Master's Degree in Artificial Intelligence and Data Engineering
Cloud Computing Course Project

Bloom Filters in Spark project documentation

Project developed by Antonio Patimo

Academic Year 2021/2022

Table of Contents

Abstract

The aim of the project was to design and implement a MapReduce algorithm to implement a bloom filter construction. The Bloom Filter has been constructed over the ratings of movies listed in the IMDb datasets. The average ratings were rounded to the closest integer value, and was computed a bloom filter for each rating value.

The MapReduce algorithm developed in pseudo-code was implemented using the Apache Spark framework in Python language. In particular the project was developed using the PySpark interface, that allows to develop Spark applications using Python APIs.

After the implementation, the constructed Bloom Filters was tested against the IMDb ratings dataset, computing the exact number of false positives for each rating.

The application was tested both locally and over an Hadoop cluster in pseudo-distributed mode using Yarn.

Bloom Filters 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 can be added to the set, but not removed; the more items added, the larger the probability of false positives. 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_i , every hash function h_1 , ... , h_k computes the corresponding 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 and 1).

The relations between theses values can be expressed as:

$$m = \frac{-nlnp}{(ln2)^2}$$

$$k = \frac{m}{n} ln2$$

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

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.

Dataset Analysis

The dataset used for the bloom filters construction is the IMDb dataset containing rating and votes information for every title. It contains:

- 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

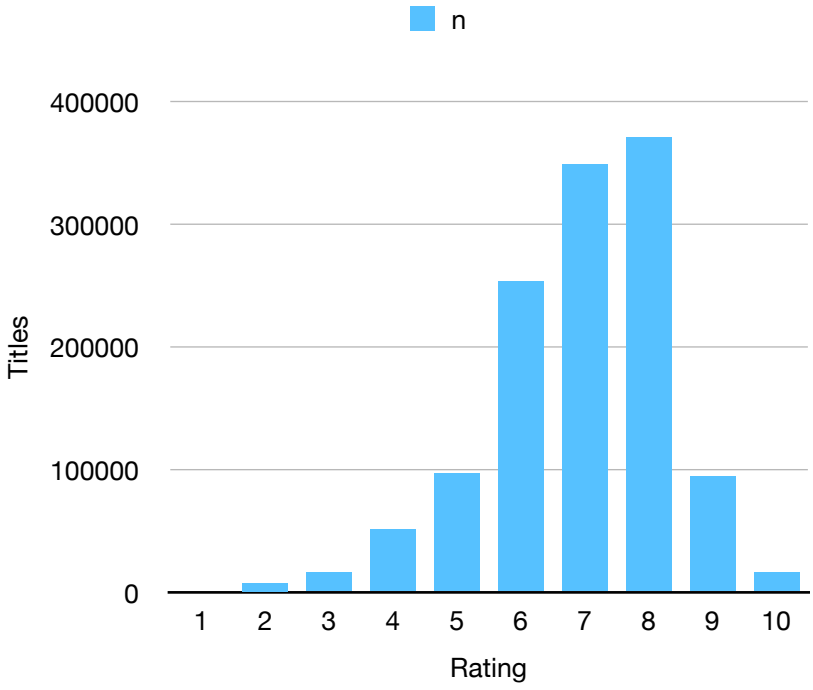
For the construction of the bloom filter only the film id and the average rating are needed for each movie. The average ratings are rounded to the closest integer value, and we've computed a bloom filter for each rating value.

The dataset has a size of 21.8 MB and contains 1260818 rows.

Below is shown the distribution of titles for every rating rounded to the closest integer.

Titles rating distribution

Rating	n
1	2484
2	7699
3	17035
4	50907
5	96854
6	253265
7	349453
8	370225
9	95158
10	17737
Sum	1260817



Algorithms Design

The Bloom filters construction and testing has been divided in tree MapReduce phases:

1. Parameters computation: in this phase the parameters for every bloom filter has been computed.
2. Bloom filters construction: in this phase the bloom filters are constructed based on the previous computed parameters.
3. Testing: in this phase the bloom filters previously constructed are evaluated based on the calculation of the false positive rate.

Parameters Computation

During this phase the n, m and k parameters are computed for every bloom filter to be constructed, using the formulas presented in the bloom filter introduction. This parameters are used during the bloom filters construction phase to match the required false positive rate (*p*). Below is described in pseudo-code the mapper and the reducer code of the MapReduce algorithm.

Mapper:

1. For every rating key emit (rating, 1).

```
1 class Mapper
2    method map(Key rating, Value film)
3       emit(rating, 1)
```

Reducer:

1. Sum the number of 1s from the map phase to compute the parameter *n* for every rating.
2. Compute the parameters *m, k* for every rating.
3. Emit (rating, [n, m, k]).

```
class Reducer
   method reduce(Key rating, Values values)
      n = 0
      for value in values
         n += value
      m = round(-((n * log(p)) / (log(2)) ** 2))
      k = round((m / n) * log(2))
      emit(rating, [n, m, k])
```

Bloom filters construction

During this phase the *n*, *m* and *k* parameters computed in the previous phase are used to construct a bloom filter for every rating. Below is described in pseudo-code the mapper and the reducer code of the MapReduce algorithm.

Mapper:

1. Compute the results of the *k* hash functions related to the film value.
2. Emit the list of indexes for every pair rating , film.

```
class Mapper
   method map(Key rating, Value film)
      indexes = []
      for i in range k
         indexes.append(hash_i(film))
      emit(rating, indexes)
```

Reducer:

1. Create an array of length *m.*
2. Insert 1 in the array index present in the indexes values. Those indexes are the concatenation of every list of indexes computed in the map phase.
3. Emit the rating and the relative bloom filter.

```
class Reducer
   method reduce(Key rating, Values indexes)
   bloom_filter = [0] * rating.m
   for i in indexes
      bloom_filter[i] = 1
   emit(rating, bloom_filter)
```

Testing

During this phase the previously computed bloom filters tested against the IMDb dataset to verify that our bloom filters provide the desired false positive rate given as parameter to our application. The testing is performed calculating for every film the k hash values and emitting for every film the rating for which it is a false positive. Then reducing by the rating we can calculate for every rating the total number of false positives and so compute the false positive rate.

Mapper:

1. For every film compute the values of the k hash functions.
2. Check for every bloom filter if those values gives a false positive.
3. For every rating for which the film is a false positive, emit (Rating, 1).

```
class Mapper
   method map(Key true_rating, Value film)
      for rating in ratings:
         if rating != true_rating:
            counter = 0
            for i in range(k):
               if bloom_filters[rating][hash_i(film)] == 1:
                  counter += 1
             if counter == k:
                emit(rating, 1)
```

Reducer:

1. For every rating count the total number of false positives.
2. Compute the false positive rate.
3. Emit rating, false positive rate.

```
class Reducer
   method reduce(Key rating, Values values)
      false_positives = 0
      for value in values
         false_positives += value
      true_negatives = total_films - rating.n
      false_positive_rate = false_positives / (false_positives + true_negatives)
      return [rating, false_positive_rate]


```

Algorithms Implementation

The project was implemented using the Apache Spark framework and was developed using Python thanks to the PySpark interface for Spark.

Initially the file was read thanks to the spark.read.csv method of the the SparkSQL API and then I transformed it from a dataframe to an RDD thanks to the dataframe method dataframe.rdd.

```
# read the csv file and return a dataframe
df = spark.read.csv(file_path, sep=r'\t', header=True).select("tconst", 'averageRating')
# get the rdd from the dataframe
rdd = df.rdd
```

Parameters computation

As discussed in the algorithm design phase initially all the ratings are associated to a 1 thanks to the map method. Then all the counters are summed thanks to the reduceByKey method and then for every rating, the *m* and *k* parameters are computed thanks to the previously computed parameter *n.* In the end a dictionary is returned having for keys the ratings and for values a list of the relative parameters .The parameters computation implementation is shown in the code below:

```
def get_bloom_filters_parameters(rdd, false_positive_ratio):
    """
    Given a rdd of films and ratings return a dictionary of parameters n, m ,k for every rating.
    Input: rdd of filmId, rating.
    Output: Dictionary of {rating: [n, m, k]} sorted for rating.
    """
    rdd = rdd.map(lambda x: [round(float(x[1])), x[0]])  # map the rdd in the form (rating, film)
    rdd = rdd.map(lambda x: (x[0], 1))
    rdd = rdd.reduceByKey(lambda x, y: x+y)
    rdd = rdd.map(lambda x: get_parameters(x, false_positive_ratio))  # map parameters to every rating
    rdd = rdd.sortByKey()  # sort ratings
    rdd.saveAsTextFile(f"./Data/Output/Parameters")
    bloom_parameters = rdd.collect()
    bloom_parameters = {parameter[0]: parameter[1] for parameter in bloom_parameters}  # transform the list of lists
    # in a dictionary
    return bloom_parameters
```

Bloom filters construction

Thanks to the get_indexes method for every row is emitted a list in the form [rating, indexes] where indexes is a list of the *k* values calculated for that film. To calculate the hash functions values the mmh3.hash method has been used. Then for every rating the list of indexes are concatenated and finally the bloom filters are constructed for every rating. At the end a dictionary is returned having the rating as key and the bloom filter as the value. The bloom filters construction implementation is shown in the code below:

```
def get_bloom_filters(rdd, bloom_parameters):
    """
    Given a rdd in the form (film, rating) return a dictionary with one bloom filter for every rating.
    Input: rdd in the form (film, rating).
    Output: {rating: bloom_filter}
    """
    rdd = rdd.map(lambda x: [round(float(x[1])), x[0]])  # map the rdd in the form (rating, film)
    indexes = rdd.map(
        lambda x: get_indexes(x, bloom_parameters.value))  # map the rdd in the form (rating, list[indexes])
    joined_indexes = indexes.reduceByKey(
        lambda x, y: concatenate_indexes(x, y))  # join the list related to every rating
    # for every rating calculate the relative bloom filter based on the indexes
    bloom_filters = joined_indexes.map(lambda x: create_bloom_filters_from_indexes(x,
                                                                                   bloom_parameters.value)).sortByKey()
    bloom_filters.saveAsTextFile(f"./Data/Output/BloomFilters")
    bloom_filters = bloom_filters.collect()
    bloom_filters = {bloom_filter[0]: bloom_filter[1] for bloom_filter in bloom_filters}  # create the dictionary
    # from the list of lists
    return bloom_filters
```

```
def get_indexes(row, bloom_parameters):
    """
    Return the indexes of the bloom filter of a specific film related to its rating.
    Input: (rating, filmId)
    Output: (rating, {indexes})
    """
    indexes = []
    rating = row[0]
    film = row[1]
    m = bloom_parameters[rating][1]
    k = bloom_parameters[rating][2]
    for i in range(k):
        indexes.append((mmh3.hash(film, i, signed=False) % m))
    return [rating, indexes]
```

```
def create_bloom_filters_from_indexes(row, bloom_parameters):
    """
    Given a set of ratings and indexes return the relative bloom filters
    Input: (rating, list[indexes])
    Output: [rating, bloom_filters]
    """
    rating = row[0]
    indexes = row[1]
    m = bloom_parameters[rating][1]

    bloom_filters = [0] * m  # creation of the array of dimension m
    for index in indexes:
        bloom_filters[index] = 1
    return [rating, bloom_filters]
```

Testing

Initially with the get_false_positives method for every film is returned in a flat-map a series of pairs (rating, 1) if that film is a false positive for the bloom filter related to that rating. Then those values are reduced by rating so we obtain for every rating the total number of false positives with which we can compute the false positive rate, with the get_false_positive_rate method. Again a dictionary is returned having for keys the ratings and for values the relative false positive rate. The testing implementation is shown in the code below:

```
def get_false_positive_rates(rdd, bloom_filters, bloom_parameters):
    """
    Given a rdd in the form (film, rating) and the bloom filters constructed return a dictionary
    of the form {rating: false positive rate}.
    Input: rdd int the form (film, rating), bloom filters, bloom parameters.
    Output: dictionary {rating: false positive rate}

    """
    false_positive_rates = rdd.map(lambda x: [round(float(x[1])), x[0]])  # map the rdd in the form (rating, film)
    false_positive_rates = false_positive_rates.flatMap(lambda x: get_false_positives(x, bloom_parameters.value,
                                                                                      bloom_filters.value))  # return a
    # list of lists where each list is in the form [rating, 1]
    # if the film is a false positive for the bloom filter related to that rating.
    false_positive_rates = false_positive_rates.reduceByKey(
        lambda a, b: a + b)  # reduce by key the number of false positives for every rating
    # get the false positive rate for every rating
    false_positive_rates = false_positive_rates.map(lambda x: get_false_positive_rate(x,
                                                                                      bloom_parameters.value)). \
        sortByKey()

    false_positive_rates.saveAsTextFile(f'./Data/Output/FalsePositiveRates')
    false_positive_rates = false_positive_rates.collect()
    false_positive_rates = {rate[0]: rate[1] for rate in false_positive_rates}
    return false_positive_rates
```

```
def get_false_positives(row, bloom_parameters, bloom_filters):
    """
    Given (rating, film) return a list of lists where each list is in the form [rating, 1]
    if the film is a false positive for the bloom filter related to that rating.
    Input: (rating, film).
    Output: list[(rating, 1)]
    """
    true_rating = row[0]
    film = row[1]
    ratings = bloom_filters.keys()
    false_positive_counter = []
    for rating in ratings:
        if rating != true_rating:
            counter = 0
            m = bloom_parameters[rating][1]
            k = bloom_parameters[rating][2]
            for i in range(k):
                hash_value = (mmh3.hash(film, i, signed=False) % m)
                if bloom_filters[rating][hash_value] == 1:
                    counter += 1
            if counter == k:
                false_positive_counter.append([rating, 1])
    return false_positive_counter
```

```
EhiSuper +1
def get_false_positive_rate(row, bloom_parameters):
    """
    Given a list [rating, falsePositives] return a dictionary of type {rating: falsePositiveRate}
    Input: [rating, falsePositives].
    Output: {rating, falsePositivesRate}
    """
    rating = row[0]
    false_positives = row[1]
    values = bloom_parameters.values()
    films = 0
    for value in values:
        films += value[0]
    true_negatives = films - bloom_parameters[rating][0]
    false_positive_rate = false_positives / (false_positives + true_negatives)
    return [rating, false_positive_rate]
```

Deployment

The application was deployed over an Hadoop pseudo-distributed cluster with Yarn. For this reason the environment with the relative imported packages has to bee distributed with the application in order to be sure that all the dependencies are satisfied. To do so, I created a python virtual environment with the venv-pack and then with the —py-files option I added a zip file containing all the file dependencies to the mai application. The venv-pack can be created with the following commands:

```
python -m venv pyspark_venv
source pyspark_venv/bin/activate
pip install mmh3 venv-pack
venv-pack -o pyspark_venv.tar.gz
```

The applications requires two parameters to run. The file path of the file to be processed and the desired false positive rate. The file must locate in the Hadoop HDFS in the folder

/user/[user]/Data/Input/

The application can be run with the following command:

```
spark-submit --archives pyspark_venv.tar.gz#environment --py-files Dependencies.zip --master yarn BloomFiltersApp.py data.tsv 0.01
```

Testing and Experimental results

The table below shows the results for every bloom filters of every rating based on the desired false positive rate passed as parameter to the application. Since the values are all really close to the desired rate we can consider that the implementation is correct.

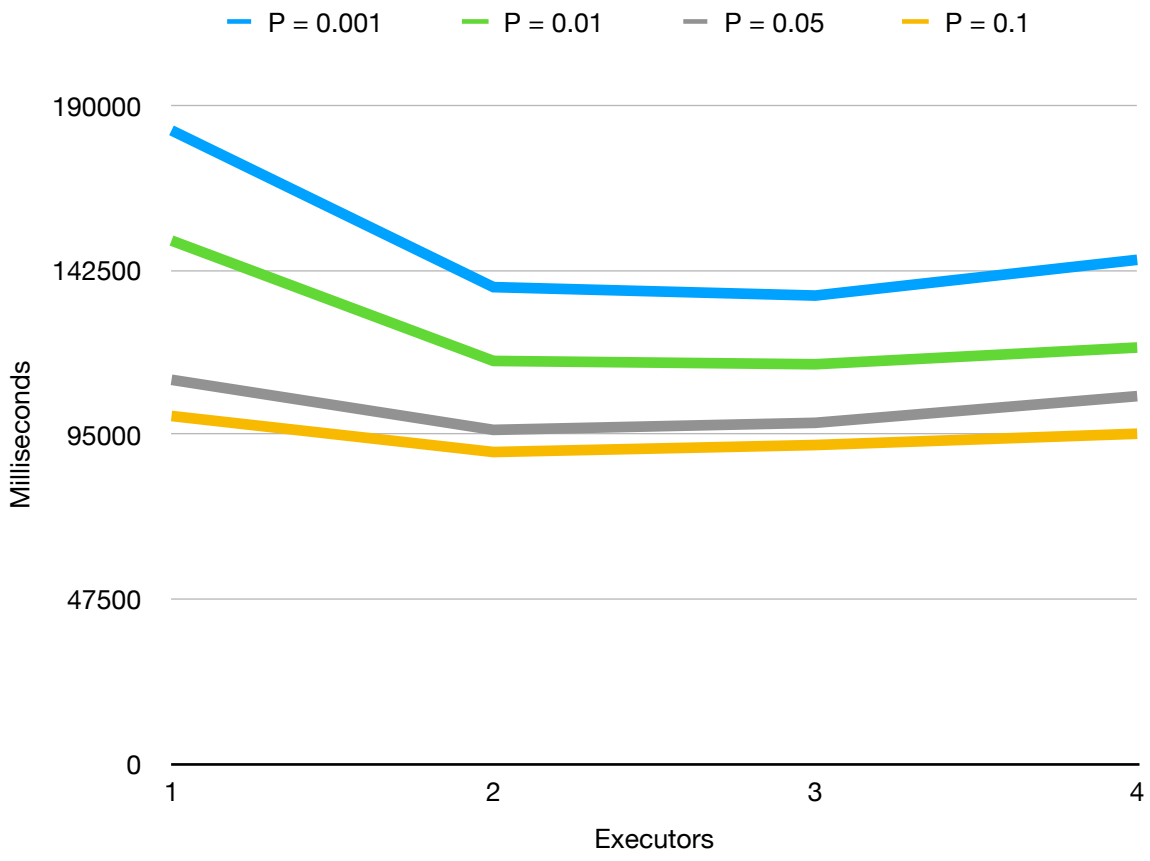
False positive rates for the constructed bloom filters

Rating	P = 0.001	P = 0.01	P = 0.05	P = 0.1
1	0.0010	0.0106	0.0490	0.0939
2	0.0010	0.0102	0.0478	0.0911
3	0.0010	0.0099	0.0486	0.0915
4	0.0010	0.0101	0.0476	0.0914
5	0.0010	0.0100	0.0480	0.0918
6	0.0010	0.0099	0.0479	0.0917
7	0.0010	0.0097	0.0479	0.0910
8	0.0010	0.0101	0.0478	0.0914
9	0.0010	0.0098	0.0476	0.0910
10	0.0010	0.0097	0.0484	0.0920

Regarding the time execution of the application The `--num-executors` option to the Spark YARN client controls how many executors it will allocate on the cluster, so I tried with 1, 2, 3 and 4 executors considering that I had a virtual machine having at most 4 vcores.

Execution times

Executors	P = 0.001	P = 0.01	P = 0.05	P = 0.1
1	182841	151041	110881	100432
2	137614	116322	96431	90039
3	135178	115362	98466	92057
4	145495	120189	106159	95315



As we can see from this chart the execution time diminish significantly when passing from 1 to 2 executors underlying the importance of parallel execution and the correct implementation in this sense. Taking into account 3 executors we can notice that the execution time is almost the same. This happens because we gain some time parallelising the execution but we loose some time in communication. This phenomenon is well noticeable when we use 4 executors where the execution time raises. This happens mostly because the node has at maximum 4 vcores so the application is not really well parallelised and this leads to a lack of performance.