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 fully featured filters (those that support deletion and merging) do not currently have a clear winner in terms of performance. Vector quotient filters are the most performant filters by far in insertion throughput, while cuckoo filters edge out vector quotient filters in terms of both random and successful query throughput. The result is that tradeoffs have to be considered and time has to be spent on deciding which particular filter design best fits an application.

In this paper, we present the *partition quotient filter* (PQF). Its design is similar to that of the vector quotient filter and prefix (quotient) filter (all ultimately based on the quotient filter). Similar to the prefix filter, it uses a two-level hierarchy to store quotients: most keys are sent to the frontyard and overflows go into the backyard. In the frontyard, there is only a single bucket (cache line) where a quotient can end up, which is responsible for the increased performance over other dynamic filter designs that have to access two cache lines for each operation. Keys are sent to the backyard using a two choice mechanism (similar to the vector quotient filter), and the innovation that enables us to support deletions is that the backyard locations are dependent purely on the frontyard location, with no rehashing of the quotient performed.

We show that the partition quotient filter is faster than all other fully dynamic filter designs. Additionally, in some scenarios it even approaches the performance of insert-only filters, showing the potential for a single unified filter design that could remove any tradeoffs between supporting merges and deletions and speed.

CCS CONCEPTS

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

KEYWORDS

Dictionary data structure; filters; membership query

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3452841

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, 12 pages. https://doi.org/10.1145/3448016.3452841

1 INTRODUCTION

Filters, such as Bloom [7], quotient [39], and cuckoo filters [28], 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, 9, 12, 17, 18, 22, 23, 26, 31, 33, 42, 46, 48–50, 52]. For example, in storage systems, filters are used to summarize the contents of on-disk data [5, 14, 19–21, 45, 47, 50]. In networks, they are used to summarize cache contents, implement network routing, and maintain probabilistic measurements [12]. In computational biology, they are used to represent huge genomic data sets compactly [2, 3, 17, 37, 38, 40, 42, 48].

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 [21, 45, 51]. For example, Monkey [21] uses an optimized allocation scheme to minimize the size of filters in-memory. PebblesDB [45] 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 [20].

Modern filters, such as quotient, cuckoo, and Morton [11] 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 [15]. 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

Filter	Num bits for n items
Bloom filter [7]	$1.44n\log(1/\varepsilon)$
Quotient filter [39]	$1.053(n\log(1/\varepsilon)+2.125n+o(n))$
Cuckoo filter* [28]	$1.053(n\log(1/\varepsilon)+3n+o(n))$
Morton filter [11]	$1.053(n\log(1/\varepsilon)+2.5n+o(n))$
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.

quotient filters [39] are based on Robin Hood hashing [16], and cuckoo and Morton filters are based on cuckoo hashing [36].

All these filters slow down as they are filled, because they experience more collisions. This shows up clearly in Figure 5a, which shows instantaneous insertion throughput as a function of 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.

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 5a, which shows that most modern filters exhibit

[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.

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 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 [7] 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 [15]. 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, 8, 13, 22, 29, 34, 43, 44]. The counting Bloom filter (CBF) [29] 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 [43] provides better cache locality than the standard Bloom filter but does not support deletion.

The quotient filter (QF) [6, 24, 25, 35] uses a new, non-Bloom-filter design. It is built on the idea of storing small fingerprints via Robin Hood hashing [16]. It supports insertion, deletion, lookups, resizing, and merging. The counting quotient filter (CQF) [39], 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 [32].

The cuckoo filter [28] 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 [11] 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 PARTITION QUOTIENT FILTER

The following is an overview of the partition quotient filter:

- The filters is organized into a frontyard and a backyard.
- In the frontyard, single-choice hashing is employed to reduce the number of operations and cache misses necessary.
- Using the frontyard location only, two locations are selected in the backyard to be used for two choice hashing. Remainders remain the same when forwarded to the backyard to support deletion.
- Mini-filters, as in vector quotient filter, are used to efficiently store several remainders in one cache line.
- Remainders in mini-filters are sorted by their "mini-bucket." This is done to be able to efficienty tell whether a key needs to be forwarded to the backyard (for an insertion or query, as it were). Only keys with the largest mini bucket are sent to the backyard.

The main source of higher throughput compared to the vector quotient filter and other deletion only filters comes from the fact that only one cache line is accessed for the majority of operations, particularly for queries. Only roughly one tenth of keys end up in the backyard.

The frontyard causes no failures, as overflows are sent to the backyard. As the backyard is implemented using two-choice hashing, it would normally be necessary to simply have an extra $\Omega(\text{loglog}n)$ in each backyard bucket versus the expected capacity to ensure that the filter does not fail with high probability..

However, the main issue with this approach comes from the fact that potentially many keys are hashed to the same backyard location from the frontyard. However, empirically, this does not cause issues, at the very least when sufficiently many keys can fit in a single bucket. Notably, one of the most common sources of failures is from a single bucket overflowing to the extent that it fills both possible backyard buckets. Since the maximum expected overflow is $O(logN)$ (and in practice with a small constant value), this is mainly an issue for the 32 byte version of the filter, where backyard buckets have pretty low capacity for remainders.

However, as discussed later, this appears to simply reduce the maximum possible load factor by a little bit, which is a tradeoff many would take. If a higher load factor is necessary, then using the full 64 bytes for each bucket empirically shows very good scaling of maximum load factor with N. Clearly, for very large values of N (such as, for example $N = 2^48$), even these larger buckets would begin to cause problems, but it is reasonable to expect that cache line sizes may increase by the time such large filters would become necessary. To give further evidence, GPUs already make use of 128 byte cache lines.

3.1 Frontyard to Backyard Rehashing (or lack thereof) to Support Deletions

The prefix filter takes a simple approach to speeding up performance, essentially treating the filter as two different filters. When an item is inserted, it is inserted into the frontyard, and, if there is overflow, it is inserted into the backyard as if the backyard were the only filter there. That is, the backyard can be implemented as any filter, and an inserted item is simply rehashed and treated as any other key. However, this approach does not support deletions and greatly limits the usefulness of the filter.

In order to support deletions, when a key is sent from the backyard to the frontyard, the hash of the backyard key used in generating quotients and remainders must be identical (or at the very least bijective with the original hash). Otherwise, unless the backyard is implemented as single-choice, it is impossible to know which remainder to delete. If you rehash to two locations, and in each of the locations there is a match to the remainder, then it is impossible to know which of the two remainders to delete. There may have been another key that only hashed to one of the locations, and, by deleting the remainder at that location, false negatives are created for that key.

Additionally, even if there is a single choice in the backyard, there is the question of what to do with backyard buckets that need to be brought back. If some frontyard elements are removed from the filter, then backyard elements need to be brought back in order to maintain the invariant that the frontyard stores the smallest items by mini bucket location. Otherwise, it may happen that an element is removed from the frontyard bucket, and then another element is inserted into the same bucket with a mini bucket index higher than that of an element in the backyard. Thus the rule for knowing when to go to the backyard breaks down, as it is impossible to tell if going to the backyard is necessary. Even if the other problem were not there, this would kill any practical attempt to make such a two-tier filter work, as the backyard would always need to be checked and the time savings of this approach would be nullified.

Therefore, we use a simple function that, when combined with a few extra bits we store in the backyard, is bijective. This means that, upon deletion or reinsertion into the frontyard, we can simply invert the function to then put the remainder back into the frontyard. The function is parametrized by one variable: the ratio R of buckets in the backyard to the frontyard. A frontyard location is converted to one backyard location simply by chopping off the least significant few ($\lceil \log R \rceil$) bits and then storing them along with the remainder in the backyard bucket. The other backyard location is chosen by splicing out the next least significant bits. The least significant bits are then moved to the front of the key. This step is important, as just splicing out one of the two groups of least significant bits of the number will mean that range of frontyard buckets corresponding to a backyard bucket would be very small. That is, taking the graph G created by treating backyard buckets as nodes and creating an edge between a pair of them if they correspond to the same frontyard bucket would be disconnected: there would be connected components corresponding to each group of frontyard buckets with the same prefix and differening least significant digits. Moving the bits, G is a very similar graph to a De Brujin graph. Of note, it is important that the less significant bits are moved rather than just, for example, splicing out the most significant bits, as the upper bits can significantly deviate from uniformly at random. Assuming that the hash function for the frontyard is good, these two sets of bits that are spliced out should be uncorrelated.

Algorithm 1 FrontyardToBackyardHash (f) -> ((b_1,b_2),(r_1,r_2))

1: $b_1 \leftarrow \left\lfloor \frac{f}{C} \right\rfloor$
2: $x_1 \leftarrow (l - b_1 * C)$
3: $r_1 \leftarrow 2x_1$
4: $b_2 \leftarrow \left\lfloor \frac{b_1}{C} \right\rfloor$
5: $x_2 \leftarrow (l - b_1 * C)$
6: $r_2 \leftarrow 2x_2 + 1$
7: $b_2 \leftarrow b_2 + Rx_1$

Specifically, the following is the pseudocode for obtaining the backyard locations from the frontyard (s corresponds to whether this is the second possible location), where f is the frontyard location, R is (slightly larger than) the range of possible backyard buckets, C is the ratio of the number of backyard buckets to the number of frontyard buckets (consolidation factor), (b_1, b_2) are the two backyard locations returned, and (r_1, r_2) are the several bits that need to be stored along with the remainders in order to uniquely identify the original frontyard location (f) from the backyard locations. One additional bit needs to be stored as part of r_2 to indicate whether this was the first or second choice, which here is done as the least significant bit to simplify the expression:

3.2 Correctness of this Hashing Method

Due to the nature of the partition quotient filter, it is difficult to analyze the failure probability. As mentioned, assuming that all keys coming out of the frontyard are rehashed independently from the original key, there are no issues and correctness follows from two choice hashing. Even if full (quotient, remainder) pairs are hashed to new values for whatever filter the backyard uses, correctness does not appear to be a major issue, as very few (quotient, remainder) pairs appear multiple times in the frontyard (assuming no duplicate keys are inserted).

However, when there are two fixed locations for an entire bucket (not even quotient, but bucket, which is quotient minus the few bits that identify the mini bucket), analysis is more difficult. Additionally, the two hashes used in partition quotient filter are not independent, which potentially causes further issues. However, we give some intuition to our claim that our choice of hash function actually improves the potential of the filter over two functions that randomly hash the bucket index:

Consider the filter if there are no frontyard buckets ($N_f = 0$). Then keys are hashed to (backyard1, remainder) and (backyard2, remainder) pairs at once, where backyard2 is generated from backyard1 *and some extra bits* (corresponding to the snipped out bits of the frontyard bucket that would be simply stored in the backyard). Considering that these bits are (pseudo)independent and (pseudo)random (assuming a good hash function), this version of two choice greatly resembles that of vector quotient filters and cuckoo filters—in both, the alternate bucket is chosen as $b_2 = b_1 \oplus r$, using the (psuedo)randomness of the bits of the remainder for the second bucket. While not exactly the same, the second location in partition quotient filters here is chosen in a similar manner and should have very similar performance. However, two issues still remain: the ratio of the number of frontyard buckets to the number of backyard buckets is relatively small (8 for all implementations

tested), meaning the number of "extra (psuedo)random bits" used in the second hash is very small, and the frontyard exists.

In fact, using the cuckoo or vector quotient filter hash is insufficient. Taking r' to be the 3 bits corresponding to the frontyard identifier for the first backyard hash, assuming that r' is random means that removing the first 3 bits out of the last 6(without moving them to the front) for the second backyard bucekt should be equivalent to taking $b_2 = b_1 \oplus r'$, as the bits are random. In either case, this is to few bits to create a good filter design (see analysis given by cuckoo filter on failures). In fact, by accident this was used in the initial implementation of the partition quotient filter, and they failed at a significantly lower load factor (data was not collected at this stage, so exact numbers are unavailable).

Essentially, in this case (still ignoring the frontyard) the graph created by connecting two buckets associated with the same key/frontyard bucket is a disjoint union of many small connected components, giving very little opportunity for the effects of an unusually full bucket to "spread out" (see figure). Since only the last few bits of the hash are affected (in this case, just the last 3), the only possible other keys hashed to a backyard bucket differ in just the last few bits in the backyard bucket. No matter how many steps are taken, keys only interact with other keys with the same prefix (excluding the last few bits).

However, in the scheme used here, where these extra bits are (conceptually) shuffled to the beginning for the second hash, the graph is a De Brujin graph, except using a different base. In this case, overflows can "trickle" down to further away parts of the graph, which matches the intuition of why two-choice hashing works in the first place. Empirically, as will be seen in the evaluation section, this results in very good load factors. This graph has as high as possible a branching factor, which similarly means that overflows trickle away maximally.

Finally, the fact that there are frontyard buckets should have only a minor effect on the quality of the filter. The one issue is that the distribution of keys kicked from the frontyard is different to that of keys in general. Whereas keys in general follow a uniformly random distribution, keys that are kicked from the backyard follow a tail normal distribution. Roughly speaking, a frontyard bucket is expected to have k keys with roughly $\sqrt{k}$ standard deviation following a roughly normal distribution (assuming the buckets are large enough, which is not exactly the case). Thus, the overflow is expected to have $O(\sqrt{k})$ keys in the overflow with a $\sqrt{k}$ standard deviation, a significantly less well behaved distribution. This should mainly serve to make it so that there are larger clumps of keys with the same two choices than in the previous case. As it is possible to reduce the relative effect of these clump sizes by making larger backyard buckets with more frontyard buckets mapping to a single backyard bucket, this has the effect of reducing the effective ratio between frontyard and backyard buckets. Additionally, it means that backyard buckets need to have a minimum size in order to have two backyard buckets even be able to store the largest overflow of any frontyard bucket (a limit which the 32 byte buckets ride dangerously close to).

It may be surprising that it is possible to get such good performance out of so few bits used to choose the two backyard buckets, but, as this hash disperses keys far and evenly, it appears to be sufficient.

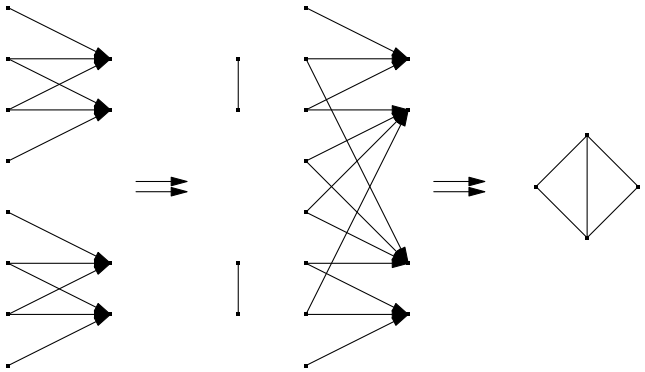


Figure 1: An example of how the cuckoo style two choice hash would work versus the system used in partition quotient filter using a backyard to frontyard ratio of 2 for simplicity. Observe that the cuckoo system results in disconnected components, which would significantly decrease maximum load factor.

Algorithm 2 Insert (x)

```
1:  (f,m,r) ← h(x)                                          ▷
    retrieving remainder r to store in minibucket m in frontyard bucket f
2:  over,(m′,r′) ←InsertFrontBucket(f,m,r)                  ▷
    returns if bucket overflowed; (r′,m′) is the overflow
3:  if over then
4:      ((b1,b2),(r1,r2)) ←FrontyardToBackyardHash(f)
5:      r1 ← r1 r′
6:      r2 ← r2 r′
7:      if full(b1) and full(b2) then
8:          return False
9:      end if
10:     if count(b1) < count(b2) then                       ▷ If b1 is emptier, insert into b1
11:         InsertBackBucket(b1,m′,r1)
12:     else
13:         InsertBackBucket(b2,m′,r2)
14:     end if
15: end if
```

4 PARTITION QUOTIENT FILTER OPERATIONS

This section describes the algorithms used to implement the insert, lookup and delete operations on a partition quotient filter. Bucket operations are as described in vector quotient filter with the exception of queries in the frontyard and removals from the backyard. Note that the bucket operations in vector quotient filters already sort the elements by their mini bucket index, as this is required for the structure to function at all, meaning that there needs to be no additional structure to implement this (with the slight exception of backyard removals). Frontyard bucket operations are generally distinguished from backyard bucket operations due to the fact that backyard buckets need to store larger remainders (for example, frontyard buckets may use 8 bit remainder operations, while backyard buckets use 12 bit operations).

Insert. Algorithm 2 shows the pseudocode for the insert operation. To perform an insert, the key is first hashed to determine the remainder r and quotient q. As in other filters that use the mini filter structure originating with vector quotient filter, this quotient is split into a (frontyard) bucket index f and mini bucket index m. m is (conceptually) compared to the other mini bucket indices in the

Algorithm 3 Lookup (x)

```
1: (f,m,r) ← h(x)   ▷ retrieving remainder r, minibucket m, frontyard bucket f
2: status ← QueryBucket(f,m,r)
3: if status == Present then
4:     return True
5: else if status == PotentialBackyard and full(f) then ▷ minibucket was at least as large as the largest minibucket of a remainder in the frontyard, meaning need to search backyard
6:     ((b1,b2),(r1,r2)) ←FrontyardToBackyardHash(f)
7:     r1 ← r1 r ▷ Concatenate r′ with r1 (r2) to get first (second) backyard remainder
8:     r2 ← r2 r
9:     return (QueryBucket(b1,m,r1) == Present) || (QueryBucket(b2,m,r2) == Present)
10: end if
11: return False
```

Algorithm 4 Remove (x)

```
1: (f,m,r) ← h(x)   ▷ retrieving remainder r, minibucket m, frontyard bucket f
2: if full(f) then ▷ If f is full, x may be in the backyard or may need to retrieve an element from the backyard
3:     e ← RemoveFromFrontBucket(f,m,r) ▷ e stands for key was eliminated (true) or not present (false)
4:     ((b1,b2),(r1,r2)) ←FrontyardToBackyardHash(f)
5:     ((b1,b2),(r1,r2)) ←FrontyardToBackyardHash(f)
6:     if e then ▷ (m,r) was removed from frontyard
7:         (m1,r1′) ← minElem(b1,r1) ▷ Grab element with the smallest minibucket index coming from the frontyard bucket determined by r1
8:         (m2,r2′) ← minElem(b2,r2)
9:         if m1 < m2 then
10:            RemoveFromBackBucket(b1,m1,r1r1′)
11:            InsertFrontBucket(f,m1,r1′)
12:        else
13:            RemoveFromBackBucket(b2,m2,r2r2′)
14:            InsertFrontBucket(f,m2,r2′)
15:        end if
16:        return True
17:    else ▷ (m,r) not found in frontyard
18:        if RemoveFromBackBucket(b1,r1r′) then
19:            return True
20:        else if RemoveFromBackBucket(b2,r2r′) then
21:            return True
22:        else
23:            return False ▷ Key was not found anywhere
24:        end if
25:    end if
26: else
27:    return RemoveFromBucket(f,m,r) ▷ Simply remove x from the bucket
28: end if
```

remainder store to determine where remainder r should be stored. r is inserted (using AVX512) into the remainders portion of the bucket using the order of m in the metadata as an index.

If the bucket overflows, then we keep track of which mini bucket index m' (possibly equal to m) and remainder r' overflowed (so m' is the max of the highest mini bucket index previously in the bucket and m). The frontyard bucket index f is then hashed to (b_1, r_1) and (b_2, r_2) as given previously. Then $(m', r_i r')$ is inserted into whichever bucket at index b_i is emptier, with a failure occuring if both buckets are full. $r_i r'$ is simply a concatenation of the two pieces of data needed to identify a remainer in the backyard—the frontyard remainder, the bits chopped off and which bits were chopped off (r_i).

Lookup. Algorithm 3 shows the pseudocode for the lookup or query operation. Lookup proceeds by obtaining (f,m,r) as in inserts and checking if the remainder is present in the frontyard bucket. If the remainder is found, then the query is done and returns true. Otherwise, it may be necessary to check the backyard when the status specifies PotentialBackyard. This check is a slight modification of the typical bucket query operation: when the frontyard bucket is full and the minibucket index of the queried key is at least as large as the largest minibucket index of a remainder in the frontyard bucket, it is possible that the key is in the backyard, as the backyard stores keys with the same or larger mini bucket indices as the largest in the frontyard.

Remove. As in the other operations, removal first attempts to remove the key from the frontyard. If the frontyard bucket was not full, then the removal is finshed whether or not the key was found in the bucket, as there is guaranteed to be nothing in the backyard.

If the key was in fact present in the frontyard bucket and the frontyard bucket is full (meaning there *may* have been keys sent to the backyard from this bucket), then a key with the smallest mini bucket index in the backyard is moved back to the frontyard. This is done for a single bucket by querying which elements in the bucket match the prefix given by r_1 (or r_2 for b_2) and then using one ctzll instruction to see what is the index of the smallest matching key in the remainders. This index is then used to identify which mini bucket the matching key belongs to. After obtaining the key with the smallest minibucket index in the two backyard buckets, it is removed from its corresponding bucket and inserted into to the frontyard.

If the key was not present, then it is first attempted to be removed from the first backyard bucket, and, if not found, then it is attempted to be removed from the second backyard bucket. If it is still not found, the removal fails (the key was either removed twice or never inserted).

5 SPACE ANALYSIS

We now do a quick analysis of the space usage of the partition quotient filter and compare it to other filters, after which we present the space usage of the filters in practice. Note that the space usage of all these filters is relatively similar.

The main factor in determining space usage is the ratio of the size of the frontyard to the size of the backyard. Call this ratio C. Estimating the asympotics of this ratio is nontrivial in the case of this filter, so we just note that empirically the best performing (in terms of maximum load factor supported) filters we implemented set $C = 8$ for both the 32 byte bucket and the 64 byte bucket configurations. It is reasonable to assume that larger buckets would enable larger values of C, if only because the number of keys that overflow a frontyard bucket scales as the square root the size of the bucket (for larger bucket sizes), meaning that the ratio of the overflow to the size of the bucket is inverse square root (and thus C should be roughly on the order of the square root of the bucket size).

Optimally, the frontyard should use on the order of roughly $1.914 + \log(1/\varepsilon)$ bits per key (the 1.914 coming from the optimal configuration of the mini filter). The backyard needs to store an additional $1 + \left\lceil \log\left(\frac{1}{C}\right) \right\rceil$ bits per key, in order to store whether it is the first choice bucket and what the bits that were cut from the frontyard location were. For our configurations with $C = 8$, this expression simplifies to 4. Therefore, the optimal average bits per key is $1.914 + \log(1/\varepsilon) + \frac{4}{8} = 2.414 + \log(1/\varepsilon)$. In our implementations, we use configurations that yeild around $2 + \log(1/\varepsilon)$ bits for the mini

filters in the frontyard, so the average space consumption per key is just slightly higher at $2.5+\log(1/\varepsilon)$.

The filter using a bucket size of 32 achieves a load factor of $75-85\%$ (for reasonable values of N), and, taking the lower figure, we get that it uses $\frac{4}{3}(2.5+\log(1/\varepsilon))$. At $\log(1/\varepsilon)=8$, this is a space usage of 14 bits per key, although, due to the lower fill of the filter, the actual space efficiency is a little better than $\frac{8}{14}\approx 0.571$ at 0.591. The filter using a bucket size of 64 achieves a load factor of over 0.9 for all ranges of N tested, which leaves a space usage of 11.67 bits per key, so this filter should be used when space is at a premium. This space usage is very comparable to other filter designs. Once again, the filter is slightly underfilled here, and the real space efficiency of it is 0.718 at 90% fill, rather than the $\frac{8}{11.67}\approx 0.686$ this would predict.

6 IMPLEMENTATION AND OPTIMIZATION

The main constraint on the buckets/minifilters in both the frontyard and the backyard is that they need to fit within a single cache line to enable fast operation. The two main components of The minifilters are the array of remainders and the metadata. The metadata uses one bit per remainder and one bit per minibucket to separate out the remainders with. Therefore, letting the number of remainders in a bucket be N_f for the frontyard and N_b for the backyard, the number of minibuckets be B, and the number of bits used in a remainder (roughly corresponding to the log false positive rate) be R. Then the space usage of a frontyard bucket is $B+N_f(R+1)$.

Additionally, the backyard buckets need to store which frontyard bucket the remainder came from out of the posssibilities. In the case of our filter configurations, there are always at most 16 possiblities, so this uses four bits. Therefore, the space usage of a backyard bucket is $B+N_b(R+5)$.

One further constraint that could lead to faster filters would be to use just one machine word for the metadata. As the metadata uses the LZCNT instruction, this maximum size is 64 bits. This is naturally the case when the size of the remainders is 16 bits—in the frontyard, we set $N_f=28$ and $B=36$, which uses precisely 64 bits, using the other 56 bytes of a cacheline for the remainders. In the backyard, $N_b=22$. This filter will be known as $PQF16$ in the evaluations.

In the case when the size of the remainders is 8 bits, there is a tradeoff to consider: do we use 64 byte buckets, leading to a better load factor and space usage, or do we use 32 byte buckets, which then have the metadata fit in a single machine word? In the first option, some possible configurations are $N_f=53$, $B=51$, and $N_b=35$ ($PQF8-53$); and $N_f=62$, $B=50$, $N_b=34$ ($PQF8-62$) (which offers a somewhat lower false positive rate and higher space efficiency but very slightly lower capacity).

In the second option, the seemingly fastest configuration is to use $N_f=22$, $B=26$, $N_b=18$ ($PQF8-22$), with similar tradeoffs with having more minibuckets per bucket possible. In addition, it is possible to alleviate the load factor issues by making buckets 64 bytes while frontyard buckets are 32 bytes (an example configuration tested is same as $PQF8-22$ but with $N_b=37$), but this means that a larger fraction of the remainders will be in the backyard, alleviating the advantage of partition quotient filters.

7 EVALUATION

In this section, we evaluate our implemention of the partition quotient filters (PQF) in different comfigurations. We compare them primarily against two similar filter data structures: vector quotient filters (VQF) and prefix filters (PF).

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 [6, 11, 28, 39]. We also report the aggregate throughput performance which is the performance of the filter going from scratch to 95% (or 90%) load factor.

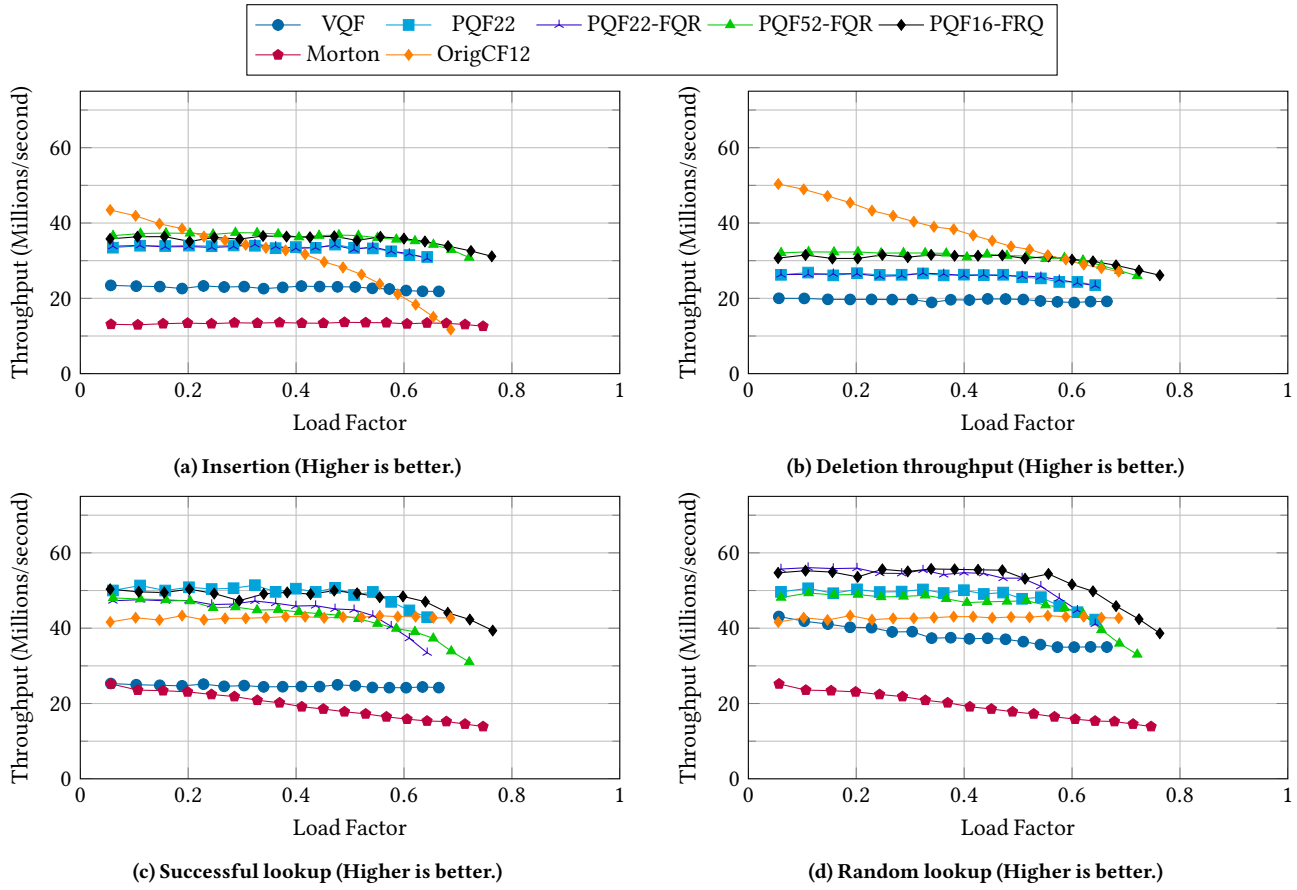
One challenge we face is that the filters do not all support the same false-positive rates. For example, the cuckoo filter implementation [27] supports only 2, 4, 8, 12, 16, and 32-bit fingerprints. The false-positive rate can further be tweaked by a small amount by adjusting the block size, but making 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 [10] 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

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



(a) Insertion (Higher is better.)

(b) Deletion throughput (Higher is better.)

(c) Successful lookup (Higher is better.)

(d) Random lookup (Higher is better.)

Figure 2: Insertion, deletion, and lookup performance of different filters in RAM for different load factors. Averaged over 5 runs.

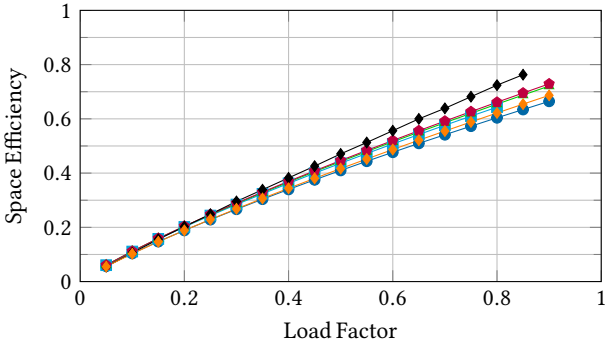


Figure 3: Space effiency (Higher is better.)

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 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 $\approx 20\%$ 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

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.

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 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.

7.2 In-RAM performance

Figure 2 shows the in-RAM performance of data structures.

Our performance results for the Morton filter are worse than the main experimental results from the Morton filter paper [11]. 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 [11] 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 4 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 **??**.

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.

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.

7.4 Filter Failures

We ran some experiments to test whether the filter design is indeed correct empirically. To do that, we inserted into different filter configurations and designs until each failed, and recorded the load factor at which this failure occured. This was done over 1000 runs. Figure 5 shows the trend over different filter capacities of the load factor at which failure occurs, comparing several different configurations versus the vector quotient filter.

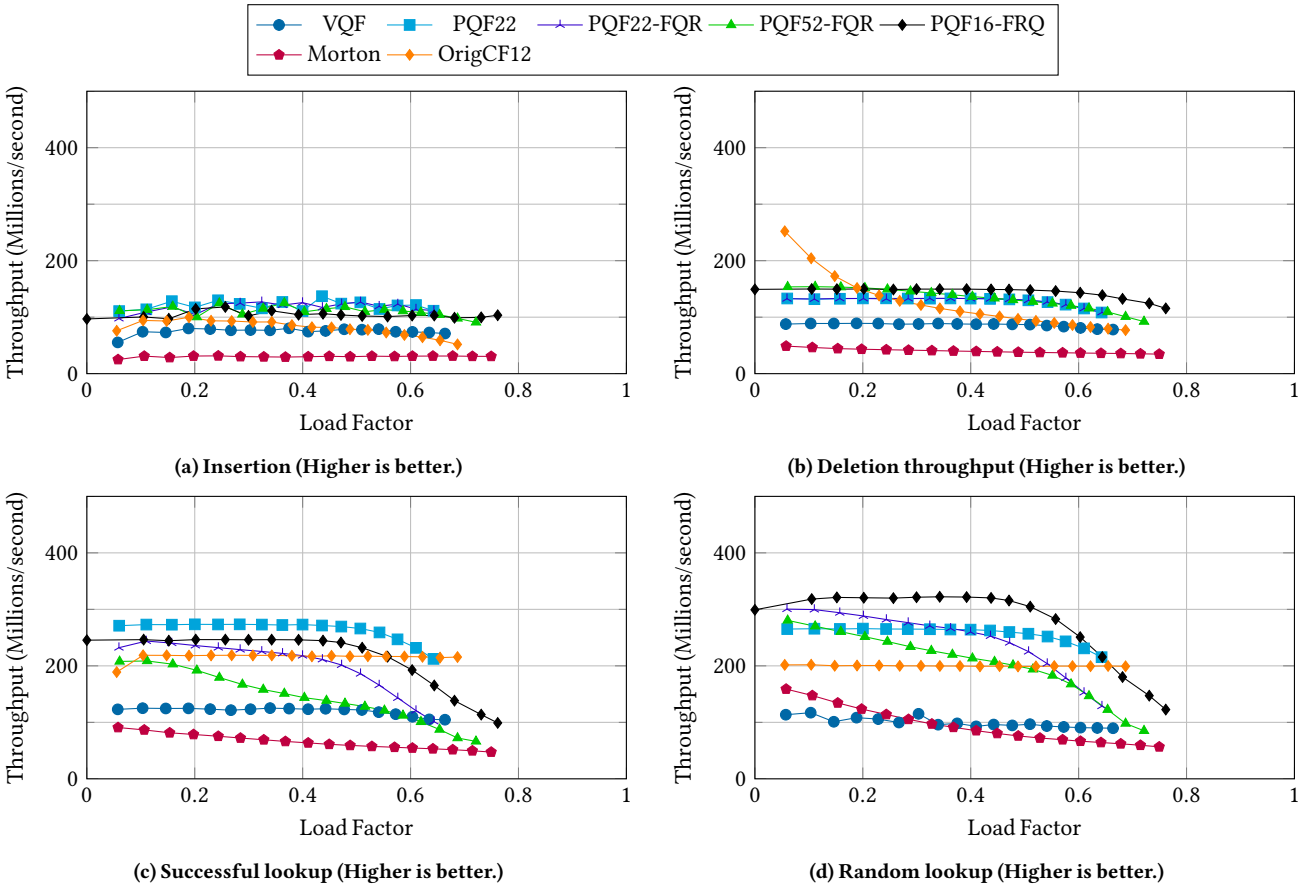
As can be seen, PQF52 outperforms the default configuration of VQF (which has more failures due to the shortcut done to improve insertion throughput) in failures, and has a very flat graph. PQF22 has failures a lot earlier in exchange for its faster query performance, which is to be expected due to the significantly smaller bucket sizes.

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, 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.



(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 L3 cache. Averaged over 5 runs.

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 [19, 30, 41]. 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.

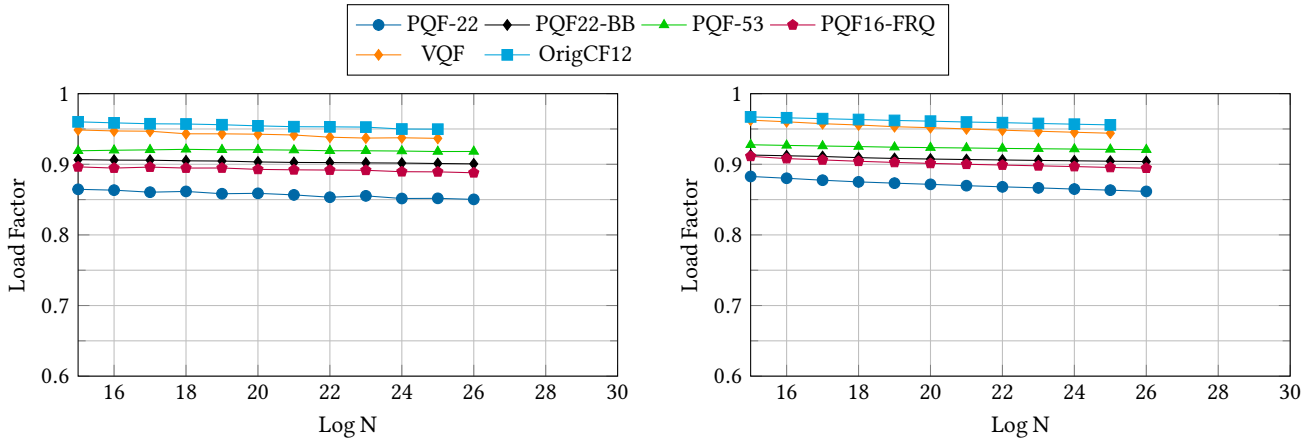
[6] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kaner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment* 5, 11 (2012).

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

[8] 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.

[9] 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.

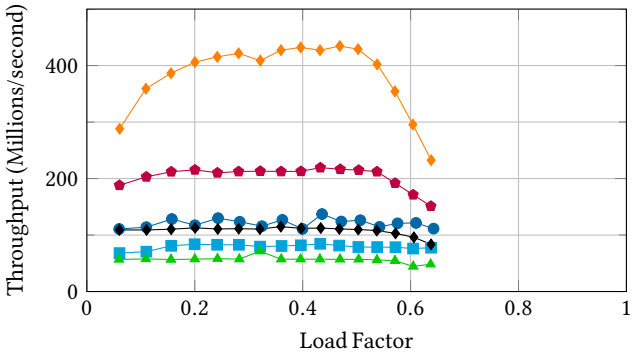
[10] Alex D Breslow. 2018. Morton Filter source code in C++. https://github.com/AMDComputeLibraries/morton_filter. [Online; accessed 19-July-2020].



(a) 1pct worst failures.

(b) Median failures.

Figure 5: Plotting the load factor at which different variants of the filter fail versus the (log of the) size of the filter. Done with 1000 runs, so 1pcts are 10th (might be 11th, check this) order statistics.



(a) Insertion (Higher is better.)

??

Figure 6: Threading testing

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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)*.

[24] 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

[25] 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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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).

[47] RocksDB [n. d.]. RocksDB. https://rocksdb.org/, Last Accessed Sep. 26, 2018.

[48] Brad Solomon and Carl Kingsford. 2016. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology* 34, 3 (2016), 300.

[49] 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.

[50] 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.

[51] Maysam Yabandeh. 2017. *Partitioned Index/Filters*. https://rocksdb.org/blog/2017/05/12/partitioned-index-filter.html.

[52] 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.