UNIVERSITÀ DI PISA

CLOUD COMPUTING

<Teletubbies>

BLOOM FILTER IMPLEMENTATION

Hadoop - Spark

Technical Report

Developed by**:**

- **Giuseppe Aniello** - badge : 643032 - email: g.aniello@studenti.unipi.it
- **Domenico Armillotta** - badge : 643020 - email: d.armillotta@studenti.unipi.it
- **Leonardo Bellizzi** - badge : 643019 - email: l.bellizzi@studenti.unipi.it
- **Edoardo Malaspina** - badge : 578355 - email: e.malaspina@studenti.unipi.it

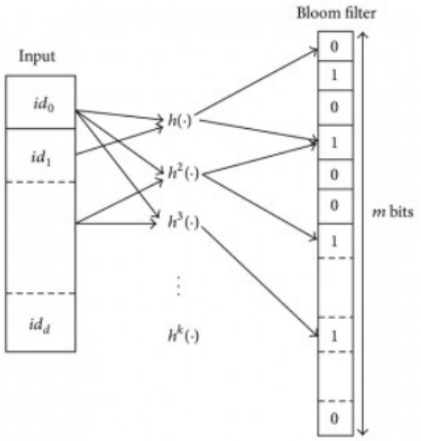
Summary

1. INTRODUCTION

A bloom filter is a space-efficient probabilistic data structure that is used for membership testing. False positive matches are possible, but false negatives are not - in other words, a query returns either “possibly in set” or “definitely not in set”.

Elements, inside this structure, can be added to the set, but not removed and more items are added, larger is false positives probability.

Data structure is shown below:



A bloom filter is a bit-vector with m elements. it uses k hash function to map n keys to the m elements of the bit-vector. Give a key id_1, every hash function h_n compute the corresponding output positions, and set the corresponding bit in that position to 1, if it is equal to 0.

A bloom filter has 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)

Following formulas were used to calculate parameters aforementioned:

$$m = -\frac{n \ln p}{(\ln 2)^2} \qquad k = \frac{m}{n} \ln 2 \qquad p \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

2. PSEUDOCODE

Pseudocode below refers to the first Map-Reduce job that counts the number of films for each rating and counts the number of bits in the bit vector.

```
Couting MR

MAPPER

function map (key, value) {

        tokens[0] = split_line[0]
        tokens[1] = round(split_line[1])

        key := tokens[1]
        value := tokens[0]
}

REDUCER

function reducer (key, values) {

        forEach (val in values){
                n++
        }

        m = (-n(log(pvalue)) / (log(2),2))+1
        write (key, m)
}
```

Following pseudocode refers to the implementation adopted both for Hadoop and Spark.

```
MAPPER

declare filtersf1[ ] --> filtersf10 [ ]
declare f1 --> f10

function map (key, value) {

        tokens[0] = split_line[0]
        tokens[1] = round(split_line[1])

        rating := tokens[1]

        if (rating == range (1.0, 10.0)) {
                addItem (filtersf1 --> filterf10, rating, pvalue)
        }

        f1 --> f10.write (filters1 --> filters10)
}
```

```
REDUCER

function reducer (key, values) {

        finalarray
        bitArray [] = bitArray[m+key]

        for (i=0; i<len(bitArray); i++){
                set(bitArray[1],0)
        }

        forEach (val in values){
                orFilter (bitArray, val)
        }
        finalArray := bitArray

}
```

3. HADOOP

For Hadoop implementation the aforementioned pseudocode was followed.

Two mapReduces were created, one to count the lines of the file and size the "m" of the blooms (the size of bloom). This first MapReduce saves the rating and associated size to an hdfs file.

The second Map-Reduce will start after the first one is completed because it needs “m” in order to build Bloom Filters.

3.1. COUNTING MAP-REDUCE

Mapper:

Mapper takes the data.tsv file as input, and reads each line, for each line it separates the rating part and the title_id part.

After splitting a write operation is done with:

- Key = rating
- Value = film_id

Reducer:

For each reducer a specific rating is assigned, it counts the number of movies with that given rating, and calculates the bloom size m of the rating.

Writes the rating to a file with the "write" associated with the value of the "m" dimension.

3.2. BlOOM FILTER MAP-REDUCE

Mapper:

Each mapper receives a fraction of the lines of the input file. Specifically 311922 lines to each mapper out of a total of 1,247,686, thanks to function "NLineInputFormat" and each mapper receives in input the dimension "m" for the creation of the blooms, from the conf of hadoop.

The "setup" function was used (a function that is executed only once at the beginning of the Map phase), to initialize the blooms of the ratings with the right size, in the IntWritable[] format, which is not serializable.

In the "map" function, each line it receives is read, and separates the title_id from the rating. The title_id is then assigned to the right bloom, with the *addItem* function, which takes care to insert the 1s in the right positions of the bloom.

Ones inside bloom filters depend on the number of k functions used, therefore on the specified p value.

The *addItem* function calculates the position using the *murmurhash* library and given that this hash method might returns negative values, the following formula was used $(hash\%\, dimension\, +\, dimension)\%\, dimension$ to avoid distribution changes.

After filling the IntWritable [] array with 1s, *.set ()* function was applied, to transform bloom filters in a serializable way i.e. IntArrayWritable, so that it can be passed to the reducer.

In the "cleanup" phase (function performed only once at the end of the mapper) bloom filters are written with a key rating assigned, furthermore these data will be passed to the reducer. So the output of each mapper is a bloom list with associated ratings.

Reducer:

Here mappers' output are taken as inputs, so each reducer will get several 1s-filled arrays of a given rating.

Logical *OR* function is exploited in order to merge filters coming in the reducer. The result of this operation is composed of 10 bloom filters, one for each rating, and for each bloom filter, the result is given by a merge of all 1s and 0s coming from mappers of its rating.

Afterwards, the output file is written in HDFS.

3.3. TESTS

In this phase following steps were followed:

1. Blooms have been created by reading the Reducer output text file from HDFS. The 10 blooms were then created, one for each rating, with the calculated size and filled with the ones read from the output file;

2. To calculate the FP, each line of the input file "data.tsv" was read, each film_id / rating was checked out in order to figure out if it was present inside the remaining 9 filters since its own filter wasn't taken into consideration.
   If the film was in another bloom filter with a different rating from the one seen in the original dataset, False Positive counter was incremented by one unit.
   To check if the film is present in the bloom, K- positions given by the same k hash functions used for creation, were checked.

3. After that False Positives were obtained the False Positive Rate was calculated for each filter following this formula:

$$fpr = \frac{fp[i]}{(numTotFilms - numFilmRating[i])}$$

3.4. HADOOP PERFORMANCE

Different values of the initial False Positive Rate were tried in order to check if bloom filters were in line with them.

In the table below results with different values are shown:

INPUT ERROR	1%		2%		5%		10%	
FILTER	FPR	FP	FPR	FP	FPR	FP	FPR	FP
Rating 1	0.82	10309	1.75	21859	4.48	55758	9.31	115965
Rating 2	0.95	11792	1.87	23220	4.84	60092	9.93	123280
Rating 3	0.96	11856	1.94	23875	4.99	61470	10.02	125791
Rating 4	0.97	11710	1.96	23696	5.04	60783	10.00	120991
Rating 5	0.96	11015	1.92	22063	4.99	57224	10.02	114846
Rating 6	0.93	9614	1.90	19581	4.89	50275	9.88	101632
Rating 7	0.90	7934	1.83	16100	4.77	50279	9.72	85231
Rating 8	0.87	7834	1.78	15970	4.64	41889	9.62	86001
Rating 9	0.89	10133	1.78	20236	4.68	53196	9.62	109213
Rating 10	0.87	10758	1.76	21727	4.67	53196	9.54	117533

Comparison tests were also carried out between spark and hadoop to compare the correct construction of the blooms and the consistency between the data.

Elapsed time for **First Map Reduce** job = 18 sec

Elapsed time for **Second Map Reduce** job = 41 sec

4. SPARK

In Spark the same reasoning of Hadoop was followed.

4.1. COUNTING MAP-REDUCE

Initially an RDD was created, reading from the input file and it was mapped to obtain title_id plus the rounded rating.

The input is partitioned into 4 parts.

From this RDD, the number of element(film_id) for each rating was computed applying serially:

.distinct() = take distinct values

.keys()

.map() = (mapper) to each rating associated with the movie is assigned 1 to do the sum in the next step

.reduceByKey(add) = (reducer) merge the values of each key by summing the value

.sortByKey() = sort ratings with associated dimension

fillM function was used to calculate the size based on the p value and the number of elements calculated in the previous function.

“m” values were needed to size bloom filters dimension.

4.2. BLOOM FILTER MAP-REDUCE

Mapper:

Mapper function was applied on 4 partitions.

The function reads each row and inserts for each movie, in the corresponding bloom, the various 1s based on the number of hash functions.

The hash function does not return a negative value, so there is no need to balance the distribution as in java.

The output of the mapper will be composed by 10 bloom filters filled with the movies given as input to the individual mapper according to the partition.

Reducer:

The reducer applies the logical OR between the various bloom filters at the output of the various mappers. "numPartitions=1" is set because results must be aggregated into one result.

Reducer output will be composed by 10 bloom filters coming from aggregation performed on mappers outputs via logical OR function.

4.3. TESTS

Tests work similarly to hadoop.

In this phase following steps were followed:

1. Blooms have been created by reading the Reducer output text file from HDFS. The 10 blooms were then created, one for each rating, with the calculated size and filled with the ones read from the output file;

2. To calculate the FP, each line of the input file "data.tsv" was read, each film_id / rating was checked out in order to figure out if it was present inside the remaining 9 filters since its own filter wasn't taken into consideration.
   If the film was in another bloom filter with a different rating from the one seen in the original dataset, False Positive counter was incremented by one unit.
   To check if the film is present in the bloom, K- positions given by the same k hash functions used for creation, were checked.

3. After that False Positives were obtained the False Positive Rate was calculated for each filter following this formula:

$$fpr = \frac{fp[i]}{(numTotFilms - numFilmRating[i])}$$

4.4. SPARK PERFORMANCE

INPUT ERROR	1%		2%		5%		10%	
FILTER	FPR	FP	FPR	FP	FPR	FP	FPR	FP
Rating 1	1.03	12884	2.07	25892	5.08	63270	10.08	125616
Rating 2	1.00	12461	2.00	24844	5.03	62540	10.02	124435
Rating 3	1.00	12375	2.05	25262	4.97	61204	10.10	124859
Rating 4	1.00	12133	2.02	24399	5.05	60870	10.05	121015
Rating 5	1.01	11657	2.00	22978	5.04	57813	10.07	115339
Rating 6	1.00	10302	2.02	20774	5.02	51613	10.07	103593
Rating 7	1.01	8896	2.01	17645	5.05	44312	10.11	88629
Rating 8	1.00	8962	2.03	18200	4.99	44654	10.08	90093
Rating 9	1.00	11401	2.02	22927	4.99	56709	10.05	113503
Rating 10	1.03	12706	2.01	24865	4.96	61096	10.04	123651

Elapsed time for **First/Second Map Reduce** job = 42 sec

5. FINAL CONSIDERATIONS

Hadoop code is quantitatively more verbose w.r.t Spark one, due to functional programming exploited with Spark.

Hadoop and Spark results are very similar, obviously the position of the 1's is different in the various bloom filters because of the different hash function used , mmh3 vs murmurhHash. The number of 1's is very similar, with little variation on the blooms with a larger number of movies.