Using The Power of Two Choices to Improve Bloom Filters

Steve Lumetta* Michael Mitzenmacher†

Abstract

We consider the combination of two ideas from the hashing literature, the power of two choices and Bloom filters. Specifically, we show via simulations that in comparison with a standard Bloom filter, using the power of two choices can yield modest reductions in the false positive probability using the same amount of space and more hashing.

1 Introduction

The Bloom filter [2] is a simple space-efficient randomized data structure for representing a set in order to support membership queries that sometimes allows false positives. The Bloom filter has proven useful, for example, in the context of routers, where off-chip memory accesses are dramatically slower than on-chip memory accesses, and sets (e.g., lists of source-destination pairs, attack signatures, etc.) must be represented space-efficiently. Further applications and related variations are discussed extensively in the survey paper [3]. While alternatives data structures have recently appeared [10], the Bloom filter's simplicity, straightforward mapping to hardware, and excellent performance ensure that it will continue to be of great use in many applications. We briefly review Bloom filters in Section 2. For now, it suffices to know that Bloom filters hash each element of the set to a vector of indices using a group of hash functions.

In this paper, we combine the Bloom filter with a seemingly unrelated but powerful technique often applied to improve load balancing, the power of two choices [1, 8, 9, 11]. With this technique, an item is stored at the less loaded of two (or more) random alternatives. We briefly review the power of two choices in Section 2.

The basic idea for combining the two techniques is to use two independent groups of hash functions for inserting elements into the Bloom filter. When checking for the membership of an element, both groups of hash functions are used, and a positive response is returned if either group indicates that the element is present. By allowing a choice of which hash function group is used to record the presence of an element in the set, we can reduce the false positive probability. Specifically, we show via simulations and numerical analysis that in comparison with a standard Bloom filter, using the power of two choices can yield modest reductions in the false positive probability using the same amount of space and more hashing. As Bloom filters are often used in situations where a fixed amount of space is the primary constraint, these gains may prove useful in practice.

The standard analysis for deriving the optimal configuration of a Bloom filter assumes the use of a single group of hash functions to check for set membership. The power of two choices allows us to reduce the false positive probability while maintaining the form and the character of the original Bloom filter.

*Supported in part by NSF grant ACI-9984492. lumetta@uiuc.edu

†Division of Engineering and Applied Sciences, Harvard University. michaelm@eecs.harvard.edu. Supported in part by NSF grant CCR-0121154 and a grant from Cisco.

2 Background

2.1 Bloom filter review

We first review the concepts behind Bloom filters; for mathematical details, see the survey [3]. A Bloom filter represents a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements from a universe U using an array of m bits, which we denote by $B[1], \ldots, B[m]$, initially all set to 0. The filter uses a group H of k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, m\}$, where it assumed that these hash functions independently map each element in the universe to a random number uniformly over the range. (While the randomness of the hash functions is clearly an optimistic assumption, it is standard and convenient for Bloom filter analyses.) For each element $x \in S$, the bits $B[h_i(x)]$ are set to 1 for $1 \le i \le k$. (A bit can be set to 1 multiple times.) To answer a query of the form "Is $y \in S$?", we check whether all $h_i(y)$ are set to 1. If not, y is not a member of S, by the construction. If all $h_i(y)$ are set to 1, it is assumed that y is in S, and hence a Bloom filter may yield a *false positive*.

The probability of a false positive for an element not in the set, or the *false positive probability*, can be easily derived. If p is the fraction of ones in the filter, it is simply p^k. A standard combinatorial argument gives that p is concentrated around $\left(1 - \mathrm{e}^{-kn/m}\right)$, and the expression $\left(1 - \mathrm{e}^{-kn/m}\right)^k$ is minimized when $k = \ln 2 \cdot (m/n)$, giving a false positive probability f of

$$f = \left(1 - \mathrm{e}^{-kn/m}\right)^k = (1/2)^k \approx (0.6185)^{m/n}. \tag{1}$$

In practice, of course k must be an integer. Also, in practical situations where Bloom filters are used, m/n (the number of bits per set element) and k should be thought of as constants. For example, when $m/n = 8$ and $k = 5$ the false positive probability is just over 0.02.

We emphasize an important intuition that guides some of our results. Note that as the number of hash functions k increases, there are more chances to find a zero when examining the Bloom filter, decreasing the chance of a false positive. On the other hand, larger values of k also produce more ones in the Bloom filter, increasing the chance of a false positive. The end result of this tradeoff yields the above optimization; changing this tradeoff, however, changes the optimal number of hash functions to use.

While the above analysis gives the optimal configuration for a standard Bloom filter, it is known that Bloom filters are not information-theoretically optimal [3, 10]. That is, it is possible to devise a data structure using less space that obtains the same false positive probability. It is therefore not surprising that we can improve on the Bloom filter construction; what is surprising is that we can improve on it while keeping essentially the same fundamental approach as used for a standard Bloom filter.

An important variation of the Bloom filter is the *counting Bloom filter*, where each entry in the Bloom filter is not a single bit but a small counter that tracks the number of elements that have hashed to that location [5]. A standard Bloom filter can be derived from a counting Bloom filter by setting all non-zero counts to 1. Counting Bloom filters allow deletions; when an element is deleted, the corresponding counters are decremented. Counters must be chosen large enough to avoid overflow; for most applications, four bits suffice [5].

2.2 Power of two choices review

The seminal example of the power of two choices stems from the results of Azar, Broder, Karlin and Upfal [1]. It is well known that when n balls are sequentially placed uniformly at random into n bins, the fullest bin has with high probability $(1 + o(1)) \ln n / \ln \ln n$ balls in it. The result of [1] is that if for each ball we instead choose two bins at random and sequentially place each ball into the one which is less full at the time of placement, then with high probability the fullest bin contains only $\ln\ln n/\ln 2 + O(1)$ balls. This idea of allowing choice to improve load balance has spawned a large literature; see, for example, [9].

A natural extension of this idea in the setting of Bloom filters is to allow each set element a choice of how to be recorded in the filter, by giving two (or more) groups of hash functions that can record an element. Here the goal is not to balance the load, but rather to minimize the number of bits set to one in the filter. The idea, however, follows the theme of providing better performance by introducing choices. We note that the conjunction of the power of choice and Bloom filters has also appeared previously in [6]. There the authors, in the context of a hash table construction, have choices in how to partition items into subtables of the hash table. Each subtable has an associated Bloom filter; the authors use the effect of the item placement on the false positive probabilities of the Bloom filters to guide their choice of subtable. Our work can be seen as an extrapolation of these ideas to the problem of creating a single Bloom filter for a set.

3 Using two choices with Bloom filters

We consider the following variation on the Bloom filter. We again have a set S of n items, and m bits available for a Bloom filter. Instead of one group of hash functions, we now have two groups: group H consists of $h_1, h_2, \ldots, h_k$ and and group G consists of $g_1, g_2, \ldots, g_k$. Again, we treat all hash functions as being perfectly independent random hash functions.

Each element in the set S should have all bits from the hashes from either group G or group H set to one. That is, if we let $B[h_i(x)]$ be the Bloom filter bit given by hashing x with h_i, we have the following *constraint*: we require that for every element $x \in S$, either $B[h_i(x)] = 1$ for all $i \in [1,k]$, or $B[g_i(x)] = 1$ for all $i \in [1,k]$. (The or is not exclusive; both events may occur.) We say that for $x \in S$ either G must cover x or H must cover x, where cover has the natural meaning. A false positive occurs for an element $y \notin S$ if either $B[h_i(y)] = 1$ for all $i \in [1,k]$ or $B[g_i(y)] = 1$ for all $i \in [1,k]$ (or both); that is, either G or H covers y even though $y \notin S$. We address the question of how to choose which bits are actually set to 1 given the set S shortly. Although this idea extends naturally to more than two groups of hash functions; we focus on the case of two groups for ease of exposition.

Using two groups of hash functions would seem to increase the chances of a false positive, in that there are now two ways for a false positive to occur. Specifically, if p is the fraction of ones in the filter, the probability of a false positive is now $1 - (1 - p^k)^2 \approx 2p^k$. On the other hand, we now have additional choice in which bits in the filter are set to one; if this choice sufficiently reduces the number of bits set to one in the filter, overall there may be a reduction in the false positive probability. This is exactly the tradeoff we explore. The cost of this scheme is that additional hashing and lookups into the Bloom filter are now required; for any specific application, this cost must be considered against the improvement in the false positive probability. As previously stated, in many cases fixed space is the primary constraint when Bloom filters are used, as one is trying to fit the Bloom filter into a cache or fast memory, and we assume henceforth that additional hashing is not a concern.

We consider two main settings. In the *online* setting, elements of S are presented one at a time, and after each element x_i appears, the Bloom filter must be updated so that either G or H covers x in the filter; moreover, a bit once set to the value one keeps the value one subsequently. This latter restriction is meant to encode the idea that we are not storing the elements of S, so we cannot naturally change which group is being used to fulfill the constraint for a previously seen element.

In the *offline* setting, all the elements of S are initially given, and we seek to minimize the number of ones in the filter while covering all elements of S. Obviously, this provides us with significantly more power in setting the bits of the Bloom filter. Ideally, we might find the optimal offline solution, which would minimize the number of ones in the filter, but this appears to be a hard problem, even in this setting, as we discuss below.

In both the online and offline settings, we find that reasonable reductions in the false positive probability are possible, using simple and efficient greedy algorithms to set up the filter. We emphasize that the additional work for our modified Bloom filter is essentially all in the construction phase; lookups are done in the same fashion, except that one must or the results from all groups of hash functions.

3.1 Online solutions

In this section, we analyze the optimal online solution, using both simulations and numerical analysis.

In the online setting, since the false positive probability depends only on the number of ones in the filter, the natural greedy approach is to determine when considering an element x how many additional bits would have to be set to one to cover it by G and by H, and then to modify the smallest number of bits from these two choices. It is intuitively clear and only slightly harder to prove that this greedy process is stochastically optimal, in that it stochastically dominates any other strategy. That is, if X is the number of bits set to one by the greedy strategy, and Y is the number of bits set to one by some other strategy (that is independent of future choices), then

$$\mathbf{Pr}(X \geq w) \leq \mathbf{Pr}(Y \geq w)$$

for any number w.

We have not found a simple closed form for the expected number $\mathbf{E}[X]$ of bits set to one (or some close approximation of this number) for the greedy algorithm. This remains an interesting open question. We note that using standard martingale techniques one can show that X is concentrated around $\mathbf{E}[X]$ with high probability.

One can approximate the distribution of the number of ones in the filter iteratively. Let X_t be the number of ones in the filter after t elements have been placed, and let $B_1(k,p)$ and $B_2(k,p)$ represent independent binomial random variables corresponding to k trials each with success probability p (and hence mean kp). A good approximation for the number of ones in the filter can be given by the equation

$$X_{t+1} = X_t + \min(B_1(k, 1 - X_t/m), B_2(k, 1 - X_t/m)). \qquad (2)$$

The intuition for this equation is that the number of ones needed so that G covers a new element has distribution B_1 and similarly for H. This expression is only an approximation, however, since it is possible that two hash values $h_i(x)$ and $h_j(x)$ will be the same; hence the number of ones needed to cover an element is not exactly a binomial, although this is correct up to $O(1/m)$ terms. (Such smaller order terms can be considered with more work, but have negligible effect, as we explain below.) It is not immediately clear how to derive a concise closed form given this equation, but one can use it to iteratively calculate the distribution of the number of ones after t items appear.

More insight can be gained into the expected performance by letting $z = X_t/m$, and re-scaling time to run from 0 to 1 instead of 0 to n, yielding a differential equation approximating the difference equation associated with the distribution X_t:

$$\frac{dz}{dt} = \frac{\mathbf{E}[\min(B_1(k, 1-z), B_2(k, 1-z)]}{m/n}. \qquad (3)$$

n	m	c	k	Observed false positive prob.
10000	80000	1	6	2.159e-02
10000	80000	2	7	2.323e-02
10000	80000	3	7	2.389e-02
10000	160000	1	11	4.588e-04
10000	160000	2	13	3.935e-04
10000	160000	3	13	3.607e-04
10000	320000	1	22	2.106e-07
10000	320000	2	24	1.285e-07
10000	320000	3	25	9.980e-08

Table 1: Average results for the greedy online algorithm from 1000 trials per experiment. Recall that n is the number of elements, m is the number of bits, c is the number of choices, and k is the number of hash functions. The value of k with the best observed false positive probability is used for each setting of n, m, and c. Note the case $c = 1$ corresponds to the standard Bloom filter.

We note that using the theory of martingales one can show that the evolution of X_t closely follows that of the differential equation, and Chernoff-like bounds can be given; see [7, 12] for details. Equation (3) does not appear to have a simple closed form solution, although it can easily be numerically evaluated and used in place of or in conjunction with simulations. Notice that equation (3) highlights the role played by the fraction m/n, the number of bits per set element, as the result of this differential equation depends only on m/n and not on m and n individually. Hence, up to asymptotically vanishing terms, the false positive probability is a function of m/n and k, as is equation (1) for the standard Bloom filter analysis. Equation (3) generalizes in the natural way when the number of choices is greater than two. We compare the results from this equation to our simulations below. Also, note that the form of equation (3) for one group of hash functions, namely

$$\frac{dz}{dt} = \frac{\mathbf{E}[(B_1(k, 1-z)]}{m/n} = \frac{k(1-z)}{m/n},$$

can be solved exactly to yield equation (1).

Our experimental results for the online setting are given in Table 1. We have considered cases with 8, 16, and 32 bits per set element, which are standard configurations. The case of $c = 1$ choice, corresponding to a standard Bloom filter, is compared with $c = 2$ and $c = 3$ choices. The results presented are for the best value of k, the number of hash functions, over 1000 trials for each setting of parameters. We model our random hash functions by assigning each hash of an element a number generated using a standard 48-bit pseudorandom number generator.

For a small number of bits per set element, ($m/n = 8$) we see no improvement. When $m/n = 16$, we see small improvement, and for $m/n = 32$, the false positive rate is reduced by a factor slightly less than two for $c = 2$ choices, and by a factor slightly greater than two for $c = 3$ choices. In intuitive terms, we require many bits per set element before the greedy algorithm reduces the number of ones in the filter sufficiently to make up for the two opportunities to obtain a false positive. Also, note that because fewer bits are set to one in the Bloom filter using this approach, the optimal results are obtained when using more hash functions than in the standard Bloom filter.

We point out that the use of 10000 elements does not affect our results; the numbers are nearly identical for 100000 elements, as the key parameter is the ratio m/n. There is some variance; over 1000 trials with $m/n = 32$, $c = 2$, and $k = 24$, the observed false positive probability (calculated

n	m	c	k	Observed fraction of ones	Prediction of (3)
10000	80000	2	7	0.5926	0.5926
10000	80000	3	7	0.5020	0.5019
10000	160000	2	13	0.5187	0.5187
10000	160000	3	13	0.4494	0.4994
10000	320000	2	24	0.5016	0.5016
10000	320000	3	25	0.5022	0.5022

Table 2: Comparing the average results for the greedy online algorithm from 1000 trials per experiment with a numerical evaluation from the differential equations (to four decimal places).

based on the number of ones in the filter) varied between 1.206e-07 and 1.378e-07; this is similar to the variance for $m/n = 32$ for the standard Bloom filter, where the false positive probability ranged from 1.955e-07 to 2.270e-07. Finally, our results are quite robust to small changes in the parameters. Changing the ratio m/n slightly gives only small changes in the observed false positive probability, and changing the number of hash functions to another number near the optimal also leads to very small changes. For example, while the best choice of 13 hash functions when $m/n = 16$ gave an average false positive probability of 3.935e-04 over 1000 trials, using 12 hash functions gave an average of 3.954e-04, and using 14 gave an average of 4.035e-04.

It is also worth noting that the differential equation (3) is very accurate, as we show in Table 2. For the optimal configurations presented in Table 1, the predicted fraction of ones in the Bloom filter from equation (3) matches the average from our experiments to essentially four decimal places (with one case off in the last digit due to rounding). We conclude that equation (3) can be used to accurately predict performance.

3.2 Offline solutions

In this section, we consider a greedy algorithm that extends our online solution to the offline case, and examine its performance via simulation.

In the offline setting, one is given the entire set in advance, and the goal is to design the Bloom filter with the smallest possible false positive probability (equivalently, the smallest number of ones given m, n, k, and the groups of hash functions) while ensuring that all elements are covered.

In its general form, the problem of finding the smallest possible covering is at least as hard as Minimum Vertex Cover. To see this, consider a graph G. Associate the vertices of the graph with filter bits and the edges with set elements. Consider the problem of covering each set element (or edge) with at least one of two filter bits (the adjacent vertices) using the minimum number of filter bits. This shows that Minimum Vertex Cover is equivalent to our covering problem for the Bloom filter in the case where $k = 1$. In our setting, because groups correspond to *random* hash values, there might still be an efficient algorithm that works with high probability. At this point, however, we know of no efficient algorithm.

Even if we have no algorithm for the optimal solution, for comparison purposes it could also be useful to have a probabilistic argument giving insight into the distribution of the number of ones required. Denote the minimum possible number of ones by the random variable X. As with the online setting, while we do not currently have an argument giving the expectation of X, we can say that X is concentrated around its expectation, using standard martingale arguments.

Although we do not have an optimal algorithm, there are several natural heuristic algorithms one

n	m	c	k	Rounds	Observed false positive prob.
10000	80000	1	6	n/a	2.159e-02
10000	80000	2	7	10	1.505e-02
10000	80000	2	8	15	1.485e-02
10000	80000	3	8	30	1.237-02
10000	80000	3	8	40	1.222-02
10000	160000	1	11	n/a	4.588e-04
10000	160000	2	14	10	2.259-04
10000	160000	2	14	15	2.223-04
10000	160000	3	15	30	1.604-04
10000	160000	3	15	40	1.582-04
10000	320000	1	22	n/a	2.106e-07
10000	320000	2	26	10	6.260e-08
10000	320000	2	26	15	6.143e-08
10000	320000	3	27	30	3.579e-08
10000	320000	3	27	40	3.525e-08

Table 3: Average results for the greedy offline algorithm from 1000 trials per experiment. Again the case $c = 1$ corresponds to the standard Bloom filter. Rounds refers to the number of times each element is inserted into the filter.

can apply to this problem: hill-climbing, tabu search, simulated annealing, etc. For our experiments, we chose an extremely simple greedy algorithm; while more sophisticated approaches may perform better, for many applications a simple and quick approach is likely to be desirable. Our algorithm repeatedly walks over the elements of the set (in some fixed order), and re-optimizes the choice of group for each element in turn (breaking ties randomly).[1] This can be done, for example, by using a counting Bloom filter to enable deletion, and then simply deleting (after the first round of insertions) and re-inserting each element. Although we present this approach as an offline algorithm, it could also be used as a background task to improve performance in the online setting if elements are stored as they arrive.

Our experimental results for the offline setting are given in Table 3. We have again considered cases with 8, 16, and 32 bits per set element. In this table we also consider the number of rounds used in the offline greedy algorithm, where in each round each element of the set is (re-)inserted one time. The results presented are for the best value of k, the number of hash functions, over 1000 trials for each setting of parameters.

Because our greedy algorithm breaks ties randomly, it is not clear when it enters a local minimum from which no further improvement will occur. We found that with 10000 items and two choices, 10 rounds appeared sufficient to get us very close to the minimum; 5 further rounds led to further decreases in the average false positive probability of around 1-2%. Similarly, about 30 rounds appear sufficient when $c = 3$. Similar results were obtained with experiments with 100000 items.

As one would expect, the gains are significantly greater than in the online case. Here, even when $m/n = 8$, the false positive probability is reduced from over 2% to near 1.5% using two choices, and below 1.25% using three choices. Similarly, two choices leads reduces the false positive probability

[1] We note that breaking ties randomly improves performance non-trivially over breaking ties by, for example, the order of the groups.

by roughly a factor of two for 16 bits, and over a factor of three for thirty-two bits. Going to three choices leads to further improvements; for 32 bits, the gain is roughly a factor of six.

Again, our results appear to scale: the numbers are similar for 100000 elements, with the same number of rounds. There is some variance, but the greedy algorithm also appears to generally concentrate around its expectation over a sufficient number of rounds. For example, over 1000 trials with $m/n = 32$, $c = 2$, $k = 26$, and 15 rounds, the false positive probability varied between 5.706e-08 and 6.486e-08. Also, as before, our results are quite robust to small changes in the parameters m/n and k, although here the number of rounds plays a key role in the overall performance.

4 Conclusion

We have demonstrated that when constructing a Bloom filter, the power of two choices can yield improvements in the false positive probability using the same amount of space, at the expense of more hashing. The advantage of this approach is that it keeps all of the appealing properties of a Bloom filter, such as its simplicity and ease of implementation, while yielding better results.

This work suggests several additional open questions. First, an analysis of the optimal offline schemes, as well as an analysis of some easily implementable offline scheme, would be desirable, as they would allow design decisions without resorting to simulations. Second, a simple closed form (or very good approximation) for equation (3) would be useful. Third, it would be useful to find techniques that reduce the amount of hashing required when choices are used, or to extend these results to hold under simple implementable hash functions instead of the random hash function model. Fourth, it is worth considering whether this approach could somehow also be useful for Bloom filter variations, such as Count-Min sketches [4]. Finally, this work opens the door to the possibility that there are further ways to improve the performance of Bloom filters through simple modifications that preserve their desirable properties. Such improvements would likely be useful in many implementations.

References

[1] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal of Computing* 29:1, pp. 180-200, 1999.

[2] B. Bloom. Space/time tradeoffs in in hash coding with allowable errors. *Communications of the ACM*, 13(7):422-426, 1970.

[3] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485-509, 2004.

[4] G. Cormode and S. Muthukrishnan. Improved Data Stream Summaries: The Count-Min Sketch and its Applications. DIMACS Technical Report 2003-20, 2003.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281-293, 2000.

[6] S. Kumar and P. Crowley. Segmented Hash: An Efficient Hash Table Implementation for High-Performance Networking Subsystems. To appear in *Proceedings of the 2005 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. October 2005.

[7] T. G. Kurtz. Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes. *Journal of Applied Probability*, Vol. 7, pp. 49-58, 1970.

[8] M. Mitzenmacher. The power of two choices in randomized load balancing. Ph. D. thesis, U.C. Berkeley, 1996.

[9] M. Mitzenmacher, A. Richa, and R. Sitaraman. *The Power of Two Choices: A Survey of Techniques and Results*. Kluwer Academic Publishers, Norwell, MA, 2001, pp. 255–312. Edited by P. Pardalos, S. Rajasekaran, J. Reif, and J. Rolim.

[10] A. Pagh, R. Pagh, and S. Srinivas Rao. An Optimal Bloom Filter Replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 823-829, 2005.

[11] B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the* 40^{th} *IEEE-FOCS*, pp. 131-140, 1999.

[12] N Wormald. Differential equations for random processes and random graphs. *Ann. Appl. Probab.*, 5:4, pp. 1217-1235, 1995.