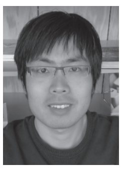
Cuckoo Filter: Better Than Bloom

BIN FAN, DAVID G. ANDERSEN, AND MICHAEL KAMINSKY



Bin Fan is a Ph.D. student in the Computer Science Department at Carnegie Mellon University. His research interests include networking systems, storage systems, and distributed systems. He is in the Parallel Data Lab (PDL) at CMU and also works closely with Intel Labs. binfan@cs.cmu.edu

Michael Kaminsky is a Senior Research Scientist at Intel Labs and an adjunct faculty member in the Computer Science Department at Carnegie Mellon University. He is part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), based in Pittsburgh, PA. His research interests include distributed systems, operating systems, and networking. michael.e.kaminsky@intel.com

David G. Andersen is an Associate Professor of Computer Science at Carnegie Mellon University. He completed his S.M. and Ph.D. degrees at MIT, and holds BS degrees in Biology and Computer Science from the University of Utah. In 1995, he co-founded an Internet Service Provider in Salt Lake City, Utah. dga@cs.cmu.edu

High-speed approximate set-membership tests are critical for many applications, and Bloom filters are used widely in practice, but do not support deletion. In this article, we describe a new data structure called the *cuckoo filter* that can replace Bloom filters for many approximate set-membership test applications. Cuckoo filters allow adding and removing items dynamically while achieving higher lookup performance, and also use less space than conventional, non-deletion-supporting Bloom filters for applications that require low false positive rates (ϵ< 3%).

Set-membership tests determine whether a given item is in a set or not. By allowing a small but tunable false positive probability, set-membership tests can be implemented by Bloom filters [1], which cost a constant number of bits per item. Bloom filters are efficient for representing large and static sets, and thus are widely used in many applications from caches and routers to databases; however, the existing items cannot be removed from the set without rebuilding the entire filter. In this article, we present a new, practical data structure that is better for applications that require low false positive probabilities, handle a mix of "yes" and "no" answers, or that need to delete items from the set.

Several proposals have extended classic Bloom filters to add support for deletion but with significant space overhead: *counting Bloom filters* [5] are four times larger and the recent *d-left counting Bloom filters* (dl-CBFs) [3, 2], which adopt a hash table-based approach, are still about twice as large as a space-optimized Bloom filter. This article shows that supporting deletion for approximate set-membership tests does not require higher space overhead than static data structures like Bloom filters. Our proposed cuckoo filter can replace both counting and traditional Bloom filters with three major advantages: (1) it supports adding and removing items dynamically; (2) it achieves higher lookup performance; and (3) it requires less space than a space-optimized Bloom filter when the target false positive rate ϵ is less than 3%. A cuckoo filter is a compact variant of a cuckoo hash table [7] that stores fingerprints (hash values) for each item inserted. Cuckoo hash tables can have more than 90% occupancy, which translates into high space efficiency when used for set membership.

Bloom Filter Background

Standard Bloom filters allow a tunable false positive rate ϵ so that a query returns either "definitely not" (with no error) or "probably yes" (with probability ϵ of being wrong). The lower ϵ is, the more space the filter requires. An empty Bloom filter is a bit array with all bits set to "0", and associates each item with k hash functions. To add an item, it hashes this item to k positions in the bit array, and then sets all k bits to "1". Lookup is processed similarly, except it reads k corresponding bits in the array: if all the bits are set, the query returns positive; otherwise it returns negative. Bloom filters do not support deletion, thus removing even a single item requires rebuilding the entire filter.

Counting Bloom filters support delete operations by extending the bit array to a counter array. An insert then increments the value of k counters instead of simply setting k bits, and lookup checks whether each of the required counters is non-zero. The delete operation decrements the values of the k counters. In practice the counter usually consists of four or more

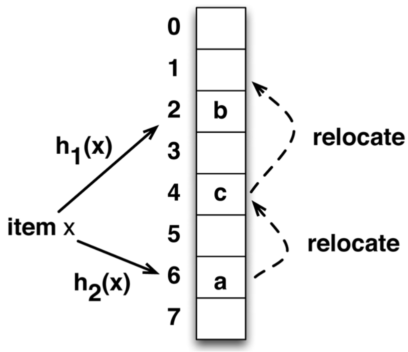


Figure 1: A cuckoo hash table with eight buckets

bits, and a counting Bloom filter therefore requires four times more space than a standard Bloom filter.

The work on d-left counting Bloom filters (dl-CBFs) [2, 3] is intellectually closest to our cuckoo filter. A dl-CBF constructs a hash table for all known items by *d-left hashing* [6], but replaces each item with a short fingerprint (i.e., a bit string derived from the item using a hash function). The dl-CBFs can reduce the space cost of counting Bloom filters, but still require twice the space of a space-optimized Bloom filter.

Cuckoo Filter

The cuckoo filter is a compact data structure for approximate set-membership queries where items can be added and removed dynamically in O(1) time. Essentially, it is a highly compact cuckoo hash table that stores fingerprints (i.e., short hash values) for each item.

Basic Cuckoo Hash Table

Cuckoo hashing is an open addressing hashing scheme to construct space-efficient hash tables [7]. A basic cuckoo hash table consists of an array of buckets where each item has two candidate buckets determined by hash functions $h_1(\cdot)$ and $h_2(\cdot)$ (see Figure 1). Looking up an item checks both buckets to see whether either contains this item. If either of its two buckets is empty, we can insert a new item into that free bucket; if neither bucket has space, it selects one of the candidate buckets (e.g., bucket 6), kicks out the existing item ("a"), and re-inserts this victim item to its own alternate location (bucket 4). Displacing the victim may also require kicking out another existing item ("c"), so this procedure may repeat until a vacant bucket is found, or until a maximum number of displacements is reached (e.g., 500 times in our implementation). If no vacant bucket is found, the hash table is considered too full to insert and an expansion process is scheduled. Though cuckoo hashing may execute a sequence of displacements, its amortized insertion time is still O(1). Cuckoo hashing ensures high space occupancy because it can refine earlier item-placement decisions when inserting new items.

Proper configuration of various cuckoo hash table parameters can ensure table occupancy more than 95%.

Dynamic Insert

When inserting new items, cuckoo hashing may relocate existing items to their alternate locations in order to make room for the new ones. Cuckoo filters, however, store only the items' fingerprints in the hash table and therefore have no way to read back and rehash the original items to find their alternate locations (as in traditional cuckoo hashing). We therefore propose *partial-key cuckoo hashing* to derive an item's alternate location using only its fingerprint. For an item x, our hashing scheme calculates the indexes of the two candidate buckets i_1 and i_2 as follows:

$$i_1 = \mathrm{HASH}(x),$$
$$i_2 = i_1 \oplus \mathrm{HASH}(x\text{'s fingerprint}).$$
Eq. (1)

The exclusive-or operation in Eq. (1) ensures an important property: i_1 can be computed using the same formula from i_2 and the fingerprint; therefore, to displace a key originally in bucket i (no matter whether i is i_1 or i_2), we can directly calculate its alternate bucket j from the current bucket index i and the fingerprint stored in this bucket by

$$j = i \oplus \mathrm{HASH}(\text{fingerprint}).$$
Eq. (2)

Hence, insertion can complete using only information in the table, and never has to retrieve the original item x.

Note that we hash the fingerprint before it is XOR-ed with the index of its current bucket, in order to help distribute the items uniformly in the table. If the alternate location is calculated by "i ⊕Fingerprint" without hashing the fingerprint, the items kicked out from nearby buckets will land close to each other in the table, assuming the size of the fingerprint is small compared to the table size. Hashing ensures that items kicked out can land in an entirely different part of the hash table.

Does Partial-Key Cuckoo Hashing Ensure High Occupancy?

The values of i_1 and i_2 calculated by Eq. (1) are uniformly distributed, individually. They are not, however, necessarily independent of each other (as required by standard cuckoo hashing). Given the value of i_1, the number of possible values of i_2 is at most 2^f where each fingerprint is f bits; when $f \le \log_2 r$ where r is the total number of buckets, the choice of i_2 is only a subset of all the r buckets of the entire hash table. For example, using one-byte fingerprints, given i_1 there are only up to 2^f=256 different possible values of i_2 across the entire table; thus i_1 and i_2 are dependent when the hash table contains more than 256 buckets. This situation is relatively common, for example, when the cuckoo

			# memory references lookup	
	Bits per item	**Load factor α**	Positive query	Negative query
Space-optimized Bloom filter	$1.44 \log_2(1/\epsilon)$	–	$\log_2(1/\epsilon)$	2
(2,4)-cuckoo filter	$(\log_2(\alpha/\epsilon)+3)/\alpha$	95.5%	2	2
(2,4)-cuckoo filter w/ semi-sort	$(\log_2(\alpha/\epsilon)+2)/\alpha$	95.5%	2	2

Table 1: Space and lookup cost of Bloom filters and two cuckoo filters

filter targets a large number of items but a moderately low false positive rate.

The table occupancy, though, can still be close to optimal in most cases (where optimal is when i_1 and i_2 are fully independent). We empirically show in the Evaluation section that this algorithm achieves close-to-optimal load when each fingerprint is sufficiently large.

Dynamic Delete

With partial-key cuckoo hashing, deletion is simple. Given an item to delete, we check both its candidate buckets; if there is a fingerprint match in either bucket, we just remove the fingerprint from that bucket. This deletion is safe even if two items stored in the same bucket happen to have the same fingerprint. For example, if item x and y have the same fingerprint, and both items can reside in bucket i_1, partial-key cuckoo hashing ensures that bucket $i_2 = i_1 \oplus \mathrm{HASH}(\mathrm{fingerprint})$ must be the other candidate bucket for both x and y. As a result, if we delete x, it does not matter if we remove the fingerprint added when inserting x or y; the membership of y will still return positive because there is one fingerprint left that must be reachable from either bucket i_1 and i_2.

Optimizing Space Efficiency

Set-Associativity: Increasing bucket capacity (i.e., each bucket may contain multiple fingerprints) can significantly improve the occupancy of a cuckoo hash table [4]; meanwhile, comparing more fingerprints on looking up each bucket also requires longer fingerprints to retain the same false positive rate (leading to larger tables). We explored different configuration settings and found that having four fingerprints per bucket achieves a sweet point in terms of the space overhead per item. In the following, we focus on the (2,4)-cuckoo filters that use two hash functions and four fingerprints per bucket.

Semi-Sorting: During lookup, the fingerprints (i.e., hashes) in a single bucket are compared against the item being tested; their relative order within this bucket does not affect query results. Based on this observation, we can compress each bucket to save one bit per item, by "semi-sorting" the fingerprints and encoding the sorted fingerprints. This compression scheme is similar to the "semi-sorting buckets" optimization used in [2]. Let us use the following example to illustrate how the compression works.

When each bucket contains four fingerprints and each fingerprint is four bits, an uncompressed bucket occupies 16 bits; however, if we sort all four four-bit fingerprints in this bucket, there are only 3,876 possible outcomes. If we precompute and store all of these 3,876 16-bit buckets in an extra table, and replace the original bucket with an index into the precomputed table, each bucket can be encoded by 12 bits rather than 16 bits, saving one bit per fingerprint (but requiring extra encoding/decoding tables).

Comparison with Bloom Filter

When is our proposed cuckoo filter better than Bloom filters? The answer depends on the goals of the applications. This section compares Bloom filters and cuckoo filters side-by-side using the metrics shown in Table 1 and several additional factors.

Space efficiency: Table 1 compares space-optimized Bloom filters and (2,4)-cuckoo filters with and without semi-sorting. Figure 2 further shows the trend of these schemes when varies from 0.001% to 10%. The information theoretical bound requires $\log_2(1/\epsilon)$ bits for each item, and an optimal Bloom filter uses 1.44 $\log_2(1/\epsilon)$ bits per item, or 44% overhead. (2,4)-cuckoo filters with semi-sorting are more space efficient than Bloom filters when < 3%.

Number of memory accesses: For Bloom filters with k hash functions, a positive query must read k bits from the bit array. For space-optimized Bloom filters that require $k=\log_2(1/\epsilon)$, when ϵ gets smaller, positive queries must probe more bits and are likely to have more cache line misses when reading each bit. For example, k equals 2 when $\epsilon = 25\%$, but the value quickly grows to 7 when $\epsilon = 1\%$, which is more commonly seen in practice. A negative query to a space optimized Bloom filter reads 2 bits on average before it returns, because half of the bits are set [8]. In contrast, any query to a cuckoo filter, positive or negative, always reads a fixed number of buckets, resulting in two cache line misses.

Static maximum capacity: The maximum number of entries a cuckoo filter can contain is limited. After reaching the maxi-

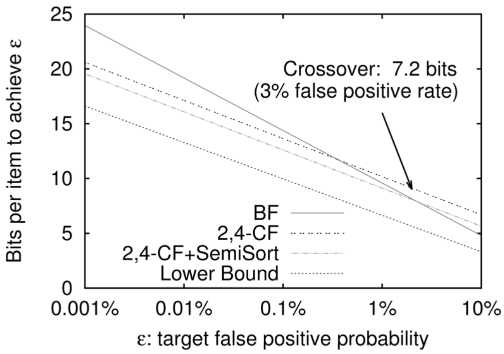


Figure 2: False positive rate vs. space cost per element. For low false positive rates (< 3%), cuckoo filters (CF) require fewer bits per element than the space-optimized Bloom filters (BF). The load factors to calculate space cost of cuckoo filters are obtained empirically.

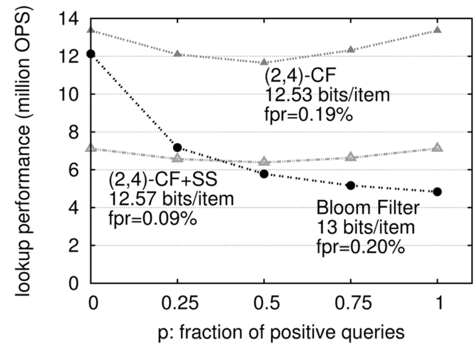


Figure 3: Lookup performance for a space-optimized Bloom filter and a (2,4)-cuckoo filter with a single thread. Each point is the average of 10 runs.

mum load factor, insertions are likely to fail and the hash table must expand in order to store more items. In contrast, one can keep inserting new items into a Bloom filter at the cost of an increasing false positive rate. To maintain the same target false positive rate, the Bloom filter must also expand.

Limited duplicate insertion: If the cuckoo filter supports deletion, it must store multiple copies of the same item. Inserting the same item kb+1 times will cause the insertion to fail. This is similar to counting Bloom filters where duplicate insertion causes counter overflow. In contrast, there is no effect from inserting identical items multiple times into Bloom filters, or a non-deletable cuckoo filter.

Evaluation

We implemented a cuckoo filter in approximately 500 lines of C++ (https://github.com/efficient/cuckoofilter). To evaluate its space efficiency and lookup performance, we ran micro-benchmarks on a machine with Intel Xeon processors (L5640@2.27 GHz, 12 MB L3 cache) and 16 GB DRAM.

Load factor: As discussed above, partial-key cuckoo hashing relies on the fingerprint to calculate each item's alternate buckets. To show that the hash table still achieves high occupancy even when the hash functions are not fully independent,

f (bits)	mean of α	(gap to optimal)	variance of α
2	17.53%,	(-78.27%)	1.39%
4	67.67%,	(-28.13%)	8.06%
6	95.39%,	(-0.41%)	0.10%
8	95.62%,	(-0.18%)	0.18%
12	95.77%,	(-0.03%)	0.11%
16	95.80%,	(0.00%)	0.11%

Table 2: Load factor achieved by different f with (2,4)-cuckoo filter. Each point is the average of 10 runs.

we built (2,4)-cuckoo filters using fingerprints of different sizes and measured the maximum load factor. We varied the fingerprint size from 2 bits to 16 bits, and each filter consists of 2^{25} (32 million) buckets. Keys are inserted to an empty filter until a single insertion relocates existing fingerprints more than 500 times (our "full" condition); then we stop and measure the mean and variance of achieved load factor α. As shown in Table 2, when the fingerprint is smaller than six bits, the table utilization is low, because the limited number of alternate buckets causes insertions to fail frequently. Once fingerprints exceed six bits, α approaches the optimal (i.e., that achieved using two fully independent hash functions).

Space efficiency: We measured the achieved false positive rates of Bloom filters and (2,4)-cuckoo filters with and without the semi-sorting optimization. When the Bloom filter uses 13 bits per item, it can achieve its lowest false positive rate of 0.20% with nine hash functions. With 12-bit fingerprints, the (2,4)-cuckoo filter uses slightly less space (12.53 bits/item), and its achieved false positive rate is 0.19%. When semi-sorting is used, a (2,4)-cuckoo filter can encode one more bit for each item and thus halve the false positive rate to 0.09%, using the same amount of space (12.57 bits/item).

Lookup Performance: After creating these filters, we also investigated the lookup performance for both positive and negative queries. We varied the fraction p of positive queries in the input workload from p=0% to 100%, shown in Figure 3. Each filter occupies about 200 MB (much larger than the L3 cache). The Bloom filter performs well when all queries are negative, because each lookup can return immediately after fetching the first "0" bit; however, its performance declines quickly when more queries are positive, because it incurs additional cache misses as it reads additional bits as part of the lookup. In contrast, a (2,4)-cuckoo filter always fetches two buckets in parallel, and thus achieves about the same, high performance for 100% positive queries and 100% negative queries. The performance drops slightly when p=50% because the CPU's branch prediction is least accurate (the probability of matching or not matching is exactly 1/2). A (2,4)-cuckoo filter with semi-sorting has a similar trend, but it is slower due to the extra encoding/decoding overhead when reading each bucket. In return for the performance penalty, the semi-sorting version reduces the false positive rate by half compared to the standard (2,4)-cuckoo filter. However, the cuckoo filter with semi-sorting still outperforms Bloom filters when more than 50% queries are positive.

References

[1] B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM,* vol. 13, no. 7 (1970), pp.422-426.

[2] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Bloom Filters via D-Left Hashing and Dynamic Bit Reassignment," in Proceedings of the Allerton Conference on Communication, Control and Computing, 2006.

[3] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," *14th Annual European Symposium on Algorithms,* 2006, pp. 684-695.

[4] U. Erlingsson, M. Manasse, and F. McSherry, "A Cool and Practical Alternative to Traditional Hash Tables, *Seventh Workshop on Distributed Data and Structures* (WDAS 2006), CA, USA, pp. 1-6.

[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,* vol. 8, no. 3 (June 2000), pp. 281-293, doi: 10.1109/90.851975.

[6] M. Mitzenmacher and B. Vocking, "The Asymptotics of Selecting the Shortest of Two, Improved," *Proceedings of the Annual Allerton Conference on Communication Control and Computing* (1999), vol. 37, pp. 326-327.

[7] R. Pagh and F. Rodler, "Cuckoo Hashing," *Journal of Algorithms,* vol. 51, no. 2 (May 2004), pp.122-144.

[8] F. Putze, P. Sanders, and S. Johannes, "Cache-, Hash- and Space-Efficient Bloom Filters," *Experimental Algorithms* (Springer Berlin / Heidelberg, 2007), pp. 108-121.