Department of Information Engineering
Artificial Intelligence and Data Engineering
Cloud Computing course

Bloom Filter implementation in Map Reduce

E. Ruffoli, F. Hudema, F. Pezzuti, T. Baldi

Academic Year 2021/2022

Contents

1 Introduction

In this project, we present two possible solutions for the implementation of a *Bloom Filter* that we tested and validated using the movie ratings of the official *IMDb dataset*. In detail, we will describe an *Hadoop* solution and a *Spark* solution to the above problem.

A *Bloom Filter* is a space-efficient probabilistic data structure that can be used to test whether or not an object is in a set; the cost to pay is that it is possible to have false positives. The *Bloom Filters* obtained can be used to minimize the response time of a search-movie query and ensuring a low rate of false positives. The two solutions we implemented allow the efficient creation of ten *Bloom Filters*, one for each possible IMDb rating, exploiting distribute processing across a cluster of four virtual machines.

The following documentation is organized as follows: in Section 2 we will describe all the design aspects of the *Bloom Filter* creation algorithm that we used in the two implementations; in Section 3 the *Hadoop* implementation will be presented and in Section 4 it will be presented the implementation of the algorithm using the *Spark* frame-work; in Section 5 the false positive rate of the *Bloom Filters* obtained with the two frame-works are compared with the expected false positive rate values; finally, in Section 6 we will describe some of tests that we performed to evaluate the *Bloom Filters* and the implementations that we have designed.

The code is available in the following repository:

`https://github.com/edoardoruffoli/BloomFilter-MapReduce`

2 Design

In the following section will be analyzed the design choices made both to realize the *Bloom Filter* objects and to implement the MapReduce workflow to create the *Bloom Filters*. The workflow can be divided in three main stages: the first used to calibrate the *Bloom Filters* parameters; the second stage is related to the actual population of the *Bloom Filters*, by inserting each film in the *Bloom Filter* correspondent to its rating; the third one is related to the computation of the false positive rate of each filter. All of the stages can be considered as MapReduce jobs and they will be analyzed in detail in the following sections.

2.1 Bloom Filter Class

Each *Bloom Filter* object must have a set of functions and data structures to implement its typical functionalities, especially the ones needed for the MapReduce workflow.

In detail, the idea to transform the *Bloom Filters* creation problem into a MapReduce problem, is to partition the input dataset among the nodes, each node computes the *Bloom Filters* locally based on the partition that it has been assigned, and, finally, merge the local *Bloom Filters* into the final ones. The key point to parallelize the algorithm is a function that allows to merge different *Bloom Filters* into a single one: the function **Or()**, by performing the *bitwise or* operation on each pair of correspondent bits, does exactly this. Of course, the *Bloom Filter* object also needs to provide the functions to perform the **Insert** operation and the **Find** operation.

Template: Bloom Filter Class

```
1  class BloomFilter
2      method Initialize(n_bits m, n_hash k)
3          bits ← new BitsArray(m)
4      method Insert(docid id)
5          seed ← 0
6          for i ← 1 to k
7              seed ← Hash(id, seed)
8              index ← |seed % m |
9              bits[index] ← 1
10     method Or(BitsArray bits2)
11         for i ← 1 to m
12                 bits[i] ← bits[i] | bits2[i]
13     method Find(docid id)
14         seed ← 0
15         for i ← 1 to k
16             seed ← Hash(id, seed)
17             index ← |seed % m |
18             if bits[index] ≠ 1
19                 return False
20         return True
```

2.2 Parameter Calibration Stage

The aim of the first stage of the workflow is to fine calibrate each *Bloom Filter* parameters by exploiting the following formulas:

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{1}$$

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

They respectively return the number of bits in the bit-vector and the number of hash functions, to obtain a *Bloom Filter* with a false positive rate p. To use both formulas is required the value of n, the number of films that will be inserted in the *Bloom Filter*. So in our case, since we have 10 *Bloom Filter*, one for each rating, we need to compute the total number of films per rating.

We have decided to use a MapReduce approach to perform this count operation due to high volume of the data set. As far as the mapper concerns, we decide to employ the *In-Mapper Combining* pattern, to reduce the number of intermediate keys sent across the cluster by aggregating information at mapper level. Each mapper stores the count of films that it has processed for each rating and, when all the records have been processed, it will emit its

partial results. The Reduce() function takes as input a list of all the partial count of a rating, computes the sum of all the elements of the list and emit the final count result for the rating.

Algorithm 2: Parameter Calibration

```
   Data: Input Data
   Result: Count by Rating
 1 class Mapper
 2     method Initialize(config [path_1, ... path_10])
 3         counter ← new Array
 4     method Map(docid id, doc d)
 5         for all rating r ∈ doc d do
 6             counter_r ← counter_r + 1
 7     method Close()
 8         for all rating r ∈ [1, ... 10] do
 9             EMIT(rating r, counter_r)
10 class Reducer
11     method Reduce(rating r, counter [c_1, c_2, ...])
12         sum ← 0
13         for all counter c ∈ [c_1, c_2, ...] do
14             sum ← sum + c
15         EMIT(rating r, sum)
```

2.3 Bloom Filter Creation Stage

In the second stage, the m and k parameters that have been calibrated for each rating using the results of the first stage, are used to initialize the corresponding *Bloom Filter* object.

In the initialization method of each *mapper*, an array of 10 *Bloom Filter* objects is instantiated using the array of values m and k passed as parameter. The Map() function takes as input a film and insert the film title in the *Bloom Filter* correspondent to its *rating*. Finally, the Close() method emits for each *rating* its *Bloom Filter*. Each *reducer* will receive all the partial *Bloom Filters* associated with a *rating* and emits as output the *bitwise or*, and so the union, of all those *Bloom Filters*.

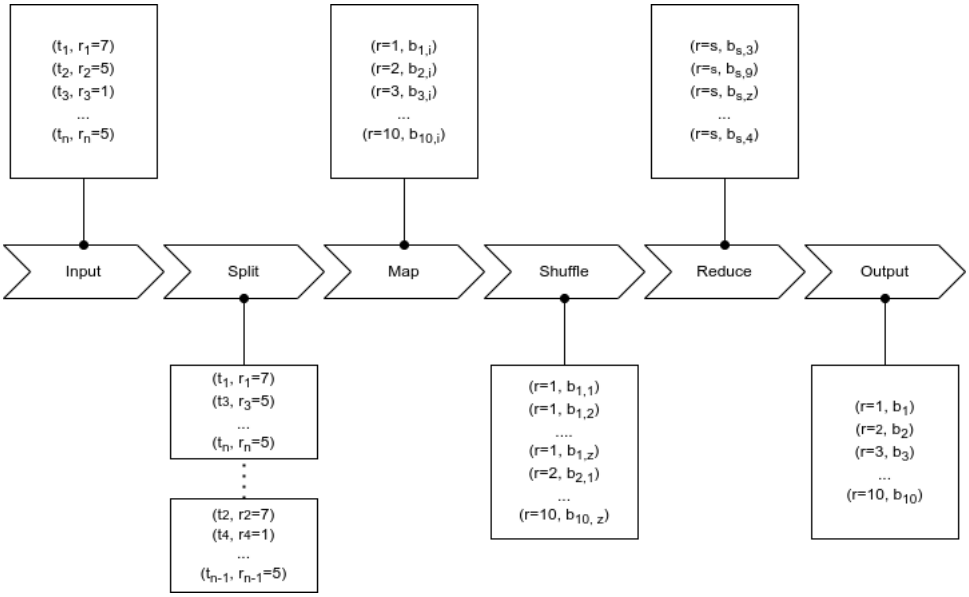

Figure 1: Bloom Filters creation stage

Algorithm 3: Bloom Filter Creation

Data: Input Data, Bloom Filters Configuration Parameters
Result: Array of Bloom Filters

```
1  class Mapper
2      method Initialize(config [c_1, ... c_10])
3          for all rating r ∈ [1, ... 10] do
4              b_r ← new BloomFilter(c_r)
5      method Map(docid id, doc d)
6          r ← d.rating
7          b_r.add(d.filmId)
8      method Close()
9          for all rating r ∈ [1, ... 10] do
10             EMIT(rating r, BloomFilter b_r)
11
12 class Reducer
13     method Reduce(rating r, BloomFilters [b_1, b_2, ...])
14         output ← new BloomFilter(b1.m)
15         for all BloomFilter b ∈ [b_1, b_2, ...] do
16             output.or(b)
17         EMIT(rating r, BloomFilter output)
```

2.4 Parameter Validation Stage

The last stage of the workflow regards the computation of the exact number of false positives for each rating: the *Bloom Filters* obtained in the previous step of the workflow are used, and a membership test is performed with all films that do not have the rating of the selected *Bloom Filter*.

In order to implement the mapper, once again the *In-Mapper Combining* pattern is employed and so an associative array with the counter of the number of false positives for each rating. In the Map() function, each film is membership tested with all the *Bloom Filter* that have been created on other ratings, if it results as a false positive the related counter is incremented by one. The Reduce() simply performs the aggregation sum of all the counters for each rating.

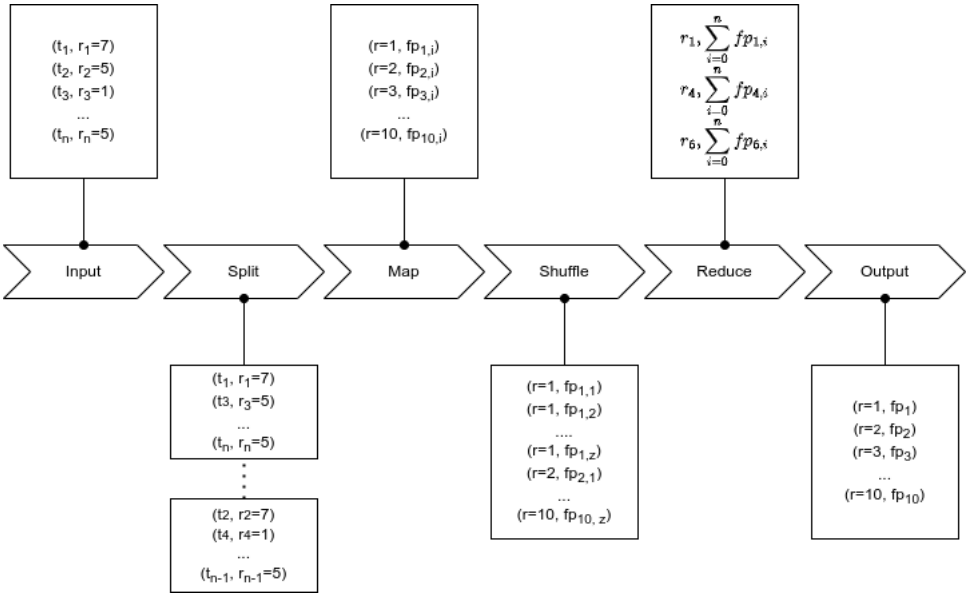

Figure 2: Bloom Filters validation stage

Algorithm 4: Parameter Validation

Data: Input Data, Bloom Filters path
Result: Array of FalsePositive Count

```
1  class Mapper
2      method Initialize(config [path_1, ... path_10])
3          counter ← new Array
4          for all rating r ∈ [1, ... 10] do
5              b_r ← load BloomFilter(path_r)
6      method Map(docid id, doc d)
7          for all rating r ∈ [1, ... 10] do
8              if d.rating = r
9                  continue
10             if b_r.find(d_id)
11                 counter_r ← counter_r + 1
12     method Close()
13         for all rating r ∈ [1, ... 10] do
14             EMIT(rating r, counter_r)
15
16 class Reducer
17     method Reduce(rating r, counter [c_1, c_2, ...])
18         sum ← 0
19         for all counter c ∈ [c_1, c_2, ...] do
20             sum ← sum + c
21         EMIT(rating r, sum)
```

3 Hadoop

In this section is presented the *Bloom Filter* creation implementation using Hadoop framework.

The *Bloom Filter* class employs the object **Bitset** to implement the array of bits. In the 6 section of this document was performed a comparison among three different possible implementations of a bit-vector in Java, and the **Bitset** provided the best results in terms of memory usage and so it was employed for the *Bloom Filter* class.

Since the *Bloom Filter* objects are sent as values between the map and reduce phases of the job stages, we need to implement the methods of the Writable Interface, that allows to serialize and de-serialize objects over Hadoop. The functions that implement the *insert()* and the *or()* operations of the *Bloom Filter* are omitted because are easily obtained from those written in the pseudo-code.

3.1 Bloom Filter Implementation

BloomFilter

```
public BloomFilter(int length, int kHash){
    bitset = new BitSet(length);
    this.length = length;
    this.kHash = kHash;
}
```

Write

```
@Override
public void write(DataOutput dataOutput) throws IOException {
    dataOutput.writeInt(this.length);
    dataOutput.writeInt(this.kHash);

    long[] longs = bitset.toLongArray();
    dataOutput.writeInt(longs.length);
    for (int i = 0; i < longs.length; i++) {
        dataOutput.writeLong(longs[i]);
    }
}
```

ReadFields

```
@Override
public void readFields(DataInput dataInput) throws IOException {
    length = dataInput.readInt();
    kHash = dataInput.readInt();
```

```
        long[] longs = new long[dataInput.readInt()];
        for (int i = 0; i < longs.length; i++) {
            longs[i] = dataInput.readLong();
        }
        bitset = BitSet.valueOf(longs);
    }
```

3.2 Parameter Calibration stage

Mapper

```
public static class CountByRatingMapper extends Mapper<Object,
    Text, IntWritable, IntWritable>{
    private int roundRating;
    private int[] counter = new int[10];

    public void map(Object key, Text value, Context context) {
        roundRating = (int)
            Math.round(Double.parseDouble(value.toString().split("\t")[1]));
        counter[roundRating-1]++;
    }

    public void cleanup(Context context) throws IOException,
        InterruptedException {
        for (int i=0; i<counter.length; i++)
            context.write(new IntWritable(i+1), new
                IntWritable(counter[i]));
    }
}
```

Reducer

```
public static class CountSumReducer extends
    Reducer<IntWritable,IntWritable,IntWritable,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(IntWritable key, Iterable<IntWritable>
        values, Context context) throws IOException,
        InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

3.3 Bloom Filter Creation stage

The parameters m, the number of bits, and k, the number of the hash functions, are computed in the Hadoop driver based on the output of the previous stage and on the user-specified false positive rate p.

Calculation of parameters - Driver

```
// Compute and set Bloom Filter parameters based on the result
    of the Parameter Calibration stage
double[] countByRating = readJobOutput(conf,
    conf.get("output.parameter-calibration"));

for (int i=0; i<countByRating.length; i++) {
    double n = countByRating[i];
    int m = (int)
        Math.round((-n*Math.log(Double.parseDouble(conf.get("input.p"))))
        /(Math.log(2)*Math.log(2)));
    int k = (int) Math.round((m*Math.log(2))/n);

    conf.set("input.filter_" + i + ".m", String.valueOf(m));
    conf.set("input.filter_" + i + ".k", String.valueOf(k));
}
```

Mapper

```
public static class BloomFilterCreationMapper extends
    Mapper<Object, Text, IntWritable, BloomFilter> {
    ArrayList<BloomFilter> bloomFilters = new ArrayList<>();
    private int roundRating;

    public void setup(Context context) {
        int m, k;
        for (int i = 0; i < 10; i++) {
            m =
                Integer.parseInt(context.getConfiguration().get("input.filter_"
                + i + ".m"));
            k =
                Integer.parseInt(context.getConfiguration().get("input.filter_"
                + i + ".k"));
            bloomFilters.add(i, new BloomFilter(m, k));
        }
    }

    public void map(Object key, Text value, Context context) {
        roundRating = (int)
            Math.round(Double.parseDouble(value.toString().split("\t")[1]));
        bloomFilters.get(roundRating-1).add(value.toString().split("\t")[0]);
    }
```

```
    public void cleanup(Context context) throws IOException,
        InterruptedException {
        for (int i = 0; i < 10; i++) {
            context.write(new IntWritable(i+1), bloomFilters.get(i));
        }
    }
}
```

Reducer

```
public static class BloomFilterOrReducer extends
    Reducer<IntWritable, BloomFilter, IntWritable, BloomFilter> {
    private BloomFilter result;

    public void reduce(IntWritable key, Iterable<BloomFilter>
        values, Context context) throws IOException {
        result = new BloomFilter(values.iterator().next());
        while(values.iterator().hasNext()) {
            result.or(values.iterator().next().getBitset());
        }

        // Save the final Bloom Filter in the file system
        Path outputFilePath = new
            Path(context.getConfiguration().get("output.bloom-filters")
                + Path.SEPARATOR + "filter"
                + key.toString());
        FileSystem fs = FileSystem.get(context.getConfiguration());

        try (FSDataOutputStream fsdos = fs.create(outputFilePath)) {
            result.write(fsdos);

        } catch (Exception e) {
            throw new IOException("Error while writing bloom filter
                to file system.", e);
        }
    }
}
```

3.4 Parameter Validation stage

Mapper

```
public static class ParameterValidationMapper extends
    Mapper<Object, Text, IntWritable, IntWritable> {
        ArrayList<BloomFilter> bloomFilters = new ArrayList<>();
        private final int[] counter = new int[10];

        public void setup(Context context) throws IOException {
```

```
            for (int i = 0; i < 10; i++) {
                Path bloomFilterCachePath = new
                    Path(context.getConfiguration().get("output.bloom-filters")
                        + "/filter" + (i+1));
                FileSystem fs =
                    FileSystem.get(context.getConfiguration());

                try (FSDataInputStream fsdis =
                    fs.open(bloomFilterCachePath)) {
                    BloomFilter tmp = new BloomFilter();
                    tmp.readFields(fsdis);
                    bloomFilters.add(i, tmp);

                } catch (Exception e) {
                    throw new IOException("Error while reading Bloom
                        Filter cache file from file system.", e);
                }
            }
        }

        public void map(Object key, Text value, Context context) {
            double rating =
                Double.parseDouble(value.toString().split("\t")[1]);
            int roundRating = (int) Math.round(rating);

            for (int i=0; i<10; i++) {
                if (roundRating == (i+1))
                    continue;
                if
                    (bloomFilters.get(i).find(value.toString().split("\t")[0]))
                    counter[i]++;
            }
        }

        public void cleanup(Context context) throws IOException,
            InterruptedException {
            for (int i = 0; i < 10; i++) {
                context.write(new IntWritable(i+1), new
                    IntWritable(counter[i]));
            }
        }
    }
```

Reducer

```
public static class ParameterValidationReducer extends
    Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {
    private final IntWritable result = new IntWritable();
    public void reduce(IntWritable key, Iterable<IntWritable>
```

```
        values, Context context) throws IOException,
        InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

The computation of the false positive rate is performed in the Hadoop driver, using the output values of the first stage (the count by rating) and the output values of the last stage (the number of false positives per rating). The actual false positive rates will be discussed in the validation section 5.

False positive rate - Driver

```
    // Compute false positive rate for each rating
    double[] falsePositiveCounter = readJobOutput(conf,
        conf.get("output.parameter-validation"));
    double[] falsePositiveRate = new double[10];
    double tot = 0;

    for (int i=0; i<10; i++)
        tot += countByRating[i];

    for (int i=0; i<10; i++) {
        falsePositiveRate[i] = (double)
            100*falsePositiveCounter[i]/(tot-countByRating[i]);
    }
```

3.5 Tuning Job Parameters

To optimize the performances of the Hadoop implementation, all the possible combinations of the configuration parameters have been tested, comparing their execution time. The script *Test.java* has been exploited for this purpose: it simply sets the *config.properties* file for each combination of parameters, it launches the Hadoop MapReduce workflow and it stores all the final output information in a json file. The script has been launched with a *false positive rate* value of 0.01. In order to have statistical relevance, each combination of parameters have been tested a total of ten times; in the following table are reported the mean values of the results.

In the first column is shown the combination of the configuration parameters with the following format:

numberOfReducerJob0 - numberOfReducerJob1 - nLineSplitJob1

Note that, with a *nLineSplitJob1* value of 200000, the second stage will have 6 mappers, with 400000 it will have 4 mappers and with 800000 it will have 2 mappers.

In the other columns are shown the execution time of to the first two stages, and the one of the overall job, all of them expressed in milliseconds. The results in the table are ordered with respect to the value of the Stage 1 Exec. Time column: our goal is to minimize primarily the execution time of the second stage, the one that actually creates the *Bloom Filters*.

Finally, the best configuration and the one that will be employed in our Hadoop implementation is the one with 2 reducers for the first two jobs and 4 mappers (400000 nLineSplit).

Hadoop Parameters Tuning			
Parameters	**Stage 0 (ms)**	**Stage 1 (ms)**	**Total (ms)**
2-2-400000	19741.9	18052.8	65024.1
1-3-400000	20362.8	18154.0	66972.6
3-1-400000	20589.0	18571.9	67048.1
3-3-400000	20380.0	18597.9	67565.5
1-1-400000	20117.2	18743.5	66704.4
1-2-400000	19372.9	18793.1	65823.8
3-2-800000	20437.5	18801.6	67678.5
1-3-800000	19474.1	18947.8	66224.0
1-2-800000	19986.2	19029.3	66326.0
2-1-400000	19985.4	19083.6	66740.7
3-2-400000	22334.6	19166.0	70727.9
2-1-200000	20142.1	19387.5	66668.0
1-2-200000	19728.3	19542.4	67179.8
3-1-800000	20330.1	19609.6	67719.0
2-1-800000	20238.2	19620.6	66938.5
2-2-800000	20205.5	19798.8	67647.9
1-3-200000	19759.7	19846.4	66452.6
2-2-200000	20108.0	20016.9	68761.3
1-1-200000	20184.7	20121.3	68953.7
2-3-200000	20089.1	20188.3	69793.1
2-3-400000	20249.7	20192.1	69470.6
3-3-800000	20577.5	20212.1	68576.4
3-3-200000	20133.4	20639.2	69081.4
1-1-800000	19877.4	20751.7	68950.1
3-2-200000	20662.4	20940.7	72178.4
2-3-800000	19878.8	21142.0	70326.7
3-1-200000	21695.0	21743.0	72663.2

4 Spark

In this section will be presented the *Bloom Filter* implementation with the Spark frame-work.

4.1 Bloom Filter Implementation

Bloom Filter Class

```
class Bloomfilter(object):
    def __init__(self, length, kHash):
        self.length = length
        self.kHash = kHash
        self.bits = bitarray(length)
        self.bits.setall(0)

    def copy(self):
        return copy.deepcopy(self)

    def add(self, id):
        seed = 0
        for i in range(self.kHash):
            seed = mmh3.hash(id, seed)
            self.bits[abs(seed % self.length)] = 1

    def bitwise_or(self, input):
        self.bits = self.bits | input.bits
        return self

    def print(self):
        print(self.bits)

    def find(self, id):
        seed = 0
        for i in range(self.kHash):
            seed = mmh3.hash(id, seed)
            if self.bits[abs(seed % self.length)] == 0:
                return False
        return True
```

4.2 Parameter Calibration stage

Bloom Filter Object

```
def id_rate_split(line):
    items = line.split("\t")
    return items[0], int(Decimal(items[1]).quantize(0,
        ROUND_HALF_UP))
```

```
# main
rdd_films = rdd_input.map(id_rate_split)
counts_by_rating = rdd_films.map(lambda x: (x[1],
    1)).reduceByKey(lambda x, y: x + y).sortByKey
counts_by_rating = rdd_counts_by_rating.collect()
```

4.3 Bloom Filter Creation stage

Bloom Filter Object

```
def init_bloomfilter(n, p):
    m = int(round((-n * math.log(p)) / (math.log(2) ** 2)))
    k = int(round(m * math.log(2) / n))
    return m, k

def insert_in_bloomfilters(lines):
    # setup
    bloomfilters = [Bloomfilter(m, k) for m, k in
        broadcast_bf_params.value]

    # computation
    for line in lines:
        (id, rate) = id_rate_split(line)
        bloomfilters[rate - 1].add(id)

    rate = range(1, 11)
    return zip(rate, bloomfilters)

# main
bf_params = [init_bloomfilter(n, p) for rating, n in
    counts_by_rating]
broadcast_bf_params = sc.broadcast(bf_params)
rdd_partial_bf = rdd_input.mapPartitions(insert_in_bloomfilters)
rdd_final_bf = rdd_partial_bf.reduceByKey(lambda filter1,
    filter2: filter1.bitwise_or(filter2)).sortByKey()
```

4.4 Parameter Validation stage

Bloom Filter Object

```
def validate_bloomfilter(line):
    counter = []
    # for line in lines:
    (film_id, rate) = id_rate_split(line)
    for bf in broadcast_bf.value:
        if rate != bf[0]:
```

```
            result = bf[1].find(film_id)
            if result is True:
                counter.append(tuple((bf[0], 1)))
    return counter

# main
broadcast_bf = sc.broadcast(rdd_final_bf.collect())
rdd_counter = rdd_input.flatMap(validate_bloomfilter)
rdd_false_positive_count = rdd_counter.reduceByKey(lambda x, y:
    x + y).sortByKey

false_positive_count = rdd_false_positive_count.collect()
counts_by_rating = rdd_counts_by_rating.collect()
film_count = rdd_input.count()

with open(OUTPUT_PATH, "w") as f:
    for i in range(10):
        false_positive_rate =
            false_positive_count[i][1]/(film_count -
            counts_by_rating[i][1])
        f.write(str(i+1) + "\t" + str(false_positive_rate) +
            "\n")
```

5 False Positive Rate Results

We created all the *Bloom Filters* setting the value of p to 1%. The *false positive rates* of each rating, obtained as final output of the MapReduce workflow of the two frameworks, are shown in the following table, along with the *expected false positive rate*, computed using the following formula:

$$p = (1 - e^{\frac{-kn}{m}})^k \tag{3}$$

where m, k, n are the *Bloom Filter* configuration parameters.

False Positive Rate (%)			
Rating	**Hadoop FPR**	**Spark FPR**	**Expected FPR**
1	1.0486199571952415	1.0131103663848982	1.003945915
2	1.015727497754138	1.0107187405075971	1.003897072
3	0.993999478215497	1.0168275502217584	1.003934271
4	0.9926721625447581	1.0018319593638105	1.003922309
5	1.0124543086654847	1.0230481318537002	1.003924689
6	1.0041459054429271	1.0157517099056029	1.003923199
7	0.9943498688423694	1.0080776627058169	1.004695403
8	1.0121098727543567	1.0084082628905758	1.00392144
9	1.005098865661509	1.001651296993013	1.003921302
10	0.9958745364144609	1.0049915830123372	1.003926691

6 Tests

6.1 Bloom Filter Structure Comparison

We have tried three different *Bloom Filter* class, each of them with a different implementation of the array of bits. In order to verify which of them is the best we compared different MapReduce parameters such as: time of execution, usage of the memory, data sent across the cluster. Note: the three implementations have been compared in the Hadoop framework.

Here are the three implementation:

- *Position*
  Instead of using a regular array, bit array is implemented by a list of indexes that have been set by the hash functions of the *Bloom Filter*.
- *Boolean*
  In this implementation the array of bits is implemented with an array of booleans.
- *Bitset*
  In the last implementation we tried the Bitset class, that is a java built-in class that implements an optimized version of a bit array.

Based on our results, that are shown in the following table, we decided to use the Bitset implementation.

Bloom Filter Implementation Comparison			
Parameter	**Position**	**Boolean**	**Bitset**
Number of bytes read (FILE)	9070052	16675520	2085304
Number of bytes written (FILE)	19473796	34684764	5504300
Number of bytes written (HDFS)	7494660	4168791	521248
Map output bytes	9069860	16675324	2085120

6.2 False Positive Rate Comparison

False Positive Rate			
framework	**FPR**	**Stage 1 (ms)**	**Std**
Hadoop	0.20	18986.2	1392.132481
	0.10	18976.9	1564.989915
	0.05	19019.2	820.913556
	0.01	19711.6	822.832648
Spark	0.20	3161.6806	222.81397
	0.10	3578.7374	199.07603
	0.05	3986.5408	179.26152
	0.01	5277.8095	341.43247

In this section we compared the performance of the *Bloom Filter* creation algorithm with different values of the false positive rate p. We tested both the Hadoop and Spark implementation (for the Hadoop tests we employed the tuned configuration parameters obtained in the previous section). In the first column of the table is show the false positive rate, while in the other columns the execution time of the *Bloom Filter* creation stage and the standard deviation. Note that the other two stages of the workflow are not affected by the value of p, and so their execution times have not been reported. As expected, by increasing the value of the false positive rate, the execution time decreases.

6.3 VM Cluster Performance Test

In this subsection we show the performance results, in terms of mean execution time, of the tests performed by shutting down some of the cluster machines and keeping the others on.

For each configuration are reported the mean execution times of the three stages over 10 iterations.

VM Cluster performance test				
framework	machines	Stage 1 (ms)	Stage 2 (ms)	Stage 3 (ms)
Hadoop	1	30080.6	41995.2	35993.8
	2	26118.1	27392.9	29416.6
	3	20322.3	19132.7	24828.2
	4	19741.9	18052.8	26024.1
Spark	local	3553.238702	5466.302586	13481.88584
	1	6454.933929	10423.93034	25598.08214
	2	5458.744216	8297.260547	17522.54481
	3	3375.617242	5277.809477	13016.33408

6.3.1 Hadoop Performances

Here below we reported a graph showing the results obtained using different number of workers.

Figure 3: Hadoop performance test's results

6.3.2 Spark Performances

As we can see from the graph in Figure 4, the better results in terms of execution time can be obtained using the *local configuration* or *3 workers*.

6.3.3 Comparison

By observing Figure 5, the *Spark* implementation is much faster in terms of execution time respect to the *Hadoop* implementation.

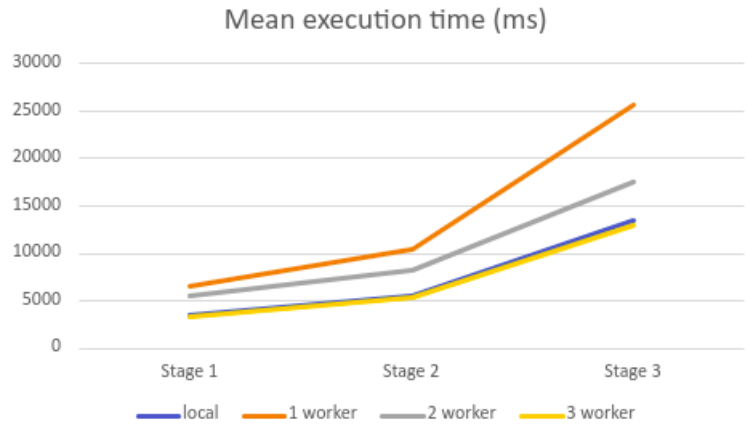

Figure 4: Spark performance test's results

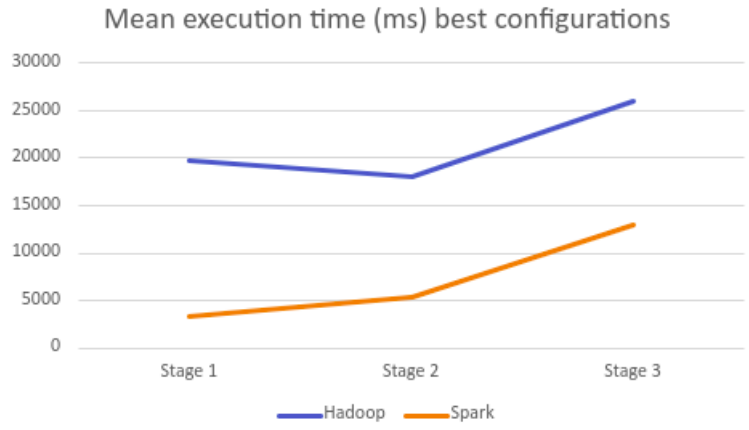

Figure 5: Final comparison of Hadoop and Spark performances