Breadcrumb Filters: Fast Fully Featured Filters

ANDREW KRAPIVIN, Carnegie Mellon University, USA
AADITYA RANGARAJAN, University of Utah, USA
ALEX CONWAY, Cornell Tech, USA
MARTÍN FARACH-COLTON, New York University, USA
ROB JOHNSON, VMware Research, USA
PRASHANT PANDEY, Northeastern University, USA

Enumerable filters are the gold-standard for full-featured filters: they support insertions, deletions, merging; they can support associated values, such as counts; and like traditional filters they support queries with a small probability of false positives. When designing a system that uses filters, high-performance enumerable filters are required to simplify system design and improve overall performance, compared to systems that must work around the limitations of traditional filters that are not full featured. The vector quotient filter (VQF) is the state of the art enumerable filter.

For limited-functionality filters, blocked Bloom filters (BBF) and the prefix filter (PF) are state of the art and offer tradeoffs. Both support insertions and queries but not deletions or merging. BBFs are faster for insertions and queries but use more space than PFs and VQFs. For small-space filters, the state-of-the-art suggests a tradeoff: PFs are faster than VQFs but VQFs are enumerable, which makes them a favorable candidate to be integrated in applications. Given these tradeoffs, we are left with the following question: *Do we need to give up features in order to achieve the highest performance?*

In this paper, we present the ***breadcrumb filter*** (BCF), a full-featured enumerable filter. For insertions, the BCF is up to 34% faster than VQF, 3.2× faster than cuckoo filter, 19.6% faster than PF. For queries, the BCF achieves competitive performance, outperforming the VQF while matching the cuckoo filter. At the same time, it achieves higher space efficiency than VQF, prefix, cuckoo filter and BBF.

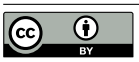
We conclude that the choice of filter is now simplified: if additional features beyond insertions, such as deletions and counting, are required, or if minimizing space is crucial, the BCF offers the best performance. On the other hand, if only insertions are needed and space is not constrained, the BBF is the right choice.

CCS Concepts: • **Theory of computation** → **Bloom filters and hashing**; *Data structures design and analysis*; • **Information systems** → *Point lookups.*

Additional Key Words and Phrases: Filters; Dictionary data structure; Databases

ACM Reference Format:
Andrew Krapivin, Aaditya Rangarajan, Alex Conway, Martín Farach-Colton, Rob Johnson, and Prashant Pandey. 2026. Breadcrumb Filters: Fast Fully Featured Filters. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 15 (February 2026), 28 pages. https://doi.org/10.1145/3786629

Authors' Contact Information: Andrew Krapivin, andrew@krapivin.net, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; Aaditya Rangarajan, aaditya2200@gmail.com, University of Utah, USA; Alex Conway, me@ajhconway.com, Cornell Tech, New York, New York, USA; Martín Farach-Colton, martin@farach-colton.com, New York University, New York, New York, USA; Rob Johnson, rob@robjohnson.io, VMware Research, USA; Prashant Pandey, p.pandey@northeastern.edu, Northeastern University, USA.



ACM 2836-6573/2026/2-ART15
https://doi.org/10.1145/3786629

1 Introduction

Filters, such as Bloom [13], cuckoo [34], morton [15], quotient [53], ribbon [31], xor [40], and vacuum filters [66], are data structures for compactly representing a set S of items. They support membership queries (or point queries), i.e. whether an item x is in set S, and save space by representing S approximately: a membership query may return "Yes" for some ε fraction of items not in S. Filters are widely used in databases, storage, networks, computational biology, and numerous other applications [4, 16, 21, 29, 36, 41–43, 50, 52, 54, 56–58, 61, 64, 67, 69, 70, 72].

The Bloom filter [13], is an ***incremental filter***, meaning that it only supports two operations: insertions and membership queries. Although the Bloom filter is widely deployed, its limited functionality leads to many inefficiencies and complications, because systems often need a richer set of operations [14, 19, 22, 32, 53, 71].

For example, applications in computational biology and stream processing build multiple, smaller filter instances and merge these filters to build an aggregate filter [1–3, 57]. These applications require the filters to support efficient *merging*. Other applications in computational biology associate counts with entries in the filter to efficiently represent key abundances and weighted graphs (de Bruijn graphs) [52, 54]. These applications require the filter to efficiently associate the *count* with each key. Key-value stores such as RocksDB [24, 28, 62] and SplinterDB [22, 23] employ filters to avoid looking into disk-resident partitions. However, SplinterDB accelerates queries by requiring the filter to associate a small *value* with each key to quickly identify relevant partitions to look for the key. Finally, applications in networking and data processing [46, 47] employ filters to represent a dynamic stream of keys and require the filter to support *deletions* to accurately represent the modifications in the set of keys. For more details please refer to a recent tutorial [56] on modern fully featured filters and applications.

Enumerable filters. *A highly desirable feature for functionality in filters is enumerability.* Enumerable filters associate a fingerprint to each key and store the fingerprints compactly in a hash table. They support the ability to list or iterate over all these fingerprints. This enables them to support *merging, deletions, associating values,* and *counting*. As a result, many systems for the above applications employ enumerable filters to achieve the necessary functionality together with high performance. This has led to a push to make enumerable filters as efficient as possible [10, 25, 26, 32, 37, 45, 46, 53, 55, 63, 65, 68].

The state-of-the-art enumerable filter is the vector quotient filter (VQF) [55]. The VQF achieves consistent high performance independent of the fullness of the filter. In contrast, previous state-of-the-art enumerable filters, such as the quotient filter [53], experience significant degradation in insertion performance as they near their capacity.

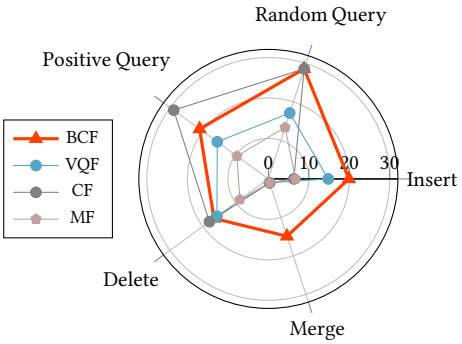
Feature and performance tradeoff. There has been a separate line of research on maximizing the performance of filters if one is willing to give up full functionality. In fact, it has been suggested that a full feature set limits the performance of filters [33]. *The main point of this paper is to explore this tradeoff between functionality, space efficiency and throughput.*

The state-of-the-art for insertion and query performance is the blocked Bloom filter (BBF) [60]. However, blocked Bloom filters are incremental and have poor space efficiency [1]. The BBF achieves lower space efficiency (62%) compared to the best enumerable filters (72% for CFs [35] and 79% for Morton filters [15]).

Recently, the prefix filter (PF) [33] was introduced with the goal of achieving a more space-efficient incremental filter. The prefix filter (PF) has better space efficiency (69%) than the BBF, but substantially worse throughput. The PF result is a mixed bag: the PF is about 38% faster than the VQF, but the VQF is enumerable and supports deletes. This suggests the following open problem:

For space-efficient filters, do you have to sacrifice performance to support enumerability?

[1] Space efficiency of a filter is the ratio of its size and the information-theoretic lower bound, i.e. the space efficiency of a filter instance of size S bits with false positive rate ε is $\frac{n \log 1/\varepsilon}{S}$.



Filter	Feature Support (Mops / sec)			
	Insert	Query (R,P)	Delete	Merge[2]
BCF	19.9	28.6, 21.0	16.6	14.9
VQF	14.8	17.1, 15.6	15.7	?[3]
CF	6.3	28.6, 28.9	18.0	?[3]
MF	6.7	13.4, 9.6	8.7	?[3]
PF	22.8	34.3, 25.4	✗	✗
BF	52.5	53.9, 55.3	✗	✗

Fig. 1. We present the performance of different filters in two different ways. For insertions (deletions), we fill (empty) the filters up to their maximum load factor, measure the time and divide by the number of keys. For queries, we measure the performance at maximum load for both positive queries (P) and uniformly randomly (R) drawn from the universe. Each filter has 2^{30} slots. Note that the BCF, VQF, CF, and MF are fully-featured, supporting insertions, deletions, and queries, whereas PF and BF only support insertions and queries. The BCF is the only filter out of these to implement merges, and it does so in a bandwidth efficient way. (Left) A radar plot depicting the performance of the fully featured filters on different operations. (Right) A table including performance of both fully-featured (top) and non-fully-featured filters (bottom).

Our Results. We introduce the ***breadcrumb filter (BCF)***, an enumerable filter that achieves both high performance and strong space efficiency. To address the previously outlined open question, we begin by comparing BCF with the prefix filter. Our results show that BCF supports insertions up to 19.6% faster, queries up to 22% faster, and offers better space efficiency overall.

While it remains possible that even faster, space-efficient non-enumerable filters could be developed in the future, our findings suggest that—at least for now—enumerability can be achieved with no additional cost in space efficiency.

Figure 1 summarizes our performance comparisons with space-efficient filters. For insertions, the BCF achieves 19.9 Mops/sec, which is 1.3× faster than VQF (14.8 Mops/sec) and approximately 3× faster than CF and MF (6.3 and 6.7 Mops/sec, respectively). For queries, the BCF achieves competitive throughput (28.6 Mops/sec for random queries, 21.0 Mops/sec for positive queries), outperforming VQF and MF while matching CF. For deletions, the BCF (16.6 Mops/sec) performs comparably to VQF (15.7 Mops/sec) and is approximately 2× faster than MF (8.7 Mops/sec). Importantly, the BCF is the only fully-featured filter among these to implement bandwidth-efficient merging (14.9 Mops/sec). The BCF achieves higher space efficiency than VQF, CF, and BBF, and slightly lower than the Morton filter.

We conclude that the blocked Bloom filter is ideal if only insertions and queries are needed without space constraints, whereas the BCF is the appropriate choice when functionality, performance, and space efficiency are all required. Among fully-featured filters, the BCF comes closest to this ideal, offering the best combination of functionality, high performance, and space efficiency.

Why is enumerability hard? BCFs, PFs, and various other filters use a related set of tools derived from the hash table literature. Fast, space-efficient hash tables that support deletions (and re-insertions) have proven difficult to design. State-of-the-art solutions periodically rebalance parts

[2]While we did not evaluate its throughput on insertions, deletions, and queries due to poor performance in previous comparisons, the counting quotient filter (CQF) is one of the few filters to implement merges. It has a merge throughput of 2.92 million keys per second, so the BCF has >5x merge throughput.

[3]By their design, VQF, CF, and MF are enumerable filters (or could potentially be modified to be enumerable). As enumerability enables merging, it should therefore be possible to implement merging for these filters. However, the BCF is the only one of these filters to implement merging, and it is designed to do so in a bandwidth efficient way.

of the data structure, for example, between a front-yard and backyard, using different hash functions. In filters, this is much more difficult, because filters store only fingerprints, rather than entire keys, and lose the ability to rehash an item with another hash function.

One of the main challenges in achieving performance, space efficiency, and functionality together has been how filters handle collisions. All current enumerable filter designs use collision resolution schemes that require insertions to access $1+\Omega(1)$ cache lines. For example, the quotient filter (QF) [53] was one of the first enumerable filters and it resolves collisions using Robin Hood hashing [18]. However, this leads to clustering, and as a result can cause operations to access too many cache lines both in the worst case and in expectation [20]. The Cuckoo filter (CF) [34] places items in one of two memory locations, reducing the number of memory accesses for queries down to 1.5 in expectation, but insertions may still require many memory accesses due to Cuckoo kick-out chains. Furthermore, insertions in both the QF and CF slow down as they fill, forcing users to choose a tradeoff between getting the maximum speed or maximum space efficiency offered by each structure.

The VQF improved this further, placing items in one of two cache lines without requiring kick-outs on insertion, therefore guaranteeing 1.5 expected memory accesses for most operations and 2 in the worst case, no matter how full the filter is. This eliminated the tradeoff between speed and space efficiency, but still leaves open the question: can we build an enumerable filter that accesses only a single cache line for almost all operations? Although going from two cache line accesses to one cache line access may seem like a small change, it is necessary for further performance improvements.

Until now, cache line accesses have been a significant difference between insert-only and enumerable filters. Insert-only filters, *i.e.* those that do not support deletions, have been able to get operations down to a single memory access. The blocked Bloom filter is guaranteed to do so, and is very fast as a result, but, as noted above, incurs larger space usage than other modern filters. The prefix filter resolves collisions using a front-yard/backyard hash table and, as a result, most operations require access only to the front-yard, yielding performance even better than the VQF in their benchmarks. However, their design does not support deletions because, during deletions of items in the front-yard, they would need to move items from the backyard to the front-yard, but their scheme inherently does not support doing so. In fact, they speculate that deletions are a hard obstacle to designing fast and space-efficient filters.

Our technical contribution. We present a new enumerable filter, the breadcrumb filter (BCF), that requires $1 + o(1)$ cache line accesses per operation, breaking through the $1 + \Omega(1)$ cache line barrier of previous enumerable filters, which explains its strong performance compared to VQFs. Moreover, the BCF has the very desirable property that performance is largely independent of how full the filter is, enabling users to enjoy both high performance and high space efficiency simultaneously.

The BCF uses a front-yard/backyard hash table design, with a large front-yard that uses single-choice hashing and buckets that fit on a single cache line. Nearly all of the items are stored in the front-yard, which means that nearly all operations need to access only a single cache line. The much smaller backyard uses minimum-of-two-choice hashing in order to achieve high occupancy while incurring at most $O(1)$ cache line accesses per operation. This design is inspired by iceberg hashing [7, 51], but requires substantial modification to work in the filter setting.

The BCF uses the prefix filter's "prefix invariant" to avoid accessing the backyard for almost all queries, even negative queries. The prefix invariant guarantees that a front-yard bucket contains the smallest fingerprints that have hashed into that bucket. Maintaining the prefix invariant during deletions is challenging because, if we delete a fingerprint from a front-yard bucket, then we may need to promote a fingerprint from the backyard to the front-yard. The PF has no promotion mechanism, and hence does not support deletions.

We propose ***Compactly Representable Uniform Map Back (CRUMB) hashing***, a new hashing design that allows us to perform backyard-to-front-yard promotions, enabling us to maintain the

prefix invariant while supporting full enumerability, including deletions. The backyard uses standard minimum-of-two-choice hashing [55] but with the following key modifications to enable promotion. In order to make it feasible to find elements for promotion to a front-yard bucket, all items that hash to the same front-yard bucket must also hash to the same two backyard buckets. As a result, during a promotion, we need to search only two backyard buckets. Furthermore, CRUMB hashing ensures that each of the two maps (one for first choice, one for second) from a front-yard bucket to a backyard bucket is ***uniform***, i.e. *exactly* C front-yard buckets map to each backyard bucket, which means that we can compactly represent a back-pointer from each backyard item to its front-yard bucket by storing a $\log C$-bit "crumb" with the item as well as a single bit for whether the first choice or second choice was made. Since the backyard is much smaller than the front-yard, these crumbs do not significantly affect space efficiency.

Like other filters that reduce the independence of the hash functions used for two-choice hashing, such as cuckoo and vector quotient filters, breadcrumb filters have some probability of failure. We empirically evaluate the failure probability of the BCF and find that the BCF fails about 1% of the time when filled to 92% occupancy, whereas the cuckoo and vector quotient filters can get to around 95% or 93% occupancy, respectively, with a 1% failure rate. Thus, for a given failure probability, the BCF is about 3% less space efficient than the cuckoo or vector quotient filter but up to 70% faster.

Thus, BCF achieves faster insertions than the best enumerable filter and faster queries than the best query filter, all without increasing space or failure probability. This is enabled by a novel front-yard/backyard hashing scheme that allows almost all operations to access a single cache line.

2 Background

In this section, we first examine the designs of various fingerprint filters. We then discuss the front-yard/backyard hashing approach and the challenges associated with adapting this hashing technique for existing filters.

Enumerable filters are ***fingerprinting*** filters, i.e. they store a set S of items by storing a multiset of fingerprints $h(S) = \{h(x) \mid x \in S\}$, where h is a hash function. The quotient filter is the classic example of a fingerprinting filter [11, 30, 48]. Fingerprinting filters generally store fingerprints space-efficiently by using a technique called ***quotienting***, i.e. they divide a fingerprint $h(x)$ into a ***quotient*** $q(x)$ and a ***remainder*** $r(x)$ [44]. The *quotient* is used to determine a location (e.g., a bucket) within the filter, and the *remainder* is stored in that bucket. Thus the space per item is proportional to the size of the remainder $r(x)$ rather than the size of the fingerprint $h(x)$.

Ideally, each filter operation should access a single cache line. To achieve this, we partition the filter into buckets, sizing each to fit within a cache line, and store item fingerprints in their respective buckets using single hashing. However, some buckets may overflow, causing insertions to fail before all buckets are filled, reducing *space efficiency*. For instance, when targeting a false-positive rate of $1/64$, each bucket could hold approximately 64 items: 6 bits are needed for each fingerprint and 2 bits are needed for the metadata for each key, for a total of 8 bits per key and 64 keys per 512-bit bucket. Experimentally, averaged over 10 trials, a simulated single-choice hashing scheme with buckets of size 64 and 2^{30} slots overflowed at a load factor of 0.476. Larger fingerprints translate to smaller buckets and only exacerbate the issue.

Prior bucket-based filters have overcome the space efficiency problem in several ways. Many fingerprinting filters employ minimum-of-two-choice hashing [12] where they use two hash functions, h_0 and h_1, to achieve better load balancing across buckets. For each item in S, they store either $h_0(x)$ or $h_1(x)$. Examples include cuckoo [34], Morton [15], vector quotient [55], and vacuum filters [66].

Using two hash functions can help load balancing because each item can be placed in one of two buckets depending on the load. For example, the cuckoo filter can move an item between its two

buckets to balance the load. The vector quotient filter places each item in the emptier of its two buckets and uses sufficiently large buckets to ensure there is no overflow until the filter is nearly full.

The downside of using two hash functions is that many operations must now access two (or more) buckets, and hence two cache lines, limiting performance.

These filters are part of a longstanding tradition of adapting collision-resolution strategies from hash tables to filters. For example, the quotient filter uses Robin Hood hashing [18], cuckoo filters employ cuckoo hashing [49], and the VQF utilizes minimum-of-two-choice hashing [12]. However, adapting these schemes is not always straightforward. For instance, the cuckoo filter had to introduce the "xor trick," which compromises the independence of the two hash functions, to ensure correct deletions.

Recently, there has been a revived interest in front-yard/backyard hash table designs [5, 9, 38, 39, 59], which divide the hash table into two subtables, called the ***front-yard*** and the ***backyard***. Typically, the front-yard is sized to hold almost all the items and uses single-choice hashing, i.e. each item hashes to a single bucket in the front-yard. To handle collisions, when too many items hash to the same front-yard bucket, items are allowed to overflow into the backyard. The current state of the art in this direction is iceberg hashing [6, 8, 51], which uses minimum-of-two-choice hashing in the backyard.

Front-yard/backyard constructions are promising for filter design because the front-yard buckets can be sized to fit on a cache line and the filter can be structured so that most operations need to access only the relevant front-yard bucket.

Unfortunately, front-yard/backyard filters proposed so far have not supported deletions. For example, Arbitman, et al. proposed backyard cuckoo hashing [5], and even sketched a filter based on their structure. Unfortunately, their construction does not support storing the same fingerprint many times, which is essential to supporting deletes in a fingerprinting filter. More recently, Even, et al. proposed the prefix filter [33], which enables almost all queries to avoid searching the backyard by ensuring that each front-yard bucket contains all the smallest fingerprints that hash to that bucket (hence the name—each front-yard bucket contains a prefix of the sorted list of fingerprints that hash to that bucket). Unfortunately, this may require actively moving items from the backyard to the front-yard after deletion of a fingerprint in the front-yard, but, since the backyard hash functions are independent of the front-yard hash function, it is not possible to find, or even identify, the fingerprint to move.

3 Breadcrumb Filter

The BCF design is based on two objectives:

(1) Each operation—insert, delete, query—should access only a single cache line most of the time, so that the average number of cache-line accesses per operation is as close to 1 as possible.
(2) The BCF should be enumerable.

3.1 Front-yard/backyard

The BCF accomplishes the first goal by employing a front-yard/backyard design. The front-yard uses single-choice hashing into F buckets of size N_f, where N_f is chosen so that buckets fit into a single cache line. When a front-yard bucket overflows, some items in the bucket are moved to the backyard of B buckets, which uses a minimum-of-two-choices hashing. The front-yard is much larger than the backyard, so most positive queries can be answered by examining a single front-yard bucket and hence by accessing only a single cache line.

Note that, as described so far, negative queries need to examine 3 buckets: the queried item's front-yard bucket and both of its backyard buckets. In the BCF, we reduce the cost of negative queries by ensuring that the front-yard bucket always contains the smallest N_f fingerprints that map to that bucket. We call this the ***prefix property***. This means that whenever a front-yard bucket overflows, we move

the largest fingerprint in the bucket to the backyard. This is similar to the technique employed in the prefix filter [33]. Note that we employ a variant of this policy in our implementation (see Algorithm 2).

3.2 Handling deletes

There are two challenges to supporting deletions: (1) maintaining the prefix property during a deletion, and (2) ensuring correctness, i.e. that we do not introduce false negatives.

The correctness issue is the same issue faced by the cuckoo filter designers. They solved this with the "xor trick", which ensures that, if two items collide in one bucket (i.e. have the same remainder and have one bucket in common) then they collide on both their buckets. This makes the two items completely indistinguishable, so deleting one will never cause a false negative for the other.

Since in our front-yard/backyard design, each item has 3 potential buckets, we need to generalize the cuckoo filter idea to what we call the ***total collision property.*** Let $S_x = \{(b_i, r_i)\}$ be the set of (bucket, remainder) pairs to which an item x hashes. The total collision property requires that for all x and y, either $S_x = S_y$ or $S_x \cap S_y = \emptyset$. Thus x and y are either indistinguishable or they do not interact at all. In either case, we can safely delete either one without causing a false negative for the other, which establishes correctness for deletions. To ensure this property, we calculate the backyard bucket remainder pairs from the front-yard bucket remainder pairs (ensuring that if two keys have the same front-yard pair, they have the same backyard pairs) and, along with the front-yard remainder, store enough additional information in each backyard remainder to identify the front-yard bucket (having the same backyard pair implies having the same front-yard pair). Naively, this uses a lot of space per backyard entry, but we will see in the next section how to fix this.

Maintaining the prefix property requires us to deviate from the iceberg hashing design. Iceberg hashing has a so-called "unmanaged" backyard, which means that items never need to be moved from the backyard to the front-yard. However, to maintain the prefix property, i.e. that each front-yard bucket always contains its N_f smallest fingerprints, we may need to move items from the backyard to the front-yard whenever we delete an item from the front-yard.

The primary challenge to moving items from the backyard to the front-yard is finding them: if each item moved from a front-yard bucket hashes independently to backyard buckets, then we would need to search the entire backyard to find an item to move back to the front-yard. We could try to solve this by keeping pointers from each front-yard bucket to its items in the backyard (e.g., using a linked-list), but the pointers would blow up the space used by our filter.

To make it easy to find items evicted from a given front-yard bucket, we enforce the ***front-yard collision property***: If two items hash to the same front-yard bucket, then they must hash to the same backyard buckets. This means that all the items moved from a given front-yard bucket will be placed into the same two backyard buckets. Thus, when we delete an item from the front-yard, we need to search only two backyard buckets to find any item that needs to be promoted back to the front-yard.

3.3 The front-yard/backyard map

The mapping from front-yard buckets to backyard buckets is crucial to the overall BCF design, and there are several constraints on this mapping, which we explain below.

Goals. The primary goal of the front-yard/backyard mapping is to spread overflows from the front-yard as evenly as possible across backyard buckets, so as to minimize the probability that any backyard bucket overflows. This would easily be achieved if each overflowing item hashed independently to two backyard buckets. However, as said above, we will compromise on this independence: all items that hash to a given front-yard bucket will hash to the same two backyard buckets. Nonetheless, even with this constraint, we want the front-yard/backyard mapping to spread overflows as evenly as possible across backyard buckets.

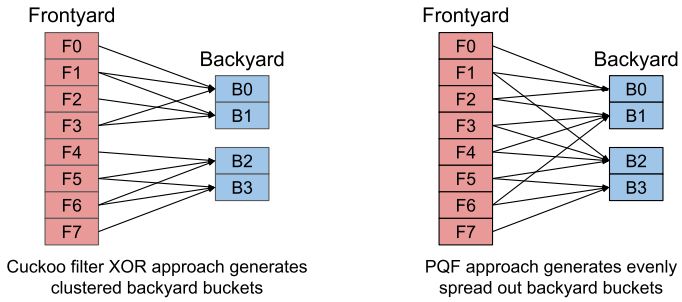


Fig. 2. An example of how cuckoo style of two choice hashing works versus how the hashing in the breadcrumb filter works. Notice how the cuckoo style hashing results in disconnected components. This can lead to insertion failures at significantly lower load factors. CRUMB hashing in BCF spreads out the mapping which is critical in achieving high load factors. Front-yard to backyard ratio C is 2.

The BCF smooths out the distribution of overflows to each backyard bucket by having each backyard bucket receive overflows from multiple front-yard buckets. Thus, the number of overflows that hash to a given backyard bucket will get averaged across several front-yard buckets, which will tend to even out the distribution of overflows mapped to each backyard bucket. Additionally, we ensure each frontyard bucket can overflow to two different backyard buckets, improving load balancing by inserting overflows into the emptier of the two possible backyard buckets. Specifically, we set the front-yard to have C times as many buckets as the backyard, i.e. $F = CB$. Let $h_1(f)$ and $h_2(f)$ be the functions that map a front-yard bucket to its two backyard buckets. We design h_1 and h_2 to be C-to-1 functions, so that they evenly map overflows across backyard buckets.

Furthermore, we want the two hash functions to spread load evenly across the entire backyard. Consider constructing a bipartite graph with two edges from each front-yard bucket to its corresponding backyard buckets, as shown in Figure 2. We want the bipartite graph to be connected to allow information about heavily loaded backyard buckets to flow across the entire graph. If the graph were not connected, then it would be like we were running several, smaller, independent instances of a minimum-of-two-choices hashing scheme. Since the probability of failure of minimum-of-two-choices load balancing goes down as a function of the number of backyard buckets, running multiple small instances would result in a higher failure probability than running one large instance. We initially considered using the xor trick, but it failed precisely because it created disconnected components, as in Figure 2.

Finally, in addition to this property helping increase the load, it enables satisfying the total collision property with little additional space per backyard entry versus front-yard entry. Since h_1 and h_2 are each C-to-1 mappings, we should be able to record the origin of each item in the backyard by storing $1 + \log C$ bits (the extra bit is to record which hash function was used)—along with the index of the backyard bucket itself, this is sufficient to uniquely identify the front-yard bucket.

Construction. The intuition behind our mapping scheme is to treat the front-yard bucket index f as a number written in base C, with digits $f_{\ell-1} \cdots f_0$. We then set $h_1(f) = f_{\ell-1} \cdots f_1$ and $h_2(f) = f_0 f_{\ell-1} f_{\ell-2} \cdots f_2$. Note that we need to record only f_0 in order to be able to invert $h_1(f)$ and we need to record only f_1 in order to invert $h_2(f)$. Since f_0 and f_1 are digits base C, we can record them each using $\log C$ bits.

It is also easy to see that, when the number B of backyard buckets is a power of C, the undirected bipartite graph on the front-yard and backyard buckets is connected (and in fact has diameter

Algorithm 1 FrontyardToBackyardHash (f) -> ((b_0, b_1), (f_0, f_1))

Input: $\boldsymbol{f}$ Front-yard location
Output: Pair of bucket indices ($b_0, b1$) and breadcrumbs (f_0, f_1)
Constant: **C** Ratio of number of front-yard to backyard buckets
$b_0 \leftarrow \left\lfloor \frac{f}{C} \right\rfloor$ ▷ $b_0 = f_{\ell-1}f_{\ell-2}\cdots f_1$
$f_0 \leftarrow (f \bmod C)$
$f_1 \leftarrow (b_0 \bmod C)$
$x \leftarrow \left\lfloor \frac{b_0}{C} \right\rfloor$ ▷ $x = f_{\ell-1}f_{\ell-2}\cdots f_2$
$b_1 \leftarrow x + f_0 \cdot \left\lfloor \frac{B}{C} + 1 \right\rfloor$
return $(b_0, b_1), (f_0, f_1)$

$O(\log_C F)$). This is because we can get from any f to any f' by iteratively shifting in the digits of f' as we go back and forth between front-yard and backyard buckets. For example, we can follow the path

$$f_{\ell-1}\cdots f_0 \xrightarrow{h_2} f_0 f_{\ell-1}f_{\ell-2}\cdots f_2 \xrightarrow{h_1^{-1}} f_0 f_{\ell-1}f_{\ell-2}\cdots f_2 f_0'$$

to cyclicly rotate f by one digit and replace the least-significant digit with a digit of f'. By repeating this process, we can shift in all the digits of f' in $O(\log_C F)$ steps. This demonstrates the connectivity and bounds the diameter of the graph.

When B is not a power of C, the hash function h_2 does not work because shifting f_0 into the most-significant-digit position can result in backyard indices that are, in the worst case, almost C times larger than $B-1$, the largest valid backyard bucket index. So, instead of shifting f_0 into the most-significant-digit position, we set the backyard bucket index to be

$$h_2(f) = f_{\ell-1}f_{\ell-2}\cdots f_2 + f_0 \cdot \left\lfloor \frac{B}{C} + 1 \right\rfloor,$$

as shown in Algorithm 1.

It is easy to see that, since $f < CB$, $h_2(f) \le B + C - 2$. Thus $h_2(f)$ may still be larger than $B-1$, but only by a small amount. We handle this by allocating an extra $C-1$ buckets in the backyard.

Second, given $h_2(f)$ and f_1, we can still recover f as follows. First, setting $x = f_{\ell-1}f_{\ell-2}\cdots f_2 \le \left\lfloor \frac{B}{C} \right\rfloor$, observe that we can recover f_0 by computing $f_0 = \lfloor h_2(f)/\lfloor B/C + 1\rfloor\rfloor = f_0 + \lfloor x/\lfloor BC + 1\rfloor\rfloor = f_0$. Then we can easily recover $x = h_2(f) - f_0 \left\lfloor \frac{B}{C} + 1 \right\rfloor$. Given x, f_0, and f_1, we easily recover f by concatenating $x||f_1||f_0$.

Space efficiency trade-off. The total collision property helps enable deletions and enumerating hashes in the filter, thereby achieving fully-functional filters. CRUMB hashing achieves the total collision property and also helps evenly distribute items in the backyard. As a result, it achieves a high load factor comparable to state-of-the-art filters along with state of the art performance. As an alternative to CRUMB hashing, we empirically evaluated the xor trick from the cuckoo filter [34] in the breadcrumb filter. That is, we considered letting $h_1(f) = f_{\ell-1}\cdots f_1$ and $h_2(f) = f_{\ell-1}\cdots f_1 \oplus f_0$, analogous to the xor trick, but this resulted in a significantly lower maximum load factor. The xor trick results in 12-20% reduction (depending on filter configuration) in the maximum load factor achieved by the breadcrumb filter. See Figure 2 for intuition.

3.4 Mini-filter design

We employ the mini-filter structure to efficiently store and retrieve remainders in the buckets. Each bucket in the front-yard and backyard is a mini quotient filter [53] which is called a ***mini-filter*** [55].

We will now describe the encoding of the mini-filters and how this encoding enables $O(1)$ time operations using AVX-512 vector instructions.

Each mini-filter is further divided into b mini-buckets, unlike the unstructured block employed in cuckoo and morton filters. Each fingerprint x inserted into a mini-filter is hashed using a hash function $h(x)$. We divide the output $h(x)$ into a $\log b$-bit mini-bucket index, $m(x)$, and an r-bit tag, $r(x)$. We then add $r(x)$ to mini-bucket $m(x)$. Note that we can recover $h(x)$ from $r(x)$ and $m(x)$. Similarly, queries for an element y return yes only if $r(y)$ is present in mini-bucket $m(y)$. The main challenge is to efficiently encode the mini-bucket structure.

Front-yard mini-filter. Figure 3 shows how the mini-filter stores tags along with their corresponding mini-buckets in the front-yard. It uses a $(b + s)$-bit mini-bucket-size vector followed by an array of up to s tags. The tags are stored in mini-bucket order, meaning all tags in mini-bucket 0 are stored together, followed by those in mini-bucket 1, and so on. The number of tags in each mini-bucket is encoded in unary in the mini-bucket-size vector. Therefore, the total number of metadata bits is at most $b + s$, and the total size required for a block is at most $b + (1 + r)s$. Figure 3 illustrates an example of the mini-filter's encoding, using colors to differentiate keys across different mini-buckets.

Backyard mini-filter. Figure 3 shows how the mini-filter stores tags along with their corresponding mini-buckets in the backyard. Compared to the front-yard mini-buckets, backyard mini-buckets also store "crumbs" that enable us to map items to their front-yard bucket. Multiple front-yard buckets can map to a single backyard bucket. However, as described earlier we only need h_1 and h_2 to be C-to-1 functions. The backyard needs only $1 + \log C$ bits per item to store the reverse mapping. The mini filters in the backyard buckets maintain this mapping along with the tags and the metadata.

3.5 Enumerability

Enumerability naturally follows from the above design. Recall that a filter is enumerable if it stores, losslessly, the multiset $h(S) = \{h(x) \mid x \in S\}$. For the filter to be enumerable, it must reconstruct $h(x)$ for any x stored in the filter.

The BCF hashes each item to $h(x) = (f, m, r)$, where f is the index of x's front-yard bucket, m is the mini-bucket index, and r is the remainder for x. The front-yard mini-filter stores (m, r) losslessly in bucket f, so it is obviously possible to reconstruct $h(x)$ for any x stored in the front-yard. Furthermore, since we can move items from the backyard back to the front-yard, we can obviously recover f and (m, r) for all items in the backyard, as well. Hence the BCF is enumerable.

4 BCF Operations

This section describes the algorithms used to implement the *insert*, *lookup* and *delete* operations in a BCF. Mini-filter operations in the BCF are implemented in a similar manner as the vector quotient filter [55]. However, the queries in the front-yard and removals from the backyard are implemented differently in the BCF. Note that the mini-filter operations in vector quotient filters already sort the elements by their mini bucket index, meaning that there needs to be no additional structure to implement the ordering (with the slight exception of backyard removals).

Front-yard bucket operations are generally distinguished from backyard bucket operations due to the fact that backyard buckets need to store additional information (specifically, $1 + \log C$ bits) to efficiently identify the corresponding front-yard bucket.

Insert. Algorithm 2 shows the pseudocode for the insert operation. To insert an item, the BCF hashes it as $h(x) = (f, m, r)$, where f is the index of x's front-yard bucket, m is the mini-bucket index, and r is the remainder.

Remainders within buckets are organized by mini-bucket index. The mini-bucket index m is conceptually compared with others in the bucket to determine where to place the remainder r. The

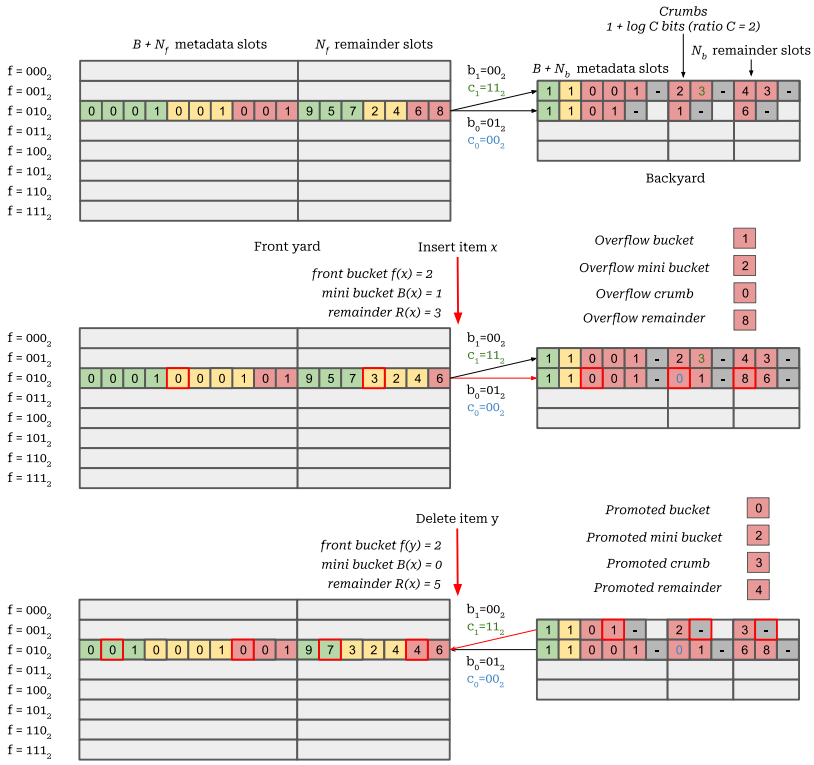


Fig. 3. An example of an insert followed by a delete operation in the breadcrumb filter with front-yard-backyard design. Each row is a mini-filter. Mini-filters have b logical buckets, s slots for storing tags, and $b+s$ metadata bits ($b = 3$ and $s = 7$ in the example). The metadata bits encode, in unary the number of tags in each bucket. The fingerprints for each bucket are stored consecutively in the fingerprint array. When the front-yard bucket is full, a tag from the largest mini bucket is kicked to the backyard. Minimum-of-two-choice hashing is used in the backyard. Front-yard to backyard ratio C is 2. Finally, we use the simplified hashing scheme for when the size of the table is a power of C. Here, we use c_0 and c_1 to refer to the crumbs, which store both h_i and whether $i = 1$ or $i = 2$ in one 1+ log C bit integer. Green boxes refer to data for the first logical bucket, yellow for the second, and red for the third.

insertion uses AVX512 vector instructions to shift existing remainders to create space for the new remainder (if needed) and store r into the appropriate slot, guided by bucket metadata. This metadata is updated after the insertion to reflect the new state of the bucket.

If the front-yard bucket overflows, we track the overflowing mini-bucket index m' (the maximum of the existing indices and m) and the remainder r'. The front-yard bucket index f is then re-hashed to (b_1, r_1) and (b_2, r_2) as described in the previous section. The combined backyard remainder $r_i r'$, formed by concatenating the front-yard remainder and crumb r_i, is inserted into the less occupied of the two backyard buckets. Insertion fails only if both buckets are full.

Lookup. Algorithm 3 presents the pseudocode for the lookup operation. A query begins by computing (f, m, r) as in the insert procedure and checking whether the remainder r is present in the front-yard bucket. If found, the query returns true.

If not found and the status indicates `PotentialBackyard`, a backyard lookup may be required. This check slightly extends the typical bucket query logic: when the front-yard bucket is full and the mini-bucket index m of the queried key is greater than or equal to the largest mini-bucket index in

Algorithm 2 Insert (x)

```
(f, m, r) ← h(x)          ▷ retrieving remainder r to store in minibucket m in front-yard bucket f
over, (m′, r′) ←InsertFrontBucket(f, m, r) ▷ returns if bucket overflowed; (r′, m′) is the overflow
if over then
    ((b1, b2), (r1, r2)) ←frontyardToBackyardHash(f)
    r1 ← r1r′
    r2 ← r2r′
    if full(b1) and full(b2) then
        return False
    end if
    if count(b1) < count(b2) then                    ▷ If b1 is emptier, insert into b1
        InsertBackBucket(b1, m′, r1)
    else
        InsertBackBucket(b2, m′, r2)
    end if
end if
return True
```

Algorithm 3 Lookup (x)

```
(f, m, r) ← h(x)          ▷ retrieving remainder r, minibucket m, front-yard bucket f
status ← QueryBucket(f, m, r)
if status == Present then
    return True
else if status == PotentialBackyard and full(f) then   ▷ minibucket was at least as large
as the largest minibucket of a remainder in the front-yard, meaning need to search backyard
    ((b1, b2), (r1, r2)) ←frontyardToBackyardHash(f)
    r1 ← r1r          ▷ Concatenate r′ with r1 (r2) to get first (second) backyard remainder
    r2 ← r2r
    return (QueryBucket(b1, m, r1) == Present) || (QueryBucket(b2, m, r2) == Present)
end if
return False
```

the front-yard, it is possible that the key resides in the backyard. This is because the backyard holds keys with mini-bucket indices at least as large as the maximum seen in the front-yard.

Remove. Algorithm 4 presents the pseudocode for the remove operation. As with other operations, removal first attempts to delete the key from the front-yard. If the front-yard bucket is not full, the operation terminates—regardless of whether the key was found—since no keys could have been offloaded to the backyard.

If the key is present and the front-yard bucket is full (i.e. some keys may have been moved to the backyard), then the key is removed from the front-yard, and a replacement is fetched from the backyard. Specifically, the key with the smallest mini-bucket index across the two corresponding backyard buckets is identified and moved back. This is done by checking which entries match the prefix r_1 (or r_2) and using the `ctzll` (count trailing zeros) instruction to find the index of the smallest matching key. This index helps identify the mini-bucket index of the candidate, which is then removed from the backyard and reinserted into the front-yard.

Algorithm 4 Remove (x)

```
(f, m, r) ← h(x)                          ▷ retrieving remainder r, minibucket m, front-yard bucket f
if full(f) then                                                                                  ▷
  If f is full, x may be in the backyard or may need to retrieve an element from the backyard
    e ← RemoveFromFrontBucket(f, m, r)                                                           ▷
  e stands for key was eliminated (true) or not present (false)
    ((b_1, b_2), (r_1, r_2)) ←front-yardToBackyardHash(f)x
    if e then                                          ▷ (m, r) was removed from front-yard
        (m_1, r'_1) ← minElem(b_1, r_1)                                          ▷ Grab element
  with the smallest minibucket index coming from the front-yard bucket determined by r_1
        (m_2, r'_2) ← minElem(b_2, r_2)
        if m_1 < m_2 then
            RemoveFromBackBucket(b_1, m_1, r_1 r'_1)
            InsertFrontBucket(f, m_1, r'_1)
        else
            RemoveFromBackBucket(b_2, m_2, r_2 r'_2)
            InsertFrontBucket(f, m_2, r'_2)
        end if
        return True
    else                                               ▷ (m, r) not found in front-yard
        if RemoveFromBackBucket(b_1, r_1 r') then
            return True
        else if RemoveFromBackBucket(b_2, r_2 r') then
            return True
        else
            return False                                          ▷ Key was not found anywhere
        end if
    end if
else
    return RemoveFromBucket(f, m, r)                  ▷ Simply remove x from the bucket
end if
```

If the key is not found in the front-yard, removal proceeds to the backyard: first attempting deletion from the bucket at b_1, and if unsuccessful, from b_2. If the key is not found in either, the removal fails—indicating that the key was either never inserted or already removed.

5 Space Analysis

We now analyze and provide a theoretical bound for the space usage of the BCF and compare it to other filters. Later we present the space usage of the filters in practice.

The primary factor impacting space usage is the ratio of the size of the front-yard to the size of the backyard. We call this ratio C. Asymptotically estimating the optimal value of C is nontrivial, so we note that empirically the best performing filter variants (in terms of maximum load factor supported) we implemented use $C = 8$ for both the 32 byte bucket and the 64 byte bucket configurations (see Section 6 for more details). It is reasonable to assume that larger buckets would enable larger values of C. This is because the number of keys that overflow a front-yard bucket scales as the square root of the size of the bucket $\sqrt{N_f}$, where N_f is the front-yard bucket size, meaning that the

Table 1. Details of the configuration of two breadcrumb filter variants employed in the evaluation. Number of remainders in a bucket be N_f for the front-yard and N_b for the backyard, the number of mini-buckets be b, and the number of bits used in a remainder be $\log \varepsilon$) be R.

Filter variant	R	b	N_f	N_b	Bucket size	FPR
BCF53	8	53	51	35	64B	0.39%
BCF36	16	36	28	22	64B	0.0018%

ratio of the number of overflows 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 front-yard 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 [55]). The backyard needs to store an additional $1 + \lceil \log(C) \rceil$ bits per key, in order to store whether it is the first choice bucket and what the bits that were cut from the front-yard location were. For our BCF variants 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 yield around $2 + \log(1/\varepsilon)$ bits for the mini-filters in the front-yard, so the average space consumption per key is just slightly higher at $2.5 + \log(1/\varepsilon)$.

The BCF variant using a bucket size of 32 achieves a load factor of $75-85\%$ (for most practical values of number of keys 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 BCF variant 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 for $\log(1/\varepsilon) = 8$, so this filter variant should be used when space is at a premium. This space usage is quite competitive to other state-of-the-art filter designs. Once again, the filter is slightly under-filled 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/mini-filters in both the front-yard and the backyard is that they need to fit within a single cache line to enable fast operation. This also enables us to employ AVX512 vector instructions that can operate on the whole cache line. The two main components of the mini-filters are the array of remainders and the metadata. The metadata uses one bit per remainder and one bit per mini-bucket to separate out the remainders with. Therefore, letting the number of remainders in a bucket be N_f for the front-yard and N_b for the backyard, the number of mini-buckets be b, and the number of bits used in a remainder (roughly corresponding to the log (false positive rate)) be R, the space usage of a front-yard bucket is $b + N_f(R + 1)$.

Additionally, the backyard buckets need to store which front-yard bucket the remainder came from out of the possibilities. In the case of our filter configurations, there are always at most 16 possibilities, so this uses four bits. Therefore, the space usage of a backyard bucket is $b + N_b(R + 5)$. Please refer to Figure 3 for an example.

One further constraint that could lead to faster filters would be to use just one machine word for the metadata. As we employ the LZCNT instruction (leading zeros count) to process metadata, the maximum size is 64 bits. This is naturally the case when the size of the remainders is 16 bits—in the front-yard, we set $N_f = 28$ and $b = 36$, which uses precisely 64 bits, using the other 56 bytes of the cache line for the remainders. In the backyard, $N_b = 22$.

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, a few possible configurations

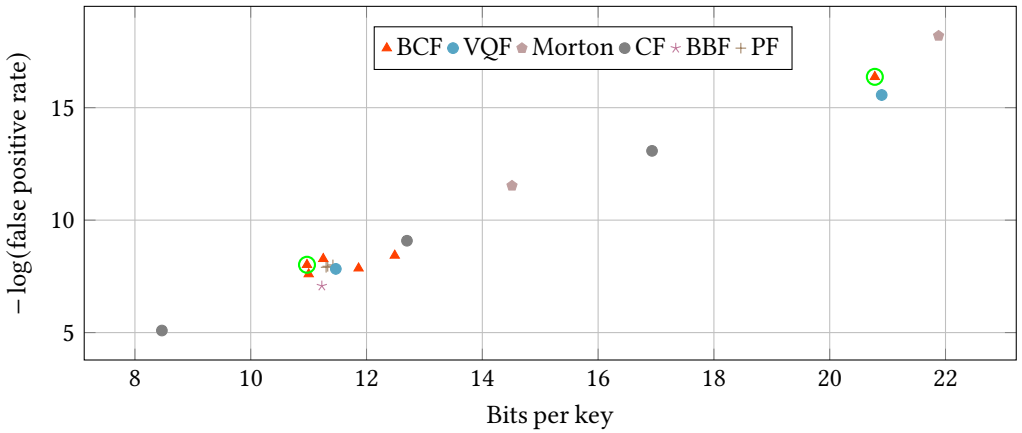


Fig. 4. False positive rates vs bits per key empirically achieved for different filters (tests are run until they succeed, at increments of 0.005 load factor). Higher is better. Several configurations are tested for each filter. The two configurations of the BCF that we evaluate in Section 7 are circled in green.

are $N_f = 53$, $b = 51$, and $N_b = 35$ (BCF8 − 53); and $N_f = 62$, $b = 50$, $N_b = 34$ (BCF8 − 62) (which offers marginally higher space efficiency but lower performance).

In the second option, the seemingly fastest configuration is to use $N_f = 22$, $b = 26$, $N_b = 18$ (BCF8 − 22), with similar tradeoffs with having more mini-buckets per bucket possible. This configuration outperforms the above configurations, but supports a significantly lower space efficiency. In addition, it is possible to alleviate the load factor issues by making backyard buckets 64 bytes while front-yard buckets are 32 bytes (an example configuration we tested but do not present is equivalent to BCF8 − 22 but with $N_b = 37$), but this means that a larger fraction of the remainders will be in the backyard, reducing the advantage of breadcrumb filters.

All configurations we considered have either 8 or 16 bit remainders in order to optimize performance by exploiting cache line sizes and hardware instructions, but this is not inherent to the filter design. With small modifications to the filter design, specifically, the mini-filter design, the breadcrumb filter can be extended to support (almost) arbitrary FPR by trading off slight performance. This is inline with existing state-of-the-art filters such as VQF, Morton, and Prefix filter, which support a small set of remainder size options but can be redesigned to support arbitrary FPRs for a slightly slower performance.

Figure 4 shows the false positive rate versus space usage per key of different configurations of the BCF in comparison with existing filters.

Concurrency. An important aspect of filter performance that is frequently overlooked in the literature is *concurrency*. Most enterprise applications are concurrent, and critical components often require a fully concurrent filter for easy integration and simplified design. However, concurrency often comes with additional overheads depending on the filter design. For example, it is relatively straightforward to achieve concurrency in the Bloom and blocked Bloom filters using atomics. In enumerable filters, the quotient filter [53] and vector quotient filter [55] provide fully concurrent implementations.[4] The quotient filter maintains separate locks for a block of contiguous slots which results in extra cache line probes and write backs resulting in higher performance overheads. On the other hand, the VQF encodes the locks inside the same cache line along with the fingerprints and achieves concurrency without an extra cache line miss. Other filters such as cuckoo, morton, and prefix do not offer concurrent implementations.

[4]As implemented, the vector quotient only supports concurrent insertions, but it can trivially be modified to support concurrent queries/deletes as well

The BCF offers concurrency without incurring an extra cache miss and achieves almost linear scalability with increasing number of threads for all operations. We employ a single bit to indicate the lock and encode the bit in the same cache line along with the metadata and remainders.

Expandability. The breadcrumb filter supports enumeration, allowing us to iterate through all stored hashes. This capability enables filter expansion through a straightforward approach: we create a new filter with double the number of mini-filters and migrate the existing data by enumerating and reinserting all hashes from the original filter. During expansion, the filter doubles its mini-filter count and uses a bit from the remainder to accommodate the increased index space. That is, when reinserting a hash $h(x) = (f, m, r)$ into the expanded filter, we compute a new hash $h'(x) = (f', m', r')$ by reallocating bits from the combined $(m + r)$ field, specifically borrowing one bit to extend the fingerprint f'. This bit reallocation is necessary because doubling the mini-filters requires an additional bit for indexing into the mini-filters.

This redistribution of bits reduces the available bits within each mini-filter, potentially increasing the false positive rate. However, it is possible to mitigate this effect by adopting techniques from recent expandable filter designs [25, 27]. These approaches use longer fingerprints for keys inserted after expansion to counterbalance the increased false positive rate from keys inserted earlier.

7 Evaluation

We now evaluate our implementation of the breadcrumb filter (BCF) variants. We include two BCF variants (BCF53 and BCF36) as described in Table 1. Our code is publicly available on Github [5]. We compare against state-of-the-art filter data structures: vector quotient filter (VQF) [55], prefix filter (PF-TC) [33][6], cuckoo filter (CF) [34], morton filter (MF) [15], and blocked Bloom filter (BBF) [60]. We evaluate each filter on three fundamental operations: *insertions, lookups, and removals*. We evaluate lookups both for items that are present (positive queries) and for items that are not present (negative queries) in the filter.

The BCF is an enumerable and full-featured filter. Among the state-of-the-art filters, only the VQF is enumerable. The PF and BBF are not enumerable and do not support deletes. The CF and MF support deletes but are not enumerable as they can't disambiguate the primary bucket of the fingerprint from the alternative and as a result can't reconstruct the original hash $h(x)$.

7.1 Experimental setup

Our experimental setup follows prior filter evaluations [15, 33, 34, 53, 55], with minor enhancements to the evaluation metric to improve practicality and ensure a fair comparison. Specifically, we measure the *throughput* as a function of *space efficiency* instead of *load factor*. We further extend beyond the typical evaluation framework by additionally measuring in-cache performance, aging behavior, concurrency scalability, and merging operations.

Dataset. Our evaluation uses keys drawn uniformly at random from the universe of 64-bit keys, consistent with all prior filter evaluations. Since the BCF and other evaluated filters hash keys before insertion, they are inherently robust to spatial skew in the input distribution.

Load factor. The breadcrumb filter uses a two-tier design in which overflows from the front-yard are redirected to the backyard. As the filter fills, the likelihood of accessing the backyard increases, which can impact performance. Since the filters we compare against also exhibit different performance characteristics at varying load levels, we evaluate each filter across the load factors it supports to ensure a fair comparison.

[5] https://github.com/saltsystemslab/BreadcrumbFilter

[6] This refers to the two-choice variant of the prefix filter.

Space efficiency. Comparing filters at different load factors is inherently challenging. First, different filters support different *maximum* load factors. Moreover, load factor alone is not always the best proxy for *space utilization*. For instance, although cuckoo filters can achieve higher load factors than breadcrumb filter, we will show that breadcrumb filter attains better space efficiency.

Similar to prior works [15, 34, 55], we define the *space efficiency* as, $n \log \varepsilon^{-1}/S$, where n is the number of items inserted into a filter, ε is the false positive rate, and S is the total number of bits used by the filter. The numerator, $n \log \varepsilon^{-1}$, represents the information-theoretic minimum space required [17], while the denominator reflects the actual space used. All the filter designs we consider (**except** the Bloom filter with $1.44n \log \varepsilon^{-1}$ space usage) incur an overhead of $\Omega(n)$ bits beyond the lower bound. This means that configuring a filter for a lower false positive rate can make its space efficiency appear artificially higher. For example, the space efficiency when setting $\log \varepsilon^{-1} = 8$ is $\frac{8}{11}$, while the space efficiency when setting $\log \varepsilon^{-1} = 16$ is $\frac{16}{19} > \frac{8}{11}$. Therefore, as the cuckoo filter configuration we compare against has a lower false positive rate than our own filters, the space efficiency of cuckoo filters appears slightly overstated. We evaluated different variants of the cuckoo filter with 8, 12, and 16 bit fingerprints and the performance variance was within 10% and CF12 being the fastest for insertions and competitive for queries. Therefore, we picked CF12 as it offers better trade-off between performance and FPR compared to other variants.

Despite these flaws, we believe space efficiency provides a fairer comparison than load factor. We evaluate filters based on space efficiency rather than raw load factor. We benchmark each filter for different load factors, with a step of 0.05 and a maximum corresponding to rounding down to a tick of 0.005 for the 1pct worst maximum load factor a filter can support with $N = 2^{30}$ (as we detail in a later subsection).

False positive rate. Filters only support a fixed set of configurations, and that makes them difficult to directly compare. For one, CF does not have an exactly comparable false positive rate to our filter design. When configured at 8 bits per fingerprint, the CF has a false positive rate of $1/32$ while at 12 bits per fingerprint, CF has a false positive rate of approximately $1/512$. With 8 bit fingerprints, both our filter design and the VQF support a false positive rate of roughly $1/256$, with our filters supporting a somewhat lower false positive rate than the VQF. This makes the choice of comparison for the VQF obvious, but the CF choice is less clear. We chose to compare to the 12 bit configuration of the cuckoo and morton filters, as this was the closest comparison point available. The BBF is configured with a false positive rate reasonably close to $1/256$.

Throughput measurement. We run each test three times and average to attain a more accurate result. To measure query performance at a load factor, we simply load the table up to that load factor and then run a number of queries to measure performance. To measure insertion and removal performance, we measure the time to load the table up to the load factor and then subtract off the time to load the table up to the previous load factor. For example, the average time to get up to 0.9 is subtracted from the average time to get up to 0.85, and then this is divided by the number of items inserted in this interval to get the amortized time per insertion.

For each filter, we run the original code supplied by the authors and integrate it into our own test suite in order to ensure that the best competing implementations are used.

System specification. Our test system is a two socket server with Intel Xeon Gold 6338 32 core 64 thread CPUs, $1TiB$ total of system memory, and 96 MiB L3 cache across the two nodes. To avoid NUMA communication penalties, we treat the system as a single socket system, and run all of our code on the first CPU and NUMA node. It is running the Linux kernel version `5.4.0-155`.

RAM vs cache setup. Each filter has different performance characteristics depending on if fingerprints need to be fetched from RAM or cache, as each filter does a different amount of computation,

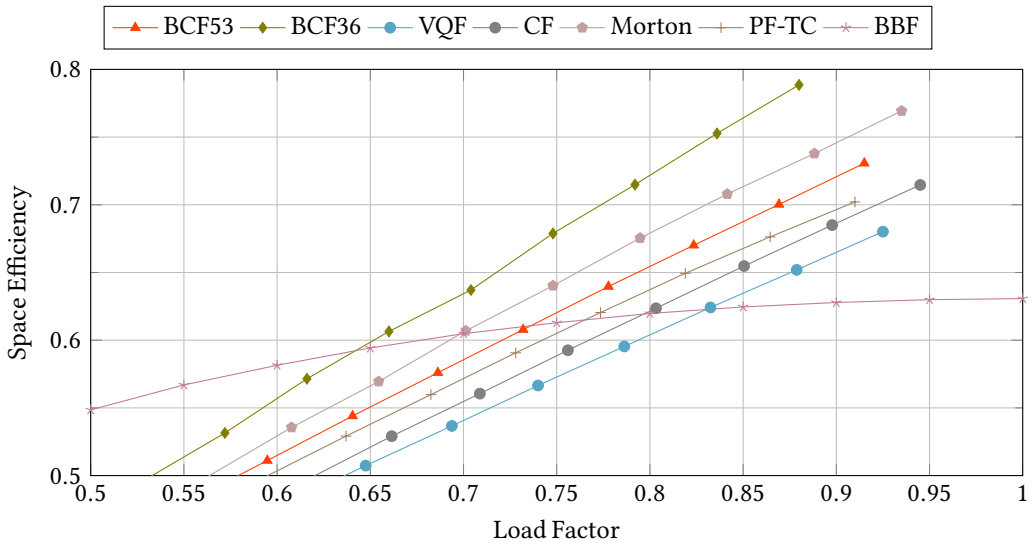


Fig. 5. Space efficiency at different load factors for different filters (Higher is better.)

affecting the processor's prefetching capabilities in different ways. Additionally, our filter in particular has a backyard that is significantly smaller than the front-yard, meaning that the backyard may reside in cache while the front-yard does not. Therefore, we test each filter with two different slot counts: 2^{22} and 2^{30}. In the first case, the filters are fully in L3 cache, so this measures the pure cache performance. In the second case, the filters take up over a gigabyte each and are guaranteed to be almost entirely in RAM, even the backyard of our filters.

7.2 Space efficiency

Figure 5 shows the space efficiency of each filter with changing load factors. The highest space efficiency is the breadcrumb filter with 16 bit remainders(BCF36). Morton filters achieve second highest space efficiency, as they target $\log \varepsilon^{-1} = 12$, so their space usage is also similarly incomparable to the rest of the filters using 8-bit remainders (or tags). Cuckoo filters achieve a lesser space efficiency than the variant of the breadcrumb filter with 8 bit fingerprints.

If the maximum possible space efficiency is required while retaining high throughput, the BCF is the best option. The maximum space efficiency achieved for CFs is 0.715, the maximum for the large front and backyard version of BCF is 0.7307, and the maximum for VQFs is 0.665.

7.3 In-RAM performance

Figure 6 shows the in-RAM performance for various filters plotted against space efficiency. Performance of all filters goes down as the space efficiency increases, but this change varies wildly by filter and operation. We now provide a brief summary of the performance for each operation.

Inserts. The different configurations of breadcrumb filter designs all perform similarly to each other at insertions, and the configurations with fewer fingerprints in a bucket perform slightly better. Up to a space efficiency of about 0.2, the cuckoo filter performs similarly to the breadcrumb filter, but, at higher space efficiencies, the insertion performance of the cuckoo filter drops precipitously. At a space efficiency of roughly 0.6, the BCF53 and BCF36 perform approximately 3.5× and 4× faster than cuckoo filters, and, at the maximum cuckoo filter space efficiency of 0.715, the BCF53 and BCF36 outperform the cuckoo filter by over 10×. Breadcrumb Filters perform between 5% — 50% faster than vector quotient

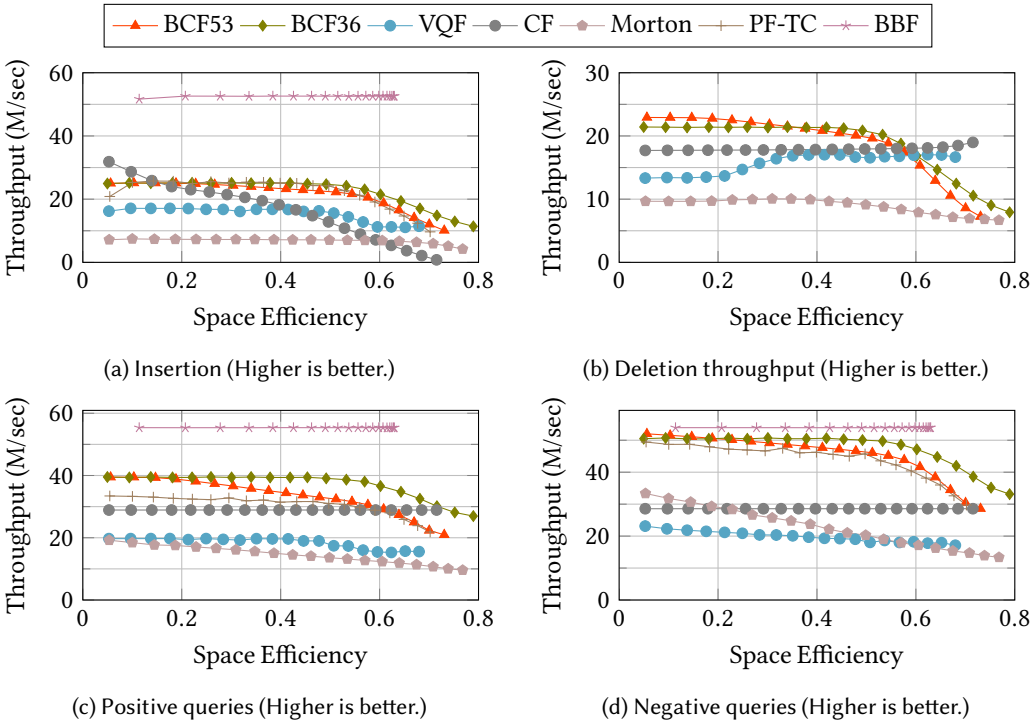


Fig. 6. Insertion, deletion, and lookup performance of different filters in RAM for different load factors. Averaged over 3 runs. BBF does not support deletions and does not have the corresponding graph.

filters when at the same space efficiency, depending on load factor and configuration. All BCF variants perform similarly to the prefix filter with BCF36 having higher performance at high space efficiency.

Queries. Query performance differs significantly between the different configurations of the breadcrumb filters. Smaller buckets (BCF36) enable competitive to significantly better query performance than all the other filters. Even the larger bucket BCF (BCF53) outperforms vector quotient filters across the different space efficiencies by 40% — 100%, depending on the load factor. At a space efficiency of 0.6, the BCF53 performs roughly 46% and 1% better than the cuckoo filter at negative and positive queries, respectively; at maximum load, the BCF53 has equivalent negative queries but 27% worse positive queries than the cuckoo filter.

Removals. Removal performance is similar across the BCF variants. The BCF variants offer higher performance compared to the vector quotient filter and cuckoo filter up to 60% space efficiency. Both the cuckoo filter and the vector quotient filter have better removal performance at high load factors. At high load factor, BCF53 under performs VQF and cuckoo by 56% and 62%, respectively. The Morton filter has the slowest removal performance across load factors.

7.4 In-cache performance

Figure 7 shows the in-cache performance for various filters design plotted against space efficiency. At a high level, the relative performance of each filter for insertion, deletion, and queries is similar to the in-RAM performance. The BCF variants with small buckets offer competitive query performance to the cuckoo filter, and all filter configurations match or outperform the vector quotient filter with the exception of deletions at high load factors, in some cases by more than 2×.

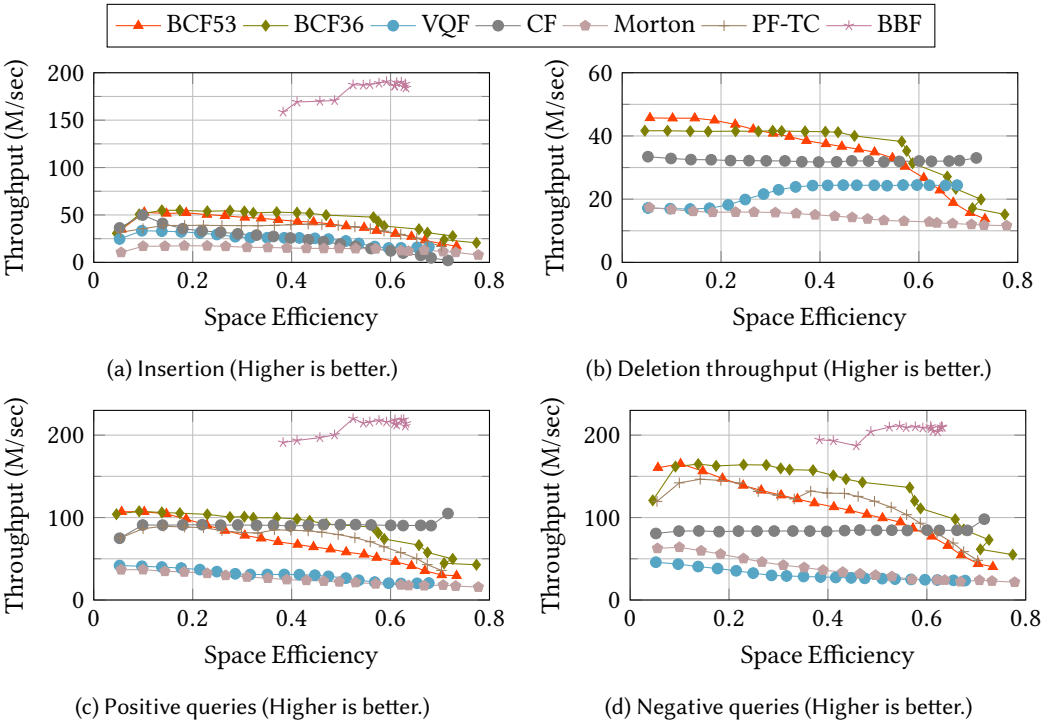


Fig. 7. Insertion, deletion, and lookup performance of different filters in L3 cache. Averaged over 3 runs. BBF does not support deletions and does not have the corrresponding graph.

7.5 Maximum load factors

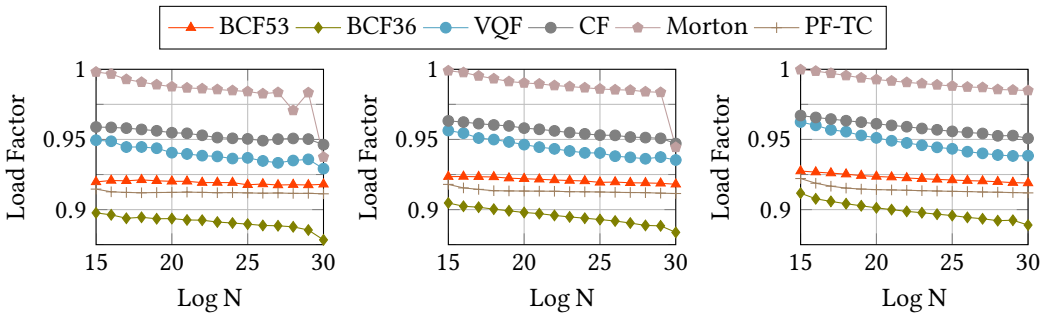
Figures 8a to 8c show maximum load factors each filter can support with 1%, 10% and 50% failure probabilities. Each point is obtained by inserting into each filter until failure. We perform multiple runs for each filter size depending on the size (1000 tests for $\log N$ from 15 to 24, 100 tests for $\log N = 25, 26$, 50 tests for $\log N = 27$, 20 tests for $\log N = 28$, and 10 tests for $\log N = 29, 30$.).

There are three conclusions to draw from these graphs. First, the max load factors are all pretty similar. For example, BCF53 reaches slightly higher load factors than the prefix filter, and is within 3% of the cuckoo filter. Second, the load factors remain close across all failure rates, suggesting that all the filters support similar load factors regardless of the target failure rate. Finally, the lines are all very close to linear and with similar slopes, suggesting that all the filters will support similar load factors and failure probabilities for even larger data sets.

In short, all the filters have similar load factors across failure probabilities and filter sizes. We exclude the BBF as it does not explicitly store fingerprints in the same manner as all the fingerprint-based filters and therefore it does not "fail" in the same way. Instead of failing to find a slot, its false positive rate sharply rises to 1 after enough keys have been inserted, but there is no specific point at which it "fails".

7.6 Filter aging

Another concern with filters that support deletions is aging. In aging experiments, we evaluate the performance of filters when they are close to full over several iterations of deletions and subsequent insertions. In theory, cuckoo hashing supports deletions without failures. On the other hand, there is no similar theory to support two choice hashing which is employed in the vector quotient filter



(a) 1% failure rate (b) 10% failure rate (c) 50% failure rate

Fig. 8. Load factor at which filters fail 1%, 10% and 50% of the time. We perform 1000 runs for $\log N$ from 15 to 24, 100 runs for $\log N = 25, 26$, 50 tests for $\log N = 27$, 20 tests for $\log N = 28$, and 10 tests for $\log N = 29, 30$. Therefore, the results become noisier for larger N, especially for the 1% results.

Table 2. The ratio of the number of items that can be inserted/deleted to a filter filled to a specific space efficiency in the plot until it fails, or 50 times the number of slots in the filter have been churned through. This experiment follows the random deletions setup.

Efficiency	BCF 53	BCF 36	VQF	CF	Morton
0.63	50	50	50	50	50
0.68	50	50	0.049	50	50
0.7	50	50	N/A	50	50
0.73	0.028	50	N/A	N/A	50
Load factor					
0.8	50	50	50	50	50
0.85	50	19.24	50	50	50
0.9	1.33	N/A	50	50	50

and in the backyard table in the partitioned quotient filter. Therefore, it is important to measure how many deletions and subsequent insertions the vector quotient filter and breadcrumb filter can handle before an insertion failure.

Random deletions. Here, we insert keys into a filter until the desired load factor is reached. Then we delete a key that is in the filter and insert a new random key. This random deletion procedure is continued until either the filter fails or a certain limit is reached, which we have set to 50 times the number of slots in the filter, as before.

We have tested our filters at several different load factors as well as several different efficiency thresholds (load factors set to roughly approximate a certain efficiency). We present the results for the random deletions in Table 2. Notice that certain load factors or space efficiencies are not supported by certain filters, so we put N/A in their place. All tests were performed so that filters have $2^{22} = 4194304$ slots.

7.7 Scaling with multiple threads

With minimal effort, the BCF is able to support multiple threads by adding lightweight spin locks to each bucket. This takes up one bit in each bucket, meaning that the number of separators that the metadata supports decreases by one—one is used for the lock bit. This locking approach is similar to the locking approach employed in the VQF [55]. Here, we report results for BCF52 instead of BCF53

Table 3. Multi-threaded aggregate throughput of insertions and queries (Millions Ops/sec) in RAM. For insertions, we fill up the filters from 0 to their max space efficiency. Queries are performed at max space efficiency. For the mixed workload throughput, we fill up filters until they operate at their respective maximum space efficiency and perform 10M operations consisting of inserts/queries/deletes in the proportion (30%/40%/30%) and measure the aggregate throughput (Million Ops/sec). $N = 2^{30}$.

	Inserts		**Queries**		**Mixed Workload**			
Num threads	BCF52T	VQFT	BCF52T	VQFT	BCF52T	VQFT	Morton	CF
1	13.0	10.1	11.5	12.1	6.3	9.3	4.4	1.3
2	25.8	20.1	23.5	24.8	12.0	17.6	N/A	N/A
4	50.6	39.2	46.3	49.7	23.4	34.4	N/A	N/A
8	93.3	76.3	89.6	94.0	44.8	64.1	N/A	N/A
16	170.8	143.2	166.8	174.7	81.6	111.1	N/A	N/A
32	304.3	245.7	237.3	217.7	116.7	137.9	N/A	N/A

as one slot is used for the lock bit. Table 3 gives the results of scaling insertions, deletions, and a mixed workload (see Section 7.8) for the threaded version of the BCF and VQF, as these are the only filters that support multithreading. Both the filters scale almost linearly going from 1 to 32 threads where performance grows between $23\times - 24\times$ for insertions and $19\times - 22\times$ for queries. We note the cuckoo and morton filter do not support multiple threads, as insertions may access many different cachelines without a clear access pattern, making it difficult to avoid deadlocking. Also, we modified the VQF to use locks for queries, which is necessary for correctness.

7.8 Mixed workloads

The 'Mixed Workload' column of Table 3 shows the performance of various filters (that support deletions) for mixed workloads consisting of insertions (30%), deletions (30%), queries (40%) while being maintained at high load factors. BCF variants perform similar to the vector quotient filter on mixed workload. The VQF is faster for single threaded runs likely due to higher query performance at high load factor compared to the BCF variants.[7] The BCF52T performed 1.4 and 4.8 times faster on a single thread than the morton and cuckoo filters, respectively, which do not support concurrent operations.[8] Prefix and blocked Bloom filters do not support deletes and are therefore excluded from mixed workloads.

7.9 Merging

In addition to all the other features, the BCF supports merging two separate filters in a bandwidth-efficient way. Due to the single-hashing approach employed in the front-yard, two corresponding buckets in the filter can be merged without affecting other buckets. The merged backyard is not constructed as efficiently, as it depends on the random order of key arrival to do load balancing, but it is small. Merging involves enumeration, so this benchmark also evaluates enumeration performance. We evaluated BCF variants as well as the counting quotient filter (CQF) in several different load factors for merging performance. As merging reduces the size of the fingerprints, we slightly reduced the load factor of each filter by 0.01 from its maximum, to 0.905 for the BCF53 and 0.94 for the CQF. We did not compare to any of the other filters we compared to previously, as none of them support merging. The results for several different values of N are seen in Table 4.

[7] We expect that, when running at the same space efficiency, our filters would have better performance, but we performed these tests with each filter at maximum load.

[8] Note that the cuckoo and morton filters do not lock on operations and thus are not incurring the associated costs.

Table 4. Merge throughput for the BCF53, BCF36, and CQF. We are unsure of the reasons why, but CQF was unable to complete merging for $\log N = 22, 24$. BCF53 is merging at 0.905 load factor, BCF36 at 0.86, and CQF at 0.94.

$\log N$	BCF53	BCF36	CQF
22	11.8	12.5	-
24	16.3	16.9	-
26	16.1	16.7	2.89
28	15.4	15.9	2.92
30	14.9	15.5	2.92

Table 5. Query throughput (Thousand queries/sec) of WiredTiger database. On-disk version has 1024 MB of total cache between the filter and database. The results are given for three different ratios of negative to positive queries: 10/1, 100/1, and all negative queries.

Inv PR	BCF53	BCF36	VQF	CF12	Morton
On disk					
10	79	82	81	83	84
100	590	781	563	695	741
∞	1921	16706	1646	3838	7162
In memory					
10	1911	2034	1928	2106	1912
100	8121	9396	7202	9883	6957
∞	12966	17385	10979	18180	10484

Results. The BCF merge performance remains with increasing filter sizes and it is $\sim 5\times$ better than the CQF merge performance.

7.10 WiredTiger benchmarks

To evaluate the impact on database performance, we conducted benchmarks using filters integrated into WiredTiger, MongoDB's production storage engine. We implemented full-featured filters (supporting deletion operations) and measured query performance under various workload conditions. Our experimental setup leveraged WiredTiger's ability to operate both with limited cache and fully in memory. For the cache-constrained configuration, we allocated a total of 1024MB to be shared between the WiredTiger cache and the filter, setting WiredTiger's cache size to $(1024 - f)$MB, where f represents the filter size in megabytes. Additionally, we tested a fully in-memory configuration with no cache constraints.

We populated the database with $0.9 \cdot 2^{28}$ key-value pairs, using 8-byte keys and 24-byte values. Initially, we attempted to insert 2^{28} keys; however, the cuckoo filter exhibited poor performance due to supporting only non-power-of-two sizes (as filters require slightly more slots than the number of inserted keys). While this constraint did not affect our filters or the VQF, we reduced the dataset to $0.9 \cdot 2^{28}$ keys to ensure a fair comparison with CF12.

Following data insertion, we adjusted the WiredTiger cache size for each filter configuration to maintain a combined cache and filter size of exactly 1024MB. We then measured average query performance across three positive query rate scenarios: 10% positive (1/10), 1% positive (1/100), and 0% positive (all negative queries). The results are presented in Table 5.

Results. In our benchmark, the BCF36 achieves the highest database throughput across most configurations and is competitive to CF12 in some configurations. This performance advantage becomes most pronounced in workloads composed entirely of negative queries, where BCF36 significantly outperforms competing filters. In contrast, workloads dominated by updates and positive queries inherently require database access, thereby diminishing the performance benefits that filters provide. Consequently, the all-negative query scenario best demonstrates BCF36's effectiveness in production database systems, where it delivers substantial performance improvements over alternative filtering approaches.

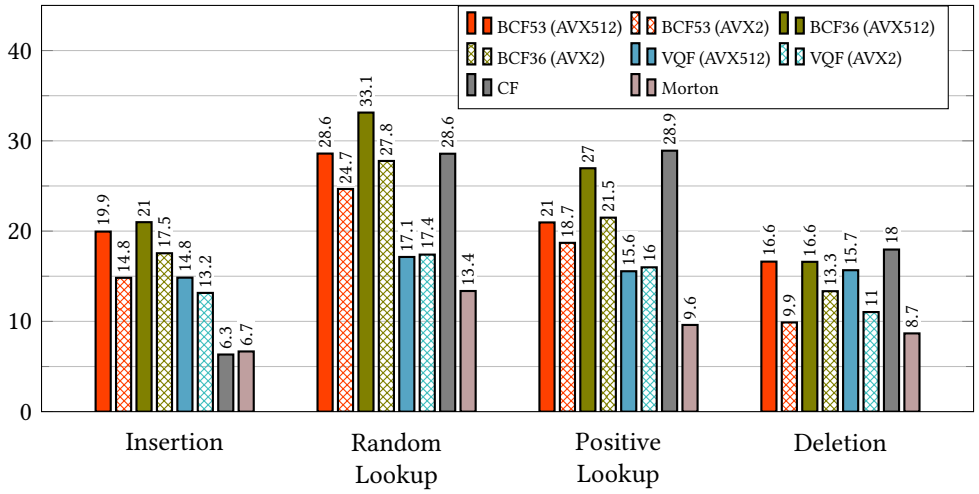


Fig. 9. Aggregate throughput for insertion, deletion, and lookup performance of AVX512 and AVX2 variants in RAM of the breadcrumb filter, with the CF filter, the Morton filter, and the AVX512 and AVX2 versions of the VQF for reference. Where filters have separate AVX512 and AVX2 versions, the AVX2 version is shown shaded.

7.11 Aggregate throughput and AVX2

We did an AVX2 implementation of breadcrumb filter variants and evaluate the impact of vector instructions on the performance of the filters. Figure 9 shows the aggregate throughput of breadcrumb filter variants in comparison to the AVX2 and AVX512 versions of the VQF as well as the CF and Morton filters for reference. The aggregate throughput for insertions (deletions) is calculated by filling (removing till empty) the filter until it reaches its max load factor, and then dividing the number of keys inserted (removed) by the time to do so. Query performance is measured when the filter is at max load factor. AVX2 is $\sim$ 12% to $\sim$ 40% slower compared to the AVX512 variant. This performance drop is inline with the performance drop in the AVX2 variant of the VQF presented by Pandey et al. [55].

8 Conclusion

This paper demonstrates an enumerable filter design using a novel front-yard/backyard hashing technique that achieves high performance and load factor. Across almost all load factors, the BCF is able to significantly outperform the state-of-the-art dynamic filters cuckoo, morton, and vector quotient filters (also enumerable) for all operations. We attribute the high performance to the fact that BCF variants often only check or update a single cacheline per query rather than two, as all the competing filters.

The proposed CRUMB hashing overcomes a traditional trade-off in filters based on a front-yard/backyard design to efficiently support deletes and achieve high load factors.

Additionally, the BCF supports bandwidth-efficient merges, whereas all of the competing filters are fundamentally unable to support faster merges than simply reinserting items due to the fact that two keys are unlikely to have the same two choices of buckets. We believe that this property holds potential for filters in database systems, as the filters within one layer of a LSM or size-tiered B^ϵ tree need to be merged on subsequent levels.

Acknowledgments

This research is funded in part by NSF grant OAC 2517201, 2513656 and CNS 2504471. Krapivin was supported by the Jeanne B. and Richard F. Berdik ARCS Pittsburgh Endowed Scholar Award.

References

[1] Fatemeh Almodaresi, Jamshed Khan, Sergey Madaminov, Michael Ferdman, Rob Johnson, Prashant Pandey, and Rob Patro. 2022. An incrementally updatable and scalable system for large-scale sequence search using the Bentley–Saxe transformation. , 3155–3163 pages.

[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] Yuriy Arbitman, Moni Naor, and Gil Segev. 2010. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Foundations of Computer Science*. 787–796.

[6] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2021. All-purpose hashing. *arXiv preprint arXiv:2109.04548* (2021).

[7] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Iceberg Hashing: Optimizing Many Hash-Table Criteria at Once. *J. ACM* 70, 6 (2023), 40:1–40:51.

[8] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. 2023. Tiny pointers. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 477–508.

[9] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. 2018. Bloom Filters, Adaptivity, and the Dictionary Problem. In *Proc. 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. Paris, France, 182–193.

[10] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, 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. *PVLDB* 5, 11 (2012), 1627–1637.

[11] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2013. Don't Thrash: How to Cache Your Hash on Flash. In *Fifth Workshop on Massive Data Algorithmics (MASSIVE 2013)*. Sophia Antipolis, France.

[12] 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

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

[14] 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.

[15] 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.

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

[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] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 30–39.

[20] Yuvaraj Chesetti, Benwei Shi, Jeff M. Phillips, and Prashant Pandey. 2025. Zombie Hashing: Reanimating Tombstones in a Graveyard. *Proc. ACM Manag. Data* 2, 4 (2025), 192:1–192:28. https://doi.org/10.1145/3725424

[21] 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.

[22] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[23] 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.

[24] 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.

[25] Niv Dayan, Ioana Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proc. ACM Manag. Data* 1, 2, Article 140 (jun 2023), 27 pages. https://doi.org/10.1145/3589285

[26] Niv Dayan, Ioana O. Bercea, and Rasmus Pagh. 2024. Aleph Filter: To Infinity in Constant Time. *CoRR* abs/2404.04703 (2024). https://doi.org/10.48550/ARXIV.2404.04703 arXiv:2404.04703

[27] Niv Dayan, Ioana-Oriana Bercea, and Rasmus Pagh. 2024. Aleph Filter: To Infinity in Constant Time. 17, 11 (Aug. 2024), 3644–3656. https://doi.org/10.14778/3681954.3682027

[28] Niv Dayan and Moshe Twitto. 2021. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

[29] 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.

[30] 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

[31] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. https://doi.org/10.48550/ARXIV.2103.02515

[32] 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

[33] Tomer Even, Guy Even, and Adam Morrison. 2022. Prefix Filter: Practically and Theoretically Better Than Bloom. *Proc. VLDB Endow.* 15, 7 (2022), 1311–1323. https://doi.org/10.14778/3523210.3523211

[34] 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.

[35] 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*. ACM, 75–88.

[36] Martin Farach-Colton, Rohan J. Fernandes, and Miguel A. Mosteiro. 2009. Bootstrapping a hop-optimal network in the weak sensor model. *ACM Transactions on Algorithms* 5, 4 (2009).

[37] 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.

[38] Michael T Goodrich, Daniel S Hirschberg, Michael Mitzenmacher, and Justin Thaler. 2011. Fully de-amortized cuckoo hashing for cache-oblivious dictionaries and multimaps. *arXiv preprint arXiv:1107.4378* (2011).

[39] Michael T Goodrich, Daniel S Hirschberg, Michael Mitzenmacher, and Justin Thaler. 2012. Cache-oblivious dictionaries and multimaps with negligible failure probability. In *Mediterranean Conference on Algorithms*. Springer, 203–218.

[40] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *ACM J. Exp. Algorithmics* 25, Article 1.5 (March 2020), 16 pages. https://doi.org/10.1145/3376122

[41] 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.

[42] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A Right-Optimized Write-Optimized File System. In *Proc. 13th USENIX Conference on File and Storage Technologies (FAST)*, Jiri Schindler and Erez Zadok (Eds.). 301–315.

[43] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: Write-Optimization in a Kernel File System. *Transactions on Storage—Special Issue on USENIX FAST 2015* 11, 4 (2015), 18:1–18:29.

[44] Donald E. Knuth. 1969. *Fundamental Algorithms* (first ed.). The Art of Computer Programming, Vol. 1. Addison-Wesley.

[45] Tobias Maier, Peter Sanders, and Robert Williger. 2019. Concurrent expandable AMQs on the basis of quotient filters. *arXiv preprint arXiv:1911.08374* (2019).

[46] Hunter McCoy, Steven A. Hofmeyr, Katherine A. Yelick, and Prashant Pandey. 2023. High-Performance Filters for GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 160–173. https://doi.org/10.1145/3572848.3577507

[47] Hunter McCoy, Steven A. Hofmeyr, Katherine A. Yelick, and Prashant Pandey. 2023. Singleton Sieving: Overcoming the Memory/Speed Trade-Off in Exascale κ-mer Analysis. In *SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2023, Seattle, WA, USA, May 31 - June 2, 2023*, Jonathan W. Berry, David B. Shmoys, Lenore Cowen, and Uwe Naumann (Eds.). SIAM, 213–224. https://doi.org/10.1137/1.9781611977714.19

[48] 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 (SODA)*. 823–829.

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

[50] 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.

[51] Prashant Pandey, Michael A. Bender, Alex Conway, Martin Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2023. IcebergHT: High Performance Hash Tables Through Stability and Low Associativity. *Proc. ACM Manag. Data* 1, 1 (2023), 47:1–47:26. https://doi.org/10.1145/3588727

[52] 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.

[53] 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, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 775–787. https://doi.org/10.1145/3035918.3035963

[54] 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.

[55] 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 *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1386–1399. https://doi.org/10.1145/3448016.3452841

[56] Prashant Pandey, Martín Farach-Colton, Niv Dayan, and Huanchen Zhang. 2024. Beyond Bloom: A Tutorial on Future Feature-Rich Filters. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 636–644. https://doi.org/10.1145/3626246.3654681

[57] 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.

[58] 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.

[59] W Wesley Peterson. 1957. Addressing for random-access storage. *IBM journal of Research and Development* 1, 2 (1957), 130–146.

[60] 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.

[61] 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).

[62] RocksDB 2014. RocksDB. rocksdb.org. Viewed April 19, 2014.

[63] Moustafa Shokrof, C Titus Brown, and Tamer A Mansour. 2021. MQF and buffered MQF: Quotient filters for efficient storage of k-mers with their counts and metadata. *BMC bioinformatics* 22 (2021), 1–14.

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

[65] Hancheng Wang, Haipeng Dai, Rong Gu, Youyou Lu, Jiaqi Zheng, Jingsong Dai, Shusen Chen, Zhiyuan Chen, Shuaituan Li, and Guihai Chen. 2024. Wormhole Filters: Caching Your Hash on Persistent Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 456–471. https://doi.org/10.1145/3627703.3629590

[66] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. *Proc. VLDB Endow.* 13, 2 (2019), 197–210. https://doi.org/10.14778/3364324.3364333

[67] 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.

[68] Robert Williger and Tobias Maier. 2019. *Concurrent dynamic quotient filters: Packing fingerprints into atomics.* Ph.D. Dissertation. Karlsruher Institut für Technologie (KIT).

[69] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A Bender, et al. 2017. Writes wrought right, and other adventures in file system optimization. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–26.

[70] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing Every Operation in a Write-Optimized File System. In *Proc. 14th USENIX Conference on File and Storage Technologies (FAST).*

[71] Zhijian Yuan, Jiajia Miao, Yan Jia, and Le Wang. 2008. Counting data stream based on improved counting Bloom filter. In *Web-Age Information Management, 2008. WAIM'08. The Ninth International Conference on.* IEEE, 512–519.

[72] 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.

Received July 2025; revised October 2025; accepted November 2025; revised July 2025; revised October 2025; accepted November 2025