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

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

Introduction

As the Internet grows, computers collect, store, search, and index data at increasingly rapid rates. A recent IDC study estimates that more than 300 Exabytes of hard-disk storage will be delivered in the next five years to data centers and clouds alone. Many large storage systems use approximate-membership-query (AMQ) data structures to deal with the massive amounts of data that they process. The canonical AMQ data structure is the Bloom filter. Bloom filters, however, do not scale well outside of main memory. Bloom filters which are larger than main memory would choke on disks with rotating platters and moving heads. A rotational disk performs only 100–200 (random) I/Os per second, and each Bloom filter operation requires multiple I/Os. Even on solid-state drives (SSD), Bloom filters only utilize a small fraction of the bits in each block write, and so continue to suffer performance problems because of their appetite for random I/Os. Hence, such storage systems typically organize their Bloom filters as in-memory data structures, which limits the dataset size for which Bloom filters can be used.

We introduce the *Quotient Filter* data structure, which is functionally similar to the popular and efficient dynamic Bloom filter. Our data structure requires only one I/O per lookup, compared to the 6–8 I/Os per lookup that dynamic Bloom filters could potentially require. A quotient filter stores a lossy representation of a multi-set of objects. It maintains a hash table of r-bit fingerprints for each item inserted into the filter. The hash table contains 2^ℓ buckets. Given a hash function h mapping objects to $(\ell + r)$-bit integers, a quotient filter stores an item x by inserting $h(x) \bmod 2^r$ into bucket $h(x)/2^r$ of the hash table. To test whether an item x is in the filter, one simply checks whether any entry in bucket $h(x)/2^r$ has value $h(x) \bmod 2^r$. It also supports efficiently iterating over the hashes of all the objects inserted into the filter, in order of increasing hash value. Therefore it is possible to merge multiple quotient filters into a single quotient filter using an algorithm analogous to the merge operation in merge-sort. By composing several Quotient Filters organized into a modified cache-oblivious lookahead array (COLA) [1] we create a *Cascade Filter*, which can support very high data ingestion rates while maintaining good lookup times, and consuming $1/4$ the space of a dynamic Bloom filter, and $2\times$ the space of a traditional (no-delete) Bloom filter.

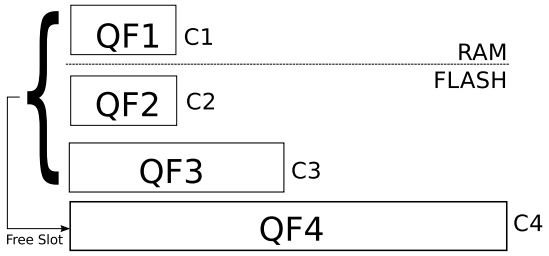

Figure 1: Cascading Filter

The COLA [1] consists of $\lceil \log_2 N \rceil$ arrays of exponentially increasing size, such that an array at level l can accommodate all the arrays at levels 1 through $l - 1$. In the Cascade Filter we replace these arrays with our quotient filters and use this property for merging quotient filters. As a simple example, in Figure 1, Quotient Filter QF1 in RAM is merged with QF2 and QF3 on FLASH, and the resulting Quotient Filter QF4 is placed in the next free level available C4. After merging, levels C1-C3 are marked free.

Work in progress

We evaluated a prototype of our system on an Intel X25-M 160GB SSD II drive. We were able to perform insertions and deletions of more than 8.72 billion elements at 911,131 insertions/s and 378 lookups/s. To put that in perspective, our insertion throughput is 607 times faster than the random write throughput of the Intel X25-M. The increasing random-read throughput of Flash devices [2] and increasing random IOPS to cost ratio [3] would boost our lookup throughput. We argue that our approach scales. For a data center holding 1PB of 512 byte keys, our results indicate that using cheap off-the-shelf parts, one can construct a Cascading Filter with a less than 0.04% false positive rate using 10TB of Flash disks. This device would be relatively inexpensive, costing less than US$35,000. The Cascading Filter is currently CPU bound; a parallel implementation could potentially perform over 70 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.

[*] Stony Brook University.
[†] Rutgers University.
[‡] MIT.
[§] Tokutek, Inc..
[¶] Author is a student.
[‖] Author would be presenting WIP/poster.

We are also exploring applications to traffic routing, deduplication, replication, write offloading, load balancing, and security in a data center or large network. With a cost-effective method of determining whether an item *doesn't* exist in a large petabyte data center from a *single point* in the network, many new scheduling and architectural possibilities become available.

References

[1] M. A. Bender, M. Farach-Colton, J. T. Fineman, et al. Cache-oblivious streaming B-trees. In *SPAA 2007*, 2007.

[2] FusionIO. IODrive octal datasheet. `http://www.fusionio.com/data-sheets/iodrive-octal/`.

[3] G. Gibson M. Polte, J. Simsa. Comparing performance of solid state devices and mechanical disks. In *PDSW 2008*, November 2008.