CS 51 Project: Scalable Bloom Filters, the Algorithms

May 1, 2015

Joseph Kahn, Grace Lin, Aron Szanto
Cambridge, MA 02138

Contents

1 Static Bloom Filters

Fundamentally, a Bloom filter is a probabilistic data structure. By giving up the ability to definitively check the presence of an element in the data structure, a Bloom filter is able to store elements with a constant element/memory ratio, regardless of the size of the element. The size of the bitset, m, and the number of hash functions, k, can be modified to accomodate any number of inputs, n, with any desired false positive percentage, *errbound*.

The probability for a given bit being set by a particular hash function for a particular entry is $\frac{1}{m}$. Therefore, the probability for the bit of interest not to be set is $1 - \frac{1}{m}$. It follows that the probability for a particular bit to still be 0 after n inputs is $(1 - \frac{1}{m})^{nk}$. Therefore, the probability for the bit to be set to 1 after n inputs is $1 - (1 - \frac{1}{m})^{nk}$, which for very large n and m can be approximated as $(1 - e^{-\frac{kn}{m}})^k$.

Note: This approach assumes that the hash functions are independent and that there will be no collisions across hash functions. However, for very large m, n, this approximation is close enough to calculate a false positive percentage.

As derived above, $errbound = (1 - e^{-\frac{kn}{m}})^k$. For a given m, n, we can treat $errbound$ as a single-variable function of k, and using some simple calculus, find the value of k with respect to at the minimum $errbound$. The zero of the derivative of $errbound$ is $k = \frac{m}{n} ln(2)$. Substituting this k value back into $errbound$, $errbound$ simplifies to $errbound = (1 - e^{-ln(2)})^{\frac{m}{n} ln(2)}$, which cleans up into $ln(errbound) = -\frac{m}{n}(ln(2))^2$. Since $errbound$ and n are given, we can thus derive $m = -\frac{nln(errbound)}{(ln(2))^2}$. With this derivation of m, we can also simplify k to be $-\frac{ln(errbound)}{ln2}$.

The explanation for the optimal $fill$ value is as follows: given an error bound and an expected number of elements, the optimal k value is $k = \frac{m}{n} ln2$. $fill = 1 - e^{-\frac{km}{n}}$. Substituting $ln2 = \frac{km}{n}$, $fill = \frac{1}{2}$.

2 Scalable Bloom Filters

A scalable Bloom filter (SBF) deals with the problem of having to a priori declare the expected number of inputted elements when creating a Bloom filter. While a Bloom filter will never reject an input, if the number of bits set to 1, or the fill ratio, increases beyond a certain threshold (we show above that this threshold is $\frac{1}{2}$), then the false positive probability of the Bloom filter will increase beyond the desired error bound. We can overcome this problem by creating a series of Bloom filters. The initial Bloom filter is simply a static Filter of some arbitrary size, and once this Bloom filter has been filled to a certain fill ratio, a subsequent Bloom filter is created, and so on. The SBF struct stores several things: an array of all the existing Bloom filters, the number of Bloom filters, a scaling factor that determines how much bigger each subsequent Bloom filter will be, and a tightening ratio that determines how much smaller the error bound on each subsequent Bloom filter will be.

We will show that such error bounds will converge. For a SBF with l filters, initial error bound e_0 and tightening ratio r, with $e_i = e_0 * r^i$. The overall errorbound, e_tot, is equal to $1 - \prod_{i=0}^{N} 1 - e_i$. This is bounded by $\prod_{i=0} e_i = \frac{e_0}{1-r}$. Thus, e_tot converges to approximately $\frac{e_0}{1-r}$. Thus, given a particular total error bound and an r value, it is possible to calculate the necessary error bound for each static filter within the SBF.

3 Partitioned Bloom Filters

For the third part of our project, we implemented partitioned Bloom filters in scalable Bloom filters. Partitioned Bloom filters are more efficient because they prevent collisions/overlaps between hash functions. Algorithmically, calculating fill ratios and error bounds for each partition are equivalent to setting k to 1. The total error bound is $err_tot = fill^k$, where $fill$ is the fill ratio of a particular slice $(e = 1 - (1 - \frac{1}{m})^n$, where m is the size of the slice. Using a Taylor expansion, $fill$ can be approximated by $1 - e^{-\frac{n}{m}}$. It easily follows, through some algebra, that n can be approximated by $M\frac{ln(1-fill)}{-k}$. This gives us a M value, which is the size of the bitset, of $n\frac{-k}{1-fill}$. $fill^k = err_tot$, so with a fill of $\frac{1}{2}$, we get $k = -\frac{ln(err_tot)}{ln2}$. So, $M = n\frac{2lnerr_tot}{ln(2)}$. If we compare the coefficient of M and the coefficient of m from the static classic Bloom filter previously discussed, the coefficient for M is about 20 percent smaller than that of m. Given the same number of values and the same total error bound, a partitioned Bloom filter is about 20 percent more memory efficient than a standard Bloom filter. Comparatively, had we chosen to increase k significantly rather than partition the bitset, the number of collisions would have also increased significantly and thus reduced the efficiency of our Bloom filter.