A General-Purpose Counting Filter: Making Every Bit Count

Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro
Stony Brook University
Stony Brook, NY, USA
{ppandey, bender, rob, rob.patro}@cs.stonybrook.edu

ABSTRACT

Approximate Membership Query (AMQ) data structures, such as the Bloom filter, quotient filter, and cuckoo filter, have found numerous applications in databases, storage systems, networks, computational biology, and other domains. However, many applications must work around limitations in the capabilities or performance of current AMQs, making these applications more complex and less performant. For example, many current AMQs cannot delete or count the number of occurrences of each input item, take up large amounts of space, are slow, cannot be resized or merged, or have poor locality of reference and hence perform poorly when stored on SSD or disk.

This paper proposes a new general-purpose AMQ, the counting quotient filter (CQF). The CQF supports approximate membership testing and counting the occurrences of items in a data set. This general-purpose AMQ is small and fast, has good locality of reference, scales out of RAM to SSD, and supports deletions, counting (even on skewed data sets), resizing, merging, and highly concurrent access. The paper reports on the structure's performance on both manufactured and application-generated data sets.

In our experiments, the CQF performs in-memory inserts and queries up to an order-of-magnitude faster than the original quotient filter, several times faster than a Bloom filter, and similarly to the cuckoo filter, even though none of these other data structures support counting. On SSD, the CQF outperforms all structures by a factor of at least 2 because the CQF has good data locality.

The CQF achieves these performance gains by restructuring the metadata bits of the quotient filter to obtain fast lookups at high load factors (i.e., even when the data structure is almost full). As a result, the CQF offers good lookup performance even up to a load factor of 95%. Counting is essentially free in the CQF in the sense that the structure is comparable or more space efficient even than non-counting data structures (e.g., Bloom, quotient, and cuckoo filters).

The paper also shows how to speed up CQF operations by using new x86 bit-manipulation instructions introduced in Intel's Haswell line of processors. The restructured metadata transforms many quotient filter metadata operations into rank-and-select bit-vector operations. Thus, our efficient implementations of rank and select may be useful for other rank-and-select-based data structures.

1. INTRODUCTION

Approximate Membership Query (AMQ) data structures maintain a probabilistic representation of a set or multiset, saving space by allowing queries occasionally to return a false-positive. Examples of AMQ data structures include Bloom filters [7], quotient filters [5], cuckoo filters [17], and frequency-estimation data structures [12]. AMQs have become one of the primary go-to data structures in systems builders' toolboxes [9].

AMQs are often the computational workhorse in applications, but today's AMQs are held back by the limited number of operations they support, as well as by their performance. Many systems based on AMQs (usually Bloom filters) use designs that are slower, less space efficient, and significantly more complicated than necessary in order to work around the limited functionality provided by today's AMQ data structures.

For example, many Bloom-filter-based applications work around the Bloom filter's inability to delete items by organizing their processes into epochs; then they throw away all Bloom filters at the end of each epoch. Storage systems, especially log-structured merge (LSM) tree [25] based systems [4,32] and deduplication systems [14, 15, 34], use AMQs to avoid expensive queries for items that are not present. These storage systems are generally forced to keep the AMQ in RAM (instead of on SSD) to get reasonable performance, which limit their scalability. Many tools that process DNA sequences use Bloom filters to detect erroneous data (erroneous subsequences in the data set) but work around the Bloom filter's inability to count by using a conventional hash table to count the number of occurrences of each subsequence [24, 29]. Moreover, these tools use a cardinality-estimation algorithm to approximate the number of distinct subsequences a priori to workaround the Bloom filter's inability to dynamically resize [20].

As these examples show, four important shortcomings of Bloom filters (indeed, most production AMQs) are (1) the inability to delete items, (2) poor scaling out of RAM, (3) the inability to resize dynamically, and (4) the inability to count the number of times each input item occurs, let alone support skewed input distributions (which are so common in DNA sequencing and other applications [24, 33]).

Many Bloom filter variants in the literature [30] try to overcome the drawbacks of the traditional Bloom filter. The counting Bloom filter [18] can count and delete items, but does not support skewed distributions and uses much more space than a regular Bloom filter (see Section 4). The spectral Bloom filter [11] solves the space problems of the counting Bloom filter, but at a high computational cost. The buffered Bloom filter [10] and forest-structured Bloom filter [23] use in-RAM buffering to scale out on flash, but still require several random reads for each query. The scalable Bloom filter [3] solves the problem of resizing the data structure by maintaining a series of Bloom filters, but queries become slower because each Bloom filter must be tested for the presence of the item. See

Section 2 for a thorough discussion of Bloom filter variants and other AMQs.

Counting, in particular, has also become important to AMQ structures, primarily as a tool for supporting deletion. The counting Bloom filter [18] was originally introduced to add support for deletion to the Bloom filter, albeit at a high space cost. More recently, the cuckoo [17] and quotient filters [5] support deletion by allowing each item to be duplicated a small number of times. However, many real data sets have a Zipfian distribution [13] and therefore contain many duplicates, so even these data structures can have poor performance or fail entirely on real data sets.

Full-featured high-performance AMQs. More generally, as AMQs have become more widely used, applications have placed more performance and feature demands on them. Applications would benefit from a general purpose AMQ that is small and fast, has good locality of reference (so it can scale out of RAM to SSD), and supports deletions, counting (even on skewed data sets), resizing, merging, and highly concurrent access. This paper proposes a new AMQ data structure that that has all these benefits. The paper reports on the structure's performance on both manufactured and application-generated data sets.

Results

This paper formalizes the notion of a *counting filter*, an AMQ data structure that counts the number of occurrences of each input item, and describes the counting quotient filter (CQF), a space-efficient, scalable, and fast counting filter that offers good performance on arbitrary input distributions, including highly skewed distributions.

We compare the CQF to the fastest and most popular counting data structure, the counting Bloom filter (CBF) [17], and to three (noncounting) approximate membership data structures, the Bloom [7], quotient [5], and cuckoo [17] filters. We perform comparisons in RAM and on disk, and using data sets generated uniformly at random and according to a Zipfian distribution. Our evaluation results are summarized in Table 1.

The CQF does counting for "free." In other words, the CQF is almost always faster and smaller than other AMQs, even though most AMQs can only indicate whether an item is present, not the number of times the item has been seen. In particular, the CQF is always smaller than the cuckoo filter, the quotient filter, and a count-min-sketch configured to match the CQF's error rate. The CQF is smaller than the Bloom filter, counting Bloom filter, and spectral Bloom filter for most practical configurations. The CQF is faster than every other AMQ and counting data structure we evaluated, except the cuckoo filter, which has comparable RAM performance.

The CQF handles skewed inputs efficiently. The CQF is actually smaller and faster on skewed inputs than on uniformly random inputs. On Zipfian inputs, the CQF is about an order-of-magnitude faster than the counting Bloom filter. We have also benchmarked the CQF on real-world data sets, and gotten similar performance.

The CQF scales well out of RAM. When stored on SSD, each insert or query in a CQF requires $O(1)$ I/Os, so that operations on an SSD-resident CQF are up to an order of magnitude faster than with other data structures.

The CQF supports deletes, merges, and resizing. These features make it easier to design applications around the CQF than other counting data structures and AMQs.

Contributions

The CQF uses three techniques to deliver good performance with small space:

- The CQF embeds variable-sized counters into the quotient filter in a cache-efficient way. Our embedding ensures that the CQF never takes more space than the quotient filter. Resizability and variable-sized counters enable the CQF to use substantially less space for Zipfian and other skewed input distributions.

- The CQF restructures the quotient filter's metadata scheme to speed up lookups when the data structure is nearly full. This enables the CQF to save space by supporting higher load factors than the QF. The original quotient filter performs poorly above 75% occupancy, but the CQF provides good lookup performance up to 95% occupancy. The metadata scheme also saves space by reducing the number of metadata bits per item.

- The CQF can take advantage of new x86 bit-manipulation instructions introduced in Intel's Haswell line of processors to speed up quotient-filter-metadata operations. Our metadata scheme transforms many quotient-filter-metadata operations into rank-and-select bit-vector operations. Thus, our efficient implementations of rank and select may be useful for other rank-and-select-based data structures. Other counting filters, such as the counting Bloom filter and cuckoo filter, do not have metadata and hence cannot benefit from these optimizations.

2. AMQ AND COUNTING STRUCTURES

2.1 AMQ data structures

Approximate Membership Query (AMQ) data structures provide a compact, lossy representation of a set or multiset. AMQs support INSERT(x) and QUERY(x) operations, and may support other operations, such as delete. Each AMQ is configured with an allowed false-positive rate, δ. A query for x is guaranteed to return true if x has ever been inserted, but the AMQ may also return true with probability δ even if x has never been inserted. Allowing false-positives enables the AMQ to save space.

For example, the classic AMQ, the Bloom filter [7], uses about $-\frac{\log_2 \delta}{\ln 2} \approx -1.44 \log_2 \delta$ bits per element. For common values of δ, e.g., in the range of 1/50 to 1/1000, the Bloom filter uses about one to two bytes per element.

The Bloom filter is common in database, storage, and network applications. It is typically used to avoid expensive searches on disk or queries to remote hosts for nonexistent items [9].

The Bloom filter maintains a bit vector A of length m. Every time an item x is inserted, the Bloom filter sets $A[h_i(x)] = 1$ for $i = 1, \ldots, k$, where k is a configurable parameter, and $h_1, \ldots, h_k$ are hash functions. A query for x checks whether $A[h_i(x)] = 1$ for all $i = 1, \ldots, k$. Assuming the Bloom filter holds at most n distinct items, the optimal choice for k is $\frac{m}{n} \ln 2$, which yields a false-positive rate of $2^{-\frac{m}{n} \ln 2}$. A Bloom filter cannot be resized—it is constructed for a specific false-positive rate δ and set size n.

The Bloom filter has inspired numerous variants [3, 8, 10, 14, 18, 23, 27, 28]. The counting Bloom filter (CBF) [18] replaces each bit in the Bloom filter with a c-bit saturating counter. This enables the CBF to support deletes, but increases the space by a factor of c. The scalable Bloom filter [3] uses multiple Bloom filters to maintain the target false-positive rate δ even when n is unknown.

The quotient filter [5] does not follow the general Bloom-filter design. It supports insertion, deletion, lookups, resizing, and merging. The quotient filter hashes items to a p-bit fingerprint and uses the upper bits of the fingerprint to select a slot in a table, where it stores the lower bits of the fingerprint. It resolves collisions using a variant of linear probing that maintains three metadata bits per slot. During an insertion, elements are shifted around, similar to insertion sort with gaps [6], so that elements are always stored in order of increasing hash value.

The quotient filter uses slightly more space than a Bloom filter, but much less than a counting Bloom filter, and delivers speed comparable to a Bloom filter. The quotient filter is also much more

Data Structure	QF	RSQF	CQF	CF	BF
Uniform random inserts per sec	11.12	12.06	11.19	14.25	2.84
Uniform successful lookups per sec	3.39	17.13	11.16	18.87	2.55
Uniform random lookups per sec	5.71	25.09	25.93	18.84	11.56
Bits per element	12.631	11.71	11.71	12.631	12.984

(a) In-memory uniform-random performance (in millions of operations per second).

Data Structure	CQF	CBF
Zipfian random inserts per sec	13.43	0.27
Zipfian successful lookups per sec	19.77	2.15
Uniform random lookups per sec	43.68	1.93
Bits per element	11.71	337.584

(b) In-memory Zipfian performance (in millions of operations per second).

Data Structure	RSQF	CQF	CF
Uniform random inserts per sec	69.05	68.30	42.20
Uniform successful lookups per sec	35.42	34.49	12.26
Uniform random lookups per sec	31.32	29.87	11.09
Bits per element	11.71	11.71	12.631

(c) On-SSD uniform-random performance (in thousands of operations per second).

Table 1: Summary of evaluation results. All the data structures are configured for 1/512 false-positive rate. We compare the quotient filter (QF) [5], rank-and-select quotient filter (RSQF), counting quotient filter (CQF), cuckoo filter (CF) [17], Bloom filter (BF) [26], and counting Bloom filter (CBF) [31].

cache-friendly than the Bloom filter, and so offers much better performance when stored on SSD. One downside of the quotient filter is that the linear probing becomes expensive as the data structure becomes full—performance drops sharply after 60% occupancy. Geil has accelerated the QF by porting it to GPUs [19].

The cuckoo filter [17] is built on the idea of cuckoo hashing. Similar to a quotient filter, the cuckoo filter hashes each item to a p-bit fingerprint, which is divided into two parts, a slot index i and a value f to be stored in the slot. If slot i (called the *primary* slot) is full then the cuckoo filter attempts to store f in slot $i \oplus h(f)$ (the *secondary* slot), where h is a hash function. If both slots are full, then the cuckoo filter kicks another item out of one of the two slots, moving it to its alternate location. This may cause a cascading sequence of kicks until the data structure converges on a new stable state. The cuckoo filter supports fast lookups, since only two locations must be examined. Inserts become slower as the structure becomes fuller, and in fact inserts may fail if the number of kicks during a single insert exceeds a specified threshold (500 in the author's reference implementation). Lookups in the cuckoo filter are less cache-friendly than in the quotient filter, since two random locations may need to be inspected. Inserts in the cuckoo filter can have very poor cache performance as the number of kicks grows, since each kick is essentially a random memory access.

2.2 Counting data structures

Counting data structures fall into two classes: counting filters and frequency estimators.

They support INSERT, QUERY, and DELETE operations, except a query for an item x returns the number of times that x has been inserted. A counting filter may have an error rate δ. Queries return true counts with probability at least $1-\delta$. Whenever a query returns an incorrect count, it must always be greater than the true count.

The counting Bloom filter is an early example of a counting filter. The counting Bloom filter was originally described as using fixed-sized counters, which means that counters could saturate. This could cause the counting Bloom filter to undercount. Once a counter saturated, it could never be decremented by any future delete, and so after many deletes, a counting Bloom filter may no longer meet its error limit of δ. Both these issues can be fixed by rebuilding the entire data structure with larger counters whenever one of the counters saturates.

The d-left Bloom filter [8] offers the same functionality as a counting Bloom filter and uses less space, generally saving a factor of two or more. It uses d-left hashing and gives better data locality. However, it is not resizable and the false-positive rate depends upon the block size used in building the data structure.

The spectral Bloom filter [11] is another variant of the counting Bloom filter that is designed to support skewed input distributions space-efficiently. The spectral Bloom filter saves space by using variable-sized counters. It offers significant space savings, compared to a plain counting Bloom filter, for skewed input distributions. However, like other Bloom filter variants, the spectral Bloom filter has poor cache-locality and cannot be resized.

The quotient filter also has limited support for counting, since supports inserting the same fingerprint multiple times. However, inserting the same item more than a handful of times can cause linear probing to become a bottleneck, degrading performance.

The cuckoo filter can also support a small number of duplicates of some items. In the authors' reference implementation, each slot can actually hold 4 values, so the system can support up to 8 duplicates, although its not clear how this will impact the probability of failure during inserts of other items. One could add counting to the cuckoo filter by associating a counter with each fingerprint, but this would increase the space usage.

Frequency-estimation data structures offer weaker guarantees on the accuracy of counts, but can use substantially less space. Frequency-estimation data structures have two parameters that control the error in their counts: an accuracy parameter ε and a confidence parameter δ. The count returned by a frequency-estimation data structure is always greater than the true count. After M insertions, the probability that a query returns a count that is more than εM larger than the true count is at most δ. For infrequently occurring items, the error term εM may dominate the actual number of times the item has occurred, so frequency-estimation data structures are most useful for finding the most frequent items.

The count-min sketch (CMS) [12] data structure is the most widely known frequency estimator. It maintains a $d \times w$ array A of uniform-sized counters, where $d = -\ln \delta$ and $w = e/\varepsilon$. To insert an item x, the CMS increments $A[i, h_i(x)]$ for $i = 1, \ldots, d$. For a query, it returns $\min_i A[i, h_i(x)]$. Deletions can be supported by decrementing the counters. As with a counting Bloom filter, the CMS can support arbitrarily large counts by rebuilding the structure whenever one of the counters saturates.

Note that we can use a CMS to build a counting filter by setting $\varepsilon = 1/n$, where n is an upper bound on the number of insertions to be performed on the sketch. However, as we will see in Section 4.1, this will be less space efficient than the counting quotient filter.

3. A RANK-AND-SELECT-BASED QUOTIENT FILTER

In this section we explain how to improve upon a quotient filter's

metadata representation and algorithms. We explain how to embed variable-sized counters into a quotient filter in Section 4.

The rank-and-select-based quotient filter (RSQF) improves the quotient filter's metadata scheme in three ways:

- It uses 2.125 metadata bits per slot, compared to the 3 metadata bits per slot used by the quotient filter.
- It supports faster lookups at higher load factors than the quotient filter. The quotient filter authors recommend filling the quotient filter to only 75% capacity due to poor performance above that limit. The RSQF performs well up to 95% capacity.
- The RSQF's metadata structure transforms most quotient filter metadata operations into bit vector *rank* and *select* operations. We show how to optimize these operations using new x86 bit-manipulation instructions.

The space savings from these optimizations make the RSQF more space efficient than the Bloom filter for false-positive rates less than $1/64$ and more space efficient than the cuckoo filter for all false-positive rates. In contrast, the original quotient filter is less space efficient than the cuckoo filter for all false-positive rates and the Bloom filter for false-positive rates larger than to 2^{-36}.

The performance optimizations make the RSQF several times faster than a Bloom filter and competitive with the cuckoo filter. The original quotient filter was comparable to a Bloom filter in speed and slower than the cuckoo filter.

3.1 Rank-and-select-based metadata scheme

We first describe a simple rank-and-select-based quotient filter that requires only 2 bits of metadata per slot, but is not cache friendly and has $O(n)$ lookups and inserts. We then describe how to solve these problems by organizing the metadata into blocks of 64 slots and adding an extra 8 bits of metadata to each block (increasing the overall metadata to 2.125 bits per slot).

The rank-and-select-based quotient filter (RSQF) implements an approximate-membership-query data structure by storing a compact lossless representation of the multiset $h(S)$, where $h : \mathcal{U} \to \{0, \ldots, 2^p - 1\}$ is a hash function and S is a multiset of items drawn from a universe U. As in the original quotient filter, the RSQF sets $p = \log_2 \frac{n}{\delta}$ to get a false-positive rate δ while handling up to n insertions (see the original quotient filter paper for the analysis [5]).

The rank-and-select-based quotient filter divides $h(x)$ into its first q bits, which we call the *quotient* $h_0(x)$, and its remaining r bits, which we call the *remainder* $h_1(x)$. The rank-and-select-based quotient filter maintains an array Q of 2^q r-bit slots, each of which can hold a single remainder. When an element x is inserted, the quotient filter attempts to store the remainder $h_1(x)$ in the *home slot* $Q[h_0(x)]$. If that slot is already in use, then the rank-and-select-based quotient filter uses a variant of linear probing, described below, to find an unused slot and stores $h_1(x)$ there.

Throughout this paper, we say that slot i in a quotient filter is *occupied* if the quotient filter contains an element x such that $h_0(x) = i$. We say that a slot is *in use* if there is a remainder stored in the slot. Otherwise the slot is *unused*. Because of the quotient filter's linear-probing scheme, a slot may be in use even if it is not occupied. However, since the quotient filter always tries to put remainders in their home slots (which are necessarily occupied) and only shifts a reminder when it is pushed out by another remainder, occupied slots are always in use.

The RSQF also maintains two metadata bit vectors that enable the quotient filter to determine which slots are currently in use and to determine the home slot of every remainder stored in the filter. Together, these two properties enable the RSQF to enumerate all the hash values that have been inserted into the filter.

The quotient filter makes this possible by maintaining a small amount of metadata and a few invariants:

Figure 1: A simple rank-and-select-based quotient filter. The colors are used to group slots that belong to the same run, along with the runends bit that marks the end of that run and the occupieds bit that indicates the home slot for remainders in that run.

Figure 2: Procedure for computing offset O_j given O_i.

- The quotient filter maintains an *occupieds* bit vector of length 2^q. The RSQF Q sets $Q.\text{occupieds}[b]$ to 1 if and only if there is an element $x \in S$ such that $h_0(x) = b$.
- For all $x, y \in S$, if $h_0(x) < h_0(y)$, then $h_1(x)$ is stored in an earlier slot than $h_1(y)$.
- If $h_1(x)$ is stored in slot s, then $h_0(x) \leq s$ and there are no unused slots between slot $h_0(x)$ and slot s, inclusive.

These invariants imply that remainders of elements with the same quotient are stored in consecutive slots. We call such a sequence of slots a *run*. After each insert, the quotient filter shifts elements as necessary to maintain the invariants.

We now describe the second piece of metadata in a rank-and-select-based quotient filter.

- The quotient filter maintains a *runends* bit vector of length 2^q. The RSQF Q sets $Q.\text{runends}[b]$ to 1 if and only if slot b contains the last remainder in a run.

As shown in the Figure 1, the bits set in the occupieds vector and the runends vector are in a one-to-one correspondence. There is one run for each slot b such that there exists an x such that $h_0(x) = b$, and each such run has an end. Runs are stored in the order of the home slots to which they correspond.

This correspondence enables us to reduce many quotient-filter-metadata operations to bit vector rank and select operations. Given a bit vector B, RANK(B, i) returns the number of 1s in B up to position i, i.e., RANK$(B, i) = \sum_{j=0}^{i} B[j]$. Select is essentially the inverse of rank. SELECT(B, i) returns the index of the ith 1 in B.

These operations enable us to find the run corresponding to any quotient $h_0(x)$; see Algorithm 1. If $Q.\text{occupieds}[h_0(x)] = 0$, then no such run exists. Otherwise, we first use RANK to count the number t of slots $b \leq h_0(x)$ that have their occupieds bit set. This is the number of runs corresponding to slots up to and including slot $h_0(x)$. We then use SELECT to find the position of the tth runend bit, which tells us where the run of remainders with quotient $h_0(x)$ ends. We walk backwards through the remainders in that run. Since elements are always shifted to the right, we can stop walking backwards if we ever pass slot $h_0(x)$ or if we reach another slot that is marked as the end of a run.

Algorithm 1 Algorithm for determining whether x may have been inserted into a simple rank-and-select-based quotient filter.

```
1: function MAY_CONTAIN(Q, x)
2:     b ← h_0(x)
3:     if Q.occupieds[b] = 0 then
4:         return 0
5:     t ← RANK(Q.occupieds, b)
6:     ℓ ← SELECT(Q.runends, t)
7:     v ← h_1(x)
8:     repeat
9:         if Q.remainders[ℓ] = v then
10:            return 1
11:        ℓ ← ℓ − 1
12:    until ℓ < b or Q.runends[ℓ] = 1
13:    return false
```

Algorithm 2 Algorithm for inserting x into a rank and select quotient filter.

```
1: function FIND_FIRST_UNUSED_SLOT(Q, x)
2:     r ← RANK(Q.occupieds, x)
3:     s ← SELECT(Q.runends, r)
4:     while x ≤ s do
5:         x ← s + 1
6:         r ← RANK(Q.occupieds, x)
7:         s ← SELECT(Q.runends, s)
8:     return x

9: function INSERT(Q, x)
10:    r ← RANK(Q.occupieds, h_0(x))
11:    s ← SELECT(Q.runends, r)
12:    if h_0(x) > s then
13:        Q.remainders[h_0(x)] ← h_1(x)
14:        Q.runends[h_0(x)] ← 1
15:    else
16:        s ← s + 1
17:        n ← FIND_FIRST_UNUSED_SLOT(Q, s)
18:        while n > s do
19:            Q.remainders[n] ← Q.remainders[n − 1]
20:            Q.runends[n] ← Q.runends[n − 1]
21:            n ← n − 1
22:        Q.remainders[s] ← h_1(x)
23:        if Q.occupieds[h_0(x)] = 1 then
24:            Q.runends[s − 1] ← 0
25:        Q.runends[s] ← 1
26:    Q.occupieds[h_0(x)] ← 1
27:    return
```

Algorithm 2 shows the procedure for inserting an item x. The algorithm uses rank and select to find the end of the run corresponding to quotient $h_0(x)$. If slot $h_0(x)$ is not in use, then the result of the rank-and-select operation is an index less than $h_0(x)$, in which case the algorithm stores $h_1(x)$ in slot $h_0(x)$. Otherwise the algorithm shifts remainders (and runends bits) to the right to make room for the new item, inserts it, and updates the metadata.

As with the original quotient filter, the false-positive rate of the RSQF is at most 2^{-r}. The RSQF also supports enumerating all the hashes currently in the filter, and hence can be resized by building a new table with $2^{q'}$ slots, each with a remainder of size $p - q'$ bits, and then inserting all the hashes from the old filter into the new one. RSQFs can be merged in a similar way.

This simple quotient filter design demonstrates the architecture of the RSQF and requires only two metadata bits per slot, but has two problems. First, rank and select on bit vectors of size n requires $O(n)$ time in the worst case. We would like to perform lookups without having to scan the entire data structure. Second, this design is not cache friendly. Each lookup requires accessing the occupieds bit vector, the runends bit vector, and the array of remainders. We prefer to reorganize the data so that most operations access only a small number of nearby memory locations.

Offsets. To compute the position of a runend without scanning the entire occupieds and runends bit vectors, the RSQF maintains an *offsets* array. The offset O_i of slot i is

$$O_i = \text{SELECT}(Q.\text{runends}, \text{RANK}(Q.\text{occupieds}, i)) - i$$

or 0 if this value is negative, which occurs whenever slot i is unused. Intuitively, O_i is the distance from slot i to the slot containing the runend corresponding to slot i. Thus, if we know O_i, we can immediately jump to the location of the run corresponding to slot i, and from there we can perform a search, insert, delete, etc.

To save space, the RSQF stores O_i for only every 64th slot, and computes O_j for other slots using the algorithm from Figure 2. To compute O_j from O_i, the RSQF uses RANK to count the number d of occupied slots between slots i and j, and then uses SELECT to find the dth runend after the end of the run corresponding to slot i.

Maintaining the array of offsets is inexpensive. Whenever the RSQF shifts elements left or right (as part of a delete or insert), it updates the stored O_i values. Only O_i values in the range of slots that were involved in the shift need to be updated.

Computing O_j from the nearest stored O_i is efficient because the algorithm needs to examine only the occupieds bit vector between indices i and j and the runends bit vector between indices $i + O_i$ and $j + O_j$. Since the new quotient filter stores O_i for every 64th slot, the algorithm never needs to look at more than 64 bits of the occupieds bit vector. And only needs to look at $O(q)$ bits in the runends bit vector based on the following theorem.

offset	occupieds	runends	remainders
8	64	64	64r

Figure 3: Layout of a rank-and-select-based-quotient-filter block. The size of each field is specified in bits.

THEOREM 1. *The length of the longest contiguous sequence of in-use slots in a quotient filter with 2^q slots and load factor α is $O(\frac{\ln 2^q}{\alpha - \ln \alpha - 1})$ with high probability.*

The theorem (from the original QF paper [5]) bounds the worst case. On average, the RSQF only needs to examine $j - i < 64$ bits of the runends bit vector because the average number of items associated with a slot is less than 1.

This theorem also shows that the offsets are never more than $O(q)$, so we can store entries in the offsets array using small integers. Our prototype implementation stores offsets as 8-bit unsigned ints. Since it stores one offset for every 64 slots, this increases the metadata overhead to a total of 2.125 bits per slot.

Blocking the RSQF. To make the RSQF cache efficient, we break the occupieds, runends, offsets, and remainders vectors into blocks of 64 entries, which we store together, as shown in Figure 3. We use blocks of 64 entries so these rank and select operations can be transformed into efficient machine-word operations as described in the Section 3.2. Each block holds one offset, 64 consecutive bits from each bit vector and the corresponding 64 remainders. An operation on slot i loads the corresponding block, consults the offset field, performs a rank computation on the occupieds bits in the block, and then performs a select operation on the runends bits in the block. If the offset is large enough, the select operation may extend into subsequent blocks, but in all cases the accesses are sequential. The remainders corresponding to slot

i can then be found by tracing backwards from where the select computation completed.

For a false-positive rate of 2^{-r}, each block will have size of $64(r+2)+8$ bits. Thus a block is much smaller than a disk block for typical values of r, so quotient filter operations on an SSD require accessing only a small number of consecutive blocks, and usually just one block.

3.2 Fast x86 rank and select

The blocked RSQF needs to perform a RANK operation on a 64-bit portion of the occupieds bitvector and a SELECT operation on a small piece of the runends vector. We now describe how to implement these operations efficiently on 64-bit vectors using the x86 instruction set.

To implement SELECT, we use the PDEP and TZCNT instructions added to the x86 instruction set with the Haswell line of processors. PDEP deposits bits from one operand in locations specified by the bits of the other operand. If $x = \text{PDEP}(v, m)$, then the ith bit of x is given by

$$x_i = \begin{cases} v_j & \text{if } m_i \text{ is the } j\text{th 1 in } m, \\ 0 & \text{otherwise.} \end{cases}$$

TZCNT returns the number of trailing zeros in its argument. Thus, we can implement select on 64-bit vectors as

$$\text{SELECT}(v, i) = \text{TZCNT}(\text{PDEP}(2^i, v))$$

POPCOUNT returns the number of bits set in its argument. We implement $\text{RANK}(v, i)$ on 64-bit vectors using the widely-known mask-and-popcount method [21]:

$$\text{RANK}(v, i) = \text{POPCOUNT}(v \ \& \ (2^i - 1))$$

We evaluate the performance impact of these optimizations in Section 6.

3.3 Lookup performance

We now explain why the RSQF offers better lookup performance than the original QF at high load factors.

Lookups in any quotient filter involve two steps: finding the start of the target run, and scanning through the run to look for the queried value. In both the original QF and the RSQF, runs have size $O(1)$ on average and $O(\log n / \log \log n)$ with high probability, and the RSQF does nothing to accelerate the process of scanning through the run.

The RSQF does accelerate the process of finding the start of the run. The original QF finds the target run by walking through the slots in the target cluster, one-by-one. Both the average and worst-case cluster sizes grow as the load factor increases, so processing each slot's metadata bits one at a time can become expensive. The RSQF, however, processes these metadata bits 64 at a time by using our efficient rank-and-select operations.

At high load factors, this can yield tremendous speedups, since it converts 64 bit-operations into single word-operations. At low load factors, the speedup is not so great, since the QF and RSQF are both doing essentially $O(1)$ operations, albeit the QF is doing bit operations and the RSQF is doing word operations.

Section 6 presents experimental results showing the performance impact of this redesign.

3.4 Space analysis

Table 2 gives the space used by several AMQs. Our rank-and-select-based quotient filter uses fewer metadata bits than the original quotient filter, and is faster for higher load factors (see Section 6). The RSQF is more space efficient than the original quotient filter, the cuckoo filter, and, for false-positive rates less than 1/64,

Filter	Bits per element
Bloom filter	$\frac{\log_2 1/\delta}{\ln 2}$
Cuckoo filter	$\frac{3 + \log_2 1/\delta}{\alpha}$
Original QF	$\frac{3 + \log_2 1/\delta}{\alpha}$
RSQF	$\frac{2.125 + \log_2 1/\delta}{\alpha}$

Table 2: Space usage of several AMQs. Here, δ is the false-positive rate and α is the load factor. The original quotient filter was less space efficient than the cuckoo filter because it only supports α up to 0.75, whereas the cuckoo filter supports α up to 0.95. The RSQF is more efficient than the cuckoo filter because it has less overhead and supports load factors up to 0.95.
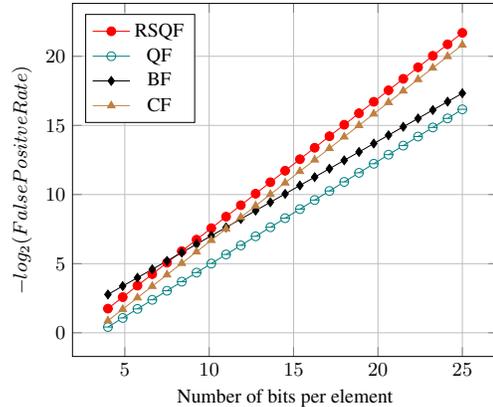
Figure 4: Number of bits per element for the RSQF, QF, BF, and CF. The RSQF requires less space than the CF amd less space than the BF for any false-positive rate less than $1/64$. (Higher is better)

the Bloom filter. Even for large false-positive rates, the RSQF never uses more than 1.55 more bits per element than a Bloom filter.

Figure 4 shows the false-positive rate of these data structures as a function of the space usage, assuming that each data structure uses the recommended load factor, i.e., 100% for the BF, 95% for the RSQF and CF, and 75% for the QF. In order to fairly compare the space requirements of data structures with different recommended load factors, we normalize all the data structures' space requirements to bits per element.

3.5 Enumeration, resizing, and merging

Since quotient filters represent a multi-set S by losslessly representing the set $h(S)$, it supports enumerating $h(S)$. Everything in this section applies to the original quotient filter, our rank-and-select-based quotient filter, and, with minor modifications, to the counting quotient filter.

The time to enumerate $h(S)$ is proportional to 2^q, the number of slots in the QF. If the QF is a constant-fraction full, enumerating $h(S)$ requires $O(n)$ time, where n is the total number of items in the multi-set S. The enumeration algorithm performs a linear scan of the slots in the QF, and hence is I/O efficient for a QF or RSQF stored on disk or SSD.

The QF's enumeration ability makes it possible to resize a filter, similar to resizing any hash table. Given a QF with 2^q slots and r-bit remainders and containing n hashes, we can construct a new, empty filter with $2^{q'} \geq n$ slots and $q + r - q'$-bit remainders. We then enumerate the hashes in the original QF and insert them into the new QF. As with a hash table, the time required to resize the QF is proportional to the size of the old filter plus the size of the new

filter. Hence, as with a standard hash table, doubling a QF every time it becomes full will have $O(1)$ overhead to each insert.

Enumerability also enables us to merge filters. Given two filters representing $h(S_1)$ and $h(S_2)$, we can merge them by constructing a new filter large enough to hold $h(S_1 \cup S_2)$ and then enumerating the hashes in each input filter and inserting them into the new filter. The total cost of performing the merge is proportional to the size of the output filter, i.e., if the input filters have n_1 and n_2 elements, the time to merge them is $O(n_1 + n_2)$.

Merging is particularly efficient for two reasons. First, items can be inserted into the output filter in order of increasing hash value, so inserts will never have to shift any other items around. Second, merging requires only linear scans of the input and output filters, and hence is I/O-efficient when the filters are stored on disk.

4. COUNTING QUOTIENT FILTER

We now describe how to add counters to the RSQF to create the CQF. Our counter-embedding scheme maintains the data locality of the RSQF, supports variable-sized counters, and ensures that the CQF takes no more space than an RSQF of the same multiset. Thanks to the variable-size counters, the structure is space efficient even for highly skewed distributions, where some elements are observed frequently and others rarely.

Encoding counters. The RSQF counts elements in unary i.e., if a given remainder occurs k times in a run, then the RSQF just stores k copies of this remainder.

The CQF saves space by repurposing some of the slots to store counters instead of remainders. In the CQF, if a particular element occurs more than once, then the slots immediately following that element's remainder hold an encoding of the number of times that element occurs.

To make this scheme work, however, we need some way to determine whether a slot holds a remainder or part of a counter. The CQF distinguishes counters from remainders as follows. Within a run, the CQF stores the remainders in increasing order. Any time the value stored in a slot deviates from this strictly increasing pattern, that slot must hold part of an encoded counter. Thus, a deviation from the strictly increasing pattern acts as an "escape sequence" indicating that the CQF is using the next few slots to store an encoded counter rather than remainders.

Once the CQF decoder has recognized that a slot holds part of the counter for some remainder x, it needs to determine how many slots are used by that counter. We again use a form of escape sequence: The counter for remainder x is encoded so that no slot holding part of the counter will hold the value x. Thus, the end of the counter is marked by another copy of x.

This scheme also requires that we encode a counter value C into a sequence of slots so that the first slot following the first occurrence of x holds a value less than x. Thus, we simply encode C as described below, and then prepend a 0 to its encoding if it would otherwise violate this requirement.

There remains one wrinkle. For the remainder 0, it is not possible to encode its counter so that the first slot holding the counter has a value less than 0. Instead, we mark a counter for remainder 0 with a special "terminator"—two slots containing consecutive 0s. If a run contains two consecutive 0s, then everything between the first slot and the two consecutive 0s is an encoding for the number of 0 remainders in the run. Otherwise, the number of 0 remainders is recorded through repetition, as in the RSQF.

This last rule means that we cannot have two consecutive 0s anywhere else in the run, including in the encoding of any counters. To ensure this, we never use 0 in the encoding for other counters.

Thus, the counter C for remainder $x > 0$ is encoded as a sequence of r-bit values, but we cannot use the values 0 or x in the

Count	Encoding	Rules
$C = 1$	x	none
$C = 2$	x, x	none
$C > 2$	$x, c_{\ell-1}, \ldots, c_0, x$	$x > 0$
		$c_{\ell-1} < x$
		$\forall i \ c_i \neq x$
		$\forall i < \ell - 1 \ c_i \neq 0$
$C = 3$	$0, 0, 0$	$x = 0$
$C > 3$	$0, c_{\ell-1}, \ldots, c_0, 0, 0$	$x = 0$
		$\forall i \ c_i \neq 0$

Table 3: Encodings for C occurrences of remainder x in the CQF.
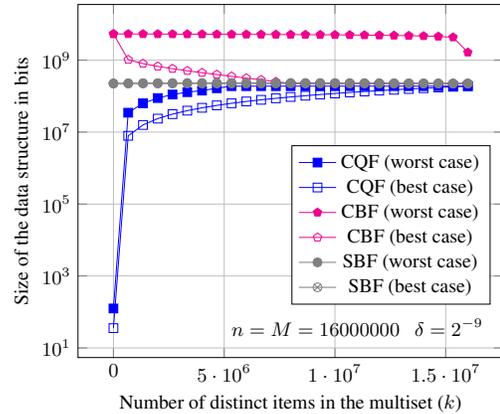
Figure 5: Space comparison of CQF, SBF, and CBF as a function of the number of distinct items. All data structures are built to support up to $n = 1.6 \times 10^7$ insertions with a false-positive rate of $\delta = 2^{-9}$.

encoding for C. Since we know $C \geq 3$ we achieve this by encoding $C - 3$ as $c_{\ell-1}, \ldots, c_0$ in base $2^r - 2$, where the symbols are $1, 2, \ldots, x-1, x+1, \ldots, 2^r - 1$, and prepend a 0 if $c_{\ell-1} \geq x$. Note that this requires $r \geq 2$ in order to have the base of the encoding be greater than 1, but this is not a serious constraint, since most applications require $r > 2$ in order to achieve their target false positive rate, anyway.

The counter C for remainder $x = 0$ is encoded as using base $2^r - 1$, since only 0 is disallowed in the counter encoding. Furthermore, since we know $C \geq 4$, we encode $C - 4$.

Table 3 summarizes the counter encoding used by the CQF.

As an example, a run consisting of

5 copies of **0**, *7* copies of **3**, and *9* copies of **8**

would be encoded as (**0**, *2*, **0**, **0** , **3**, *0*, *6*, **3** , **8**, *7*, **8**).

4.1 CQF Space Analysis

For data sets with no or few repeated items, the CQF uses essentially the same space as a RSQF, but never more. When the input distribution is skewed, then the CQF can use substantially less space than a RSQF because the CQF encodes the duplicate items much more efficiently.

The exact size of the CQF after M inserts depends on the distribution of the inserted items, but we can give an upper bound as follows. In the following, n is the total number of items to be inserted, k is the total number of distinct items to be inserted, and M is the number of item inserted so far.

When the data structure has r-bit slots, the encoding of an item that occurs C times consumes at most three slots plus $\left\lceil \frac{\log_2 C}{r-1} \right\rceil \leq$

$\frac{\log_2 C}{r-1} + 1$ slots for its counter. Thus, the total number of bits used to encode a remainder and its count is at most $4r + \frac{r}{r-1}\log_2 C \leq 4r + 2\log_2 C$, since $r \geq 2$. After inserting $k < n$ distinct items, there are at least k occupied slots, so r is at most $p - \log_2 k$. Since $p = \log_2 n/\delta$, this means that $r \leq \log_2 \frac{n}{\delta} - \log_2 k = \log_2 \frac{n}{k\delta}$. The total size of all the counters is maximized when all the distinct items occur an equal number of times, i.e., when $C = M/k$ for each item. Putting this together, we get the following space bound:

THEOREM 2. *Let Q be a CQF with capacity n and false-positive rate δ. Suppose we initially build Q with $s = 1$ slot and resize to double s whenever the number of used slots exceeds $0.95s$. Then, after performing M inserts consisting of k distinct items, the size of the CQF will be $O(k \log \frac{nM}{\delta k^2})$ bits. The worst case occurs when each item occurs an equal number of times.*

To understand this bound, consider the extremal cases when $k = 1$ and $k = M$. When $k = 1$, the CQF contains M instances of a single item. The space bound reduces to $O(\log \frac{nM}{\delta}) = O(\log \frac{n}{\delta} + \log M)$ bits. This is exactly the size of a single hash value plus the size of a counter holding the value M. At the other extreme, when $k = M$, the space bound simplifies to $O(M \log \frac{n}{\delta M}) = O(M(\log \frac{n}{\delta} - \log M))$, i.e., the CQF has $O(M)$ slots, each of size $\log \frac{n}{\delta} - \log M$, which is exactly the space bound for the RSQF.

Figure 5 gives bounds, as a function of the number k of distinct items, on the space usage of the counting quotient filter, counting Bloom filter, and spectral Bloom filter, for $n = M = 1.6 \times 10^7$ and $\delta = 2^{-9}$. As the graph shows, the worst-case space usage for the CQF is better than the best-case space usage of the other data structures for almost all values of k. Although it is difficult to see in the graph, the spectral Bloom filter uses slightly less space than the CQF when k is close to M. The counting Bloom filter's space usage is worst, since it stores the most counters and all counters have the same size—large enough to hold the count of the most frequent element in the data set. This is also why the counting Bloom filter's space usage improves slightly as the number of distinct items increases, and hence the count of the most frequent item decreases. The spectral Bloom filter (SBF) uses space proportional to a plain Bloom filter plus optimally-sized counters for all the elements. As a result, its space usage is largely determined by the Bloom filter and hence is independent of the input distribution. The CQF space usage is best when the input contains many repeated items, since the CQF can be resized to be just large enough to hold those items. Even in its worst case, its space usage is competitive with the best-case space usage of the other counting filters.

Comparison to count-min sketch. Given a maximum false-positive rate δ and an upper bound n on the number of items to be inserted, we can build a CMS-based counting filter by setting the CMS's parameter $\varepsilon = 1/n$. After performing $M \leq n$ inserts, consisting of $k \leq M$ distinct items, at least one counter must have value at least M/k. Since CMS uses uniform-sized counters, each counter must be at least $1 + \log \frac{M}{k}$ bits. Thus the total space used by the CMS must be at least $\Omega((1 + \log \frac{M}{k})n \ln \frac{1}{\delta})$ bits. One can use the geometric-arithmetic mean inequality to show that this is asymptotically never better than (and often worse than) the CQF space usage for all values of δ, k, M, and n.

4.2 Configuring the CQF

When constructing a Bloom filter, the user needs to preset the size of the array, and this size cannot change. In contrast, for a CQF, the only parameter that needs to be preset is the number of bits p output by the hash function h. The CQF can be dynamically resized, and resizing has no affect on the false-positive rate.

As Section 3.1 explains, the user derives p from from the error rate δ and the maximum possible number n of items; then the user sets $p = \lceil \log_2(n/\delta) \rceil$.

One of the major advantages of the CQF is that its space usage is robust to errors in estimating n. This is important because, in many applications, the user knows δ but not n. Since underestimating n can lead to a higher-than-acceptable false-positive rate, users often use a conservatively high estimate.

The space cost of overestimating n is much lower in the CQF than in the Bloom filter. In the Bloom filter, the space usage is linear in n. Thus, if the user overestimates n by, say, a factor of 2, then the Bloom filter will consume twice as much space as necessary. In the CQF, on the other hand, overestimating n by a factor of 2 causes the user to select a value of p, and hence the remainder size r, that is merely one bit larger than necessary. Since $r \approx \log_2(1/\delta)$, the relative cost of one extra remainder bit in each slot is small. For example, in typical applications requiring an approximately 1% false-positive rate, $r \approx 7$, so each slot contains at least 9 bits, and hence overestimating n by a factor of 2 increases the space usage of the CQF by at most 11%.

5. MULTI-THREADED QUOTIENT FILTERS

To implement a thread-safe counting quotient filter, we divide the CQF into regions of 4096 contiguous slots. The thread performing the insert operation locks two consecutive regions, the region in which the item hashes and the next one, before modifying the data structure. As explained in Section 3.1, existing remainders are shifted during an insert operation in order to put the incoming remainder in its home slot. Taking locks on two consecutive regions in the CQF avoids the corruption of the data structure in-case shifting overflows to the next region.

The above locking scheme scales well for data sets that are not very skewed. However, when a data set has a lot of repetitions, the above scheme can cause lock contention among insertion threads.

To avoid lock contention among insertion threads, each insertion thread maintains a small local counting quotient filter. During an insert operation, an insertion thread first tries to acquire a lock on the region where the item hashes in the main CQF. If it gets the lock in the first attempt, it inserts the item in the main CQF. Otherwise, it inserts the item in its local CQF. Once the local CQF fills up, the insertion thread dumps the local CQF into the main CQF. While dumping the local CQF, the insertion thread spins if it does not get the lock in the first attempt. This scheme amortizes the overhead of acquiring a lock and reduces the contention among multiple insertion threads.

The above locking scheme is appropriate when queries can tolerate slightly stale data, since some inserts may be delayed while they sit in their thread's local CQF. For example, the above scheme works for applications that have an insert phase followed by a query phase, which is common in computational biology and LSM-tree uses of AMQs.

6. EVALUATION

In this section we evaluate our implementations of the counting quotient filter (CQF) and the rank-and-select quotient filter (RSQF). The counting quotient filter is our main AMQ data structure that supports counting and the rank-and-select quotient filter is our other AMQ data structure, which strips out the counting ability in favor of slightly faster query performance.

We compare the counting quotient filter and rank-and-select quotient filter against four other AMQs: a state-of-the-art Bloom filter [26], Bender et al.'s quotient filter [5], Fan et al.'s cuckoo filter [16], and Vallentin's counting Bloom filter [31].

We evaluate each data structure on the two fundamental operations, insertions and queries. We evaluate queries both for items that are present and for items that are not present.

We address the following questions about how AMQs perform in RAM and on SSD:

1. How do the rank-and-select quotient filter (RSQF) and counting quotient filter (CQF) compare to the Bloom filter (BF), quotient filter (QF), and cuckoo filter (CF) when the filters are in RAM?

2. How do the RSQF and CQF compare to the CF when the filters reside on SSD?

We do a deep dive into how performance is affected by the data distribution, metadata organization, and low-level optimizations:

1. How does the CQF compare to the counting Bloom filter (CBF) for handling skewed data sets?

2. How does our rank-and-select-based metadata scheme help performance? (I.e., how does the RSQF compare to the QF?) We are especially interested in evaluating filters with occupancy higher than 60%, when the QF performance starts to degrade.

3. How much do the new x86 bit-manipulation instructions (PDEP and TZCNT) introduced in Intel's Haswell architecture contribute to performance improvements?

4. How does the CQF's insert speed scale with multiple threads?

5. How efficient is the average merge throughput when merging multiple counting quotient filters?

We also evaluate and address the following questions about the counting quotient filter when used with data sets from real-world applications:

1. How does the CQF performs when used with real-world data sets? We use data sets from k-mer counting (a sub-task of DNA sequencing) and the firehose benchmark, which simulates a network-event monitoring task, as our real-world applications.

6.1 Experiment Setup

We evaluate the performance of the data structures in terms of the *load factor* and *capacity*. The *capacity* of the data structure is the number of items that can be inserted without causing the data structure's false-positive rate to become too high (which turns out to be the number of elements that can be inserted when there are no duplicates). We define the *load factor* to be the ratio of the number of distinct items in the data structure to the capacity of the data structure. For most experiments, we report the performance on all operations as a function of the data structures' load factor, i.e., when the data structure's load factor is 5%, 10%, 15%, etc.

In all our experiments, the data structures were configured to have a false-positive rate of $1/512$. Experiments with other false-positive rates gave similar results.

All the experiments (except the multi-threaded experiments) were run on an Intel Skylake CPU (Core(TM) i5-6500 CPU @ 3.20GHz with 2 cores and 6MB L3 cache) with 8 GB of RAM and a 480GB Intel SSDSC2BW480A4 Serial ATA III 540 MB/s 2.5" SSD. Experiments were single-threaded, unless otherwise mentioned. The multi-threaded experiments were run on an Intel Skylake CPU (Core(TM) i7-6700HQ CPU @ 2.60GHz with 4 cores and 6MB L3 cache) with 32 GB RAM.

Microbenchmarks. The microbenchmarks measure performance on raw inserts and lookups and are performed as follows. We insert random elements into an empty data structure until its load factor is sufficiently high (e.g., 95%). We record the time required to insert every 5% of the items. After inserting each 5% of items, we measure the lookup performance for that load factor.

We perform experiments both for uniform and skewed data sets. We generate 64-bit hash values to be inserted or queried in the data structure.

We configured the BF and CBF to be as small as possible while still supporting the target false-positive rate and number of insertions to be performed in the experiment. The BF and CBF used the optimal number of hash functions for their size and the number of insertions to be performed.

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

For the on-SSD experiments, the data structures were allocated using mmap and the amount of in-memory cache was limited to 800MBs of RAM, leading to a RAM-size-to-filter-size ratio of roughly 1:2. Paging was handled by the OS. The point of the experiments was to evaluate the IO efficiency of the quotient filter and cuckoo filter. We omit the Bloom filter from the on-SSD experiments, because Bloom filters are known to have poor cache locality and run particularly slowly on SSDs [5].

We evaluated the performance of the counting filters on two different input distributions, uniformly random and Zipfian. We use a Zipfian distribution to evaluate the CQF's performance on realistic data distributions and its ability to handle large numbers of duplicate elements efficiently. We omit the Cuckoo filters from the Zipfian experiment, because they are not designed to handle duplicate elements.

We also evaluated the merge performance of the counting quotient filter. We created K (i.e., 2, 4, and 8) counting quotient filters and filled them to 95% load factor with uniformly random data. We then merged these counting quotient filters into a single counting quotient filter. While merging multiple counting quotient filters, we add the number of occupied slots in each input counting quotient filter and take the next closest power of 2 as the number of slots to create in the output counting quotient filter.

Application benchmarks. We also benchmarked the insert performance of the counting quotient filter with data sets from two real-world applications: k-mer counting [29, 33] and FireHose [22].

K-mer counting is often the first step in the analysis of DNA sequencing data. This helps to identify and weed out erroneous data. To remove errors, one counts the number of times each k-mer (essentially a k-gram over the alphabet A, C, T, G) occurs [29, 33]. These counts are used to filter out errors (i.e., k-mers that occur only once) and to detect repetitions in the input DNA sequence (i.e., k-mers that occur very frequently). Many of today's k-mer counters typically use a Bloom filter to remove singletons and a conventional, space-inefficient hash table to count non-singletons.

For our experiments, we counted 28-mers, a common value used in actual DNA sequencing tasks. We used SRA accesion SRR072006 [1] for our benchmarks. This data set has a total of $\approx 330M$ 28-mers in which there are $\approx 149M$ distinct 28-mers. We measured the total time taken to complete the experiment.

Firehose [22] is a suite of benchmarks simulating a network-event monitoring workload. A Firehose benchmark setup consists of a *generator* that feeds packets via a local UDP connection to a *monitor*, which is being benchmarked. The monitor must detect "anomalous" events as accurately as possible while dropping as few packets as possible. The anomaly detection task is as follows: each packet has an ID and value, which is either "SUSPICIOUS" or "OK". When the monitor sees a particular ID for the 25th time, it must determine whether that ID occurred with value SUSPICIOUS more than 20 times, and mark it as anomalous if so. Otherwise, it is marked as non-anomalous.

The Firehose suite includes two generators: the power-law generator generates items with a Zipfian distribution, the active-set generator generates items with a uniformly random distribution. The power-law generator picks keys from a static range of 100,000 keys, following a power-law distribution. The active-set generator selects keys from a continuously evolving active-set of 128,000

keys. The probability of selection of each key varies with time and roughly follows a bell-shaped curve. Therefore, in a stream, a key appears occasionally, then appears more frequently, and then dies off. Firehose also includes a reference implementation of a monitor. The reference implementation uses conventional hash tables for counting the occurrences of observations.

In our experiments, we inserted data from the above application data sets into the counting quotient filter to measure the raw insertion throughput of the CQF. We performed the experiment by first dumping the data sets to files. The benchmark then read the files and inserted the elements into the CQF. We took 50M items from each data set. The CQF was configured to the next closest power of 2, i.e., to $\approx 64M$ slots.

To evaluate how the counting quotient filter scales with multiple threads we performed multi-threaded insertion experiments for all our synthetic and application data sets. However, instead of measuring the instantaneous insertion throughput, we measured the overall insertion throughput with increasing numbers of threads to show how the CQF scales with multiple threads.

6.2 In-RAM Performance

Figure 6 shows the in-memory performance of the RSQF, CQF, CF and BF when inserting ≈ 67 million items.

The RSQF and CQF outperform the Bloom filter on all operations and are roughly comparable to the cuckoo filter. Our QF variants are slightly slower than the cuckoo filter for inserts and lookups of existing items. They are faster than the CF for lookups of non-existent items at low load factors and slightly slower at high load factors. Overall, the CQF has lower throughput than the RSQF because of the extra overhead of counter encodings.

6.3 On-SSD Performance

Figure 7 shows the insertion and lookup throughputs of the RSQF, CQF, and CF when inserting 1 billion items. For all three data structures, the size of the on-SSD data was roughly $2\times$ the size of RAM.

The quotient filters significantly outperform the cuckoo filter on all operations because of their better cache locality. The cuckoo filter insert throughput drops significantly as the data structure starts to fill up. This is because the cuckoo filter performs more kicks as the data structure becomes full, and each kick requires a random I/O. The cuckoo filter lookup throughput is roughly half the throughput of the quotient filters because the quotient filters need to look at only one location on disk, whereas the cuckoo filter needs to check two locations.

6.4 Performance with skewed data sets

Figure 8 shows the performance of the counting quotient filter and counting Bloom filter on a data set with a Zipfian distribution with Zipfian coefficient 1.5 and universe size of 201 million elements. We don't evaluate the cuckoo filter in this setting because it fails after ≈ 200 insertions. This is because the cuckoo filter cannot handle more than 8 duplicates of any item, but Zipfian distributed data contains many duplicates of the most common items.

The counting quotient filter is 6 to $10\times$ faster than the counting Bloom filter for all operations and, as shown in Table 1, uses 30 times less space.

As explained in Section 4, the counting quotient filter encodes the counters in the slots instead of storing a separate copy for each occurrence of an item. Figure 9c shows the percentage of slots in use in the counting quotient filter during the experiment. Combined with Figure 8, this shows that even when the counting quotient filter is nearly full, i.e., most of its slots are in use, it still offers good performance on skewed data.

Data set	Num distinct items	Max frequency
uniform-random	49927180	3
zipfian	10186999	2559775
K-mer	34732290	144203
Firehose (active-set)	17438241	24965994
Firehose (power-law)	85499	16663304

Table 4: Characteristics of data sets used for multi-threaded experiments. The total number of items in all of the data sets is 50M.

Number of CQFs (K)	Average merge throughput
2	12.398565
4	12.058525
8	11.359184

Table 5: CQF K-way merge performance. All the CQFs to be merged are created with 16M slots and filled up to 95% load factor. The insert throughput is in millions of items merged per second.

6.5 Applications

K-mer counting. Figure 9a shows the instantaneous insert throughput of the counting quotient filter. The throughput is similar to that in the Zipfian experiments, showing that the counting quotient filter performs well with real-world data sets.

FireHose. We benchmarked the instantaneous insertion throughput of the counting quotient filter for the data sets generated by the Firehose generators. In Figure 9a we show the insertion performance of the counting quotient filter for data from the active-set and power-law generators. Due to huge repetitions in the data set from the power-law generator, the insertion throughput is constantly very high. For the active-set data set, the insertion throughput is similar to our experiments with uniformly random data.

6.6 Concurrency

Figure 9b shows the average insert throughput of the counting quotient filter with increasing numbers of threads for various data sets. The average insert throughput is linearly increasing with increasing number of threads for all data sets. Table 4 shows the prevalence of repetitions in these data sets. The data sets from Firehose (both active-set and power-law) have many repetitions. The counting quotient filter scales linearly with multiple threads even for these data sets. The multi-threading scheme as explained in Section 5 not only reduces the lock contention among multiple insertion threads, but also amortizes the cost of acquiring a lock.

6.7 Mergeability

Table 5 shows the average merge throughput during a K-way merge of counting quotient filters with increasing K. The average merge throughput is always greater than the average insert throughput. This is because, during a merge, we insert hashes into the output counting quotient filter in increasing order, thereby avoiding any shifting of remainders. Although the average merge throughput is greater than the average insert throughput, the merge throughput decreases slightly as we increase K. This is because, with bigger K, we spend more time finding the smallest hash in each iteration. For very large values of K, one could use a min-heap to determine the smallest hash quickly during each iteration.

6.8 Impact of optimizations

Figure 10 shows the performance of two RSQF implementations; one using the fast x86 rank and select implementations described in Section 3 and one using C implementations. The optimizations

(a) Inserts. (b) Successful lookups. (c) Uniformly random lookups.
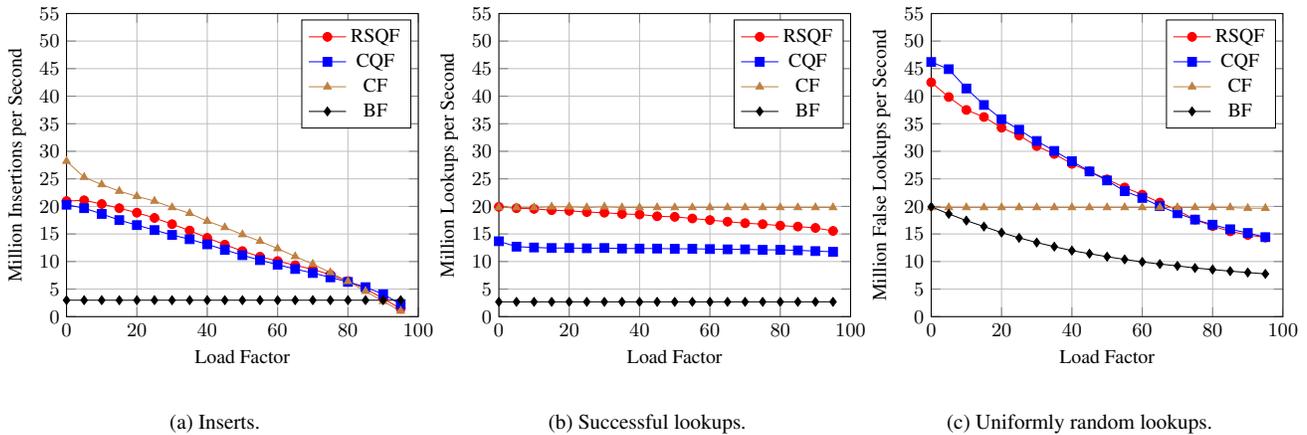
Figure 6: In-memory performance of the QF, CQF, CF, and BF on uniformly random items. The first graph shows the insert performance against changing load factor. The second graph shows the lookup performance for existing items. The third graph shows the lookup performance for uniformly random items. (Higher is better.)

(a) Inserts on SSD. (b) Successful lookups on SSD. (c) Uniformly random lookups on SSD.
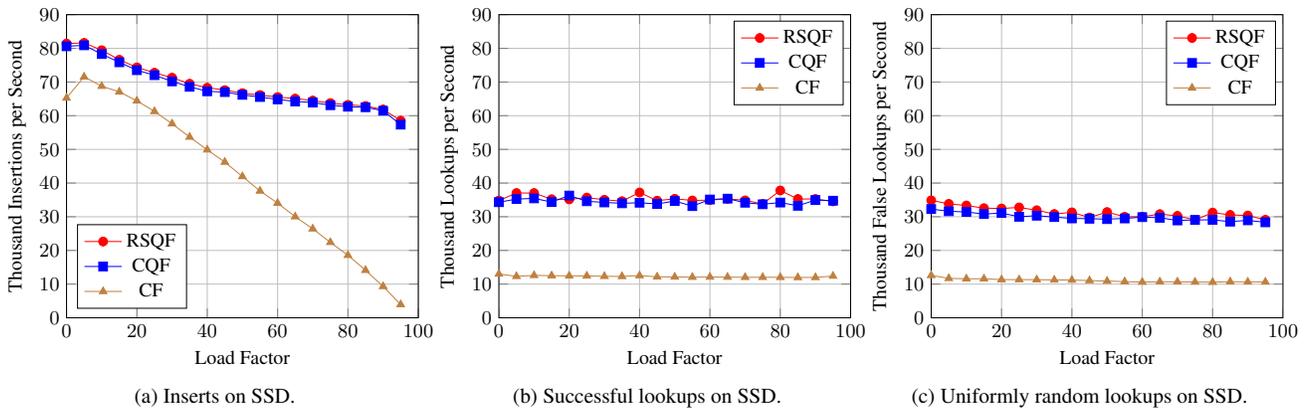
Figure 7: On-SSD performance of the RSQF, CQF, and CF on uniformly random inputs. The first graph shows the insert performance against changing load factor. The second graph shows the lookup performance for existing items. The third graph shows the lookup performance for uniformly random items. (Higher is better.)

speed up lookups by a factor of 2-4, depending on the load factor. The optimizations speed up inserts less than lookups because inserts are bottlenecked by the time required to shift elements around (which does not involve performing rank or select operations).

Figure 10 shows the insert and lookup performance of the original quotient filter and the RSQF. The original quotient filter lookup throughput drops as it passes 60% load factor because it must examine an entire cluster, and the average cluster size grows quickly as the load factor increases. RSQF lookup performance drops more slowly because it must only examine a single run, and the average run size is bounded by a constant for any load factor.

Note that performance for the QF and RSQF on lookups for non-existent items drops for a different reason. Both filters first check $Q.occupieds[h_0(x)]$ during a lookup for x. If this bit is 0 they can immediately return false. When looking up elements that are in the filter, this fast-path never gets taken. When looking up non-existent items, this fast-path is frequently taken at low load factors, but less frequently at high load factors. As a result, for both filters, lookups of non-existent items start off very fast at low load factors and drop to roughly the same performance as lookups for existing items as the load factor increases, as can be seen in Figures 10b and 10c.

7. CONCLUSION

This paper shows that it is possible to build a counting data structure that offers good performance and saves space, regardless of the input distribution. Our counting quotient filter uses less space than other counting filters, and in many cases uses less space than non-counting, membership-only data structures.

Our counting quotient filter also offers several features that are important for real applications. It has good data locality, so that it can operate efficiently on SSD. Quotient filters can be merged to compute their union, a feature that has found wide use in parallel computing [2]. Mergeability also means the counting quotient filter can be used to build a write-optimized counting cascade filter, similar to the cascade filter in the original quotient filter paper.

Finally, we revealed a connection between the quotient filter's metadata and the RANK and SELECT operations widely used in other compact data structures. We then described an efficient method for implementing SELECT on 64-bit integers in the x86 instruction set. This technique may be of interest to other rank-and-select-based data structures.

8. ACKNOWLEDGMENTS

(a) Inserts.

(b) Successful lookups.

(c) Uniformly random lookups.
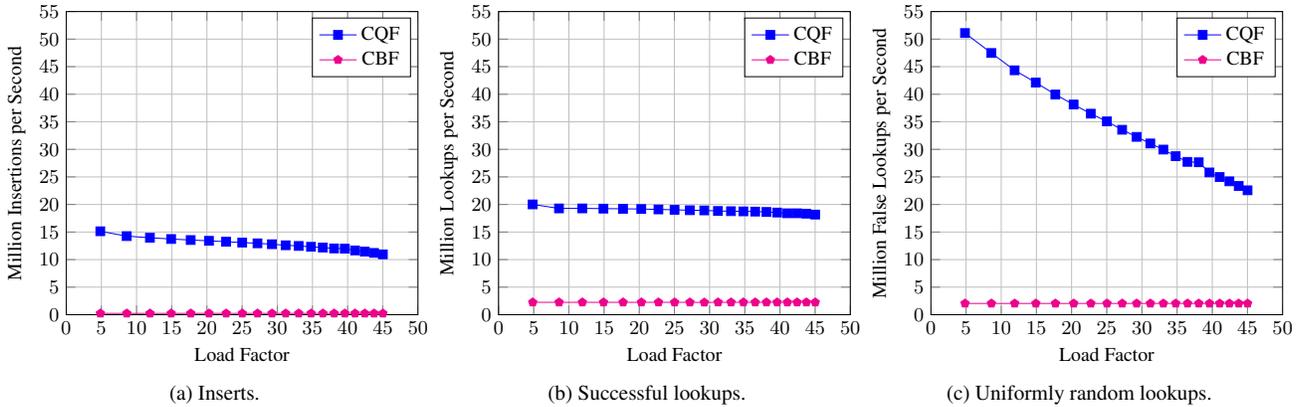
Figure 8: In-memory performance of the CQF and CBF on data with a Zipfian distribution. We don't include the CF in these benchmarks because the CF fails on a Zipfian input distribution. The load factor does not go to 95% in these experiments because load factor is defined in terms of the number of distinct items inserted in the data structure, which grows very slowly in skewed data sets. (Higher is better.)

(a) CQF in-memory insert performance on application data sets. (Higher is better.)

(b) CQF multi-threaded in-memory insert performance. (Higher is better.)

(c) Percent of slots in use in a counting quotient filter vs. the number of distinct items inserted from a Zipfian distribution with C=1.5 and a universe of 201M. We performed a total of 201M inserts.
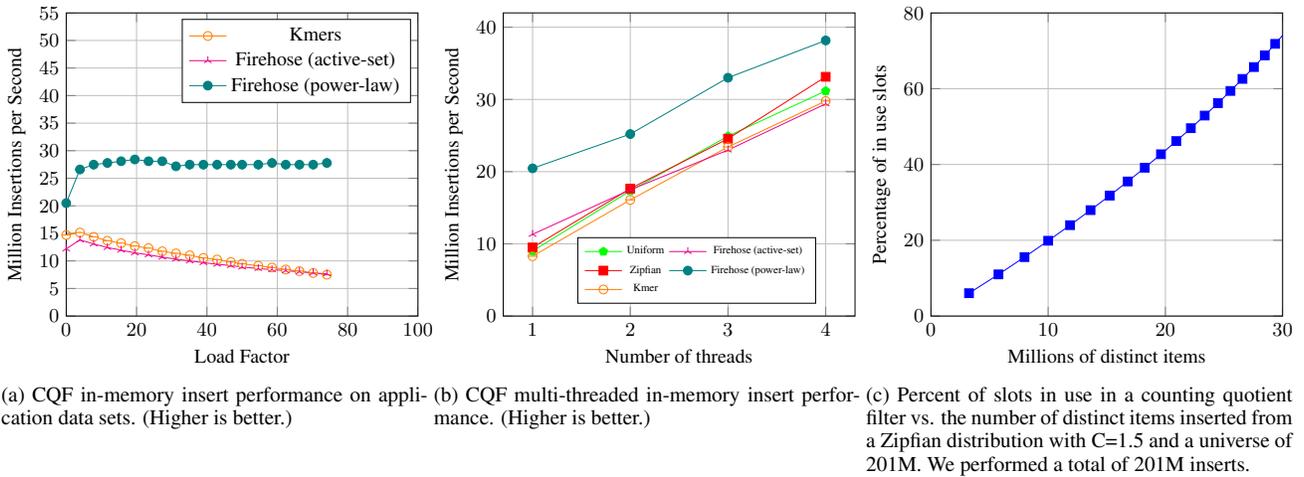
Figure 9: In-memory performance of the counting quotient filter with real-world data sets and with multiple threads, and percent slot usage with skewed distribution.

(a) Inserts.

(b) Successful lookups.

(c) Uniformly random lookups.
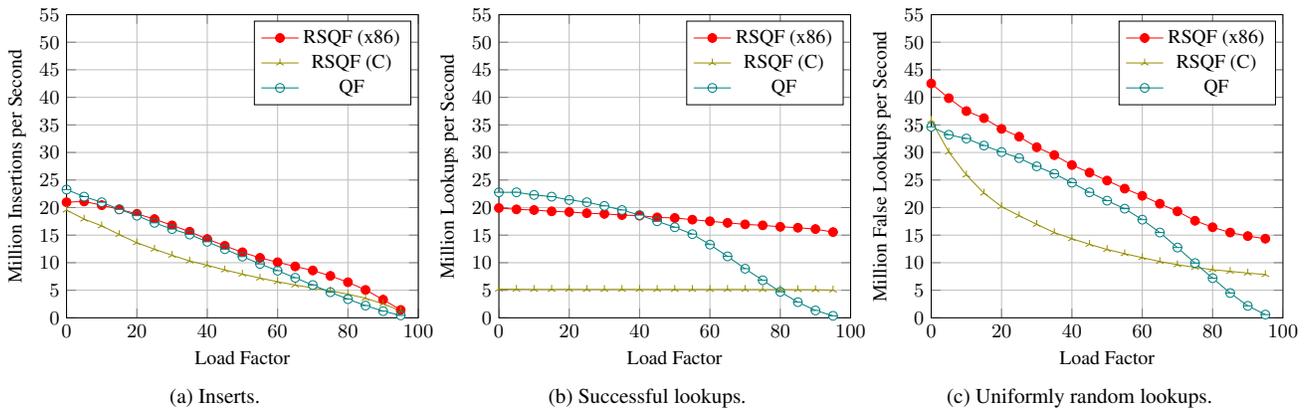
Figure 10: In-memory performance of the RSQF implemented with x86 pdep & tzcnt instructions, the RSQF with C implementations of rank and select, and the original QF, all on uniformly random items. The first graph shows the insert performance against changing load factor. The second graph shows the lookup performance for existing items. The third graph shows the lookup performance of uniformly random items. (Higher is better.)

9. REFERENCES

[1] *F. vesca* genome read dataset. ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA020/SRA020125/SRX030576/SRR072006.fastq.bz2. [Online; accessed 19-February-2016].

[2] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):26, 2013.

[3] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable Bloom filters. *Journal of Information Processing Letters*, 101(6):255–261, 2007.

[4] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in AsterixDB. *Proceedings of the VLDB Endowment*, 7(10):841–852, 2014.

[5] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kaner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11), 2012.

[6] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro. Insertion sort is $O(n \log n)$. *Theory of Computing Systems*, 39(3):391–397, 2006. Special Issue on *FUN '04*.

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

[8] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. In *European Symposium on Algorithms (ESA)*, pages 684–695. Springer, 2006.

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

[10] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross. Buffered Bloom filters on solid state storage. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–8, 2010.

[11] S. Cohen and Y. Matias. Spectral Bloom filters. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 241–252, 2003.

[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[13] B. Corominas-Murtra and R. V. Solé. Universality of Zipf's law. *Physical Review E*, 82(1):011102, 2010.

[14] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du. BloomFlash: Bloom filter on flash-based storage. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 635–644, 2011.

[15] B. K. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2010.

[16] B. Fan. Cuckoo filter source code in C++. https://github.com/efficient/cuckoofilter, 2014. [Online; accessed 19-July-2014].

[17] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*,

[18] pages 75–88, 2014.

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

[19] A. Geil. Quotient filters: Approximate membership queries on the GPU. http://on-demand.gputechconf.com/gtc/2016/presentation/s6464-afton-geil-quoetient-filters.pdf, 2016.

[20] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. Parallel de Bruijn graph construction and traversal for de novo genome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 437–448, 2014.

[21] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[22] S. P. Karl Anderson. Firehose. http://firehose.sandia.gov/, 2013. [Online; accessed 19-Dec-2015].

[23] G. Lu, B. Debnath, and D. H. Du. A forest-structured Bloom filter with flash memory. In *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, 2011.

[24] P. Melsted and J. K. Pritchard. Efficient counting of k-mers in DNA sequences using a Bloom filter. *BMC Bioinformatics*, 12(1):1, 2011.

[25] P. O'Neil, E. Cheng, D. Gawlic, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[26] A. Partow. C++ Bloom filter library. https://code.google.com/p/bloom/. [Online; accessed 19-July-2014].

[27] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121, 2007.

[28] Y. Qiao, T. Li, and S. Chen. Fast Bloom filters and their generalization. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(1):93–103, 2014.

[29] R. S. Roy, D. Bhattacharya, and A. Schliep. Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, 30:1950–1957, 2014.

[30] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of Bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2012.

[31] M. Vallentin. Counting Bloom filter source code in C++. https://github.com/mavam/libbf, 2014. [Online; accessed 19-July-2015].

[32] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. 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)*, pages 16:1–16:14, 2014.

[33] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown. These are not the k-mers you are looking for: Efficient online k-mer counting using a probabilistic data structure. *PLoS One*, 9(7):e101271, 2014.

[34] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2008.