Data Structures

Term Paper

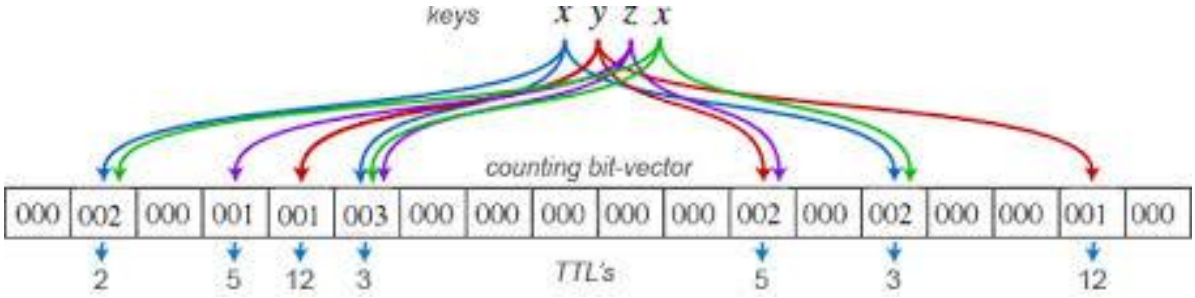
BLOOM FILTERS

By:-

Udyan Khurana (201202101)

Gaurav Mishra (201202057)

1. What Are Bloom Filters?



A Bloom filter, conceived by Burton Howard Bloom in 1970, is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. It uses hashing to insert values into a table and the same to retrieve data back from that table. The basic operation of a bloom filter, besides insertion, is a membership query (checking membership of an element in the bloom filter set). Elements can be added to the bloom filter set, but not removed (though this can be addressed with a counting filter). The more elements that are added to the set, the larger the probability of false positives. The main goal while designing a good bloom filter is to ensure that:

1. The probability of false positives are kept minimal
2. The retrieval time is very fast.

2. Some Important Terms:-

1. False Positives:- False positives are those results when we assert that the result is true but in actuality the result is not true. For example, assume that a number 237 is not a member of a set. We give a query on that set to check whether 237 is a member of a set or not and the answer we get is true, while actually it should have been false.

2. False Negatives:- False Negatives are those results when we assert that the result is false but in actuality the result is true.

In a bloom filter, False positive retrieval results are possible, but false negatives are not; i.e. a query returns either "inside set (may be wrong)" or "definitely not in set". This is because while inserting an element into a bloom filter set, the identity of that element in that set is due to k values obtained using the k hash functions, as only those bits are set to 1 when this value is added to the set. But

suppose some other value is hashed by those hash functions and suppose hash6(val1) returns same value as hash4(val2). Now if this is possible, a scenario is also possible where the identity of an element is overwritten by combination of other values getting hashed and returning values that are same as the k identities of our element. So, it may seem that this element is a part of the set, when it is actually not.

3. How A Bloom Filter Functions:-

A bloom filter consists of an n bit array with all bits initially set to 0. m values are inserted to it through k hash functions. Each of the hash functions we have implemented map the given input value to a 32-bit constant and then all the k 32-bit hashed constants corresponding to that value are set to 1.

Consider a set $A = \{a_1, a_2, ..., a_n\}$ of n elements. Bloom filters describe membership information of A using a bit vector V of length m. For this, k hash functions, $h_1, h_2, ..., h_k$ with $h_i : X \to \{1..m\}$, are used.
The following procedure builds an m bits Bloom filter, corresponding to a set A and using $h_1, h_2, ..., h_k$ hash functions:

```
Procedure BloomFilter(set A, hash_functions, integer m)
      filter = allocate m bits initialized to 0
      for each ai in A:
            for each hash function hj:
                  filter[hj(ai)] = 1
                  end for
      end for
      return filter
```

To find whether a value is inserted or not we hash the value through the same k hash functions and see if all the corresponding bits are 1. If any of them is 0, we can surely say that it has not been inserted and any other bit which is 1 it would have been set by some other value.

```
Procedure Membership(set A, hash_functions, integer m)
      for(i = 0; i < k; i++)
            if(!array[h_i(m)])
                  return false
      end for
      return filter
```

But it may so happen that a combination of other values have set all these k bits to 1, giving us the illusion that this value had been inserted while actually it was not. This phenomenon gives us False positive results, which need to be minimized. This can be done by using the following 2 ways:-

1. Choosing optimal values of k, m, and n to minimize the Probability of false positives. How this can be done mathematically is explained in the following section.
2. Using good non-cryptographic hash functions like: MurmurHash3, FNV-1 Hash, Jenkins Hash functions etc.

4. Bloom Filters - The Math:-

One prominent feature of Bloom filters is that there is a clear tradeoff between the size of the filter and the rate of false positives.

Observe that after inserting m keys into a filter of size n using k hash functions, the probability that a particular bit is still 0 is:

$$p_0 = \left(1 - \frac{1}{n}\right)^{km} \approx 1 - e^{-\frac{km}{n}} \tag{1}$$

Note that we assume perfect hash functions that spread the elements of A evenly throughout the space {1..n}. Hence, the probability of a false positive (the probability that all k bits have been previously set) is:

$$p_{err} = (1 - p_0)^k = \left(1 - \left(1 - \frac{1}{n}\right)^{km}\right)^k \approx \left(1 - e^{-\frac{km}{n}}\right)^k \tag{2}$$

In formula (2), p_{err} is minimized for number of hash functions = $k = \frac{n}{m} \ln 2$

In practice however, only a small number of hash functions are used. The reason is that the computational overhead of each hash additional function is constant while the incremental benefit of adding a new hash function decreases after a certain threshold, which is 22 hash functions (determined experimentally).

To summarize:

Bloom filters are compact data structures for probabilistic representation of a set in order to support membership queries. The main design tradeoffs are:

1. The number of hash functions used (driving the computational overhead)
2. The size of the filter.
3. The error (collision) rate.

Formula (2) is the main formula to tune parameters according to application requirements.

5. IMPLEMENTATION OF THE BLOOM FILTER:-

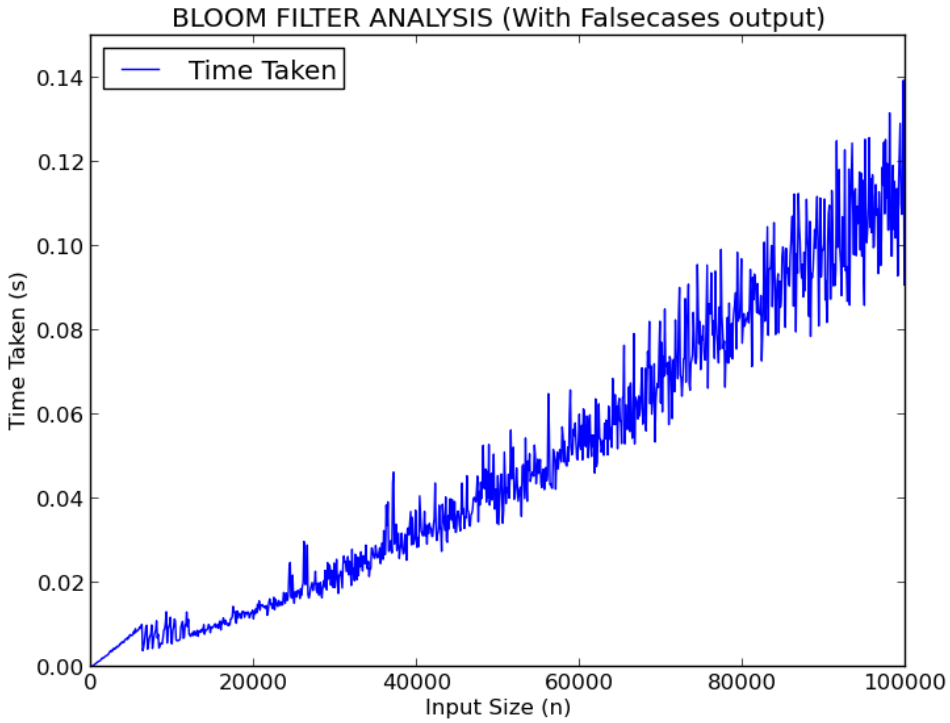
In our project on Bloom filters, initially we made the basic bloom filter for numbers. Later on we compared the performance of Bloom filters on the operations of storing and searching on random number datasets of various sizes ranging from 1 element to the order of 10^5 elements against AVL tree and arrays by using a total of 10 hash functions where 3 are standard hash functions and the other are a combination of those. We came at the number 10 as we first set a probability target of 0.1% and if we choose k optimally using the formula given above,

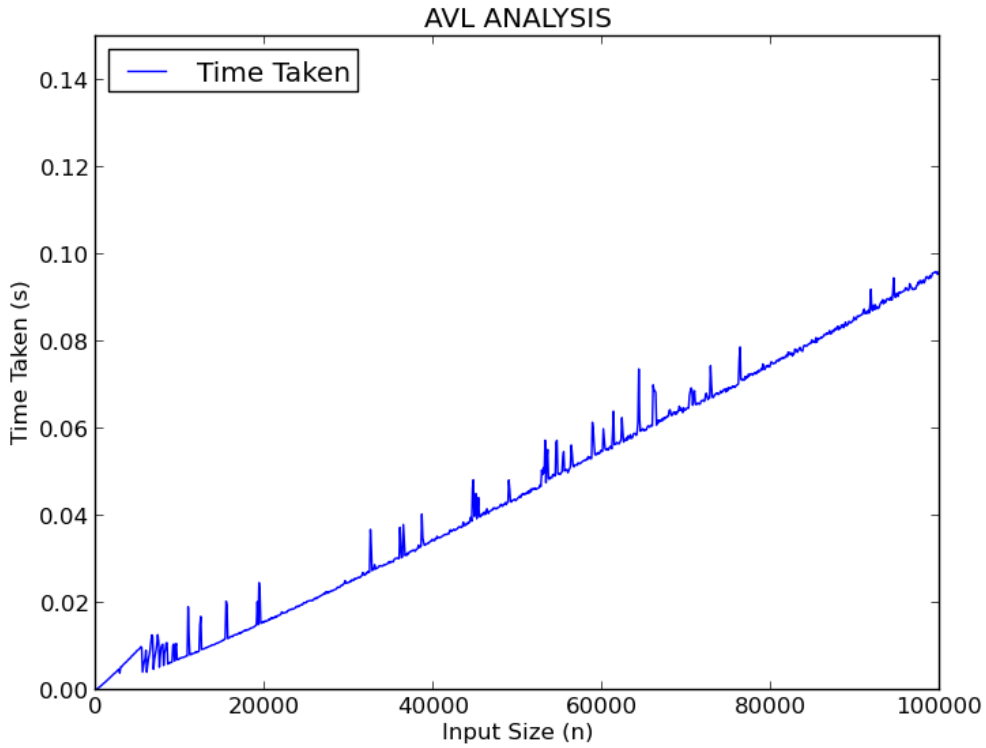
Probability p that we want = $(0.5)^k$

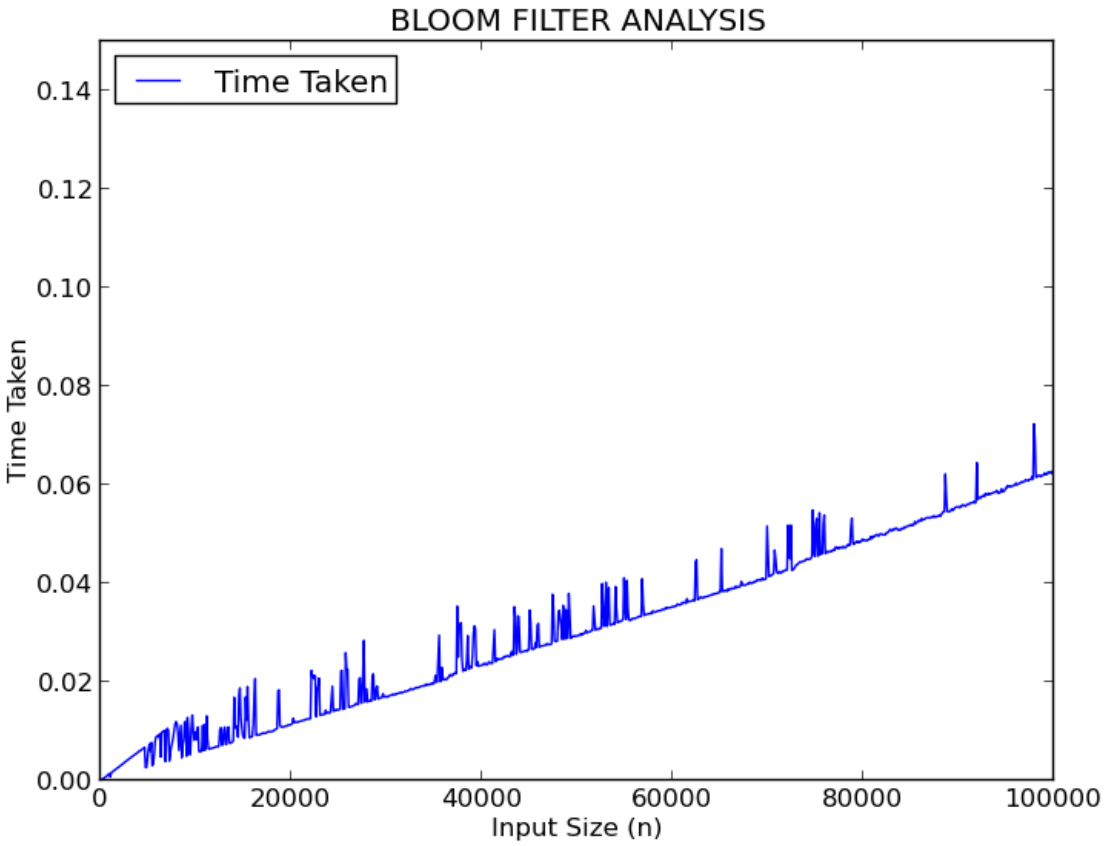
= 0.001

This gives k=10. We assumed n as 10000000, using which upper limit of m can be found. Our tested bloom filter gives us a better-than-expected 0.08% error rate.

Performance Graphs:







These graphs were made for a number of test files (approx. 1000) with number of queries and number of entries in the bloom filter set varying from 1 to the order of 10^5. This was done using a python script which generated all the input files using 'random' library of python and using os module system commands were given to execute the files and time was taken inside the C codes ('-time' appended to these codes at the end) using time.h header file. Then these 'times' were used by 'matplotlib' python library to plot the graphs.

The input files folder was too large so we have attached 4 sample files with their outputs as were run on the terminal, and the test file generator python script that we used.

Results and Analysis:

1. graph-bloom.png - Too much variation in output time due to variable time taken by linear searching.
2. For the other 2 graphs (graph-bloom_without_linearsearch.png and graph-avl.png) - Small variations due to small variations in:
    - search depth variation in AVL
    - hash time in BLOOM FILTER

We also observed that although time for insertion and searching in bloom filter was O(1), but as the input size increases from 1 to 10^5, total time to process and output the result will obviously increase, thus the linear-looking graph.

6. APPLICATION - SPELL CHECKER

We made a spellchecker using our bloom filter after making it so as to function for string inputs. We included a version of the British English Dictionary containing approx. 80,000 words through our 10 hash functions. Then we asked the user to input a paragraph and we gave him all words with incorrect spellings.