CS-261P – Data Structures

Lab Assignment Project 2

Empirical Analysis of False Positive Rate for Two variants of Bloom Filters

Rishabh Saxena (rsaxena3) | Chaitanya Prafull Ujeniya (cujeniya)

1. Introduction

This project explores a new way of implementing a Bloom filter and empirically comparing the performance to a regular bloom filter. The new bloom filter (called k-array bloom filter) is implemented by having different arrays for each of the k hash functions. ($h_0(x)$, $h_1(x)$, … $h_{k-1}(x)$)

The project has been implemented in C++ and the analysis on False Positive Rates has been done using python. Key highlights regarding the project's implementation are as follows:

- **OOP design & Implementation:** The entire project is implemented using Object-Oriented features. Separate abstract classes have been defined for Bloom Filter and Hash Functions, which are implemented by one or more sub-classes (inheritance). The classes also exhibit data hiding and abstraction which doesn't allow the testing and Data collection framework to access and manipulate key features of the class objects.
- **Unit Test Framework:** UnitTest.cpp file has been implemented that verifies the sanity of all bloom filter methods that have been implemented. This ensures the validity of our implementation for both types before we start collecting any data for false positives.
- **Data Collection Framework:** MainTester.cpp file has been implemented to collect data on False Positive Rates (in %) for both types of Bloom Filter on a huge range of values for n (number of elements inserted into Bloom Filter) and k (number of hash functions in bloom filter).

2. Problem Description

In a regular Bloom Filter of size *N* & *k* hash functions, after n insertions, a total of *n.k* set bit operations are done. The formula for a false positive in such a bloom filter is:

$$(1-(1-\frac{1}{N})^{nk})^{k}$$

Therefore, each of the *k* bits in a false positive map to one of the bits set by the *n.k* set bit operations.

We define a new bloom filter (called k-array bloom filter) such that, for *n* insert operations, each of *k* hash functions sets n bits in its own individual array (Refer to Fig 2). Therefore, after n insertions, there are a total of *n* set bit operations in *k* individual arrays. **For such a bloom filter, each of the *k* bits in a false positive map to one of the bits set by *n* set bit operations.**

Finally, to ensure that the comparisons are fair, we will ensure that the 2 bloom filters being compared have the same total memory consumption i.e. we will consider the size of a regular bloom filter as *N* and the size of the k-array bloom filter as *k* arrays of *N/k* length. (Refer to Fig 1 & 2 below)

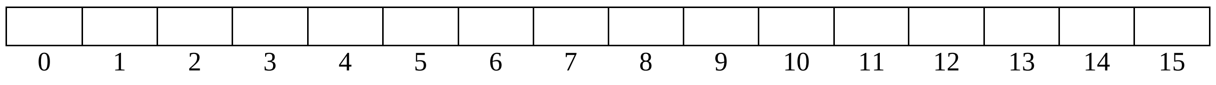


Fig 1. Regular Bloom Filter with N = 16, and k=4

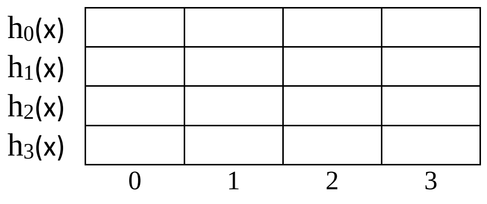
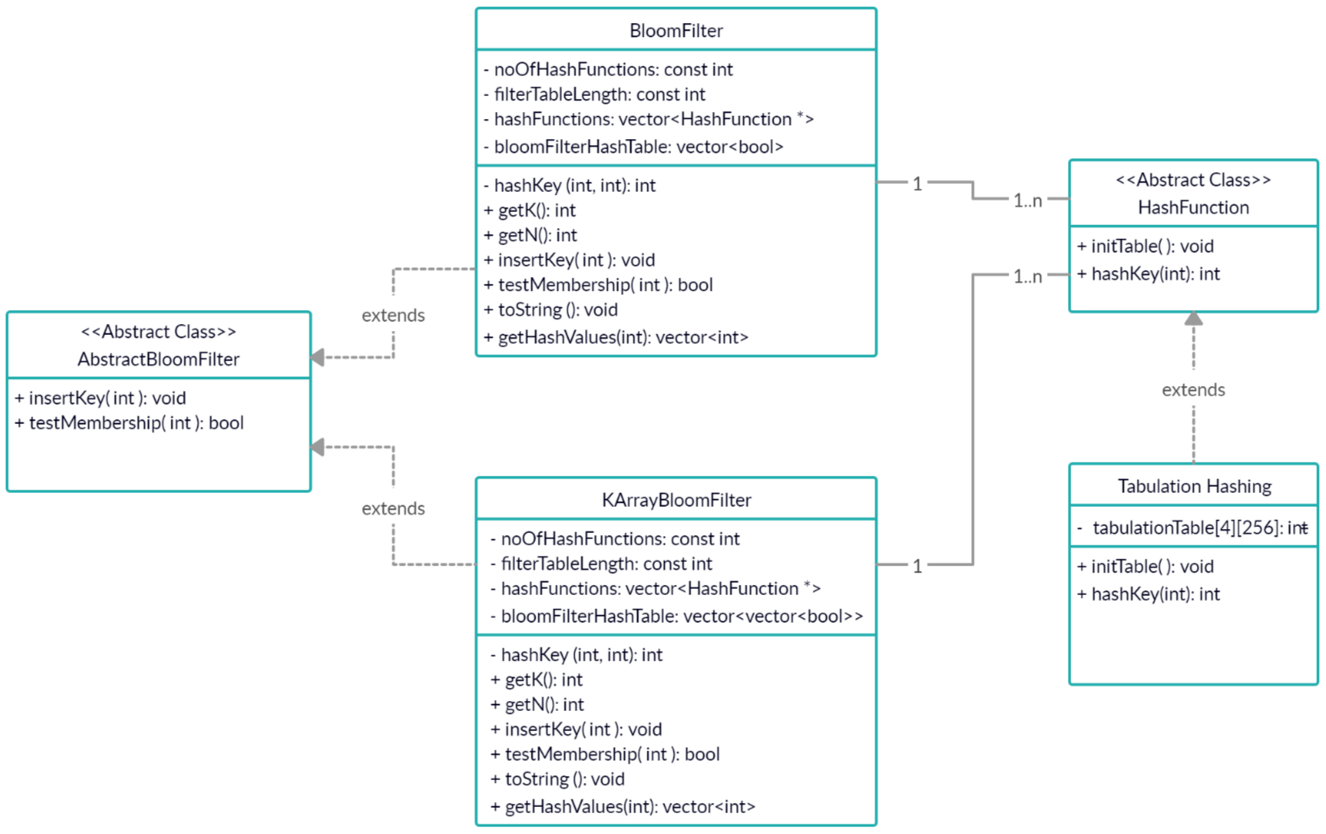


Fig 2. *k-array* Bloom Filter with k=4 and k arrays of length 4

3. Design

UML Class Diagram showing class relationships


BloomFilter
- noOfHashFunctions: const int
- filterTableLength: const int
- hashFunctions: vector<HashFunction *>
- bloomFilterHashTable: vector<bool>
- hashKey (int, int): int
+ getK(): int
+ getN(): int
+ insertKey(int): void
+ testMembership(int): bool
+ toString (): void
+ getHashValues(int): vector<int>
1
1..n
<<Abstract Class>>
HashFunction
+ initTable( ): void
+ hashKey(int): int
1..n
extends
<<Abstract Class>>
AbstractBloomFilter
+ insertKey(int): void
+ testMembership(int): bool
extends
extends
KArrayBloomFilter
- noOfHashFunctions: const int
- filterTableLength: const int
- hashFunctions: vector<HashFunction *>
- bloomFilterHashTable: vector<vector<bool>>
- hashKey (int, int): int
+ getK(): int
+ getN(): int
+ insertKey(int): void
+ testMembership(int): bool
+ toString (): void
+ getHashValues(int): vector<int>
1
Tabulation Hashing
- tabulationTable[4][256]: int
+ initTable( ): void
+ hashKey(int): int

Following are some of the design decisions made during the design phase of the project:

- Code implementation will be in C++ and analysis on data collected will be in Python.
- HashFunction is an abstract class which functions as an interface. Tabulation Hashing is a subclass of HashFunction which implements Tabulation Hashing algorithm.
- AbstractBloomFilter is an abstract class for a Bloom Filter. BloomFilter and KArrayBloomFilter are 2 different sub-classes that implement the General and KArrayBloomFilter Bloom filters respectively.

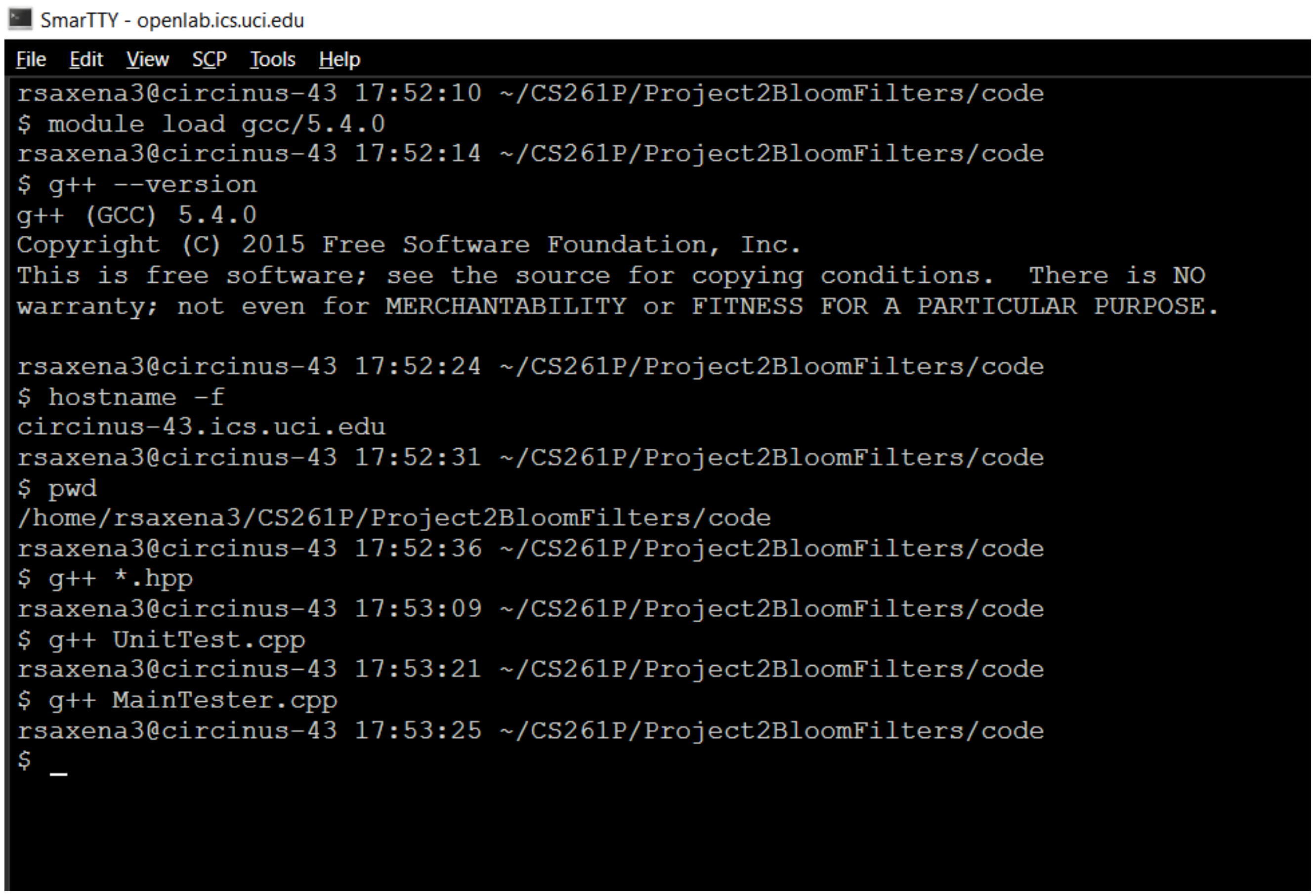
4. Implementation

After implementing the classes depicted in the UML diagram, 2 Test modules were developed:

- UnitTest.cpp: Implements sanity tests
- MainTester.cpp: Implements False Positive Rate data collection

we will discuss their details in this section:

Screenshot for compilation of the project on OpenLab



```
rsaxena3@circinus-43 17:52:10 ~/CS261P/Project2BloomFilters/code
$ module load gcc/5.4.0
rsaxena3@circinus-43 17:52:14 ~/CS261P/Project2BloomFilters/code
$ g++ --version
g++ (GCC) 5.4.0
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

rsaxena3@circinus-43 17:52:24 ~/CS261P/Project2BloomFilters/code
$ hostname -f
circinus-43.ics.uci.edu
rsaxena3@circinus-43 17:52:31 ~/CS261P/Project2BloomFilters/code
$ pwd
/home/rsaxena3/CS261P/Project2BloomFilters/code
rsaxena3@circinus-43 17:52:36 ~/CS261P/Project2BloomFilters/code
$ g++ *.hpp
rsaxena3@circinus-43 17:53:09 ~/CS261P/Project2BloomFilters/code
$ g++ UnitTest.cpp
rsaxena3@circinus-43 17:53:21 ~/CS261P/Project2BloomFilters/code
$ g++ MainTester.cpp
rsaxena3@circinus-43 17:53:25 ~/CS261P/Project2BloomFilters/code
$ _
```

4.1 Unit Test Implementation

We inserted 5 unique keys in each bloom filter with k=3 hash functions to check if the correct bit was being set (output image attached below). The total number of buckets in both bloom filters are 30.

For Regular Bloom Filter

E.g.: key=10

Table Entries set:14,6,16 (3 hash values were provided by the 3 hash functions)

The index 14,6,16 are set as 1 in the bloom filter.

For k-array Bloom Filter

E.g.: Key=60

Table Entries Set:2,4,5

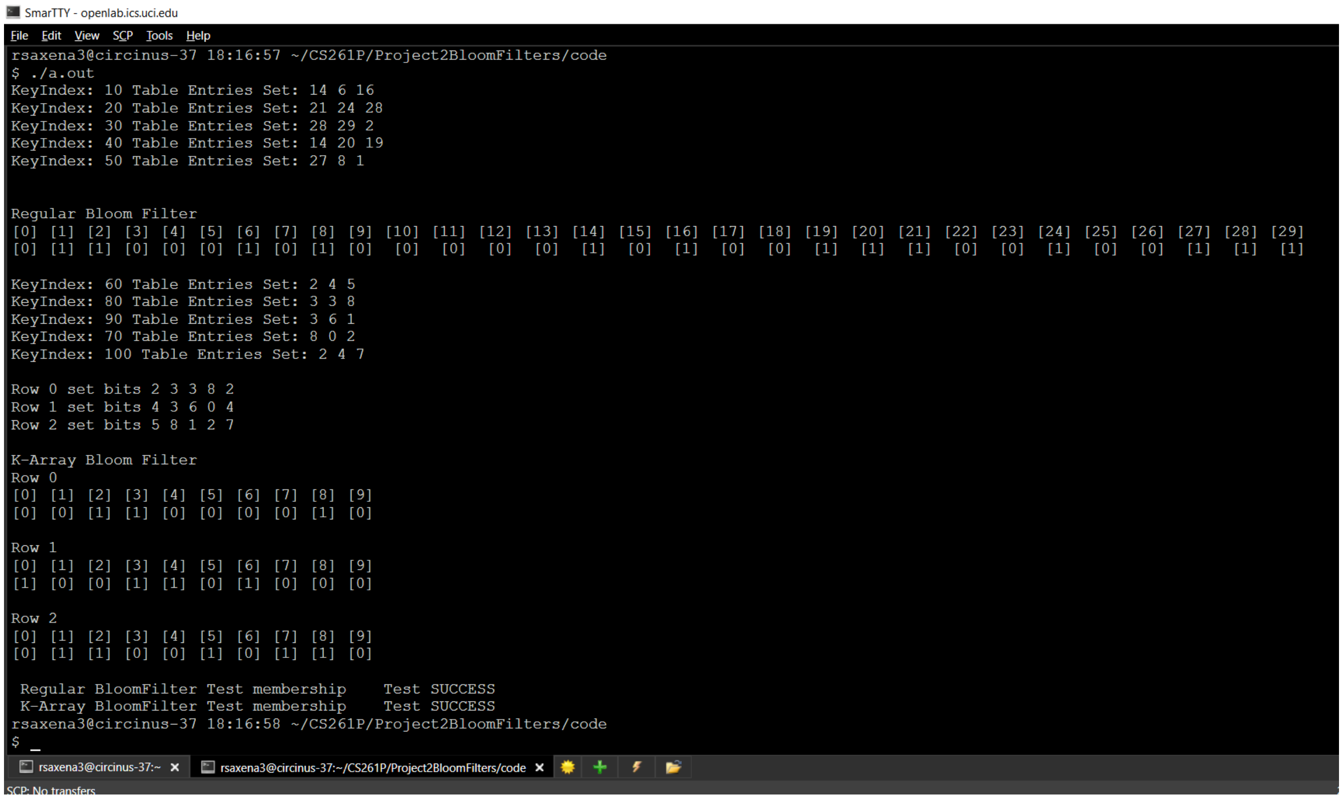
Each hash function has separate arrays for setting the bit to 1.

Index 2 in Row 0(1st array) is set as 1.

Index 4 in Row 1(2nd array) is set as 1.

Index 5 in Row 2(3rd array) is set as 1.

We tested the membership for 10 inserted keys by checking if all the bits provided by hash functions are set to 1 and concluded the test to be SUCCESS if all the 10 inserted keys got positive result.


SmarTTY - openlab.ics.uci.edu
File Edit View SCP Tools Help
rsaxena3@circinus-37:~
rsaxena3@circinus-37:~/CS261P/Project2BloomFilters/code
SCP: No transfers

```
rsaxena3@circinus-37 18:16:57 ~/CS261P/Project2BloomFilters/code
$ ./a.out
KeyIndex: 10 Table Entries Set: 14 6 16
KeyIndex: 20 Table Entries Set: 21 24 28
KeyIndex: 30 Table Entries Set: 28 29 2
KeyIndex: 40 Table Entries Set: 14 20 19
KeyIndex: 50 Table Entries Set: 27 8 1


Regular Bloom Filter
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29]
[0] [1] [1] [0] [0] [0] [1] [0] [1] [0]  [0]  [0]  [0]  [0]  [1]  [0]  [1]  [0]  [0]  [1]  [1]  [1]  [0]  [0]  [1]  [0]  [0]  [1]  [1]  [1]

KeyIndex: 60 Table Entries Set: 2 4 5
KeyIndex: 80 Table Entries Set: 3 3 8
KeyIndex: 90 Table Entries Set: 3 6 1
KeyIndex: 70 Table Entries Set: 8 0 2
KeyIndex: 100 Table Entries Set: 2 4 7

Row 0 set bits 2 3 3 8 2
Row 1 set bits 4 3 6 0 4
Row 2 set bits 5 8 1 2 7

K-Array Bloom Filter
Row 0
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] [0] [1] [1] [0] [0] [0] [0] [1] [0]

Row 1
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[1] [0] [0] [1] [1] [0] [1] [0] [0] [0]

Row 2
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
[0] [1] [1] [0] [0] [1] [0] [1] [1] [0]

 Regular BloomFilter Test membership    Test SUCCESS
 K-Array BloomFilter Test membership    Test SUCCESS
rsaxena3@circinus-37 18:16:58 ~/CS261P/Project2BloomFilters/code
$ _
```

4.2 Main Tester Implementation

This module implements the framework to collect information on the False Positive Rate (FPR) for varying values of ***n*** (number of keys inserted) and ***k*** (number of hash functions).

- The values of n range from [50, 1000] in incrementing steps of 25. (much more than the proposed [50,250])
- The values of k range from [2,10] in incrementing steps of 1. (same as proposed)
- These two parameters are easily tunable in the MainTester.cpp.
- Both Bloom Filters for each test scenario have a total of 5000 bits in them available to be set. This is slightly different from the proposal but doesn't change the analysis. A general bloom filter has 5000 buckets in one row and a k-array bloom filter has k arrays with 5000/k buckets each.
- The total n keys to be inserted are generated randomly in the range [0, 1000000). This max value can be easily changed in the code, and the value was chosen based on other parameters to allow a good range of random values and also providing room for false positives and true positives to occur.
- After inserting n keys, it is verified that all the keys are returned as present in the bloom filter. An error message is thrown if that is not the case, which wasn't seen during execution.
- For calculating False positive rate, a fixed number (10,000) of completely new random keys are generated and membership is tested on such keys. Therefore, for each pair of *(n,k)* we have a fixed number of membership tests. If such a key is returned as true it is checked if it was inserted in the Bloom Filter in which case it is a True Positive. However, if the key was not inserted then keep the count of such False positives. After all iterations, the percentage of False Positives is recorded in a .csv file.
- All random keys generated are also logged in a separate .csv file.

5. Analysis & Observations

Colab Link for Analysis:

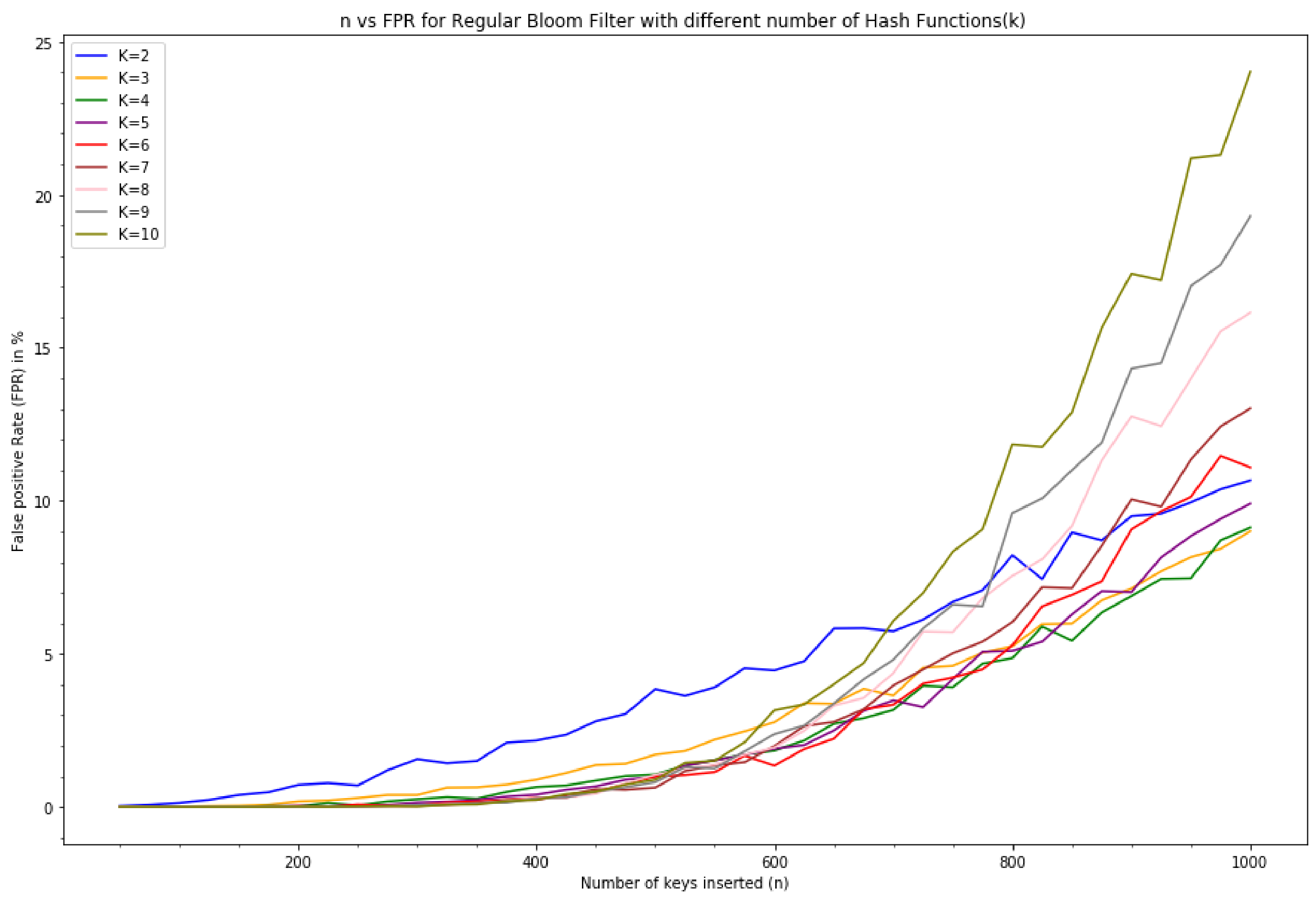
https://colab.research.google.com/drive/13vlQPALnxcF9vtEXEodb2Q520AGOgaFt

Range for n (no of insertions in filter): 50 – 1000 (in steps of 25)

Range of k (no of hash functions): 2 – 10 (in steps of 1)

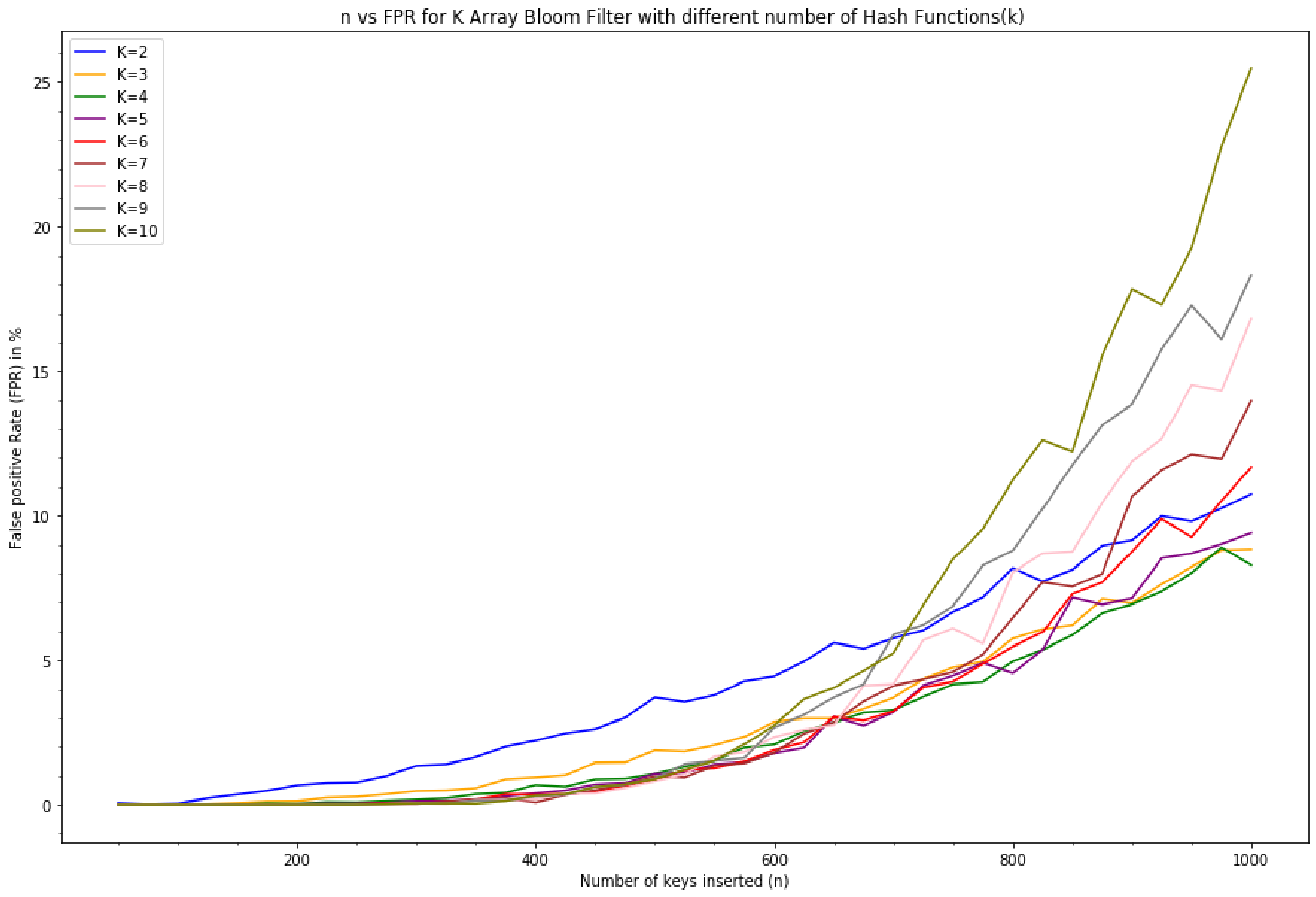
Total Data points collected: 351

The next few pages will contain graphs plotted from analyzing the data.



This graph represents the variation of False-positive Rate(FPR) with the number of keys inserted(n) for a **General Bloom Filter**. The different lines correspond to the different values of no hash functions(k). The following trends can be observed using this graph:

- The performance for k=3,4,5 is the best across all values of n.
- The performance for k=2,3 is worst for small values of n but gets relatively better for higher values of n.
- The performance for k=8,9,10 gets progressively worse as n increases.

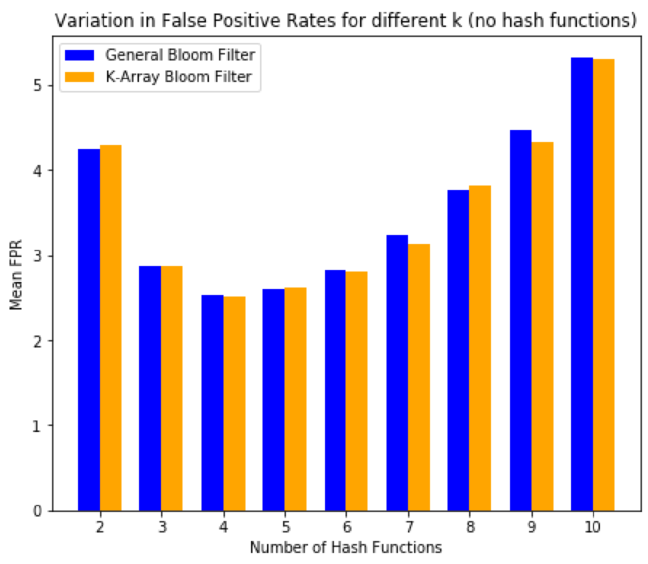


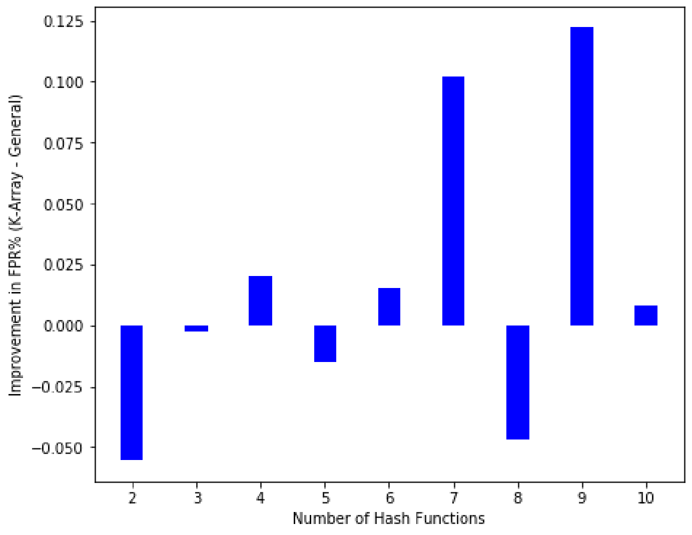
This graph represents the variation of False-positive Rate(FPR) with the number of keys inserted(n) for a **K Array Bloom Filter**. The different lines correspond to the different values of no hash functions(k). The following trends can be observed using this graph:

- The performance for k=3,4,5 is the best across all values of n.
- The performance for k=2,3 is worst for small values of n but gets relatively better for higher values of n.
- The performance for k=8,9,10 gets progressively worse as n increases.

The general trends observed for both the bloom filters are very similar and almost indifferentiable. This gives us a clue that there is only a **marginal difference in performance between the two Bloom Filters**.

To illustrate this result we have done some further analysis



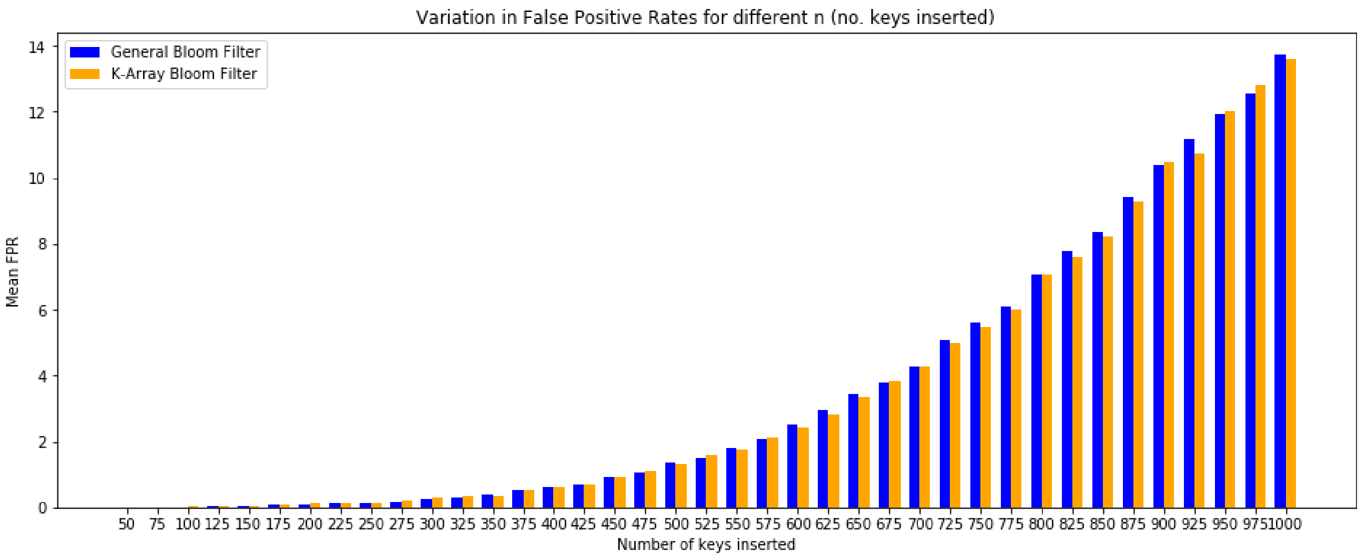


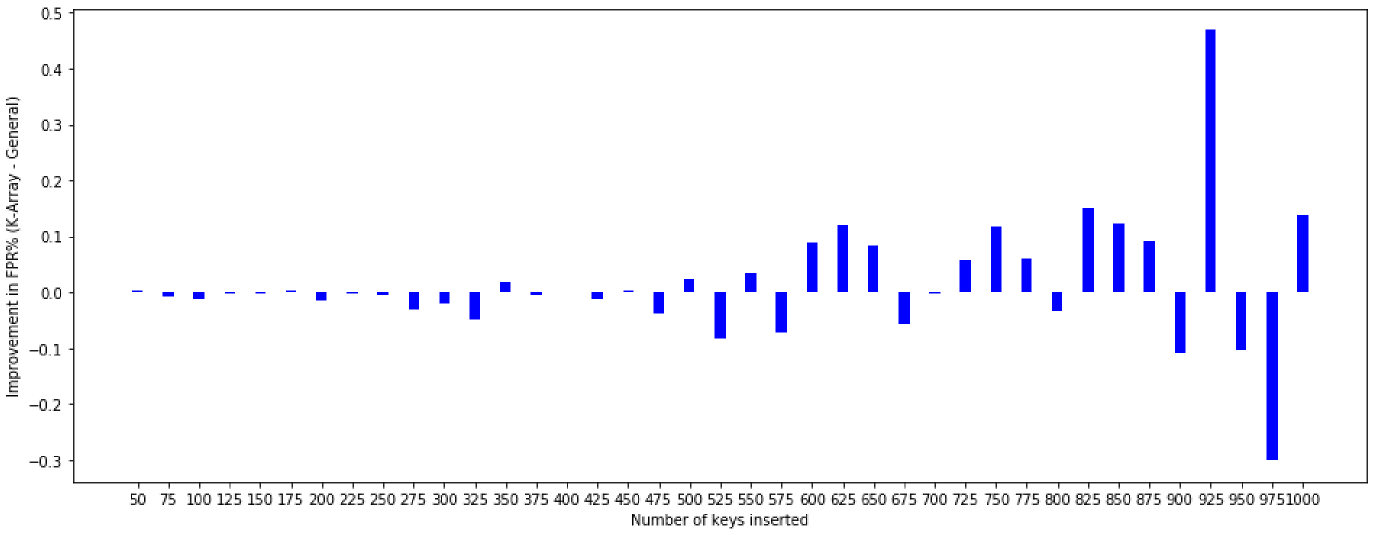
To analyze the difference in performance, we have calculated the average False Positive Rate (FPR) for both the bloom filters for varying number of hash functions ***(k)***. The average values for both Bloom filters are plotted in the bar graph and the graph on the right plots the percentage improvement of using a K-Array Bloom filter over a general Bloom filter.

The following information can be observed from the 2 graphs:

- The maximum improvement in average FPR observed for the K-Array Bloom filter is of 0.125% when k=9. This is an extremely marginal improvement and could be attributed to the exact data used.
- The maximum degradation in average FPR observed for the K-Array Bloom filter is of 0.05% for k=2. This performance degradation is again marginal at best.
- There is no clear pattern observable in the improvement and hence no relations can be inferred.

In order to improve the completeness of our analysis, we tried to plot the average False Positive Rate(FPR) vs. number of insertions***(n)*** for both the Bloom filters. Such an analysis would help us find out if one of the two bloom filter is performing better for some specific values of ***n***. We have plotted 2 graphs like the previous analysis (refer to next page). The obtained results were very similar to the one observed up until nows. There is no significant performance improvement between the two Bloom Filters.


Variation in False Positive Rates for different n (no. keys inserted)
General Bloom Filter
K-Array Bloom Filter
Mean FPR
Number of keys inserted


Improvement in FPR% (K-Array - General)
Number of keys inserted

6. Inference

The following conclusions can be drawn from our analysis:

- If the number of Hash Functions ***(k)*** in a Bloom Filter is very low (k=2,3), then False Positive Rates increases. This is because there are very few bits that are checked for each membership test, which increases the probability of False positive.
- If the number of Hash Functions ***(k)*** in a Bloom Filter is very high (k=9,10), then False Positive Rates increases. This is because the total number of bits being set in the entire filter increase by a lot, which increases the probability of False positive.
- Both the above trends are observable for both Bloom Filters.
- There is no significant improvement (or degradation) in performance observed by using a K-Array Bloom Filter over the general Bloom Filter.
- Due to the slightly complex implementation of K-Array Bloom Filter, it is probably better to use a general Bloom Filter for all real-world applications.