Rank-Indexed Hashing: 
A Compact Construction of Bloom Filter and its Variants*

Nan Hua¹, Haiquan (Chuck) Zhao¹, 
Bill Lin², Jun (Jim) Xu¹

¹College of Computing, Georgia Tech
²Dept of ECE, UCSD
* Supported in part by CNS-0519745, CNS-0626979, CNS-0716423, CAREER Award ANI-0238315, and a gift from CISCO
Index

- What is Bloom Filter?
- Alternative Construction Way: Fingerprint Hash Table
- Scheme of Rank-Indexed Hashing
- Tail Bound Guarantee
- Numerical Results
- Conclusion
Bloom Filter (standard)

- A space-efficient data-structure to approximately represent a set and answer membership query
- $k$ hash functions and an m-bit-long vector
- Insert element by setting the bits in $k$ hashed locations to 1
- Answer membership query by checking the bits in the $k$ hashed locations
Bloom Filter (Standard) (II)

- In one word, only approximately answers “in the set” or not, nothing else.
  - No False Negative, while False Positive is possible and predictable
  - Can’t support deletion, since once bit is set, never reset
- Tradeoff of false positive rate $p$, hash functions $k$, and storage per inserted item
  - The best tradeoff for storage is achieved when $k=\lg(1/p)=\ln2/n$ ($p$ is false positive rate)
  - $1.44\lg(1/p)$ bits per item ($1.44=1/\ln2; \lg$ means $\log_2$)
Variants on Bloom Filters (I)

- Support Deletion:
  - Counting Bloom Filter (CBF)

- Support Query of Associated Values
  - Associated value: such as counts, state, address, etc.
  - Spectral Bloom Filter (SBF)
  - Bloomier Filter
  - Approximate Concurrent State Machines (ACSM) (based on d-left Counting Bloom Filter, dlCBF)
Variants on Bloom Filters (II)

- **Compressed Bloom Filter**
  - optimized for best compressed size
  - The optimal point of standard Bloom Filter is also the worst point of compressed Bloom Filter:
    - $1.44 \log(1/p)$ bits per item
  - The limit is of compressed Bloom Filter
    - $\log(1/p)$ bits per item (also the information theory limit)
  - Compressed Bloom Filter trade original (uncompressed) size for compressed size.

- [Graph showing false positive rate against hash functions]
Application of Bloom Filter and Variants

- Although discovered in database area, widely used in networking
  - IP Lookup & Packet Classification
  - Hash table acceleration
  - Pattern Matching & Deep Packet Inspection
  - P2P File Sharing
  - Routing Protocols Design
  - ...

- Design consideration and optimization targets
  - Space (bloom filters are always small and on-chip-able)
  - Query speed (HW/SW? Need deterministic query time bound or flexible?)
  - Update speed (static, quasi-static or fast)
  - Support of extra functions
An alternative way to build “Bloom Filter”: Fingerprint Hash Table (I)

- The weakness of standard Bloom Filter:
  - the best storage of Bloom Filter is $1.44 \lg(1/p)$ bits per element, still far from the limit, $\lg(1/p)$.
  - Not easy to support deletions and associated values

- We could revisit “Bloom filter” problem, i.e. approx. representation of a set, by another way: fingerprint hash table
  - Make fingerprint for each element (by hash) and store it in a hash table
An alternative way to build Bloom Filter: Fingerprint Hash Table (II)

- For each incoming element \( x \), get location \( d(x) \) and \( r \)-bit-long fingerprint \( r(x) ([0,2^r-1]) \) by hash function.
- Store the \( r \)-bit-long fingerprint \( r(x) \) in the \( d(x) \)th hash chain.
- To test whether \( x \) is in the set, simply check all fingerprints in the \( d(x) \)th hash chain to find \( r(x) \).
- Naturally support deletion and query of associated values.

\[ \text{m Hash Chains} \]

\[ r \text{ bits fingerprint per element} \]
An alternative way to build Bloom Filter: Fingerprint Hash Table (III)

- Suppose $n$ items inserted, false positive rate $p$ would be only $(n/m)^{2^r}$
- When $n=m$, if only counting the bits for fingerprint, only $\lg(1/p)$ bits per inserted item is needed, which is the lower bound by information theory!
- We haven’t included the chaining cost…

$\begin{array}{c}
\text{m Hash Chains} \\
\vdots \quad \vdots \quad \vdots \\
\vdots \quad \vdots \quad \vdots \\
\vdots \quad \vdots \quad \vdots \\
\vdots \quad \vdots \quad \vdots \\
\end{array}$

$r$ bits fingerprint per element
An alternative way to build Bloom Filter: Fingerprint Hash Table (III)

- However, the indexing cost for chaining is very high, if using conventional hash table techniques
  - Suppose using pointers, at least another lg(n) bits per item are needed.
  - Memory organization is hard
  - Some well-known hash table constructions are unsuitable here, such as Linear Probe (can’t know which chain is in during query, while the information of d(x) is also important for fingerprint hash table)
- In nature, dl-CBF is using multiple-choices hash table to lower the chaining cost
- We use Rank-Indexing technique to lower chaining cost
Our approach– Rank Indexing

From a horizontal view, hash chain is very fluctuating. However, we could pack them together from a vertical view. Then, how to link??!!
Rank Indexing (II)

The link is established by “rank indexing”

- Named by “Dictionaries using variable-length keys and data, with applications” [SODA05]
- Also called bitmap technique in [Varghese04]
Rank Indexing (III)

- How Rank Indexing works?
  - The location of the linked element is calculated by “rank” operation
  - \( \text{rank}(A, b) \) “popcount” the bits in \( A[1…b] \)

\[
\begin{align*}
\text{rank}(I_0, 7) &= 3 \\
&= \text{(3 bits in } I_0[1…7] \text{ are set)}
\end{align*}
\]

\[
\begin{align*}
\text{rank}(I_1, 3) &= 2 \\
\text{rank}(I_2, 2) &= 1
\end{align*}
\]
• We could even pack these $I_1, I_2, I_3...$ together, since the size of $I_1, I_2, I_3$, etc, are calculable.
Rank Indexing (V)

- To insert one fingerprint,
  - Bit shift of the bitmap index
  - Bit shift of the fingerprints

To insert one fingerprint, we perform the following steps:

1. **Base Index**
   - Initialize the base index with the bitmap index.

2. **Higher Index**
   - Shift the base index to the left by the number of bits in the higher index.

3. **Fingerprints**
   - Shift the fingerprints to the left by the number of bits in the higher index.

4. **Insertion**
   - Insert the fingerprint into the corresponding chain.

5. **Popcount Check**
   - Check the popcount of the shifted index and fingerprints.

For example, if we have a fingerprint `0101110`, and the shifted indexes are `I_0 = popcnt(I_0) = 4` and `I_1 = popcnt(I_1) = 2`, we insert the fingerprint into the 4th chain.
Bucket is needed

- We can’t directly use rank indexing on the whole fingerprint hash table
  - Too large bitmap
- We could group hash chains into buckets and only use rank indexing in the “local” bucket.
Bucket Overflow

- There is some probability that bucket might overflow!
  - X: fingerprints inserted into bucket
  - Z: the number of entries pre-allocated in each bucket

<table>
<thead>
<tr>
<th>Threshold Z</th>
<th>Overflow (X&gt;Z) Probability (When E[X]=56)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z=64</td>
<td>0.13</td>
</tr>
<tr>
<td>Z=74</td>
<td>0.01</td>
</tr>
<tr>
<td>Z=128</td>
<td>8e-17</td>
</tr>
</tbody>
</table>
Handle Bucket Overflow

The number of 2nd-level and 3rd-level bucket extensions are defined as $J_2$ and $J_3$. 
Tail Bound (I)

- Random Variable $X_{i(n)}$ denotes the number of fingerprints inserted in $i^{th}$ bucket (when $n$ fingerprints inserted in total)

- The probability that 2nd-level bucket extensions are insufficient is

$$\text{Pr}[O_2 > J_2], \text{ where } O_2 \equiv \sum_{i=1}^{B} 1_{\{X_{i(n)} > Z_i\}}$$

- The probability $P_o$ that the bucket extensions are insufficient is bound by:

$$P_o \leq \sum_{l=2}^{4} \text{Pr}[O_l > J_l], \text{ where } J_4 \equiv 0, O_l = \sum_{i=1}^{B} 1_{\{X_{i(n)} > Z_{l-1}\}}$$
Tail Bound (II)

- The hardest point of the proof: Random Variables $X_{i}^{(n)}$, i=1,…,B, are weakly correlated.
- We could construct Random Variables $Y_{i}^{(n)}$, i=1,…,B, which is i.i.d random variables with distribution $\text{Poisson}(n/m)$.
- From [MM05], it could be proved that:
  \[ E[f(X_{1}^{(n)},...,X_{m}^{(n)})] \leq 2E[f(Y_{1}^{(n)},...,Y_{m}^{(n)})] \]

where $f$ is an nonnegative and increasing function.

- Then, we could use the following increasing indicator function to get bound for $\Pr[O>J]$
  \[ f(x_{1},...,x_{B}) = 1 \left\{ \sum_{i=1}^{B}1_{x_{i}>W} > J \right\} \]

- We could get: $\Pr[O > J] \leq 2\text{Binotail}(B, \text{Poisson}(n/B, Z), J)$
Numerical Results (I)

----Parameter Configuration

- Typical Configuration of Parameters

**Representative results for rank-indexed hashing under various parameter settings ($P_0 = 10^{-10}, n = 10^5$).**

<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>$\lambda$</th>
<th>$R$</th>
<th>$L$</th>
<th>$Z_1$</th>
<th>$Z_2$</th>
<th>$Z_3$</th>
<th>$J_2/B$</th>
<th>$J_3/B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>0.64</td>
<td>6 bits</td>
<td>60</td>
<td>45</td>
<td>8</td>
<td>45</td>
<td>17.9%</td>
<td>2.7%</td>
</tr>
<tr>
<td>0.1%</td>
<td>0.92</td>
<td>10 bits</td>
<td>64</td>
<td>63</td>
<td>17</td>
<td>50</td>
<td>36.0%</td>
<td>1.7%</td>
</tr>
<tr>
<td>0.01%</td>
<td>0.86</td>
<td>13 bits</td>
<td>61</td>
<td>59</td>
<td>13</td>
<td>48</td>
<td>23.3%</td>
<td>1.8%</td>
</tr>
</tbody>
</table>
Numerical Results (II)

--- Storage

- Compared with standard Bloom Filter

<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>standard</th>
<th>$d$-left</th>
<th>Rank</th>
<th>Comparison vs. standard</th>
<th>Comparison vs. $d$-left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>9.6</td>
<td>15.0</td>
<td>10.6</td>
<td>+10.1%</td>
<td>-29.5%</td>
</tr>
<tr>
<td>0.1%</td>
<td>14.6</td>
<td>18.0</td>
<td>14.4</td>
<td>-1.1%</td>
<td>-26.3%</td>
</tr>
<tr>
<td>0.01%</td>
<td>19.1</td>
<td>22.2</td>
<td>18.2</td>
<td>-4.4%</td>
<td>-23.2%</td>
</tr>
</tbody>
</table>

- Compared with Counting Bloom Filter

<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>standard CBF</th>
<th>$d$-left CBF</th>
<th>Rank CBF</th>
<th>Comparison vs. standard</th>
<th>Comparison vs. $d$-left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>38.3</td>
<td>17.6</td>
<td>13.0</td>
<td>-66%</td>
<td>-27%</td>
</tr>
<tr>
<td>0.1%</td>
<td>58.4</td>
<td>22.3</td>
<td>16.8</td>
<td>-71%</td>
<td>-24%</td>
</tr>
<tr>
<td>0.01%</td>
<td>76.3</td>
<td>26.4</td>
<td>20.6</td>
<td>-73%</td>
<td>-22%</td>
</tr>
</tbody>
</table>
Numerical Results (III) ----for transferring Purpose

- When used for transferring (P2P sharing, etc), no need to transfer the pre-allocated vacant entries
- The bits per item would be only $1 + m/n + \lg(1/p)$

<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>Scheme</th>
<th>Packed Size</th>
<th>Uncompressed Compression BF(^1)</th>
<th>Compression ratio(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>Rank</td>
<td>8.6</td>
<td>30</td>
<td>89%</td>
</tr>
<tr>
<td>1%</td>
<td>$d$-left</td>
<td>11.7</td>
<td>$-$ $^3$</td>
<td>122%</td>
</tr>
<tr>
<td>0.1%</td>
<td>Rank</td>
<td>12.1</td>
<td>150</td>
<td>83%</td>
</tr>
<tr>
<td>0.1%</td>
<td>$d$-left</td>
<td>15.2</td>
<td>$-$ $^3$</td>
<td>104%</td>
</tr>
<tr>
<td>0.01%</td>
<td>Rank</td>
<td>15.2</td>
<td>$1.3 \times 10^3$</td>
<td>79%</td>
</tr>
<tr>
<td>0.01%</td>
<td>$d$-left</td>
<td>18.3</td>
<td>27</td>
<td>96%</td>
</tr>
</tbody>
</table>

\(^1\)To achieve the same packed size, how much big original array size is needed by a Compressed Bloom Filter
(Note that Compressed Bloom Filter trade original size for compressed size)
Conclusion

- A compact construction of Bloom Filters and Variants
  - Based on popcnt operation, which is supported by modern processors
  - Very Memory-Efficient
  - Sharing all advantages of fingerprint hash table approaches
    - Support deletions, support query of associated value
  - Also great to substitute Compressed Bloom Filter

- Disadvantages
  - The access time is not deterministic (due to the chaining), although the expected access time is very good

- Related work that have been done
  - Combination with d-left Counting Bloom Filter (Rank-Indexed d-left)
    - the storage performance is even better
    - A little more memory accesses
  - Using similar techniques in statistical counters (ANCS’08)
Q&A

- nanhua@gatech.edu
- Thanks!