UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Artificial Intelligence and Data Engineering

Bloom Filters in MapReduce

Project report

Luana Bussu

Luca Caprioli

Martina Marino

Roberta Matrella

Academic Year: 2021/2022

Contents

1 Introduction **2**

2 Design **3**
2.1 Dataset 3
2.2 Assumptions 3
2.3 Parameter Calibration 4
2.4 Bloom Filters Creation 5
2.5 Bloom Filters Validation 7

3 Hadoop **9**
3.1 Implementation 9
3.1.1 Bloom Filter 9
3.1.2 Parameter Calibration 9
3.1.3 Bloom Filter Creation 9
3.1.4 Parameter Validation 10
3.2 Results 11

4 Spark **15**
4.1 Implementation 15
4.2 Results 16

5 Experimental results **17**

6 Conclusion **20**

1 Introduction

A *Bloom Filter* is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. The cost paid for efficiency is that there might be some false positive results. False positive means, it might tell that an element is present but actually it is not.

It is initialized as an array of bits set to 0. Then some hashing functions are needed, each of which can map an element of the database to one of the m positions of the array. The mapping must also have a uniform distribution. To add an element the k hashing functions are recalculated to obtain the positions of the relative bits and setting them to 1. The verification of the presence of an element is accomplished by calculating the hashing functions on the controlled element, determining which bits should be set. The element is absent if even one of the bits is 0. For this reason, as the number of elements inserted in the *Bloom Filter* increases, the percentage of false positives becomes higher.

This data structure is efficient because the time needed to add or to find an element is constant and depending from the number of hash functions selected but independent from the number of elements that are already present.

In this project is presented a possible MapReduce implementation of a *Bloom Filter* using *Hadoop* and *Spark* and finally a comparison of the results obtained.

The code is available in the following repository:
https://github.com/martimarino/Bloom-Filters-in-MapReduce.git

2 Design

The workflow that has been followed is structured in 3 stages: the first consists in the calibration of the parameters, then the Bloom Filters are created and populated and finally the performances are evaluated in terms of false positive rate.

2.1 Dataset

The dataset exploited is taken from IMDb and contains a list of films with the relative ratings. It is composed by the attributes [*tconst*, *averageRating*, *numVotes*] but only the first two have been considered in this work, i.e. id of the film and relative average rating.

In Figure 1 the bar diagram shows the films distribution considered their rounded ratings.

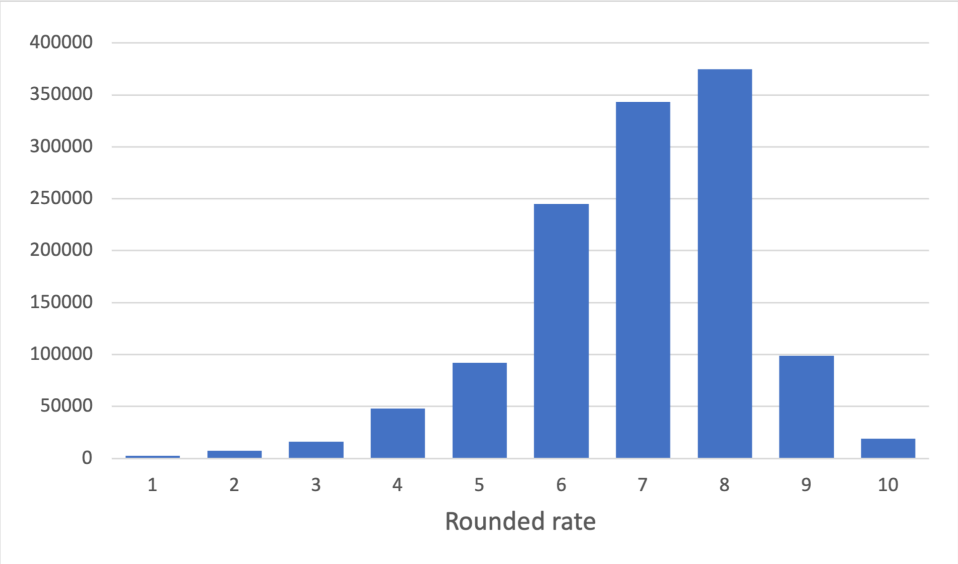

Figure 1: Rate distribution

2.2 Assumptions

To build a Bloom filter four main parameters are needed:

- p: false positive rate;
- k: number of hash functions;
- m: number of bits of the bloom filter;
- n: number of keys in input to the bloom filter.

The relations between these values can be expressed as:

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

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

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

To design a *Bloom Filter* with a given false positive rate p, first the number of keys n to be added have to be estimated, then the number of bits are computed and finally the number of hash functions k to use can be obtained.

The average ratings in the dataset are rounded to the closest integer value, and a *Bloom Filter* is computed for each rating value. For this reason the number of *Bloom Filters* will be 10, given that the rating value is in the range [1,10].

Given n and decided p a priori, the parameter m is dependent from both whereas k is only dependent from p as proved replacing m in its formula. Hence every *Bloom Filter* may have a different m but all share the same number k of hash functions.

2.3 Parameter Calibration

The first stage calibrates the necessary parameters for the following steps.

Every **mapper** build an associative array counting how many films per rate. In particular, it parses each film of those that locally have, rounds its rating and increases the relative counter in the array. Then it emits $< rate, count >$ pairs. The in-mapper combiner has been used in order to reduce the number of intermediate data produced.

class MAPPER

```
1: method INITIALIZE
2:     counter ← new Array[rates]
3: method MAP (film f)
4:     rate ← round(f.rate);
5:     counter[rate] ← counter[rate] + 1;
6: method CLOSE
7:     for all r ∈ rates do
8:         EMIT(r, counter[r])
9:     end for
```

The **reducer** takes in input the $< rate, count >$ values, computes n for every *Bloom Filter* summing the counters of the mappers and define the relative m and k.

class REDUCER

1: **method** REDUCE (rate, counters[c1,c2,...])
2: $n \leftarrow 0$
3: **for all** c $\in$ counters **do**
4: n $\leftarrow$ n+c
5: **end for**
6: $m \leftarrow -n * \ln p/(\ln 2)^2$
7: k $\leftarrow \ln 2 * m/n$
8: EMIT($rate, \{m, k, n\}$)

A second solution has been implemented in order to emit a lower number of bytes in output from each reducer. The **reducer** computes only the parameter n.

class REDUCER

1: **method** REDUCE (rate, counters[c1,c2,...])
2: $n \leftarrow 0$
3: **for all** c $\in$ counters **do**
4: n $\leftarrow$ n+c
5: **end for**
6: EMIT($rate, n$)

2.4 Bloom Filters Creation

The second stage consists in the creation of the *Bloom Filter*. They have been implemented as arrays of bits filled with the result of the hashing functions generated for every film.

Every **mapper** creates an array of *Bloom Filters*, one for each rate, using the parameters calculated during the Parameter Calibration stage. In the *Map* function, for each film the indices of the correspondent *Bloom Filter* are set. At the end, in the *Close* function, for each rating is emitted the correspondent *Bloom Filter*.

class MAPPER

1: **method** INITIALIZE
2: **for all** i ∈ rates **do**
3: bf ← **new** BloomFilter(m,k)
4: **end for**
5: **method** MAP (film f)
6: $rate \leftarrow round(f.rate)$
7: $bf[rate].insert(f)$;
8: **method** CLOSE (film f)
9: **for all** i ∈ rates **do**
10: EMIT(rate, bf)
11: **end for**

Each **reducer** computes the *or* operation between the partial *Bloom Filters* for each rating in order to get the unified result.

class REDUCER

1: **method** REDUCE (rate, bf[bf1,bf2,...])
2: $result \leftarrow$ **new** $BloomFilter(b1.m)$
3: **for all** bf ∈ [bf1,bf2,...] **do**
4: result.or(bf)
5: **end for**
6: EMIT(rate, result)

Another solution for the creation *Bloom Filter* has been implemented: every **mapper** computes the k hash functions on the films' ID and emits $< rate, indices >$ pairs concerning the rounded rate and an integer array representing the binary configuration of the bits to set.

class MAPPER

1: **method** INITIALIZE
2: $indices \leftarrow$ **new** $Array[k]$
3: **method** MAP (film f)
4: $rate \leftarrow round(f.rate)$
5: $counter[rate] \leftarrow counter[rate] + 1$;
6: **for** i ∈ 0,...,k **do**
7: indices[i] ← abs(MurmurHash(f.id) % m[rate])
8: **end for**
9: EMIT(rate, indices)

The **reducer** receives the output of the mappers, creates the *Bloom Filters*, each with its own length m, fills them with the indices received and writes them to the HDFS.

class REDUCER

```
1: method REDUCE (rate, indices)
2:     BitSet bs ← new BitSet(m)
3:     for all arr ∈ indices do
4:         for all i ∈ arr do
5:             bs.set(arr[i])
6:         end for
7:     end for
8:     BloomFilter bf ← new BloomFilter(k, m, bs)
9:     EMIT(rate, bit)
```

2.5 Bloom Filters Validation

The last stage is the evaluation of the false positive rate for each rating. First of all the *Bloom Filters* previously created and populated are taken from HDFS, then the MapReduce paradigm does what follows.

Every **mapper** builds an array containing a counter for each rate representing the number of false positives of the relative *Bloom Filter*.

class MAPPER

```
1: method INITIALIZE
2:     fp_counters ← new Array[rates]
3:     for all r ∈ rates do
4:         bf ← readFromFile()
5:         bloomFilters[r] ← new BloomFilter(r, bf)
6:     end for
7: method MAP (film f)
8:     rate ← round(f.rate) - 1
9:     for all r ∈ rates do
10:        if rate == r then
11:            continue
12:        end if
13:        if bloomFilters[r].find(f) == true then
14:            fp_counters[r] + +
15:        end if
16:    end for
17: method CLOSE
18:    for all r ∈ rates do
19:        EMIT(rate, fp_counters)
20:    end for
```

The **reducer** sums all the counters relative to the same rating and emits them.

```
class Reducer
1: method Reduce (rate, mapper_counts[1,2,...])
2:     counter = 0
3:     for all i ∈ mapper_counts do
4:         counter+ = i
5:     end for
6:     Emit(rate, counter)
```

3 Hadoop

3.1 Implementation

Concerning the hashing, the family `org.apache.hadoop.util.hash.Hash.MURMUR_HASH` hash function family has been used. This is a very fast, non-cryptographic hash suitable for general hash-based lookup.

For what concerns the different solutions compared in 3.1.2 and 3.1.3 they can be found in the *test* branch of the GitHub link. In particular, the *parametersDriver* module presents in both stages the implementation of the second solution, while the *sendBF* one has the combination of the others (all parameters evaluated in the Reducer phase and sending BF instead of indices).

3.1.1 Bloom Filter

The *Bloom Filter* class makes use of the object Bitset to implement the array of bits because it provides good results in terms of memory usage. It extends the Writable interface, so the methods for serialize and de-serialize objects over Hadoop have been redefined.

3.1.2 Parameter Calibration

Regarding the Calibration Reducer, two different solutions have been considered in order to compare the output bytes.

In the first solution, the Reducer compute the value of m and k starting from the total count of films per rate and emit, for each rate, an array of IntWritable objects (n, m, k).

In the second solution, the Reducer compute and emit only the total number of films per rate (n) and the values of m and k are computed on the driver allowing the reducer to emit a single IntWritable instead of an array. This solution is possible because the value of m depends on n and p, while the value of k only depends on p, so we avoid to emit it for each rate being equal for all of them.

3.1.3 Bloom Filter Creation

The creation stage also has two different implementations considering where the *BloomFilter* object is created.

In the first solution in the map phase a local $ArrayList < BloomFilter >$ is created and sent to the reduce phase that does the or operation to create the final 10 filters.

In the second solution the *map* function emits an IntArrayWritable filled with the indices of the positions of the filter to be set. The BitSet object is then created and filled in the reduce phase and then passed to the *BloomFilter* constructor.

3.1.4 Parameter Validation

The last stage evaluates the false positive rate basing on the Bloom Filters and the parameters computed in the first two stages. It also exploits the MapReduce paradigm to parallelize the process. The filters just created are read and the **reduce** function takes care of the false positives count. This result is finally compared, for every rating, with the total number of film for each filter in the Driver.

3.2 Results

Using $p = 0.01$ the result is a value of k = 6 and the values of m are shown in Table 1.

Rating	n	m
1	2555	24489
2	6467	61986
3	17051	163434
4	42128	403799
5	97625	935741
6	215643	2066950
7	369543	3542091
8	352852	3382107
9	118182	1132781
10	19028	182384

Table 1: m values for each rating

Table 2 shows the result in terms of false positive count and false positive rate for every class.

Rating	p = 0.01		p = 0.05		p = 0.1	
	FP	FPR	FP	FPR	FP	FPR
1	12690	0.0102	61008	0.0493	141458	0.1142
2	12523	0.0101	62557	0.0507	144156	0.1168
3	12208	0.0099	61982	0.0506	142771	0.1166
4	12024	0.0100	60186	0.0502	139430	0.1163
5	11638	0.0101	57499	0.0503	134094	0.1173
6	10536	0.0102	51551	0.0503	120113	0.1171
7	9015	0.0103	43675	0.0501	101586	0.1166
8	8796	0.0099	44940	0.0506	102946	0.1159
9	11406	0.0101	56570	0.0504	130603	0.1163
10	12117	0.0099	62011	0.0507	141230	0.1156

Table 2: FP and FPR with different p values

The different solutions for each stage were compared in terms of output bytes and execution time by varying the number of mappers and reducers. For each one, the one that guarantees the best trade off has been chosen.
For the first stage, we considered the two different implementations of the *Reducer* explained in 2.3: in the first solution (red one), the *Reducer* computes the values of n, m and k and emit, for each rate, an array of IntWritable objects (n, m, k); in the second solution (blue one), the Reducer compute only the total number n of films per rate, while the values of m and k are computed on the *Driver*, allowing the reducer to emit, for each rate, a single IntWritable instead of an array. This solution is possible because the value of m depends on n and p, while the value of k only depends on p, so we avoid to emit it for each rate being equal for all of them. In Figure 2 the execution time of the two different solutions have been compared using different numbers of mappers and reducers.

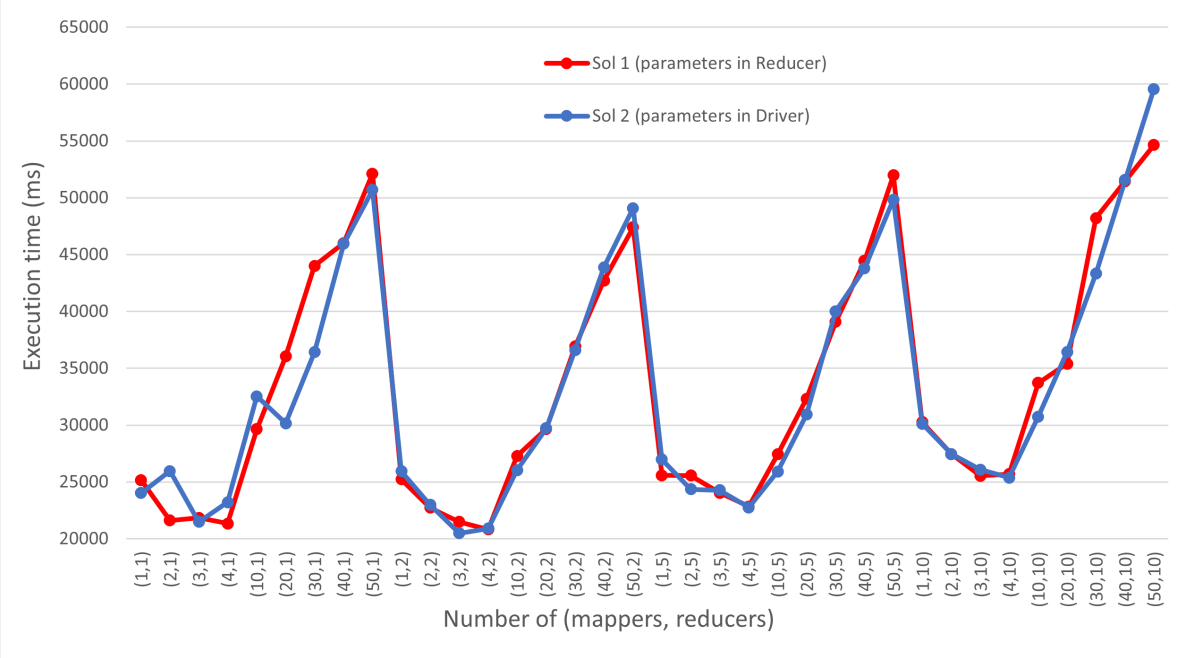

Figure 2: First stage solution execution time comparison

The two solutions have been compared also in terms of output bytes.

	Solution 1	Solution 2
Number of bytes written (HDFS)	374	252

As expected, the output bytes of the second solution are lower then the first. The second, indeed, needs to emit all the parameters computed, the first one emits only the n parameter.
The two solutions both provide comparable performances in terms of execution time, but the second one ensures also a lower number of output bytes. For this reason, **the second**

one has been chosen.

Regarding the second stage, the two solutions of 2.4 have been compared in terms of mapper output bytes. In the first solution (yellow one), each mapper creates a *Bloom Filter* for each rate and sends all of them to the reducer. In the second solution (green one) the mapper sends, for each film, the indices calculated with the hash function. As shown in Figure 3, the mapper output bytes of the second solution are constant. It was expected because the output bytes of the mapper are linear w.r.t. the input lines. Instead, in the first solution, increasing the number of mappers the number of output bytes emitted increases too because each mapper creates and emits ten *Bloom Filters*, one for each rate. For this reason, more mappers means more output bytes.

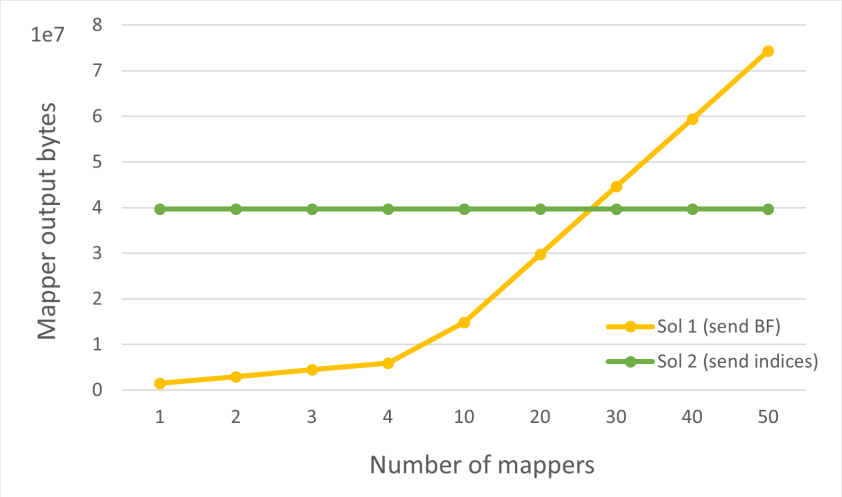

Figure 3: Second stage solution mapper bytes comparison

Considering the memory overhead, the first solution maintains as class fields the bloom filters and we can encounter scalability problems due to the store of all the Bloom Filters in memory, so the second solution has been preferred.

For the sake of completeness in Figure 4 the execution time of the two different solutions are shown using different numbers of mappers and reducers.

Figure 4: Second stage solution execution time comparison

Basing on the aspects previously highlighted, **the second solution has been chosen** because of the higher number of bytes emitted with a higher number of mappers and the memory overhead. Indeed, even if the execution time of the first solution is higher than the second, the difference becomes smaller with the increasing of the number of mapper.

4 Spark

4.1 Implementation

Following the implementation made in Hadoop, a similar one was built in the Spark framework. The following graphs show the workflow of the the different stages.

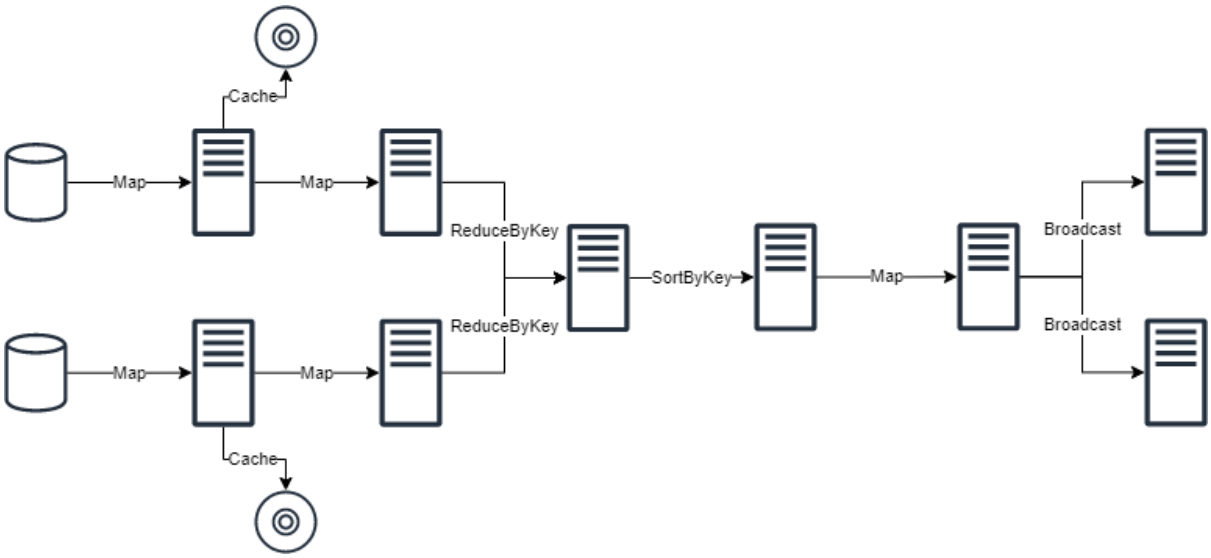

Figure 5: Spark's Parameter Calibration Workflow

The application takes as inputs the *dataset* and the value of p. In the Parameters Calibration stage (Figure 5) the data set is loaded and mapped inverting the original $< key, value >$ pairs, caching it as it will be used in every stage. An aggregation is then executed in order to count for each vote the number of movies. Finally k, an array of m values and the number of films per vote are computed and saved in broadcast variables for all the worker to be accessed.

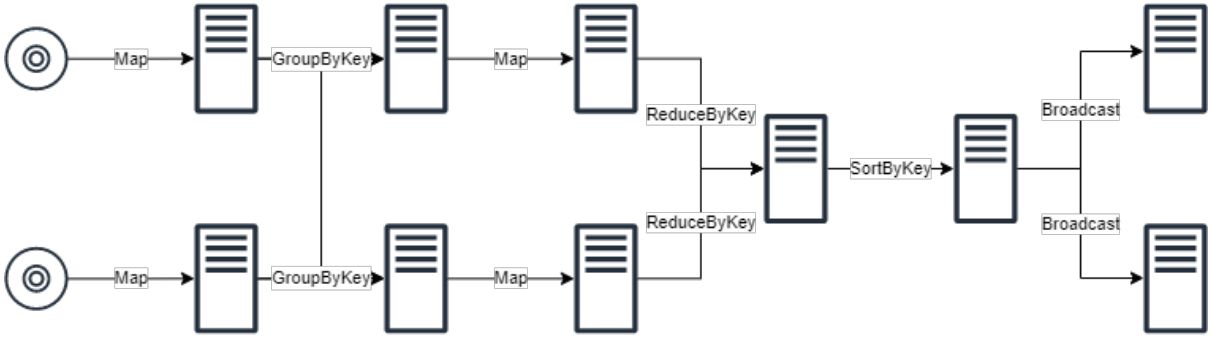

Figure 6: Spark's Bloom Filter Creation Workflow

In the Bloom Filter Creation stage (Figure 6) the partitioned data set is loaded from the cache and the different movies are grouped in lists by their respective keys, shuffling them between workers. Each list is the used to calculate the k hashes of each title in order

to populate local bloom filters. Finally the global bloom filters are created by aggregating the local ones and are then sent in broadcast to all workers following the same reasoning of the previous stage.

4.2 Results

Table 3 shows the result in terms of false positive count and false positive rate for every bloom filter at different values of p.

Rating	p = 0.01		p = 0.05		p = 0.1	
	FP	FPR	FP	FPR	FP	FPR
1	12466	0.0100	61435	0.0496	127149	0.1026
2	12545	0.0101	60313	0.0488	123988	0.1005
3	12244	0.0099	61709	0.0503	123247	0.1006
4	12464	0.0104	60108	0.0503	120097	0.1006
5	11838	0.0102	57728	0.0502	115003	0.1000
6	10114	0.0101	50230	0.0503	100590	0.1008
7	9222	0.0102	45259	0.0503	90746	0.1008
8	8913	0.0102	43689	0.0502	87619	0.1008
9	11591	0.0101	57485	0.0503	115517	0.1011
10	12380	0.0101	61066	0.0499	123819	0.1013

Table 3: Spark Framework FP and FPR with different p values

To check the correctness of the results an accumulator array was implemented and then populated by each worker by iterating the data set. In this last part the k hashes were computed on each film for each other vote different from its own, increasing the respective position of each vote in the accumulator in case of a False Positive.

5 Experimental results

In this paragraph the execution of the first two stages of the Hadoop and the Spark implementation are compared both in terms of FPR and execution time.

Rating	p = 0.01		p = 0.05		p = 0.1	
	Hadoop	Spark	Hadoop	Spark	Hadoop	Spark
1	0.0102	0.0100	0.0493	0.0496	0.1142	0.1026
2	0.0101	0.0101	0.0507	0.0488	0.1168	0.1005
3	0.0099	0.0099	0.0506	0.0503	0.1166	0.1006
4	0.0100	0.0104	0.0502	0.0503	0.1163	0.1006
5	0.0101	0.0102	0.0503	0.0502	0.1173	0.1000
6	0.0102	0.0101	0.0503	0.0503	0.1171	0.1008
7	0.0103	0.0102	0.0501	0.0503	0.1166	0.1008
8	0.0099	0.0102	0.0506	0.0502	0.1159	0.1008
9	0.0101	0.0101	0.0504	0.0503	0.1163	0.1011
10	0.0099	0.0101	0.0507	0.0499	0.1156	0.1013

Table 4: Hadoop and Spark FPR for different values of p

The obtained results concerning FPR shown in Table 4 are similar between the Hadoop and Spark implementation, specifically at $p = 0.01$ and $p = 0.05$, while at $p = 0.1$ there is an mean improvement of 0.016 on the Spark side.

In order to compare the results of the Hadoop and Spark implementations, it is necessary to choose a configuration of mappers-reducers for the Hadoop framework. In Figure 7 and Figure 8 are reported the selected solutions for stage 1 and stage 2 respectively. The trend shows that the execution time increases proportionally with the mappers if the number of reducers is fixed and, overall, also with the increasing of the number of reducers. From the plots a pattern can be identified: the execution time decreases from 1 to 4 mappers and increases from 4 to 10 so, at number of mappers equal to 4 the plot has its local minima and these local minima do not differ significantly from each other.

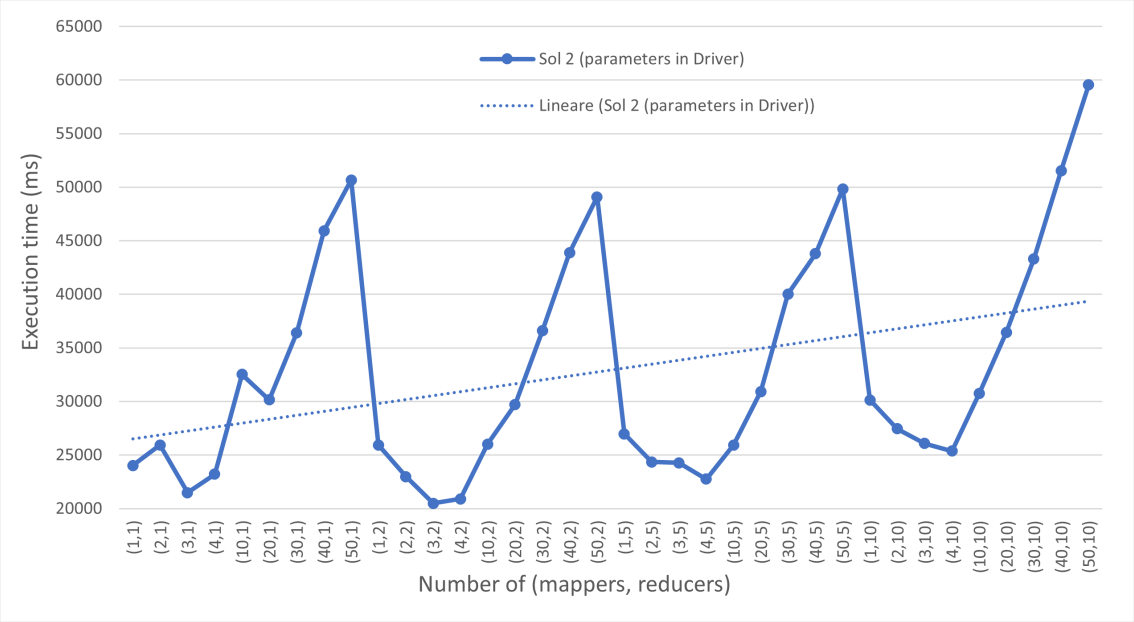

Figure 7: stage1

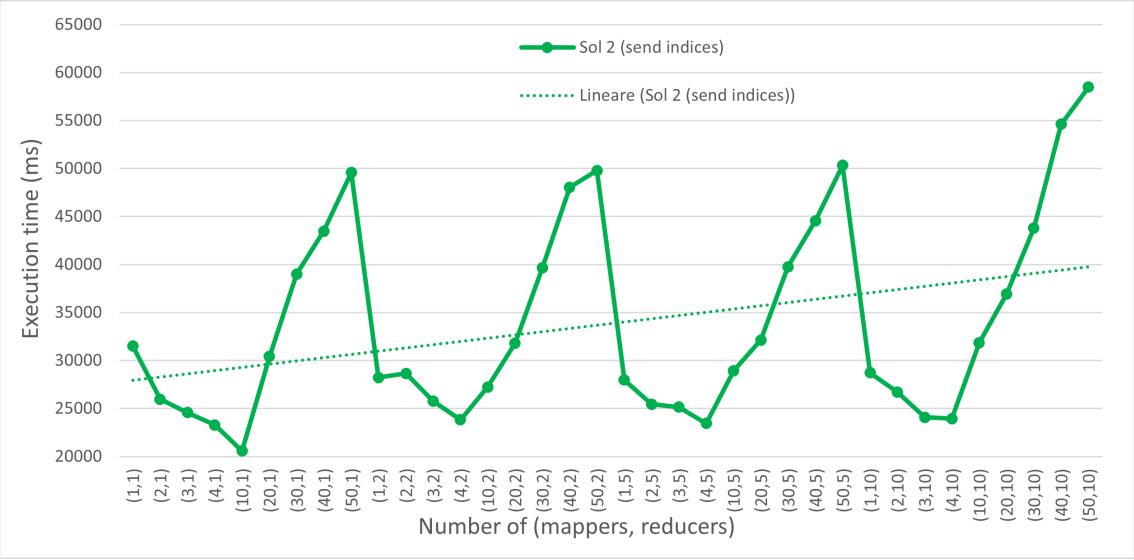

Figure 8: stage2

For simplicity, the configuration with the lowest total execution time of the algorithm (stage 1 and stage 2) has been chosen (Figure 9), corresponding to the one with 4 mappers and 2 reducers.

Figure 9: Total execution time plot

In Table 5 are shown the mean execution times of the different stages in both the Hadoop and Spark frameworks over 10 iterations. As expected, the execution times are lower in Spark, specifically in the second stage were more computation is needed for each row of the data set.

	Stage 1 (ms)	Stage 2 (ms)	Total (ms)
Hadoop	20906	23821	44727
Spark	17833	7857	25690

Table 5: Hadoop-Spark comparison

6 Conclusion

In this report the Bloom Filter data structure was implemented and tested in both the Hadoop and Spark frameworks, considering both the False Positive Rate and execution time of the two to highlight the differences. Spark's capability of in-memory processing, exploited by caching the mapped input data set, built-in support for distributed computing, which allows it to divide the filter into smaller chunks and process them in parallel, and for iterative processing, which makes it more suitable for implementing algorithms that require multiple passes over the data, reduce the execution time needed for computing the parameters and filling the Bloom Filters of almost half compared to its Hadoop counterpart. Hadoop, on the other hand, is designed for batch processing and does not have native support for distributed computing and in-memory processing, which can make it less efficient for creating Bloom Filters and in general for computation on large data sets.