K-mer counting, cardinality estimation, Approximate Membership Query (AMQ) data structures & perfect hashes

UNIVERSITY OF MARYLAND

Scalability at the forefront

I've spoken a lot 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:

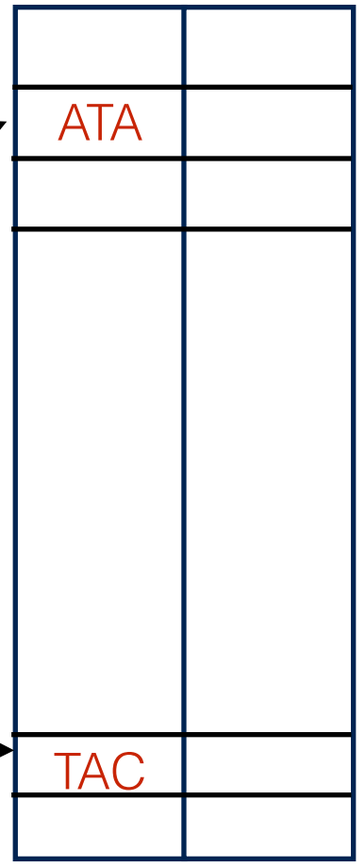
> Tallymer, Jellyfish, DSK, KMC{1,2,3}, BFCounter, scTurtle, Girbil, KAnalyze, khmer, … and many more

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

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

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

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

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

Easily get to > 100GB of RAM

Smart, parallel hashing actually pretty good

If we put some thought and engineering effort into the hashing approach, it can actually do pretty well. This is the insight behind the Jellyfish program.

Massively parallel, *lock-free*, k-mer counting

— most parallel accesses *won't* cause a collision

Efficient storage of hash table values

— bit-packed data structure

— small counter with multiple entries for high-count k-mers

Efficient storage of keys

— f: $U_k \rightarrow U_k$, and let hash(k) = f(k) mod M

— Can reconstruct k from pos in hash table (quotient) and remainder.

Smart, parallel hashing actually pretty good

Efficient storage of keys

— f: $U_k \rightarrow U_k$, and let hash(k) = f(k) mod M

— recall: we can represent f(k) as f(k) = qM + r

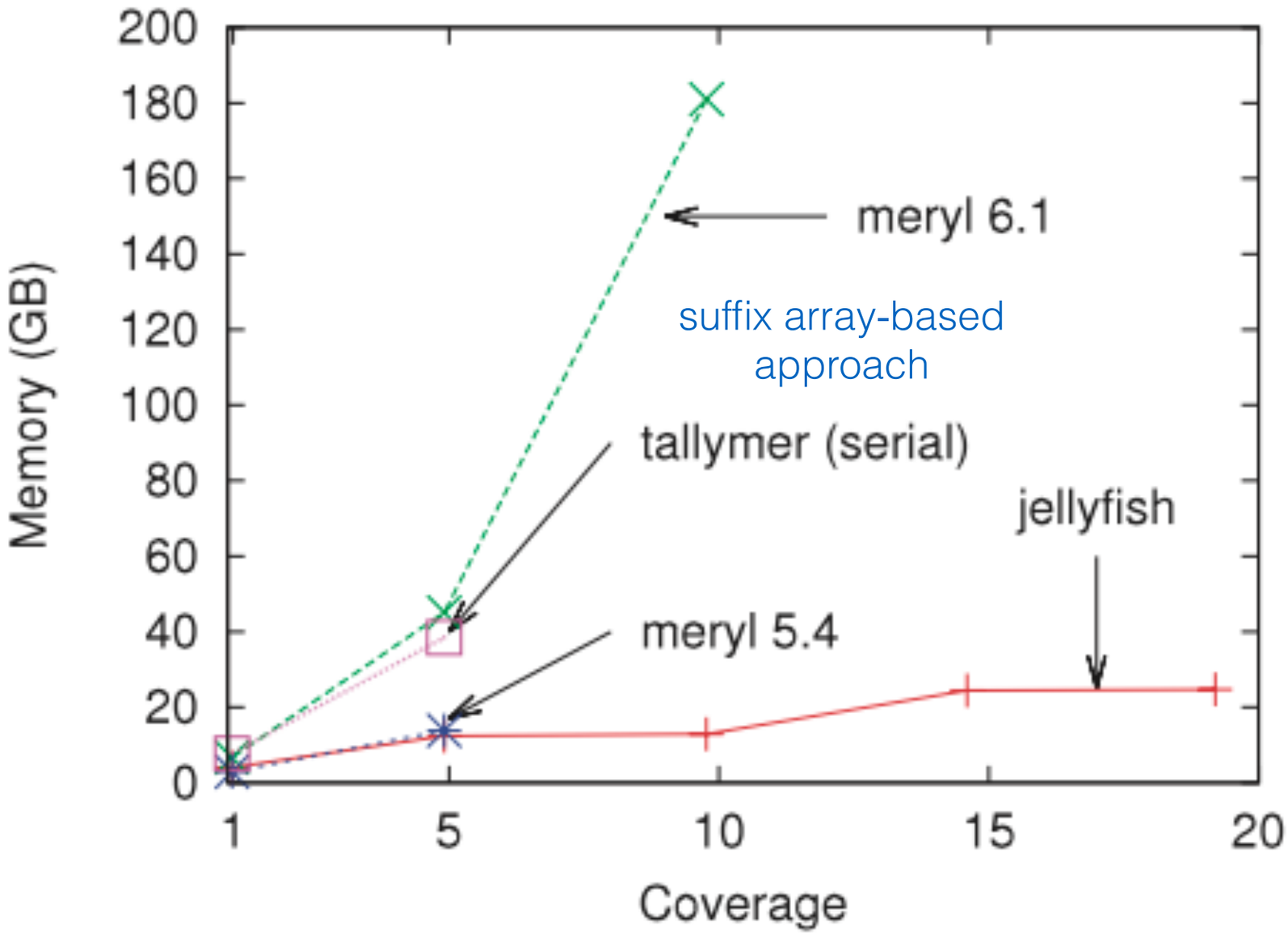
— Can reconstruct k from pos in hash table (quotient, q) and remainder, r. The quotient is simply encoded as the position.

— Extra work must be done since collisions can occur

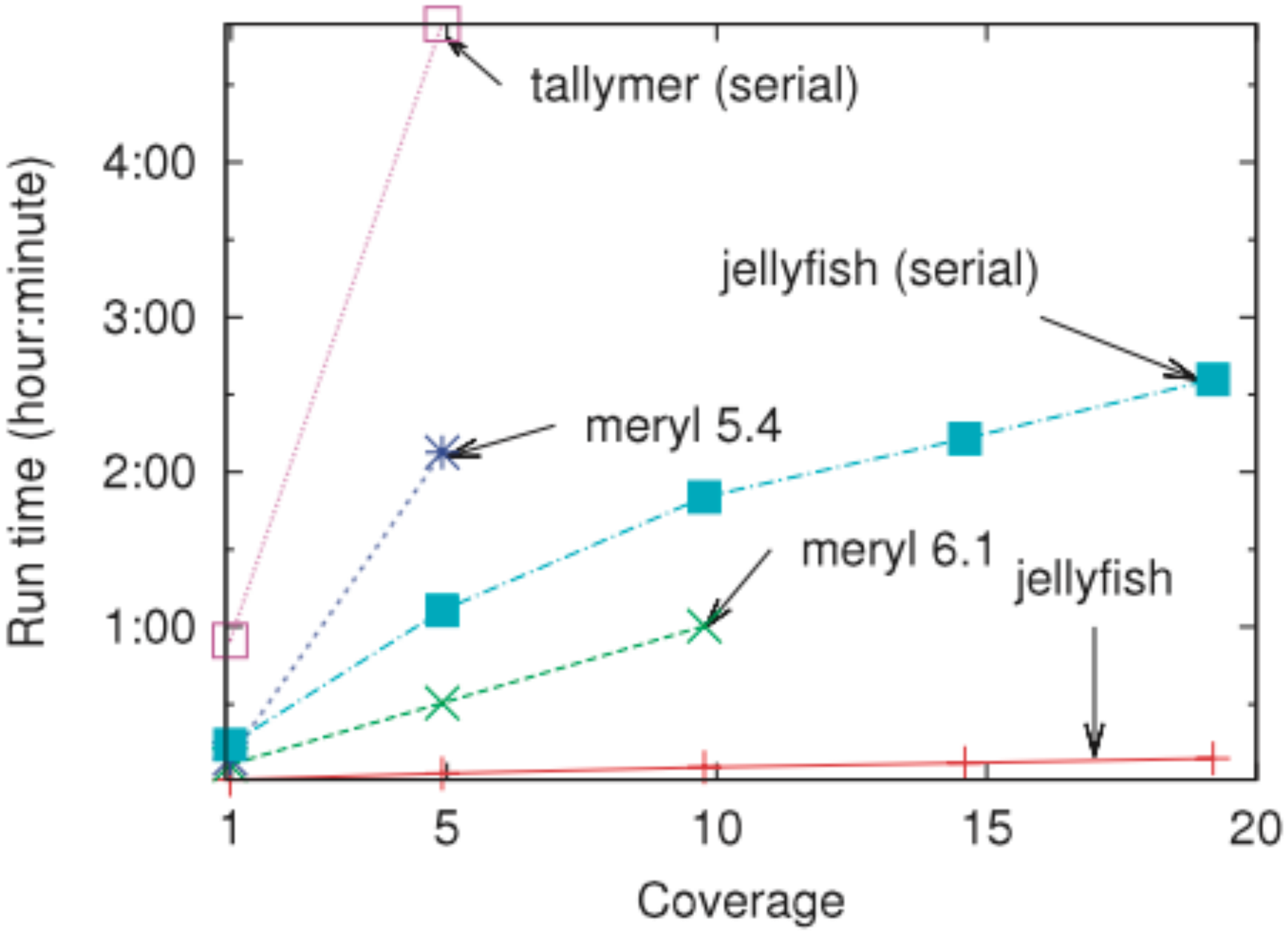
— For a general coverage of this idea, see the Quotient Filter data structure by Bender et al. (2011)

"Don't thrash: how to cache your hash on flash" (PDF). *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems (HotStorage'11)*. Retrieved 21 July 2012.

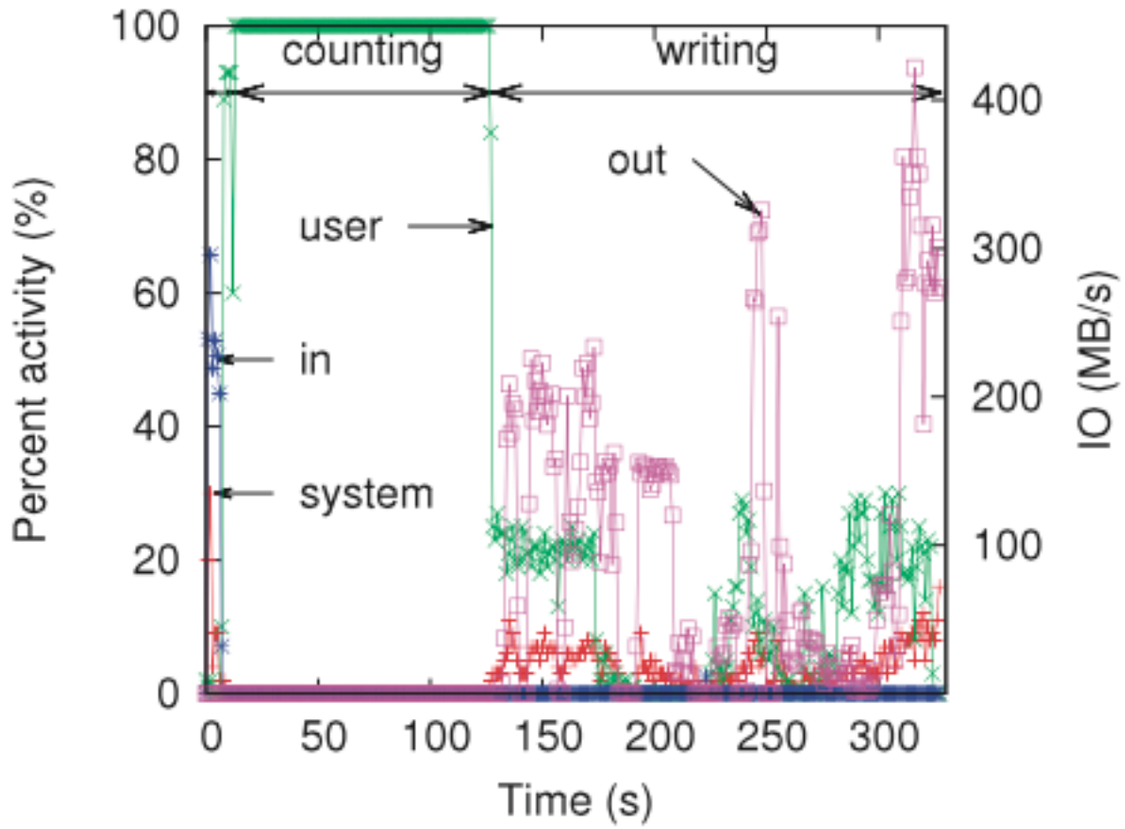
Memory usage of Jellyfish

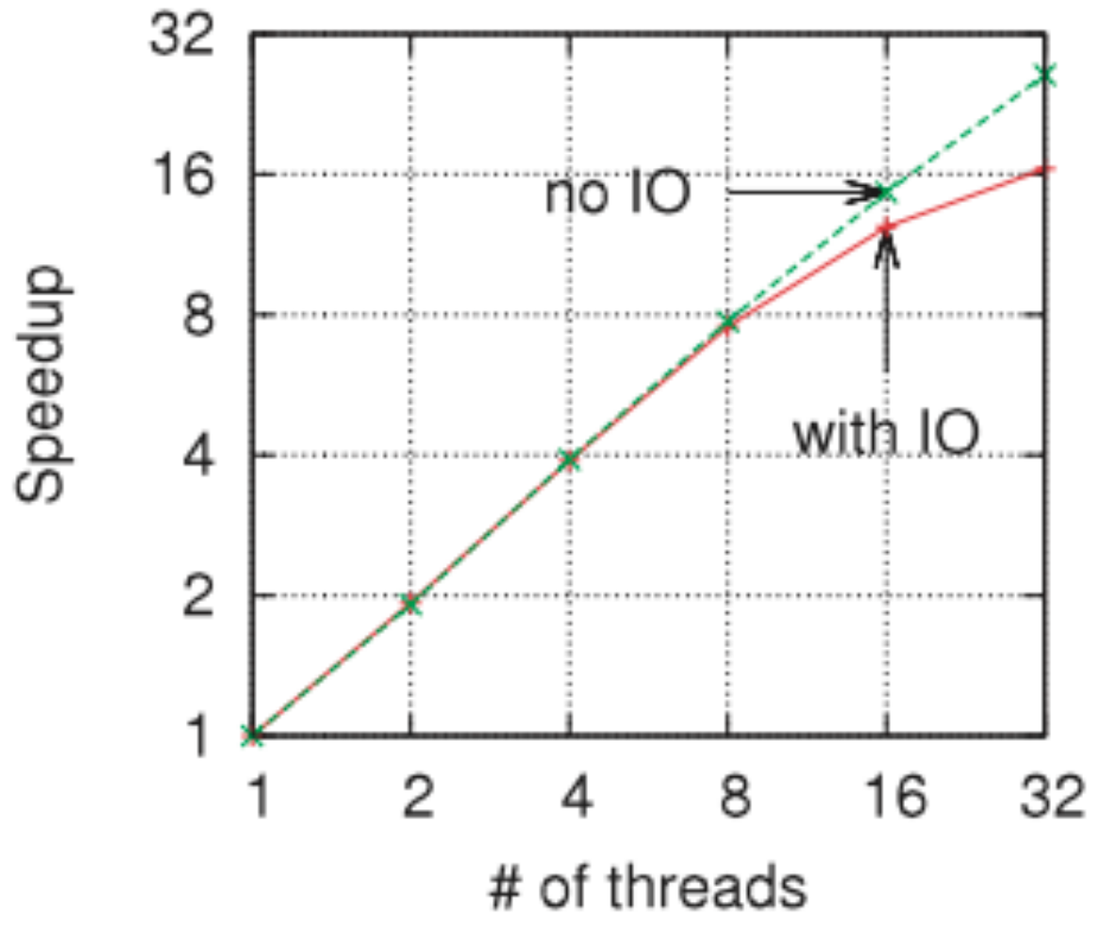


Runtime of Jellyfish



System utilization of Jellyfish





Even bigger data

For very large datasets, even this approach may use too much memory. How can we do better?

Even bigger data

For very large datasets, even this approach may use too much memory. How can we do better?

Solve a different (but closely-related) problem

What if we just want to know “if” a k-mer is present?

What if we just wanted “approximate” counts?

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 $0 < i < k$

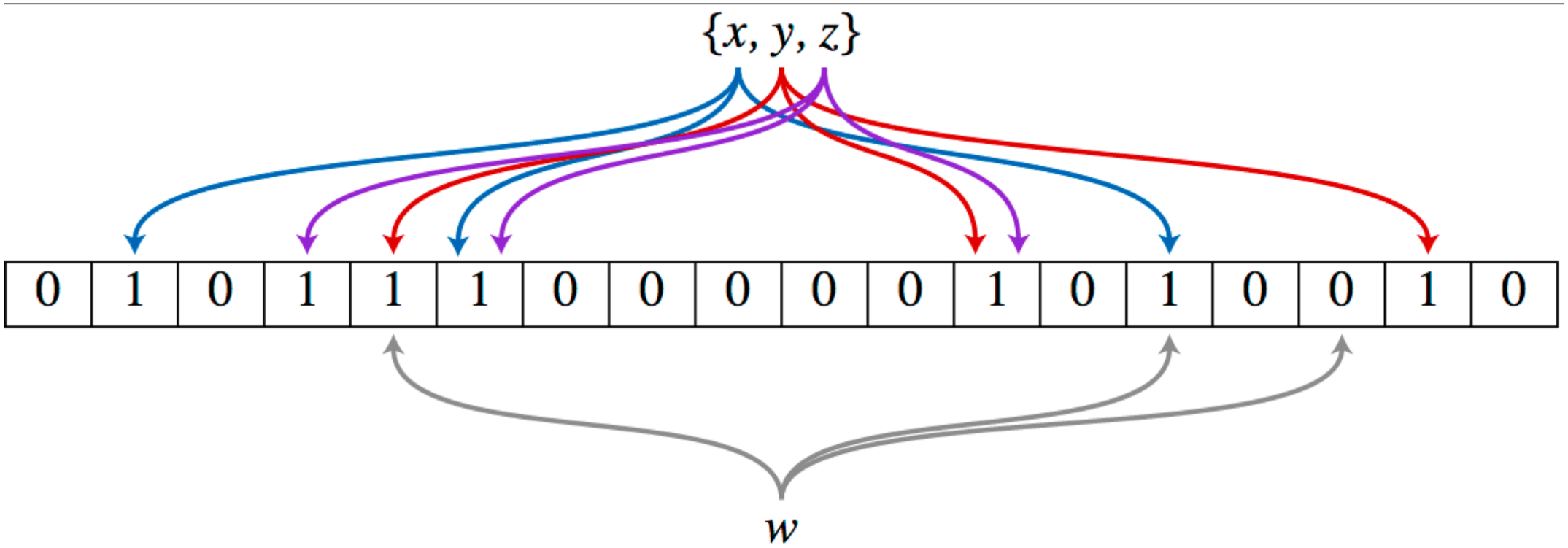


Image by David Eppstein - self-made, originally for a talk at WADS 2007

Bloom Filters

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

How low?

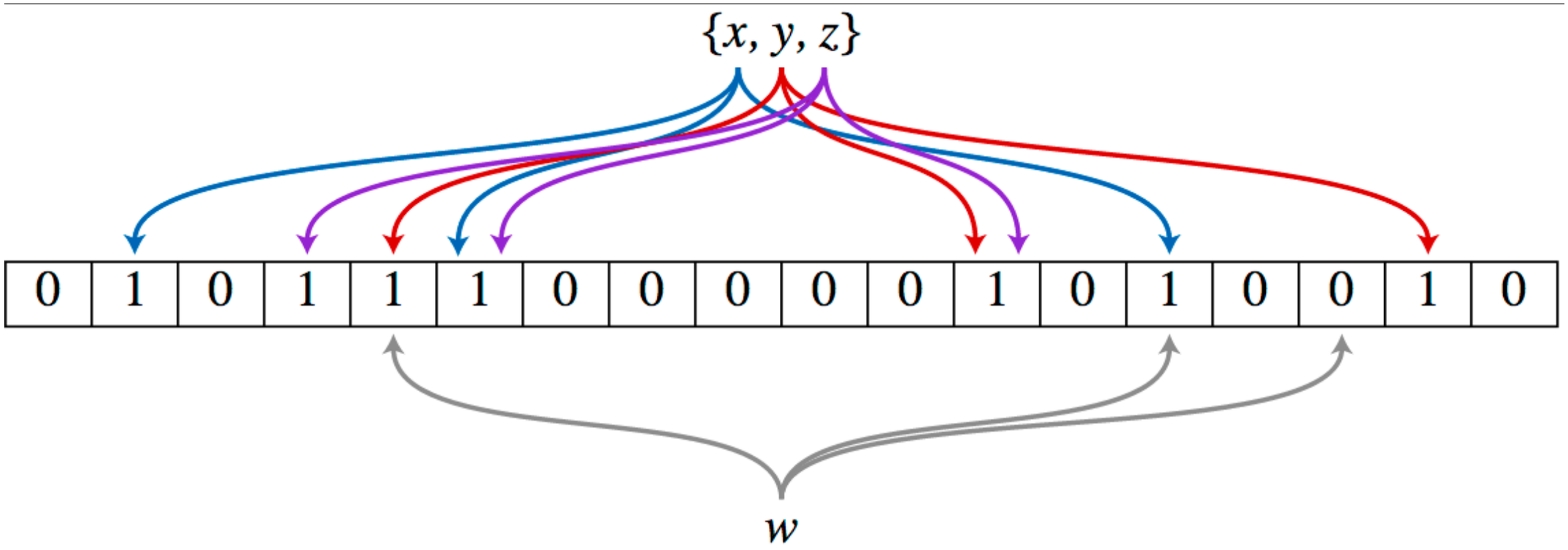


Image by David Eppstein - self-made, originally for a talk at WADS 2007

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 probability 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

Cardinality Estimation

Consider a “simpler” problem than indexing, or even k-mer counting:

Given: A collection of sequencing reads S and paramater k and t.

Find: The number of k-mers that occur 1 time, 2 times, …, t times.

This is the *k-mer cardinality estimation* problem

Cardinality Estimation

There is the hope that we can solve this (approximately) very efficiently.

Why: We need not record information for each distinct k-mer, the output is simply a vector of length t.

We'll discuss one particular approach for solving this, introduced in ntCard

Bioinformatics, 33(9), 2017, 1324–1330
doi: 10.1093/bioinformatics/btw832
Advance Access Publication Date: 5 January 2017
Original Paper

OXFORD

Sequence analysis

ntCard: a streaming algorithm for cardinality estimation in genomics data

Hamid Mohamadi[1,2,*], Hamza Khan[1,2] and Inanc Birol[1,2,*]

[1]Canada's Michael Smith Genome Sciences Centre, British Columbia Cancer Agency, Vancouver, BC, V5Z 4S6, Canada and [2]Faculty of Science, University of British Columbia, Vancouver, BC, V6T 1Z4, Canada

*To whom correspondence should be addressed.
Associate Editor: Bonnie Berger

Received on October 31, 2016; revised on December 21, 2016; editorial decision on December 25, 2016; accepted on December 27, 2016

let f_i be the number of distinct k-mers that appear i times

frequency histogram is list of $f_i, i \geq 1$

Define k-th frequency moment as $F_k = \sum_{i=1}^{\infty} i^k \cdot f_i$

Goal: Estimate the f_i , usually only care about smallish maximum i (e.g. 64).

Basic idea: Hash the k-mers

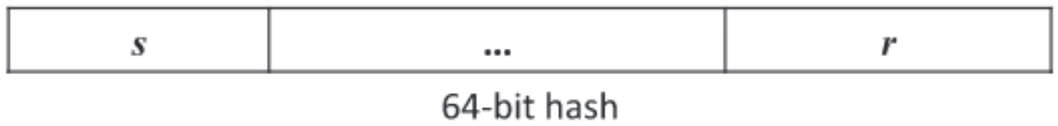
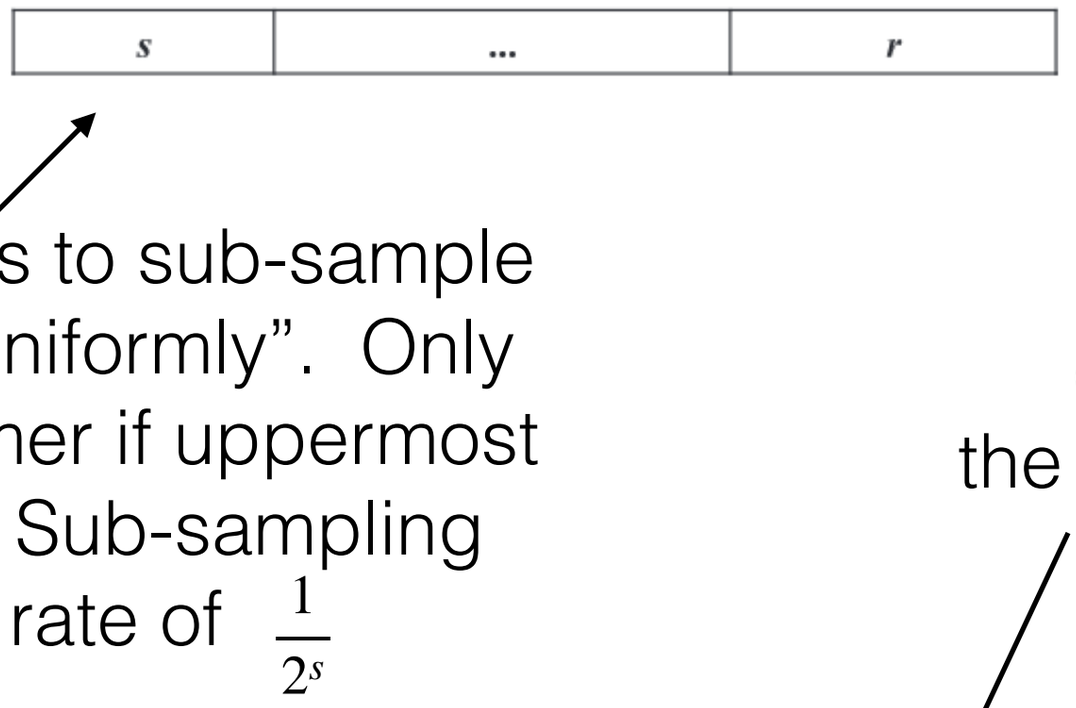
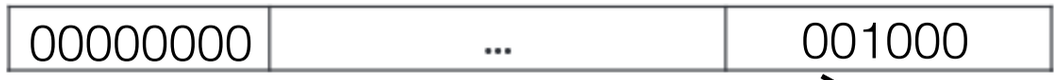


Fig. 1. 64-bit hash value generated by ntHash. The *s* left bits are used for sampling the *k*-mers in input datasets and the *r* right bits are used as resolution bit for building the reduced multiplicity table, with $r + s < 64$



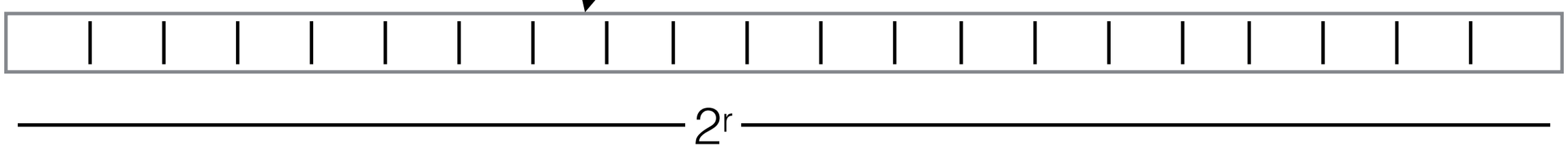
When we encounter a k-mer's hash:



If the uppermost s bits are 0

Then we increment the count in the cell determined by the lowermost r bits

$t^{(r)}$ array holding counts



The *true* cardinality histogram is the histogram we would have if we used $r=\infty$. Clearly, we can't do this, so we will instead *estimate* that value given a fixed, finite r.

We want to estimate $t^{(\infty)}$, what is the relationship between $t^{(r)}$ and $t^{(r+1)}$?

$$t_n^{(r)} = t_n^{(r+1)} + t_{2^r+n}^{(r+1)}, \qquad \forall n \in [0, \ldots, 2^r - 1] \tag{5}$$

where $t_n^{(r)}$ denotes the count for entry n in table $t^{(r)}$

Let $p_i^{(r)}$ be the relative frequency (probability) of count $i \geq 0$ in table $t^{(r)}$

Observe: $t_i^{(r)} = 0$ iff $t_i^{(r+1)} = 0$ and $t_{2^r+i}^{(r)} = 0$

Mohamadi, Hamid, Hamza Khan, and Inanc Birol. "ntCard: a streaming algorithm for cardinality estimation in genomics data." *Bioinformatics* 33.9 (2017): 1324-1330.

Assuming distributions in both half of $t^{(r+1)}$ are the same

$$p_0^r = (p_0^{(r+1)})^2 \qquad \text{(eq 6)}$$

relates the frequencies of 0 counts in $t^{(r)}$ to $t^{(r+1)}$

Similarly, a count of 1 in $t_n^{(r)}$ can happen only if

$$t_n^{(r)} = 0 \text{ and } t_{2^r+n}^{(r)} = 1$$

or

$$t_n^{(r)} = 1 \text{ and } t_{2^r+n}^{(r)} = 0$$

We can express this mathematically as:

$$p_1^{(r)} = 2p_0^{(r+1)}p_1^{(r+1)} \qquad \text{(eq 7)}$$

Mohamadi, Hamid, Hamza Khan, and Inanc Birol. "ntCard: a streaming algorithm for cardinality estimation in genomics data." *Bioinformatics* 33.9 (2017): 1324-1330.

This rule can be generalized

$$p_i^{(r)} = \sum_{i'=0}^{i} p_{i'}^{(r+1)} p_{i-i'}^{(r+1)} \qquad \text{(eq 8)}$$

Note that, Equations (6)–(8) can be solved for $p_i^{(r+1)}$ through the recursive formula

$$p_i^{(r+1)} = \begin{cases} \left(p_0^{(r)}\right)^{1/2} & \text{for } i = 0 \\ \dfrac{p_1^{(r)}}{2p_0^{(r+1)}} & \text{for } i = 1 \\ \dfrac{1}{2p_0^{(r+1)}} \left(p_i^{(r)} - \displaystyle\sum_{i'=1}^{i-1} p_{i'}^{(r+1)} p_{i-i'}^{(r+1)} \right) & \text{for } i > 1 \end{cases} \qquad (9)$$

This tells us how to go from r to r+1, we want to compute these values for r+x as $x \to \infty$

Mohamadi, Hamid, Hamza Khan, and Inanc Birol. "ntCard: a streaming algorithm for cardinality estimation in genomics data." *Bioinformatics* 33.9 (2017): 1324-1330.

we will call our estimates $\hat{f}_i$

$$\hat{f}_i = \frac{p_i^{(\infty)}}{1 - p_0^{(\infty)}}$$

For example, for $i = 1$, this can be calculated as

$$\hat{f}_1 = \lim_{x \to \infty} \frac{\frac{p_1^{(r)}}{2^x (p_0^{(r)})^{\frac{2^x - 1}{2^x}}}}{1 - (p_0^{(r)})^{\frac{1}{2^x}}} = \frac{-p_1^{(r)}}{p_0^{(r)} \ln p_0^{(r)}}$$

and for $i = 2$ as

$$\hat{f}_2 = \frac{-p_0^{(r)} p_2^{(r)} + \frac{1}{2} (p_1^{(r)})^2}{(p_0^{(r)})^2 \ln p_0^{(r)}}$$

Mohamadi, Hamid, Hamza Khan, and Inanc Birol. "ntCard: a streaming algorithm for cardinality estimation in genomics data." *Bioinformatics* 33.9 (2017): 1324-1330.

In general, for $\hat{f}_i, i \geq 1$, we can write the following equation

$$\hat{f}_i = \frac{1}{(p_0^{(r)})^i \ln p_0^{(r)}} \sum_{j=0}^{i-1} \frac{(-1)^{i+j} (p_0^{(r)})^j}{i-j} \Bigg(\sum_{\substack{\forall (l,u) \in \mathbb{Z}^2 \text{ s.t.} \\ \sum_k u_k = i-j \\ \sum_k l_k u_k = i}} \prod_{k=1}^{|u|} \binom{i-j-\sum_{k'=0}^{k-1} u_{k'}}{u_k} (p_{l_k}^{(r)})^{u_k} \Bigg)$$

where $u_0 = 0$, $u_k \neq u_{k'}$ for all $k \neq k'$, and $|u| = \text{argmax}_k \{u_k\}$.

UGLY!

Mohamadi, Hamid, Hamza Khan, and Inanc Birol. "ntCard: a streaming algorithm for cardinality estimation in genomics data." *Bioinformatics* 33.9 (2017): 1324-1330.

This complex-looking formula can also be written in the following recursive form

$$\hat{f}_i = \frac{-p_i^{(r)}}{p_0^{(r)} \ln p_0^{(r)}} - \frac{1}{i} \sum_{j=1}^{i-1} \frac{j p_{i-j}^{(r)} \hat{f}_j}{p_0^{(r)}} \tag{14}$$

The two terms of this equation can be interpreted as follows. The first term corresponds to count frequencies i in table $t^{(r)}$ assuming none of the entries collided with any non-zero entries through folding rounds from $\lim_{x \to \infty}(r + x)$ to r. The second term is a correction to the first term, accounting for all collisions of $(i - j), 0 < j < i$ and j, result of which is a count frequency of i.

Mohamadi, Hamid, Hamza Khan, and Inanc Birol. "ntCard: a streaming algorithm for cardinality estimation in genomics data." *Bioinformatics* 33.9 (2017): 1324-1330.

We can also estimate the 0th order moment as

$$F_0 = \lim_{x \to \infty} 2^s (1 - p_0^{(r+x)}) 2^{r+x} = -2^{s+r} \ln p_0^{(r)}$$

together with the $\hat{f}_i$, this is enough to compute all we want

Algorithm 1. The ntCard algorithm

```
1: function Update(k-mer)
2:     for each read seq do
3:         for each k-mer in seq do
4:             h ← ntHash (k-mer)                ▷ Compute 64-bit h using ntHash
5:             if h_{64:64−s+1} = 0^s then       ▷ Checking the s left bit in h
6:                 i ← h_{r:1}                   ▷ r is resolution parameter
7:                 t_i ← t_i + 1
8: function Estimate
9:     for i ← 1 to 2^r do
10:        p_{t[i]} ← p_{t[i]} + 1
11:    for i ← 1 to t_max do
12:        p_i ← p_i / 2^r
13:    F_0 = −ln p_0 × 2^{s+r}                   ▷ F_0 estimate
14:    for i ← 1 to t_max do
15:        f̂_i ← −p_i / (p_0 ln p_0) − (1/i) Σ_{j=1}^{i−1} j p_{i−j} f̂_j / p_0   ▷ Relative estimates
16:    for i ← 1 to t_max do
17:        f_i ← f̂_i × F_0                       ▷ f_i estimates
18:    return f, F_0
```

Mohamadi, Hamid, Hamza Khan, and Inanc Birol. "ntCard: a streaming algorithm for cardinality estimation in genomics data." *Bioinformatics* 33.9 (2017): 1324-1330.

Results

Table 1. Dataset specification

Dataset	Read number	Read length	Total bases	Size
HG004	868,593,056	250 bp	217,148,264,000	480 GB
NA19238	913,959,800	250 bp	228,489,950,000	500 GB
PG29	6,858,517,737	250 bp	1,714,629,434,250	2.4 TB

Table 2. Accuracy of algorithms in estimating F_0 and f_1 for HG004 reads

k		DSK	ntCard	KmerGenie	KmerStream	Khmer
32	f_1	13,319,957,567	**0.01%**	0.97%	7.04%	–
	F_0	16,539,753,749	**0.02%**	0.64%	5.12%	0.67%
64	f_1	17,898,672,342	**0.02%**	0.35%	0.73%	–
	F_0	21,343,659,785	**0.00%**	0.22%	0.66%	0.15%
96	f_1	18,827,062,018	0.36%	0.87%	**0.00%**	–
	F_0	22,313,944,415	0.24%	0.69%	**0.05%**	0.31%
128	f_1	18,091,241,186	**0.36%**	0.76%	0.40%	–
	F_0	21,555,678,676	0.25%	0.62%	**0.20%**	0.30%

The DSK column reports the exact k-mer counts, and columns for the other tools report percent errors.

Table 3. Accuracy of algorithms in estimating F_0 and f_1 for NA19238 reads

k		DSK	ntCard	KmerGenie	KmerStream	Khmer
32	f_1	14,881,561,565	**0.00%**	0.53%	6.36%	–
	F_0	18,091,801,391	**0.00%**	0.40%	4.64%	1.82%
64	f_1	19,074,667,480	**0.02%**	0.75%	0.68%	–
	F_0	22,527,419,136	**0.01%**	0.77%	0.65%	1.22%
96	f_1	19,420,503,673	0.22%	0.66%	**0.09%**	–
	F_0	22,932,238,161	0.16%	0.66%	**0.07%**	0.46%
128	f_1	17,902,027,438	0.21%	0.85%	**0.19%**	–
	F_0	21,421,517,759	0.13%	0.76%	**0.03%**	1.05%

Table 4. Accuracy of algorithms in estimating F_0 and f_1 for PG29 reads

k		DSK	ntCard	KmerGenie	KmerStream	Khmer
32	f_1	27,430,910,938	**0.02%**	15.33%	9.41%	–
	F_0	42,642,198,777	**0.01%**	11.02%	7.37%	8.86%
64	f_1	44,344,130,469	**0.04%**	16.36%	2.61%	–
	F_0	67,800,291,613	**0.02%**	11.14%	1.73%	11.18%
96	f_1	43,300,244,443	**0.66%**	17.51%	0.73%	–
	F_0	69,855,690,006	**0.46%**	11.13%	0.57%	9.36%
128	f_1	32,089,613,024	0.40%	14.82%	**0.06%**	–
	F_0	58,195,246,941	0.30%	8.35%	**0.27%**	7.39%

Captures the whole histogram well

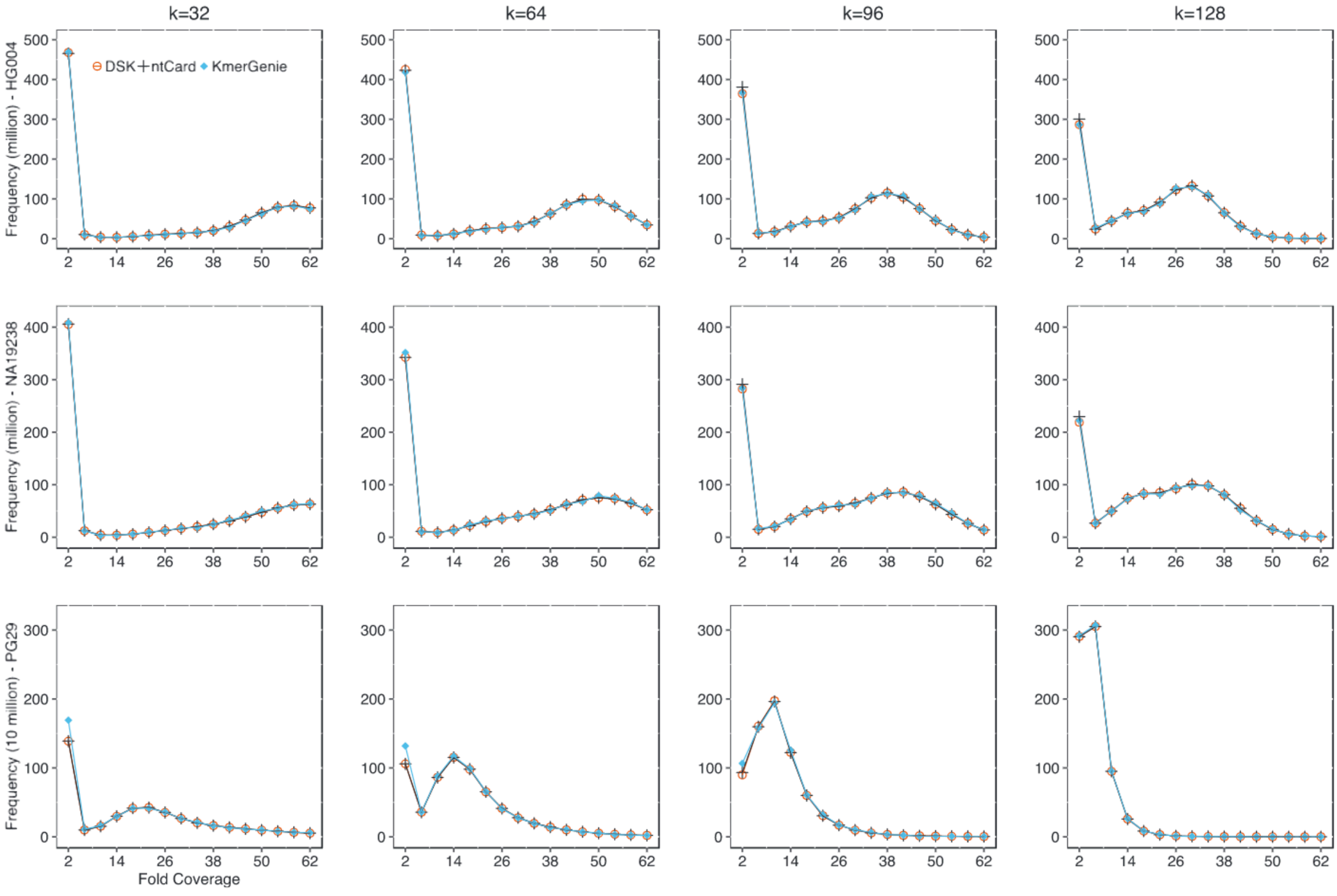


Fig. 2. *k*-mer frequency histograms for human genomes HG004 and NA19238 (rows 1 and 2, respectively), and the white spruce genome PG29 (row 3). We have used DSK *k*-mer counting results as our ground truth in evaluation (orange circle data points). The *k*-mer coverage frequency results, $f_2..f_{62}$ of ntCard and KmerGenie for different values of $k = 32, 64, 96, 128$ (the four columns from left to right) are shown with the symbols (+) and ($\diamond$), respectively

The ntCard algorithm is fast

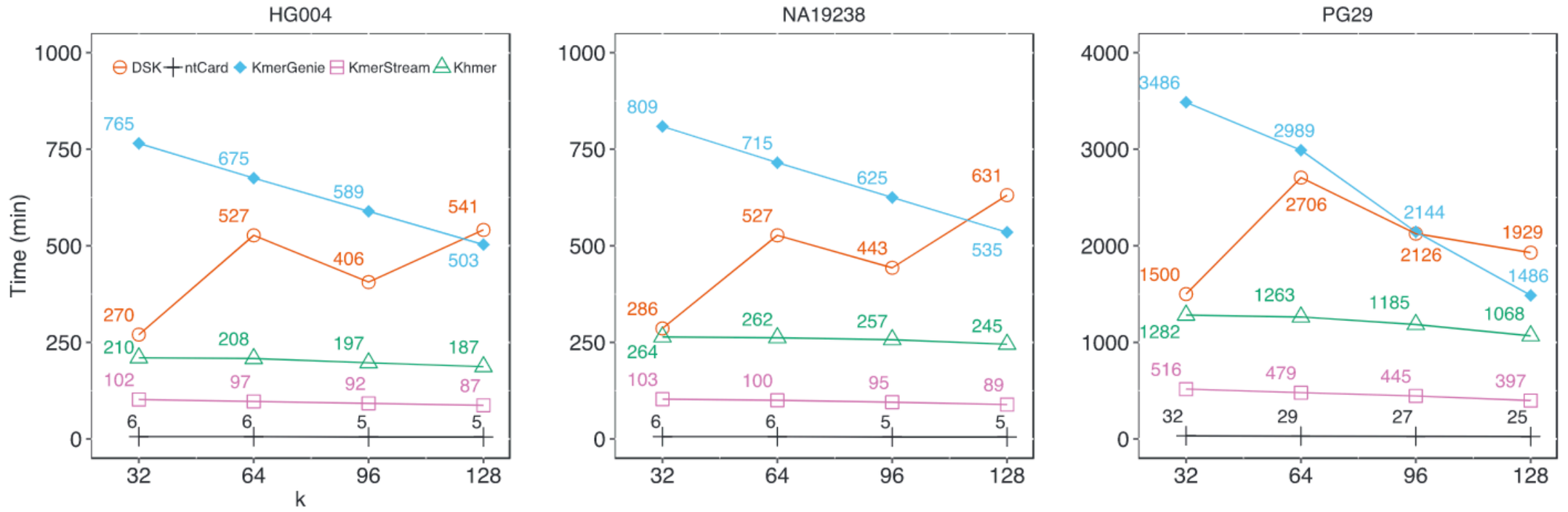


Fig. 3. Runtime of DSK, ntCard, KmerGenie, KmerStream and Khmer for all three datasets, HG004, NA19238 and PG29. We have calculated the runtime of all algorithms for different values of k in $\{32, 64, 96, 128\}$. As we see in the plots, ntCard estimates the full k-mer coverage frequency histograms $>15\times$ faster than KmerStream

The memory usage of ntCard on all 3 datasets is ~500MB

Open challenge: Can we solve this problem *sub-linearly* (without looking at all input k-mers)?

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 → {0,…,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.

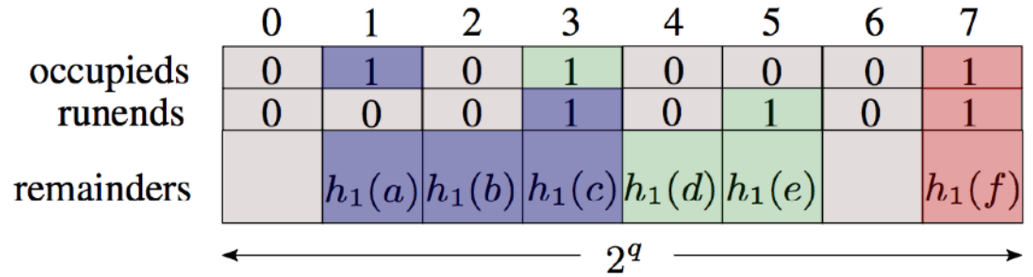
Q is an array of 2^q r-bit slots

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))]

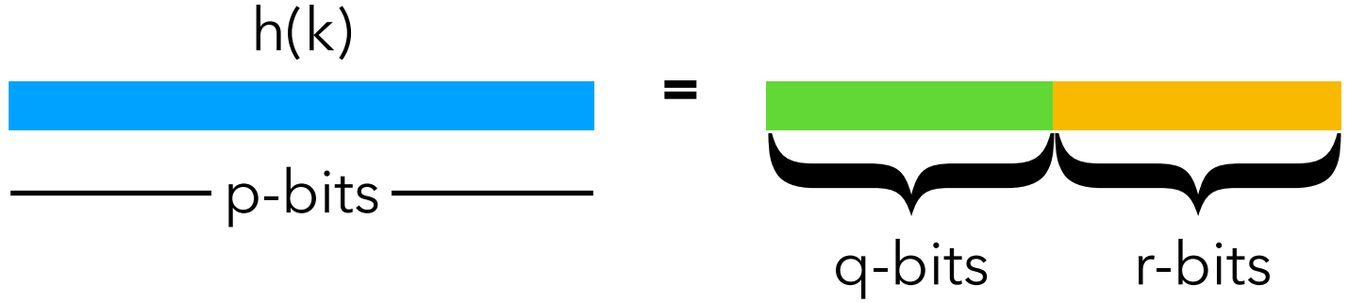
The Counting Quotient Filter (CQF)

Approximate *Multiset* Representation



Works based on quotienting* & fingerprinting keys

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



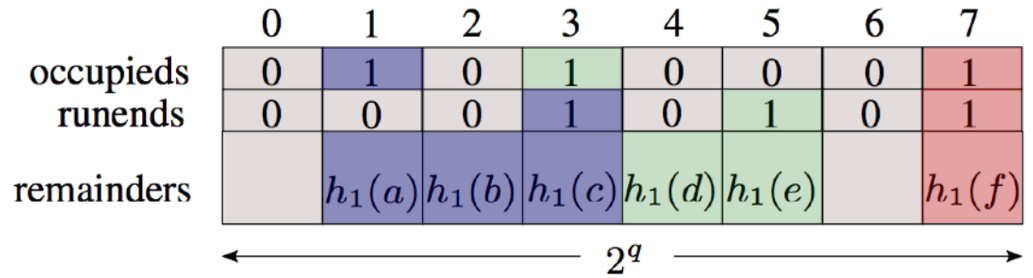
Clever encoding allows low-overhead storage of element counts
(use *key* slots to store *values* in base 2^r-1; smaller values ⇒ fewer bits)

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)

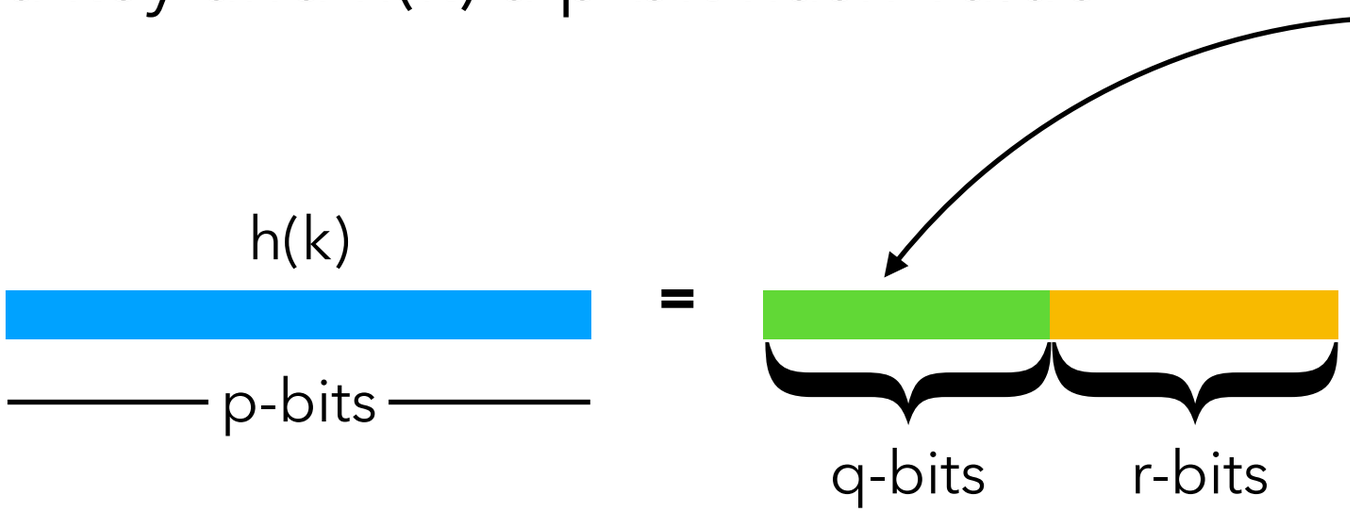
Approximate *Multiset* Representation



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



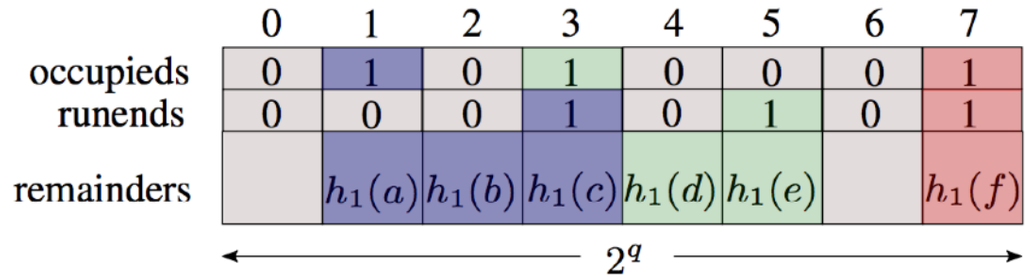
Clever encoding allows low-overhead storage of element counts (use *key* slots to store *values* in base 2^r-1; smaller values ⇒ fewer bits)

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)

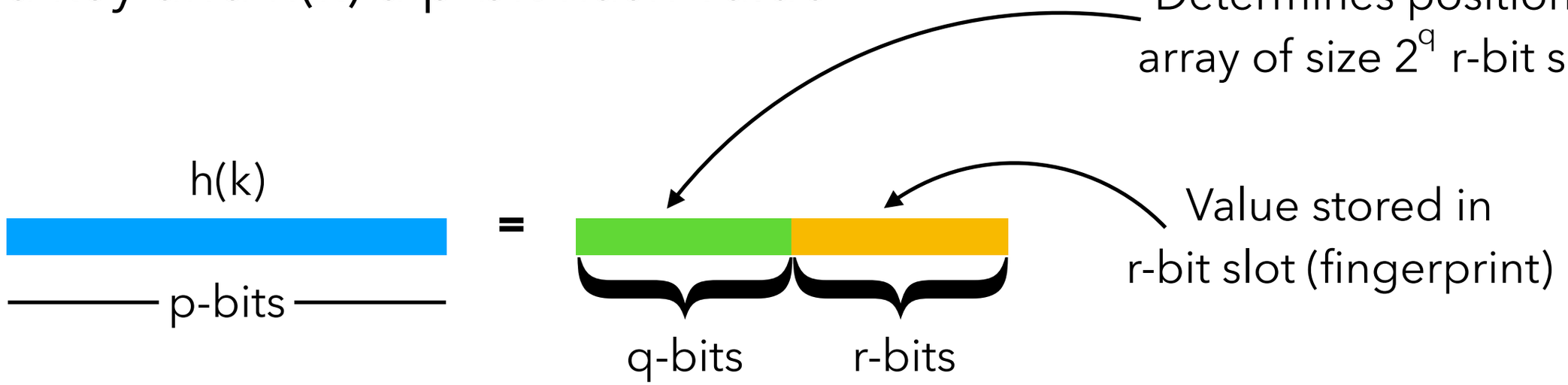
The Counting Quotient Filter (CQF)

Approximate *Multiset* Representation



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)?

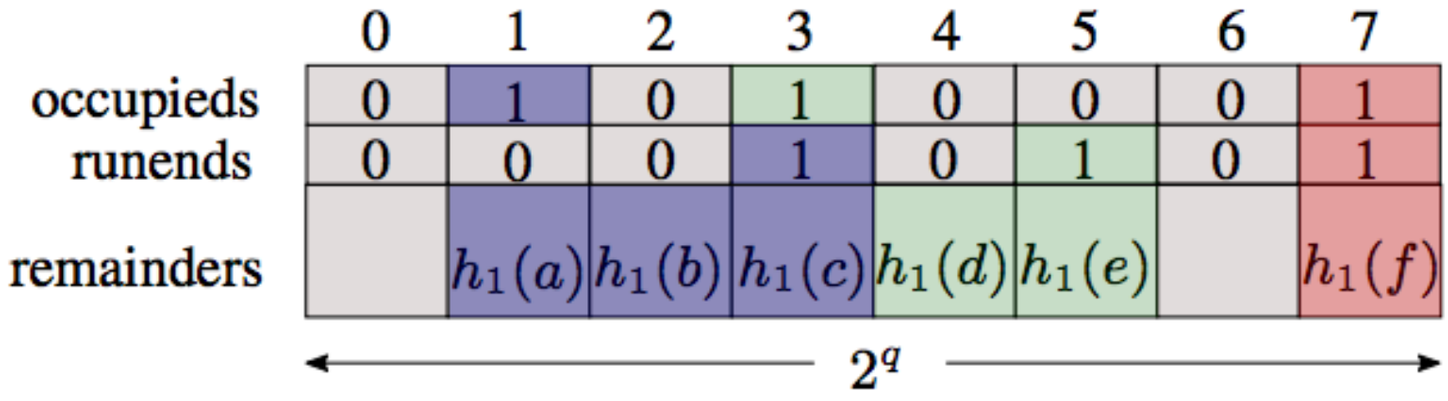


Figure 1: A simple rank-and-select-based quotient filter. The colors are used to group slots that belong to the same run, along with the runends bit that marks the end of that run and the occupieds bit that indicates the home slot for remainders in that run.

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

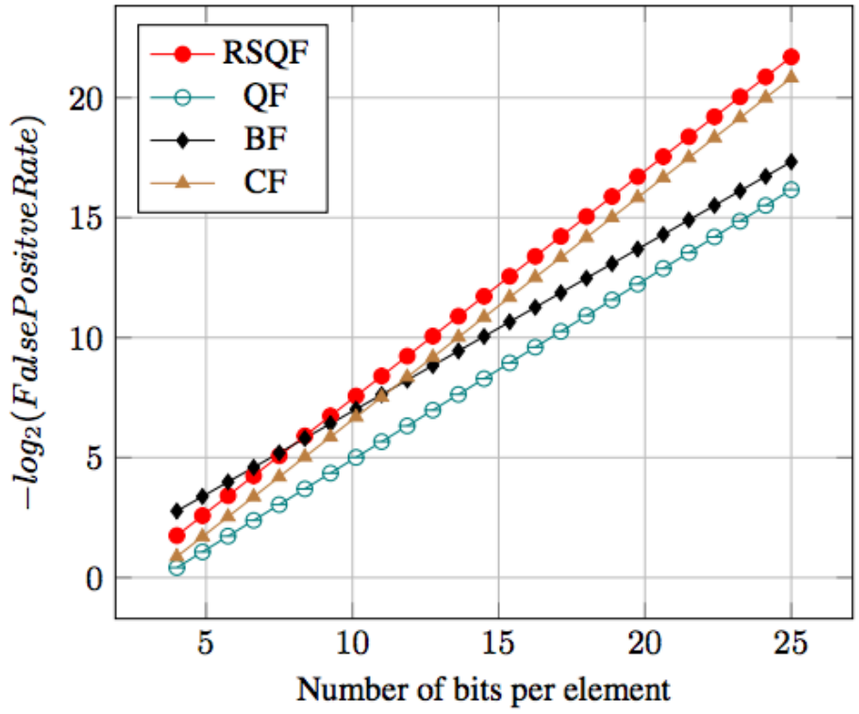


Figure 4: Number of bits per element for the RSQF, QF, BF, and CF. The RSQF requires less space than the CF amd less space than the BF for any false-positive rate less than $1/64$. (Higher is better)

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).

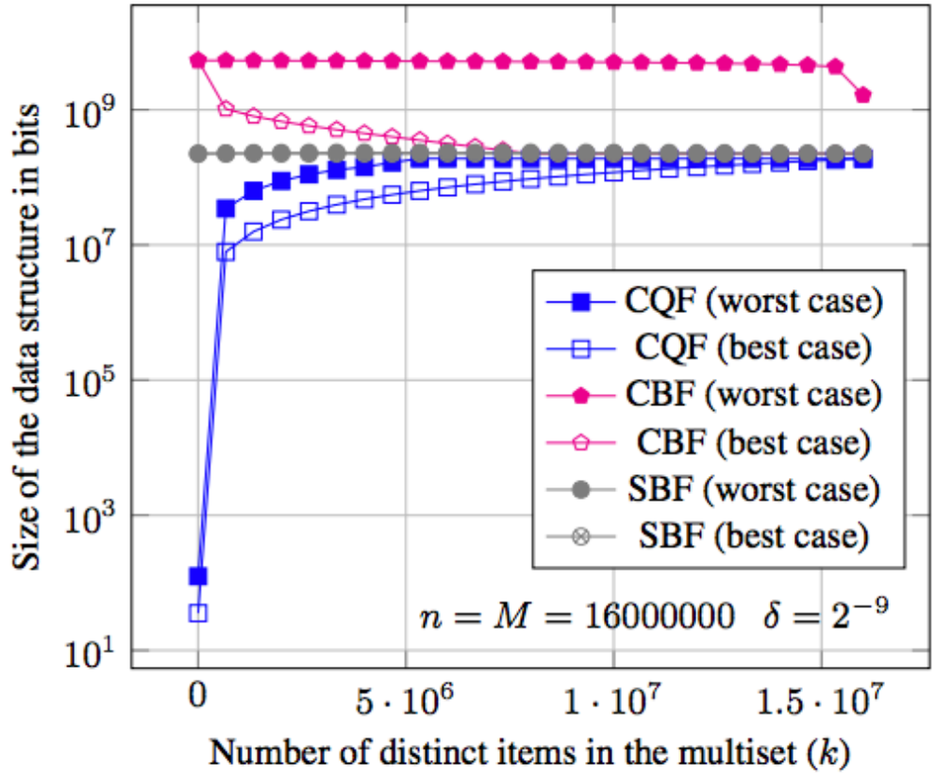
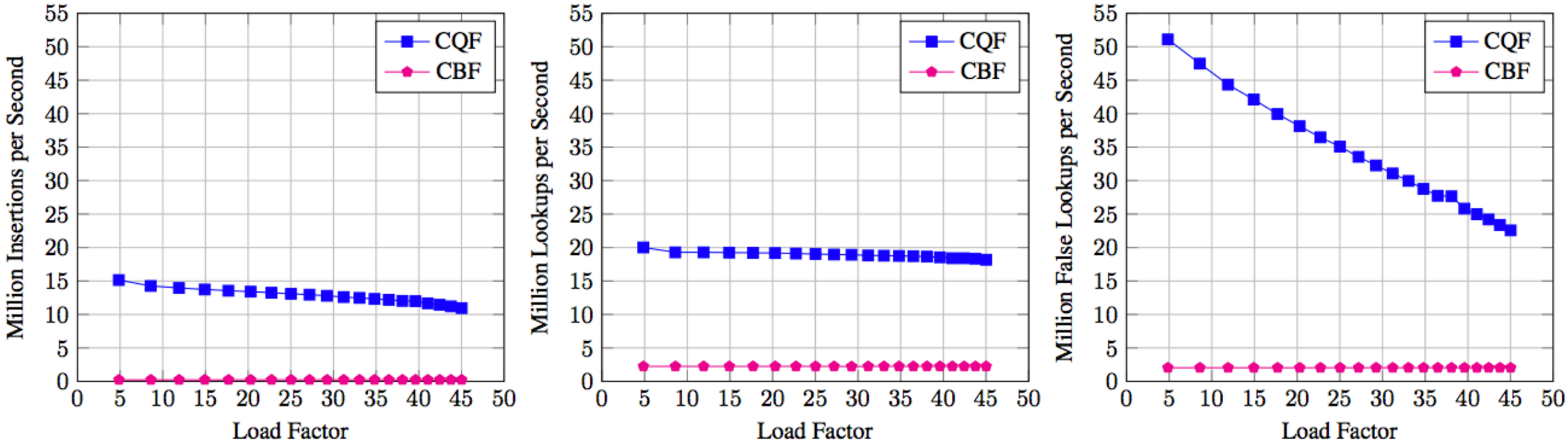


Figure 5: Space comparison of CQF, SBF, and CBF as a function of the number of distinct items. All data structures are built to support up to $n = 1.6 \times 10^7$ insertions with a false-positive rate of $\delta = 2^{-9}$.

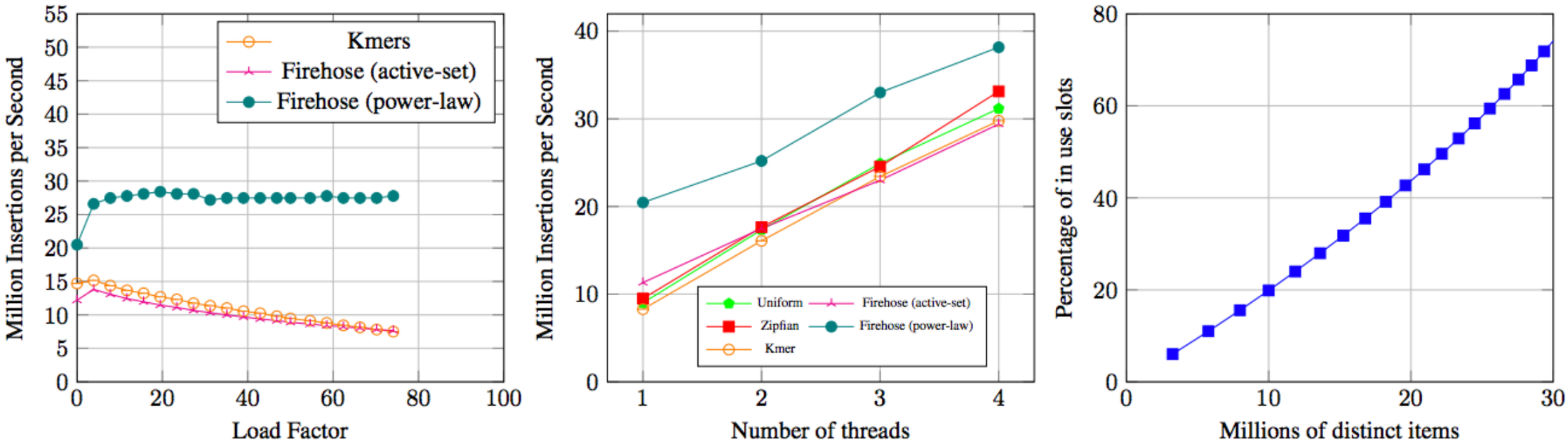
The Counting Quotient Filter, results



(a) Inserts. (b) Successful lookups. (c) Uniformly random lookups.

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

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

Squeakr, applying the CQF to k-mer counting

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).

(Minimum) Perfect Hash Functions

We've been using the idea of *hashing* a lot in this lecture.

One class of hash functions that are particularly interesting are Minimum Perfect Hash Functions (MPHF).

S : set of keys

$f : S \rightarrow \{1,2,\ldots|S|\}$ s.t.

$$\forall\ u,v \in S,\ u \neq v \text{ then } f(u) \in \{1,2,\ldots|S|\}$$
$$\text{and } f(v) \in \{1,2,\ldots|S|\} \text{ and } f(u) \neq f(v)$$

In other words. f is an injective function from S to the integers 1..|S| (or 0..|S|-1) such that every element of S maps to some *distinct* integer.

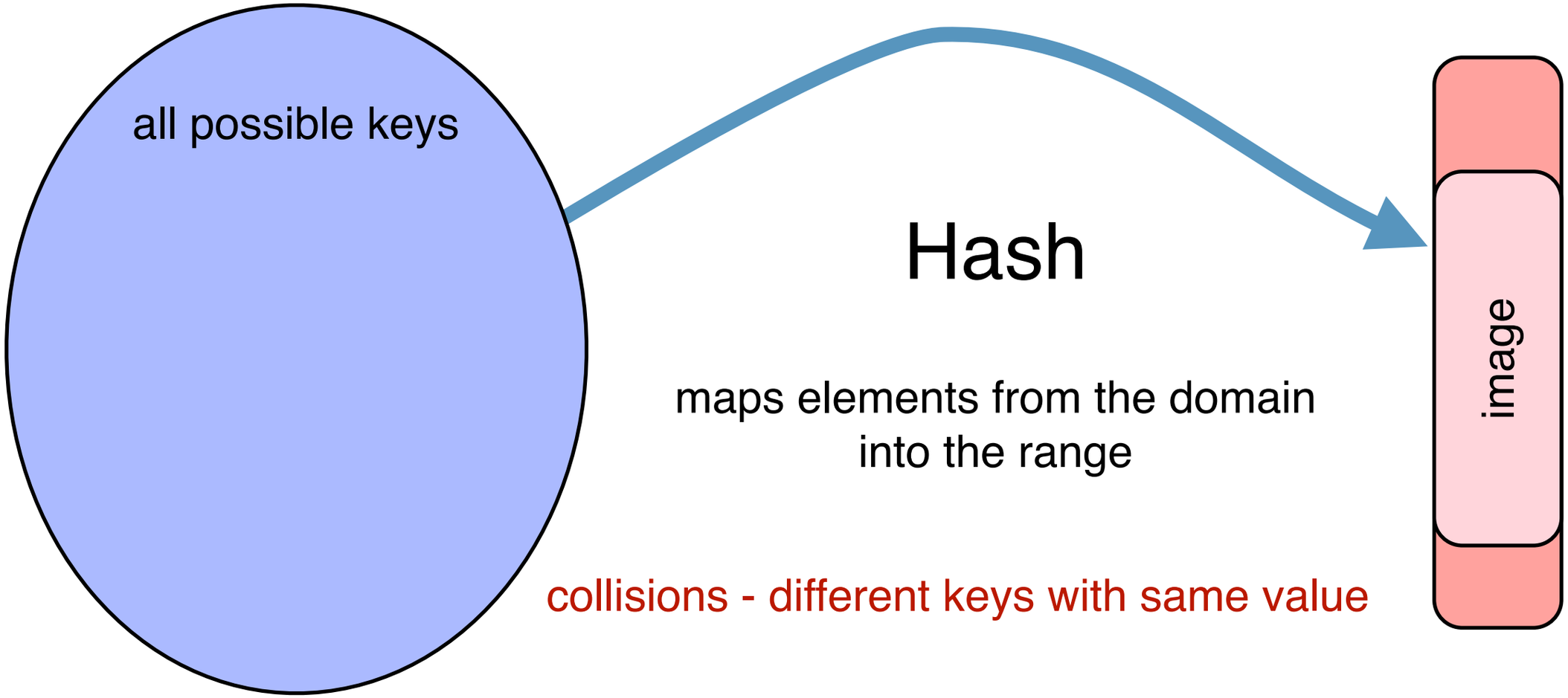
Note: for $x \notin S$, no property is guaranteed about f(x)

Construction of a Perfect Hash Index

Key: We know meaningful sub-patterns ahead of time

Domain (e.g. keys)

Range (e.g. [0, mlDl])



maps elements from the domain
into the range

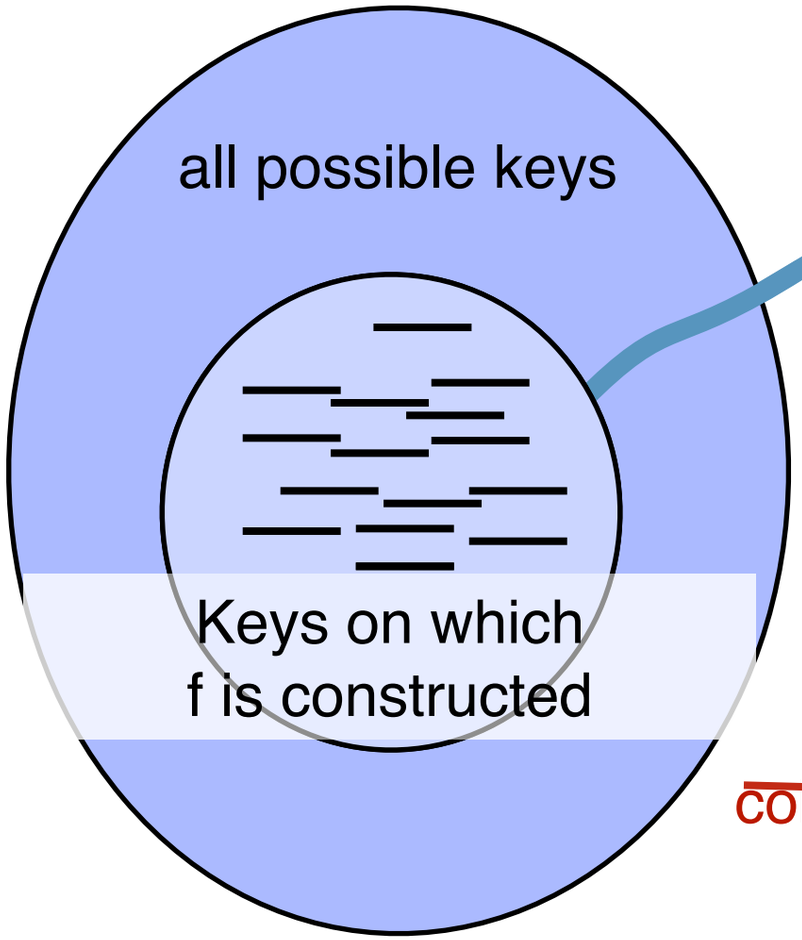
collisions - different keys with same value

Construction of a Perfect Hash Index

Key: We know meaningful sub-patterns ahead of time

Domain (e.g. keys)

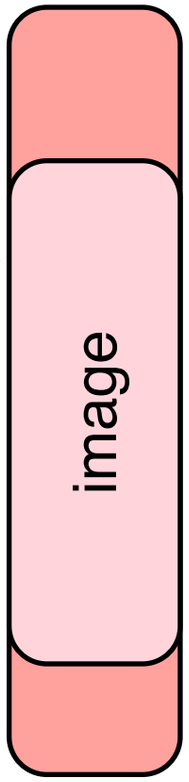
Range (e.g. [0, mlDl])



PerfectHash

maps elements from the domain
into the range

~~collisions - different keys with same value~~

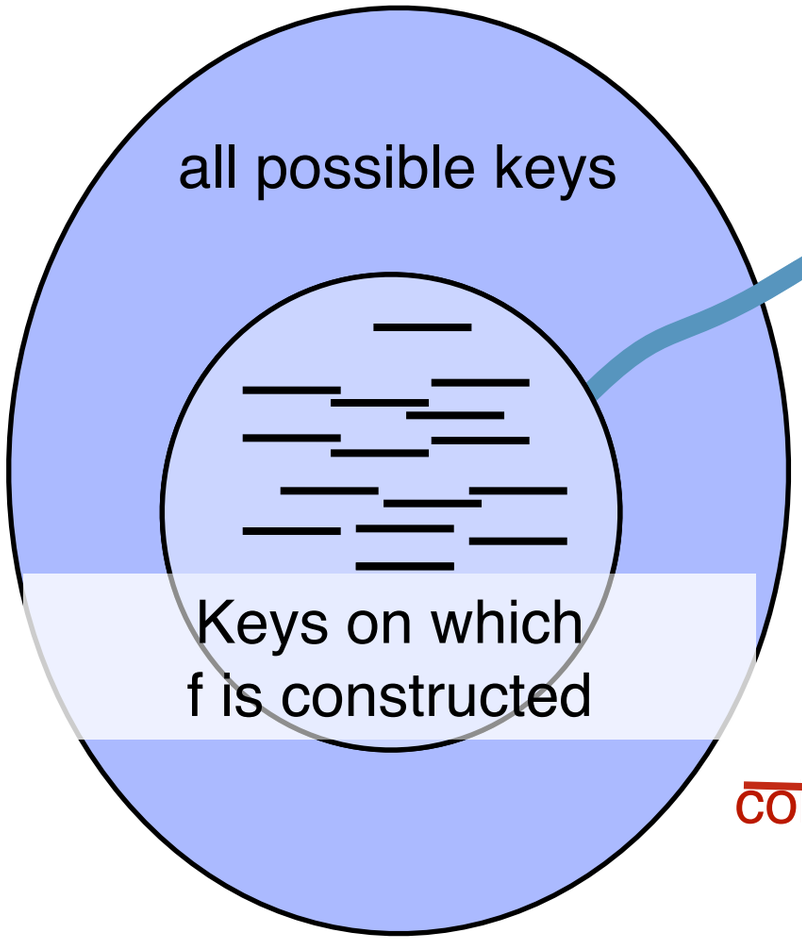


Construction of a Perfect Hash Index

Key: We know meaningful sub-patterns ahead of time

Domain (e.g. keys)

Range (e.g. [0, mlDl])

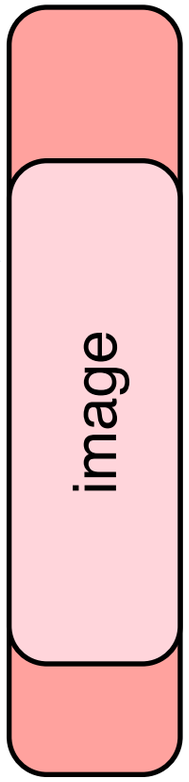


Minimal
PerfectHash

maps elements from the domain
into the range

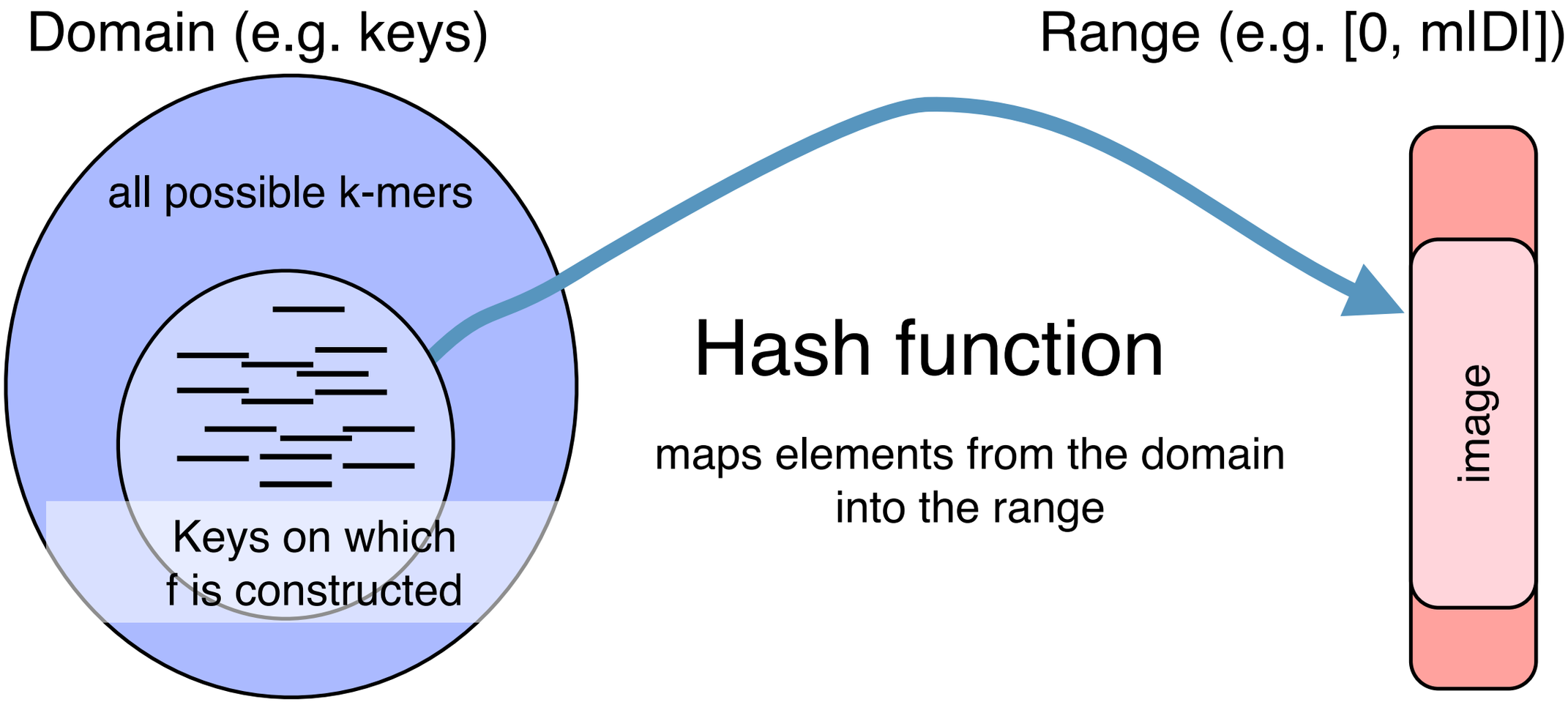
collisions - different keys with same value

maps keys to distinct integers in [0, lDl-1]



Construction of a Perfect Hash Index

Key: We know meaningful sub-patterns ahead of time



Minimal maps keys to distinct integers in [0, IDI-1]

Perfect no collisions - every key maps to its own value

(Minimum) Perfect Hash Functions

We'll talk about BBhash. My *favorite* algorithm for minimal perfect hash construction. It's *not* the most sophisticated algorithm in the literature, but it is by-far the most practical.

https://github.com/rizkg/BBHash

Fast and Scalable Minimal Perfect Hashing for Massive Key Sets*

Antoine Limasset[1], Guillaume Rizk[2], Rayan Chikhi[3], and Pierre Peterlongo[4]

1 IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France
2 IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France
3 CNRS, CRIStAL, Université de Lille, Inria Lille – Nord Europe, Lille, France
4 IRISA Inria Rennes Bretagne Atlantique, GenScale team, Rennes, France

Observation: often MPHF are used to map keys (not stored) to values (stored). The set of values is often much larger, per-element than the MPHF. It's worth spending a few more bits per-element on the MPHF if we can construct it efficiently

Minimum Perfect Hash Functions

Theoretical minimum is $\log_2(e)N \approx 1.44N$ bits / key (regardless of # of keys).

Best methods, in practice, provide just under ~3 bits/key.

This approach provides a parameter γ to trade off between construction speed and final MPHF size and query time.

Successive hashing for construction

For a set of keys F_0 construct a bit-array of size $A_0 = |F_0|$.

Insert the keys into A_0 using hash function $h_0()$

$A_0[i] = 1$ if *exactly* one element from F_0 hashes to i

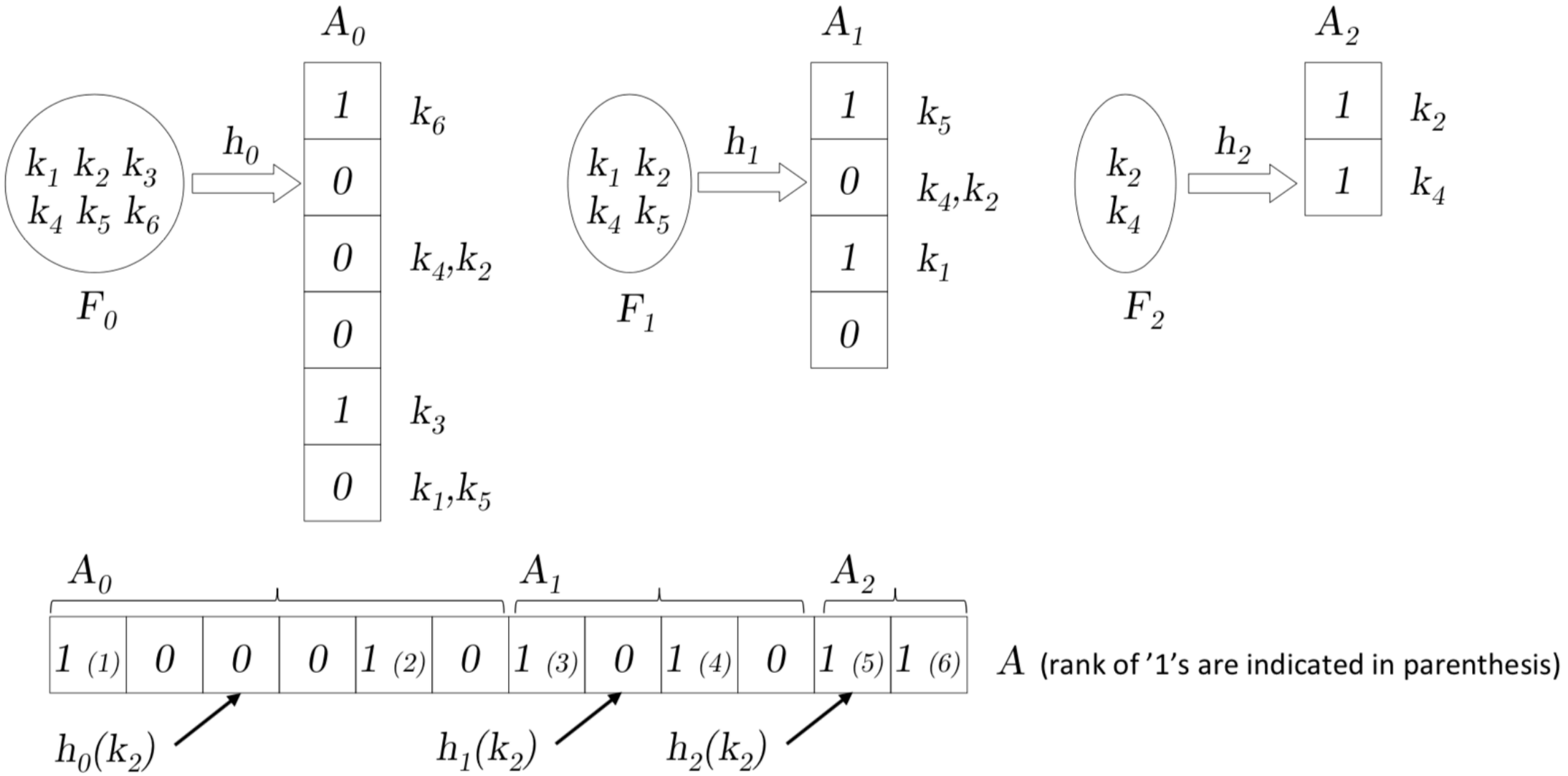
For all keys that collide under $h_0()$, create a new key set F_1 of size $|F_1|$

Create a corresponding new bit vector A_1.

Repeat this process until there are no collisions.

In practice: Repeat this process until F_k is sufficiently small, and use a traditional hash table to store it.

Successive hashing for construction



Detecting Collisions

During construction at each level d, collisions are detected using a temporary bit array C_d of size $|A_d|$. Initially all C_d bits are set to '0'. A bit of $C_d[i]$ is set to '1' if two or more keys from F_d have the same value i given by hash function h_d. Finally, if $C_d[i] = 1$, then $A_d[i] = 0$. Formally:

$$C_d[i] = 1 \Rightarrow A_d[i] = 0;$$

$$(h_d[x] = i \text{ and } A_d[i] = 0 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 1 \,(\text{and } C_d[i] = 0)\,;$$

$$(h_d[x] = i \text{ and } A_d[i] = 1 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 0 \text{ and } C_d[i] = 1.$$

Querying & Minimality

A query of a key x is performed by finding the smallest d such that $A_d[h_d(x)] = 1$. The (non minimal) hash value of x is then $(\sum_{i<d} |F_i|) + h_d(x)$.

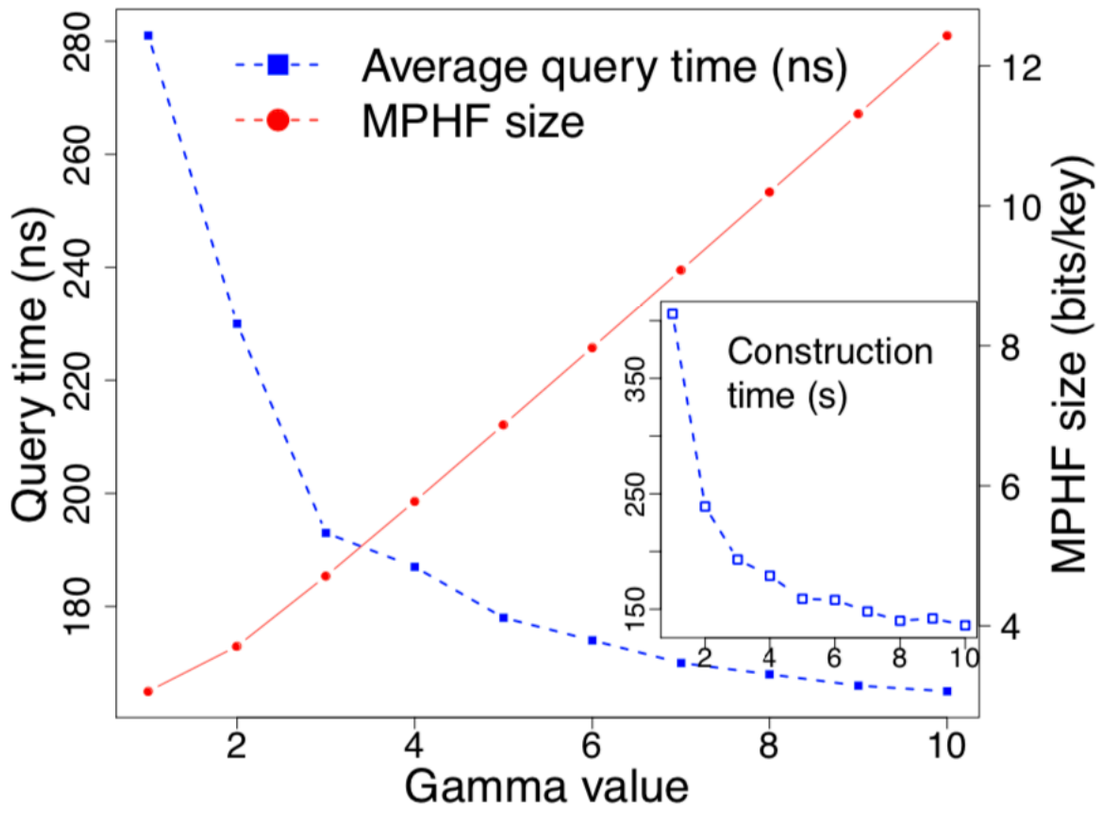
To ensure that the image range of the function is $[1, |F_0|]$, we compute the cumulative rank of each '1' in the bit arrays A_i. Suppose, that d is the smallest value such that $A_d[h_d(x)] = 1$. The minimal perfect hash value is given by $\sum_{i<d}(weight(A_i) + rank(A_d[h_d(x)])$, where $weight(A_i)$ is the number of bits set to '1' in the A_i array, and $rank(A_d[y])$ is the number of bits set to 1 in A_d within the interval $[0, y]$, thus $rank(A_d[y]) = \sum_{j<y} A_d[j]$. This is a classic method also used in other MPHFs [3].

Tradeoff with the γ parameter

The running time of the construction depends on the number of collisions on the A_d arrays, at each level d. One way to reduce the number of collisions, hence to place more keys at each level, is to use bit arrays (A_d and C_d) larger than $|F_d|$. We introduce a parameter $\gamma \in \mathbb{R}$, $\gamma \geq 1$, such that $|C_d| = |A_d| = \gamma|F_d|$. With $\gamma = 1$, the size of A is minimal. With $\gamma \geq 2$, the number of collisions is significantly decreased and thus construction and query times are reduced, at the cost of a larger `MPHF` structure size.

▶ **Lemma 1.** *For $\gamma > 0$, the space of our `MPHF` is $S = \gamma e^{\frac{1}{\gamma}} N$ bits. The maximal space during construction is S when $\gamma \leq \log(2)^{-1}$, and $2S$ bits otherwise.*

Tradeoff with the γ parameter



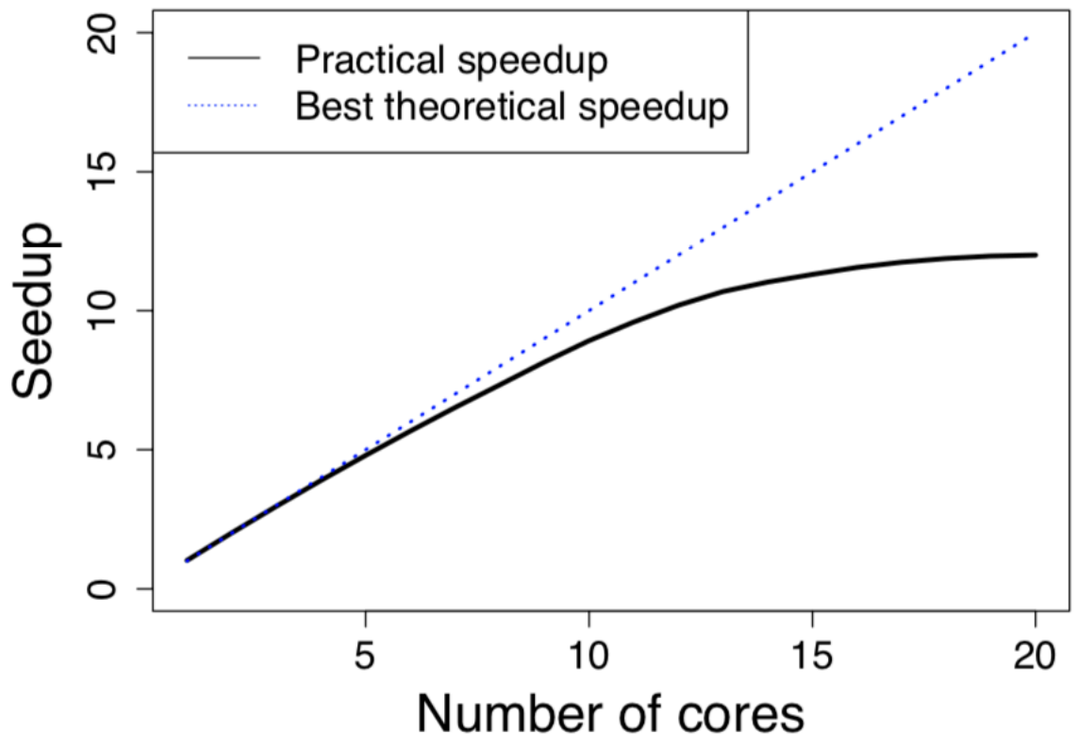
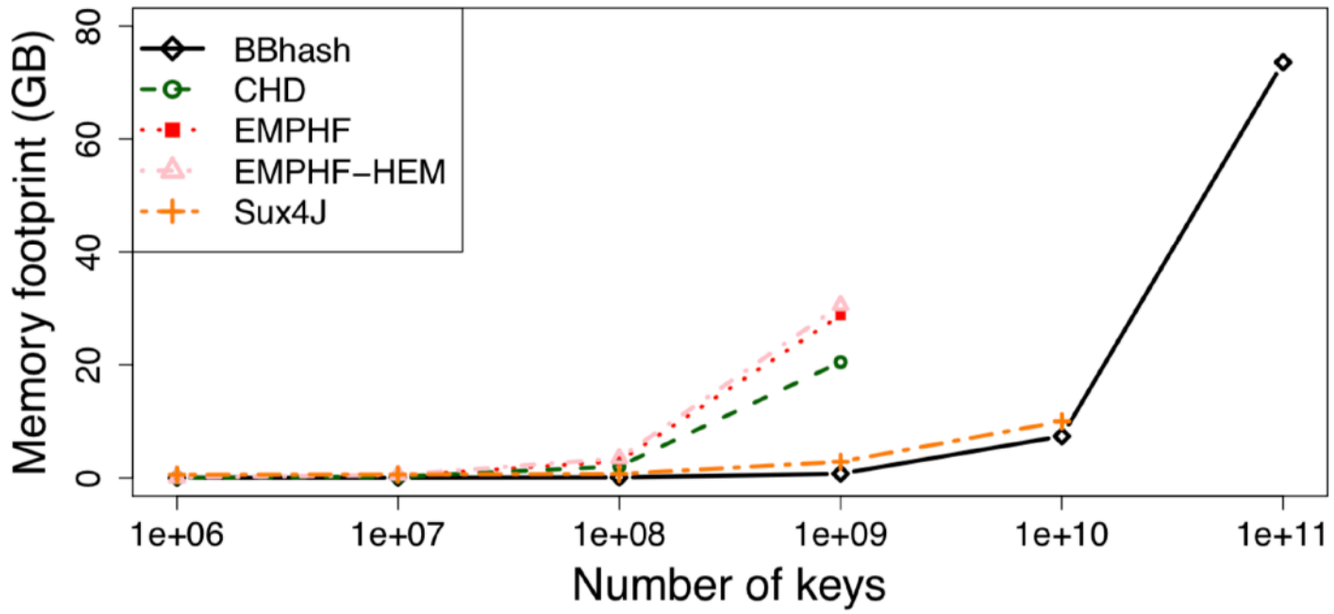


Figure 2 Left: Effects of the gamma parameter on the performance of *BBhash* when run on a set composed of one billion keys, when executed on a single CPU thread. Times and `MPHF` size behave accordingly to the theoretical analysis, respectively $O(e^{(1/\gamma)})$, and $O(\gamma e^{(1/\gamma)})$. Right: Performance of the *BBhash* construction time according to the number of cores, using $\gamma = 2$.

Comparison with other MPHF schemes



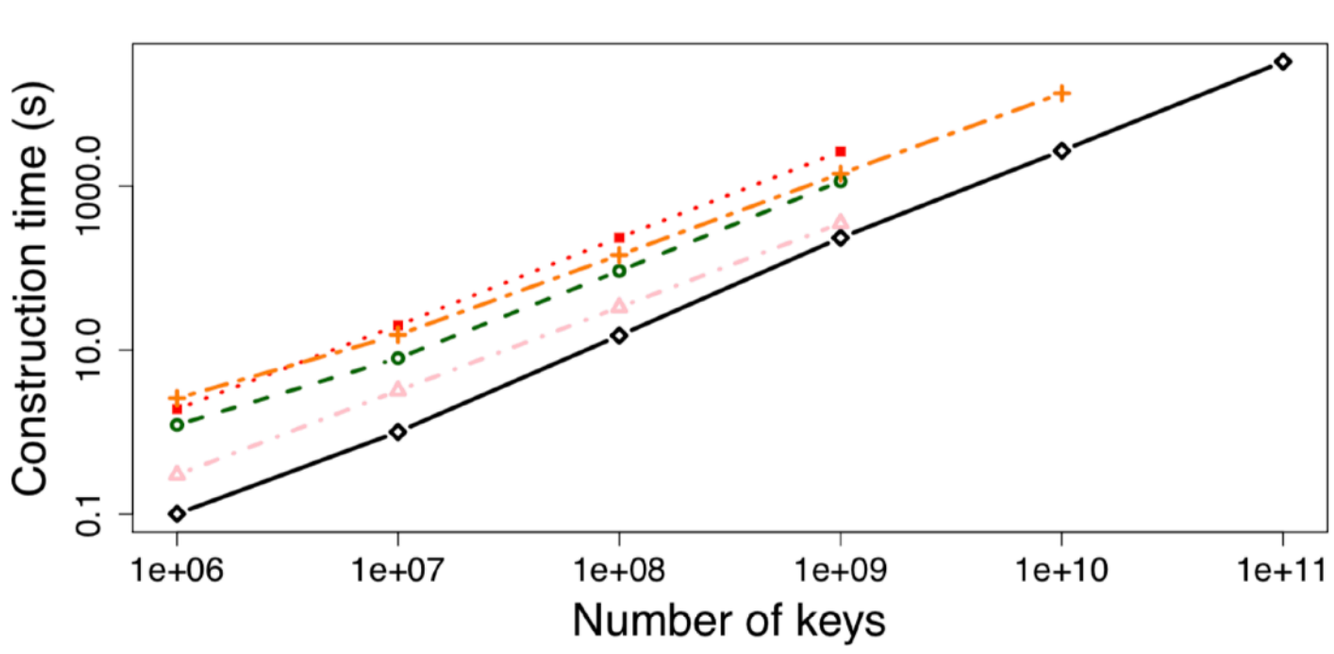


Figure 3 Memory footprint and construction time with respect to the number of keys. All libraries were run using default parameters, including $\gamma = 2$ for *BBhash*. For a fair comparison, *BBhash* was executed on a single CPU thread. Except for Sux4J, missing data points correspond to runs that exceeded the amount of available RAM. Sux4J limit comes from the disk usage, estimated at approximately 4TB for 10^{11} keys.

Comparison with other MPHF schemes

Method	Query time (ns)	`MPHF` size (bits/key)	Const. time* (s)	Const. memory**	Disk. usage (GB)
BBhash $\gamma = 1$	271	3.1	60 (393)	3.2 (376)	8.23
BBhash $\gamma = 1$ minirank	279	2.9	61(401)	3.2 (376)	8.23
BBhash $\gamma = 2$	216	3.7	35 (229)	4.3 (516)	4.45
BBhash $\gamma = 2$ nodisk	216	3.7	80 (549)	6.2 (743)	0
BBhash $\gamma = 5$	179	6.9	25 (162)	10.7 (1,276)	1.52
EMPHF	246	2.9	2,642	247.1 (29,461)†	20.8
EMPHF HEM	581	3.5	489	258.4 (30,798)†	22.5
CHD	1037	2.6	1,146	176.0 (20,982)	0
Sux4J	252	3.3	1,418	18.10 (2,158)	40.1

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!