Notes 10 for CS 170

1 Bloom Filters

If the key distribution is not known, or too complicated to yield to analysis, then the use of a particular hash function may have adverse effects: it may magnify correlations among keys and fill the hash table non-uniformly. Last lecture we saw one technique that deals with this phenomenon but still allows us to use simple hash functions: in *universal hashing*, one of several hash functions is chosen at random. Today, we see a different technique: in *Bloom filters*, several hash functions are applied to each key. Again, this allows us to use simple hash functions while at the same time minimizing the effects of any particular hash function.

The main purpose of Bloom filters is to build a space-efficient data structure for set membership. Indeed, to maximize space efficiency, correctness is sacrificed: if a given key is not in the set, then a Bloom filter may give the wrong answer (this is called a *false positive*), but the probability of such a wrong answer can be made small.

A typical application of Bloom filters is web caching. An ISP may keep several levels of carefully located caches to speed up the loading of commonly viewed web pages, in particular for large data objects, such as images and videos. If a client requests a particular URL, then the service needs to determine quickly if the requested page is in one of its caches. False positives, while undesirable, are acceptable: if it turns out that a page thought to be in a cache is not there, it will be loaded from its native URL, and the "penalty" is not much worse than not having the cache in the first place.

So here is the formal set-up: we want to represent n-element sets $S = \{s_1, \ldots, s_n\}$ from a very large universe U, with $|U| = u >> n$. (Think of U as the set of URLs, n as the cache size, and S as the URLs of those web pages that are currently in the cache.) We want to support insertions and membership queries ("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. (Bloom filters can also be made to support deletions, but we won't worry about those.)

A *Bloom filter* is a bit vector B of m bits, with k independent hash functions $h_1, \ldots, h_k$ that map each key in U to the set $R_m = \{0, 1, \ldots, m-1\}$. 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 the hash functions are independent, it follows that the vector $(h_1(x), \ldots, h_k(x))$ 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 $h_1(x), \ldots, h_k(x)$ and set $B[h_1(x)] = B[h_2(x)] = \cdots = B[h_k(x)] = 1$.

- *Query if* $x \in S$. Compute $h_1(x), \ldots, h_k(x)$. If $B[h_1(x)] = B[h_2(x)] = \cdots = B[h_k(x)] = 1$ then answer Yes, else answer No.

The running times of both operations depend only on the number k of hash functions, and we will later see how to choose a suitable value for k in order to minimize the probability of false positives. The space requirement of the data structure is m bits, and we will later see that a reasonable value for m is, say, $8n$. Note that any non-randomized data structure that represents n-element subsets of U must use $\Omega(n \cdot \log u)$ bits (why?).

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.

2 Example

This is an unrealistically small example; its only purpose is to illustrate the possibility of false positives. Choose $m = 5$ (number of bits) and $k = 2$ (number of hash functions):

$$h_1(x) = x \bmod 5$$
$$h_2(x) = (2x + 3) \bmod 5$$

We first initialize the Bloom filter $B[1..5]$, and then insert 9 and 11:

	$h_1(x)$	$h_2(x)$	B				
Initialize:			0	0	0	0	0
Insert 9:	4	1	0	1	0	0	1
Insert 11:	1	0	1	1	0	0	1

Now let us attempt some membership queries:

	$h_1(x)$	$h_2(x)$	Answer
Query 15:	0	3	No, not in B (correct answer)
Query 16:	1	0	Yes, in B (wrong answer: false positive)

Note that 16 was never inserted into the filter.

3 Analysis

We compute the probability of a false positive. The probability that one hash fails to set a given a bit is $1 - \frac{1}{m}$. Hence, after all n elements of S have been inserted into the Bloom filter, the probability that a specific bit is still 0 is

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}.$$

(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

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k = (1 - p)^k$$

for $p = e^{-kn/m}$. 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* $f = (1-p)^k$).

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. We can find the minimum by taking the derivative of f. To simplify the math, we minimize the logarithm of f with respect to k (why is minimizing the logarithm of a function equivalent to minimizing the function itself?). Let

$$g \;=\; \ln(f) \;=\; k \cdot \ln(1-p) \;=\; k \cdot \ln\left(1 - e^{-\frac{kn}{m}}\right).$$

Then,

$$\frac{dg}{dk} \;=\; \ln\left(1 - e^{-\frac{kn}{m}}\right) + \frac{kn}{m} \cdot \frac{e^{-\frac{kn}{m}}}{1 - e^{-\frac{kn}{m}}}.$$

We find the optimal k, or right number of hash functions to use, when the derivative is 0. This occurs when $k = (\ln 2) \cdot \frac{m}{n}$. This can be shown to be a global minimum. For the optimal value of k, the false positive rate is

$$\left(\frac{1}{2}\right)^k \;=\; (0.6185)^{\frac{m}{n}}.$$

Of course as m grows in proportion to n, the false positive rate decreases. Already $m = 8n$ reduces the chance of error to roughly 2%, and $m = 10n$ to less than 1%. More precisely, for $m = 8n$, if $k = 3$, then $f = 0.0306$; if $k = 4$, then $f = 0.0240$; if $k = 5$, then $f = 0.0217$; if $k = 6$, then $f = 0.0216$; if $k = 7$, then $f = 0.0229$. Note that the minimum is achieved at $k = 6$, but certainly $k = 5$ and perhaps $k = 4$ may be preferable from a running-time point of view. If $m = 10n$, then the optimum is $k = 7$ with $f = 0.0082$.

4 References

1. B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM, 13:422-426, 1970.

2. J. Byers, M. Fayed. Lecture Notes: Algorithmic Aspects of Computer Networks. http://www.cs.bu.edu/fac/byers/courses/591/S02/SCRIBE/Lecture1.pdf.

3. L. Fan, P. Cao, J. Almeida, and A.Z. Broder. Summary Cache: A Scalable Wide-area Cache Sharing Protocol, Proceedings of ACM SIGCOMM, 1998.

4. M. Mitzenmacher. Compressed Bloom Filters. Proceedings of ACM PODC 2001.