Bioinformatics, 34(4), 2018, 568–575
doi: 10.1093/bioinformatics/btx636
Advance Access Publication Date: 9 October 2017
Original Paper

Sequence analysis

Squeakr: an exact and approximate *k*-mer counting system

Prashant Pandey[1,*], Michael A. Bender[1], Rob Johnson[1,2] and Rob Patro[1]

[1]Department of Computer Science, Stony Brook University, Stony Brook, NY 11790, USA and [2]VMware Research, Palo Alto, CA 94304, USA

*To whom correspondence should be addressed.

Associate Editor: Bonnie Berger

Received on March 29, 2017; revised on September 19, 2017; editorial decision on October 3, 2017; accepted on October 6, 2017

Abstract

Motivation: *k*-mer-based algorithms have become increasingly popular in the processing of high-throughput sequencing data. These algorithms span the gamut of the analysis pipeline from *k*-mer counting (e.g. for estimating assembly parameters), to error correction, genome and transcriptome assembly, and even transcript quantification. Yet, these tasks often use very different *k*-mer representations and data structures. In this article, we show how to build a *k*-mer-counting and multiset-representation system using the counting quotient filter, a feature-rich approximate membership query data structure. We introduce the *k*-mer-counting/querying system Squeakr (Simple Quotient filter-based Exact and Approximate Kmer Representation), which is based on the counting quotient filter. This off-the-shelf data structure turns out to be an efficient (approximate or exact) representation for sets or multisets of *k*-mers.

Results: Squeakr takes 2×–4.3× less time than the state-of-the-art to count and perform a random-point-query workload. Squeakr is memory-efficient, consuming 1.5×–4.3× less memory than the state-of-the-art. It offers competitive counting performance. In fact, it is faster for larger *k*-mers, and answers point queries (i.e. queries for the abundance of a particular *k*-mer) over an order-of-magnitude faster than other systems. The Squeakr representation of the *k*-mer multiset turns out to be immediately useful for downstream processing (e.g. de Bruijn graph traversal) because it supports fast queries and dynamic *k*-mer insertion, deletion, and modification.

Availability and implementation: https://github.com/splatlab/squeakr available under BSD 3-Clause License.

Contact: ppandey@cs.stonybrook.edu

Supplementary information: Supplementary data are available at *Bioinformatics online*.

1 Introduction

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 (Bankevich *et al.*, 2012; Grabherr *et al.*, 2011; Pevzner *et al.*, 2001; Schulz *et al.*, 2012; Simpson *et al.*, 2009; Zerbino and Birney, 2008). 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 (Koren *et al.*, 2016).

k-mer-based methods are also heavily used for preprocessing HTS data to perform error correction (Heo *et al.*, 2014; Liu *et al.*, 2013; Song *et al.*, 2014) and digital normalization (Brown *et al.*, 2012; Zhang *et al.*, 2014). Even in long-read ('3rd-generation') sequencing-based assembly, the *k*-mer acts as a building block to help find read overlaps (Berlin *et al.*, 2015; Carvalho *et al.*, 2016) and to perform error correction (Salmela and Rivals, 2014; Salmela *et al.*, 2016).

k-mer-based methods reduce the computational costs associated with many types of HTS analysis. These include transcript

quantification using RNA-seq (Patro *et al.*, 2014; Zhang and Wang, 2014), taxonomic classification of metagenomic reads (Ounit *et al.*, 2015; Wood and Salzberg, 2014), and search-by-sequence over large repositories of HTS-based sequencing experiments (Solomon and Kingsford, 2016).

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 non-trivial 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 (Chikhi and Rizk, 2013), searches (Solomon and Kingsford, 2016), and inner products between datasets (Murray *et al.*, 2016; Vinga and Almeida, 2003).

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.

1.1 k-mer counting systems and approximate membership query 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 (Marçais and Kingsford, 2011), BFCounter (Melsted and Pritchard, 2011), DSK (Rizk *et al.*, 2013), KMC2 (Deorowicz *et al.*, 2015), and Turtle (Roy *et al.*, 2014).

These tools, as well as other sequence-analysis systems (Chikhi and Rizk, 2013), use the Bloom filter (Bloom, 1970) 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 (Marçais and Kingsford, 2011), BFCounter (Melsted and Pritchard, 2011), and Turtle (Roy *et al.*, 2014) 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 look up 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 (Almeida *et al.*, 2007; Bonomi *et al.*, 2006; Canim *et al.*, 2010; Debnath *et al.*, 2011; Fan *et al.*, 2000; Lu *et al.*, 2011; Putze *et al.*, 2007; Qiao *et al.*, 2014). The counting Bloom filter (CBF)Fan *et al.*, (2000) 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, Zhang *et al.*, (2014) demonstrate that the count-min sketch (Cormode and Muthukrishnan, 2005) 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 (Pandey *et al.*, 2017).

There do exist more feature-rich AMQs. In particular, the counting quotient filter (CQF) (Pandey *et al.*, 2017), supports operations such as insertions, deletions, counting (even on skewed datasets), resizing, merging, and highly concurrent accesses. The false-positives in a CQF, similar to a Bloom filter, are always one sided and result in an over estimation of the actual count.

1.2 Results

In this article, we show how to build a k-mer-counting and multiset-representation system using the recently introduced CQF (Pandey *et al.*, 2017), a feature-rich AMQ data structure.

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 CQF 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 (Bloom, 1970), which is commonly used in k-mer-counting applications, Squeakr has the ability to modify and remove k-mers. Unlike the count-min sketch (Cormode and Muthukrishnan, 2005), 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 article; for example it could be instrumental in improving the

sequence Bloom tree structure used for large-scale search (Solomon and Kingsford, 2016).

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 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 traversal workload, Squeakr is 2×–4.3× faster than KMC2.

2 Materials and methods

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

2.1 Counting quotient filter

The CQF implements a counting filter (Pandey *et al.*, 2017) representation of a multiset S by storing a compact, lossless representation of the multiset $h(S)$, where $h : \mathcal{U} \rightarrow \{0, \ldots, 2^p - 1\}$ is a hash function and $\mathcal{U}$ is the universe from which S is drawn. The CQF sets $p = \log_2 \frac{n}{\delta}$ to obtain a false-positive rate δ while handling up to n insertions (Bender *et al.*, 2012).

The CQF saves space by representing some of the bits of $h(x)$ implicitly using a technique called ***quotienting***. The CQF divides $h(x)$ into its first q bits, which we call the ***quotient*** $h_0(x)$, and its remaining $r = p - q$ bits, called the ***remainder*** $h_1(x)$. The CQF 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 CQF 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 CQF uses a variant of linear probing (using few metadata-bits per slot), to find an unused slot where it can store $h_1(x)$ (Pandey *et al.*, 2017).

Instead of storing multiple copies of the same item to count, like a quotient filter, the CQF employs an encoding scheme to count the multiplicity of items. The encoding scheme enables the CQF to maintain *variable-sized* counters. This is achieved by using slots originally reserved to store the remainders to, instead, store count information. The metadata bits maintained by the CQF allow this dynamic reuse of remainder slots for large counters while still ensuring the correctness of all CQF operations.

The variable-sized counters in the CQF enable the data structure to handle highly skewed datasets efficiently. By reusing the allocated space, the CQF avoids wasting extra space on counters and naturally and dynamically adapts to the frequency distribution of the input data. The CQF never takes more space than a quotient filter for storing the same multiset. For highly skewed distributions, like those observed in HTS-based datasets, it occupies only a small fraction of the space that would be required by a comparable (in terms of false-positive rate) quotient filter.

2.2 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 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 AMQ data structure to maintain all the singletons and a hash table (Marçais and Kingsford, 2011) or compaction-and-sort based data structure (Roy *et al.*, 2014) 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; Roberts *et al.*, 2004) to achieve faster performance (Deorowicz *et al.*, 2015; Roberts *et al.*, 2004).

Squeakr uses a single, off-the-shelf data structure, the CQF, 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 dataset. Each thread performs the same set of operations; reading data from disk, parsing and extracting k-mers, and inserting k-mers in the CQF. 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* CQF 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 (Threads actually read slightly <16 MB, since threads always break chunks at inter-read boundaries.), 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 CQF (Pandey *et al.*, 2017) for synchronizing insert operations among multiple threads. Using a thread-safe CQF, 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 CQF) 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 CQF 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. (For the *G. gallus* dataset, though only 0.000561% of attempts did not obtain the lock in the first try the average and maximum spinning times were quite high, 2.137 ms and 3.072 s, respectively.) This is due, in part, to the fact that these data have many highly repetitive k-mers, causing multiple threads to attempt

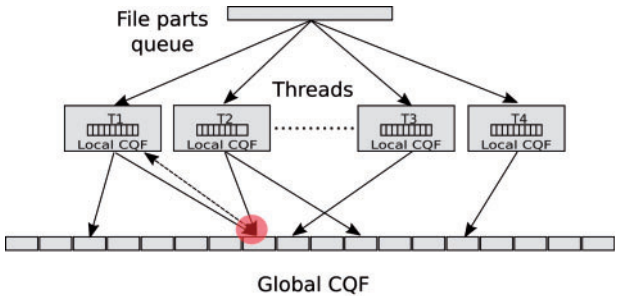

Fig. 1. Squeakr system design: each thread has a local CQF and there is a global CQF. The dotted arrow shows that one thread did not get the lock in the first attempt and had to insert the item in the local CQF

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.

2.2.1 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 CQF. 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 (Pandey *et al.*, 2017), the time spent by threads while waiting for a lock can be utilized to do local work. As shown in Figure 1, we assign a local CQF to each thread. Now, during insertion, each thread first tries to insert the k-mer in the thread-safe, global CQF. 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 CQF and continues. Once the local CQF becomes full, the thread dumps the k-mers present in the local CQF into the global CQF 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 CQF. When a thread encounters such a k-mer and fails to obtain the corresponding lock in the global CQF, the thread instead immediately inserts those k-mers in the local (lockless) CQF and continues processing data. Moreover, these repetitive k-mers are first counted in the local CQF before being inserted with their corresponding counts in the global CQF. Under this design, instead of inserting multiple instances of the k-mer in the global CQF, requiring multiple acquisitions of a global CQF lock, Squeakr only insert the k-mers a few times with their counts aggregated via the local CQF s.

In Squeakr even while maintaining multiple data structures, a local CQF per thread and a global CQF, 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 CQF 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 CQF and only insert them into the global CQF once the local CQF gets full.

2.3 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 2bits. As explained in Section 2.1, to achieve a maximum allowable false-positive rate δ the CQF 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 (Appleby, 2016) (https://sites.google.com/site/murmurhash/), by default, for hashing k-mers.

In the CQF, the p-bit hash is divided into q quotient bits and r remainder bits. The maximum false-positive rate is bounded by 2^{-r} (Bender *et al.*, 2012). In Squeakr, we assign $q = \log n$, where n is the number of distinct k-mers in the dataset and we use 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 (Li, 2016) (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=26bits. In addition, the CQF uses 2.125 metadata bits per item, see Pandey *et al.*, (2017). This is still far less than the 56bits that would be required to store each k-mer key explicitly.

3 Results

In this section, we evaluate our implementations of Squeakr and Squeakr-exact. Squeakr-exact, as described in Section 2.3, is an exact k-mer counting system that uses the CQF 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 CQF 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 (Danek, 2016) and Jellyfish2 (Marçais and Kingsford, 2011). KMC2 is currently the fastest k-mer counting system (Deorowicz *et al.*, 2015), 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 (Zhang *et al.*, 2014) is the only approximate multiset representation and uses a count-min sketch. Here, we do not 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 CQF.

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 (Pell *et al.*, 2012). To evaluate the ability of the CQF 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 (Vinga and Almeida, 2003). Recently, (Murray *et al.*, 2016) 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.

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 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 CQF 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.

3.1.1 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 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.

Table 1. datasets used in the experiments

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

Note: The file size is in GB. All the datasets are compressed with gzip compression.

Table 2. Gigabytes of RAM used by KMC2, Squeakr, Squeakr-exact, and Jellyfish2 for various datasets for in-memory experiments for k=28

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

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 usedMohamadi *et al.*, (2017) to estimate the number of distinct k-mers in datasets. As explained in Section 2.2, 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.

3.1.2 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*. In addition, 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 Pell *et al.*, (2012). 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 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. In addition,

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

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

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 MashOndov *et al.*, (2016). 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 CQFs. We report the time taken to compute the intersection and write the output to disk.

3.2 Memory requirement

Table 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 ≈30GB of RAM. Across all datasets, Squeakr takes 1.5×–4.3× 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 approximately the same amount of RAM as Jellyfish2.

3.3 Counting performance

Tables 3 and 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 >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.

3.4 Query performance

3.4.1 Random query for existing k-mers

Table 5 shows the random query performance for existing k-mers. Squeakr is 3.2×–4.9× 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.

3.4.2 Random query for uniformly random k-mers

Table 5 shows the random query performance for uniformly random k-mers. For uniformly random k-mers, Squeakr is 8.9×–10.8× 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 CQF. For non-existing items, the CQF 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 CQF 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 3.1, the theoretical false-positive rate is 1/256, i.e. 00390625. The empirical false-positive rate reported during the

Table 5. Random query performance of KMC2, Squeakr, Squeakr-exact, and Jellyfish2 on two different datasets for k=28

System	*G.gallus*		*M.balbisiana*	
	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

benchmark is 0.0012414. The false-positive rate in Squeakr is uniformly distributed across k-mers irrespective of their abundance.

3.4.3 de Bruijn graph traversal

Table 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 2.7×–6.7× faster than KMC2 for de Bruijn graph traversal queries. For the de Bruijn graph traversal, we perform four queries for each k-mer. To continue on the path, only one out of the four queries should return true. In the whole benchmark, ≈75% of the queries are false queries.

Table 6 also reports the longest path present in the graph reported by both the 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 1.7×–6.4× faster than KMC2 in terms of total time.

3.5 Inner-product queries

For inner-product queries, we first counted k-mers from two different datasets, each having ≈975 Million k-mer instances and ≈91 Million and ≈136 Million distinct k-mers respectively and stored the output on disk. It took 349.46s 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 329.65s 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 4.29s using six threads. While Squeakr took 4.57s using only three threads. This shows that Squeakr is faster than KMC2 for workloads comparing two datasets.

4 Conclusion

We demonstrate that the CQF can serve as a memory-efficient, fast, and feature-rich representation of k-mer multisets by building a CQF-based k-mer counting system, Squeakr. Despite its straightforward use of an off-the-shelf data structure, Squeakr offers great counting performance and exceptional query performance.

Table 6. de Bruijn graph query performance on different datasets

System	Dataset	Max path len	Running times		
			Counting	Query	Total
KMC2	*G.gallus*	122	266	23 097	23 363
Squeakr	*G.gallus*	92	412	3415	3827
KMC2	*M.balbisiana*	123	607	6817	7424
Squeakr	*M.balbisiana*	123	662	1471	2133

Note: 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).

Squeakr is much more space efficient than other k-mer-counting solutions (even when representing the k-mer multiset exactly), and scales with multiple threads. Squeakr's k-mer representation can be further used to store and traverse the weighted de Bruijn graph over the k-mers with high query throughput, an order-of-magnitude faster than other systems like KMC2 and Jellyfish2. Furthermore, Squeakr is dynamic, since k-mers can be added or removed, and their counts can be updated.

We believe that these properties make Squeakr's k-mer representation a strong candidate for many downstream analysis tasks. For example, the fast query performance and ability to accurately record k-mer counts makes the CQF an enticing candidate atop which to build a de Bruijn graph-based transcriptome assembler. We anticipate that, with some domain specific optimizations, Squeakr's CQF can be made even more memory efficient and can be used to efficiently store an exact weighted de Bruijn graph.

Funding

We gratefully acknowledge support from National Science Foundation grants [grant numbers BBSRC-NSF/BIO-1564917, IIS-1247726, IIS-1251137, CNS-1408695, CCF-1439084, and CCF-1617618], and Sandia National Laboratories.

Conflict of Interest: none declared.

References

Almeida,P.S. *et al.* (2007) Scalable Bloom filters. *J. Inform. Proc. Lett.*, **101**, 255–261.

Appleby,A. (2016) Murmurhash. (Online; 19 July 2016, last date accessed).

Bankevich,A. *et al.* (2012) SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *J. Comput. Biol.*, **19**, 455–477.

Bender,M.A. *et al.* (2012) Don't thrash: how to cache your hash on flash. *Proc. VLDB Endowment*, **5**, 1627.

Berlin,K. *et al.* (2015) Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat. Biotechnol.*, **33**, 623–630.

Bloom,B.H. (1970) Spacetime trade-offs in hash coding with allowable errors. *Commun. ACM*, **13**, 422–426.

Bonomi,F. *et al.* (2006) An improved construction for counting Bloom filters. *ECA'06*. In *14th Annual European Symposium on Algorithms, LNCS 4168*, 2006, pp. 684–695.

Brown,C.T. *et al.* (2012) A reference-free algorithm for computational normalization of shotgun sequencing data. *arXiv preprint arXiv: 1203.4802*.

Canim,M. *et al.* (2010) Buffered Bloom filters on solid state storage. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pp. 1–8.

Carvalho,A.B. *et al*. (2016) Improved assembly of noisy long reads by k-mer validation. *Genome Res*., **26**, 1710–1720.

Chikhi,R. and Rizk,G. (2013) Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms Mol. Biol*., **8**, 1.

Cormode,G. and Muthukrishnan,S. (2005) An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, **55**, 58–75.

Danek,A. (2016) Kmc2 github. (Online; 29 April 2016, last date accessed).

Debnath,B. *et al*. (2011) BloomFlash: Bloom filter on flash-based storage. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS), 2011*, pp. 635–644.

Deorowicz,S. *et al*. (2015) Kmc 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, **31**, 1569–1576.

Fan,L. *et al*. (2000) Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM T. Netw*., **8**, 281–293.

Grabherr,M.G. *et al*. (2011) Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nat. Biotechnol*., **29**, 644–652.

Heo,Y. *et al*. (2014) BLESS: bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, **30**, 1354–1362.

Koren,S. *et al*. (2016) Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *bioRxiv*, 071282.

Li,H. (2016) Inthash. (Online; 19 July 2016, last date accessed).

Liu,Y. *et al*. (2013) Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, **29**, 308–315.

Lu,G. *et al*. (2011) A forest-structured Bloom filter with flash memory. In *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies (MSST), 2011*, pp. 1–6.

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

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

Mohamadi,H. *et al*. (2017) ntcard: A streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, btw832.

Murray,K.D. *et al*. (2016) kwip: The k-mer weighted inner product, a de novo estimator of genetic similarity. *bioRxiv*, 075481.

Ondov,B.D. *et al*. (2016) Mash: fast genome and metagenome distance estimation using minhash. *Genome Biol*., **17**, 132.

Ounit,R. *et al*. (2015) Clark: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC Genomics*, **16**, 1.

Pandey,P. *et al*. (2017) A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 775–787. ACM.

Patro,R. *et al*. (2014) Sailfish enables alignment-free isoform quantification from RNA-Seq reads using lightweight algorithms. *Nat. Biotechnol*., **32**, 462–464.

Pell,J. *et al*. (2012) Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proc. Natl. Acad. Sci. USA*, **109**, 13272–13277.

Pevzner,P.A. *et al*. (2001) An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA*, **98**, 9748–9753.

Putze,F. *et al*. (2007) Cache-, hash-and space-efficient Bloom filters. In *Proceedings 6th International Conference on Experimental Algorithms*. Springer-Verlag, Berlin, Heidelberg, pp. 108–121.

Qiao,Y. *et al*. (2014) Fast Bloom filters and their generalization. *IEEE Trans. Parallel Distributed Syst*., **25**, 93–103.

Rizk,G. *et al*. (2013) DSK: k-mer counting with very low memory usage. *Bioinformatics*, **29**, 652–653.

Roberts,M. *et al*. (2004) Reducing storage requirements for biological sequence comparison. *Bioinformatics*, **20**, 3363–3369.

Roy,R.S. *et al*. (2014) Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, **30**, 1950–1957.

Salmela,L. and Rivals,E. (2014) LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, **30**, 3506–3514.

Salmela,L. *et al*. (2016) Accurate selfcorrection of errors in long reads using de Bruijn graphs. *Bioinformatics*, **33**, 799–806.

Schulz,M.H. *et al*. (2012) Oases: robust de novo RNA-Seq assembly across the dynamic range of expression levels. *Bioinformatics*, **28**, 1086–1092.

Simpson,J.T. *et al*. (2009) Abyss: a parallel assembler for short read sequence data. *Genome Res*., **19**, 1117–1123.

Solomon,B. and Kingsford,C. (2016) Fast search of thousands of short-read sequencing experiments. *Nat. Biotechnol*., **34**, 300.

Song,L. *et al*. (2014) Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biol*., **15**, 1.

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

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

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

Zhang,Q. *et al*. (2014) These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PLoS One*, **9**, e101271.

Zhang,Z. and Wang,W. (2014) RNA-Skim: a rapid method for RNA-Seq quantification at transcript level. *Bioinformatics*, **30**, i283–i292.