

# CSE 549: Efficiently Dealing with k-mers

# 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 parameter  $k$

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

This is the  *$k$ -mer counting* problem

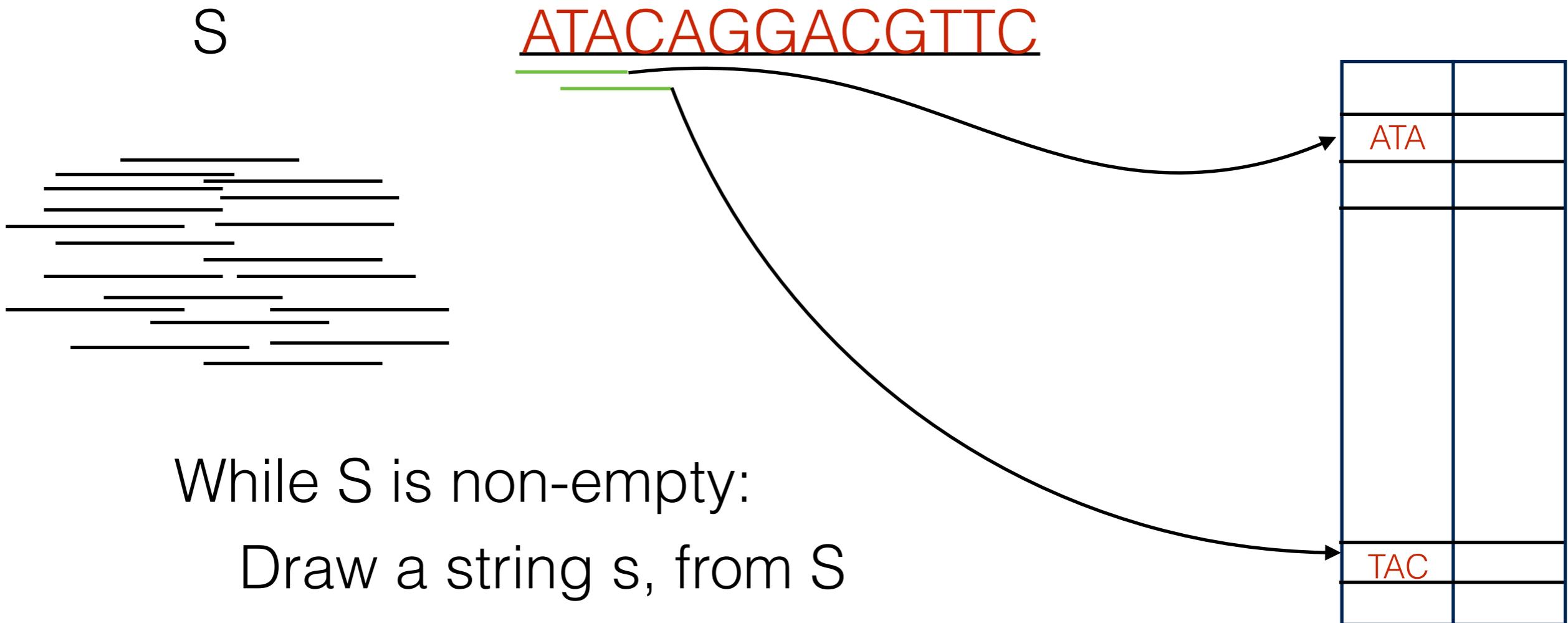
# k-mer counting

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

Tallymer, Jellyfish, DSK, KMC, BFCounter,  
scTurtle, KAnalyze, khmer, ... and many more

# How might we count k-mers

A naive approach:



While  $S$  is non-empty:

    Draw a string  $s$ , from  $S$

    For every k-mer,  $k$  in  $s$ :

$\text{counts}[k] += 1$

# What's wrong with this approach?

Speed & Memory usage

Routinely encounter datasets with  $10 - 100 \times 10^9$  nucleotides

Just hashing the k-mers and resolving collisions takes time

On the order of  $1-10 \times 10^9$  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  $\text{hash}(k) = f(k) \bmod M$
- 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  $\text{hash}(k) = f(k) \bmod M$
- 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)

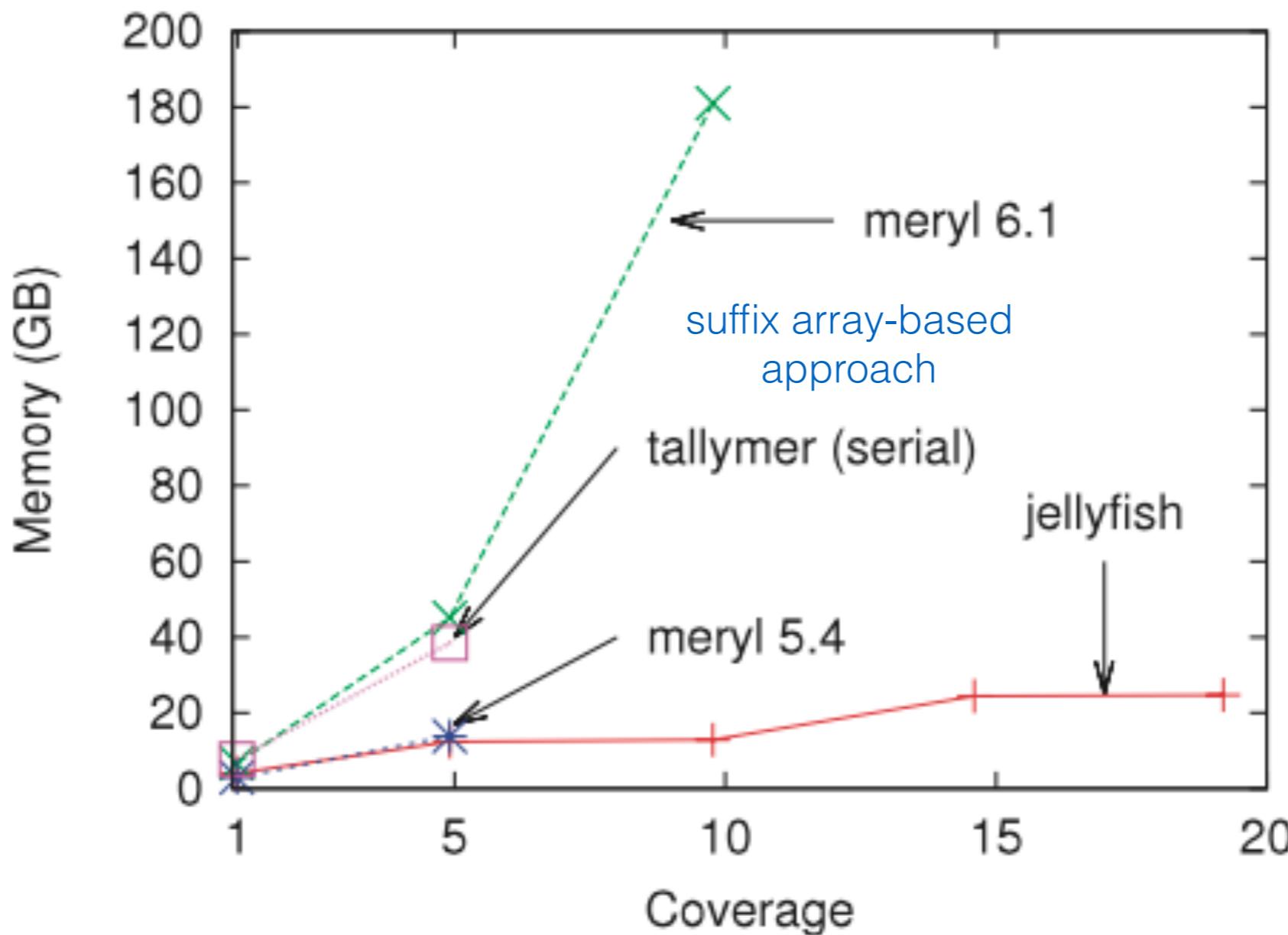
## Squeakr: An Exact and Approximate $k$ -mer Counting System

Prashant Pandey , Michael A Bender, Rob Johnson, Rob Patro [Author Notes](#)

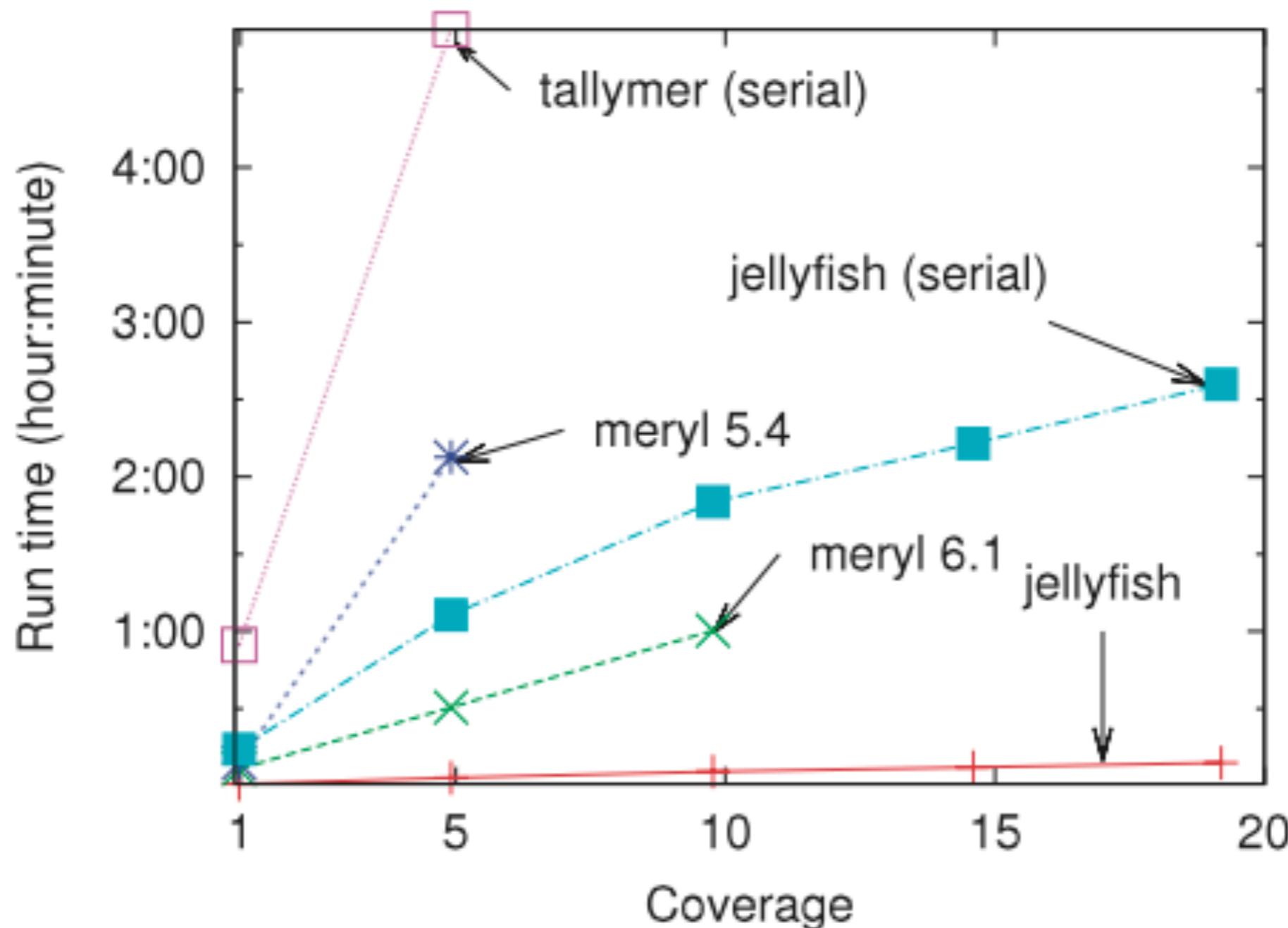
*Bioinformatics*, btx636, <https://doi.org/10.1093/bioinformatics/btx636>

**Published:** 09 October 2017 [Article history](#) ▾

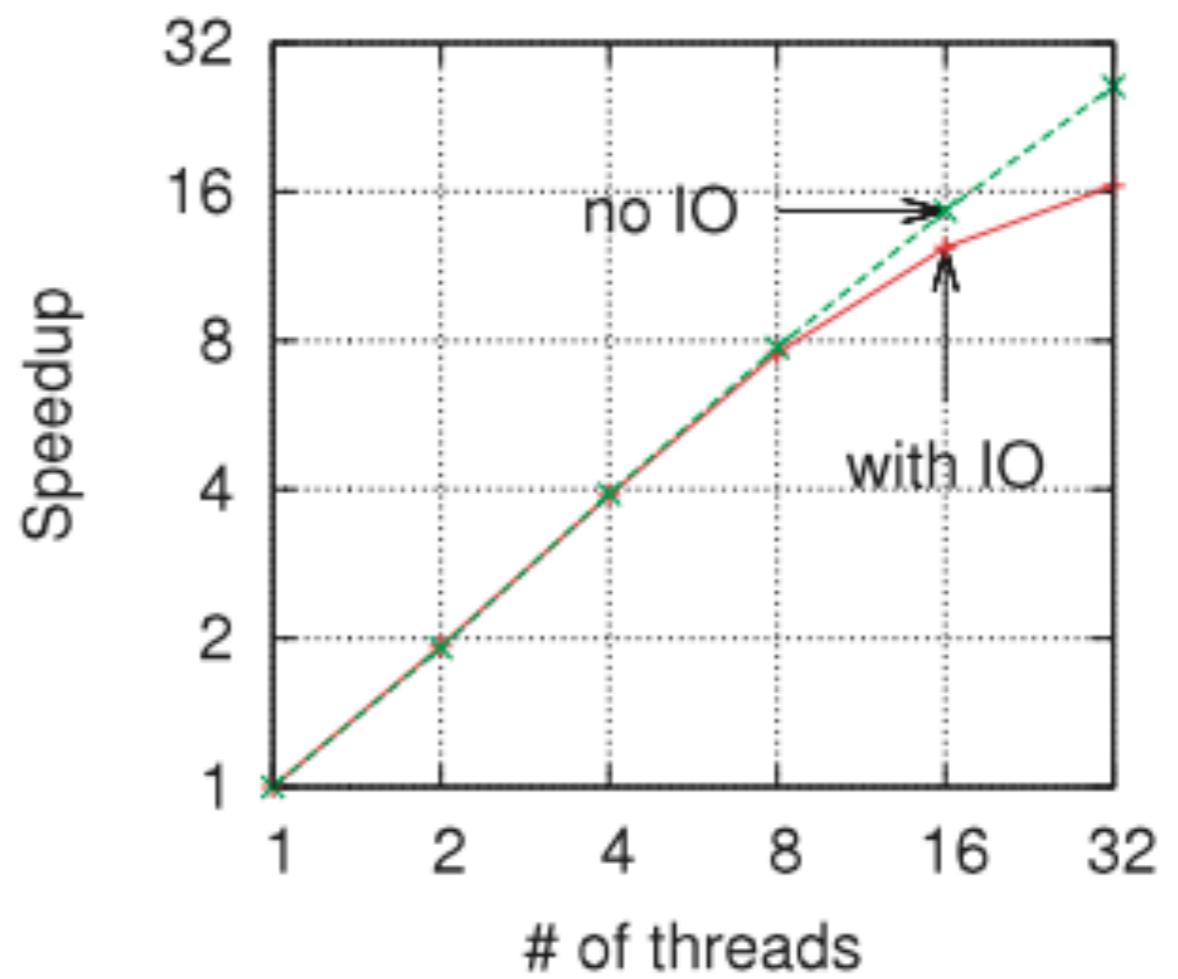
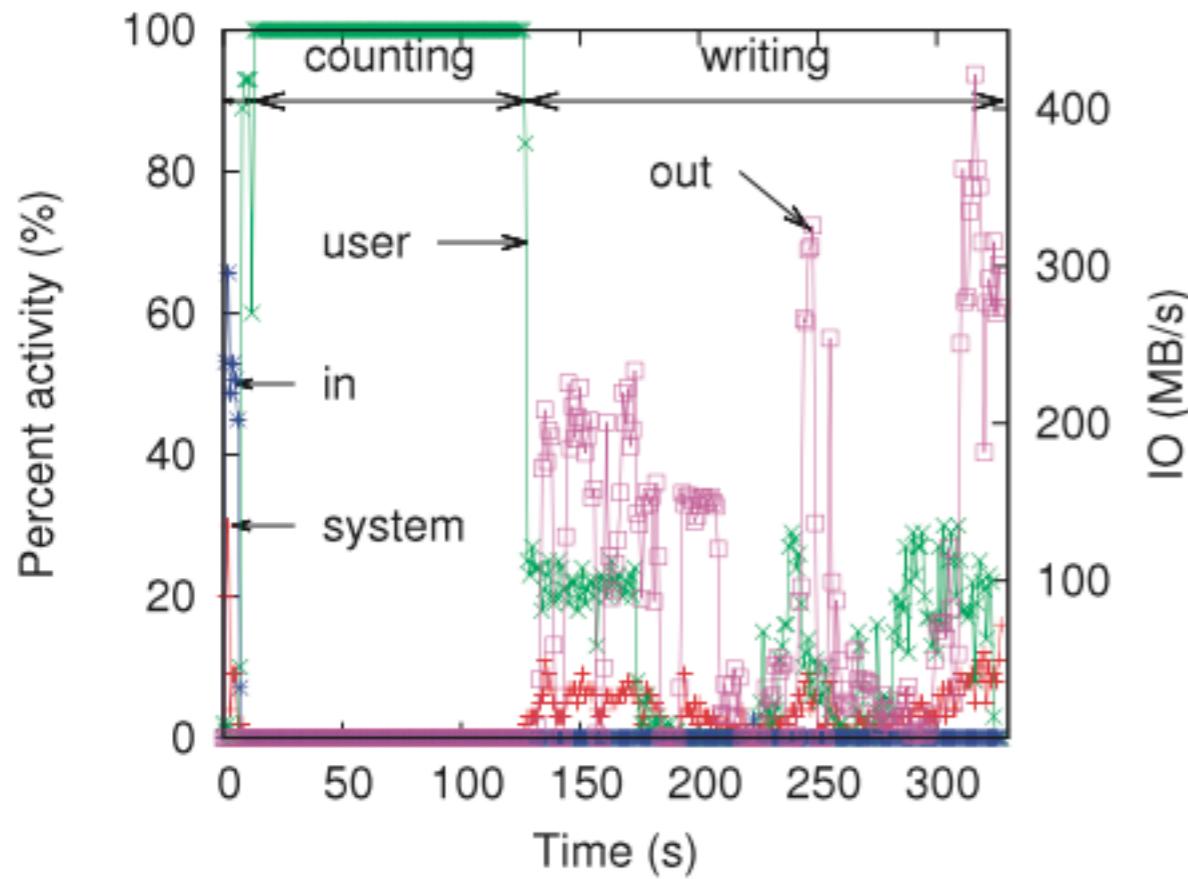
# Memory usage of Jellyfish



# 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,  $\{h_0, \dots, h_{k-1}\}$

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  $0 < i < k$

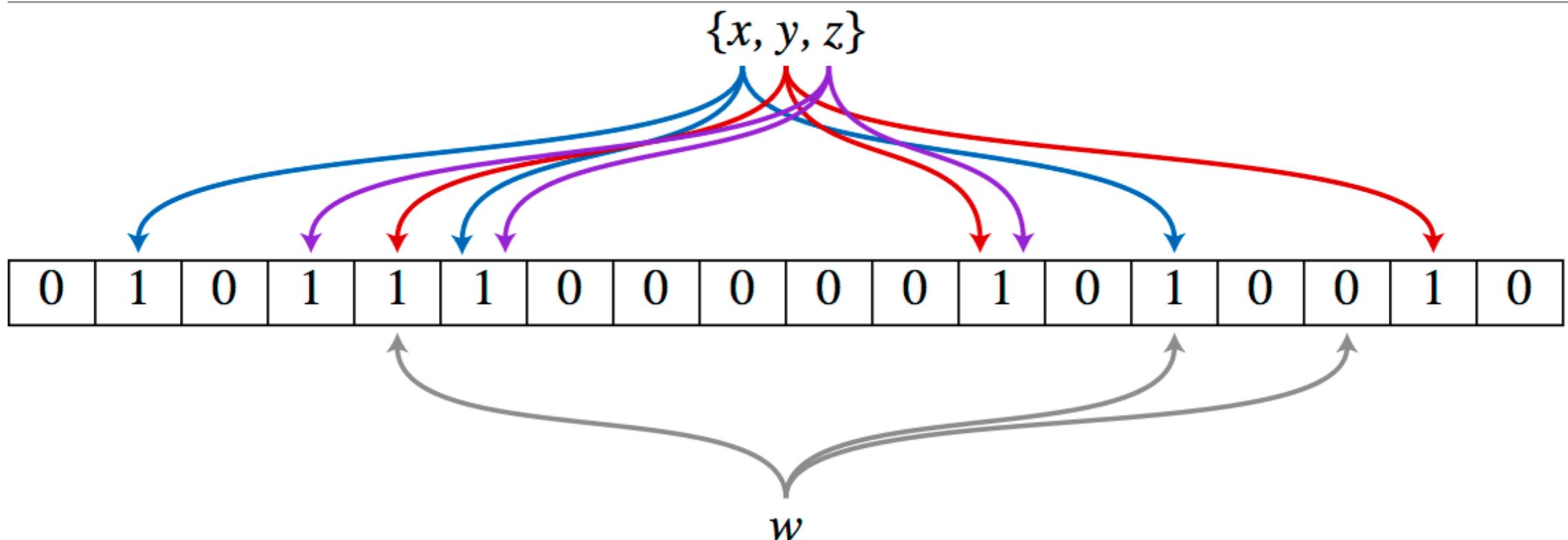


Image by David Eppstein - self-made, originally for a talk at WADS 2007

# Bloom Filters

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

How low?

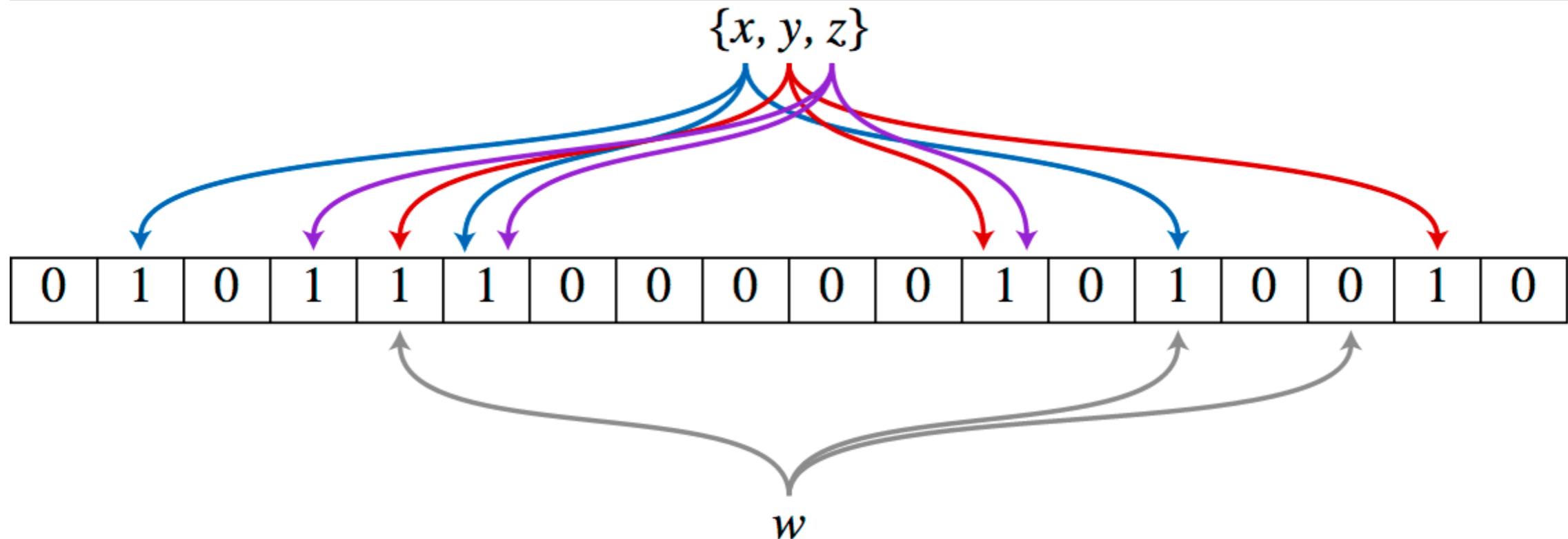


Image by David Eppstein - self-made, originally for a talk at WADS 2007

# 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:

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

# False Positives

Mitzenmacher & Upfal used the Azuma-Hoeffding inequality 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:

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

# 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

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

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

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

# 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  $k$

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

We can then compute the false positive prob

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

given an **expected**  
**# elems**

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

and a desired  
false positive rate

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

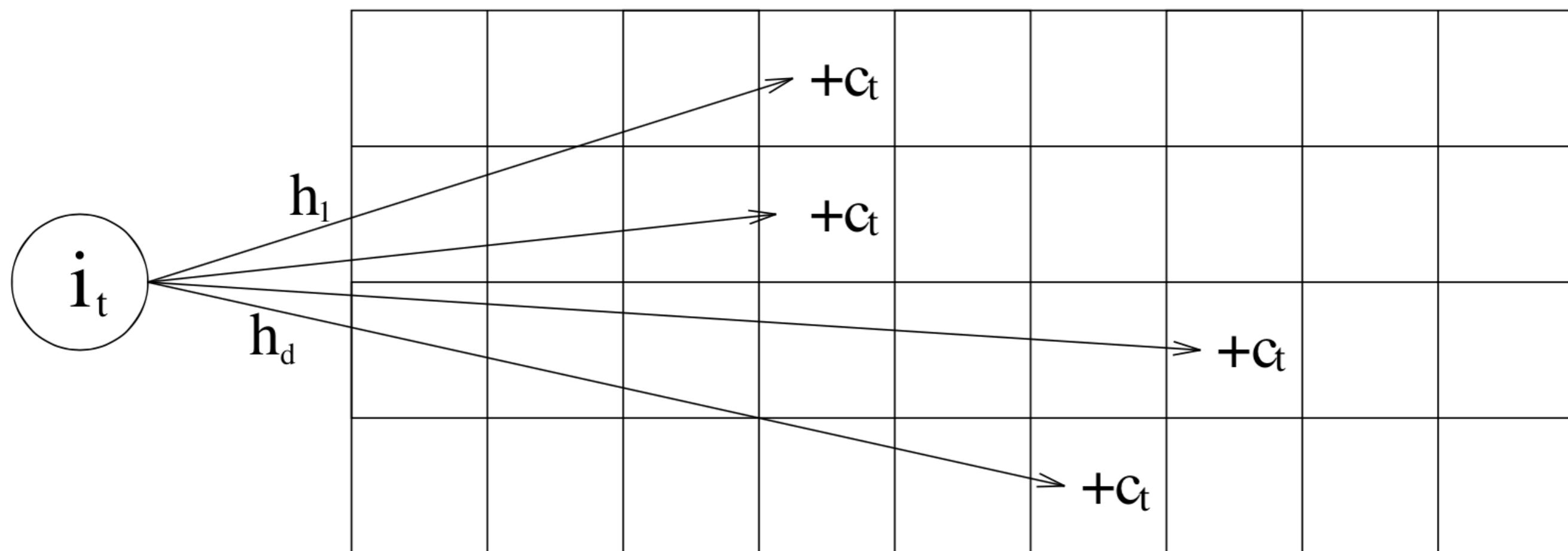
we can compute  
the **optimal size** and  
**# of has functions**

# Probabilistic Data Structures & k-mer Counting

Some recent methods apply Bloom filters or related ideas to the problem of k-mer counting. One such method is khmer, which uses the count-min sketch data structure.

# Probabilistic Data Structures & k-mer Counting

Instead of a an array of  $m$ -bits, store a 2D, array,  $CM$ , of size  $d \times w$  —  $d$  is called the depth of the array, and there are  $d$  *independent* hash functions,  $w$  is called width of the array. This is an  $O(wd)$  data structure.



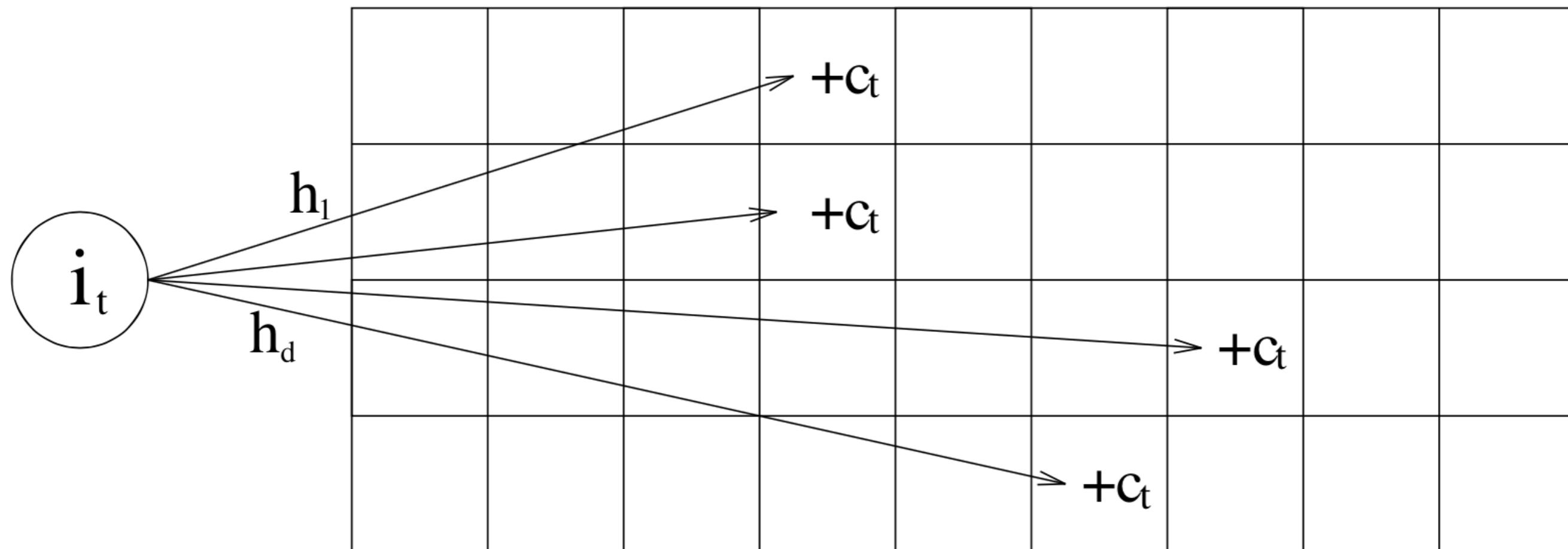
# Probabilistic Data Structures & k-mer Counting

Like Bloom filters, 2 mains operations:

Update ( $k, v$ ) — for each entry  $CM[i, h_i(k)]$ , where  $0 < i < d$ , increment the value by  $v$ .

Query ( $k$ ) — compute  $v = \min_{0 < i < d} CM[i, h_i(k)]$

Both are  $O(d)$  operations



# Probabilistic Data Structures & k-mer Counting

Similar error analysis to Bloom filters (won't prove bounds)

Let  $\hat{a}_i$  be the result returned by  $\text{Query}(i)$ . We have that:

$$a_i \leq \hat{a}_i \quad (\text{always})$$

$$\hat{a}_i \leq a_i + \epsilon \|\mathbf{a}\|_1 \quad (\text{with probability at least } \frac{1}{\delta})$$

where,

$$w = \left\lceil \frac{e}{\epsilon} \right\rceil, d = \left\lceil \ln\left(\frac{1}{\delta}\right) \right\rceil, \text{ and } \|\mathbf{a}\|_1 = \sum_{i=1}^n |a_i|$$

# Probabilistic Data Structures & k-mer Counting

Similar error analysis to Bloom filters (won't prove bounds)

Let  $\hat{a}_i$  be the result returned by  $\text{Query}(i)$ . We have that:

$$a_i \leq \hat{a}_i \quad (\text{always})$$

$$\hat{a}_i \leq a_i + \epsilon \|\mathbf{a}\|_1 \quad (\text{with probability at least } \frac{1}{\delta})$$

where,

$$w = \left\lceil \frac{e}{\epsilon} \right\rceil, d = \left\lceil \ln\left(\frac{1}{\delta}\right) \right\rceil, \text{ and } \|\mathbf{a}\|_1 = \sum_{i=1}^n |a_i|$$

base of nat. log

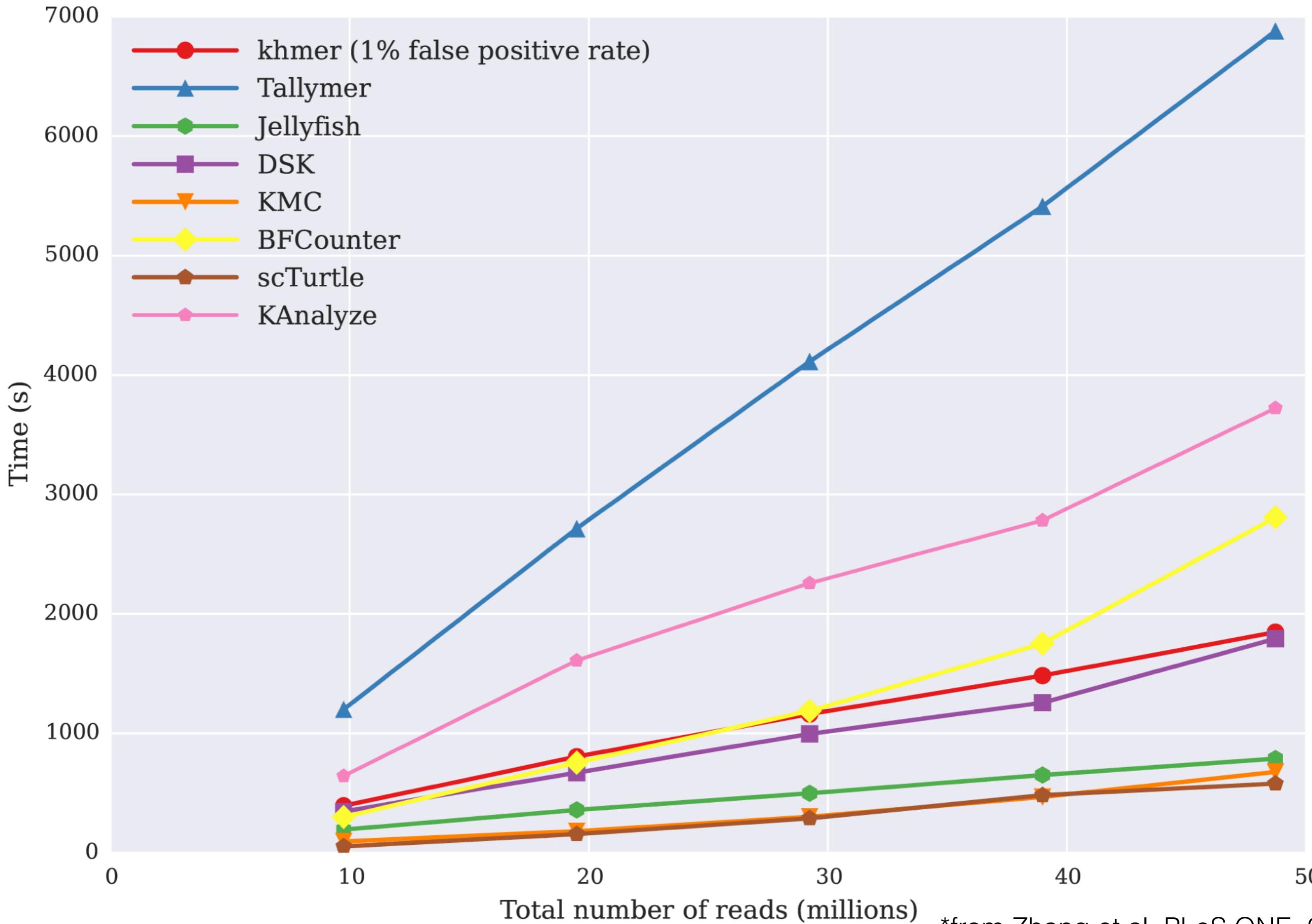
# The Count-Min Sketch for k-mer counting

This approach is used in the k-mer counting software khmer

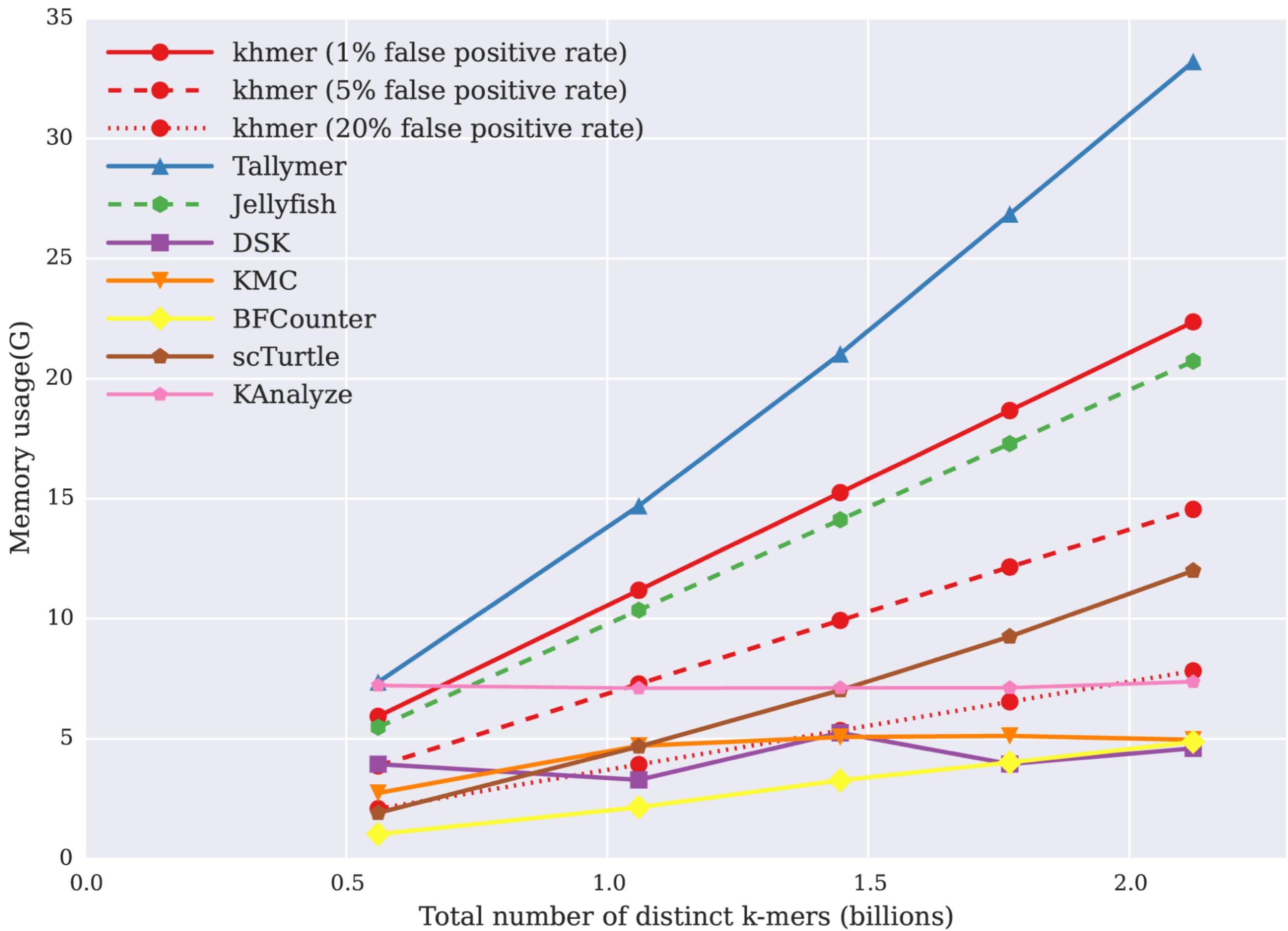
No exact data structure is maintained, just a CMS

This allows for answering approximate count queries efficiently.

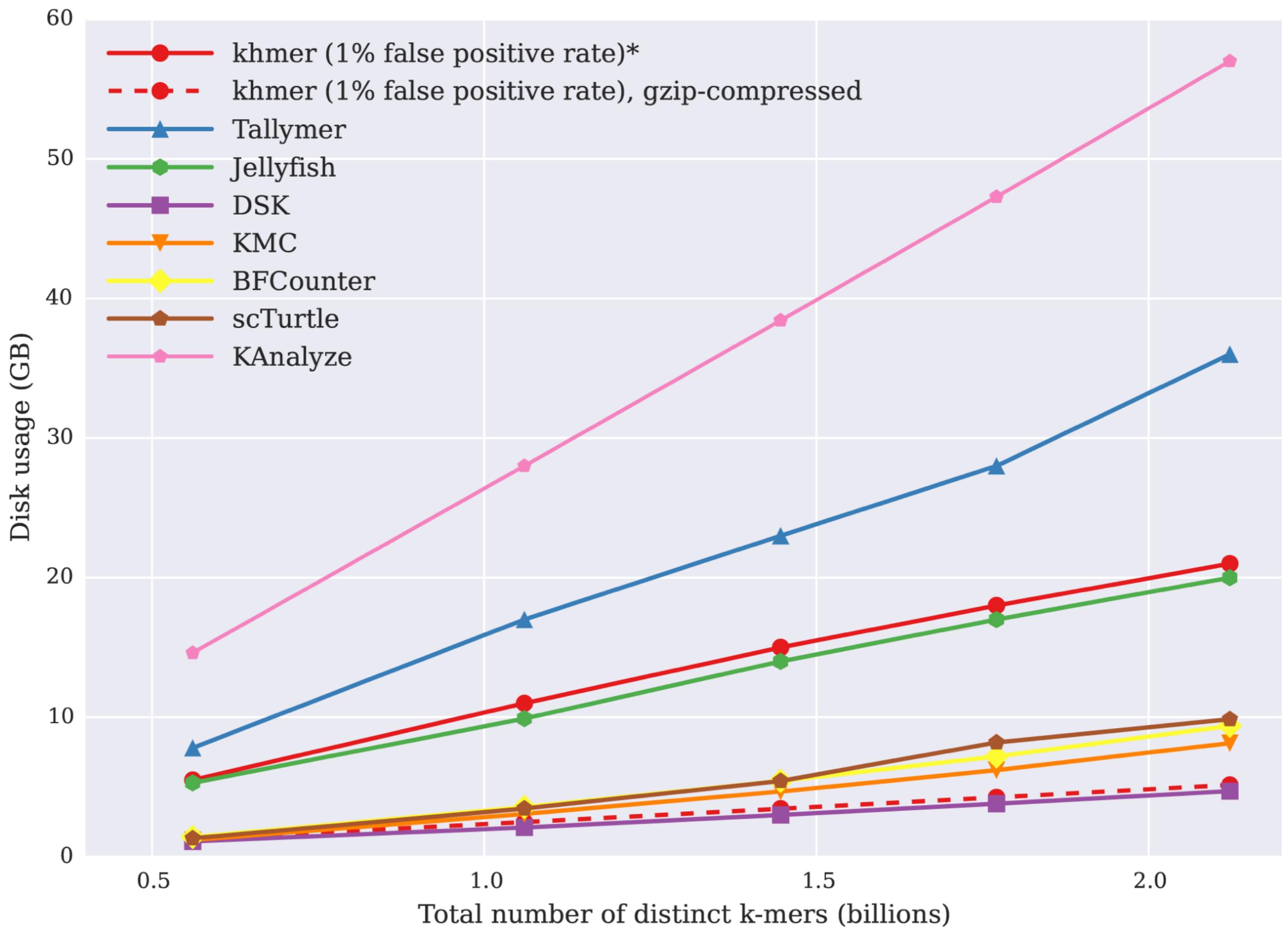
Authors compared to a large number of other k-mer counters under several different metrics.



\*from Zhang et al. PLoS ONE, 2014

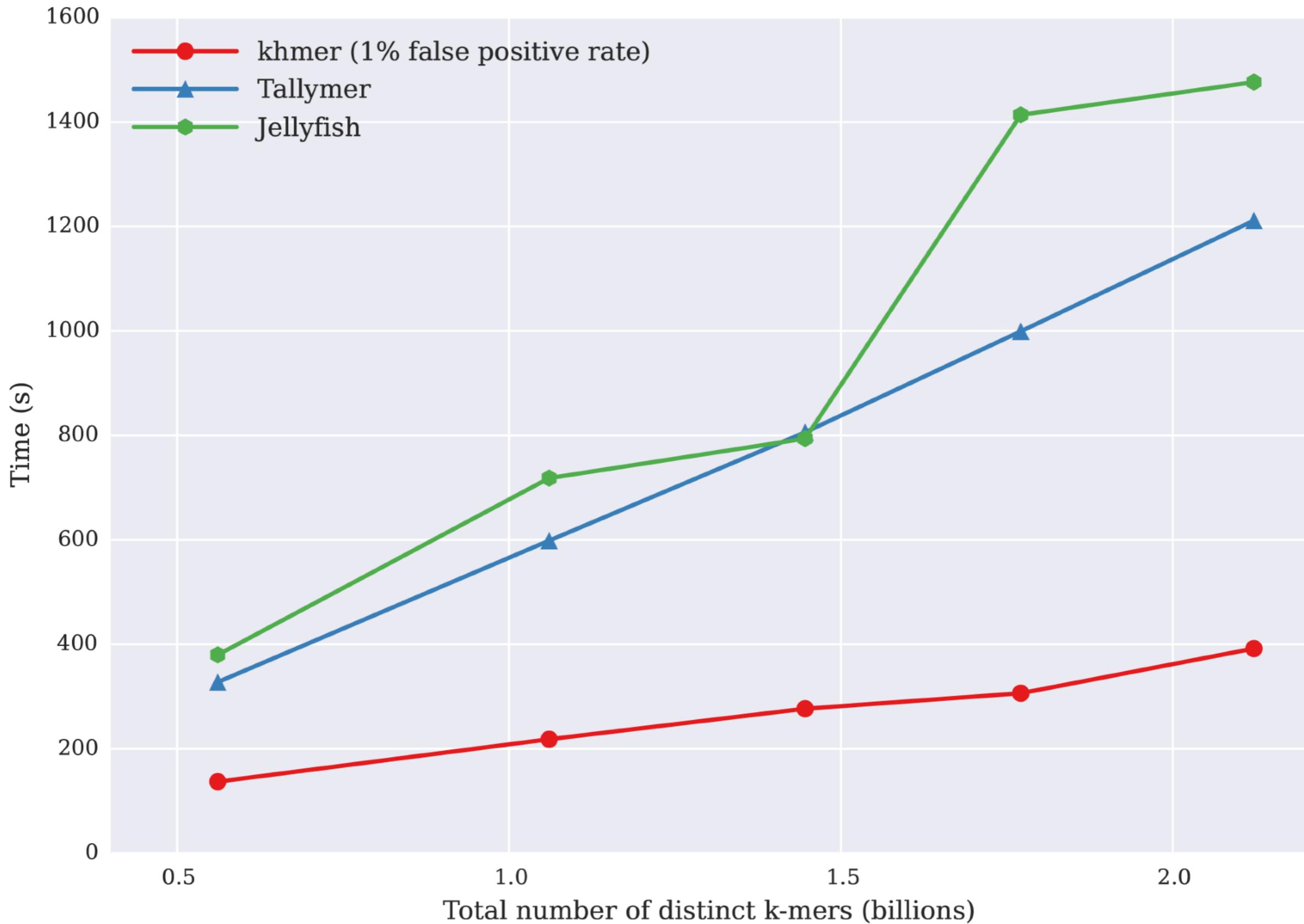


\*from Zhang et al. PLoS ONE, 2014



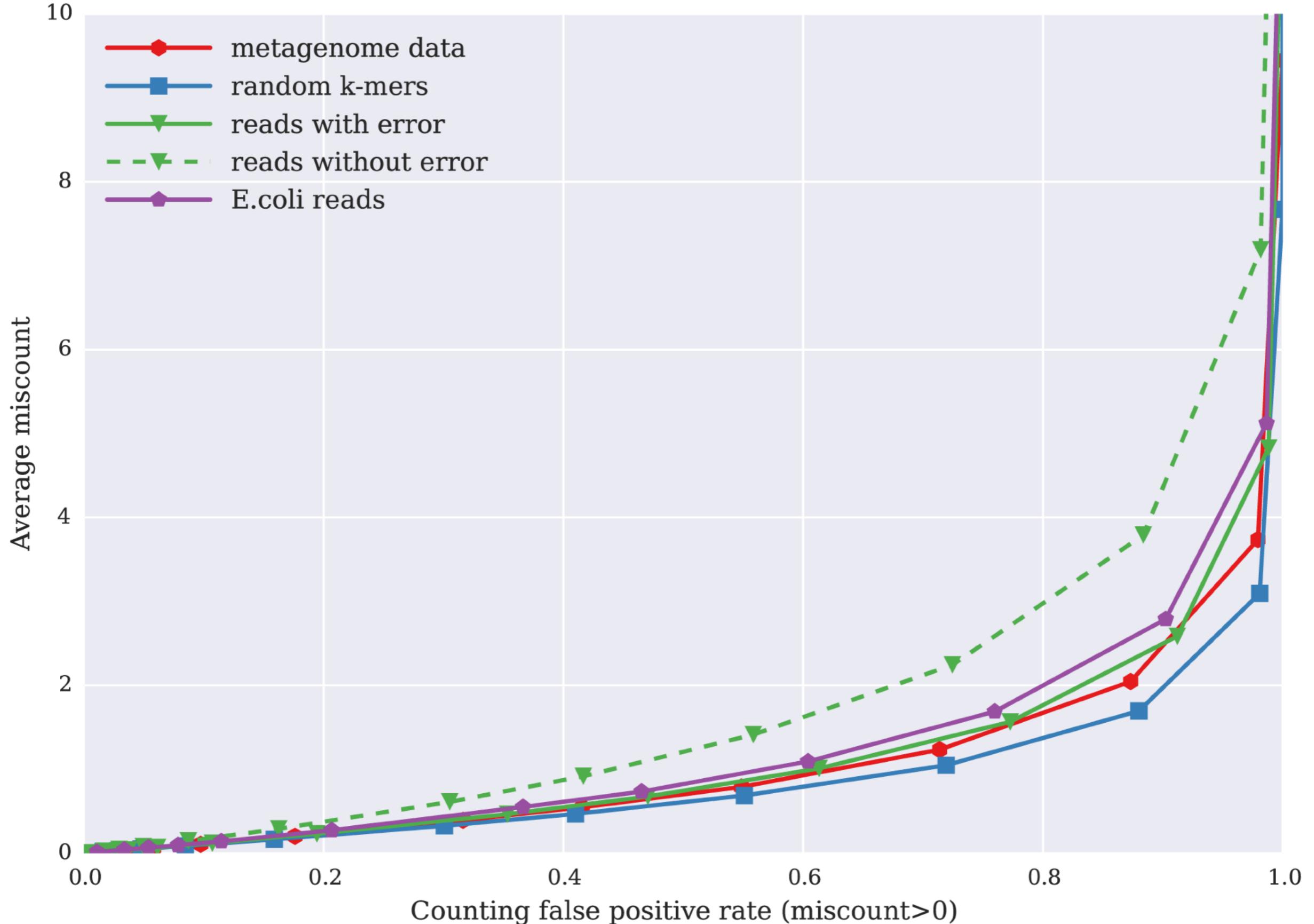
\*from Zhang et al. PLoS ONE, 2014

# Querying for random k-mers

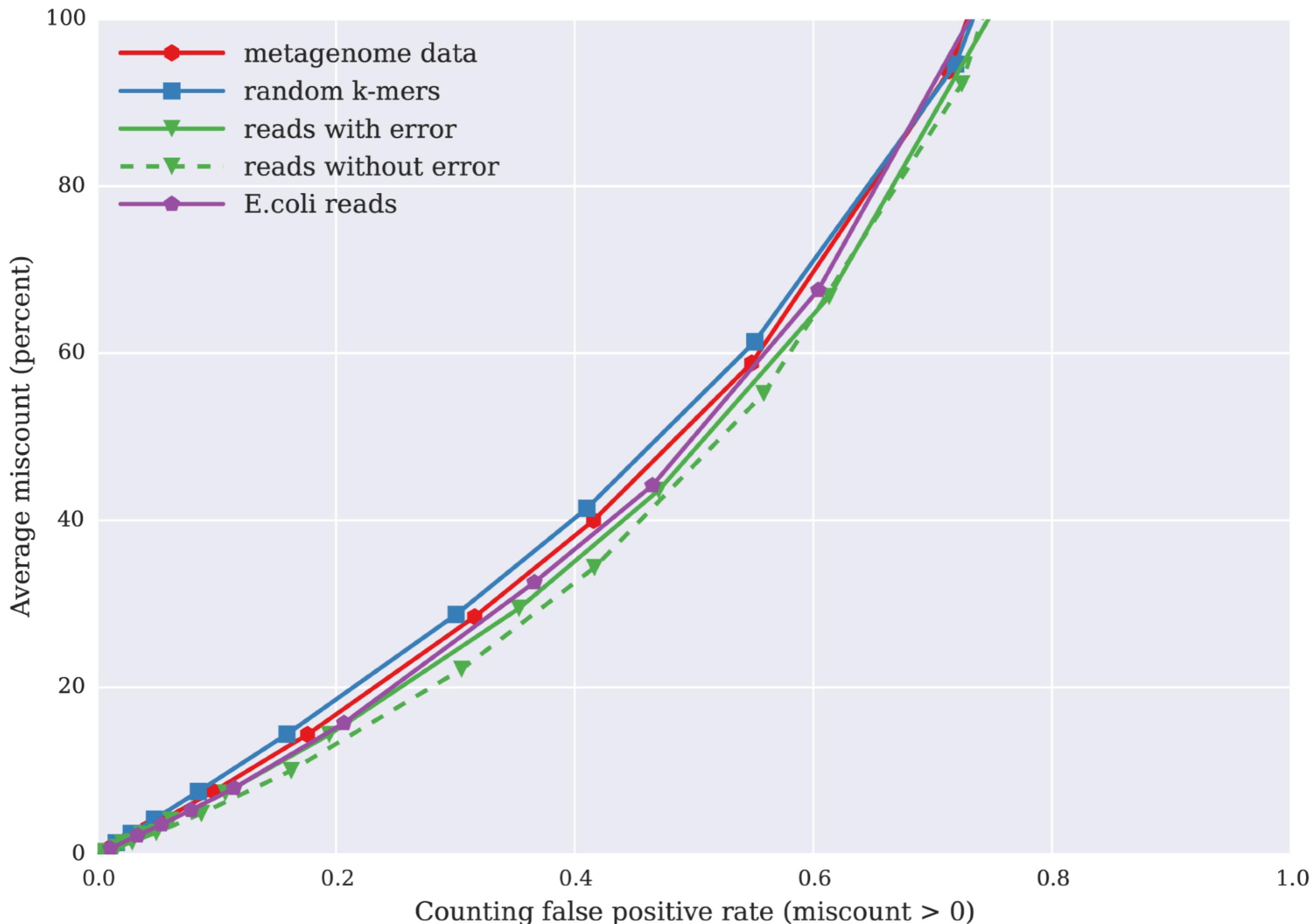


\*from Zhang et al. PLoS ONE, 2014

# Miscount & FP rate; changing $\epsilon$ and $\delta$



# Miscount & FP-rate



\*from Zhang et al. PLoS ONE, 2014

# Other uses of this approach

Khmer has been used successfully for other tasks e.g.  
digital normalization:

diginorm algo:

```
for read in dataset:  
    if estimated_coverage(read) < C:  
        accept(read)  
    else:  
        discard(read)
```

median k-mer abundance of  
k-mers in the read

memory	FP rate	retained reads	retained reads %	true k-mers missing	total k-mers
before diginorm	-	5,000,000	100.0%	170	41.6 m
2400 MB	0.0%	1,656,518	33.0%	172	28.1 m
240 MB	2.8%	1,655,988	33.0%	172	28.1 m
120 MB	18.0%	1,652,273	33.0%	172	28.1 m
60 MB	59.1%	1,633,182	32.0%	172	27.9 m
40 MB	83.2%	1,602,437	32.0%	172	27.6 m
20 MB	98.8%	1,460,936	29.0%	172	25.7 m
10 MB	100.0%	1,076,958	21.0%	185	20.9 m

The results of digitally normalizing a 5 m read *E. coli* data set (1.4 GB) to C = 20 with k = 20 under several memory usage/false positive rates. The false positive rate (column 1) is empirically determined. We measured reads remaining, number of “true” k-mers missing from the data at each step, and the number of total k-mers remaining. Note: at high false positive rates, reads are erroneously removed due to inflation of k-mer counts.  
doi:10.1371/journal.pone.0101271.t004

# Cardinality Estimation

Consider yet one more related problem.

**Given:** A collection of sequencing reads  $S$  and parameter  $k$  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



---

Sequence analysis

## ntCard: a streaming algorithm for cardinality estimation in genomics data

**Hamid Mohamadi<sup>1,2,\*</sup>, Hamza Khan<sup>1,2</sup> and Inanc Birol<sup>1,2,\*</sup>**

<sup>1</sup>Canada's Michael Smith Genome Sciences Centre, British Columbia Cancer Agency, Vancouver, BC, V5Z 4S6, Canada and <sup>2</sup>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).

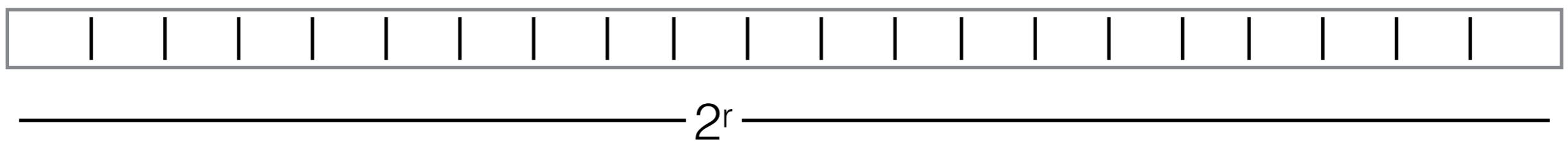
# Basic idea: Hash the k-mers



**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  $r + s < 64$

Use these bits to sub-sample input data “uniformly”. Only process a k-mer if uppermost *s* bits are 0. Sub-sampling at a rate of  $\frac{1}{2^s}$

Maintain an array of size  $2^r$  and, count the number of occurrences of each *r*-bit pattern



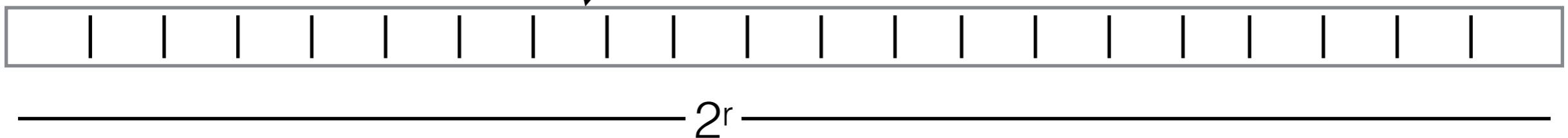
When we encounter a k-mer's hash:

00000000	...	001000
----------	-----	--------

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  $t^{(r+1)}$ ?

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

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  $t_{2^r+i}^{(r+1)} = 0$

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

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

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:

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

This rule can be generalized

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

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

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

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

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

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

and for  $i=2$  as

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

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

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

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

**UGLY!**

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

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

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  $\lim_{x \rightarrow \infty} (r + x)$  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  $i$ .

We can also estimate the 0th order moment as

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

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:        $h \leftarrow \text{ntHash}(k\text{-mer})$   $\triangleright$  Compute 64-bit  $h$  using ntHash
5:       if  $h_{64:64-s+1} = 0^s$  then  $\triangleright$  Checking the  $s$  left bit in  $h$ 
6:          $i \leftarrow h_{r:1}$   $\triangleright r$  is resolution parameter
7:          $t_i \leftarrow t_i + 1$ 
8: function Estimate
9:   for  $i \leftarrow 1$  to  $2^r$  do
10:     $p_{t[i]} \leftarrow p_{t[i]} + 1$ 
11:   for  $i \leftarrow 1$  to  $t_{max}$  do
12:      $p_i \leftarrow p_i / 2^r$ 
13:      $F_0 = -\ln p_0 \times 2^{s+r}$   $\triangleright F_0$  estimate
14:   for  $i \leftarrow 1$  to  $t_{max}$  do
15:      $\hat{f}_i \leftarrow \frac{-p_i}{p_0 \ln p_0} - \frac{1}{i} \sum_{j=1}^{i-1} \frac{j p_{i-j} \hat{f}_j}{p_0}$   $\triangleright$  Relative estimates
16:   for  $i \leftarrow 1$  to  $t_{max}$  do
17:      $f_i \leftarrow \hat{f}_i \times F_0$   $\triangleright f_i$  estimates
18:   return  $f, F_0$ 
```

---

# 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.

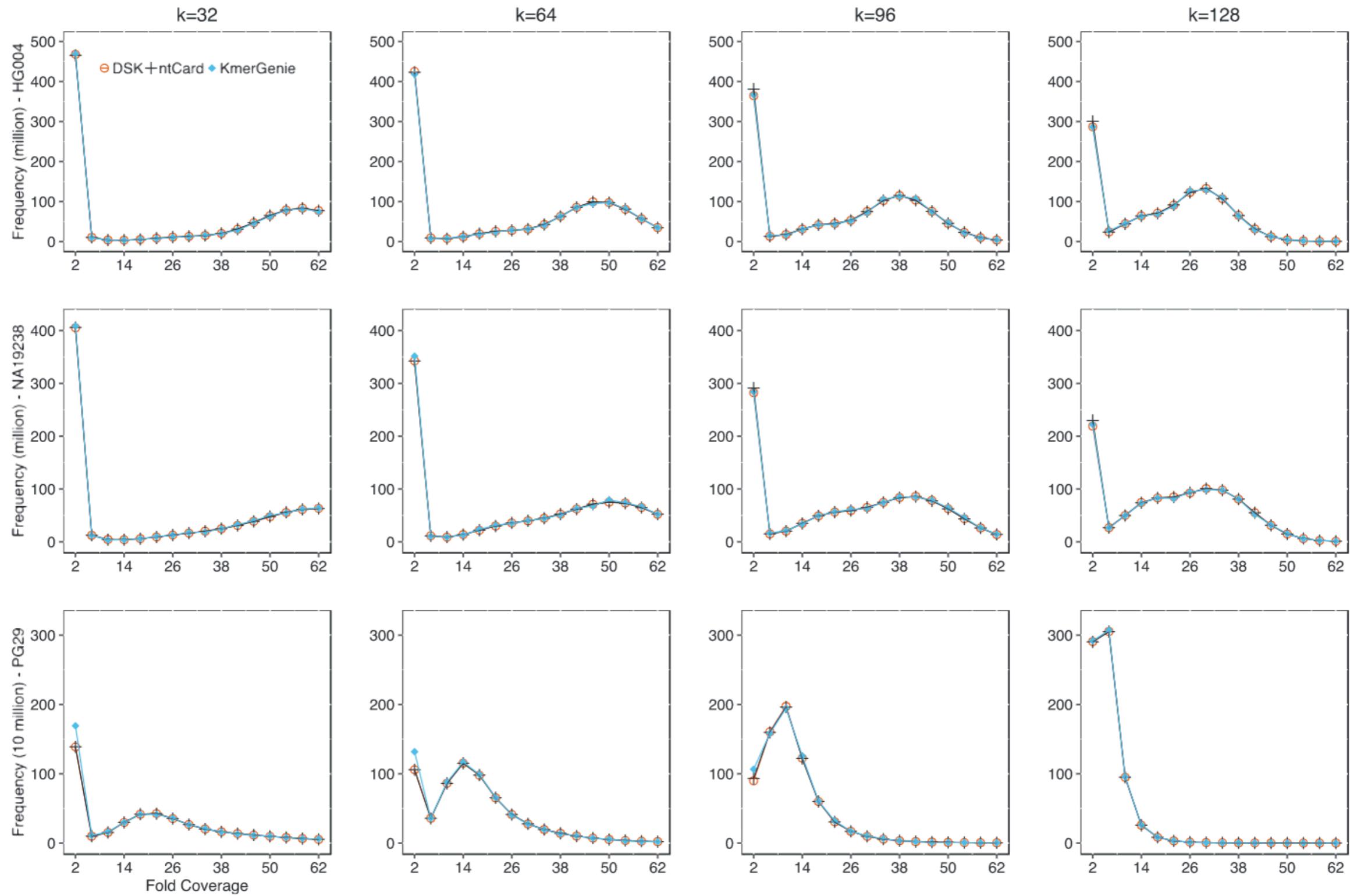
**Table 3.** Accuracy of algorithms in estimating  $F_0$  and  $f_1$  for NA19238 reads

$k$		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

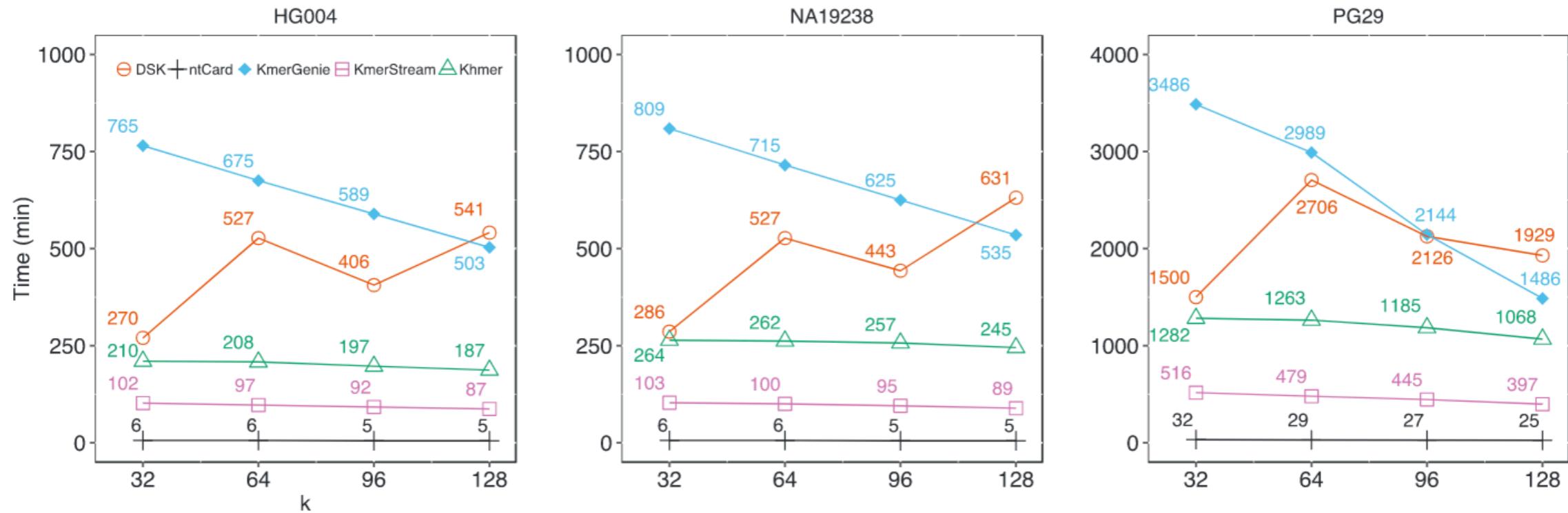
$k$		DSK	ntCard	KmerGenie	KmerStream	Khmer
32	$f_1$	27,430,910,938	0.02%	15.33%	9.41%	–
	$F_0$	42,642,198,777	0.01%	11.02%	7.37%	8.86%
64	$f_1$	44,344,130,469	0.04%	16.36%	2.61%	–
	$F_0$	67,800,291,613	0.02%	11.14%	1.73%	11.18%
96	$f_1$	43,300,244,443	0.66%	17.51%	0.73%	–
	$F_0$	69,855,690,006	0.46%	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  $\sim 500\text{MB}$

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

## Interlude: Succinct Data Structures & Rank and Select

Slides for the following taken from:

<https://www.cs.helsinki.fi/u/puglisi/dct2015/slides10.pdf>

**credit to Simon J. Puglisi, University of Helsinki**

# Succinct Data Structures

- Succinct data structure
  - = succinct representation of data + a succinct index
- (usually static)
- High-level goal: reduce space so the data structure might fit in RAM and therefore be faster to use
- Examples
  - Sets
  - Trees, graphs
  - Strings
  - Permutations, functions

# Succinct Representation

- A representation of data whose size (roughly) matches the information-theoretic lower bound
- If the input is taken from  $L$  distinct possible inputs, then its information-theoretic lower bound is  $\text{ceil}(\log L)$  bits
  - To be considered succinct a data structure must use:  
 $\text{ceil}(\log L) + o(\log L)$  bits
- Example: a lower bound for a set  $S$ , subset of  $\{1,2,\dots,n\}$ 
  - $\log(2^n) = n$  bits
  - $n = 3$  we have 8 distinct sets... so d.s. will need at least 3 bits
    - $\emptyset$
    - $\{1\}$
    - $\{2\}$
    - $\{3\}$
    - $\{1,2\}$
    - $\{1,3\}$
    - $\{2,3\}$
    - $\{1,2,3\}$

# Succinct Index

- Auxiliary data structure to support queries on the succinct representation
- Size:  $\text{O}(\log L)$  bits
- The index should allow queries/operations on the succinct representation in (almost) the same time complexity as using a conventional data structure
  - This is the aim anyway
- Computational model is the word RAM
  - Assume word length  $w = \log \log L$
  - (this is the same pointer size as conventional data structures)
  - read/write  $w$  bits of memory in  $O(1)$  time
  - arithmetic/logical operations on  $w$  bit numbers take  $O(1)$  time
  - $+, -, *, /, \log, \&, |, !, >>, <<$

# Binary rank and select

- The ability to answer *rank* and *select* queries over bit vectors (binary strings, bit arrays) is essential for implementing succinct data structures
- Given a binary string  $B[1..n]$ 
  - $\text{rank}_B(i)$  returns the number of 1 bits in  $B[1..i]$
  - $\text{select}_B(i)$  returns the position of the  $i^{\text{th}}$  1 bit in  $B$

# Naïve rank

- To answer  $\text{rank}(i)$  scan  $B[1..i]$  and count 1-bits
- Simple but slow
  - $O(i)$  time =  $O(n)$  time in the worst case
- How can we do better?
  - After all, what are we?

# (Slightly) Less naïve rank

- Store an table  $A[1..n]$ , containing the rank answers

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0
A	1	1	1	2	2	3	4	5	5	6	6	6	7	7	8	8

- $A[i] = \text{rank}(i)$ 
  - Now  $\text{rank}(i)$  takes constant time - just an array lookup!
- Drawback:
  - A requires  $n \log n$  bits -  $\log n$  times the size of B - not succinct!
  - We'd like a solution with  $O(1)$  queries and  $o(n)$  extra space...

# We want $O(1)$ queries with $o(n)$ extra bits...

- General approach will be to precompute some tables
- Each table stores part of the answer to every query
  - For any given query, we can extract needed parts in  $O(1)$  time
  - The total size of the tables is  $o(n)$  bits

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0

- Premise:
  - Can read  $O(\log n)$  bits into an integer in range  $1..n$  in  $O(1)$  time
  - However, to inspect each of those bits take  $O(\log n)$  time

# Tables : Superblocks

- Divide B into superblocks of size  $s = \log^2 n / 2 = 4 * 4 / 2 = 8$
- Build a small table  $R_s$  containing ranks for only some positions

B	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0
	0								1							
$R_s$	0								5							

- Store in  $R_s[j] = \text{rank}_B(j*s)$ , for all  $0 \leq j < n/s$

# Tables : Blocks

- Divide each superblock into blocks of size  $b = \log n / 2 = 2$
- Build a table  $R_b$  which contains the rank from the start of each block to the start of its superblock

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0

	0		1
$R_s$	0		5

	0	1	2	3	0	1	2	3
$R_b$	0	1	2	3	0	1	1	2

- Store  $R_b[k/b] = \text{rank}_B(k*s) - \text{rank}_B(j*s)$ , for all  $0 \leq k < n/b$

# Intermission

- What we have so far (tables  $R_s$  and  $R_b$ ) almost gets us the answer we're after

B	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0
$R_s$	0															1
$R_b$	0															5
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
	0	1	2	3	0	1	1	2	0	1	1	2	0	1	1	2

- $\text{rank}_B(i) \approx R_s[i/s] + R_b[i/b]$ 
  - Just need to answer in-block queries in  $O(1)$  time

# Tables : Resolving in-block queries

- Solution? Use another table!
- Blocks have size  $b = \log_2 n / 2$ 
  - There are  $2^b$  such blocks possible
  - In each block there are  $b$  possible rank queries
  - Each answer (relative to the block) is in the range 1.. $b$

	Type			rank(0)	rank(1)
		0	1		
$R_p$	0	0	0	0	0
	1	0	1	0	1
	2	1	0	1	1
	3	1	1	1	2

# Final Data Structure

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	1	0	0	1	0	1	1	1	0	1	0	0	1	0	1	0

	0	1
R <sub>s</sub>	0	5

	0	1	2	3	0	1	2	3
R <sub>b</sub>	0	1	2	3	0	1	1	2

R <sub>p</sub>	Type	0    1		rank(0)    rank(1)	
		0	1	0	0
	0	0	0	0	0
	1	0	1	0	1
	2	1	0	1	1
	3	1	1	1	2

# Size of table for within-block queries

- Blocks have size  $b = \log_2 n / 2$ 
  - There are  $2^b$  such blocks possible
  - In each block there are  $b$  possible rank queries
  - Each answer (relative to the block) is in the range 1.. $b$

	Type	0	1	rank(0)	rank(1)
		0	0	0	0
$R_p$	1	0	1	0	1
	2	1	0	1	1
	3	1	1	1	2

- Therefore size of  $R_p$ , the in-block data structure is
  - $2^b * b * \log b = n^{1/2} * \log n * \log \log n / 2$  bits =  $o(n)$  bits

# Summing up sizes...

- The size of  $R_s$ , the superblock data structure is
  - $2n/\log^2 n$  superblocks, each of size  $\log n$  bits
  - $(n/\log^2 n) * \log n = 2n/\log n$  bits =  $o(n)$  bits
- The size of  $R_b$ , the block data structure is
  - $2n/\log n$  blocks, each of size  $\log \log n$  bits
  - $2n \log \log n / \log n$  bits =  $o(n)$  bits
- $R_s + R_b + R_p = o(n)$  extra bits for  $O(1)$  time rank queries
  - It is possible to construct this data structure in  $O(n)$  time

# Variations

- Just store  $R_s$  + use manual counting within superblocks
  - Saves space for  $R_b$  and  $R_p$ , takes time  $O(\log^2 n)$  per query
- Store  $R_s$  and  $R_b$  + use manual counting within blocks
  - Saves only space for  $R_p$ , takes time  $O(\log n)$  per query
- Use different superblock & block sizes
  - No more theoretical guarantees, but...
  - Perhaps faster in practice: blocks that are multiples of word sizes (32-bits) can be faster to handle

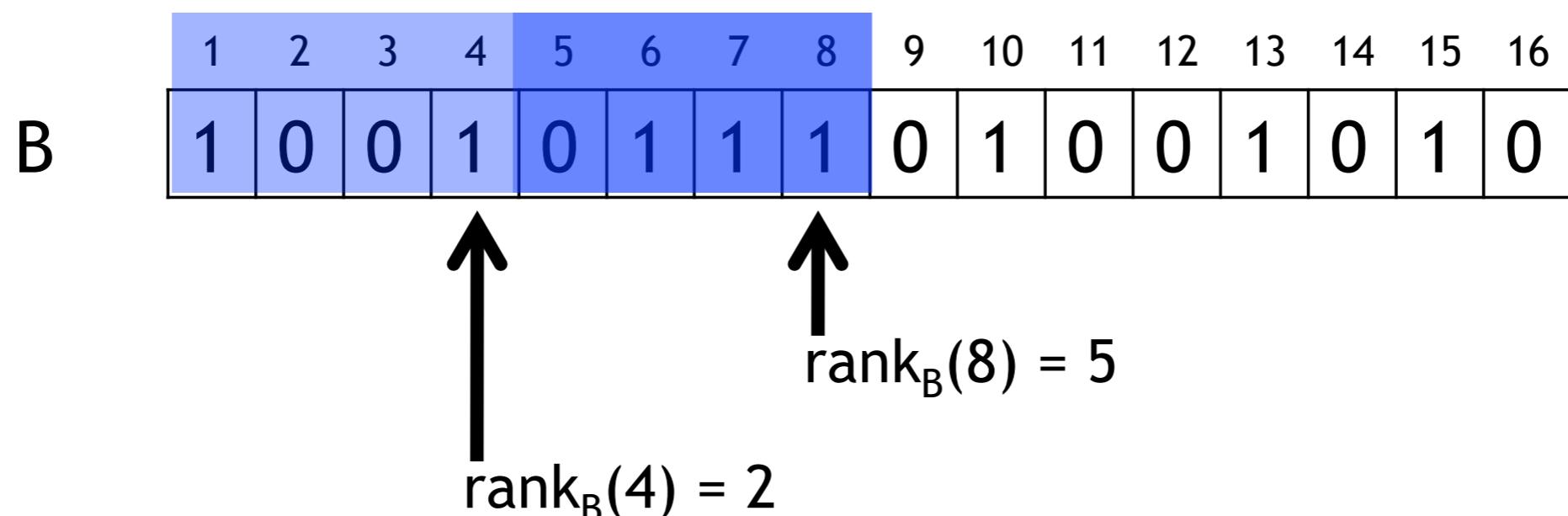
# Summary of rank

- Rank index takes  $O(n \log \log n / \log n) = o(n)$  bits so we use  $n + o(n)$  overall and can answer queries in  $O(1)$  time
- While it is sublinear, we'd still like the  $o(n)$  term to be small
  - Best is by Patrascu:  $O(n / \log^k n)$  bits,  $O(k)$  time queries
- Dynamic solutions exist
  - Queries no longer constant:  $O(\log n / \log \log n)$  time (Raman et al.)

# Relationship to select( $i$ )

- We can use our solution to rank to get a (fairly) efficient solution to  $\text{select}(i)$ , with this observation:
- If  $\text{rank}(n/2) > i$ , then the  $i^{\text{th}}$  1-bit is in  $B[1..n/2]$ 
  - Otherwise it is in  $B[n/2+1..n]$

$\text{select}_B(3)$



# Relationship to select(i)

- Applying this idea recursively to arrive at select(i)
  - $O(\log_2 n)$  time,  $o(n)$  space
- $O(1)$  time,  $o(n)$  space solutions for select also exist
  - Slightly more complicated than  $O(1)$  rank
  - (Munro and Clark)
- Similar variations as we discussed with rank (trading space for query time) are also possible

Information Systems 73 (2018) 25–34



Contents lists available at [ScienceDirect](#)

Information Systems

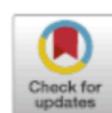
journal homepage: [www.elsevier.com/locate/is](http://www.elsevier.com/locate/is)



Rank and select: Another lesson learned

Szymon Grabowski\*, Marcin Raniszewski

Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, Łódź 90-924, Poland



# Work along these lines at SBU

## 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](mailto:{ppandey,bender,rob,rob.patro}@cs.stonybrook.edu)

Bioinformatics  
33/bioinformatics/xxxxxx  
n Date: Day Month Year  
Manuscript Category



---

Genome analysis

## Squeakr: An Exact and Approximate $k$ -mer Counting System

Prashant Pandey<sup>1,\*</sup>, Michael A. Bender<sup>1</sup>, Rob Johnson<sup>1,2</sup>, and Rob Patro<sup>1</sup>

<sup>1</sup>Department of Computer Science, Stony Brook University, Stony Brook, NY 11790, USA

<sup>2</sup>VMware Research, 3425 Hillview Ave, Palo Alto, CA 94304, USA

33, 2017, i133–i141  
nformatics/btx261  
ISMB/ECCB 2017



---

## deBGR: an efficient and near-exact representation of the weighted de Bruijn graph

Prashant Pandey<sup>1</sup>, Michael A. Bender<sup>1</sup>, Rob Johnson<sup>1,2</sup> and Rob Patro<sup>1,\*</sup>

<sup>1</sup>Department of Computer Science, Stony Brook University, Stony Brook, NY 11790, USA, <sup>2</sup>VMWare, Inc., Palo Alto, CA 94304

## Mantis: A Fast, Small, and Exact Large-Scale Sequence-Search Index

Prashant Pandey<sup>1</sup>, Fatemeh Almodaresi<sup>1</sup>, Michael A. Bender<sup>1</sup>, Michael Ferdman<sup>1</sup>, Rob Johnson<sup>2,1</sup>, and Rob Patro<sup>1</sup>

<sup>1</sup> Computer Science Dept., Stony Brook University

[{ppandey,falmodaresit,bender,mferdman,rob.patro}@cs.stonybrook.edu](mailto:{ppandey,falmodaresit,bender,mferdman,rob.patro}@cs.stonybrook.edu)

<sup>2</sup> VMware Research

[robj@vmware.com](mailto:robj@vmware.com)

# The Counting Quotient Filter

Compact, lossless representation of multiset  $h(S)$

$h : U \rightarrow \{0, \dots, 2^p - 1\}$  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

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)?

	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		$h_1(a)$	$h_1(b)$	$h_1(c)$	$h_1(d)$	$h_1(e)$		$h_1(f)$
	$\longleftrightarrow 2^q \longrightarrow$							

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 encounter).

# 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

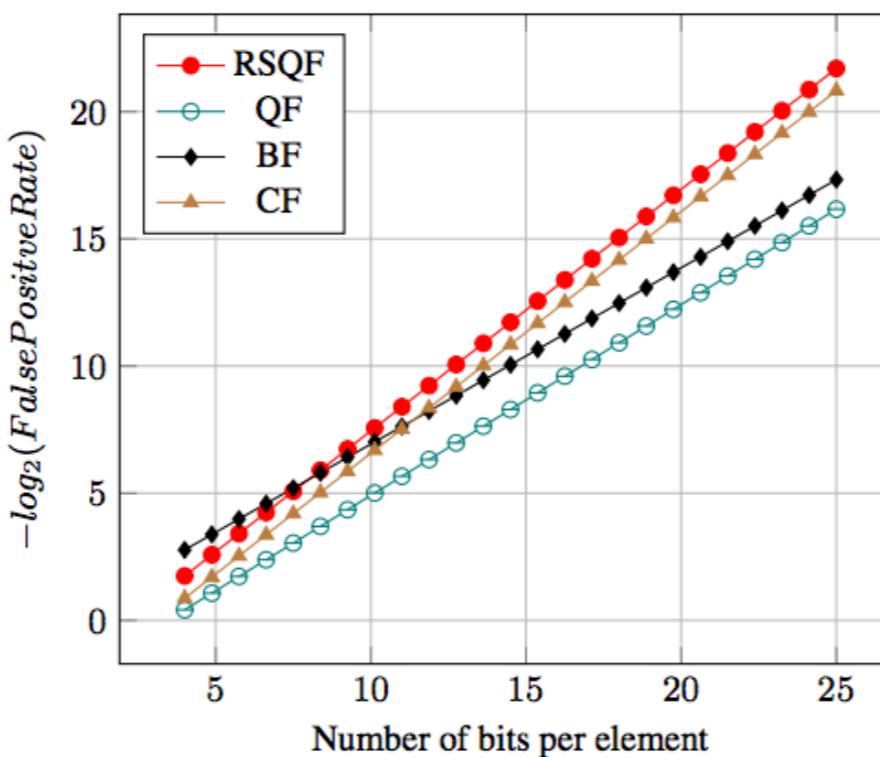


Figure 4: Number of bits per element for the RSQF, QF, BF, and CF. The RSQF requires less space than the CF and less space than the BF for any false-positive rate less than 1/64. (Higher is better)

# 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).

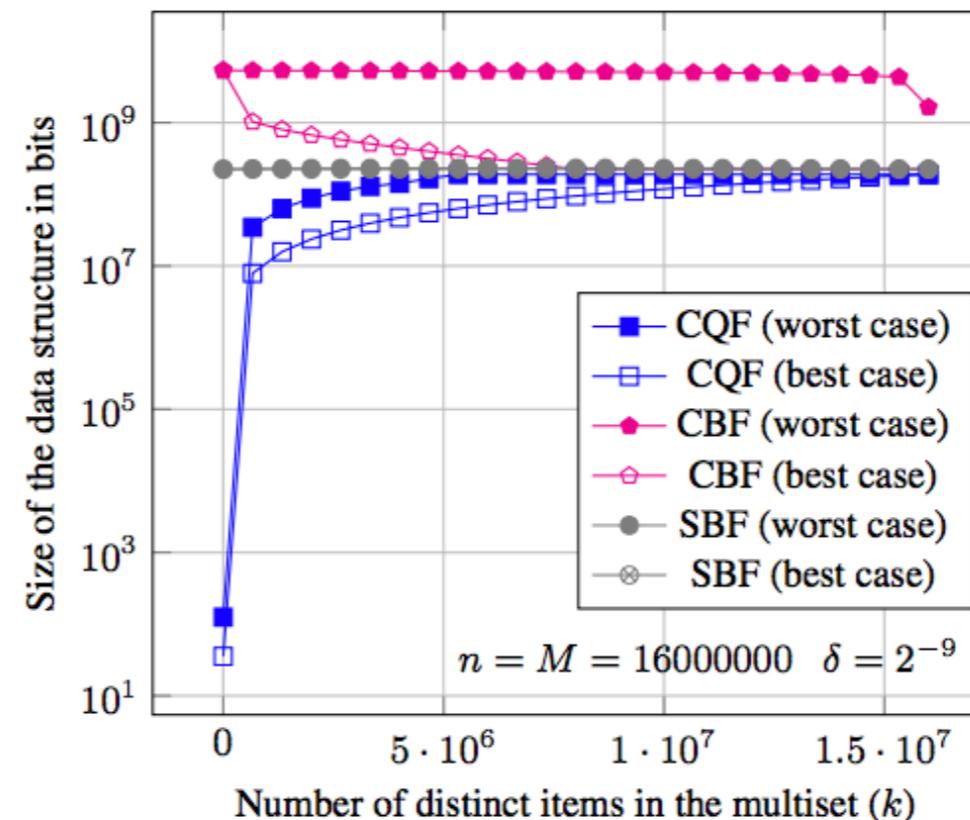
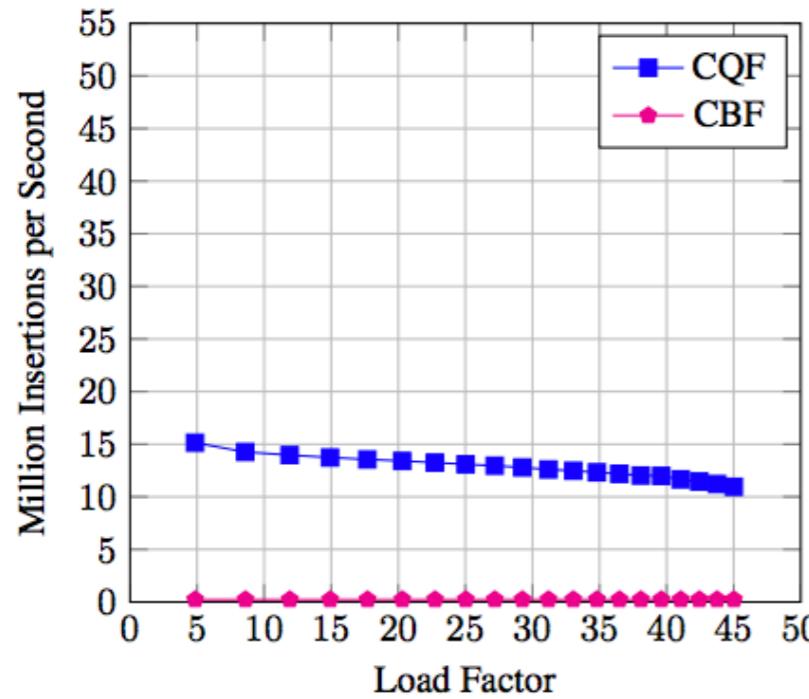
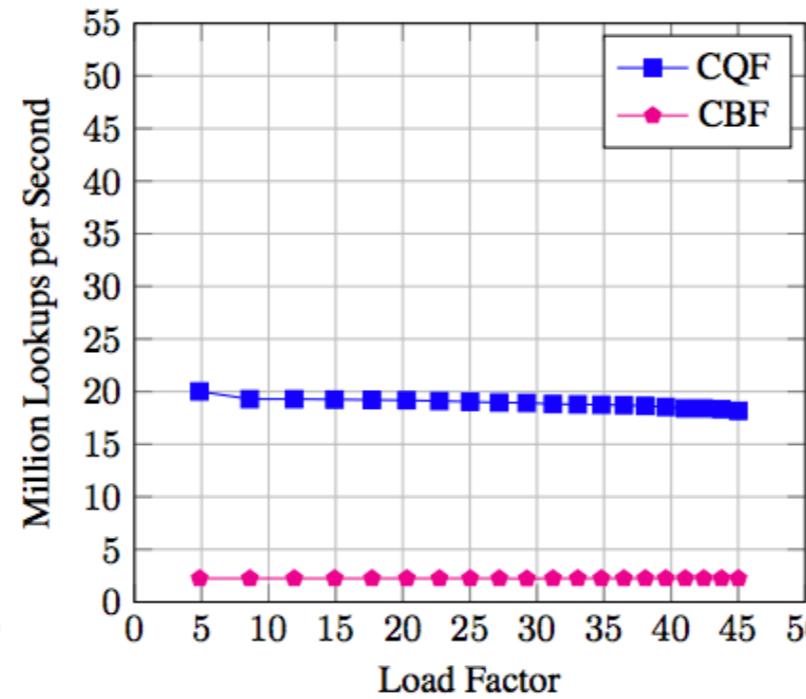


Figure 5: Space comparison of CQF, SBF, and CBF as a function of the number of distinct items. All data structures are built to support up to  $n = 1.6 \times 10^7$  insertions with a false-positive rate of  $\delta = 2^{-9}$ .

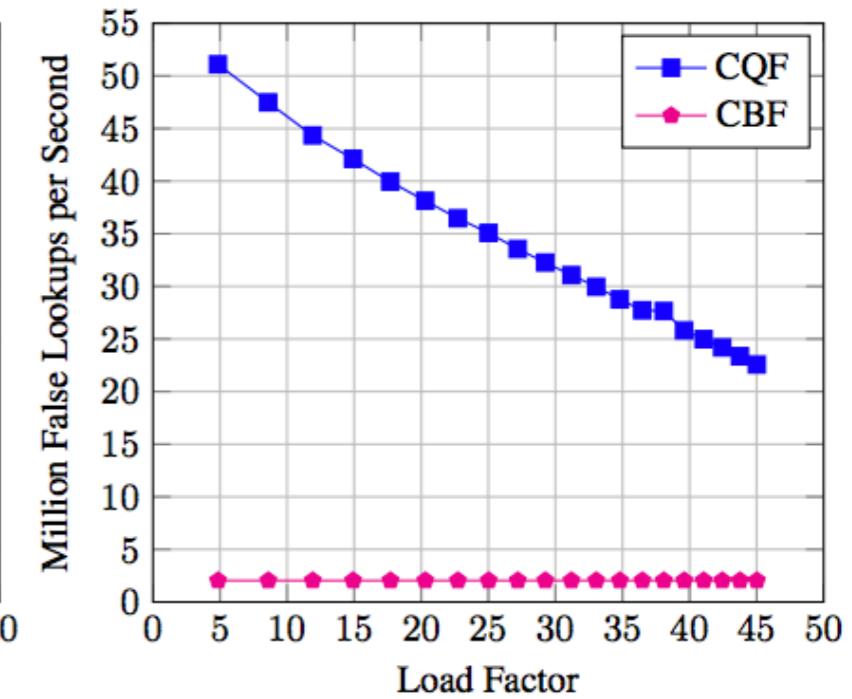
# The Counting Quotient Filter, results



(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

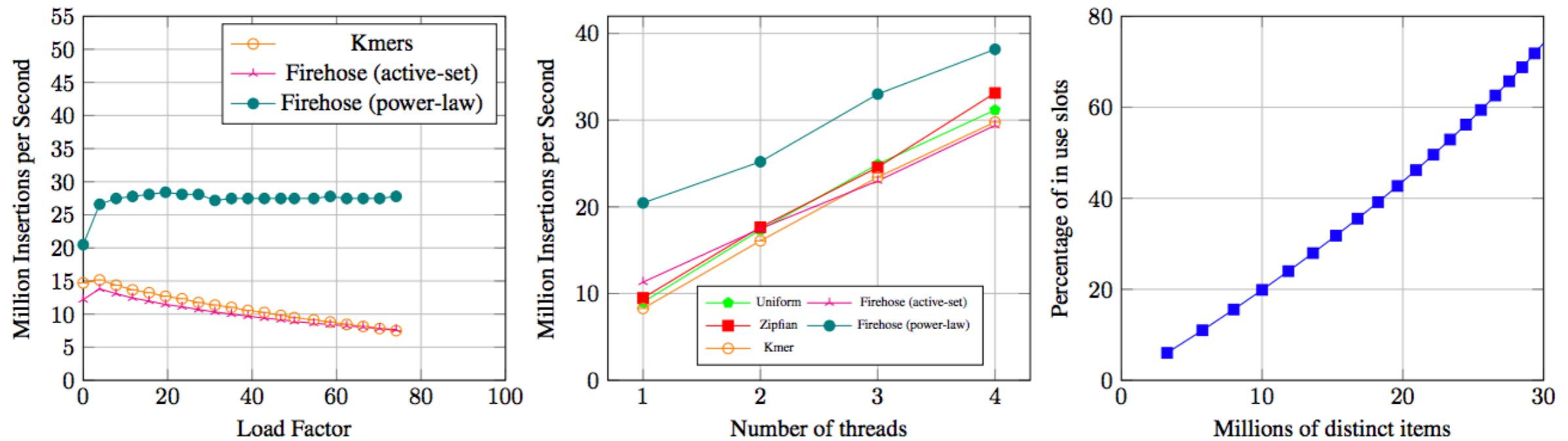


Figure 9: In-memory performance of the counting quotient filter with real-world data sets and with multiple threads, and percent slot usage with skewed distribution.

# 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
<i>E.vesca</i>	3.3	11	4 134 078 256	632 436 468
<i>G.gallus</i>	25.0	15	25 337 974 831	2 727 529 829
<i>M.balbisiana</i>	46.0	2	41 063 145 194	965 691 662
<i>H.sapiens</i> 1	67.0	6	62 837 392 588	6 353 512 803
<i>H.sapiens</i> 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
<i>E.vesca</i>	8.3	4.8	9.3	8.3
<i>G.gallus</i>	32.8	13.0	28.8	31.7
<i>M.balbisiana</i>	48.3	11.1	14.2	16.3
<i>H.sapiens</i> 1	71.4	22.1	51.5	61.8
<i>H.sapiens</i> 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	<i>F.vesca</i>		<i>G.gallus</i>		<i>M.balbisiana</i>		<i>H.sapiens</i> 1		<i>H.sapiens</i> 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	<i>F.vesca</i>		<i>G.gallus</i>		<i>M.balbisiana</i>		<i>H.sapiens</i> 1		<i>H.sapiens</i> 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	<i>G. gallus</i>		<i>M. balbisiana</i>	
	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	<i>G. gallus</i>	122	266	23 097	23 363
Squeakr	<i>G. gallus</i>	92	412	3415	3827
KMC2	<i>M. balbisiana</i>	123	607	6817	7424
Squeakr	<i>M. balbisiana</i>	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!