Fast and Space-Efficient Maps for Large Data Sets

A Dissertation presented

by

Prashant Pandey

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2018

Stony Brook University

The Graduate School

Prashant Pandey

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Michael A. Bender, Thesis Advisor
Professor, Computer Science

Rob Johnson, Thesis Advisor
Research Assistant Professor, Computer Science

Michael Ferdman, Thesis Committee Chair
Associate Professor, Computer Science

Rob Patro
Assistant Professor, Computer Science

Guy Blelloch
Professor, Computer Science
Carnegie Mellon University

This dissertation is accepted by the Graduate School

Dean of the Graduate School

Abstract of the Dissertation

Fast and Space-Efficient Maps for Large Data Sets

by

Prashant Pandey

Doctor of Philosophy

in

Computer Science

Stony Brook University

2018

Approximate Membership Query (AMQ) data structures, such as the Bloom 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, nor can they store values corresponding to items. They 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.

In this dissertation, we show how to use recent advances in the theory of compact and succinct data structures to build a general-purpose AMQ. We further demonstrate how we use this new feature-rich AMQ to improve applications in computational biology and streaming in terms of space, speed, and simplicity.

First, we present a new general-purpose AMQ, the counting quotient filter (CQF). The CQF supports approximate membership testing and counting occurrences of items in a data set and offers an order-of-magnitude faster performance than the Bloom filter. The CQF can also be extended to be a fast and space-efficient map for small keys and values.

Then we discuss three applications in computational biology and streaming that use the CQF to solve big data problems. In the first application, Squeakr, we show how we can compactly represent huge weighted de Bruijn Graphs in computational biology using the CQF as an approximate multiset representation. In deBGR, we further show how to use a weighted de Bruijn Graph invariant to iteratively correct all the approximation errors in the Squeakr representation. In the third application, Mantis, we show how to use the CQF as a map for small keys and values to build a large-scale sequence-search index for large collections of raw sequencing data. In the fourth application, we show how to extend the CQF to build a write-optimized key-value store (for small keys and values) for timely reporting of heavy-hitters using external memory.

Dedication

To my collaborators, cohabitators, and everyone else who had to put up with me. Especially, my parents, Sudha and Dharmendra, who taught me the importance of hard work and humbleness in life. My brother and sister-in-law, Siddharth and Pooja, who helped me at every step of my career and encouraged me to pursue research. And finally, my wife, Kavita, who stood with me as a friend all through the good and bad times of my grad life.

Contents

List of Figures

List of Tables

Acknowledgements

My grad school life has been an amazing ride due to the support and encouragement of several people whom I can hardly ever repay. This acknowledgment is just a tiny effort to thank them.

I would like to start with my advisors, Michael Bender and Rob Johnson, for showing enduring patience with me in my early days of grad school while I was learning how to do research. From Michael, I learned the value of keeping an open mind while tackling new problems and how to become a good academic peer. From Rob, I learned the importance of keeping a rational attitude toward problems that come your way while doing research and working in a group. It has been an extremely rewarding experience doing pair programming with Rob. I would especially like to mention the nurturing environment that Michael and Rob provided which helped me immensely in working at my own pace and learning from my own mistakes. Finally, I would like to thank them for making grad school fun by encouraging me to travel and attend conferences and hiking with me.

I also want to thank Rob Patro, Mike Ferdman, and Don Porter for their constant guidance and fruitful research discussions. I would like to especially thank Rob Patro for taking my random excursions with data structures and converting them into meaningful research projects in computational biology. I would like to thank Mike for his deep insight into systems and his quirky replies making research meetings fun. I would like to thank Don for guiding me in my early grad school days and making me realize that I am interested in both theory and systems.

I want to thank all my collaborators, especially, Martin Farach-Colton, Bradley Kuszmaul, Cynthia Philips, Jon Berry, and Tom Kroeger, who taught me how to do research, collaborate, and treat your juniors with respect.

I want to thank all my lab mates from algorithms lab, OSCAR lab, and Combine lab. Especially, Pablo, Dzejla, Mayank, Sam, Rishabh, Shikha, Tyler, and Rathish from algorithms lab, Bill, Jun, Chia-Che, Amogh, Yang, and Yizheng from OSCAR lab, and Fatemeh from Combine lab. Due to their guidance, company, and support my ride through the grad school was smooth and enjoyable.

Finally, I want to mention the contribution of our admin department, Kathy Germana, Betty Knitweiss, and Cindy SantaCruz-Scalzo, in looking after me through my grad school. Because of their diligent and untiring efforts, I got paid on time, got my travel reimbursements, and graduate without any hiccups.

Chapter 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 [29], quotient filters [21], cuckoo filters [69], and frequency-estimation data structures [57]. AMQs have become one of the primary go-to data structures in systems builders' toolboxes [39].

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.

Current AMQs, such as the Bloom filter, have several shortcomings, including a relatively small set of supported operations and poor performance due to poor cache locality. The Bloom filter has inspired numerous variants that try to overcome one drawback or the other, e.g., [5, 30, 43, 62, 70, 108, 138, 139]. For example, the counting Bloom filter (CBF) [70] 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. However, there is not one single variant that overcomes all the drawbacks.

AMQs in systems. 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 [124] based systems [8, 164] and deduplication systems [62, 63, 172], 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.

AMQs in computational biology. 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 [110, 112, 146]. 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 [77].

Many tools use Bloom filters and a small auxiliary data structure to exactly represent large substring graphs in a small space [48, 135, 136, 149]. However, the Bloom filter omits critical information—the frequency of each substring—that is necessary in many other DNA analyses applications.

Large-scale sequence-search indexes [155,156,160] build a binary tree of Bloom filters where each leaf Bloom filter represents the subsequences present in a raw sequencing data set. Due to limitations of the Bloom filter, these indexes are forced to balance between the false-positive rate at the root and the size of the filters representing the individual data sets. Because two Bloom filters can only be merged if they have the same size and Bloom filters can not be resized, filters at the top of the tree contain orders-of-magnitude more items than leaf nodes.

AMQs in streaming. Many streaming applications [39, 118, 152, 170] use Bloom filters and other frequency-estimation data structures, such as the count-min sketch [57]. However, these applications are forced to perform all analyses in RAM due to the inability of the Bloom filters and count-min sketch to scale out of RAM efficiently [83].

Full-featured high-performance AMQs. 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, (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 [112, 168]).

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), can store values, and supports deletions, counting (even on skewed data sets), resizing, merging, and highly concurrent access.

In this dissertation, we show how to use recent advances in the theory of compact and succinct data structures to build a general-purpose AMQ data structure that has all these features. We further demonstrate how we use this new feature-rich AMQ data structure to improve applications in computational biology and streaming in terms of space, speed, and simplicity.

1.1 Outline

In Chapter 2, we talk about various AMQ and counting filter data structures. We start with giving a formal definition of the AMQ data structure and then describing common AMQ data structures. We then describe how counting data structures generalize the notion of AMQ data structures and explain how common counting data structures work. Finally, we describe succinct data structures. We borrow some tricks from succinct data structures in the implementation of our high-performant AMQ. The goal of this chapter is to provide the reader with some important background in AMQ, counting filter, and succinct data structures that we will draw upon throughout the rest of the dissertation.

In Chapter 3, we formalize the notion of a ***counting filter***, an AMQ data structure that counts the number of occurrences of each input item, and describe 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 also evaluate the performance of the counting quotient filter comparing it to other AMQ data structures, like the Bloom filter and cuckoo filter. This chapter's content expands upon two published works [129, 131].

In Chapter 4, we describe Squeakr, which uses the CQF as a counting filter data structure to build a smaller and faster k-mer (a length-k substring) counter and which can also be used as an approximate weighted de Bruijn Graph representation. This chapter's content expands upon one of our published papers [132].

We then show in Chapter 5 how we extend this work in deBGR to use a weighted de Bruijn Graph invariant and approximate counts in the CQF to iteratively self correct all the approximation errors in the weighted de Bruijn Graph representation in Squeakr. This chapter's content expands upon one of our published papers [130].

In Chapter 6, we describe Mantis which is a large-scale sequence-search index built over large collections of raw sequencing data. Mantis uses the CQF as a map for small keys and values to store a mapping from k-mers to the set of raw sequencing samples in which the k-mer appears. This chapter's content expands upon my most recent work [128].

In Chapter 7, we show how I scale the CQF out-of-RAM to build write-optimized data structures (time-stretch filter and popcorn filter) which support timely reporting of heavy-hitters using external memory (SSD). The work in this chapter expands upon two manuscripts that are currently in preparation.

Chapter 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 [29], 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 non-existent items [39].

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 [5, 30, 43, 62, 70, 108, 138, 139]. The counting Bloom filter (CBF) [70] 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 [5] uses multiple Bloom filters to maintain the target false-positive rate δ even when n is unknown.

The quotient filter [21] 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 [24], 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 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 et al. have accelerated the QF by porting it to GPUs [75].

The cuckoo filter [69] is built on the idea of cuckoo hashing [126]. 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 [30] 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 [51] 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) [57] 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 3.7, this will be less space efficient than the counting quotient filter.

2.3 Succinct data structures

A ***succinct data structure*** consumes an amount of space that is close to the information theoretically optimal. More precisely, if Z denotes the information-theoretically optimal space usage for a given data-structure specification, then a succinct data structure would use $Z + o(Z)$ space. There is much research on how to replace traditional data structures with succinct alternatives [59, 67, 71, 79, 119, 140, 147], especially for memory-intensive applications, such as genetic databases, newspaper archives, dictionaries, etc. Researchers have proposed numerous succinct data structures, including compressed suffix arrays [85], FM indexes [72], and wavelet trees [119].

Two basic operations—namely ***rank*** and ***select*** [92]—are commonly used for navigating within succinct data structures. For bit vector $B[0, \ldots, n-1]$, $\textsc{rank}(j)$ returns the number of 1s in prefix $B[0, \ldots, j]$ of B; $\textsc{select}(r)$ returns the position of the rth 1, that is, the smallest index j such that $\textsc{rank}(j) = r$. For example, for the 12-bit vector $B[0, \ldots, 11] =$`100101001010`, $\textsc{rank}(5) = 3$, because there are three bits set to one in the 6-bit prefix $B[0, \ldots, 5]$ of B, and $\textsc{select}(4) = 8$, because $B[8]$ is the fourth 1 in the bit vector.

Researchers have proposed many ways to improve the empirical performance of rank and select [80, 82, 120, 162, 171] because faster implementations of rank and select yield faster succinct data structures. Many succinct data structures are asymptotically optimal, but have poor performance in practice due to the constants involved in bit-vector rank and select operations [162].

Chapter 3: The counting quotient filter (CQF)

In this chapter we explain how to improve upon a quotient filter's [21] metadata representation and algorithms. We explain how to embed variable-sized counters into a quotient filter in Section 3.6.

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 smaller 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 AMQ 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 [21]).

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*** $h_1(x)$ [98, Section 6.4, exercise 13]. 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 chapter, 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:

- The quotient filter maintains an ***occupieds*** bit vector of length 2^q. The RSQF Q sets Q.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.

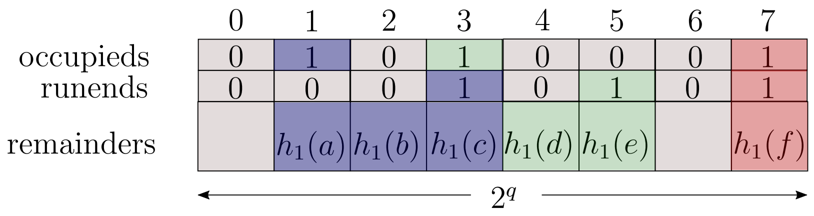


Figure 3.1: A simple rank-and-select-based quotient filter. The colors are used to group slots that belong to the same run, along with the runends bit that marks the end of that run and the occupieds bit that indicates the home slot for remainders in that run.

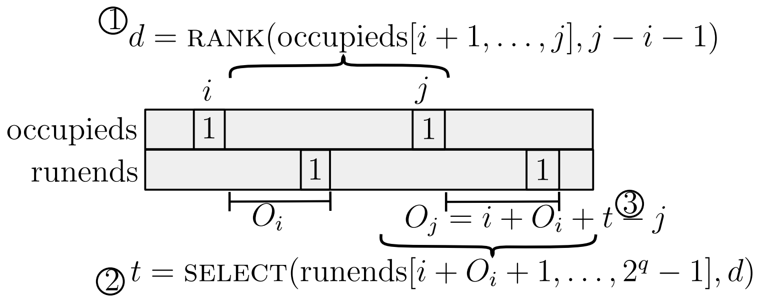

① $d = \text{RANK}(\text{occupieds}[i+1,\ldots,j], j-i-1)$
i
j
occupieds
runends
O_i
$O_j = i + O_i + t$ ③ $- j$
② $t = \text{SELECT}(\text{runends}[i+O_i+1,\ldots,2^q-1], d)$

Figure 3.2: Procedure for computing offset O_j given O_i.

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.runends[b] to 1 if and only if slot b contains the last remainder in a run.

As shown in the Figure 3.1, 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, $\text{RANK}(B, i)$ returns the number of 1s in B up to position i, i.e., $\text{RANK}(B, i) = \sum_{j=0}^{i} B[j]$. Select is essentially the inverse of rank. $\text{SELECT}(B, i)$ 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.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:     $b \leftarrow h_0(x)$
3:     if $Q$.occupieds[$b$] $= 0$ then
4:         return 0
5:     $t \leftarrow$ RANK($Q$.occupieds, b)
6:     $\ell \leftarrow$ SELECT($Q$.runends, $t$)
7:     $v \leftarrow h_1(x)$
8:     repeat
9:         if $Q$.remainders[$\ell$] $= v$ then
10:            return 1
11:        $\ell \leftarrow \ell - 1$
12:    until $\ell < b$ or $Q$.runends[$\ell$] $= 1$
13:    return false
```

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

offset	occupieds	runends	remainders
8	64	64	$64r$

Figure 3.3: Layout of a rank-and-select-based-quotient-filter block. The size of each field is specified in bits.

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 3.2. 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.

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 [21]) 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 3.3. 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

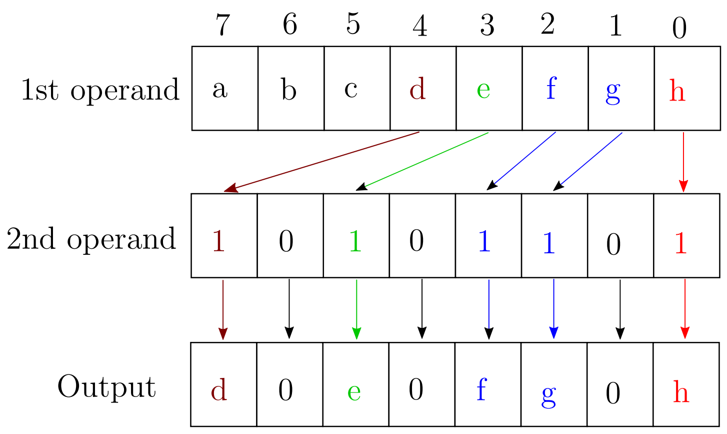


Figure 3.4: A simple example of pdep instruction [87]. The 2nd operand acts as a mask to deposit bits from the 1st operand.

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, as shown in Algorithm 3. Both PDEP and TZCNT instructions were introduced in Intel's Haswell line of CPUs.

PDEP deposits bits from one operand in locations specified by the bits of the other operand, as illustrated in Figure 3.4. If $p = \text{PDEP}(v, x)$, then the ith bit of p is given by

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

TZCNT returns the number of trailing zeros in its argument. If B is a 12-bit vector such that $B[0,11] =$`110010100000` then $\text{TZCNT}(B) = 5$.

The implementation works because performing $\text{PDEP}(2^j, x)$ produces a 64-bit integer

p with a single bit set—in the same position as x's jth bit. TZCNT then finds the position of this bit by counting the number of trailing zeros in p.

POPCOUNT returns the number of bits set in its argument. We implement $\textsc{rank}(v, i)$ on 64-bit vectors using the widely-known mask-and-popcount method [82]:

$$\textsc{rank}(v, i) = \textsc{popcount}(v \ \& \ (2^i - 1))$$

We evaluate the performance impact of these optimizations in Section 3.9.

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 3.9 presents experimental results showing the performance impact of this redesign.

3.4 Space analysis

Table 3.1 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 3.9). The RSQF is more space efficient than the original quotient filter, the cuckoo filter, and, for false-positive rates less than 1/64, 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 3.5 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.,

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 3.1: 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.

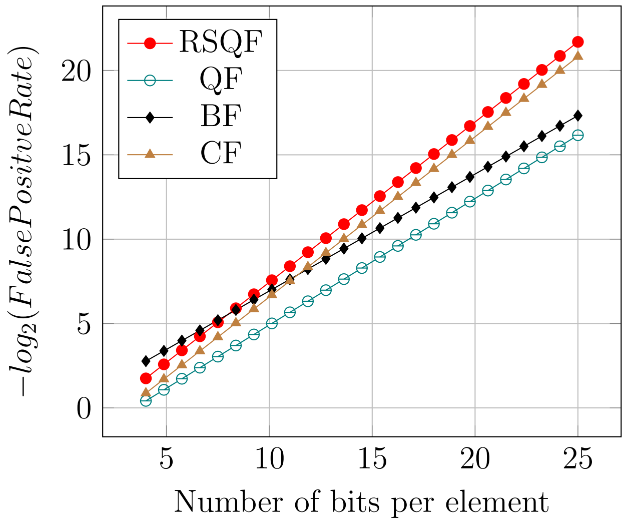


Figure 3.5: Number of bits per element for the RSQF, QF, BF, and CF. The RSQF requires less space than the CF amd less space than the BF for any false-positive rate less than 1/64. (Higher is better)

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.

3.6 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 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 3.2 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 ($\boxed{\mathbf{0}, \mathit{2}, \mathbf{0}, \mathbf{0}}$, $\boxed{\mathbf{3}, \mathit{0}, \mathit{6}, \mathbf{3}}$, $\boxed{\mathbf{8}, \mathit{7}, \mathbf{8}}$).

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 3.2: Encodings for C occurrences of remainder x in the CQF.

3.7 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 $k = M$.

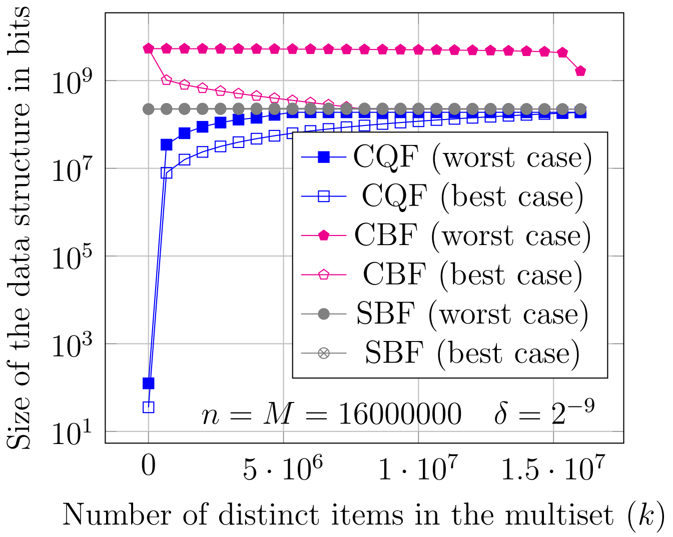


Figure 3.6: Space comparison of CQF, SBF, and CBF as a function of the number of distinct items. All data structures are built to support up to $n = 1.6 \times 10^7$ insertions with a false-positive rate of $\delta = 2^{-9}$.

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 3.6 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.

3.8 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%.

3.9 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 [133], Bender et al.'s quotient filter [21], Fan et al.'s cuckoo filter [68], and Vallentin's counting Bloom filter [161].

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 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.

3.9.1 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 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.

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 [21].

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 [146, 168] and FireHose [94].

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 [146, 168]. 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 [2] 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 [94] 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 24th 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.

3.9.2 In-RAM performance

Figure 3.7 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.

3.9.3 On-SSD performance

Figure 3.8 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.

3.9.4 Performance with skewed data sets

Figure 3.9 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 uses 30 times less space.

As explained in Section 3.6, the counting quotient filter encodes the counters in the slots instead of storing a separate copy for each occurrence of an item. Figure 3.10b shows the percentage of slots in use in the counting quotient filter during the experiment. Combined with Figure 3.9, 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 3.3: 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 3.4: 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.

3.9.5 Applications

K-mer counting. Figure 3.10a 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 3.10a 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.

3.9.6 Mergeability

Table 3.4 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.

3.9.7 Impact of optimizations

Figure 3.11 shows the performance of two RSQF implementations; one using the fast x86 rank and select implementations described in Section 3.2 and one using C implementations. The optimizations 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 3.11 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.

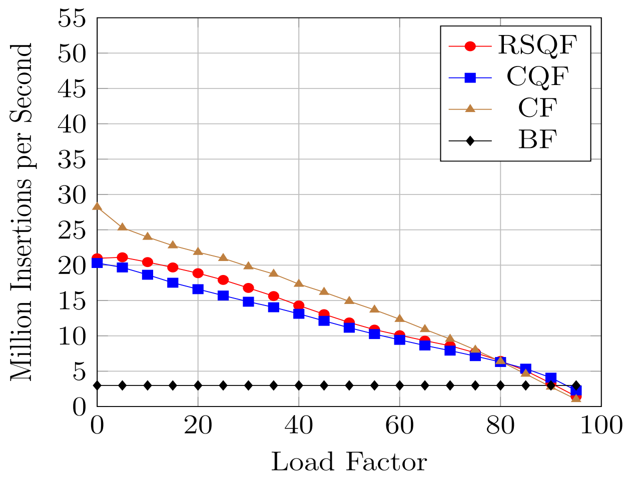
Note that performance for the QF and RSQF on lookups for non-existent items drops for a different reason. Both filters first check $Q.\text{occupieds}[h_0(x)]$ during a lookup for x. If this bit is 0 they can immediately return false. When looking up elements that are in the filter, this fast-path never 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 3.11b and 3.11c.

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

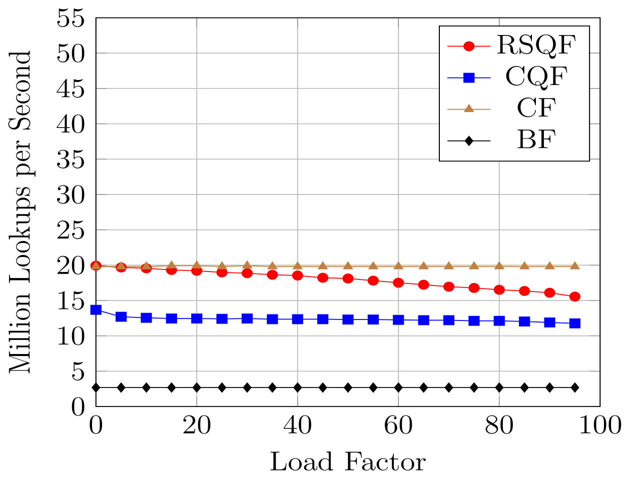
```
1: function FIND_FIRST_UNUSED_SLOT($Q$, $x$)
2:     $r \leftarrow$ RANK($Q$.occupieds, $x$)
3:     $s \leftarrow$ SELECT($Q$.runends, $r$)
4:     while $x \leq s$ do
5:         $x \leftarrow s + 1$
6:         $r \leftarrow$ RANK($Q$.occupieds, $x$)
7:         $s \leftarrow$ SELECT($Q$.runends, $s$)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

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

Algorithm 3 Algorithm for determining the position of the jth 1 in a machine word.

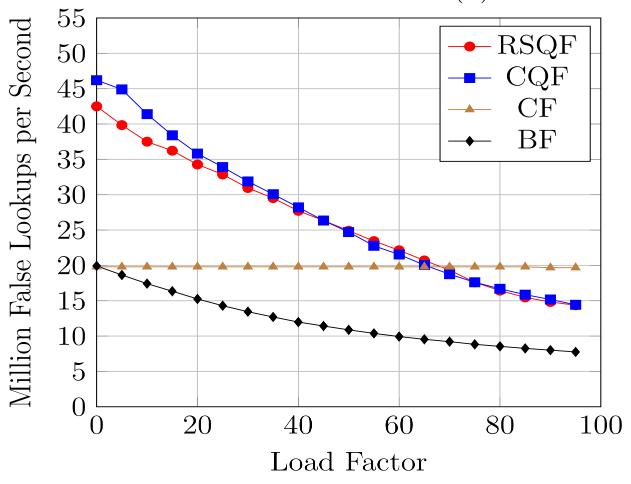
```
1: function SELECT($x$, $j$)
2:     $i \leftarrow$ ShiftLeft($1, j$)
3:     $p \leftarrow$ PDEP($i, x$)
4:     return TZCNT($p$)
```



(a) Inserts.

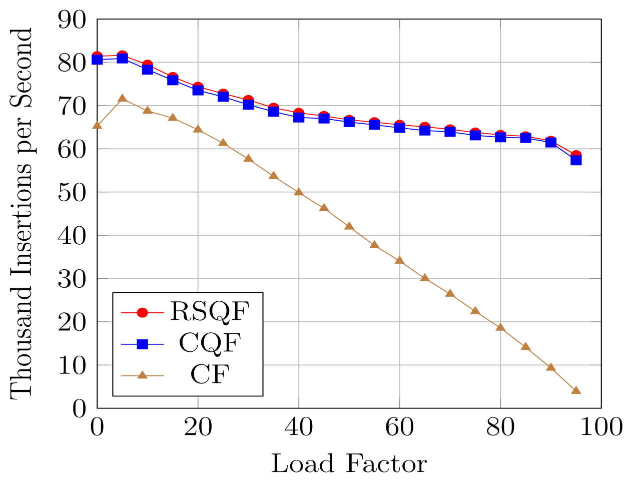


(b) Successful lookups.

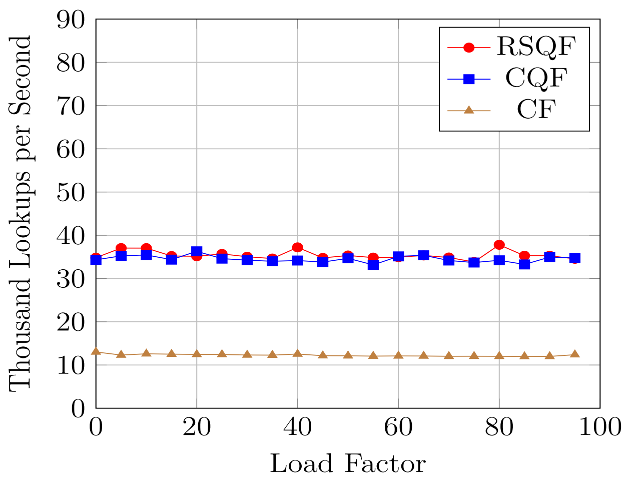


(c) Uniformly random lookups.

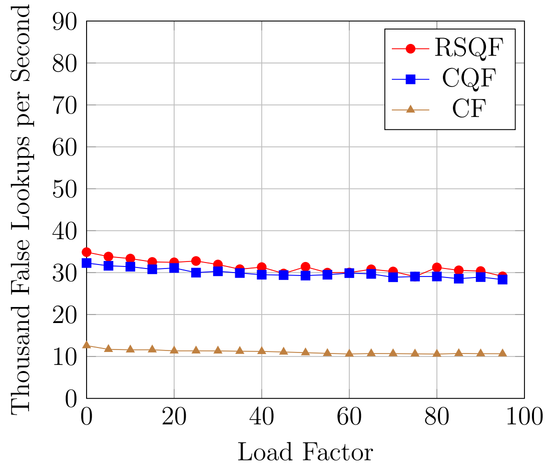
Figure 3.7: In-memory performance of the QF, CQF, CF, and BF on uniformly random items. The first graph shows the insert performance against changing load factor. The second graph shows the lookup performance for existing items. The third graph shows the lookup performance for uniformly random items. (Higher is better.)



(a) Inserts on SSD.

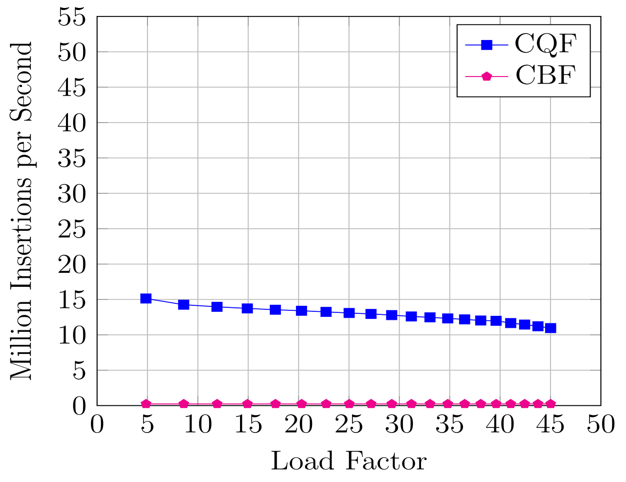


(b) Successful lookups on SSD.

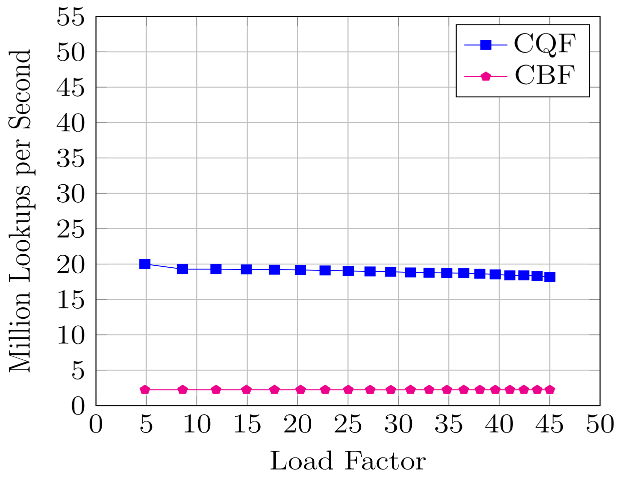


(c) Uniformly random lookups on SSD.

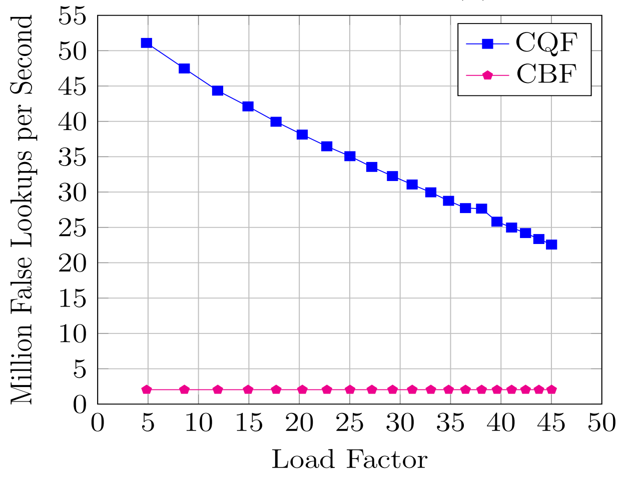
Figure 3.8: On-SSD performance of the RSQF, CQF, and CF on uniformly random inputs. The first graph shows the insert performance against changing load factor. The second graph shows the lookup performance for existing items. The third graph shows the lookup performance for uniformly random items. (Higher is better.)



(a) Inserts.

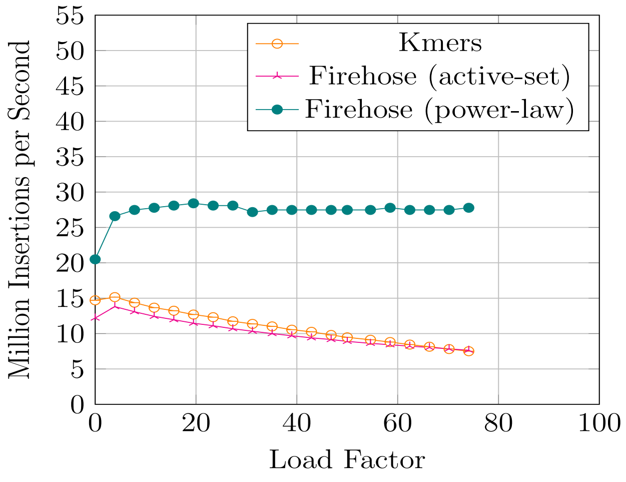


(b) Successful lookups.

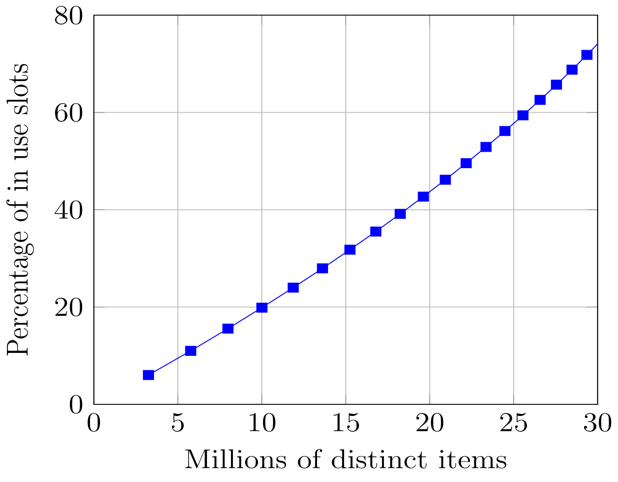


(c) Uniformly random lookups.

Figure 3.9: 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.)

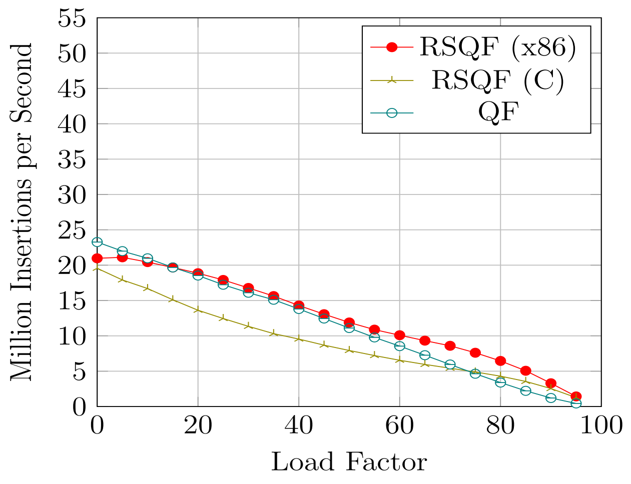


(a) CQF in-memory insert performance on application data sets. (Higher is better.)

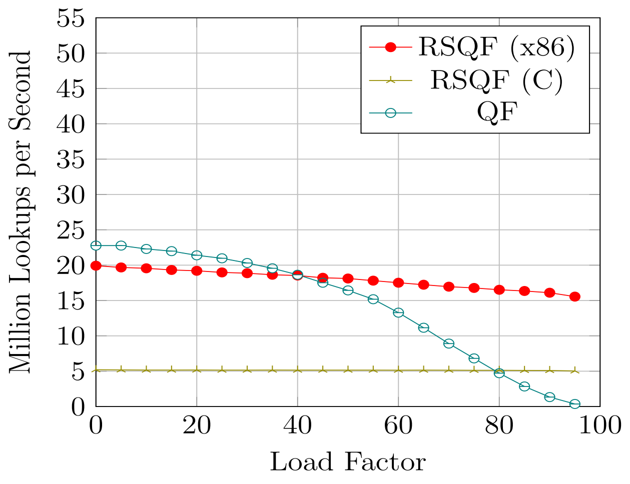


(b) Percent of slots in use in a counting quotient filter vs. the number of distinct items inserted from a Zipfian distribution with C=1.5 and a universe of 201M. We performed a total of 201M inserts.

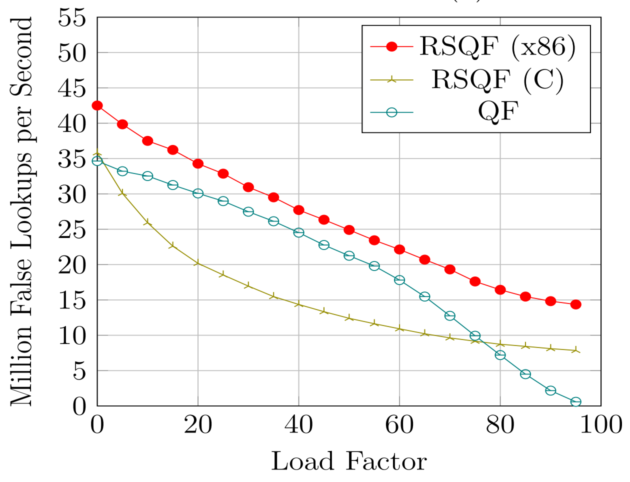
Figure 3.10: In-memory performance of the counting quotient filter with real-world data sets and with multiple threads, and percent slot usage with skewed distribution.



(a) Inserts.



(b) Successful lookups.



(c) Uniformly random lookups.

Figure 3.11: 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.)

Chapter 4: Squeakr: An exact and approximate k-mer counting system

We now describe the first application of the counting quotient filter to computational biology. This application makes use of variable-sized counters in the counting quotient filter to store k-length subsequences (as integer hashes) in the underlying datasets and their frequencies in a space-efficient way. This application also benefits from the fast insertion and lookup throughput of the CQF.

This chapter also describes the design of a thread-safe counting quotient filter. It further shows how to scale the insertion throughput of the thread-safe counting quotient filter with increasing number of threads when the input data is highly skewed.

4.1 k-mer counting

There has been a tremendous increase in sequencing capacity thanks to the rise of massively parallel high-throughput sequencing (HTS) technologies. Many of the computational methods for dealing with the increasing amounts of HTS data use ***k-mers***—strings of k nucleotides—as the atomic units of sequence analysis. For example, most HTS-based genome and transcriptome assemblers use k-mers to build de Bruijn Graphs (see e.g., [13,84,137,153,154,167]). De-Bruijn-graph-based assembly is favored, in part, because it eliminates the computationally burdensome "overlap" approach of the more traditional overlap-layout-consensus assembly [100].

k-mer-based methods are also heavily used for preprocessing HTS data to perform error correction [86, 107, 158] and digital normalization [40, 168]. Even in long-read ("3rd-generation") sequencing-based assembly, the k-mer acts as a building block to help find read overlaps [26, 44] and to perform error correction [150, 151].

k-mer-based methods reduce the computational costs associated with many types of HTS analysis. These include transcript quantification using RNA-seq [134, 169], taxonomic classification of metagenomic reads [125, 165], and search-by-sequence over large repositories of HTS-based sequencing experiments [155].

Many of the analyses listed above begin by counting the number of occurrences of each k-mer in a sequencing dataset. In particular, k-mer counting is used to weed out erroneous data caused by sequencing errors. These sequencing errors most often give rise to "singleton" k-mers (i.e. k-mers that occur only once in the dataset), and the number of singletons grows linearly with the size of the underlying dataset. k-mer counting identifies all singletons, so that they can be removed.

k-mer counting is nontrivial because it needs to be done quickly, the datasets are large, and the frequency distribution of the k-mers is often skewed. There are many different system architectures for k-mer counters, because there are many different competing performance issues, including space consumption, cache-locality, and scalability with multiple threads.

Generally, k-mer-counter research has focused on counting performance and memory usage, but has given relatively little attention to the performance of ***point queries***, i.e., queries for the count of an arbitrary k-mer. But fast point queries would be helpful to many downstream analyses, including de Bruijn Graph traversal [48], searches [155], and inner-products between datasets [117, 163].

k-mer Counting Systems and AMQ Data Structures

Many k-mer-counting approaches have been proposed in recent years, and are embodied in popular k-mer-counting tools such as Jellyfish [110], BFCounter [112], DSK [143], KMC2 [65], and Turtle [146].

These tools, as well as other sequence-analysis systems [48], use the Bloom filter [29] as a data-structural workhorse. The Bloom filter is a well known example of an ***approximate membership query*** (AMQ) data structure, which maintains a compact and probabilistic representation of a set or multiset. AMQs save space by allowing membership queries occasionally to return false positive answers.

k-mer-counting systems such as Jellyfish2 [110], BFCounter [112], and Turtle [146] use a Bloom filter to identify and filter out singleton k-mers, thus reducing the memory consumption. Then the systems resort to larger hash tables, or other, more traditional data structures, for the actual counting. Under such a strategy, k-mers are inserted into the Bloom filter upon first observation, and they are stored in a hash table (or other exact counting data structure) along with their counts upon subsequent observations.

Bloom filters have several shortcomings, including a relatively small set of supported operations and poor performance due to poor cache locality. k-mer-counting systems based on Bloom filters have to work around these performance and feature limitations. The limitations of the Bloom filter mean that (at least) two separate data structures need to be maintained: one for membership and one for counting. This requires all inserts to lookup and insert in multiple structures. Additionally, a single counting data structure is often more space efficient than a Bloom filter and hash table combination. Moreover, a Bloom filter does not support resizing, deletions, or counting.

The Bloom filter has inspired numerous variants that try to overcome one drawback or the other, e.g., [5, 30, 43, 62, 70, 108, 138, 139]. The counting Bloom filter (CBF) [70] 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. However, there is not one single variant that overcomes all the drawbacks.

Furthermore, exact k-mer counts are often not required. In such applications, memory usage can be reduced even further, and the simplicity of the underlying algorithm improved, by replacing the Bloom filter and exact counting data structure by a single probabilistic data structure. For example, [168] demonstrate that the count-min sketch [57] can be used to answer k-mer presence and abundance queries approximately. Such approaches can yield order-of-magnitude improvements in memory usage. However, a frequency estimation data structure, such as count-min sketch, can also blow up the memory usage for skewed data distributions, as often occur with k-mers in sequencing datasets [132].

There do exist more feature-rich AMQs. In particular, the counting quotient filter (CQF) [132], supports operations such as insertions, deletions, counting (even on skewed datasets), resizing, merging, and highly concurrent accesses. The false-positives in a counting quotient filter, similar to a Bloom filter, are always one sided and result in an over estimation of the actual count.

Results

In this chapter we show how to build a k-mer-counting and multiset-representation system using the counting quotient filter (CQF) [132].

We show that this off-the-shelf data structure is well suited to serve as a natural and efficient structure for representing and operating on multisets of k-mers (exactly or approximately). We make our case by developing and evaluating a k-mer-counting/querying system Squeakr (Simple Quotient filter-based Exact and Approximate Kmer Representation), which is based on the CQF.

Our CQF representation is space efficient and fast, and it offers a rich set of operations. The underlying CQF is easily tuned to trade off between space and accuracy/precision, depending upon the needs of the particular application. In the application of k-mer counting, we observe that the counting quotient filter is particularly well suited to the highly skewed distributions that are typically observed in HTS data.

Our representation is powerful, in part, because it is dynamic. Unlike, the Bloom filter [29], which is commonly used in k-mer-counting applications, Squeakr has the ability to modify and remove k-mers. Unlike the count-min sketch [57], Squeakr maintains nearly exact (or lossless) representations of the counts compactly.

In Squeakr, one can enumerate the hashes of the k-mers present in the structure, allowing k-mer multisets to be easily compared and merged. One interesting feature is that approximate multisets of different sizes can be efficiently compared and merged. This capability is likely to have other advantages beyond what we explore in this chapter; for example it could be instrumental in improving the sequence Bloom tree structure used for large-scale search [155].

We benchmark two settings of our system, Squeakr and Squeakr (exact), the latter of which supports exact counting via an invertible hash function, albeit at the cost of using more space. We compare both Squeakr and Squeakr (exact) with state-of-the-art k-mer counting systems KMC2 and Jellyfish2. Squeakr takes 2×–4.3× less time to count and perform a random-point-query workload (de Bruijn Graph graph traversal) than KMC2. Squeakr uses considerably less memory (1.5×–4.3×) than KMC2 and Jellyfish2. Squeakr offers insertion performance similar to that of KMC2 and faster than Jellyfish2. Squeakr offers an order-of-magnitude improvement in point query performance. We test the effect of query performance under both random and application-specific workloads (e.g., de Bruijn Graph traversal). For point query workloads Squeakr is 3.2×–24× faster than KMC2 and Jellyfish2. For de Bruijn Graph graph traversal workload Squeakr is 2×–4.3× faster than KMC2.

Squeakr offers orders-of-magnitude faster queries than existing systems while matching or beating them on counting performance and memory usage. Because most applications perform a combination of counting and querying, they can complete in less overall time with Squeakr than with other k-mer-counting systems.

4.2 Methods

We begin by first describing the design of Squeakr; how we use the counting quotient filter in Squeakr for counting k-mers, and how we efficiently parallelize Squeakr to scale with multiple threads.

4.2.1 Squeakr design

A k-mer counting system starts by reading and parsing the input file(s) (i.e., `FASTA/FASTQ` files) and extracting reads. These reads are then processed from left to right, extracting each read's constituent k-mers. The k-mers may be considered as they appear in the read, or, they may first be *canonicalized* (i.e., a k-mer is converted to the lexicographically smaller of the original k-mer and its reverse-complement). The goal of a k-mer

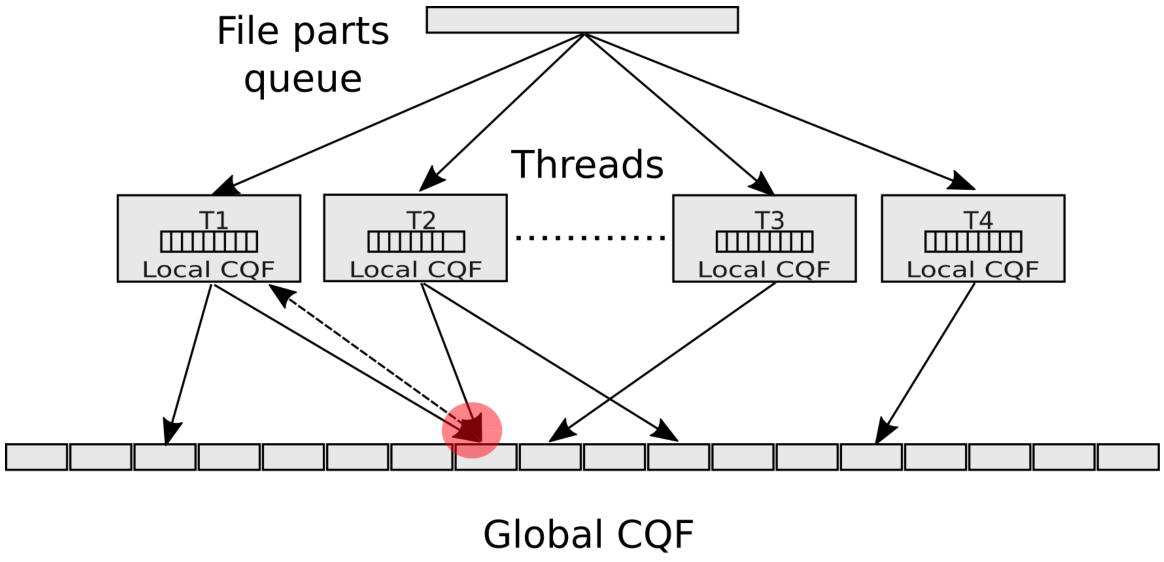


Figure 4.1: Squeakr system design: each thread has a local CQF and there is a global CQF. The dotted arrow shows that one thread did not get the lock in the first attempt and had to insert the item in the local CQF.

counting system is to count the number of occurrences of each k-mer (or canonical k-mer) present in the input dataset.

Squeakr has a relatively simple design compared to many existing k-mer counting systems. Many k-mer counting systems use multiple data structures, e.g., an Approximate Membership Query (AMQ) data structure to maintain all the singletons and a hash table [110] or compaction-and-sort based data structure [146] for the actual counting. The motivation behind using multiple data structures is primarily to reduce the memory requirements. Yet, having to maintain and modify multiple data structures when processing each k-mer can slow the counting process, and add complexity to the design of the k-mer counter. Other k-mer counting systems use domain specific optimizations (e.g., minimizers [144]) to achieve faster performance [65, 144].

Squeakr uses a single, off-the-shelf data structure, the counting quotient filter, with a straightforward system design, yet still offers superior performance in terms of memory and running and query time.

In Squeakr we have a single-phase process for counting k-mers in a read data set. Each thread performs the same set of operations; reading data from disk, parsing and extracting k-mers, and inserting k-mers in the counting quotient filter. The input file is read in chunks of 16MB, and each chunk is then parsed to find the last complete read record.

To synchronize operations among multiple threads, Squeakr uses a lock-free queue for storing the state of each file being read from disk, and a *thread-safe* counting quotient filter for inserting k-mers. Each thread executes a loop in which it grabs a file off the lock-free queue, reads the next 16MB chunk from the file[1], returns the file to the lock-free queue, and then parses the reads in the 16MB chunk and inserts the resulting k-mers into a shared, thread-safe quotient filter. This approach enables Squeakr to parallelize file reading and parsing, improving its ability to scale with more threads. KMC2, on the other hand, uses a single thread for each file to read and decompress which can sometimes become a bottleneck e.g., when one input file is much larger than others.

Squeakr uses a thread-safe counting quotient filter [132] for synchronizing insert operations among multiple threads. Using a thread-safe counting quotient filter, multiple threads can simultaneously insert k-mers into the data structure. Each thread acquires a lock on the region (each region consists of 4096 consecutive slots in the counting quotient filter) where the k-mer must be inserted and releases the lock once it is done inserting the k-mer. k-mers that hash to different regions of the counting quotient filter may be inserted concurrently.

This scheme of using a single, thread-safe CQF scales well with an increasing number of threads for smaller datasets, where the skewness (in terms of k-mer multiplicity) is not very high. However, for larger highly-skewed datasets, the thread-safe CQF scheme does not scale well. [2] This is due, in part, to the fact that these data have many highly-repetitive k-mers, causing multiple threads to attempt to acquire the same locks. This results in excessive lock contention among threads trying to insert k-mers, and prevents an increase in the number of threads.

Scaling with multiple threads. Large datasets with high skewness contain k-mers with very high multiplicity (of the order of billions). Such k-mers causes hotspots, and lead to excessive lock contention among threads in the counting quotient filter. To overcome the issue of excessive lock contention, Squeakr tries to reduce the time spent by threads waiting on a lock by amortizing the cost of acquiring a lock.

As explained in [132], the time spent by threads while waiting for a lock can be utilized to do local work. As shown in Figure 4.1, we assign a local counting quotient filter to each thread. Now, during insertion, each thread first tries to insert the k-mer in the thread-safe, global counting quotient filter. If the thread acquires the lock in the first attempt, then it inserts the k-mer and releases the lock. Otherwise, instead of waiting on the lock to be released, it inserts the k-mer in the local counting quotient filter and continues. Once the local counting quotient filter becomes full, the thread

[1]Threads actually read slightly less than 16MB, since threads always break chunks at inter-read boundaries.

[2]For the *G. gallus* dataset, though only .000561% of attempts did not obtain the lock in the first try the average and maximum spinning times were quite high, 2.137 Milliseconds and 3.072 seconds, respectively.

dumps the k-mers present in the local counting quotient filter into the global counting quotient filter before processing any new k-mers.

The above approach helps to reduce the time spent by threads while waiting on a lock and also amortizes the cost of acquiring a lock. Intuitively, repetitive k-mers are the ones for which it is hardest to acquire the lock in the global counting quotient filter. When a thread encounters such a k-mer and fails to obtain the corresponding lock in the global counting quotient filter, the thread instead immediately inserts those k-mers in the local (lockless) counting quotient filter and continues processing data. Moreover, these repetitive k-mers are first counted in the local counting quotient filter before being inserted with their corresponding counts in the global counting quotient filter. Under this design, instead of inserting multiple instances of the k-mer in the global counting quotient filter, requiring multiple acquisitions of a global counting quotient filter lock, Squeakr only insert the k-mers a few times with their counts aggregated via the local counting quotient filters.

In Squeakr even while maintaining multiple data structures, a local counting quotient filter per thread and a global counting quotient filter, one operation is performed for the vast majority of k-mers. While inserting k-mers that occur only a small number of times, threads obtain the corresponding lock in the global counting quotient filter in the first attempt. These k-mers are only inserted once. On the other hand, for repetitive k-mers, instead of acquiring the lock for each observation, we insert them in the local counting quotient filter and only insert them into the global counting quotient filter once the local counting quotient filter gets full.

4.2.2 Squeakr can also be an exact k-mer counter

Squeakr is capable of acting as either an approximate or an exact k-mer counter. In fact, this can be achieved with no fundamental changes to the underlying system; but simply by increasing the space dedicated to storing each hash's remainder, and by adopting an invertible hash function.

In Squeakr each k-mer in the read dataset is represented as a bit vector using $2k$ bits, i.e., each base-pair is represented using 2 bits. As explained in Chapter 3, to achieve a maximum allowable false-positive rate δ the counting quotient filter requires a p-bit hash function, where $p = \log_2 \frac{n}{\delta}$ and n is the number of distinct k-mers. For example, to achieve a false-positive rate of $1/512$ for a dataset with 2^{30} distinct k-mers, we need a 39-bit hash function. In Squeakr, we use the Murmur hash function [11] (`https://sites.google.com/site/murmurhash/`), by default, for hashing k-mers.

In the counting quotient filter, the p-bit hash is divided into q quotient bits and r remainder bits. The maximum false-positive rate is bounded by 2^{-r} [21]. In Squeakr, we assign $q = \log n$, where n is the number of distinct k-mers in the dataset and we use

dataset	File size	#Files	#k-mer instances	#Distinct k-mers
F. vesca	3.3	11	4,134,078,256	632,436,468
G. gallus	25.0	15	25,337,974,831	2,727,529,829
M. balbisiana	46.0	2	41,063,145,194	965,691,662
H. sapiens 1	67.0	6	62,837,392,588	6,353,512,803
H. sapiens 2	99.0	48	98,892,620,173	6,634,382,141

Table 4.1: Datasets used in the experiments. The file size is in GB. All the datasets are compressed with gzip compression.

9-bit remainders to achieve a false-positive rate of 1/512.

In order to convert Squeakr from an approximate k-mer counter to an exact k-mer counter, we need to use a p-bit *invertible* hash function, where $p = 2k$. In Squeakr (exact), we use the Inthash hash function [104] (`https://gist.github.com/lh3/974ced188be2f90422cc`) for hashing k-mers. For a dataset with n distinct k-mers and a p-bit hash function, the remainder $r = p - \log_2 n$. For example, for $n = 2^{30}$ and $k = 28$ (i.e., $p = 56$), we need $r = 26$ bits. Also, the counting quotient filter uses 2.125 metadata bits per item, see [132]. This is still far less than the 56 bits that would be required to store each k-mer key explicitly.

4.3 Evaluation

In this section we evaluate our implementations of Squeakr and Squeakr (exact). Squeakr (exact), as described in Section 4.2.2, is an exact k-mer counting system that uses the counting quotient filter with a p-bit invertible hash function, where p is the number of bits to represent a k-mer in binary. Squeakr is an approximate k-mer counting system that also uses the counting quotient filter but takes much less space than Squeakr (exact). The space savings comes from the fact that Squeakr allows a small false-positive rate.

We compare both versions of Squeakr with state-of-the-art k-mer counting systems in terms of speed, memory efficiency, and scalability with multiple threads. We compare Squeakr against two k-mer counting systems; KMC2 [60] and Jellyfish2 [110]. KMC2 is currently the fastest k-mer counting system [65], although not the most frugal in terms of memory usage (when not run in disk-based mode). Jellyfish2, though not the fastest or most memory-frugal system, is very widely used, and internally uses a domain specific hash-table to count k-mers, and is thus methodologically similar to Squeakr.

Khmer [168] is the only approximate multiset representation and uses a count-min sketch. Here, we don't compare Squeakr against Khmer, since they are geared toward somewhat different use-cases. Squeakr exhibits a very small error rate, and is intended to be used in places where one might otherwise use an exact k-mer counter, while Khmer is designed much more as a sketch, to perform operations on streams of k-mers for which near-exact counts are not required.

Squeakr is an in-memory k-mer counter and we compare it against other in-memory k-mer counting systems. We compare Squeakr with other systems for two different k-mer sizes. Squeakr (exact) is an in-memory and exact k-mer counter. Squeakr (exact) currently only support k-mers of maximum length 32, though, it is not a fundamental limitation of the counting quotient filter.

We evaluate each system on two fundamental operations, counting and querying. We use multiple datasets to evaluate counting performance, and a subset of those datasets for query performance. We evaluate queries for existing k-mers and absent k-mers (uniformly random k-mers) in the dataset. We also evaluate Squeakr for performing queries for k-mers as they appear in the context of de Bruijn Graph traversal. Traversing the de Bruijn Graph is a critical step in any De-Bruijn-graph-based assembly, and using an AMQ for a compact representation and fast traversal of the de Bruijn Graph has been shown in the past [135]. To evaluate the ability of the counting quotient filter to represent a de Bruijn Graph and deliver fast query performance during traversal, we performed a benchmark where we query k-mers as they appear in the de Bruijn Graph.

Other than counting and queries, we also evaluate Squeakr for computing the inner-product between the k-mer abundance vectors of a pair of datasets. The comparison of the k-mer composition of two different strings (or entire datasets) has proven a fruitful approach for quickly and robustly estimating their overall similarity. In fact, many methods exist for the so-called alignment-free comparison of sequencing data [163]. Recently, [117] introduced the k-mer weighted inner product as an estimate of the similarity between genomic/metagenomic datasets. Prior work suggests that ability to compute fast inner-product between two datasets is important. To assess the utility of the CQF in enabling such types of comparisons, we performed a benchmark to evaluate the performance of Squeakr to compute the inner-product (or cosine-similarity) between two datasets.

4.3.1 Experimental setup

Each system is tested for two k-mer sizes, 28 and 55, except for Squeakr (exact) which is only evaluated for 28-mers (because the current implementation is limited to $k \leq 32$). All the experiments are performed in-memory. We use several datasets for our experiments, which are listed in Table 4.1. All the experiments were performed on an Intel(R) Xeon(R) CPU (E5-2699 v4 @ 2.20GHz with 44 cores and 56MB L3 cache) with 512GB RAM and a 4TB TOSHIBA MG03ACA4 ATA HDD. In order to evaluate the scalability of the systems with multiple threads, we have reported numbers with 8 and 16 threads for all the systems and for each dataset. In all our experiments, the counting quotient filter was configured with a maximum allowable false-positive rate of 1/256. For reporting time and memory metrics, we have taken the average over two runs for all the benchmarks. The time reported in all the benchmarks is in seconds and memory (RAM) is in GBs.

Counting benchmarks. For each dataset we have only counted the canonical k-mers. The time reported for counting benchmarks is the total time taken by the system to read data off disk, parse it, count k-mers, and write the k-mer representation to disk.

The memory reported is the maximum RAM required by the system while counting k-mers, as given by `/usr/bin/time`, which is the maximum resident set size. In our experiments, RAM usage was almost always the same, regardless of the number of threads, so the RAM mentioned in Table 4.2 is the average RAM required by the system for 8 and 16 threads. Both Squeakr and Jellyfish2 require to give as a parameter the number of distinct k-mers in the dataset. Squeakr needs the number of distinct k-mers (approximate to next closet power of 2) as an input. Squeakr takes the approximation of number of distinct k-mers as the number of slots to create the CQF. We used [115] to estimate the number of distinct k-mers in datasets. As explained in Section 4.2.1, KMC2 can be bottlenecked to decompress bzip2 compressed files. Therefore, we recompress all the files using gzip. In our experiments gzip decompression was never a bottleneck.

Query benchmarks. We performed three different benchmarks for queries. First, we randomly queried for all the k-mers in the dataset. First, we randomly queried for all k-mers that we knew existed in the dataset. Second, we queried for 1.5 billion uniformly random k-mers (i.e., uniformly random 28-mers), most of which are highly unlikely to exist in the dataset. Third, we performed a de Bruijn Graph traversal, walking the paths in the de Bruijn Graph and querying the neighbors of each node. We have performed the query benchmarks on two different datasets, *G. gallus* and *M. balbisiana*. Also, we excluded Jellyfish2 in the de Bruijn Graph benchmark because Jellyfish2's random query performance was very slow for the first two query benchmarks. We note here that this appears to be a result of the fact that Jellyfish2 uses a sorted, compacted list to represent the final counts for each k-mer, rather than the hash table that is used during counting. This helps to minimize on-disk space, but results in logarithmic random query times.

The de Bruijn Graph traversal benchmark measures the time to search the graph for the longest non-branching path. That is, for each k-mer from the original dataset, we take the suffix of length $k-1$ and append each of the four possible bases to generate four new k-mers [135]. Then we perform four separate queries for these newly generated k-mers in the database. If there is more than one adjacent k-mer present (i.e., a fork) then we stop. Otherwise, we continue this process. At the end, we report the total

dataset	**KMC2**	**Squeakr**	**Squeakr (exact)**	**Jellyfish2**
F. vesca	8.3	4.8	9.3	8.3
G. gallus	32.8	13.0	28.8	31.7
M. balbisiana	48.3	11.1	14.2	16.3
H. sapiens 1	71.4	22.1	51.5	61.8
H. sapiens 2	107.4	30.8	60.1	61.8

Table 4.2: Gigabytes of RAM used by KMC2, Squeakr, Squeakr (exact), and Jellyfish2 for various datasets for in-memory experiments for $k = 28$.

time taken and the longest path in the de Bruijn Graph.

For query benchmarks, we report only the time to query k-mers in the database. We exclude the time to read k-mers from an input file or to generate the uniformly random query k-mers. Also, we first load the database completely in memory for all the systems before performing any queries. In case of KMC2 we load the database in random-access mode. For the de Bruijn Graph traversal queries, we generate k-mers to traverse the graph on-the-fly. The time reported for the de Bruijn Graph traversal query includes the time to generate these k-mers.

For the inner-product query benchmark, in Squeakr, we first count the k-mers from two datasets and store the k-mer representations on disk. We then compute the inner-product between the two representations by querying k-mers from one dataset in the other dataset. We compare inner-product performance of Squeakr against Mash [123]. For Mash, we first compute the representations of the fastq files and store them on disk. We then perform inner-product of these representations. For both systems, we report the total time taken to create the representations and computing inner-product. KMC2 exposes an API to perform intersection on two k-mer representations. To compare against KMC2, we implemented an API in Squeakr for computing intersection on two on-disk counting quotient filters. We report the time taken to compute the intersection and write the output to disk.

4.3.2 Memory requirement

Table 4.2 shows the maximum memory required by KMC2, Squeakr, Squeakr (exact), and Jellyfish2 for counting k-mers from different datasets. Squeakr requires the least RAM compared to the other systems. Even for the human datasets, Squeakr is very frugal in memory usage and completes the experiment in $\approx$ 30GB of RAM. Across all datasets, Squeakr takes 1.5X—4.3X less RAM than KMC2 (in in-memory mode).

Squeakr (exact) takes less RAM than KMC2 for all (except *F. vesca*) datasets and Jellyfish2 for human datasets. For smaller datasets, Squeakr (exact) takes approxi-

System	*F. vesca*		*G. gallus*		*M. balbisiana*		*H. sapiens 1*		*H. sapiens 2*	
	8	16	8	16	8	16	8	16	8	16
KMC2	91.68	67.76	412.19	266.546	721.43	607.78	1420.45	848.79	1839.75	1247.71
Squeakr	116.56	64.44	739.49	412.82	1159.65	662.53	1931.97	1052.73	3275.20	1661.77
Squeakr (exact)	146.56	80.58	966.27	501.77	1417.48	763.88	2928.06	1667.98	5016.46	2529.46
Jellyfish2	257.13	172.55	1491.25	851.05	1444.16	886.12	4173.3	2272.27	6281.94	3862.82

Table 4.3: k-mer counting performance of KMC2, Squeakr, Squeakr (exact), and Jellyfish2 on different datasets for $k = 28$.

System	*F. vesca*		*G. gallus*		*M. balbisiana*		*H. sapiens 1*		*H. sapiens 2*	
	8	16	8	16	8	16	8	16	8	16
KMC2	233.74	123.87	979.20	1117.35	1341.01	1376.51	3525.41	2627.82	4409.82	3694.85
Squeakr	138.32	75.48	790.83	396.36	1188.15	847.83	2135.71	1367.56	3320.67	2162.97
Jellyfish2	422.220	294.93	1566.79	899.74	2271.33	1189.01	3716.76	2264.70	6214.81	3961.53

Table 4.4: k-mer counting performance of KMC2, Squeakr, and Jellyfish2 on different datasets for $k = 55$.

mately the same amount of RAM as Jellyfish2.

4.3.3 Counting performance

Table 4.3 and Table 4.4 show the time taken by different systems to count the k-mers present in the datasets.

For 55-mers, Squeakr is 11%—64% faster than KMC2 and 28%—74% faster than Jellyfish2. For 28-mers, Squeakr's counting speed is between 5% faster and 79% slower than KMC2. However, as weshow in the next section, Squeakr is orders-of-magnitude faster than KMC2 on queries, so applications finish faster overall with Squeakr than with KMC2.

Although Squeakr is slower than KMC2 for 28-mers on most datasets it is much faster than KMC2 for 55-mers. This is because increasing the size of k-mers has little effect on hashing (in case of Squeakr) and bigger effect on string processing (in case of KMC2).

Squeakr exhibits better concurrency than KMC2. For 28-mers, Squeakr speeds up by 43%—49% when going from 8 to 16 threads, whereas KMC2 speeds up by only 16%—40%. For 55-mers, Squeakr speeds up by 29%—50% when going from 8 to 16 threads, whereas KMC2 sometimes slows down and never speeds up by more than 26% except on *f.vesca*. Squeakr and Jellyfish2 exhibit similar scaling, perhaps with a slight edge to Squeakr, which is also much faster in absolute terms.

Squeakr (exact) is slower than Squeakr for all datasets tested. However, we find that it is always faster than Jellyfish2.

	G. gallus		*M. balbisiana*	
System	**Existing**	**Non-existing**	**Existing**	**Non-existing**
KMC2	1495.82	470.14	866.93	443.74
Squeakr	303.68	52.45	269.24	40.73
Squeakr (exact)	389.58	58.46	280.54	42.67
Jellyfish2	884.17	978.57	890.57	985.30

Table 4.5: Random query performance of KMC2, Squeakr, Squeakr (exact), and Jellyfish2 on two different datasets for $k = 28$.

4.3.4 Query performance

Random query for existing k-mers. Table 4.5 shows the random query performance for existing k-mers. Squeakr is 3.2X—4.9X faster than KMC2 for random queries for existing k-mers. Jellyfish2 is the slowest. This is likely because the on-disk representation used by Jellyfish2 is a compacted, sorted list, not the hash table used during the counting phase.

Random query for uniformly-random k-mers. Table 4.5 shows the random query performance for uniformly-random k-mers. For uniformly-random k-mers, Squeakr is 8.9X—10.8X faster than KMC2. Squeakr is even faster for uniformly-random queries than when querying for existing k-mers because there is a fast path for non-existing items in the counting quotient filter. For non-existing items, the counting quotient filter often returns the result by examining a single bit. Jellyfish2 is the slowest among the three for uniformly-random k-mer queries.

Both Squeakr and Squeakr (exact) have similar query performance, with Squeakr (exact) being slightly slower because the exact version requires a larger counting quotient filter structure.

We also evaluated the empirical false-positive rate of Squeakr, which we find to be close to the theoretical false-positive rate. As mentioned in Section 4.3.1, the theoretical false-positive rate is 1/256 i.e., .00390625. The empirical false-positive rate reported during the benchmark is 0.0012414. The false-positive rate in Squeakr is uniformly distributed across k-mers irrespective of their abundance.

de Bruijn Graph traversal. Table 4.6 shows the de Bruijn Graph traversal performance of Squeakr and KMC2.

Since Squeakr is much faster than KMC2 for both type of queries, existing k-mers and non-existing k-mers, it performs 2.7X—6.7X faster than KMC2 for de Bruijn Graph traversal queries. For the de Bruijn Graph traversal, we perform 4 queries for each k-mer. To continue on the path, only one out of the four queries should return true. In the whole benchmark, $\approx 75\%$ of the queries are false queries.

Table 4.6 also reports the longest path present in the graph reported by both the

			Running Times		
System	**dataset**	**Max path len**	**Counting**	**Query**	**Total**
KMC2	*G. gallus*	122	266	23097	23363
Squeakr	*G. gallus*	92	412	3415	3827
KMC2	*M. balbisiana*	123	607	6817	7424
Squeakr	*M. balbisiana*	123	662	1471	2133

Table 4.6: de Bruijn Graph query performance on different datasets. The counting time is calculated using 16 threads. The query time is calculated using a single thread. Time is in seconds. We excluded Jellyfish2 from this benchmark because Jellyfish2 performs slowly compared to KMC2 and Squeakr for both counting and query (random query and existing k-mer query).

systems. Squeakr, being an approximate k-mer counter, has some false-positives. The length of the longest path reported by Squeakr is shorter for *G. gallus* dataset and same for *M. balbisiana*. The shorter path is seen due to a fork caused by a false-positive k-mer during the traversal.

In the table, we also present the time taken to count k-mers in the dataset and the total time (i.e., counting time and de Bruijn Graph traversal time). Squeakr is 1.7X—6.4X faster than KMC2 in terms of total time.

4.3.5 Inner-product queries

For inner-product queries, we first counted k-mers from two different datasets, each having $\approx$ 975 Million k-mer instances and $\approx$ 91 Million and $\approx$ 136 Million distinct k-mers respectively and stored the output on disk. It took 349.46 seconds for Squeakr to read raw fastq files, count k-mers, and compute the inner-product between the two datasets. We gave raw fastq files as input to both Squeakr and Mash. Mash took 329.65 seconds to compute the distance between the two datasets. This suggests that the CQF k-mer multiset representation provides comparable performance to the state-of-the-art tool for computing similarity between two k-mer datasets. This feature can be used for large-scale comparison and organization of sequencing datasets.

We performed intersection on the same two datasets as used for inner-product query. KMC2 took 4.29 seconds using 6 threads. While Squeakr took 4.57 seconds using only 3 threads. This shows that Squeakr is faster than KMC2 for workloads comparing two datasets.

Chapter 5: deBGR: An efficient and near-exact representation of the weighted de Bruijn Graph

We now describe the second application of the counting quotient filter to computational biology. In this application, we show how we can iteratively correct all approximation errors in the CQF-based approximate weighted de Bruijn Graph representation described in Chapter 4. We use the approximate counts stored in the counting quotient filter and an invariant of the weighted de Bruijn Graph to iteratively correct errors.

5.1 de Bruijn Graphs

The de Bruijn Graph has become a fundamental tool in genomics [53] and the de Bruijn Graph underlies almost all short-read genome and transcriptome assemblers—[45, 84, 93, 106, 137, 153, 154, 167]—among others.

Despite the computational benefits that the de Bruijn Graph provides above the overlap-layout-consensus paradigm, the graph still tends to require a substantial amount of memory for large data sets. This has motivated researchers to derive memory-efficient de Bruijn Graph representations. Many of these representations build upon approximate membership query (AMQ) data structures (such as Bloom filters) to achieve an economy of space.

Approximate membership query data structures are set (or multiset) representations that achieve small space requirements by allowing queries, occasionally, to return false positive results. The Bloom filter [29] is the archetypal example of an AMQ. Bloom filters began to gain notoriety in bioinformatics when [112] showed how they can be coupled with traditional hash tables to vastly reduce the memory required for k-mer counting. By inserting k-mers into a Bloom filter the first time they are observed, and adding them to the higher-overhead exact hash table only upon subsequent observations. Later, [168] demonstrated that the count-min sketch [57] (a frequency estimation data structure) can be used to approximately answer k-mer presence and abundance queries when one requires only approximate counts of k-mers in the input. Such approaches can yield order-of-magnitude improvements in memory usage over competing methods.

These ideas were soon applied to the construction and representation of the de Bruijn Graph. For example, [135] introduce a completely probabilistic representation of the de Bruijn Graph using a Bloom filter to represent the underlying set of k-mers. Though this representation admits false positives in the edge set, they observe that this has little effect on the large-scale structure of the graph until the false positive rate becomes very high (i.e., ≥ 0.15).

Building upon this probabilistic representation, [48] introduce an exact de Bruijn Graph representation that couples a Bloom-filter-based approximate de Bruijn Graph with an exact table storing *critical false positive* edges. Chikhi and Rizk's de Bruijn Graph representation exploits the fact that, in the de Bruijn Graph, there are very few edges connecting true-positive k-mers to false-positive k-mers of the Bloom filter representation of the k-mer set. Such edges are called critical false positives. Further, they observe that eliminating these critical false positives is sufficient to provide an exact (navigational) representation of the de Bruijn Graph. This compact representation allows large de Bruijn Graphs to be held in RAM, which enables relatively efficient assembly of even large and complex genomes.

Subsequently, the representation of Chikhi and Rizk was refined by [149], who improved the memory requirements even further by replacing the exact table with a cascading Bloom filter. The cascading Bloom filter stores an approximate set using a combination of an approximate (i.e., Bloom filter-based) representation of the set and a smaller table to record the relevant false-positives. This construction can be applied recursively to substantially reduce the amount of memory required to represent the original set. [149] provide a representation that requires as little as $8-9$ bits per k-mer, yet remains exact from a navigational perspective. Even more memory-efficient exact representations of the unweighted de Bruijn Graph are possible. For example, [32] introduced the succinct de Bruijn Graph (often referred to as the BOSS representation), which provides an exact navigational representation of the de Bruijn Graph that uses < 5 bits per k-mer, which compares favorably to the lower bound of ≈ 3.24 bits per k-mer on navigational representations [47].

While the above approaches used auxiliary data structures to correct errors in an approximate representation of the de Bruijn Graph, [136] showed how to exploit redundancy in the de Bruijn Graph itself to correct errors. Essentially, they observed that true k-mers are not independent—each true k-mer will have a $k-1$-base overlap with another true k-mer. If a Bloom filter representation of the de Bruijn Graph indicates that a particular k-mer x exists, but that no k-mer overlapping x exists, then x is likely to be a false positive. Thus, by checking for the existence of all overlapping k-mers, they can dramatically reduce the false-positive rate of a Bloom-filter-based de Bruijn Graph representation. Our representation of the weighted de Bruijn Graph can be viewed as an extension and generalization of this basic idea. See Sections 5.2.1 and 5.3 for details.

However, the Bloom filter omits critical information—the frequency of each k-mer—that is necessary when performing assembly of a transcriptome. Thus, "topology-only" representations are inadequate in the case where knowing the abundance of each transcript, and by extension, each k-mer in the de Bruijn Graph that is part of this transcript, is essential. In the transcriptomic context, then, one is interested primarily in the *weighted de Bruijn Graph* (see Definition 2). The weighted de Bruijn Graph associates with each k-mer its abundance in the underlying data set upon which the de Bruijn Graph was constructed. Unlike the case of genomic assembly, we expect the counts in the weighted de Bruijn Graph for transcriptomic data to have a very large dynamic range, and maintaining exact or near-exact counts for each k-mer can be important for accurately identifying transcripts.

In this chapter we introduce a memory-efficient and near-exact representation of the weighted de Bruijn Graph. Our representation is based upon a recently-introduced counting filter data structure [132] which, itself, provides an approximate representation of the weighted de Bruijn Graph. Observing certain abundance-related invariants that hold in an *exact* weighted de Bruijn Graph, we devise an algorithm that uses this approximate data representation to iteratively self-correct approximation errors in the structure. The result is a data structure that takes 18%–28% more space than the approximate representation and has zero errors. This makes our new representation, which we call deBGR, a near-exact representation of the weighted de Bruijn Graph. In datasets with billions of distinct k-mers, deBGR typically exhibits zero topological errors. Further, our algorithm corrects not only the topology of the approximate representation, but also misestimates of abundance that result from collisions in the underlying counting filter.

Additionally, while existing space-efficient representations of the de Bruijn Graph, are static, i.e., k-mers cannot easily be deleted from the graph; our representation supports removal of edges from the de Bruijn Graph. This capability is enabled by the counting quotient filter's ability to delete items (which cannot be done reliably in Bloom filters). Since aggressive simplification of the de Bruijn Graph (e.g., to remove spurious topology like bubbles and tips) is typically done prior to assembly, this deletion capability is important. Previous approaches avoided the standard simplification step by instead adopting more complicated traversal algorithms [48]. By removing this limitation of the Bloom filter, our representation benefits both from simpler traversal algorithms which allow the in-memory creation of a more manageable *simplified* weighted de Bruijn Graph. Recently, [15] have introduced a dynamic representation of the unweighted de Bruijn Graph based on perfect hashing, and it will be interesting to explore the ability of this approach to represent the weighted de Bruijn Graph. However, to the best of our knowledge, this representation has not yet been implemented.

We believe that our representation of the weighted de Bruijn Graph can be successfully applied to considerably reduce the computational requirements for de Bruijn Graph-based transcriptome assembly [45, 84, 93, 106]. One of the major benefits of our approach is that weighted de Bruijn Graph construction should require considerably less memory than the approaches taken by these other tools. This will allow for the assembly of larger and more complicated transcriptomes on smaller and less expensive computers. Further, since our compact representation of the de Bruijn Graph can be kept completely in memory, even for relatively large transcriptomes, we can avoid the *ad hoc* and potentially complicated step of partitioning the de Bruijn Graph for further processing [93, 135].

5.2 Background

deBGR is built on our prototype k-mer counter Squeakr [132], which is in turn built on our counting quotient filter data structure [132]. We explain the key features of these systems that are needed to understand deBGR. We then review prior work on exploiting redundancy in de Bruijn Graphs to correct errors in approximate de Bruijn Graph representations. We also note that, throughout the chapter, we assume a DNA (i.e., 4 character) alphabet.

5.2.1 Prior approximate de Bruijn Graph representations

deBGR extends and generalizes an idea first suggested by [136], for correcting errors in approximate de Bruijn Graph representations.

The Bloom filter false-positive rate is calculated assuming all the items inserted in the Bloom filter are independent. However, when we use a Bloom filter to represent a de Bruijn Graph, the items (or k-mers in this case) are not independent. Each k-mer has a $k-1$-base overlap with adjacent k-mers in the sequence.

[136] use this redundancy to detect false positives in a Bloom filter representation of the de Bruijn Graph. Whenever they want to determine whether a k-mer x is present in the de Bruijn Graph, they first query the Bloom filter for x. If the Bloom filter indicates that x is not present, then they know that x is not in the de Bruijn Graph. If, however, the Bloom filter indicates that x might be in the de Bruijn Graph, they then query the Bloom filter for every possible k-mer that overlaps x in $k-1$ bases. If the Bloom filter indicates that none of these k-mers is part of the de Bruijn Graph, then x is very likely to be a false positive. If the Bloom filter returns true for at least one of

the k-mers overlapping with x, then they conclude that x is very likely to be in the de Bruijn Graph.

[136] present two versions of the k-mer Bloom filter, a one-sided k-mer Bloom filter and a two-sided k-mer Bloom filter. The one-sided k-mer Bloom filter only looks for the presence of a single overlapping neighbor out of the eight possible neighbors (four on each side) of a k-mer x. The one-sided k-mer Bloom filter achieves a smaller false-positive rate than a standard Bloom filter using the same space.

The two-sided k-mer Bloom filter achieves an even lower false-positive rate by requiring that there is an overlapping k-mer present on either side of x. However, this approach can result in false-negative results for k-mers that are at the edges of reads, since the k-mers at the edges might not have neighbors on both sides.

The two-sided k-mer Bloom filter deals with the k-mers at the edges of reads (i.e., start and end k-mers) specially. It maintains a separate list that contains all the k-mers that occur at the beginning or end of a read. While constructing the k-mer Bloom filter, the first and last k-mer of each read are stored in separate lists. During a query for x, if it finds a neighboring k-mer on only one side of x, then it checks whether x is in the list of edge k-mers. If yes, then it returns positive; else it returns negative.

5.2.2 Lower bounds on weighted de Bruijn Graph representation

In the experiments we perform in Section 5.4, we find that deBGR is practically exact from a navigational perspective (i.e., it yields zero errors in terms of topology or abundance). It is useful, therefore, to keep in mind some lower bounds for what is achievable in representing the weighted de Bruijn Graph exactly from a navigational perspective. We know that a navigational structure for the unweighted de Bruijn Graph requires at least 3.24 bits per kmer [47], and that exactly representing the counts requires at least $F = \sum_{k \in K} \lceil \log_2(f_k) \rceil$ bits where K is the set of k-mers in a data set and f_k is the frequency of k-mer k, so that a reasonable lower bound would be $3.24 + \frac{F}{|K|}$ bits per k-mer. To make such a representation efficient would likely require more space (e.g., a fast way to index the encoded, variable-size frequency data).

We consider what this bound implies for the data set `GSM984609` in Section 5.4. Here we have 1 146 347 598 distinct k-mers and F =1 119 742 769, yielding a lower bound of ≈ 4.217 bits per k-mer for an exact navigational representation of this weighted de Bruijn Graph. Approaching such a bound closely, is, of course, a challenge. For example, the cosmo [1] implementation of the BOSS data structure requires ≈ 5.995 bits per k-mer on this data set, but does not encode the weight of each edge. If we couple this with an array of fixed-size counters large enough to represent the frequency distribution

[1] `https://github.com/cosmo-team/cosmo`

losslessly (for this data set, 23 bits per k-mer), it yields a representation requiring $\approx$ 28.995 bits per k-mer. deBGR, on the other hand, requires 26.52 bits per k-mer. Thus, this example shows that there is still a considerable gap between what existing approaches achieve and the absolute theoretical lower bound for an exact navigational representation of a weighted de Bruijn Graph. However, deBGR is dynamic, supports membership queries, and provides efficient access (expected $O(1)$) to k-mer abundances.

5.3 Methods

We begin by first presenting an invariant of de Bruijn Graphs that we exploit in our compact de Bruijn Graph representation. We then describe how we use this invariant to extend Squeakr [132] to create a near-exact representation of the weighted de Bruijn Graph.

5.3.1 A weighted de Bruijn Graph invariant

This section explains the structure of weighted de Bruijn Graphs that we exploit to correct errors in approximate weighted de Bruijn Graph representations, such as that provided by Squeakr.

Definition 1. *For a k-mer x, we will denote its reverse complement as x^{-1}. The* ***canonical form*** *of a k-mer x, denoted $\widehat{x}$, is the lexicographically smaller of x and x^{-1}. For two k-mers x and y, we write $x \simeq y$ if $\widehat{x} = \widehat{y}$.*

A ***read*** is a string of bases over the DNA alphabet A, C, T, and G.

Definition 2. *The weighted* ***de Bruijn Graph G of k-mers for a set of reads R*** *has a node $\widehat{n}$ for each $(k-1)$-mer n that occurs in R. For each k-mer b_1xb_2 in R, where b_1 and b_2 are bases and x is a $(k-2)$-mer, there is an edge $\widehat{b_1xb_2}$ connecting the nodes $\widehat{b_1x}$ and $\widehat{xb_2}$. The* ***abundance*** *of an edge $\widehat{e}$, denoted $a(\widehat{e})$, is the number of times that $\widehat{e}$ (i.e., e or e^{-1}) occurs in R.*

In this formalization, a read of length ℓ corresponds to a walk of length $\ell - k$ edges in the de Bruijn graph.

Definition 3. *For a node $\widehat{n}$ and edge $\widehat{e}$, we say that $\widehat{e}$ is a* ***duplex edge of $\widehat{n}$*** *if there exist bases b and b' (and possibly $b = b'$) such that $\widehat{e} \simeq b\widehat{n}$ and $\widehat{e} \simeq \widehat{n}b'$. We say that $\widehat{e}$ is a* ***left edge of $\widehat{n}$*** *if $\widehat{e}$ is not a duplex edge of $\widehat{n}$ and there exists a base b such that $\widehat{e} \simeq b\widehat{n}$. Similarly, $\widehat{e}$ is a* ***right edge of $\widehat{n}$*** *if $\widehat{e}$ is not a duplex edge of $\widehat{n}$ and there exists a base b such that $\widehat{e} \simeq \widehat{n}b$.*

There are several subtleties to this definition. Left, right, and duplex are defined relative to a node $\widehat{n}$. An edge $\widehat{e}$ connecting nodes $\widehat{n_1}$ and $\widehat{n_2}$ may be a left edge of $\widehat{n_1}$ and a right edge of $\widehat{n_2}$, or a left edge of both $\widehat{n_1}$ and $\widehat{n_2}$, or any other combination. Note also that left, right, and duplex are mutually exclusive—every edge of $\widehat{n}$ is exactly one of left, right, or duplex, with respect to $\widehat{n}$.

Our compact representation of the de Bruijn graph is based on the following observation:

Observation 1. *Let $\widehat{e_1}, \widehat{e_2}, \ldots, \widehat{e_\ell}$ be the sequence of edges in a walk corresponding to a read, and $\widehat{n_0}, \ldots, \widehat{n_\ell}$ the corresponding sequence of nodes in the walk. Then for $i = 1, \ldots, \ell - 1$, $\widehat{e_i}$ and $\widehat{e_{i+1}}$ cannot both be left edges of $\widehat{n_i}$, nor can they both be right edges of $\widehat{n_i}$.*

In other words, whenever a read arrives at a node $\widehat{n}$ via a left edge of $\widehat{n}$, it must depart via a right or duplex edge of $\widehat{n}$, and whenever it arrives via a right edge of $\widehat{n}$, it must depart via a left or duplex edge of $\widehat{n}$. (When a walk arrives via a duplex edge, it may leave via any kind of edge.) This is because two successive edges of the walk correspond to a substring $b_1 n b_2$ of the read, where b_1 and b_2 are bases and n is a $(k-1)$-mer. If $\widehat{n} = n$, then $\widehat{b_1 n}$ is a left (or duplex) edge of $\widehat{n}$ and $\widehat{n b_2}$ is a right (or duplex) edge of $\widehat{n}$. If $\widehat{n} = n^{-1}$, then $\widehat{b_1 n} \simeq \widehat{n} b_1^{-1}$ is a right (or duplex) edge of $\widehat{n}$, and $\widehat{n b_2} \simeq b_2^{-1} \widehat{n}$ is a left (or duplex) edge of $\widehat{n}$.

The following lemma implies that duplex edges are rare, since only nodes of a special form can have duplex edges.

Lemma 1. *If a node $\widehat{n}$ has a duplex edge, then either (1) $n = n^{-1}$ or (2) $\widehat{n}$ is equal to either A^{k-1} or C^{k-1}, where A and C are the DNA bases.*

Proof. Suppose node $\widehat{n}$ has a duplex edge $\widehat{e}$. Without loss of generality, we can assume $\widehat{n} = n$ (by replacing n with n^{-1} if necessary). Then there exist (possibly equal) bases b and b' such that $bn \simeq \widehat{e} \simeq nb'$, i.e., $bn \simeq nb'$. Let $n = n_1 \cdots n_{k-1}$, i.e., n_i are the bases constituting n. Then there are two cases:

- $bn = nb'$. In this case, $b = n_1 = n_2 = \cdots = n_{k-1} = b'$, i.e., n is a string of equal bases. Thus n is equivalent to A^{k-1} or C^{k-1}.
- $bn = b'^{-1} n^{-1}$. Thus $n = n^{-1}$. □

□

We call nodes that can have duplex edges ***duplex nodes***, for example see Figure 5.2.

We say that a walk, path, or cycle is ***left-right-alternating*** if, for every successive pair of edges $\widehat{e}$ and $\widehat{e'}$ in the path, walk, or cycle, one is a left edge of $\widehat{n}$ and one is a right edge of $\widehat{n}$, where $\widehat{n}$ is the node common to $\widehat{e}$ and $\widehat{e'}$. We say that nodes $\widehat{n}$ and $\widehat{n'}$

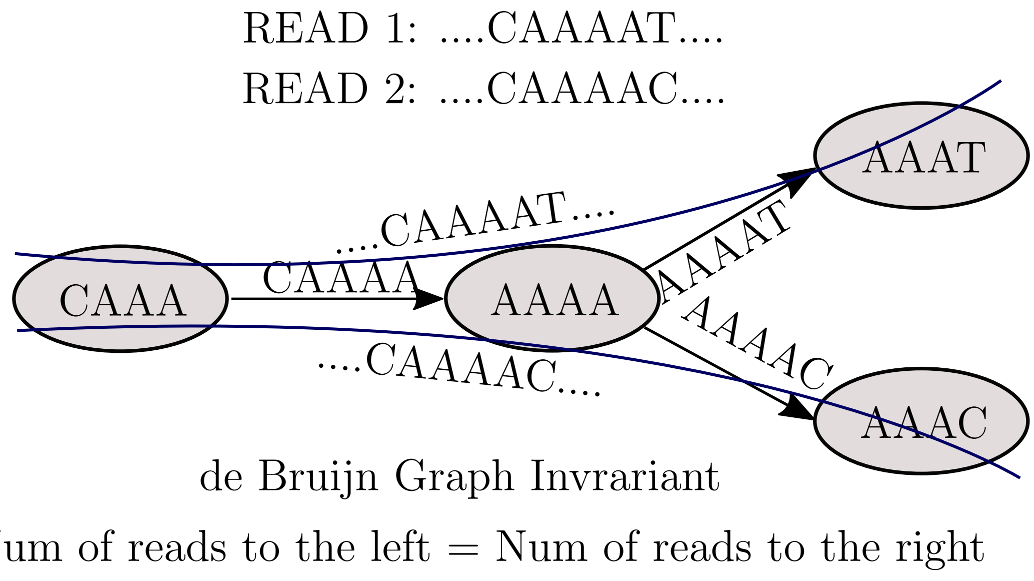


Figure 5.1: **Weighted de Bruijn Graph invariant.** The nodes are 4-mers and edges are 5-mers. The nodes and edges are drawn from Read1 and Read2 mentioned in the figure. The solid curve shows the read path. The nodes/edges are not canonicalized.

have left-right-alternating distance d if the shortest left-right-alternating path from $\widehat{n}$ to $\widehat{n'}$ has length d.

We now explain the main invariant used in our compact weighted de Bruijn Graph representation, as illustrated in Figure 5.1.

This observation leads to the following invariant.

Theorem 3 (The weighted de Bruijn Graph invariant)**.** *Let $\mathcal{R}$ be a set of reads that does not include any duplex edges. Let $a(\widehat{e})$ be the number of occurrences of the edge $\widehat{e}$ in a set of reads. Let $\ell(\widehat{n})$ be the number of reads that begin or end with a left edge of $\widehat{n}$, and $r(\widehat{n})$ the number of reads that begin or end with a right edge of $\widehat{n}$. Let $s(\widehat{e})$ be 1 if $\widehat{e}$ is a self-loop, and 0 otherwise. Let $\widehat{n}$ be a node and assume, WLOG, that $\widehat{n} = n$. Then*

$$\left(\sum_{\substack{b \in \{A,C,\\ G,T\}}} 2^{s(\widehat{bn})} a(\widehat{bn}) \right) - \ell(\widehat{n}) = \left(\sum_{\substack{b \in \{A,C,\\ G,T\}}} 2^{s(\widehat{nb})} a(\widehat{nb}) \right) - r(\widehat{n}).$$

Proof. We argue the invariant for a single read. The overall invariant is established by summing over all the reads.

Let W be a read. Since W contains no duplex edges, it corresponds to a left-right alternating walk in the de Bruijn Graph. Thus, every time W visits $\widehat{n}$, it must arrive via a right edge of $\widehat{n}$ and depart via a left edge of $\widehat{n}$ (or vice versa), unless W starts or ends at $\widehat{n}$. We call an arrival at or departure from $\widehat{n}$ a ***threshold***. Each occurrence of $\widehat{n}$ in W corresponds to two thresholds (except with the possible exception of occurrences

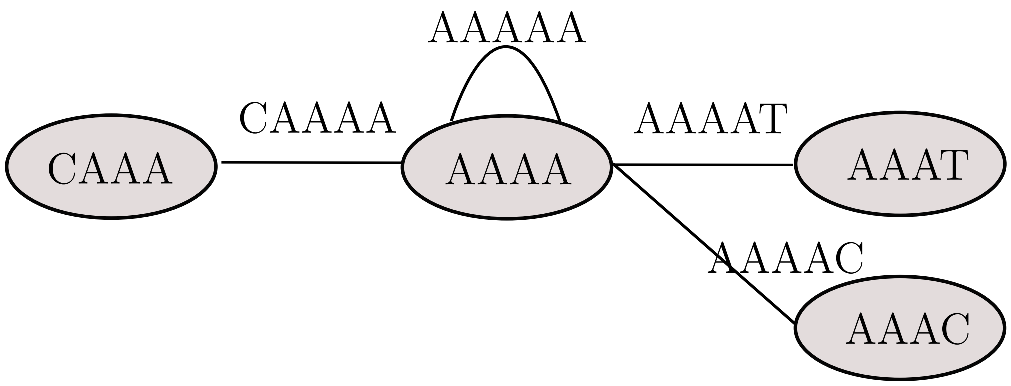


Figure 5.2: **Types of edges in a de Bruijn Graph.** The nodes are 4-mers and edges are 5-mers. For node AAAA, CAAAA is a left edge and AAAAT, AAAAC are right edges. We introduced another edge AAAAA in order to show a duplex edge. All the nodes/edges are canonicalized and the graph is bi-directional.

of $\widehat{n}$ at the beginning or end of W). We call an arrival at or departure from $\widehat{n}$ via a left edge of $\widehat{n}$ a left threshold of $\widehat{n}$, and define right thresholds similarly.

Thus, ignoring occurrences of $\widehat{n}$ at the beginning or end of W, the number of left thresholds of $\widehat{n}$ must equal the number of right thresholds of $\widehat{n}$. Let $L_W(\widehat{n})$ and $R_W(\widehat{n})$ be the number of left and right thresholds, respectively, of $\widehat{n}$ in W. Let $\ell_W(\widehat{n})$ be the number of left thresholds of occurrences of $\widehat{n}$ at the beginning or end of W, and define $r_W(\widehat{n})$ analogously for right thresholds of $\widehat{n}$. Thus we have the equality

$$L_W(\widehat{n}) - \ell_W(\widehat{n}) = R_W(\widehat{n}) - r_W(\widehat{n}).$$

Each occurrence of an left edge $\widehat{e}$ of $\widehat{n}$ in W corresponds to a single threshold of $\widehat{n}$, unless $\widehat{e}$ is a loop connecting $\widehat{n}$ to itself, in which case each occurrence of $\widehat{e}$ corresponds to two thresholds. Note that, since by assumption $\widehat{e}$ is not a duplex edge, if it is a loop, it corresponds to two left thresholds or two right thresholds of $\widehat{n}$ (i.e. it does not correspond to one left and one right threshold of $\widehat{n}$). Let $a_W(\widehat{e})$ be the number of occurrences of $\widehat{e}$ in W. Then

$$L_W(\widehat{n}) = \sum_{\substack{b\in\{A,C,\\ G,T\}}} 2^{s(\widehat{bn})} a_W(\widehat{bn})$$

and

$$R_W(\widehat{n}) = \sum_{\substack{b\in\{A,C,\\ G,T\}}} 2^{s(\widehat{nb})} a_W(\widehat{nb}).$$

Thus

$$\sum_{\substack{b\in\{A,C,\\ G,T\}}} 2^{s(\widehat{bn})} a_W(\widehat{bn}) - \ell_W(\widehat{n}) = \sum_{\substack{b\in\{A,C,\\ G,T\}}} 2^{s(\widehat{nb})} a_W(\widehat{nb}) - r_W(\widehat{n}).$$

The final result is obtained by summing over all reads $W \in \mathcal{R}$. □

5.3.2 deBGR: A compact de Bruijn graph representation

We now describe our compact weighted de Bruijn Graph representation. Given a set R of reads, we build counting quotient filters representing the functions a, ℓ, and r. For ℓ and r, we use exact CQFs, so these tables will have no errors. Since ℓ and r have roughly one entry for each read, these tables will be relatively small (see Section 5.3.6). For a, we build a space-efficient approximate CQF, which we call a_{CQF}. Since we build exact representations of ℓ and r, we will use ℓ and r to refer to both the actual functions and our tables representing these functions. The CQF guarantees that, for every edge $\widehat{e}$, $a_{\mathrm{CQF}}[h(\widehat{e})] \geq a(\widehat{e})$. We then compute a table c of *corrections* to a_{CQF} (we explain how to compute c below). After computing c, a query for the abundance of an edge $\widehat{e}$ returns $g(\widehat{e})$, where g is defined to be $g(\widehat{e}) = a_{\mathrm{CQF}}[h(\widehat{e})] - c[\widehat{e}]$. Thus, since c is initially 0, we initially have that $g(\widehat{e}) \geq a(\widehat{e})$ for all $\widehat{e}$.

Definition 4. *We say that g* ***satisfies the weighted de Bruijn Graph invariant for $\widehat{n}$*** *if*

$$\left(\sum_{\substack{b\in\{A,C,\\G,T\}}} 2^{s(\widehat{bn})} g(\widehat{bn}) \right) - \ell[\widehat{n}] = \left(\sum_{\substack{b\in\{A,C,\\G,T\}}} 2^{s(\widehat{nb})} g(\widehat{nb}) \right) - r[\widehat{n}].$$

5.3.3 Local error-correction rules

We first describe our local algorithm for correcting errors in g. This algorithm can be used to answer arbitrary k-mer membership queries by correcting errors on the fly. Thus this algorithm can be used to perform queries on a dynamic weighted de Bruijn Graph.

The process for computing c maintains the invariant that $g(\widehat{e}) \geq a(\widehat{e})$ for every edge $\widehat{e}$ in the weighted de Bruijn Graph, while using the following three rules to correct errors in g.

1. If we know that g is correct for all but one edge of some node $\widehat{n}$, then we can use the weighted de Bruijn Graph invariant to solve for the true abundance of the remaining edge.

2. Since $g(\widehat{e}) \geq a(\widehat{e})$ for all $\widehat{e}$, if (1) g satisfies the weighted de Bruijn Graph invariant for some node $\widehat{n}$ and, (2) we know that g is correct for all of $\widehat{n}$'s left edges, then we can conclude that g is correct for all of $\widehat{n}$'s right edges, as well (and vice versa for "left" and "right").

3. If $\sum_{b\in\{A,C,G,T\}} 2^{s(\widehat{bn})} g(\widehat{bn}) = \ell[\widehat{n}]$ and $r[\widehat{n}] = 0$, then the abundance of all of $\widehat{n}$'s right edges must be 0 (and vice versa for "right" and "left").

Given an initial set C of edges for which we know g is correct, we can use the above rules to correct errors in g and to expand C. But how can we get the initial set of edges C that is required to bootstrap the process? Our algorithm uses two approaches.

First, whenever $g(\widehat{e}) = 0$, it must be correct. This is because g can never be smaller than a. Thus, the above rules always apply to leaves of the approximate weighted de Bruijn Graph and, more generally, to any nodes that have only left or only right edges.

Leaves and nodes with only right or only left edges are not sufficient to bootstrap the error correction process, however, because weighted de Bruijn Graphs can contain cycles in which each node has both left and right edges that are part of the cycle. Starting only from leaves and one-sided nodes, the above rules are not sufficient to infer that g is correct on any edge in such a cycle, because each node in the cycle will always have a left and right edge for which g is not known to be correct.

We can overcome this problem by exploiting the random nature of errors in the CQF to infer that g is correct, with very high probability, on almost all edges of the approximate weighted de Bruijn Graph, including many edges that are part of cycles.

Theorem 4. *Suppose that errors in g are independent and random with probability ε. Suppose $\widehat{n}$ is not part of a left-right-alternating cycle of size less than d. Suppose also that g satisfies the weighted de Bruijn Graph invariant at every node within a left-right-alternating distance of $\lceil d/2 \rceil$ from $\widehat{n}$. Then the probability that g is incorrect for any edge attached to $\widehat{n}$ is less than $(4\varepsilon)^d$.*

Proof. Since g is never smaller than a, if g is incorrect for a left edge of some node $\widehat{n}$ and g satisfies the weighted de Bruijn Graph invariant at $\widehat{n}$, then g must be incorrect for at least one right edge of $\widehat{n}$. (And symmetrically for right/left). Thus, if g is incorrect for some edge attached to $\widehat{n}$, then since g satisfies the weighted de Bruijn Graph invariant for all nodes within a radius $d/2$ of $\widehat{n}$, it must be the case that g is incorrect for every edge along some left-right-alternating path of length at least d edges. Since $\widehat{n}$ is not part of a cycle of length less than d, all the edges in this path must be distinct. Since errors in g are independent and have probability ε, the probability of this occurring along any single path is at most ε^d.

Since each node of the weighted de Bruijn Graph has at most 4 left and 4 right edges, the total number of left-right-alternating paths of length d centered on node $\widehat{n}$ is at most 4^d. Hence, by a union bound, the probability that such a path exists is at most $(4\varepsilon)^d$. □

We can use this theorem to infer, with high probability, that g is correct for many edges in the graph. We can choose larger or smaller d to control the probability that we incorrectly infer that g is correct on an edge. By choosing $d \geq \log n / \log(1/4\varepsilon)$, where n is the number of edges in the approximate weighted de Bruijn Graph, we can make the expected number of such edges less than 1.

On the other hand, we expect many nodes from actual weighted de Bruijn Graphs to meet the criteria of Theorem 4. The vast majority of nodes in a weighted de Bruijn Graph are simple, i.e., they have exactly 1 left edge and 1 right edge. Therefore, for most nodes, there are only $O(d)$ nodes within left-right-alternating distance $\lceil d/2 \rceil$. Thus, for most nodes, the probability that they fail to meet the criteria is $O(d\varepsilon)$. When $d = \log n / \log(1/4\varepsilon)$, this becomes $O(\varepsilon \log n / \log(1/4\varepsilon))$. This means that for most values of n and ε that arise in practice, the vast majority of nodes will meet the criteria of Theorem 4. For example, when $n \leq 2^{40}$ and $\varepsilon \leq 2^{-8}$, the fraction of nodes expected to fail the criteria of Theorem 4 is less than 3%.

The above analysis suggests that large cycles (i.e. cycled of length at least $\log n / \log(1/4\varepsilon)$) in the weighted de Bruijn Graph will have at least a few nodes that meet the criteria of Theorem 4, so the correction process can bootstrap from those nodes to correct any other incorrect edges in the cycle. Small cycles (i.e. of size less than $\log n / \log(1/4\varepsilon)$) , however, still pose a problem, since Theorem 4 explicitly forbids nodes in small cycles.

We can handle small cycles as follows. Any kmer that is part of a cycle of length $q < k$ must be periodic with periodicity q, i.e., it must be a substring of a string of the form $x^{\lceil k/q \rceil}$, where x is a string of length q. Thus, small cycles are quite rare. We can detect k-mers that might be involved in a cycle of length less than d during the process of building a_{CQF} and record their abundance in a separate, exact CQF. Since periodic k-mers are rare, this exact CQF will not consume much space. Later, during the correction phase, we can add all the edges corresponding to these k-mers to the set C.

As mentioned before, the weighted de Bruijn Graph invariant only applies to nodes without duplex edges. Our weighted de Bruijn Graph representation handles duplex edges as follows. Suppose a read corresponds to a walk visiting the sequence of nodes $\widehat{n_1}, \widehat{n_2}, \ldots, \widehat{n_q}$. We treat every time the read visits a duplex node as the end of one read and the beginning of a new read. By breaking up reads whenever they visit a duplex node, we ensure that whenever a walk arrives at a node via a left or right edge, it either ends or departs via a left or right edge. Thus we can use the weighted de Bruijn Graph invariant to correct errors in a_{CQF} as described above.

5.3.4 Global, CQF-specific error-correction rules

So far, our error-correction algorithm uses only local information about discprencies in the weighted de Bruijn Graph invariant to correct errors. It also uses the CQF in a black-box fashion—the same algorithm could work with, for example, a counting Bloom filter approximation of a.

We now describe an extension to our error-correction algorithm that, in our experiments, enables it to correct all errors in the approximate weighted de Bruijn Graph. This extension exploits the fact that the CQF represents a multiset S by storing, exactly, the multiset $h(S)$, where h is a hash function. It also performs a global analysis of the graph in order to detect more errors than can be detected by the local algorithm. Thus this algorithm is appropriate for applications that need a static, navigational weighted de Bruijn Graph representation.

Note that applications can mix-and-match the two error correction algorithms. Both the local and global algorithms can be run repeatedly, in any order, and even intermixed with intervening modifications to the weighted de Bruijn Graph (e.g., after inserting additional k-mers).

For a read set R, let K be the set of distinct k-mers occurring in R. During the k-mer counting phase, every time we see a k-mer e, we increment the counter associated with $h(\widehat{e})$ in a_{CQF}. Thus, after counting is completed, for any edge $\widehat{e}$,

$$a_{\text{CQF}}[h(\widehat{e})] = \sum_{\widehat{x} \in K \cap h^{-1}(\widehat{e})} a(\widehat{x})$$

where $h^{-1}(\widehat{e}) = \{\widehat{x} \mid h(\widehat{x}) = h(\widehat{e})\}$.

The above equation enables us to use knowledge about the abundance of an edge $\widehat{e}$ to infer information about the abundance of other edges that collide with $\widehat{e}$ under the hash function h. For example, if we know that we have inferred the true abundance for all but one edge in some set $h^{-1}(\widehat{e})$, then we can use this equation to infer the abundance of the one remaining edge.

Our algorithm implements this idea as follows. Recall that, with high probability, whenever an edge $\widehat{x} \in C$, then $g(\widehat{x}) = a(\widehat{x})$. Thus we can rewrite the above equation as:

$$a_{\text{CQF}}[h(\widehat{e})] - \sum_{\widehat{x} \in C \cap h^{-1}(\widehat{e})} g(\widehat{x}) = \sum_{\widehat{x} \in \overline{C} \cap h^{-1}(\widehat{e})} a(\widehat{x}),$$

where $\overline{C} = K \setminus C$. For convenience, write $z(\widehat{e}) = a_{\text{CQF}}[h(\widehat{e})] - \sum_{\widehat{x} \in C \cap h^{-1}(\widehat{e})} g(\widehat{x})$. Note that z factors through h, i.e., if $h(\widehat{x}) = h(\widehat{y})$, then $z(\widehat{x}) = z(\widehat{y})$, and hence $z(\widehat{x})$ is the same for all $\widehat{x}$ in some set $h^{-1}(\widehat{e})$.

The above equation implies two invariants that our algorithm can use to infer additional information about edge abundances:

- For all $\widehat{e}$, $a(\widehat{e}) \leq z(\widehat{e})$. This is because, by definition, $a(\widehat{e}) \geq 0$. Thus, if the algorithm ever finds an edge $\widehat{e}$ such that $g(\widehat{e}) > z(\widehat{e})$, then it can update $c[\widehat{e}]$ so that $g(\widehat{e}) = z(\widehat{e})$.

- If, for some $\widehat{e}$, $\sum_{\widehat{x} \in \overline{C} \cap h^{-1}(\widehat{e})} g(\widehat{x}) = z(\widehat{e})$, then $g(\widehat{x}) = a(\widehat{x})$ for all $x \in \overline{C} \cap h^{-1}(\widehat{e})$. This is because $0 \leq a(\widehat{x}) \leq g(\widehat{x})$ for all $\widehat{x}$. Thus, in this case, the algorithm can add all the elements of $x \in \overline{C} \cap h^{-1}(\widehat{e})$ to C.

5.3.5 An algorithm for computing abundance corrections

Algorithm 4 shows our algorithm for computing c based on these observations. To save RAM, the algorithm computes the complement M of C, since for typical error rates C would contain almost all the edges in the weighted de Bruijn Graph. The algorithm also implements an optimized version of the test in Theorem 4 as follows. Rather than iterating over every node $\widehat{n}$ of the graph and performing a breadth-first visit to all the nodes within distance d of $\widehat{n}$, the algorithm iterates over every node $\widehat{n}$ of the graph and, for each node that fails to satisfy the weighted de Bruijn Graph invariant or might be in a small cycle, marks all edges within distance d as not-known-to-be-correct. Since few nodes violate the weighted de Bruijn Graph invariant, this is much faster than performing a breadth-first visit starting from each node of the weighted de Bruijn Graph.

The algorithm also implements an optimized version of the test in Theorem 4 as follows. Rather than iterating over every node $\widehat{n}$ of the graph and performing a breadth-first visit to all the nodes within distance d of $\widehat{n}$, the algorithm iterates over every node $\widehat{n}$ of the graph and, for each node that fails to satisfy the weighted de Bruijn Graph invariant or might be in a small cycle, marks all edges within distance d as not-known-to-be-correct. Since few nodes violate the weighted de Bruijn Graph invariant, this is much faster than performing a breadth-first visit starting from each node of the weighted de Bruijn Graph.

The algorithm then initializes two maps used in the CQF-specific extension to the algorithm. For each $x \in h(M)$, the algorithm sets $I[x]$ to be the set of all edges $\widehat{e}$ in the approximate weighted de Bruijn Graph such that $h(\widehat{e}) = x$. This will enable the algorithm to quickly compute $z(\widehat{e})$, and apply the correction rules, for any edge that is not yet known to have a correct abundance. The algorithm initializes I using a single extra pass over the edges of the approximate weighted de Bruijn Graph. The algorithm then removes from M and I any edges that it discovers have no hash collisions – a_{CQF} must have the correct abundance for these edges. The algorithm then computes a table Z such that $Z[h(\widehat{x})] = z(\widehat{x})$ for any $\widehat{x} \in M$.

The algorithm then initializes a work queue of all nodes adjacent to any edge in M (these are the only nodes to which our weighted de Bruijn Graph invariant rules can apply). The rest of the algorithm pulls nodes off the work queue and attempts to apply our weighted de Bruijn Graph invariant rules to correct a_{CQF} and/or remove edges from M. Whenever the algorithm decides that it has determined an edge's true abundance w.h.p., it invokes DECLARE-CORRECT. This helper function updates c if necessary, removes the given edge from M, adds its nodes to the work-queue, and then applies the CQF-specific correction rules.

The worst-case running time of the algorithm is $O(n^{1+1/\log(1/4\varepsilon)})$ but, for real weighted de Bruijn Graphs, the algorithm runs in $O(n \log n / \log(1/4\varepsilon))$ time. The running time is dominated by initializing M, which requires traversing the graph and and finding any nodes within distance $\log n / \log(1/4\varepsilon$ of a weighted de Bruijn Graph invariant discrepancy. Since real weighted de Bruijn Graphs have nodes mostly of degree 2, there will usually be $O(d) = O(\log n / \log(1/4\varepsilon))$ such nodes, giving a total running time of $O(n \log n / \log(1/4\varepsilon))$.

The rest of the algorithm runs in linear time. This is because, after initializing Q, a node can be added to Q only if at least one of its edges is in M and, whenever a node gets added to Q, at least one of its edges must have been removed from M. Since nodes can have degree at most 8, each node can enter Q at most 8 times. Thus the total number of iterations of the while loop is at most linear in the size of the weighted de Bruijn Graph. Each iteration runs in constant time.

When used to perform a local correction as part of an abundance query, we use the same algorithm, but restrict it to examine the region of the weighted de Bruijn Graph within $O(d)$ hops of the edge being queried. In the worst case, this could require examining the entire graph, resulting in the same complexity as above. In the common case, however, the number of nodes within distance d of the queried edge is $O(d)$, so the running time of a local correction is $O(\log n / \log(1/4\varepsilon))$.

The space for deBGR can be analyzed as follows. To represent a multiset S with false positive rate ε, the CQF takes $O(|S| \log_2 1/\varepsilon + C(S))$, where $C(S)$ is the sum of the logs of the counts of the items in S. To represent S exactly, assuming that each element of S is a b-bit string, takes $O(|S| \log_2 b/|S| + C(S))$. So let K be the multiset of k-mers, and let $E \subseteq K$ be the multiset of k-mer instances in K that occur at the beginning or end of a read or visit a duplex node. Then the space required to represent a_{CQF} is $O(|K| \log 1/\varepsilon + C(K))$. The space required for ℓ and r is $O(|E| \log 4^k/|E| + C(E))$. Note that since $E \subseteq K$, $C(E) \leq C(K)$. The space required to represent c is $O(\varepsilon |K| \log 4^k/\varepsilon |K| + C(K))$. Thus the total space required for deBGR is

$$O(|K| \log \frac{1}{\varepsilon} + |E| \log \frac{4^k}{|E|} + \varepsilon |K| \log \frac{4^k}{\varepsilon |K|} + C(K)).$$

5.3.6 Implementation

We extended Squeakr to construct the exact CQFs ℓ and r as described above, in addition to the approximate CQF a_{CQF} that it already built. We then wrote a second tool to compute c from a_{CQF}, ℓ, and r. Our prototype handles duplex nodes and small cycles as described. Our current prototype uses a standard hash table to store M and standard set to store Q. Also, we use a standard hash table to store c. An exact CQF would be more space efficient, but c is small enough in our experiments that it doesn't matter.

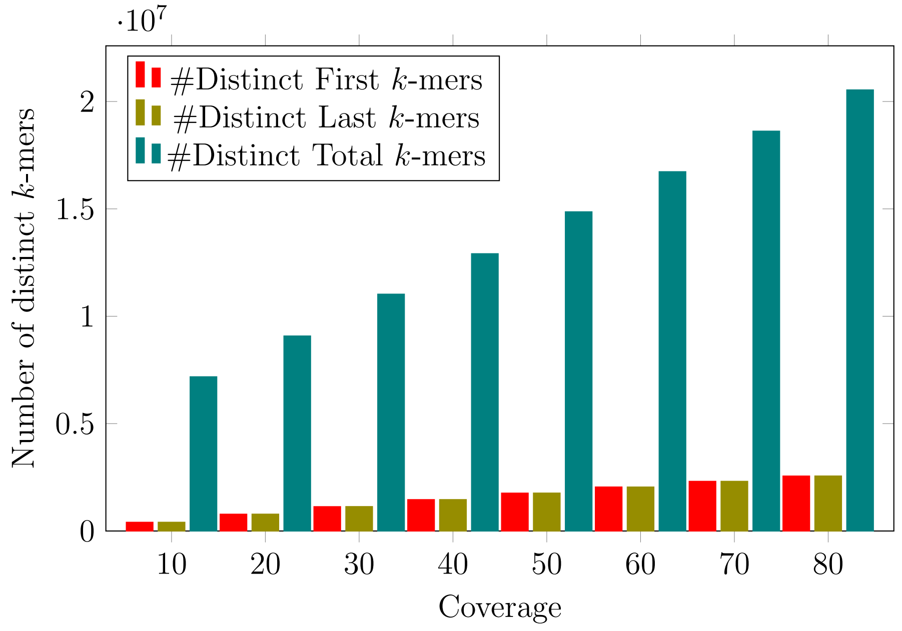


Figure 5.3: Total number of distinct k-mers in First QF, Last QF, and Main QF with increasing coverage of the same dataset. We generate dataset simulations using [89].

Size of the first and last tables. We explore, through simulation, how the sizes of the first and last tables ℓ and r grow with the coverage of the underlying data. Here, for simplicity, we focus on genomic (rather than transcriptomic) data, as coverage is a well-defined notion. We simulated reads generated from the *Escheria coli* (*E. coli*) (strain E1728) reference genome at varying levels of coverage, and recorded the number of total distinct k-mers, as well as the number of distinct k-mers in ℓ and r (Figure 5.3). Reads were simulated using the Art [89] read simulator, using the error profiles 125 bp, paired-end reads sequences on an Illumina HiSeq 2500. As expected, the number of distinct k-mers in all of the tables grows with the coverage (due to sequencing error), Yet, even at $80x$ coverage, the ℓ and r tables, together, contain fewer than 25% of total distinct k-mers. On the experimental data examined in Section 5.4, the ℓ and r tables, together, require between than 18% – 28% of the total space required by the deBGR structure.

5.4 Evaluation

In this section we evaluate deBGR, as described in Section 5.3.

We evaluate deBGR in terms of space and accuracy. The space is the size of the data structure(s) needed to represent the weighted de Bruijn Graph. The accuracy is the measure of how close the weighted de Bruijn Graph representation is to the actual weighted de Bruijn Graph. We also report the time taken by deBGR to construct the

Data set	File size	#Files	#k-mer instances	#Distinct k-mers
GSM984609	26	12	19662773 330	1146347598
GSM981256	22	12	16470774825	1118090824
GSM981244	43	4	37897872977	1404643983
SRR1284895	33	2	26235129875	2079889717

Table 5.1: Data sets used in our experiments. The file size is in GB. All the datasets are compressed with gzip compression.

weighted de Bruijn Graph representation, perform global abundance correction, and perform local abundance correction for an edge.

As described in Section 5.3.6, deBGR uses two exact counting quotient filters (ℓ and r) in addition to the approximate counting quotient filter that stores the number of occurrences for each k-mer. The error-correction algorithm then computes a table c of corrections. In our evaluation we report the total size of all these data structures, i.e. a_{CQF}, ℓ, r, and c.

We measure the accuracy of systems in terms of errors in the weighted de Bruijn Graph representation. There are two kind of errors in the weighted de Bruijn Graph, abundance errors and topological errors. An ***abundance error*** is an error when the weighted de Bruijn Graph representation returns an over-count for the query k-mer (deBGR never resulted in an undercount in any of our experiments). ***Topological errors*** are abundance error for edges whose true abundance is 0. Topological errors are also known as false-positives.

In both cases, we report the number of ***reachable*** errors. Let G be the true weighted de Bruijn Graph and G' our approximation. Since g is never smaller than a, the set of edges in G' is a superset of the set of edges in G. An error on edge $\widehat{e}$ of G' is ***reachable*** if there exists a path in G' from $\widehat{e}$ to an edge that is also in G. Note that reachable false positives are not the same as Chikhi, et al's notion of critical false positives [48]. Critical false positives are false positives that are false positive edges that share a node with a true positive edge. Reachable false positives, on the other hand, may be multiple hops away from a true edge of the weighted de Bruijn Graph.

We compare deBGR to Squeakr in both its approximate and exact configurations. Recall that the exact version of Squeakr stores k-mers in a CQF using a $2k$-bit invertible hash function, so that it has no false positives. We use the exact version of Squeakr as the reference weighted de Bruijn Graph for computing the number of reachable errors in Squeakr and deBGR.

We do not compare deBGR against other Bloom filter based de Bruijn Graph representations [48, 149] because Bloom filter based de Bruijn Graph representations do not have abundance information.

5.4.1 Experimental setup

All experiments use 28-mers. In all our experiments, the counting quotient filter was configured with a maximum allowable false-positive rate of 1/256.

All the experiments are performed in-memory. We use several datasets for our experiments, which are listed in Table 5.1. All the experiments were performed on an Intel(R) Xeon(R) CPU (E5-2699 v4 @ 2.20GHz with 44 cores and 56MB L3 cache) with 512GB RAM and a 4TB TOSHIBA MG03ACA4 ATA HDD.

5.4.2 Space vs accuracy trade-off

In Table 5.2 we show the space needed and the accuracy (in terms of navigational errors) offered in representing the weighted de Bruijn Graph by deBGR and the exact and approximate versions of Squeakr. For deBGR, Table 5.2 gives the final space usage (i.e., a_{CQF}, ℓ, r, and c).

deBGR offers 100% accuracy and takes 48%–52% less space than the exact version of Squeakr that also offers 100% accuracy. deBGR takes 18%–28% more space than the approximate version but the appropriate version has millions of navigational errors.

The space required by deBGR in Table 5.2 is the total space of all data structures (a_{CQF}, ℓ, r, and c). In Table 5.3 we report the maximum number of items stored in auxiliary data structures (see Algorithm 4) while performing abundance correction. This gives an upper bound on the amount of space needed by deBGR to perform abundance correction.

5.4.3 Performance

In Table 5.4 we report the time taken by deBGR to construct the weighted de Bruijn Graph representation and perform global abundance correction. The time information for construction and global abundance correction is averaged over two runs.

We also report the time taken to perform local abundance correction for an edge. The time for local abundance correction per edge is averaged over 1M local abundance corrections. After performing abundance correction, computing $g(\widehat{e}) = a_{\text{CQF}}[h(\widehat{e})] - c[\widehat{e}]$ takes 3.45 microseconds on average.

System	**Data set**	**Space** (bits/k-mer)	**Navigational Errors** Topological	Abundance
Squeakr	GSM984609	18.9	14263577	16655318
Squeakr (exact)		50.8	0	0
deBGR		26.5	0	0
Squeakr	GSM981256	19.4	13591254	15864754
Squeakr (exact)		52.1	0	0
deBGR		27.1	0	0
Squeakr	GSM981244	30.9	10462963	12257261
Squeakr (exact)		79.8	0	0
deBGR		37.0	0	0
Squeakr	SRR1284895	20.9	23272114	27200821
Squeakr (exact)		53.95	0	0
deBGR		25.38	0	0

Table 5.2: Space vs Accuracy trade-off in Squeakr and deBGR. Topological errors are false-positive k-mers. Abundance errors are k-mers with an over count.

Data set	**#Edges in M**	**#Edges in work queue (Q)**
GSM984609	30815799	76178634
GSM981256	29359913	72606572
GSM981244	22674515	56309858
SRR1284895	50320986	124558299

Table 5.3: The maximum number of items present in auxiliary data structures, edges (k-mers) in MBI and nodes (($k-1$)-mers) in work queue as described in the Algorithm 4, during abundance correction.

Algorithm 4 Global algorithm for computing corrections to a_{CQF}. (For clarity, we omit the special-handling code for k-mers that may be part of a small cycle.) For the local version of the algorithm, simply omit all the code dealing with I and Z.

```
function COMPUTE-CORRECTIONS($a_{\text{CQF}}$, $\ell$, $r$)
    Let $n$ be the number of nodes in the approximate de Bruijn Graph
    $d \leftarrow \log n / \log(1/4\varepsilon)$
    Initialize $c$ so that $c(\widehat{e}) = 0$ for all $\widehat{e}$.
    $M \leftarrow \emptyset$, $Q \leftarrow \emptyset$, $I \leftarrow \emptyset$, $Z \leftarrow \emptyset$
    for each node $\widehat{n} \in V$ do
        if $\widehat{n}$ might be part of a cycle of size $\leq d$
          or $g$ does not satisfy the dBG invariant at $\widehat{n}$ then
           for each edge $\widehat{e}$ within left-right-alternating distance $d$ of $\widehat{n}$ do
               insert $\widehat{e}$ into $M$
    for each edge $x \in h(M)$ do
        $I[x] \leftarrow h^{-1}(x) \cap E$
    for each hash value $x$ in the domain of $I$ do
        if $|I[x]| = 1$ then
            $I[x] \leftarrow \perp$, $M \leftarrow M \setminus I[x]$
    for each hash value $x$ in the domain of $I$ do
        $Z[x] \leftarrow a_{\text{CQF}}[x]$
    for each edge $\widehat{e} \in M$ do
        insert both nodes of $\widehat{e}$ into $Q$
    while $Q \neq \emptyset$ do
        Extract a node $\widehat{n}$ from $Q$.
        if $g$ satisfies the dBG invariant at $\widehat{n}$ then
            for $s \in \{\text{left}, \text{right}\}$ do
                if none of the $s$ edges of $\widehat{n}$ is in $M$ then
                    for each edge $\widehat{e}$ on the other side of $\widehat{n}$ do
                        DECLARE-CORRECT($a_{\text{CQF}}, c, M, Q, I, Z, \widehat{e}, g(\widehat{e})$)
        else if $\sum_{\widehat{e} \in \widehat{n}\text{'s left edges}} g(\widehat{e}) - \ell[\widehat{n}] = r[\widehat{n}] = 0$ then
            for each right edge $\widehat{e}$ of $\widehat{n}$ do
                DECLARE-CORRECT($a_{\text{CQF}}, c, M, Q, I, Z, \widehat{e}, 0$)
        else if $\sum_{\widehat{e} \in \widehat{n}\text{'s right edges}} g(\widehat{e}) - \ell[\widehat{n}] = r[\widehat{n}] = 0$ then
            for each left edge $\widehat{e}$ of $\widehat{n}$ do
                DECLARE-CORRECT($a_{\text{CQF}}, c, M, Q, I, Z, \widehat{e}, 0$)
        else if $\widehat{n}$ has exactly one edge $\widehat{e}$ in $M$ then
            $t \leftarrow \sum_{b \in \{A,C,G,T\}} \left(2^{s(\widehat{bn})} g(\widehat{bn}) - 2^{s(\widehat{nb})} g(\widehat{nb})\right) - \ell[\widehat{n}] + r[\widehat{n}]$
            DECLARE-CORRECT($a_{\text{CQF}}, c, M, Q, I, Z, \widehat{e}, |t|$)
    return $c$
```

Algorithm 5 Subroutine for recording a newly-discovered edge abundance and applying the global, CQF-specific error-correction rules. The algorithm records that $\widehat{e}$ has abundance v. For the local version of the algorithm, simply omit all the code dealing with I and Z.

```
function DECLARE-CORRECT($a_{\mathrm{CQF}}$, $c$, $M$, $Q$, $I$, $Z$, $\widehat{e}$, $v$)
    if $\widehat{e} \in M$ then
        if $v \neq g(\widehat{e})$ then
            $c[\widehat{e}] \leftarrow a_{\mathrm{CQF}}[h(\widehat{e})] - v$
        remove $\widehat{e}$ from $M$
        add $\widehat{e}$'s nodes to $Q$

                                        ▷ The rest of this function is the global extension
        remove $\widehat{e}$ from $I[h(\widehat{e})]$
        $Z[h(\widehat{e})] \leftarrow Z[h(\widehat{e})] - g(\widehat{e})$
        for each edge $\widehat{x} \in I[h(\widehat{e})]$ do
            if $g(\widehat{x}) > Z[h(\widehat{x})]$ then
                $c[\widehat{x}] \leftarrow a_{\mathrm{CQF}}[h(\widehat{x})] - Z[h(\widehat{x})$
                add $\widehat{x}$'s nodes to $Q$
        $t \leftarrow \sum_{\widehat{x} \in I[h(\widehat{e})]} g(\widehat{x})$
        if $t = Z[h(\widehat{e})]$ then
            for each edge $\widehat{x} \in I[h(\widehat{e})]$ do
                remove $\widehat{x}$ from $M$
                add $\widehat{x}$'s nodes to $Q$
            $I[h(\widehat{e})] \leftarrow \perp$, $Z[h(\widehat{e})] \leftarrow \perp$
```

Data set	Construction	Global correction	Local correction
	(seconds)	(seconds)	(microseconds)
GSM984609	6605.65	14 857.68	12.93
GSM981256	5470.83	15 390.56	18.25
GSM981244	13 373.78	22 266.86	16.50
SRR1284895	8429.17	41 218.85	16.62

Table 5.4: Time to construct the weighted de Bruijn Graph, correct abundances globally in the weighted de Bruijn Graph, and perform local correction per edge in the weighted de Bruijn Graph (averaged over 1M local corrections).

Chapter 6: Large-scale sequence-search index

We now describe the third application of the counting quotient filter to computational biology. In this application, we use the counting quotient filter to build a large-scale search index for biological sequences on thousands of raw samples. This application uses the counting quotient filter to map k-length subsequences to values (i.e., the set of samples in which the subsequence appears). The values are usually variable-sized and highly skewed. Mantis efficiently represents the mappings by using the variable-sized counters in the CQF to instead store variable-sized values.

6.1 Sequence-search indexes

The ability to issue sequence-level searches over publicly available databases of assembled genomes and known proteins has played an instrumental role in many studies in the field of genomics, and has made BLAST [9] and its variants some of the most widely-used tools in all of science. Much subsequent work has focused on how to extend tools such as BLAST to be faster, more sensitive, or both [41, 61, 142, 159]. However, the strategies applied by such tools focus on the case where queries are issued over a database of reference sequences. Yet, the vast majority of publicly-available sequencing data (e.g., the data deposited in the SRA [99]) exists in the form of raw, unassembled sequencing reads. As such, this data has mostly been rendered impervious to sequence-level search, which substantially reduces the utility of such publicly available data.

There are a number of reasons that typical reference-database-based search techniques cannot easily be applied in the context of searching raw, unassembled sequences. One major reason is that most current techniques do not scale well as the amount of data grows to the size of the SRA (which today is ≈ 4 petabases of sequence information). A second reason is that searching unassembled sequences means that relatively long queries (e.g., genes) are unlikely to be present in their entirety as an approximate substring of the input.

Recently, new computational schemes have been proposed that hold the potential to allow searching raw sequence read archives while overcoming these challenges. Solomon and Kingsford introduced the sequence Bloom tree (SBT) data structure [155] and an associated algorithm that enables an efficient type of search over thousands of sequencing experiments. Specifically, they re-phrase the query in terms of k-mer set membership in a way that is robust to the fact that the target sequences have not been assembled. The resulting problem is coined as the *experiment discovery* problem, where the goal is to return all experiments that contain at least some user-defined fraction θ of the k-mers present in the query string. The space and query time of the SBT structure has been further improved by [156] and [160] by applying an all-some set decomposition over the original sets of the SBT structure. This seminal work introduced both a formulation of this problem, and the initial steps toward a solution.

Sequence Bloom trees build on prior work using Bloom filters [29]. A Bloom filter is a compact representation of a set S. Bloom filters support insertions and membership queries, and they save space by allowing a small false-positive probability. That is, a query for an element $x \notin S$ might return "present" with probability δ. Allowing false positives enables the Bloom filter to save space—a Bloom filter can represent a set of size n with a false-positive probability of δ using $O(n \log_2(1/\delta))$ bits. Bloom filters have an interesting property that the bitwise-or of two Bloom filters representing S_1 and S_2 yields a Bloom filter for $S_1 \cup S_2$. However, the false-positive rate of the union may increase substantially above δ.

In k-mer-counting tools, Bloom filters are used to filter out single-occurrence (and likely erroneous) k-mers from raw read data [112]. In a high-coverage genomic data set, any k-mer that occurs only once is almost certainly an error and can thus be ignored. However, such k-mers can constitute a large fraction of all the k-mers in the input—typically 30-50%—so allocating a counter and an entry in a hash table for these k-mers can waste a lot of space. Tools such as BFCounter [112] and Jellyfish [110] save space by inserting each k-mer into a Bloom filter the first time it is seen. For each k-mer in the input, the tool first checks whether the k-mer is in the Bloom filter. If not, then this is the first time this k-mer has been seen, so it is inserted into the filter. If the k-mer is already in the filter, then the counting tool stores the k-mer in a standard hash table, along with a count of the number of times this k-mer has been seen. In this application, a false positive in the Bloom filter simply means that the tool might count a few k-mers that occur only once. Using Bloom filters in this way can reduce space consumption by roughly 50%.

Sequence Bloom trees repurpose Bloom filters to index large sets of raw sequencing data probabilistically. In an SBT, each experiment is represented by a Bloom filter of all the k-mers that occur a sufficient number of times in that experiment. A k-mer counter can create such a Bloom filter by first counting all the k-mers in the experiment and then inserting every k-mer that occurs sufficiently often into a Bloom filter. The SBT then builds a binary tree by logically or-ing Bloom filters until it reaches a single root node. To find all the experiments that contain a k-mer x, start from the root and test whether x is in the Bloom filter of each of the root's children. Whenever a Bloom filter indicates that an element might be present in a subtree, recurse into that subtree.

SBTs support queries for entire transcripts as follows. First compute the set Q of k-mers that occur in the transcript. Then, when descending down the tree, only descend into subtrees whose root Bloom filter contains at least a θ fraction of the k-mers in Q. Typical values for θ proposed by [155] are in the range $0.7 - 0.9$. That is, any experiment that contains 70-90% of the k-mers in Q has a reasonable probability of containing the transcript (or a closely-related variant).

The SSBT and the AllSomeSBT have a similar structure to the SBT, but they use more efficient encodings. The AllSomeSBT has a shorter construction time and query time than the SBT. The SSBT has a slower construction time than the SBT, answers queries faster than the SBT, and uses less space than either the SBT or the AllSomeSBT.

Both structures use a similar high-level approach for saving space and thus making queries fast. Namely, instead of retaining a single Bloom filter at each internal node, the structures maintain two Bloom filters. One Bloom filter stores k-mers that appear in every experiment in the descendant leaves. These k-mers do not need to be stored in any descendants of the node, thus reducing the space consumption by reducing redundancy. If a queried k-mer is found in this Bloom filter, then it is known to be present in all descendant experiments. If the required fraction of k-mers for a search (i.e. θ) ever appear in such a filter, then search of this subtree can terminate early as all descendant leaf nodes satisfy the query requirements. The other Bloom filter stores the rest of the k-mers, those that appear in some, but not all, of the descendants. AllSomeSBT saves additional space by clustering similar leaves into subtrees, so that more k-mers can be stored higher up in the tree and with less duplication.

Due to limitations of the Bloom filter, all of these SBT-like structures are forced to balance between the false-positive rate at the root and the size of the filters representing the individual experiments. Because Bloom filters cannot be resized, they must be created with enough space to hold the maximum number of elements that might be inserted. Furthermore, two Bloom filters can only be logically or-ed if they have the same size (and use the same underlying hash functions). Thus, the Bloom filters at the root of the tree must be large enough to represent every k-mer in every experiment indexed in the entire tree, while still maintaining a good false positive rate. On the other hand, the Bloom filters at the leaves of the tree represent only a relatively small amount of data and, since there are many leaves, should be as small as possible.

Further complicating the selection of Bloom filter size is that individual dataset sizes vary by orders of magnitude, but all datasets must be summarized using Bloom filters of the same size. For these reasons, most of the Bloom filters in the SBT are, of necessity, sub-optimally tuned and inefficient in their use of space. SBTs partially mitigate this issue by compressing their Bloom filters using an off-the-shelf compressor (Bloom filters that are too large are sparse bit vectors and compress well.) Nonetheless, SBTs are typically constructed with a Bloom filter size that is too small for the largest experiments, and for the root of the tree. As a result, they have low precision: typically only about 57-67% of the results returned by a query are actually valid (i.e., contain at least a fraction θ of the query k-mers).

We present a new k-mer-indexing approach, which we call Mantis, that overcomes these obstacles. Mantis has several advantages over prior work:

- Mantis is *exact*. A query for a set Q of k-mers and threshold θ returns exactly those data sets containing at least fraction θ of the k-mers in Q. There are no false positives or false negatives. In contrast, we show that SBT-based systems exhibit only 57-67% precision, meaning that many of the results returned for a given query are, in fact, false positives.

- Mantis supports much faster queries than existing SBT-based systems. In our experiments, queries in Mantis ran up to 100$\times$ faster than in SSBT.

- Mantis supports much faster index construction. For example, we were able to build the Mantis index on 2652 data sets in 16 hours. SSBT reported 97 hours to construct an index on the same collection of data sets.

- Mantis uses less storage than SBT-based systems. For example, the Mantis index for the 2652 experiments used in the SSBT evaluation is 20% smaller than the compressed SSBT index for the same data.

- Mantis returns, for each experiment containing at least 1 k-mer from the query, the number of query k-mers present in this experiment. Thus, the full spectrum of relevant experiments can be analyzed. While these results can be post-processed to filter out those not satisfying a θ-query, we believe the Mantis output is more useful, since one can analyze which experiments were close to achieving the θ threshold, and can examine if there is a natural "cutoff" at which to filter experiments.

Mantis builds on Squeakr [132], a k-mer counter based on the counting quotient filter (CQF) [131]. The CQF is a Bloom-filter alternative that offers several advantages over the Bloom filter. First, the CQF supports counting, i.e., queries to the CQF return not only "present" or "absent," but also an estimate on the number of times the queried item has been inserted. Analogous to the Bloom filter's false-positive rate, there is a tunable probability δ that the CQF may return a count that is higher than the true count for a queried element. CQFs can also be resized, and CQFs of different sizes can be merged together efficiently. Finally, CQFs can be used in an "exact" mode where they act as a compact exact hash table, i.e., we can make $\delta = 0$. CQFs are also faster and more space efficient than Bloom filters for all practical configurations (i.e., any false-positive rate $< 1/64$).

Prior work has shown how CQFs can be used to improve performance and simplify the design of k-mer-counting tools [132] and de Bruijn Graph representations [130]. For example, Squeakr is essentially a thin wrapper around a CQF—it just parses fastq files, extracts the k-mers, and inserts them into a CQF. Other k-mer counters use multiple data structures (e.g., Bloom filters plus hash tables) and often contain sophisticated domain-specific tricks (e.g., minimizers) to get good performance. Despite its simplicity, Squeakr uses less than half the memory of other k-mer counters, offers competitive counting performance, and supports queries for counts up to $10\times$ faster than other k-mer counters. Performance is similar in exact mode, in which case, the space is comparable to other k-mer counters.

In a similar spirit, Mantis uses the CQF to create a simple space- and time-efficient index for searching for sequences in large collections of experiments. Mantis is based on colored de Bruijn graphs. The "color" associated with each k-mer in a colored de Bruijn graph is the set of experiments in which that k-mer occurs. We use an exact CQF to store a table mapping each k-mer to a color ID, and another table mapping color IDs to the actual set of experiments containing that k-mer. Mantis uses an off-the-shelf compressor [140] to store the bit vectors representing each set of experiments.

Mantis takes as input the collection of CQFs representing each data set, and outputs the search index. Construction is efficient because it can use sequential I/O to read the input and write the output CQFs. Similarly, queries for the color of a single k-mer are efficient since they require only two table lookups.

We believe that, since Mantis is also a colored de Bruijn Graph representation, it may be useful for more than just querying for the existence of sequences in large collections of data sets. Mantis supports the same fast de Bruijn Graph traversals as Squeakr, using the same traversal algorithm as described in the Squeakr [132] and deBGR papers [130]. Hence Mantis may be useful for topological analyses such as computing the length of the query covered in each experiment (rather than just the fraction of k-mers present). Mantis can be used for de Bruijn Graph traversal by querying the possible neighboring k-mers of a given k-mer and extending the path in the de Bruijn Graph [16, 116, 135]. It can also naturally support operations such as bubble calling [91], and hence could allow a natural, assembly-free way to analyze variation *among* experiments.

6.2 Methods

6.2.1 Method details

Mantis builds on Squeakr [132], a k-mer counter that uses a counting quotient filter [131] as its primary data structure.

Mantis

Mantis takes as input a collection of experiments and produces a data structure that can be queried with a given k-mer to determine the set of experiments containing that k-mer. Mantis supports these queries by building a colored de Bruijn graph. In the colored de Bruijn graph, each k-mer has an associated color, which is the set of experiments containing that k-mer.

The Mantis index is essentially a colored de Bruijn graph, represented using two dynamic data structures: a counting quotient filter and a color-class table. The counting quotient filter is used to map each k-mer to a color ID, and then that ID can be looked up in the color-class table to find the actual color (the list of experiments containing that k-mer). This approach of using color classes was also used in Bloom filter Trie [88] and Rainbowfish [6].

Mantis re-purposes the CQF's counters to store color IDs instead. In other words, to map a k-mer k to a color ID c, we insert c copies of k into the CQF. The CQF supports not only insertions and deletions, but directly setting the counter associated with a given k-mer, so this can be done efficiently (i.e., we do not need to insert k repeatedly to get the counter up to c, we can just directly set it to c).

Each color class is represented as a bit vector in the color-class table, with one bit for each input experiment (Figure 6.1). All the bit vectors are concatenated and compressed using RRR compression [140] as implemented in the sdsl library [78].

Construction. To construct the Mantis index, we first count k-mers for each input experiment using Squeakr. Because Squeakr can either perform exact or approximate k-mer counting, the user has the freedom to trade off space and count accuracy. The output of Squeakr is a counting quotient filter containing k-mers and their counts. Mantis builds its index by performing a k-way merge of the CQFs, creating a single counting quotient filter and a color-class table. The merge process follows a standard k-way merge approach [131] with a small tweak.

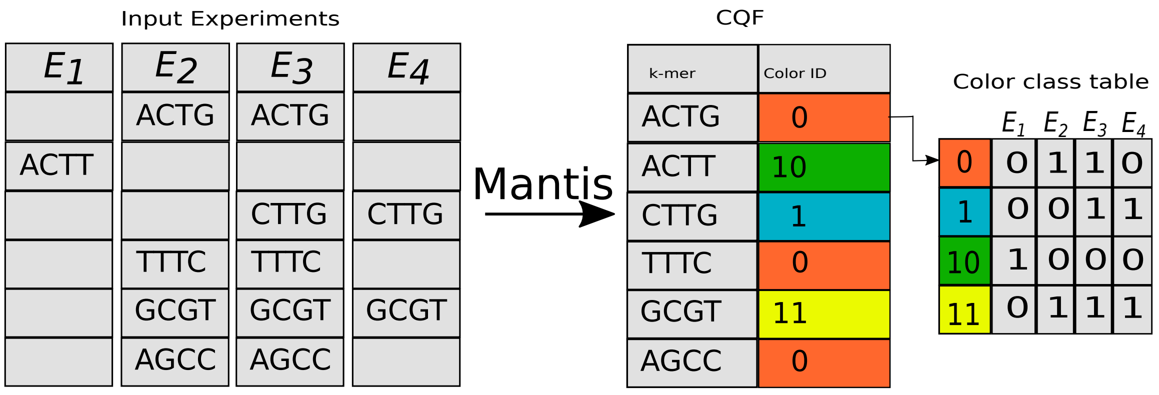


Figure 6.1: The Mantis indexing data structures. The CQF contains mappings from k-mers to color-class IDs. The color-class table contains mappings from color-class IDs to bit vectors. Each bit vector is N bits, where N is the number of experiments from which k-mers are extracted. The CQF is constructed by merging N input CQFs each corresponding to an experiment. A query first looks up the k-mer(s) in the CQF and then retrieves the corresponding color-class bit vectors from the color-class table.

During a standard CQF merge, the merging algorithm accumulates the total number of occurrences of each key by adding together its counters from all the input CQFs. In Mantis, rather than accumulating the total count of a k-mer, we accumulate the set of all input experiments that contain that k-mer.

As with SBT-based indexes, once we have computed the set of experiments containing a given k-mer, we filter out experiments that contain only a few instances of a given k-mer. This filtered set is the color class of the k-mer. The merge algorithm then looks up whether it has already seen this color class. If so, it inserts the k-mer into the output CQF with the previously assigned color-class ID for this color class. Otherwise, it assigns the next available color-class ID for the new color, adds the k-mer's color class to the set of observed color classes, and inserts the k-mer into the output CQF with the new color-class ID. Figure 6.1 gives an overview of the Mantis build process and indexing data structure.

We detect whether we have seen a color class previously by hashing each color-class bit vector to a 128-bit hash. We then store these hashes in an in-memory hash table. Each time we compute the color class of a new k-mer, we check whether we have seen this color class before by looking up its hash in this table. This approach may have false positives if two distinct color classes collide under the hash function, but that never happened in our experiments. Furthermore, assuming that the hash function behaves like a random function and that there are less than 4 billion distinct color classes, the odds of having a collision are less than 2^{-64}.

Each color-class bit vector is stored in a separate buffer. The size of the buffer may vary based on the amount of RAM available for the construction and the final size of the index representation. In our experiments, we used a buffer of size $\approx$ 6GB. Once the buffer gets full we compress the color-class bit vector using RRR compression [140] and write it to disk.

Sampling color classes based on abundance. Mantis stores the color-class ID corresponding to each k-mer as its count in the counting quotient filter. In order to save space in the output CQF, we assign the smaller IDs to the most abundant color classes. We could achieve this using a two-pass algorithm. In the first pass, we count the number of k-mers that belong to each color class. In the second pass, we sort the color classes based on their abundances and assign the IDs in increasing order [6]. However, two passes through the data is expensive.

In Mantis, we improve upon the two-pass algorithm as follows. We perform a sampling phase in which we analyze the color-class distribution of a subset of k-mers.[1] We sort the color classes based on their abundances and assign IDs giving the smallest ID to the most abundant color class, an observation that helps in optimizing the size of other data structures like the Bloom Filter Trie [88] and Rainbowfish [6]. We then use this color-class table as the starting point for the rest of the k-mers. Given the uniform-randomness property of the hash function used in Squeakr to hash k-mers there is a very high chance that we will see the most abundant colors class in the first few million k-mers and will assign the smallest ID to it. Figure 6.2 shows that the smallest ID is assigned to the color class with the most number of k-mers. Also, the number of k-mers belonging to a color class reduces monotonically as the color class ID increases.

Queries. A query consists of a transcript T and a threshold θ. Let Q be the set of the k-mers in T. The query algorithm should return the set of experiments that contain at least a fraction θ of the k-mers in Q.

Given a query transcript T and a threshold θ, Mantis first extracts the set Q of all k-mers from T. It then queries the CQF for each k-mer $x \in Q$ to obtain x's color-class ID c_x. Mantis then looks up c_x in the color-class table to get the bit vector v_x representing the set of experiments that contain x. Finally, Mantis performs vector addition (treating the vectors as vectors of integers) to obtain a single vector v, where $v[i]$ is the number of k-mers from Q that occur in the ith experiment. It then outputs each experiment i such that $v[i] \geq \theta|Q|$.

In order to avoid decoding the same color-class bit vector multiple times, we maintain a map from each color-class ID to the number of times a k-mer with that ID has

[1]In Mantis, we look at first $\approx 67M$ k-mers in the sampling phase.

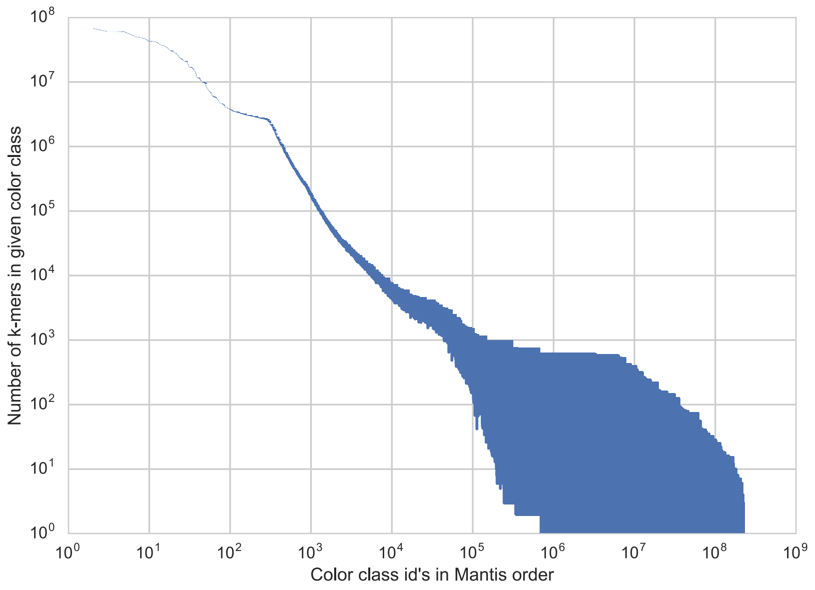


Figure 6.2: The distribution of the number of k-mers in a color class and ID assigned to the color class. The color class with the most number of k-mers gets the smallest ID. And the color class with least number of k-mers gets the largest ID. This distribution of IDs helps save space in Mantis.

appeared in the query. This is done so that, regardless of how many times a k-mer belonging to a given color-class appears in the query, we need to decode each color-class at most one time. Subsequently, these IDs are looked up in the color-class table to obtain the bit vector corresponding to the experiments in which that k-mer is present. We maintain a hash table that records, for each experiment, the number of query k-mers present in this experiment, and the bit vector associated with each color-class is used to update this hash table until the color-classes associated with all query k-mers have been processed. This second phase of lookup is particularly efficient, as it scales in the number of *distinct* color-classes that label k-mers from the query. For example, if all n query k-mers belonged to a single color-class, we would decode this color-class' bit vector only once, and report each experiment present in this color class to contain n of the query k-mers.

Mantis supports both approximate and exact indexes. When used in approximate mode, queries to Mantis may return false positives, i.e. experiments that do not meet the threshold θ. Theorem 5 shows that, with high probability, any false positives returned by Mantis are close to the threshold.[2] Thus, false positives in Mantis are not simply random experiment—they are experiments that contain a significant fraction of the queried k-mers.

[2]An event $E_{c,n}$ occurs ***with high probability*** if it occurs with probability at least $1 - 1/n^c$.

Theorem 5. *A query for q k-mers with threshold θ returns only experiments containing at least $\theta q - O(\delta q + \log n)$ queried k-mers w.h.p.*

Proof. This follows from Chernoff bounds and the fact that the number of queried k-mers that are false positives in an experiment is upper bounded by a binomial random variable with mean δq.

$\square$

Dynamic updates. We now describe a method that a central authority could use to maintain a large database of searchable experiments. Researchers could upload new experiments and submit queries over previously uploaded experiments. In order to make this service efficient, we need an way to incorporate new experiments without rebuilding the entire index each time a new experiment is uploaded.

Our solution follows the design of cascade filters [131] and LSM-trees [124]. In this approach, the index consists of a logarithmic number of levels, each of which is a single index. The maximum size of each level is a constant factor (typically 4—10) larger than the previous level's maximum size. New experiments are added to the index at the smallest level (i.e. level 0). Since level 0 is small, it is feasible to add new experiments by simply rebuilding it. When the index at level i exceeds its maximum size, we run a merge algorithm to merge level i into level $i + 1$, recursively merging if this causes level $i + 1$ to exceed its maximum size.

We now describe how to rebuild a Mantis index (i.e. level 0) to include new experiments. First compute CQFs for the new experiments using Squeakr. Then update the bit vectors in the Mantis index to include a new entry (initially set to 0) for each new experiment. Then run a merge algorithm on the old index and the CQFs of the new experiments. The merge will take as input the old Mantis CQF, the old Mantis mapping from color IDs to bit vectors, and the new CQFs, and will produce as output a new CQF and new mapping from color IDs to bit vectors.

For each k-mer during the merge, compute the new set of experiments containing that k-mer by adding any new experiments containing that k-mer to the old bit vector for that k-mer. Then assign that set a color ID as described above and insert the k-mer and its color ID into the output CQF.

Merging two levels together is similar. During the merge, simply union the sets of experiments containing a given k-mer. This process is I/O efficient since it requires only sequentially reading each of the input indexes.

During a query, we have to check each level for the queried k-mers. However, since queries are fast and there are only a logarithmic number of levels, performance should still be good.

6.2.2 Quantification and statistical analysis

We used precision as the main accuracy metric to compare Mantis with SSBT. For a query string with a set Q of $|Q| = m$ distinct k-mers, an experiment-query pair (E, Q) is defined as a true positive (TP) at threshold θ if E contains at least θm fraction of the k-mers occuring in Q. By the same definition, false positive (FP) hits are those experiment-query pairs reported as found that do not contain at least θm fraction of the k-mers occuring in Q. Finally, we define a false negative (FN) as an experiment-query pair where E does contain at least θm fraction of the k-mers occuring in Q, but the experiment is not reported by the system as containing the query. Having all of these, we define precision as $\frac{TP}{TP+FP}$ and sensitivity as $\frac{TP}{TP+FN}$. None of the tools, Mantis, SBT, or SSBT, should have any FNs, so the sensitivity defined as $\frac{TP}{TP+FN}$ is 1 for all these tools. In addition, the precision is also 1 for Mantis, since it is exact and doesn't report any FPs.

6.3 Evaluation

In this section we compare the performance and accuracy of Mantis against SSBT [156].

6.3.1 Evaluation metrics

Our evaluation aims to compare Mantis to the state of the art, SSBT [156], on the following performance metrics:

- **Construction time.** How long does it take to build the index?
- **Index size.** How large is the index, in terms of storage space?
- **Query performance.** How long does it take to execute queries?
- **Quality of results.** How many false positives are included in query results?

6.3.2 Experimental procedure

For all experiments in this chapter, unless otherwise noted, we consider the k-mer size to be 20 to match the parameters adopted by [155].

To facilitate comparison with SSBT, we use the same set of 2652 experiments used in the evaluation done by [157], and as listed on their website [97]. These experiments consist of RNA-seq short-read sequencing runs of human blood, brain, and breast tissue. We obtained these files directly from European Nucleotide Archive (ENA) [122] since they provide direct access to gzipped FASTQ files, which are more expedient for our

Min size	Max size	Cutoff
0	$\leq$ 300MB	1
$>$ 300MB	$\leq$ 500MB	3
$>$ 500MB	$\leq$ 1GB	10
$>$ 1GB	$\leq$ 3GB	20
$>$ 3GB	∞	50

Table 6.1: Minimum number of times a k-mer must appear in an experiment in order to be counted as abundantly represented in that experiment (taken from the SBT paper).

purposes than the SRA format files. We discarded 66 files that contained only extremely short reads (i.e., less than 20 bases).[3] Thus the actual number of files used in our evaluation was 2586.

We first used Squeakr (exact) to construct counting quotient filters for each experiment. We used 40-bit hashes and an invertible hash function in Squeakr (exact) to represent k-mers exactly. Before running Squeakr (exact), we needed to select the size of the counting quotient filter for each experiment. We used the following rule of thumb to estimate the counting quotient filter size needed by each experiment: singleton k-mers take up 1 slot in the counting quotient filter, doubletons take up 2 slots, and almost all other k-mers take up 3 slots. We implemented this rule of thumb as follows. We used ntCard [115] to estimate the number of distinct k-mers F_0 and the number of k-mers of count 1 and 2 (f_1 and f_2, respectively) in each experiment. We then estimated the number of slots needed in the counting quotient filter as $s = f_1 + 2f_2 + 3(F_0 - f_1 - f_2)$. The number of slots in the counting quotient filter must be a power of 2, so let s' be the smallest power of 2 larger than or equal to s. In order to be robust to errors in our estimate, if s was more than $0.8s'$, then we constructed the counting quotient filter with $2s'$ slots. Otherwise, we constructed the counting quotient filter with s' slots.

We then used Squeakr (exact) to construct a counting quotient filter of the counts of the k-mers in each experiment. The total size of all the counting quotient filters was 2.7TB.

We then computed cutoffs for each experiment according to the rules defined in the SBT paper [156], shown in Table 6.1. The SBT paper specifies cutoffs based on the size of the experiment, measured in bytes, but does not specify how the size of an experiment is calculated (e.g. compressed or uncompressed). We use the size of the compressed file downloaded from ENA.

We then invoked Mantis to build the search index using these cutoffs. Based on a few trial runs, we estimated that the number of slots needed in the final counting quotient filter would be 2^{34}, which turned out to be correct.

[3]We believe that the SBT authors just treated these as files containing zero 20-mers.

Query datasets. For measuring query performance, we randomly selected sets of 10, 100, and 1000 transcripts from the Gencode annotation of the human transcriptome [76]. Also, before using these transcripts for queries we replaced any occurrence of "N" in the transcripts with a pseudo-randomly chosen valid nucleotide. We then performed queries for these three sets of transcripts in both Mantis and SSBT. For SSBT, we used $\theta = 0.7$, 0.8, and 0.9. For Mantis, θ makes no difference to the run-time, since it is only used to filter the list of experiments at the very end of the query algorithm. Thus performance was indistinguishable, so we only report one number for Mantis's query performance.

For SSBT, we used the tree provided to us by the SSBT authors via personal communication.

We also compared the quality of results from Mantis and SSBT. Mantis is an exact representation of k-mers and therefore all the experiments reported by Mantis should also be present in the results reported by SSBT. However, SSBT results may contain false-positive experiments. Therefore, we can use Mantis to empirically calculate the precision of SSBT. Precision is defined as $TP/(TP + FP)$, where TP and FP are the number of true and false positives, respectively, in the query result.

6.3.3 Experimental setup

All experiments were performed on an Intel(R) Xeon(R) CPU (E5-2699 v4 @2.20GHz with 44 cores and 56MB L3 cache) with 512GB RAM and a 4TB TOSHIBA MG03ACA4 ATA HDD running ubuntu 16.10 (Linux kernel 4.8.0-59-generic), and were carried out using a single thread. The data input to the construction process (i.e., fastq files and the Squeakr representations) was stored on 4-disk mirrors (8 disks total), each is a Seagate 7200rpm 8TB disk (ST8000VN0022). They were formatted using ZFS and exported to via NFS over a 10Gb link.

All the input CQF files were mmaped. However, we also used asynchronous reads (*aio_read*) to perform prefetch of data from the remote storage. Since the input CQFs were accessed in sequential order, prefetching can help the kernel cache the data that will be accessed in the immediate future. We adopted the following prefetching strategy: each input CQF had a separate buffer wherein the prefetched data was read. The sizes of the buffers were proportional to the number of slots in the input CQF. We used 4096B buffers for the smallest CQFs and 8MB for the largest CQF.

The time reported for construction and query benchmarks is the total time taken measured as the wall-clock time using `/usr/bin/time`.

We compare Mantis and SSBT on their in-memory query performance. For Mantis, we warmed the cache by running the query benchmarks twice; we report the numbers from the second run. We followed the SSBT author's procedure for measuring SSBT's

	Mantis	SSBT
Build time	16 Hr 35 Min	97 Hr
Representation size	32 GB	39.7 GB

Table 6.2: Time and space measurement for Mantis and SSBT. Total time taken by Mantis and SSBT to construct the representation. Total space needed to store the representation by Mantis and SSBT. Numbers for SSBT were taken from the SSBT paper [156].

in-RAM performance [157], as explained to us in personal communication. We copied all the nodes in the tree to a ramfs (i.e. an in-RAM file system). We then ran SSBT on the files stored in the ramfs. (We also tried running SSBT twice, as with Mantis, and performance was identical to that from following the SSBT authors' procedures.)

SSBT query benchmarks were run with thresholds 0.7, 0.8 and 0.9, and the `max-filter` parameter was set to 11000. By setting `max-filter` to 11000, we ensured that SSBT never had to evict a filter from its cache.

6.3.4 Experiment results

In this section we present our benchmark results comparing Mantis and SSBT to answer questions posed above.

Build time and index space. Table 6.2 shows that Mantis builds its index 6× faster than SSBT. Also, the space needed by Mantis to represent the final index is 20% smaller than SSBT.

Most of the time is spent in merging the hashes from the input counting quotient filters and creating color-class bit vectors. The merging process is fast because there is no random disk IO. We read through the input counting quotient filters sequentially and also insert hashes in the final counting quotient filter sequentially. The number of k-mers in the final counting quotient filter was $\approx$ 3.69 billion. Writing the resulting CQF to disk and compressing the color-class bit vectors took only a small fraction of the total time.

The maximum RAM required by Mantis to construct the index, as given by `/usr/bin/time` (maximum resident set size), was 40GB. The maximum RAM usage broke down as follows. The output CQF consumed 17GBs. The buffer of uncompressed bit vectors used 6GBs, and the compressor uses another 6GB output buffer. The table of hashes of previously seen bit vectors consumed about 5GBs. The prefetch buffers used about 1GB. There were also some data structural overheads, since we used several data structures from the C++ standard library.

	Mantis	SSBT (0.7)	SSBT (0.8)	SSBT (0.9)
10 Transcripts	25 Sec	3 Min 8 Sec	2 Min 25 Sec	2 Min 7 Sec
100 Transcripts	28 Sec	14 Min 55 Sec	10 Min 56 Sec	7 Min 57 Sec
1000 Transcripts	1 Min 3 Sec	2 Hr 22 Min	1 Hr 54 Min	1 Hr 20 Min

Table 6.3: Time taken by Mantis and SSBT to perform queries on three sets of transcripts. The set sizes are 10, 100, and 1000 transcripts. For SSBT we used three different threshold values 0.7, 0.8, and 0.9. All the experiments were performed by either making sure that the index structure is cached in RAM or is read from ramfs.

Query performance. Table 6.3 shows the query performance of Mantis and SSBT on three different query datasets. Even for $\theta = 0.9$ (the best case of SSBT), Mantis is $5\times$, $16\times$, and $75\times$ faster than SSBT for in-memory queries. For $\theta = 0.7$, Mantis is up to 137 times faster than SSBT.

The query time for SSBT reduces with increasing θ values because with higher θ queries will terminate early and have to perform fewer accesses down the tree. In Table 6.3, for Mantis we only have one column because Mantis reports experiments for all θ values.

In Mantis, only two memory accesses are required per k-mer—one in the counting quotient filter and, if the k-mer is present, then the ID is looked up in the color-class table. SSBT has a fast case for queries that occur in every node in a subtree or in no node of a subtree, so it tends to terminate quickly for queries that occur almost everywhere or almost nowhere. However, for random transcript queries, it may have to traverse multiple root-to-leaf paths, incurring multiple memory accesses. This can cause multiple cache misses, resulting in slow queries.

Quality of results. Table 6.4 compares the results returned by Mantis and SSBT on the queries described above. All the comparisons were performed for $\theta = 0.8$. Since Mantis is exact, we use the results from Mantis to calculate the precision of SSBT results. The precision of SSBT results varied from 0.577 to 0.679.

Mantis is an exact index and SSBT is an approximate index with one-sided error (i.e., only false positives). Therefore the experiments reported by SSBT should be a super-set of the experiments reported by Mantis. However, when we compared the results from Mantis and SSBT there were a few experiments reported by Mantis that were not reported by SSBT. These could be because of a bug in Mantis or SSBT.

In order to determine whether there was a bug in Mantis, we randomly picked a subset of experiments reported only by Mantis and used KMC2 to validate Mantis' results. The results reported by KMC2 were exactly the same as the results reported by Mantis. This means that there were some experiments that actually had at least 80% of the k-mers from the query but were not reported by SSBT.

	Both	Only-Mantis	Only-SSBT	Precision
10 Transcripts	2018	19	1476	0.577
100 Transcripts	22466	146	10588	0.679
1000 Transcripts	160188	1409	95606	0.626

Table 6.4: Comparison of query benchmark results for Mantis and SSBT. Both means the number of those experiments that are reported by both Mantis and SSBT. Only-Mantis and Only-SSBT means the number of experiments reported by only Mantis and only SSBT. All three query benchmarks are taken from Table 6.3 for $\theta = 0.8$.

We contacted the SSBT authors about this anomaly, and they found that some of the data sets were corrupted during download from SRA. This resulted in the SSBT-tree having corrupted data for those particualar data sets, which was revealed by comparison with results from Mantis. We believe only a handful of data sets were corrupted, so that these issues do not materially impact the results reported in the SSBT paper.

Chapter 7: Timely reporting of heavy-hitters using external memory

In this chapter, we describe the application of the counting quotient filter to streaming. We show how to modify the cascade filter (a write-optimized quotient filter [21]) to solve the online event-detection problem using SSDs. We modify the cascade filter to design two new data structures, the time-stretch filter and popcorn filter and analyse their I/O performance. We further describe how we implement these two data structures, deamortize them to multi-threaded insertions, and perform empirical analysis to validate our theoretical results and evaluate the insertion performance.

7.1 Online event detection problem

Given a stream $S = (s_1, s_2, \ldots, s_N)$, a ***$\phi$-heavy hitter*** is an element s that occurs at least ϕN times in S. We say that there is a ***ϕ-event at time step t*** if s_t occurs exactly $\lceil \phi N \rceil$ times in $(s_1, s_2, \ldots, s_t)$. Thus, there is a single ϕ-event for each ϕ-heavy hitter, which occurs when its count reaches the ***reporting threshold*** $T = \lceil \phi N \rceil$.

We define the ***online event-detection problem*** (OEDP) to be: given stream $S = (s_1, s_2, \ldots, s_N)$ and reporting threshold $T = \lceil \phi N \rceil$, where each s_i arrives one at a time, for each $i \in [1, N]$, decide if there is a ϕ-event at time i before seeing s_{i+1}. A solution to the online event-detection problem must report

(a) all events[1] (no FALSE NEGATIVES)

(b) with no errors and no duplicates (no FALSE POSITIVES)

(c) as soon as an element crosses the threshold (ONLINE).

Furthermore, an online event detector must scale to

(d) small reporting thresholds T and large N, i.e., very small ϕ (SCALABLE).

[1]We drop the ϕ when it is understood.

Algorithms to solve the OEDP are critical components of monitoring and defense systems for cybersecurity [103, 141, 166] and physical systems, such as water or power distribution [27, 96, 105]. In such a monitoring system, changes of state are translated into the stream elements. Each detected/reported event triggers an intervention by an analyst. They use more specialized tools to gauge the actual threat level. Newer systems are even beginning to take defensive actions, such as blocking a remote host, automatically based on detected events [49, 81, 111, 127]. When used in an automated system, accuracy (i.e., low false-positive and false-negative rates) and timeliness of event detection are essential.

As an example of the demands placed on event detection systems, the US Department of Defense (DoD) and Sandia National Laboratories developed the Firehose streaming benchmark suite [1, 10] for the online event-detection problem. In the FireHose benchmark, the reporting threshold is preset to the representative value of $T = 24$, independent of the length of the stream.

Online event-detection requires large space. The online event-detection problem is at least as hard as the error-free one-pass heavy-hitters problem, which requires a large memory of $\Omega(N)$ words[2] in the worst case [28, 55]. In fact, even if we allow the detector to return $O(1 + \beta t)$ false positives in a stream with t true positives, for any constant β, a bound $\Omega(N \log N)$ bits can be obtained by an analogous communication-complexity reduction from the probabilistic indexing problem [101, 145]. Since there is no space savings to be had from allowing a bounded false-positive rate, this chapter focuses on exact solutions to the OEDP.

Event detection and the streaming model. There is a vast literature on relaxations of the OEDP in the streaming model [7, 25, 28, 31, 34, 35, 46, 57, 58, 64, 66, 102, 109]. Relaxations are necessary because, in the streaming model [12, 173], the available memory is small—usually just polylog(N)—while, as noted above, the OEDP requires $\Omega(N)$ space in the absence of errors, even without the ONLINE constraint [28].

If we relax conditions (a)-(d), the stringent space requirements diminish. For example, the classic Misra-Gries algorithm [55, 114] finds ϕ-heavy hitters using about $1/\phi$ words of space. But this algorithm makes two passes over the data, so it is not ONLINE. Bose et al. [31] show that a single-pass Misra-Gries algorithm provides estimates on the key count. Bhattacharyya et. al. use Misra-Gries as a subroutine to solve the heavy-hitter problem with upper and lower reporting thresholds, where the difference between the thresholds is inversely related to the space used [28]. Thus, prior work has investigated solutions with FALSE POSITIVES, FALSE NEGATIVES, or a mix.

[2]A word has $\Omega(\log N + \log |\mathcal{U}|)$ bits, where $\mathcal{U}$ is the universe of values to be stored.

Finally, existing streaming algorithms assume $\phi > 1/\text{polylog}(N)$ since they keep all candidates in memory. Even if we allow ε fuzziness in the reporting threshold, as in the (ε, ϕ)-heavy hitters problem[3], there is a lower bound of $(1/\varepsilon)\log(1/\phi) + (1/\phi)\log|\mathcal{U}| + \log\log N$ bits [28], which is large when ε is small.

In short, even though there are efficient and elegant algorithms that match the lower bounds of what is possible in the streaming model, there is a provably insurmountable gap between the requirements of the original problem and what is possible using limited storage.

Online event-detection in external memory. The goal of this chapter is to make the storage costs of the OEDP more palatable by presenting algorithms that make it possible to shift most of the storage from expensive RAM to lower-cost external storage, such as SSDs or hard drives. In the external-memory model, RAM has size M, storage has unbounded size, and any I/O access to external memory transfers blocks of size B. Typically, blocks are large, i.e., $\log N < B$ [4, 73, 74].

At first, it may appear trivial to detect heavy hitters using external memory: we can store the entire stream, so what is there to solve? And this would be true in an offline setting. We could find all events by logging the stream to disk and then sorting it.

The technical challenge to online event-detection in external memory is that searches are slow. A straw-man solution is to maintain an external-memory dictionary to keep track of the count of every item, and to query the dictionary after each stream item arrives. But this approach is bottlenecked on dictionary searches. In a comparison-based dictionary, queries take $\Omega(\log_B N)$ I/Os, and there are many data structures that match this bound [14, 18, 38, 52]. This yields an I/O complexity of $O(N \log_B N)$. Even if we use external-memory hashing, queries still take $\Omega(1)$ I/Os, which still gives a complexity of $\Omega(N)$ I/Os [54, 90]. Both these solutions are bottlenecked on the latency of storage, which is far too slow for stream processing.

Note that data ingestion is *not* the bottleneck in external memory. Optimal external-memory dictionaries (including write-optimized dictionaries such as B^{ε}-trees [20, 38], COLAs [19], xDicts [37], buffered repository trees [42], write-optimized skip lists [23], log structured merge trees [124], and optimal external-memory hash tables [54, 90]) can perform inserts and deletes extremely quickly. The fastest can index using $O\left(\frac{\log N/M}{B}\right)$ I/Os per stream element, which is far less than 1 I/O per item. In practice, this means that even a system with just a single disk can ingest hundreds of thousands of items per second. For example, at SuperComputing 2017, a single computer was easily able

[3]Given a stream of size N and $1/N \leq \varepsilon < \phi \leq 1$, report every item that is a ϕ-heavy hitter and no item that is not a $(\phi - \varepsilon)$-heavy hitter. It is optional whether we report elements with counts between $(\phi - \varepsilon)N$ and ϕN.

to maintain a B^{ε}-tree [38] index of all connections on a 600 gigabit/sec network [17]. The system could also efficiently answer offline queries. What the system could not do, however, was detect events online. The results of this chapter show that we can get online (or nearly online) event detection for essentially the same cost as simply inserting the data into a B^{ε}-tree or other optimal external-memory dictionary.

Results

Our main result is that, for $\phi = \Omega(1/M)$ we can solve the OEDP in external memory substantially cheaper than performing a query for each element:

Result 1. *Given $\phi > 1/M + \Omega(1/N)$ and a stream S of size N, the online event-detection problem can be solved at an amortized cost of $O\left(\left(\frac{1}{B} + \frac{M}{(\phi M - 1)N}\right) \log N/M\right)$ I/Os per stream item.*

To put this in context, suppose that $\phi \gg 1/M$ and $\phi > B/N$. Then, we get complexity of $O\left(\frac{1}{B} \log\left(\frac{N}{M}\right)\right)$, which is only a logarithmic factor larger than the naïve scanning lower bound. In this case, we eliminate the query bottleneck and match the data ingestion rate of B^{ε}-trees.

Our algorithm builds on the classic Misra-Gries algorithm, and thus supports many of its generalizations, such as fuzzy thresholds. Thus our algorithm can be used to solve the (ε, ϕ)-heavy hitters problem. See Section 7.4 for details.

We then show that, by allowing a bounded amount of reporting delay, we can extend this result to arbitrarily small ϕ. Intuitively, we allow the reporting delay for an event s_t to be proportional to the time it took for the element s_t to go from 1 to ϕN occurrences. More formally, for a ϕ-event s_t, define the ***flow time*** of s_t to be $F_t = t - t_1$, where t_1 is the time step of s_t's first occurrence. We say that an event-detection algorithm has ***time stretch*** $\boldsymbol{1+\alpha}$ if it reports each event s_t at or before time $t + \alpha F_t = t_1 + (1+\alpha)F_t$.

Result 2. *Given $\alpha > 0$ and a stream S of size N, the OEDP can be solved with time stretch $1+\alpha$ at an amortized cost of $O\left(\frac{\alpha+1}{\alpha}\frac{\log N/M}{B}\right)$ I/Os per stream item.*

For constant α, this is asymptotically as fast as simply ingesting and indexing the data [19, 38, 42]. As above, this generalizes to fuzzy thresholds, and hence yields an almost-online solution to the (ε, ϕ)-heavy hitters problem for arbitrarily small ε and ϕ. See Section 7.5.

Finally, we design a data structure for the OEDP problem that supports a smaller threshold ϕ and achieves a better I/O complexity when the count of elements in the input stream are drawn from a power-law distribution with exponent $\theta > 2 +$

$1/(\log_2(N/M))$,[4] which are extremely common in practical data [3, 25, 36, 50, 121]. See Section 7.6 for details.

Result 3. *Given an input stream S of size N where the count of elements is drawn from a power-law distribution with exponent $\theta > 1$, and $\phi > \gamma/N + \Omega(1/N)$, where $\gamma = 2\left(\frac{N}{M}\right)^{\frac{1}{\theta-1}}$, the* OEDP *on S can be solved with an amortized I/O complexity $O\Big(\Big(\frac{1}{B} + \frac{1}{(\phi N - \gamma)^{\theta-1}}\Big)\log\frac{N}{M}\Big)$ per stream item.*

Summary. Our results show that, by enlisting the power of external memory, we can solve event detection problems at a level of precision that is not possible in the streaming model, and with little or no sacrifice in terms of the timeliness of reports.

We believe these results are exciting because they show that, even though streaming algorithms, such as Misra-Gries, were originally developed for a space-constrained setting, they nonetheless are useful in external memory, where storage is plentiful but I/Os are expensive. On the other hand, by using external memory to solve problems that have traditionally been analyzed in the streaming setting, we enable solutions that can scale beyond the provable limits of fast RAM.

7.2 The Misra-Gries algorithm and heavy hitters

This section reviews the Misra-Gries heavy-hitters algorithm [114]. There are two main strategies that have been used for finding heavy hitters: deterministic counter-based approaches [31,64,95,109,113,114] and randomized sketch-based ones [46,56]. The first is based on the classic Misra and Gries (MG) algorithm [114], which generalizes the Boyer-Moore majority finding algorithm [33].

The Misra-Gries frequency estimator. The Misra-Gries algorithm estimates the frequency of elements in a stream. Given an error bound ε and a stream S of N elements from a universe $\mathcal{U}$, the MG algorithm uses a single pass over S to construct a table $\mathcal{C}$ with at most $\lceil 1/\varepsilon \rceil$ entries. Each table entry is an element $s \in \mathcal{U}$ with a count, denoted $\mathcal{C}[s]$. For each $s \in \mathcal{U}$ not in table $\mathcal{C}$, we define $\mathcal{C}[s] = 0$. Let f_s be the number of occurrences of element s in stream S. The MG algorithm guarantees that $\mathcal{C}[s] \le f_s < \mathcal{C}[s] + \varepsilon N$ for all $s \in \mathcal{U}$.

MG initializes $\mathcal{C}$ to an empty table and then processes the items in the stream one after another. For each s_i in S,

- If $s_i \in \mathcal{C}$, increment counter $\mathcal{C}[s_i]$.
- If $s_i \notin \mathcal{C}$ and $|\mathcal{C}| < \lceil 1/\varepsilon \rceil$, insert s_i into $\mathcal{C}$ and set $\mathcal{C}[s_i] \leftarrow 1$.

[4]If the element-counts follow a power-law distribution with exponent θ, then the probability that an element appears c times in the stream is $Zc^{-\theta}$, where Z is a normalization constant.

- If $s_i \notin \mathcal{C}$ and $|\mathcal{C}| = \lceil 1/\varepsilon \rceil$, then for each $x \in \mathcal{C}$ decrement $\mathcal{C}[x]$ and delete its entry if $\mathcal{C}[x]$ becomes 0.

We now argue that $\mathcal{C}[s] \leq f_s < \mathcal{C}[s] + \varepsilon N$. We have $\mathcal{C}[s] \leq f_s$ because $\mathcal{C}[s]$ is incremented only for an occurrence of s in the stream. MG underestimates counts only through the decrements in the third condition above. This step decrements $\lceil 1/\varepsilon \rceil + 1$ counts at once: the element s_i that caused the decrement, since it is never added to the table, and each element in the table. There can be at most $\lfloor N/\lceil 1/\varepsilon + 1 \rceil \rfloor < \varepsilon N$ executions of this decrement step in the algorithm. Thus, $f_s < \mathcal{C}[s] + \varepsilon N$.

Heavy hitters. The core MG algorithm can be used to solve the ***(ε, ϕ)-heavy hitters problem***, which requires us to report all elements s with $f_s \geq \phi N$ and not to report any element s with $f_s \leq (\phi - \varepsilon)N$. Elements that occur between $(\phi - \varepsilon)N$ and ϕN times in S are neither required nor forbidden in the reported set.

To solve the problem, run MG on the stream with error parameter ε. Then iterate over the set $\mathcal{C}$ and report any element s with $\mathcal{C}[s] > (\phi - \varepsilon)N$. Correctness follows since (1) if $f_s \leq (\phi - \varepsilon)N$, then, since $\mathcal{C}[s] \leq f_s \leq (\phi - \varepsilon)N$, s will not be reported, and (2) if $f_s \geq \phi N$, then $\mathcal{C}[s] > f_s - \varepsilon N \geq \phi N - \varepsilon N$, so s is reported.

Space usage. For a frequency estimation error of ε, Misra-Gries uses $O(\lceil 1/\varepsilon \rceil)$ words of storage, assuming each stream element and each count occupy $O(1)$ words. Bhattacharyya et al. [28] showed that, by using hashing, sampling, and allowing a small probability of error, Misra-Gries can be extended to solve the (ε, ϕ)-Heavy Hitters problem using $1/\phi$ slots that store counts and an additional $(1/\varepsilon)\log(1/\phi) + \log\log N$ bits, which they show is optimal.

For the exact ϕ-hitters problem, that is, for $\varepsilon = 1/N$, the space requirement of the MG algorithm is large—N slots. Even the optimal algorithm of Bhattacharyya uses $\Omega(N)$ bits of storage in this case, regardless of ϕ.

7.3 External-memory Misra-Gries and online event detection

In this section, we design an efficient external-memory version of the core Misra-Gries frequency estimator. This immediately gives an efficient external-memory algorithm for the (ε, ϕ)-heavy hitters problem. We then extend our external-memory Misra-Gries algorithm to support I/O-efficient immediate event reporting, e.g., for online event detection.

When $\varepsilon = o(1/M)$, then simply running the standard Misra-Gries algorithm can result in a cache miss for every stream element, incurring an amortized cost of $\Omega(1)$

I/Os per element. Our construction reduces this to $O(\frac{\log 1/(\varepsilon M)}{B})$, which is $o(1)$ when $B = \omega\big(\log\big(\frac{1}{\varepsilon M}\big)\big)$.

7.3.1 External-memory Misra-Gries

Our external-memory Misra-Gries data structure is a sequence of Misra-Gries tables, $\mathcal{C}_0, \ldots, \mathcal{C}_{L-1}$, where $L = 1 + \lceil \log_r(1/\varepsilon M) \rceil$ and r is a parameter we set later. The size of the table $\mathcal{C}_i$ at level i is $r^i M$, so the size of the last level is at least $1/\varepsilon$.

Each level acts as a Misra-Gries data structure. Level 0 receives the input stream. Level $i > 0$ receives its input from level $i-1$, the level above. Whenever the standard Misra-Gries algorithm running on the table $\mathcal{C}_i$ at level i would decrement a key count, the new data structure decrements that key's count by one on level i and sends one instance of that key to the level below $(i+1)$.

The algorithm processes the input stream by inserting each item in the stream into $\mathcal{C}_0$. To insert an item x into level i, do the following:

- If $x \in \mathcal{C}_i$, then increment $\mathcal{C}_i[x]$.
- If $x \notin \mathcal{C}_i$, and $|\mathcal{C}_i| \leq r^i M - 1$, then $\mathcal{C}_i[x] \leftarrow 1$.
- If $x \notin \mathcal{C}_i$ and $|\mathcal{C}_i| = r^i M$, then, for each x' in $\mathcal{C}_i$, decrement $\mathcal{C}_i[x']$; remove it from $\mathcal{C}_i$ if $\mathcal{C}_i[x']$ becomes 0. If $i < L-1$, recursively insert x' into $\mathcal{C}_{i+1}$.

Correctness. We first show that this algorithm meets the original requirements of the Misra-Gries frequency estimation algorithm. In fact, every prefix of levels $\mathcal{C}_0, \ldots, \mathcal{C}_j$ is a Misra-Gries frequency estimator, with the accuracy of the frequency estimates increasing with j:

Lemma 2. *Let $\widehat{\mathcal{C}}_j[x] = \sum_{i=0}^{j} \mathcal{C}_i[x]$ (where $\mathcal{C}_i[x] = 0$ if $x \notin \mathcal{C}_i$). Then $\widehat{\mathcal{C}}_j[x] \leq f_x < \widehat{\mathcal{C}}_j[x] + \frac{N}{r^j M}$. Furthermore, $\widehat{\mathcal{C}}_{L-1}[x] \leq f_x < \widehat{\mathcal{C}}_{L-1}[x] + \varepsilon N$.*

Proof. Decrementing the count for an element x in level $i < j$ and inserting it on the next level does not change $\widehat{\mathcal{C}}_j[x]$. This means that $\widehat{\mathcal{C}}_j[x]$ changes only when we insert an item x from the input stream into $\mathcal{C}_0$ or when we decrement the count of an element in level j. Thus, as in the original Misra-Gries algorithm, $\mathcal{C}[x]$ is only incremented when x occurs in the input stream, and is decremented only when the counts for $r^j M$ other elements are also decremented. As with the original Misra-Gries algorithm, this is sufficient to establish the frequency estimation bounds.

The second claim follows from the first, and the fact that $r^{L-1} M \geq 1/\varepsilon$. □

Heavy hitters. Since our external-memory Misra-Gries data structure matches the original Misra-Gries error bounds, it can be used to solve the (ε, ϕ)-heavy hitters problem when the regular Misra-Gries algorithm requires more than M space. First, insert each element of the stream into the data structure. Then, iterate over the sets $\mathcal{C}_i$ and report any element x with counter $\widehat{\mathcal{C}}_{L-1}[x] > (\phi - \varepsilon)N$.

I/O complexity. We now analyze the I/O complexity of our external-memory Misra-Gries algorithm. For concreteness, we assume each level is implemented as a B-tree, although the same basic algorithm works with sorted arrays (including with fractional cascading from one level to the next, similar to cache-oblivious lookahead arrays [19]) or hash tables with linear probing and a consistent hash function across levels (similar to cascade filters [22]).

Lemma 3. *For a given $\varepsilon \geq 1/N$, the amortized I/O complexity of insertion in the external-memory Misra-Gries data structure is $O(\frac{\log(1/\varepsilon M)}{B})$.*

Proof. We call the process of decrementing the counts of all the keys at level i and incrementing all the corresponding key counts at level $i+1$ a ***flush***. A flush can be implemented by rebuilding the B-trees at both levels, which can be done in $O(r^{i+1}M/B)$ I/Os.

Each flush from level i to $i+1$ moves r^iM stream elements down one level, so the amortized cost to move one stream element down one level is $O(\frac{r^{i+1}M}{B}/(r^iM)) = O(r/B)$ I/Os.

Each stream element can be moved down at most L levels. Thus, the overall amortized I/O cost of an insert is $O(rL/B) = O((r/B)\log_r(1/\varepsilon M))$, which is minimized at $r = e$. □

7.4 Online event-detection

We now extend our external-memory Misra-Gries data structure to solve the online event-detection problem. In particular, we show that for a threshold ϕ that is sufficiently large, we can report ϕ-events as soon as they occur.

One way to add immediate reporting to our external-memory Misra-Gries algorithm would be to compute $\widehat{\mathcal{C}}_{L-1}[s_i]$ for each stream event s_i and report s_i as soon as $\widehat{\mathcal{C}}_{L-1}[s_i] > (\phi - \varepsilon)N$. However, this would require querying $\mathcal{C}_i$ for $i = 0, \ldots, L-1$ for every stream element. This could cost up to $O(\log(1/\varepsilon M))$ I/Os per stream element.

We avoid these expensive queries by using the properties of $\mathcal{C}_0$, which is a Misra-Gries frequency estimator. If $\mathcal{C}_0[s_i] \leq (\phi - 1/M)N$, then we know that $f_{s_i} \leq \phi N$ and we therefore do not have to report s_i, regardless of the count for s_i in the lower levels of the external-memory data structure.

Online event-detection in external memory. We amend our external-memory Misra-Gries algorithm to support online event detection as follows. Whenever we increment $\mathcal{C}_0[s_i]$ from a value that is at most $(\phi - 1/M)N$ to a value that is greater than $(\phi - 1/M)N$, we compute $\widehat{\mathcal{C}}_{L-1}[s_i]$ and report s_i if $\widehat{\mathcal{C}}_{L-1}[s_i] = \lceil(\phi - \varepsilon)N\rceil$. For each entry $\mathcal{C}_0[x]$, we store a bit indicating whether we have performed a query for $\widehat{\mathcal{C}}_{L-1}[x]$. As in our basic external-memory Misra-Gries data structure, if the count for an entry $\mathcal{C}_0[x]$ becomes 0, we delete that entry. This means we might query for the same element more than once if its in-memory count crosses the $(\phi - 1/M)N$ threshold, it gets removed from $\mathcal{C}_0$, and then its count crosses the $(\phi - 1/M)N$ threshold again.

As we will see below, this will not affect the I/O cost of the algorithm.[5]

In order to avoid reporting the same element more than once, we can store, with each entry in $\mathcal{C}_i$, a bit indicating whether that key has already been reported. Whenever we report a key x, we set the bit in $\mathcal{C}_0[x]$. Whenever we flush a key from level i to level $i+1$, we set the bit for that key on level $i+1$ if it is set on level i. When we delete the entry for a key that has the bit set on level $L-1$, we add an entry for that key on a new level $\mathcal{C}_L$. This new level contains only keys that have already been reported. When we are checking whether to report a key during a query, we stop checking further and omit reporting as soon as we reach a level where the bit is set. None of these changes affect the I/O complexity of the algorithm.

I/O complexity. In our analysis, we assume that computing $\widehat{\mathcal{C}}_{L-1}[x]$ requires $O(L)$ I/Os. This is true if the levels of the data structure are implemented as sorted arrays with fractional cascading.

We first state the result for the approximate version of the online event-detection problem.

Theorem 6. *Given a stream S of size N and ε and ϕ, where $1/N \le \varepsilon < \phi$ and $(1/\gamma + \Omega(1/N)) < \phi < 1$, the approximate* OEDP *can be solved with an amortized I/O complexity $O\left(\left(\frac{1}{B} + \frac{M}{(\phi M - 1)N}\right)\log\frac{1}{\varepsilon M}\right)$ per stream item.*

Proof. Correctness follows from the arguments above. We need only analyze the I/O costs. We analyze the I/O costs of the insertions and the queries separately.

The amortized cost of performing the insertions is $O(\frac{1}{B}\log\frac{1}{\varepsilon M})$.

To analyze the query costs, let $\varepsilon_0 = 1/M$, i.e., the frequency-approximation error of the in-memory level of our data structure.

Since we perform at most one query each time an element's count in $\mathcal{C}_0$ goes from 0 to $(\phi - \varepsilon_0)N$, the total number of queries is at most $N/((\phi - \varepsilon_0)N) = 1/(\phi - \varepsilon_0) = M/(\phi M - 1)$. Since each query costs $O(\log 1/\varepsilon M)$ I/Os, the overall amortized I/O complexity of the queries is $O\left(\left(\frac{M}{(\phi M - 1)N}\right)\log\frac{1}{\varepsilon M}\right)$. □

[5] It is possible to prevent repeated queries for an element but we allow it as it does not hurt the asymptotic performance.

Exact reporting. If no false positives are allowed, we set $\varepsilon = 1/N$ in Theorem 6. The cost of error-free reporting is that we have to store all the elements, which increases the number of levels and thus the I/O cost. In particular, we have the following result on OEDP.

Corollary 1. *Given $\phi > 1/M + \Omega(1/N)$ and a stream S of size N, the* OEDP *can be solved with amortized I/O complexity $O\Big(\Big(\frac{1}{B} + \frac{M}{(\phi M-1)N}\Big)\log\frac{N}{M}\Big)$ per stream item.*

Summary. The cascading external-memory MG sketch supports a throughput at least as fast as optimal write-optimized dictionaries [19, 20, 23, 37, 38, 42], while estimating the counts as well as an enormous RAM. The external-memory MG maintains count estimates at different granularities. (Not all estimates are actually needed for all structures.) Given the small number of levels, we can refine any given estimate by looking in only a small number of additional locations.

The external-memory MG helps us solve the online event-detection problem. The smallest MG sketch (which fits in memory) is the most important estimator here, because it serves to sparsify queries to the rest of the structure. When such a query gets triggered, we need the total counts from the remaining $\log\frac{N}{M}$ levels for the (exact) online event-detection problem but only $\log\frac{1}{\varepsilon M}$ levels when approximate thresholds are permitted. In the next two sections, we exploit other advantages of this cascading technique to support much lower ϕ without sacrificing I/O efficiency.

7.5 Online event detection with time-stretch

The external-memory Misra-Gries algorithm described in Section 7.4 reports events immediately, albeit at a higher amortized I/O cost for each stream element. In this section, we show that, by allowing some delay in the reporting of events, we can perform event detection asymptotically as cheaply as if we reported all events only at the end of the stream.

Time-stretch filter. We design a new data structure to guarantee time-stretch called the ***time-stretch filter***. Recall that, in order to guarantee a time-stretch of α, we must report an element x no later than time $t_1 + (1+\alpha)F_t$, where t_1 is the time of the first occurrence of x, and F_t is the flow time of x.

As in our external-memory MG scheme, the time-stretch filter consists of $L = \log_r(1/(\varepsilon M))$ levels $\mathcal{C}_0, \ldots, \mathcal{C}_{L-1}$, where the ith level has size $r^i M$, and elements are flushed from higher levels to lower levels.

Unlike the external-memory MG structure in Section 7.4, all events are detected during flush operations. Thus, we never need to perform point queries. This means that (1) we can use simple sorted arrays to represent each level and, (2) we don't need to maintain the invariant that level 0 is a Misra-Gries data structure on its own.

Layout and flushing schedule. We split the table at each level i into $\ell = (\alpha+1)/\alpha$ equal sized ***bins*** $b_1^i, \ldots, b_\ell^i$, each of size $\frac{\alpha}{\alpha+1}(r^i M)$. The capacity of a bin is defined by the sum of the counts of the elements in that bin, i.e., a bin at level i can become full because it contains $\frac{\alpha}{\alpha+1}(r^i M)$ elements, each with count 1, or 1 element with count $\frac{\alpha}{\alpha+1}(r^i M)$, or any other such combination.

We maintain a strict flushing schedule to obtain the time-stretch guarantee. The flushes are performed at the granularity of bins (rather than entire levels). Each stream element is inserted into b_1^0. Whenever a bin b_1^i becomes full (i.e., the sum of the counts of the elements in the bin is equal to its size), we shift all the bins on level i over by one (i.e., bin 1 becomes bin 2, bin 2 becomes bin 3, etc), and we move all the elements in b_ℓ^i into bin b_1^{i+1}. Since the bins in level $i+1$ are r times larger than the bins in level i, b_1^{i+1} becomes full after exactly r flushes from b_ℓ^i. When this happens, we perform a flush recursively on level $i+1$.

Thus, in the time-stretch filter, each element inserted at level i waits $(r^i M)/\alpha$ time steps before it can be flushed down to level $i+1$. This ensures that elements that are placed on a deeper level have aged sufficiently that we can afford to not see them again for a while.

Finally, during a flush involving levels $0, \ldots, j$, we scan these levels and for each element k in the input levels, we sum the counts of each instance of k and, if the total count is greater than $(\phi - \varepsilon)N$, and (we have not reported it before) then we report[6] k.

Theorem 7. *Given a stream S of size N and $1/N \le \varepsilon < \phi < 1$, the approximate* OEDP *can be solved with time-stretch $1+\alpha$ at an amortized I/O complexity $O(\frac{\alpha+1}{\alpha}(\frac{1}{B}\log\frac{1}{\varepsilon M}))$ per stream item.*

Proof. A flush from level i to $i+1$ costs $O(r^{i+1}M/B)$ I/Os, and moves moves $\frac{\alpha}{\alpha+1}r^i M$ stream elements down one level, so the amortized cost to move one stream element down one level is $O(\frac{r^{i+1}M}{B} / \frac{\alpha}{\alpha+1} r^i M) = O(\frac{\alpha+1}{\alpha}\frac{r}{B})$ I/Os.

Each stream element can be moved down at most L levels, thus the overall amortized I/O cost of an insert is $O(\frac{\alpha+1}{\alpha}\frac{rL}{B}) = O\left(\frac{\alpha+1}{\alpha}\frac{r}{B}\log_r \frac{1}{\varepsilon M}\right)$, which is minimized at $r = e$.

We now prove correctness. Consider an element s_t with flow time $F_t = t - t_1$, where t is a ϕ-event and t_1 is the time step of the first occurrence of s_t.

Let $\ell \in \{0, 1, \ldots, L\}$ be the largest level s_t is stored at time t of its ϕNth occurrence. By the flushing algorithm, we are guaranteed that the s_t must have survived at least $r^{\ell-1}/\alpha$ flushes since it was first inserted in the data structure. Thus, $r^{\ell-1}M/(\alpha+1) \le F_t$.

[6]For each reported element, we set a flag that indicates it has been reported, to avoid duplicate reporting of events.

Furthermore, level ℓ is involved in a flush again after $t_\ell = r^{\ell-1}M\alpha/(\alpha+1) \leq \alpha F_t$ time steps. At time t_ℓ during the flush all counts of the element will be consolidated to a total count estimate of $\tilde{c}$. As $\ell \leq L$ and the count-estimate error can be at most εN_{t_ℓ}, where N_{t_ℓ} is the size of the stream up till t_ℓ, we have that $\phi N\tilde{c} + \varepsilon N_t \leq \tilde{c} + \varepsilon N$. Thus, $\tilde{c} \geq (\phi - \varepsilon)N$ and s_t gets reported during the flush at time step t_ℓ, which is no more than αF_t time steps away from t_1.

□

Similar to Section 7.4, if we do not want any false positives among the reported events, we set $\varepsilon = 1/N$. The cost of error-free reporting is that we have to store all the elements, which increases the number of levels and thus the I/O cost. In particular, we have the following result on TEDP.

Corollary 2. *Given $\alpha > 0$ and a stream S of size N, the* OEDP *can be solved with time stretch $1+\alpha$ at an amortized cost of $O\left(\frac{\alpha+1}{\alpha}\frac{\log N/M}{B}\right)$ I/Os per stream item.*

Summary. By allowing a little delay, we can solve the timely event-detection problem at the same asymptotic cost as simply indexing our data [19, 20, 23, 37, 38, 42]. We do not even need to use most of the MG estimates from Section 7.3.1, and what we need we learn essentially for free. These results show that we get different reporting guarantees depending on whether flushing decision from one level to the next are based on age or count. Finally, our results show that there is a spectrum between completely online and completely offline, and it is tunable with little I/O cost.

7.6 Online event detection on power-law distributions

In this section, we assume that the element-counts in the input stream follow a power-law distribution with exponent $\theta > 1$. We design a data structure that solves the OEDP problem and that has better I/O performance and allows for a smaller threshold ϕ when $\theta \geq 2 + 1/(\log_2 N/M)$.[7] We make no assumptions about the order of arrivals.

We call out previous work on power-law assumptions on MG, but, in fact, the assumptions are different. Berinde et al. [25] assume the following. Let $f_1, \ldots, f_u$ be the ranked-frequency vector, that is, $f_1 \geq f_2 \geq \ldots f_u$ of the elements, where $u = |\mathcal{U}|$. They assume that the probability that the rank i element has frequency f_i is equal to $\mathcal{Z} \cdot i^{-\alpha}$, where α is the Zipfian exponent and $\mathcal{Z}$ is the normalization constant. They showed that if $\alpha > 1$, then only $\varepsilon^{-1/\alpha}$ words are sufficient to solve the ε-approximate

[7]The most common power-law distributions found in nature have $2 \leq \theta \leq 3$ [121].

heavy hitter problem. We assume a power-law distribution on the count of the elements in the stream; see Definition 5.

7.6.1 Power-law distributions

We define and state properties of the power-law distribution; for additional details, see [3, 36, 50, 121].

Definition 5 (Continuous Power Law [121]). *A continuous real variable x with a power-law distribution has a probability $p(x)\,dx$ of taking a value in the interval from x to $x+dx$, where $p(x) = Z \cdot x^{-\theta}$ for $\theta > 1$ and Z is the normalization constant.*

In the context of this chapter, the random variable x represents the count of an element in the input stream. In general, the power-law distribution on x may hold above some minimum value $c_{\min}$ of x. For simplicity, we let $c_{\min} = 1$. The normalization constant Z is calculated as follows.

$$1 = \int_1^\infty p(x)\,dx = Z \int_1^\infty x^{-\theta}\,dx = \frac{Z}{\theta - 1}\left[\frac{-1}{x^{\theta-1}}\right]_1^\infty = \frac{Z}{\theta-1}.$$

Thus, $Z = (\theta - 1)$.[8] We will use the cumulative distribution of a power law, that is,

$$\text{Prob } (x > c) = \int_{j=c}^\infty \text{Prob } (x = c) = \int_{j=c}^\infty (\theta - 1)x^{-\theta}dx = \big[-x^{-\theta+1}\big]_c^\infty = \frac{1}{c^{\theta-1}}. \quad (7.1)$$

7.6.2 Popcorn filter

First, we present the layout and data structure parameters. Then, we discuss its main algorithm, the shuffle merge, and finally we analyze its performance.

Layout. The popcorn filter consists of a cascade of Misra-Gries tables, where M is the size of the table in RAM and there are $L = \log_r(2/\varepsilon M)$ levels on disk, where the size of level i is $2/r^{L-i}\varepsilon$.

Each level on disk has an *explicit upper bound* on the number of instances of an element that can be stored on that level (unlike in MG, where it is implicit). In particular, each level i has a ***level threshold*** τ_i for $1 \le i \le L$, $(\tau_1 \ge \tau_2 \ge \ldots \ge \tau_L)$, indicating that the maximum count on level i can be τ_i.

Threshold invariant. We maintain the invariant that at most τ_i instances of an element can be stored on level i. Later, we show how to set τ_i's based on the element-count distribution.

[8]In principle, one could have power-law distributions with $\theta < 1$, but these distributions cannot be normalized and are not common [121].

Shuffle merge. Sections 7.3 and 7.5 use two different flushing strategies, and here we use a third.

Similar to external-memory MG, the level in RAM receives inputs from the stream one at a time. If we are trying to insert at a level i and it is at capacity, instead of flushing elements to the next level, we find the smallest level $j > i$, which has enough empty space to hold all elements from levels $0, 1, \ldots, i$. We aggregate the count of each element k on levels $0, \ldots, j$, resulting in a consolidated count c_k^j. If $c_k^j \geq (\phi - \varepsilon)N$, we report k. Otherwise, we pack instances of k in a bottom-up fashion on levels $j, \ldots, 0$, while maintaining the threshold invariants. In particular, we place $\min\{c_k^j, \tau_j\}$ instances of k on level j, and $\min\{c_k^j - (\sum_{\ell=y+1}^{j} \tau_y), \tau_y\}$ instances of k on level y for $0 \leq y \leq j-1$.

Immediate reporting. As soon as the count of an element k in RAM (level 0) reaches a threshold of $\phi N - 2\tau_1$, the data structure triggers a sweep of the entire L-level data structure, consolidating the count estimates of k at all levels. If the consolidated count reaches $(\phi - \varepsilon)N$, we report k; otherwise we update the k's consolidated count in RAM and "pin" it in RAM—a pinned element does not participate in future shuffle merges. Reported elements are remembered, so that each event gets reported exactly once.

Setting thresholds. We now show how to set the level thresholds based on the power-law exponent so that the data structure does not get "clogged" even though the high-frequency elements are being sent to higher levels of the data structure.

The threshold invariant prevent us from flushing too many counts of an element downstream. As a result, elements get ***pinned***, that is, they cannot be flushed out of a level. Specifically, we say an element is ***pinned at level ℓ*** if its count exceeds $\sum_{i=L}^{\ell+1} \tau_i$.

Too many pinned elements at a level can clog the data structure. We show that if the element-counts in the input stream follow a power-law distribution with exponent θ, we can set the thresholds based on θ in a way that no level has too many pinned elements. In particular, we show the following.

Lemma 4. *Let the element-counts in an input stream S of size N be drawn from a power-law distribution with exponent $\theta > 1$. Let $\tau_i = r^{\frac{1}{\theta-1}}\tau_{i+1}$ for $1 \leq i \leq L-1$ and $\tau_L = (r\varepsilon N)^{\frac{1}{\theta-1}}$. Then the number of keys pinned at any level i is at most half its size, i.e., $1/(r^{L-i}\varepsilon)$.*

Proof. We prove by induction on the number of levels. We start at the penultimate level $L-1$. An element is placed at level $L-1$ if its count is greater than $\tau_L = (r\varepsilon N)^{\frac{1}{\theta-1}}$. By Equation 7.1, there can be at most $N/\tau_L^{\theta-1} = N/(r\varepsilon N) = 1/r\varepsilon$ such elements which proves the base case.

Now suppose the lemma holds for level $i+1$. We show that it holds for level i. An element gets pinned at level $i+1$ if its count is greater than $\sum_{\ell=L}^{i+2} \tau_\ell$. Using Equation 7.1

again, the expected number of such elements is at most $N/(\sum_{\ell=L}^{i+2} \tau_\ell)^{\theta-1} < N/{\tau_{i+2}}^{\theta-1}$. By the induction hypothesis, this is no greater than the size of level $i+1$, that is, $N/{\tau_{i+2}}^{\theta-1} < 1/(\varepsilon r^{L-i-1})$.

Using this, we prove that the expected number of elements pinned at level i is at most $1/(r^{L-i}\varepsilon)$. By Equation 7.1, the expected number of pinned elements at level i is $N/(\sum_{\ell=L}^{i+1} \tau_\ell)^{\theta-1} < N/({\tau_{i+1}}^{\theta-1}) = N/\big((r^{1/\theta-1} \cdot \tau_{i+1})^{\theta-1}\big) < (1/r) \cdot N/({\tau_{i+2}}^{\theta-1}) < 1/(r\varepsilon r^{L-i-1}) = 1/(r^{L-i}\varepsilon)$. □

Theorem 8. *Let S be a stream of size N where the count of elements is drawn from a power-law distribution with exponent $\theta > 1$. Let $\gamma = 2\left(\frac{N}{M}\right)^{\frac{1}{\theta-1}}$. Given ε and ϕ such that $1/N \le \varepsilon < \phi$ and $\phi = \Omega(\gamma/N)$, the approximate* OEDP *on S can be solved with an amortized I/O complexity $O\Big(\Big(\frac{1}{B} + \frac{1}{(\phi N-\gamma)^{\theta-1}}\Big)\log\frac{1}{\varepsilon M}\Big)$ per stream item.*

Proof. Let $\tilde{c}_i$ denote the count estimate of an element i in RAM in the power-law filter. Let f_i be the frequency of i in the stream. Since at most $\sum_{\ell=L}^{1} \tau_\ell < 2\tau_1$ instances of a key can be stored on disk, we have that: $\tilde{c}_i \le f_i \le \tilde{c}_i + 2\tau_1$. Suppose element s_t reaches the threshold ϕN at time t, then its count estimate s_t in RAM must be at least $\tilde{c}_i \ge \phi N - 2\tau_1 = \phi N - 2r^{L/\theta-1}(\varepsilon N)^{1/\theta-1} = \phi N - 2(N/M)^{\frac{1}{\theta-1}} = \phi N - \gamma > 0$.

When the count estimate in RAM reaches $(\phi N - \gamma)$ is exactly when an ODP is triggered which consolidates the count of s_t across all L levels; if the consolidated count reaches $(\phi N - \varepsilon)N$, we report it. This proves correctness as the consolidated count can have an error of at most εN.

We now give I/O complexity of the data structure. The insertions cost $O(rL/B) = O((1/r)\log_r(1/\varepsilon M))$ as we are always able to flush out a constant fraction of a level during a shuffle merge using Lemma 4. This cost is minimized at $r = e$.

Since we perform at most one query each time an element's count in RAM reaches $(\phi N - \gamma)$. The total number of elements in the stream with count at least $(\phi N - \gamma)$ is at most $N/(\phi N - \gamma)^{\theta-1}$. Since each query costs $O(\log 1/\varepsilon M)$ I/Os, the overall amortized I/O complexity of the queries is $O\Big(\frac{1}{(\phi N-\gamma)^{\theta-1}}\log\frac{1}{\varepsilon M}\Big)$. □

Similar to previous section, if we do not want to allow any false positives, we set the error term $\varepsilon = 1/N$. Thus, we get the following result about the OEDP when the element-counts in the input stream follow a power law distribution.

Corollary 3. *Let S be a stream of size N where the count of elements is drawn from a power-law distribution with exponent $\theta > 1$. Let $\gamma = 2\left(\frac{N}{M}\right)^{\frac{1}{\theta-1}}$. Given ϕ $\phi = \Omega(\gamma/N)$, the* OEDP *on S can be solved with an amortized I/O complexity $O\Big(\Big(\frac{1}{B} + \frac{1}{(\phi N-\gamma)^{\theta-1}}\Big)\log\frac{N}{M}\Big)$ per stream item.*

Note that our data structure on an input stream with a power-law distribution allows for strictly smaller thresholds ϕ compared to Theorem 6 and Corollary 1 on worst-case-distributions, when $\theta > 2 + 1/(\log_2(N/M))$. Recall that we need $\phi \ge \Omega(1/M)$ for solving OEDP on worst-case input streams. In Theorem 8 and Corollary 3, we need $\phi \geq \Omega(\gamma/N)$. When we have a power-law distribution with $\theta \geq 2 + 1/(\log_2 N)$, we have $\gamma/N = \frac{2}{M^{1/(\theta-1)} N^{\theta-2}} < \frac{1}{M}$ for $\theta \geq 2 + 1/(\log_2(N/M))$.

Remark on dynamic thresholds We can assign level thresholds dynamically. Initially, each level on disk has a threshold 0 (i.e., $\forall i \in 1, \ldots, L\ \tau_i = 0$). During the first shuffle-merge involving RAM and the first level on disk, we determine the minimum threshold for level 1 (τ_1) required in-order to move at least half of the items from RAM to the first level on disk. When multiple levels, $0, 1, \ldots, i$, are involved in a shuffle-merge, we use a bottom-up strategy to assign thresholds. We determine the minimum threshold required for the bottom most level involved in the shuffle-merge (τ_i) to flush at least half the items from the level just above it (τ_{i-1}). We then apply the same strategy to increment thresholds for levels $i-1, \ldots, 1$.

This means (1) the τ_is for levels $1, \ldots, L$ increase monotonically. Moreover, during shuffle-merges, we increase thresholds of levels involved in the shuffle-merge from bottom-up and to the minimum value so as to not clog the data structure, which means that the τ_is take their minimum possible values. If the τ_i have a feasible setting, then this adaptive strategy will find it.

Then the number of keys pinned at any level i is at most half its size. This means that the τ_is, for assigned dynamically

Lemma 5. *Let the element-counts in an input stream S of size N be drawn from a power-law distribution with exponent $\theta > 1$. During shuffle-merges, we increase thresholds of levels involved in the shuffle-merge from bottom-up and to the minimum value so as to not clog the data structure. Then the number of keys pinned at any level i is at most its size.*

Proof. In the dynamic strategy, thresholds always grow monotonically with the incoming stream. At any time during the input stream, the threshold assigned to any on disk level is the minimum threshold required to not clog the data structure. Therefore, if any threshold was smaller by 1 then the data structure will be clogged. And based on Lemma 4, if the element-counts in the input stream follow a power-law distribution then we can find thresholds so that the data structure does not get clogged. □

Summary. With a power law distribution, we can support a much lower threshold ϕ for the online event-detection problem. In the external-member MG sketch from Section 7.3.1, the upper bounds on the counts at each level are implicit. In this section, we can get better estimates by making these bounds explicit. Moreover, the data structure can learn these bounds adaptively. Thus, the data structure can automatically tailor itself to the power law exponent without needing to be told the exponent explicitly.

7.7 Implementation

In this section we describe our implementation of the time-stretch filter and popcorn filter. As described in Sections 7.5 and 7.6, time-stretch filter and popcorn filter consist of multiple levels. We represent each level as an exact counting quotient filter [131]. We also store a small value (usually a few bits) with each key, in addition to the count. In the time-stretch filter, we use the value bits to track the age of elements. In the popcorn filter, we use the value bits to mark whether an element has its absolute count at a level.

7.7.1 Time-stretch filter

In the time-stretch filter, we need to split each level i into $\ell = (\alpha+1)/\alpha$ equal-sized bins $b_1^i, \ldots, b_\ell^i$, each of size $\frac{\alpha}{\alpha+1}(r^i M)$. In our implementation, instead of actually splitting levels into physical bins we assign a value (i.e., age of the element) of size $\log \ell$-bits to each element which determines the bin an element belongs to. The age of the element on a level determines whether the element is ready to be flushed down from that level during a shuffle-merge. We can do this easily in the counting quotient filter because the counting quotient filter supports storing a small value with every element along with the count of the element as described in Chapter 3.

In addition to assigning a value to each element we also assign an age of size $\log \ell$-bits to each level. The age of a level is incremented if it is involved in a shuffle-merge but not the last level involved in that shuffle-merge. The age of a level gets wrapped around back to 0 after l increments.

We initialize all levels in the time-stretch filter with age 0. When an element is inserted in a level it gets the current age of the level as its age. However, if the level already has an instance of the element then it gets the age of that instance. We increment the age of the level just before a shuffle-merge. During the shuffle-merge, the current age of the level acts as the death age (a death age is the age when an element is eligible to be flushed down during a shuffle-merge) for elements in that level. This is because if an element's age is same as the current age of the level then it means that the element has survived ℓ shuffle-merges on that level.

Shuffle-merge schedule. In the time-stretch filter we follow a fixed schedule for shuffle-merges. Every r^{i-1}-th shuffle-merge involves levels $0, 1, \ldots, i$. Algorithm 6 shows the pseudocode for the shuffle-merge schedule in a time-stretch filter.

In order to decide how many levels to involve in the shuffle-merge we maintain a counter per level to record the number of times a level has been involved in a shuffle-merge. We increment this counter for all the levels involved in the shuffle-merge after each shuffle-merge. This counter is wrapped around back to 0 after r (or the growth factor) increments.

A shuffle-merge is performed every $\frac{\alpha}{\alpha+1}M$-th observation in the stream. As described above we use the age value to split the levels into bins. For example, in order to guarantee a stretch of $1+\alpha$ we need $\log\frac{1+\alpha}{\alpha}$ bits for the age. After every shuffle-merge the age of all the levels but the last level involved in the shuffle-merge is incremented.

Shuffle-merge. The shuffle-merge works like the merge step in a k-way merge sort. We first aggregate the count of an element across all the levels involved in the shuffle-merge. We then decide based on the age of the instance of the element in the last level whether to move it to the next level or not. If the instance of the element in the last level is aged then we insert the element with the aggregate count in the next level. Otherwise, we just update the count of the instance in the last level to the aggregate count. Algorithm 7 shows the pseudocode for the shuffle-merge in a time-stretch filter.

7.7.2 Popcorn filter

In the popcorn filter we assign thresholds to each level on disk. These thresholds acts as an upper bound on the number of instances of an element that can exist on that level.

In our implementation of the popcorn filter we follow a fixed shuffle-merge schedule. A shuffle-merge is invoked from RAM (level 0) after every M observations. However, to determine the number of levels involved in the shuffle-merge we find the smallest level i, such that $i < L$, which has enough empty space to hold all elements from levels $0, 1, \ldots, i$. We use the information about the total occupied slots in the counting quotient filter at each level for this purpose.

During a shuffle-merge, we first aggregate the count of each element and then smear it across all levels involved in the shuffle-merge in a bottom-up fashion without violating the thresholds. Algorithm 8 shows the pseudo code for the shuffle-merge in the popcorn filter.

Immediate reporting. As explained in Section 7.6.1, in order to report an element immediately when it reaches a count T, we aggregate the count of the element on disk when the count in RAM reaches $T - \sum_{i=1}^{L}\tau_i$, where $\sum_{i=1}^{L}\tau_i$ is the maximum count an element can have across all levels on disk.

To compute the aggregate count we perform point queries to each level on disk and sum the counts. If the aggregate count in RAM and on disk is T then we report the element, else we insert the aggregate count in RAM and set a bit (the pinning bit) corresponding to the element that says this is the absolute count of the element.

Therefore, in the future we do not need to perform any point queries to disk because we know that we already have the absolute count of the element in RAM. We employ a lazy policy to delete the instances of the element from disk. They get garbage collected during the next shuffle merge.

7.8 Greedy shuffle-merge schedule

As described above, both data structures follow a fixed shuffle-merge schedule. In the time-stretch filter, we invoke a shuffle-merge after every $\frac{\alpha}{\alpha+1}M$ observations and in the popcorn filter after every M observations. The time-stretch filter upper bounds the amount of time within which an element will be reported after its T-th occurrence and a fixed shuffle-merge schedule is necessary for the timeliness guarantee.

However, the popcorn filter upper bounds the count an element can attain before being reported. Each element appears independent of other elements in the stream and thus we do not need to follow a fixed shuffle-merge schedule in the popcorn filter. Therefore, we can optimize the shuffle-merge schedule in the popcorn filter but not in time-stretch filter.

In the greedy shuffle-merge optimization, instead of invoking a shuffle-merge in the popcorn filter after every M observations, we can instead invoke a shuffle-merge only when it is needed, i.e., invoke a shuffle-merge when the RAM is at capacity. This reduces the frequency of shuffle-merges because the actual number of slots needed for storing M observations can be much smaller than M slots, if there are duplicates in the stream. This is due to the variable-length encoding for counters in the counting quotient filter which packs the counters in the slots allocated for remainders (see Section 3.6). Especially, in case of streams with a skewed distribution, such as the one from Firehose, where counts have a power-law distribution, the space needed to store M observations is much smaller than M slots.

The greedy shuffle-merge optimizations can help avoid unnecessary I/Os to disk that a fixed schedule would incur during shuffle-merges.

7.9 Deamortizing

The I/O analysis in Sections 7.5 and 7.6 is amortized. In this section we describe how to deamortize shuffle-merges in the time-stretch filter and popcorn filter. Deamortization is also necessary to implement the thread-safe version of these data structures. A thread-safe version enables ingesting elements using multiple threads. The scalability with multiple threads is crucial to support ingestion of elements from multiple incoming streams at a high rate. Our deamortization strategy applies to both the time-stretch filter and popcorn filter.

In deamortization, we divide the data structure into multiple smaller data structures which we call ***cones***. A cone is an independent data structure consisting of multiple levels each growing with factor r, where each level is a independent counting quotient filter. Each element in the stream can only go to a specific cone. We use higher order bits of the hash of the element to decide which cone it will go to. For example, if we want to create four cones then we use 2 higher order bits of the hash of the element to determine the cone in which to insert the element.

During ingestion, each thread performs the same set of operations. Multiple threads can simultaneously pick elements from the stream(s) and insert them in their respective cones. Each thread first acquires a lock on the cone before performing an insertion. We use lightweight spin locks to lock a cone. During the insert operation in a cone, if a shuffle-merge is needed to be performed then the same insertion thread performs the shuffle-merge and holds the locks for the duration of the shuffle-merge.

However, the above design does not scale with increasing number of threads due to two reasons. First, some insert operations takes longer than others due to shuffle-merges and the thread performing the shuffle-merge holds the lock the whole time. Second, the input streams often contain elements with skewed distributions and a good proportion of all the elements tends to go to a single cone (or a small set of cones). Due to this, multiple threads try to acquire the lock on a single cone and only one of them can succeed and others wait while spinning for the lock to be released.

To avoid this issue we use a buffering technique similar to the one described in Section 4.2.1. We assign a small buffer (a small counting quotient filter) to each insertion thread. Each thread tries to acquire a lock on the cone but if it does not succeed in the first attempt then it does not wait and spin. Instead it comes back and inserts the element in the local buffer and continues to pick the next element from the stream. The thread then needs to dump the elements from the buffer into the main data structure.

We use two strategies to decide when to dump elements from the local buffer into the main data structure. First, we dump the elements when the local buffer is close to capacity. However, this alone will result in nondeterministic delays in the reporting of elements. Thus we also dump the local buffer at regular intervals to guarantee an upper bound on the delay in the reporting of elements.

7.10 Evaluation

In this section, we evaluate our implementations of the time-stretch filter and popcorn filter. We evaluate our implementations for timeliness, performance, and scalability with multiple threads.

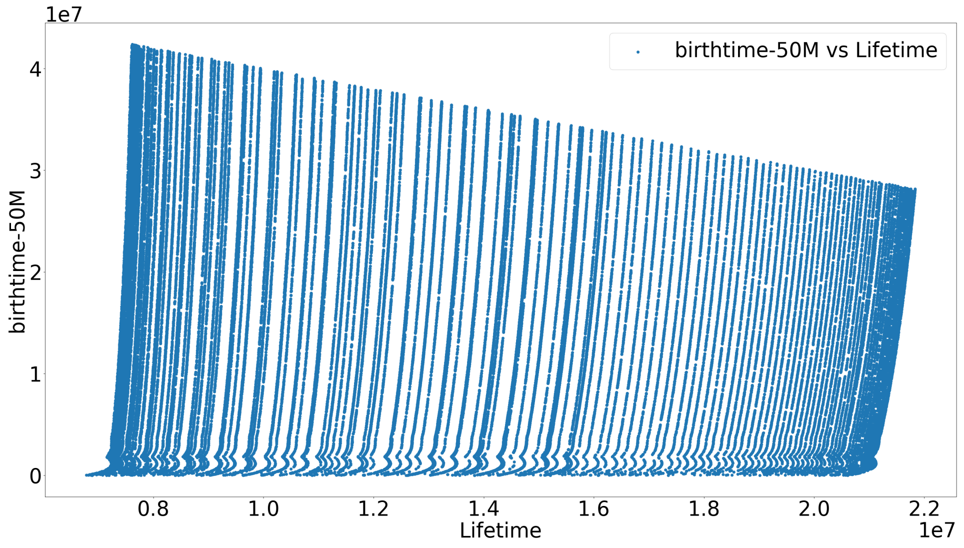


Figure 7.1: Birthtime vs. the lifetime of each reportable element in the active-set generator dataset.

Timeliness. Both the time-stretch filter and popcorn filter provide an upper bound on how late after hitting the threshold value an element can be reported. In timeliness evaluation we validate whether each reportable element is reported within the provided upper bounds of time and count.

Performance. Both the time-stretch filter and popcorn filter provide a bound on the number of I/Os per element during insertion. We evaluate the I/O bounds by calculating the actual number of I/Os performed by the time-stretch filter and popcorn filter to ingest a stream of N observations.

Scalability. We evaluate how the performance (i.e., insertion throughput) of the time-stretch filter and popcorn filter scales with increasing number of ingestion threads.

7.10.1 Experimental setup

Here we describe how we design our experiments to evaluate the time-stretch filter and popcorn filter on the parameters stated above. We first describe the workload we use for the evaluation. We then describe how we validate timeliness and measure the number of I/Os.

Workload. Firehose [94] 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 24th 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 power-law distribution, the active-set generator generates items with a uniformly random distribution but assigns counts to them from a power-law distribution. The power-law generator picks keys from a static range of 100,000 keys, following a power-law distribution. Therefore, the stream generated is highly skewed and is easy to handle in our data structures. The active-set generator selects keys from a continuously evolving active-set of 128,000 keys as the default. The generator first creates a power-law distribution (with exponent 2.5) over counts and then random picks a key from a from the set of active keys and assigns it a count. 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.

For our evaluation, we use the active-set generator because streams from the active-set generator are better at representing real-life workloads compared to the power-law generator. However, in the default Firehose settings (i.e., active set size 128K) the longest lifetime was $\approx$ 3M observations which is relatively short and poses no challenge to report keys out of RAM. Therefore, we changed the size of the active set to 1M keys so that keys can have a longer lifetime in the stream and would force the data structure to perform shuffle-merges.

Figure 7.1 shows the distribution of birthtime vs. the lifetime of keys in the stream from active-set generator. The stream contains 50M observations and the active set size is 1M. The longest lifetime is $\approx$ 22M. Whenever a new key is added to the active set it is assigned a count value from the set of counts based on the power-law distribution. Therefore, we see these bands of keys that have similar lifetime but born at different times throughout the stream. The lifetime of keys in these bands tend to increase slightly as the keys are born later in the stream due the way the different selection probabilities of keys from the active set. In all our experiments we have used dataset from the active-set generator unless noted otherwise.

Validation. In the time-stretch filter, the upper bound is on the time (i.e., the index of the observation in the stream) within which an element can be reported based on its lifetime. To validate the timeliness guarantee of the time-stretch filter we first perform an offline analysis on the stream. Given a reporting threshold T, we record the index of the first occurrence of the element (I_0) and index of the T-th occurrence of the element (I_T). We then record the index at which the element is actually reported by the time-stretch filter (I_R). We then calculate the time stretch (ts) for each reported element as $ts = (I_R - I_0)/(I_T - I_0)$ and validate that $ts \leq (1 + \alpha)$.

In the popcorn filter, the upper bound is on the actual count of the element when it is reported based on the reporting threshold. To validate we first record the index at which an element is reported by the data structure (I_R). We then perform an offline analysis of the stream to determine the count of the element at index I_R (C_{I_R}). We then calculate the count stretch (cs) as $cs = C_{I_R}/T$ and validate that $cs \leq 2$.

To perform the offline analysis of the stream we first generate the stream from the active-set generator and dump it in a file on disk. We then read the stream from disk to do the analysis and also feed it to the data structure. This offline analysis ensures that we do not drop any packets while analyzing the stream feeding to the data structure. For timeliness validation experiments we use a stream of 50M observations from the active-set generator.

I/O performance. In our implementation, we allocate space for the data structure by mmap-ing each the counting quotient filters to a file on disk representing the levels of the data structure. We then control the available RAM to the insertion process using the "cgroups" utility in linux. We calculate the total RAM needed by the process by adding the size of the RAM CQF, the space used by the anomaly CQF to record reported keys, the space used by thread local buffers, and a small amount of extra space to read stream sequentially from disk. We then provision the RAM to the next power-of-two of the sum.

To count the number of I/Os performed by the data structure we use the "iotop" utility in linux. Using `iotop` we can count the total reads and writes in KB performed by the process doing insertions.

Similar to validation experiments, for I/O evaluation experiments we first dump the stream to disk and then feed it to the data structure. We use a stream of 50M observations from the active-set generator.

Throughput. We measure the raw throughput of the data structure in terms of the number of observations inserted over time. For the scalability evaluation we measure how the throughput of the data structure changes with increasing number of threads.

Machine specifications. All benchmarks were performed on 64-bit Ubuntu 18.04 running Linux kernel 4.15.0-34-generic. The machine has an Intel Skylake CPU (Core(TM)

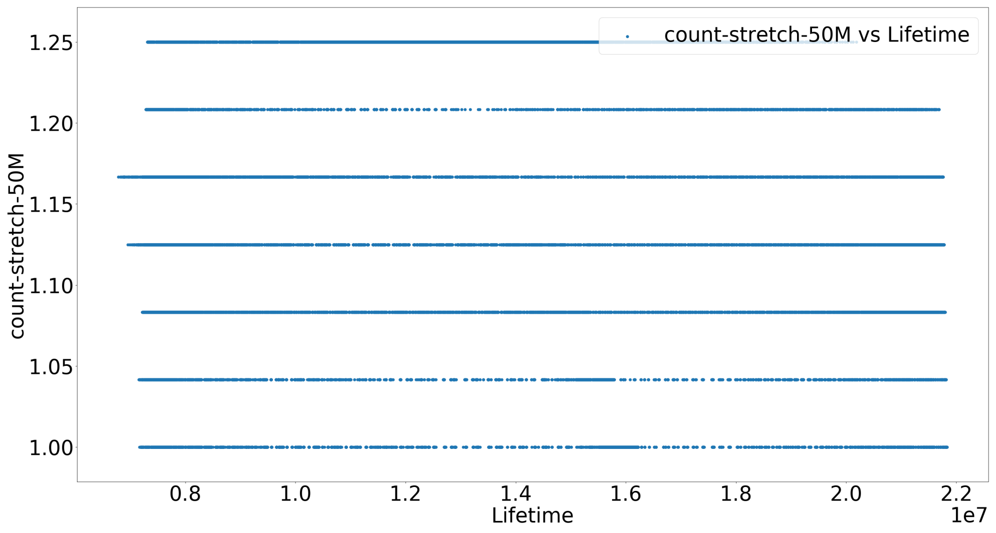


Figure 7.2: Count-stretch vs. the lifetime of each reportable element in the active-set generator dataset. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, level thresholds for on-disk level($\ell_2 \dots \ell_1$): (2, 4), number of observations: 50M.

i7-6700HQ CPU @ 2.60GHz with 4 cores and 6MB L3 cache) with 32 GB RAM and a 1TB Toshiba SSD.

For all the experiments, we use a reporting threshold 24 since it is used as the default in the Firehose benchmarking suite.

7.10.2 Timely reporting

Popcorn filter. We first validate the popcorn filter without immediate reporting. The maximum count on disk is 6, so the maximum count of an element when it is reported can be 30. The count stretch can have seven different values between $1, \dots, 1.25$. Figure 7.2 shows the count stretch of each reported element. We see that the maximum reported count stretch is 1.25. And all reported elements have count stretch ranging from 1 to 1.25.

Time-stretch filter. For the time stretch validation, we performed four experiments with different α values. We changed α values by changing the number of bins that we divide each level into, which in-turn is done by using different numbers of bits to represent the ages of elements and levels. Table 7.1 shows the relation between the number of age bits and α values.

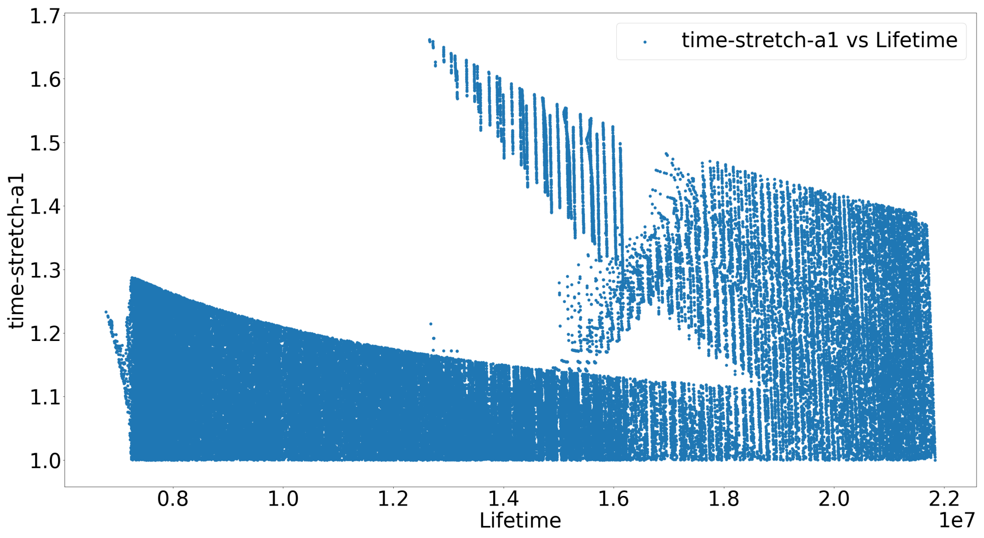


Figure 7.3: Time-stretch vs. the lifetime of each reportable element in the active-set generator dataset. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of bins: 2, number of observations: 50M.

#age bits	**#bins**	α
1	2	1
2	4	1.33
3	8	1.14
4	16	1.06

Table 7.1: Relation between number of age bits, number of bins, and α values.

Figures 7.3 to 7.6 show the time stretch of each reported element. We see that the time stretch of any reported element is always smaller than the maximum allowable time stretch listed in Table 7.1.

In all the time-stretch plots, there is an initial period (items with $\approx$ 4M lifetime) when all items have a time-stretch of 1. These are all those items that are reported from the RAM level before the first flush in which items are moved to the next level. Therefore, all these items have a time stretch 1.

Also, as the plots show there are two waves of time-stretch values. Both waves start with hitting the highest time stretch initially and then gradually going down. The highest time stretch happens when the items are first moved to the next level. The gradual decay in the time stretch occurs because elements that are reported from the same level but during later shuffle-merges have smaller time stretches compared to

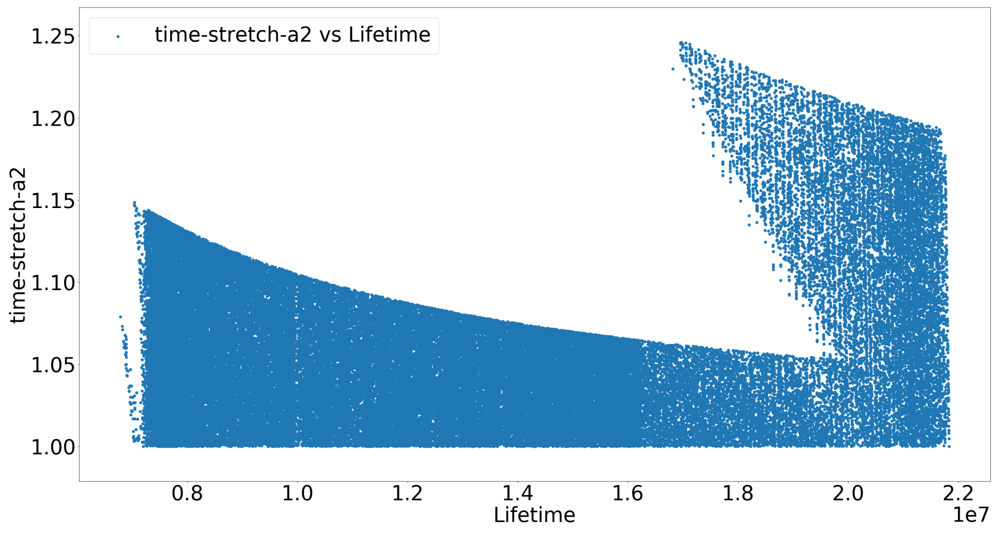


Figure 7.4: Time-stretch vs. the lifetime of each reportable element in the active-set generator dataset. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of bins: 4, number of observations: 50M.

elements that are reported in earlier shuffle-merges. This is due to the fact that we calculate the time stretch using the overall lifetime of the element in the stream and not the total time spent by the element on each level.

The first wave consists of the items that are moved to the first level on disk without being reported. And the second wave consists of the items that are moved to the second level on disk. The first starts to appear around the same time for every time-stretch value but the second wave starts to appear later as we decrease the time stretch. This is because as we decrease the time stretch items require more time to be moved to the last level on disk.

Based on our experiments we can validate that the popcorn filter and time-stretch filter empirically provide the intended timeliness guarantees.

7.10.3 I/O performance

Table 7.2 shows the total sequential reads and writes performed, calculated based on the number of shuffle-merges during the insertion. We also measured the total number of point queries (or random reads) performed by the popcorn filter with immediate reporting. To validate, we measured the total reads and writes performed by the process using “iotop”.

For all experiments, the total reads and writes as given by `iotop` is similar to what

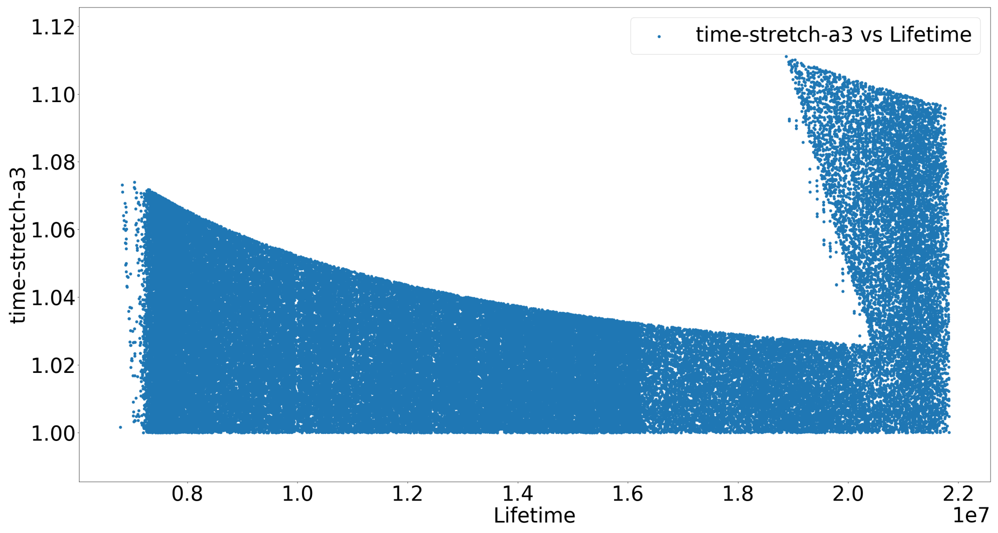


Figure 7.5: Time-stretch vs. the lifetime of each reportable element in the active-set generator dataset. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of bins: 8, number of observations: 50M.

we calculated based on the shuffle-merges. The popcorn filter does the least amount of I/Os because it performs the least number of shuffle-merges. For the time-stretch filter the amount of I/O grows by the factor of two as we increase the number of bins as predicted by the theory.

7.10.4 Scaling with multiple threads

Table 7.3 and Figures 7.7 and 7.8 show the throughput of the time-stretch filter and popcorn filter in terms of number of insertions per second.

The popcorn filter has the highest throughput among all the data structures. This is

Filter	**Seq reads**	**Seq writes**	**point queries**	**reads reported**	**writes reported**
popcorn filter	1 398 141	1 615 197	0	1 157 115	1 811 719
popcorn filter (immediate reporting)	1 453 542	1 678 316	191 322	1 782 303	1 684 229
time-stretch filter-2	2 311 195	2 655 848	0	2 034 367	2 698 935
time-stretch filter-4	5 070 866	5 799 339	0	4 680 427	5 390 067
time-stretch filter-8	10 797 431	12 318 755	0	10 207 707	11 194 569
time-stretch filter-16	22 592 907	25 733 310	0	21 591 663	23 095 296

Table 7.2: Total number of I/Os performed by the time-stretch filter and popcorn filter. All numbers are in KB. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of observations: 50M.

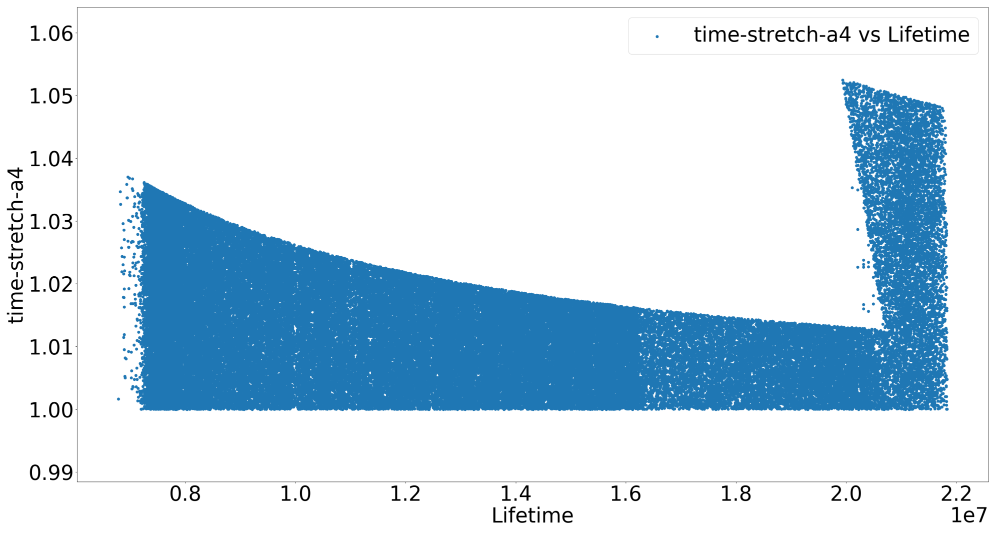


Figure 7.6: Time-stretch vs. the lifetime of each reportable element in the active-set generator dataset. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of bins: 16, number of observations: 50M.

because the popcorn filter performs the least amount of I/Os compared to the other data structures. The popcorn filter with immediate reporting has lower throughput because of the extra random point queries performed by the data structure. The throughput in the time-stretch filter goes down as we add bins and decrease the stretch.

The throughput of all these data structures increases as we add more threads. However, the throughput increases more for bigger datasets (e.g., 500M) because we are able to create more cones with the same number of threads. Having more cones reduces the amount of contention among threads.

Filter	Number of insertions per second
popcorn filter	938 086.30
popcorn filter (immediate reporting)	469 351.35
time-stretch filter-2	485 389.76
time-stretch filter-4	246 390.38
time-stretch filter-8	122 835.03
time-stretch filter-16	60 852.42

Table 7.3: Total number of items inserted per second by the time-stretch filter and popcorn filter. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of observations: 50M.

Algorithm 6 Time-Stretch filter shuffle-merge schedule

```
1: procedure NEED_FLUSH($num\_obs$, $max\_age$)
2:     $flush\_size \leftarrow levels[0].size/max\_age$
3:     if $num\_obs \bmod flush\_size = 0$ then
4:         return $true$
5:     return $false$
6: procedure FIND_NUM_LEVELS($flushes$)
7:     $i \leftarrow 1$
8:     for $i$ in TOTAL_LEVELS $- 1$ do
9:         if $levels[i].flushes < growth\_factor - 1$ then
10:            break
11:    return i
12: procedure PERFORM_SHUFFLE_MERGE_IF_NEEDED
13:    $max\_age \leftarrow 2^{num\_age\_bits}$
14:    if NEED_FLUSH($num\_obs, max\_age$) then
15:        $num\_levels \leftarrow$ FIND_NUM_LEVELS($num\_flushes$)
16:        for $i$ in $num\_levels$ do
17:            $levels[i].age \leftarrow (levels[i].age + 1)\%max\_age$
18:        SHUFFLE_MERGE($num\_levels$)
19:        for $i$ in $num\_levels$ do
20:            $levels[i].flushes \leftarrow (levels[i].flushes + 1)\%growth\_factor$
21:        $num\_flushes \leftarrow num\_flushes + 1$
```

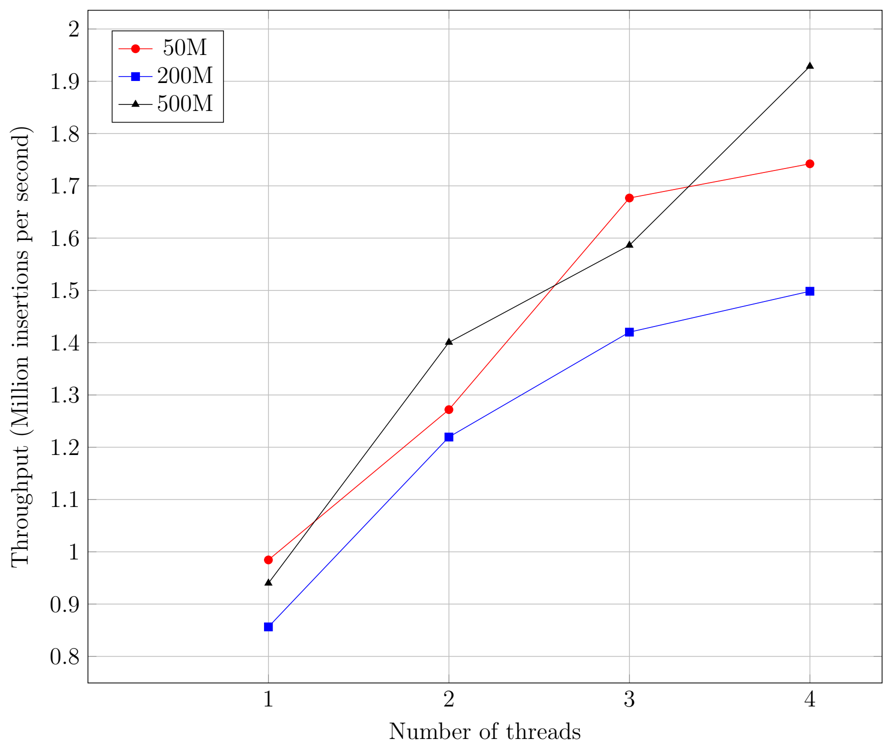


Figure 7.7: Insertion throughput with increasing number of threads for the popcorn filter. Data structure configuration: RAM level: 4194304 (50M), 8388608 (200M), and 16777216 (500M) slots in the CQF, levels: 3(50M), 4(200M, 500M), growth factor: 4, level thresholds for on-disk level($\ell_3 \ldots \ell_1$): (2, 4, 8), cones: 8(50M and 200M) and 16 (500M) with greedy optimization.

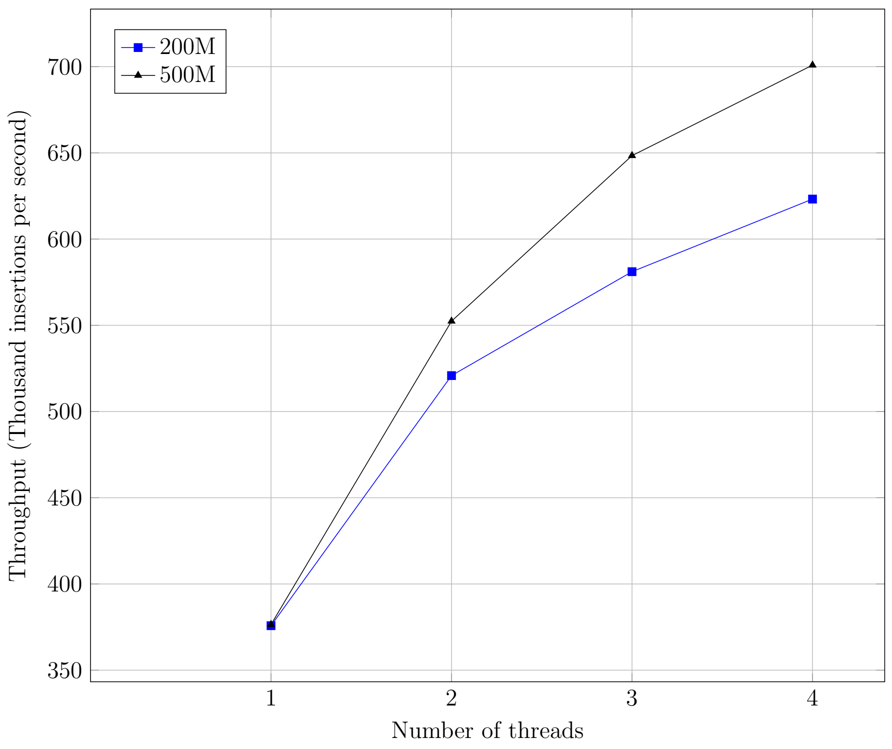


Figure 7.8: Insertion throughput with increasing number of threads for the time-stretch filter. Data structure configuration: RAM level: 8388608 (200M), and 16777216 (500M) slots in the CQF, levels: 4, growth factor: 4, cones: 8(200M) and 16 (500M).

Algorithm 7 Time Stretch Shuffle Merge

```
procedure IS_AGED($age$, $level$)
    if $age = level.age$ then
        return $true$
    return $false$
procedure SHUFFLE_MERGE($levels$)
    $e \leftarrow -\infty$
    while true do
        $e \leftarrow$ SHUFFLE_MERGE_STEP($levels, e$)
        if $e = \infty$ then
            break
procedure SHUFFLE_MERGE_STEP($nlevels$, $prev$)
    $min \leftarrow \infty$
    $count \leftarrow 0$
    $age \leftarrow 0$
    $level \leftarrow 0$
    for $i$ in $nlevels$ do
        $next[i] \leftarrow levels[i].succ(prev)$
        if $min > next[i]$ then
            $min \leftarrow next[i]$
    if $min = \infty$ then
        return $\infty$
    for $i$ in $nlevels$ do
        if $min = next[i]$ then
            $count \leftarrow count + next[i].count$
            $level \leftarrow next[i].level$
            $age \leftarrow next[i].age$
            $levels[i].delete(min)$
    if $level < nlevel$ and IS_AGED($age, level$) then
        $l\_age \leftarrow levels[level + 1].age$
        $levels[level + 1].insert(min, count, l\_age)$
    else
        $levels[min.level].insert(min, count, age)$
    return $min$
```

Algorithm 8 Popcorn filter Shuffle Merge

```
1: procedure SHUFFLE_MERGE($levels$, $\tau$)
2:     $e \leftarrow -\infty$
3:     while true do
4:         $e \leftarrow$ SHUFFLE_MERGE_STEP($levels, \tau, e$)
5:         if $e = \infty$ then
6:             break
7: procedure SHUFFLE_MERGE_STEP($levels$, $\tau$, $prev$)
8:     $min \leftarrow \infty$
9:     for $i$ in $levels$ do
10:        $next[i] \leftarrow levels[i].succ(prev)$
11:        if $min > next[i]$ then
12:            $min \leftarrow next[i]$
13:    if $min = \infty$ then
14:        return $\infty$
15:    $count \leftarrow 0$
16:    for $i$ in $levels$ do
17:        if $min = next[i]$ then
18:            $count \leftarrow count + next[i].count$
19:            $levels[i].delete(min)$
20:    for $i$ in $levels$ (Bottom to Top) do
21:        if $count \geq \tau[i]$ then
22:            $levels[i].insert(min, \tau[i])$
23:            $count \leftarrow count - \tau[i]$
24:        else if $count > 0$ then
25:            $levels[i].insert(min, count)$
26:        else
27:            break
28:    return $min$
```

Chapter 7: Bibliography

[1] FireHose streaming benchmarks. `firehose.sandia.gov`. Accessed: 2017-1-11.

[2] *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].

[3] L. Adamic. Zipf, power law, pareto: a ranking tutorial. HP Research, 2008.

[4] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

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

[6] F. Almodaresi, P. Pandey, and R. Patro. Rainbowfish: A Succinct Colored de Bruijn Graph Representation. In R. Schwartz and K. Reinert, editors, *17th International Workshop on Algorithms in Bioinformatics (WABI 2017)*, volume 88 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:15, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 20–29, 1996.

[8] 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.

[9] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[10] K. Anderson and S. Plimpton. Firehose streaming benchmarks. Technical report, Sandia National Laboratory, 2015.

[11] A. Appleby. Murmurhash. `https://sites.google.com/site/murmurhash/`, 2016. [Online; accessed 19-July-2016].

[12] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[13] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, et al. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[14] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Journal of Acta Informatica*, 1(3):173–189, Feb. 1972.

[15] D. Belazzougui, T. Gagie, V. Mäkinen, and M. Previtali. *Fully Dynamic de Bruijn Graphs*, pages 145–152. Springer International Publishing, Cham, 2016.

[16] D. Belazzougui, T. Gagie, V. Mäkinen, and M. Previtali. Fully dynamic de bruijn graphs. In *International Symposium on String Processing and Information Retrieval*, pages 145–152. Springer, 2016.

[17] M. A. Bender, J. W. Berry, M. Farach-Colton, J. Jacobs, R. Johnson, T. M. Kroeger, T. Mayer, S. McCauley, P. Pandey, C. A. Phillips, A. Porter, S. Singh, J. Raizes, H. Xu, and D. Zage. Advanced data structures for improved cyber resilience and awareness in untrusted environments: LDRD report. Technical Report SAND2018-5404, Sandia National Laboratories, May 2018.

[18] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[19] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 81–92, San Diego, CA, June 9–11 2007.

[20] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. An introduction to b$^{\varepsilon}$-trees and write-optimization. *:login; magazine*, 40(5):22–28, October 2015.

[21] 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.

[22] M. A. Bender, M. Farach-Colton, R. Johnson, 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. In *Proceedings of the 3rd USENIX conference on Hot Topics in Storage and File Systems*, HotStorage'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[23] M. A. Bender, M. Farach-Colton, R. Johnson, S. Mauras, T. Mayer, C. A. Phillips, and H. Xu. Write-optimized skip lists. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 69–78. ACM, 2017.

[24] 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*.

[25] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems (TODS)*, 35(4):26, 2010.

[26] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology*, 33(6):623–630, 2015.

[27] J. Berry, R. D. Carr, W. E. Hart, V. J. Leung, C. A. Phillips, and J.-P. Watson. Designing contamination warning systems for municipal water networks using imperfect sensors. *Journal of Water Resources Planning and Management*, 135, 2009.

[28] A. Bhattacharyya, P. Dey, and D. P. Woodruff. An optimal algorithm for l1-heavy hitters in insertion streams and related problems. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 385–400, 2016.

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

[30] F. Bonomi, M. Mitzenmacher, R. Panigrahy, et al. An improved construction for counting Bloom filters. In *ECA'06*, 2006.

[31] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Bounds for frequency estimation of packet streams. In *SIROCCO*, pages 33–42, 2003.

[32] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In *Proceedings of the International Workshop on Algorithms in Bioinformatics*, pages 225–235. Springer, 2012.

[33] R. S. Boyer and J. S. Moore. *A fast majority vote algorithm.* SRI International. Computer Science Laboratory, 1981.

[34] V. Braverman, S. R. Chestnut, N. Ivkin, J. Nelson, Z. Wang, and D. P. Woodruff. Bptree: an ℓ_2 heavy hitters algorithm using constant memory. *arXiv preprint arXiv:1603.00759*, 2016.

[35] V. Braverman, S. R. Chestnut, N. Ivkin, and D. P. Woodruff. Beating countsketch for heavy hitters in insertion streams. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 740–753, 2016.

[36] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134, 1999.

[37] G. S. Brodal, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, and J. I. Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *SODA*, pages 1448–1456, 2010.

[38] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, 2003.

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

[40] C. T. Brown, A. Howe, Q. Zhang, A. B. Pyrkosz, and T. H. Brom. A reference-free algorithm for computational normalization of shotgun sequencing data. *arXiv preprint arXiv:1203.4802*, 2012.

[41] B. Buchfink, C. Xie, and D. H. Huson. Fast and sensitive protein alignment using DIAMOND. *Nature methods*, 12(1):59–60, 2015.

[42] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 859–860, San Francisco, CA, 2000.

[43] 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.

[44] A. B. Carvalho, E. Dupim, and G. Goldstein. Improved assembly of noisy long reads by k-mer validation. *Genome Research*, 2016.

[45] Z. Chang, G. Li, J. Liu, Y. Zhang, C. Ashby, D. Liu, C. L. Cramer, and X. Huang. Bridger: a new framework for de novo transcriptome assembly using rna-seq data. *Genome Biol*, 16(1):30, 2015.

[46] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the Twenty-Ninth International Colloquium on Automata, Languages and Programming (ICALP 2002)*, pages 693–703, Malaga, Spain, July 8–13 2002.

[47] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev. On the representation of de Bruijn graphs. In *Proceedings of the International Conference on Research in Computational Molecular Biology*, pages 35–55. Springer, 2014.

[48] R. Chikhi and G. Rizk. Space-efficient and exact de bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):1, 2013.

[49] Cisco. Cisco IOS intrusion prevention system (IPS). https://www.cisco.com/c/en/us/products/security/ios-intrusion-prevention-system-ips/index.html.

[50] A. Clauset, C. R. Shalizi, and M. E. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[51] S. Cohen and Y. Matias. Spectral Bloom filters. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2003.

[52] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[53] P. E. Compeau, P. A. Pevzner, and G. Tesler. How to apply de Bruijn graphs to genome assembly. *Nature biotechnology*, 29(11):987–991, 2011.

[54] A. Conway, M. Farach-Colton, and P. Shilane. Optimal hashing in external memory. In *Proceedings of the Forty-Fifth International Colloquium on Automata, Languages, and Programming*, page 39:139:14, 2018.

[55] G. Cormode. Misra-Gries summaries. *Encyclopedia of Algorithms*, pages 1–5, 2008.

[56] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *Latin American Symposium on Theoretical Informatics*, pages 29–38, 2004.

[57] 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.

[58] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. *ACM Transactions on Database Systems (TODS)*, 30(1):249–278, 2005.

[59] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *European Symposium on Algorithms*, pages 194–205. Springer, 2010.

[60] A. Danek. Kmc2 github. `https://github.com/refresh-bio/KMC`, 2016. [Online; accessed 29-Apr-2016].

[61] N. M. Daniels, A. Gallant, J. Peng, L. J. Cowen, M. Baym, and B. Berger. Compressive genomics for protein databases. *Bioinformatics*, 29(13):i283–i290, 2013.

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

[63] B. K. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proc. USENIX annual technical conference*, 2010.

[64] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.

[65] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz. Kmc 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[66] X. Dimitropoulos, P. Hurley, and A. Kind. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *ACM SIGCOMM Computer Communication Review*, 38(1):5–5, 2008.

[67] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[68] B. Fan. C++ cuckoo filter. `https://github.com/efficient/cuckoofilter`, 2014. [Online; accessed 19-July-2014].

[69] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proc. 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[70] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM T. Netw.*, 8:281–293, June 2000.

[71] R. M. Fano. *On the number of bits required to implement an associative memory.* Massachusetts Institute of Technology, Project MAC, 1971.

[72] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[73] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, NY, Oct. 17–19 1999.

[74] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, Jan. 2012.

[75] A. Geil, M. Farach-Colton, and J. D. Owens. Quotient filters: Approximate membership queries on the gpu. 2018.

[76] Gencode. Release 25. `https://www.gencodegenes.org/releases/25.html`, 2017. [online; accessed 06-Nov-2017].

[77] 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 *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 437–448, 2014.

[78] S. Gog. Succinct data structure library. `https://github.com/simongog/sdsl-lite`, 2017. [online; accessed 01-Feb-2017].

[79] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[80] S. Gog and M. Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014.

[81] J. M. Gonzalez, V. Paxson, and N. Weaver. Shunting: A hardware/software architecture for flexible, high-performance network intrusion prevention. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 139–149, 2007.

[82] 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.

[83] M. Goswami, D. Medjedovic, E. Mekic, and P. Pandey. Buffered count-min sketch on SSD: theory and experiments. In Y. Azar, H. Bast, and G. Herman, editors, *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, volume 112 of *LIPIcs*, pages 41:1–41:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[84] M. G. Grabherr, B. J. Haas, M. Yassour, J. Z. Levin, D. A. Thompson, I. Amit, X. Adiconis, L. Fan, R. Raychowdhury, Q. Zeng, et al. Full-length transcriptome assembly from rna-seq data without a reference genome. *Nature biotechnology*, 29(7):644–652, 2011.

[85] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[86] Y. Heo, X.-L. Wu, D. Chen, J. Ma, and W.-M. Hwu. BLESS: bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, page btu030, 2014.

[87] Y. Hilewitz and R. B. Lee. Fast bit compression and expansion with parallel extract and parallel deposit instructions. In *Application-specific Systems, Architectures and Processors, 2006. ASAP'06. International Conference on*, pages 65–72. IEEE, 2006.

[88] G. Holley, R. Wittler, and J. Stoye. Bloom filter trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms Mol. Biol.*, 11:3, 2016.

[89] W. Huang, L. Li, J. R. Myers, and G. T. Marth. ART: A next-generation sequencing read simulator. *Bioinformatics*, 28(4):593, 2012.

[90] J. Iacono and M. Pătraşcu. Using hashing to solve the dictionary problem. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 570–582, 2012.

[91] Iqbal Zamin, Caccamo Mario, Turner Isaac, Flicek Paul, and McVean Gil. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44:226—232, January 2012.

[92] G. Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

[93] S. Kannan, J. Hui, K. Mazooji, L. Pachter, and D. Tse. Shannon: An information-optimal de novo rna-seq assembler. *bioRxiv*, 2016.

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

[95] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, 2003.

[96] M. Kezunovic. Monitoring of power system topology in real-time. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, volume 10, pages 244b–244b, Jan 2006.

[97] C. Kingsford. Srr list. `https://www.cs.cmu.edu/~ckingsf/software/bloomtree/srr-list.txt`, 2017. [online; accessed 06-Nov-2017].

[98] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[99] Y. Kodama, M. Shumway, and R. Leinonen. The sequence read archive: explosive growth of sequencing data. *Nucleic acids research*, 40(D1):D54–D56, 2011.

[100] S. Koren, B. P. Walenz, K. Berlin, J. R. Miller, and A. M. Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *bioRxiv*, page 071282, 2016.

[101] E. Kushilevitz. Communication complexity. In *Advances in Computers*, volume 44, pages 331–360. Elsevier, 1997.

[102] K. G. Larsen, J. Nelson, H. L. Nguyen, and M. Thorup. Heavy hitters via cluster-preserving clustering. In *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 61–70, 2016.

[103] Q. Le Sceller, E. B. Karbab, M. Debbabi, and F. Iqbal. SONAR: Automatic detection of cyber security events over the twitter stream. In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES)*, 2017.

[104] H. Li. Inthash. `https://gist.github.com/lh3/974ced188be2f90422cc`, 2016. [Online; accessed 19-July-2016].

[105] E. Litvinov. Real-time stability in power systems: Techniques for early detection of the risk of blackout [book review]. *IEEE Power and Energy Magazine*, 4(3):68–70, May 2006.

[106] J. Liu, G. Li, Z. Chang, T. Yu, B. Liu, R. McMullen, P. Chen, and X. Huang. Binpacker: Packing-based de novo transcriptome assembly from rna-seq data. *PLOS Comput Biol*, 12(2):e1004772, 2016.

[107] Y. Liu, J. Schröder, and B. Schmidt. Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, 29(3):308–315, 2013.

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

[109] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002.

[110] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[111] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX Conference on Security*, 2010.

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

[113] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

[114] J. Misra and D. Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[115] H. Mohamadi, H. Khan, and I. Birol. ntcard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, 33(9):1324–1330, 2017.

[116] M. D. Muggli, A. Bowe, N. R. Noyes, P. Morley, K. Belk, R. Raymond, T. Gagie, S. J. Puglisi, and C. Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, page btx067, 2017.

[117] K. D. Murray, C. Webers, C. S. Ong, J. O. Borevitz, and N. Warthmann. kwip: The k-mer weighted inner product, a de novo estimator of genetic similarity. *bioRxiv*, 2016.

[118] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In J. A. Stankovic, A. Arora, and R. Govindan, editors, *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004, Baltimore, MD, USA, November 3-5, 2004*, pages 250–262. ACM, 2004.

[119] G. Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys (CSUR)*, 46(4):52, 2014.

[120] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *International Symposium on Experimental Algorithms*, pages 295–306. Springer, 2012.

[121] M. E. Newman. Power laws, pareto distributions and Zipf's law. *Contemporary physics*, 46(5):323–351, 2005.

[122] NIH. Sra. `https://www.ebi.ac.uk/ena/browse`, 2017. [online; accessed 06-Nov-2017].

[123] B. D. Ondov, T. J. Treangen, P. Melsted, A. B. Mallonee, N. H. Bergman, S. Koren, and A. M. Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome biology*, 17(1):132, 2016.

[124] 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.

[125] R. Ounit, S. Wanamaker, T. J. Close, and S. Lonardi. Clark: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC genomics*, 16(1):1, 2015.

[126] R. Pagh and F. F. Rodler. Cuckoo hashing. In F. M. auf der Heide, editor, *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August*

28-31, 2001, Proceedings, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

[127] Palo Alto Networks. What is an intrusion prevention system? https://www.paloaltonetworks.com/cyberpedia/what-is-an-intrusion-prevention-system-ips.

[128] P. Pandey, F. Almodaresi, M. A. Bender, M. Ferdman, R. Johnson, and R. Patro. Mantis: A fast, small, and exact large-scale sequence-search index. In *Research in Computational Molecular Biology*, page 271. Springer.

[129] P. Pandey, M. A. Bender, and R. Johnson. A fast x86 implementation of select. *CoRR*, abs/1706.00990, 2017.

[130] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. debgr: an efficient and near-exact representation of the weighted de bruijn graph. *Bioinformatics*, 33(14):i133–i141, 2017.

[131] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. In Salihoglu et al. [148], pages 775–787.

[132] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. Squeakr: An exact and approximate k-mer counting system. *Bioinformatics*, page btx636, 2017.

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

[134] R. Patro, S. M. Mount, and C. Kingsford. Sailfish enables alignment-free isoform quantification from rna-seq reads using lightweight algorithms. *Nature biotechnology*, 32(5):462–464, 2014.

[135] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *In Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[136] D. Pellow, D. Filippova, and C. Kingsford. Improving Bloom filter performance on sequence data using k-mer Bloom filters. In *International Conference on Research in Computational Molecular Biology*, pages 137–151. Springer, 2016.

[137] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to DNA fragment assembly. *In Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[138] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient Bloom filters. In *Proc. Experimental Algorithms*, pages 108–121. Springer, 2007.

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

[140] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242. Society for Industrial and Applied Mathematics, 2002.

[141] S. Raza, L. Wallgren, and T. Voigt. Svelte: Real-time intrusion detection in the internet of things. *Ad Hoc Networks*, 11(8):2661–2674, 2013.

[142] M. Remmert, A. Biegert, A. Hauser, and J. Söding. Hhblits: lightning-fast iterative protein sequence searching by hmm-hmm alignment. *Nature methods*, 9(2):173–175, 2012.

[143] G. Rizk, D. Lavenier, and R. Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, page btt020, 2013.

[144] M. Roberts, W. Hayes, B. R. Hunt, S. M. Mount, and J. A. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[145] T. Roughgarden et al. Communication complexity (for algorithm designers). *Foundations and Trends® in Theoretical Computer Science*, 11(3–4):217–404, 2016.

[146] R. S. Roy, D. Bhattacharya, and A. Schliep. Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, page btu132, 2014.

[147] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[148] S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors. *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 2017.

[149] K. Salikhov, G. Sacomoto, and G. Kucherov. Using cascading Bloom filters to improve the memory usage for de brujin graphs. In *Algorithms in Bioinformatics*, pages 364–376. Springer, 2013.

[150] L. Salmela and E. Rivals. LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, page btu538, 2014.

[151] L. Salmela, R. Walve, E. Rivals, and E. Ukkonen. Accurate selfcorrection of errors in long reads using de bruijn graphs. *Bioinformatics*, page btw321, 2016.

[152] S. Schechter, C. Herley, and M. Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8. USENIX Association, 2010.

[153] M. H. Schulz, D. R. Zerbino, M. Vingron, and E. Birney. Oases: Robust de novo RNA-seq assembly across the dynamic range of expression levels. *Bioinformatics*, 28(8):1086–1092, 2012.

[154] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[155] B. Solomon and C. Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, advance online publication, Feb 2016. Research.

[156] B. Solomon and C. Kingsford. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. In *International Conference on Research in Computational Molecular Biology*, pages 257–271. Springer, 2017.

[157] B. Solomon and C. Kingsford. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. *Journal of Computational Biology*, 25(7):755–765, 2018.

[158] L. Song, L. Florea, and B. Langmead. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome biology*, 15(11):1, 2014.

[159] M. Steinegger and J. Söding. MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets. *Nature biotechnology*, 2017.

[160] C. Sun, R. S. Harris, R. Chikhi, and P. Medvedev. Allsome sequence bloom trees. In *International Conference on Research in Computational Molecular Biology*, pages 272–286. Springer, 2017.

[161] M. Vallentin. C++ counting Bloom filter. `https://github.com/mavam/libbf`, 2015. [Online; accessed 19-July-2014].

[162] S. Vigna. Broadword implementation of rank/select queries. In *International Workshop on Experimental and Efficient Algorithms*, pages 154–168. Springer, 2008.

[163] S. Vinga and J. Almeida. Alignment-free sequence comparisona review. *Bioinformatics*, 19(4):513–523, 2003.

[164] 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 *Proc. 9th European Conference on Computer Systems*, page 16, 2014.

[165] D. E. Wood and S. L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):1, 2014.

[166] H. Yan, R. Oliveira, K. Burnett, D. Matthews, L. Zhang, and D. Massey. Bgpmon: A real-time, scalable, extensible monitoring system. In *2009 Cybersecurity Applications Technology Conference for Homeland Security*, pages 212–223, March 2009.

[167] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[168] 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.

[169] Z. Zhang and W. Wang. Rna-skim: a rapid method for rna-seq quantification at transcript level. *Bioinformatics*, 30(12):i283–i292, 2014.

[170] Q. G. Zhao, M. Ogihara, H. Wang, and J. J. Xu. Finding global icebergs over distributed data sets. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 298–307. ACM, 2006.

[171] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *International Symposium on Experimental Algorithms*, pages 151–163. Springer, 2013.

[172] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. 6th USENIX Conference on File and Storage Technologies (FAST 08)*, pages 1–14, 2008.

[173] B.-Y. Ziv, J. T.S., K. Ravi, S. D., and T. Luca. Counting distinct elements in a data stream. In *Randomization and Approximation Techniques in Computer Science. RANDOM 2002. Lecture Notes in Computer Science, vol 2483*. Springer, 2002.