University of Pisa
Department of Information Engineering
Cloud Computing course

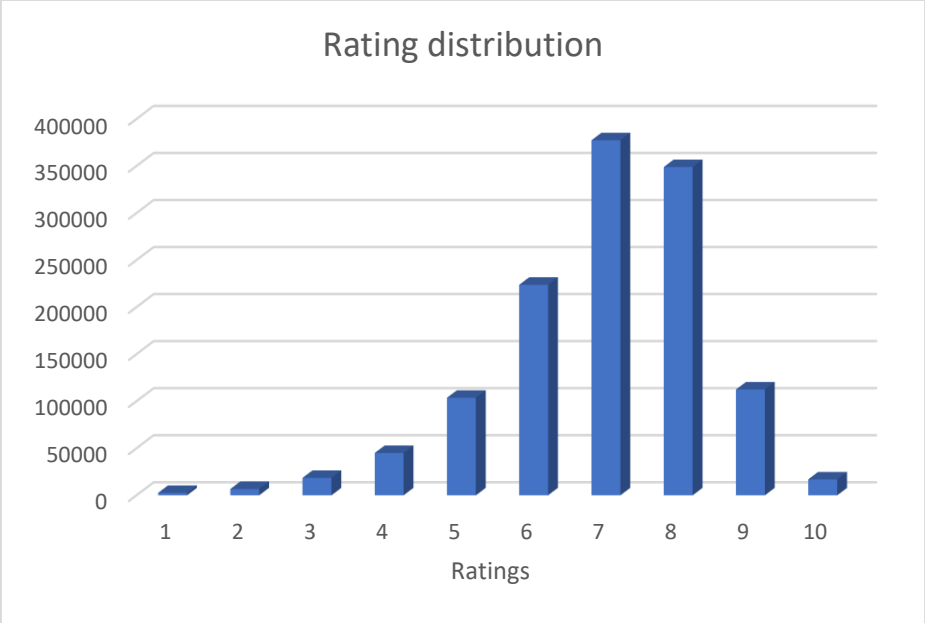
Bloom Filter implementation in Hadoop and Spark

Abaterusso Matteo, Arancio Febbo Salvatore, Di Donato Mattia, Giorgi Matteo

Contents

1 Introduction

In this project we present a possible solution for the implementation of a distributed Bloom Filter using the map reduce approach on Hadoop and Spark framework. The data used to create and populate these Bloom Filter came from an IMDb dataset on average rating related to movies. Ten Bloom Filter will be realized since the IMDb rating is a measure that goes from 1 to 10. In the following histogram the distribution of films over the rating is shown.



The Bloom Filter is a space-efficient probabilistic data structure used to test if an element is a member of a set or not. False positive matches are possible but false negatives not. In this context, this type of data structure can be used, for instance, to speed up the response time of search query on movies.

Before the population of the Bloom Filters, a parameter calibration stage is needed. First of all, is necessary to fix the desired p, expected false positive rate, then we have to calculate n, the number of keys added for membership testing. After this we can finally discover the m and k parameters which respectively represent the number of bits of the bloom filters and the number of used hash functions. Parameters can be calculated using the following formulas.

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

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

$$\mathrm{p} \cong \left(1 - e^{-\frac{kn}{m}}\right)^{\mathrm{k}}$$

The documentation is organised as follow: a design phase, in which the approach to the problem will be explained, the implementation section in which a Hadoop and Spark implementation is presented and a result paragraph in which the false positive rate and the experimented duration of the application are shown.

The code is available in the following GitHub repository:
https://github.com/mattiadido95/Distributed-BloomFilter

2 Design

In this section we will see which are the main players on which the *Distributed-BloomFilter* is based. The main components are Bloom Filter type objects on which will be applied 3 Map-Reduce stages for their realization.

The first one is the *ParameterCalculation* in which the dimensional parameters for each Bloom Filter are calculated; after this follows the *BloomFilterCreation* phase in which Bloom Filters are created and populated using the dataset. Finally, the *BloomFilterValidation* phase in which the performance of the data structures, in terms of false positive rate, are evaluated.

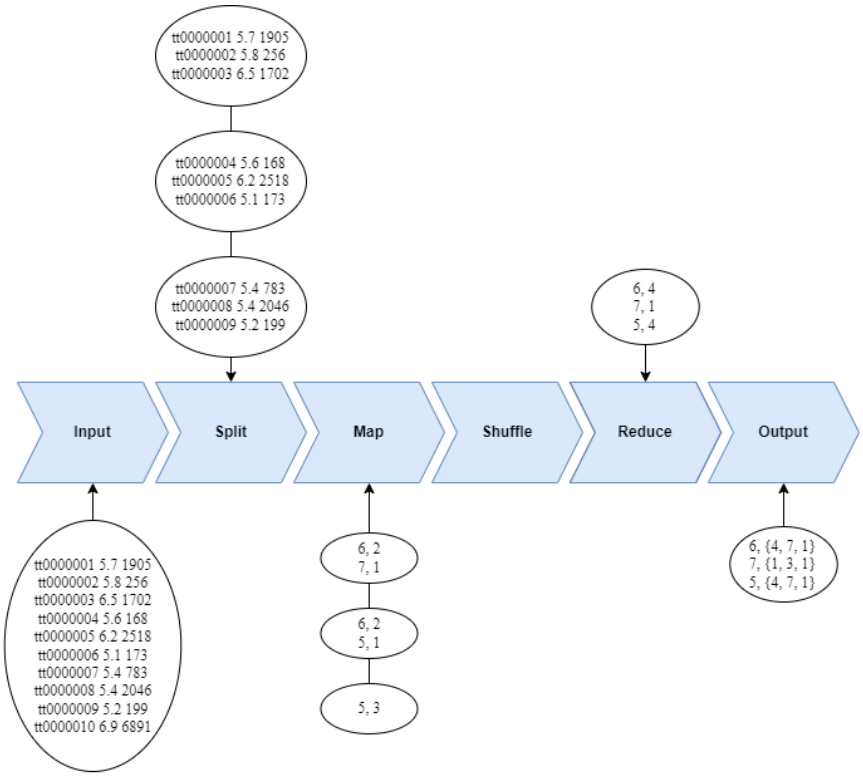
2.1 Bloom Filter

Bloom Filter class is a data structure containing information and methods that allow to work with the Map-Reduce model. The attributes of this class are: m which is the size of the bloom filter, k which gives information about the number of hash functions to be used and a bit array representing the bloom filter structure.

Among the main methods made available by the Bloom Filter there are the Add() method which adds elements to the bloom filter array through the Murmur hash function, the Find() method which searches for elements within a bloom filter returning true or false and finally a very important feature for the Reduce phase which is the bitwise-or operation, used to merge results between bloom filters.

```
class BloomFilter:
    method Inizialize(m, k):
        arrayBF = new BitSet(m)

    method Inizialize(m,k,bfs):
        arrayBF = new BitSet(m)
        for bf in bfs:
            arrayBF.or(bf)

    method Add(title):
        for i in range(0, k-1):
            index = Hash(title, i) % m
            arrayBF[index] = 1

    method Find(title):
        for i in range(0, k-1):
            index = Hash(title, i) % m
            if(arrayBF[index] == 0):
                return false
            return true
```

2.2 Parameter Calibration



In this phase, for each bloom filter, are calculated the parameters useful for their creation, considering the type and distribution of the dataset.

Having fixed the parameter p which expresses the desired error rate through the following formulas: $m = -\frac{n \ln p}{(\ln 2)^2}$, $k = \frac{m}{n} \ln 2$ the parameters m and k will be calculated.

Essential for this calculation is the value of n, which represents the number of data that will be as input to each bloom filter. Following project specification 10 Bloom Filter will be created, one for each rating present in the dataset. In this way, through a Map-Reduce approach on the initial dataset, we calculate how many inputs we will have for each rating and therefore for each bloom filter.

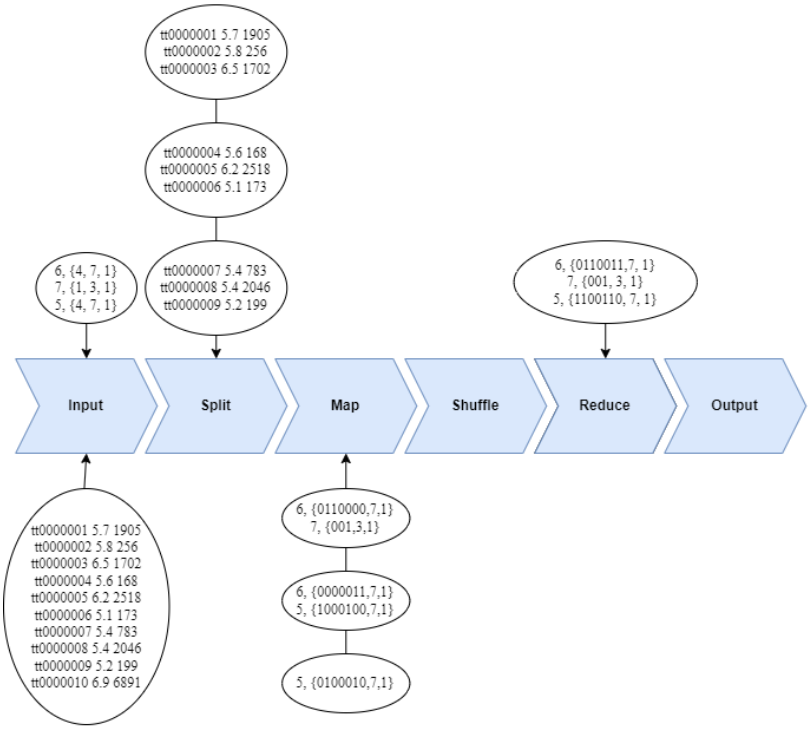
Specifically, in the map phase we are going to divide the inputs on several mappers to enhance the calculation phase for each rating. To do this, we take advantage of the optimization offered by the In-Mapper Combining pattern which reduce the number of emitted key-value couples in the cluster, obtaining a lower network traffic.

In this way, each mapper will process a partial count for each rating then in the reducer phase the partial counters will be aggregated for each rating, providing for each bloom filter the n parameter.

```
Input: title, rating
Output: [m, n, k] for each rating i in range(1,11)

class Mapper(title, rating):
    method Inizialize():
        counter = new array[10]
```

```
    method Map():
        rating = round(rating)
        counter[rating-1]++

    method Close():
        for i in range(0,10):
            emit(i+1, counter[i])

class Reducer(rating, counters[]):
    n = 0
    for counter in counters:
        n += counter
    m = -(n * ln(p)) / (ln(2)^2)
    k = (m / n) * ln(2)
    emit(rating, [n, m, k])
```

2.3 Bloom Filter Creation



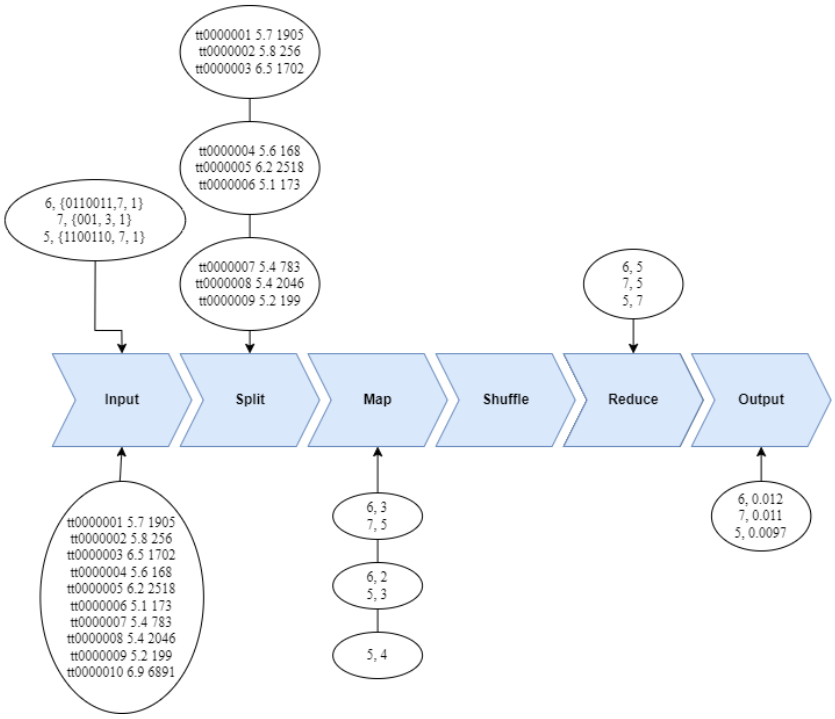
At this point in the workflow, the calibrated parameters for each rating are known and the Bloom Filter Creation phase can start. In the initialization phase 10 bloom filters are created for all mapper with the corresponding m an k parameter. Then this phase continues with the insertion of films into the corresponding bloom filters based on the rating.

After the map, the reduce phase is based on the bitwise-or operation, where the bloom filters belonging to the same rating are merged together. A list of 10 bloom filters, initialized with the entire dataset, will be given as output at the end of this phase.

```
Input: title, (rating, [n, m, k]) foreach ratings in range(1,11)
Output: rating, bloomFilters[10]
```

```
class Mapper(title, rating):
    method Inizialize():
        BloomFilter[] bf = new BloomFilter[10]
        for i in range(0,10):
            bf[i] = new BloomFilter(m, k)

    method Map():
        r = round(rating)
        bf[r-1].Add(title)

    method Close():
        for i in range(0,10):
            emit(i+1, bf[i])

class Reducer(rating, bloomFilters[]):
    bloomFilter = new BloomFilter(m, k, bloomFilters)
    emit(rating, bloomFilter)
```

2.4 Bloom Filter Validation



In this final phase a performance analysis is performed over the data structure created. For each bloom filter, the number of false positives is obtained by giving as input to the find method the portion of the dataset which do not belong to that analysed bloom filter.

To process the count, the optimization provided by the In-Mapper Combining pattern is used. This one, trough the map phase, will provide a partial count for each bloom filter which then will be aggregated in the reduce phase. So, for each single bloom filter, a count on the searches that returned

false positives results is obtained. With these values the percentage of the false positive rate will be calculated and compared to the initially chosen p to evaluate the performance.

```
Input: title, rating, bloomFilter in bloomFilters[10]
Output: rating, FalsePositiveRate

class Mapper(title, rating):
    method Inizialize():
        counter = new array[10]

    method Map():
        r = round(rating)
        for i in range(0,10):
            if ( i == r-1 ):
                continue
            if ( bloomFilter.Find(title) ):
                counter[i]++

    method Close():
        for i in range(0,10):
            emit(i+1, counter[i])

class Reducer(rating, fp_counters[]):
    fp_tot_counter = 0
    for counter in counters:
        fp_tot_counter += counter
    emit(rating, fp_tot_counter)
```

3 Hadoop

In this paragraph will be explained the Bloom Filter implementation realized on Hadoop.

3.1 Bloom Filter Class

The BitSet class has been used for the realization of the bloom filter because it is more optimized than a Boolean array for large number of bits. Moreover, it already provides method like set and bitwise-or.

Since is needed to create a Bloom Filter merging a set of existing Bloom Filters, has been decided to implement a constructor that take as input that iterator of them and apply the bitwise-or.

To exchange the Bloom Filters between Mappers and Reducers, the Write and Read method of the Writeable interface have been overridden and the serialized information are m and k parameters, a byte array that represent the BitSet and its length.

Merger constructor

```
// used to create a bloom filter from an iterator of bloom filters
public BloomFilter(Iterable<BloomFilter> arrayBFs){
    BloomFilter first = arrayBFs.iterator().next();
    this.m = first.m;
    this.k = first.k;
    this.arrayBF = (BitSet) first.arrayBF.clone();

    while(arrayBFs.iterator().hasNext()) {
        or(arrayBFs.iterator().next().getArrayBF());
    }
}

public void or(BitSet input){
    this.arrayBF.or(input);
}
```

Add method

```
// compute k MURMUR_HASH for a given title and set relative bits in bloom filter
public void add(String title) {
    int index;
    for (int i = 0; i < k; i++) {
        index =
Math.abs(Hash.getInstance(Hash.MURMUR_HASH).hash(title.getBytes(StandardCharsets.UTF_8),
i)) % m;
        arrayBF.set(index, true);
    }
}
```

Find method

```
// return true only if it finds all the k bits, associated to the title, set
public boolean find(String title) {
    int index;
    for (int i = 0; i < k; i++) {
        index =
Math.abs(Hash.getInstance(Hash.MURMUR_HASH).hash(title.getBytes(StandardCharsets.UTF_8),
i)) % m;
        if (arrayBF.get(index) == false)
            return false;
```

```
    }
    return true;
}
```

Write method

```
public void write(DataOutput out) throws IOException {
    out.writeInt(this.m);
    out.writeInt(this.k);
    byte[] bytes = arrayBF.toByteArray();

    out.writeInt(bytes.length);
    for (int I = 0; I < bytes.length; i++) {
        out.writeByte(bytes[i]);
    }
}
```

Read method

```
public void readFields(DataInput in) throws IOException {
    this.m = in.readInt();
    this.k = in.readInt();
    int length = in.readInt();
    byte[] bytes = new byte[length];

    for (int i = 0; i < length; i++) {
        bytes[i] = in.readByte();
    }
    this.arrayBF = BitSet.valueOf(bytes);
}
```

3.2 Parameter Calculation Stage

The Setup and Cleanup method have been used to implement the in-mapper combining optimization described in the design chapter. Moreover, before emitting the result in the cleanup phase a check on the value of the counter is performed to avoid unnecessary network traffic. At the beginning of this stage the P parameter is stored into the context, in this way every reducer can access it.

Mapper

```
protected void setup(Context context){
    counter = new int[maxRating];
}

public void map(Object key, Text value, Context context) throws NumberFormatException {
    String record = value.toString();
    if (record == null || record.length() == 0)
        return;

    String[] tokens = record.split("\t");

    //skip file header
    if(tokens[0].equals("tconst"))
        return;

    // <title, rating, numVotes>
    if (tokens.length == 3) {
        int roundedRating = (int) Math.round(Double.parseDouble(tokens[1]));
        this.counter[roundedRating-1]++;
```

```
    }
}

protected void cleanup(Context context) throws IOException, InterruptedException {
    for (int i=0; i < maxRating; i++)
        //emit only if there are films of rating i
        if(counter[i] > 0)
            context.write( new IntWritable(i+1), new IntWritable(counter[i]) );
}
```

Reducer

```
public void reduce(IntWritable key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
    int n = 0;
    int m,k;
    double p = context.getConfiguration().getDouble("parameter.calculation.p",0.05);

    // merge mapper's counters for rating = key
    while (values.iterator().hasNext())
        n += values.iterator().next().get();

    m = (int) (- ( n * Math.log(p) ) / (Math.pow(Math.log(2),2.0)));
    k = (int) ((m/n) * Math.log(2));
    Text value = new Text(n + "\t" + m + "\t" + k);
    context.write(key, value);
}
```

3.3 Bloom Filter Generation Stage

In this stage the m and k parameter of the bloom filters, computed in the previous stage, are shared among maps and reduces through context. Like the first stage, the in-mapper combining has been implemented and in the cleanup method a Bloom Filter is emitted only if it is not empty.

Mapper

```
protected void setup(Context context) {
    this.bf = new BloomFilter[maxRating];

    for (int i = 0; i < maxRating; i++) {
        int index = i + 1;
        int m = context.getConfiguration().getInt("filter." + index + ".parameter.m", 0);
        int k = context.getConfiguration().getInt("filter." + index + ".parameter.k", 0);

        //no film for rating i
        if (m == 0 || k == 0)
            bf[i] = null;
        else
            bf[i] = new BloomFilter(m, k);
    }
}

public void map(Object key, Text value, Context context) throws NumberFormatException {
    String record = value.toString();
    if (record == null || record.length() == 0)
        return;

    String[] tokens = record.split("\t");

    // skip file header
    if (tokens[0].equals("tconst"))
```

```
            return;

        // <title, rating, numVotes>
        if (tokens.length == 3) {
            int roundedRating = (int) Math.round(Double.parseDouble(tokens[1]));
            if (bf[roundedRating - 1] != null) {
                // set bits in the bloom filter for that title
                bf[roundedRating - 1].add(tokens[0]);
            }
        }
    }


    protected void cleanup(Context context) throws IOException, InterruptedException {
        for (int i = 0; i < maxRating; i++)
            // emit only if bloom filter exists
            if (bf[i] != null) {
                context.write(new IntWritable(i + 1), bf[i]);
            }
    }
```

Reducer

```
public void reduce(IntWritable key, Iterable<BloomFilter> values, Context context) throws
IOException, InterruptedException {
    BloomFilter bfTot = new BloomFilter(values);

    context.write(key, bfTot);
}
```

3.4 Bloom Filter Validation Stage

In the Setup method every Mapper loads all the Bloom Filters that have been created in the previous stage. The result of this Map-Reduce phase is the number of false positive of each Bloom Filter used by the Driver to calculate the false positive rate (the effective p).

Mapper

```
protected void setup(Context context) throws IOException {
    this.bf = readFilter(context.getConfiguration());
    counter = new int[maxRating];
}

public void map(Object key, Text value, Context context) throws NumberFormatException {
    String record = value.toString();
    if (record == null || record.length() == 0)
        return;

    String[] tokens = record.split("\t");

    // skip file header
    if (tokens[0].equals("tconst"))
        return;

    // <title, rating, numVotes>
    if (tokens.length == 3) {
        int roundedRating = (int) Math.round(Double.parseDouble(tokens[1]));

        for (int i = 0; i < maxRating; i++) {
            if (roundedRating == i+1)
                continue;
```

```
            // increment counter if title is found in bloom filter
            if (bf[i].find(tokens[0]))
                counter[i]++;
        }
    }
}

protected void cleanup(Context context) throws IOException, InterruptedException {
    for (int i = 0; i < maxRating; i++) {
        // emit only if we found false positive counter for rating i
        if (counter[i] > 0) {
            context.write(new IntWritable(i + 1), new IntWritable(counter[i]));
        }
    }
}
```

Reducer

```
public void reduce(IntWritable key, Iterable<IntWritable> values, Context context) throws
IOException, InterruptedException {
    int falsePositive = 0;

    //merge false positive counter
    while (values.iterator().hasNext())
        falsePositive += values.iterator().next().get();

    context.write(key, new IntWritable(falsePositive));
}
```

False Positive Rate Calculation

```
double[] percentage = new double[totalRating];
for (int i = 0; i < totalRating; i++) {
    percentage[i] = (double) result2[i] / (double) (total_n - result[i]);
}
```

3.5 Configuration File

In order to set up the execution of the code without changing it, a Json configuration file has been used. As follow an example of it is shown.

```
{
  "falsePositiveRate" : 0.01,
  "input" : "data.tsv",
  "output" : {
    "stage1" : "parameter",
    "stage2" : "filter",
    "stage3" : "falsePositive"
  },
  "linesPerMap" : {
    "stage1" : 1256196,
    "stage2" : 1256196,
    "stage3" : 1256196
  },
  "nReducer" : {
```

```
    "stage1" : 1,
    "stage2" : 1,
    "stage3" : 1
  },
  "root" : "hdfs://hadoop-namenode:9820/user/hadoop/",
  "statsFile": "stage-duration-hadoop.txt",
  "outputFile": "output-hadoop.csv"
}
```

4 Spark

In this section will be presented the Bloom Filter implementation with the Spark framework.

4.1 Bloom Filter Implementation

Bloom Filter Class

```
class BloomFilter:
    def __init__(self, m, k):
        self.m = m
        self.k = k
        self.array_bf = [0 for _ in range(m)]


    def add(self, title):
        for i in range(self.k):
            self.array_bf[abs(mmh3.hash(title, i) % self.m)] = 1


    def find(self, title):
        for i in range(self.k):
            if self.array_bf[abs(mmh3.hash(title, i) % self.m)] == 0:
                return False
        return True


    def merge(self, bf):
        self.array_bf = [x|y for x,y in zip(self.array_bf, bf.get())]
        return self


    def get(self):
        return self.array_bf
```

4.2 Calculation of parameters

Having a file with n rows and 2 columns (with title and rating), the map is made by using the "word_split" function which will separate each row, returning the title as the key and the rounded rating as a value.

With the obtained results, the mapPartition will be executed and it will perform the "counter_rating" function. This function is applied on each partition of data and returns key-value pairs of rating and quantity of titles for that rating. All the partial results will be aggregated by the Reducer which will return the corresponding "n" related to each rating and consequently the k and m values will be calculated.

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


def counter_rating(rows):
```

```
    counter = [0 for _ in range(10)]
    for row in rows:
        counter[row[1] - 1] = counter[row[1] - 1] + 1
    return zip(range(1, 11), counter)


#main
rdd_title_rating = rdd_input.map(word_split)
count_rating = rdd_title_rating.mapPartitions(counter_rating)
count_rating = count_rating.reduceByKey(lambda x, y: x + y).sortByKey()
```

4.3 Bloom Filter Creation

The m and k parameters and the dataset are used to create and populate the Bloom Filters from the mapPartition through the "bf_creation" function and aggregated by the Reducer.

```
def bf_creation(rows):
    bloomfilters = [BloomFilter(m, k) for r, n, m, k, in parameter_rating.value]
    for row in rows:  # (title, rating) = row
        bloomfilters[row[1] - 1].add(row[0])
    return zip(range(1, 11), bloomfilters)

#main
rdd_bf = sc.broadcast(
rdd_title_rating.mapPartitions(bf_creation).reduceByKey(lambda x, y:
x.merge(y)).sortByKey().collect())
parameter_rating.destroy()
```

4.4 False Positive Validation

False positives are calculated for each rating via mapPartition using the “bf_compare” function and aggregated by the Reducer. Each percentage must be strictly close to the expected p.

```
def bf_compare(rows):
    counter = [0 for _ in range(10)]
    for row in rows:
        for i in range(10):
            if (i + 1) == row[1]:
                continue
            if rdd_bf.value[i][1].find(row[0]):
                counter[i] = counter[i] + 1
    return zip(range(1, 11), counter)


#main
rdd_compare = rdd_title_rating.mapPartitions(bf_compare).reduceByKey(lambda x, y: x +
y).sortByKey()
```

5 Results

In the following table the resulting false positive rate are shown for Hadoop and Spark. The expected false positive rate, used for calculating bloom filter parameters, is p=0,01.

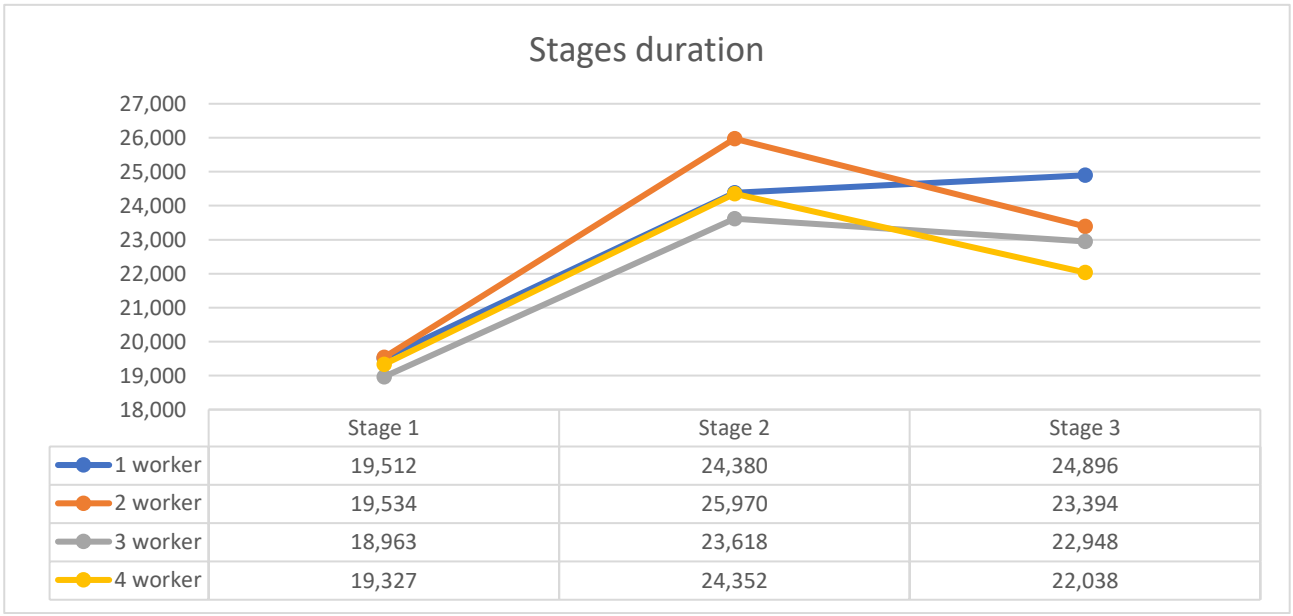
False Positive Rate		
Rating	**Hadoop**	**Spark**
1	0,01011161	0,010194566
2	0,00980107	0,010017159
3	0,0099891	0,010146657
4	0,00997115	0,010179212
5	0,0100763	0,010139644
6	0,01004176	0,010169616
7	0,01000122	0,010265216
8	0,01011615	0,010154729
9	0,01009923	0,010109719
10	0,01024656	0,00988263

The slightly differences between the two frameworks probably are due to different representation of data in Java and Python.

6 Performance

6.1 Hadoop

In the following table are shown mean durations of 5 repetitions for each stage, varying the number of mappers from 1 to 4 and keeping the number of reducers fixed at 1. In addition, a p = 0.01 was used.

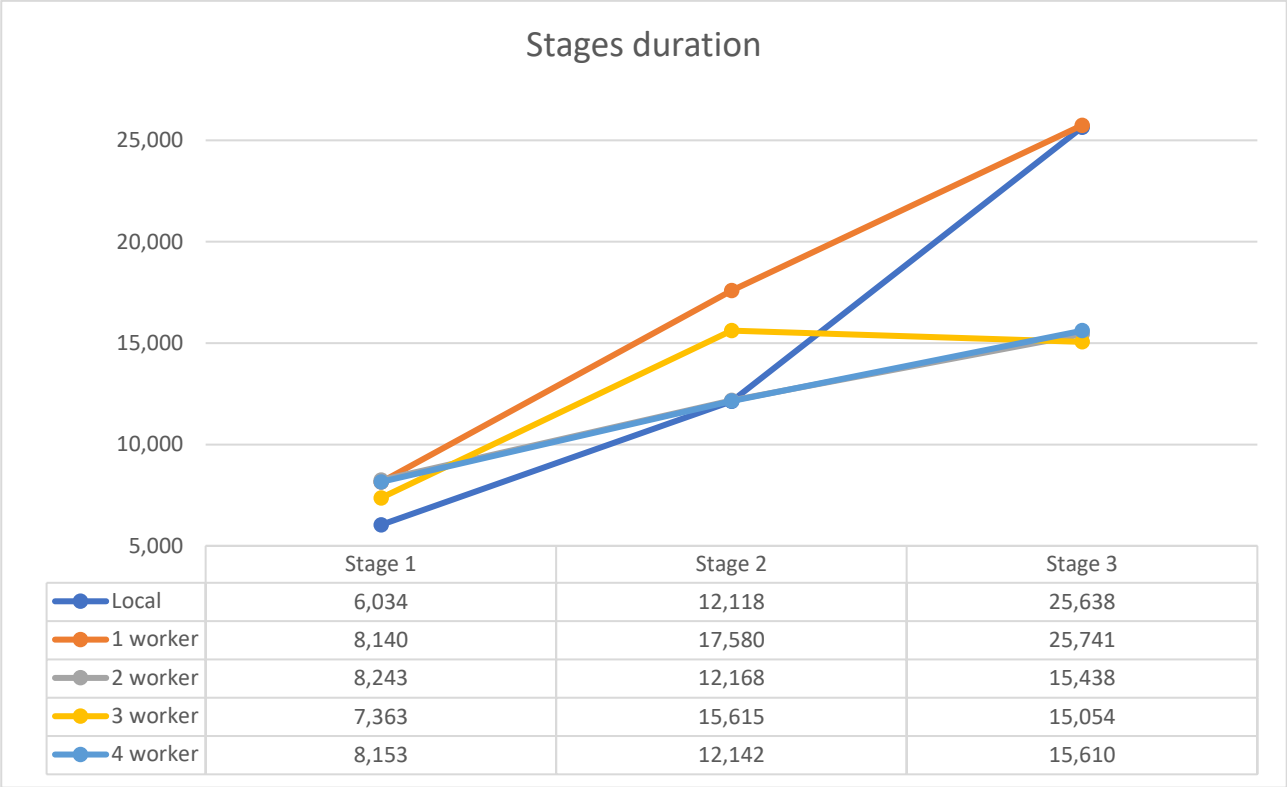


	Stage 1	Stage 2	Stage 3
1 worker	19,512	24,380	24,896
2 worker	19,534	25,970	23,394
3 worker	18,963	23,618	22,948
4 worker	19,327	24,352	22,038

As expected, the best performance is achieved by exploiting the parallelization of multiple maps. In particular, this can be seen more in stage 3, which requires more computational power (maps performs a find on each bloom filter); in this case the slowest execution is achieved with only 1 worker and the fastest execution is achieved with 4 workers. Instead, in the other stages, where the required computational power is lower, the communication overhead affects the duration more.

6.2 Spark

In the following table are shown mean durations of 5 repetitions for each stage, varying the number of workers from 1 to 4. In addition, a p = 0.01 was used.



	Stage 1	Stage 2	Stage 3
Local	6,034	12,118	25,638
1 worker	8,140	17,580	25,741
2 worker	8,243	12,168	15,438
3 worker	7,363	15,615	15,054
4 worker	8,153	12,142	15,610

As expected, by increasing the number of workers the duration tends to decrease. Although, good results are also obtained with the local configuration because dataset is not huge and the network communication overhead is absent.