frac{n}{10}$$

$$replication\ rate\ r = k$$

Considering the equations of the reducer size and the replication rate, we've been able to determine the communication-computation trade-off of our solution.

$$\frac{q}{r} = \frac{n}{10}$$

2.5. SPARK DESIGN

Starting from the solutions highlighted in the Hadoop Design, the following lineage graphs will show the computation workflow of the Spark Framework

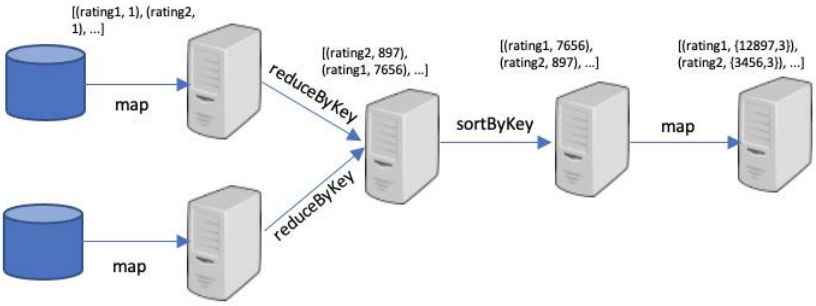


Figure 10. Spark Workflow Parameters Calculation Stage

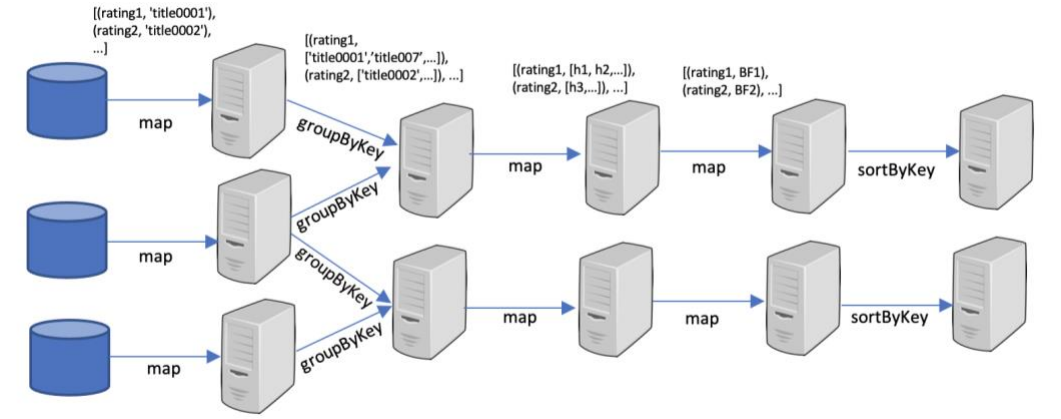
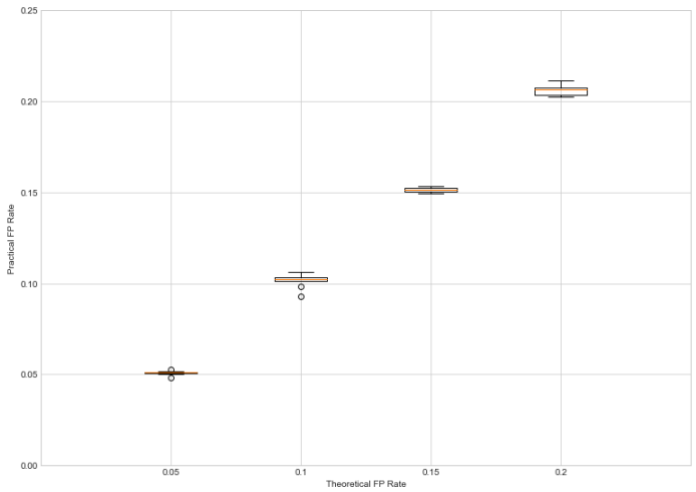


Figure 11. Spark Workflow Construction Stage

2.6. FALSE POSITIVE RATE EVALUATION

Once terminated the construction phase of our Bloom Filters we've verified the FP Rate considering the number of elements which has a false positive result for each Bloom Filter and comparing it with the value given by the theoretical formula.



FP Rate	Standard Deviation
0.05	0.0011
0.10	0.0034
0.15	0.0013
0.20	0.0027

Figure 12. False Positive Rate Verification

Results show that the practical FP Rate is close to theoretical one, with a negligible standard deviation.

2.7. COMPLEXITY

Our solution has a space complexity of $O(m * l)$ where m is the maximum size among all Bloom Filters and l is the total number of possible ratings.

Moreover, the operation of insertion has a complexity of $O(k * n)$ where k is the maximum number of hash functions among all Bloom Filters and n is the total number of elements.

CONCLUSIONS

This project presents a statistical efficient data structure which allows to answer whether an element belongs to a multi-set of elements. The proposed structure is space-efficient and thus can be cached in RAM to provide faster access compared to disk-stored structures.