University of Pisa
MSc in Computer Engineering

Cloud Computing

Academic Year 21/22

Design bloom filters over IMDB dataset using map-reduce algorithm

Kamran Mehravar
Gryffindor Group

July, 2022

https://github.com/kamran-mehravar/Cloud-Computing

Table of Contents

Overview

The purpose of this project is to create a bloom filter based on the ratings of movies from the IMDb databases. The average ratings are rounded to the nearest integer number, and for each rating value, I should compute a bloom filter.

Goals

1. Create a Map-Reduce algorithm (using *pseudo code*) to build the bloom filter.

2. use the Hadoop framework to implement the Map-Reduce bloom filter creation technique.

3. Use the Spark framework to implement the Map-Reduce bloom filter creation technique.

4. Put both solutions through their paces on the IMDb ratings dataset, calculating the precise amount of false positives for each rating.

5. Create a project report including the design, implementations, and experimental outcomes.

For each IMDB rating number, we must create a *map-reduce* technique to compute a bloom filter.

Every movie has an average rating of 1-10, and we need first round this rating to the nearest integer before creating the bloom filter for each rating number. The movie IDs are the keys to each bloom filter.

First, we must compute the '*n'* (number of movies in each rate), and then for each bloom filter, we may determine the '*m'* (number of bits in bloom filter) and '*k'* (number of hashing techniques).

Phases

1. Create bloom filters using a map-reduce technique *(pseudo code)*

2. Perform the following tasks for Hadoop and Spark implementations:
    - Calculate the parameters required for the construction of bloom filters.
    - Develop bloom filters
    - For each bloom filter, calculate the false positive rate.

Calculate the following parameters for creating bloom filters:

First, we must compute" *n* "(*the number of videos in each rate*) and determine the false positive rate.

In the following step, we will construct our bloom filters based on these *two* characteristics.

1. Build bloom filters:

We should create our bloom filters and fill the cells with the parameters computed in the previous phase.

2. Determine the false positive rate for each bloom filter:

In this phase we should calculate the real false positive rate of our filter and compare it to the expected value.

Map-reduce pseudocode:

Here, we have two jobs. The ***first*** task is to count the movies in each average rating, and the ***second*** task is to fill the bloom filters with regard to the determined "***n***".

```
Input: IMDB dataset tsv file
Result: 10 bloom filters
class Mapper1
        method Map(conf key, line inputline )
                avg ← get_avg_rate(inputline)
                Rounded ←  round(avg)
                EMIT(IntWritable Rounded, IntWritable one )
class Reducer1
        method Reduce(IntWritable r, IntWritable ones)
        sum ← 0
        for all one ∈ ones do
                sum ← sum + one
                  EMIT(IntWritable r, sum)
```

```
class Mapper2
        attribute list bloomfilters
        method Configure(conf jobConf)
                for i in range(0,10) do
                        m ← jobConf.get('bloomfilters'+i+'.m')
                        m ← jobConf.get('bloomfilters'+i+'.m')
                        bloomfilters.add(bloomfilter(m,k))
        method Map(conf key, line inputline )
                avg ← get_avg_rate(inputline)
                Rounded = round(avg)
                bloomfilters[avg-1].add(inputline.id)
        method Close(conf key, line inputline )
                for i in range(0,10) do
                        EMIT(IntWritable i, BloomFilter bloomfilters[i] )
class Reducer2
        method Reduce(IntWritable r, bloomfilters [])
        result ← new BloomFilter()
        for all bloomfilter  b ∈ [bloomfilters] do
                result.or(b)
        EMIT(IntWritable r, bloomfilter result)
```

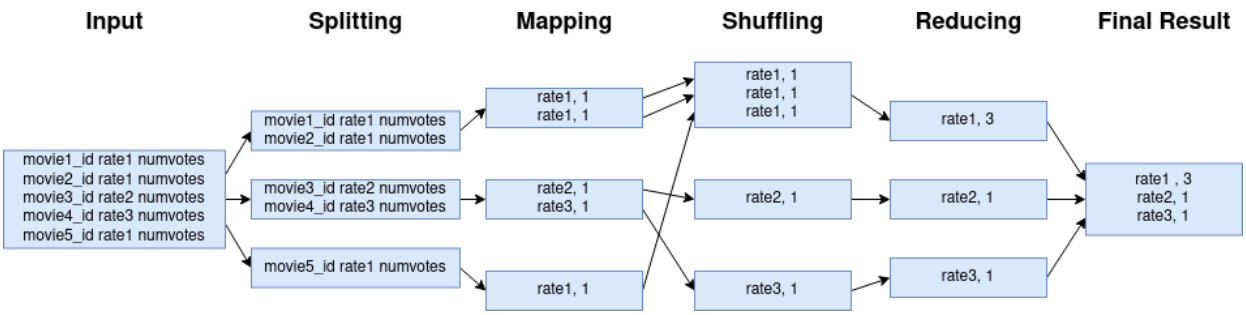


***Figure 1**: The overall map-reduce movie count process*

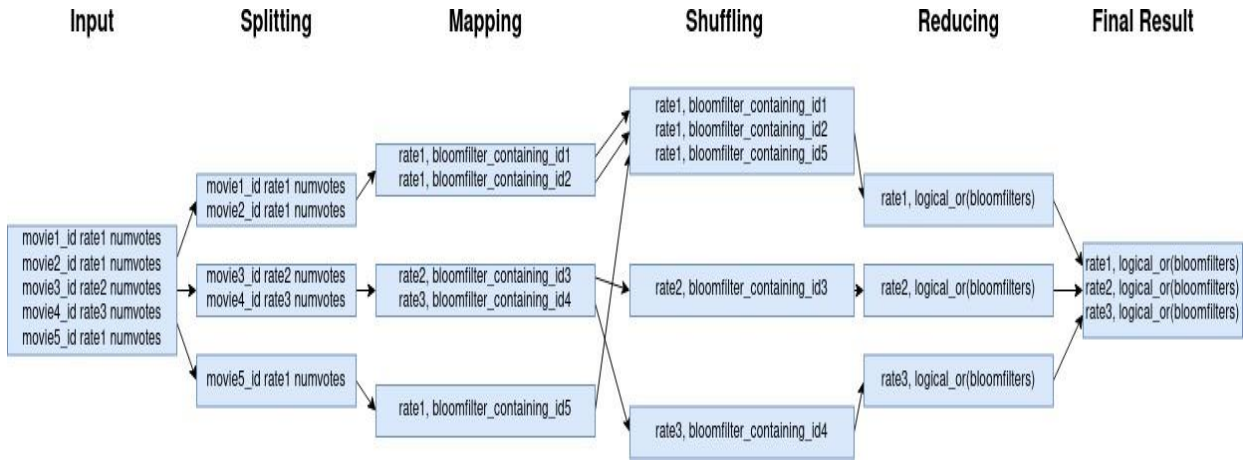


***Figure 2**: The overall map-reduce bloom filter building process*

Hadoop implementation:

First, we must install Hadoop and initialize the Hadoop clusters. Since I didn't have access to several machines, I had used Hadoop Docker container to deploy my Hadoop map-reduce code.

```
git clone https://github.com/big-data-europe/docker-hadoop.git
```

With this command we can clone the *docker-hadoop repository*. After that we just need one command to run a Hadoop container. (all the images needed to run *name-node* and *data-nodes* and resource manager will be downloaded):

```
docker-compose up -d
```

Then after copying necessary files (dataset) into the container we can run our java code easily using this command:

```
hadoop jar hadoop-mapreduce-examples-2.7.1-sources.jar org.apache.hadoop.examples.WordCount input output
```

We have 2 important java class files. *Main.java* and *BloomFilter.java*.

The Primary class is our main class, and the main class file is ***Main.java***. We have three maps- reduce jobs in the main class to completing the following:

1. Count the number of movies in each rate (*calculate n*)
2. Create bloom filters according to results of previous job
3. Test the false positive rate of our filters

In ***BloomFilter.java*** contains the Bloom Filter class, which we use to store our bloom filters and access its characteristics and functions.

The Bit set class is used to maintain bloom filter bit cells, while the Murmur hash class is used for hashing operations.

The API 1 of Hadoop have used for the implementation. It is the old API Version of Hadoop but we used it this is because we found many Hadoop examples developed by the old API Version.

The maven has used as the build automation tool for our ***java code.***

We can compile our code using this command:

```
mvn clean install
```

We must have noticed that every map or reduce class defined in the project is inheriting from Map-Reduce class of:

org.apache.hadoop.mapred.MapReduceBase.

For implementing the map classes, defining the *map()* method is mandatory and implementing Two other functions *(configure, close)* are arbitrary.

In the configure method we can initialize our needing for map work. In addition, in *close()* method we can specify the pairs*(keys , values)* to be emitted.

We have 3 map classes and 2 reducers. It is because of that our third job is using the firstreducer class as its reducer.

I have commented on the code to clarify the number of mapper and reducer classes.

Mapper1: It is like word count example that maps each rounded rating value to 1

Reducer1: sums all the ones for each rating value and emits *(rate, n)*

Mapper2: maps each rate to its bloom filter (training the bloom filter: just one *movie_id* for each mapper is hashed)

Reducer2: combines all of the bloom filters that have the same key by doing logical or between them and saves the bloom filters as *hdfs* strings

Mapper3: reads the trained bloom filters from files; counts all of the false positives for each bloom filter. False positives for any bloom filter are those samples that do not have the same rating as the bloom filter has and the *contains()* function returns true if the sample passed to it.

Reducer3: uses Reducer1 that emits the total false positives for each IMDB rate.

Spark implementation

A Spark program typically follows a simple paradigm:

- A *driver* is the main program.
- One or more workers, called *executors*, run code sent to them by the driver on their partitions of the RDD which is distributed across the cluster.
- Results are then sent back to the driver for aggregation or compilation.

To facilitate the installation and configuration of spark we used the *jupyter* lab Docker image. To install this image we just need to run this command in Linux terminal:

```
sudo docker run -p 8880:8888 jupyter/pyspark-notebook
```

Like Hadoop implementation we have two important files here. The bloomfilter.py and main.py.

In bloomfilter module we have implemented a bloomfilter class that has the needed attributes and methods. A simple implementation of bloomfilter class is shown here:

https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/

We used bit array class from the bit array module to store the bit cells of our bloom filters in it.

We have an add function to add new keys to the bloom filter, a copy function to copy the bloom filter, a bitwise or way to do bitwise or between two bloom filters, a print method to output the bloomfilter bits, and a find method to determine whether or not the key exists in the bloom filter.

The main function is our driver for the spark. To start using Spark, we have to create a *RDD*.

The *Spark-Context* provides a number of methods to do this. We have used the *text-File* method, which reads a file and creates a *RDD* of strings, one for each line in the file.

Then we used the map function to map each rating to 1 and then the reduce function to sum all the ones for each key *(rate)*. It is the same thing that happened in Hadoop implementation.

We computed the m and '*k'* based on the movie counts *(n)* and the specified '*p'* value. These computed parameters should be disseminated so that distributed functions may read them.

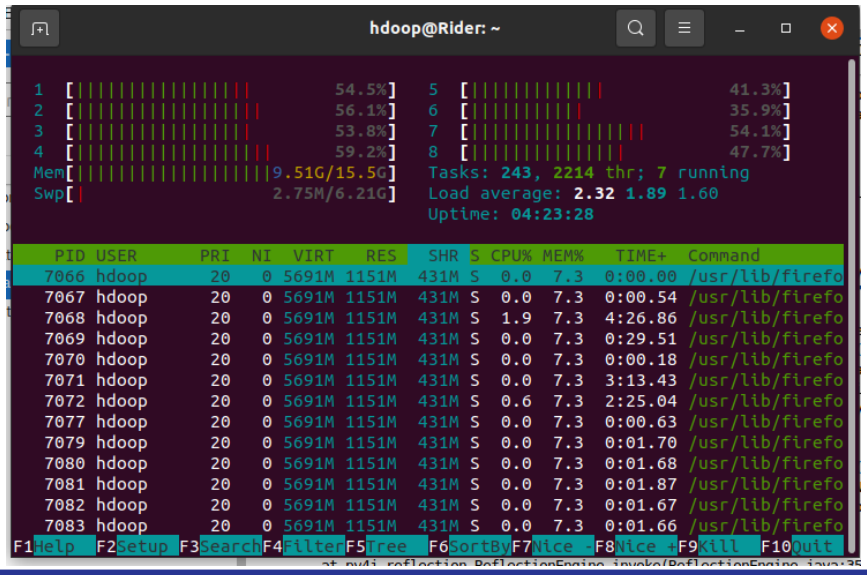
Now we have a map-reduce like the previous map-reduce. We passed the *add_in_bloomfilters()* function to *map-partitions* of our *RDD*; Then by reducing using *bitwise_or()* function we have got the **trained** bloom filters.

The next step is testing the false positive rate. To do so we used the flat-Map of our input *RDD*

We used the ***splitter function*** to split the lines of input correctly. The *flat-Map* method applies the function to all elements of the RDD and flattens the results into a single list of words. We used *bloomfilter_test* function and passed it to *flat-Map* to be applied to all elements in *rdd_input*. After mapping we reduced the data using the *lambda function* that returns summation of false positives for each rate, then we wrote the false positive rate output in an *arbitrary address*.

Results

image of how the spark wasusing the resources (running in Docker containerand interacting using jupyter notebook opened at firefox browser)



False positive rates in Hadoop implementation with constant p = 0.01 :

The *FPR* for each rating is calculated using below formula:

$$calculated\ FPR\ =\ (\frac{false\ positive\ count\ of\ this\ rate}{total\ num\ of\ samples - number\ of\ movies\ in\ this\ rate})$$

Results table (Hadoop implementation)

average rate	n	FP count	Calculated FPR
1	1909	10286	0.0098
2	5526	10437	0.0100
3	15205	10417	0.0100
4	37960	10335	0.0102
5	89595	9510	0.0099
6	189027	8558	0.0099
7	314860	7473	0.01018
8	291900	7542	0.0099
9	89017	9522	0.0099
10	13576	10629	0.0102

Results table (spark implementation)

average rate	n	FP count	Calculated FPR
1	1909	10422	0.0099
2	5526	10523	0.0100
3	15205	10404	0.0100
4	37960	10229	0.0101
5	89595	9674	0.0100
6	189027	8636	0.0100
7	314860	7299	0.0099
8	291900	7581	0.0100
9	89017	9587	0.0099
10	13576	10510	0.010

Runtime of whole code without prints:

The runtime calculation consists of calculating all the main jobs runtimes. Our main jobs in this project are: counting the movies by rate, training bloom filters and also testing the false positive rates for each IMDB *avg* rate.

Hadoop runtime (single machine on Docker)	Spark runtime (single machine on Docker)
9.6433 (s)	**23.6099 (s)**

While it was unbelievable these results are showing that the spark runtime is much greater thanthe Hadoop runtime.

The spark runtime was much bigger than this but with changing *sc = SparkContext('local', 'test')* to *sc = SparkContext('local[*]')* I got this result. running spark using *local[*]* means run spark locally with as many worker threads as logical cores on your machine.