CSE 549: Efficiently Dealing with k-mers and De Bruijn Graphs

Stony Brook University

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

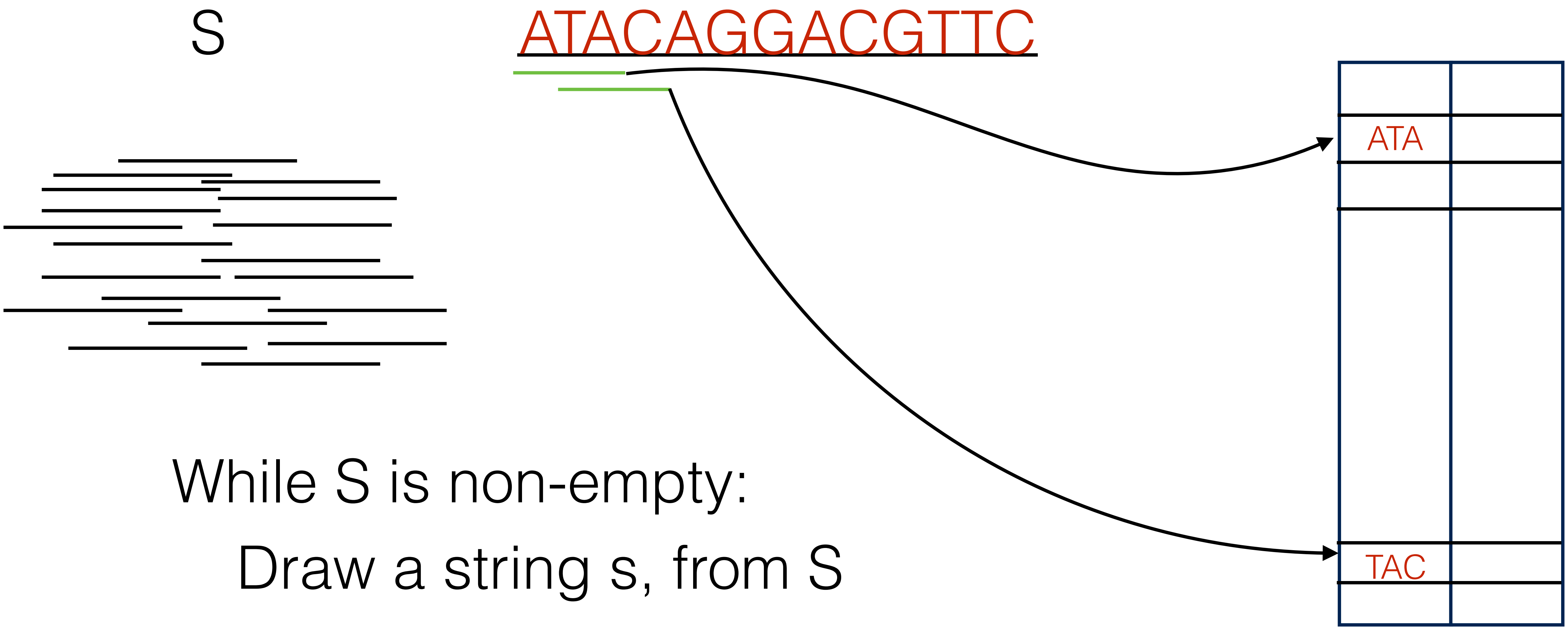
k-mer counting

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

Tallymer, Jellyfish, DSK, KMC, 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:
        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

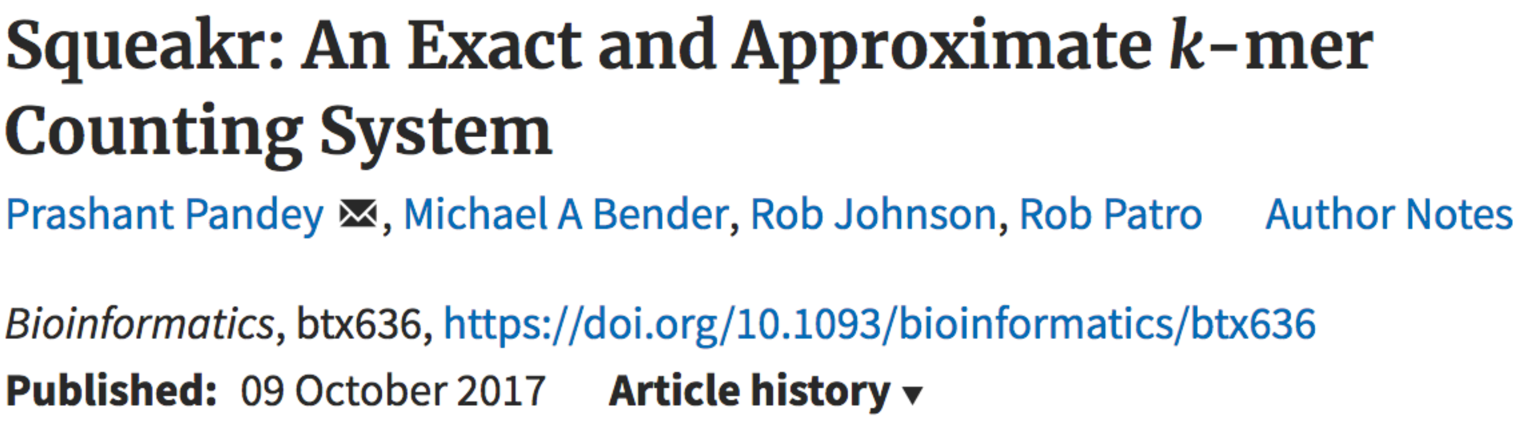
— f: $U_k \rightarrow U_k$, and let hash(k) = f(k) mod M

— 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)



Squeakr: An Exact and Approximate *k*-mer Counting System

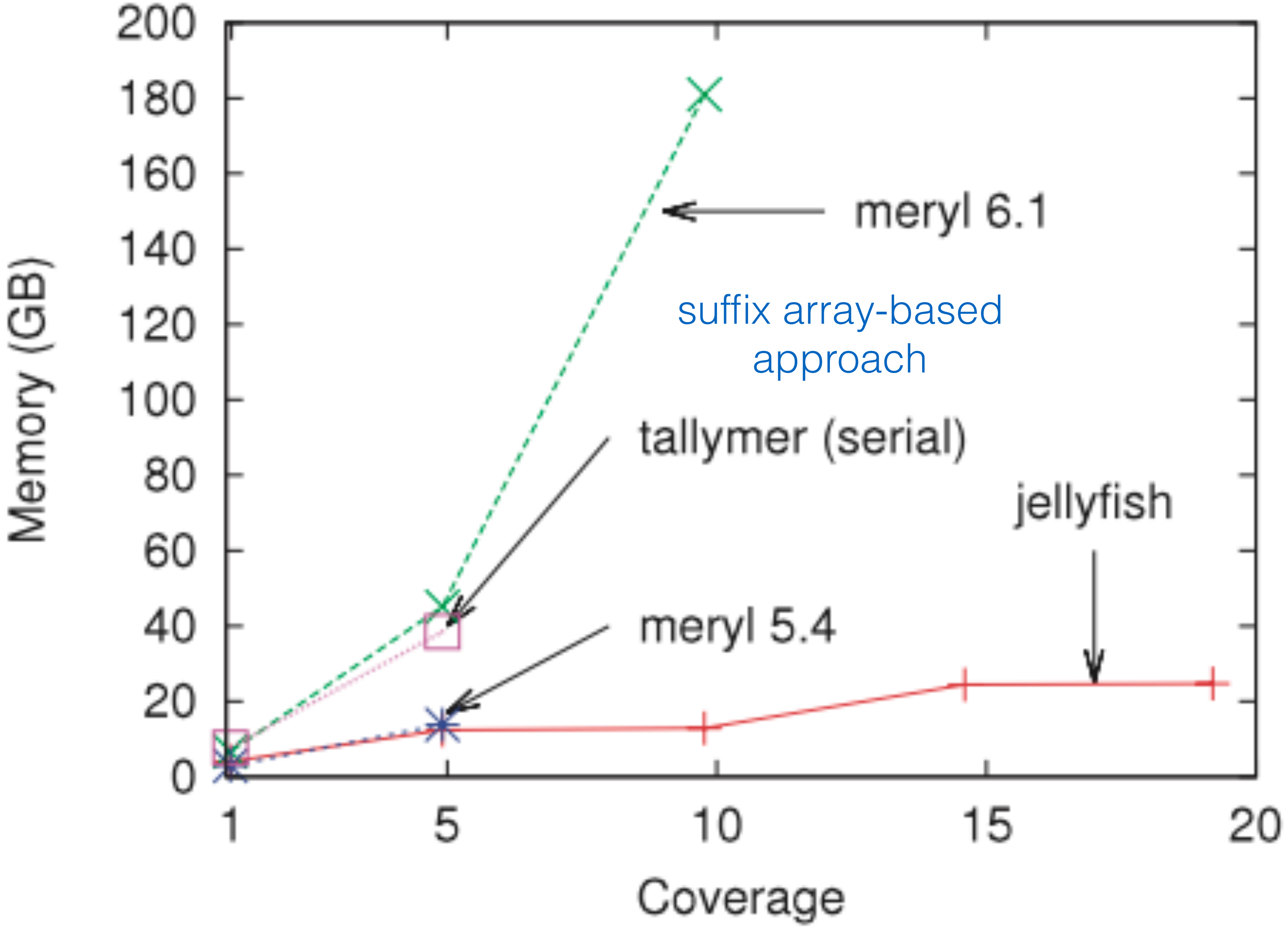
Prashant Pandey ✉, Michael A Bender, Rob Johnson, Rob Patro Author Notes

Bioinformatics, btx636, https://doi.org/10.1093/bioinformatics/btx636

Published: 09 October 2017 **Article history** ▾

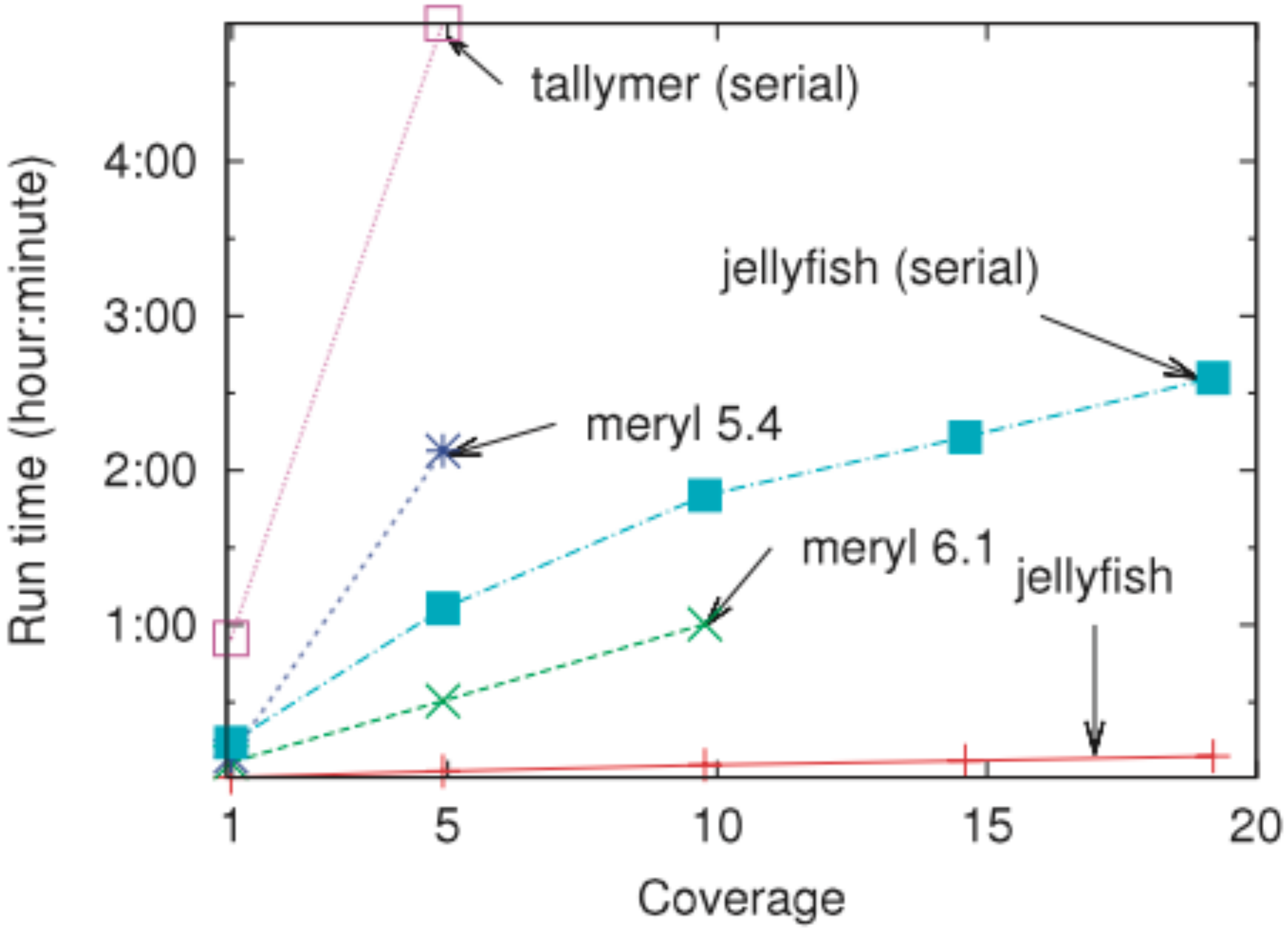
"Don't thrash: how to cache your hash on flash" (PDF). *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems (HotStorage'11)*. Retrieved 21 July 2012.

Memory usage of Jellyfish

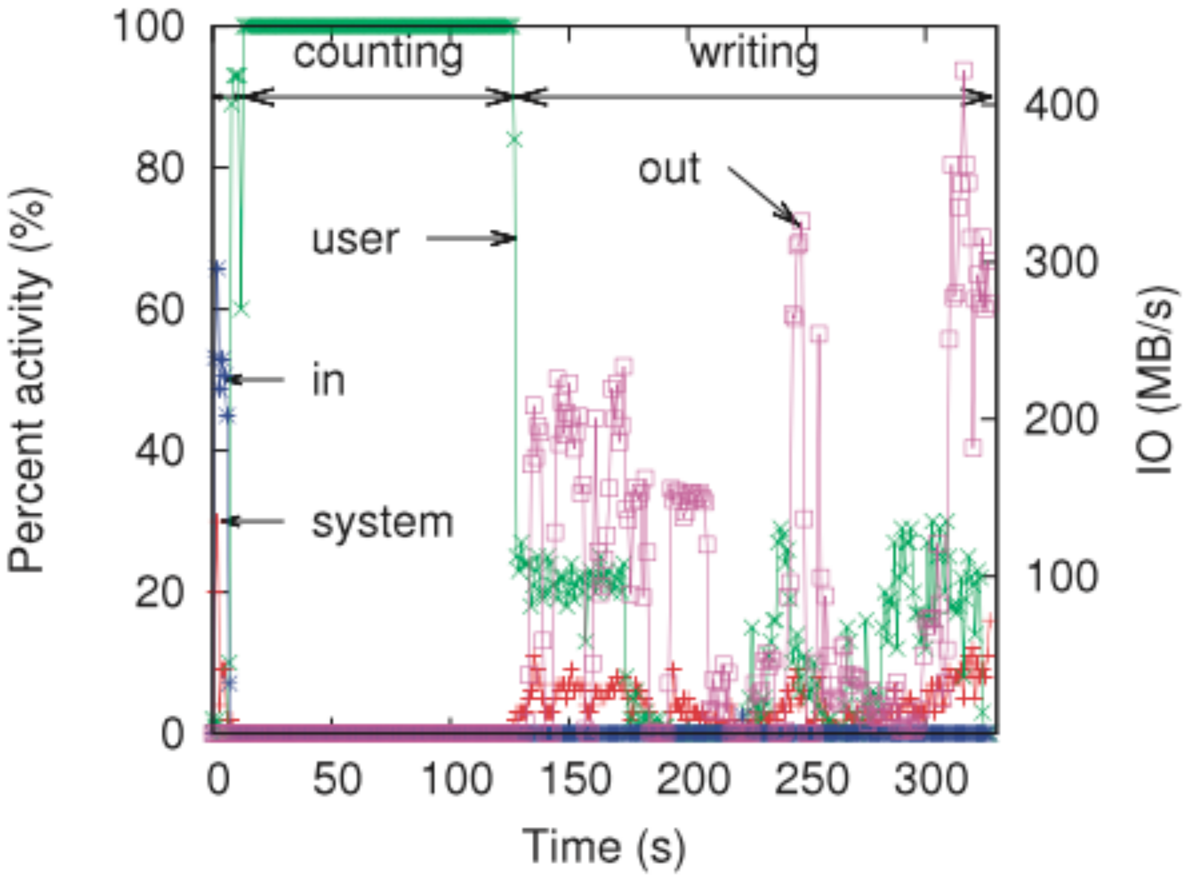


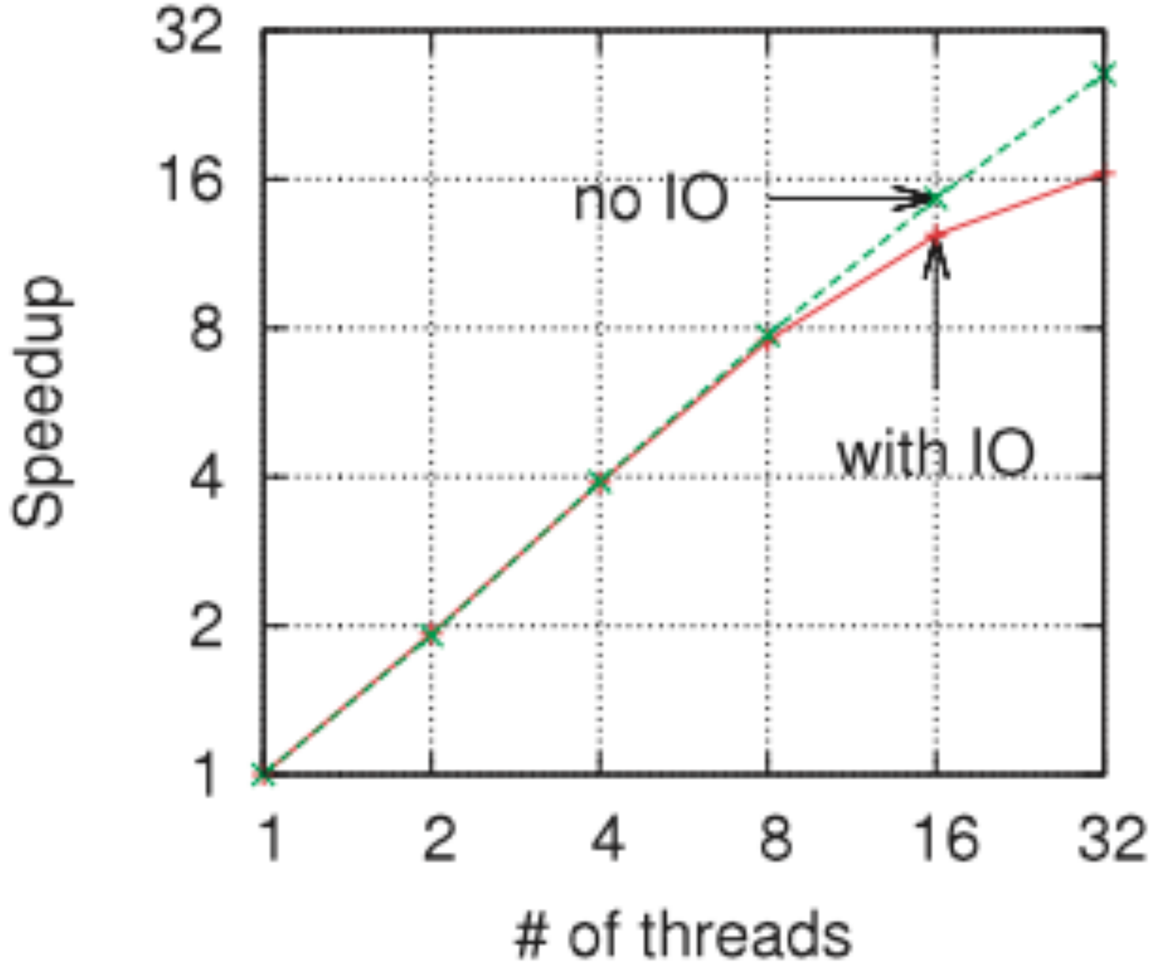
suffix array-based approach

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.

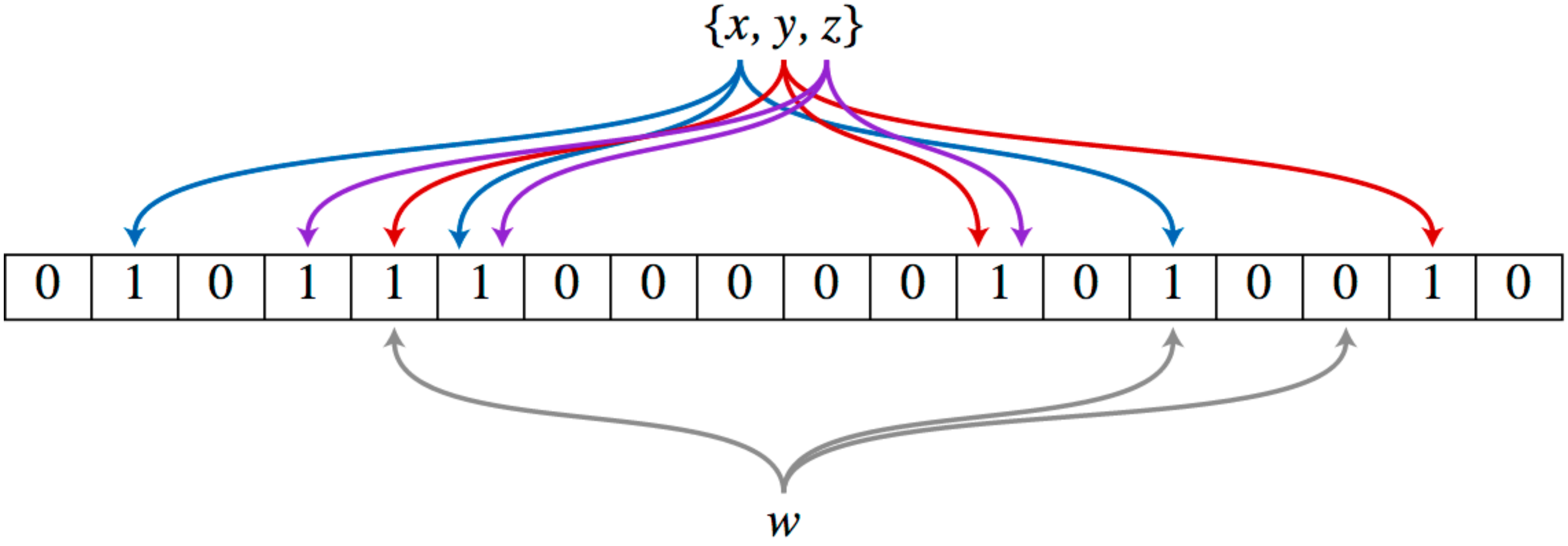
Bloom Filters

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 $0 < i < k$



0	1	0	1	1	1	0	0	0	0	0	1	0	1	0	0	1	0

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?

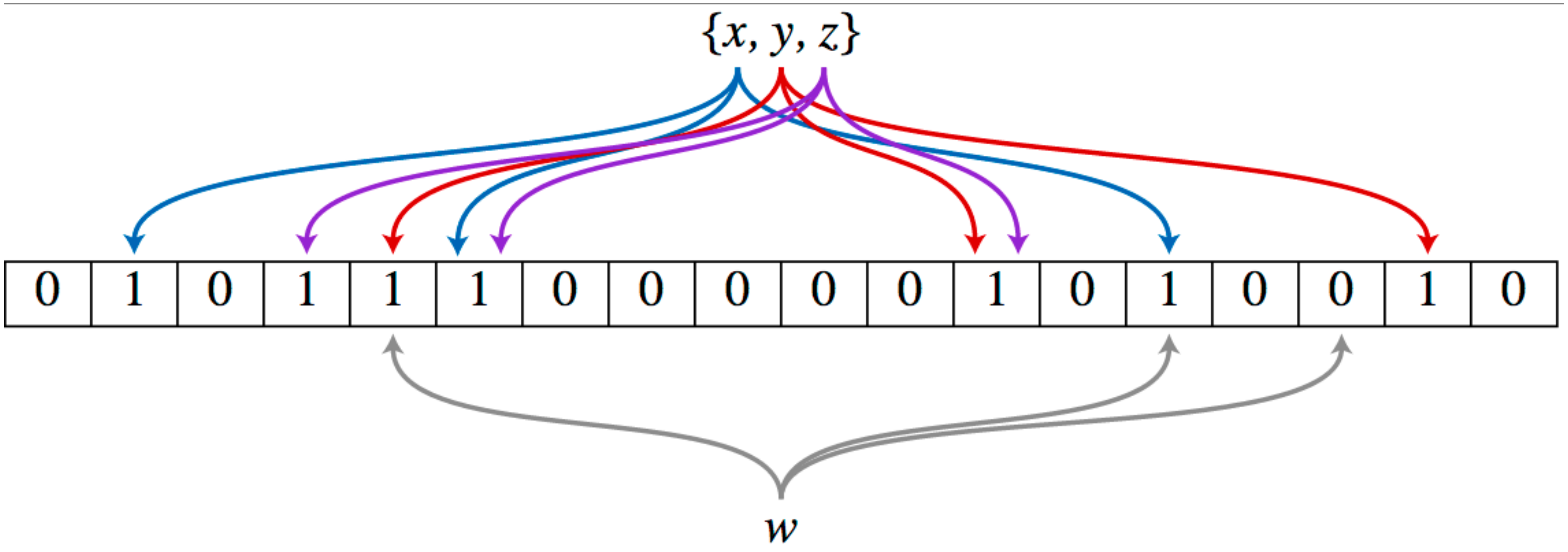


Image by David Eppstein - self-made, originally for a talk at WADS 2007

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

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

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

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

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

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

given an expected # elems

and a desired false positive rate

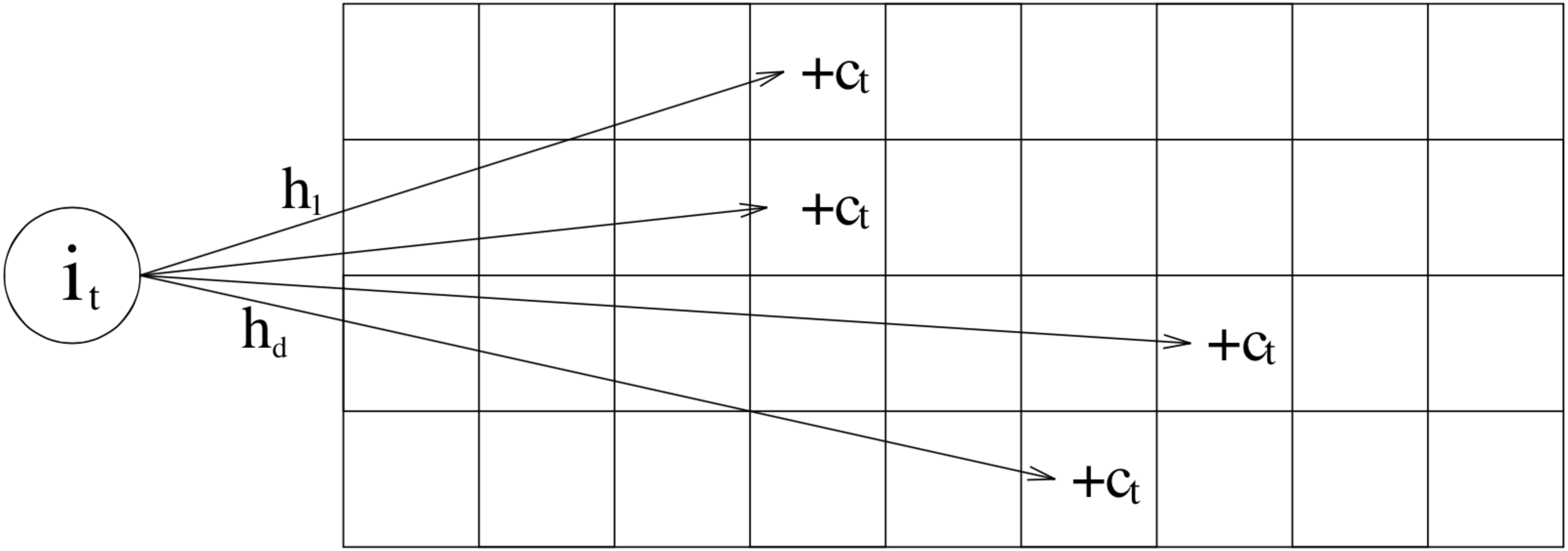
we can compute the optimal size and # of has functions

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 x w — d is called the depth of the array, and there are d *independent* hash functions, w is called with width of the array. This is an O(wd) data structure.



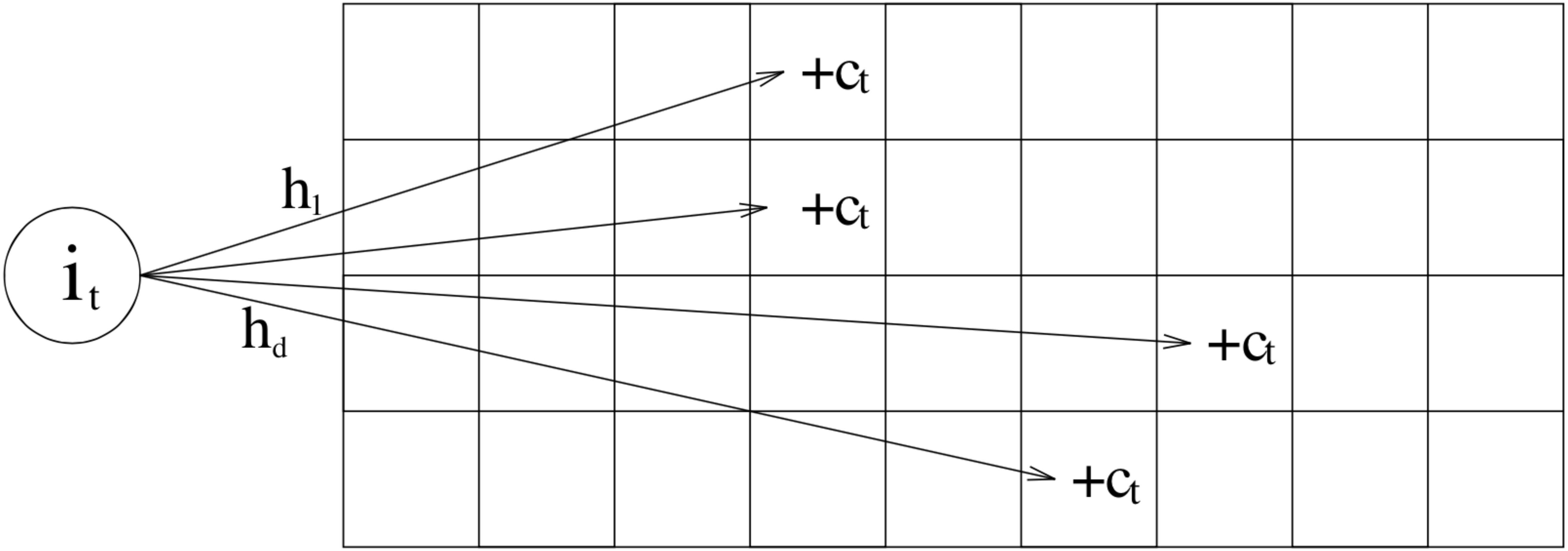
Probabilistic Data Structures & k-mer Counting

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 âᵢ be the result returned by Query(i). We have that:

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

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

where,

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

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 Query(i). We have that:

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

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

where,

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

base of nat. log

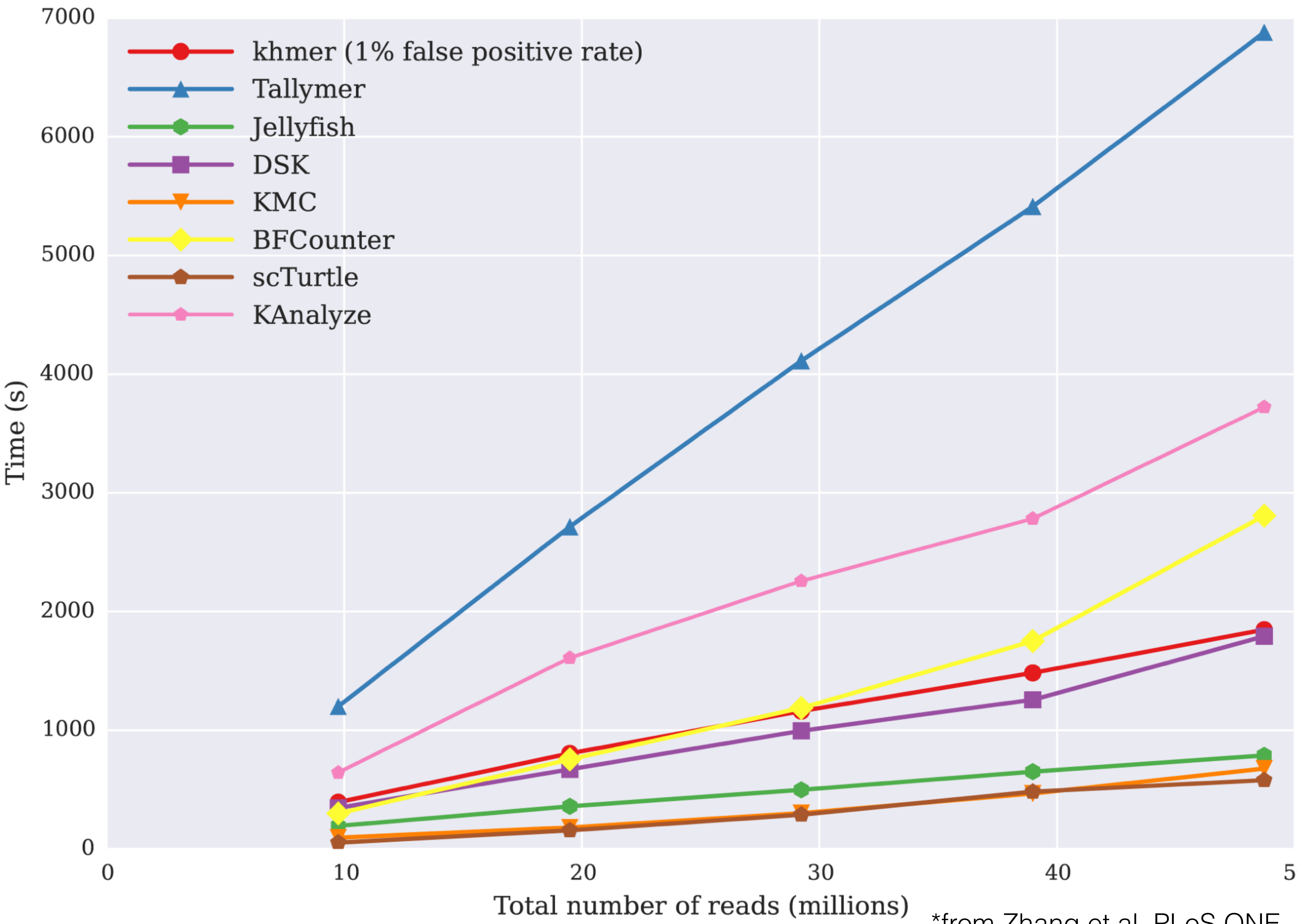
The Count-Min Sketch for k-mer counting

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

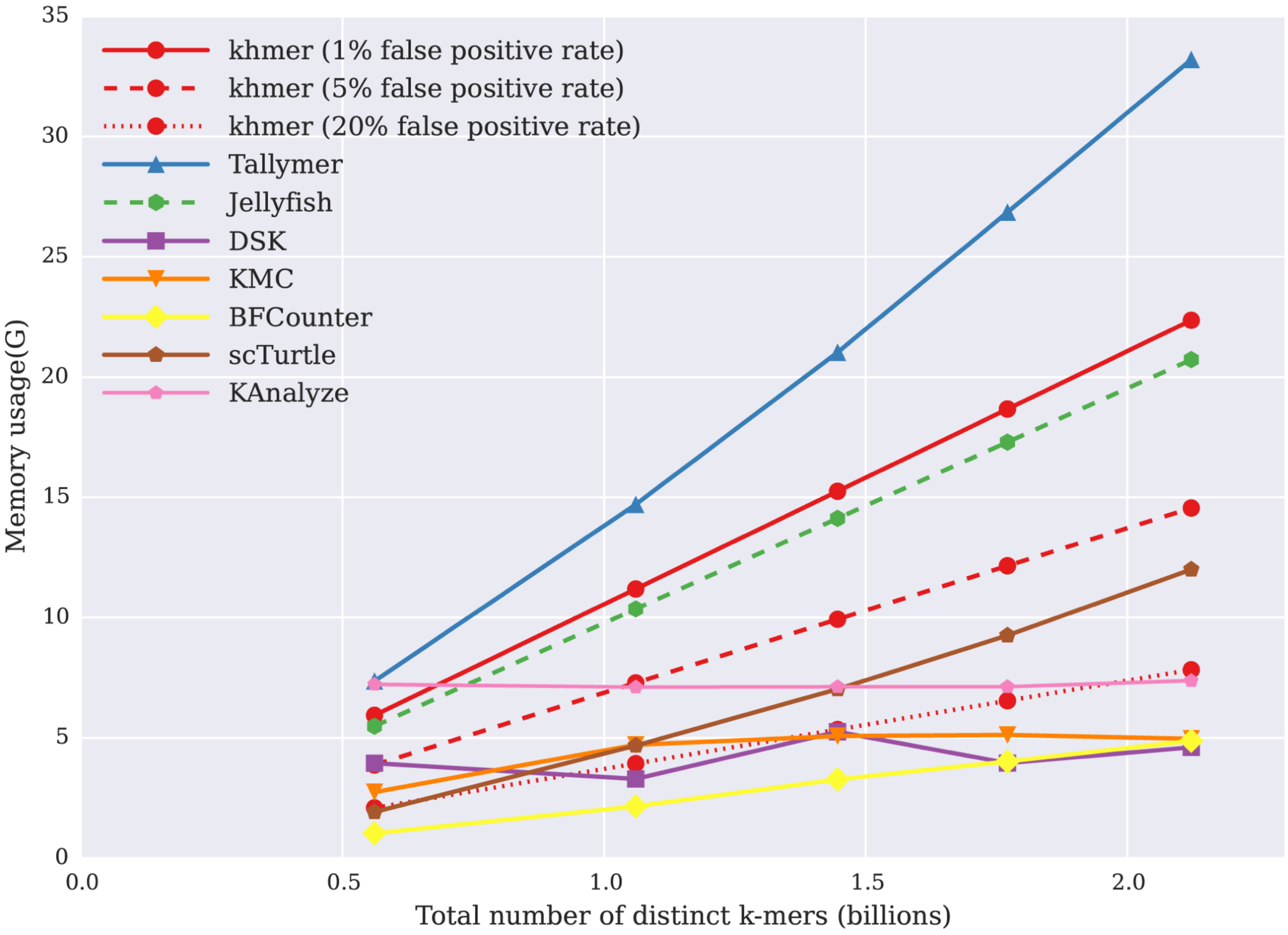
No exact data structure is maintained, just a CMS

This allows for answering approximate count queries efficiently.

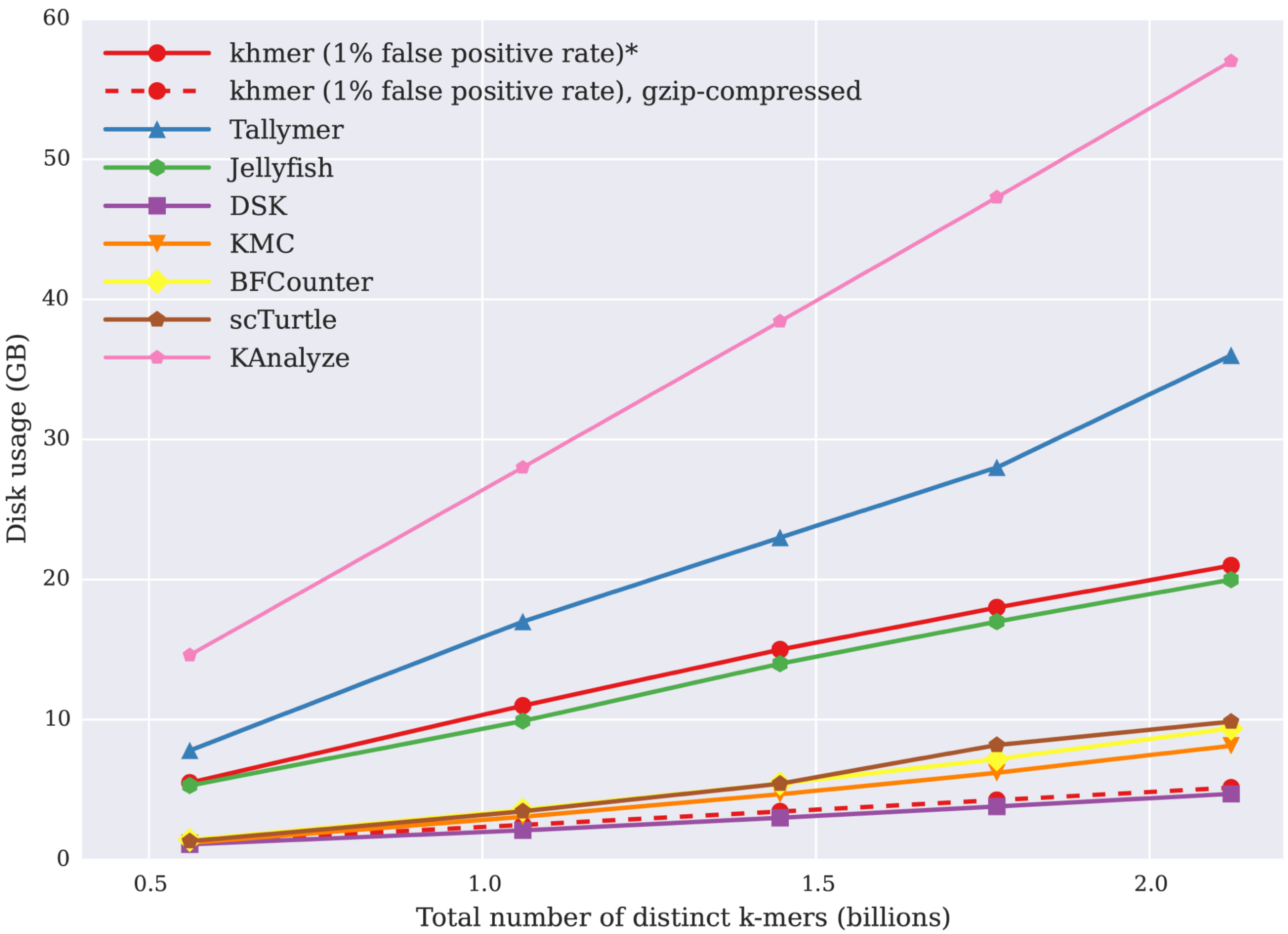
Authors compared to a large number of other k-mer counters under several different metrics.



*from Zhang et al. PLoS ONE, 2014

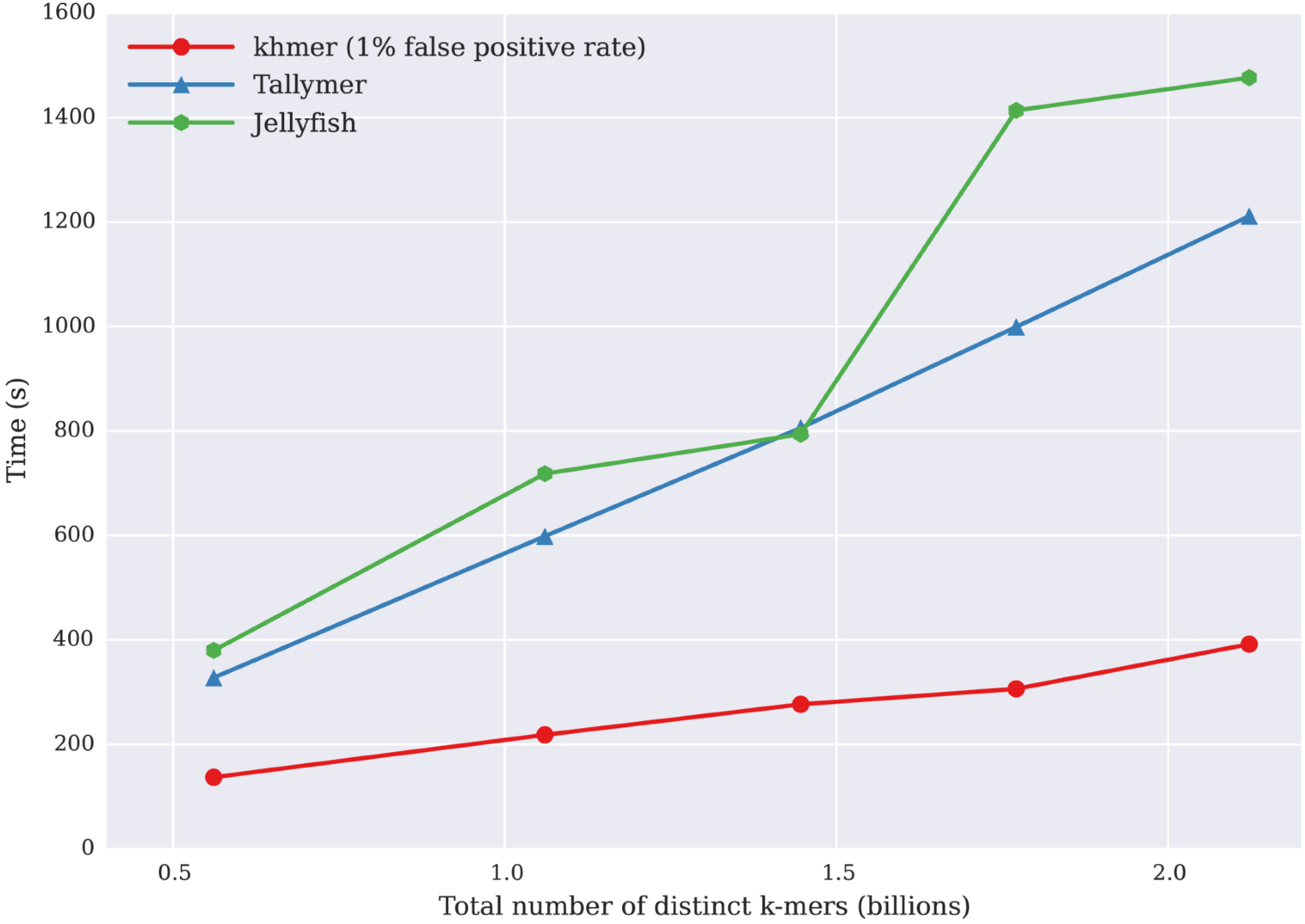


*from Zhang et al. PLoS ONE, 2014



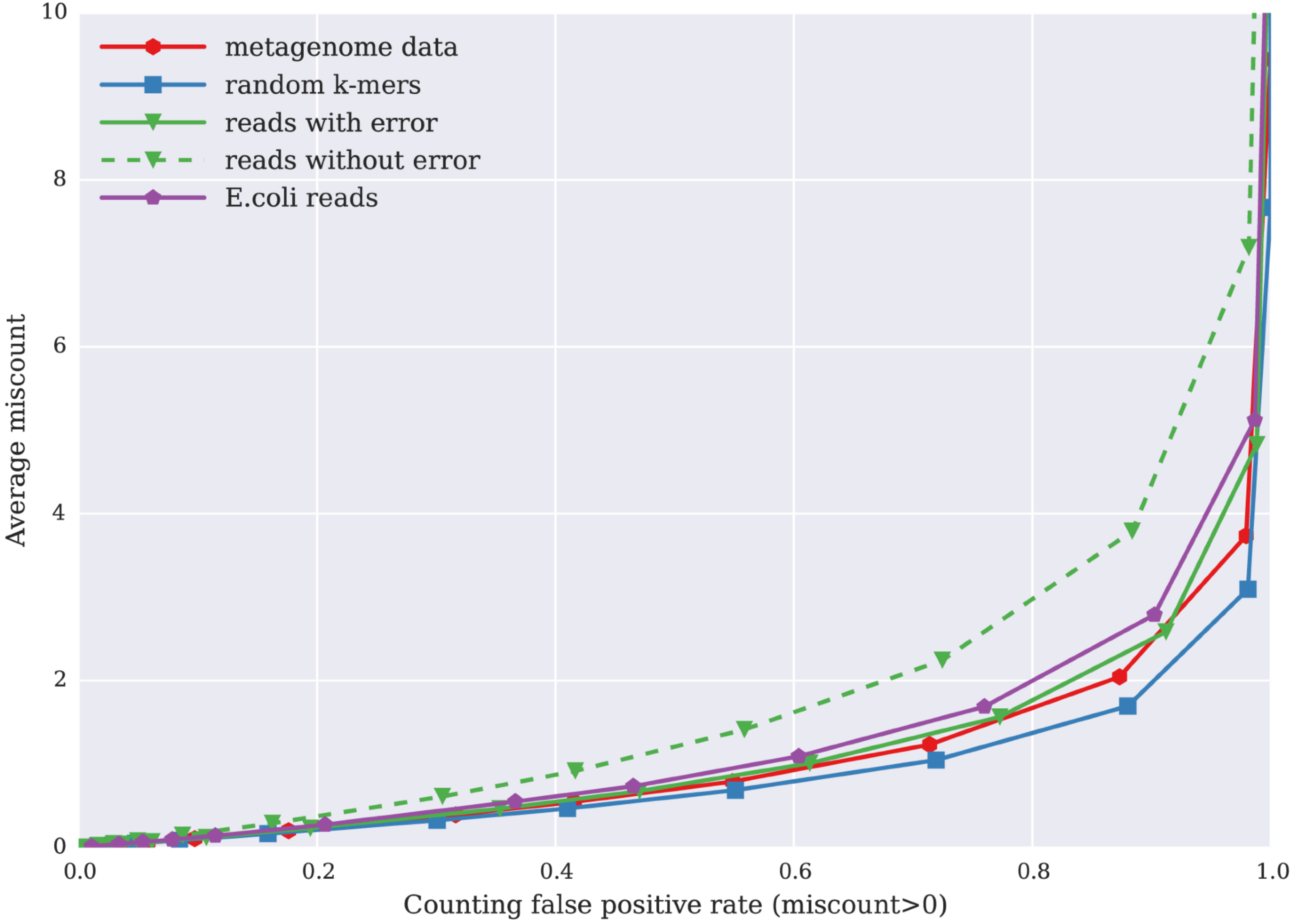
*from Zhang et al. PLoS ONE, 2014

Querying for random k-mers



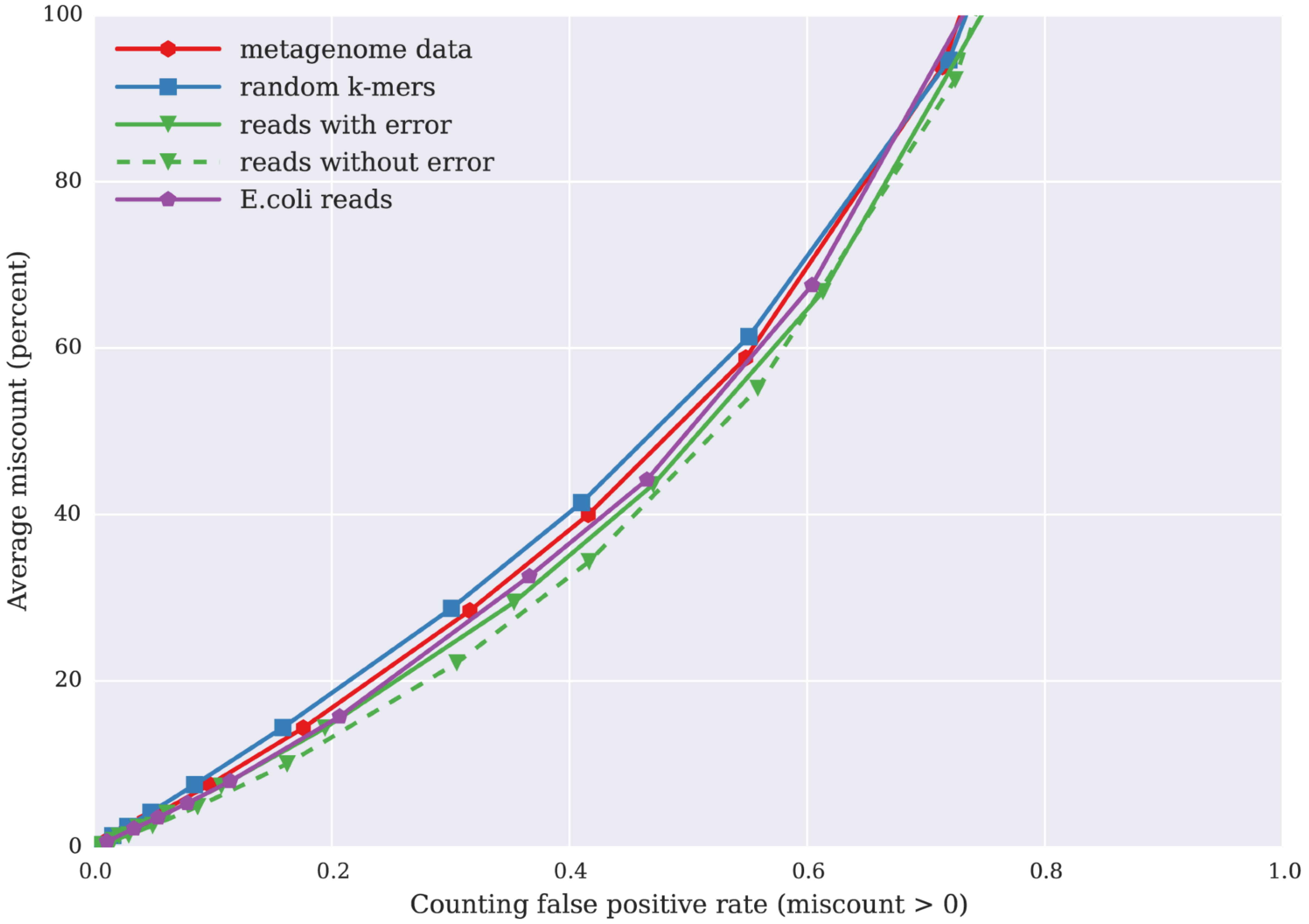
*from Zhang et al. PLoS ONE, 2014

Miscount & FP rate; changing ϵ and δ



*from Zhang et al. PLoS ONE, 2014

Miscount & FP-rate

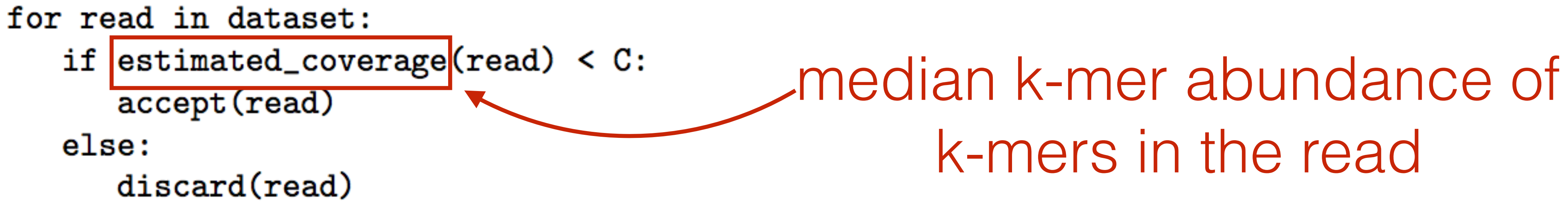


*from Zhang et al. PLoS ONE, 2014

Other uses of this approach

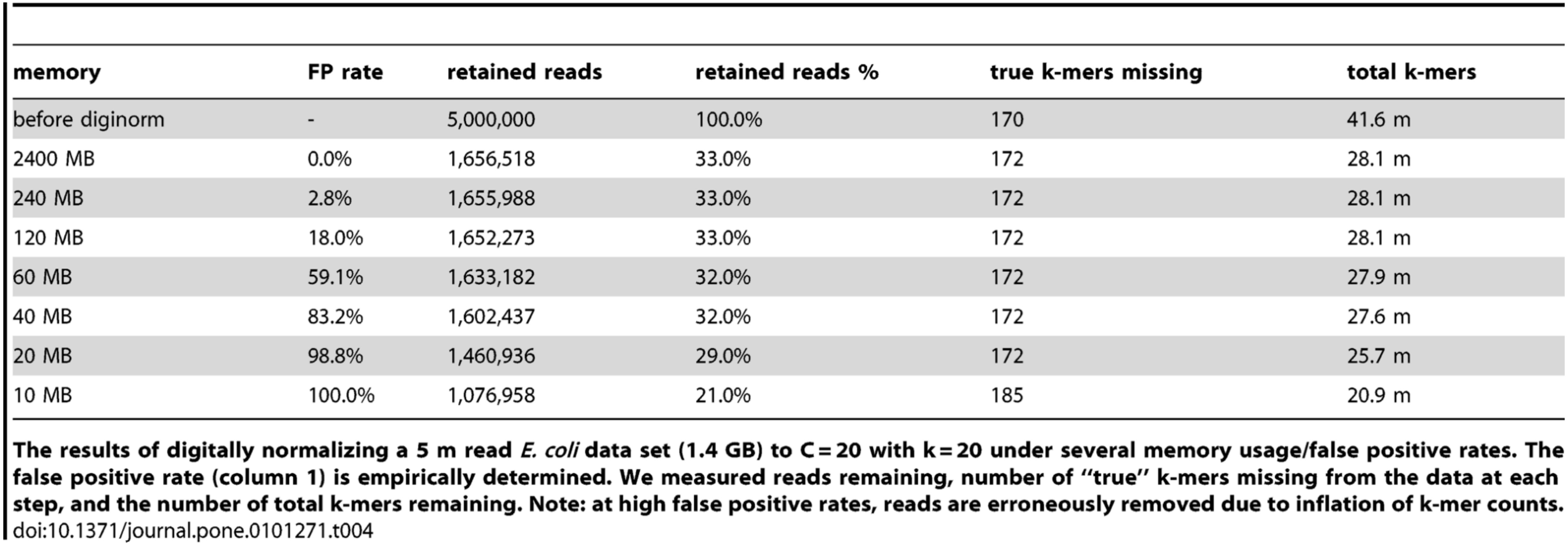
Khmer has been used successfully for other tasks e.g. digital normalization:

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



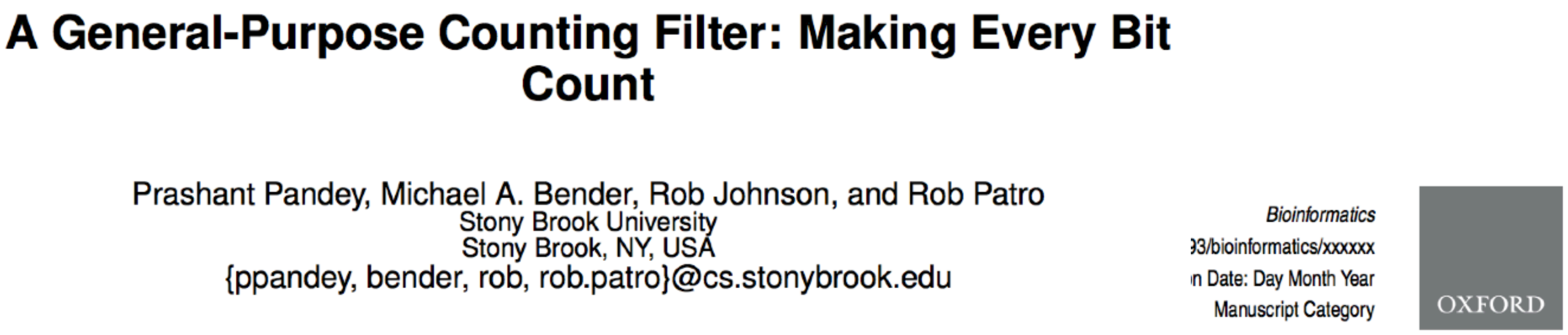
memory	FP rate	retained reads	retained reads %	true k-mers missing	total k-mers
before diginorm	-	5,000,000	100.0%	170	41.6 m
2400 MB	0.0%	1,656,518	33.0%	172	28.1 m
240 MB	2.8%	1,655,988	33.0%	172	28.1 m
120 MB	18.0%	1,652,273	33.0%	172	28.1 m
60 MB	59.1%	1,633,182	32.0%	172	27.9 m
40 MB	83.2%	1,602,437	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

*from Zhang et al. PLoS ONE, 2014

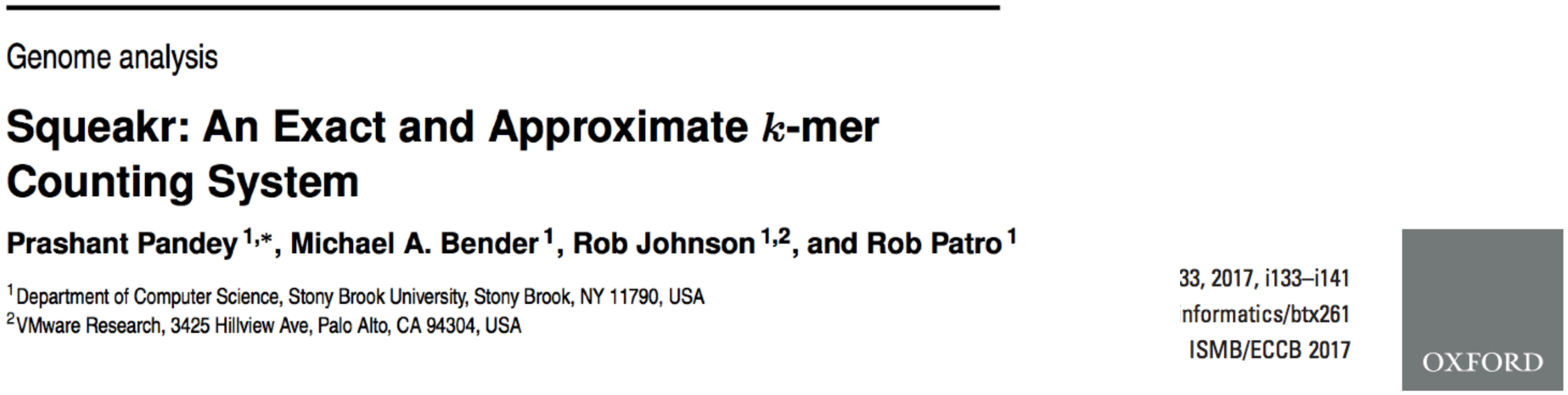
Work along these lines at SBU



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

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



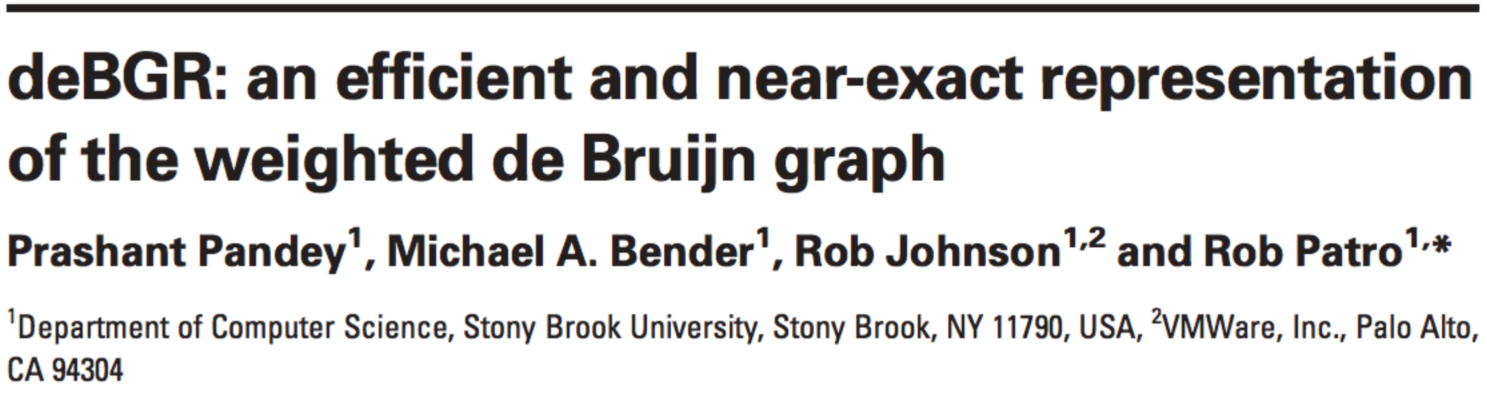
Genome analysis

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

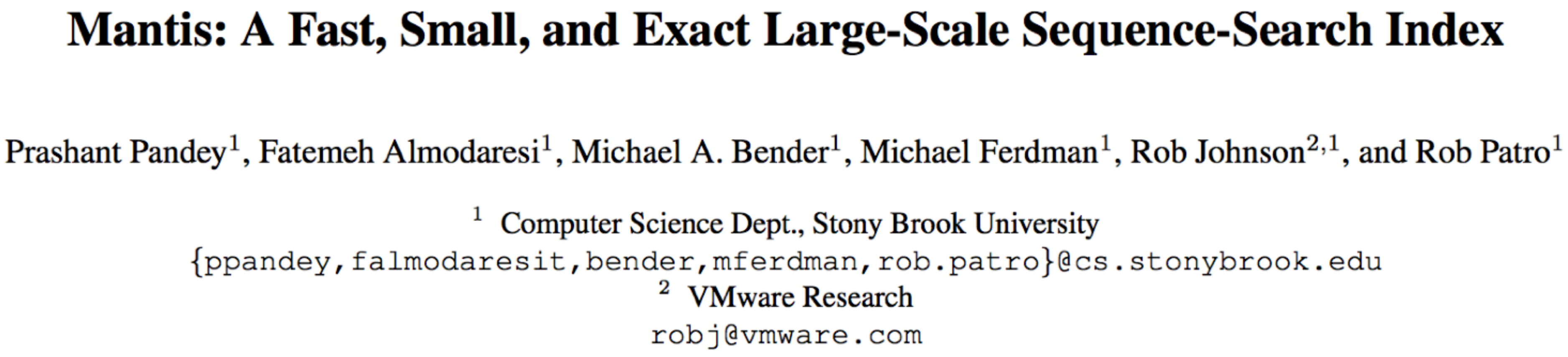
33, 2017, i133–i141
nformatics/btx261
ISMB/ECCB 2017
OXFORD



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

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, Inc., Palo Alto, CA 94304



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

Prashant Pandey[1], Fatemeh Almodaresi[1], Michael A. Bender[1], Michael Ferdman[1], Rob Johnson[2,1], and Rob Patro[1]

[1] Computer Science Dept., Stony Brook University
{ppandey,falmodaresit,bender,mferdman,rob.patro}@cs.stonybrook.edu
[2] VMware Research
robj@vmware.com

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 ∊ S, h(x) is a p-bit number.

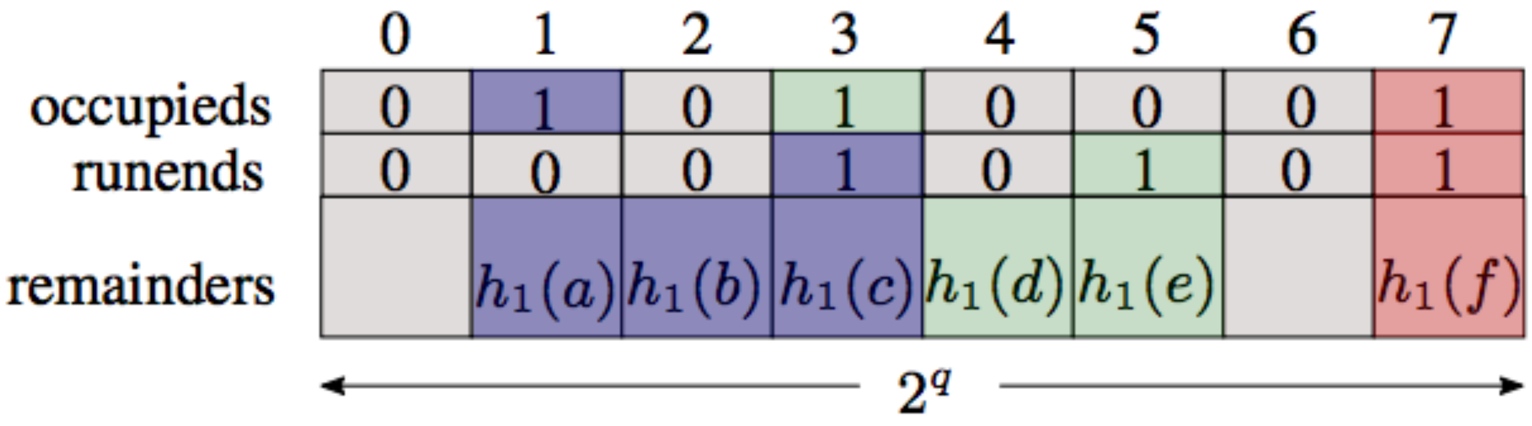
Q is an array of 2^q r-bit slots

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		$h_1(a)$	$h_1(b)$	$h_1(c)$	$h_1(d)$	$h_1(e)$		$h_1(f)$

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

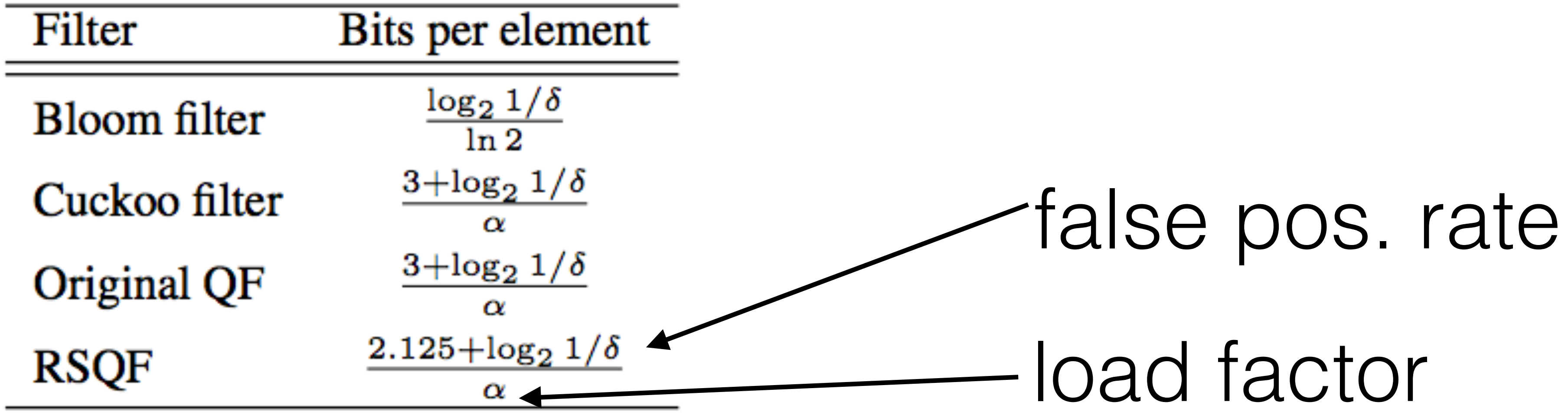
How to count?

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

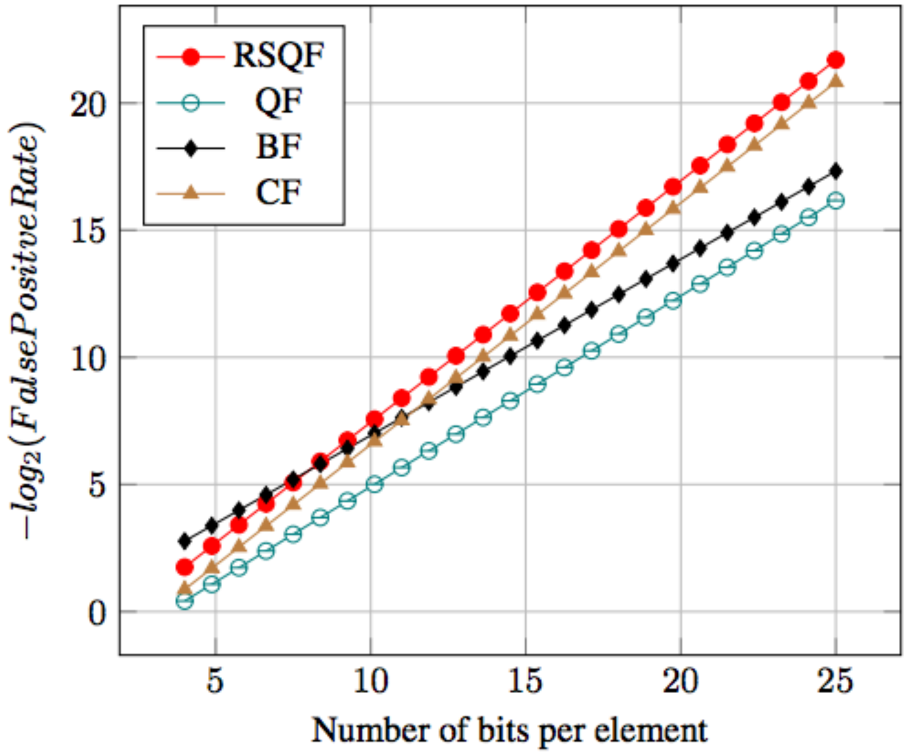
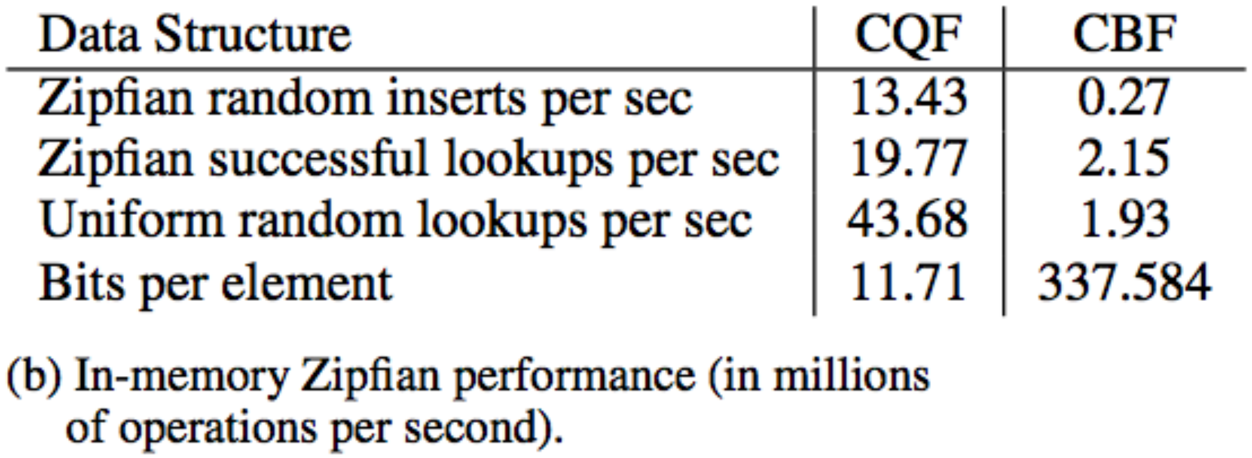


Figure 4: Number of bits per element for the RSQF, QF, BF, and CF. The RSQF requires less space than the CF amd less space than the BF for any false-positive rate less than 1/64. (Higher is better)

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).

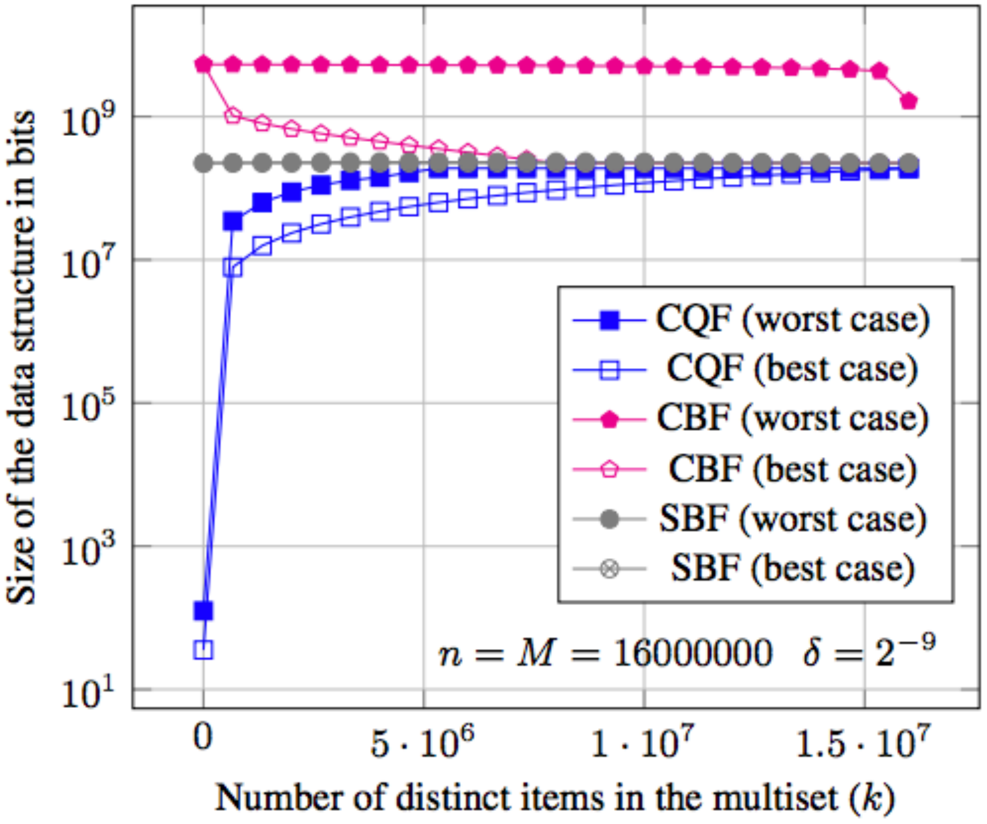
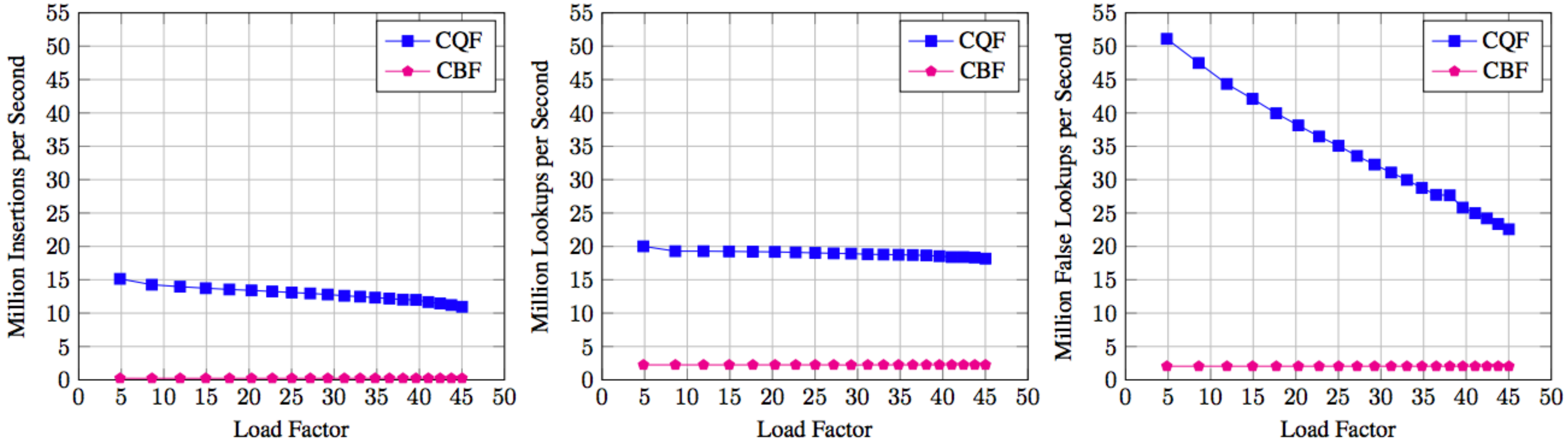


Figure 5: Space comparison of CQF, SBF, and CBF as a function of the number of distinct items. All data structures are built to support up to $n = 1.6 \times 10^7$ insertions with a false-positive rate of $\delta = 2^{-9}$.

The Counting Quotient Filter, results



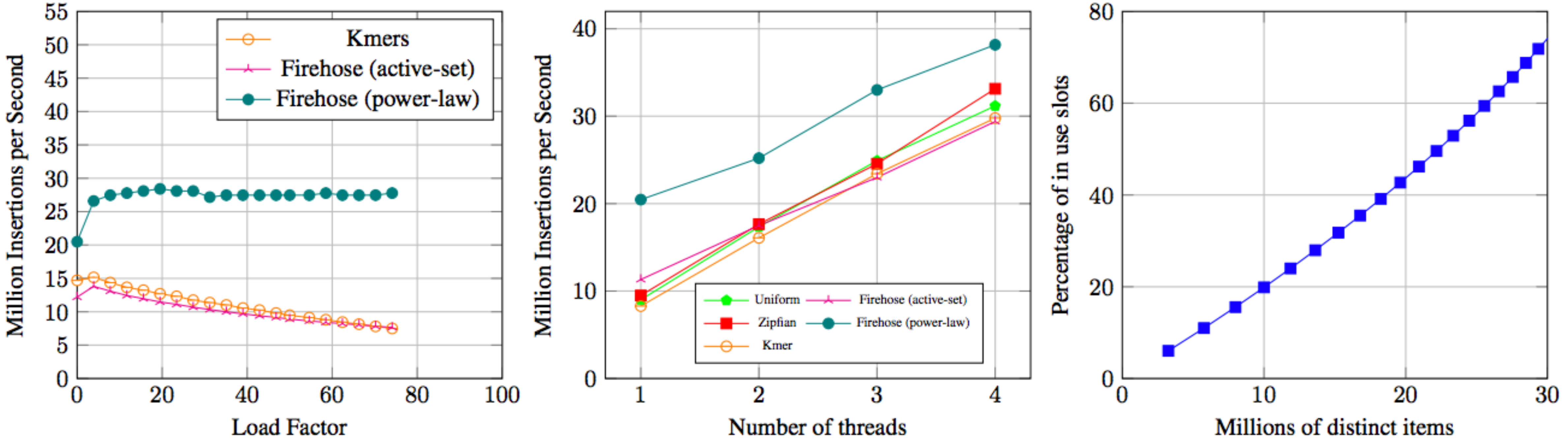
(a) Inserts.

(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

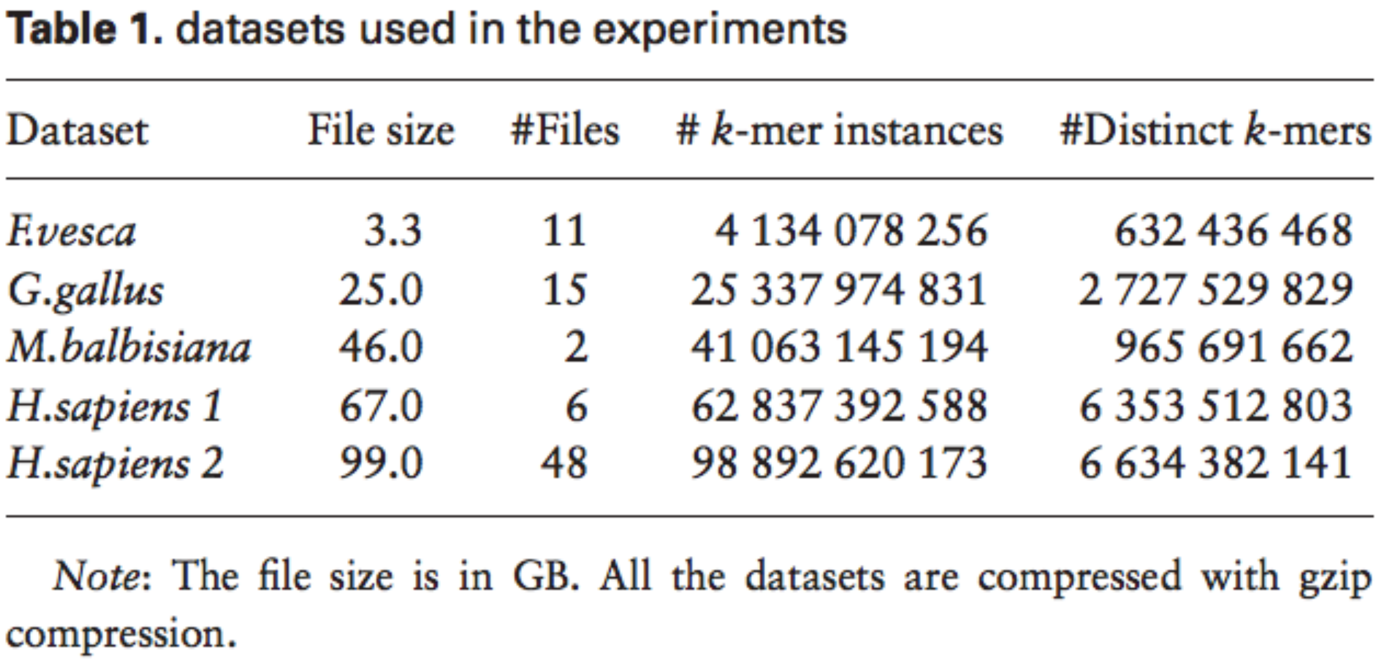


Table 1. datasets used in the experiments

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.

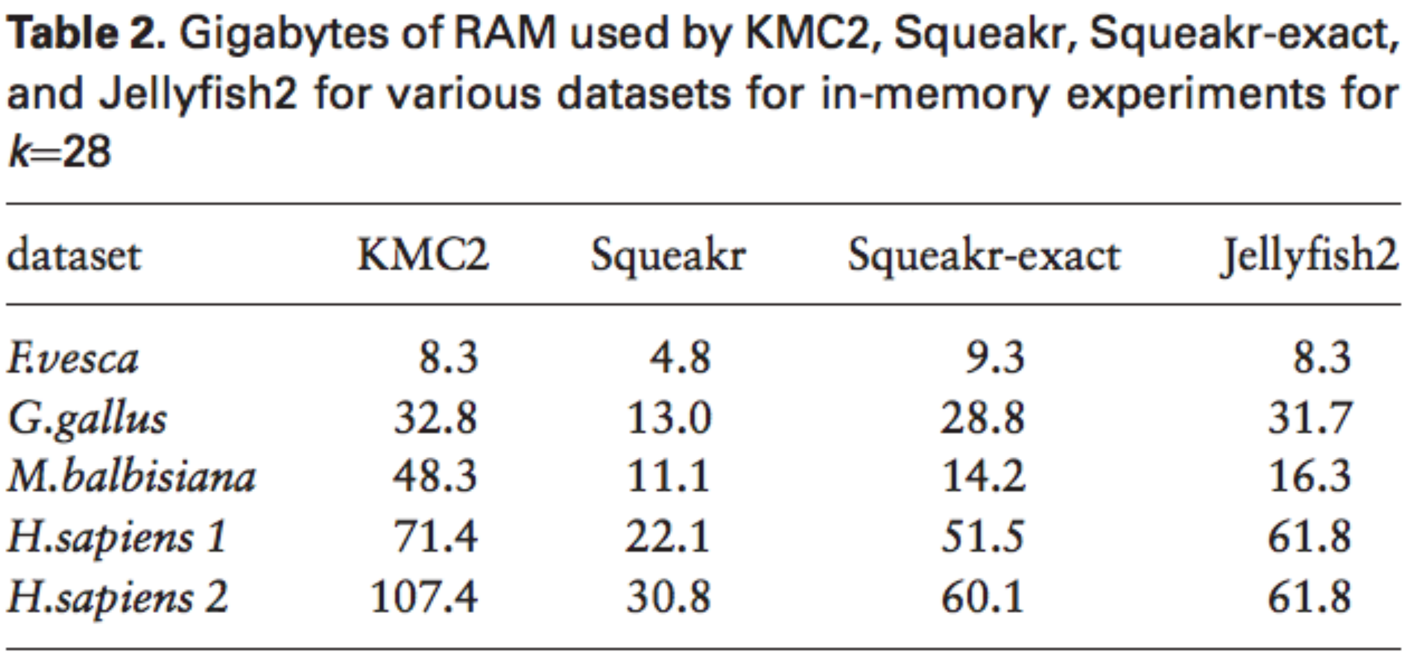


Table 2. Gigabytes of RAM used by KMC2, Squeakr, Squeakr-exact, and Jellyfish2 for various datasets for in-memory experiments for *k*=28

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

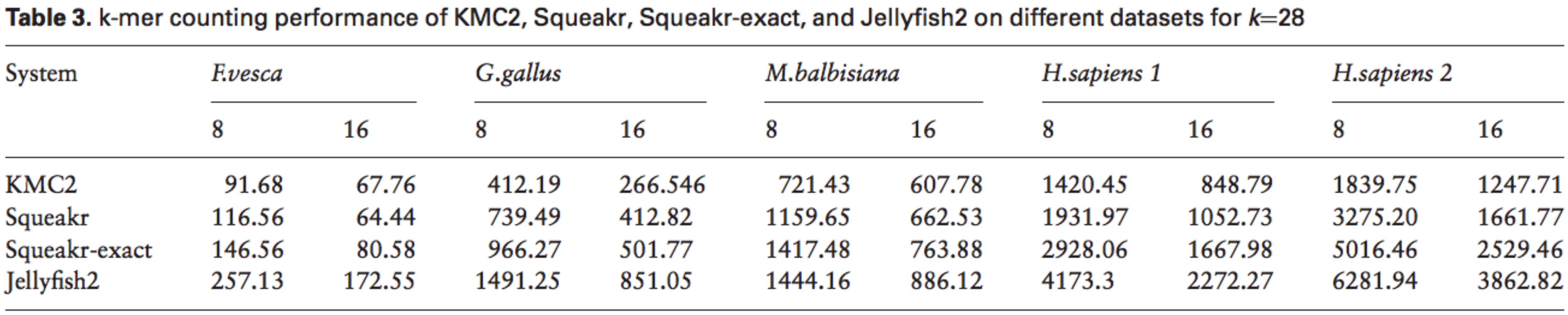


Table 3. k-mer counting performance of KMC2, Squeakr, Squeakr-exact, and Jellyfish2 on different datasets for *k*=28

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

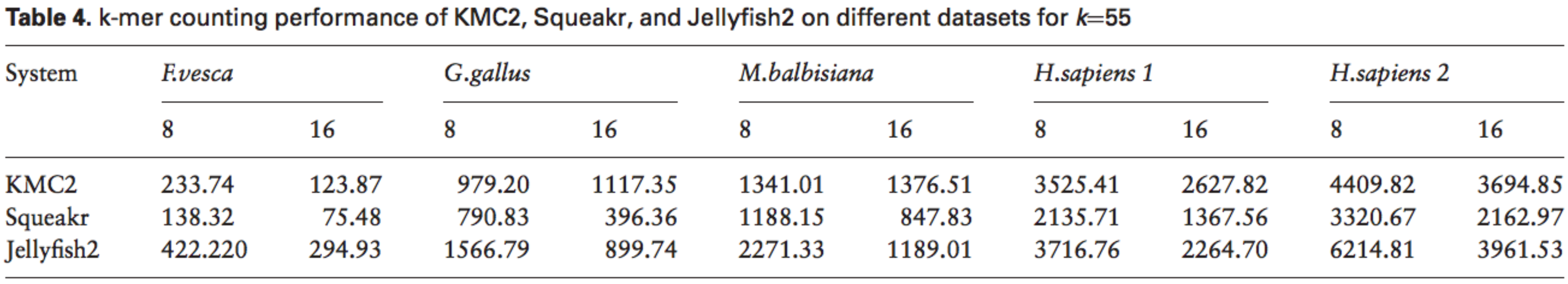


Table 4. k-mer counting performance of KMC2, Squeakr, and Jellyfish2 on different datasets for *k*=55

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

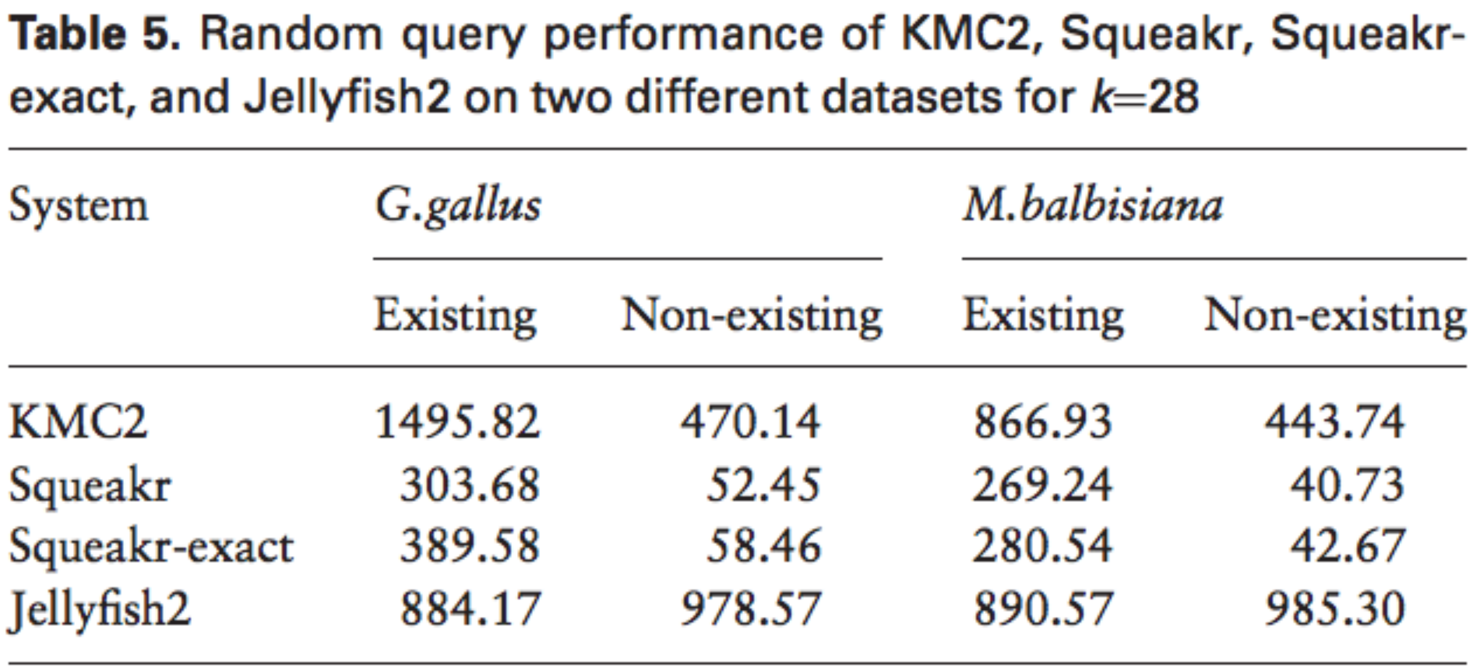


Table 5. Random query performance of KMC2, Squeakr, Squeakr-exact, and Jellyfish2 on two different datasets for *k*=28

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

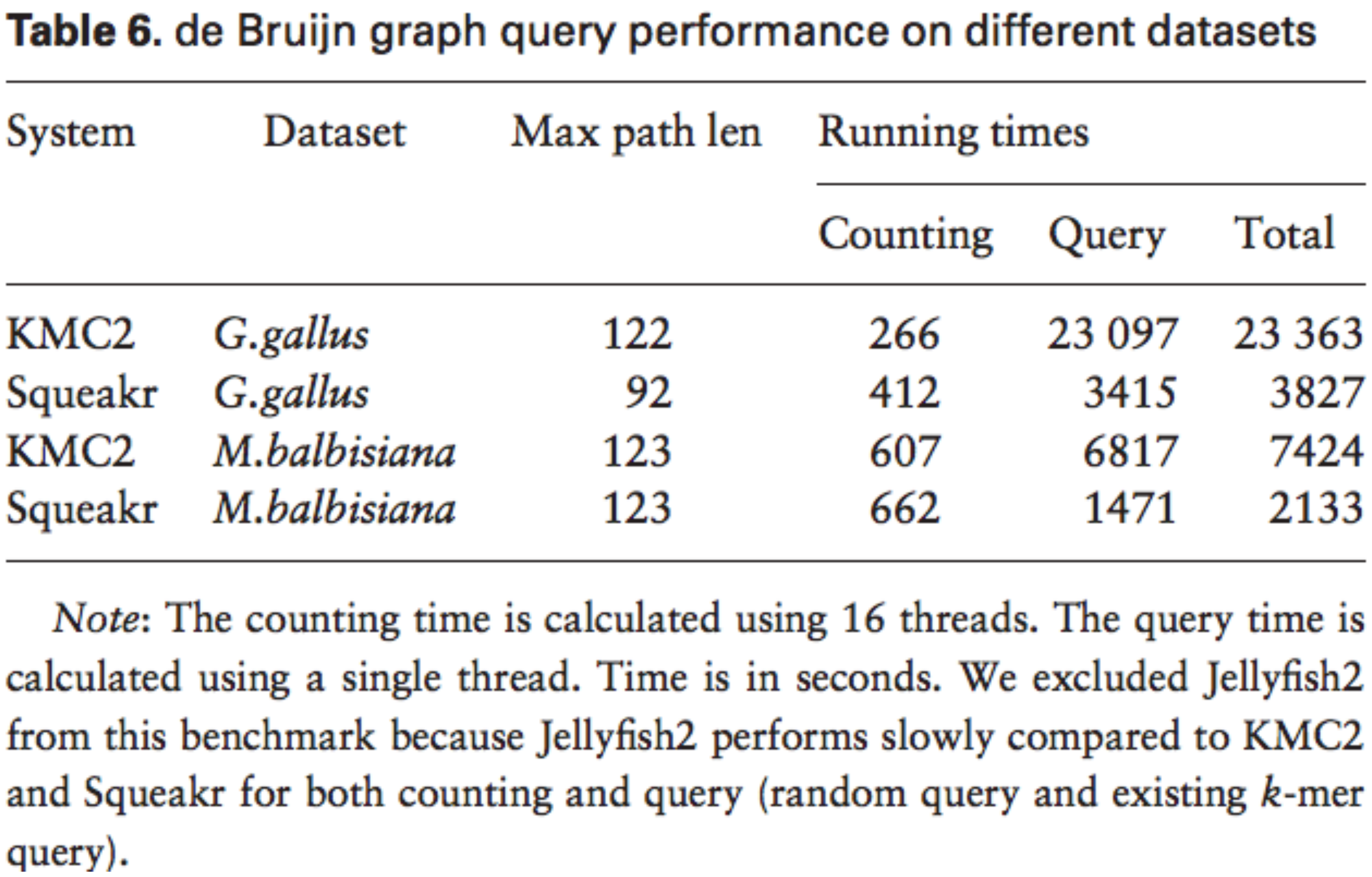


Table 6. de Bruijn graph query performance on different datasets

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).

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!