Bloom filters and Hashing

1 Introduction

The Bloom filter, conceived by Burton H. Bloom in 1970, is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not in this structure. Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

Bloom filter is widely used in various computer science areas. People use Bloom filter to consisely represent a set of data. For example, in the networking area, a Bloom filter will often be constructed by one system and sent over a network to a recipient. The Bloom filter for a set is much smaller than the set itself, which makes it appropriate for sending over a network (or for storing the filter at a higher level in memory hierarchy when the full set will not fit).

2 Set up of Bloom filter

We want to represent n-element sets $S = s_1, s_2, ..., s_n$ from a very large universe U, with $|U| = u >> n$. We want to support insertions and membership queries (as to say "Given $x \in U$, is $x \in S$?") so that :

1. If the answer is No, then $x \notin S$.

2. If the answer is Yes, then x may or may not be in S, but the probability that $x \notin S$ (false positive) is low.

Both insertions and membership queries should be performed in constant time.

A Bloom filter is a bit vector B of m bits, with k independent hash functions $h_1, ..., h_k$ that map each key in U to the set $R_m = 0, 1, ..., m-1$. We assume that each hash function h_i maps a uniformly at random chosen key $x \in U$ to each element of R_m with equal probability. Since we assume the hash functions are independent, it follows that the vector $(h_1(x), .., h_k(x))$ is equally likely to be any of the m^k k-tuples of elements from R_m.

- Initially all m bits of B are set to 0.
- Insert x into S. Compute $h_1(x), ..., h_k(x)$ and set $B[h_1(x)] = B[h_2(x)] = ... = B[h_k(x)] = 1$.
- Query if $x \in S$. Compute $h_1(x), ..., h_k(x)$. If $B[h_1(x)] = B[h_2(x)] = ... = B[h_k(x)] = 1$ then answer Yes, else answer No.

Bloom filters are popular with software engineers as they achieve provably good performance (see analysis below) with little effort: simple hash functions, simple algorithms (no collision handling etc.), efficient use of space. The main drawback of Bloom filters is that it is difficult to store additional information with the keys in S.

3 Analysis

Now we will analyze the probability of a false positive. The probability that one hash do not set a given bit is $1 - \frac{1}{m}$ given the setting in previous section. The probability that it is not set by any of the hash functions is $(1-\frac{1}{m})^k$ .Hence, after all n elements of S have been inserted into the Bloom filter, the probability that a specific bit is still 0 is:

$$f = (1-\frac{1}{m})^{kn} = e^{-\frac{kn}{m}}$$

(Note that this uses the assumption that the hash functions are independent and perfectly random.) The probability of a false positive is the probability that a specific set of k bits are 1, which is

$$(1-(1-\frac{1}{m})^{kn})^k = (1-e^{-\frac{kn}{m}})^k$$

This shows that there are three performance metrics for Bloom filters that can be traded off: computation time (corresponds to the number k of hash functions), size (corresponds to the number m of bits), and probability of error (corresponds to the false positive rate).

Suppose we are given the ratio $\frac{m}{n}$ and want to optimize the number k of hash functions to minimize the false positive rate f. Note that more hash functions increase the precision but also the number of 1's in the filter, thus making false positives both less and more likely at the same time. Let

$$g = ln(f) = kln(1-e^{-\frac{kn}{m}})$$

Suppose $p = e^{-\frac{kn}{m}}$. Then, the derivative of g is

$$\frac{dg}{dk} = ln(1-p) + \frac{kn}{m}\cdot\frac{p}{1-p}$$

We find the optimal k, or right number of hash functions to use, when the derivative is 0. The solution is $k = (ln2)\cdot\frac{m}{n}$. So for this optimal value of k, the false positive rate is:

$$(\frac{1}{2})^k = (0.6185)^{\frac{m}{n}}$$

As m grows in proportion to n, the false positive rate decreases.

4 Lower Bound

One way to determine how efficient Bloom filters are is to consider how many bits m are necessary to represent all sets of n elements from a universe in a manner that allows false positives for at most a fraction ϵ of the universe but allows no false negatives. We derive a simple lower bound on m for this case.

Suppose that our universe has size u. Our representation must associate an m-bit string with each of the $\binom{u}{n}$ possible sets. Consider a specific set X of n elements. Any string s that is used to represent X must accept every element x of X, but it may also accept $\epsilon(u-n)$ other elements of the universe while maintaining a false positive rate of at most ϵ. Each string s therefore accepts at most $n+\epsilon(un)$ elements. A fixed string s can be used to represent any of the $\binom{n+\epsilon(un)}{n}$ subsets of size n of these elements, but it cannot be used to represent any other sets. If we use m bits, then we have 2^m distinct strings that must represent all the $\binom{u}{n}$ sets. Hence we must have

$$2^m\binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n}$$

then we can get

$$m \geq log_2 \frac{\binom{u}{n}}{\binom{n+\epsilon(u-n)}{n}} \approx log_2 \frac{\binom{u}{n}}{\binom{\epsilon u}{n}} \geq log_2 \epsilon^{-n} = nlog_2(1/\epsilon)$$

We, therefore, find that m needs to be essentially $nlog_2(1/\epsilon)$ for any representation scheme with a false positive rate bounded by ϵ.

We already know:

$$f = (1/2)^k \geq (1/2)^{mln2/n}$$

We can find that $f \leq \epsilon$ requires

$$m \geq n\frac{log_2(1/\epsilon)}{ln2} = nlog_2 e \cdot log_2(1/\epsilon)$$

Thus, space-wise Bloom filters are within a factor of $log_2 e \approx 1.44$ of the lower bound. Alternatively, keeping the space constant, if we use $n \cdot j$ bits for the table, optimal Bloom filters will yield a false positive rate of about $(0.6185)^j$, while the lower bound error rate is only $(0.5)^j$.

5 Partitioned Bloom Filters

Now we will describe a simple variant of the standard Bloom filter presented in the previous sections. This variant won't perform better than the standard version, but it will be useful for the analysis of other interesting variants.

While in the standard version the hash functions had range $\{1, \ldots, m\}$ in this variant every hash function has a disjoint set of bit locations and range $\{0, \ldots, p-1\}$, with $m = pk$. When inserting element x, we will set to 1 the bits $h_1(x)$, $h_2(x) + p$, $h_3(x) + 2p$, and so on.

In this setting, if the Bloom filter contains set $S = \{x_1, \ldots, x_n\}$ and we check for an element y that is not in S, a false positive happens if and only if:

$$\forall i \in [1..k] \exists x \in S : h_i(y) = h_i(x)$$

In other words, there is false positive if there is a collision in all the hash functions. We can compute the false positive probability:

$$
\begin{aligned}
P(F) &= \prod_{i=1}^{k} P(\text{Collision in hash i}) \\
&= \prod_{i=1}^{k}(1 - \prod_{x\in S} P(h_i(x) \neq h_i(y))) \\
&= \prod_{i=1}^{k}(1 - \prod_{x\in S}(1 - \frac{1}{p}) \\
&= \prod_{i=1}^{k}(1 - \prod_{x\in S}(1 - \frac{k}{m}) \\
&= \prod_{i=1}^{k}(1 - (1 - \frac{k}{m})^n) \\
&= (1 - (1 - \frac{k}{m})^n))^k \\
&\approx (1 - e^{-kn/m})^k
\end{aligned}
$$

This variant has the same asymptotical false positive probability, but in practice it behaves worse than the standard version., since $(1 - \frac{k}{m})^n)) < (1 - \frac{1}{m})^{kn}$

6 Using only two hash functions

One of the biggest problems of the standard bloom filter is the need for a big number of random hash functions. We will now see a variant of the partitioned Bloom filter that needs only two independent hash functions and achieves the same asymptotic false positive probability.

In this variant we start from two indipendent random hash function $h_1(x)$ and $h_2(x)$ with range $\{0, 1, \ldots, p-1\}$ for a prime p. We will obtain a Bloom filter with k hash functions with range $\{0, 1, \ldots, p-1\}$ and an hash table of $m = kp$ bits. The k hash functions are defined like this:

$$g_i(x) = h_1(x) + i * h_2(x) \pmod p$$

Lemma 1 *For every $x, y \in U$ and any choice of h_1, h_2, we can have only one of the following three cases:*

1. *$g_i(x) \neq g_i(y)$ for all i*
2. *$g_i(x) = g_i(y)$ for one i*
3. *$g_i(x) = g_i(y)$ for all i*

This lemma means that if two different elements have at least two collisions between themselves then they have the same starting hash functions. Proof:

$$\begin{aligned}
g_i(x) &= g_i(y) \\
g_j(x) &= g_j(y) \\
g_j(x) - g_i(x) &= g_j(y) - g_i(y) \\
(h_1(x) + j * h_2(x)) - ((h_1(x) + i * h_2(x)) &\equiv (h_1(y) + j * h_2(y)) - (h_1(y) + i * h_2(y)) \pmod p \\
(j-i) * h_2(x) &\equiv (j-i) * h_2(y) \pmod p \\
(j-i) * (h_2(x) - h_2(y)) &\equiv 0 \pmod p
\end{aligned}$$

Since p is prime and $j \neq i$ we can see that $h_2(x) = h_2(y)$ and then $h_1(x) = h_1(y)$. Let's call this event ε.

$$Pr(\varepsilon) = 1 - (1 - \frac{1}{p^2})^n = 1 - (1 - \frac{k^2}{m^2})^n$$

Since this probability is very small and we are interted in the asymptotic false positive probability, we can ignore this case.

Now, we have to analize the false positive probability in the case that there are not more than 1 collision betweeen the new element and any other element. We can see this setting like a balls and bins problem: if the new element and an old element have a collision on hash g_i we see as if the old element has sent a ball on bin i. Every element send at most one ball and there is a false positive if and only if all bins are covered. Let's study the probability that the new element (y) has a collision with an old element(x) on hash i.

$$\begin{aligned}
g_i(x) &= g_i(y) \\
h_1(x) + i * h_2(x) &\equiv h_1(y) + i * h_2(y) \pmod p \\
h_1(x) &\equiv h_1(y) + i * h_2(y) - i * h_2(x) \pmod p
\end{aligned}$$

We can see that for every choice of $h_2(x)$ there is a $h_1(x)$ so that there is a collision. Also, if $h_2(y) = h_2(x)$ then we obtain that $h_1(y) = h_1(x)$ and that there is more than one collision between x and y. Since we are working with the assumption that there cannot be more than one collision between a new element and an old element, we don't consider this case. We can conclude that there is a total of $p-1$ possible pairs of $(h_1(x), h_2(x))$ so that y has a collision with x on hash i.The pair $(h_1(x), h_2(x))$ is uniformly distributed on $p^2 - 1$ pairs (the pair in which we get the same values as for y is not allowed), so we get the following result. The probability there is a collision between x and y is equal to $k * (p-1)/(p^2-1) = \frac{k}{p+1}$.

Coming back to the bin and balls setting, we have $Bin(n, k/(p+1))$ balls and k bins, we need to show the probability that all bins are covered. We can approximate the binomial distribution using the poission distribution : $Bin(n, k/(p+1)) \approx Bin(n, k^2/m) \approx Po(nk^2/m)$. Since every ball has the same priobability of landing in any bin, the distribution of balls in each bin is asymptotically equal to $Po(nk/m)$.

$$Pr(\text{false positive}) = Pr(Po(nk/m) > 0)^k = (1 - e^{-kn/m})^k$$

That is the asymptotic false positive probability we have found for the standard bloom filter.

7 Application of Bloom Filters

1. Dictionary:

Bloom filters were first introduced by Bloom to keep a dictionary of words that require frequent lookup. Bloom filters were also used in early UNIX spell-checkers since a mispelled word is tolerable.

2. Databases:

   Bloom filters also found very early uses in databases especially in distributed databases. One use is to speed up semi-join operations.

3. Network Application:

   - Collaborating in overlay and peer-to-peer networks: Bloom filters can be used for summarizing content to aid collaborations in overlay and peer-to-peer networks.
   - Resource routing: Bloom filters allow probabilistic algorithms for locating resources.
   - Packet routing: Bloom filters provide a means to speed up or simplify packet routing protocols.
   - Measurement: Bloom filters provide a useful tool for measurement infrastructures used to create data summaries in routers or other network devices.

8 Weakness in Bloom filter

Bloom filters have found many applications, in theory and practice, in particular in distributed systems. In spite of its strength, the data structure has several weaknesses.

1. Dependence on k. The lookup time grows as the false positive rate decreases. For example, to get a false positive rate below 1%, 7 memory accesses are needed. If S is large this will almost surely mean 7 cache misses, since the addresses are random.

2. Suboptimal space usage. The space usage is a factor $\frac{1}{ln2} = 1.44$ from the information theoretically best possible.

3. Lack of hash functions. There is no known way of choosing the hash functions such that the scheme can be shown to work. Choosing independent hash functions for large set S sometimes can be difficult.

4. No deletions. Deletion operations are not supported. It is not easy to modify the set after it is constructed.

9 Solution to weakness

1. Reduce space usage: We already know that the space usage of traditional Bloom filters is about 1.44 times than optimal structure. Therefore, there are methods to represent sets that use fewer bits than Bloom filters while maintaining the same rate of false positives, including the compressed Bloom filters and techniques based on perfect hashing.

   Suppose instead that we optimize the false positive rate of the Bloom filter under the constraint that the number of bits to be sent after compression is z, but the size m of the array in its uncompressed form can be larger. It turns out that using a larger, but sparser, Bloom filter can yield the same false positive rate with a smaller number of transmitted bits.

2. Reduce hashing functions: In "Less hashing, same performance: building a better Bloom filter", they are able to provide theoretical guarantees while using only pairwise independent hash functions; this is a signicant advance, since pairwise independent hash functions are generally easy to implement and quite efficient in practice.

3. Support key for each elements in S and reduce space usage: Chazelle et al. (2004) designed a generalization of Bloom filters that could associate a value with each element that had been inserted, implementing an associative array. Like Bloom filters, these structures achieve a small space overhead by accepting a small probability of false positives. In the case of "Bloomier filters", a false positive is defined as returning a result when the key is not in the map. The map will never return the wrong value for a key that is in the map.

4. Counting Bloom filters: Counting filters provide a way to implement a delete operation on a Bloom filter without recreating the filter afresh. In a counting filter the array positions (buckets) are extended from being a single bit, to being an n-bit counter. In fact, regular Bloom filters can be considered as counting filters with a bucket size of one bit. The insert operation is extended to increment the value of the buckets and the lookup operation checks that each of the required buckets is non-zero. The delete operation, obviously, then consists of decrementing the value of each of the respective buckets.

10 References

- Network Applications of Bloom Filters: A Survey, Andrei Broder and Michael Mitzenmacher:

  Sections 1-4, 7-9

- Less hashing, same performance: building a better Bloom filter, A. Kirsch and M. Mitzenmacher

  Sections 5-6