Department of Information Engineering (DII)
M.Sc in Computer Engineering

Cloud Computing Project

Arsalen Bidani, Bashar Hayani, Pietrangelo Manco

Academic Year 2021/2022

Contents

1 Introduction

The goal of the project is to design a MapReduce algorithm to implement a set of Bloom Filters in both Hadoop and Spark.

A Bloom Filter is a space-efficient probabilistic data structure, which is used to establish if an element is a member of a set or not. It's possible to obtain false positives, but not false negatives.

The important parameters to consider while projecting such a Filter are the size of the input set (n), the size of the filter in the memory, given in bits (m), the optimal number of Hash Functions to use (k), and the False Positives Rate (p).

The input set here is given: the aim is to realize 10 Bloom Filters, each of which will store the films of the list with a fixated rounded evaluation. The Bloom Filter itself is a bit array initialized to m zeros. Adding an element means to set k of the m bits of the filter to 1, according to the Hash Functions.

This report contains the most important information about the Projectation of the Filter, the Algorithm and the associated Pseudo Code, the Hadoop and Spark implementations, and the Testing Phase.

The full codebase is avaiable at:

`https://github.com/Arselene/Cloud-Computing-Project`.

2 Projectation

There is a set of relations that hold between the Bloom Filter's parameters previously introduced.

Relation between m, n and p:

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

Relation between k, m and n:

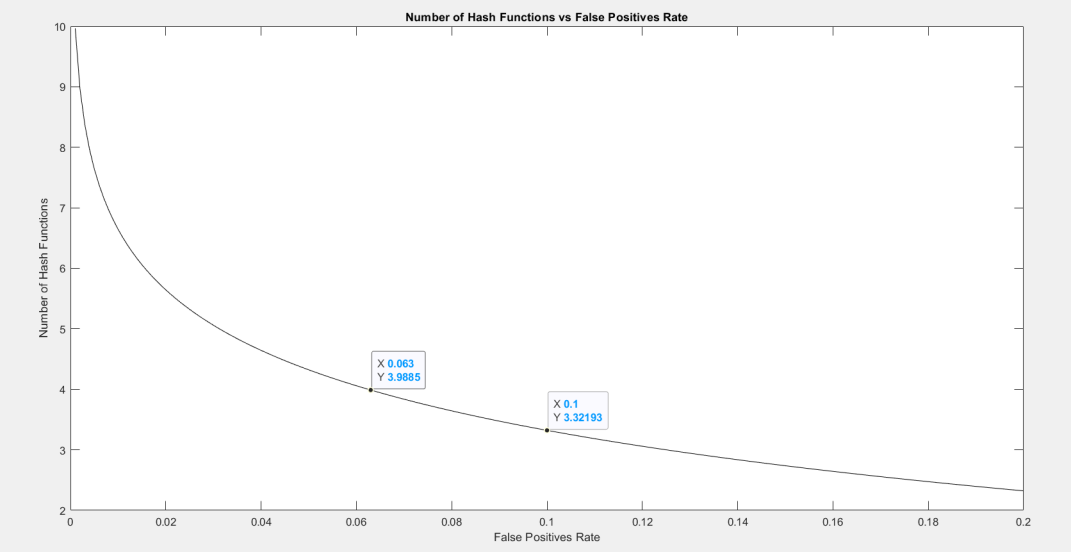
$$k = \frac{m \ln 2}{n} \tag{2}$$

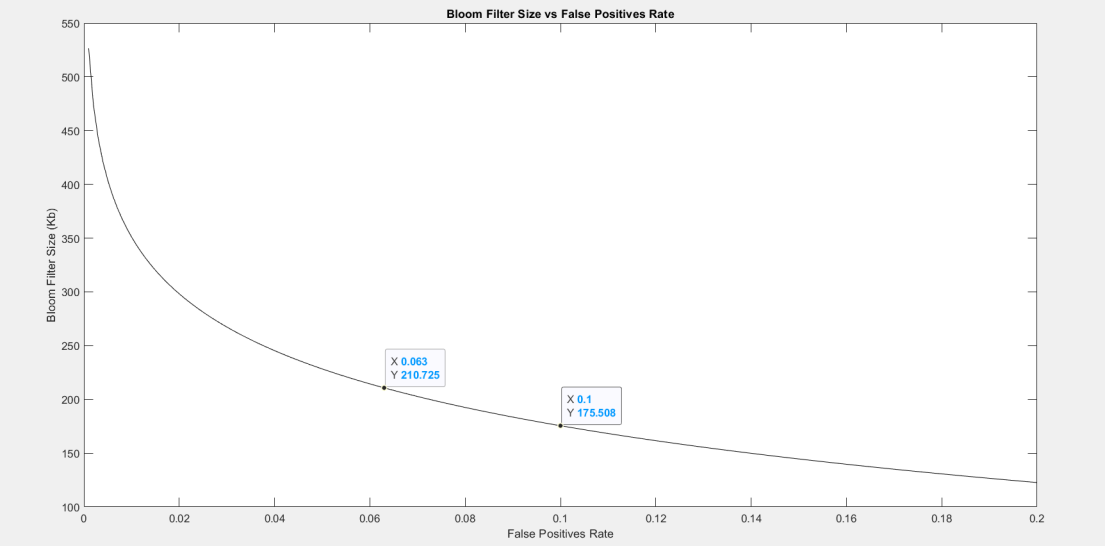
Relation between p, k, n, m:

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

According to the formulas, a trade-off between the False Positive Rate and the memory efficiency has to be made. Since n is fixated, and since k depends only on p, one between m and p has to be fixated as well, in order to compute the other quantities. Here p is chosen as a parameter. In order to have a reasonable balance between precision and memory expense, we estimated a range of good False Positives Rates between 5% and 10%.

The final choice was taken with the help of a MatLab Script, which was used in order to study the dependecies of both k and m with regard to p, producing the following graphs:





The chosen value for p is $\approx 0.063(6.3\%)$, as it's the closest possible value to an integer for the optimal number of Hash Functions in the feasible interval. This is showed clearly in the first picture. In the second curve, the kbytes size of the filter with respect to p is represented. A comparison between the chosen parameter and a sample p of 0.1 is made: the $\approx 37\%$ of precision increase comes with a drawback of the $\approx 17.3\%$ of memory increase, which is partially compensated by the fact that, by chosing $p = 0.1$, the closest optimal integer for the number of Hash Functions is 3, that is the $\approx 90\%$ of the actual optimal value for k.

3 Algorithm Pseudocode

Across the Bloom Filters creation and implementation and including the validation in which we put them to test phase, we were able to create three different MapReduce jobs.
Two first stages for the actual implementation of Bloom Filters and an additional one that was carried out in the validation phase.

Implementation Phase: Creation of a Bloom Filter for each rating

Job 1: Counts the number of movies per rating

Data: Movies data input that includes movie IDs and ratings
Result: Number of movies per each *rating*

```
1  class Mapper
2      method Map(lineid, line)
3          rating ← round( line.parse_rating() )
4          for all rating ∈ line do
5              EMIT(rating, 1)
6  class Reducer
7      method Reduce(rating, [c1, c2, ...])
8          sum ← 0
9          for all c ∈ [c1, c2, ...] do
10             sum ← sum + c
11         Emit(rating,sum)
```

Job 2: Bloom Filter creation and implementation

Data: Movies data input
Result: Bloom Filters in a bytearray format files

```
1  class MAPPER
2      method SETUP()
3          FP_rate ← Hadoop.config.get("False_Postive_Rate")
4          ratings_counts ← Hadoop.context.getFile("ratings_counts")
5          for all line ∈ ratings_counts do
6              rating ← line.parse_rating()
7              count ← line.parse_count()
8              bloomFilters[rating] ← bloomFilter(count,FP_rate)
9      method MAP(lineid, line)
10         rating ← round( line.parse_rating() )
11         movieId ← line.parse_movieId()
12         bf ← bloomFilters[rating]
13         indexes ← bf.getIndexes(movieId)
14         for all index ∈ indexes do
15             EMIT(rating,index)
16 class REDUCER
17     method SETUP()
18         FP_rate ← Hadoop.config.get("False_Postive_Rate")
19         ratings_counts ← Hadoop.context.getFile("ratings_counts")
20         for all line ∈ ratings_counts do
21             rating ← line.parse_rating()
22             count ← line.parse_count()
23             bloomFilters[rating] ← bloomFilter(count,FP_rate)
24     method REDUCE(rating, [i_1, i_2, ...])
25         bf ← bloomFilters[rating]
26         for all index ∈ [i_1, i_2, ...] do
27             bf.setIndex(index)
28         EMIT_SEPERATE_FILE(Null,bf)
```

Validation Phase: Putting each Bloom Filter on testing phase to view its statistics on the movies dataset

---

Data: Bloom Filters each in bytearray format & the movie dataset
Result: Stats and results of the Bloom Filter

```
1  class Mapper
2      method Setup()
3          bloomFilter_files ← Hadoop.getCacheFiles()
4          for all bf_file ∈ bloomFilter_files do
5              bloomFilters[rating] ← bf_file.read()
6      method Map(lineid, line)
7          rating ← round( line.parse_rating() )
8          movieId ← line.parse_movieId()
9          for all bf ∈ bloomFilter do
10             contained_in_bf ← bf.contained(movieId)
11             correct_bf ← (bf.Id = rating)
12             if (contained_in_bf and not correct_bf) do
13                 Emit(bf.Id, 0)
14             if (not contained_in_bf and correct_bf) do
15                 Emit(bf.Id, 1)
16 class Reducer
17     method Setup()
18         movieSum ← 0
19         ratingCounts ← Hadoop.context.getFile("ratings_counts")
20         for all line ∈ ratingCounts do
21             movieSum ← movieSum+ count
22             numMovies[rating] ← count
23     method Reduce(bf.id, [c1, c2, ...])
24         FPcounts ← 0
25         FNcounts ← 0
26         for all c ∈ [c1, c2, ...] do
27             If (c = 0)
28                 FPcount++
29             Else
30                 FNcount++
31         FP_percentage ← FPcount / (movieSum - numMovies(bf.id))
32         Emit(bf.id,FP_percentage)
```

---

4 Hadoop Implementation

The Bloom Filter Hadoop implementation was divided into 2 MapReduce jobs:

- job0: counts how many films have a given rating, rounded to the closest integer. The input data is splitted into 4 distinct mappers using NLineInputFormat, that take care of emitting (movie id, movie rating) pairs. The reducer performs the sum between movies that share the same rating, outputting (movie rating, number of movies) pairs.
- job1: creates 10 Bloom Filters as 10 multiple outputs, one for each possible movie rating. They are stored in byte files, which are used during the Testing Phase. Again, the input is splitted into 4 mappers, that take care of hashing each movie id; those ids are then emitted alongside the Bloom Filter Key for that specific rating. In the end, the reducer sets to 1 the corresponding bits for each hashed id in the associated Bloom Filter, and creates the 10 byte files containing them.

The Java implementation consists of the following classes:

- Create.java: This is the MapReduce jobs driver.
- CounterMapper.java: mapper for the first job; extracts each movie rate and rounds it. The map function is called on each line of a movie record, composed by the movie id and its rating.
- CounterReducer.java: reducer for the first job; it takes care of summing each record with a given rating, outputting (rating, number of movies) pairs.
- BFCreateMapper.java: mapper for the second job; it hashes each movie id in a bit sequence. The Setup() function initializes 10 empty Bloom Filters depending on the number of elements and probability rate. The Map() functions run on each movie record and outputs the hashed indexes for each Bloom Filter as (Bloom Filter Key, Hashed Index) pairs.

- BFCreateReducer.java: reducer for the second job; it sets to 1 the bits highlighted by the Hashed Id in the corresponding Bloom Filter, then it outputs them all in 10 separated byte files using MultipleOutputs class. The Setup() function is used to set up each of 10 empty bloom-filters using the number of elements and the probability rate. The Reduce() function sets the bits in each Bloom Filter and outputs it.

- BloomFilterOutputFormat.java: this custom output format class was built to enable us to output the Bloom Filters in a byte file format. Its usage is based on serialization and deserialization functions implemented in the BloomFilterWritable Class.

- BloomFilterWritable.java: this class defines the Bloom Filter as an object and manages all of its functionalities. It includes a constructor, methods for setting the indexes, to verify if an element is in it or not, to write it in a byte file and to read it out of such files, and also a method to perform the union between Bloom Filters, with a logical OR, of course since the class implemented the Writable class , it is fully serializable and deserializable (with custom write() and readFields() functions).

5 Spark Implementation

The Bloom Filter's Spark implementation was written using Python, almost the same structure with Hadoop was maintained as we have the bloom filter class as well.

The code is fully documented, and the MapReduce paradigm was applied just as in the pseudocode we provided, we also tried to improve the running time by exploiting the caching of **RDD** since it saves us from the recalculation and it improves the performance and also through exploring the serialization module of Python (Pickle).

```
1 import sys
2 import math
3 from bfClass import BloomFilter
4 from pyspark import SparkContext
5
6 falsePositiveRate=0.2
```

```
7
8
9
10 # A fuction to mimick java's half up rounding
11 def round_half_up(n, decimals=0):
12     multiplier = 10 ** decimals
13     return int(math.floor(n*multiplier + 0.5) / multiplier)
14
15
16 def map1(line):
17     result=line.split()
18     roundedRatings =round_half_up((float(result[1])))
19     return (roundedRatings,1)
20
21 def map2(line,bloomFilters):
22     result=line.split()
23     movieId=result[0]
24     rating=round_half_up(float(result[1]))
25     bloomFilter=bloomFilters.get(rating)
26     hashIndexes = bloomFilter.getHashIndexes(movieId)
27     return(rating,hashIndexes)
28
29 def reducer1(a, b):
30     return a+b
31
32 def reducer2(a, b):
33     for x in b:
34         a.append(x)
35     return a
36
37
38 # Usage : <input1> : Dataset file || <input2> : FP Probability Rate ||  <output> ... : BloomFilter
39
40 if __name__ == "__main__":
41
42
43     # Setup Spark Context with the bloom Filter class
44
45     sc = SparkContext("yarn", "BF Creation Phase",pyFiles=['bfClass.py'])
46
47     # Create Input file RDD and Read FP Rate
48     lines = sc.textFile(sys.argv[1])
49
50     # In order to mimick the NLineInputFormat on hadoop which
51     # takes N lines in a split , we are going to use repartion
52     # function to split the Input in equal 4 partitions
53     lines_splitted = lines.repartition(4)
54
55     # Reads the false positive rate from the arguments
56     falsePositiveRate = float(sys.argv[2])
57
58     # Persist the RDD in memory after the first calculation to save ... recomputations cost in second use
59     lines_splitted.cache()
60
61     # 1st Map-Reduce Job
62     # Computes How Many Movies Per Rating, output is list of (K,V) where K = ... rating & V = count
63     ratingCounts = lines_splitted.map(map1).reduceByKey(reducer1)
64     ratingCounts_InMemory = ratingCounts.collect()
65
66     # Save the rating counts pickled in HDFS for later use in test phase
67     ratingCounts.saveAsPickleFile("spark_ratingCounts")
68
69     # Initialize bloomfilters and assign them in a dictionnary
70
71     bloomFilters = {}
72     for (rating, count) in ratingCounts_InMemory:
73         bloomFilters[rating] = BloomFilter(count,falsePositiveRate)
74
75     # 2nd Map-Reduce job
76     # Mapper outputs a list of (K,V) , K = (BF key) & V = (list of an movieID ... hashed indexes)
77     mapped_rdd =lines_splitted.map(lambda line : map2(line,bloomFilters))
78
79     # Reducer return the aggregation of hashed indexes list for each BF
```

```
80      result_rdd = mapped_rdd.reduceByKey(reducer2).sortByKey().collect()
81
82      # Set each bloomfilter with the collected list of hashed indexes
83      for i in range(10):
84          for index in result_rdd[i][1]:
85              bloomFilters[i+1].setIndex(index)
86
87      # Save Bloomfilters into HDFS as a pickled file
88      bfRDD = sc.parallelize(list(bloomFilters.items()))
89      bfRDD.repartition(1).saveAsPickleFile(sys.argv[3])
```

The class 'bfClass' was implemented as follows.

```
1   import math
2   from bitarray import bitarray
3   import mmh3
4
5   ### Bloom filter class
6   class BloomFilter(object):
7
8       def __init__(self, items_count, fp_prob=0.2):
9
10
11          # False possible probability
12          self.fp_prob = fp_prob
13
14          # Size of bit array to use
15          self.size = self.round_half_up((-(items_count * ...
                math.log(fp_prob))/(math.log(2)**2)),0)
16
17
18          # number of hash functions to use
19          self.hash_count = math.ceil( (self.size/items_count) * math.log(2))
20
21          # Bit array of given size
22          self.bit_array = bitarray(self.size)
23
24          # initialize all bits as 0
25          self.bit_array.setall(0)
26
27      def add(self, item):
28          '''
29          Add an item in the filter
30          '''
31          hashIndexes = []
32          for i in range(self.hash_count):
33
34              # create hash for given item.
35              # i iterator work as seed to mmh3.hash() function
36              # With different seed, hashes created are different
37
38              hashIndex = abs(mmh3.hash(item, i) % self.size)
39              hashIndexes.append(hashIndex)
40
41              # set the bit to 1 in bit_array
42              self.bit_array[hashIndex] = True
43
44      def setIndex(self, index):
45          '''
46          Set an index in the filter to 1
47          '''
48          self.bit_array[index] = True
49
50      def getHashIndexes(self, item):
51          '''
52          an array of hash indexes of an item is returned
53          '''
54          hashIndexes = []
55          for i in range(self.hash_count):
56              hashIndex = abs(mmh3.hash(item, i) % self.size)
57              hashIndexes.append(hashIndex)
58
```

```
59          return hashIndexes
60
61      def union(self, bloomFilter):
62          '''
63          Performers an OR bitwise operation between two bloomfilters
64          '''
65          self.bit_array |= bloomFilter.bit_array
66
67      def check(self, item):
68          '''
69          Check for existence of an item in filter
70          '''
71          for i in range(self.hash_count):
72              hashIndex = abs(mmh3.hash(item, i) % self.size)
73              if self.bit_array[hashIndex] == False:
74
75                  # if any of bit is False then,its not present
76                  # in filter
77                  # else there is probability that it exist
78                  return False
79          return True
80
81      # A fuction to mimick java's half up rounding
82      @classmethod
83      def round_half_up(self, n, decimals=0):
84          multiplier = 10 ** decimals
85          return int(math.floor(n*multiplier + 0.5) / multiplier)
```

6 Testing Phase

In order to test the Bloom Filters implementations, a testing script was realized in both Hadoop and Spark. The tests have the goal to output the false negatives rate, which is expected to be 0, and the false positives rate, which is expected to be at around 6.3%. In order to count the false positives and false negatives, we perform the following steps for each movie id:

- a: we check if the movie id is contained one of the 10 Bloom Filters;
- b: we check if the movie id is in the correct Bloom Filter.

Depending on the results of these steps, we get the following outputs for the test:

For each Bloom Filter, the False Positive Rate calculation was carried out using the following formula, where the **True Negative Elements** is the total elements in dataset **minus** the number of elements in the specific bloomfilter:

$$FPRate = FPCount/TrueNegativeElements \tag{4}$$

Case	Output
$a == b$	Nothing
a && !b	False Positive
!a && b	False Negative

6.1 Hadoop Test

The Hadoop testing phase was implemented in Java using again MapReduce, just as the relative Bloom Filter implementation. It consists of a single MapReduce job, which takes as input the training set, the Bloom Filters and outputs the results. The Bloom Filters are saved in a cached manner on each node of the cluster for the duration of the test, then they get deleted. The Java implementation consists of the following classes:

- Test.java: This is the MapReduce job driver.
- BFTestMapper.java: the Mapper class. Its goal is to count the false positives and false negatives for each Bloom Filter. The Setup() function reads the Bloom Filters from the relative byte files. The Map() function performs the test on each movie record for each Bloom Filter and outputs results as (K,V) pairs, where K is the movie rating, V = 0 or 1; 0 stands for false positive and 1 stands for false negative. The function also checks if each movie is contained in one of the Bloom Filters or not.
- BFTestReducer.java: the Reducer class. It performs the sum of the false positives and false negatives for each Bloom Filter, and outputs them together with the relative rates. The Setup() function imports the rating count file; the Reducer() function performs the sum and outputs (K,V) pairs, where K is the movie ratinng and V the string containing the test results.

6.2 Hadoop Test Results

In this section we report the output string of the Bloom Filter Hadoop implementation and the relative test's output, as we can notice the we were able to achieve an excellent degree precision to the value that we initially chose of 0.063 for false positive rate.

We would like to note also that the false negatives are nonexistent throughout all of the Bloom Filters, and that proves that the implementation and the validation was correct across the two phases.

Movie count for each rate's Bloom Filter:

1 2544

2 6648

3 17819

4 43559

5 102433

6 219531

7 371114

8 354062

9 113157

10 16079

Testing Phase's results for each rate's Bloom Filter:

1 False Positive: 78859, False Negatives: 0, False Positive Rate: 6.34%;

2 False Positive: 78772, False Negatives: 0, False Positive Rate: 6.35%;

3 False Positive: 77720, False Negatives: 0, False Positive Rate: 6.32%;

4 False Positive: 75290, False Negatives: 0, False Positive Rate: 6.26%;

5 False Positive: 72353, False Negatives: 0, False Positive Rate: 6.32%;

6 False Positive: 64829, False Negatives: 0, False Positive Rate: 6.31%;

7 False Positive: 54952, False Negatives: 0, False Positive Rate: 6.27%;

8 False Positive: 56314, False Negatives: 0, False Positive Rate: 6.31%;

9 False Positive: 71662, False Negatives: 0, False Positive Rate: 6.32%;

10 False Positive: 77402, False Negatives: 0, False Positive Rate: 6.29%;

6.3 Spark Test

The Spark Test implementation was written using Python, using a map reduce paradigm that is depicted in the pseudocode above, the code is well commented and it is already abstracted enough by Spark.

We implemented the MurmurHash version 3.0 which is different that the one implemented in **Hadoop** which is the 2.0 because mainly it was nonexistent on Python libraries, this could results in small difference of results which we tried to mitigate even more with the implementation of a function that mimics the Java's half rounding.

```
1  import sys
2  import math
3  from bfClass import BloomFilter
4  from pyspark import SparkContext
5
6
7
8
9  # A fuction to mimick java's half up rounding
10 def round_half_up(n, decimals=0):
11     multiplier = 10 ** decimals
12     return math.floor(n*multiplier + 0.5) / multiplier
13
14
15 def mapF(line,bloomFilters):
16     result=line.split()
17     movieRating = round_half_up((float(result[1])))
18     movieId = result[0]
19     for i in range(1,11):
20         bfContained = bloomFilters[i].check(movieId);
21         correctBf= (int(movieRating) == (i))
22
23         if bfContained and not correctBf:  #False Postive
24             yield(i,0)
25         elif not bfContained and correctBf: # False Negative
26             yield(i,1)
27
28
29 # Usage : <input1> : Training set file || <input2> : BloomFilters Input ||  ...
       <output> : Test Phase Results
30
31 if __name__ == "__main__":
32
33
34     # Setup Spark Context with the bloom Filter class
35
36     sc = SparkContext("yarn", "BF Test Phase",pyFiles=['bfClass.py'])
37
38     # Create RDDs of input files and pickled rating counts
39     lines = sc.textFile(sys.argv[1])
40     bloomFilters_RDD = sc.pickleFile(sys.argv[2])
41     ratings_RDD = sc.pickleFile("spark_ratingCounts").collect()
42
43     # In order to mimick the NLineInputFormat on hadoop which
44     # takes N lines in a split , we are going to use repartion
45     # function to split the Input in equal 4 partitions
46     lines_splitted = lines.repartition(4)
47
48     # Insert ratings into a dictionnary , and sum the total num of movies
49     ratings = {}
50     movieSum=0
51     for (rating,count) in ratings_RDD:
52         movieSum+=count
53         ratings[rating]=count
54
55     # Assign bloomFilters from RDD to memory
56     bloomFilters_InMemory=bloomFilters_RDD.collect()
57
58
59     # Assign each BF into a dictionnary
60     bloomFilters={}
61     for (rating, bloomFilter) in bloomFilters_InMemory:
62         bloomFilters[rating] = bloomFilter
63
64     # Map each movie record to output False Postive and False Negative Results ...
           In Each BF
65     mappedValues = lines_splitted.flatMap(lambda line: mapF(line, bloomFilters))
66
67     # Persist the RDD in memory after the first calculation to save ...
           recomputations cost in second use
68     mappedValues.cache()
69
70     # Count for each BF the Number of FP and FN
71     numFP = mappedValues.filter(lambda x: x[1]==0).countByKey()
72     numFN = mappedValues.filter(lambda x: x[1]==1).countByKey()
73
```

```
74
75
76      # Insert each BF result in a dictionnary in string format
77      rates={}
78      for i in range(1,11):
79
80          fpRate= str(round((numFP[i] / (movieSum - ratings[i])) * 100, 2)) # ...
                False Postive Percentage Formula
81          fpCounts = str(numFP[i] if i in numFP else 0)
82          fnCounts = str(numFN[i] if i in numFN else 0)
83          rates[i]="False Positive : " + fpCounts + " False Negatives : " + ...
                fnCounts + " False Positive Rate : " + fpRate + "%"
84
85      # Output Test Results To HDFS
86      results = sc.parallelize(rates.items())
87      results.repartition(1).saveAsTextFile(sys.argv[3])
```

6.4 Spark Test Results

In this section we report the output string of the Bloom Filter Spark implementation test; and throughout we were able to get a very close results both to the value that we initially set and also the **Hadoop** Implementation results.

Testing Phase's results for each rate's Bloom Filter:

1, False Positive: 78859, False Negatives: 0, False Positive Rate: 6.16%;

2, False Positive: 78772, False Negatives: 0, False Positive Rate: 6.35%;

3 False Positive: 77720, False Negatives: 0, False Positive Rate : 6.27%;

4 False Positive: 75290, False Negatives: 0, False Positive Rate: 6.33%;

5 False Positive: 72353, False Negatives: 0, False Positive Rate: 6.30%;

6 False Positive: 64829, False Negatives: 0, False Positive Rate: 6.33%;

7 False Positive: 54952, False Negatives: 0, False Positive Rate: 6.32%;

8 False Positive: 56314, False Negatives: 0, False Positive Rate: 6.29%;

9 False Positive: 71662, False Negatives: 0, False Positive Rate: 6.32%;

10 False Positive: 77402, False Negatives: 0, False Positive Rate: 6.30%;