Efficient, approximate representations of sets (k-mer counting and the de Bruijn graph)

Scalability at the forefront

I've spoken a good bit, already, in this class about the need for scalable solutions, but how big of a problem is it?

Take (one of) the simplest problems you might imagine:

Given: A collection of sequencing reads S and a paramater k

Find: The multiplicity of every length-k substring (k-mer) that appears in S

This is the *k-mer counting* problem

k-mer counting

A large number of recent papers tackle this (or a closely related) problem

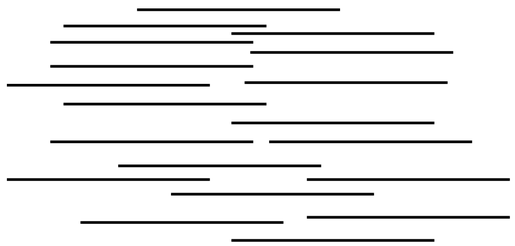
Tallymer, Jellyfish, DSK, KMC{1,2,3}, BFCounter, scTurtle, Girbil, KAnalyze, khmer, … and many more

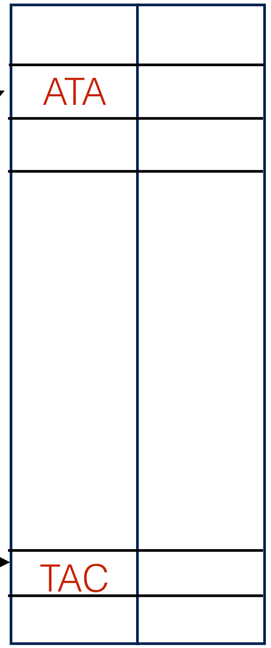
How might we count k-mers

A naive approach:

S

ATACAGGACGTTC





While S is non-empty:

Draw a string s, from S

For every k-mer, k in s:

counts[k] += 1

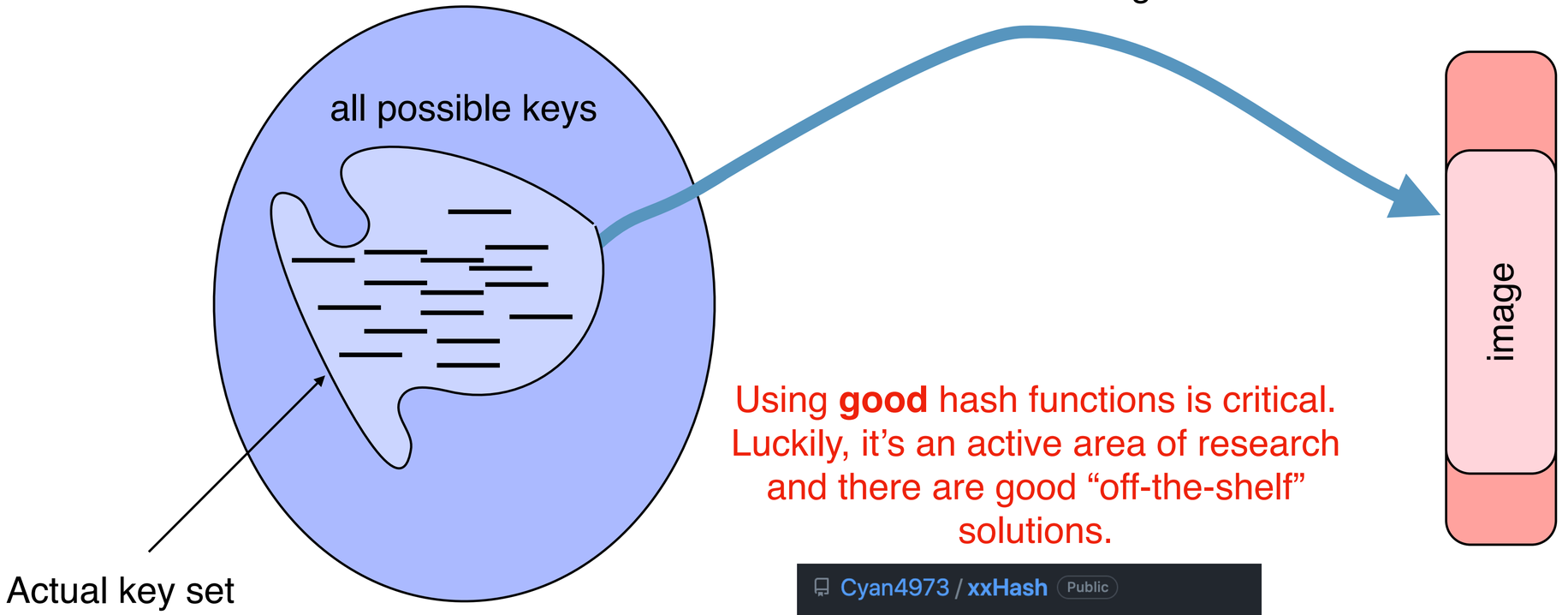
Tiny refresher on hashing

Hash function h : k →[0, R]

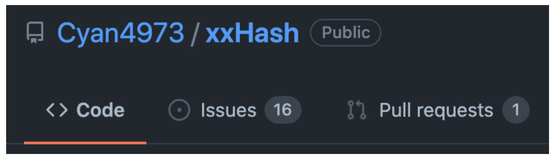
Domain (e.g. keys)

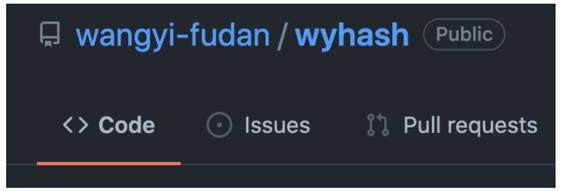
maps elements from the domain into the range

Range

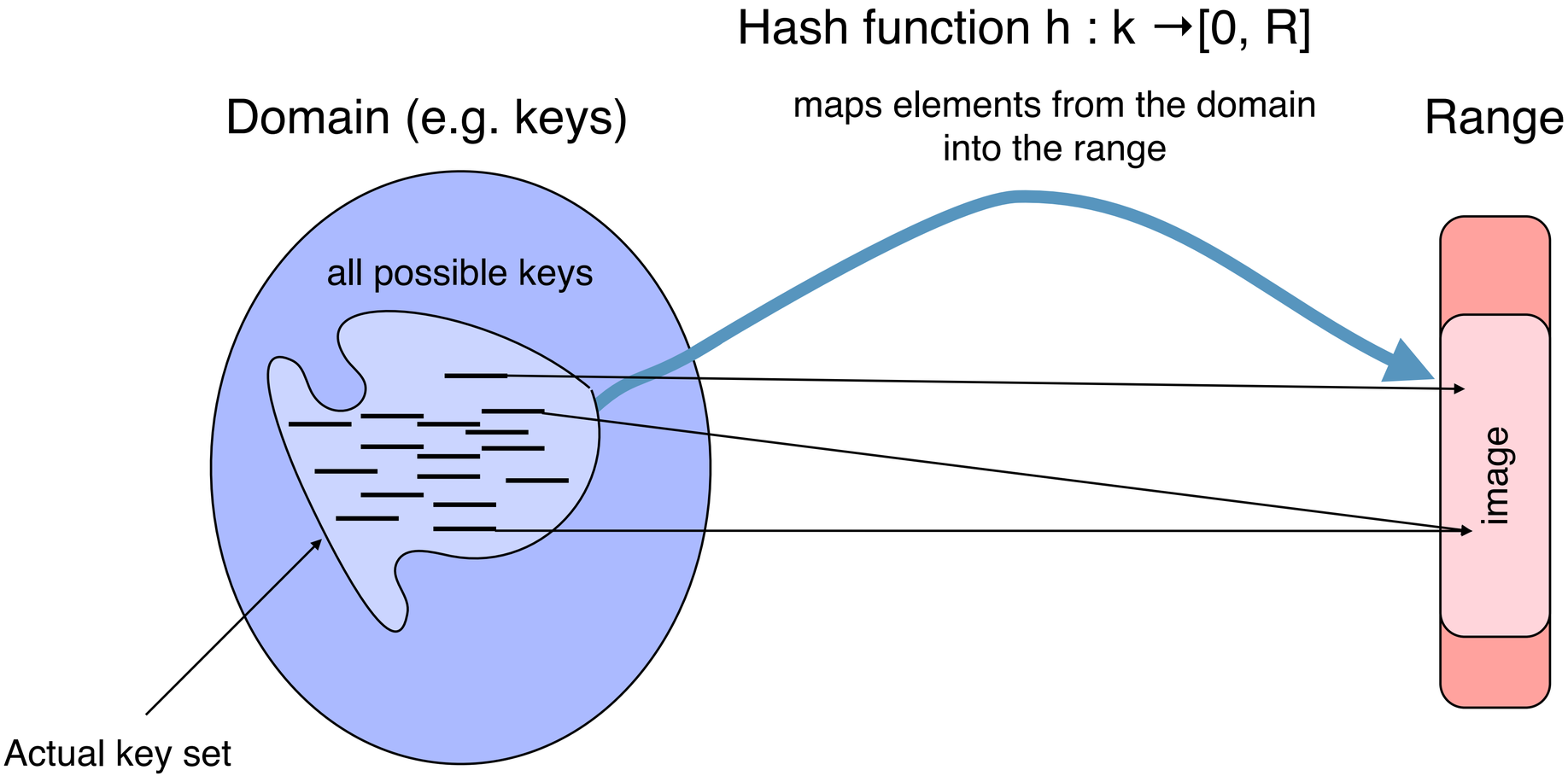


Using **good** hash functions is critical. Luckily, it's an active area of research and there are good "off-the-shelf" solutions.





Tiny refresher on hashing

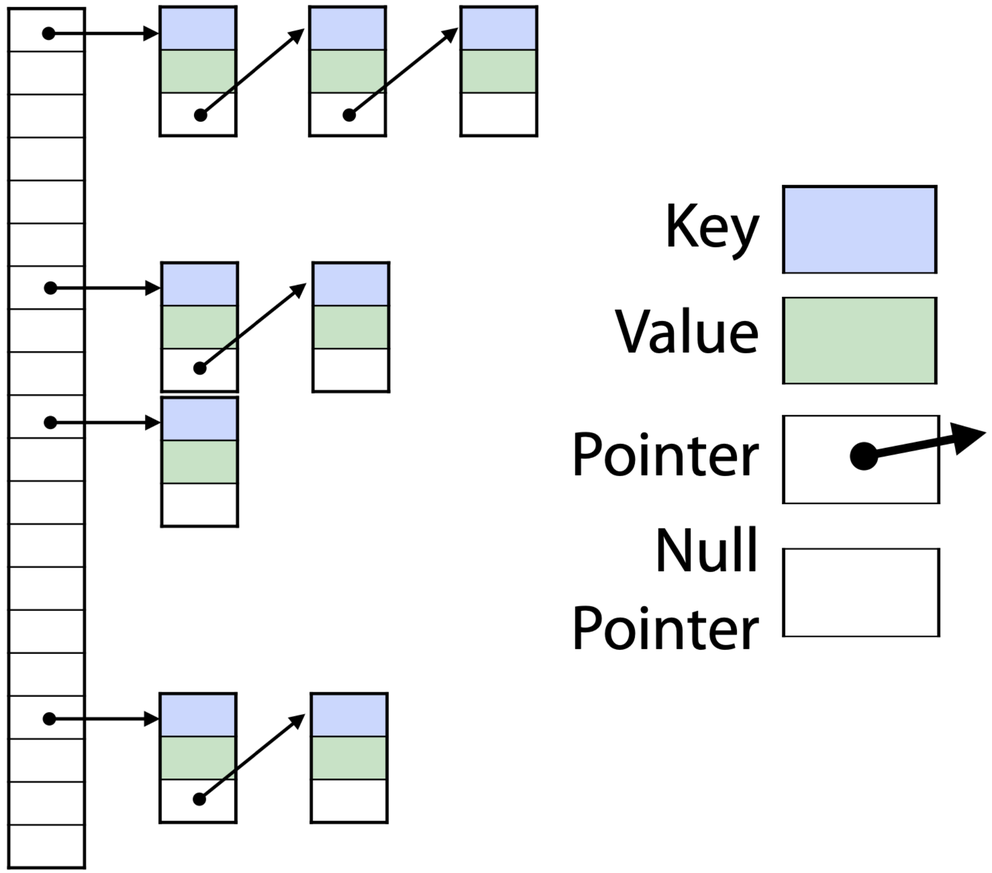


Since, in general, R << |k|, **collisions** are inevitable. That is, there can exist distinct keys k_1, k_2 such that $k_1 \neq k_2$, but $h(k_1) = h(k_2)$.

There are many different ways to deal with these collisions, including chaining and probing — refresh yourself on these, we won't discuss these in detail here.

Collision Resolution (e.g. separate chaining)

"*Hashing with chaining*" or "*chain hashing*"



Note: Since the hash is a lossy "fingerprint" of the key**,** we generally have to store the key itself as well.

To look up the value associated with an element k, we compute h(k) to find the "cell" in the hash table to which k maps, then walk the list to check if any of the keys we see are = k; if so, the associated value is k's, else, k has not been added to the table.

In reality, separate chaining is quite expensive in terms of both memory and lookup time, so other "probing" strategies are more popular common.

Bonus nugget: Because of requirements of behavior specified in the language standard, the standard library hash table in C++ essentially *has* to use chaining. Therefore, if you want a high-perf. Hash table in C++, use a different library!

Taken from Ben Langmead's slides @ https://www.cs.jhu.edu/~langmea/resources/lecture_notes/105_hash_and_prob_pub.pdf

What's wrong with this approach?

Speed & Memory usage

Routinely encounter datasets with 10 - 100 x 10^9 nucleotides

Just hashing the k-mers and resolving collisions takes time

How should we represent the keys (k-mers)?

Naive string representation requires at least k bytes / key!

At ~3,000,000,000 k-mers in the human genome with k=31
we are already looking at 93,000,000,000 bytes!

Since Σ is small, we generally consider more efficient encodings are common (e.g. 2 bits / nucleotide; the best encoding isn't obvious though (see e.g. https://github.com/COMBINE-lab/kmers/issues/3)

What's wrong with this approach?

Speed & Memory usage

How many distinct k-mers should we expect?

If just looking at a reference genome G, and k is large, can expect O(G) k-mers (most k-mers are unique, but total is bounded by reference size).

What if we are looking at a read set consisting of N reads, each of length ℓ?

Each read contains ℓ - k + 1 k-mers, but not all are distinct. How many are distinct?

Eventually, we will see all O(G) k-mers from the reference, but if we have an error rate ϵ, then we will continue to see new k-mers at a rate proportional to ϵ; why?

Can expect an error every $1/\epsilon$ -th sequenced nucleotide, and (ignoring edge effects) each such error will affect k different k-mers.

What's wrong with this approach?

Speed & Memory usage

Commonly on the order of 1-10 x 10^9 or more distinct k-mers

What about storing the *values* (i.e. counts)?

If we used a 4-byte unsigned int to store the count, we'd be using 40GB just for counts

Also, hash tables have overhead (load factor < 1), and often need to store the *key* as well as the *value*

Can easily get to > 100GB of RAM

Bloom Filters

Originally designed to answer *probabilistic* membership queries:

Is element e in my set S?

If yes, **always** say yes

If no, say no **with large probability**

False positives can happen; false negatives cannot.

Bloom Filters

For a set of size N, store an array of M bits

Use k different hash functions, $\{h_0, \ldots, h_{k-1}\}$

To insert e, set $A[h_i(e)] = 1$ for $0 < i < k$

To query for e, check if $A[h_i(e)] = 1$ for $0 < i < k$

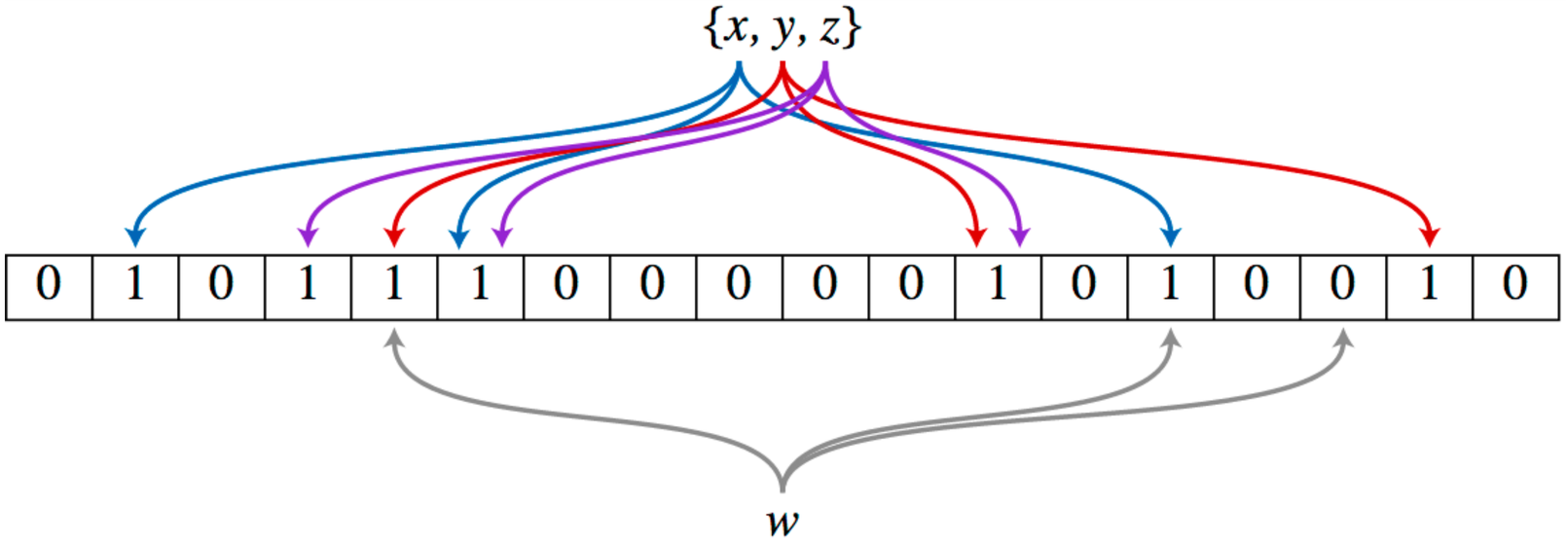


Image by David Eppstein - self-made, originally for a talk at WADS 2007

Bloom Filters

If hash functions are good and sufficiently independent,
then the probability of false positives is
low and controllable.

How low?

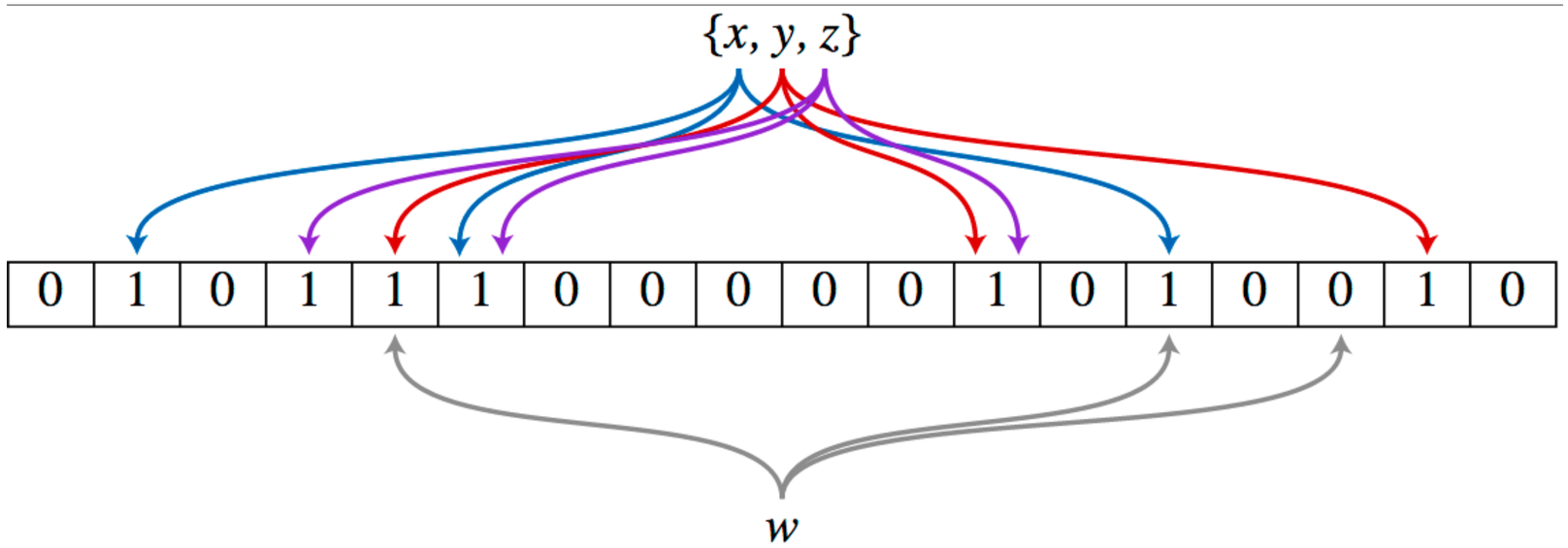


Image by David Eppstein - self-made, originally for a talk at WADS 2007

False Positives

Let q be the fraction of the m-bits which remain as 0 after n insertions.

The probability that a randomly chosen bit is 1 is 1-q.

But we need a 1 in the position returned by k different hash functions; the
of this is $(1-q)^k$

We can derive a formula for the expected value of q,
for a filter of m bits, after n insertions with k different hash functions:

$$E[q] = (1 - 1/m)^{kn}$$

*analysis of Mitzenmacher and Upfal

False Positives

Mitzenmacher & Unfal used the Azuma-Hoeffding inequaltiy to prove (without assuming the probability of setting each bit is independent) that

$$\Pr(|q - E[q]| \geq \frac{\lambda}{m}) \leq 2\exp(-2\frac{\lambda^2}{m})$$

That is, the random realizations of q are highly concentrated around E[q], which yields a false positive prob of:

$$\sum_t \Pr(q = t)(1-t)^k \approx (1 - E[q])^k = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx (1 - e^{-\frac{kn}{m}})^k$$

*analysis of Mitzenmacher and Upfal

False Positives

$$\sum_t \Pr(q=t)(1-t)^k \approx (1-E[q])^k = \left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^k \approx (1-e^{-\frac{kn}{m}})^k$$

This lets us choose optimal values to achieve a target false positive rate. For example, assume m & n are given. Then we can derive the optimal k

$$k = (m/n) \ln 2 \Rightarrow 2^{-k} \approx 0.6185^{\,m/n}$$

We can then compute the false positive prob

$$p = (1-e^{-(\frac{m}{n}\ln 2)\frac{n}{m}})^{(\frac{m}{n}\ln 2)} \implies$$

$$\ln p = -\frac{m}{n}(\ln 2)^2 \implies$$

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

False Positives

$$\sum_t \Pr(q=t)(1-t)^k \approx (1-E[q])^k = \left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^k \approx (1-e^{-\frac{kn}{m}})^k$$

This lets us choose optimal values to achieve a target false positive rate. For example, assume m & n are given. Then we can derive the optimal k

$$k = (m/n) \ln 2 \Rightarrow 2^{-k} \approx 0.6185^{\,m/n}$$

We can then compute the false positive prob

$$p = (1-e^{-(\frac{m}{n}\ln 2)\frac{n}{m}})^{(\frac{m}{n}\ln 2)} \Longrightarrow$$

$$\ln p = -\frac{m}{n}(\ln 2)^2 \Longrightarrow$$

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

given an expected # elems

and a desired false positive rate

we can compute the optimal size and # of has functions

Bloom Filters & De Bruijn Graphs

How could the Bloom filter be useful for representing a De Bruijn graph?

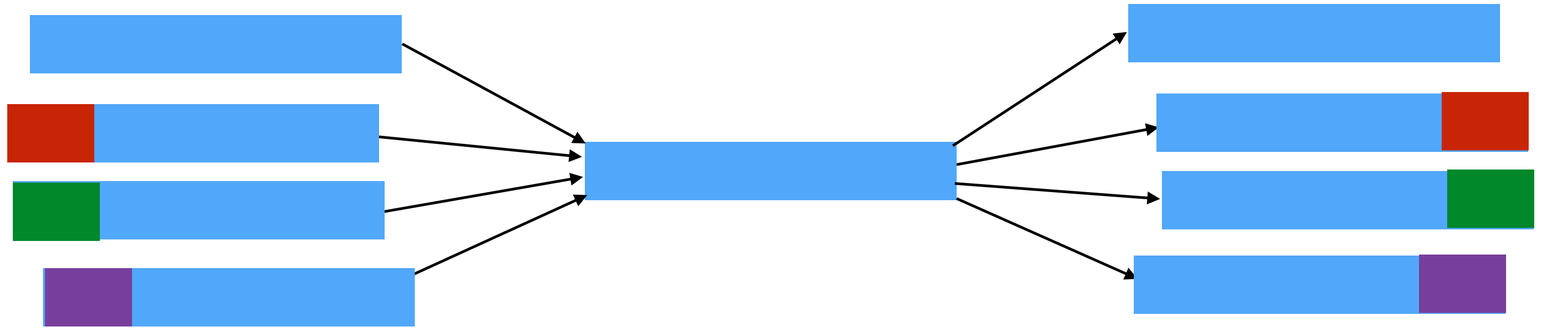
Say we have a bloom filter B, for all of the k-mers in our data set, and say I give you one k-mer that is truly present.

Remember this term, it will come up in a couple of slides

We now have a "*navigational*" representation of the De Bruijn graph (can return the set of neighbors of a node, but not select/iterate over nodes); why?

Detour: Bloom Filters & De Bruijn Graphs

How could this data structure be useful for representing a De Bruijn graph?



A given (k-1)-mer can only have 2*|Σ| neighbors; |Σ| incoming and |Σ| outgoing neighbors — for genomes |Σ| = 4

To navigate in the De Bruijn graph, we can simply query all possible successors, and see which are actually present.

Bloom Filters & De Bruijn Graphs

But, a Bloom filter still has false-positives, right?

May return some neighbors that are not actually present.

Pell et al., PNAS 2012, use a lossy Bloom filter directly

Chikhi & Rizk, WABI 2012, present a *lossless* data structure based on Bloom filters

Salikhov et al., WABI 2013 extend this work and introduce the concept of “cascading” Bloom filters

Pellow, Filippova & Kingsford, RECOMB 2016. Take advantage of “independence” of false positives to lower FP rate for Bloom Filter representations

First, some bounds

JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 22, Number 5, 2015

Pp. 336–352
DOI: 10.1089/cmb.2014.0160

Research Articles

On the Representation of De Bruijn Graphs

RAYAN CHIKHI,[1,6] ANTOINE LIMASSET,[3] SHAUN JACKMAN,[4]
JARED T. SIMPSON,[5] and PAUL MEDVEDEV[1,2,6]

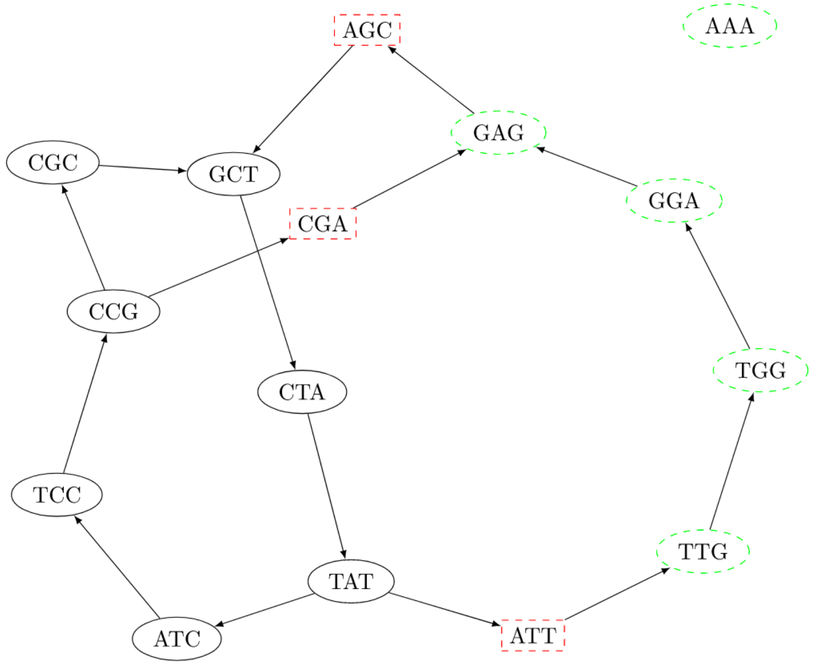
We use the term *membership data structure* to refer to a way of representing a dBG and answering k-mer membership queries. We can view this as a pair of algorithms: (CONST, MEMB). The CONST algorithm takes a set of k-mers S (i.e., a dBG) and outputs a bit string. We call CONST a constructor, since it constructs a representation of a dBG. The MEMB algorithm takes as input a bit string and a k-mer x and outputs true or false. Intuitively, MEMB takes a representation of a dBG created by CONST and outputs whether a given k-mer is present. Formally, we require that for all $x \in \Sigma^k$, MEMB(CONST(S), x) is true if and only if $x \in S$.

An NDS is a pair of algorithms, CONST and NBR. As before, CONST takes a set of k-mers and outputs a bit string. NBR takes a bit string and a k-mer and outputs a set of k-mers. The algorithms must satisfy that for every dBG S and a k-mer $x \in S$, NBR(CONST(S), x) = $ext(x) \cap S$. Note that if $x \notin S$, then the behavior of NBR(CONST(S), x) is undefined. We observe that a membership data structure immediately implies an NDS because an NBR query can be reduced to eight MEMB queries.

In this section, we prove that a navigational data structure on de Bruijn graphs needs at least 3.24 bits per k-mer to represent the graph:

Theorem 1. *Consider an arbitrary NDS and let* CONST *be its constructor. For any* $0 < \epsilon < 1$, *there exists a* k *and* $x \subseteq \Sigma^k$ *such that* $|\text{CONST}(x)| \geq |x| \cdot (c - \epsilon)$, *where* $c = 8 - 3 \lg 3 \approx 3.25$.

Critical False Positives



(a)

$a_1 \ldots a_k$	$\sum_{i=1}^{k} a_i^i \bmod 10$
ATC	0
CCG	0
TCC	5
CGC	6
...	...

(b)

Bloom filter
1
0
0
0
0
1
1
0
0
0

(c)

Nodes self-information:

$$\lceil log_2 \binom{4^3}{7} \rceil = 30 \text{ bits}$$

Structure size:

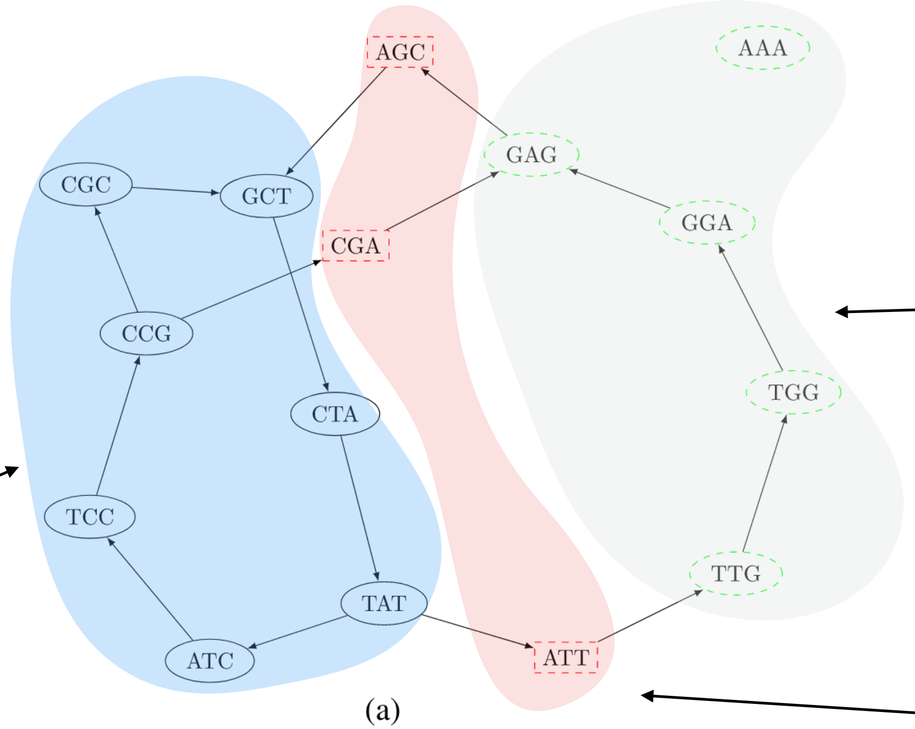
$$\underbrace{10}_{\text{Bloom}} + \underbrace{3 \cdot 6}_{\text{False positives}} = 28 \text{ bits}$$

(d)

“toy” hash func (not used in practice)

Chikhi & Rizk

Critical False Positives



(a)

encoded in BF, but not "reachable"

"true" k-mers

critical FP

$a_1 \ldots a_k$	$\sum_{i=1}^{k} a_i^i \bmod 10$
ATC	0
CCG	0
TCC	5
CGC	6
...	...

(b)

Bloom filter
1
0
0
0
0
1
1
0
0
0

(c)

Nodes self-information:

$$\lceil log_2 \binom{4^3}{7} \rceil = 30 \text{ bits}$$

Structure size:

$$\underbrace{10}_{\text{Bloom}} + \underbrace{3 \cdot 6}_{\text{False positives}} = 28 \text{ bits}$$

(d)

"toy" hash func (not used in practice)

Chikhi & Rizk

The quotient filter for exact & approximate counting

A General-Purpose Counting Filter: Making Every Bit Count

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

Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

OXFORD

Genome 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
[2]VMware Research, 3425 Hillview Ave, Palo Alto, CA 94304, USA

The Counting Quotient Filter

Compact, lossless representation of multiset h(S)

$h : U \rightarrow \{0,\ldots,2^p-1\}$ is a hash function, S is multiset, U is the universe from which S is drawn

$x \in S$, h(x) is a p-bit number.

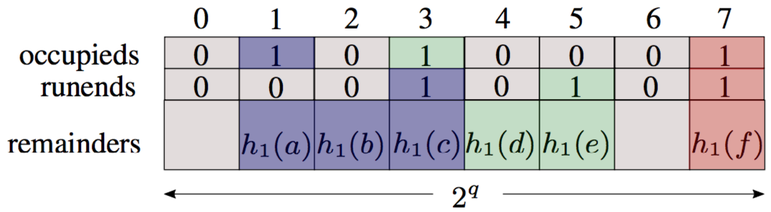
Q is an array of 2^q r-bit slots

The quotient filter divides h(x) into q(h(x)), r(h(x)); the first q and remaining r bits of h(x) where p=q+r

Put r(h(x)) into Q[q(h(x))]

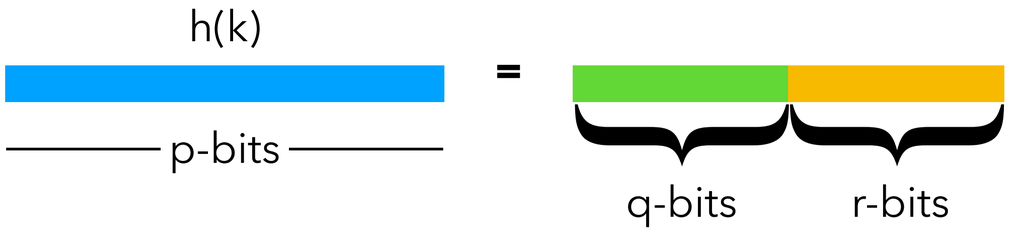
The Counting Quotient Filter (CQF)

Approximate *Multiset* Representation



Works based on quotienting* & fingerprinting keys

Let k be a key and h(k) a p-bit hash value



Clever encoding allows low-overhead storage of element counts
(use *key* slots to store *values* in base 2^r-1; smaller values ⇒ fewer bits)

Careful engineering & use of efficient rank & select to resolve collisions leads to a **fast**, **cache-friendly** data structure

* Idea goes back at least to Knuth (TACOP vol 3)

The Counting Quotient Filter (CQF)

Approximate *Multiset* Representation

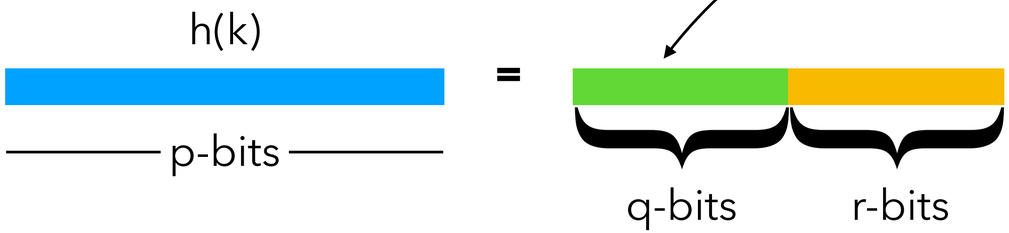
	0	1	2	3	4	5	6	7
occupieds	0	1	0	1	0	0	0	1
runends	0	0	0	1	0	1	0	1
remainders		$h_1(a)$	$h_1(b)$	$h_1(c)$	$h_1(d)$	$h_1(e)$		$h_1(f)$

2^q

Works based on quotienting* & fingerprinting keys

Let k be a key and h(k) a p-bit hash value

Determines position in array of size 2^q r-bit slots



Clever encoding allows low-overhead storage of element counts
(use *key* slots to store *values* in base 2^r-1; smaller values ⇒ fewer bits)

Careful engineering & use of efficient rank & select to resolve collisions leads to a **fast**, **cache-friendly** data structure

* Idea goes back at least to Knuth (TACOP vol 3)

The Counting Quotient Filter (CQF)

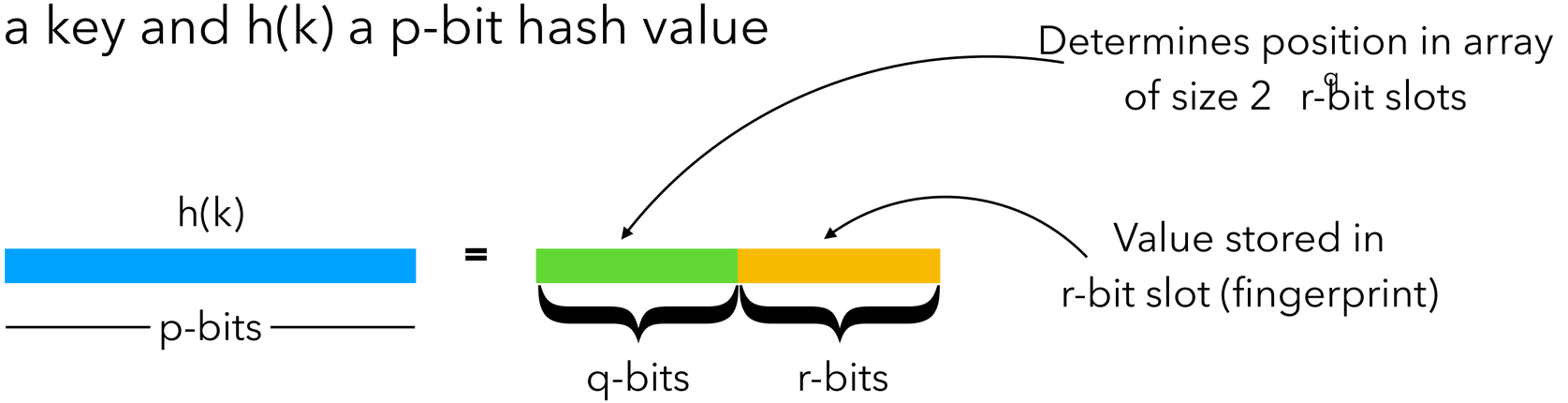
Approximate *Multiset* Representation

	0	1	2	3	4	5	6	7
occupieds	0	1	0	1	0	0	0	1
runends	0	0	0	1	0	1	0	1
remainders		$h_1(a)$	$h_1(b)$	$h_1(c)$	$h_1(d)$	$h_1(e)$		$h_1(f)$

2^q

Works based on quotienting* & fingerprinting keys

Let k be a key and h(k) a p-bit hash value



Clever encoding allows low-overhead storage of element counts (use *key* slots to store *values* in base 2^r-1; smaller values ⇒ fewer bits)

Careful engineering & use of efficient rank & select to resolve collisions leads to a **fast**, **cache-friendly** data structure

* Idea goes back at least to Knuth (TACOP vol 3)

The Counting Quotient Filter

In reality, a bit more complicated because collisions can occur. What if Q[q(h(x))] is occupied by some other element (as the result of an earlier collision)?

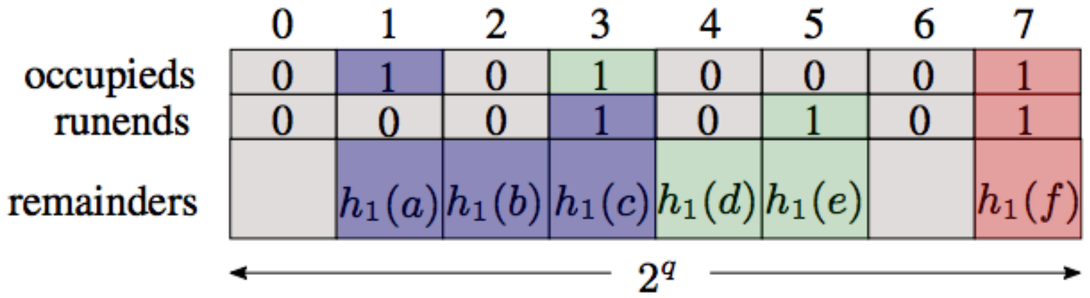


Figure 1: A simple rank-and-select-based quotient filter. The colors are used to group slots that belong to the same run, along with the runends bit that marks the end of that run and the occupieds bit that indicates the home slot for remainders in that run.

Move along until you find the next free slot.
Metadata bits allow us to track “runs” and skip elements other than the key of interest efficiently.

The Counting Quotient Filter

How to count?

Rather than having a separate array for counting (*a la* the counting Bloom filter), use the slots of Q directly to encode either r(h(x)), or counts!

The CQF uses a somewhat complex encoding scheme (base 2^r-2), but this allows arbitrary variable length counters.

This is a **huge** win for highly-skewed datasets with non-uniform counts (like most of those we encoutner).

The Counting Quotient Filter, results

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	$\frac{2.125+\log_2 1/\delta}{\alpha}$

false pos. rate

load factor

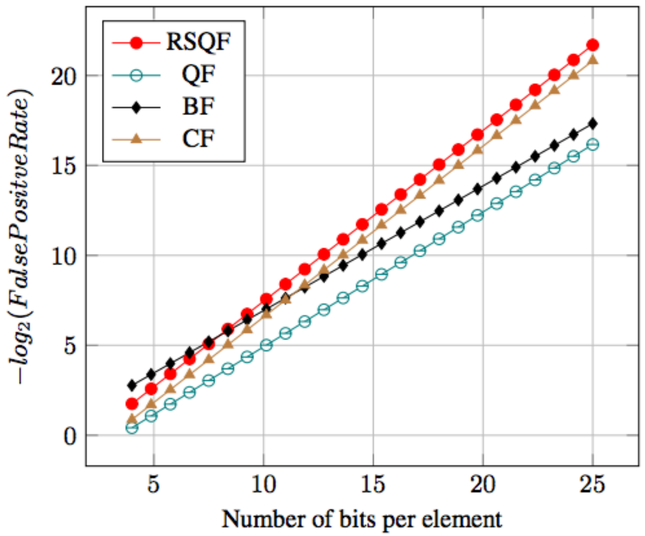


Figure 4: Number of bits per element for the RSQF, QF, BF, and CF. The RSQF requires less space than the CF amd less space than the BF for any false-positive rate less than 1/64. (Higher is better)

The Counting Quotient Filter, results

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

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

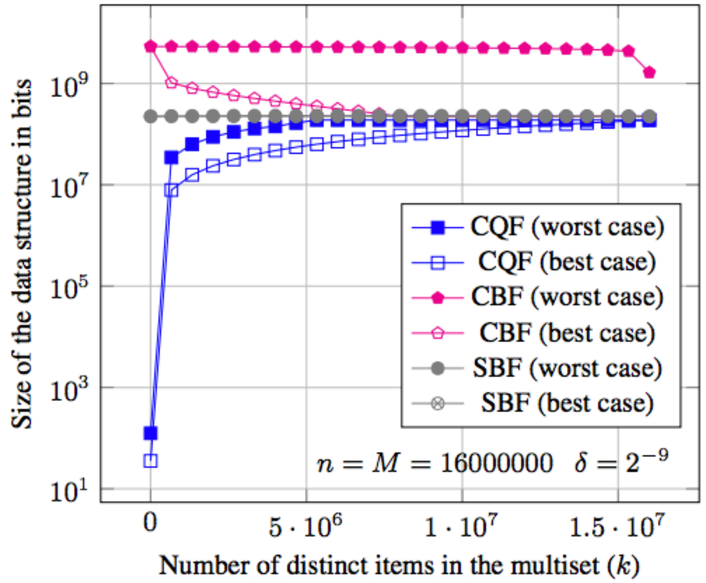
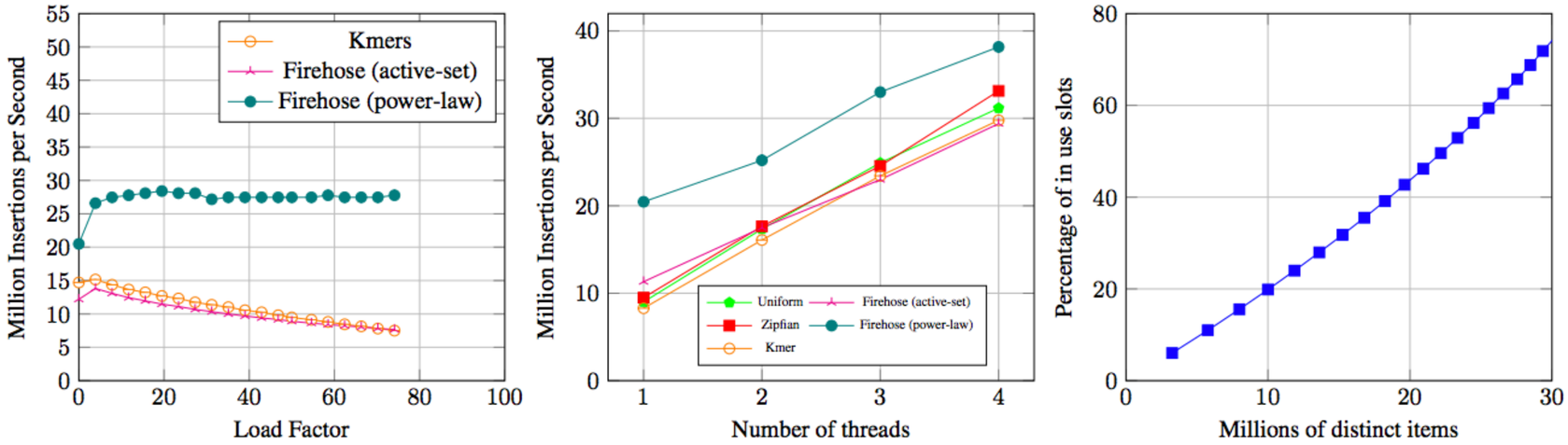


Figure 5: Space comparison of CQF, SBF, and CBF as a function of the number of distinct items. All data structures are built to support up to $n = 1.6 \times 10^7$ insertions with a false-positive rate of $\delta = 2^{-9}$.

The Counting Quotient Filter, results



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

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

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

Figure 9: In-memory performance of the counting quotient filter with real-world data sets and with multiple threads, and percent slot usage with skewed distribution.

Squeakr, applying the CQF to k-mer counting

Counting Memory

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

Counting performance

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

Squeakr, applying the CQF to k-mer counting

Query performance

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

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).

Take-home message

The sheer scale of the data we have to deal with makes
even the most simple tasks (e.g. counting k-mers) rife with opportunities for the development and application of interesting and novel data structures and algorithms!