PC-2018/19 Final Project: Implementation of Bloom Filter using Java and Java Hadoop

Luisa Collodi
luisa.collodi@stud.unifi.it

Matteo Ghera
matteo.ghera@stud.unifi.it

Abstract

In this article we present our Java and Java Hadoop Bloom Filter implementation. Regarding the Java implementation we present both a sequential and a parallel version using Java threads.

1. Introduction

When dealing with data streams you must be careful, because if they are not processed immediately or stored, they are lost forever. Often the speed at which data arrive is such that not all of them can be stored. Sometimes, therefore, a representative subset of them is selected or only data that verify a certain property are filtered out. It's easy filtering data that verify a computable math property, it's more difficult if the filtering criterion is based on belonging to a set and that set is too large to be stored. The solution could be: Bloom Filter.

Bloom Filter is a very compact data structure designed in 1970 by Burton Howard Bloom. It provides certain information useful to test whether an element is a member of a set, but only the non-appearance of an element to the set is surely correct. If the filter gives an affirmative answer, this does not necessarily mean that it is correct, there is in fact the possibility to have false positives. So what we want is to select all the elements whose value belongs to a particular set S and to reject the largest number of elements whose value is not in S. For example we want to reject as many spam mails as possible and we have a list of allowed addresses (set S).
More formally:
A Bloom Filter consists of:

- a n bit array all at zero.
- a collection $h_1, h_2, .., h_r$ of hash function such that $h_i : E \longrightarrow \{0, 1, ..., n-1\}$ per $i = 1, 2, ..., r$, where E is the elements set that we are considering.
- a set S which contains the elements (or the elements key attribute) that we want the filter will select.

The **insertion** in the Bloom Filter of an element is done applying on the element in sequence the r hash functions to get r array positions and then setting bits at all these positions to 1. And this is repeated for all the elements of S. So the S set has been mapped in the Bloom Filter.

When we want instead to verify if a new generic element k belongs to the set S, we can proceed with a **lookup** in the filter. More precisely the r hash functions are applied to k, so $h_1(k), h_2(k), .., h_r(k)$ are computed. If at least one of the corresponding positions in the array (filter) has value 0, we are sure that $k \notin S$, so we reject k, instead if all the values are 1, k is allowed to pass because $k \in S$, but in this case there is the possibility of false positives. In particular the probability to have a false positive is:

$$p = (1 - e^{\frac{-rm}{n}})^r$$

where r is the number of hash functions, m the cardinality of the S set and n the length of the

filter. A detailed description of the Bloom filter can be found in [3]

1.1. Language

The first version of the algorithm is written in Java using for the parallelization both low level primitives then higher level abstractions. An overview of Java parallel features can be found in [4, 1]. Instead for the second one we have used Hadhoop, some rifements can be found in [2].

2. Java version

In this section we introduce the main ideas we followed writing the Java code. We have represented the filter as a boolean array: `map` and then we have considered a list of hash functions.

The choice of the hash functions to use in Bloom Filters is very important for performance, they should be uniformly distributed, independent and as fast as possible. Hash encryption functions ensure stability, but they are computationally expensive. In fact with the increasing of the number of functions utilized, the Bloom Filter progressively becomes slower. Using non cryptographic hash functions we lose in guarantee, but we gain significantly in performance. For example murmur is a non cryptographic hash functions family, they are fast and simple, in our implementation we use a murmur-like function: `hash = (R * hash + s.charAt(i)) % n`, where `R` is a random integer and hash at the beginning is 0.

2.1. Sequential implementation

In the program we assume to read from files both the data used to initialize the filter (set S), and those on which it will then be applied. The first step of the algorithm is the filter creation and its initialization. In particular the entire data set is scrolled through and the following function is invoked for each element:

```
public void initializeMap(T element) {
Iterator<HashFunction<T>>hashFunctionIterator
  = myHashFunctionList.iterator();
while (hashFunctionIterator.hasNext()) {
  HashFunction<T> currentHashFunction
    = hashFunctionIterator.next();
  int hash
    = currentHashFunction.compute(element);
   map[hash] = true;
  }
}
```

For each element, all the hash functions we are considering are computed and the corresponding location of the array representing the filter is set to 1.

Then we proceed with the second phase, that is we apply the Bloom filter to the elements of the data flow to verify whether or not they belong to the S set. In particular elements of the second file taken as input are read and processed invoking the `check` method on them:

```
public boolean check(T element) {
 boolean result=true;
 Iterator<HashFunction<T>>hashFunctionIterator
   =myHashFunctionList.iterator();
 while(hashFunctionIterator.hasNext()) {
   HashFunction<T> currentHashFunction
     =hashFunctionIterator.next();
   result=result &&
   map[currentHashFunction.compute(element)];
 }
return result;
}
```

For each element, all the considered r hash functions are computed and so r positions on the filter array are found. If in all the positions found there is a 1 then we increase the counter of the number of admitted elements at S, otherwise that of the rejected elements.

Finally, we compute the number of functions used, the size of the S set, the number of elements to be analyzed, those that are admitted as S set element and those rejected by the algorithm.

2.2. Parallel implementation

Both the initialization and the analysis phases have been parallelized. We also use semaphores to control the accesses to the initialization phase, to the analysis phase and the final report phase. We must ensure that all threads have completed the initialization of the filter, before starting analyzing new elements, likewise it's necessary to ensure that before printing the final report, all the

elements to be evaluated, assigned to each thread, have been evaluated.

1. *Initialization phase*:
   the data set used to initialize the filter is divided into as many parts as the number of threads and each of them is associated with the data chunk that has to process.

   In particular data are divided by the number of threads with the foresight that, if the result of the division is not an integer number, then all the remaining data are assigned to the last thread, so for him the work will be slightly greater. However we thought not to allocate an extra thread, because in the case of a few remaining data the cost of creation and management would be unbalanced.

   In this implementation we used the *Executor Famework*. It abstracts the concept of creating and managing threads, simplifying their use and improving their performance and scalability. In fact threads are created from pools of threads, so we use already existing threads, thus keeping low the overhead. It also improves readability because it separates threads management and creation from the rest of the application.

   An executor takes in input the thread that has to be executed, it is represented by a runnable object. In our implementation we use an *Executor* sub-interface: *ExecutorService*, it has allowed us to manage objects in a more versatile way than *Executor*, in fact it makes available features to help the lifecycle management, both of the individual tasks and of the executor, such as *Callable* and *Future* objects.

   However in this first phase we aren't looking for the thread execution result, we just want the threads to process their data chunk to initialize the Bloom Filter, so we simply give in input to the *ExecutorService* a *Runnable* interface: `Runnable currentInitializeBloomThread` and then we invoke the method *execute* on it:

```
Runnable currentInitializeBloomThread =
  new InitializeBloomThreads(currentList.
  subList(fromIndex, toIndex));
myThreads.execute(currentInitializeBloomThread);
```

   The *execute* method invocation ensures that, for each thread, the `currentInitializeBloomThread.run()` method is executed. It initializes the filter using the data associated with the corresponding thread and it "releases" a permission on `analysisPhaseSem` semaphore, increasing the number of available permissions for the next analysis phase.

2. *Analysis phase*:
   you can access the analysis phase only when all the threads have completed the filter initialization. In fact an `acquire` blocks the code execution until `numThreads` permissions are available and then it decreases the number of available permissions. So all the initialization threads have to terminate their execution releasing a permission each one.

   Subsequently the data to be analyzed are divided between the various threads, similarly to what we did above with the data for the initialization of the filter. In this case, however, we are very interested in the result of the execution of the thread, because we want to know if the element considered belongs or not to the set of interest. For this reason we use a *Callable* instead of a *Runnable* object, which allows to return to the caller a result or to launch an exception, through its *call* method.

   The method *submit* is invoked on an object of type *ExecutorService* with input the object of type *Callable*, to indicate to the thread the task that has to carry out. As it happened before with the *run* method, also the *call* method is executed, but it returns a result that is represented by the following object of type *Future*. The *Future* type objects allow us to receive the returned value from the correspondent task, which is the object *Callable* that they take as input.

```
Callable<List<Integer>>currentAnal..Thread
  = new AnalyzeBloomThreads(
currentList.subList(fromIndex, toIndex));
Future<List<Integer>> result
  = myThreads.submit(currentAnal...Thread);
resultList.add(result);
```

In this case the *submit* method invocation ensures that, for each thread, the *call* method of the class `AnalyzeBloomThreads` is executed.

Each thread analyzes the data assigned to it, returning the number of elements that belong to the set, or rather those that have a high probability of belonging to the set and the number of those that certainly do not belong to it. It also "releases" a permission on `reportPhaseSem` semaphore , increasing the number of available permissions of the semaphore.

Once the call method has been executed, we save all the results: the number of admitted elements and the number of rejected elements from each thread, in a *Future* list and then with a for loop we calculate the total number of admitted elements and the total number of those rejected.

The `reportPhaseSem` semaphore makes sure that the final computations could be executed only when all the threads have completed their chunck's analysis. In the same way as `analysisPhaseSem` semaphore in the previous phase.

Finally, the report containing the results of the computation and the execution time is printed.

2.3. Initialization phase parallelization

In the previous implementation we have also proposed a parallelization of the filter initialization phase, however this operation is not performed frequently in real cases. For this reason the performance measurement will be performed neglecting the initialization times of the filter to make the times more realistic.

2.4. Experiments

2.4.1 Data description

We used two different types of datasets to measure the performance of the parallel implementation of our Bloom filter: the *sample dataset*, which is the dataset used to initialize the Bloom Filter and the *data-flow dataset*, which is the dataset that contains the elements analyzed by Bloom Filter. This dataset is called `people` because we imagined that the strings of the dataset were the people who wanted to take a plane and our job was to push back the terrorists.

We have created five datasets with different lengths using the following bash script:

```
gpw 1 30 > people.txt
for i in $(seq 1 9999);
do
gpw 1 30 >> people.txt
done
```

where the `gpw` command Linux-bash generate pronounceable passwords of whatever length. Then, using the following Java method, we have created the five sample datasets associated to them:

```
public static List<String> generatorSamples
  (List<String> data, int limit) {

  List<String> samples=new ArrayList<String>();
  Iterator<String> dataIterator=data.iterator();
  int chosenIndex=(int) (Math.random()*limit);
  int i=0;
  while(dataIterator.hasNext()) {
    String currentData=dataIterator.next();
    if(i==chosenIndex) {
      samples.add(currentData);
      chosenIndex+=(int) (Math.random()*limit)+1;
    }
  i++;
  }
  return samples;
}
```

So we have:

- the first dataset contains 10000 strings, the corresponding sample dataset contains 39 strings. The size of the Bloom Filter map is 100;

Dataset	Sequential	Parallel
10K	2.67	2.11
100K	2.00	2.56
500K	5.00	5.00
1M	5.67	5.44
2M	7.67	6.56

Table 1: Average execution time for sequential and parallel implementation obtained by varying the number of strings in the data set. The time is in *ms*.

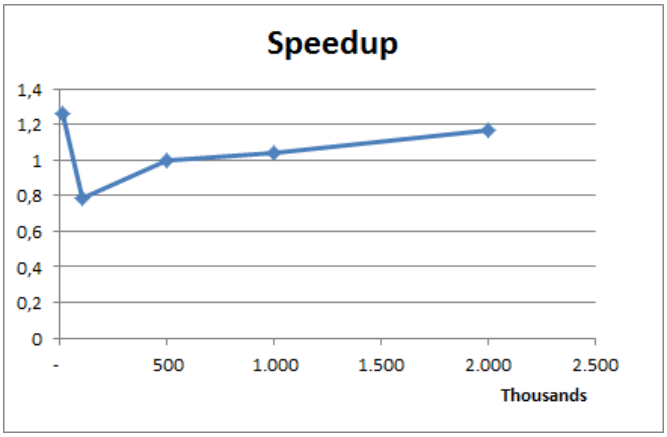

Figure 1: Speedup obtained by varying the number of strings in the data set.

- the second dataset contains 100000 strings, the corresponding sample dataset contains 409 strings. The size of the Bloom Filter map is 1050;

- the third dataset contains 500000 strings, the corresponding sample dataset contains 2012 strings. The size of the Bloom Filter map is 5160;

- the fourth dataset contains one million strings, the corresponding sample dataset contains 39176 strings. The size of the Bloom Filter map is 10200;

- the fifth dataset contains two millions strings, the corresponding sample dataset contains 8018 strings. The size of the Bloom Filter map is 20560.

2.4.2 Speedup

We run our Bloom Filter implementations on a Oracle Virtual Machine Virtualbox using Ubuntu 18.04 64-bit operating system. The number of available cores is two.

We decided to analyze the speedup in the following way: first varying the number of strings in the input dataset and then increasing the number of threads. We performed all the tests ten times and calculate the average of the results.

The first test set was performed using only two threads, that is a number of threads equal to that of the available cores. Table 1 shows the average time execution for sequential and parallel implementation of Bloom Filter in milliseconds on datasets with different number of strings. We note that the gap between the execution time for the parallel implementation and that of the sequential implementation increases as the number of strings in the data set increases. However the increase is very small this is probably due to the fact that the operations performed are very fast; this means that to have greater variations it is necessary to perform tests on very large datasets. Figure 1 shows the speedup trend varying the number of strings in data set. We observe that the speedup tends to 2 (the number of cores available) for the datasets with more strings.

In the second test set we run the sequential and the parallel implementation on the dataset with two millions of strings because it is the most stable. In particular, we run the sequential implementation and the parallel implementation using 2, 3, 5, 10 and 15 threads on the dataset with two milions of strings ten times and then, as before, we calculate the average of the results. Table 2 shows the average time execution for sequential and parallel implementation (using 2, 3, 5, 10 and 15 threads) of Bloom Filter in milliseconds. The execution time increases by increasing the number of threads due to the excessive overhead for the initialization of many threads and beacause the considered dataset is not very big. Figure 2 shows the speedup trend varying the number of threads in the parallel implementation. We ob-

Implementation	Average
Sequential	7.22
2 threads	6.44
3 threads	11.22
5 threads	12.00
10 threads	15.78
15 threads	17.78

Table 2: Average execution time for sequential and parallel implementation obtained by varying the number of threads. The time is in *ms*.

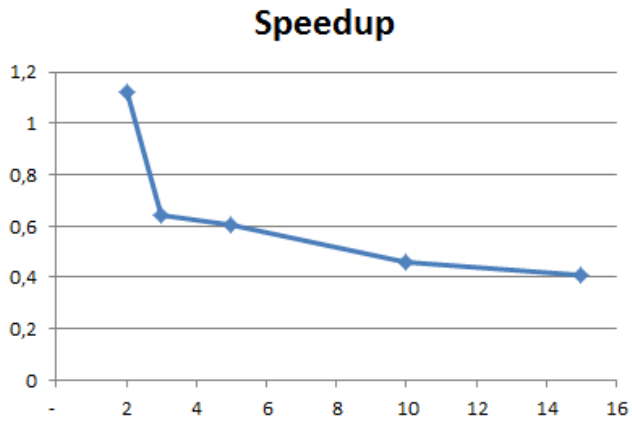

Figure 2: Speedup obtained by varying the number threads.

serve that the speedup is closer to two when we use a number of threads near to the number of cores available, further increasing the number of threads the speedup decreases.

3. Hadoop version

Hadoop is a software platform and framework for distributed computing of data, it can process data faster than a single computer and it can also store more data. It's composed by:

1. HDFS: a filesystem that stores data across multiple machines
2. YARN: a framework to manage Hadoop resources (CPU, RAM, disk space)
3. MapReduce; a framework to process and analyze data in a distributed fashion

Distributed systems are not used because of their excellent characteristics, but because we couldn't do otherwise. In fact, their use determines a significant increase in complexity, as the various computers that make up the cluster (nodes) must communicate with each other over a network. Moreover, with the use of more than one machine, the probability of failure also increases, so there is a kind of trade off between complexity and necessity. However, Hadoop is user-friendly, in fact it hides the entire distributed system, showing a superficial view that looks very like a single system so that the user can focus exclusively on his task, instead of worrying about coordinating the different machines. It's also fault tolerant, because several copies of the data are kept on different machines and the code is very portable, in fact we can use the same code to test both a small data sample and a larger cluster: no modifications to the code are needed as the cluster size changes. More information on Hadoop can be found in [2, 5].

3.1. General structure

Our progam structure is the typical strucure of a MapReduce program. In our program we have identified two Jobs: one related to the initialization phase and one related to the analysis phase. Both jobs have a class that defines them: `InitializeBloomFilterMapReduce` and `AnalyzeBloomFilterMapReduce`. We also create, as required, the `Mapper` and `Reducer` classes as static classes. In particular, we insert them as internal classes of the two jobs because they are rather small, even if they will not be executed by the customer, but by the different Hadoop nodes The `map` method takes in input as parameters a pair key-value, an instance of the `Context` class which is useful to communicate with the Hadoop framework.

We can also note that the `map` method of the `Mapper` class processes a single key-value pair, so we have to write methods to process a single record at a time, we will never have to manage the entire dataset, but it will be the Hadoop framework to transform a huge set of data into a stream of key-value pairs. It's very useful because it allows us to describe code regardless of the size of

the dataset. In the `Mapper` class there are also `setup` and `cleanup` methods that are called before and after the `map` method invocation, in our case we mantain their default implementation where they do nothing.

The main method of each job is the `run` method, it creates the object related to that particular job, initializes it and configures it. It's also needed to specify some parameters such as Mapper class, Reducer class and input/output paths Finally, the job is started, invoking the `runJob` method on it. In general, MapReduce returns the output data using the `OutputFormat` class, each reducer writes the output to its own file, and the output files are usually contained in the same folder. In fact Hadoop has also an input/output mechanisms that provides common file formats implementations, so that we don't have to write file parsers. However in our case we don't use Hadoops MapReduce output mechanism, we directly write the result out to a file. More information on MapReduce programs in Hadoop can be found in [6].

3.2. Implementation

It's often useful to 'summarize' a set of data through a Bloom Filter and when the dataset considered is particularly large Hadoop is a very useful tool. Bloom Filter is also often used for merging datasets, in fact merging datasets and then calculating the Bloom Filter is equivalent to calculating the Bloom Filter on each of the two datasets separately and then merging the two filters, making the OR of their respective vectors.

1. *Initialization phase*:
   The implementation of the first phase of the Bloom Filter with Hadoop is based on the previous observation. The main idea is that each mapper calculates the Bloom Filter on its data chunk and at the end the reducer merges the filters, returning the Bloom Filter of the entire data set as output. In particular, the `map` method calls the `initializeMap` method on the data assigned to it for filter initialization. Note that the `initializeMap` method is the same as the previous implementation in java. In our case the input values of the map function are the lines of the text file that each represents an element of the dataset, so we use a Text object to encapsulate the read text: `Text value`, the same for the key, even if we don't care because the reducer is only one.

```
public void map(Text key, Text value,
  OutputCollector<Text,BloomFilter>output,
  Reporter reporter) throws IOException {

    if (oc == null)
      oc = output;
    bf.initializeMap(key.toString());
}
```

All the Bloom Filters generated by the mappers are sent to a single reducer, which manages data through `reduce` method. More precisely mappers output a key-value pair where the value is a Bloom Filter, in this case the key doesn't matter because we have only a single reducer. The mapper output is contained in the appropriate output collector. Then the `reduce` method makes the union of the Bloom Filters taken as input to return finally the filter relative to the total dataset.

```
public void reduce(Text key,
  Iterator<BloomFilter> values,
  OutputCollector<Text, Text> output,
  Reporter reporter) throws IOException {

    while (values.hasNext()) {
      myBloomFilter.union(
        (BloomFilter)values.next());
      }
    }
```

In particular an output collector is used to collect the mappers outputs, however as previously mentioned, we don't use the Hadoop MapReduce output mechanism, as we want a binary file directly in output, all output operations are concentrated in the `close` method. So we have to set the reducers `OutputFormat`

to `NullOutputFormat`. We have to keep in mind however that the `close` method doesn't take in input an `OutputCollector`, for this it's necessary that the same `MapClass` class that implements `Mappers` maintains a reference to the `OutputCollector` as to make it accessible to the `close` method.

In the `Reducer` class the `close` method must write the output files, so it must know the output path of the file and, in order to obtain it, it must access the dedicated property of the `JobConf` type object and for this reason it keeps it as a class variable in a similar way how it was done with `OutputFormat` in the `Mapper` class. In fact, even in this case the `close` method does not accept a `JobConf` object as input. Finally, `close` method writes the binary Bloom filter to a file in HDFS.

```
public void close() throws IOException {
  Path file = new Path(job.get(
    "mapred.output.dir") + "/bloomfilter_log");
  FSDataOutputStream out
        = file.getFileSystem(job).create(file);
  myBloomFilter.write(out);
  out.close();
}
```

2. *Analysis phase*:
   The analysis phase follows the same programming structure described above but the jobs to be executed changing.

   More precisely in the mapping phase each mapper calculates the number of allowed and rejected elements in its data chunk using the Bloom filter initialized in the previous phase. In particular, each mapper returns a key-value pair $(\text{"}\textit{Number of elements admitted :}\text{"}, 1)$ each time the `check` method applied to the considered element returns true, otherwise it returns a key-value pair $(\text{"}\textit{Number of elements rejected :}\text{"}, 1)$. Also in this case the `check` method remains unchanged from the previous implementation in java

```
public void map(Object key, Text value,
  Context context) throws IOException,
  InterruptedException {

  String word=value.toString();
  if(myBloomFilter.check(word)) {
    context.write(new Text(
      "Number of elements admitted: "), one);
  } else {
    context.write(new Text(
    "Number of elements rejected: "), one);
  }
}
```

In the reduction phase all the values returned in output by the various mappers are added up, thus finally obtaining the number of elements admitted and rejected by the filter.

```
public void reduce(Text key,
  Iterable<IntWritable> values, Context context)
  throws IOException, InterruptedException {

  int total=0;
  for(IntWritable val: values) {
    total+=val.get();
  }
  context.write(key, new IntWritable(total));
}
```

4. Conclusions

Nowadays, it's easy to have to work with really large datasets, so representing them in a compact form is essential to be able to manage them and make queries. In particular, the Bloom filter is a very useful tool to determine the belonging of a certain element to a set of data, in this paper we propose a dual implementation: in Java and Java Hadoop.

In the Java version we have implemented a parallel Bloom Filter using package `java.util.concurrent` tools and applying the master-worker scheme. We have also written a sequential version to compare the performances. Clearly the gap between the execution time for the parallel implementation and that of the sequential implementation increases as the number of strings in the data set increases,

in fact these works make sense only considering large datasets. The speedup obtained is 1.3 with two cores and we can note it increases as the size of the dataset increases, but slightly for the dataset we have considered that are medium-sized datasets. We also observed that the speedup has a peak when the number of threads is close to the number of available cores.

We have also developed a second version of the filter using Hadoop. Sometimes, in-fact, the size of the dataset can be such that it requires the use of a distributed system such as Hadhoop, trying to find the right balance between complexity and necessity. The Bloom filter in Hadoop has been implemented applying the map-reduce framework and basing on the following observation: merging datasets and then calculating the Bloom Filter is equivalent to calculating the Bloom Filter on each of the two datasets separately and then merging the two filters. We did not perform the speedup analysis with the implementation in Hadoop because Hadoop handles the assignment of tasks autonomously and is not sure that it always parallelizes.

References

[1] T. Dauber and G. Runger. *Parallel Programming for Multicore and Cluster Systems*. Springer, 2010.

[2] C. Lam. *Hadoop in Action*. Manning, 2011.

[3] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Stanford InfoLab, 2010.

[4] C. Lyn and L. Snyder. *Principles of Parallel Programming*. Pearson, 2009.

[5] D. Miner. *Hadoop: What You Need to Know*. Oreilly, 2016.

[6] G. Turkington and G. Modena. *Learning Hadoop 2*. Packt, 2015.