Bloom Filters and its Variants

Deke Guo
guodeke@gmail.com

National University of Defense Technology
2012.01.13
Outline

1. Standard Bloom Filters
2. Compressed Bloom Filters
3. Counting Bloom Filters
4. Representation of a set of (key, f(key))
5. Invertible Bloom Filters
The main point

- Whenever you have a set or list, and space is an issue, a Bloom filter may be a useful alternative.
The Problem Solved by BF: Approximate Set Membership

- Given a set $S = \{x_1, x_2, \ldots, x_n\}$, construct data structure to answer queries of the form “Is $y$ in $S$?”

- Data structure should be:
  - Fast (Faster than searching through $S$).
  - Small (Smaller than explicit representation).

- To obtain speed and size improvements, allow some probability of error.
  - False positives: $y \not\in S$ but we report $y \in S$
  - False negatives: $y \in S$ but we report $y \not\in S$
1. Standard Bloom Filters
Set representation

A hash function family

Data set B

A bit vector

0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0

Add ‘a’
1. Standard Bloom Filters

Set representation

A hash function family

A bit vector

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
\ldots & \ldots & \ldots & \ldots \\
1 & 1 & 0 & 0 \\
\end{array}
\]
1. Standard Bloom Filters

   Membership query

- “Constant” time (time to hash).
- Small amount of space.
- But with some probability of being wrong.
A hash function family

Data set A

Data set B

A bit vector

A false positive

query
False positive probability

- **Assumption:** We have good hash functions, look random.

- Given $m$ bits for filter and $n$ elements, choose number $k$ of hash functions to minimize false positives:
  - Let $p = \Pr[\text{cell is empty}] = (1 - 1/m)^{kn} \approx e^{-kn/m}$
  - Let $f = \Pr[\text{false pos}] = (1 - p)^k \approx (1 - e^{-kn/m})^k$

- As $k$ increases, more chances to find a 0, but more 1’s in the array.

- Find optimal at $k = (\ln 2)m/n$ by calculus.

$$f = \Pr[\text{false pos}] = 0.61285^{m/n}$$
Hash functions

False positive rate

Opt $k = 8 \ln 2 = 5.45...$

$m/n = 8$
Alternative Approach for Bloom Filters

- Folklore Bloom filter construction.
  - Recall: Given a set $S = \{x_1, x_2, x_3, \ldots, x_n\}$ on a universe $U$, want to answer membership queries.
  - Method: Find an $n$-cell *perfect hash function* for $S$.
    - Maps set of $n$ elements to $n$ cells in a 1-1 manner.
    - Then keep $\lceil \log_2(1/\varepsilon) \rceil$ bit fingerprint of item in each cell. Lookups have false positive $\leq \varepsilon$.
    - Advantage: each bit/item reduces false positives by a factor of $1/2$, vs $\ln 2$ for a standard Bloom filter.

- Negatives:
  - Perfect hash functions non-trivial to find.
  - Cannot handle on-line insertions.
Perfect Hashing Approach
Classic Uses of BF: Spell-Checking

- Once upon a time, memory was scarce...
- `/usr/dict/words` -- about 210KB, 25K words
- Use 25 KB Bloom filter
  - 8 bits per word.
  - Optimal 5 hash functions.
- Probability of false positive about 2%
- False positive = accept a misspelled word
- BF's still used to deal with list of words
  - Password security [Spafford 1992], [Manber & Wu, 94]
  - Keyword driven ads in web search engines, etc
Classic Uses of BF: Data Bases

- **Join**: Combine two tables with a common domain into a single table.
- **Semi-join**: A join in distributed DBs in which only the joining attribute from one site is transmitted to the other site and used for selection. The selected records are sent back.
- **Bloom-join**: A semi-join where we send only a BF of the joining attribute.
Example

<table>
<thead>
<tr>
<th>Empl</th>
<th>Salary</th>
<th>Addr</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>60K</td>
<td>…</td>
<td>New York</td>
</tr>
<tr>
<td>George</td>
<td>30K</td>
<td>…</td>
<td>New York</td>
</tr>
<tr>
<td>Moe</td>
<td>25K</td>
<td>…</td>
<td>Topeka</td>
</tr>
<tr>
<td>Alice</td>
<td>70K</td>
<td>…</td>
<td>Chicago</td>
</tr>
<tr>
<td>Raul</td>
<td>30K</td>
<td></td>
<td>Chicago</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>City</th>
<th>Cost of living</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>60K</td>
</tr>
<tr>
<td>Chicago</td>
<td>55K</td>
</tr>
<tr>
<td>Topeka</td>
<td>30K</td>
</tr>
</tbody>
</table>

- Create a table of all employees that make < 40K and live in city where COL > 50K.

- **Join**: send (City, COL) for COL > 50. **Semi-join**: send just (City).

- **Bloom-join**: send a Bloom filter for all cities with COL > 50.
A Modern Application: Distributed Web Caches
Web Caching

- Summary Cache: [Fan, Cao, Almeida, & Broder]
- If local caches know each other’s content...
  …try local cache before going out to Web
- Sending/updating lists of URLs too expensive.
- Solution: use Bloom filters.
- False positives
  - Local requests go unfulfilled.
  - Small cost, big potential gain
2. Compressed Bloom Filters

- Insight: Bloom filter is not just a data structure, it is also a message.
- If the Bloom filter is a message, worthwhile to compress it.
- Compressing bit vectors is easy.
  - Arithmetic coding gets close to entropy.
- Can Bloom filters be compressed?
Optimization, then Compression

- Optimize to minimize false positive.

\[ p = \Pr[\text{cell is empty}] = (1 - 1/m)^{kn} \approx e^{-kn/m} \]
\[ f = \Pr[\text{false pos}] = (1 - p)^k \approx (1 - e^{-kn/m})^k \]

\[ k = (m \ln 2) / n \text{ is optimal} \]

- At \( k = m (\ln 2)/n, \) \( p = 1/2. \)

- Bloom filter looks like a random string.
  - Can’t compress it.
Compressed Bloom Filters

- “Error optimized” Bloom filter is ½ full of 0’s, 1’s.
  - Compression would not help.
  - But this optimization for a fixed filter size $m$.
- Instead optimize the false positives for a fixed number of transmitted bits.
  - Filter size $m$ can be larger, but mostly 0’s
  - Larger, sparser Bloom filter can be compressed.
  - Useful if transmission cost is bottleneck.
- Claim: transmission cost limiting factor.
  - Updates happen frequently.
  - Machine memory is cheap.
### Benefits of Compressed Bloom Filters

<table>
<thead>
<tr>
<th>Array bits per elt.</th>
<th>m/n</th>
<th>8</th>
<th>14</th>
<th>92</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans. Bits per elt.</td>
<td>z/n</td>
<td>8</td>
<td>7.923</td>
<td>7.923</td>
</tr>
<tr>
<td>Hash functions</td>
<td>k</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>False positive rate</td>
<td>f</td>
<td>0.0216</td>
<td>0.0177</td>
<td>0.0108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array bits per elt.</th>
<th>m/n</th>
<th>16</th>
<th>28</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans. Bits per elt.</td>
<td>z/n</td>
<td>16</td>
<td>15.846</td>
<td>15.829</td>
</tr>
<tr>
<td>Hash functions</td>
<td>k</td>
<td>11</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>False positive rate</td>
<td>f</td>
<td>4.59E-04</td>
<td>3.14E-04</td>
<td>2.22E-04</td>
</tr>
</tbody>
</table>

- Examples for bounded transmission size.
  - 20-50% of false positive rate.
Benefits of Compressed Bloom Filters

<table>
<thead>
<tr>
<th>Array bits per elt.</th>
<th>m/n</th>
<th>8</th>
<th>12.6</th>
<th>46</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans. Bits per elt.</td>
<td>z/n</td>
<td>8</td>
<td>7.582</td>
<td>6.891</td>
</tr>
<tr>
<td>Hash functions</td>
<td>k</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>False positive rate</td>
<td>f</td>
<td>0.0216</td>
<td>0.0216</td>
<td>0.0215</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Array bits per elt.</th>
<th>m/n</th>
<th>16</th>
<th>37.5</th>
<th>93</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans. Bits per elt.</td>
<td>z/n</td>
<td>16</td>
<td>14.666</td>
<td>13.815</td>
</tr>
<tr>
<td>Hash functions</td>
<td>k</td>
<td>11</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>False positive rate</td>
<td>f</td>
<td>4.59E-04</td>
<td>4.54E-04</td>
<td>4.53E-04</td>
</tr>
</tbody>
</table>

- Examples with fixed false probability rate.
  - 5-15% compression for transmission size.
Example

\[ \frac{z}{n} = 8 \]
Results

- At $k = m \frac{(\ln 2)}{n}$, false positives are maximized with a compressed Bloom filter.
  - Best case without compression is worst case with compression; compression always helps.
- Side benefit: Use fewer hash functions with compression; possible speedup.
3. Counting Bloom Filters and Deletions

- Cache contents change
  - Items both inserted and deleted.
- Insertions are easy – add bits to BF
- Can Bloom filters handle deletions?
- Use Counting Bloom Filters to track insertions/deletions at hosts; send Bloom filters.
Handling Deletions

- Bloom filters can handle insertions, but not deletions.

\[ B = \begin{array}{cccccccccccc}
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array} \]

- If deleting \( x_i \) means resetting 1s to 0s, then deleting \( x_i \) will “delete” \( x_j \).
Counting Bloom Filters

Start with an $m$ bit array, filled with 0s.

$$B = \begin{array}{cccccccccccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}$$

Hash each item $x_j$ in $S$ $k$ times. If $H_i(x_j) = a$, add 1 to $B[a]$.

$$B = \begin{array}{cccccccccccccccccc} 0 & 3 & 0 & 0 & 1 & 0 & 2 & 0 & 0 & 3 & 2 & 1 & 0 & 2 & 1 & 0 \\
\end{array}$$

To delete $x_j$ decrement the corresponding counters.

$$B = \begin{array}{cccccccccccccccccc} 0 & 2 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 3 & 2 & 1 & 0 & 1 & 1 & 0 \\
\end{array}$$

Can obtain a corresponding Bloom filter by reducing to 0/1.

$$B = \begin{array}{cccccccccccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
\end{array}$$
Counting Bloom Filters: Overflow

- Must choose counters large enough to avoid overflow.
- Poisson approximation suggests 4 bits/counter.
  - Average load using $k = (\ln 2)m/n$ counters is $\ln 2$.
  - Probability a counter has load at least 16:

- Failsafes possible.

$$\approx e^{-\ln 2} (\ln 2)^{16} / 16! \approx 6.78E-17$$
Bloom Filters: Other Applications?

- P2P Keyword Search
- P2P Collaboration
- Resource Location
- Loop detection
- Scalable Multicast Forwarding
Efficient P2P keyword searching [Reynolds & Vadhat, 2002].

- Distributed inverted word index, on top of an overlay network. Multi-word queries.
- Peer A holds list of document IDs containing Word1, Peer B holds list for Word2.
- Need intersection, with low communication.
- A sends B a Bloom filter of document list.
- B returns possible intersections to A.
- A checks and returns to user; no false positives in end result.
- Equivalent to Bloom-join
P2P Collaboration

- Informed Content Delivery
  - Delivery of large, encoded content.
    - Redundant encoding.
    - Need a sufficiently large (but not all) number of distinct packets.
  - Peers A and B have lists of encoded packets.
  - Can B send A useful packets?
  - A sends B a Bloom filter; B checks what packets may be useful.
  - False positives: not all useful packets sent
  - Method can be combined with
    - Min-wise sampling (determine a-priori which peers are sufficiently different)
Resource Location

Queries sent to root. Each node keeps a list of resources reachable through it, through children.

List = Bloom filter.
Resource Location: Examples

- Secure Discovery Service
  - [Czerwinski, Zhao, Hodes, Joseph, Katz 99]
  - Tree of resources.

- OceanStore distributed file storage
  - [Kubiatowicz & al., 2000], [Rhea & Kubiatowicz, 2002]
  - Attenuated BFs – go d levels down in the tree

- Geographical region summary service
  - [Hsiao 2001]
  - Divide square regions recursively into smaller sub squares.
  - Keep and update Bloom filters for each level in hierarchy.
Loop detection

☐ Idea: Carry small BF in the packet header
☐ Whenever passing a node, the node mask is OR-ed into the BF
☐ If BF does not change there might be a loop
Scalable Multicast Forwarding

- Usual arrangement for multicast trees: for each source address keep list of interfaces where the packet should go
  - For many simultaneous multicasts, substantial storage required

- Alternative idea: trade computation for space:
  - For each interface keep BF of addresses
  - Packets checked against the BF. Check can be parallelized
  - False positives lead to (few) spurious transmissions
4. Representation of a set of \((key, f(key))\)

- **Hash-Based Approximate Counting**
  - Multiset problem: \((Key, frequency)\)
  - Space-code Bloom filters (INFOCOM 2004)
  - Spectral Bloom filters (SIGMOD 2003)

- **Bloomier Filter**
  - \((key, f(key))\)

- **Approximate Concurrent State Machines**
  - \((Key, state)\)
  - Beyond Bloom Filters: Approximate Concurrent State Machines (SIGCOMM 2006)
  - Fast Statistical Spam Filter by Approximate Classifications (Sigmetric 2006)
Hash-Based Approximate Counting

- Use min-counter associated with flow as approximation.
  - Yields approximation for all flows simultaneously.
  - Gives lower bound, and good approx.
  - Can prove rigorous bounds on performance.

- This hash-based approximate counting structure has many uses.
  - Any place you want to keep approximate counts for a data stream.
  - Databases, search engines, network flows, etc.
Use Counting Bloom filter to track bytes per flow. Potentially heavy flows are recorded.

The flow associated with $y$ can only have been responsible for 3 packets.
Example

The flow associated with \( y \) can only have been responsible for 3 packets; counters should be updated to 5.
Bloomier Filter

- Bloom filters handle set membership.
- Counters to handle multi-set/count tracking.
- Bloomier filter:
  - Extend to handle *approximate static functions*.
  - Each element of set has associated function value.
  - Non-set elements should return null.
  - Want to always return correct function value for set elements.
Approximate Concurrent State Machines

Motivation: Router State Problem

- Suppose each flow has a state to be tracked. Applications:
  - Intrusion detection
  - Quality of service
  - Distinguishing P2P traffic
  - Video congestion control
  - Potentially, *lots* of others!

- Want to track state for each flow.
  - But *compactly*; routers have small space.
  - Flow IDs can be ~100 bits. Can’t keep a big lookup table for hundreds of thousands or millions of flows!
Approximate Concurrent State Machines

- Model for ACSMs
  - We have underlying state machine, states 1...X.
  - Lots of concurrent flows.
  - Want to track state per flow.
  - Dynamic: Need to insert new flows and delete terminating flows.
  - *Can allow some errors.*
  - Space, hardware-level simplicity are key.
ACSM Basics

- **Operations**
  - Insert new flow, state
  - Modify flow state
  - Delete a flow
  - Lookup flow state

- **Errors**
  - False positive: return state for non-extant flow
  - False negative: no state for an extant flow
  - False return: return wrong state for an extant flow
  - **Don’t know**: return don’t know
    - Don’t know may be better than other types of errors for many applications, e.g., slow path vs. fast path.
ACSM via Counting Bloom Filters

- Dynamically track a set of current (FlowID, FlowState) pairs using a CBF.
- Consider first when system is well-behaved.
  - Insertion easy.
  - Lookups, deletions, modifications are easy *when current state is given*.
  - If not, have to search over all possible states. Slow, and can lead to don’t knows for lookups, other errors for deletions.
Direct Bloom Filter (DBF) Example

\[(123456,3) \rightarrow (123456,5)\]
Stateful Bloom Filters

- Each flow hashed to $k$ cells, like a Bloom filter.
- Each cell stores a state.
- If two flows collide at a cell, cell takes on don’t know value.
- On lookup, as long as one cell has a state value, and there are not contradicting state values, return state.
- Deletions handled by timing mechanism (or counters in well-behaved systems).
- Similar in spirit to [KM], Bloom filter summaries for multiple choice hash tables.
Fingerprint-compressed Filter Approach

- Store a fingerprint of flow + state in a d-left hashtable
Fingerprint-compressed Filter Approach

- Insert - hash the element, and find the corresponding bucket in each hash table, insert the fingerprint + state in the bucket with least number of elements
- Lookup – retrieve the state of the fingerprint
- Delete – remove the fingerprint
- Update – direct update or remove old + add new
- Timing-based deletion can still be applied
Stateful Bloom Filter (SBF) Example

(123456,3) → (123456,5)

```
1 4 3 4 3 3 0 0 2 1 0 1 4 ? 0 2
```

```
1 4 5 4 5 3 0 0 2 1 0 1 4 ? 0 2
```
5. Invertible Bloom Filters

What’s the Difference?
Efficient Set Reconciliation without Prior Context
Motivation

- Distributed applications often need to compare remote state.

Must solve the Set-Difference Problem!
What is the Set-Difference problem?

- What objects are unique to host 1?
- What objects are unique to host 2?
Example 1: Data Synchronization

- Identify missing data blocks
- Transfer blocks to synchronize sets

Diagram:

Host 1:
- A
- B
- C
- D
- E
- F

Host 2:
- A
- B
- C
- D
- E
- F
Example 2: Data De-duplication

- Identify all unique blocks.
- Replace duplicate data with pointers
Set-Difference Solutions

- Trade a sorted list of objects.
  - $O(n)$ communication, $O(n \log n)$ computation

- Approximate Solutions:
  - Approximate Reconciliation Tree (Byers)
    - $O(n)$ communication, $O(n \log n)$ computation

- Polynomial Encodings (Minsky & Trachtenberg)
  - Let “$d$” be the size of the difference
    - $O(d)$ communication, $O(dn+d^3)$ computation

- Invertible Bloom Filter
  - $O(d)$ communication, $O(n+d)$ computation
Difference Digests

- Efficiently solves the set-difference problem.

- Consists of two data structures:
  - Invertible Bloom Filter (IBF)
    - Efficiently computes the set difference.
    - Needs the size of the difference
  - Strata Estimator
    - Approximates the size of the set difference.
    - Uses IBF’s as a building block.
Invertible Bloom Filters (IBF)

- Encode local object identifiers into an IBF.

- Host 1

- Host 2

- Encode local object identifiers into an IBF.
IBF Data Structure

- Array of IBF cells
  - For a set difference of size, \( d \), require \( \alpha d \) cells \((\alpha > 1)\)
- Each ID is assigned to many IBF cells
- Each IBF cell contains:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idSum</td>
<td>XOR of all ID’s in the cell</td>
</tr>
<tr>
<td>hashSum</td>
<td>XOR of hash(ID) for all ID’s in the cell</td>
</tr>
<tr>
<td>count</td>
<td>Number of ID’s assign to the cell</td>
</tr>
</tbody>
</table>
IBF Encode

IBF:

- idSum ⊕ A
- hashSum ⊕ H(A)
- count++

Assign ID to many cells

Hash1, Hash2, Hash3

All hosts use the same hash functions

“Add” ID to cell

Not O(n), like Bloom Filters!
Invertible Bloom Filters (IBF)

- Trade IBF’s with remote host
Invertible Bloom Filters (IBF)

- “Subtract” IBF structures
  - Produces a new IBF containing only unique objects
$$B_2 = \langle W, Y, Z \rangle$$

<table>
<thead>
<tr>
<th>idSum:</th>
<th>hashSum:</th>
<th>count:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y \ H(Y)</td>
<td>W \ O \ Z \ H(W) \ O \ H(Z)</td>
<td>1</td>
</tr>
<tr>
<td>Z \ H(Z)</td>
<td>W \ O \ Y \ H(W) \ O \ H(Y)</td>
<td>2</td>
</tr>
<tr>
<td>Z \ H(Z)</td>
<td>W \ O \ Y \ O \ Z \ H(W) \ O \ H(Y) \ O \ H(Z)</td>
<td>3</td>
</tr>
</tbody>
</table>
Timeout for Intuition

- After subtraction, all elements common to both sets have disappeared. Why?
  - Any common element (e.g. W) is assigned to same cells on both hosts (assume same hash functions on both sides)
  - On subtraction, $W \ XOR \ W = 0$. Thus, W vanishes.

- While elements in set difference remain, they may be randomly mixed $\Rightarrow$ need a decode procedure.
Invertible Bloom Filters (IBF)

- Decode resulting IBF
  - Recover object identifiers from IBF structure.
IBF Decode

Step 1: Initial Scan

<table>
<thead>
<tr>
<th>Index</th>
<th>idSum: V ⊕ X</th>
<th>hashSum: H(V) ⊕ H(X)</th>
<th>count: 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V</td>
<td>H(V) ⊕ H(X)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>H(V) ⊕ H(X) ⊕ H(Z)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>V</td>
<td>H(V)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>H(Z)</td>
<td>-1</td>
</tr>
</tbody>
</table>

Pure: {3, 4}

$D_{A-B}: \{\}$

$D_{B-A}: \{\}$

Test for Purity:

$H(\text{idSum}) = \text{hashSum}$

$H(V) = H(V)$
## IBF Decode

### Step 2: Record

<table>
<thead>
<tr>
<th>IdSum: ( \text{IdSum} )</th>
<th>HashSum: ( \text{HashSum} )</th>
<th>Count: ( \text{Count} )</th>
<th>Index:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V \oplus X )</td>
<td>( H(V) \oplus H(X) )</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( V \oplus X \oplus Z )</td>
<td>( H(V) \oplus H(X) \oplus H(Z) )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( X \oplus Z )</td>
<td>( H(X) \oplus H(Z) )</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>( V )</td>
<td>( H(V) )</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>( Z )</td>
<td>( H(Z) )</td>
<td>-1</td>
<td>4</td>
</tr>
</tbody>
</table>

- Pure: \( \{3, 4\} \)
- \( D_{A-B} \): \( \{V\} \)
- \( D_{B-A} \): \( \{\} \)
IBF Decode

Step 3: Remove

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>X</th>
<th>X</th>
<th>V</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>hashSum:</td>
<td>X</td>
<td>X⊕Z</td>
<td>X⊕Z</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>idSum:</td>
<td>H(X)</td>
<td>H(X)⊕H(Z)</td>
<td>H(X)⊕H(Z)</td>
<td>0</td>
<td>H(Z)</td>
</tr>
<tr>
<td>count:</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Pure: \{4\}
\[D_{A-B}: \{V\}\]
\[D_{B-A}: \{\}\]
IBF Decode

Step 4: Update Pure List

<table>
<thead>
<tr>
<th>Index</th>
<th>idSum: H(X)</th>
<th>hashSum: H(X) ⊕ H(Z)</th>
<th>count:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X ⊕ Z</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>X ⊕ Z</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>X ⊕ Z</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Z</td>
<td>H(Z) ⊕ H(Z)</td>
<td>-1</td>
</tr>
</tbody>
</table>

Pure: \{4, 0\}
\[D_{A-B}: \{V\}\]
\[D_{B-A}: \{\}\\]
How many IBF cells?

Overhead to decode at >99%

Small Diffs: 1.4x – 2.3x

Large Differences: 1.25x - 1.4x
How many hash functions?

- 1 hash function produces many pure cells initially but nothing to undo when an element is removed.
How many hash functions?

- 1 hash function produces many pure cells initially but nothing to undo when an element is removed.
- Many (say 10) hash functions: too many collisions.
How many hash functions?

- 1 hash function produces many pure cells initially but nothing to undo when an element is removed.
- Many (say 10) hash functions: too many collisions.
- We find by experiment that 3 or 4 hash functions works well. Is there some theoretical reason?
Theory

- Let $d =$ difference size, $k =$ # hash functions.
- **Theorem 1**: With $(k + 1) d$ cells, failure probability falls exponentially.
  - For $k = 3$, implies a 4x tax on storage, a bit weak.
- [Goodrich, Mitzenmacher]: Failure is equivalent to finding a 2-core (loop) in a random hypergraph
- **Theorem 2**: With $c_k d$, cells, failure probability falls exponentially
  - $c_4 = 1.3x$ tax, agrees with experiments
How many IBF cells?

Overhead to decode at >99%

Large Differences:
1.25x - 1.4x
Connection to Coding

- **Mystery**: IBF decode similar to peeling procedure used to decode Tornado codes. Why?

- **Explanation**: Set Difference is equivalent to coding with insert-delete channels

- **Intuition**: Given a code for set A, send **codewords only** to B. Think of B’s set as a corrupted form of A’s.

- **Reduction**: If code can correct D insertions/deletions, then B can recover A and the set difference.

Reed Solomon <---> Polynomial Methods
LDPC (Tornado) <---> Difference Digest
Difference Digests

- Consists of two data structures:
  - Invertible Bloom Filter (IBF)
    - Efficiently computes the set difference.
    - Needs the size of the difference
  - Strata Estimator
    - Approximates the size of the set difference.
    - Uses IBF’s as a building block.
Strata Estimator

- Divide keys into partitions of containing $\sim 1/2^k$
- Encode each partition into an IBF of fixed size
  - $\log(n)$ IBF’s of $\sim 80$ cells each
Strata Estimator

• Attempt to subtract & decode IBF’s at each level.
• If level $k$ decodes, then return: $2^k \times$ (the number of ID’s recovered)
• Attempt to subtract & decode IBF’s at each level.
• If level $k$ decodes, then return: $2^k$ x (the number of ID’s recovered)

Strata Estimator
Observation: Extra partitions hold useful data
Sum elements from all decoded strata & return: $2^{(k-1)} \times \text{(the number of ID’s recovered)}$
Estimation Accuracy

Average Estimation Error (15.3 KBytes)

- Strata good for small differences.
- Min-Wise good for large differences.
Hybrid Estimator

- Combine Strata and Min-Wise Estimators.
  - Use IBF Stratas for small differences.
  - Use Min-Wise for large differences.
Hybrid Estimator Accuracy

Average Estimation Error (15.3 KBytes)

- Hybrid matches Strata for small differences.
- Converges with Min-wise for large differences.
Promising Applications:
- File Synchronization
- P2P file sharing
- Failure Recovery
Difference Digests Summary

- **Strata & Hybrid Estimators**
  - Estimate the size of the Set Difference.
  - For 100K sets, 15KB estimator has <15% error
  - $O(\log n)$ communication, $O(\log n)$ computation.

- **Invertible Bloom Filter**
  - Identifies all ID’s in the Set Difference.
  - 16 to 28 Bytes per ID in Set Difference.
  - $O(d)$ communication, $O(n+d)$ computation.

- **Implemented in KeyDiff Service**
Conclusions: Got Diffs?

- New randomized algorithm (difference digests) for set difference or insertion/deletion coding
- Could it be useful for your system? Need:
  - Large but roughly equal size sets
  - Small set differences (less than 10% of set size)
Comparison to Logs

- IBF work with no prior context.
- Logs work with prior context, BUT
  - Redundant information when sync’ing with multiple parties.

IBF’s may out-perform logs when:
  • Synchronizing multiple parties
  • Synchronizations happen infrequently
The main point revised again

☐ Whenever you have a set or list or function or concurrent state machine or whatever-will-be-next?, and space is an issue, an approximate representation, like a Bloom filter may be a useful alternative.

☐ Just be sure to consider the effects of the false positives!
Extension: Distance-Sensitive Bloom Filters

- Instead of answering questions of the form \( Is \ y \in S. \)

we would like to answer questions of the form \( Is \ y \approx x \in S. \)

- That is, is the query close to some element of the set, under some metric and some notion of close.

- Applications:
  - DNA matching
  - Virus/worm matching
  - Databases

- Some initial results [KirschMitzenmacher].
Variation: Simpler Hashing

- [DillingerManolios],[KirschMitzenmacher]
- Let $h_1$ and $h_2$ be hash functions.
- For $i = 0, 1, 2, \ldots, k - 1$ and some $f$, let
  \[ g_i(x) = h_1(x) + ih_2(x) \mod m \]
  - So 2 hash functions can mimic $k$ hash functions.
- Hash functions:
  - SDBM, BUZ
- Fast generation of very high quality pseudorandom numbers
  - Mersenne twister