Bloom filters

Approximate membership data structures
Bloom filters: generalities

- Bloom (1970)
- generalizes the bitmap representation of sets
- *approximate membership data structure*: supports INSERT and LOOKUP
- LOOKUP only checks for the presence, no satellite data
- produces false positives (with low probability)
- cannot iterate over the elements of the set
- DELETE is not supported (in the basic variant)
- very space efficient, keys themselves are not stored
- *Example*: forbidden passwords
Bloom filter: how it works

- $U$: universe of possible elements
- $K$: subset of elements, $|K| = n$
- $m$: size of allocated bit array

- Define $d$ hash functions $h_1, \ldots, h_d: U \rightarrow \{0, \ldots, m - 1\}$

- INSERT($k$): set $h_i(k) = 1$ for all $i$
- LOOKUP($k$): check $h_i(k) = 1$ for all $i$

- False positives but no false negatives
Bloom filters: analysis

- $P[\text{specific bit of filter is 0}] = (1 - 1/m)^{dn} \approx e^{-dn/m} \equiv p$

- $P[\text{false positive}] = (1 - p)^d = (1 - e^{-dn/m})^d$

- Optimal number $d$ of hash functions: $d = \ln 2 \cdot \frac{m}{n} \approx 0.693 \cdot \frac{m}{n}$

- Therefore, for the optimal number of hash functions,

$$P[\text{false positive}] = 2^{-\ln 2 \cdot \frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

- E.g. with 10 bits per element, $P[\text{false positive}]$ is less than 1%

- To insure the FP rate $\varepsilon$: $m = \log_2 e \cdot n \cdot \log_2 \frac{1}{\varepsilon} \approx 1.44 \cdot n \cdot \log_2 \frac{1}{\varepsilon}$
Dependence on the nb of hash functs

Opt $d = 8 \ln 2 = 5.45 \ldots$

$m/n = 8$

$n$ elements
$m$ bits
$d$ hash functions
Lower bound on the size of approximate membership data structures (AMD)

- Bloom filter takes $1.44 \cdot \log \frac{1}{\varepsilon}$ bits per key, is this optimal?

- How many AMDs are there to store all sets of size $n$ drawn from universe $U$ with FPP $\varepsilon$?
- Each AMD specifies a set of size $\varepsilon |U|$ (assuming $|U|$ large) containing a set of size $n$
- Any set of size $n$ should be covered, and the number of such sets is $\geq \left( \frac{|U|}{n} \right) / \left( \frac{\varepsilon |U|}{n} \right) \approx \left( \frac{1}{\varepsilon} \right)^n$ (cf Erdős & Spencer 74, Rödl 85)
- $\Rightarrow$ each FPP must take $\geq n \cdot \log \frac{1}{\varepsilon}$ bits
Bloom filter: properties/operations

- For the optimal number of hash function, about a half of the bits is 1 \([immedate\ from\ the\ formula]\)
- The Bloom filter for the union is the OR of the Bloom filters
- Is similar true for the intersection? \([explain]\)
- If a Bloom filter is sparse, it is easy to halve its size
Bloom filters: applications

- Bloom filters are very easy to implement
- Used e.g. for
  - spell-checkers (in early UNIX-systems)
  - unsuitable passwords, "approximate" unsuitable passwords (Manber&Wu 1994)
  - online applications (traffic monitoring, …)
  - distributed databases
  - malicious sites in Google Chrome
  - read articles in publishing systems (Medium)
  - Google Bigtable, Apache HBase, Bitcoin, bioinformatics, …
- Sometimes (when the set of possible queries is limited) it is possible to store the set of false positives in a separate data structure
Cuckoo filters

filters via Cuckoo hashing
Filters via MPHF

- Given a set $K \subseteq U$, build an MPHF $h: K \rightarrow [1..n]$
- Given $\varepsilon$, pick a hash function $f$ mapping keys of $K$ into fingerprints of $\log \frac{1}{\varepsilon}$ bits
- Build an array $F$ of fingerprints: $F[h(k)] = f(k)$

$$P[f(x) = f(y)] = \frac{1}{2^{\log \frac{1}{\varepsilon}}} = \varepsilon \text{ (false positive proba)}$$

- Space: $n \cdot \log \frac{1}{\varepsilon} + \text{<size of MPFR>}$
- lower bound: size of MPFR $\geq 1.44n$
- $K$ must be static, does not support insertions/deletions
Cuckoo filter: ideas

- Use Cuckoo hash table (e.g. (2,4)-table) instead of MPHF

- **Problem**: How to move a fingerprint? i.e. how to know its alternative bucket?
Cuckoo filter: ideas

- Use Cuckoo hash table (e.g. (2,4)-table) instead of MPHF

- **Problem**: How to move a fingerprint? i.e. how to know its alternative bucket?

  \[
  h_1: K \rightarrow 2^{\log |T|}, \quad h_2: 2^{\log \frac{1}{\varepsilon}} \rightarrow 2^{\log |T|}
  \]

  location 1: \( h_1(k) \)

  location 2: \( h_1(k) \oplus h_2(f(k)) \)

- Alternative location of a fingerprint \( \alpha \) at location \( i \) is \( i \oplus h_2(\alpha) \)
Remarks

- Two locations of a key are not fully independent. E.g. two keys sharing the same bucket and the same fingerprint have the same alternative location. (⇒ store multisets in $b$-element buckets)

- *Practical*: Cuckoo vs. Bloom: for small false positive rate ($< 3\%$) and $b = 4$, Cuckoo filter achieves the same performance as Bloom with smaller space

![Figure 4: False positive rate vs. space cost per element. For low false positive rates ($< 3\%$), cuckoo filters require fewer bits per element than the space-optimized Bloom filters. The load factors to calculate space cost of cuckoo filters are obtained empirically.](image)

[Fan et al. Cuckoo filter: practically better than Bloom, CoNEXT 2014]
Count-Min sketch (aka Spectral Bloom filter)

Storing count information
How to support deletions in Bloom filters?
How to support deletions in Bloom filters?

- **Counting Bloom filter**: Bloom filter that, instead of 0 and 1, stores (small) counters
  
  - **INSERT**($k$): $B[h_i(k)] \leftarrow B[h_i(k)] + 1$ for all $i$
  - **LOOKUP**($k$): check $B[h_i(k)] > 0$ for all $i$
  - **DELETE**($k$): $B[h_i(k)] \leftarrow B[h_i(k)] - 1$ for all $i$

- Also works for multi-sets
- Analysis shows that
  
  $$P \left[ \text{max counter} \geq 16 \right] < 1.37m \cdot 10^{-15}$$

  i.e. 4 bits/counter suffices for practical purposes

  [Fan et al. IEEE/ACM Trans. on Networking, 2000]
Count-Min sketch

- What if we want to estimate the multiplicities (number of occurrences) of elements of a multi-set stored in a counting Bloom filter?
- Streaming framework
- Example: $m = 8$, $C[0..7]$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$h_2$</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

b a d a e f c a ...

#a? #e? #f? #c? #e? #f?
Count-Min: operations

- **UPDATE**$(k)$: $C[h_i(k)] \leftarrow C[h_i(k)] + 1$ for all $i$
- $\hat{f}(k) = \min_i h_i(k)$
Count-Min sketch: analysis

- **Theorem**: if \( d = \log_2 \frac{1}{\delta} \) and \( m = \frac{ed}{\epsilon} \), then

\[
P[\hat{f}(k) \geq f(k) + \epsilon n] \leq \delta,
\]

where \( d \) is the number of hash functions, \( f(k) \) is the true count of \( k \) and \( n \) is the total number of elements in the stream.

**Proof**: \( C[h_i(k)] = f(k) + X_i(k) \)

\[
E[X_i(k)] = \frac{1}{m} \sum_{l \neq k} f(l) + \sum_{j \neq i} \frac{1}{m} \sum_k f(k) \leq \frac{d}{m} n = \frac{\epsilon}{e} n
\]

\[
P[X_i(k) > \epsilon n] < \frac{1}{e}
\]

Since \( \hat{f}(k) = f(k) + \min_i X_i(k) \), we have

\[
P[\hat{f}(k) - f(k) \geq \epsilon n] \leq e^{-d} = \delta
\]
Count-Min: properties

- Total space $m = \frac{e}{\varepsilon} \log \frac{1}{\delta}$

- **Example (“Heavy Hitters”):** Assume we want to output all elements that occur $n/50$ of times. Set $\varepsilon = 1/100$, i.e. $m = 271 \cdot \log \frac{1}{\delta}$. Then we will output all desired elements, but also some elements occurring less, but not less than $n/100$, with $p = 1 - \delta$.

- Bound in Theorem is in terms of $n$

- CM-sketch also applies to increments $> 1$

- Decrements (“deletions”) are also supported provided that counters remain non-negatives

What if \( n \) is not known in advance?

- **Idea**: Maintain a min-heap of current frequent items, update after each element
- After processing each element \( x \), estimate \( \hat{f}(x) \)
- If \( \hat{f}(x) \geq \varepsilon n \) (\( n \) current stream size), insert \( x \) to the heap with value \( \hat{f}(x) \) (or update if it was already there)
- If the smallest value of the heap (computed in \( O(1) \)) is \( < \varepsilon m \), delete it from the heap
- At the end, output all elements of the heap