Engineering Department

Master's Degree Artificial Intelligence and Data Engineering

Università degli studi di Pisa
Accademic year - 2021-2022

Bloom Filter implementation using Map-Reduce paradigm in Hadoop and Spark

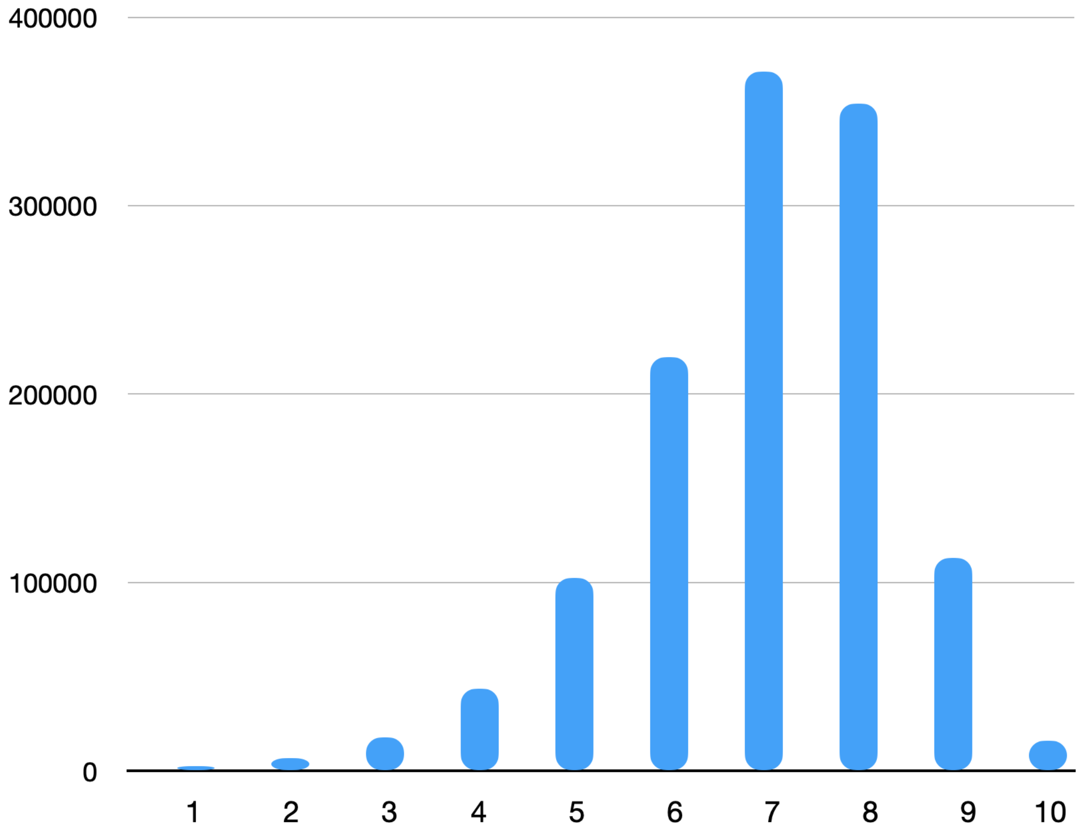
STUDENTS:
Marco Bellia, Marco Ralli, Stefano Bianchettin, Giulio Fischietti

Cloud Computing

Contents

1 Introduction

In this project we want to investigate in deep on how a Bloom Filter can be implemented in a parallelized way using the map-reduce approach whit Hadoop and Spark.
Moreover we want to test our implementation to prove the literature formulas we have about bloom filters, finally we'll make a performance comparison between Hadoop and Spark. We have to say that now-days, Spark offers much more flexibility than Hadoop and it is 10 times faster to code on it.
To play with Bloom Filters we used an IMDb datasets with more than 1.5M ratings of movies, starting from this dataset we wanted to create 10 Bloom Filters each one representing the movies with a certain rating.
Once realized the algorithm to built such bloom filters, we compute the false positive rating for each bloom filter to extract the final average rate.
The ratings dataset was not balanced, as you can see in the distribution above. For such reason we will see later how the dataset composition influence our results for the false positive rate.



We concluded our paper with performance analysis and a simple correlation study between the p value and the false positive rate.

2 Bloom Filters

Bloom filter is a space-efficient probabilistic data structure used to test whether an element belongs to a set. False positives matches are possible, but false negatives are not. So returning either "possibly in set" or "definitely not in set".
The data structure is a bit-vector with *m* elements. It uses *k* 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.
If we want to test if an element belongs to a set, we compute the hash value of that element and check if any of the element pointed is 0: if this is true, then that element surely does not belong to that set. Otherwise, if we have all elements pointed to 1, then, that element probably belongs to that set.

2.1 Parameters

We consider a multiset Bloom filter, hence *l* different bloom filters each one map an unique rounding rating value: 1 to 10. So *l* is equal to 10.
The *i*-th Bloom filter typically has the following parameters:

- m_i: number of bits in the bit-vector
- k_i: number of hash functions
- n_i: number of keys added for membership testing
- *p*: false positive rate (probability between 0 and 1)

Are defined as

$$m_i = -\frac{n_i \ln p}{(\ln 2)^2}$$

$$k_i = \frac{m_i}{n_i} \ln 2$$

$$p \approx (1 - e^{-\frac{kn_i}{m_i}})^{k_i}$$

n_i is determined as the number of inserted elements, so the total number of movies in the dataset with *i*-th rating of the *i*-th Bloom filter.
Fixed *p* can calculate respectively m_i and k_i. The Bloom filters and its params depend on *p* assumed.

2.1.1 Murmurhash

The set of hash functions, used to determine the corresponding bits to be set to 1 inside each Bloom filter, consider are a 32-bit MurmurHash non-cryptographic hash function suitable for general hash-based lookup.
The bits to be flag will be the following:

MurmurHash(MovieID) mod m_i, *i=1...l*

2.2 False Positive Rate

The average FP rate can be computed as following

$$AverageFPR = \frac{\sum_{i=1}^{l} FPR_i}{l} = \frac{\sum_{i=1}^{l} \frac{FP_i}{FP_i+TP_i}}{l}$$

The objective of the project is to minimize it in order to optimize the Bloom filters. In fact, we can control the probability of getting a false positive by controlling the size of the Bloom filter. More space means fewer false positives. If we want to decrease probability of false positive result, we have to use more number of hash functions and larger bit array. This would add latency in addition to the item and checking membership. So regulate *p* we optimize them.

2.3 Structure algorithms

The Bloom filter support the add and check:

Algorithm 1: Add item

```
Data: $movieID \neq NULL$
for $hashFunction \in hashFunctions$ do
    $index \leftarrow hashFunction(movieID)$ ;
    $bitSet[index] \leftarrow true$
end
```

Algorithm 2: Check if item belongs to set

```
Data: $movieID \neq NULL$
for $hashFunction \in hashFunctions$ do
    $index \leftarrow hashFunction(movieID)$ ;
    if $bitSet[index]$ is false then
        return false
    end
end
return true
```

The sequential algorithm to estimate the average false positive rate of all bloom filters:

Algorithm 3: Compute Average False positive rate

```
Data: $Dataset \neq NULL$, $BloomFilters \neq NULL$,$i \leftarrow 0$ $FPR_{tot} \leftarrow 0$ ;
for $bf \in BloomFilters$ do
    $FP_i \leftarrow 0$ ;
    $TP_i \leftarrow 0$ ;
    for $movies \in Dataset$ do
        $movieID \leftarrow movies[0]$ ;
        $rating \leftarrow movies[1]$ ;
        if $bf.idx <> rating$ then
            if $bf.check(movieID)$ is true then
                $FP_i \leftarrow FP_i + 1$ ;
            else
                $TP_i \leftarrow TP_i + 1$ ;
            end
    end
    $FPR_{tot} \leftarrow FPR_{tot} + \frac{FP_i}{FP_i+TP_i}$ ;
    $i \leftarrow i + 1$ ;
end
return $\frac{FPR_{tot}}{|BloomFilters|}$
```

3 MapReduce

The algorithms presented so far are sequential, they can be made more efficient and faster by parallelizing them using MapReduce paradigm.
The solution we came up with is made up of three phases, which corresponds one job for each phase.

3.1 Stage 1: Compute Parameters

In this stage we compute the parameters for the creation of the bloom filters: below are shown the algorithms in map-reduce used.

Algorithm 4: Method MAP(Movies)

```
for all movies $m \in Movies$ do
    $rating \leftarrow round(m.rating)$ ;
    EMIT($rating$,1);
end
```

Algorithm 5: Method COMBINE(Rating r,counts[$c_1, c_2, ..$])

```
$n \leftarrow 0$ ;
for all count $c \in counts[c_1, c_2, ...]$ do
    $n \leftarrow n + 1$ ;
end
EMIT($rating, n$);
```

Algorithm 6: Method REDUCE(Rating r,counts[$c_1, c_2, ..$])

```
$n \leftarrow 0$ ;
for all count $c \in counts[c_1, c_2, ...]$ do
    $n \leftarrow n + c$ ;
end
$m \leftarrow -\frac{n \ln p}{(\ln 2)^2}$ ;
$k \leftarrow \frac{m}{n} \ln 2$ ;
EMIT($rating, \{n, m, k\}$);
```

3.2 Stage 2: Creation of Bloom Filters

Creation of Bloom Filters and insertion of the data.

We decide to use a pattern design, the In-Mapper combiner:

Algorithm 7: Class Mapper

```
class MAPPER;
    method INITIALIZE;
        $BloomFilter \leftarrow$ new AssociativeArray;
        $Combiner \leftarrow$ new AssociativeArray;
        for all params $p \in BloomFilterParameters$ do
            $BloomFilter\{p \rightarrow n\} \leftarrow BF(p \rightarrow m, p \rightarrow k)$ ;
        end
    method MAP(Movies);
        for all movies $m \in Movies$ do
            $rating \leftarrow round(m \rightarrow rating)$ ;
            $movieID \leftarrow (m \rightarrow MovieID)$ ;
            if $movieID \in Combiner\{rating\}$ then
                $Combiner\{rating\}.add(movieID)$ ;
                $Combiner\{rating\} \leftarrow Combiner\{rating\}$ ;
            else
                $BloomFilter\{rating\}.add(movieID)$ ;
                $Combiner\{rating\} \leftarrow BloomFilter\{rating\}$ ;
            end
        end
    method CLEANUP();
        for all AssociativeArray $temp \in Combiner$ do
            EMIT($temp \rightarrow rating$,$temp\{rating\}$);
        end
```

Algorithm 8: Method REDUCE(Rating r,bloomfilters[$bf_1, bf_2, ..$])

```
$tempBloomFilter \leftarrow$ new BloomFilter ;
for all bloomfilters $bf \in bloomfilters[bf_1, bf_2, ...]$ do
    $tempBloomFilter \leftarrow tempBloomFilter$ OR $bf$ ;
end
EMIT($rating, tempBloomFilter$);
```

The combiner is the same of the reducer.

3.3 Stage 3: Compute average false positive rate

We decide to use a pattern design, the In-Mapper combiner:

Algorithm 9: Class Mapper

```
class MAPPER;
  $FP \leftarrow$ new Array ;
  $TP \leftarrow$ new Array ;
    method MAP(MOVIES $m$,BLOOMFILTERS $bloomFilter[bf_1, bf_2, ..., bf_{10}]$);
      for all movies $m \in Movies$ do
        $rating \leftarrow round(m \rightarrow rating)$ ;
        $movieID \leftarrow (m \rightarrow MovieID)$ ;
        for all bloomfilters $bf_i \in bloomFilter$ do
          if $bf_i \rightarrow rating <> rating$ then
            if $movieID \in$ checkItemBF($bf_i$) then
              $FP_i \leftarrow FP_i + 1$
            else
              $TP_i \leftarrow TP_i + 1$
            end
        end
      end
    method CLEANUP();
      $i \leftarrow 0$ ;
      for $falsepositive \in FP$ do
        EMIT($i + 1, \frac{falsepositive}{falsepositive+TP_i}$);
        $i \leftarrow i + 1$ ;
      end
```

Algorithm 10: Method REDUCE(Rating r,falsepositivesrate[$fpr_1, fpr_2, ..$])

```
$FPRtot \leftarrow 0$ ;
$l \leftarrow 0$ ;
for $FPR \in falsepositivesrate[fpr_1, fpr_2, ...]$ do
  $FPRtot \leftarrow FPRtot + FPR$ ;
  $l \leftarrow l + 1$ ;
end
EMIT($rating, \frac{FPRtot}{l}$);
```

4 Implementation on Hadoop

4.1 Configuration

IP	name node	data node	hostname
172.16.4.175	yes	yes	hadoop-namenode
172.16.4.148	no	yes	hadoop-datanode-2
172.16.4.183	no	yes	hadoop-datanode-3
172.16.4.181	no	yes	hadoop-datanode-4

Hadoop was configured according to the table above following a guide proposed by Professor.
Tonellotto of the University of Pisa.
https://github.com/tonellotto/cloud-%computing/blob/master/notes/hadoop3-installation.md

4.2 Stages

The multiset Bloom filters implementation using Hadoop was divided into 3 MapReduce Jobs:

4.2.1 Job 1: Compute Parameters

- **RatingMapper.java:** receives the dataset splitted in N batches by NLineInputFormat; it parses each line of the batch extracting the rounded rating used as key with value one to emit the output.
- **CreateParameterReducer.java:** sums the values outputted by the mapper to obtain the final count of how many movies of such rating are found *n*; calculates the *m* and *k* parameters.

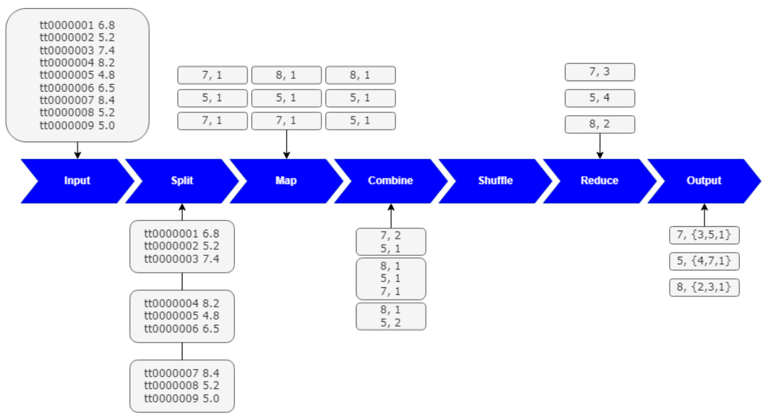


Figure 1: Stage Job 1

4.2.2 Job 2: Creation of Bloom Filters

- **BloomFiltersMapper.java:** Implement the pattern design In-Mapper combiner.
  The setup method reads the hdsf file where it's memorize the output of the previous stage to initialize the hash maps: each rating is associated with an empty class bloom filter relative to its parameters.
  The map method receives the dataset splitted in N batches by NLineInputFormat and it parses each line of the batch extracting the rounded rating and its movieID. It inserts the movieID into the bloom filter

associate to its rating.
The cleanup method emits the key rating and its associate Bloom filter as a value, for each rating.

- **BloomFIlterReducer.java:** merges the Bloom filters associate to the same rating using the OR logic operator and emits the result.

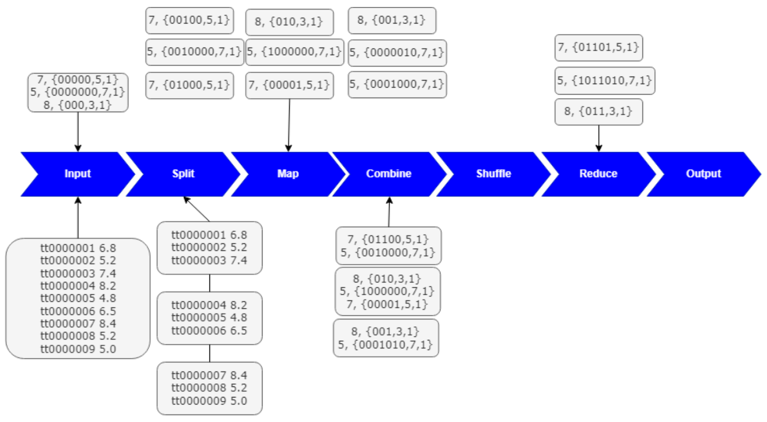


Figure 2: Stage Job 2

4.2.3 Job 3: Compute average false positive rate

- **TestMapper.java:** Implement the pattern design In-Mapper combiner.
  The setup method reads the Bloom filters and initializes the array of class Bloom filter.
  The map method counts the false positives and true positives for all the Bloom filters excepts for the one associates to the rating.
  The cleanup method emits the key rating and as value the false positive rate calculated.

- **TestReducer.java:** sums the false positive rates from the cleanup method of the mapper and emits the average false positive rate.

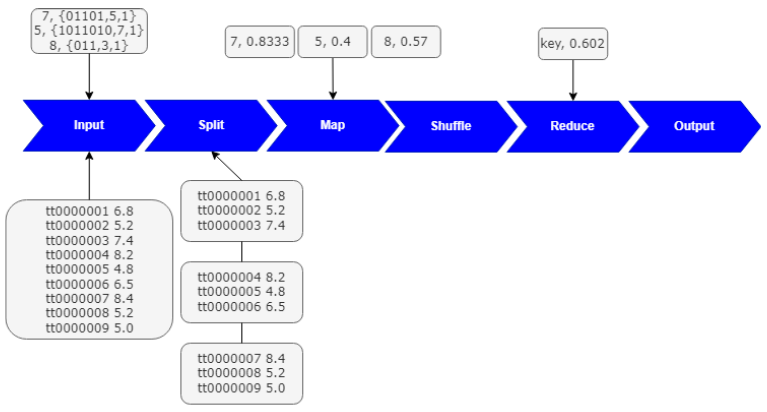


Figure 3: Stage Job 3

4.3 Performance analysis

The solution has 2 degree of freedom: the number of lines per map and the p value.
We can notice that with the increase of the number of mappers, obtained reducing the number of lines of input, the CPU time execution increases dramatically.

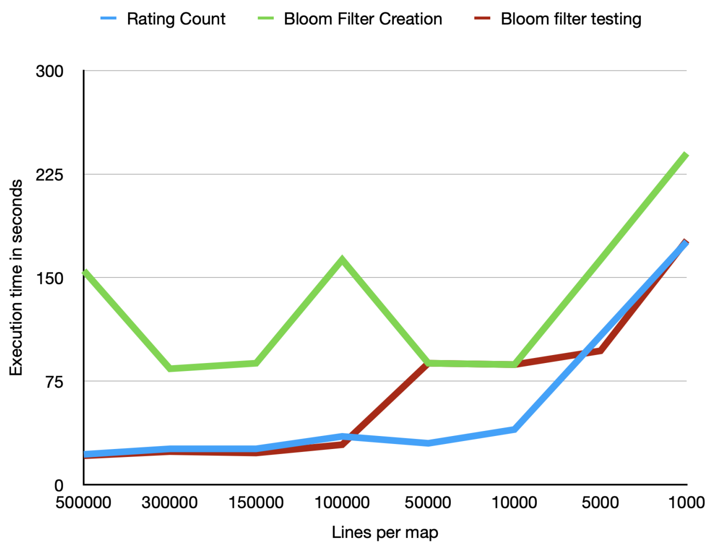


Figure 4: CPU time execution for n lines per mapper with $p = 0.01$

The testing for p values demonstrates the increasing of the average construction time with the decreasing of the FP Rate. In particular the CPU time execution increases dramatically in Job 2; for job 3 remains constant. So the Bloom filter Creation is the most expensive phase.

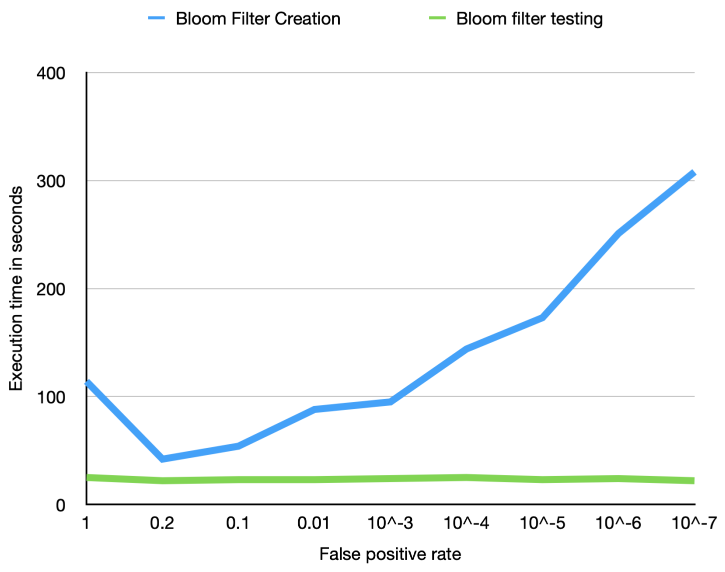


Figure 5: CPU time execution for FPR with Nline=150000

4.4 Map Reduce Complexity

4.4.1 Job 1 complexity

We managed to reduce the cost complexity of this job by using a combiner:

- **replication rate:** 1. For each rating we emit the pair (rating, 1)
- **Combiner size:** ranges from |Ri|, to min(Rj, NLineInputSplit) (worst case in which the mapper elaborates all lines with the same rating, all of them will be passed into a combiner task), supposing Ri is the smallest set of ratings and Rj the biggest, with |Ri|<NLineInputSplit
- **Reducer size q** is equal to the number of input splits.

4.4.2 Job 2 complexity

Also in this job we managed to reduce as much as possible the complexiy:

- **Replication rate:** 10/NLineInputSplit. Thanks to the InMapperCombiner approach we reduce the mapper replication rate: as input we have the whole split, as output (from the mapper) we obtain directly 10 bloom filters.
- **Reducer size:** |input split|. Each reducer will work on an iterable of bloom filters, each of them computed on an input split: this means that for each rating i, a reducer will get to elaborate a number of bloom filters equal to the number of data splits.

4.4.3 Job 3 complexity

In this testing phase we perform the testing right in the mapper, so to reduce the communication overhead.

- **Replication rate:** 10/NLineInputSplit. In the mapper we elaborate and test against each bloom filter NLineInputSplit movie ids, giving as output a fixed number of false positive rates.
- **Reducer size:** |Input splits|. Each reducer will have as value input a list of false positive rates calculated in each input split, so its size will be equal to the number of total input splits obtained.

5 Implementation on Spark

We used Spark over the same Yarn cluster configuration we used for Hadoop, in this case we choose to develop the solution in python. We performed the Spark installation following this guide https://github.com/tonellotto/cloud-computing/blob/master/notes/spark-installation-notes.md.

5.1 Design

In this section we explain the solutions considered in spark and our final choise. The way they differ is in the creation of the bloom filters (Job 2), which is the core phase.

5.1.1 Job 0

In this job the data is prepared for usage, rounding the values of ratings and sorting the movieIDs by key (rating): then the data source is saved in memory with .cache() method, which allows us to speed up the process relying on main memory rather than storage only.

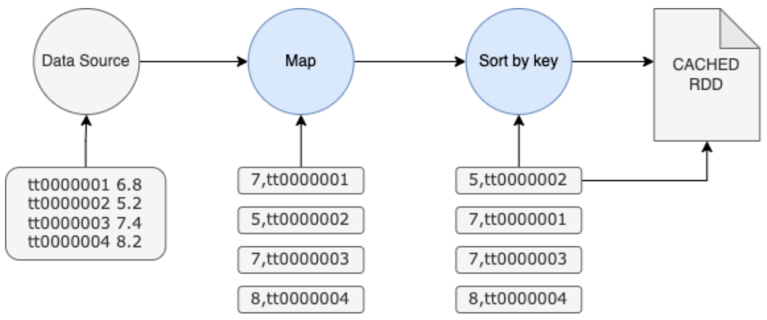


Figure 6: Preparing Cached RDD starting from file data source

5.1.2 Job 1

The following job count the number of ratings which is then used to calculate the parameters of the bloom filter, followed by the collect() action.

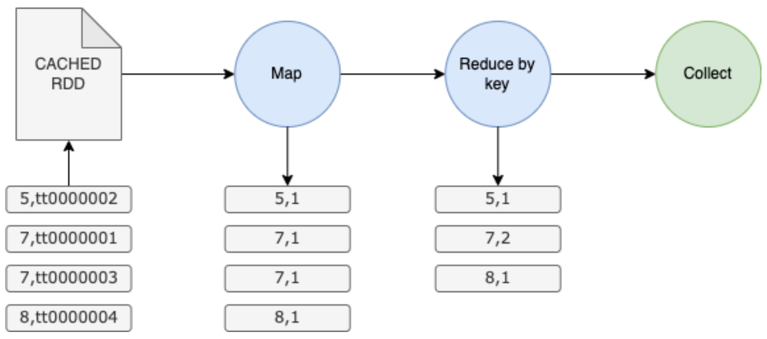


Figure 7: Rating count occurrences

5.1.3 Job 2 - Solution 1

The first solution considered for job 2 creates one bloom filter for each movie id, with only one movie id inserted: after this, reduce by key is performed merging the bloom filters and the process is completed with another collect() action.

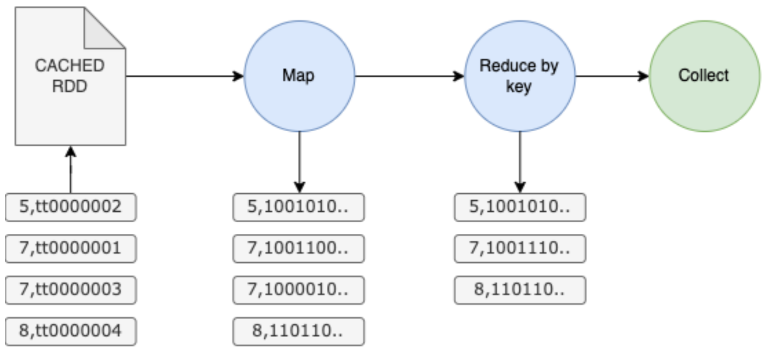


Figure 8: Bloom filter creation - solution 1

5.1.4 Job 2 - Solution 2

In this alternative solution we have a totally different approach: instead of creating one bloom filter for each movie ID, we first group by rating all the movies performing a wide transformation, and then we create one bloom filter for each rating, adding in it all the grouped movie IDs.

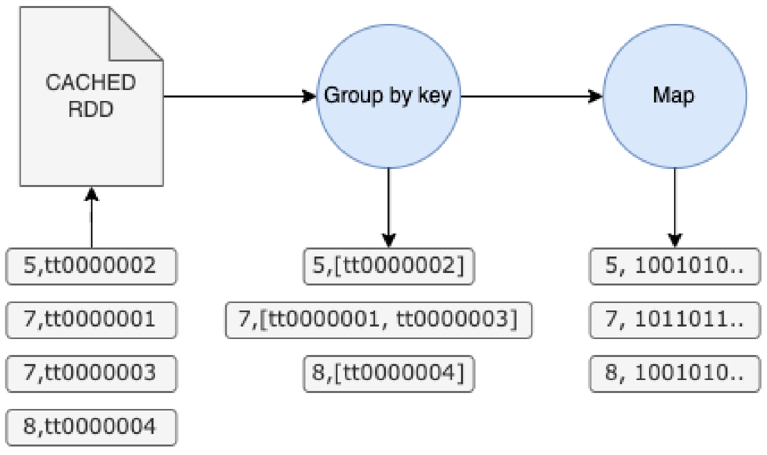


Figure 9: Bloom filter creation - solution 2

5.1.5 Job 3

In this job we perform the test on all the bloom filters: first we perform a FlatMap to check if the item is possibly present or not, then we reduce and sort by key, returning the results with the collect() action.

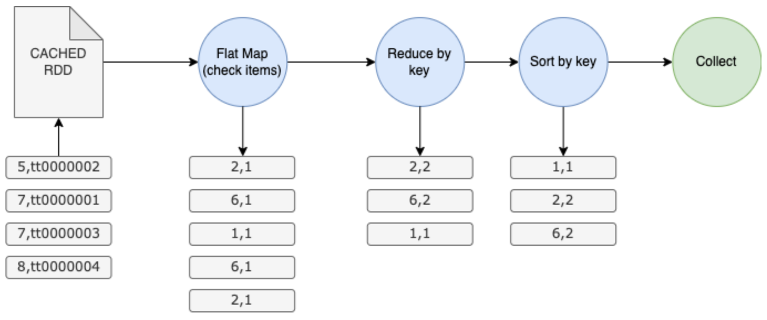


Figure 10: Test

5.2 Performance Analysis

In this section we analyze the performance of both solutions of this implementation with different values of P, to check how it affects the jobs running time.

We first wanted to check which is the most efficient implementation, then we tested it with different values of P.

In the following table we show the test that compares the performance of the 2 solutions with a fixed P (=0.00001).

Table 2 : Execution time in seconds of Spark's jobs (p= 0.00001)

	Count ratings	Create bloom filter	Test bloom filters
Solution 1	5.19	236.91	9.30
Solution 2	4.40	9.75	9.85

As we could expect solution 1 demonstrates to be less efficient than 2, as it generates a huge communication overhead since it generates one bloomfilter for each id which must be transferred to the reducers.
What's more, in this particular in which we use a small dataset, the difference of usage between narrow and wide transformation cannot be fully appreciated: in contrast to what we would expect, the second approach which implements a wide, less efficient transformation in which each partition must be aware if the others, shows to perform more than 20x faster, possibly also thanks to the RDD cache storing.

We performed then some tests with many different values of P for the solution 2, ranging from 0.2 to 10e-9 and kept track of the execution time for each job: below the graph summarizes the results obtained.

Table 3 : Execution time in seconds of Spark's jobs (p= 0.00001)

P	Count ratings	Create bloom filter	Test bloom filters
0.2	4.49	2.85	9.56
0.1	4,87	3,41	9,80
10e-2	4,46	4,36	9,72
10e-3	4,46	3,74	10,02
10e-4	4,61	4,55	10,70
10e-5	4,72	5,24	9,79
10e-6	4,52	5,65	10,71
10e-7	4,64	11,34	12,17
10e-8	4,62	10,30	11,88
10e-9	4,67	11,51	12,1

While the count of ratings was afflicted minimally, as we could expect, the greater impact was noticed on the creation and testing of bloom filters, which increased almost linearly with time.
In the image below we can have a clearer view of what stated above.

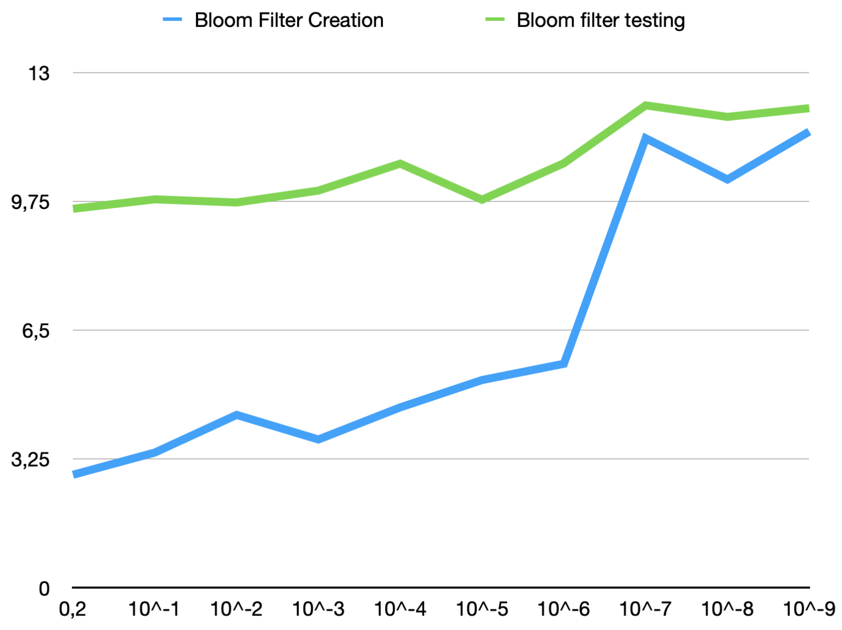
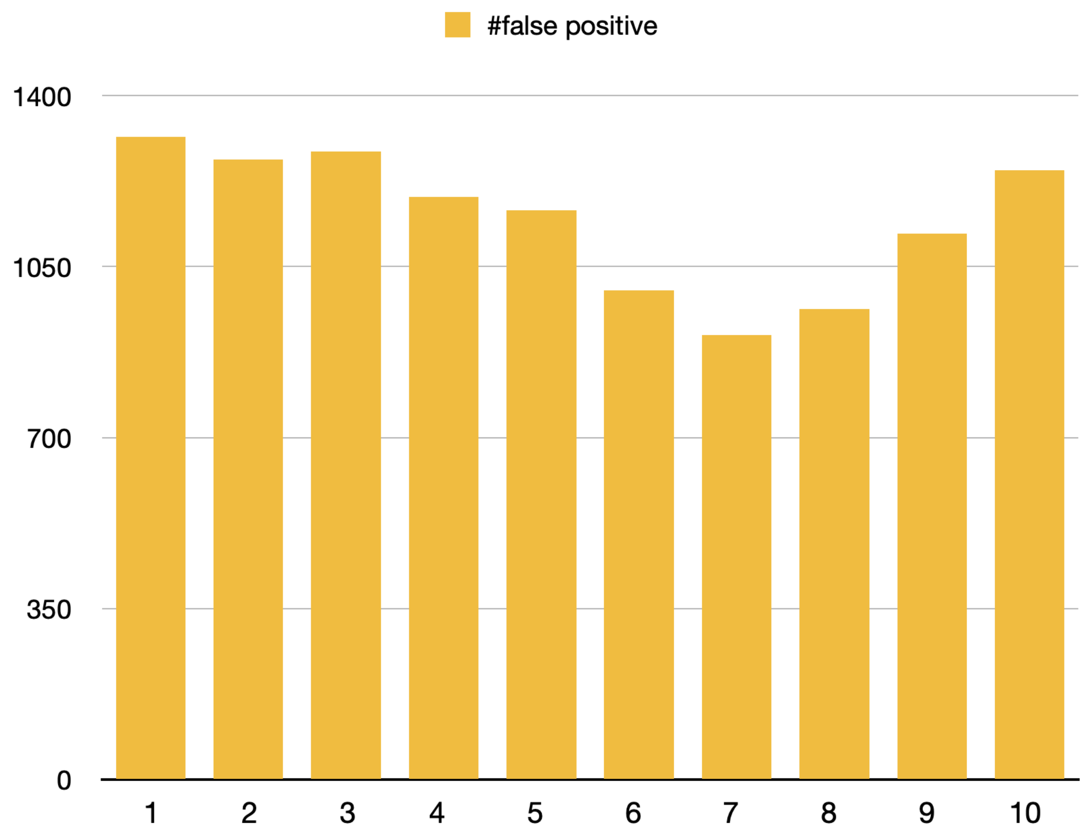


Figure 11: Bloom filter creation - solution 2

6 Comparison and Conclusion

6.1 Bloomfilter FPR

Our initial The histogram above shows the number of false positive occurrences in each bloomfilter having p set to 0.001



It is very interesting to notice how the false positive ratio has a small variance. This means that doesn't matter how the data are distributed. In fact taking the top and the bottom value there are 2551 movies with rating 1 and 371192 with rating 7, while the false positive cases are 1316 and 910 respectively. There ratios between movies with rating 7 and movies with rating 1 is 145.51 while the ratio between their false positive occurrences is just 1.45.

Based on the value of the standard deviation that has been calculated over 10 different trials, the variance obtained is around 0. This is an optimal value cause it is next to zero both for Hadoop and Spark.

6.2 Performance Comparison

As reported in previous results, we developed a very similar solution in Spark and Hadoop, but we have that Spark performs much more better, in average is 24 times faster in the bloom filter creation phase, as the following graph shows.

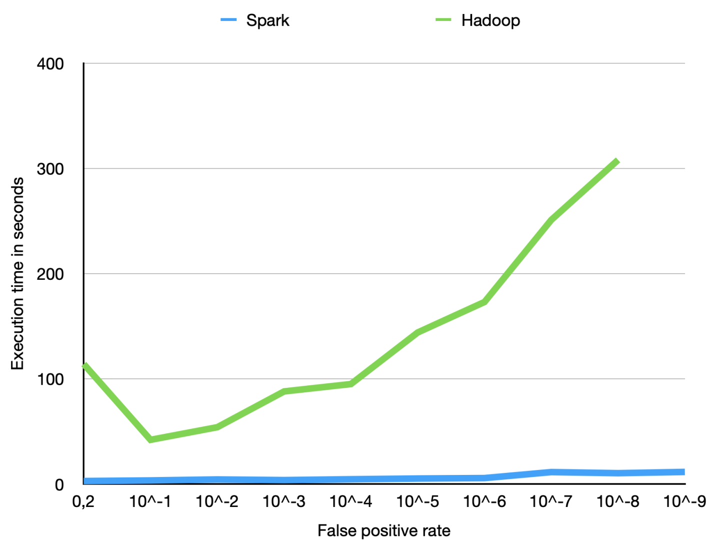


Figure 12: Bloom filter creation - solution 2

6.3 Final notes

With our implementation not only we proved bloom filter's literature formulas, we also found an optimal solution in terms execution time and data transfer between stages. Anyway we know that our analysis could be biased from the dataset size, it would be very interesting to test our implementation with a bigger dataset especially to compare the 2 solutions implemented for Spark, such experiments could lead to literature facts about wide and narrow transformations in real application contexts.