Cloud Computing Project Report

Lorenzo Catoni Andrea Di Marco Leonardo Giovannoni
Marco Lucio Mangiacapre

Contents

1 Introduction

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set (i.e. to verify the presence of a username inside a website database).
It is usually represented as an array of bits of variable size. When a new element is added to the structure (in the example, a username), it is processed by k hash functions that will return as many values, each belonging to the filter size. Each bit corresponding to these indexes will then be set to 1.
If someone wanted to verify the presence of an element within the database, it's just needed the recalculation of the k hash functions to which the username was passed and the verification that the resulting indexes are set to one.
It is important to note that collisions are a possibility, since the number of filter indexes is limited and cannot be too large, thus leading to the possibility of false positives (in the example, a false positive occurs when the presence of a username that does not exist in the actual database is reported).
The objective of this project is to create an implementation of a bloom filter using the MapReduce paradigm, in particular exploiting the Hadoop and Spark frameworks.

1.1 The Dataset

The datasets used for this project are movie datasets offered by IMDb for personal and non-commercial use (`https://datasets.imdbws.com/`).
The website offered different kind of datasets, but for the construction of the filter, these ones were used:

- *title.basics.tsv.gz* - which contains the following information for titles:
  - *tconst* (*string*) - alphanumeric unique identifier of the title;
  - *titleType* (*string*) – the type/format of the title (e.g. movie, short, tvseries, tvepisode, video, etc);
  - *primaryTitle* (*string*) – the more popular title / the title used by the filmmakers on promotional materials at the point of release;
  - *originalTitle* (*string*) - original title, in the original language;
  - *isAdult* (*boolean*) - 0: non-adult title; 1: adult title;
  - *startYear* (*YYYY*) – represents the release year of a title. In the case of TV Series, it is the series start year;
  - *endYear* (*YYYY*) – TV Series end year. '\N' for all other title types;
  - *runtimeMinutes* – primary runtime of the title, in minutes;
  - *genres* (*string array*) – includes up to three genres associated with the title;
- *title.ratings.tsv.gz* – which contains the IMDb rating and votes information for titles
  - *tconst* (*string*) - alphanumeric unique identifier of the title;
  - *averageRating* – weighted average of all the individual user ratings;
  - *numVotes* - number of votes the title has received;

In order to optimize the dataset used, a join operation was performed to merge the two datasets and obtain only the information needed to construct the filter. The resulting dataset has 10 elements and contains the following fields:

- *tconst* (*string*) - alphanumeric unique identifier of the title;
- *originalTitle* (*string*) - original title, in the original language;
- *averageRating* – weighted average of all the individual user ratings;

As a second operation to optimize the data, the dataset was distributed by average grade, resulting in 10 datasets having the distribution shown in Fig.1

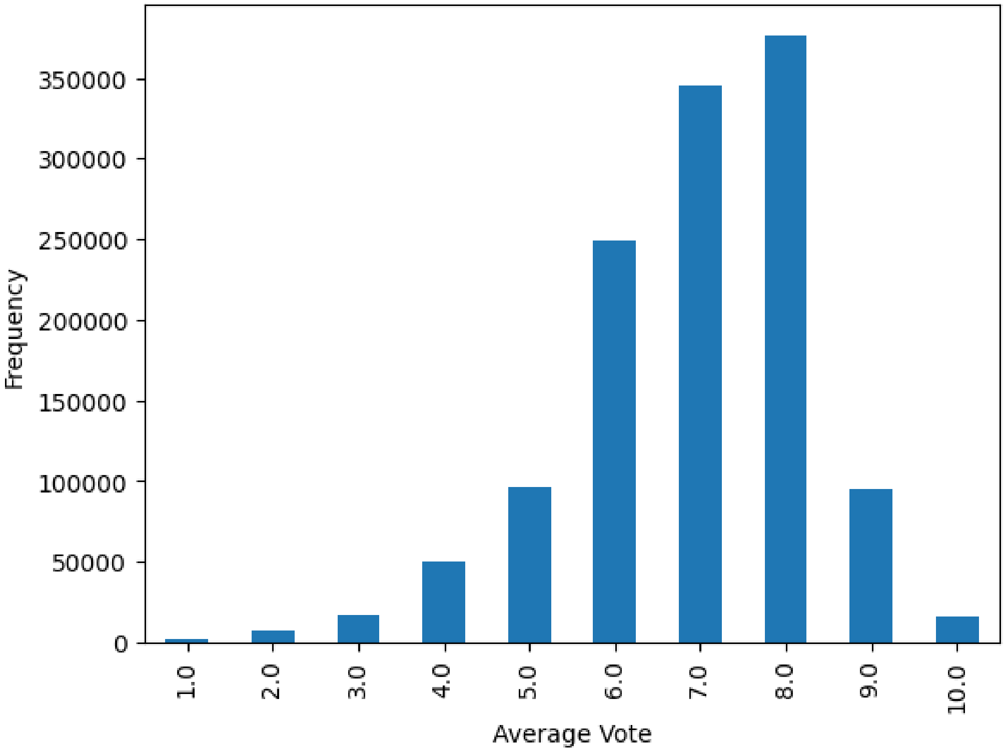


Figure 1: Values Distribution

1.2 MurmurHash

As described by the design specification, *MurmurHash* was used as the hash function for the filter, which is a non-cryptographic hash function suitable for general hash-based lookup. In particular the version used is a port of the The C version of MurmurHash 2.0 defined inside the package *org.apache.hadoop.util.hash.*

2 Hadoop Module

2.1 Implementation

The Hadoop module used in this project presents a four phases algorithm:

- **First Phase:** a join phase that merges the *title.basics.tsv* and the *title.ratings.tsv* databases by *id* (*tconst* parameter).
- **Second Phase:** a counting phase which will group every movie inside the database by their rounded vote.
- **Third Phase:** the third phase will generate *ten* files, each one containing every index of the bloom filters that must be set to 1. Every value will be obtained passing the movies' id by k murmurhash functions, each one generated by a different seed.
- **Fourth Phase:** the last phase will build the 10 bloom filter structures, obtained by taking as input the file returned by the third phase. Note that each bloom filter will be returned as a JSON file for the sake of readability

2.2 Join Phase

The *join phase* involves the use of two mappers and a reducer. Specifically, the first mapper (*JoinTitleMapper*), will be responsible for loading the dataset "*title.basics.tsv*" and extracting its "*originalTitle*" field line by line.

Algorithm 1 Method JoinTitleMapper(Filename, Movies)

```
for all movies m ∈ Movies do
    id ← m.id
    title ← m.title
    EMIT (id, label-title)
end for
```

The second mapper (*JoinRatingMapper*), will instead take care of loading the dataset "*title.ratings.tsv*" and extracting its "*averageRating*" field line by line.

Algorithm 2 Method JoinRatingMapper(Filename, Movies)

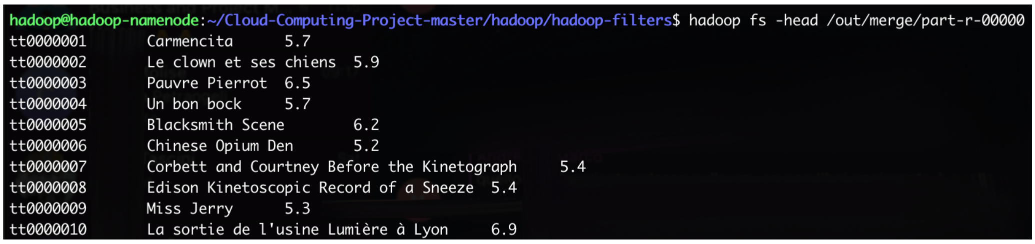
```
for all movies m ∈ Movies do
    id ← m.id
    score ← m.score
    EMIT (id, label-score)
end for
```

Finally, the reducer (*JoinReducer*) will be in charge of joining the two extracted datasets on the *id* (*tconst* field) of each movie. Both title and score are saved on a support class *Res* which stores the two fields as strings (note that the method supports multiple inputs thanks to the usage of the *MultipleInputs* class offered by hadoop).
Note that the reducer is able to distinguish between a title dataset line and a score one thanks to the label field given as output in both mappers.

Algorithm 3 Method JoinReducer(title_list, value_list, Movies)

```
Res ← new Res
counter ← 0
vote ← false
title ← false
for all split s ∈ title_list and ∈ value_list do
    if counter++ is equal to 2 then
        return failure
    else
        str ← v
        split ← str
        if s[0] is equal to a title dataset line then
            if title is doubled then
                return failure
            end if
            if title is true then
                Res.title ← s[1]
            end if
        end if
        if s[0] is equal to a rating dataset line then
            if rating is doubled then
                return failure
            end if
            if vote is true then
                Res.vote ← s[1]
            end if
        end if
    end if
end for
for all movies m ∈ Movies do
    if counter++ is equal to 2 then
        EMIT (id, score, title)
    end if
end for
```

If possible as a last optimization, the reducer exploits a custom hadoop counter to preemptively count the number of movies that must be represented within each bloom filter.
After phase one the output produced will be put in the *merge* folder and will be as shown in Fig.2



```
hadoop@hadoop-namenode:~/Cloud-Computing-Project-master/hadoop/hadoop-filters$ hadoop fs -head /out/merge/part-r-00000
tt0000001       Carmencita      5.7
tt0000002       Le clown et ses chiens  5.9
tt0000003       Pauvre Pierrot  6.5
tt0000004       Un bon bock     5.7
tt0000005       Blacksmith Scene        6.2
tt0000006       Chinese Opium Den       5.2
tt0000007       Corbett and Courtney Before the Kinetograph     5.4
tt0000008       Edison Kinetoscopic Record of a Sneeze  5.4
tt0000009       Miss Jerry      5.3
tt0000010       La sortie de l'usine Lumière à Lyon     6.9
```

Figure 2: Join Phase Output

2.3 Counting Phase

The *Counting Phase* involves the use of one mapper and a reducer. Specifically, the mapper (*CounterMapper*), will take every elements of the merged dataset, then each value will be rounded up and returned as a <vote, 1> pair by exploiting a procedure similar to the WordCount algorithm.
In order to optimize the execution of the task, an associative array is used in the map phase, expoliting the In-Mapper approach.

Algorithm 4 Method CounterMapper(Merged_Movie_Dataset, Movies)

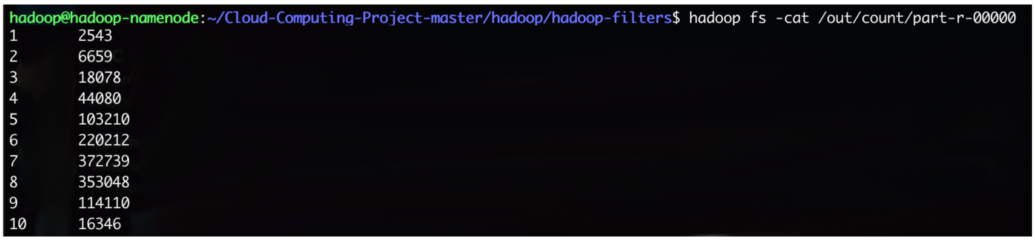
```
H ← new AssociativeArray
for all movies m ∈ Movies do
    rounded_score ← round(m.score)
    H(rounded_score) ← H(rounded_score)+1
end for
EMIT (rounded_score, H(rounded_score))
```

The reducer (*CounterReducer*) will merge every movie with the same score, increasing the counter of the <vote, count> pair for each match found.

Algorithm 5 Method CounterReducer(rounded_score, counts[c_1...c_n])

```
sum ← 0
for all counts c ∈ counts[$c_1$...$c_n$] do
    sum ← sum + c
    EMIT (rounded_score, count sum)
end for
```

At the end of the second phase, the output will be be put in the *count* folder and will be as shown in Fig.3



```
hadoop@hadoop-namenode:~/Cloud-Computing-Project-master/hadoop/hadoop-filters$ hadoop fs -cat /out/count/part-r-00000
1       2543
2       6659
3       18078
4       44080
5       103210
6       220212
7       372739
8       353048
9       114110
10      16346
```

Figure 3: Counting Phase Output

2.4 Hash Generation Phase

The third phase will use a single mapper-reducer pair. The mapper (*UniqueMapper*) will be used to generate the *k* <average_vote, hash> pairs for each of the 10 bloom_filters (one per vote). Note that before the mapper is executed, it will receive as input a hashmap containing the complete list of movies in the database, with complete removal of duplicates. This is taken care of by the *getMapIIFromPairs* method, whose code is described below:

```java
/*
* key value in the map are put in consecutive positions,
* even index: rounded vote
* odd index:  movie count
*/
private static Map<Integer,Integer> getMapIIFromPairs(int[] pairs) {
    Map<Integer,Integer> ans = new TreeMap<>();
    for (int i=0; i<pairs.length; i+=2) {
        ans.put(pairs[i], pairs[i+1]);
    }
    return ans;
}
```

Listing 1: Code responsible to generate the hashmap

At the end of the execution of the mapper, it will output a list of integer pairs, (defined by overriding the *IntPair*class), in which will be added the bloom filter and the index that must be set to 1.
Notice that before passing the movie id to the hash function, the string is converted to lowercase characters only using the standard UTF-8 encoding, so as to avoid possible errors during the final testing phase.

Also note that the k number of hash seeds and the bloom filter size m are computed using formulas derived from those provided in the design specifications.

$$k = -\frac{ln(p)}{ln(2)} \tag{1}$$

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

Algorithm 6 Method UniqueMapper(Movies, hash_seeds[k_1,...k_n])

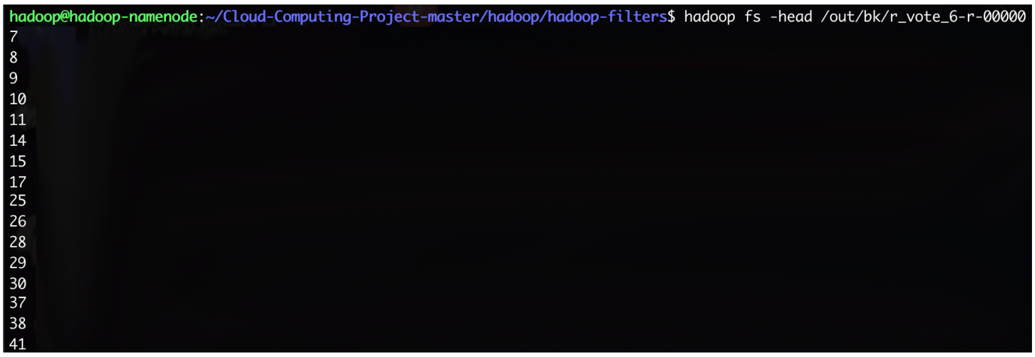
```
hasher ← MurmurHash.getInstance()
outk ← new IntPair
for all movies m ∈ Movies do
  outk.first ← Math.round(m.vote)
  bloom_size ← movie_bloom_size.get(rounded_vote)
  bytes ← new Byte Array
  bytes ← line[0] converted as UTF_8 standard charset
  for all hash_seeds ∈ hash_seeds[$k_1$,...$k_n$] do
    outk.second ← (hasher.hash(bytes, k)%bloom_size)+bloom_size)%bloom_size
    EMIT (outk)
  end for
end for
```

It will finally be the job of the reducer (*UniqueReducer*) to split this document into 10 different files and return them as output (the *MultipleOutputs* class provided by hadoop is used to support this operation).

Algorithm 7 Method UniqueReducer(bloom_numbers, hash_intPairs_result[i_1...i_n])

```
for all IntPairs i ∈ hash_intPairs_result[$i_1$...$i_n$] do
  for each bloom_filters_documents do
    EMIT (i.first) {▷ Correct bloom filter is selected from the i.second field}
  end for
end for
```

At the end of this phase, the 10 output files will be named *r_vote_**x**-r-00000* (where **x** is the average vote that identifies the filter) and put in the *bk* folder and will be as shown in Fig.4



```
hadoop@hadoop-namenode:~/Cloud-Computing-Project-master/hadoop/hadoop-filters$ hadoop fs -head /out/bk/r_vote_6-r-00000
7
8
9
10
11
14
15
17
25
26
28
29
30
37
38
41
```

Figure 4: Hash Generation Phase Output

2.5 Building Phase

The last phase will deal with the construction of the *10* bloom filters using a mapper and a reducer. The mapper (*BFMapper*) will take the *10* files produced by the Hash Generation Phase as input, through which it will extract the fields needed to create the filter. It will produce the <*rounded_vote*, *index_to_set*> pairs.

```
Algorithm 8 Method BFMapper(hash_index_files[$k_1$,...$k_n$])
  for all file f $\in$ hash_index_files[$k_1$,...$k_n$] do
    outk $\leftarrow$ vote_value from f.filename {$\triangleright$ will take the correct index based on the actual filename}
    outv $\leftarrow$ filter_index from each file row
    EMIT (outk, outv)
  end for
```

The reducer (*BFReducer*) will take these pairs as input and create *10 JSON* files in which the bloom filters will be contained. To perform this operation a class *BloomFilter* having this structure has been created:

- *private int rating*: will store the score referred by the bloom filter;
- *private int N*: will store the bloom filter size;
- *private BitSet bs*: will store the actual bloomfilter structure;
- *private int[] seeds*: will store all seeds used to generate the hash values;

After the creation of the *BloomFilter* object, the *JSON* file will be created using an ad hoc method *toJSON()*:

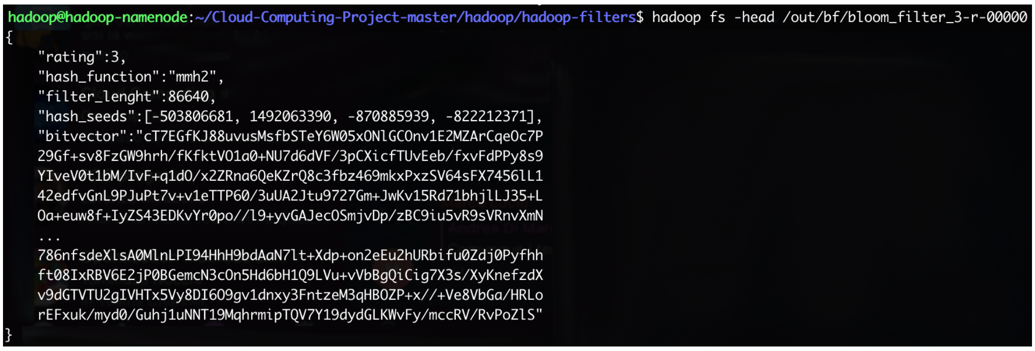
```java
public String toJson() {
    StringBuilder str_builder = new StringBuilder(512 << 8).append("{");
    // add vote
    str_builder.append("\"rating\":").append(Integer.toString(rating)).append(",");
    // specify hash function
    str_builder.append("\"hash_function\":\"mmh2\"").append(",");
    // filter lenght
    str_builder.append("\"filter_lenght\":").append(Integer.toString(N)).append(",");
    // add hash seeds
    str_builder.append("\"hash_seeds\":").append(Arrays.toString(seeds)).append(",");
    // add bit vector
    str_builder.append("\"bitvector\":\"").append(getBase64()).append("\"");
    return str_builder.append("}").toString();
}
```

Listing 2: Code responsible for the creation of the JSON file

Finally, all 10 filters will be exported using the already mentioned *MultipleOutputs* class provided by hadoop.

```
Algorithm 9 Method BFReducer(votes[1...10], hash_indexes[$i_1$...$i_n$])
  for all votes v $\in$ votes[1...10] do
    bf $\leftarrow$ new BloomFilter
    for each index i $\in$ hash_indexes[$i_1$...$i_n$] do
      bf.set[i] {$\triangleright$ set the index i of the bloom filter array to 1}
    end for
    EMIT (bf.toJson())
  end for
```

At the end of this phase, the 10 Bloom Filter will be named *bloom_filter_**x**-r-00000* (where **x** is the average vote that identifies the filter) and put in the *bf* folder and will be as shown in Fig.5



```
hadoop@hadoop-namenode:~/Cloud-Computing-Project-master/hadoop/hadoop-filters$ hadoop fs -head /out/bf/bloom_filter_3-r-00000
{
    "rating":3,
    "hash_function":"mmh2",
    "filter_lenght":86640,
    "hash_seeds":[-503806681, 1492063390, -870885939, -822212371],
    "bitvector":"cT7EGfKJ88uvusMsfbSTeY6W05xONlGCOnv1E2MZArCqeOc7P
    29Gf+sv8FzGW9hrh/fKfktVO1a0+NU7d6dVF/3pCXicfTUvEeb/fxvFdPPy8s9
    YIveV0t1bM/IvF+q1dO/x2ZRna6QeKZrQ8c3fbz469mkxPxzSV64sFX7456lL1
    42edfvGnL9PJuPt7v+v1eTTP60/3uUA2Jtu9727Gm+JwKv15Rd71bhjlLJ35+L
    Oa+euw8f+IyZS43EDKvYr0po//l9+yvGAJecOSmjvDp/zBC9iu5vR9sVRnvXmN
    ...
    786nfsdeXlsA0MlnLPI94HhH9bdAaN7lt+Xdp+on2eEu2hURbifu0Zdj0Pyfhh
    ft08IxRBV6E2jP0BGemcN3cOn5Hd6bH1Q9LVu+vVbBgQiCig7X3s/XyKnefzdX
    v9dGTVTU2gIVHTx5Vy8DI6O9gv1dnxy3FntzeM3qHBOZP+x//+Ve8VbGa/HRLo
    rEFxuk/myd0/Guhj1uNNT19MqhrmipTQV7Y19dydGLKWvFy/mccRV/RvPoZlS"
}
```

Figure 5: Building Phase Output

2.6 Code Optimization

In order to optimizatize the code, both Hash Generation Phase and Building Phase can be merged in a single MapReduce job which will be composed by:

- **MapFunction**: will include the Generation phase and so will create the <rounded vote, index to set> pairs (*UniqueMapper*).
- **ReduceFunction**: the reduce function will create the JSON files containg the filters (*BFReducer*).

As a further optimization, where possible *IntWritable* objects have been replaced so that space and memory can be saved. In particular:

- **ByteWritable**: was used for the keys that associate the value to the filter, as the *4* bytes size offered by *IntWritable* is excessive and *1* byte was more than enough to represents these values.
- **VIntWritable**: was used for the indexes to be set in the filter. In fact, this class allows an integer to be saved on fewer bytes than the fixed *4* bytes offered by *IntWritable*.

3 Spark Module

3.1 Implementation

The Spark module is significantly simpler than the hadoop one, we can basically split the execution in three phases:

- **First Phase:** reading the dataset from the hdfs and convert it to a spark rdd
- **Second Phase:** counting the number of movies per vote
- **Third Phase:** building the bloom filters

Figure 6 shows an high level overview of the data flow in spark.

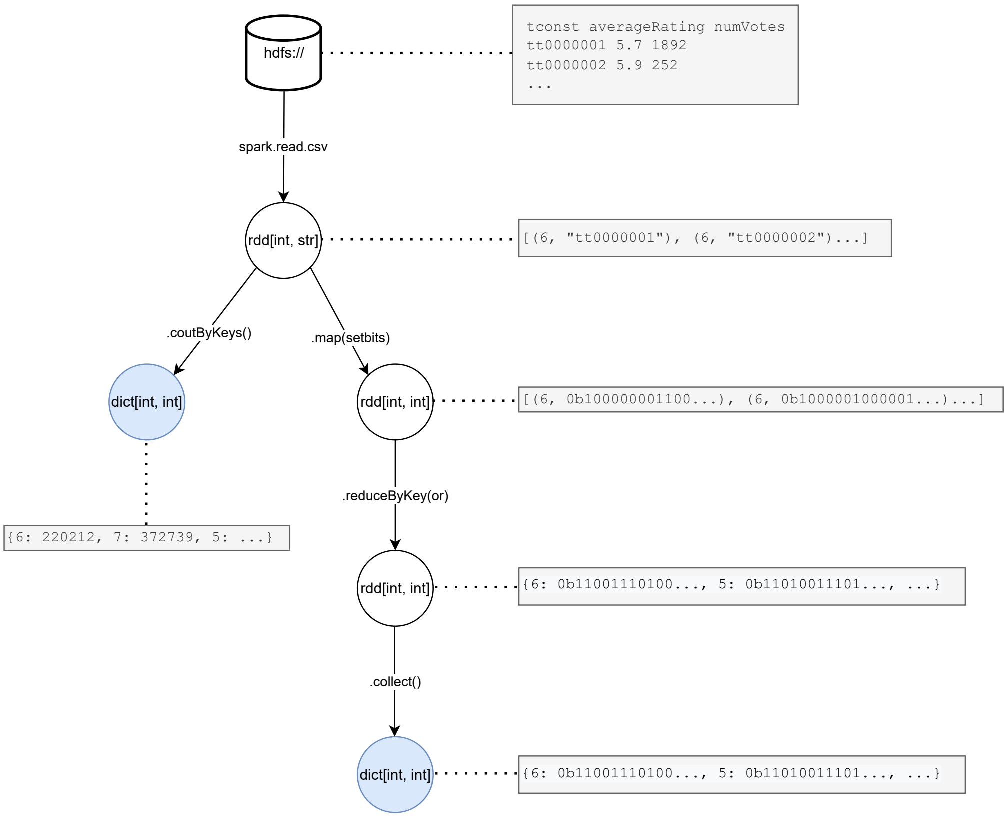

hdfs://
tconst averageRating numVotes
tt0000001 5.7 1892
tt0000002 5.9 252
...
spark.read.csv
rdd[int, str]
[(6, "tt0000001"), (6, "tt0000002")...]
.coutByKeys()
.map(setbits)
dict[int, int]
{6: 220212, 7: 372739, 5: ...}
rdd[int, int]
[(6, 0b100000001100...), (6, 0b1000001000001...)...]
.reduceByKey(or)
rdd[int, int]
{6: 0b11001110100..., 5: 0b11010011101..., ...}
.collect()
dict[int, int]
{6: 0b11001110100..., 5: 0b11010011101..., ...}

Figure 6: Overview of the spark process

3.2 Reading Phase

The input file is parsed automatically via the `spark.read.csv` and converted to a rdd of tuples (`rounded_rating`, `title_id`). See code snippet 3.

```python
def load_dataset(ratings_file: str) -> pyspark.rdd.RDD:
  """ Load dataset and return a rdd in the form ‘(rating, title)‘
  """

  # load ratings from file
  ratings_schema = StructType([StructField("tconst", StringType(), False), StructField("averageRating", FloatType(), False), StructField("numVotes", IntegerType(), False)])
  df_ratings = spark.read.csv(ratings_file, sep=’\t’, schema=ratings_schema, header=True)\
    .withColumn(’averageRating’, pyspark_round(pyspark_col("averageRating")).cast(ByteType()))\
    .withColumnRenamed("averageRating", ’roundedRating’)\
    .select(’roundedRating’, ’tconst’)

  return df_ratings.rdd
```

Listing 3: Code responsible for reading the input file

3.3 Counting Phase

This phase is very simple because the function `countByKeys()` does exactly what we need. See code snippet 4.

```python
# count number of movies per rating
num_movies_per_rating = dataset.countByKey()
print(num_movies_per_rating)
# -> defaultdict(<class 'int'>, {6: 220212, 7: 372739, 5: 103210, 4: 44080, 3: 18078, 8: 353048, 9: 114110, 2: 6659, 1: 2543, 10: 16346})
```

Listing 4: Code responsible for counting the number of movies per rating

3.4 Building Phase

In this phase we build the bloom filters, we exploit the fact that the builtin `int` data type has arbitrary precision to implement the internal bit vector of the bloom filter. In the first step: `map`, we calculate k times the hash function for every title_id and set the bits in the bit vector. In the second step: `reduceByKey` we aggregate all the bit vectors related to the same rating with a binary or operation. See code snippet 5

```python
# function that takes a pair (rating, title) and returns a pair (rating, int with the bits set to one at the positions specified by the hash function)
def setbits(x: Tuple[int, str]) -> Tuple[int, int]:
  rating, title = x
  m = BloomFilter.optimal_m(num_movies_per_rating[rating], epsilon)
  k = BloomFilter.optimal_k(num_movies_per_rating[rating], epsilon)
  rv = 0
  for seed in range(k):
    i = murmurhash2(title.lower().encode("utf-8"), seed) % m
    rv |= 1<<i
  return rating, rv

# calculate bloom filter
bf_tmp = dataset.map(setbits)\
  .reduceByKey(int.__or__)\
  .collect()
```

Listing 5: Code responsible for building the bloom filters

4 Testing Phase

4.1 Hadoop

4.1.1 Hadoop Module

The testing phase for the Hadoop module involves the use of a *MapReduce* function that will be responsible for fetching JSON files containing the Bloom Filters and checking the false positive rate.
The mapper (*TestMapper*), in the setup phase will pick up the filters from the *JSON* files: these will be *deserialized* and placed inside a *TreeMap*.
Following setup, the map function will check, for each movie in the db, which bloom filter it is in and then check that the same movie it's not present in any other filter.
To state the presence of a false positive, an IntPair class has been used again. The class presents this structure:

- **first**: stores the count of movies tested;
- **second**: stores the number of objects found as false positives;

so, if the filter will find the movie as a false positive, both first and second value will be set to one and returned as a value; otherwise only the first field will be set to 1 and the second field will be 0.
In any case the map function will output the <*movie_score*, *IntPair*> pair.

Algorithm 10 Method TestMapper(Movies, bloom_filters[1...10])

```
for all movies m ∈ Movies do
  bytes ← new Byte Array
  bytes ← m.id converted as UTF_8 standard charset
  rVote ← round(m.score)
  if m ∉ the right filter then
    return failure
  end if
  for all bloom filters b ∈ bloom_filters[1...10] do
    {▷ checks only the filters that should not include the movie}
    if m ∉ b then
      outk ← b.avg_score
      outv ← new IntPair
      outv.first ← 1 {▷ increase checked} elements
      if m ∈ b then
        {▷ m is a false positive}
        outv.second ← 1
      else
        outv.second ← 0
      end if
    end if
    EMIT (outk, outv)
  end for
end for
```

The reducer (*TestReducer*), on the other hand, will take the pairs returned by the mapper as input and will increment them if it finds duplicate keys.
In this way, the output of the reducer will have a complete report of the items tested and of the rate of false positives for each filter.

Algorithm 11 Method TestReducer(rounded_score, int_pairs[$i_1 ... i_n$])

```
for all bloom filters b ∈ bloom_filters[1...10] do
  outv ← new IntPair
  outv.first ← 0
  outv.second ← 0
  for all int_pairs i ∈ int_pairs[$i_1 ... i_n$] do
    outv.first ← outv.first + 1
    if i.second is 1 then
      outv.second ← outv.second + 1
    end if
  end for
  EMIT (b, outv)
end for
```

4.1.2 Results Analysis

In order to evaluate the false positive rate, this formula has been used:

$$p = \frac{false\_positives}{processed\_elements} \tag{3}$$

First Test: For the first application test, a value of p = 0.1 was set as target. The results obtained are listed in the table below:

Rounded Vote	Processed Elements	False Positives	False Positives Rate
1	1247481	88580	0.071007095
2	1243424	122464	0.09848933
3	1232211	115750	0.09393683
4	1206287	110456	0.091566935
5	1147333	98207	0.0855959
6	1029738	78023	0.07576976
7	877897	56052	0.06384803
8	895542	50671	0.056581378
9	1136503	74927	0.06592768
10	1233863	83673	0.06781385

Second Test: For the second application test, a value of p = 0.01 was set as target. The results obtained are listed in the table below:

Rounded Vote	Processed Elements	False Positives	False Positives Rate
1	1247481	12301	0.009860671
2	1243424	12119	0.009746474
3	1232211	12518	0.010158975
4	1206287	12174	0.010092125
5	1147333	11699	0.010196691
6	1029738	10386	0.0100860605
7	877897	9025	0.01028025
8	895542	8861	0.009894567
9	1136503	11362	0.009997334
10	1233863	12566	0.010184275

Third Test: For the third and final application test, a value of p = 0.001 was set as target. The results obtained are listed in the table below:

Rounded Vote	Processed Elements	False Positives	False Positives Rate
1	1247481	1230	0.000985987
2	1243424	1316	0.0010583679
3	1232211	1241	0.0010158975
4	1206287	1142	0.0009467067
5	1147333	1121	0.0009770485
6	1029738	1056	0.0010255036
7	877897	872	0.0009932828
8	895542	924	0.0010317774
9	1136503	1144	0.0010065966
10	1233863	1261	0.0010219936

4.2 Spark

4.2.1 Spark Code

To check the false positive rate we took a sample of the input data and checked how many movies that should not be in the bloom filter actually are. See code snippet 6.

```
testset = dataset.sample(withReplacement=False, fraction=0.05).collect()
total, falsep = 0, 0
for t_rating, t_title in testset:
  if t_rating == rating:
    assert t_title.lower().encode("utf-8") in bf
    continue
  if t_title.lower().encode("utf-8") in bf:
    falsep += 1
  total += 1
print(falsep/total)
```

Listing 6: Code responsible for testing the false positive rate

4.2.2 Results Analysis

First Test: For the first application test, a value of p = 0.1 was set as target. The results obtained are listed in the table below:

Rounded Vote	Processed Elements	False Positives	False Positives Rate
1	6772	61939	0.1093
2	6256	61754	0.1013
3	6356	61191	0.1039
4	6202	59906	0.1035
5	5780	56958	0.1015
6	5220	51244	0.1019
7	4346	43632	0.0996
8	4513	44537	0.1013
9	5775	56267	0.1026
10	6326	61292	0.1032

Second Test: For the first application test, a value of p = 0.01 was set as target. The results obtained are listed in the table below:

Rounded Vote	Processed Elements	False Positives	False Positives Rate
1	591 62303	0.0095	0.1093
2	611 62077	0.0098	0.1013
3	644 61530	0.0105	0.1039
4	570 60212	0.0095	0.1035
5	562 57197	0.0098	0.1015
6	550 51527	0.0107	0.1019
7	453 43749	0.0104	0.0996
8	459 44973	0.0102	0.1013
9	580 56688	0.0102	0.1026
10	634 61569	0.0103	0.1032

Third Test: For the first application test, a value of p = 0.001 was set as target. The results obtained are listed in the table below:

Rounded Vote	Processed Elements	False Positives	False Positives Rate
1	75	62729	0.0012
2	60	62502	0.0010
3	72	61920	0.0012
4	71	60588	0.0012
5	53	57731	0.0009
6	39	51775	0.0008
7	39	44065	0.0009
8	47	45223	0.0010
9	58	57164	0.0010
10	50	62034	0.0008

5 Conclusions

As shown by the tables, all p-values are fairly close to the imposed thresholds, so the result of the project was satisfactory and in line with the requirements provided.