

# Com S 435/535 Programming Assignment 1

## 600 Points

Due: Oct 20, 11:59PM

In this programming assignment, you will design (variants of ) Bloom Filter, and use it in an application.

The description of a programming assignment is not a linear narrative and may require multiple readings before things start to click. You are encouraged to consult the instructor/Teaching Assistant for any questions/clarifications regarding the assignment.

Your programs must be in Java, preferably Java 8.1. **You are allowed to work in groups of size 2.** Please do not forget to read the guidelines before you start implementation.

## 1 Bloom Filter

Recall that Bloom Filter is a probabilistic, memory-efficient data structure to store as set  $S$ . You will design the following classes/programs.

- BloomFilterFNV
- BloomFilterRan
- MultiMultiBloomFilter
- NaiveBloomFilter
- FalsePositives
- BloomDifferential
- NaiveDifferential
- EmpericalComparison

If we store a set of size  $N$  using Bloom filter that uses  $M$  bits of memory, then *bitsPerElement* is defined as  $M/N$ .

### 1.1 BloomFilterFNV

Recall that to implement a Bloom filter, one needs to choose the appropriate hash function(s). In lecture, we discussed two types of hash functions—*deterministic hash functions* and *random hash functions*. For this class, you must use the deterministic hash function 64-bit FNV. Your class should have following constructors and methods.

`BloomFilterFNV(int setSize, int bitsPerElement)`. Creates a Bloom filter that can store a set  $S$  of cardinality `setSize`. The size of the filter should approximately be `setSize * bitsPerElement`. The number of hash functions should be the optimal choice which is  $\ln 2 \times \text{filterSize} / \text{setSize}$ .

`add(String s)`. Adds the string  $s$  to the filter. Type of this method is `void`. This method should be case-insensitive. For example, it should not distinguish between “Galaxy” and “galaxy”.

`appears(String s)`. Returns `true` if  $s$  appears in the filter; otherwise returns `false`. This method must also be case-insensitive.

`filterSize()` Returns the size of the filter.

`dataSize()` Returns the number of elements added to the filter.

`numHashes()` Returns the number of hash function used.

`getBit(int j)`. Returns `true` if the  $j$ th bit of the filter is 1. Returns `false` otherwise.

In addition to the above methods, you may include additional public/private methods and constructors.

## 1.2 BloomFilterRan

This class is exactly the same as before, except you use a random hash function. To pick a random hash function with range (close to  $M$ ), (randomly) pick a prime  $p$  that is (slightly) larger than  $M$ . Randomly pick  $a$  and  $b$  between  $\{0, \dots, p-1\}$  and use  $(ax+b)\%p$  as hash function. This class should have the same methods and similar constructors as `BloomFilterFNV`.

### 1.2.1 BloomFilterRanPlus

Very similar to `BloomFilterRan`, but to pick a random hash function, (randomly) pick a prime, and pick  $a, b, c$  uniformly at random from  $\{0, \dots, p-1\}$ . The hash function is  $(ax^2 + bx + c)\%p$ .

### 1.2.2 MultiMultiBloomFilter

The standard Bloom Filter uses one table and multiple hash functions. Here, you will explore a variant that use multiple hash functions along with multiple hash functions.

To store a set  $S$  of size  $N$ , create  $k$  arrays  $A_1, A_2, \dots, A_k$ , each of size  $N$ . Initialize all cells of the arrays to 0. Uniformly and independently  $k$  hash functions  $h_1, h_2, \dots, h_k$ . Store  $S$  as follows: For every  $x \in S$ , place 1 in  $A_1[h_1(x)], A_2[h_2(x)], \dots, A_k[h_k(x)]$ . This class should have the same methods and similar constructor as `BloomFilterFNV`. This class must use *random* hash functions.

## 1.3 FalsePositives

For Bloom Filters (that use a random hash function), we have seen how to theoretically determine a bound on the false positives. Thus, the performance a Bloom Filter can be captured by  $M/N$  (*bitsPerElement*) and the false positive rate. Here you will experimentally/empirically determine the false positives for the above Bloom Filters (and there) variants. For each of the Bloom Filters above, design a program to compare their performances by determining False Positives.

## 1.4 Statistics

Suppose we store  $n$  elements in a filter of size  $m$  using  $k$  hash functions. Recall that the probability that a  $i$ th bit of the filter remains zero is  $(1 - 1/m)^{kn}$ . Thus we expect  $m(1 - 1/m)^{kn}$  entries of the filter to be zeros. It can be shown that the number of bits that are zero is “close” to the expectation with high probability (the proof is beyond the scope of this course).

Suppose you are given a Bloom Filter of size  $m$  and you know that the filter was constructed using  $k$ -hash functions by storing elements of an unknown set. In particular, you do not know the size of the set  $S$ . How can you estimate the size of the set by accessing the Bloom Filter?

Suppose  $B_1$  and  $B_2$  are bloom filters of size  $m$  that are constructed by storing sets  $S_1$  and  $S_2$  (by using the same set of  $k$  hash functions). Let  $B$  be a filter obtained by taking bit-wise and of  $B_1$  and  $B_2$ , and  $Z$  be the number of zeros in  $B$ . Let  $Z_1$  and  $Z_2$  be number so of zeros in  $B_1$  and  $B_2$ . It can be shown that

$$\frac{Z_1 + Z_2 - Z}{Z_1 Z_2}$$

approximately equals

$$\frac{1}{m} \left( 1 - \frac{1}{m} \right)^{-kt}$$

where  $t$  is number of elements that appear in both  $S_1$  and  $S_2$ .

Create a class named `Statistics`. This will have following `static` methods.

`estimateSetSize(BloomFilterFNV f)`. This method returns an estimate for the number of elements stored in the filter, without calling the method `dataSize`.

`estimateIntersectSize(BloomFilterFNV f1, BloomFilterFNV f2)`. Returns an estimate for the size of  $S_1 \cap S_2$  when  $S_1$  is stored in `f1` and  $S_2$  is stored in `f2`.

## 1.5 Application 1—Differential Files

Bloom filters can be used in some low-level applications that involve file management. An example is that of *differential files*. Consider a large database of records, where each record is of the form  $\langle key, value \rangle$ . *key* and *value* could be strings. Suppose that such data is stored in a single file. Let us name this file as `database.txt`. This file has one record per line. Often we would like to make changes to records. However, it is very inefficient to make changes to the file `database.txt` every time a record is changed (and is prone to errors). A better approach is to create a differential file. Whenever a particular record is changed, then that record is written to a differential file (lets call this file `differential.txt`), and `database.txt` is not changed. Once the differential file becomes large enough (and when the load on the server is low), then `database.txt` is updated and all contents of `differential.txt` are removed. Typically, the number of records in the differential file is less than 10% of the number of records in `database.txt`. Since both the files are large, they are often stored in the external memory.

Now consider the query mechanism with such a system. The user has a `key` and wants to retrieve the `value` associated with that key. For this, the system should first check whether `key` appears in the differential file. If it does not appear in the differential file, then the system will access `database.txt`. Since the number of records in `differential.txt` is much smaller (10%) compared to the number of records in `database.txt`, for most of the queries (90% of the time), the system accesses both the files `differential.txt` and `database.txt` that are stored in the secondary memory. This is inefficient. A solution to speed up this process is to store `differential.txt` (or

just the keys of `differential.txt`) in main memory. However, this consumes more memory as the number of records in `differential.txt` can be quite large. A solution is via using Bloom filters. Create a Bloom filter of all keys in `differential.txt`. Store the filter in the main memory. When a key arrives as a query, first check if that key appears in the Bloom Filter. If it appears in the filter, then consult `differential.txt`, otherwise, access `database.txt` directly. Now, for most keys that do not appear in `differential.txt`, the system does not consult `differential.txt` (unless there is a false positive). This is more efficient.

Your job is to write a program that simulates this. You are given the following files.

- `database.txt`: This is the data base file. Each line of this file is of the following from:  
word1 word2 word3 word4 year1 n1 m1 year2 n2 m2 .....

For example a record would look like

Archbishop had given him 1720 8 6 1727 10 4 1758 20 6 .....

This means the 4-word phrase Archbishop had given him appeared 8 times in 6 books in year 1720, and appeared 10 times in 4 books in year 1727 and so on. Here key is the 4-word phrase.

This file was created by processing data from Google Books Ngram Viewer (<http://storage.googleapis.com/books/ngrams/books/datasetsv2.html>). This file has nearly 12 Million records.

- `DiffFile.txt`. This is the differential file. This has nearly 1.2 Million records. This is created by picking random records from `database.txt` and making changes to those records.
- You are also given a file named `grams.txt`. This contains all keys that appear in `database.txt`.

Your task is the following. Write two programs named `BloomDifferential` and `NaiveDifferential` that does the following. Program `BloomDifferential` has

- a method named `createFilter()` that returns a Bloom Filter corresponding to the records in the differential file `DiffFile.txt`.
- a method named `retrieveRecord(String key)` that gets a key as parameter and returns the record corresponding to the record by consulting the Bloom Filter first.

Program `NaiveDifferential` also has a method named `retrieveRecord(String key)` that returns the record corresponding to the key. It does not use Bloom Filter. It may use a hash table to store the contents of `DiffFile.txt`.

How do you compare the performances of the programs `NaiveDifferential` and `BloomDifferential`? Finally, design a program named `EmpericalComparison` that empirically compares the performances of both the programs.

## 2 Application 2: Distributed Join

This section of the PA will not be graded. You may want to work on this to gain more experience with Bloom Filter applications. Please do not submit the code for this part.

Recall join operation on two relations. For our purposes we only work with 2-ary relations, such a relation has a key and an attribute. Join is performed based on the attribute that is common to both relations. Let us call this common attribute as “join attribute”. Consider following two relations  $R1$  and  $R2$ .

Name	Sport
Harry	Baseball
Sally	Cricket
Harriet	Baseball
Harry	Basketball
Jim	Football

Sport	Team
Baseball	Royals
Football	Vikings
Baseball	Mets
Basketball	Bulls
Hockey	Sabres

Here **Sport** is the join attribute between the relations. Natural join of these two relations (based on join attribute **Sport**) is the following table.

Harry	Baseball	Royals
Harry	Baseball	Mets
Harriet	Baseball	Royals
Harriet	Baseball	Mets
Harry	Basketball	Bulls
Jim	Football	Vikings

Suppose that  $R1$  is resides in one server and  $R2$  is on a different server. How can they compute join? A natural solution is to send one of the tables to the other server. However, this incurs a huge communication cost. A solution to reduce communication is via Bloom Filters. The first server creates a bloom filter consisting all values corresponding to the join attribute’. Upon receiving this the second server prunes its table as follows: For each tuple  $\langle a_1, a_2 \rangle$  in its relation check if  $a_1$  ( $a_1$  is a value corresponding to the join attribute) is in the filter or not. If it is in the filter, then place  $\langle a_1, a_2 \rangle$  in to a new table/relation  $R3$ . Now second server sends  $R3$  to the first server and the first server performs the joint of  $R1$  with  $R3$ . The size  $R3$  could be much smaller than the size of  $R2$  and this decreases communication cost.

Your goal is to use Bloom filters to simulate this. You are given two relations  $R1$  and  $R2$  stored in two files. Assume that they are stored in a tabular format. Each row has exactly two columns. Also, assume that the join attribute is the first column.

Write a program named `BloomJoin` that computes the join of the two relations using Bloom Filter. This class will have

`BloomJoin(String f1, String f2)`. A constructor that takes names of two files that hold relations  $R1$  and  $R2$ .

`join(String f3)` Writes the join of  $R1$  and  $R2$  to a file named `f3`.

### 3 Report

Write a brief report that includes the following

- Your Name(s).
- For the class, `BloomFilterFNV` explain the process via which you are generating  $k$ -hash values, and the rationale behind your process.
- The random hash function that you used for the class `BloomFilterRan`, explain how you generated  $k$  hash values.
- The experiment designed to compute false positives and your rationale behind the design of the experiment.
- For all the Bloom filter classes, report the false probabilities when `bitsPerElement` are 4, 8 and 10. How do false positives depend on `bitsPerElement`? Which filter has smaller false positives? If there is a considerable difference between the false positives, can you explain the difference? How far away are the false positives from the theoretical predictions?
- Evaluate the (approximate) efficiency of the Bloom Filter for differential files application as follows: Recall that in this application that Bloom Filter is stored in the main memory, and all other files are stored in secondary memory. Suppose that accessing the contents of a file in the secondary memory takes 1 second, whereas accessing the main memory takes one milli second. Compare the time taken by programs that use Bloom Filter and that do not use Bloom Filter. Use your experiment that compared the performances of `NaiveDifferential` and `BloomDifferential` to arrive at the times.

### 4 Guidelines

An obvious choice is to store the bits of the filter in a Boolean array. However, in Java, Boolean array uses a lot more memory. Each cell of the Boolean array may use up to 4 bytes! This defeats the purpose of creating Bloom Filters. Thus your code must use the class `BitSet`. See Java API for more on this class.

Your code should not use any of the Java's inbuilt functionalities to create hash tables in the Bloom filter classes. For example, your code should not use classes such as `Hashtable` or `HashMap`. You may `Hashtable` or `Hashmap` for the differential file application. If you wish you may use the method `hashCode()` from the class `String`. This method converts a `String` to an `int`.

**You are allowed to work in groups of size 2.** You are allowed to discuss with your classmates for this assignment. Definition of *classmates*: Students who are taking Com S 435/535 in Fall 23. However, You should write your programs without consulting other groups. In your report you should acknowledge the students with whom you discussed and any online resources that you consulted. This will not affect your grade. Failure to acknowledge is considered *academic dishonesty*, and it will affect your grade.

## 5 Grading

Below is the point distribution

- Report: 50 Points.
- BloomFilterFNV: 150 Points
- BloomFilterRand: 150 Points
- BloomFilterRandPlus: 50 Points
- MultiMultiBloomFilter: 100 Points
- BloomDifferential and NaiveDifferential: 100 Points

While grading programs, the following rubric is applied:

- Code looks correct: 25%
- Test Cases: 75%

Test cases will check for accuracy (false positives and false negatives), time and memory usage.

## 6 What to Submit

Please submit all .java files and the report via Canvas. Your report should be in PDF format. Please do not submit .class files. Please zip all .java files and the report, and name the file PA1YourUserID.zip. If you are working in a group, concatenate the user IDs. **Only one submission per group, please. Please do not submit .class files**

Have Fun!