UNIVERSITÀ DI PISA

BLOOM FILTERS in MAP-REDUCE

Computer Engineering / Artificial Intelligence and Data Engineering

Cloud Computing

Fabiano Pilia

Riccardo Sagramoni

Emanuele Tinghi

Veronica Torraca

A.Y. 2021-2022

Table of Contents

1. Introduction

A bloom filter is a space efficient probabilistic structure used for membership testing. The aim of this project is to implement and design a bloom filter and the algorithm that keeps it updated both in Spark and in Hadoop.

1.1. Project Specification

Bloom filter, given their space efficient nature, are not perfect. Actually, it has a fixed size, so, independently of how many keys you will pass to it later, you even get to decide how much memory you want it to use. As already said, this comes at a certain cost, which is the probability of having false positives. But there can never be false negatives.

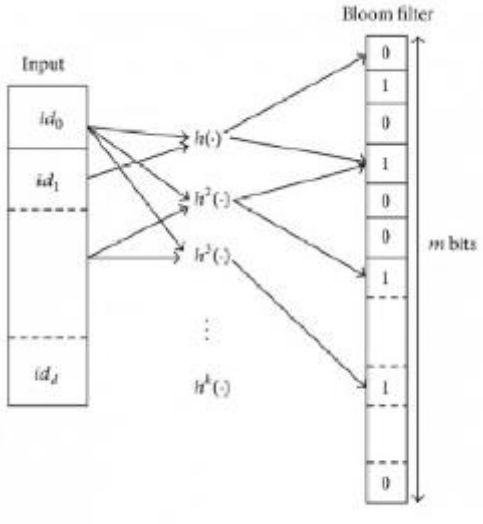


Figure 1: *Bloom filter example.*

A bloom filter is a bit-vector with m elements. It uses k hash functions to map n keys to the m elements of the bit-vector. Given a key *id*, every hash function $(h_1, \dots, h_k)$ computes the output positions and sets the corresponding bit in that position to 1, if it is equal to 0.

Let's consider a bloom filter with 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 ad 1)

The relation between these values can be expressed as:

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

$$k = \frac{m}{n} * \ln(2)$$

$$p \approx \left(1 - e^{-\frac{k*n}{m}}\right)^k$$

To design a bloom filter with a given false positive rate *p*, you need to estimate the number of keys *n* to be added to the bloom filter, then compute the number of bits *m* in the bloom filter and finally compute the number of hash functions *k* to use.

1.2. Project Goal

The goal of this project is to build a bloom filter over the rating of movies listed in this IMDB dataset. The average ratings must be rounded to the closest integer value and every rating value must have its own bloom filter.

The bloom filter construction must be implemented following the MapReduce workflow, both in Hadoop and Spark frameworks. Moreover, the exact number of false positive for each rating must be computed.

In the Hadoop implementation the following classes must be followed:

- org.apache.hadoop.mapreduce.lib.input.NLineInputFormat: splits N lines of input as one split
- org.apache.hadoop.util.hash.Hash.MURMUR_HASH: the hash function family to use

In the Spark implementation analogous classes must be used/implemented.

2. Pseudocode

In our project we choose to design two types of MapReduce applications, both in Hadoop and Spark implementations:

- The **Builder** application: to build the actual Bloom filter for each rating
- The **Tester** application: to test the Bloom filter created in the Builder, computing the number of false positives

For this aim, we also randomly split the initial dataset in **Train** and **Test** partitions, using another Spark application we realized.

2.1. Mapper

The mapper receives a text line containing the tuple `(movieId, rating, numberOfVotes)`, from which extracts the `movieId` and the `rating`, rounded to the closest integer. Then, for each movie computes the hashes of the ID, that correspond to the positions to set in the bloom filter and emits the key-value pair `<rating, positions_list>`.

```
3   class Mapper:
4       method SETUP()
5           BLOOM_FILTERS_SIZE <- {key_1: size_1, ..., key_m: size_m}
6           HASH_FUNCTION_NUMBER <- # of hash functions to compute
7       method MAP(line_id, line_text)
8           movieId <- first token from line_text
9           rating <- second token from line_text
10          for h = 0 to HASH_FUNCTION_NUMBER do
11              positions[i] <- hash(movieId, i) % BLOOM_FILTER_SIZE
12          EMIT(rating, positions[])
```

2.2. Reducer

The reducer receives several lists of positions to set to true in the bloom filter associated with the specific rating. It creates the bloom filter, sets to true the positions specified and emits the key-value pair `<rating, bloom_filter>`.

```
14  class Reducer:
15      method SETUP()
16          BLOOM_FILTERS_SIZE <- {bf_1: size_1, ..., bf_m: size_m}
17      method REDUCE(rating, list_of_positions)
18          for i = 0 to BLOOM_FILTERS_SIZE[rating] do
19              bloom_filter[i] <- false
20          for all positions in list_of_positions do
21              for all pos in positions do
22                  bloom_filter[pos] <- true
23          EMIT(rating, bloom_filter)
```

2.3. Tester

During the test phase, given a set of films that shouldn't be inside the bloom filter previously created, the Tester application computes the hashes of all the `movieIds` (as in the Builder Mapper) and then check, for each movie, if all the corresponding `positions`, i.e. the hash values, are set to true in the bloom filter. Every movie of the test dataset presents in the bloom filter is a false positive one and it is added to the total count.
The ReducerTester emits the key-value pair `<rating, (number_of_false_positives, number_of_tests, false_positive_percentage)>`.

```
27  class MapperTesterHash:
28      same as Mapper (builder)
```

```
33  class ReducerTester:
34      method REDUCE(rating, list_of_positions)
35          number_of_tests <- 0
36          number_of_false_positives <- 0
37          for all positions in list_of_positions do
38              if bloom_filter[pos] is true for all pos in positions
39                  number_of_false_positive <- number_of_false_positive + 1
40          number_of_tests <- number_of_tests + 1
41          false_positive_percentage <- number_of_false_positives / number_of_tests
42          EMIT(rating, (number_of_false_positives, number_of_tests, false_positive_percentage))
```

3. Preparation of dataset and infrastructure

3.1. Partitioning of the dataset

In order to test the bloom filters, the application **requires a set of records which weren't used during the building process**. So, we decided to split the dataset **imdb.tsv** into two partitions:

- **train_imdb.tsv**, used for the building application.
- **test_imdb.tsv**, used for the testing application.

The partitioning of the dataset has been implemented as a Spark application in the file `split-dataset.py`, in order to exploits both the benefits of MapReduce and the speed of Spark.

- Read the dataset from file
- Remove the first line, which contains the head of the TSV file
- Randomly split the lines into two RDDs
- Save the RDDs into text files

The final partition will be *approximately* (because of the random split) 60% for building and 40% for testing.

This script is executed only once when preparing the data for the project.

3.2. Count the number of movies for each rating

In order to estimate the size of the bloom filters, we require to know **how many movies exist for each rating value** (i.e. how many keys will be inserted in the bloom filter).

A Spark implementation of the process was provided (`count-number-of-keys.py`), in order to benefit from the task and data parallelization.

The application simply reads the "train" partition of the dataset, counts how many records exists for each rating (rounded to the closest integer) and outputs the results to a file in the HDFS.

The generated files (one for each executor) are merged into a single file by the **`getmerge`** HDFS command and reupload to HDFS, in order to be accessible by Java and Python when reading its content.

This script is executed only once when preparing the data for the project.

3.3. Infrastructure configuration

The infrastructure was configured with the settings for 2GB nodes, as presented during the course.

In order to fully exploit the cluster during the Spark execution, the number of executor instances has been set to 4 (equals to the available nodes in the cluster) int the `spark-defaults.conf` file.

The cluster we used consists of 4 nodes on as many machines, available at the ip addresses:

- `172.16.4.141 -> hadoop-namenode`
- `172.16.4.163 -> hadoop-datanode-2`
- `172.16.4.172 -> hadoop-datanode-3`
- `172.16.4.210 -> hadoop-datanode-4`

4. Hadoop implementation

For the Hadoop implementation of the Bloom filter, the classes and structures described in the following sections were realized.

Moreover, we chose to set the number of lines for the mapper (`lines_per_map` parameter) to **15000 lines** for each mapper, because, based on the dataset size, we obtained about 10 mappers for each node of the cluster, which is coherent with the Hadoop guidelines.

As regards the number of reducers, we chose to use **4 reducers** in order to exploit all the cluster nodes and we saw that empirically that is more efficient with respect the usage of 3 reducers, in fact we get the following results:

```
Launched reduce tasks=3
        Total time spent by all reduces in occupied slots (ms)=293336
        Total time spent by all reduce tasks (ms)=146668


Launched reduce tasks=4
        Total time spent by all reduces in occupied slots (ms)=290606
        Total time spent by all reduce tasks (ms)=145303
```

4.1. IntArrayWritable and BooleanArrayWritable

The classes `IntArrayWritable` and `BooleanArrayWritable` were created to properly handled the output values between the Mapper and the Reducer and between Builder and Tester applications.

These two new classes extend the `ArrayWritable` class and can be serialized and deserialized to allow Hadoop to use them inside the cluster.

4.2. BloomFilterConfigurationName and MapReduceParameters

These two structures were realized to store the configuration parameters both for the Bloom filters and the MapReduce implementation. In particular:

- **`BloomFilterConfigurationName`** is an enumerator type that provides the names of the properties for the configuration of the MapReduce application in order to avoid to use directly the parameter name inside the code
- **`MapReduceParameters`** is a Singleton class for reading the parameters from file for the MapReduce jobs configuration and contains information about the numbers of Reducers and lines received by a mapper (both for Builder and Tester application).

4.3. BloomFilterUtil

`BloomFilterUtil` is a class containing helpful methods used in the code, both in Builder and in Tester application, such as:

- `generateConfiguration`: creates the configuration for the MapReduce jobs, in particular, sets how many bloom filters must be created and the size each of them must have and how many hash functions must be used
- `readConfigurationBloomFilterSize`: reads the configuration of the bloom filters to create a map structure containing the key-value pair `<rating, size>` for each bloom filter
- `getBloomFiltersSizeParameters`: reads from file the number of input key for each bloom filter and compute the required size of the data structure, given as argument also the false positive rate to achieve

4.4. BloomFilterMapper

The **`BloomFilterMapper`** class receives as input the id of the line extracted from the text file (the key) and the text line itself (the value). After the parsing of the line, getting the `movieId` and the `rating`, it **computes the hash values** required and maps those values to the corresponding rating.

```
28  public class BloomFilterMapper extends Mapper<LongWritable, Text, ByteWritable, IntArrayWritable> {
29      // Logger
30      private static final Logger LOGGER = LogManager.getLogger(BloomFilterMapper.class);
31
32      // Number of hash functions that must be applied
33      private int HASH_FUNCTIONS_NUMBER;
34      // Size of each bloom filter
35      private Map<Byte, Integer> BLOOM_FILTER_SIZE;
36
37      // Array of IntWritable for the output value of the mapper
38      private final IntArrayWritable outputValue = new IntArrayWritable();
39      // ByteWritable for the output key
40      private final ByteWritable outputKey = new ByteWritable();
41
42      // Instance of the hash function MURMUR_HASH
43      private final Hash hash = Hash.getInstance(Hash.MURMUR_HASH);
```

Before the map function, a `setup function` is executed for loading from the context all the information needed to correctly apply the map method, such as the `BLOOM_FILTER_SIZE` and the `HASH_FUNCTIONS_NUMBER` parameters

```java
@Override
public void setup (Context context) {
    // Retrieve configuration
    Configuration configuration = context.getConfiguration();

    // Read size of the bloom filters
    BLOOM_FILTER_SIZE = BloomFilterUtils.readConfigurationBloomFiltersSize(configuration);

    // Read how many hash functions must be implemented
    HASH_FUNCTIONS_NUMBER = context.getConfiguration().getInt(
            BloomFilterConfigurationName.NUMBER_HASH.toString(),
            defaultValue: -1
    );
    LOGGER.debug("Number of hash functions = " + HASH_FUNCTIONS_NUMBER);
}
```

As already mentioned, the map function emit the key-value pair `<rating, list of hashes>`, where each hash value corresponds to the position to set to true in the bloom filter, and the rating is the vote of a movie, rounded to the closest integer number.

```java
@Override
public void map (LongWritable key, Text value, Context context)
        throws IOException, InterruptedException
{
    // Create string tokenizer to extract movie id and rating
    StringTokenizer itr = new StringTokenizer(value.toString());

    // Retrieve the tokens obtained
    if (!itr.hasMoreTokens()) {
        LOGGER.error("Input line has not enough tokens: " + value);
        return;
    }
    String movieId = itr.nextToken();
    LOGGER.debug("movieId = " + movieId);

    if (!itr.hasMoreTokens()){
        LOGGER.error("Input line has not enough tokens: " + value);
        return;
    }
    byte rating = (byte) Math.round(Double.parseDouble(itr.nextToken()));
    LOGGER.debug("Rating = " + rating);


    Integer bloomFilterSize = BLOOM_FILTER_SIZE.get(rating);
    if (bloomFilterSize == null) {
        LOGGER.error("Rating key " + rating + " doesn't exist in linecount");
        return;
    }

    // Local array of IntWritable to contain the hashes of the movie's id
    IntWritable[] hashes = new IntWritable[HASH_FUNCTIONS_NUMBER];

    // Apply the hash function with different seeds, to obtain HASH_FUNCTIONS_NUMBER values
    for (int i = 0; i < HASH_FUNCTIONS_NUMBER; i++){
        int hashValue = hash.hash(movieId.getBytes(StandardCharsets.UTF_8), i);

        hashes[i] = new IntWritable(
                Math.abs(hashValue % bloomFilterSize)
        );

        LOGGER.debug("Computed hash n." + i + ": " + hashes[i]);
    }

    LOGGER.debug("WRITE (key, value) = ( " + rating + ",  " + Arrays.toString(hashes) + " )");



    // Setting the output values
    outputKey.set(rating);
    outputValue.set(hashes);

    // Emit the key-value pair
    context.write(outputKey, outputValue);
}
```

4.5. BloomFilterReducer

The **`BloomFilterReducer`** class receives the movie rating as key, and the list of positions to set to true in the Bloom filter.

```
24  public class BloomFilterReducer
25          extends Reducer<ByteWritable, IntArrayWritable, ByteWritable, BooleanArrayWritable>
26  {
27      // Logger
28      private static final Logger LOGGER = LogManager.getLogger(BloomFilterReducer.class);
29
30      // Size of each bloom filter
31      private Map<Byte, Integer> BLOOM_FILTER_SIZE;
32
33      // Writable array for the result of the reducer (i.e. the bloom filter)
34      private final BooleanArrayWritable SERIALIZABLE_BLOOM_FILTER = new BooleanArrayWritable();
```

In the `setup` method, `BLOOM_FILTER_SIZE` is loaded from the context, to correctly perform the reduce function.

```
38      @Override
39  @   public void setup (Context context) {
40          BLOOM_FILTER_SIZE = BloomFilterUtils.readConfigurationBloomFiltersSize(context.getConfiguration());
41          LOGGER.debug("Bloom filter size = " + BLOOM_FILTER_SIZE);
42      }
```

In the **`reduce`** method a temporary `bloomFilter` array of `BooleanWritable` is created and initialized with all the values to `false`. Then the arrays of positions received from the mapper are iterated and the corresponding indexes are set to `true`. After these steps, the reduce method returns the key-value pairs **`<rating, bloomFilter>`**, where the `bloomFilter` is the actual bloom filter array associated with the rating key.

```
46      @Override
47  @   public void reduce (ByteWritable key, Iterable<IntArrayWritable> values, Context context)
48              throws IOException, InterruptedException
49      {
50          LOGGER.debug("Reducer key = " + key);
51
52          // Instantiate the temporary bloom filter, i.e. an array of BooleanWritable
53          BooleanWritable[] bloomFilter = new BooleanWritable[BLOOM_FILTER_SIZE.get(key.get())];
54          for (int i = 0; i < bloomFilter.length; i++) {
55              bloomFilter[i] = new BooleanWritable( value: false);
56          }
```

```
60          // Iterate the intermediate data
61          for (IntArrayWritable array : values) {
62              // Generate an iterable array
63              IntWritable[] arrayWithHashedIndexes = (IntWritable[]) array.toArray();
64              LOGGER.debug("IntWritable array = " + Arrays.toString(arrayWithHashedIndexes));
65
66              // Iterate the list of BF indexes produced by mapper's hash functions
67              for (IntWritable i : arrayWithHashedIndexes) {
68                  int indexToSet = i.get();
69                  LOGGER.debug("indexToSet = " + indexToSet);
70
71                  // Check if index is a valid number
72                  if (indexToSet < 0 || indexToSet >= bloomFilter.length) {
73                      LOGGER.warn("Index " + indexToSet +
74                              " for key " + key + " is not valid");
75                      continue;
76                  }
77
78                  // Set to true the corresponding item of the array
79                  if (!bloomFilter[indexToSet].get()) {
80                      bloomFilter[indexToSet].set(true);
81                  }
82
83              }
84
85          }
86          LOGGER.debug("Bloom filter length = " + bloomFilter.length);
87          LOGGER.debug("Bloom filter = " + Arrays.toString(bloomFilter));
88
89          // Emit the reducer's results
90          SERIALIZABLE_BLOOM_FILTER.set(bloomFilter);
91          context.write(key, SERIALIZABLE_BLOOM_FILTER);
```

4.6. Tester

In the **testing phase**, we have to **count how many false positives** there are in the generated bloom filters.

In order to achieve our goal, we have to do the following action during the map phase:

- *Parse* each record of the "test" partition of the dataset and compute the corresponding list of `indexes_to_set` in the bloom filter (this step is the same of the builder application, but it's applied to a different input file)
- *Read* the generated bloom filters from file and distribute them to the reducers.

Since Hadoop lacks of an efficient method to distribute big amount of data between nodes (such as the *broadcasting variables* in Spark), we decided to distribute the bloom filters together with the arrays of `indexes_to_set` during the map phase.

Because of that we decided to implement **two different Mapper** classes (one for the array of `indexes_to_set`, while the other for the bloom filters) and to emit a **`TesterGenericWritable`** as map output value, i.e. a generic wrapper used to uniformly access all the values emitted by the mappers.

Moreover, we used the **`MultipleInputs`** class to support both files as mapper inputs (the text file containing the "test" dataset and the sequence file containing the bloom filters).

As regards the **reducer**, in order to properly compute the number of false positives, we needed to make sure that the bloom filter would be the first value in the **iterable structure** which contains the inputs. Thus, we implemented a **Secondary sorting** mechanism:

- We created a new key for the map output (**`IntermediateKeyWritable`**), which contains the rating of the movie and a boolean which indicates if the corresponding `TesterGenericWritable` is a Bloom filter (`true`) or an array of `indexes_to_set` (`false`)
- We modified the MapReduce **partitioner**, so that it would distribute the map output values only by the rating value
- We modified the MapReduce **key comparator**, so that Bloom filter structure would be the first in the list map output values for each map output key
- We modified the MapReduce **group comparator**, so that the map output values would be still grouped by the rating

4.6.1. Writables

These writable classes are used to wrap the values needed in the Tester application:

- **`TesterGenericWritable`**: extension of `GenericWritable`, is a wrapper for the output of the mappers
- **`TesterResultsWritable`**: implementation of `Writable`, contains the results of the Tester application as a tuple of *<false positive occurrences, total occurrences, false positive percentage>*
- **`IntermediateKeyWritable`**: implementation of a `WritableComparable`, is a wrapper for the intermediate values key between the two mappers and the reducer, to sort correctly the values before reaching the reducer.

```
13  public class TesterGenericWritable extends GenericWritable {
14
15      private static Class[] CLASSES = {
16              BooleanArrayWritable.class, // bloom filter
17              IntArrayWritable.class      // hash of the inputs (indexes to set)
18      };
```

```
16  public class TesterResultsWritable implements Writable {
17
18      private int falsePositiveOccurrences;
19      private int totalOccurrences;
20      private double falsePositivePercentage;
```

```
15  public class IntermediateKeyWritable implements WritableComparable<IntermediateKeyWritable> {
16
17      // Actual key (rating of the movies)
18      private Byte rating;
19
20      // Boolean flag for the corresponding value:
21      // true if the value is a bloom filter,
22      // false if it's the array of indexes to set
23      private Boolean isBloomFilter;
```

4.6.2. Implementation

The implementation for the Tester is like that of the MapReduce algorithm used in the Builder application of the Bloom filter. The major difference lays in the fact that there are two separate mapper classes:

- **`MapperTesterForHashValues`**: mapper used as in the builder mapper, to **compute the hashes** of the `movieId`, values corresponding to the positions to check if are set in the bloom filter, in order to count the exact number of false positive in the reduce phase. The difference from the Builder is the dataset file received in input, that contains movies not present in the other dataset, and the emitted value type, that in this case is yet an `IntArrayWritable`, but

wrapped in a `TesterGenericWritable` structure, for obtaining an **uniform output value type** from the second mapper tester

- **`MapperTesterForBloomFilter`**: mapper used to **distribute the bloom filters** generated in the builder to the right reducer task, in order to test if there are false positives. As in the first mapper tester, the output (the bloom filter) is wrapped in the `TesterGenericWritable` structure

About the **`ReducerTester`**, it receives in input both the `Bloom filter` previously generated and the list of `IntArrayWritable`, each of which represents the array containing the positions to check if set to true in the Bloom filter, for each movie in the test dataset.

To properly receive both mapper outputs, in order to have the Bloom filter as the first item in the input list of the reducer, and the arrays of positions after, we introduced a **composite key** `<rating, isBloomFilter>`, wrapped in the `IntermediateKeyWritable`, described earlier, and we implemented a Secondary Sort on that composite key through the classes **`KeyComparator`**, **`GroupComparator`**, **`PartitionerTester`**:

```
class KeyComparator extends WritableComparator {

    protected KeyComparator() { super(IntermediateKeyWritable.class,  createInstances: true); }

    @Override
    public int compare(WritableComparable w1, WritableComparable w2) {
        IntermediateKeyWritable keyWritable1 = (IntermediateKeyWritable) w1;
        IntermediateKeyWritable keyWritable2 = (IntermediateKeyWritable) w2;

        int cmp = Byte.compare(keyWritable1.getRating(), keyWritable2.getRating());
        if (cmp != 0) {
            return cmp;
        }

        return -Boolean.compare(keyWritable1.getIsBloomFilter(), keyWritable2.getIsBloomFilter());  // reverse order
    }

}
```

```
class GroupComparator extends WritableComparator {
    protected GroupComparator() { super(IntermediateKeyWritable.class,  createInstances: true); }
    @Override
    public int compare(WritableComparable w1, WritableComparable w2) {
        IntermediateKeyWritable keyWritable1 = (IntermediateKeyWritable) w1;
        IntermediateKeyWritable keyWritable2 = (IntermediateKeyWritable) w2;
        return Byte.compare(keyWritable1.getRating(), keyWritable2.getRating());
    }
}
```

```
class PartitionerTester extends Partitioner<IntermediateKeyWritable, TesterGenericWritable> {
    @Override
    public int getPartition(IntermediateKeyWritable key,
                            TesterGenericWritable value,
                            int numReduceTasks)
    {
        return (key.getRating().hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}
```

Then the actual **reduce** phase is very similar to that of the reducer used in the construction of the Bloom filters. The main difference is that the reducer in the Builder has to set the positions indicated to generate the Bloom filter, while the reducer Tester has to **check** if the positions, computed through the hash functions in the mapper for each test movie, are set or not in the Bloom filter. If at least one position is not set, then the movie is not at **false positive**, otherwise it is and has to be counted as such.

The output of this reducer is then composed by the key-value pair **<rating, (numberOfFalsePositives, totalTests, falsePositiveRate)>**, where the value of the pair is wrapped in a `TesterResultsWritable` structure.

```
class ReducerTester
        extends Reducer<IntermediateKeyWritable, TesterGenericWritable, ByteWritable, TesterResultsWritable>
{
    // Logger
    private static final Logger LOGGER = LogManager.getLogger(ReducerTester.class);

    private final ByteWritable outputKey = new ByteWritable();

    // Result of tester execution
    // (number of false positives, total number of test samples, false positive probability)
    private final TesterResultsWritable testResults = new TesterResultsWritable();
```

```
    @Override
    public void reduce (IntermediateKeyWritable key, Iterable<TesterGenericWritable> values, Context context)
            throws IOException, InterruptedException
    {
        LOGGER.debug("Reducer key = " + key);

        // Get the bloom filter from the input values
        TesterGenericWritable wrappedBloomFilter = values.iterator().next();
        if (!(wrappedBloomFilter.get() instanceof BooleanArrayWritable)) {
            LOGGER.error("BloomFilter " + key + " doesn't exist");
            return;
        }

        BooleanWritable[] bloomFilter = (BooleanWritable[])
                        ( (BooleanArrayWritable) wrappedBloomFilter.get() )
                        .toArray();

        LOGGER.debug("bloomFilter = " + Arrays.toString(bloomFilter));
        LOGGER.debug("bloomFilter length = " + bloomFilter.length);
```

```
69          int numberOfTests = 0, numberOfFalsePositives = 0;
70
71          // Get the intermediate results from the mapper
72          for (TesterGenericWritable object : values) {
73
74              // Convert to array of IntWritable (the outputs of the hash functions)
75              IntWritable[] intArray = (IntWritable[]) ( (IntArrayWritable)object.get() ).toArray();
76              LOGGER.debug("intArray = " + Arrays.toString(intArray));
77
78              boolean isFalsePositive = true;
79
80              // Iterate the array of IntWritable in order to check the outputs of the hash functions
81              // (i.e. the position to hit in the bloom filter)
82              for (IntWritable i : intArray) {
83                  int index = i.get();
84                  LOGGER.debug("Index = " + index + " key = " + key +
85                          " BF_size = " + bloomFilter.length);
86
87                  if (index < 0 || index >= bloomFilter.length) {
88                      LOGGER.error("Index " + index + " for key " + key +
89                              " not valid - out of bound");
90                      return;
91                  }
92
93                  // Check current value
94                  if (!bloomFilter[index].get()) {
95                      // If AT LEAST ONE output is NOT set, then the sample is NOT a false positive
96                      isFalsePositive = false;
97                      break;
98                  }
99
100             }
101
102             // Update statistics
103             numberOfTests++;
104             if (isFalsePositive) {
105                 numberOfFalsePositives++;
106             }
107
108         }
109
110         // Set the results of the reducer
111         outputKey.set(key.getRating());
112         testResults.set(numberOfFalsePositives, numberOfTests);
113         context.write(outputKey, testResults);
```

5. Spark implementation

5.1. Bloom filters builder

The **`bloomfilters_builder.`**py file implements the building process of the Bloom filters in Spark. It reads the dataset and the `linecount` files from HDFS, compute the Bloom filter for each rating value and generate a *pickle file* containing them.

5.1.1. Input parameters

The following arguments are taken from command line:

- `false_positive_prob`: contains a float representing probability of false positive
- `dataset_input_file`: contains the path of the dataset used to build the Bloom filters as a string
- `linecount_file`: contains the path of the file containing the amount of the movies for each rating, used to compute the size of every single Bloom filter (see `bloomfilters_util.get_size_of_bloom_filters`)
- `output_file`: contains the path on which put the output file containing the bloom filters

5.1.2. Broadcast variables

We use three broadcast variables in order to efficiently distribute shared data between the worker nodes:

- `broadcast_hash_function_number`: contains the number of hashes to compute for each movie id.
- `broadcast_size_of_bloom_filters`: contains a dictionary that has as keys the ratings and as values the size of the corresponding bloom filter.

5.1.3. MapReduce job

To build the bloom filters using spark, the following steps are done:

a. *Create* the broadcast variable `broadcast_hash_function_number`, containing the number of hashes to compute for each movie.
b. *Create* the broadcast variable `broadcast_size_of_bloom_filters`, containing a dictionary that has as key the ratings and as values the size of the bloom filter of the corresponding rating.
c. *Read and parallelize* the dataset using `textFile(dataset_input_file).`
d. *Parse* each line to extract the movie id and the average rating, round the average rating to the closest integer and compute a list with the hash values of the movie's id.
e. ***Group by rating*** the lists computed previously and merge them into a unique list for each rating value, containing all the indexes of its bloom filter to be set to true (could contain duplicates).

f. For each rating ***create a bloom filter*** by setting to true the indexes contained in the list obtained in the previous step.
g. *Save* the result as a pickle file.

The following image shows the implementation of the code.

```
sc.textFile(dataset_input_file) \
    .map(lambda line:
        util.create_pair_rating_indexes(line,
                                        broadcast_size_of_bloom_filters.value,
                                        broadcast_hash_function_number.value)
        ) \
    .reduceByKey(extend_list) \
    .map(lambda pair_rating_indexes:
            (pair_rating_indexes[0],
             generate_bloom_filter(pair_rating_indexes[1],
                                   broadcast_size_of_bloom_filters.value.get(pair_rating_indexes[0]))
             )
        ) \
    .saveAsPickleFile(output_file)
```

5.1.3.1. *extend_list*

This function requires as input two lists and gives as output the first list extended with the values of the second list.

```
def extend_list(list1: list, list2: list) -> list:
    """
        Concat the second list to the first one and return the first one

        :param list1: first list
        :param list2: second list

        :return: merged list
    """
    list1.extend(list2)
    return list1
```

5.1.3.2. *generate_bloom_filter*

Given as input a list of `indexes_to_set` for a particular rating and the `size` of the Bloom filter, this function generates an array of that given size, at first with all the elements initialized to `False`

and then, setting to `True` each position of the array contained into the `indexes_to_set` list. The returned value is such array that represents the Bloom filter

The implementation can be seen in the next image.

```python
def generate_bloom_filter(indexes_to_set: list, size: int) -> list:
    """
    Set to True the corresponding item of the Bloom Filter

    :param indexes_to_set: list of indexes to set True (computed through hash functions)
    :param size: size of the bloom filter

    :return: the Bloom filters structure
    """
    if size is None:
        return []

    bloom_filter = [False for _ in range(size)]

    for i in indexes_to_set:
        bloom_filter[i] = True

    return bloom_filter
```

5.2. Bloom filters tester

The **Bloom filter tester** application reads the *`generated bloom filters`* and the *test partition* of the dataset from HDFS and computes:

- The **number of false positives**
- The **total number of test records**
- The **false positive percentage**, defined as the ratio between the previous two values

5.2.1. Input parameters

The application requires the following console parameters:

- `false_positive_prob`: a float containing the desired false positive probability.
- `dataset_input_file`: the path to the test partition of the dataset.
- `bloom_filters_input_file`: the path to the file which contains the bloom filters.
- `linecount_file`: the path to the file containing the number of movies for each rating value, in the original training dataset.
- `output_file`: the file in which the output records must be stored.

5.2.2. Broadcast variables

We use three broadcast variables in order to efficiently distribute shared data between the worker nodes:

- `broadcast_hash_function_number`: contains the number of hashes to compute for each movie id.
- `broadcast_size_of_bloom_filters`: contains a dictionary that has as keys the ratings and as values the size of the corresponding bloom filter.
- `broadcast_bloom_filters`: contains all the bloom filters, generated by the ***builder*** application, as a dictionary object `{key: rating, value: bloom_filter}`.

5.2.3. MapReduce job

The tester is implemented as a MapReduce job with one parallelization, four transformations and one action.

1. *Read* the test partition of the dataset and parallelize it
2. A first **`map`** *transformation* generates a k-v pair consisting of the rating value and the list of indexes to set to true in the bloom filter

```
sc.textFile(dataset_input_file) \
    .map(lambda line:
        util.create_pair_rating_indexes(line,
                                        broadcast_size_of_bloom_filters.value,
                                        broadcast_hash_function_number.value)
        ) \
```

3. A second **map** *transformation* emits a k-v pair containing as key the rating value and as value a list containing:

    i. **1** or **0** if the testing record is a false positive or not (checked via the function `check_false_positive`, described later)
    ii. **1**, used to compute the number of occurrences for the current rating

```
.map(lambda rating_indexes: (rating_indexes[0],
                             [check_false_positive(rating_indexes[1],
                                                   broadcast_bloom_filters.value[rating_indexes[0]]),
                              1]
                             )
    ) \
```

4. The k-v pairs obtained in the previous map transformation are **reducedByKey**. The grouped values are the lists containing [1 or 0, 1]. These lists are summed element-wise, in order to parallelly compute the number of false positives and the total number of test for each rating

```
.reduceByKey(sum_elem_lists) \
```

```
def sum_elem_lists(list1: list, list2: list) -> list:

    list1 = [x + y for x, y in zip(list1, list2)]
    return list1
```

5. The last **map** stage emits a k-v pair containing as key the rating and as value a tuple composed by, in order, the number of false positives, the `number of tested movies` (with that rating) and the `false positive rate`

```
.map(lambda rating_counters: (rating_counters[0], (rating_counters[1][0],
                                                   rating_counters[1][1],
                                                   rating_counters[1][0] / rating_counters[1][1])
                              )
    ) \
```

6. The result is saved using *saveAsTextFile* action

5.2.3.1. *check_false_positive*

To count the false positive occurrences, the function `check_false_positive` is used. This function requires as input the indexes of the Bloom filter to be checked and the Bloom filter itself.

The function iterates the list of indexes. If at least one index is equals to false, it means that the movie is not a false positive and return 0. Otherwise, if all the indexes are equals to true, the function returns 1.

```
def check_false_positive(indexes: list, bloom_filter: list) -> int:

    for i in indexes:
        if not bloom_filter[i]:
            return 0

    return 1
```

5.3. bloomfilters_util.py

This module defines some **utility functions** used by the two Spark applications. We will briefly discuss only the interesting ones.

5.3.1. get_size_of_bloom_filters

Generate a MapReduce job that reads the file with the number of keys for each rating in the "training" dataset and **compute the size of bloom filter for each rating**.

```
def get_size_of_bloom_filters(sc: SparkContext, linecount_file: str, false_positive_prob: float) -> dict:
    size_of_bf_list = sc.textFile(linecount_file) \
        .map(lambda line: line.split('\t')[0:2]) \
        .map(lambda split_line: (int(split_line[0]),
                                 compute_size_of_bloom_filter(false_positive_prob, int(split_line[1])))
             ) \
        .collect()

    # Convert the list of tuples into a dictionary
    return dict(size_of_bf_list)
```

5.3.2. compute_indexes_to_set

It generates a list containing the indexes to set in the bloom filter, by computing the hash values of the movieId and by applying the modulo operation on them (so that it will be in the range of the `bloom_filter_size`).

The hash values are obtained by applying the `MurmurHash` algorithm with different seeds (the seed is equal to the position).

```
def compute_indexes_to_set(movie_id: str, rating: int, size_of_bloom_filters: dict, hash_function_number: int) -> list:

    # Get bloom filter size
    bloom_filter_size = size_of_bloom_filters.get(rating)

    if bloom_filter_size is None:
        return []

    # Compute hash values
    return [mmh3.hash(movie_id, i) % bloom_filter_size for i in range(hash_function_number)]
```

5.3.3. create_pair_rating_indexes

Given a line from the dataset, it parses the line to obtain the rating and the movie's id and generate a tuple with the rating and a list with the indexes to set in the bloom filter (computed by calling the function `compute_indexes_to_set`)

```python
def create_pair_rating_indexes(line: str, size_of_bloom_filters: dict, hash_function_number: int) -> Tuple[int, list]:

    # Split line and parse arguments
    split_line = line.split('\t')[0:2]
    movie_id = split_line[0]
    rating = int(float(split_line[1]) + 0.5)  # round() is bugged in Python 3.6!!!

    return (
        rating,
        compute_indexes_to_set(movie_id,
                               rating,
                               size_of_bloom_filters,
                               hash_function_number)
    )
```

6. Experimental results

In order to standardize and automatize the execution of the application, **Linux shell scripts** were provided.

6.1. Hadoop

The following images contains, respectively, the output of the hadoop tester and the result of three different tests using different values of false positive probability.

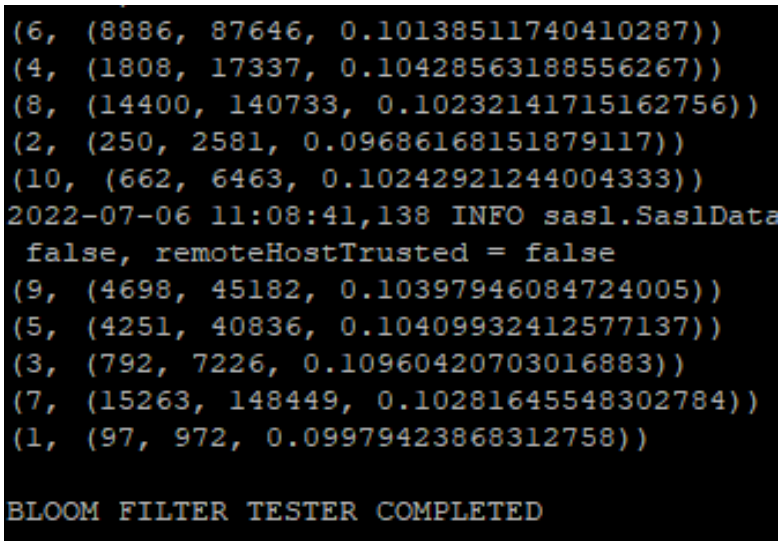
```
4       (1825, 17337, 0.1052661936897964)
8       (14357, 140733, 0.10201587403096644)
2022-07-06 13:43:07,388 INFO sasl.SaslDataTra
check: localHostTrusted = false, remoteHostTr
1       (104, 972, 0.10699588477366255)
5       (4229, 40836, 0.10356058379860907)
9       (4630, 45182, 0.10247443672258864)
2022-07-06 13:43:07,393 INFO sasl.SaslDataTra
check: localHostTrusted = false, remoteHostTr
2       (266, 2581, 0.1030608291359938)
6       (8945, 87646, 0.10205827989868334)
10      (693, 6463, 0.10722574655732632)
3       (796, 7226, 0.11015776363133l3)
7       (15118, 148449, 0.10183968905145875)

BLOOM FILTER TESTER COMPLETED
```

Rating	False Positive Probability		
	0.01	0.05	0.1
1	0.00925926	0.04629629	0.10699588
2	0.00891127	0.05695466	0.10306082
3	0.00899529	0.04926653	0.11015776
4	0.00911345	**0.05012401**	**0.10526619**
5	**0.01021158**	0.05135174	0.10356058
6	**0.01014307**	**0.04979120**	**0.10205827**
7	**0.00987544**	**0.05103436**	**0.10183968**
8	0.00983422	**0.05167942**	0.10201587
9	0.01002612	0.05307423	0.10247443
10	**0.01299706**	**0.04487080**	**0.10722574**
avg	**0,00993668**	**0,05044432**	**0,10446552**

6.2. Spark

The following pictures contain the output of the spark tester and the results of three tests.



Rating	**False Positive Probability**		
	0.01	0.05	0.1
1	0.01028806	0.05452674	0.09979423
2	0.00929872	0.04533126	0.09686168
3	0.01093274	0.05175754	0.10960420
4	0.00911345	**0.05075849**	**0.10428563**
5	**0.00999118**	0.05066607	0.10409932
6	**0.01039408**	**0.04972274**	**0.10138511**
7	**0.01024594**	**0.05079859**	**0.10281645**
8	0.01002607	**0.05057804**	0.10232141
9	0.00960559	0.04964366	0.10397946
10	**0.01005724**	**0.04858424**	**0.10242921**
avg	**0.00999531**	**0.05023674**	**0.10275767**