UNIVERSITÀ DI PISA

Computer Engineering

Cloud Computing

Movies Bloom Filters

Project Documentation

TEAM MEMBERS:
Matteo Pierucci
Elisa De Filomeno
Giacomo Pacini
Biagio Cornacchia

Academic Year: 2021/2022

Contents

1 Introduction

This is a report on a project which investigates how to build Bloom Filters using the MapReduce approach in Hadoop and Spark. The dataset used contains a list of movies and their relative ratings. Ratings can be rounded to their closest integer values, obtaining numbers from 1 to 10. This work aims to design and implement a MapReduce algorithm for the construction of 10 Bloom Filters, one for each rating value, using both Hadoop and Spark frameworks. The performances of the implementations are tested by computing the false positive rate for each rating and by registering the execution times and resources needed for each implementation.

1.1 Bloom Filter

A Bloom Filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. In our case it's used to check whether a movie has a certain rounded rating, the bloom filter relative to that rounded rating will answer True or False. False negatives are not possible for how Bloom filter construction is defined, only false positives.

Bloom filter is a bit-array of m bits. 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 bits to 1, if it was equal to 0. Let's consider a Bloom Filter with the following characteristics:

- 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 basic operations that a Bloom Filter supports are:

- add(element): to insert an element in the Bloom Filter
- lookup(element): to check whether an element is present in the filter with a false positive probability p.

There is a trade-off between the space advantage of this data structure and the false positive rate p. In fact, given n and p, other parameters (m,k) can be optimized using the formulas:

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

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

The formula to compute false positive rate is:

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

where FP indicates the number of false positives and TN the number of true negatives.

1.2 Dataset

The dataset used in this work contains 1256807 movies and their average ratings. It was extracted on 30/08/2022 from IMDb datasets.

The attributes of the dataset are:

- movie id: uniquely identifies a movie (e.g. "tt0000001")
- average rating: reported as a float number from 1 to 10 (e.g. 5.7)
- number of votes: reported as an integer (e.g. 1882)

We analysed the frequency and probability distributions of rounded ratings in the dataset, extracting the graphs:

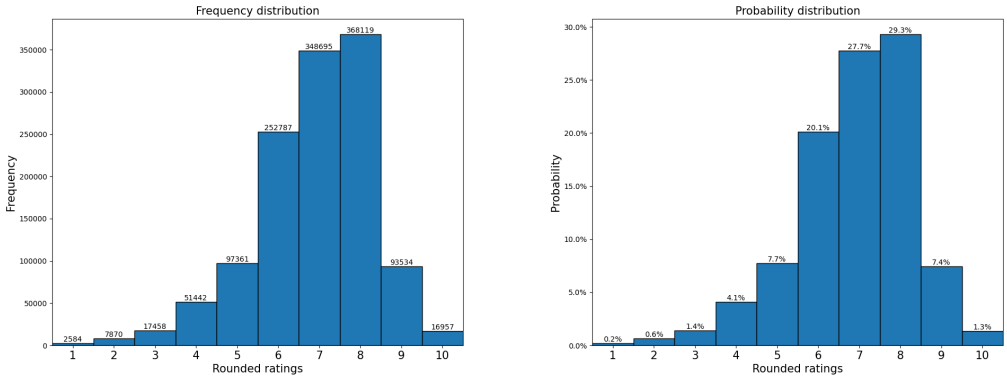

Figure 1: Frequency and Probability distributions of rounded ratings in the dataset

As we can see the dataset is not balanced, movies with a rounded rating equals to 6,7,8 make up over 75% of the entire dataset, being much more frequent than movies of other ratings.

2 The P parameter

The choice of p (false positive rate) is essential to obtain good performances of our filters. For example, a value of p equals 0.01 for one of our filters means that there is a probability of 1% that the filter will answer True when in fact the movie doesn't have that specific rounded value. In general p value should be low to minimize the *FPR*.

Since we will have to build 10 different Bloom filters, each with a different value of the numerosity n, we will obtain 10 different combinations of m and k.

Our aim is to determine the best compromise between performances (memory occupation and execution times) and False Positive Rate over the given dataset.

2.1 Multipositive rows rate

An interesting metric that we have found useful to better tune the P parameter is the ratio between the number of movies that triggered more than one Bloom filter and the total number of movies.

$$MPR = count(rows\ false\ positive)/tot\ rows$$

This score can be very useful for the tuning of P for a real application that exploits more Bloom filters to obtain some kind of information.

For example, assuming a service that exploits the Bloom filters to return the rating class to which a given movie belongs without effectively searching inside the list of movies for each rating class, we should note that each time that it happens that a record triggers more than one Bloom filter, it becomes necessary to iterate over the whole list of movies belonging to the rating classes whose filter was triggered, slowing down the application execution by far.

In order to find how often this happens on our implementation, we implemented a procedure in the step of Bloom filters testing that counts the number of rows that triggered more than one Bloom filter.

2.2 Memory occupation

In the following plot are reported the calculated m values for each rating class (according to its numerosity) for some remarkable values of P:

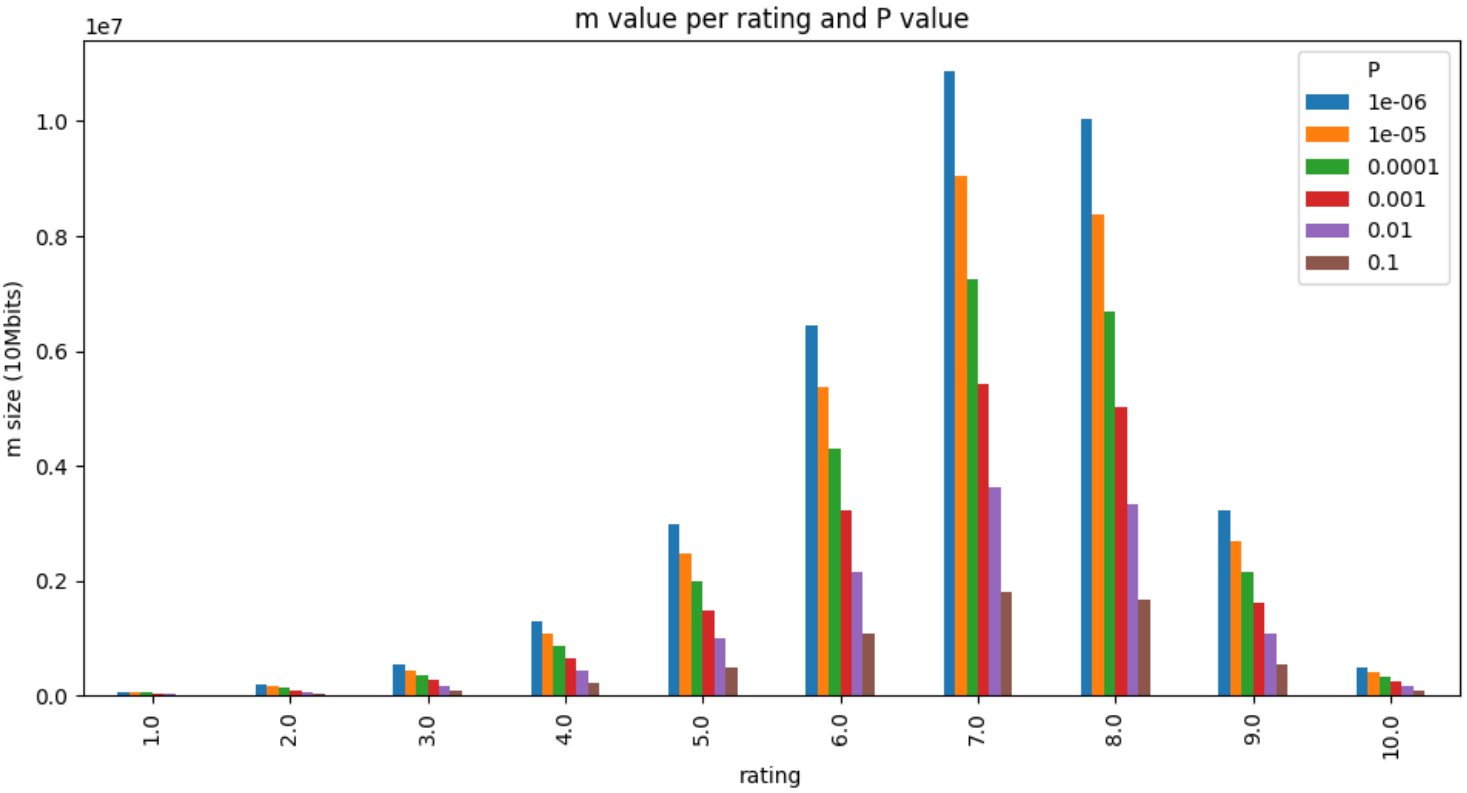

Figure 2: m values for each rating class at different Ps

We can observe that the memory occupation caused by the minority classes (like the rate 1 or 2) are very low, while the most part of memory is occupied by the filters of the more frequent ratings.

We can also observe that the memory occupation increases linearly with $-log_{10}(P)$. This

fact is even clearer if we consider the sum of the values of m for each Bloom filter at a given P and we represent those values with a logarithmic scale:

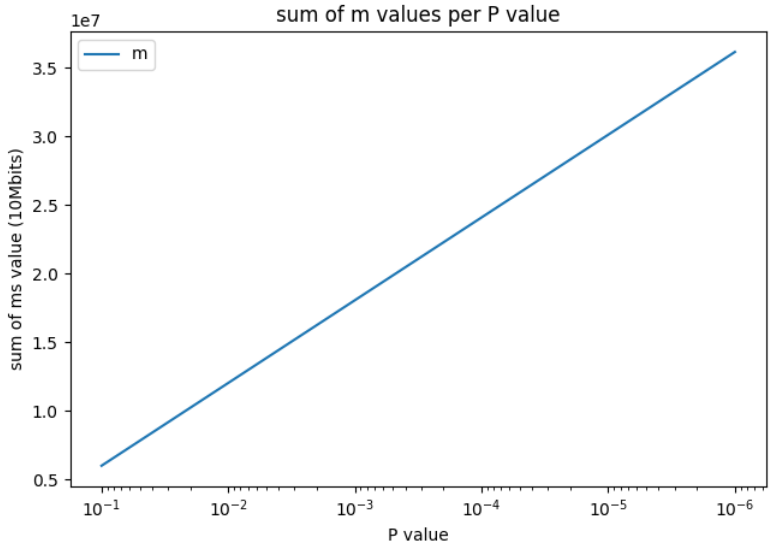

Figure 3: m values for each rating class at different Ps

2.3 Execution time on P value

While we can obtain the amount of memory necessary for the filters even before testing a real implementation, the relation between execution time and P value cannot be calculated a priori, and strongly depends on the particular implementation. This is why we will report those statistics in the following chapters.

In our study we will test our implementations of the Bloom filters creation with the following values of P:

- 10^{-1}
- 10^{-2}
- 10^{-3}
- 10^{-4}
- 10^{-5}
- 10^{-6}

3 Pseudocode

We implement the algorithms to create and test the Bloom Filters with the MapReduce paradigm in order to make them more efficient and faster. We divided our solution in three main phases, that corresponds to three different jobs:

- JOB 1: films per rating count,
- JOB 2: Bloom Filters creation,
- JOB 3: Bloom filters test.

In all the three phases we exploit the In-Mapper Combiner design pattern, which aggregates the local results of the mapper to reduce the amount of traffic between the nodes and to make the Reducer task more efficient.

3.1 Job 1: films per rating count

The first JOB counts the number of films per rating. The IN-MAPPER COMBINER processes the dataset rows, it extracts the rounded rating of each film and increases the counter of that specific rating, which is saved in memory. After the processing of all films, it emits the local cumulative number of films per rating.

Algorithm 1 IN-MAPPER COMBINER: films per rating count

```
class MAPPER
    method INITIALIZE
        counters ←new AssociativeArray
    method MAP(key k, doc d)
        roundedRating ←round(d.rating)
        counters[roundedRating − 1] ← counters[roundedRating − 1] + 1
    method CLOSE
        i ← 1
        for all counter c ∈ counters do
            EMIT(i, c)
            i ← i + 1
end class
```

The REDUCER aggregates the results of different IN-MAPPER COMBINERS using the rating as key.

Algorithm 2 REDUCER: films per rating count

```
class REDUCER
    method REDUCE(key i, list[c1, c2, ...])
        n ← 0
        for all counter c ∈ list[c1, c2, ...] do
            n ← n + c
        EMIT(i, n)
end class
```

After this first phase we obtain the number of films per rating, then we use it with p to evaluate the m and k parameters.

3.2 Job 2: Bloom Filters creation

The second Job builds the Bloom Filters.

First, the In-Mapper Combiner creates a Bloom Filter for each rounded rating, it processes each film extracting its rounded rating and it adds such film to the corresponding Bloom Filter. The *add*() function sets the bits inside the Bloom Filter that correspond to the images of the k hash functions.

Algorithm 3 In-Mapper Combiner: Bloom Filters creation

```
class Mapper
    method Initialize(context ctx)
        bloomFilterArray ← getBloomFilters(ctx)
    method Map(key k, value movieRow)
        roundedRating ←round(movieRow.rating)
        bloomFilterArray[roundedRating - 1].add(movieRow.movieId)
    method Close
        i ← 1
        for all BloomFilter bf ∈ bloomFilterArray do
            Emit(i, bf)
end class
```

The Reducer aggregates, with a bit wise OR, the results of the In-Mapper Combiners returning a final Bloom Filter for each rating.

Algorithm 4 Reducer: Bloom Filters creation

```
class Reducer
    method Reduce(key rating, list bloomFilters)
        newBloomFilter ← null
        for all BloomFilter bf ∈ bloomFilters do
            if newBloomFilter == null then
                newBloomFilter ← bf
            newBloomFilter ← newBloomFilter OR bf
        Emit(rating, newBloomFilter)
end class
```

3.3 Job 3: Bloom Filters test

The third Job tests the Bloom Filters obtained in the previous phases.

The In-Mapper Combiner retrieves the Bloom Filters previously built, it processes the dataset row obtaining the rounded rating, it tests all the Bloom Filters over the $movieID$ and increases the right counter of the statistics. After the processing of all films, it emits the local results that includes the rounded rating and the statistics of the latter.

Algorithm 5 In-Mapper Combiner: Bloom Filters test

```
class Mapper
    method Initialize(context ctx)
        counterFP ← new Array
        counterFN ← new Array
        counterTP ← new Array
        counterTN ← new Array
        bloomFilterArray ← ctx.getBloomFilters()
    method Map(key k, value movieRow)
        roundedRating ← round(movieRow.rating)
        i ← 1
        for all BloomFilter bf ∈ bloomFilterArray do
            if bf.contains(movieRow.movieId) then
                if i == roundedRating then
                    counterTP[i − 1] ← counterTP[i − 1] + 1
                else
                    counterFP[i − 1] ← counterFP[i − 1] + 1
            else
                if i <> roundedRating then
                    counterTN[i − 1] ← counterTN[i − 1] + 1
                else
                    counterFN[i − 1] ← counterFN[i − 1] + 1
            i ← i + 1
    method Close
        i ← 1
        while i <= 10 do
            Emit(i, (counterFP[i-1], counterFN[i-1], counterTP[i-1], counterTN[i-1]))
            i ← i + 1
end class
```

The Reducer then aggregates the results of different In-Mapper Combiners emitting the rating and the cumulative statistics for it.

Algorithm 6 Reducer Combiner: Bloom Filters test

```
class Reducer
    method Reduce(key rating, list counters)
        counterFP ← 0
        counterFN ← 0
        counterTP ← 0
        counterTN ← 0
        for all counter ∈ counters do
            counterFN ← counterFN + counter[0]
            counterFP ← counterFP + counter[1]
            counterTN ← counterTN + counter[2]
            counterTP ← counterTP + counter[3]
        Emit(rating, (counterFP, counterFN, counterTP, counterTN))
end class
```

4 Hadoop implementation and experimental results

4.1 Configuration

Hadoop cluster was composed by a total of four nodes: a NAMENODE that acts also as DATANODE, and three other DATANODES as described in the table below.

Hostname	IP Address	Namenode	Datanode
hadoop-namenode	172.16.4.188	✓	✓
hadoop-datanode-2	172.16.4.167		✓
hadoop-datanode-3	172.16.4.178		✓
hadoop-datanode-4	172.16.4.157		✓

4.2 Job 1 Films per rating count

We propose two different solution for the JOB 1 that is responsible for counting the number of films per rating: the first one exploits a classical MAPREDUCE approach, the second one uses IN-MAPPER COMBINER design pattern.

4.2.1 MapReduce Classical approach

This solution consists in a MAPPER and a REDUCER:

- MAPPER: the map function is responsible for emitting a Key-Value pair that contains the rating and a counter equals to 1 that corresponds to the processed film,
- REDUCER: the reduce function is responsible for aggregating the films per rating counters received by the combiners.

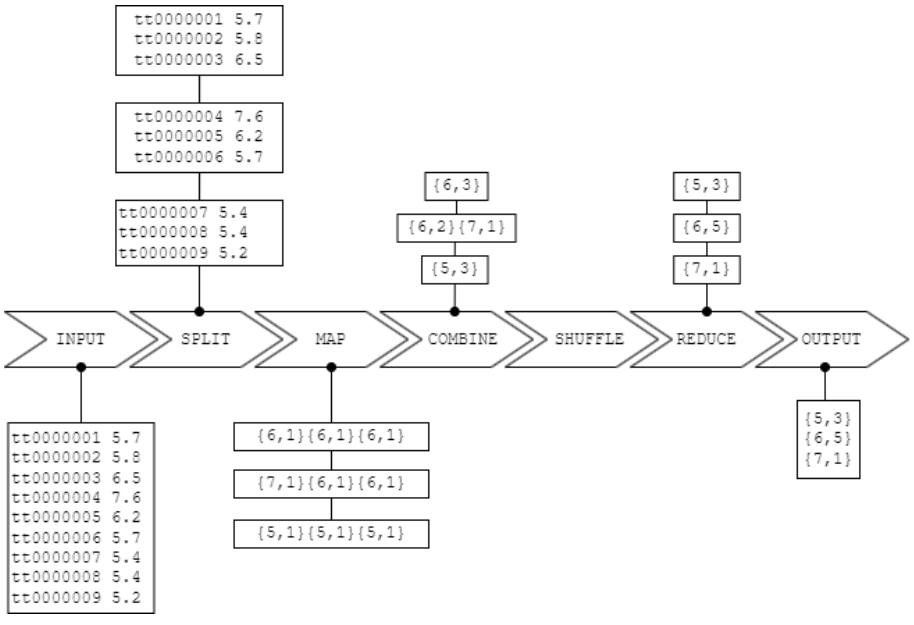

Figure 4: JOB 1 Classical approach

4.2.2 MapReduce with In-Mapper Combiner

This second solution uses an In-Mapper Combiner and a Reducer:

- In-Mapper Combiner: the map function increases the counter that corresponds to a certain rating, as the counters are saved in memory, after the processing of all films, we have the local cumulative films per rating that are then emitted,
- Reducer: the reduce function combines the data from In-Mapper combiners and emits e Key-Value pair with rounded rating and total number of films for that rating.

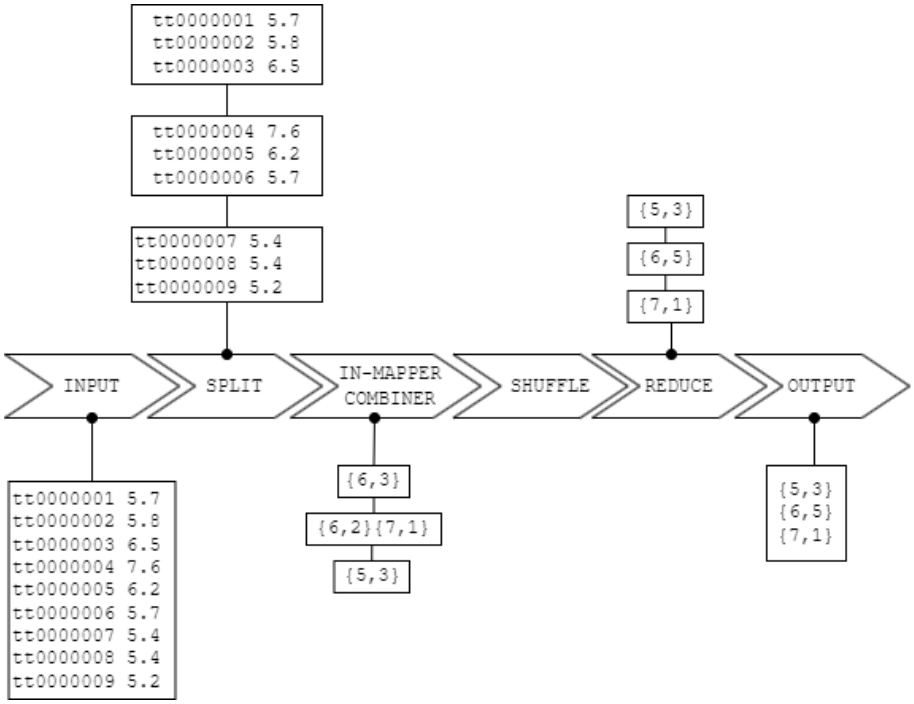

Figure 5: Job 1 Optimized approach

We expect that both implementation have the same efficiency, in fact they use similar data structures during evaluation.

4.3 Job 2 Bloom Filters creation

We propose three different solution for the Job 2 that is responsible for the Bloom Filters creation: the first and second exploits a classical MapReduce approach, the third uses instead the In-Mapper Combiner design pattern.

4.3.1 MapReduce Classical approach

The first two solutions consists in a Mapper and a Reducer:

- Mapper: the map function return a Key-Value pair with the rating and a Bloom Filter with particular bits set, these bits are the ones set by the k hash functions applied over the given *movieID*,
- Reducer: the reduce function aggregates the results from the combiners, executing the bit-wise OR between filters belonging to the same rating.

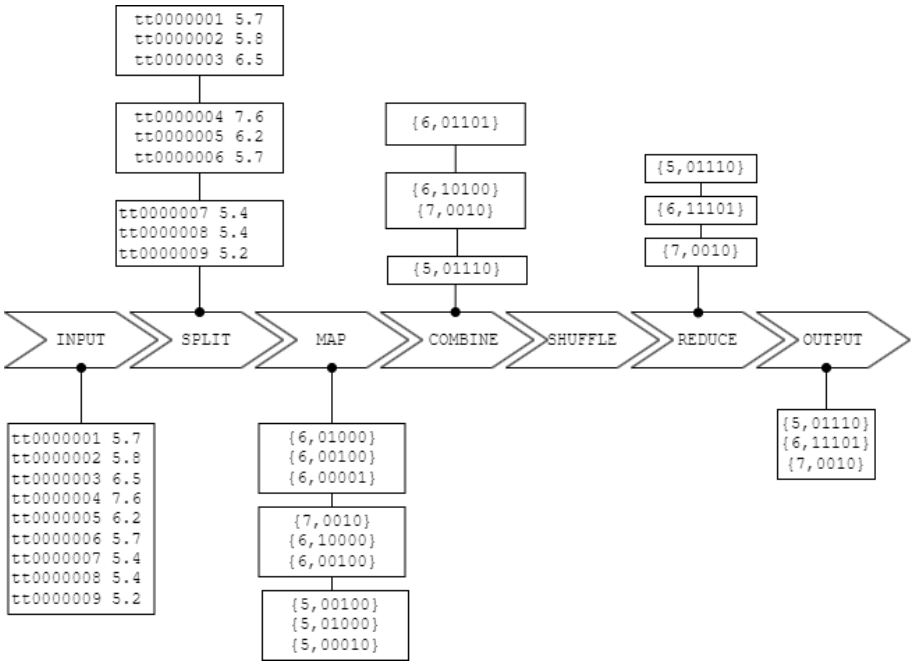

Figure 6: Job 2 Classical approach

4.3.2 Optimized Mapper

The second solution is similar to the previous one, but this time the map function returns a *Bitposition* variable, that contains the indexes of the bits that were set inside the Bloom Filter.

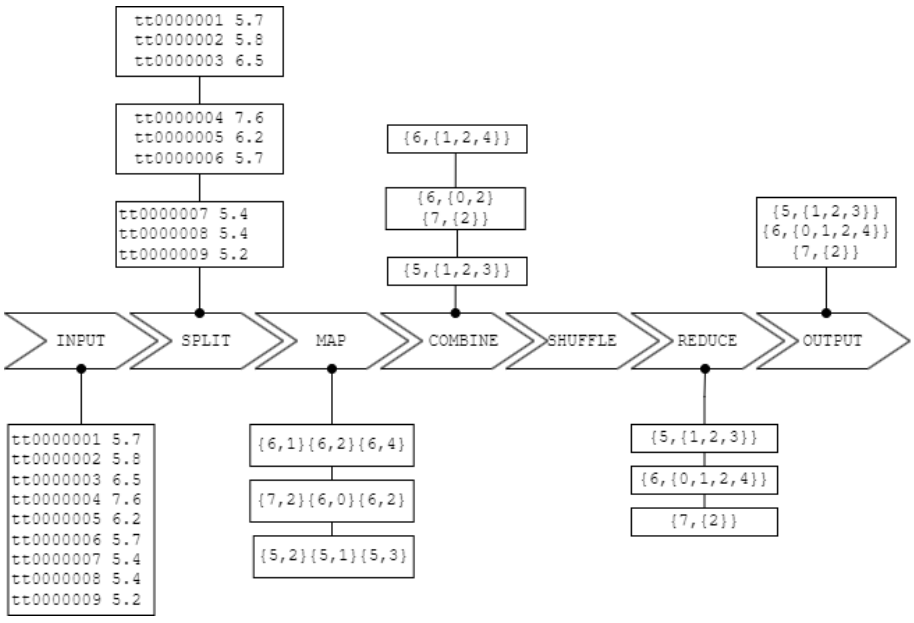

Figure 7: Job 2 Classical approach with Bitposition (Optimized)

We expect that this second solution is more efficient, in fact, considering high sized Bloom Filters, the amount of set bits is low, that means is better to pass the indexes of bits instead of the entire Bloom Filter.

In fact, as we can observe from the following table, calculating the values of m and k from P and n with the optimized formulae, we have that k tends to be $\ll$ m, and from that we have a significant reduction in the amount of data to be transferred from the Map phase.

P	rating	n	k	m
0.000001	1	2584	19	74303
0.000001	2	6752	19	194154
0.000001	3	18576	19	534156
0.000001	4	45072	19	1296053
0.000001	5	103731	19	2982803
0.000001	6	223904	19	6438398
0.000001	7	377578	19	10857321
0.000001	8	349048	19	10036936
0.000001	9	112605	19	3237976
0.000001	10	16957	19	487601

Table 1: some values of m and k for given P and n values

4.3.3 MapReduce with In-Mapper Combiner

In a third implementation we exploit the In-Mapper Combiner design pattern, the solution uses an in In-Mapper Combiner and a Reducer:

- In-Mapper Combiner: the map function adds each received film to the corresponding Bloom Filter, using its rounded rating to choose. The *add*() function sets the bits inside the Bloom Filter that correspond to the images of the k hash functions. As the Bloom Filters are saved into memory, after the processing of all films, we have the local cumulative Bloom Filter for each rating,
- Reducer: the reduce function aggregates by key the outputs of the In-Mapper Combiners executing the bit-wise OR of the Bloom Filters received that belongs to the same rating.

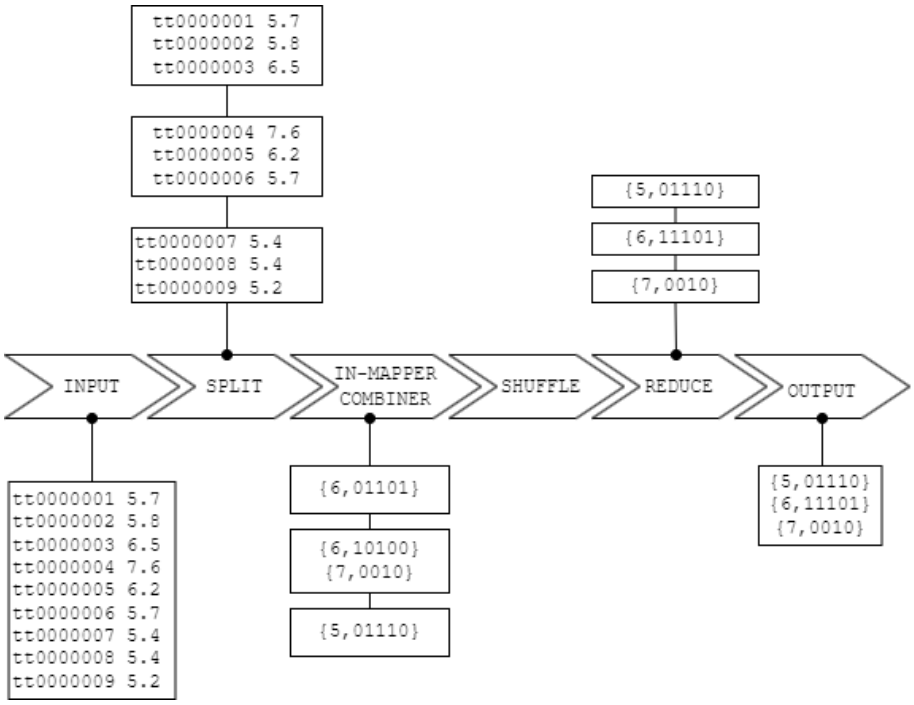

Figure 8: Job 2 In-Mapper combiner approach

We expect that this second approach is faster and more efficient than the Classical approach without *Bitposition*, because we use a single Bloom Filter object for each Mapper, reducing the amount of memory usage.

4.4 Performances and results

In the following section we analyze the performances of Job 2, that is responsible for the creation of the Bloom Filters and it is also crucial for our application. Our work consists in testing the performances in execution time and memory usage for the various implementation of Job 2.

In the following graph, we have a comparison between execution times of the different implementations varying the p value. We can easily understand that implementations with Optimized Mapper and In-Mapper Combiner outperform the first implementation of the Mapper (without *Bitposition*), and also that the latter significantly increases execution time when p becomes lower.

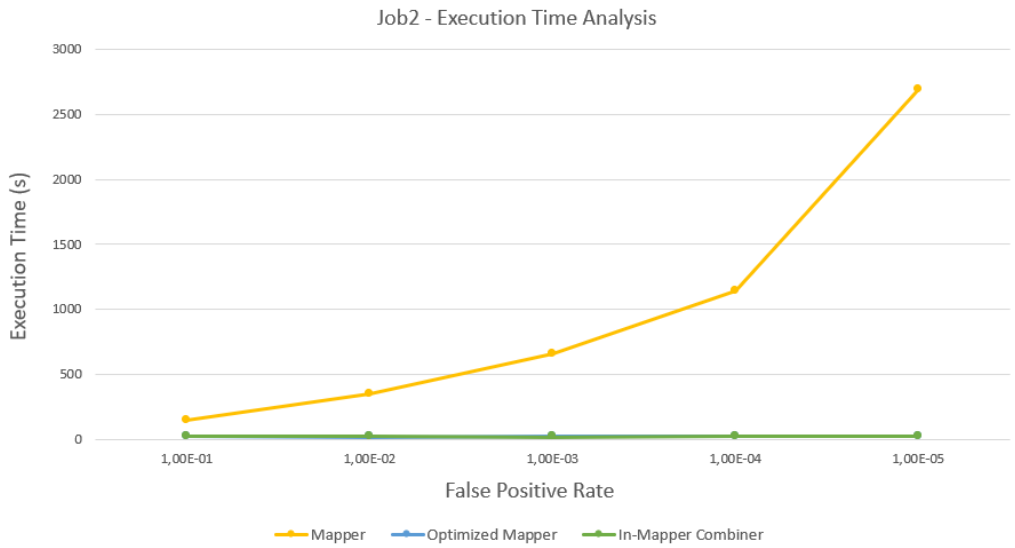

Figure 9: Job 2 execution time analysis

Talking about the Physical Memory usage, the solution with In-Mapper Combiner is preferred as it needs less memory for execution, probably because it instantiates only one Bloom Filter object per rating for each Mapper instead of many. In general we can appreciate an increasing memory usage when p decreases in all implementation, with a significant rise in the Mapper base solution when p is particularly low.

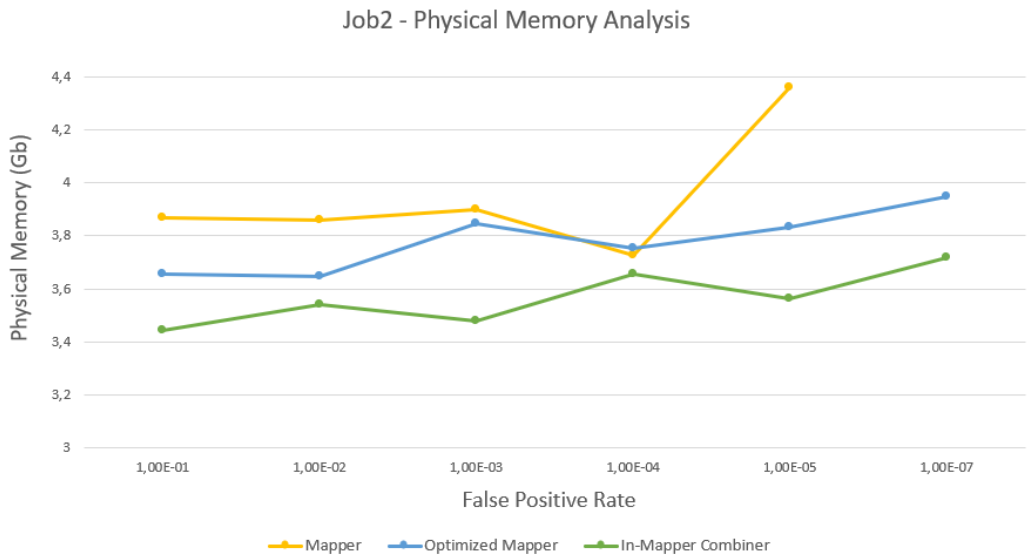

Figure 10: Job 2 physical memory analysis

We notice a similar behavior in the Heap committed usage, the Optimized Mapper solution is comparable with the In-Mapper Combiner solution, but the latter has in general less Heap usage.

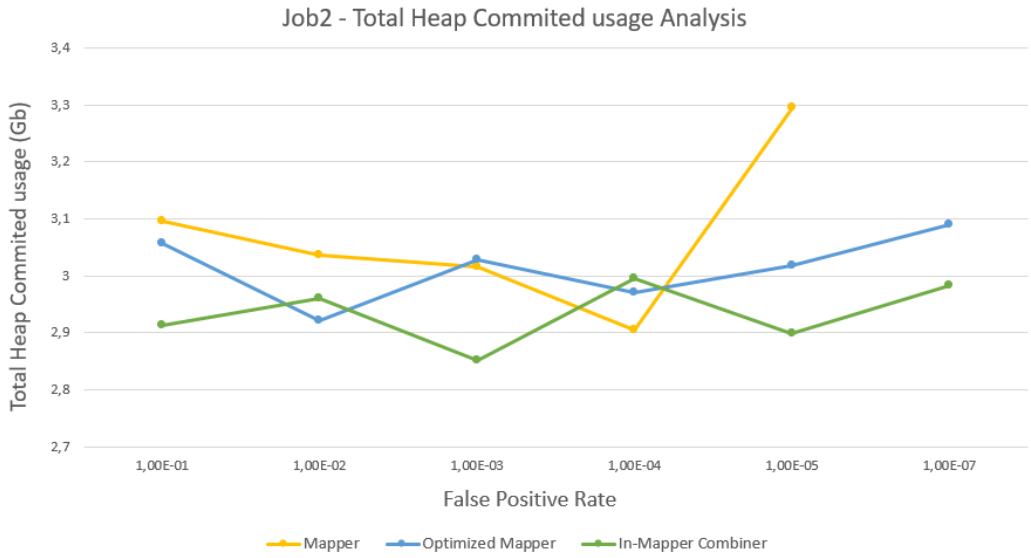

Figure 11: Job 2 heap committed usage analysis

5 Spark implementation and experimental results

5.1 Configuration

We deployed our Spark application in cluster-YARN mode using the same four nodes of Hadoop cluster.

For the test of the performances of our implementations we set the total number of executors in the cluster equal to 12 and the number of cores for each executor to 4.

For the hash functions of our Bloom Filters we used the package *mmh3*, which contains a set of fast and robust non-cryptographic hash functions.

5.2 Broadcast Variables

In order to achieve better performances we decided to use broadcast variables for m, k and the Bloom filters. In particular, the latter is used by the job 3. This choice is due to the fact that they are shared across multiple stages and tasks, so it could be useful to ship those values only once to the working nodes to reduce communication costs.

5.3 RDD repartitioning

We noticed that by default the RDD created from Spark from the input file was split in two partitions.

Since our aim was to distribute the load and the data among the cluster as much as possible, we exploited the method $RDD.repartition(num)$ to repartition the Resilient Distributed Dataset in a higher number of partitions.

In particular for the reported time results we splitted the RDD in a number of partition equal to the number of executors in the first phase, for the last plot the number of partition is equal to *cores per executor* times *number of executors*.

5.4 RDD preprocessing

We have read the dataset file from HDFS using the "pyspark.SparkContext.textFile(path)" function, which returns it as an RDD of Strings. Each string represents a row of the dataset (e.g. 'tt0000001 5.7 1905'). After that we transformed the rows so that each of them was in the form <key, value>, with the movie's rounded rating as key and the movie id of the same row as value.

5.5 Job 1: films per rating count

We deployed one simple solution that uses *countByKey* Action for counting the number of films for each rounded rating. The returned type is a dictionary with its elements in the form (rounded rating, number of films for that rating). We report the code implemented:

```
1 def job1(ratings : RDD) -> dict:
2     return ratings.countByKey()
```

5.6 Job 2: Bloom Filters creation

For job2 we proposed 3 different solutions in Spark, below are reported the details.

5.6.1 solution 1: "Classic Map Reduce" approach

```
1  def job2_base(ratings : RDD, M : Broadcast, K : Broadcast) -> list:
2      def map(tup : tuple) -> tuple:
3          (in_key, movie_id) = tup
4          key = int(in_key)
5          return ( key, initializeBloomFilter(movie_id,
6                          M.value[key-1], K.value[key-1]) )
7
8      def reduce(bf1 : bitarray, bf2 : bitarray) -> bitarray:
9          return bf1.__or__(bf2)
10
11     return ratings.map(map).reduceByKey(reduce).collect()
```

The code shows how the classic MapReduce solution was implemented, *ratings* is an RDD of tuples in the form (<rounded rating, movie id>). In this solution, at row 11 in the code reported, the *map* transformation creates a new Bloom Filter (a bitset) for each movie in *ratings*, while the *reduceByKey* transformation combines filters of the same rating using the OR logic operation. Lastly the *collect()* action returns them.

The problem with this solution is that it may introduce a lot of overhead in the communication to the reducer. In fact bloom filters are potentially huge bitsets, and we send a new bloom filter for each movie!

5.6.2 solution 2: "GroupByKey" approach

```
1  def job2_groupByKey(ratings : RDD, M : Broadcast, K : Broadcast) -> list:
2
3      def fun(tuple : tuple) -> tuple:
4          # input tuple in the form
5          # <key: int, list_of_movies: Iterable(str)>
6          in_key, list_of_movies = tuple
7          key = int(in_key)
8          bs = bitarray(M.value[key - 1])
9          bs.setall(0)
10         for movie_id in list_of_movies:
11             bs = addToFilter(bs, movie_id, M.value[key - 1],
12                     K.value[key - 1])
13         return (key, bs)
14     return ratings.groupByKey().map(fun).collect()
```

This solution is implemented in three phases. In the first phase (*ratings.groupByKey()* called in the last row of the code) RDD records are grouped so that all the rows of the dataset containing movies with the same rounded rating (our key) are processed by the same Spark Executor.

This phase can be very expensive in terms of network traffic and leads to tuples in the

form <key , list of values>, where, in our case, the key is still a rounded rating and the list of values is the list of all the movie ids with that rounded rating.

The second step (*map(fun)* called in the last row of the code after *groupByKey()*) consists of applying a mapping function that iterates over the movie_ids and adds each movie_id to the Bloom filter for that specific rate.

The third phase is directly the *collect()* Action, which retrieves bitsets from the cluster nodes towards the main Spark process in the master node.

A reduce phase isn't necessary since each node has already all the tuples related to a key.

A drawback of this solution is the possibility of having an imbalance workload among different Spark Executors if some keys contain more values than others. This is precisely our case, as we have seen in the Dataset distribution plot in the introduction of this report, some ratings contain a much higher number of movies than others.
On the other hand, the advantage of this solution is that it minimizes the overhead of passing filters between nodes (the reduce phase doesn't exist at all).

This approach works very well on small datasets or on datasets with many different keys, each with few elements. In this way, every executor has a reasonable amount of records to process and can store their partition in its Executor Memory, since it has to iterate over the whole partition in a *for* statement.

5.6.3 solution 3: "AggregateByKey" approach

```
def job2_aggregateByKey(ratingsKKV: RDD, M : Broadcast, K : Broadcast) -> list:
    #each rating is in the form <roundedRating, movieID>

    def seqFunc(bs : bitarray, row2 : tuple) -> bitarray:
        key = row2[0]
        if( not bs ): # zeroValue case
            return initializeBloomFilter(row2[1],
                M.value[key - 1], K.value[key - 1])
        return addToFilter(bs, row2[1], M.value[key - 1], K.value[key - 1])

    def combFunc(bitset1 : bitarray, bitset2 : bitarray) -> bitarray:
        if( not bitset1):
            return bitset2
        if not bitset2:
            return bitset1
        return bitset1.__or__(bitset2)

    bitsets = ratingsKKV.aggregateByKey(
        zeroValue=bitarray(), seqFunc=seqFunc, combFunc=combFunc)
    return bitsets.collect()
```

The difference of this solution from the previous one is that executors process movies that can have different rounded rating values.
The first phase consists in aggregating, within each executor, rows related to the same

rounded rating using the *SeqFun* function. This function takes in input a bitarray and the value of a new tuple.

At the end of this phase each executor will create new bloom filters from their local RDD.

In the picture below it's reported a schema of this phase for one value of the key. Note that, on each Executor, partitions of rows in which there are different keys could be processed, unlike in the *GroupByKey* solution. The data flow is the same for each key.

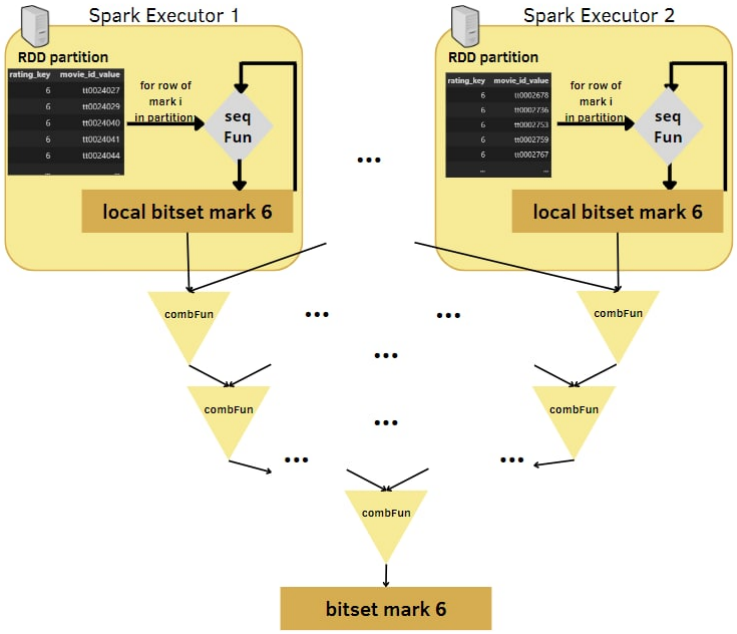

Figure 12: AggregateByKey data flow for the key *6*

At this point, the local bitarray results are aggregated using the *combFun* function, which merges the Bloom filters from different executors with an OR logic operation. Finally, the collect() action retrieves the result from the RDD and returns it to the main Spark process in the master node.

We can observe that this approach has both the two previous solutions' advantages. As we can see from the scheme of *seqFun*, it doesn't create a different bitarray for each row (as in the classic MapReduce approach) nor does it have to send rows containing movies with different rounded rating to different Executors (as in GroupByKey solution). The load related to each rating is distributed among nodes without the risk of overworking some Executors, forcing them to process a huge amount of data in a very sequential and very little distributed way.

This approach is the one that is best suited to work with considerably high amount of data. It's scalable and guarantees that the workload will be distributed in a (fairly) uniform way among the various nodes (the way the RDD was originally distributed by Spark is respected).

5.7 Job 3: Bloom filters test

This job aims to compute for each filter the number of true positives, true negatives, false positives and false negatives testing all the movies in the dataset on the 10 constructed filters. We also decided to do two more counts: the number of times at least two filters return a positive answer and the number of movies evaluated. We'll use them for the analysis of different values of p.

Using the formula in the introduction is then possible to compute the false positive rate and compare the performances of the three different solutions proposed for job 2.

```
def job3(ratingsKKV : RDD, bitsets : Broadcast, M : Broadcast,
                    K : Broadcast) -> list:
    FP_OFFSET = 0
    FN_OFFSET = 1
    TP_OFFSET = 2
    TN_OFFSET = 3
    MULTI_POSITIVE_COUNTER_INDEX = 40
    EVALUATED_COUNTER_INDEX = 41

    def seqOp(scores : list, row) -> list:
        positive_results_counter = 0
        key, movie_id = row
        for index, bloom_filter in enumerate(bitsets.value):
            if checkInFilter(bloom_filter, movie_id, M.value[index],
                    K.value[index]):
                positive_results_counter += 1
                if index == (key - 1):    #case True Positive
                    scores[ ( index ) * 4 + TP_OFFSET] += 1
                    valid = True
                else:                             #case False Positive
                    scores[ (index) * 4 + FP_OFFSET] += 1
            else:
                if index == (key - 1):    #case False Negative
                    scores[ ( index ) * 4 + FN_OFFSET] += 1
                else:                             #case True Negative
                    scores[( index ) * 4 + TN_OFFSET] += 1

        if positive_results_counter > 1:
            #more than one Bloom filter
            #returned positive result
            scores[MULTI_POSITIVE_COUNTER_INDEX] += 1
        scores[EVALUATED_COUNTER_INDEX] += 1
        return scores

    def combOp(scores1: list, scores2: list) -> list:
        return [sum(x) for x in zip(scores1, scores2)]

    return ratingsKKV.aggregate(
        zeroValue=( [0] * 42), seqOp=seqOp, combOp=combOp)
```

This solution uses the *aggregate* Action (row 38 in code snippet) which is the same as *AggregateByKey* except for the latter operates on Pair RDD (key, value) and separates the rows on the key, but for this job there is no need to do that. In the first phase of this solution the SeqFun function counts, within each Executor, the number of true/false positives and negatives of their local RDD, and saves them in a matrix (*scores*). In the second phase the combOp function sum the counts obtained from different Executors.

5.8 Performances and results

5.8.1 Job 2 different solutions performances

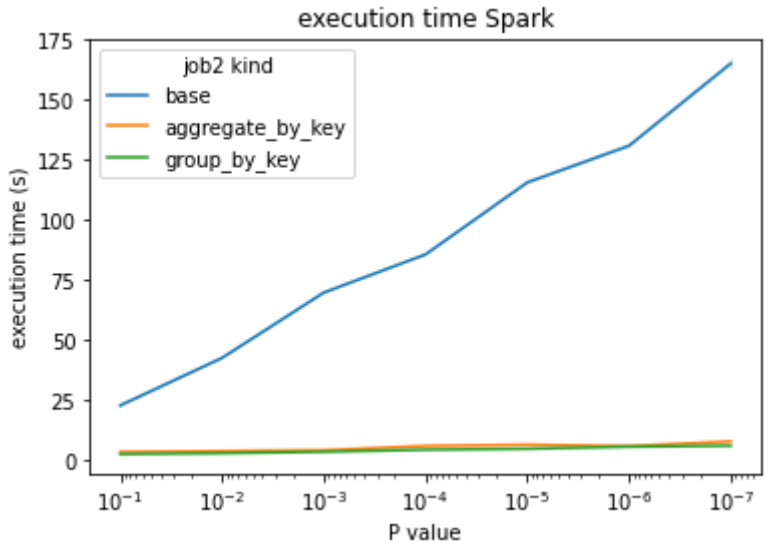

Figure 13: Comparison of the three proposed solutions for job 2 in Spark

This plot shows the execution times of the three solutions previously proposed for the Bloom Filters generation with different p-values. We can observe how the classic MapReduce approach (the blue line) is always slower than the other two approaches, in fact the blue line is higher than the green and orange line for all values of p. Furthermore its execution time increases linearly with $-log_{10}(P)$. The reason is probably that the bitsets of the Bloom Filters become larger (fixed n, the parameter m increases linearly with $-log_{10}(P)$), and the classic *MapReduce* approach creates a new filter for each movie id. This enormous amount of data generated may lead to filling the RAM memory and slowing down the application (for example with swaps from memory to disk).

On the other hand the *AggregateByKey* and *GroupByKey* solutions' execution times don't seem to be particularly affected by the change of p, their network traffic is much smaller than in the first solution.

The two lines are similar but in this case *GroupByKey* performs a little better than *AggregateByKey*. This result could be attributed to the fact our dataset is quite small and every Executor can easily process all the movies of a certain key without the need of splitting the rows between different Executors.

On the other hand we can observe that the *AggregateByKey* approach introduces some overhead caused by the necessity of combining results from different Executors on the

same key, this is why for this dataset *GroupByKey* is the best performing solution.

5.8.2 Number of Executors, Number of cores per Executor and Number of partitions

Until now all Spark performances reported were found considering:

- Number of Executors = 12
- Number of Partitions = Number of Executors = 12
- Number of Cores per Executor = 4

This means that we have 12 x 4 = 48 threads running Spark tasks simultaneously. However this configuration is not the one that optimizes best our application. In the following plots we show how different values of these three parameters can influence the total execution times of all the jobs together.

Let's start by *setting the number of partitions equal to the number of executors*:

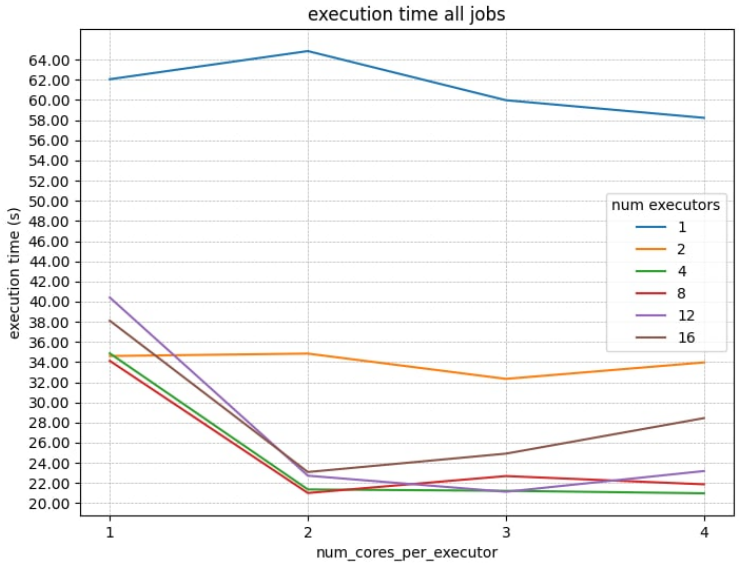

Figure 14: Execution times of all jobs with different executor configurations

Looking at the plot we can see that for a low number of executors (1 or 2) our application is very slow, with high execution times of all jobs. With the increase of executors the performances improve considerably and we reach the best performances, depending on cores, with 4 or 6 executors. However after that an even higher number of executors (12 or 16) decreases the performances.

Considering now the number of cores we can notice that having just 1 core per executor is generally a lot worse than having at least 2 or more cores per executor. Only with 1 or 2 executors (blue and orange lines) the execution times are almost constant with all the number of cores considered, but they are still too high and not satisfactory. On the other hand with 4 or more executors it's clear that having at least 2 cores per executor increase significantly the performances.

In the plot the best performing configuration is the minimum point of these lines, in which we have the lowest execution time of all jobs. We can see that it coincides with having 4 executors (green line) and 4 cores per executor, so 4 x 4 = 16 threads running Spark tasks simultaneously. The number of partitions is also 4.

Next let's set the number of partitions equal to the product of the number of executors with the number of cores per executor. The corresponding plot obtained is:

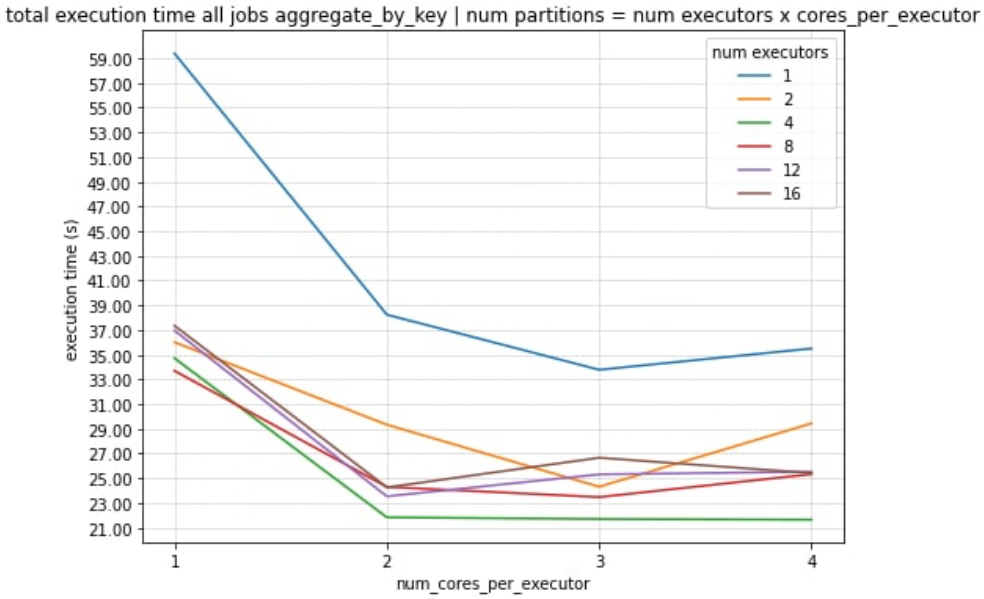

Figure 15: Execution times of all jobs with different executor configurations

Considering the number of executors we can see that the considerations done for the previous plot are valid even in this case. In fact the green and red lines are generally lower than the other lines, the best performances are still obtained using 4 or 8 executors depending on the number of cores.

Looking at the number of cores per executor the situation is similar to the one in the previous plot but in this case the blue and orange lines decrease when they come to have from 1 to at least 2 cores per executor. It seems that the increasing of the number of partitions (we have from 1 to 8 partitions in this two lines, while in the previous graph were 1 or 2) help in case of 1 or 2 executors to get better results.

The best configurations for this number of partitions are the ones having 4 executors and 2-4 cores per executor, so 8 to 16 threads running Spark tasks simultaneously. Also the number of partitions will vary from 8 to 16.

The execution times in the most optimized configurations don't differ much from the two plots and both of them are approximately 21 seconds, despite we changed the number of partitions.

6 Conclusions

It is important to note, however, that all tests were performed using a small dataset (about 20Mb). This makes it difficult to appreciate the real differences between the various proposed implementations and how they can scale. Although, among the various solutions, we were able to identify those that performed best in terms of space and execution time.

6.1 Hadoop considerations

As stated in *Chapter 2*, when p decreases the memory required for the Bloom filters will increase as well. This has consequences for the three different implementations made on Hadoop. In the following paragraphs will be given possible explanations of why these differences between them, analyzing their behavior when:

- p is fixed
- p changes

6.1.1 Implementations performances at fixed P

The first analysis consists in choosing a specific p and seeing the performance of different implementations. We can see a big difference in terms of execution time between the MapReduce classic approach and the other two implementations since they have the same execution time more or less. This behavior seems to be obvious because the classic version of map reduce will generate high intermediate data traffic (one bloom filter object per record). Despite the presence of the combiner, before the data produced by the mapper can be locally aggregated, there will be an high number of writes. When there is not enough memory, **spilling to disk** may occur which significantly slows the execution of the job.
In the in-mapper combiner implementation this does not happen probably because even if the size of the bloom filters grows up, the entire array of filters can be saved in memory or, in the worst case, the number of times spilling to disk occurs is very low compared to the previous implementation.

Regarding the differences in terms of memory, a consequence of what was just said is that in the classic map reduce approach we will obviously need more physical memory in order to store a bloom filter per record, whereas the in-memory mapper combiner approach is the best one since each mapper instantiates 10 bloom filters per input splits, saving a lot of memory.

6.1.2 Performances variations changing P

This second analysis consists in seeing the behavior of each implementation as p changes. The classic map reduce implementation is the one most affected while in the other cases the increase of the execution time and the required memory seems to be approximately constant.

In the classic map reduce approach is obvious that the decrease of p will cause an increase of m. This means that the bloom filter objects will occupy more and more memory, deteriorating the performances.

In the second implementation which emits the bit positions, the element to be taken into consideration is k. In fact, in this case the mapper emits vector of size k for each record which does not grow much with decreasing p, so the amount of the intermediate data does not vary much. In the following table it is possible to see how k varies with the different p values:

P value	k
10^{-1}	2
10^{-2}	6
10^{-3}	9
10^{-4}	13
10^{-5}	15
10^{-6}	19
10^{-7}	22

Table 2: The value of k should change as the rating changes. This is due to the fact that n changes but in our case k remains the same for all the ratings.

Finally, in the in-mapper combiner approach, the element to take into consideration is the vector of bloom filters which size increases when p decreases. That increase will begin to be significant as p decreases. But despite this, the memory continues to have a constant trend. This could be due to the use of serialization methods exploited by Hadoop.

6.2 Spark implementation performances

We propose three different versions of the Job2 in Spark.
The classic MapReduce approach is in general slower than the others, because it creates a new Bloom Filter for every movieID and the huge amount of data generated from these operations can cause swaps from disk to the main memory slowing down the application.
Considering the *GroupByKey* and *AggregateByKey* solutions, we can say that both have similar execution performances even if the first has a slightly lower execution time on our dataset.
We can notice that the first one is less scalable, in fact in this implementation some executors could receive more data to process than others, but using our dataset we don't notice any important decrease in performances.
The second one, on the other hand, distributes better the work load among the executors, but it also introduce some overhead to aggregate data from different executors, in fact, using our dataset, it is slightly less efficient than the *GroupByKey* solution.

We also find out that the optimal number of partitions and executors of the Spark application depends on the available nodes in the cluster and the dataset we use. First we fix the number of partitions equals to the number of executors and then to the product of the number of executors and the number of cores, we find out that in general best performances are reached with 4 executors and 4 cores per executors and also that when we set

the number of executors to 12 or 16 we reach worse performances, because we increase the amount of data exchanged between the executors and because the actually available resources force some executors tasks to be enqueued, taking to a sequential processing that does not improve the parallelization of the jobs.

6.3 P values considerations

From the experimental results obtained with our implementations, we can observe the trend of the *Multipositive counter*, reported in the following plot with a logarithmic scale for the P value:

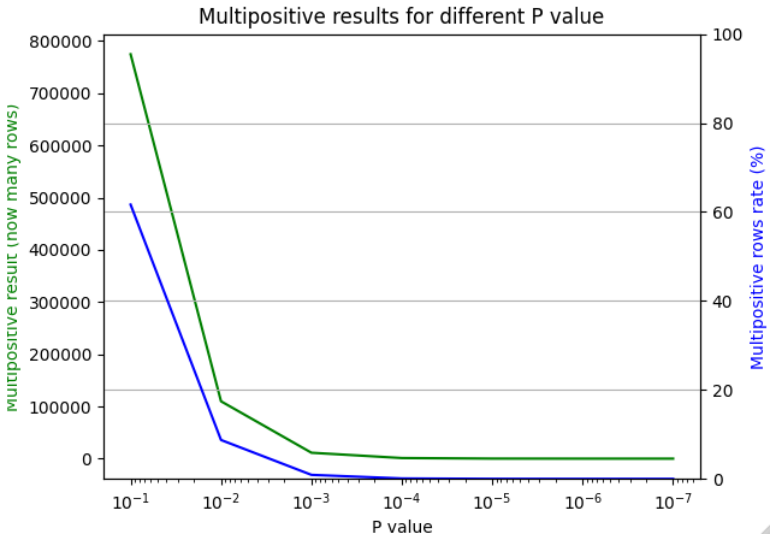

Figure 16: Multipositive records at different P values

While in the following we have the table of the MultiPositive results per P value:

P_value	multipositive_counter	MPR
10^{-1}	774845	61.6519%
10^{-2}	109907	8.7449%
10^{-3}	11331	0.9016%
10^{-4}	1165	0.0927%
10^{-5}	114	0.0091%
10^{-6}	8	0.0006%
10^{-7}	1	0.0001%

We can observe that for a P value of each Bloom filter of 0.1, we have a global *MPR* over 61%, that is not acceptable.

That rate decreases while increasing (in absolute terms) the negative exponent of 10 for the value of P.

We can observe that the rate becomes acceptable for P = 10^{-3}, but there is still a good improvement for 10^{-4}, 10^{-5} and 10^{-6} in fact it decreases for a 10 factor, becoming about 10 times the P value of each Bloom filter itself.

We can conclude that a good value of P for the given dataset considering the trade between performances and False positives, is P = 10^{-4}.

6.4 False Positive values experimental results

We report the false positive rates and other statistics obtained in Hadoop and Spark.

To verify if our filters are constructed correctly we test the false positive rate, computed at the end of job 3, with the p value set before the start of all the jobs. We expecte that these two values must almost coincide to verify the correctness of the procedures.

Hadoop results using P = 10^{-4} and P = 10^{-7}:

rating	FP	FN	TP	TN	FPR
1	117	0	2584	1254106	0.0000932848
2	124	0	6752	1249931	0.00009919563
3	133	0	18576	1238098	0.0001074112
4	145	0	45072	1211590	0.0001196631
5	127	0	103731	1152949	0.0001101401
6	114	0	223904	1032789	0.0001103685
7	101	0	377578	879128	0.0001148733
8	94	0	349048	907665	0.0001035517
9	99	0	112605	1144103	0.00008652318
10	141	0	16957	1239709	0.0001137234

Table 3: Hadoop False Positive using $P = 10^{-4}$

rating	FP	FN	TP	TN	FPR
1	0	0	2584	1254223	0.0000000000
2	1	0	6752	1250054	0.0000007999
3	0	0	18576	1238231	0.0000000000
4	1	0	45072	1211734	0.0000008252
5	0	0	103731	1153076	0.0000000000
6	0	0	223904	1032903	0.0000000000
7	0	0	377578	879229	0.0000000000
8	0	0	349048	907759	0.0000000000
9	1	0	112605	1144201	0.00000087397
10	0	0	16957	1239850	0.0000000000

Table 4: Hadoop False Positive using $P = 10^{-7}$

From these two tables above we can notice that the FN column values are always 0, and this is right since bloom filters don't admit false negatives. Another observation is that with the decrease of the P value from 10^{-4} to 10^{-7} comes the decrease of false positives, (only 0 and 1's in the second table!), and since FPR depends on FP (and TPs and TNs don't change) it has decreased as well.

Confronting P and FPRs the first table shows clearly how they are almost the same, the absolute difference FPR - P, with $P = 10^{-4}$, is always less than 1.3 x 10^{-5} in all bloom filters.

On the other hand in the second table we expect that many filters have 0 or 1 false positive. In each filter having $P = 10^{-7}$ means that it's likely to have 1 error (a false positive) out of 10^7 tests of actual negatives. However our dataset's size is approximately 10^6, so many filters shouldn't give any wrong answer. And this is exactly what has happened in our experiments: 7 bloom filters out of 10 have 0 FPs (therefore also their FPR is 0), and the other 3 filters have only 1 false positive!

Spark results using P = 10^{-4} and P = 10^{-7}:

rating	FP	FN	TP	TN	FPR
1	114	0	2584	1254109	0.0000908929
2	111	0	6752	1249944	0.0000887961
3	133	0	18576	1238098	0.0001074113
4	137	0	45072	1211598	0.0001130610
5	132	0	103731	1152944	0.0001144764
6	96	0	223904	1032807	0.0000929419
7	110	0	377578	879119	0.0001251096
8	87	0	349048	907672	0.0000958404
9	115	0	112605	1144087	0.0001005067
10	130	0	16957	1239720	0.0001048514

Table 5: Spark False Positive using $P = 10^{-4}$

rating	FP	FN	TP	TN	FPR
1	0	0	2584	1254223	0.0000000000
2	0	0	6752	1250055	0.0000000000
3	0	0	18576	1238231	0.0000000000
4	0	0	45072	1211735	0.0000000000
5	0	0	103731	1153076	0.0000000000
6	0	0	223904	1032903	0.0000000000
7	0	0	377578	879229	0.0000000000
8	1	0	349048	907758	0.0000011016
9	0	0	112605	1144202	0.0000000000
10	0	0	16957	1239850	0.0000000000

Table 6: Spark False Positive using $P = 10^{-7}$

In Spark we have analogous results, in the first Spark table ($P = 10^{-4}$) the absolute difference FPR - P is always less than 1.3 x 10^{-5} in all bloom filters. In the second Spark table we expect, like in Hadoop, many filters not to give any wrong answers, or just very few false positives. From the results we can see that it is what happened, 9 filters out of 10 have 0 FPs and only 1 filter has 1 FPs.

We can also observe that the values in Hadoop and Spark are similar but not the same because the hash functions utilized are different. Another difference is that Hadoop and Spark have a different rounding mode if the value after the decimal is equal to 5, in Python we round it to the nearest even number while in Java always upward. This means we can have a slightly different data distribution (different n parameters) and so a slightly different m and k depending on the Bloom Filter.

In conclusion from all the observations we have made until now we can conclude that our results are coherent with the expected.