

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

# 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

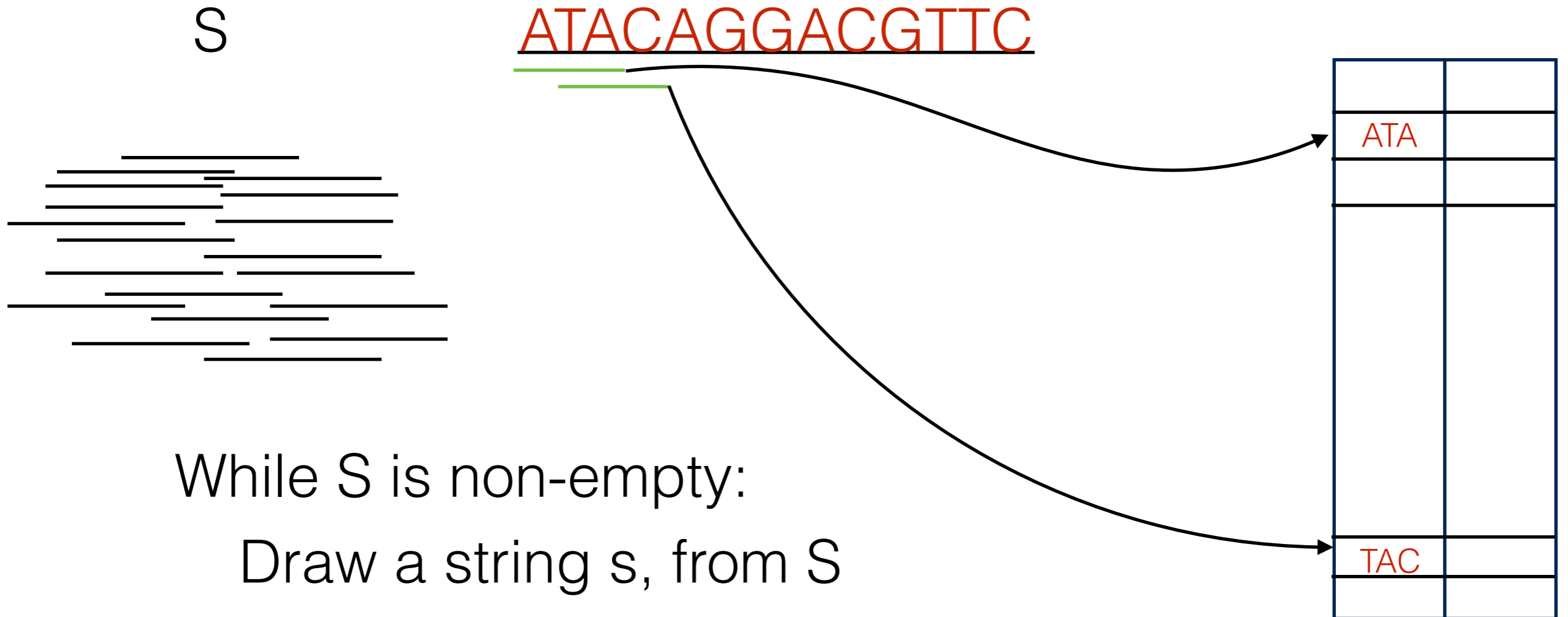
# k-mer counting

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

Tallymer, Jellyfish, DSK, KMC, BFCCounter, 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  $> 100$ GB 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

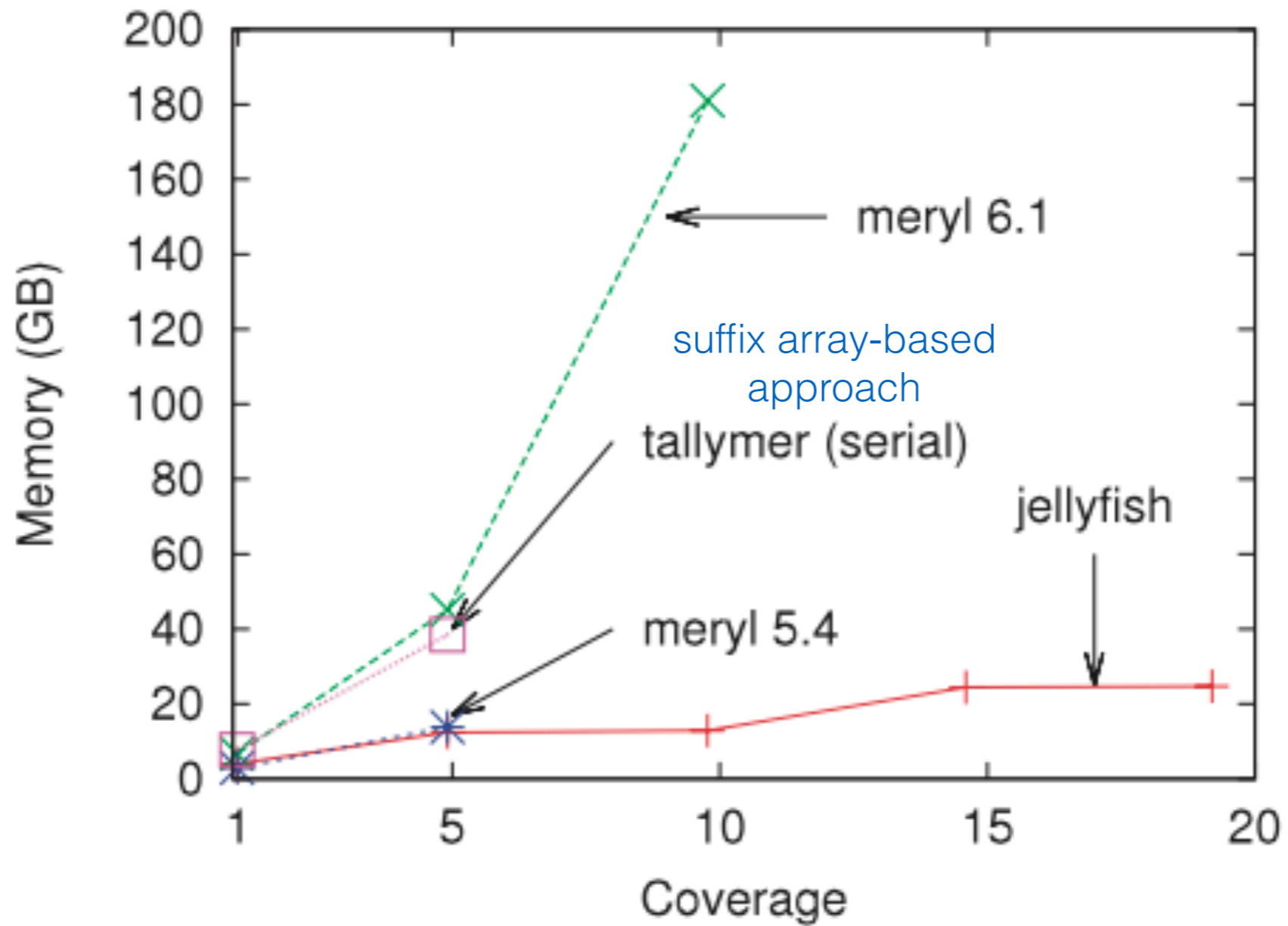
— small counter with multiple entries for high-count k-mers

Efficient storage of keys

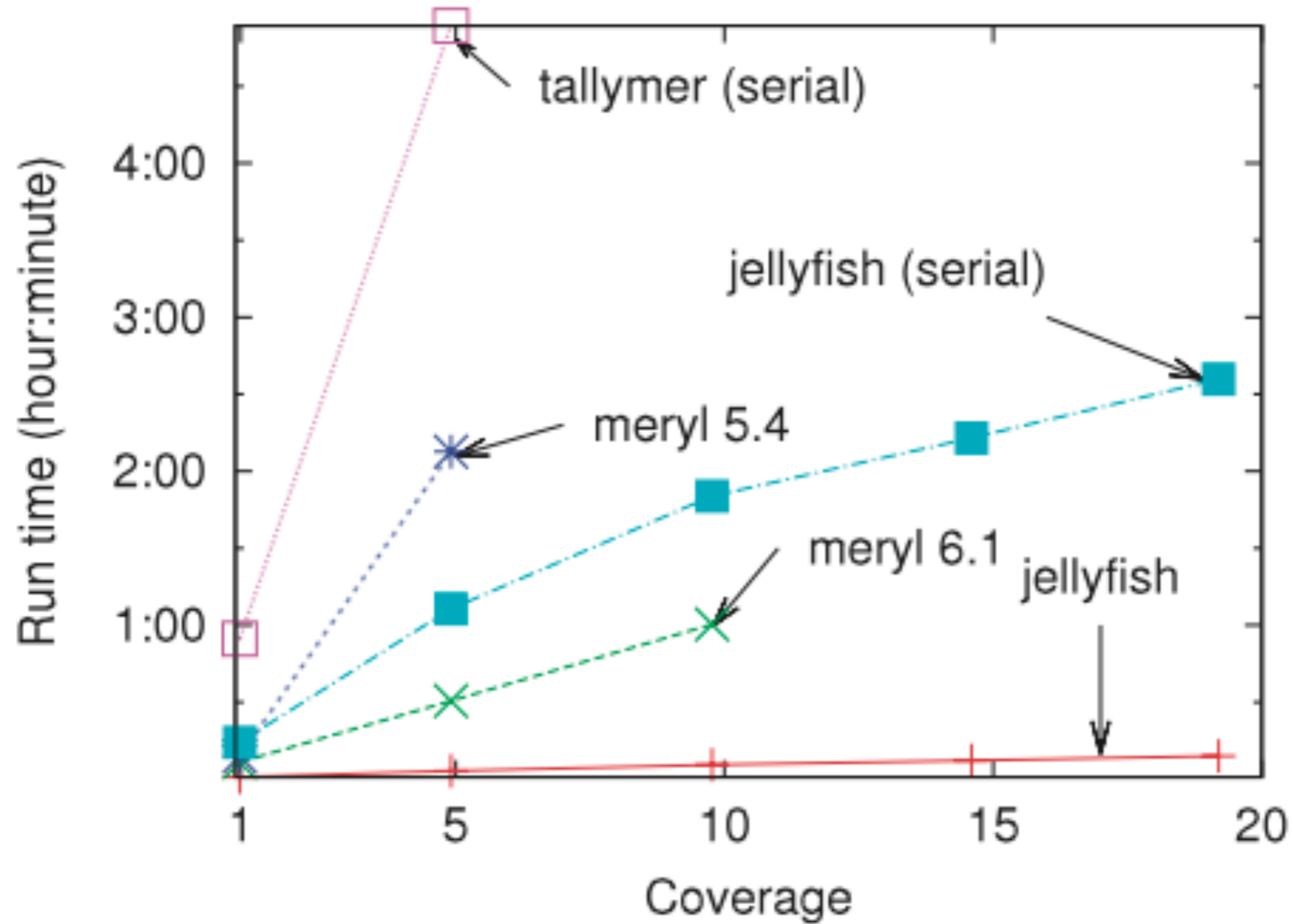
—  $f: U_k \rightarrow U_k$ , and let  $\text{hash}(k) = f(k) \bmod M$

— Can reconstruct  $k$  from pos in hash table, remainder and number of re-probes (small # )

# Memory usage of Jellyfish

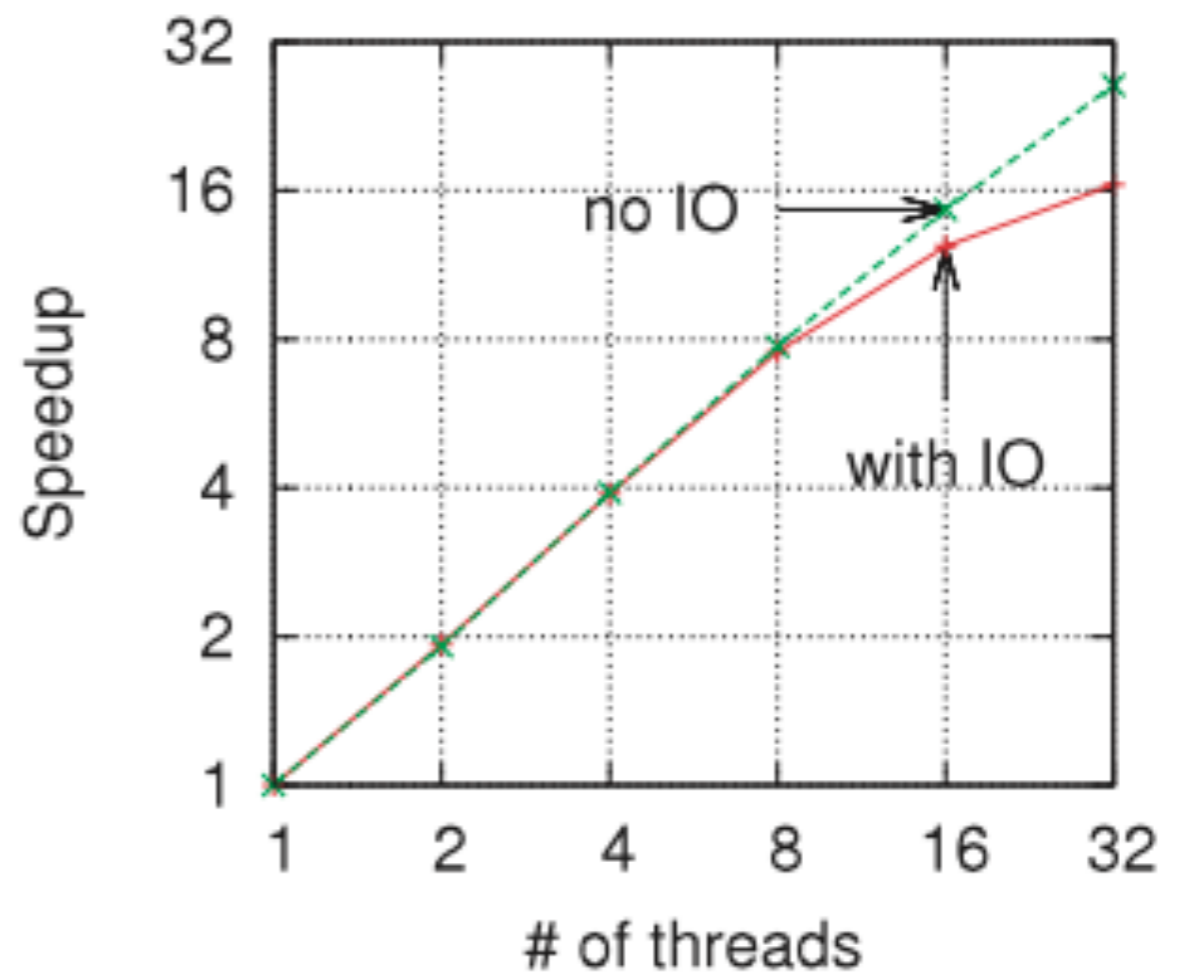
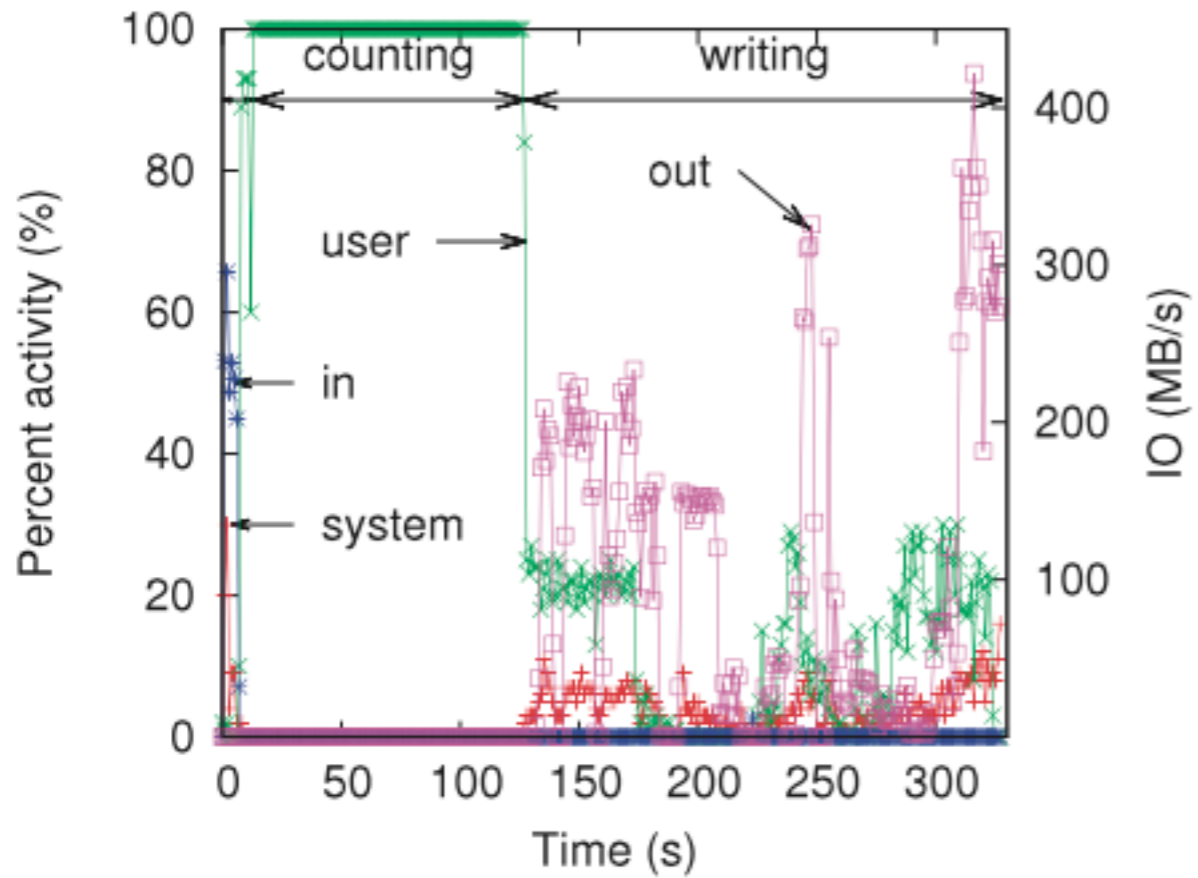


# Runtime of Jellyfish





# 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.

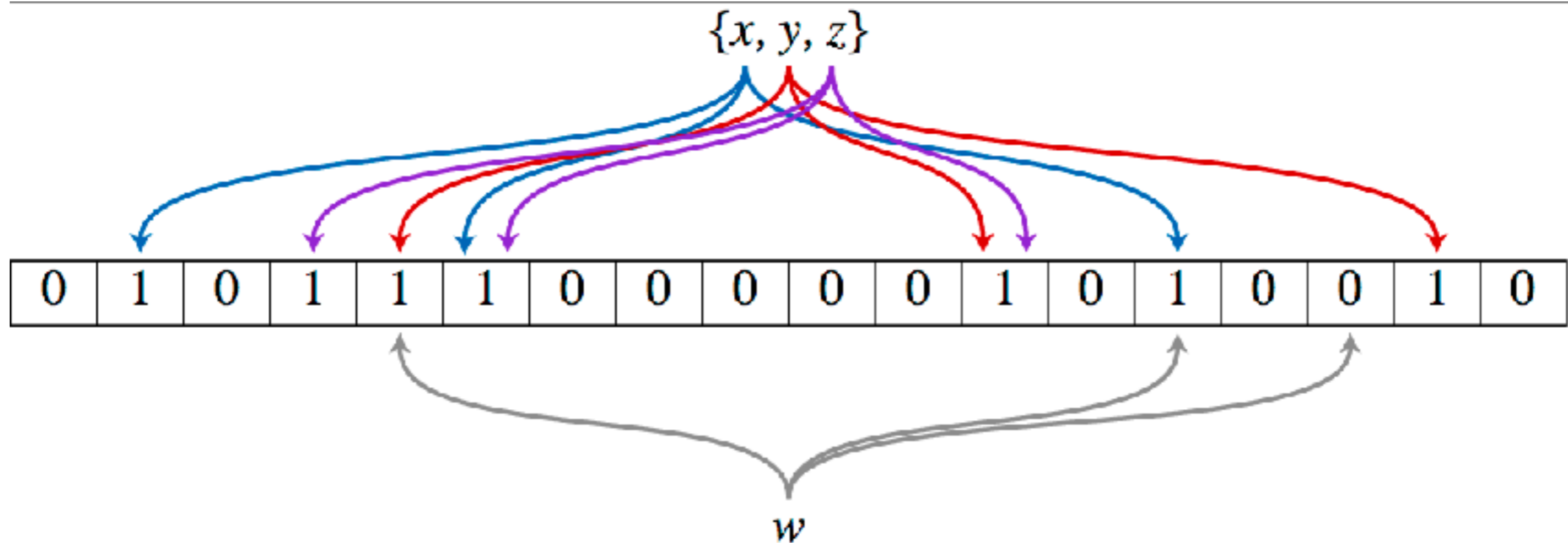
# Bloom Filters

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  $0 < i < k$

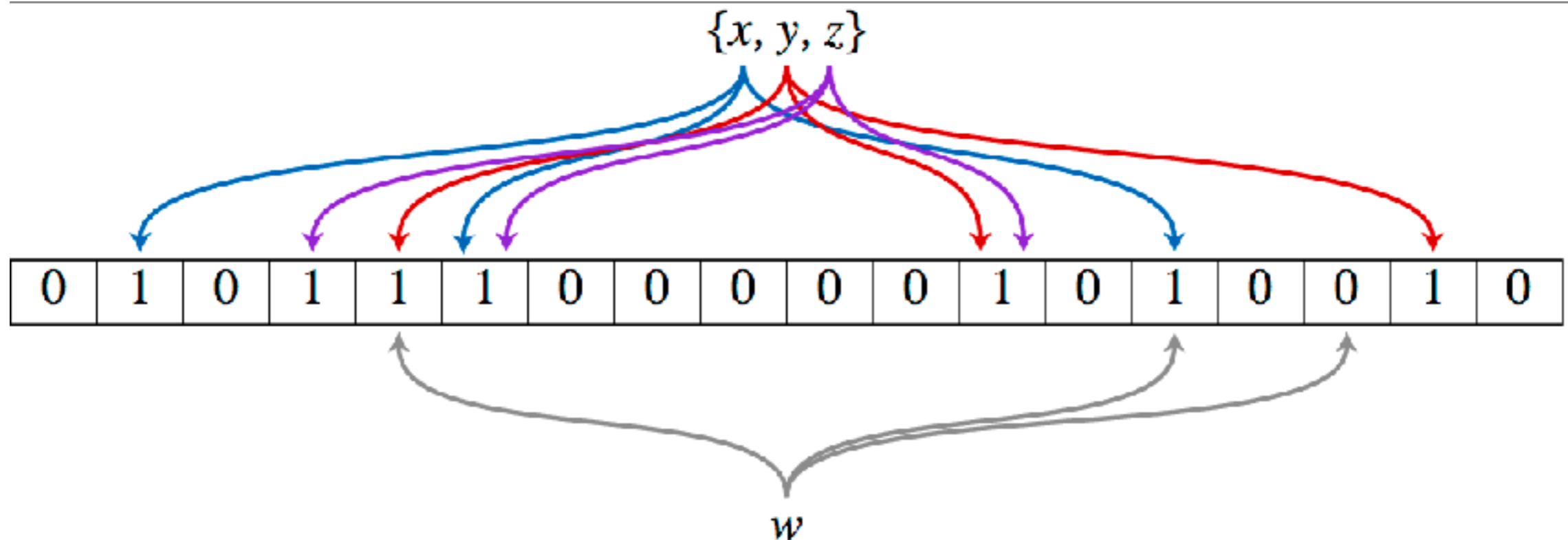
To query for  $e$ , check if  $A[h_i(e)] = 1$  for  $0 < i < k$



# Bloom Filters

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

How low?



# 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 & Unfal 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

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

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

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

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

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

given an **expected # elems** and a **desired false positive rate** we can compute the **optimal size** and **# of has functions**

# Detour: Bloom Filters & De Bruijn Graphs

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

# Detour: Bloom Filters & De Bruijn Graphs

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

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

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

# Detour: Bloom Filters & De Bruijn Graphs

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



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

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

# Bloom Filters & De Bruijn Graphs

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

May return some neighbors that are not actually present.

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

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

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

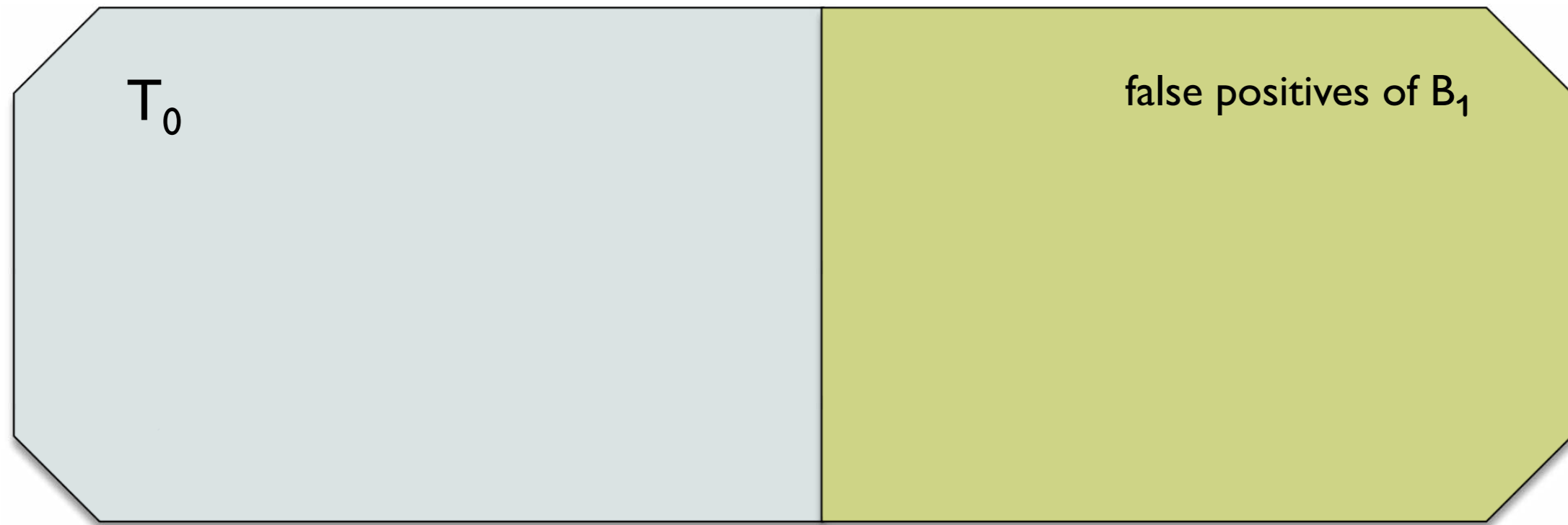
# Idea of Chkhi and Rizk

Assume we want to represent specific set  $T_0$  of  $k$ -mers with a Bloom filter  $B_1$

*Key observation:* in assembly, not all  $k$ -mers can be queried, only those having  $k-1$  overlap with  $k$ -mers known to be in the graph.

The set  $T_1$  of “critical false positives” (false neighbors of true  $k$ -mers) is *much* smaller than the set of all false positives and can be stored explicitly

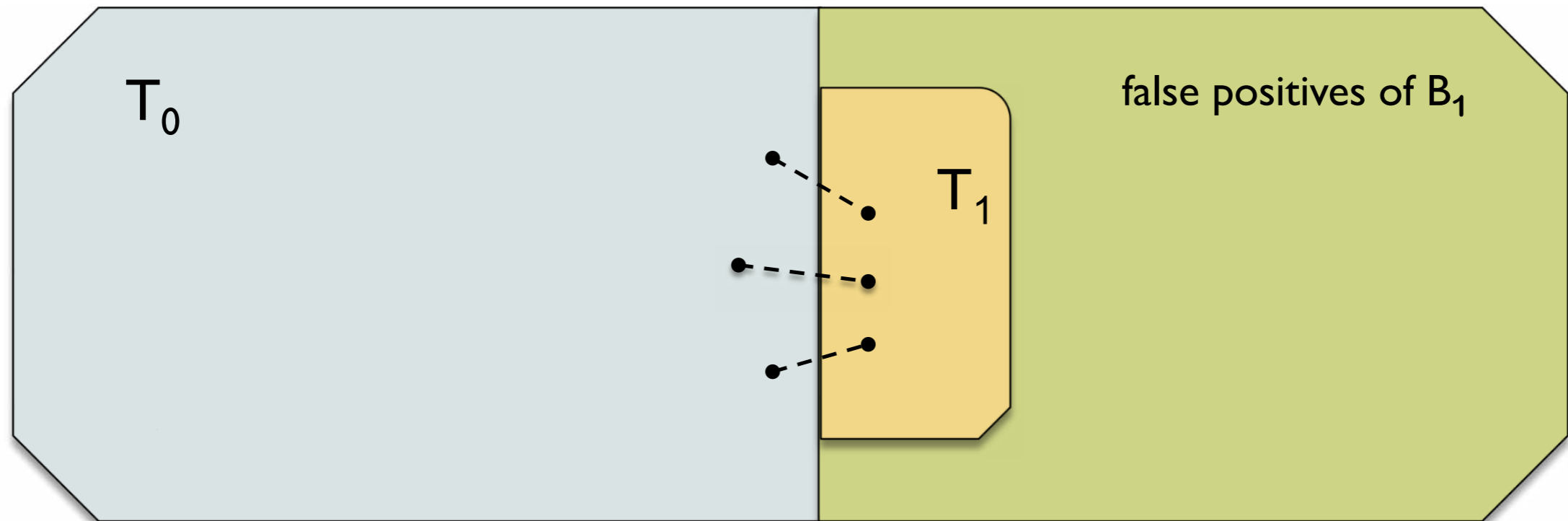
Storing  $B_1$  and  $T_1$  is much more space efficient than other exact methods for storing  $T_0$ . Membership of  $w$  in  $T_0$  is tested by first querying  $B_1$ , and if  $w \in B_1$ , check that it is *not* in  $T_1$ .



- ▶ Represent  $T_0$  by Bloom filter  $B_1$

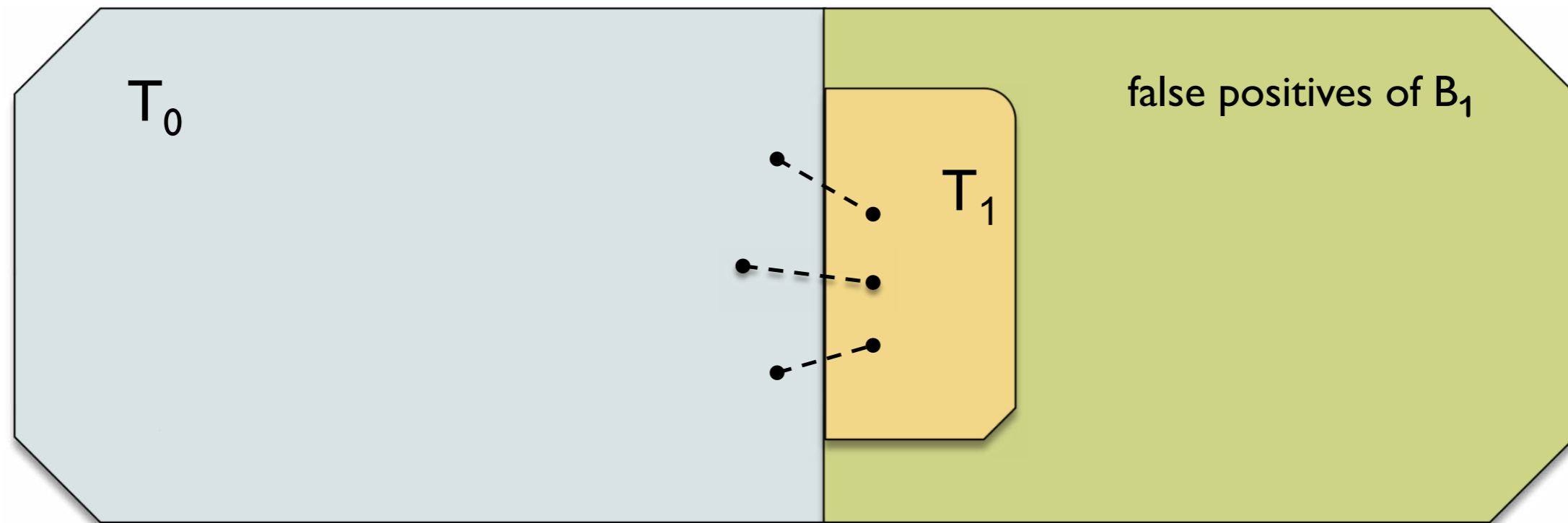






- ▶ Represent  $T_0$  by Bloom filter  $B_1$
- ▶ Compute  $T_1$  ('critical false positives') and represent it e.g. by a hash table




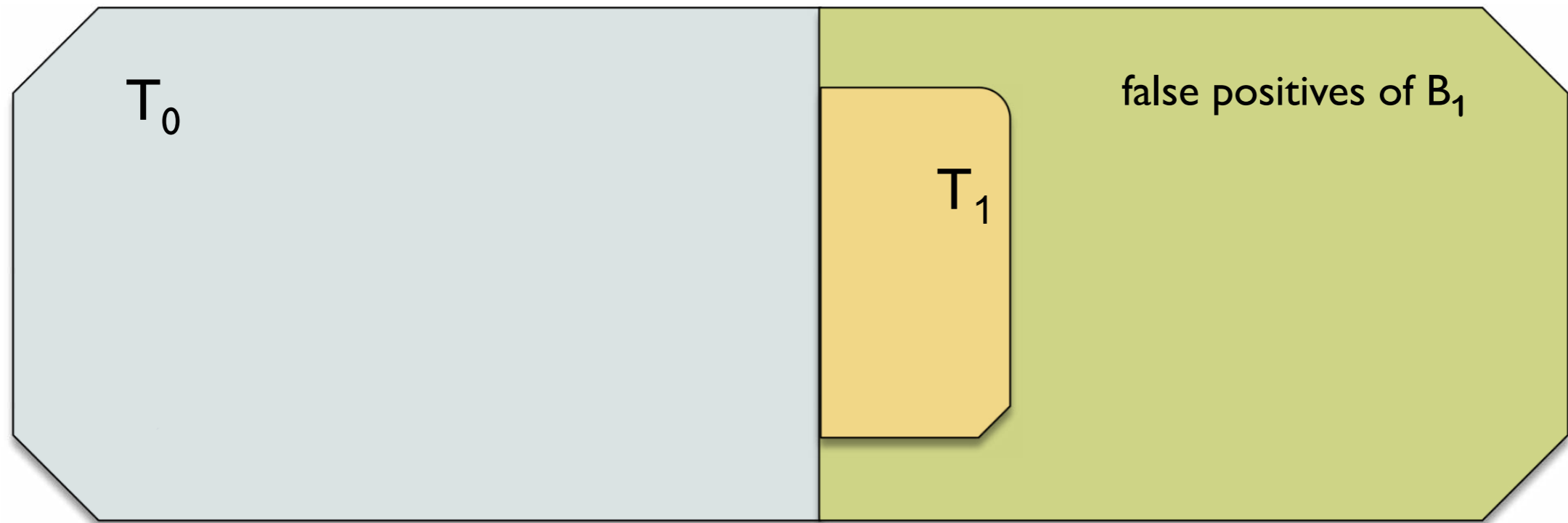


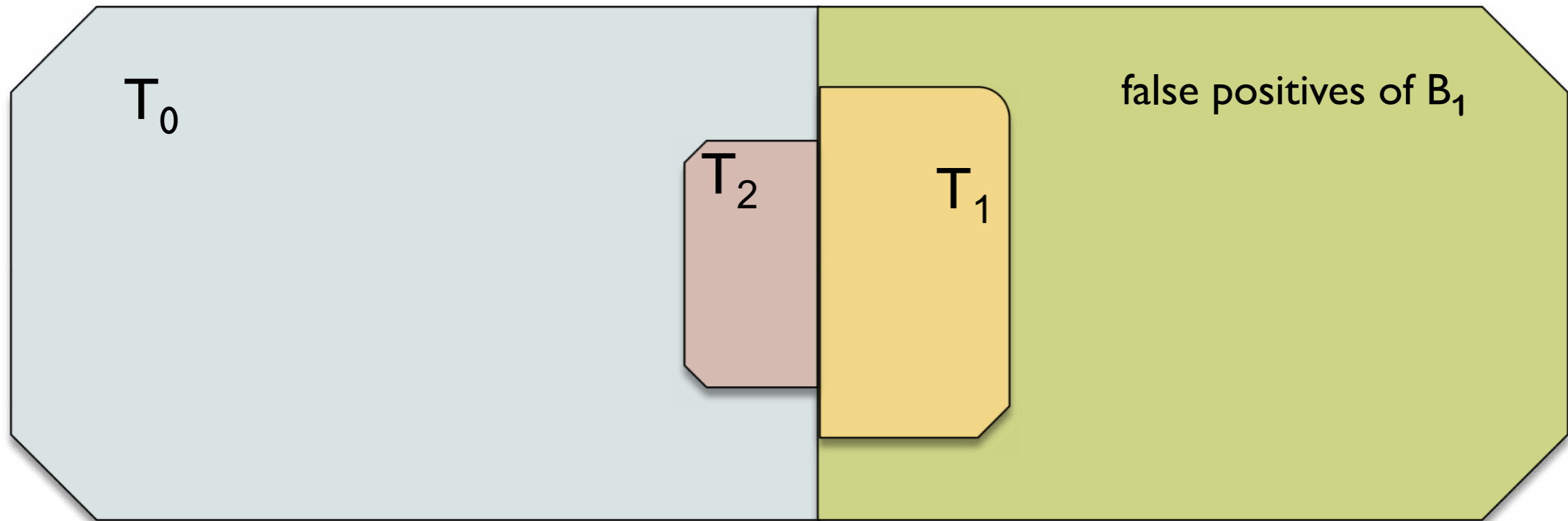
- ▶ Represent  $T_0$  by Bloom filter  $B_1$
- ▶ Compute  $T_1$  ('critical false positives') and represent it e.g. by a hash table
- ▶ Result (example): **13.2** bits/node for  $k=27$  (of which 11.1 bits for  $B_1$  and 2.1 bits for  $T_1$ )



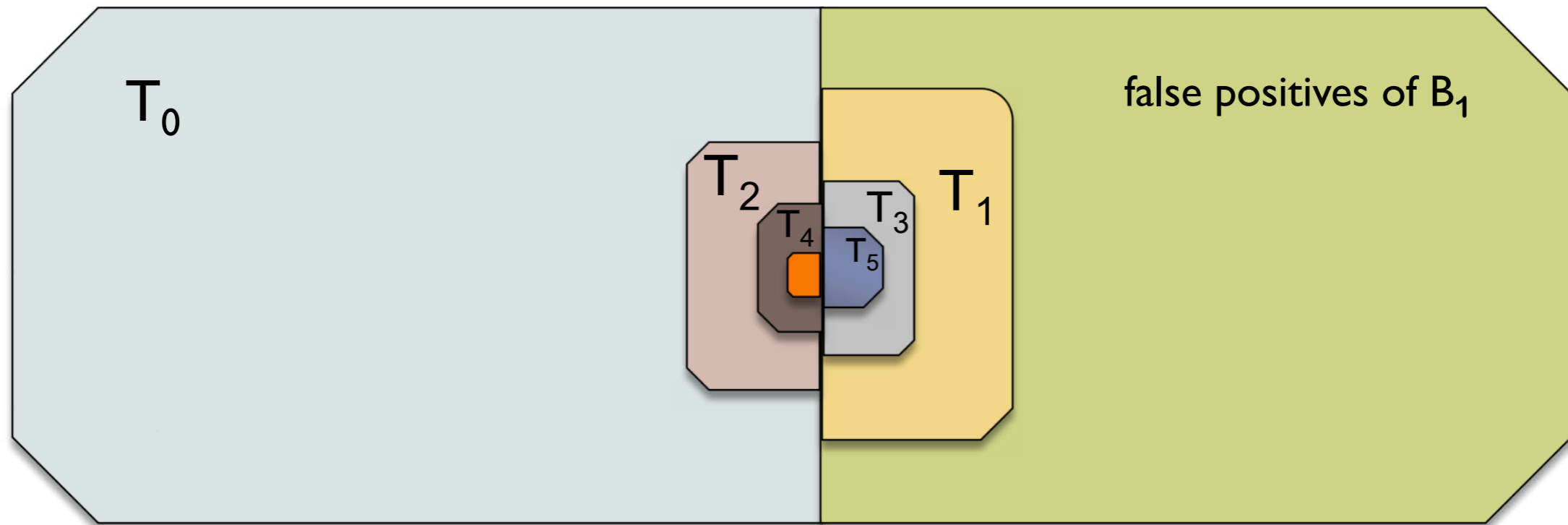
# Improving on Chikhi and Rizk's method

- ▶ *Main idea*: iteratively apply the same construction to  $T_1$  i.e. encode  $T_1$  by a Bloom filter  $B_2$  and set of 'false-false positives'  $T_2$ , then apply this to  $T_2$  etc.
- ▶  *cascading Bloom filters*

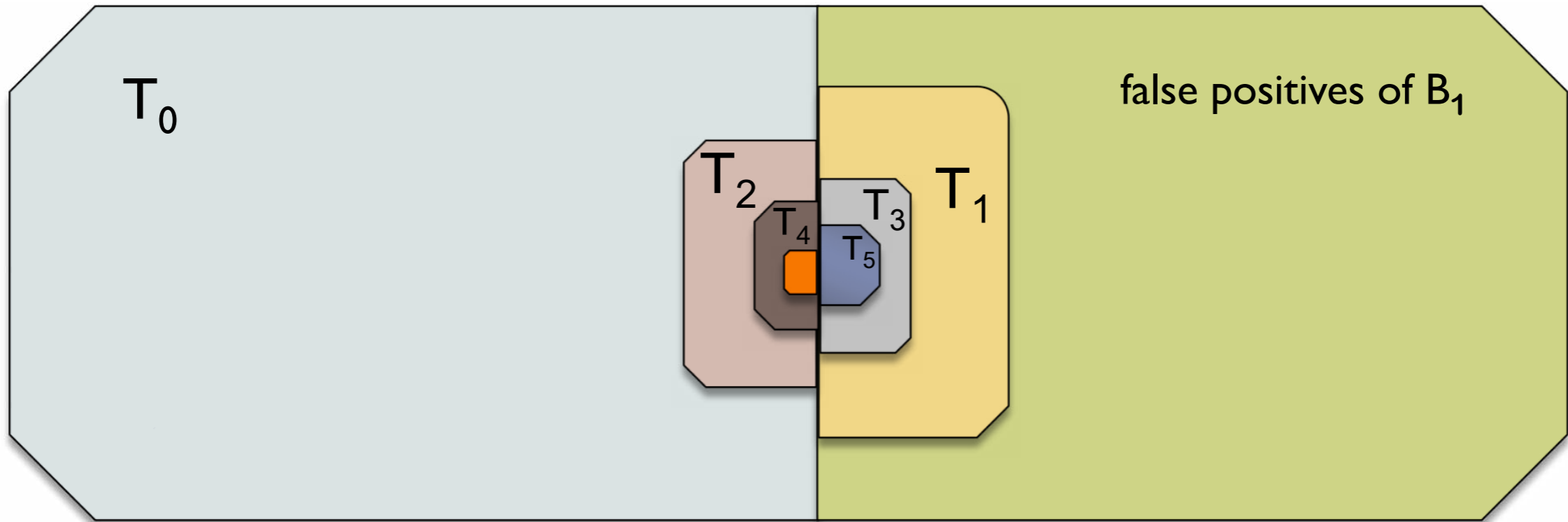




- ▶ further encode  $T_1$  via a Bloom filter  $B_2$  and set  $T_2$ , where  $T_2 \subseteq T_0$  is the set of  $k$ -mers stored in  $B_2$  by mistake ('false<sup>2</sup> positives')



- ▶ further encode  $T_1$  via a Bloom filter  $B_2$  and set  $T_2$ , where  $T_2 \subseteq T_0$  is the set of  $k$ -mers stored in  $B_2$  by mistake ('false<sup>2</sup> positives')
- ▶ iterate the construction on  $T_2$
- ▶ we obtain a sequence of sets  $T_0, T_1, T_2, T_3, \dots$  encode by Bloom filters  $B_1, B_2, B_3, B_4, \dots$  respectively
- ▶  $T_0 \supseteq T_2 \supseteq T_4 \supseteq \dots, T_1 \supseteq T_3 \supseteq T_5 \supseteq$



**Lemma [correctness]:** For a  $k$ -mer  $w$ , consider the smallest  $i$  such that  $w \notin B_{i+1}$ . Then  $w \in T_0$  if  $i$  is odd and  $w \notin T_0$  if  $i$  is even.

- ▶ if  $w \notin B_1$  then  $w \notin T_0$
- ▶ if  $w \in B_1$ , but  $w \notin B_2$  then  $w \in T_0$
- ▶ if  $w \in B_1$ ,  $w \in B_2$ , but  $w \notin B_3$  then  $w \notin T_0$
- ▶ etc.

# Assuming infinite number of filters

Let  $N=|T_0|$  and  $r=m_i/n_i$  is the same for every  $B_i$ . Then the total size is

$$\underbrace{rN}_{|B_1|} + \underbrace{6rNc^r}_{|B_2|} + \underbrace{rNc^r}_{|B_3|} + \underbrace{6rNc^{2r}}_{|B_4|} + \underbrace{rNc^{2r}}_{|B_5|} + \dots = N(1+6c^r) \frac{r}{1-c^r}$$

The minimum is achieved for  $r=5.464$ , which yields the memory consumption of **8.45** bits/node





# Infinity difficult to deal with ;)

- In practice we will store only a small finite number of filters  $B_1, B_2, \dots, B_t$  together with the set  $T_t$  stored explicitly
- $t=1 \Rightarrow$  Chkhi&Rizk's method
- The estimation should be adjusted, optimal value of  $r$  has to be updated, example for  $t=4$

$k$	optimal $r$	bits per $k$ -mer
16	5.776737	8.555654
32	6.048557	8.664086
64	6.398529	8.824496
128	6.819496	9.045435

**Table:** Estimations for  $t=4$ . Optimal  $r$  and corresponding memory consumption



# Compared to Chikhi&Rizk's method

$k$	“Optimal” (infinite) Cascading Bloom Filter	Cascading Bloom Filter with $t = 4$	Data structure of Chikhi & Rizk
16	8.45	8.555654	12.0785
32	8.45	8.664086	13.5185
64	8.45	8.824496	14.9585
128	8.45	9.045435	16.3985

**Table:** Space (bits/node) compared to Chikhi&Rizk  
for  $t=4$  and different values of  $k$ .

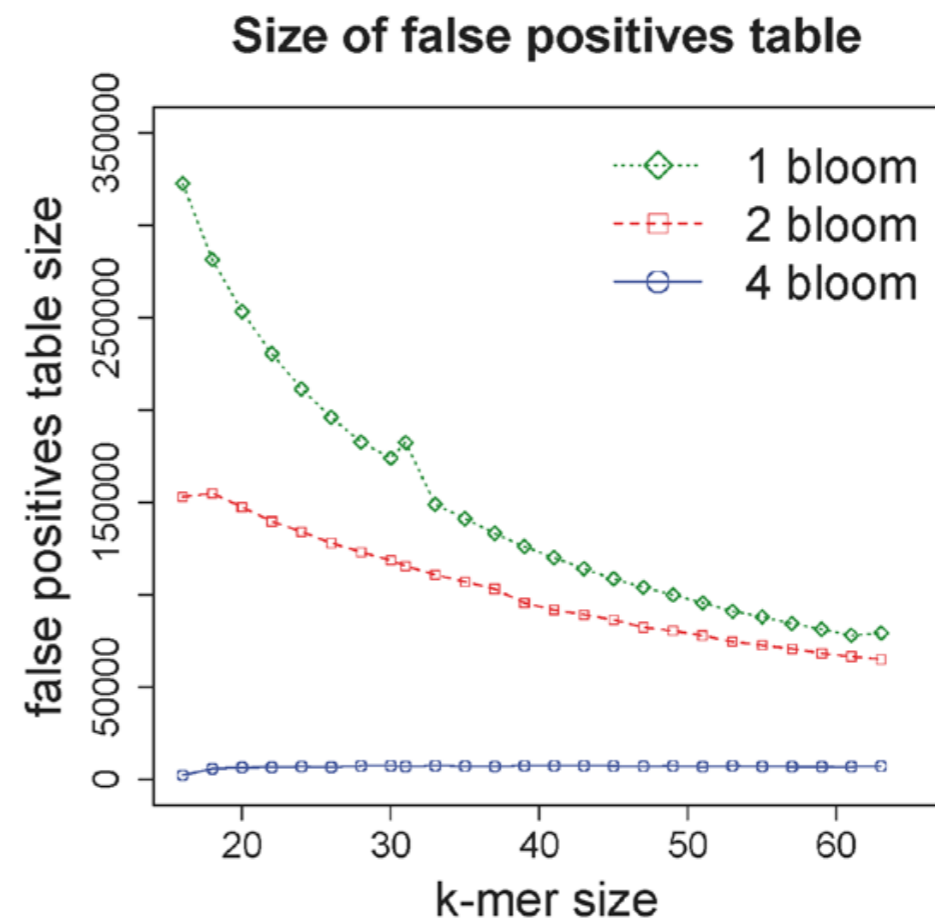
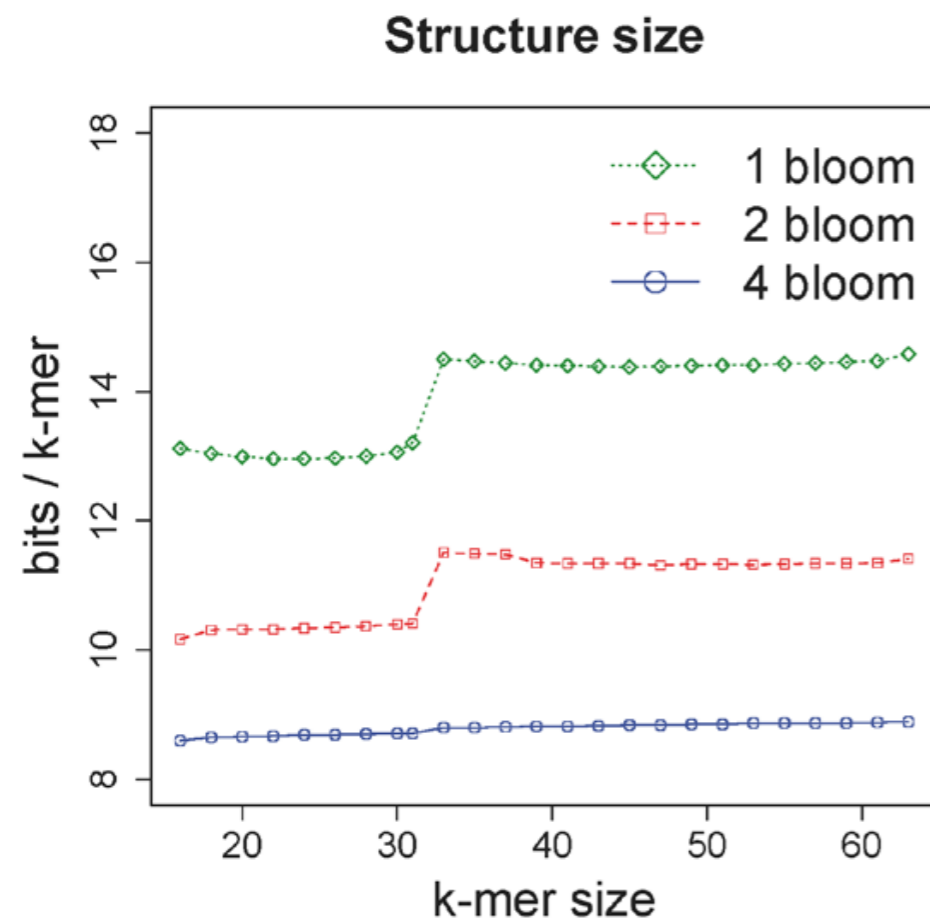
# We can cut down a bit more ...

- Rather than using the same  $r$  for all filters  $B_1, B_2, \dots$ , we can use different properly chosen coefficients  $r_1, r_2, \dots$
- This allows saving another 0.2 – 0.4 bits/ $k$ -mer



# Experiments I: E.Coli, varying $k$

- 10M E.Coli reads of 100bp
- 3 versions compared: 1 Bloom (=Chikhi&Rizk), 2 Bloom ( $t=2$ ) and 4 Bloom ( $t=4$ )

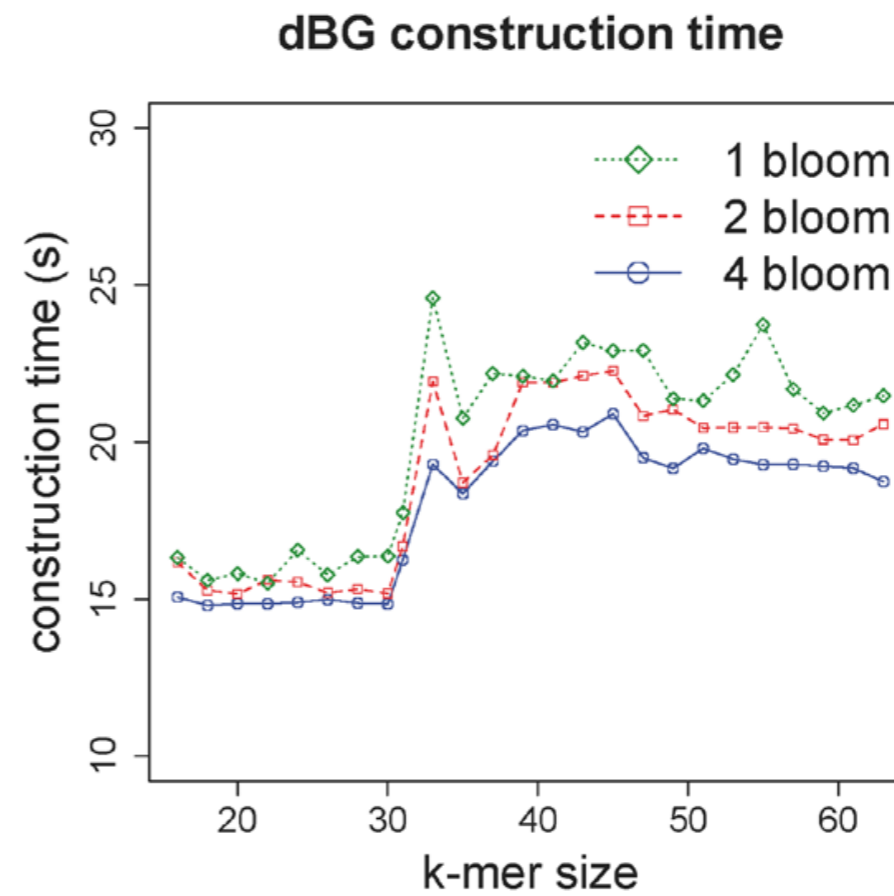
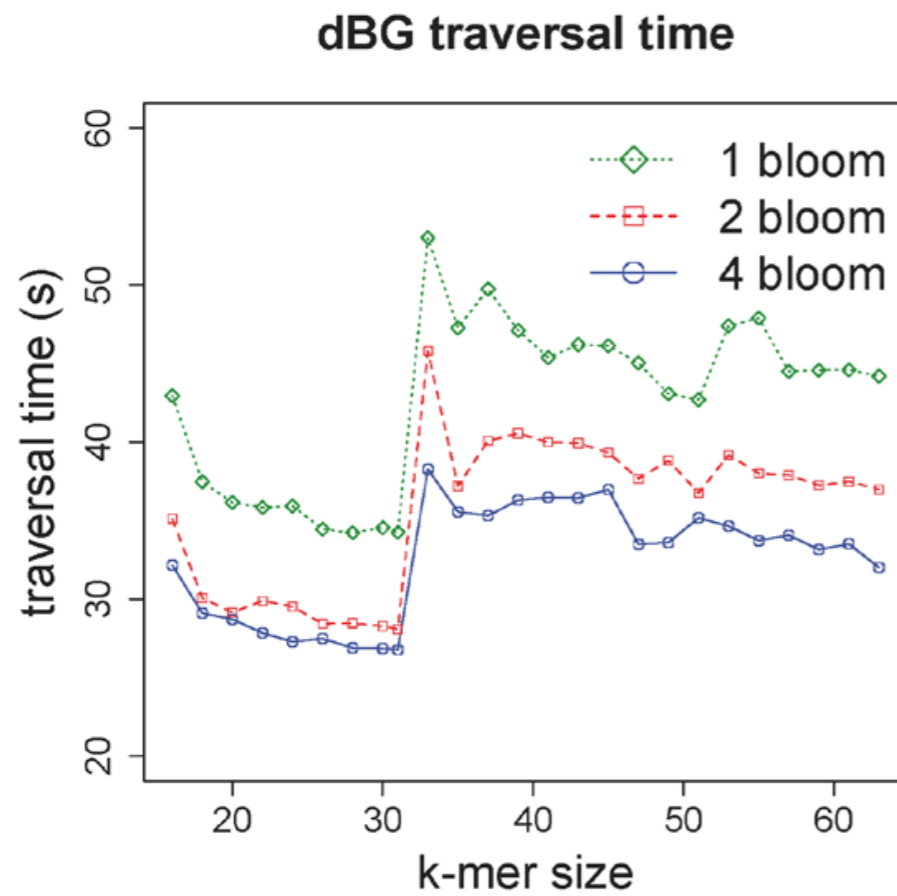


# Experiments II: Human dataset

- 564M Human reads of 100bp (~17X coverage)

Method	1 Bloom	2 Bloom	4 Bloom
Construction time (s)	40160.7	43362.8	44300.7
Traversal time (s)	46596.5	35909.3	34177.2
$r$ (bits)	11.10	8.10	6.56
Bloom filters size (MB)	$B_1 = 3250.95$	$B_1 = 2372.51$ $B_2 = 292.65$	$B_1 = 1921.20$ $B_2 = 496.92$ $B_3 = 83.39$ $B_4 = 21.57$
False positive table size (MB)	$T_1 = 545.94$	$T_2 = 370.96$	$T_4 = 24.07$
Total size (MB)	3796.89	2524.12	2547.15
<b>Size (bits/<math>k</math>-mer)</b>	<b>12.96</b>	<b>10.37</b>	<b>8.70</b>

# Experiments I (cont)



# Efficiently enumerating cFP

---

**Algorithm 1** Constant-memory enumeration of critical false positives

---

- 1: **Input:** The set  $\mathcal{S}$  of all nodes in the graph, the Bloom filter constructed from  $\mathcal{S}$ , the maximum number  $M$  of elements in each partition (determines memory usage)
  - 2: **Output:** The set cFP
  - 3: Store on disk the set  $\mathcal{P}$  of extensions of  $\mathcal{S}$  for which the Bloom filter answers *yes*
  - 4: Free the Bloom filter from memory
  - 5:  $D_0 \leftarrow \mathcal{P}$
  - 6:  $i \leftarrow 0$
  - 7: **while** end of  $\mathcal{S}$  is not reached **do**
  - 8:    $P_i \leftarrow \emptyset$
  - 9:   **while**  $|P_i| < M$  **do**
  - 10:      $P_i \leftarrow P_i \cup \{\text{next } k\text{-mer in } \mathcal{S}\}$
  - 11:   **for** each  $k$ -mer  $m$  in  $D_i$  **do**
  - 12:     **if**  $m \notin P_i$  **then**
  - 13:        $D_{i+1} \leftarrow D_{i+1} \cup \{m\}$
  - 14:   Delete  $D_i, P_i$
  - 15:    $i \leftarrow i + 1$
  - 16: cFP  $\leftarrow D_i$
-

# Bloom filters & De Bruijn Graphs

So, we can use Bloom filters to help efficiently represent De Bruijn Graphs.

Other variants (e.g. counting Bloom filters (Melsted & Pritchard, BMC Bioinformatics, 2011)) allow us to count approximate occurrences of each k-mer, allowing us to sidestep huge storage requirements for k-mers occurring exactly once.

Such an idea is implemented in BFCCounter, and brings us back, full-circle, to the problem of counting k-mers!

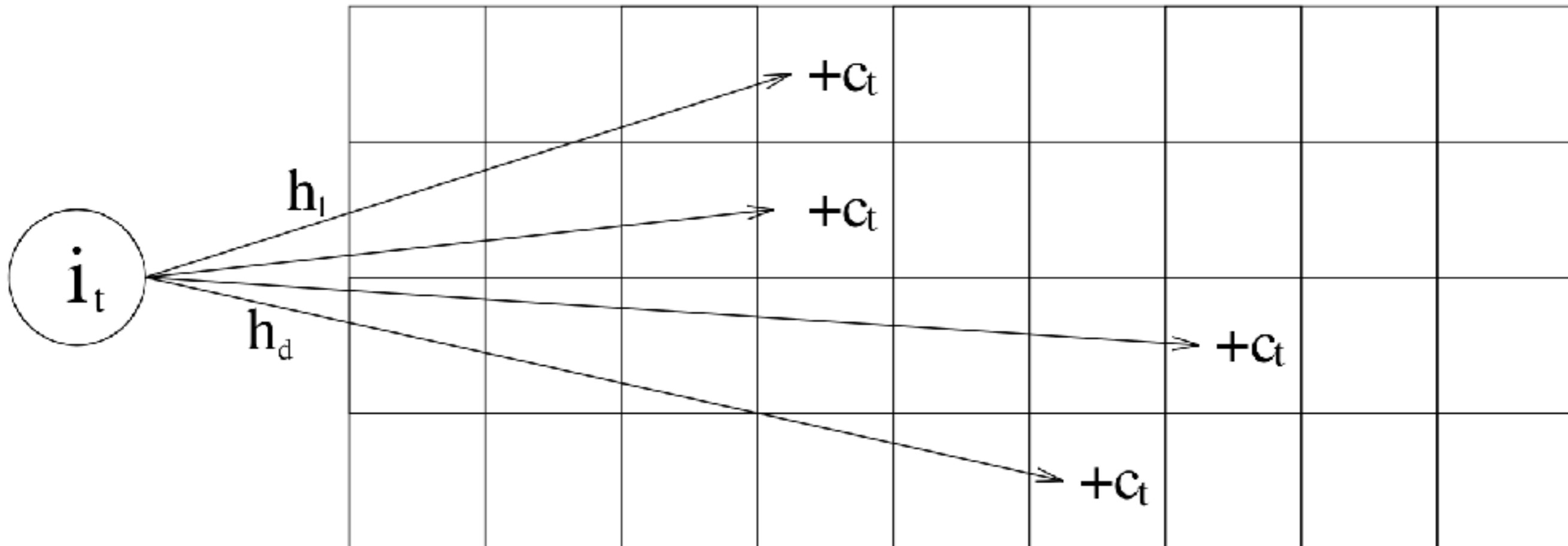


# Probabilistic Data Structures & k-mer Counting

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 with width of the array. This is an  $O(wd)$  data structure.



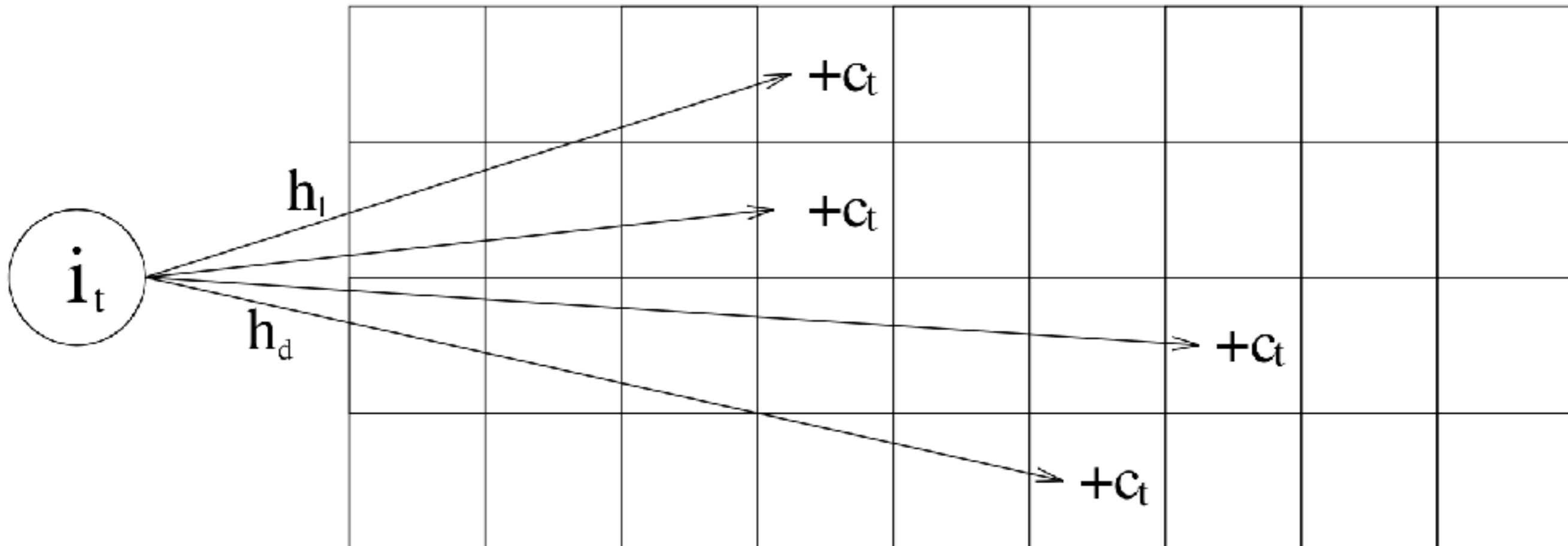
# Probabilistic Data Structures & k-mer Counting

Like Bloom filters, 2 main 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 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 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,

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

base of nat. log

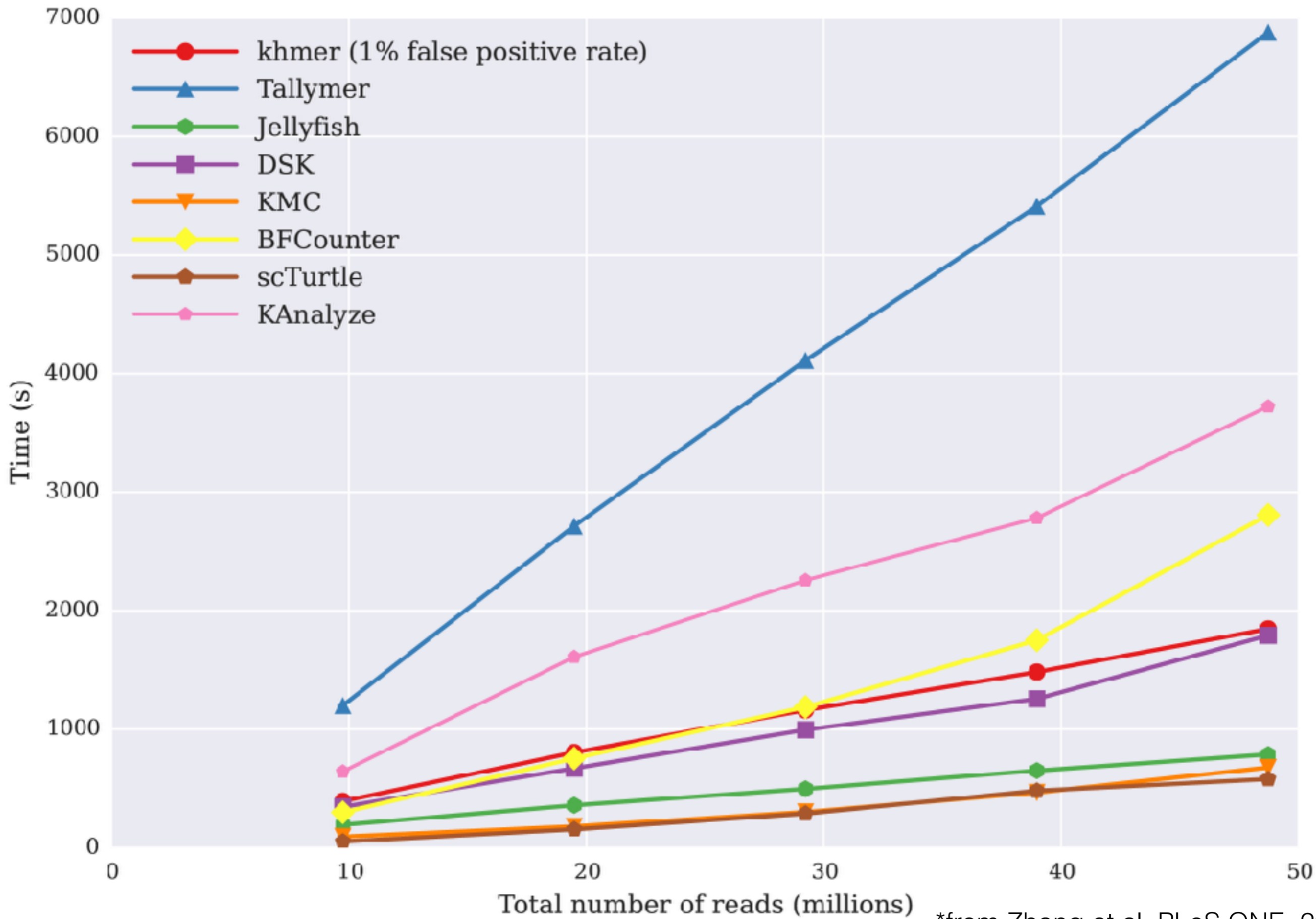
# The Count-Min Sketch for k-mer counting

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

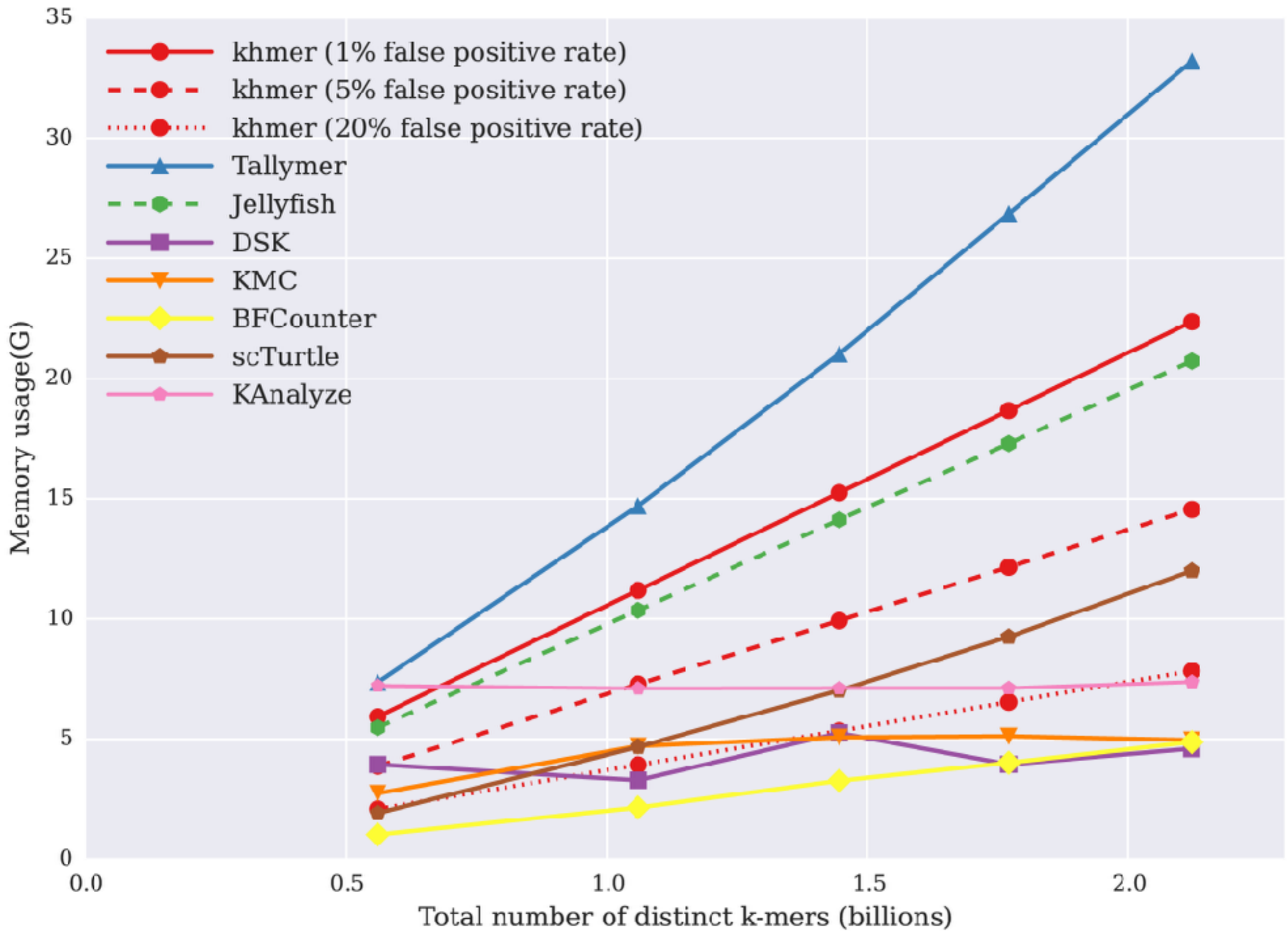
No exact data structure is maintained, just a CMS

This allows for answering approximate count queries efficiently.

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

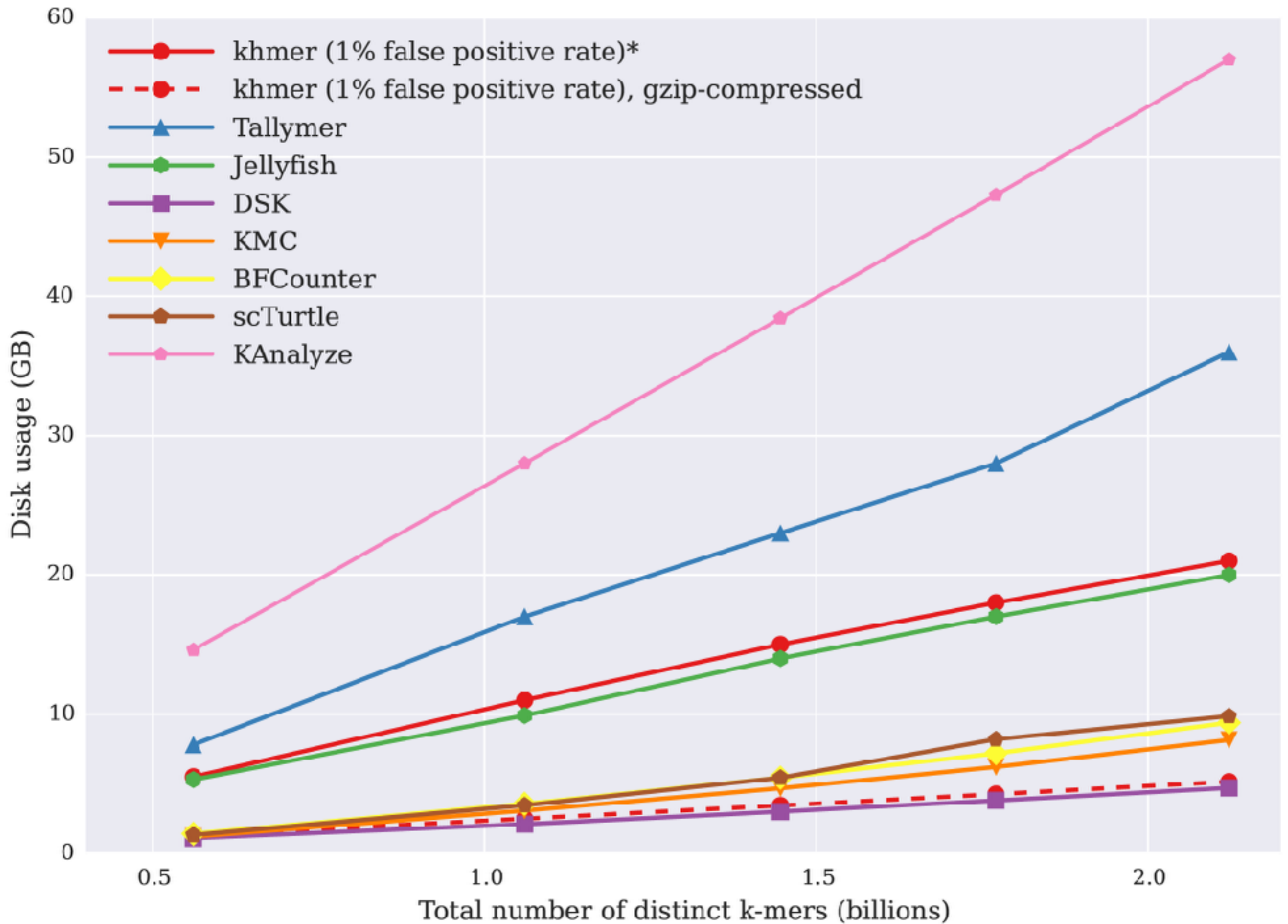


\*from Zhang et al. PLoS ONE, 2014



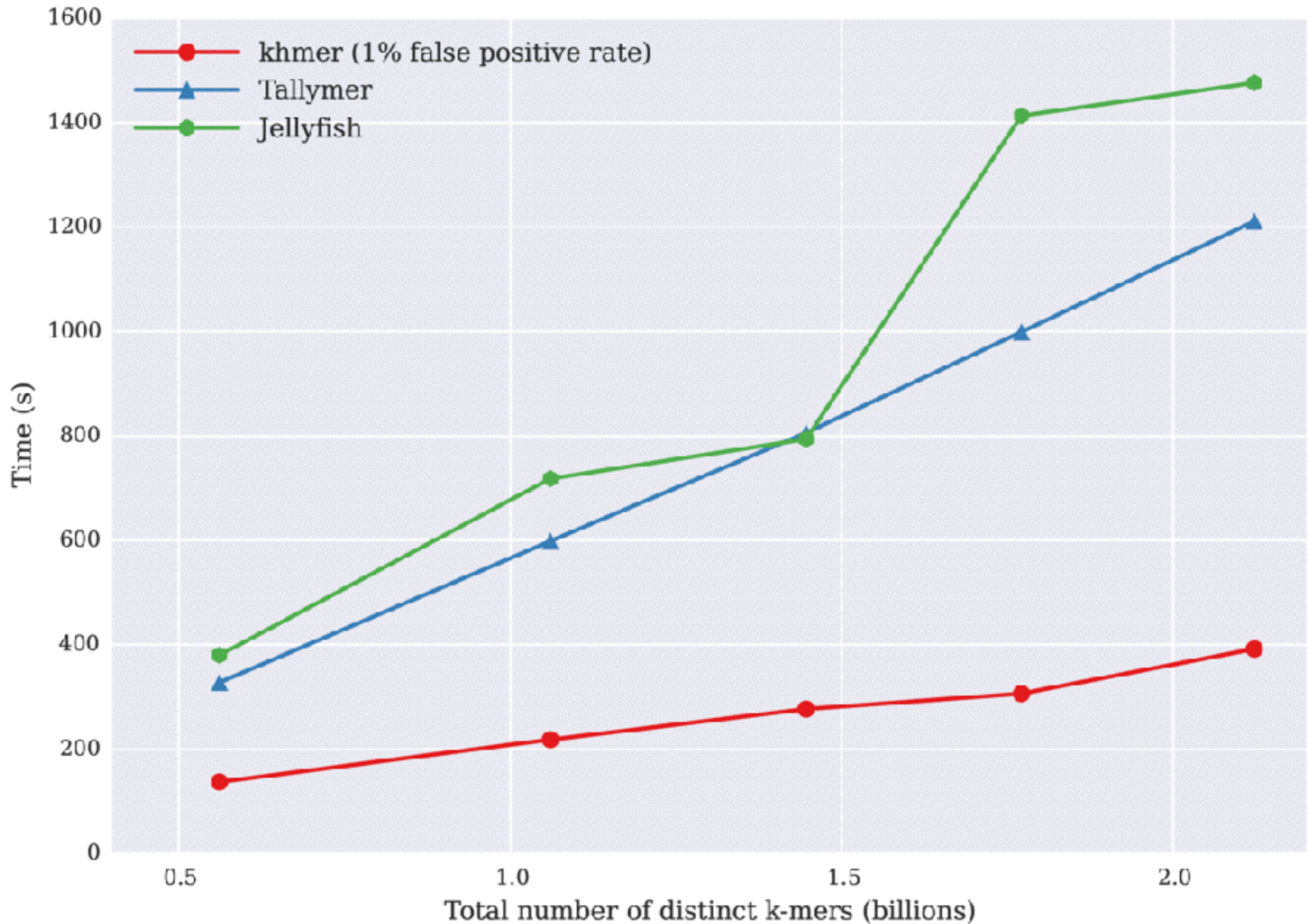
\*from Zhang et al. PLoS ONE, 2014



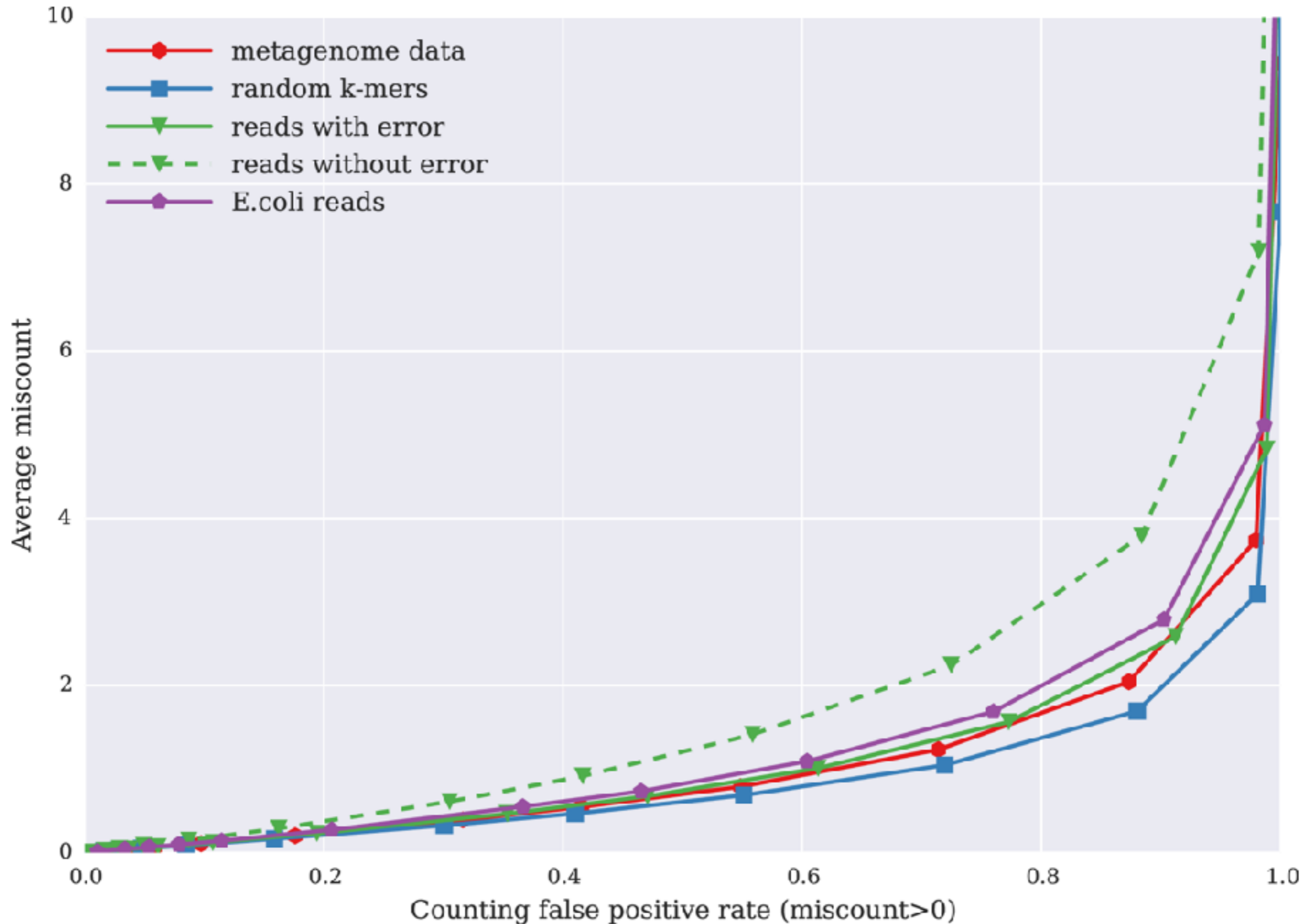


\*from Zhang et al. PLoS ONE, 2014

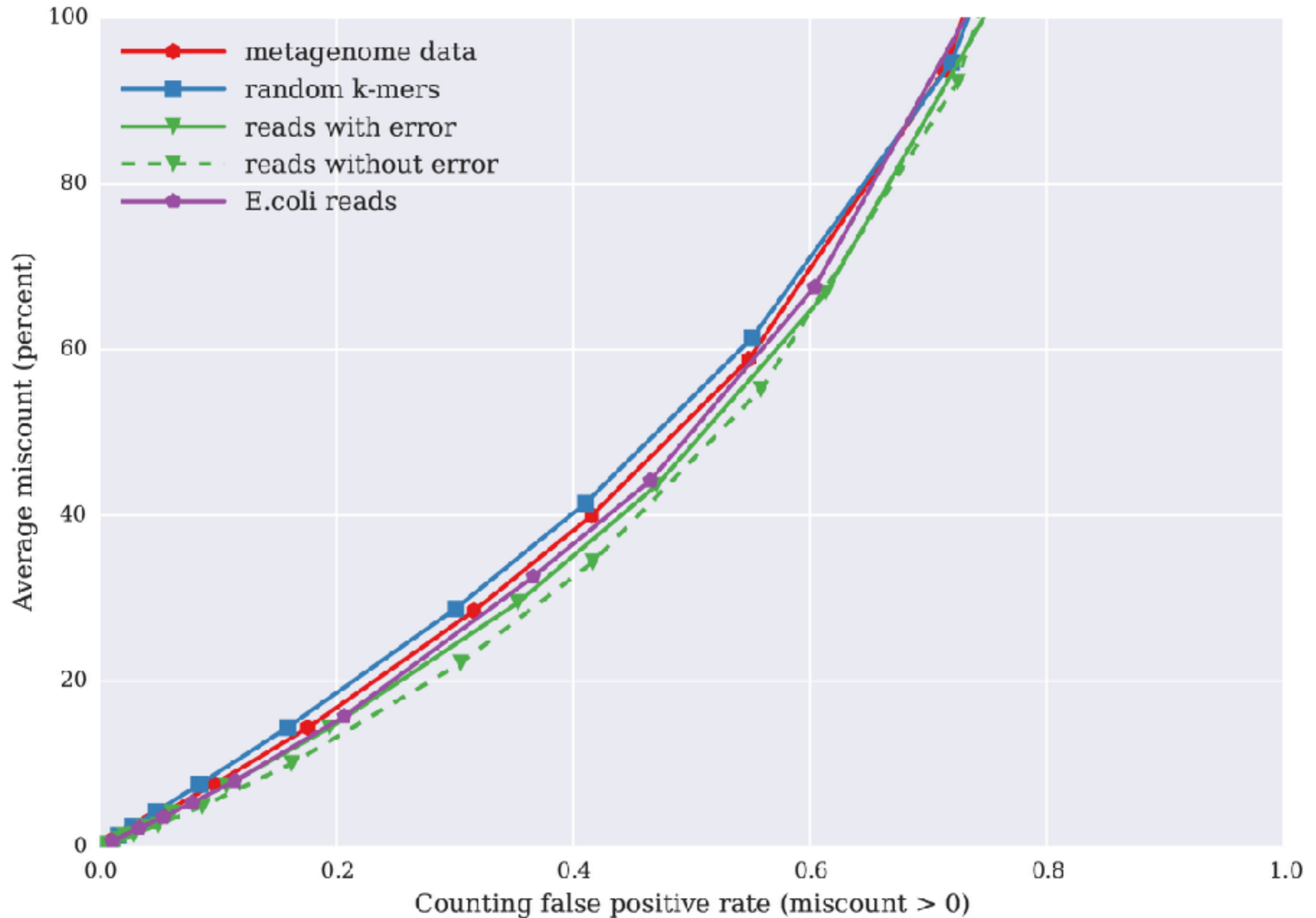
# Querying for random k-mers



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



# Miscount & FP-rate



# 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	41.5 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

# 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 or storing and traversing a De Bruin graph) rife with opportunities for the development and application of interesting and novel data structures and algorithms!