Bloom Filters Project Report **Ravish Chawla**
CS 6505
Prof. Vigoda April 1, 2016

Project Overview

Bloom Filters are an efficient Data Structure for Hashing a large number of elements. In this project, the performance of Bloom Filters was tested, and compared against optimal values. A Bloom Filter was implemented in C++, using a standard hashing algorithm. It was tested on different input sizes, for each of which the False Positive rate was calculated for the elements inserted.

Implementation and Performance Testing

The Bloom Filter is programmed in C++, and the implementation uses standard libraries and functions. The Bloom Filter keeps an underlying array structure. Inserting an element sets multiple array slots, based on the indices the element maps to. A query for an element checks that all the Hash Indices that the element maps to have been set, returning false if at least one such index exists that has not been set.

The Hash Functions used in this implementation are also programmed in C++, and are based on the Non-cryptographic FNV-1 Hashing algorithm [1]. The Hash is computed by iterating over each byte of the input, and calculated as its XOR with the product of the previous Hash value with a large prime number. The initial value is obtained by the standard c++ *rand*() function. The Bloom Filter creates a set of Hash Functions on initialization. The initialization takes two values as input, n and m. The second value is provided as a ratio, $c = n/m$, which is used to calculate the number of Hash Functions k, as $c\ln(2)$, the optimal number to minimize False Positives.

The main Driver of the test program tests for the False Positive rate, for different values of n and c. False Positive rate is a good metric for Bloom Filters because the design of Bloom Filters permits them. Since multiple Hash indices are set when an element is inserted into a Bloom Filter, all indices that an element maps to could be set by different insertions.

The program initializes different Bloom Filters, with sizes ranging from $1e2$ to $1e9$, and c ratio values in the range 2 through 10. The dataset used is a random sample of n integers. For each Bloom Filter, $m = n/c$ elements are inserted, after which all n values were queried, to accumulate the number of false positives. For each value of c, the theoretical False Positive rate was also calculated. The data accumulated from Performance testing was then written to a file, which is used to plot the values on a chart, generated in Javascript.

Optimal value for False Positive Rate

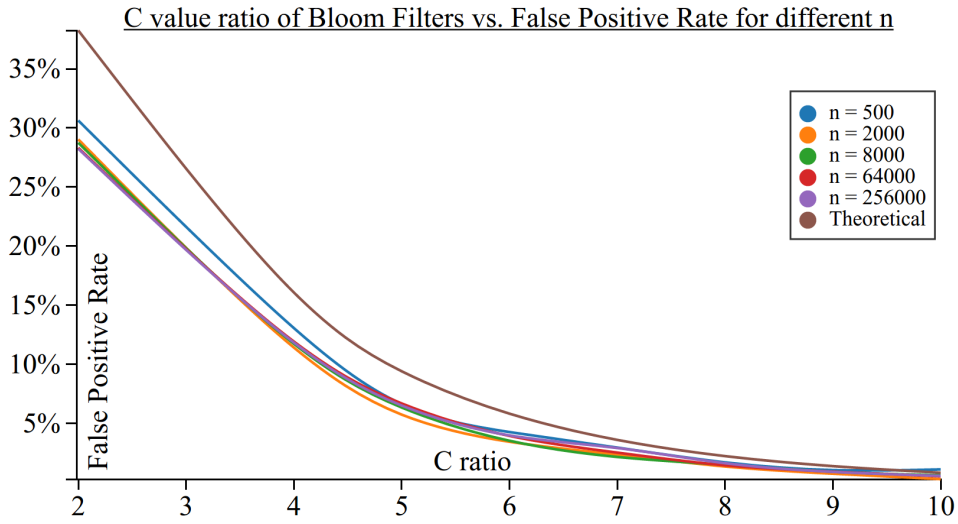
In an ideal Bloom Filter of size n, each Hash Function has a probability of selecting a Hash Index with equal probability. Using this assumption, the probability that an element will be a False Positive can be computed. Since an insertion sets k hashed slots, for a value to be False Positive, all k indices that the value maps to would have to be set. The rate can be minimized by using the optimal value of k, which is $k = c\ln(2)$. probability can then be computed as:

$$\begin{aligned} &Pr(\text{false positive}) = Pr(\text{for all } i, H[h(i)] = 1) = Pr(H[a] = 1)^k = (1 - Pr(h[a] = 0)^k \\ &Pr(H[a] = 0)^k = (1 - 1/n)^{mk} \approx (e^{-1/n})^{mk} = e^{-k/c}, \text{where } c = n/m \\ &Pr(H[a] = 1)^k = (0.6185)^c, fork = c\ln(2), \text{the most optimal value for } k \end{aligned} \tag{1}$$

Results

The figure below shows how the False Positive Rate as the value c increases, for different Bloom Filters, including the Theoretical. The c ratio, which is on the bottom axis, dictates how many elements are

added. Because the number of elements inserted are proportional to n, such that $m = n/c$, a lower c value means that more elements were inserted. It also means that fewer Hash Functions were used for each element. The results show that the False Positive rate increases exponentially with the number of inserted elements in a Bloom Filter.



Another important observation from the graph is the relation between the False Positive Rate and c, the size of the filter.As the capacity of the Bloom Filters increase, the False Positive rate decreases. This is shown in the chart where the path for $n = 5000$ is visibly higher than the paths for n larger than it. Another important observation from this chart is that the Theoretical rate is higher than the experimental rates, when c is small. An explanation for this is that the Theoretical value is an approximation of the equation that represents the probability of a Theoretical False Positive, as shown by 1. Another reason for this is that the probability equation is based on two assumptions, that the selection of each Hash Index is equally likely, and that the dataset is completely random and that the selected Hash Indices are independent of each other. The latter condition is not completely satisfied in this implementation. It arises as a limitation of the random function used in this program, *rand*(). Although the function is seeded with a unique value upon each instantiation, it is only pseudo-random, and the value it returns uses previous values in the sequence in its computation. As a result, the implementation produces results that are better than the theoretical, when c is small.

Final Conclusion

Implementing a Bloom Filter and visualizing its results helps show how effective and useful Bloom Filters are for hashing. It illustrates both the advantages and shortcomings of Bloom Filters. As the experimental results showed, the False Positive rate is lowest when the c ratio is smaller, which means when proportionally fewer elements are inserted. This is a major disadvantage of Bloom Filters. To be able to insert m elements efficiently, a Bloom Filter of size 8 - 10 times m must be used. However, regardless, it is only a constant amount of space. The space complexity of the Bloom Filter is still linear, and in the order of $O(\text{cm}) = O(\text{n})$.

The operations of the Bloom Filter are also efficient, with both the insertion and querying operations running in the order of $O(\text{k})$ time, where k is the number of Hash Functions. A major advantage of Bloom Filters is that its design permits extremely efficient data structures for storage, that require only 1 bit to be set. The efficiency of Bloom Filters, in both time and space, make them extremely useful as a Hashing structure.

References

[1] Noll, Landon Curt 2015. *FNV Hash*[online]. [Accessed 31 March 2016]. Available: http://www.isthe.com/chongo/tech/comp/fnv.