Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design

Prashant Pandey
ppandey@berkeley.edu
Lawrence Berkeley National Lab
and University of California Berkeley

Alex Conway
aconway@vmware.com
VMware Research

Joe Durie
joedurie17@gmail.com
Rutgers University

Michael A. Bender
bender@cs.stonybrook.edu
Stony Brook University

Martin Farach-Colton
farach@rutgers.edu
Rutgers University

Rob Johnson
robj@vmware.com
VMware Research

ABSTRACT

Today's filters, such as quotient, cuckoo, and Morton, have a trade-off between space and speed; even when moderately full (e.g., 50%-75% full), their performance degrades nontrivially. The result is that today's systems designers are forced to choose between speed and space usage.

In this paper, we present the *vector quotient filter* (VQF). Locally, the VQF is based on Robin Hood hashing, like the quotient filter, but uses power-of-two-choices hashing to reduce the variance of runs, and thus offers consistent, high throughput across load factors. Power-of-two-choices hashing also makes it more amenable to concurrent updates, compared to the cuckoo filter and variants. Finally, the vector quotient filter is designed to exploit SIMD instructions so that all operations have $O(1)$ cost, independent of the size of the filter or its load factor.

We show that the vector quotient filter is 2× faster for inserts compared to the Morton filter (a cuckoo filter variant and state-of-the-art for inserts) and has similar lookup and deletion performance as the cuckoo filter (which is fastest for queries and deletes), despite having a simpler design and implementation. The vector quotient filter has minimal performance decline at high load factors, a problem that has plagued modern filters, including quotient, cuckoo, and Morton. Furthermore, we give a thread-safe version of the vector quotient filter and show that insertion throughput scales 3× with four threads compared to a single thread.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; **Bloom filters and hashing**.

KEYWORDS

Dictionary data structure; filters; membership query

ACM Reference Format:
Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. 2021. Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3452841

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3452841

1 INTRODUCTION

Filters, such as Bloom [9], quotient [43], and cuckoo filters [31], maintain compact representations of sets. They tolerate a small *false-positive rate* ε: a membership query to a filter for set S returns `present` for any $x \in S$, and returns `absent` with probability at least $1-\varepsilon$ for any $x \notin S$. A filter for a set of size n uses space that depends on ε and n but is much smaller than explicitly storing all items of S.

Filters offer performance advantages when they fit in cache but the underlying data does not. Filters are widely used in networks, storage systems, machine learning, computational biology, and other areas [4, 11, 14, 19, 20, 25, 26, 29, 34, 36, 46, 50, 52–54, 56]. For example, in storage systems, filters are used to summarize the contents of on-disk data [5, 16, 21–23, 49, 51, 54]. In networks, they are used to summarize cache contents, implement network routing, and maintain probabilistic measurements [14]. In computational biology, they are used to represent huge genomic data sets compactly [2, 3, 19, 40, 42, 44, 46, 52].

In these applications, filter performance—i.e., space usage, query speed, and update speed—is often the bottleneck. In fact it is often the case that most of the working set of an application is from filters, and the application is impractically slow unless the filters fit in DRAM. Often systems are designed around the constraint that they do not have enough space for their filters [23, 49, 55]. For example, Monkey [23] uses an optimized allocation scheme to minimize the size of filters in-memory. PebblesDB [49] uses over 2/3rds of its working memory for constructing and storing filters. Furthermore, storage devices, such as NVMe SSDs, are fast enough that CPU bottlenecks are common [22].

Modern filters, such as quotient, cuckoo, and Morton [13] filters, are all bumping up against the lower bound on space usage for a dynamic filter, which is $n\log(1/\varepsilon)+\Omega(n)$ bits [17]. As Table 1 shows, these filters differ by less than 1 bit per element, which is less than a 10% difference for typical values of ε (e.g. 1%).

These filters have converged on a common overall design—they encode fingerprints into hash tables. Quotient filters and counting quotient filters [43] are based on Robin Hood hashing [18], and cuckoo and Morton filters are based on cuckoo hashing [39].

All these filters slow down as they are filled, because they experience more collisions. This shows up clearly in Figure 4a, which shows instantaneous insertion throughput as a function of

Filter	Num bits for n items
Bloom filter [9]	$1.44n\log(1/\varepsilon)$
Quotient filter [43]	$1.053(n\log(1/\varepsilon)+2.125n+o(n))$
Cuckoo filter* [31]	$1.053(n\log(1/\varepsilon)+3n+o(n))$
Morton filter [13]	$1.053(n\log(1/\varepsilon)+2.5n+o(n))$
Vector quotient filter	$1.0753(n\log(1/\varepsilon)+2.914n+o(n))$

Table 1: The space usage of different filters in terms of number of items n and false-positive rate ε. Moderns filters use essentially the same space. Quotient, cuckoo, and Morton filters support a maximum load factor of 0.95 and hence face a multiplicative overhead of 1.053. The vector quotient filter supports a load factor of 0.93, for a multiplicative overhead of 1.0753. The different additive overheads (e.g. 2.125 vs. 2.5) come from the different collision-resolution schemes used by the filters. *The cuckoo filter referred throughout the paper has 4 slots per block and 3 bits of space overhead. We picked the standard version as it offers superior performance compared to the semi-sorting variant.

load factor. Even at moderate load factors (e.g., 50%-75% occupancy), their performance degrades nontrivially.[1] For example, the insertion throughput in the cuckoo filter drops 16× when going from 10% occupancy to 90% occupancy and in the quotient filter it drops 4×. The Morton filter is arguably the fastest and most robust of existing filters, and, impressively, its insert throughout does not really degrade substantially until 70% occupancy, at which point it slows down by 2× by the time it reaches 95% occupancy.

As these observations show, the costs of collision resolution have become one of the main roadblocks to further advances in filter performance.

This paper. We present a new filter, the vector quotient filter, that overcomes the collision-resolution roadblock to improving filter update performance. The vector quotient filter shows that it is possible to build a filter that offers high performance and does not slow down across load factors. The vector quotient filter shows how to combine power-of-two-choice hashing with new vector-instruction hardware to build a filter with $O(1)$ insertion time, independent of load factor. Furthermore, these improvements come at no cost to query performance. Empirically,

Insertions: • Insertions in the vector quotient filter have constant high performance from empty to full. We also describe an optimization that further improves insertion performance at low load factors without sacrificing performance at higher load factors. • The vector quotient filter is 10×, 4.5×, and 2× faster at insertions than the cuckoo filter, quotient filter, and Morton filter at 90% load factor. • The vector quotient filter supports aggregate insertions (i.e., from empty to full) over 2× faster than the next fastest filter (the Morton filter).

Deletions: • Vector quotient filter deletions are roughly as fast as in the cuckoo filter, roughly 2× faster than the Morton filter, and 4× faster than the quotient filter. • At high load factors, the vector quotient filter is the clear winner for deletion performance.

Queries: • Queries in the vector quotient filter are roughly 80% as fast as in the cuckoo filter, 50% faster than in the Morton filter, and over twice as fast as in the quotient filter.

Space: The vector quotient filter is nearly as space-efficient as other modern filters (see Table 1). In practice, the vector quotient filter uses around 1 to 2% more space than the cuckoo filter.

Concurrency: • Insertion throughput on a machine with 4 physical cores scales over 3× with 4 threads compared to single-threaded insertion performance in the vector quotient filter, demonstrating nearly linear scaling.

查询和删除稍慢

Limitations. While the vector quotient filter is substantially faster than other filters for insertions, it is slightly slower than the fastest filter (i.e. the cuckoo filter) for queries and deletes. Query-intensive applications might be better served by the cuckoo filter. The vector quotient filter uses similar space as the cuckoo filter and is about 10 to 12% larger than the quotient filter. If space is at an absolute premium, then applications might consider the quotient filter. The vector quotient filter also lacks some of the advanced features of the quotient filter, such as resizability.

The vector quotient filter uses the same xor trick as the cuckoo filter in order to support deletion. Thus, like the cuckoo filter, the probability of failure increases as the filter becomes larger. However, because the vector quotient filter never kicks items from one block to another, it needs the xor trick only in order to support deletions. The cuckoo filter, on the other hand, always needs to use the xor trick, so that it can find an item's alternate block during kicks. Thus, if deletions are not needed, the vector quotient filter can use independent hash functions, and hence the failure probability can be made independent of the filter size.

Where performance comes from. Vector quotient filters achieve these performance gains in three steps.

First, they use power-of-two-choice hashing instead of cuckooing, which avoids the need to perform kicking in order to achieve high load factors.

In power-of-two-choice hashing, items are hashed to two blocks and placed in the emptier block. However, unlike cuckoo hashing, blocks are sized so that they never overflow, so items never need to be kicked from one block to another. Power-of-two-choice hashing ensures that the variance in block occupancies is low, so that all blocks get filled to high occupancy before any block overflows, which means we can get good space efficiency.

Power-of-two-choice hashing makes operations on the vector quotient filter cache efficient. Insertions and lookups access at most two cache lines, and insertions modify at most a single cache line, regardless of the load factor. Insertions into cuckoo and Morton filters, however, perform kicking, and hence access and modify multiple cache lines, and this increases as the filter becomes fuller. This also compares favorably to standard quotient filters where, at high load factors, a single insert may need to touch dozens of cache lines. See Figure 4a, which shows that most modern filters exhibit different amounts of performance degradation as they fill up; and this is due, in a large part, to the increasing cost of collision resolution. We expect that vector quotient filters should perform well on non-volatile memories, where writes are more expensive than reads.

Power-of-two-choice hashing also makes it easy to support concurrent updates, since each updates examines at most two cache

[1] All of these filters define "full" to be somewhat less than 100% occupancy. The quotient filter suggests limiting occupancy to 95% in order to limit collision-resolution costs. The cuckoo and Morton filter limit occupancy to 95% because their failure probabilty shoots up above 95%. This is why all these filters have a 1.053× space overhead, as shown in Table 1. 1/0.95=1.053

lines and modifies at most one. Simple locks on each block or even hardware transactional memory are all that is needed to support concurrent updates. Cuckoo and Morton filters, on the other hand, are difficult to make concurrent, since each update may touch a large number of locations, in essentially random order.

Second, vector quotient filters use a quotient-filter-like metadata scheme to keep the false-positive rate from increasing as we increase the block size. (In cuckoo and Morton filters, the false-positive rate increases with the block size, which is why they keep blocks small and use kicking to achieve high load factors.)

2 RELATED WORK

For decades, the Bloom filter [9] was essentially the only game in town, but Bloom filters are suboptimal in terms of space usage, running time, and data locality, and they support a bare-bones set of operations (insert and lookup).

In particular, Bloom filters consume $\log(e)\ n \log(1/\varepsilon)$ space, which is roughly $\log(e) \approx 1.44$ times more than the lower bound of $n \log(1/\varepsilon) + \Omega(n)$ bits [17]. Bloom filters also incur $\log(1/\varepsilon)$ cache-line misses on inserts and positive queries, giving them poor insertion and query performance.

The Bloom filter has inspired numerous variants [1, 10, 15, 25, 32, 37, 47, 48]. The counting Bloom filter (CBF) [32] replaces each bit in the Bloom filter with a c-bit saturating counter. This enables the CBF to support deletes, but increases the space by a factor of c. The blocked Bloom filter [47] provides better cache locality than the standard Bloom filter but does not support deletion.

The quotient filter (QF) [7, 27, 28, 38] uses a new, non-Bloom-filter design. It is built on the idea of storing small fingerprints via Robin Hood hashing [18]. It supports insertion, deletion, lookups, resizing, and merging. The counting quotient filter (CQF) [43], improves upon the performance of the quotient filter and adds variable-sized counters to count items using asymptotically optimal space, even in large and skewed datasets.

The quotient filter uses $1.053(2.125+\log_2 1/\varepsilon)$ bits per element, which is less than the Bloom filter whenever $\varepsilon \leq 1/64$, which is the case in almost all applications. Quotient filters are also much faster than Bloom filters, since most operations access only one or two cache lines. Geil et al. accelerated the QF by porting it to GPUs [35].

The cuckoo filter [31] uses the idea from quotient filters of hashing small fingerprints but uses cuckoo hashing instead of Robin Hood hashing. Cuckoo filters use $1.053(3+\log_2 1/\varepsilon)$ bits per item, that is, somewhat more than a quotient filter.

The Morton filter [13] is a variant of the cuckoo filter that is designed to speed up insertion using optimizations designed for hierarchical systems. The Morton filter biases insertions towards the primary hash slot and uses an overflow tracking array to speed up negative queries. In addition, the Morton filter employs a compression-based physical representation to store fingerprints in blocks and achieves better space utilization than the cuckoo filter. The Morton filter offers faster insertion throughput compared to the cuckoo filter and also less throughput degradation at high occupancy. The Morton filter offers even faster insertion throughput for bulk insertion scenarios which are often seen in practice. The Morton filter space usage depends on several configuration parameters, but the version benchmarked in the original Morton filter uses approximately $1.053(2.5+\log_2 1/\varepsilon)$.

From the above summary, we can see that the quotient, cuckoo, and Morton filters all use $1.053(K+\log_2 1/\varepsilon)$ bits per element, where K is 2.125, 3, or 2.5, respectively. The main remaining challenge is speed, especially at higher load factors.

3 VECTOR QUOTIENT FILTER

The vector quotient filter uses three techniques to get good performance at all load factors:

- Power-of-two-choice hashing to allocate items to blocks.
- Space-efficient mini-filters within each block.
- SIMD-optimized encoding of mini-filters.

The vector quotient filter uses a power-of-two-choice hashing scheme to allocate keys to blocks. Items are mapped to two blocks and placed into the emptier one. Items are never "kicked" from one block to another, avoiding the complexity and cost of kicking that cuckoo and Morton filters incur. Power-of-two-choice hashing also avoids the long chains of shifted elements in the quotient filter.

The main challenge to using power-of-two-choice hashing instead of cuckooing is that blocks must have large capacity (e.g. $\omega(\log\log n)$) in order to be able to achieve high load factor (and hence high space efficiency). In contrast, cuckoo and Morton filters use small blocks, which means that some blocks fill much sooner than others, but they handle this problem by kicking items from one block to another.

Because they use small (i.e. constant-sized) blocks, cuckoo and Morton filters can use a relatively simple block structure: each block is simply an array of fingerprints, and any query that maps to the block can match with any fingerprint in the block. This means that the false-positive rate is proportional to the block size, which is not a problem in cuckoo and Morton filters because they use small blocks.

So we need a block structure that supports large blocks without blowing up the false-positive rate.

We resolve this dilemma by structuring each block in the vector quotient filter as a mini-filter in its own right. Our mini-filter structure is based on ideas from the quotient filter, and ensures that we can make each mini-filter as large as we want without increasing its false-positive rate. In the VQF, blocks can be as large as a cache line, or even larger, and do not require rebalancing or cuckooing at all.

Finally, we encode the mini-filters to take advantage of recent Intel SIMD instructions for operating on entire cache lines in a single instruction. As a result, all operations on the VQF take constant time. This improves upon the cuckoo, quotient, and Morton filters, for which the cost of insertions grows as both a function of the filter size and its occupancy.

3.1 Power-of-two-choice hashing

As shown in Figure 1, a VQF consists of an array of k blocks, where each block is a small filter with capacity s and false-positive rate $\varepsilon/2$. (In our implementation, s is 48 or 28, see section 6.) Each block is implemented as a mini-filter, described in Section 3.2.

To insert an item, x, we compute two *block indexes* $b_1(x)$ and $b_2(x)$ using independent uniformly random hash functions. We then insert x into the emptier of blocks $b_1(x)$ and $b_2(x)$, following the power-of-two-choices hashing paradigm [6]. If both blocks are full, then the insertion fails.

The following theorem of Berenbrink, et al. enables us to ensure that all the blocks are heavily loaded before any insertion fails.

Theorem 1 (Berenbrink, et al. [8]). *If we toss m balls into n bins using the power-of-two-choices, then, with high probability, the maximum load of any bin is $m/n+O(\ln\ln n)$.*

Furthermore, the constants are quite good—for most practical purposes, we can treat the bound as $m/n+\ln\ln n$.

Thus, to create a VQF for n items, we allocate an array of $k=O(n/\ln n)$ blocks, each with capacity $s=n/k+\Theta(\ln\ln n)$ items and false-positive rate $\varepsilon/2$. By Theorem 1, we can insert all n items into the filter without causing any block to reach its maximum capacity, and hence all the insertions will succeed with high probability.

Note that this filter supports a load factor of $\frac{n}{n+\Theta(n\frac{\ln\ln n}{\ln n})}$. When n is large, this approaches 1, since $\frac{\ln\ln n}{\ln n}=o(1)$. When n is small, say $n<2^{48}$, we can simply pick $k=n/48$, so that the average block occupancy n/k will still be substantially larger than the variance $O(\ln\ln n)$.

To perform a query, we hash x to $b_1(x)$ and $b_2(x)$ and return "present" if x is present in either block. As long as each block has a false-positive rate of at most $\varepsilon/2$, then the filter as a whole will have a false-positive rate of at most ε.

3.2 Mini-filters

In the vector quotient filter, each of the blocks described in Section 3.1 is itself a small quotient filter, which we call a *mini-filter*. We now describe an efficient encoding that we use to implement the mini-filter. This encoding is both space-efficent and, as we will see in Section 3.3 enables insert, lookup and delete operations in $O(1)$ time using AVX-512 instructions.

The key difference between our mini-filter and the unstructured blocks of cuckoo and Morton filters is that we divide the block into b buckets. Each item x inserted into a block is hashed using a hash function $h(x)$. We divide the output $h(x)$ into a $\log b$-bit bucket index, $\beta(x)$, and an r-bit fingerprint, $f(x)$. We then add $f(x)$ to bucket $\beta(x)$. Note that we can recover $h(x)$ from $f(x)$ and $\beta(x)$. Similarly, queries for an element y return yes only if $f(y)$ is present in bucket $\beta(y)$. The main challenge is to efficiently encode the bucket structure.

Figure 1 shows how the mini-filter stores fingerprints and their corresponding buckets. The mini-filter stores a $(b+s)$-bit *bucket-size vector* followed by an array of up to s fingerprints. The fingerprints are stored in bucket order, i.e. all the fingerprints in bucket 0 are stored together, followed by all the fingerprints in bucket 1, and so on. The number of fingerprints in each bucket is stored in unary in the bucket-size vector. The total number of metadata bits is therefore at most $b+s$, and the total size required for a block is at most $b+(1+r)s$. Figure 1 shows an example of the encoding of a mini-filter using colors to distinguish keys across different buckets. For example, in Figure 1, bucket 3 of block 2 has 1 fingerprint, indicated in blue.

This mini-filter encoding improves upon both the cuckoo and quotient filters. In the standard quotient filter, $b=s$. In that case, the mini-filter has 2 bits of metadata overhead per element, whereas the quotient filter has 2.125. The extra bits of overhead in the quotient filter are there to enable queries and updates without parsing the entire filter, which can be huge. Mini-filters, however, are never large, so we can dispense with those extra metadata bits.

Compared to the structure of cuckoo-filter blocks, the mini-filter is even more space efficient. Since cuckoo filter blocks have no structure—just a set of fingerprints—the false-positive rate of queries in a cuckoo-filter block grows roughly linearly in the size of the block. This is why cuckoo filters keep blocks small and are forced to use cuckooing to rebalance blocks. In the mini-filter, however, the false-positive rate can be made indepdendent of the number of slots in a block (see the analysis in Section 5). Thus we can make blocks large enough to make block-occupancy variance a lower order term, without the need for cuckooing.

3.3 SIMD Mini-filter operations

The mini-filter structure described above is specifically designed to be amenable to vector operations. Specifically, operations on metadata can be performed using word-level rank, select, and similar operations, and operations in the fingerprint array can be performed using vector permutation and comparison operations.

We can perform queries in a block using bitvector-select and SIMD-compare instructions, as follows. Let m be the metadata string and t the vector of fingerprints in a block. Define $select(m,i)$ to be the index of the ith 1 in m, where the bits of m and the ones are counted from the left, starting at 0. So, for example, $select(001000000,0)=2$ because the first 1 appears at index 2. Then the first fingerprint for bucket $j>0$ is stored in slot $select(m,j-1)-j$. (The first fingerprint for bucket $j=0$ is stored in slot 0.) Thus all the fingerprints for keys in bucket $j>0$ can be found in slots $[select(m,j-1)-j+1, select(m,j)-j)$ of t. The fingerprints for bucket $j=0$ can be found in the first $select(m,0)$ slots. Furthermore, bitvector-select is fast on modern CPUs. Since the mini-filter metadata vector contains only $O(\log n)$ bits, we can use word operations, such as the x86 PDEP instruction, to perform select in constant time (first used in the counting quotient filter [41, 43]).

During an insert, we must first choose the emptier of two blocks. We can compute the occupancy of a block by computing $select(m,b-1)-b+1$.

Once we have identified the range of slots to check, we can use a SIMD comparison instruction to check all the candidate slots against the queried fingerprint in constant time.

To insert a new key k^* with bucket b^* and fingerprint f^*, we insert f^* into slot $select(m,b^*)-b^*$ of t, shifting over subsequent fingerprints in t, and insert a 0 bit at index $select(m,b^*)$ in m. We can shift the metadata bits and insert the 0 into the metadata using the x86 PDEP instruction and some lookup tables.

We can shift the fingerprints by using a SIMD table-lookup instruction, similar to the AVX-512 VPERMB (Permute Packed Bytes Elements) instruction. These fingerprint and metadata shifting operations can be performed in a small constant number of instructions irrespective of the load factor and enable the VQF to maintain a high and consistent insertion throughput even at high load factors.

3.4 Deletes

Naively, we would like to implement deleting an element x by just finding an instance of $h(x)$ in either of the blocks $b_1(x)$ or $b_2(x)$ and removing it.

The only problem this could cause is false negatives. Note that the only tricky case is when we have inserted two elements x and y with $h(x)=h(y)$ and $b_i(x)=b_j(y)$ for some i and j, and we are now

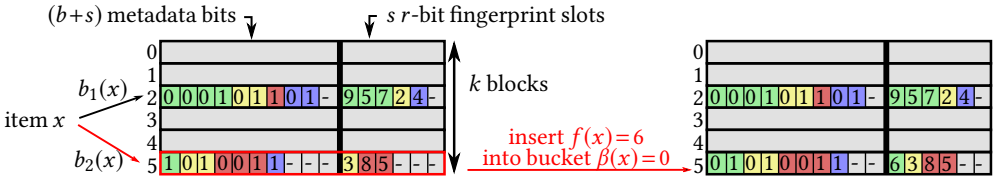


Figure 1: A vector quotient filter and the process of inserting a new item. Each row is a minifilter. Minifilters have b logical buckets, s slots for storing fingerprints, and $b+s$ metadata bits ($b=4$ and $s=6$ in the example). The metadata bits encode, in unary the number of fingerprints in each bucket. The fingerprints for each bucket are stored consecutively in the fingerprint array. So, for example, bucket 0 in block 2 has three fingerprints, 9, 5, amd 7 (indicated in green). To insert an item x, we hash it to blocks $b_1(x)$ and $b_2(x)$ and insert it into the emptier block. In the example, we insert x into block 5 since it is emptier than block 2. To insert x into the block, we add x's fingerprint $f(x)$ to its bucket $\beta(x)$, shifting over metadata bits and other fingerprints as necessary.

trying to delete one of these elements (say x). This is because, if two elements do not have the same value under hash function h, then there is no way they can be confused, and hence no way that deleting one of them could cause a false negative in future queries of the other. Similarly, if two elements have no block in common, then deleting one cannot affect the result of future queries for the other.

The vector quotient filter supports deletes by using the same XOR trick as in the cuckoo filter. We handle these cases by setting the block index $b_2(x) = b_1(x) \oplus h(x)$. Thus, if we insert any two items x and y where $h(x) = h(y)$ and $b_i(x) = b_j(y)$ for any i and j, then we must have that $\{b_1(x), b_2(x)\} = \{b_1(y), b_2(y)\}$. Thus we will have at least two copies of $h(x)$ across blocks $b_1(x)$ and $b_2(x)$, one for x and one for y (and possibly more for other elements). Hence, if we delete x (or y), we can delete one copy of $h(x)$ without causing any false negatives.

The data structure will still guarantee the same false positive behavior even after deleting the item. Because if x (or y) is queried after deletion then it would cause a false positive but that is the expected behavior of filters. Moreover, the secondary hash is computed from the primary hash using a simple multiply-and-xor technique. Thus, the total cost to perform an operation is less than 2 hash computations.

One important requirement for safely deleting (without introducing a false negative) an item is that it must have been inserted. Otherwise, deleting a non-inserted item might unintentionally remove a real, different item that happens to share the same fingerprint. This requirement also holds true for all other deletion-supporting filters.

Using the XOR operation to compute the second hash does not guarantee independence between the first and second hash functions, which is a requirement for the power-of-two-choice hashing. In practice, however, the number of bits in the fingerprints introduce enough randomness to achieve a very high load factor. This same idea is used in the cuckoo filter to support deletion without introducing false negatives. Empirically, the XOR trick has marginal impact on the maximum load factor. In our measurements, it only reduced the maximum load factor from 94.85% to 94.40%.

4 VECTOR QUOTIENT FILTER OPERATIONS

This section describes the algorithms used to implement the insert, lookup and delete operations on a vector quotient filter.

Insert. Algorithm 1 shows the pseudocode for the insert operation. To perform an insert, the item is first hashed to determine the fingerprint u that we will store in either the element's primary or secondary block. Then, it is hashed again to determine the indices

Algorithm 1 Insert (x)

```
1:  u ← h(x)                                   ▷ u is hash to be inserted in mini-filter
2:  b1 ← h1(x)                                 ▷ Compute primary and secondary block indexes
3:  b2 ← b1 ⊕ u
4:  i ← b1                                     ▷ Select index i of emptier block
5:  if SELECT(block[b2].metadata, b−1) < SELECT(block[b1].metadata, b−1) then
6:      i ← b2
7:  end if
8:  if SELECT(block[i].metadata, b−1) = s+b−1 then
9:      return False                           ▷ Filter is full.
10: end if
11: y ← u/2^r                                  ▷ y is bucket index within the block
12: t ← u mod 2^r                              ▷ t is fingerprint
13: m ← SELECT(block[i].metadata, y)           ▷ m is metadata index
14: z ← m−y                                    ▷ z is slot for this fingerprint
15: n ← b+s
16: while n > m do                             ▷ Implemented using PDEP
17:     block[i].metadata[n] ← block[i].metadata[n−1]
18:     n ← n−1
19: end while
20: n ← s
21: while n > z do                             ▷ Implemented using VPERMB
22:     block[i].fingerprints[n] ← block[i].fingerprints[n−1]
23:     n ← n−1
24: end while
25: block[i].metadata[m] ← 0
26: block[i].fingerprints[s] ← t
```

b_1 and b_2 of the primary and secondary blocks and put in whichever is emptier. We compute occupancy of a block by using SELECT, as described in Section 3.3.

Inside a block, the item must be placed at the end of the run of elements in its bucket. We first compute the target bucket y for the element and its fingerprint t. We then use SELECT to compute the position m of the 1 indicating the end of the run of fingerprints in bucket y. From m and y, we compute the correct slot z to store the fingerprint. From here, the rest of the algorithm performs simple shifts over the fingerprints and metadata bits. The while loop on line 12 shifts over the metadata bits from position m onwards, inserting a 0 at position m. In our implementation, the while loop is implemented as two PDEP instructions and some precomputed tables. The while loop on line 17 shifts the fingerprints from position z onwards, and inserts the new fingerprint at position s. In our implementation, this loop is implemented as a single VPERMB instruction and some lookup tables.

Lookup. Algorithm 2 shows the pseudocode for the lookup operation. Analoguously to the insertion algorithm, performing a lookup begins by computing hashes of the key to determine its remainder u as well as its primary and secondary block indices. Then the primary

Algorithm 2 Lookup (x)

```
1:  u ← h(x)                                   ▷ u is hash to be queried in mini-filter
2:  b1 ← h1(x)                                 ▷ Compute primary and secondary block indexes
3:  b2 ← b1 ⊕ u
4:  return 0 ≤ FIND_FINGERPRINT(b1, u) ∨ 0 ≤ FIND_FINGERPRINT(b2, u)
5:
6:  procedure FIND_FINGERPRINT(i, u)
7:      y ← u/2^r                              ▷ y is bucket index within the block
8:      t ← u mod 2^r                          ▷ t is fingerprint
9:      if y = 0 then                          ▷ Compute start of run of elements in bucket y
10:         start ← 0
11:     else
12:         start ← SELECT(block[i].metadata, y−1) − y + 1
13:     end if
14:     end ← SELECT(block[i].metadata, y) − y
15:     while start < end do                   ▷ Implemented using VPCMPB
16:         if block[i].fingerprints[start] = t then
17:             return start
18:         end if
19:         start ← start + 1
20:     end while
21:     return -1
22: end procedure
```

Algorithm 3 Remove (x)

```
1:  u ← h(x)                                   ▷ u is hash to be queried in mini-filter
2:  b1 ← h1(x)                                 ▷ Compute primary and secondary block indexes
3:  b2 ← b1 ⊕ u
4:  if REMOVE_FINGERPRINT(i1, u) then
5:      return True
6:  else
7:      return REMOVE_FINGERPRINT(i2, u)
8:  end if
9:
10: procedure REMOVE_FINGERPRINT(i, u)
11:     y ← u/2^r                              ▷ y is bucket index within the block
12:     ℓ ← FIND_FINGERPRINT(i, u)             ▷ ℓ is position of fingerprint to be removed
13:     if ℓ < 0 then
14:         return False
15:     end if
16:     m ← ℓ + y                              ▷ m is index of metadata bit to be deleted
17:     while m < s + b do                     ▷ Implemented using PEXT
18:         block[i].metadata[m] ← block[i].metadata[m+1]
19:         m ← m + 1
20:     end while
21:     while ℓ < s do                         ▷ Implemented using VPERMB
22:         block[i].fingerprints[ℓ] ← block[i].fingerprints[ℓ+1]
23:         ℓ ← ℓ + 1
24:     end while
25:     return True
26: end procedure
```

and secondary blocks are checked to see if either contains u; if so, the lookup returns "present," otherwise it returns "not present."

Inside a block, we first compute the bucket y for the given item, and use SELECT on the bucket-size bitvector to find the start and end of the run of fingerprints in bucket y. Then we compare each fingerprint in the run to the queried fingerprint t, and return the position of the match if one exists, and -1 otherwise. The comparison loop on line 14 is implemented as a single AVX-512 VCMPB instruction.

Remove. Algorithm 3 shows the pseudocode for the remove operation. The remove operation uses FIND_FINGERPRINT to find the fingerprint in the item's primary or secondary block. If it exists, then it reverses the operations of INSERT. As with insert, we can replace all the loops with AVX-512 instructions, PDEP/PEXT instructions, and lookup tables.

Because the vector quotient filter uses XOR to link the hash functions which determine the primary and secondary blocks, it is safe to remove a remainder found in this way. See section 3.4.

5 SPACE ANALYSIS

We now analyze the space usage of the vector quotient filter and compare it against the space usage of other filters. We perform the analysis on a generalized version of our optimized vector quotient filter, parameterized by b, r, and s, as defined in the table below.

The notation used for analysis is:

ε	target false positive rate
α	maximum allowed load factor
S	number of bits per item
r	number of bits in the fingerprint
b	number of buckets per block
s	number of slots per block
m	number of blocks

We first compute the false-positive rate ε as a function of b, s, and r. Each item maps to two blocks. Within a block, it maps to one of b buckets. The total number of items in a block is at most s, so the average number of items in an item's bucket is s/b. For each item in a query's bucket, there is a 2^{-r} probability that its fingerprint matches that of the query. Thus, by a union bound, we can upper bound the probability of a match within one block as $\frac{s}{b}2^{-r}$. Since each query maps to two blocks, the probability of a match in either block is at most $\varepsilon \le 2\frac{s}{b}2^{-r} = \frac{s}{b}2^{1-r}$. Solving for r gives $r \le \log(1/\varepsilon)+\log(s/b)+1$.

We now compute the bits per item, S, as a function of b, s, r, and the load factor α. Each slot occupies an r-bit fingerprint in a block. Each block also has $b+s$ metadata bits that are shared by the s slots in the block. So the total bits per slot is $r+\frac{b+s}{s}$. If only a fraction α of slots have items stored in them, then the bits per item is

$$S = \frac{r+\frac{b+s}{s}}{\alpha} = \frac{r+b/s+1}{\alpha}.$$

By subtituting $r \le \log(1/\varepsilon)+\log(s/b)+1$ for r in our equation for S, we get

$$S \le \frac{\log(1/\varepsilon)+\log(s/b)+b/s+2}{\alpha}.$$

Thus we can minimize S by minimizing $\log(s/b) + b/s$, which is minimized when $s/b = \ln 2$.

Thus the space used by the vector quotient filter is

$$S \le \frac{\log(1/\varepsilon)+\log\ln 2^e+2}{\alpha} \approx \frac{\log(1/\varepsilon)+2.914}{\alpha}.$$

Note that space usage and false-positive rate are functions of s/b. Thus we can make blocks as large as we like (i.e. we can make s arbitrarily large) without affecting the false-positive rate or space efficiency, so long as we maintain the same ratio of s/b. In the cuckoo and Morton filters, however, the false-positive rate increases with the number of slots per block, which is why they keep blocks small and use kicking.

The additive overhead of 2.914 bits/element is slightly less than the cuckoo filter's 3 bits of overhead. However, as we will see experimentally in Section 7, the vector quotient filter supports load factors only up to 93%. Thus we expect these differences to cancel out, so that the vector quotient filter uses very close to the same space as the cuckoo filter.

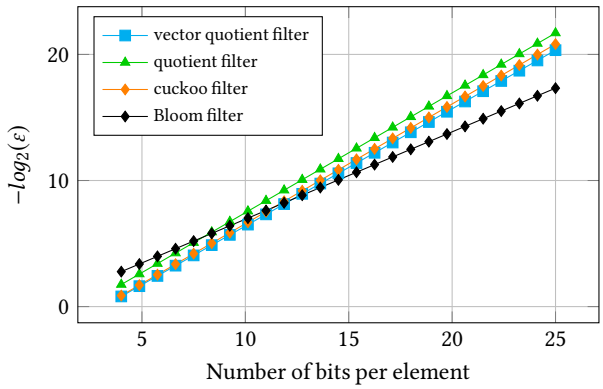


Figure 2: False-positive rate verus the number of bits per element for the vector quotient filter, quotient filter, cuckoo filter, and Bloom filter. The vector quotient filter requires similar space as the cuckoo filter. ε is the false positive rate. The y-axis shows the negative log of the false-positive rate for a clearer interpretation. (Higher is better.)

Figure 2 shows the comparison of the false positive rate and the number of bits per item for the vector quotient filter, counting quotient filter, cuckoo filter, and Bloom filter. We assume a 93% load factor for the vector quotient filter, 95% for the quotient filter and cuckoo filter, and 100% for the Bloom filter. The vector quotient filter space usage for a given false positive rate is similar to the cuckoo filter and slightly higher than the counting quotient filter. The Bloom filter has no additive overhead, so is smaller for large false-positive rates, but its larger multiplicative overhead means that it is larger for small false-positive rates.

Note that the optimal vector quotient filter configuration constrains only the ratio s/b, but says nothing about s or b individually. Thus we can make s and b as large or small a we want, as long as we keep $s/b \approx \ln 2$. In practice, for a given fingerprint size r, we set s and b as large as possible given the constraint that a block, consisting of $s+b+sr$ bits, fit on a single cache line.

We note that Figure 2 shows the theoretical relationship between the false-positive rate and the bits-per-element in different filters. In practice, only the counting quotient filter implementation supports arbitrary fingerprint sizes with relatively good efficiency. The vector quotient filter, cuckoo, and Morton filter implementations evaluated in this paper, on the other hand, support only a few discrete false-positive rates in practice. Our vector quotient filter prototype supports only two false-positive rates—0.004 and 0.000023. The cuckoo filter implementation supports more false-positive rates (by adjusting both the fingerprint size and the block size), but many false-positive rates are not practical. This is because the cuckoo filter implementation supports only 8, 12, 16, and 32-bit fingerprints and, given a fingerprint size r and block size b, it has a false positive rate of $2b2^{-r}$. Thus the only way to get a false-positive rate of say, 2^{-16}, would be to use 32-bit fingerprints and blocks of size $b = 2^{15}$, which would need to be searched on every query, resulting in very slow queries.

6 IMPLEMENTATION AND OPTIMIZATION

This section describes details and optimizations of the vector quotient filter implementation. We begin by analyzing the optimal parameters for the vector quotient filter, then describe a shortcut optimization that we use to speed up insertions, and also how to convert single-threaded operations in the vector quotient filter into thread-safe operations using lightweight spin locks.

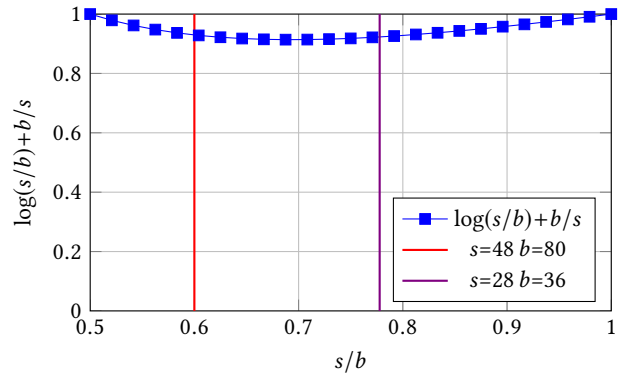


Figure 3: The relation between s/b and the overhead bits in a vector quotient filter. (Lower is better.)

6.1 Optimizing the vector quotient filter for x86

We now describe how we design the vector quotient filter to fit mini quotient filters in a single x86 cache line.

There are several constraints on the design. Ideally, each mini-filter should fit in a 512-bit cache line with no wasted bits. In order to use the VPERM and VCMP instructions to shift and search fingerprints (see section 3.3), those fingerprints should be 8, 16, or 32 bits and aligned. Finally, we want to minimize the false-positive rate and maximize the capacity.

Therefore, when setting the parameters of the mini-filter, we want to choose b and s so that the total space of the mini-filter, $b+(r+1)s$, is as close to 512 as possible, without going over. Subject to this constraint, we want to keep s/b as close as possible to ln2.

Our prototype supports 8 and 16-bit fingerprints. For 8-bit fingerprints, we choose $s = 48$ and $b = 80$. For 16-bit fingerprints, we use $s = 28$ and $b = 36$. We choose these values because they result in 128-bit and 64-bit metadata, respectively, waste no bits in a block, and have fast multiply, divide, and mod algorithms. Furthermore, they both achieve close to the optimal bit overhead. Figure 3 shows the bit overhead from various choices of s/b, and the two points chosen in our implementation. As the graph shows, the overhead curve is relatively flat in the region around its minimum, so there is not much cost in choosing convenient points near the minimum. For example, our choices result in 0.93 and 0.923 bits of overhead for 8 and 16 bits fingerprints, respectively, compared to the optimal of 0.914 bits.

6.2 Shortcuts during insertion

The insert operation in the vector quotient filter described in section 4 must check the occupancy of both the blocks given by the two block indexes and pick the less loaded block. This causes two cache-line misses during every insert operation irrespective of the load factor. Thus, while the power-of-two-choices allocation scheme balances load extremely well, it can lead to a higher insertion cost than simply allocating with a single hash function.

The *shortcut optimization* balances these two schemes: allocating to a single block when the occupancy is low and allocating with power-of-two-choices when the occupancy is high. This is implemented on insertion by checking the occupancy of the first block first and always selecting it if it is less than 75% occupied, thereby eliding the access to the second block completely. As a result, the insert operation incurs only a single cache-line miss during low load factors and speeds up the average insertion throughput.

Empirically, we find that this optimization only slightly reduces the maximum load factor of the vector quotient filter, and increases insert performance substantially. We empirically evaluated the effect of shortcut optimization on the maximum load factor for different thresholds. For the 75% threshold the optimization has only a marginal effect reducing the maximum load factor from 94.40% to 93.56%. However, increasing the threshold higher than 75% did show a sharp decrease in the maximum load factor. For example, with 87.5% and 95.83% threshold the maximum load was reduced to 90% and 64.83%. See section 7 for an evaluation of the insert operation performance with and without the shortcut optimization.

6.3 Multi-threading

We now describe an implementation of thread-safe operations in the vector quotient filter. In the vector quotient filter, each insert, lookup, and remove operation touches at most two blocks and each block fits in a cache line. Therefore, most operations occur on independent cache lines, so that threads rarely contend for the same cache lines. As a result, the vector quotient filter is especially amenable to highly concurrent thread-safe operations.

In the thread-safe implementation, the rightmost bit in the metadata bitvector of each block is used as a lock bit. As explained in section 3.2, $b+s$ metadata bits are required to store s fingerprints belonging to b buckets in a block. However, if it isn't full, it uses fewer, and if it is full, then the last (rightmost) bit is always 1. Therefore, we can use it as the lock bit and treat it as though it were 1 in the bucket-size bitvector.

A lock is acquired or released using atomic instructions ("__sync_fetch_and_or" to lock and "__sync_fetch_and_and" to release). To acquire the lock on a block, we set the bit to 1 and reset the bit to release the lock. When multiple locks are held at once, they are always acquired in increasing order of block index. This protocol avoids any potential deadlocks.

During an insert operation, the lock is acquired on the primary block before checking its occupancy. Following the shortcut optimization, if the occupancy is high enough the secondary block is checked as well. But in that case, if the secondary block has a lower index than the primary block, the lock on the primary block is released and then reobtained after acquired the lock on the secondary block, as per the locking order protocol.

During the lookup and remove operations, we acquire the lock on the block only during FIND_FINGERPRINT or REMOVE_FINGERPRINT.

7 EVALUATION

In this section, we evaluate our implementation of the vector quotient filter (VQF). We compare the vector quotient filter against three other filter data structures: Fan et al.'s cuckoo filter (CF) [30], Breslow et al.'s Morton filter (MF) [13], and Pandey et al.'s quotient filter (QF) [43].

We evaluate each data structure on three fundamental operations: insertions, lookups, and removals. We evaluate lookups both for items that are present and for items that are not present in the filter.

This section tries to address the following questions about how filters perform in RAM and L3 cache:

(1) How does the vector quotient filter (VQF) compare to the cuckoo filter (CF), Morton filter (MF), and quotient filter (QF) when the filters are in RAM?
(2) How does the vector quotient filter (VQF) compare to the cuckoo filter (CF), Morton filter (MF), and quotient filter (QF) when the filters fit in L3 cache?
(3) How does the vector quotient filter (VQF) compare to the cuckoo filter (CF) and Morton filter (MF) when running a mixed workload at high occupancy?
(4) How does the insertion throughput of the vector quotient filter (VQF) scales with multiple threads?

7.1 Experimental setup

In order to see the impact of collision resolution, we report the performance on all operations as a function of the data structures' load factor. This also eases comparison with prior work, which uses the same methodology [7, 13, 31, 43]. We also report the aggregate throughput performance which is the performance of the filter going from scratch to 95% (or 90%) load factor.

One challenge we face is that the filters do not all support the same false-positive rates. For example, the cuckoo filter implementation [30] supports only 2, 4, 8, 12, 16, and 32-bit fingerprints. The false-positive rate can further be tweaked by a small amount by adjusting the block size, but making the blocks too small increases the failure probability, and making them too large decreases performance. This is why the cuckoo filter authors recommend a block size of 4. The Morton filter implementation [12] has similar limitations.

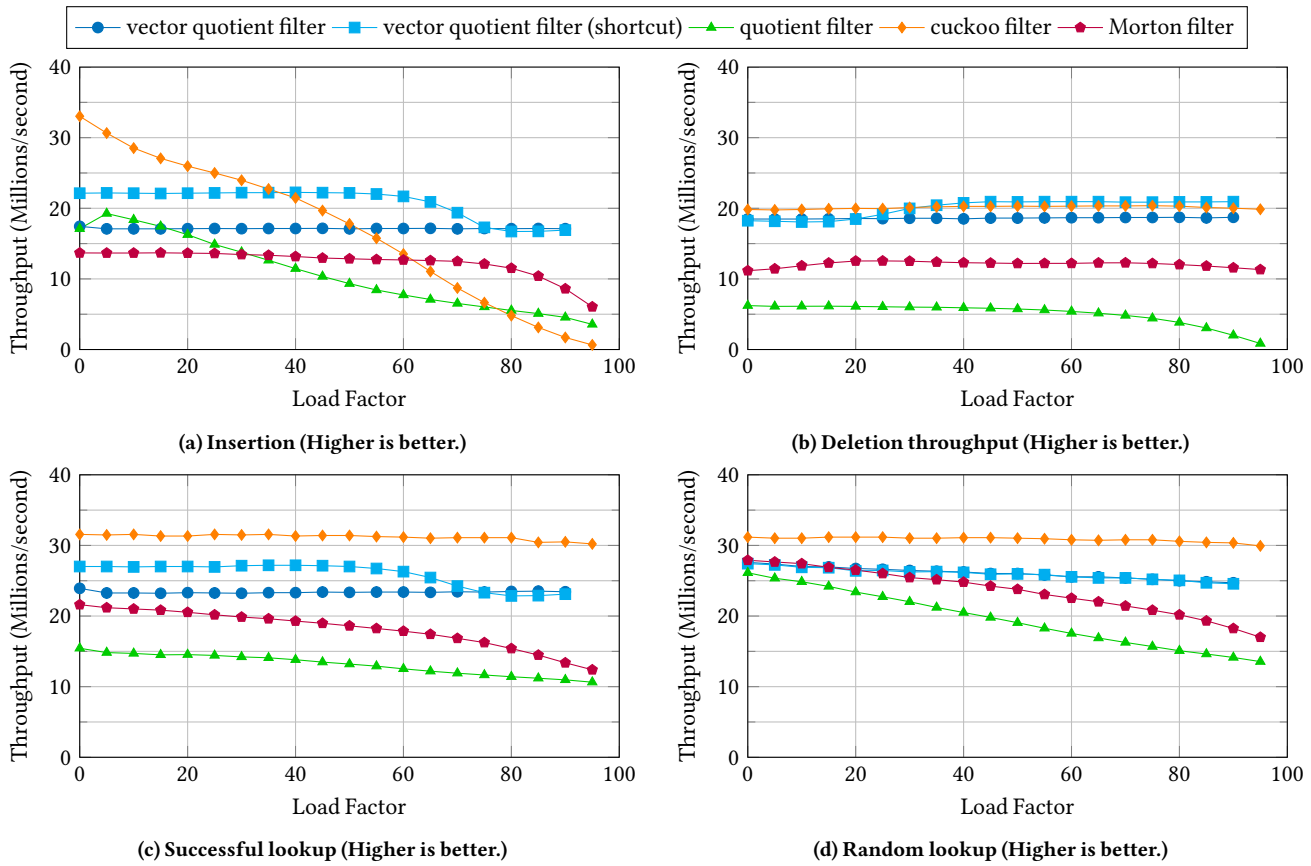
Thus we pick two target false positive rates and configure each filter to get as close as possible to those false-positive rates without sacrificing performance. Our target false-positive rates are 2^{-8} and 2^{-16}. We configure the vector quotient filter with 8 and 16-bit fingerprints, respectively and slots and buckets as described in Section 6. We use 8- and 16-bit fingerprints in the quotient filter. We use 12- and 16-bit fingerprints and blocks of size 4 in the cuckoo filter. We use 8- and 16-bit fingerprints and blocks of size 3 in the Morton filter.

Table 2 shows the empirical space usage and false-positive rate of different filters in these experiments. In the 8-bit experiments, all the filters are within roughly a factor of two in terms of false-positive rate. In the 16-bit experiments, the cuckoo filter false-positive rate is significantly higher than the other filters due to limitations of the implementation.

To compare these filters space and false-positive rate, we compute each filter's *space efficiency* in Table 2, which is defined to be

$$\frac{n \log 1/\varepsilon}{S},$$

where n is the number of items in a full filter (i.e. at the maximum supported occupancy), ε is the false-positive rate achieved by the filter, and S is the total number of bits used by the filter. As Table 2 shows, the quotient filter is the most space efficient, followed by the Morton filter. The cuckoo filter is more space efficient than the vector quotient filter for our 8-bit experiments, but the vector quotient



(a) Insertion (Higher is better.)

(b) Deletion throughput (Higher is better.)

(c) Successful lookup (Higher is better.)

(d) Random lookup (Higher is better.)

Figure 4: Insertion, deletion, and lookup performance of different filters in RAM. All filters were configured with a target false-positive rate of 2^{-8}, as described in Table 2. Shortcut refers to the optimization described in Section 6.2. Note that in Figure 4d, the lines for the vector quotient filters with and without the shortcut optimization nearly coincide. The vector quotient filter is shown with throughput only up to 90% because it reaches full capacity at 93%.

filter is more efficient than the cuckoo filter for 16-bit experiments. Nonetheless, the differences are relatively small across the board.

The configurations used in our experiments are consistent with the author's recommendations and show these filters at or near their best performance. For example, all other configurations that we tried for the Morton filter were slower. The cuckoo filter is ≈ 20% faster with 8-bit fingerprints, but this gives a false-positive rate of 1/32, which is too high for many applications.

We evaluate the performance of the data structures in RAM as well as in L3 cache. This is because applications use filters in multiple different scenarios and filters are often small enough to completely fit in L3 cache. We perform two sets of benchmarks. For the in-RAM benchmark, we create the data structures with 2^{28} (268M) slots which makes all the data structures substantially larger than the L3 cache. For the in-cache benchmark, we create the data structures with 2^{22} (4M) slots (and 2^{21} slots for 16-bit fingerprints) which keeps them well smaller than the size of the L3 cache (8MB).

All the experiments were run on an Intel Ice Lake CPU (Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz with 4 cores and 8MB L3 cache) with 15 GB of RAM running Ubuntu 19.10 (Linux kernel 5.3.0-26-generic).

Microbenchmarks. We measure performance on raw inserts, removals, and lookups which are performed as follows. We generate 64-bit hash values from a uniform-random distribution to be inserted, removed or queried in the data structure. Items are inserted into an empty filter until it reaches its maximum recommended load factor (e.g., 95%). The workload is divided into slices, each of which is 5% of the load factor. The time required to insert each slice is recorded, and after each slice, the lookup performance for that load factor is measured. Once the data structure is 95% full, items that were inserted are removed—again in slices of 5% of the load factor—until the data structure is empty and measure the performance after removing each slice.

We measure the query performance for items that exist (successful lookups) and items that do not exist in the filter (random lookups). For successful lookups, we query items that are already inserted and for random lookups we generate a different set of 64-bit hashes than the set used for insertion. The random lookup set contains almost entirely non-existent hashes because the hash space is much bigger than the number of items in the filter. Empirically, 99.9989% of hashes in the random lookup query set were non-existent in the input set.

The vector quotient filter supports up to only 93% load factor for in-RAM experiments and was able to support up to 95% load factor

Target log(FPR)	8			16		
Filter	log(FPR)	Space (MB)	Efficiency	log(FPR)	Space (MB)	Efficiency
Quotient filter	8.16	324.20	0.76	16.44	580.35	0.76
Cuckoo filter*	9.15	384.00	0.72	13.17	512.00	0.70
Morton filter	8.50	356.19	0.73	16.96	606.88	0.72
Vector quotient filter	7.84	341.34	0.68	15.15	585.14	0.72

Table 2: Empirical space usage and false-positive rate of filters used in the benchmarks. All filters were created with 2^{28} slots (in-RAM experiments). Space is given in MB. *In our 8-bit experiments, we configure the cuckoo filter with 12-bit fingerprints so that its false-positive rate roughly matches the other filters. In our 16-bit experiments, there is no practical way to configure the cuckoo filter for a matching false-positive rate, so we just use 16-bit fingerprints, which gives a much higher false-positive rate.

for in-cache experiments due to the difference in the number of items inserted in the data structure. Therefore, for in-RAM experiments, the vector quotient filter plots do not show the throughput at 95% load factor.

In order to isolate the performance differences between the data structures, we do not count the time required to generate the random inputs to the filters.

Application workload. We also measure the performance of the data structures on workloads consisting of equal portions of insertions, removals, and lookups when the data structure is maintained at a high load factor (90%). This workload is characterized as a *write heavy* (WH) workload [24] because it involves inserting and removing items from the data structure when it is almost full. This type of workload is often seen in real-world applications and the performance of the data structure at a high load factor and under a write heavy workload is critical for applications to scale.

For the application workload, we first fill up all the data structures to 90% load factor. We then perform operations from a mixed workload and compute the aggregate throughput of the data structure to execute the set of operations.

The Morton filter supports a batch API for insertions and queries [13]. Nonetheless, we use the one-at-a-time API for two reasons. First, this makes an apples-to-apples comparison with the other filters. Second, many applications cannot use batching, and we want our benchmarks to reflect the performance that such applications would see.

7.2 In-RAM performance

Figure 4 shows the in-RAM performance of data structures. The vector quotient filter has the highest insertion throughput compared to other data structures. It is 2× and 2.5× faster than the Morton filter and cuckoo filter, respectively. Aggregate throughput of different operations are shown in Figure 6a.

The insertion throughput of the vector quotient filter without the shortcut optimization stays consistent across different load factors. With the shortcut optimization, the insertion throughput is ≈ 1.25× higher until ≈ 75% load factor and then becomes similar to the no shortcut optimization. However, the aggregate insertion throughput is higher with the shortcut optimization. The shortcut optimization does not affect maximum load factor for the vector quotient filter. For removals and lookups, the vector quotient filter has similar throughput to the cuckoo filter and is 1.5× faster than the Morton filter.

The quotient filter has the lowest throughput for all operations. This is due to the additional overheads of maintaining counters with unbounded size and support for storing associated values with items.

Filter	Throughput (Million/sec)
vector quotient filter	20.268
cuckoo filter	3.147
Morton filter	11.958

Table 3: Aggregate throughput for application workload. Workload includes 100M operations (equally divided into insertions, deletions, and queries) at 90% load factor of different filters in RAM. All filters were configured for a target false-positive rate of 2^{-8}, as described in Table 2.

Our performance results for the Morton filter are worse than the main experimental results from the Morton filter paper [13]. This is because the Morton filter implementation is optimized for AMD CPUs, but we evaluate it on an Intel CPU, where performance is known to be worse. For example, Figure 17 in the Morton filter paper [13] shows that the Morton filter speed on a Skylake-X CPU is similar or worse than the CF. Our results are consistent with that.

7.3 In-cache performance

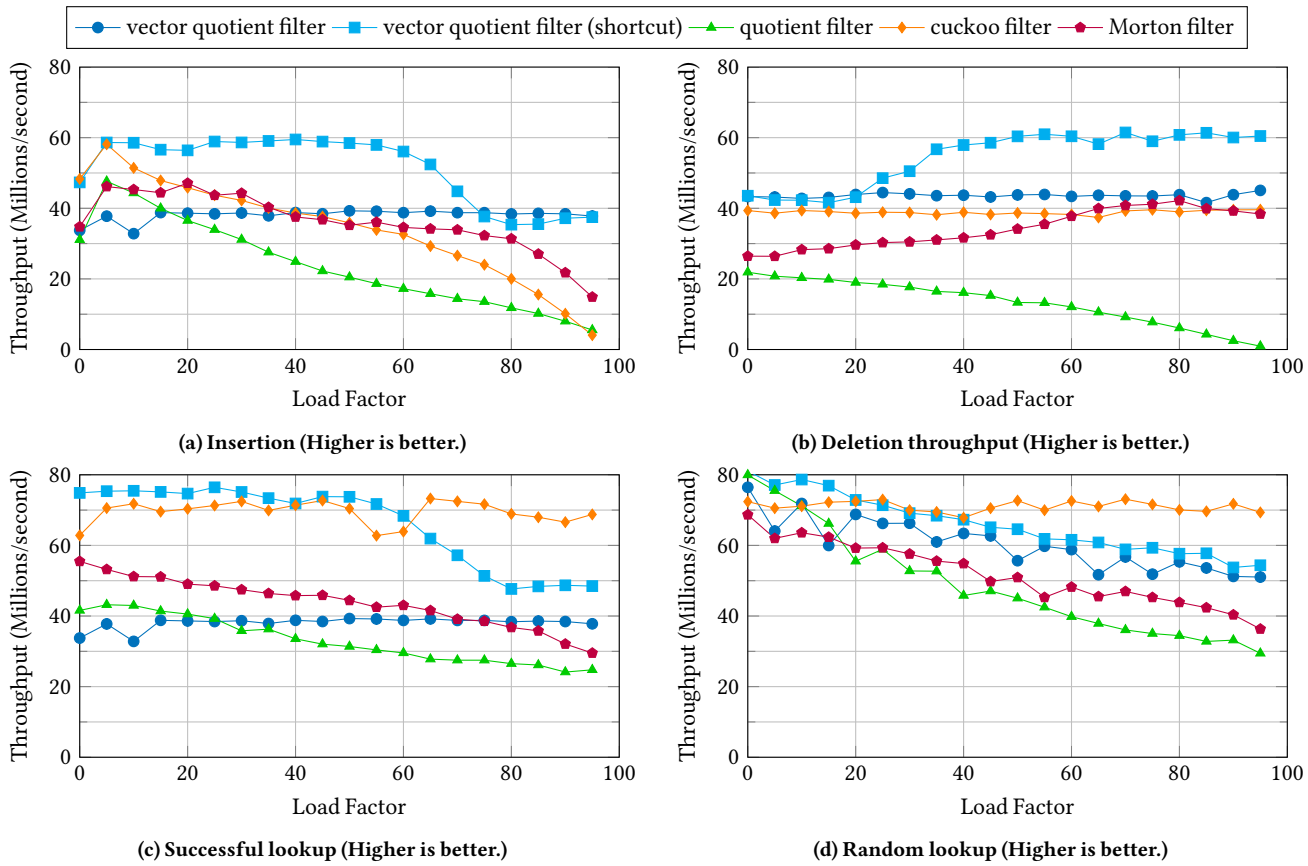
Figure 5 shows the in-cache performance of data structures. Throughput for all operations when the filters are in-cache operation is much higher compared to their corresponding throughput in RAM. The relative performance of different operations in-cache across data structures shows similar trend as the in-RAM performance. The vector quotient filter has the highest insertion and removal throughput and offers lookup performance similar to the cuckoo filter. Aggregate throughput of different operations are shown in Figure 6b.

7.4 Low false-positive rate performance

Figures 6c and 6d show the performance (aggregate throughput) of the filters at very low ($\approx 2^{-16}$) false-positive rates. The relative performance of the filters with 16-bit fingerprints shows similar trends as the 8-bit performance. One difference from the 8-bit results is that, with 16-bits the random lookup performance of the vector quotient filter is higher than the cuckoo filter. This is because the false-positive rate is very low and almost all random lookups are negative queries. The vector quotient filter has an early exit condition in this case. The instantaneous throughput performance of all the data structures for 16-bit fingerprints shows similar trends as the 8-bit fingerprint. We omit the instantaneous throughput plots due to space constraints.

7.5 Write heavy workload

Table 3 shows the throughput of data structures for a write heavy workload when the filters are maintained at 90% load factor. We did not use the quotient filter for this workload as it was the slowest



(a) Insertion (Higher is better.)

(b) Deletion throughput (Higher is better.)

(c) Successful lookup (Higher is better.)

(d) Random lookup (Higher is better.)

Figure 5: Insertion, deletion, and lookup performance of different filters in L3 cache. The vector quotient filter with shortcut refers to the optimization described in Section 6.2. All filters were configured with a target false-positive rate of 2^{-8}, as described in Table 2.

Num threads	Throughput (Million/sec)
1	16.059
2	31.154
3	43.737
4	54.282

Table 4: Insertion throughput with increasing number of threads in RAM. All filters were configured for a target false-positive rate of 2^{-8}, as described in Table 2.

data structure for both in-RAM and in-cache benchmarks. We also only use the vector quotient filter with shortcut optimization as it has higher aggregate throughput compared to no optimization. All filters were configured for a target false-positive rate of 2^{-8}, as described in Table 2. The vector quotient filter is 1.6× faster than the Morton filter and 6.4× faster than the cuckoo filter. It is due to the slow insertion performance of the cuckoo and Morton filters at high load factors that they become really slow to operate for write heavy workloads at high load factors.

7.6 Scaling with multiple threads

Table 4 shows the insertion throughput of the vector quotient filter with multiple threads. All filters were configured for a target false-positive rate of 2^{-8}, as described in Table 2. The insertion throughput increases almost linearly with increasing number of threads with ≈3× increase from 1 thread to 4 threads. We scale up to only 4 threads as the machine only had 4 physical cores and we do not have access to a machine with more than 4 cores that also supports AVX512BW instructions. The multi-threaded benchmark was performed using the same configuration as the In-RAM experiments (Figure 4).

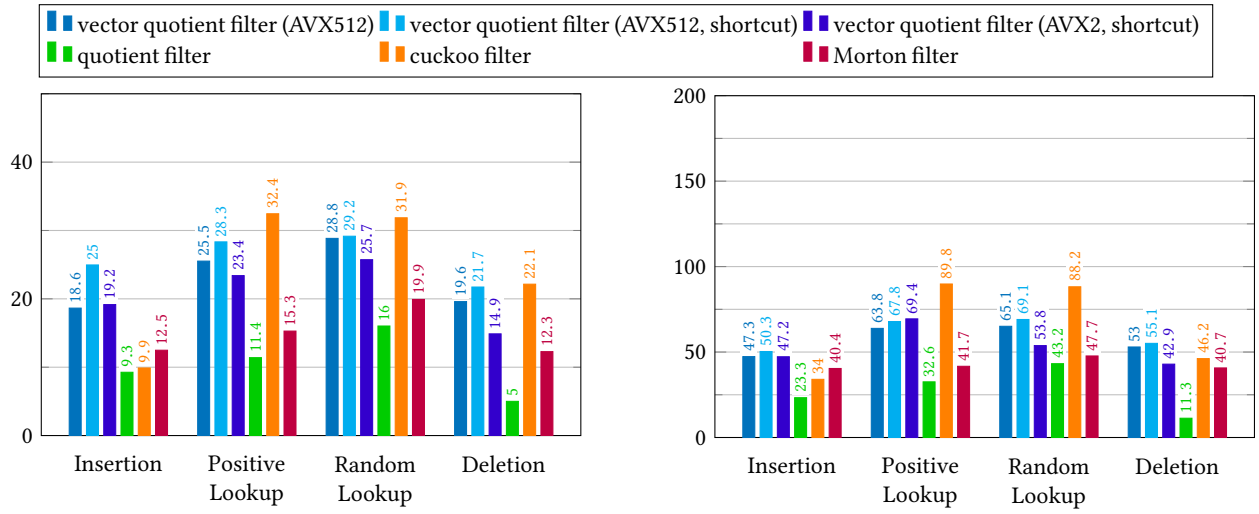
7.7 Impact of AVX512 intrinsics on performance

We implemented a variant of the vector quotient filter using only AVX2 instruction set and not using any of the AVX512 intrinsics. In our evaluation (using the same configuration as experiments in Figure 6a), the AVX2 variant was between 13% to 46% slower than AVX512 variant for different operations. The biggest impact of AVX512 is on the deletion performance. However, even without AVX512 intrinsics the vector quotient filter is between 17% to 34% faster than the Morton filter for all operations and 48% faster than the cuckoo filter for inserts.

8 CONCLUSION

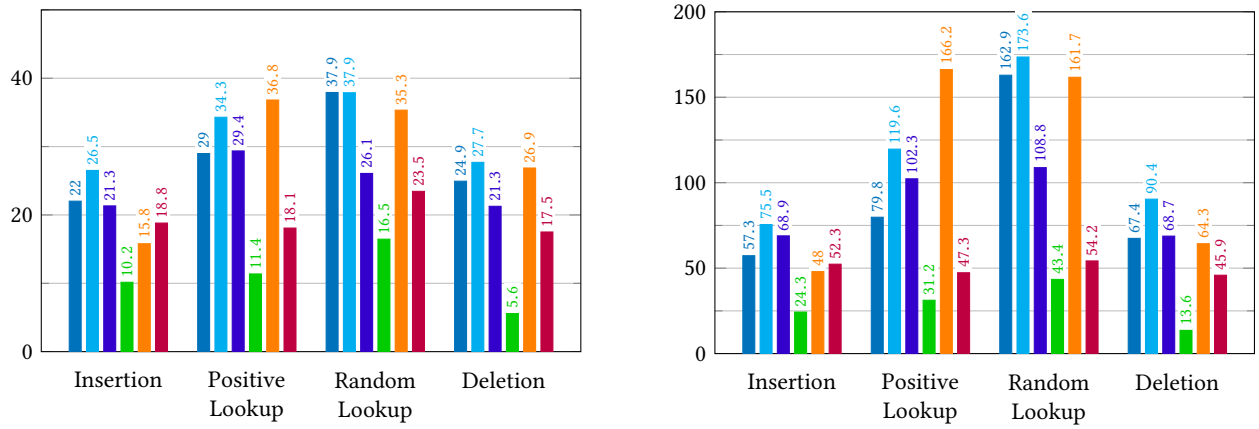
This paper shows that it is possible to build a filter that is space-efficient and offers consistently high insertion and deletion throughput even at very high load factors.

The vector quotient filter offers superior insertion performance compared to the state-of-the-art filters, especially at high load factors,



(a) Aggregate throughput in RAM. Filters were configured with a target false-positive rate of 2^{-8}, per Table 2.

(b) Aggregate throughput in L3 cache. Filters were configured with a target false-positive rate of 2^{-8}, per Table 2.



(c) Aggregate throughput in RAM. Filters were configured with a target false-positive rate of 2^{-16}, per Table 2.

(d) Aggregate throughput in L3 cache. Filters were configured with a target false-positive rate of 2^{-16}, per Table 2.

Figure 6: Aggregate throughput for insertion, deletion, and lookup performance of different filters in RAM and L3 cache. The vector quotient filter with shortcut refers to the optimization in section 6.2. (Higher is better.)

where vector quotient filter insertions are over 2× faster other modern filters. Vector quotient filter queries are slightly slower than in the cuckoo filter, but faster than the other filters in our experiments.

We attribute the high throughput and space-efficiency of the vector quotient filter to two things, the power-of-two-choice hashing and SIMD instructions. Power-of-two-choice hashing reduces the mini filter occupancy variance, enabling high occupancy. The SIMD instructions enable the vector quotient filter to perform constant-time operations in mini filters.

Like the quotient filter, the vector quotient filter also has the ability to associate a small value with each item. Applications often use the value bits to store some extra information with each item in the filter [21, 33, 45]. We believe the ability to associate a value with each key makes the vector quotient filter a go-to data structure in every application builder's toolbox.

ACKNOWLEDGMENTS

We gratefully acknowledge support from NSF grants CCF 805476, CCF 822388, CCF 1724745, CCF 1715777, CCF 1637458, IIS 1541613, CNS 1408695, CNS 1755615, CCF 1439084, CCF 1725543, CSR 1763680, CCF 1716252, CCF 1617618, CNS 1938709, IIS 1247726, CNS-1938709. This research is funded in part by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE under contract number DE-AC02-05CH11231. We used resources of the NERSC supported by the Office of Science of the DOE under Contract No. DEAC02-05CH11231. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable Bloom filters. *Journal of Information Processing Letters* 101, 6 (2007), 255–261.

[2] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. 2019. An Efficient, Scalable and Exact Representation of High-Dimensional Color Information Enabled via de Bruijn Graph Search. In *International Conference on Research in Computational Molecular Biology (RECOMB)*. Springer, 1–18.

[3] Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. 2020. An Efficient, Scalable, and Exact Representation of High-Dimensional Color Information Enabled Using de Bruijn Graph Search. *Journal of Computational Biology* 27, 4 (2020), 485–499.

[4] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. 2014. Storage management in AsterixDB. *Proceedings of the VLDB Endowment* 7, 10 (2014), 841–852.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. 53–64.

[6] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. 1999. Balanced allocations. *SIAM J. Comput.* 29, 1 (1999), 180–200.

[7] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kaner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment* 5, 11 (2012).

[8] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. 2006. Balanced Allocations: The Heavily Loaded Case. *SIAM J. Comput.* 35, 6 (June 2006), 1350–1385. https://doi.org/10.1137/S009753970444435X

[9] Burton H. Bloom. 1970. Space/time Trade-offs in Hash Coding With Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[10] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting Bloom filters. In *European Symposium on Algorithms (ESA)*. Springer, 684–695.

[11] Phelim Bradley, Henk C Den Bakker, Eduardo PC Rocha, Gil McVean, and Zamin Iqbal. 2019. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology* 37, 2 (2019), 152–159.

[12] Alex D Breslow. 2018. Morton Filter source code in C++. https://github.com/AMDComputeLibraries/morton_filter. [Online; accessed 19-July-2020].

[13] Alex D Breslow and Nuwan S Jayasena. 2018. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1041–1055.

[14] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of Bloom filters: A survey. *Internet Mathematics* 1, 4 (2004), 485–509.

[15] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Christian A Lang, and Kenneth A Ross. 2010. Buffered Bloom Filters on Solid State Storage.. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. 1–8.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[17] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. 1978. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*. 59–65.

[18] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (FOCS)*. 281–288.

[19] Rayan Chikhi and Guillaume Rizk. 2013. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology* 8, 1 (2013), 22.

[20] Justin Chu, Sara Sadeghi, Anthony Raymond, Shaun D Jackman, Ka Ming Nip, Richard Mar, Hamid Mohamadi, Yaron S Butterfield, A Gordon Robertson, and Inanc Birol. 2014. BioBloom tools: fast, accurate and memory-efficient host species sequence screening using bloom filters. *Bioinformatics* 30, 23 (2014), 3402–3404.

[21] Alexander Conway, Martin Farach-Colton, and Philip Shilane. 2018. Optimal Hashing in External Memory. In *ICALP (LIPIcs)*, Vol. 107. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 39:1–39:14.

[22] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *USENIX Annual Technical Conference*. USENIX Association, 49–63.

[23] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

[24] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G Khatib, and Cristian Ungureanu. 2016. Revisiting hash table design for phase change memory. *ACM SIGOPS Operating Systems Review* 49, 2 (2016), 18–26.

[25] Biplob Debnath, Sudipta Sengupta, Jin Li, David J Lilja, and David HC Du. 2011. BloomFlash: Bloom filter on flash-based storage. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*. 635–644.

[26] Biplob K Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory.. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[27] Peter C. Dillinger and Panagiotis (Pete) Manolios. 2009. Fast, All-Purpose State Storage. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*. Springer-Verlag, Berlin, Heidelberg, 12–31. https://doi.org/10.1007/978-3-642-02652-2_6

[28] Gil Einziger and Roy Friedman. 2016. Counting with TinyTable: Every Bit Counts!. In *Proceedings of the 17th International Conference on Distributed Computing and Networking (ICDCN '16)*. Association for Computing Machinery, New York, NY, USA, Article 27, 10 pages. https://doi.org/10.1145/2833312.2833449

[29] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS Streaming File System. In *Proc. 4th USENIX Workshop on Hot Topics in Storage (HotStorage)*. Boston, MA, USA.

[30] Bin Fan. 2014. Cuckoo Filter source code in C++. https://github.com/efficient/cuckoofilter. [Online; accessed 19-July-2014].

[31] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. 75–88.

[32] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)* 8, 3 (2000), 281–293.

[33] Martin Farach and S. Muthukrishnan. 1996. Perfect Hashing for Strings: Formalization and Algorithms. In *CPM (Lecture Notes in Computer Science)*, Vol. 1075. Springer, 130–140.

[34] Martin Farach-Colton, Rohan J. Fernandes, and Miguel A. Mosteiro. 2009. Bootstrapping a hop-optimal network in the weak sensor model. *ACM Trans. Algorithms* 5, 4 (2009), 37:1–37:30.

[35] Afton Geil, Martin Farach-Colton, and John D Owens. 2018. Quotient filters: Approximate membership queries on the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 451–462.

[36] Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S Austin Hammond, Golnaz Jahesh, Hamza Khan, Lauren Coombe, Rene L Warren, et al. 2017. ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter. *Genome research* 27, 5 (2017), 768–777.

[37] Guanlin Lu, Biplob Debnath, and David HC Du. 2011. A Forest-structured Bloom Filter with flash memory. In *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–6.

[38] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. 2005. An optimal Bloom filter replacement. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 823–829.

[39] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *European Symposium on Algorithms*. Springer, 121–133.

[40] Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. 2018. Mantis: A fast, small, and exact large-scale sequence-search index. *Cell systems* 7, 2 (2018), 201–207.

[41] Prashant Pandey, Michael A Bender, and Rob Johnson. 2017. A fast x86 implementation of select. *arXiv preprint arXiv:1706.00990* (2017).

[42] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics* 33, 14 (2017), i133–i141.

[43] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 775–787.

[44] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. 2017. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics* 34, 4 (2017), 568–575.

[45] Prashant Pandey, Shikha Singh, Michael A Bender, Jonathan W Berry, Martín Farach-Colton, Rob Johnson, Thomas M Kroeger, and Cynthia A Phillips. 2020. Timely Reporting of Heavy Hitters using External Memory. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1431–1446.

[46] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. 2012. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences* 109, 33 (2012), 13272–13277.

[47] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*. 108–121.

[48] Yan Qiao, Tao Li, and Shigang Chen. 2014. Fast Bloom Filters and Their Generalization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 25, 1 (2014), 93–103.

[49] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[50] Brandon Reagen, Udit Gupta, Robert Adolf, Michael M Mitzenmacher, Alexander M Rush, Gu-Yeon Wei, and David Brooks. 2017. Weightless: Lossy weight encoding

for deep neural network compression. *arXiv preprint arXiv:1711.04686* (2017).
[51] RocksDB [n. d.]. RocksDB. https://rocksdb.org/, Last Accessed Sep. 26, 2018.
[52] Brad Solomon and Carl Kingsford. 2016. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology* 34, 3 (2016), 300.
[53] Henrik Stranneheim, Max Käller, Tobias Allander, Björn Andersson, Lars Arvestad, and Joakim Lundeberg. 2010. Classification of DNA sequences using Bloom filters. *Bioinformatics* 26, 13 (2010), 1595–1600.
[54] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 16:1–16:14.
[55] Maysam Yabandeh. 2017. *Partitioned Index/Filters*. https://rocksdb.org/blog/2017/05/12/partitioned-index-filter.html.
[56] Benjamin Zhu, Kai Li, and R Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*. 1–14.