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

$S$

ATACAGGACGTTC

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What’s wrong with this approach?

Speed & Memory usage

Routinely encounter datasets with 10 - 100 x 10⁹ nucleotides

Just hashing the k-mers and resolving collisions takes time

On the order of 1-10 x 10⁹ 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, \textit{lock-free}, k-mer counting

— most parallel accesses \textit{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

— \( f: U_k \to U_k \), 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 $\text{hash}(k) = f(k) \mod M$

— 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

Published: 09 October 2017  Article history ▼
Memory usage of Jellyfish

The graph shows the memory usage (in GB) of different tools with increasing coverage. The tools compared are Meryl 6.1, Tallymer (serial), and Jellyfish. The suffix array-based approach is indicated by the green dashed line.
Runtime of Jellyfish

The graph shows the runtime of different tools as a function of coverage. The tools compared are tallymer, jellyfish, and meryl 5.4 and meryl 6.1. The runtime is measured in hours and minutes, and the coverage is indicated on the x-axis.
System utilization of Jellyfish

![Graph showing system utilization over time and speedup with and without IO.](image-url)
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 $0 < i < k$
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:

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

*analysis of Mitzenmacher and Upfal*
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:

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

*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

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

We can then compute the false positive prob

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

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

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

\[ k = \frac{m}{n} \ln 2 \Rightarrow 2^{-k} \approx 0.6185 \frac{m}{n} \]

We can then compute the false positive prob

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

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

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

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?
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.
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 that 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 $T_1$.

* slide courtesy of Salikhov, Sacomoto & Kucherov
Nodes self-information:

\[ \left\lceil \log_2 \left( \frac{4^3}{7} \right) \right\rceil = 30 \text{ bits} \]

Structure size:

\[ 10 + 3 \cdot 6 = 28 \text{ bits} \]
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 $T_1$)

* slide courtesy of Salikhov, Sacomoto & Kucherov
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-negative positives’ $T_2$, then apply this to $T_2$ etc.
- *cascading Bloom filters*
false positives of $B_1$
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$^2$ positives')

* slide courtesy of Salikhov, Sacomoto & Kucherov
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$^2$ positives')

- Iterate the construction on $T_2$
- We obtain a sequence of sets $T_0, T_1, T_2, T_3, \ldots$ encode by Bloom filters $B_1, B_2, B_3, B_4, \ldots$ respectively

$T_0 \supseteq T_2 \supseteq T_4 \supseteq \ldots, T_1 \supseteq T_3 \supseteq T_5 \supseteq \ldots$

* Slide courtesy of Salikhov, Sacomoto & Kucherov
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.

* slide courtesy of Salikhov, Sacomoto & Kucherov
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

\[
rN + 6rNc^r + rNc^r + 6rNc^{2r} + rNc^{2r} + ... = N(1 + 6c^r) \frac{r}{1 - c^r}
\]

The minimum is achieved for \( r = 5.464 \), which yields the memory consumption of 8.45 bits/node

* slide courtesy of Salikhov, Sacomoto & Kucherov
Infinity difficult to deal with ;)

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

<table>
<thead>
<tr>
<th>$k$</th>
<th>optimal $r$</th>
<th>bits per k-mer</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>5.776737</td>
<td>8.555654</td>
</tr>
<tr>
<td>32</td>
<td>6.048557</td>
<td>8.664086</td>
</tr>
<tr>
<td>64</td>
<td>6.398529</td>
<td>8.824496</td>
</tr>
<tr>
<td>128</td>
<td>6.819496</td>
<td>9.045435</td>
</tr>
</tbody>
</table>

**Table:** Estimations for $t=4$. Optimal $r$ and corresponding memory consumption.
Compared to Chikhi&Rizk’s method

<table>
<thead>
<tr>
<th>$k$</th>
<th>“Optimal” (infinite) Cascading Bloom Filter</th>
<th>Cascading Bloom Filter with $t = 4$</th>
<th>Data structure of Chikhi &amp; Rizk</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8.45</td>
<td>8.555654</td>
<td>12.0785</td>
</tr>
<tr>
<td>32</td>
<td>8.45</td>
<td>8.664086</td>
<td>13.5185</td>
</tr>
<tr>
<td>64</td>
<td>8.45</td>
<td>8.824496</td>
<td>14.9585</td>
</tr>
<tr>
<td>128</td>
<td>8.45</td>
<td>9.045435</td>
<td>16.3985</td>
</tr>
</tbody>
</table>

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

* slide courtesy of Salikhov, Sacomoto & Kucherov
We can cut down a bit more ...

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

* slide courtesy of Salikhov, Sacomoto & Kucherov
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 ($t=4$)

* slide courtesy of Salikhov, Sacomoto & Kucherov
Experiments II: Human dataset

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

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

* slide courtesy of Salikhov, Sacomoto & Kucherov
Experiments I (cont)

* slide courtesy of Salikhov, Sacomoto & Kucherov
Efficiently enumerating cFP

Algorithm 1 Constant-memory enumeration of critical false positives

1: **Input:** The set $S$ of all nodes in the graph, the Bloom filter constructed from $S$, 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 $S$ 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 $S$ is not reached **do**
8: $P_i \leftarrow \emptyset$
9: **while** $|P_i| < M$ **do**
10: $P_i \leftarrow P_i \cup \{\text{next } k\text{-mer in } S\}$
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 BFCounter, 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.
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 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 \(v = \min \{CM[i, h_i(k)] : 0 < i < d\}\)

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:

$$a_i \leq \hat{a}_i$$

(always)

$$\hat{a}_i \leq a_i + \epsilon \|a\|_1$$

(with probability at least $\frac{1}{\delta}$)

where,

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

\[
\begin{align*}
    a_i & \leq \hat{a}_i \\
    \hat{a}_i & \leq a_i + \epsilon \|a\|_1 \\
\end{align*}
\]

(always)

(with probability at least \( \frac{1}{\delta} \))

where,

\[
\begin{align*}
    w &= \left[ \frac{e}{\epsilon} \right], \\
    d &= \left[ \ln\left(\frac{1}{\delta}\right) \right], \\
    \text{and } \|a\|_1 &= \sum_{i=1}^{n} |a_i|
\end{align*}
\]

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 $\delta$

*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:

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

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

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

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*
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

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

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, falamodaresi, bender, mferdman, rob, patro}@cs.stonybrook.edu
2 VMware Research
robj@vmware.com
The Counting Quotient Filter

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

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

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

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

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

Put $r(h(x))$ into $Q[q(h(x))]$
The Counting Quotient Filter

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.](image)

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

<table>
<thead>
<tr>
<th>Filter</th>
<th>Bits per element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom filter</td>
<td>$\frac{\log_2 \frac{1}{\delta}}{\ln 2}$</td>
</tr>
<tr>
<td>Cuckoo filter</td>
<td>$\frac{3+\log_2 \frac{1}{\delta}}{\alpha}$</td>
</tr>
<tr>
<td>Original QF</td>
<td>$\frac{3+\log_2 \frac{1}{\delta}}{\alpha}$</td>
</tr>
<tr>
<td>RSQF</td>
<td>$\frac{2.125+\log_2 \frac{1}{\delta}}{\alpha}$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>CQF</th>
<th>CBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zipfian random inserts per sec</td>
<td>13.43</td>
<td>0.27</td>
</tr>
<tr>
<td>Zipfian successful lookups per sec</td>
<td>19.77</td>
<td>2.15</td>
</tr>
<tr>
<td>Uniform random lookups per sec</td>
<td>43.68</td>
<td>1.93</td>
</tr>
<tr>
<td>Bits per element</td>
<td>11.71</td>
<td>337.584</td>
</tr>
</tbody>
</table>

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

![Graph: Space comparison of CQF, SBF, and CBF](image)

**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

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>File size</th>
<th>#Files</th>
<th># k-mer instances</th>
<th>#Distinct k-mers</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Rvesca</em></td>
<td>3.3</td>
<td>11</td>
<td>4 134 078 256</td>
<td>632 436 468</td>
</tr>
<tr>
<td><em>G.gallus</em></td>
<td>25.0</td>
<td>15</td>
<td>25 337 974 831</td>
<td>2 727 529 829</td>
</tr>
<tr>
<td><em>M.balbisiana</em></td>
<td>46.0</td>
<td>2</td>
<td>41 063 145 194</td>
<td>965 691 662</td>
</tr>
<tr>
<td><em>H.sapiens 1</em></td>
<td>67.0</td>
<td>6</td>
<td>62 837 392 588</td>
<td>6 353 512 803</td>
</tr>
<tr>
<td><em>H.sapiens 2</em></td>
<td>99.0</td>
<td>48</td>
<td>98 892 620 173</td>
<td>6 634 382 141</td>
</tr>
</tbody>
</table>

*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

<table>
<thead>
<tr>
<th>dataset</th>
<th>KMC2</th>
<th>Squeakr</th>
<th>Squeakr-exact</th>
<th>Jellyfish2</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Rvesca</em></td>
<td>8.3</td>
<td>4.8</td>
<td>9.3</td>
<td>8.3</td>
</tr>
<tr>
<td><em>G.gallus</em></td>
<td>32.8</td>
<td>13.0</td>
<td>28.8</td>
<td>31.7</td>
</tr>
<tr>
<td><em>M.balbisiana</em></td>
<td>48.3</td>
<td>11.1</td>
<td>14.2</td>
<td>16.3</td>
</tr>
<tr>
<td><em>H.sapiens 1</em></td>
<td>71.4</td>
<td>22.1</td>
<td>51.5</td>
<td>61.8</td>
</tr>
<tr>
<td><em>H.sapiens 2</em></td>
<td>107.4</td>
<td>30.8</td>
<td>60.1</td>
<td>61.8</td>
</tr>
</tbody>
</table>
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 $k=28$

<table>
<thead>
<tr>
<th>System</th>
<th><em>F. vesca</em></th>
<th></th>
<th><em>G. gallus</em></th>
<th></th>
<th><em>M. balbisiana</em></th>
<th></th>
<th><em>H. sapiens 1</em></th>
<th></th>
<th><em>H. sapiens 2</em></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td></td>
<td>8</td>
<td>16</td>
<td></td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>KMC2</td>
<td>91.68</td>
<td>67.76</td>
<td>412.19</td>
<td>266.546</td>
<td>721.43</td>
<td>607.78</td>
<td>1420.45</td>
<td>848.79</td>
<td>1839.75</td>
<td>1247.71</td>
</tr>
<tr>
<td>Squeakr</td>
<td>116.56</td>
<td>64.44</td>
<td>739.49</td>
<td>412.82</td>
<td>1159.65</td>
<td>662.53</td>
<td>1931.97</td>
<td>1052.73</td>
<td>3275.20</td>
<td>1661.77</td>
</tr>
<tr>
<td>Squeakr-exact</td>
<td>146.56</td>
<td>80.58</td>
<td>966.27</td>
<td>501.77</td>
<td>1417.48</td>
<td>763.88</td>
<td>2928.06</td>
<td>1667.98</td>
<td>5016.46</td>
<td>2529.46</td>
</tr>
<tr>
<td>Jellyfish2</td>
<td>257.13</td>
<td>172.55</td>
<td>1491.25</td>
<td>851.05</td>
<td>1444.16</td>
<td>886.12</td>
<td>4173.3</td>
<td>2272.27</td>
<td>6281.94</td>
<td>3862.82</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>System</th>
<th><em>F. vesca</em></th>
<th></th>
<th><em>G. gallus</em></th>
<th></th>
<th><em>M. balbisiana</em></th>
<th></th>
<th><em>H. sapiens 1</em></th>
<th></th>
<th><em>H. sapiens 2</em></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td></td>
<td>8</td>
<td>16</td>
<td></td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>KMC2</td>
<td>233.74</td>
<td>123.87</td>
<td>979.20</td>
<td>1117.35</td>
<td>1341.01</td>
<td>1376.51</td>
<td>3525.41</td>
<td>2627.82</td>
<td>4409.82</td>
<td>3694.85</td>
</tr>
<tr>
<td>Squeakr</td>
<td>138.32</td>
<td>75.48</td>
<td>790.83</td>
<td>396.36</td>
<td>1188.15</td>
<td>847.83</td>
<td>2135.71</td>
<td>1367.56</td>
<td>3320.67</td>
<td>2162.97</td>
</tr>
<tr>
<td>Jellyfish2</td>
<td>422.220</td>
<td>294.93</td>
<td>1566.79</td>
<td>899.74</td>
<td>2271.33</td>
<td>1189.01</td>
<td>3716.76</td>
<td>2264.70</td>
<td>6214.81</td>
<td>3961.53</td>
</tr>
</tbody>
</table>
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 $k=28$

<table>
<thead>
<tr>
<th>System</th>
<th>G. gallus</th>
<th></th>
<th>M. balbisiana</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Existing</td>
<td>Non-existing</td>
<td>Existing</td>
<td>Non-existing</td>
</tr>
<tr>
<td>KMC2</td>
<td>1495.82</td>
<td>470.14</td>
<td>866.93</td>
<td>443.74</td>
</tr>
<tr>
<td>Squeakr</td>
<td>303.68</td>
<td>52.45</td>
<td>269.24</td>
<td>40.73</td>
</tr>
<tr>
<td>Squeakr-exact</td>
<td>389.58</td>
<td>58.46</td>
<td>280.54</td>
<td>42.67</td>
</tr>
<tr>
<td>Jellyfish2</td>
<td>884.17</td>
<td>978.57</td>
<td>890.57</td>
<td>985.30</td>
</tr>
</tbody>
</table>

**Table 6.** de Bruijn graph query performance on different datasets

<table>
<thead>
<tr>
<th>System</th>
<th>Dataset</th>
<th>Max path len</th>
<th>Running times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Counting</td>
</tr>
<tr>
<td>KMC2</td>
<td>G. gallus</td>
<td>122</td>
<td>266</td>
</tr>
<tr>
<td>Squeakr</td>
<td>G. gallus</td>
<td>92</td>
<td>412</td>
</tr>
<tr>
<td>KMC2</td>
<td>M. balbisiana</td>
<td>123</td>
<td>607</td>
</tr>
<tr>
<td>Squeakr</td>
<td>M. balbisiana</td>
<td>123</td>
<td>662</td>
</tr>
</tbody>
</table>

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