**BFLUF Experiment – readme file**

In the Bloom Filter (BF) experiment we build a setup in which we retrieve address as a function of a key. The key is a 32 bytes randomly generated while the address is a 10 bytes long randomly selected and attached to one of the keys. Unlike regular hash table which saves value in it the bloom table doesn’t, we achieve the same by discover the address performing a set of simple instructions.

We used two approaches to manage the bloom filter object, one by computing the size and number of hash to perform and the other was fixed size and number of hashes while inserting specific number of key-addresses to it. In addition, we also used two approaches for inserting values one was singe BF instance and the other is multi instances of BL :

* Single BF – in single BF setup we will push all values to the same BF
* Multi BF – each BF instance responsible for single address size

|  |  |
| --- | --- |
|  |  |

Test Setup

The entire test implemented in python file bllutclass.py however the actual bloom filter is implemented in BLF.py , a detailed section about it found below.



Address/key - generate randomize keys and addresses on demand and push it to container

Address list – holds two lists – present list and absent list, the absent list used as a monitor group to verify that randomize key could not get extracted from the bloom filter; the present list container will be pushed to bloom filters with an address and shall be extracted after it.

Hash Table – used for comparison – stored key-address inside

Extraction unit- gets a key and try to find if in bloom filter, we implement two extraction functions, recursive and iterative.

**Recursive extraction algorithm**

The below is a simplified pseudo code:

|  |
| --- |
| def check\_if\_a\_word\_is\_in\_bf\_recursive (word,depth)  If max depth:  Extractaddress (word)  Return  If check\_bf(word+’0’)  check\_if\_a\_word\_is\_in\_bf\_recursive (word+’0’,depth+1)  If check\_bf(word+’1’)  check\_if\_a\_word\_is\_in\_bf\_recursive (word+’1’,depth+1) |

The recursive algorithm extracts words which passes the address length bloom filter checks

**Iterative extraction algorithm**

Used for the scenario of multiple BF here is a simplified pseudo code:

|  |
| --- |
| def check\_if\_in\_multi\_instance\_bloom (key)  index=0  result\_bf = disct() // dictionary for results  while(index<=address\_size)  if (index<address\_size)  plus\_0 = word+’0’  plus\_1 = word+’1’  found\_plus\_0 = check\_if\_in\_mbf( index, plus\_0) // check if plus\_0 exist in mbf array  found\_plus\_1 = check\_if\_in\_mbf( index,pluse\_1) // check if plus\_1 exist in mbf array  result\_bf[index] = found\_plus\_0+ found\_plus\_1  if not found\_plus\_0 and not found\_plus\_1:  nothing\_found = true  if (index==address\_size or nothing\_found)  //nothingfound  Trace back to last position were 2 option found using result\_bf updates index and key (code can be found in py file )  If found:  found\_plus\_0 = False # because 0 was already checked  found\_plus\_1 = True # because we need to progress with 1 now  if found\_plus\_0  result\_bf[index]-- //removes 1 from result buffer  key = plus\_0  index++  if (index==address\_size)  add\_to\_found\_list(key)  key = key – last char // remove last char  continue // go to next while iteration plus 1 will be checked later  if found\_plus\_1  result\_bf[index]-- //removes 1 from result buffer  key = plus\_1  index++  if (index==address\_size)  add\_to\_found\_list(key )  key = key – last char // remove last char |

Tester – perform E2E test in python file look for run\_test procedure

|  |
| --- |
| **- Build key-address tables for absent a present words**  **- Push present words to hash**  **- Push present words to single instance bloom filter**  **- Push present words to Multi instance bloom filter**  **- Check the monitor group for absent words**  **- For each present word**  **o Check if in single instance bloom filter**  **o Check it in multi instance bloom filter**  **o Check if in hash table**  **- Collect statistics**  **-** |

**BLF.py**

Initialization of BF has two options

1. By inserting expected number of items and Fault positive probability
2. By making static allocation with number of hashes and size of bit-array

The supported operations are

* Add item
* Check if item exist
* Get size
* Get number of hashes
* Get load factor

**Main.py**

Orchestrate the test , perform number of operations collect statistics and display results the supported methods are:

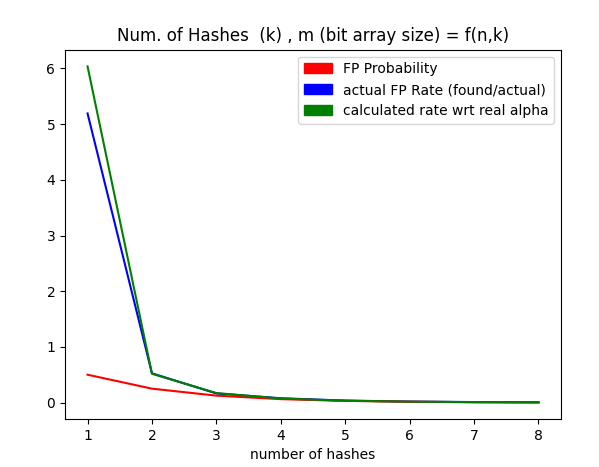
***check\_inc\_static\_bf\_k\_iteration:***

Runs 8 tests iterates on k which is the number of hashes value.

Each iteration is a 100 of iteration of bflutclass runtest, so eventually we each time average from 100 samples

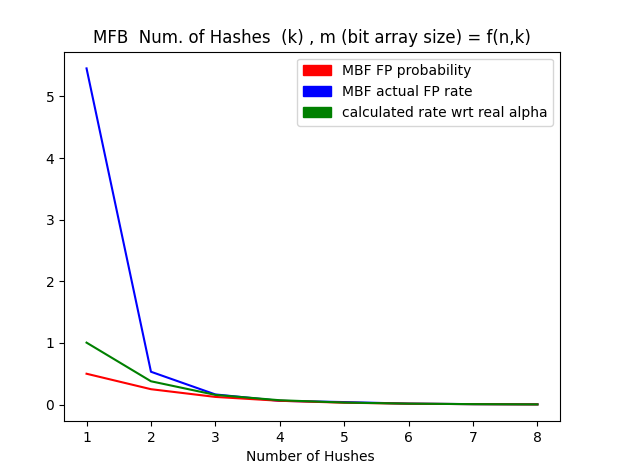
**Results:**

For single Hash function:



This graph shows

* The FP rate probability is dropping when raising the number of hash functions

For Multi Bloom filters:

This graph shows

* Calculated FP rate is for single entry to Bloom filter as such the rate is lower in one hash function .
* Actual FP rate is similar tosigle hash meaning this approach didn’t improve anything

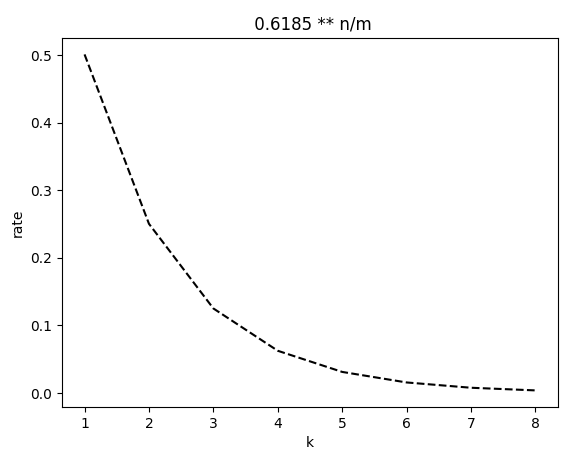
The size of BF is computed as follows:

n- the number of expected items (in our case it is 16)

m- the size of bit array = int(k \* n \* 10 / math.log(2)) while k is the number of hashs.

If gives as to a situation that the probability for false positive as describe in Probability and Computing Randomized Algorithms and Probabilistic Analysis by Michael Mitzenmacher & Eli Upfal:





The drop of probability explains the behavior of the FP rate above

***check\_inc\_static\_bf***

Iterates the n size while the k is statically 2