Don’t Thrash: How to Cache your Hash on Flash

Appears in the proceedings of the 3rd USENIX Workshop in Hot Topics in Storage and File Systems (HotStorage 2011)

Michael A. Bender[*§] Martin Farach-Colton[†§] Rob Johnson[*] Bradley C. Kuszmaul[‡§]
Dzejla Medjedovic[*] Pablo Montes[*] Pradeep Shetty[*] Richard P. Spillane[*] Erez Zadok[*]

Abstract

Many large storage systems use approximate-membership-query (AMQ) data structures to deal with the massive amounts of data that they process. An AMQ data structure is a dictionary that trades off space for a false positive rate on membership queries. It is designed to fit into small, fast storage, and it is used to avoid I/Os on slow storage. The Bloom filter is a well-known example of an AMQ data structure. Bloom filters, however, do not scale outside of main memory.

This paper describes the Cascade Filter ™, an AMQ data structure that scales beyond main memory, supporting over half a million insertions/deletions per second and over 500 lookups per second on a commodity flash-based SSD.

1 Introduction

Many large storage systems employ data structures that give fast answers to approximate membership queries (AMQs). The Bloom filter [2] is a well-known example of an AMQ.

An AMQ data structure supports the following dictionary operations on a set of keys: insert, lookup, and optionally delete. For a key in the set, lookup returns “present.” For a key not in the set, lookup returns “absent” with probability at least $1 - \varepsilon$, where ε is a tunable false-positive rate. There is a tradeoff between ε and the space consumption.

To understand how an AMQ data structure such as a Bloom filter is used, consider Webtable [6], a database table that associates domain names of websites with website attributes. An automated web crawler inserts new entries into the table while users independently perform queries. The system optimizes for a high insertion rate by splitting the database tables into smaller subtables.

When a user performs a search, this search is replicated on all subtables. To achieve fast lookups, the system assigns a Bloom filter to each subtable. Most subtables do not contain the queried element, meaning that the system can avoid I/Os in those subtables. Thus, searches are usually satisfied with one or zero I/Os.

Similar workloads to Webtable, which also require fast insertions and independent searches, are growing in importance [7, 11, 15]. Bloom filters are also used for deduplication [24], distributed information retrieval [20], network computing [4], stream computing [23], bioinformatics [8, 18], database querying [19], and probabilistic verification [12]. For a comprehensive review of Bloom filters, see Broder and Mitzenmacher’s survey [4].

Bloom filters work well when they fit in main memory. Bloom filters require about one byte per stored data item. Counting Bloom filters—those supporting insertions and deletions [10]—require 4 times more space [3].

What goes wrong when Bloom filters grow too big to fit in main memory? On disks with rotating platters and moving heads, Bloom filters choke. A rotational disk performs only 100–200 (random) I/Os per second, and each Bloom filter operation requires multiple I/Os. Even on flash-based solid-state drives, Bloom filters achieve only hundreds of operations per second in contrast to the order of a million per second in main memory.

One way to improve insertions into Bloom filters for flash is to employ buffering techniques [5]. The idea is to use an in-memory buffer to collect writes destined for the same flash page, executing multiple writes with one I/O. Buffering helps to some degree, achieving over a factor of two improvement over a simple Bloom filter in [5]. With larger buffers and data sets, we measured that buffering can give an 80-fold improvement.

However, buffering scales poorly as the Bloom-filter size increases compared to the in-memory buffer size, resulting in only a few buffered updates per flash page on average.

This paper demonstrates that AMQ data structures can be efficient, scalable, flexible, and cost-effective for data sets much larger than main memory. We describe a new data structure, called the *Cascade Filter* ™, designed to scale out of RAM onto flash.

In our experiments an Intel X25-M 160GB SATA II SSD using a Cascade Filter was able to perform 670,000 insertions per second and 530 lookups per second on a data set containing more than 8.59 billion elements. The Cascade Filter supports insertions at rates 40 times faster than a Bloom filter with buffering and 3,200 times faster than a traditional Bloom filter. Lookup

[*]Stony Brook University. [†]Rutgers University. [‡]MIT. [§]Tokutek.

This research was supported in part by a grant from DOE Grant DE-FG02-08ER25853, NetApp, a NetApp Faculty Award, and NSF Grants CCF-0540897, CNS-0627645, CCF-0634793, CCF-0937829, CCF-0937833, CCF-0937854, CCF-0937860, and CCF-0937822, and Politécnico Grancolombiano. Cascade Filter, Quotient Filter, and Quotient Filters are trademarks of Stony Brook University, MIT, Rutgers University, and Tokutek.

throughput is 3 times slower than that of a Bloom filter or about the cost of 6 times random reads on flash.

To put these numbers in perspective, on the Intel X25-M, we measured 5,603 random 4K block writes per second (21.8 MB/sec) and 3,218 random 4K block reads per second (12.5 MB/sec). Random bit reads/writes have comparable speeds. Sequential writes run at roughly 110MB/sec.

The Cascade Filter can be implemented cost effectively. For example, given a data center holding 1PB of 512 byte keys, our results indicate that one can construct a Cascade Filter with a less than 0.04% false positive rate using 10TB of consumer-grade flash disks. This Cascade Filter would be relatively inexpensive, costing less than $35,000, a small fraction of the data-center cost.

Our three contributions are as follows: (1) We introduce the Quotient Filter ^{TM}(QF), which supports insertions and deletions, as well as merging/resizing of two QFs. A QF is an in-memory AMQ data structure that is functionally similar to a Bloom filter, but lookups incur a single cache miss, as opposed to at least two in expectation for a Bloom filter. QFs are 20% bigger than Bloom filters, which compares favorably with the $4\times$ blowup associated with counting Bloom filters. (2) We introduce the Cascade Filter (CF), an AMQ data structure designed for flash. The CF comprises a collection of QFs organized into a data structure inspired by the Cache-Oblivious Lookahead Array (COLA) [1]. (3) We theoretically analyze and experimentally verify the performance of the CF. The CF performs insertions and deletions fast enough to keep pace with Cassandra [17], TokuDB [21], and other write-optimized indexing systems, as well as systems such as Vertica [22] and InnoDB [13], that use insertion buffers.

The remainder of this paper is organized as follows. Section 2 describes the QF and CF data structures and presents a theoretical analysis. Section 3 presents our experiments. Section 4 offers some concluding remarks.

2 Design and Implementation

This section presents the CF data structure and gives a brief theoretical analysis of its performance. The CF comprises a collection of quotient filters organized into a data structure resembling a Cache-Oblivious Lookahead Array (COLA) [1]. The COLA-like CF achieves its fast insertion performance by merging and writing QFs onto disk in an I/O-efficient manner. The section describes the QF, and then shows how to combine QFs into a CF.

The QF stores p-bit fingerprints of elements. The QF is a compact hash table similar to that described by Cleary [9]. The hash table employs *quotienting*, a technique suggested by Knuth [16, Section 6.4, exercise 13], in which the fingerprint is partitioned into the q most significant bits (the quotient) and the r least significant bits

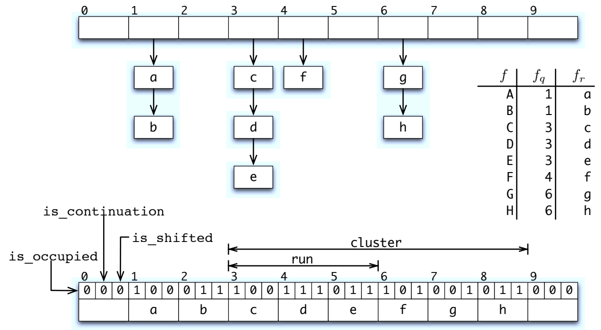

Figure 1: An example quotient filter and its representation. This filter contains values A through H. The table on the right shows, for each value, the corresponding quotient and remainder. The top of the figure shows a chained hash table storing the values A through H by storing a list of remainders in a bucket identified by the quotient of the values. The bottom of the figure shows how the remainders are stored in the QF. Each bucket contains three bits in addition to the remainder. The three bits are the `is_occupied`*,* `is_continuation`*, and* `is_shifted` *values, in that order. In this example, C, D, and E have all the same quotient, so together they form a run. Value C is stored in its canonical position, so it is the beginning of a cluster. Although F should have been stored in bucket 4, it is pushed forward by D and E to bucket 6. Values C through H together form a cluster.*

(the remainder). The remainder is stored in the bucket indexed by the quotient. Figure 1 illustrates a quotient filter.

If the quotients of two stored fingerprints are equal then we say we have a *soft collision*. The QF employs linear probing as a collision-resolution strategy, and stores the remainders in sorted order. Thus, a remainder may end up shifted forward and stored in a subsequent slot. The slot in which a fingerprint's remainder would be stored if there were no collisions is called the *canonical slot*. All of the remainders with the same quotient are stored contiguously, and are called a *run*.

A *cluster* is a maximal sequence of occupied slots whose first element is the only element of the cluster stored in its canonical slot. A cluster may contain one or more runs.

The first element of the cluster acts as an anchor that, in combination with three additional bits in each slot, allows us to recover the full fingerprint of each stored remainder in the cluster.

The three additional bits in each slot are as follows:

`is_occupied` specifies whether a slot is the canonical slot for some value stored in the filter.

`is_continuation` specifies whether a slot holds a remainder that is part of a run (but not the first).

`is_shifted` specifies whether a slot holds a remainder that is not in its canonical slot.

Whenever we insert a fingerprint we mark as occupied the slot indexed by its quotient and shift any remainders forward as necessary, updating the bits accordingly.

There is a design that uses two indicator bits instead of three, and which identifies an empty bucket by storing dummy data in reverse sorted order. However, our implementation of this scheme is more CPU intensive, and we opted for a three-bit scheme instead in our experiments.

A false positive can occur only when two elements map to the same fingerprint. For a good hash function, let the load factor of the hash table be $\alpha = n/m$, where n is the number of elements, and $m = 2^q$ is the number of slots. Then the probability of such a *hard collision* is approximately $1 - e^{-\alpha/2^r} \leq 2^{-r}$.

The space required by a QF is comparable to that of a Bloom filter, depending on parameter choices. For a QF and a Bloom filter that can hold the same number of elements and with the same false positive rate, a QF with $\alpha = 3/4$ requires 1.2 times as much space as a Bloom filter with 10 hash functions.

The QF supports several useful operations efficiently. One can merge two QFs into a single QF efficiently in a manner analogous to a merge of two sorted arrays because the fingerprints are stored in ascending order. One can also double or halve the size of a QF without rehashing the fingerprints because the fingerprints can be fully recovered from the quotients and remainders.

Since lookups, inserts, and deletes in a quotient filter all require decoding an entire cluster, we must argue that clusters are small. If we assume that the hash function h generates uniformly distributed independent outputs, then an analysis using Chernoff bounds shows that, with high probability, a quotient filter with m slots has all runs of length $O(\log m)$; most runs have length $O(1)$.

From QF to CF

Updating a QF that fits in main memory is fast. If the QF does not fit, then updates may incur random writes. Although the I/O performance is better than a traditional Bloom filter with the same false-positive rate and maximum number of insertions, we can do better by using several QFs to build a CF.

The overall structure of the CF is loosely based on a data structure called the COLA [1], and is illustrated in Figure 2. The CF comprises an in-memory QF, called QF_0. In addition, for RAM of size M, the CF comprises $\ell = \log_2(n/M) + O(1)$ in-flash QFs of exponentially increasing size, $QF_1, QF_2 \ldots QF_\ell$ stored contiguously. For simplicity, we explain here the case for insertions (deletions can be handled with tombstones at the cost of a fourth tombstone bit). In the case of insertions-only, each in-flash QF is either empty or has reached its maximum load factor. Insertions are made into QF_0. When QF_0 reaches its maximum load factor, we find QF_i the smallest empty QF, and merge $QF_0 \ldots QF_{i-1}$ into QF_i.

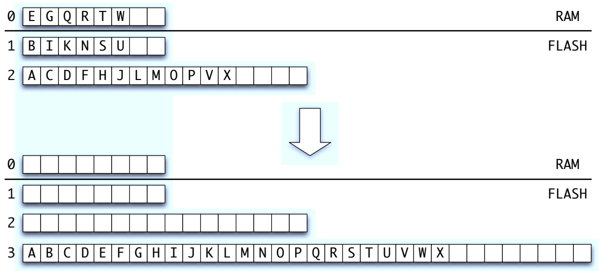
0 E G Q R T W
RAM
1 B I K N S U
FLASH
2 A C D F H J L M O P V X
0
RAM
1
FLASH
2
3 A B C D E F G H I J K L M N O P Q R S T U V W X

Figure 2: Merging QFs. Three QFs of different sizes are shown above, and they are merged into a single large QF below. The top of the figure shows a CF before a merge, with one QF stored in RAM, and two QFs stored in flash. The three QFs above have all reached their maximum load factors (which is 3/4 in this example). The bottom of the figure shows the same CF after the merge. Now the QF at level 3 is at its maximum load factor, but the QFs at levels 0, 1, and 2 are empty.

To perform a CF lookup, we examine all nonempty QFs, fetching one page from each.

The theoretical analysis of CF performance follows from the COLA: a search requires one block read per level, for a total of $O(\log(n/M))$ block reads, and an insert requires only $O((\log(n/M))/B)$ amortized block writes/erases, where B is the natural block size of the flash. Typically, $B \gg \log(n/M)$, meaning the cost of an insertion or deletion is much less than one block write per element.

Like a COLA, a CF can be deamortized to provide better worst-case bounds [1]. This deamortization removes delays caused by merging large QFs.

The false positive rate of the CF is similar to its component QFs. The CF is a multiset of integers, each of width p bits. If the largest level is configured to store $\alpha 2^{q-1}$ elements, then the entire CF can store $\alpha 2^q$ elements; by the same argument as for the component QF, the expected false positive rate is $1 - e^{-\alpha/2^r} \leq 2^{-r}$.

3 Evaluation

This section evaluates the insertion and lookup throughput of the QF and CF. We compare a QF to a traditional Bloom filter (BF) in RAM, and we compare a CF with a traditional BF and an elevator BF on flash.

We ran our experiments on a quad-core 2.4GHz Xeon E5530 with 8MB cache and 24GB RAM, running Linux (CentOS 5.4). We booted the machine with 0.994GB of RAM to test out-of-RAM performance. We used a 159.4GB Intel X-25M SSD (second generation). To ensure a cold cache and an equivalent block layout on disk, we ran each iteration of the relevant benchmark on a newly formatted file system, which we zeroed out first with `/bin/dd`. We ensured that there was no swapping. The partition size was fixed at 90GB, or 58% of the drive's capacity which is nearly optimal for the SSD [14]. The CF was configured to use 256MB of RAM. The elevator BF was configured to use 256MB

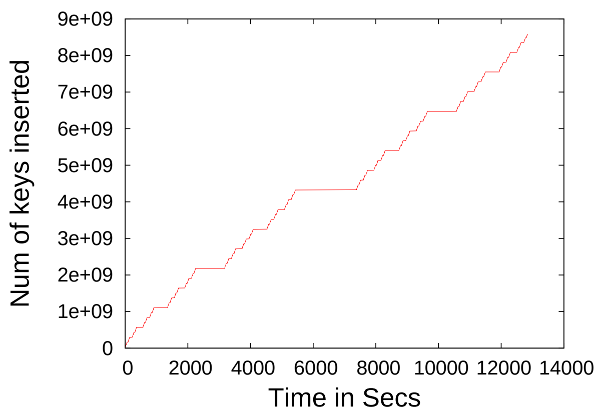

Figure 3: CF Insertion Throughput. The x axis shows elapsed time and the y axis shows the number of insertions performed up to that point. Due to periodic compactions, there are long periods of time in which no insertions take place. The sustained throughput averages 670,000 insertions per second.

worth of keys in RAM, but due to memory fragmentation this algorithm used close to 512MB. The remainder of RAM was used for file system caching. The traditional BF used all of RAM for buffer caching. All filters had the same false positive rate of 0.04%. The traditional and elevator BFs were configured to use 11 hash functions, and CF was configured with 11 r-bits in the lowest level.

QF insertion throughput. We compared the in-RAM performance of the QF and a BF with the optimal number of hash functions for the same number of elements and false-positive rate. For inserts, the cumulative throughputs of BF and QF were 690,000 and 2,400,000 inserts per second, respectively. Although the performance of QF deteriorated as the number of elements increased, it was always significantly better than that of BF. For lookups, the behavior of both BF and QF was stable throughout the benchmark. The BF performed 1,900,000 lookups per second on average, whereas the QF performed 2,000,000.

CF insertion throughput. We inserted 8.59 billion 64-bit keys into the CF. Figure 3 shows that the CF sustained an average of 670,000 insertions per second even taking into account the time during which long merges stalled insertions. The largest stall was in the middle, where all but one of the QFs were merged into the largest QF of the CF. Deamortization techniques, which we did not implement, can remove the long stalls [1]. We performed the largest merge at 8.4MB/s, well below flash's serial write throughput (110MB/s). We found that the system was CPU-bound, spending its time on bitpacking operations within the QF. In fact, it was so CPU-bound that the disk subsystem ran at only a few percent of capacity even at high insertion rates.

For comparison, we evaluated two other data structures: (1) a traditional BF and (2) a large elevator BF. The traditional BF uses the target disk as storage and hashes keys into this storage, though its writes are allowed to use the file cache. The elevator BF has the following optimization: it maintains a large buffer of locations it has recently written to, and when this buffer is full, it flushes each bit to storage in order of offset.

The traditional BF achieved an insertion throughput of 200 insertions per second, whereas the elevator BF achieved an insertion throughput of 17,000 per second, which is a considerable improvement, but far less than that of our CF. The performance for both algorithms was constant as the data structures filled because it was bounded by the flash's random-write throughput.

Lookup throughput. We compare the lookup throughput of the traditional BF and CF with each other as well as with a theoretical prediction of their performance.

In our setup, the CF has at most 6 levels on flash. The CF performs one read at each level when searching for keys that are not in the CF (6 I/Os). Our drive's random-read throughput is 3,218 4KB pages per second, and so the read throughput of the CF should have been about 530 lookups per second. A BF with an equivalent false-positive rate of 0.04% requires 11 hash functions and 16GB of space. In order to predict its lookup throughput, note that in an optimally configured BF, each bit is set to 1 with probability 1/2. A lookup on a BF uses one hash function after another until it finds a 0, meaning that the expected number of I/Os per negative lookup is 2. Thus, the expected lookup throughput is half the random read throughput of the flash drive, which in this case is 1600 lookups per second.

When measured, the actual BF lookup performance is 1609 lookups per second, which is what the model predicts. Negative CF lookups run at 530 per second, which matches what the model predicts (6 reads per lookup).

CF with tombstone bit. We re-ran the CF throughput experiments with an identical experimental setup, except we used 4 bits per element instead of 3 to measure the overhead of supporting deletes. We found that the insertion throughput dropped from 670,000 insertions per second to 630,000 insertions per second. Lookup throughput remained unchanged.

Evaluation summary. The CF trades a 3 fold slowdown in lookup throughput on flash in exchange for a 40x speedup in insertion throughput over a BF optimized to use all of its buffer for queueing random writes. Unlike the traditional BF, the CF is CPU bound and not I/O bound.

4 Conclusions and Future Work

We designed two efficient data structures: a *Quotient Filter* (QF) and a *Cascade Filter* (CF), specifically to utilize the best features of modern flash drives. We designed them to have high throughput for insertions, queries, and deletions. Our analytical results, coupled with our evaluations, demonstrate superior performance, beating optimized implementations of traditional Bloom filters by over two orders of magnitude.

The relative cost of I/O compared to CPU operations has increased by orders of magnitude over the past several decades, and with the advent of multicore, that trend is likely to continue. Most storage systems underuse their CPUs while waiting for I/O. In contrast, our data structure makes efficient use of I/O and is CPU-bound for insertions. The merge operation is parallelizable, potentially offering additional performance.

Future work. We will explore applications to traffic routing, deduplication, replication, write offloading, load balancing, and security in a data center or large network. The Cascade Filter is currently CPU bound; a parallel implementation could potentially perform upwards of 50 million inserts and updates per second with a drive performing 400MB/s serial writes. An efficient implementation could potentially be made very cost-effective by utilizing parallel GPU programming. The Cascade Filter is capable of a variety of read/write optimized configurations, and can dynamically shift between them at run-time. We will explore application of the Cascade Filter to write-optimized indexing.

5 Acknowledgments

Thanks to Guy Blelloch for helpful discussions and especially for suggesting that we use quotient filters.

References

[1] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *SPAA*, 2007.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In *ECA*, 2006.

[4] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. In *Internet Mathematics*, 2002.

[5] M. Canim, G. A. Mihaila, B. Bhattacharhee, C. A. Lang, and K. A. Ross. Buffered Bloom filters on solid state storage. In *ADMS*, 2010.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[8] Y. Chen, B. Schmidt, and D. L. Maskell. A reconfigurable Bloom filter architecture for BLASTN. In *ARCS*, pages 40–49, 2009.

[9] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE T. Comput.*, 33(9):828–834, 1984.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM T. Netw.*, 8:281–293, June 2000.

[11] Larry Freeman. How netapp deduplication works - a primer, April 2010. `http://blogs.netapp.com/drdedupe/2010/04/how-netapp-deduplication-works.html`.

[12] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., 1991.

[13] Innobase Oy. Innodb. `www.innodb.com`, 2011.

[14] Intel. Over-provisioning an Intel SSD, October 2010. `cache-www.intel.com/cd/00/00/45/95/459555_459555.pdf`.

[15] Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. Lazybase: freshness vs. performance in information management. *SIGOPS Oper. Syst. Rev.*, 44:15–19, March 2010.

[16] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1973.

[17] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. *OS Rev.*, 44(2):35–40, 2010.

[18] K. Malde and B. O'Sullivan. Using Bloom filters for large scale gene sequence analysis in Haskell. In *PADL*, 2009.

[19] J.K. Mullin. Optimal semijoins for distributed database systems. *IEEE T. Software Eng.*, 16(5):558 –560, May 1990.

[20] A. Singh, M. Srivatsa, L. Liu, and T. Miller. Apoidea: A decentralized peer-to-peer architecture for crawling the world wide web. In *SIGIR Workshop Distr. Info. Retr.*, pages 126–142, 2003.

[21] Tokutek, Inc. TokuDB for MySQL Storage Engine, 2009. `http://tokutek.com`.

[22] Vertica. The Vertica Analytic Database. `http://vertica.com`, March 2010.

[23] Z. Yuan, J. Miao, Y. Jia, and L. Wang. Counting data stream based on improved counting Bloom filter. In *WAIM*, pages 512–519, July 2008.

[24] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, 2008.