Bloom Filters

• Bloom filters offer a succinct way to represent a set of items

• Representation reduces space, but pays with it with false positives

• **Bloom Filter Principle**: Wherever a list or a set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.
Standard Bloom Filter

• goal: represent a set of \( S = \{x_1, x_2, \ldots, x_n\} \)

• representation is an array of \( m \) bits

• uses \( k \) independent hash functions: \( h_1, h_2, \ldots, h_k \)
  – hash functions have output range \( \{1,\ldots,m\} \)
Entering Data

To enter a set $S$ into a bloom filter:
   for every $x$ in $S$
      for every hash function $h_i$ ($i = 1$ to $k$)
         set $h_i(x)^{th}$ bit of $m$ to 1

Example: entering $x_1$ and $x_2$ into bloom filter

$h_1(x_1)=2$
$h_2(x_1)=5$
$h_3(x_1)=9$

$h_1(x_2)=5$
$h_2(x_2)=7$
$h_3(x_2)=11$
Checking Membership

To check membership of \( y \):
check \( h_i(y) \)th bit of \( m \) for every \( i = 1 \) to \( k \)

if any bit is 0: \( y \) is not in the set
if all bits are 1: \( y \) is in set (or false positive)

Example: checking membership of \( y_1 \) and \( y_2 \)

\[
\begin{align*}
h_1(y_1) &= 2 \\
h_2(y_1) &= 4 \\
h_3(y_1) &= 8 \\
h_1(y_2) &= 5 \\
h_2(y_2) &= 7 \\
h_3(y_2) &= 11
\end{align*}
\]
False Positives

(1/m) chance specific hash of specific element hit specific bit
(1 - (1/m)) chance specific hash of specific element missed specific bit
(1 - (1/m))^k chance all hashes of specific element missed specific bit
(1 - (1/m))^{kn} chance all hashes of all elements missed specific bit

After all elements of S have been hashed into the Bloom filter…

\[ P(\text{a specific bit is 0}) = p' = (1 - (1/m))^{kn} \approx e^{-kn/m} = p \]

Let \( \rho \) = proportion of all bits that are 0 after all \( n \) elements entered

\[ E(\rho) = p' \]
False Positives

- False positives occur when both of the following hold:
  - \( y \) is not an element in the set
  - bits corresponding to all hashes of \( y \) are set in the bit vector

\[(1 - \rho) = \text{chance corresponding bit of a specific hash of } y = 1\]
\[(1 - \rho)^k = \text{chance all corresponding bits of all hashes of } y = 1\]

- Let \( f \) and \( f' \) represent the probability of a false positive:

\[f = (1 - p)^k \approx (1 - \rho)^k \approx (1 - p')^k = f'\]
\[f' = (1 - (1 - (1/m))^{kn})^k\]
\[f = (1 - e^{-kn/m})^k\]
Alternate Forms

Bloom filters are sometimes implemented such that each hash function has an output range of \( \frac{m}{k} \) consecutive bits, disjoint from all other hash function outputs.

\[
P(\text{specific bit is 0 after all elements added}) = (1 - \frac{k}{m})^n \approx e^{-kn/m}
\]

Note that for all \( k \) greater than or equal to 1:
\[
(1 - \frac{k}{m})^n \leq (1 - \frac{1}{m})^{kn}
\]

Probability of a false positive is always at least as large in this form.
Choosing # of Hash Functions

• given \( m \) and \( n \), how many hash functions to use?
  – more hash functions yields more chances to find a 0 bit for elements not in \( S \)
  – fewer hash functions increases the fraction of the bits that are 0

• to find optimal number, minimize probability of false positive (\( f \)) with respect to number of hash functions (\( k \))
  – minimum occurs when \( p = 1/2 \)
  – equivalently, when \( k = \ln 2 \times (m/n) \)
  – number of 0 bits in Bloom filter may not exactly equal \( p \)
    • with high probability it will be very close to \( p \), for large arrays
Lower Bound

• How many bits $m$ are necessary to represent all sets of $n$ elements in a manner that allows false positives for at most a fraction $\varepsilon$ of the universe (and no false negatives)?

$u =$ size of universe
$(u \text{ choose } n) =$ number of possible sets with $n$ elements
$F(X) =$ $m$-bit string to which our representation maps set $X$

• Given an $m$-bit string $s$ and an element $x$:
  – $s$ accepts $x$ if there exists a set $X$ st. $x$ is in $X$ and $F(X) = s$
  – $s$ rejects $x$, otherwise
Consider a specific set $X$ of $n$ elements

- Any string $s$ that is used to represent $X$
  - must accept every element in $X$
  - may accept $\varepsilon(u - n)$ other elements

Therefore...

- $n + \varepsilon(u - n) =$ number of elements that a string $s$ can accept
- a string $s$ can represent any subset of these elements that is of size $n$
- $((n + \varepsilon(u - n)) \choose (n)) =$ number of such subsets
Lower Bound

• We must be able to represent all possible sets using the distinct strings that we have.

\((# \text{ distinct strings}) \times (# \text{ sets a string can represent}) \geq (# \text{ sets in universe})\)

\[
2^m \binom{n + \epsilon(u-n)}{n} \geq \binom{u}{n},
\]

\[
m \geq \log_2 \frac{\binom{u}{n}}{\binom{n}{n} \frac{1}{n+\epsilon(u-n)}} \approx \log_2 \frac{\binom{u}{n}}{\binom{u}{u} \frac{1}{\epsilon u}} \geq \log_2 \epsilon^{-n} = n \log_2 (1/\epsilon).
\]
Bloom vs. Lower Bound

Probability of a false positive in Bloom filter:

\[ f = (1/2)^k \geq (1/2)^{m \ln 2/n}, \]

$f \leq \epsilon$ requires

\[ m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} = n \log_2 e \cdot \log_2(1/\epsilon). \]

Bloom filter is within $\log_2 e$ of the lower bound (space-wise).

Keeping space constant, using $n \times j$ bits for table, false positives are:

Lower Bound false positive rate = $(0.5)^j$
Bloom Filter false positive rate = $(0.6185)^j$
Hashing vs. Bloom Filters

• Hashing
  – each item in set is hashed into $\Theta(\log n)$ bits
  – sorted list of hash values represents set
  – very small error probabilities
    • using $2\log_2 n$ bits/element $\Rightarrow$ probability that the hash of an element not in the set matches hash of an element in the set is at most $1/n$

• Bloom filters: generalization of hashing
  – more tradeoffs between # of bits used and false positives
  – example: constant false positive rate for constant number of bits/element
  – not interesting theoretically, very important practically
    • constant false positives are worth it to keep number of bits per element constant
Bloom Filter Tricks

Given two bloom filters representing sets $S_1$ and $S_2$ respectively

- $S_1 \cup S_2 = \text{OR of the bit vectors from the bloom filters}$

- $|S_1 \cap S_2|$ can be approximated
  
  $E(\text{magnitude of inner product}) =$
  
  $$m \left( 1 - \left(1 - \frac{1}{m}\right)^{k|S_1|} - \left(1 - \frac{1}{m}\right)^{k|S_2|} + \left(1 - \frac{1}{m}\right)^{k(|S_1|+|S_2|-|S_1 \cap S_2|)} \right).$$

- Given $|S_1|$, $|S_2|$, $k$, $m$ and magnitude of inner product, one can calculate an estimate of the $|S_1 \cap S_2|$

Bloom filters can be easily halved in size

- OR the first and second halves
- when hashing to do a lookup, mask the higher bit
Counting Bloom Filters

• How do we handle sets that change over time?
  – insertion is easy: compute hashes and set corresponding bits to 1
  – deletion is harder: compute hashes and set corresponding bits to 0
  – problem: what if we are unsetting a bit that is used by a different element in the set?

• Counting Bloom filters
  – each entry in the Bloom filter is a counter
  – insertion: compute hashes and increment corresponding counters
  – deletion: compute hashes and decrement corresponding counters
Counting Bloom Filters

- Counter size should be large enough to prevent overflow
- How likely is overflow?

\[
P(\text{a specific counter} = j) = {n \choose j} \left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{nk-j}
\]

\[
P(\text{a specific counter} \geq j) = {nk \choose j} \frac{1}{m^j} \leq \frac{e^{nk}}{j^m}
\]

\[
P(\text{any counter} \geq j) \leq m \cdot P(\text{a specific counter} \geq j)
\]

// Plugging in \( k = (\ln 2)m/n \)

\[
P(\text{any counter} \geq j) \leq m \left(\frac{e \ln 2}{j}\right)^j
\]
Counting Bloom Filters

- Plugging in for 4 hash functions…
  \[ P(\text{max } c(i) \geq 16) \leq 1.37 \times 10^{-15} \times m \]

- For a Bloom filter that holds \( t \) different sets of at most \( n \) elements, probability of an overflow is \( = 1.37 \times 10^{-15} \times mt \)

- What happens when counter does overflow?
  - Just leave it at maximum value
  - Only a problem if counter gets decremented all the way down to 0
  - Expected time for this to happen is very long
Compressed Bloom Filters

- Each bit in a standard Bloom filter bit vector has a (1/2) chance of being set to 1
  - not much opportunity for compression

- Use a larger (and sparser) Bloom filter
  - sparser means it can be compressed
  - same false positive rate
  - compressed filter may be fewer bits than uncompressed optimal

<table>
<thead>
<tr>
<th>Array bits per element</th>
<th>$m/n$</th>
<th>16</th>
<th>28</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmission bits per element</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 probability</td>
<td>$f$</td>
<td>0.000459</td>
<td>0.000314</td>
<td>0.000222</td>
</tr>
</tbody>
</table>

Table 1. Using at most sixteen bits per element after compression, a bigger but sparser Bloom filter can reduce the false positive probability.
Historical Applications : Dictionaries

Hyphenation
• most words can be hyphenated using simple rules
• a few require table lookup
• solution:
  – store words that require table lookup in Bloom filter
  – false positives: unnecessary table lookup

Spell-Checking
• store all legal words in Bloom filter
• false positives: misspelled word may be missed

Dictionary of Unsuitable Passwords
• store all unsuitable passwords
• false positives: suitable passwords get rejected
Historical Applications: Databases

Semi-joins

- database A holds data on cost of living
- database B holds data on where employees live
- which employees live in a city where the cost of living is greater than $50,000?

- solution:
  - A sends a Bloom filter of matching cities to B
  - B compiles list of potential employee/city pairs, sends to A
  - database A examines list and removes false positives

- potential to reduce amount of communication
- also used to evaluate size of semi-join
Historical Applications: Databases

**Differential File**

- differential file contains all changes made to a database
- when reading a field, can we read straight from database or do we have to read from differential file? has file been changed?

- solution:
  - store which files have been changed in Bloom filter
  - false positives: unnecessarily read differential file
Network Application: Distributed Caching

Summary Cache

• a collection of proxies attempt to share their web caches
• whenever a proxy has a cache miss it:
  – determines if another proxy has the desired web page
  – if so, request the page from that proxy
• solution:
  – proxies share a Bloom filter of what their cache contains
  – false positives: a proxy attempts to download a page from a proxy that doesn’t hold the page
    • this happens sometimes anyway!
• contents of caches change frequently
  – proxies use a counting Bloom filter internally
  – communicate standard 1/0 bloom filter to other proxies
Applications: P2P/Overlay Networks

Moderate-Sized P2P Networks
- Bloom filter instead of DHT
- keeping list of all items stored at each node is intractable
- keeping Bloom filter for every other node may be tractable
- false positives: extraneous requests for objects

Approximate Set Reconciliation for Content Delivery
- peers A and B have sets of items $S_A$ and $S_B$ respectively
- A wants all items that B has but A does not
- solution:
  - A sends Bloom filter of its items to B
  - B sends all items that are not in Bloom filter
- false positives: some items in $S_B - S_A$ will not be sent
- example application: collaboration during downloading
Applications: P2P/Overlay Networks

Set Intersection for Keyword Searches

• to determine $S_A$ intersection $S_B$

• Solution: same as database solution
  – B sends a Bloom filter representing $S_B$ to A
  – A sends all items that appear to be in $S_B$ to B
  – B checks for false positives

• allows for fewer bits to be transmitted
Applications: Resource Routing

Basic Routing Protocol

• network in the form of a rooted tree, nodes hold resources
• each node maintains:
  – unified list of all resources available to it and all descendents
  – list of its own resources and unified lists of children
• when resource request is received
  – if resource not in unified list, pass up the tree
  – if resource is in unified list, check individual lists and forward appropriately

• Bloom filters as unified list
  – easily compute union of children’s lists by performing OR on bit vectors
  – false positives: request gets passed down incorrect path
Applications: Resource Routing

Resource Routing in P2P Networks: OceanStore

- each node maintains array of Bloom filters for every adjacent edge

- $d^{th}$ Bloom filter in array represents files reachable within $d$ hops

- false positives: results in file not found in $d$ hops, resort to more expensive algorithm

- major challenge: keeping Bloom filters up to date without generating too much traffic
Applications: Resource Routing

**Geographic Routing**

- root node represents entire geographic space
- 4 children represents 4 quadrants of this space
- each of them have 4 children, etc.,
- each node contains Bloom filter representing mobile hosts reachable through itself and its siblings
- source finds level that has contains it and its destination
- forwards a message to the center of sibling that contains destination
Applications: Packet Routing

Detecting Loops

- normally packets in a loop die off because of Time-To-Live
- if loops are small, many loops will be taken before packets die
- not an issue in Internet traffic, but maybe in P2P

- solution
  - place a small Bloom filter in the header of each packet
  - add nodes visited to Bloom filter
  - if node already visited, there may be a loop
  - false positives: detects loop when there wasn’t one, packet must be resent?
Applications: Packet Routing

Queue Management: Stochastic Fair Blue

- goal: detect overly aggressive or non-responsive flows
- counting Bloom filter is used (hashes of source-destination pair)
  - packets from same flow will affect same counters
  - when packet is received, all k counters are incremented
  - when packet processed, all k counters are decremented
  - each counter has a marking probability associated with it
  - when packet received
    - if corresponding counter is above some threshold, p is incremented
    - if corresponding counter is 0, p is decremented
  - mark packet with probability equal to minimum of corresponding p’s
  - result: flows filling a buffer / unresponsive flows will be marked with high probability
  - false positives: flow may get punished unfairly, change hash function often
Applications: Packet Routing

**Multicast**

- Normally:
  - store list of interfaces for each multicast address
  - when packets with multicast address and interface associated with multicast address, packet should be forwarded to all interfaces on list
- Bloom filters
  - each interface has a Bloom filter
  - Bloom filter stores addresses associated with the interface
  - when packet received, each interface checks its Bloom filter to see if packets with that address should be forwarded
Applications: Measurement Infrastructure

Recording Heavy Flows

- goal: easily determine heavy flows in a router
- when a packet enters the router hash it k times, increment each entry in bit vector by the # of bytes
- if the minimum of the k counters associated with a packet is over a threshold, corresponding flow is considered heavy
- false positives: light flows may happen to hash into locations that heavy flows hash in to (or many other light flows)
- conservative update
  - when packet enters the router, maximum number of bytes previously received from the flow is minimum of packet’s k counters (M_k)
  - if new packet has B bytes, number of bytes associated with flow is M_k + B
  - increase each buffer to a maximum of M_k + B
  - this reduces likelihood of multiple light flows raising the counter over the threshold
Applications: Measurement Infrastructure

**IP Tracebook**

- each router in network stores all packets that have passed through it
- when tracing a packet, simply ask each router if it had passed the packet and trace backwards
- instead of storing each packet, use Bloom filter to represent
- false positives: means that there will be branching when tracing back
- if branching is not too high, this is not a problem
Recent Work

• Spectral Bloom Filters
  – extended to handle multiple sets
  – second filter to handle elements that have a unique minimum counter
• Count-Min Sketch
  – theoretical guarantees while using only pairwise hash functions
• Space-code Bloom Filter
  – multiple Bloom filters and maximum likelihood estimation in order to approximate multisets
• Bloomier function
  – stores a function for every value in a set S
Summary Cache
Web Cache Sharing

- proxy caches cooperate and serve each other’s cache misses
- reduces bandwidth consumption
- reduces traffic through bottlenecks

- Internet Cache Protocol (ICP)
  - most common form of web cache sharing
  - on cache miss, a proxy multicasts the request to all other proxies
  - communication costs and CPU costs scale \textit{quadratically} with # of proxies

- Summary Cache
  - alternative to ICP
  - each proxy stores a summary of every other proxy’s cache
  - on cache miss, proxy examines the summaries and forwards the request to proxies that might have the web page cached
Data For Experiments

- 5 sets of traces of HTTP requests were collected
- DEC, UCB, Upisa:
  - data on HTTP traces from user computers
- Questnet, NLANR
  - traces of request received by parent proxies from child proxies

To simulate cache sharing for this project
- DEC, UCB, UPisa are partitioned into groups
  - each group is a proxy
  - 16, 8, and 8 proxies respectively
- Questnet and NLANR
  - child proxies are the proxies
  - 12 and 4 proxies respectively
Does Cache Sharing Work?

- 4 Different Strategies Evaluated
  - No Cache Sharing
  - Simple Cache Sharing (ICP)
    - proxies serve each other’s cache misses
    - proxies locally cache documents received from other proxies
    - no coordination of cache replacement
  - Single-Copy Cache Sharing
    - proxies serve each other’s cache misses
    - proxies do NOT cache documents received from other proxies
    - serving proxy updates the document to served to be most-recently-used
  - Global Cache
    - one unified cache with global cache replacement strategy
    - proxies all share contents and coordinate replacements
    - effectively one global cache
Evaluating Cache Strategies

- infinite cache size = total size of all data that gets cached
- hit ratio = percentage of requests that are satisfied by local or remote cache
- evaluate strategies using different cache sizes
  - 0.5%, 5%, 10%, and 20% of infinite cache
Cache Sharing Results

- all cache sharing schemes are better than no cache sharing
- **single-copy** and **simple** perform as well as **global**
  - global LRU works worse than local group-wise
- not much difference between **single-copy** and **simple**
  - slightly smaller cache doesn’t matter that much
- **ICP-style simple cache sharing** performs as well as more complicated cooperative caching
Overhead of ICP

• Considerable overhead, even when number of proxies is low (4)
  – number of UDP messages increased by factor of 73 to 90
  – total network traffic seen by proxies increased 8%
  – user CPU time increased by 20%
  – system CPU time increased by 7%
  – average HTTP latency increased by 8%

Dilemma:
  clear benefits of cache sharing, but also great overhead
Summary Cache

- Each proxy maintains a summary of every other proxy’s caches

- When cache miss occurs
  - proxy examines its list of summaries
  - forwards request to proxy that might contain requested document

- Two type of errors affect total cache hit ratio
  - false hit:
    - summary says the document is held by a proxy, but it is not
    - wasted query message to remote proxy
  - false miss:
    - summary says the document is not held by a proxy, but it is
    - proxy does not take advantage of cached copy, decreased hit ratio

- Errors affect hit ratio and interproxy traffic, but not correctness

- remote stale hits: remote proxy provides cache that is old
Evaluating Update Delays

- when percentage of documents in cache that are “new” crosses some threshold, update summary

- tested thresholds of 0.1%, 1%, 2%, 5%, and 10% of total cache size

- for this test, summaries are copies of cache directories
Evaluating Update Delays

- degradation of total cache hit ratio grows almost linearly with update threshold
- remote stale hit ratio not affected by update delay
- false hit ratio increases linearly with threshold (but is very small)
- update delay threshold of 1% to 10% yields tolerable degradation of total cache hit ratio
Summary Representations

• How do we represent the summaries?

• 3 options considered
  
  – exact-directory: cache directory with each URL represented by its MD5 signature
  
  – server-name: list of server name component of URLs in cache
  
  – counting Bloom filter
    • load factors of 8, 16, and 32 were tested
    • load factor = ratio of bits in filter to number of entries in filter
    • 4 hash functions used in each configuration
Summary Representations

- Bloom filters have virtually same cache hit ratio as exact-dir
- server has higher hit ratio because of so many false hits
Summary Representations

- **Bloom filter** has slightly higher ratio than **exact-dir**
- **server** is much much higher
Summary Representations

- **exact-dir** and **Bloom filters** perform well
- ICP and **server-name** generate a lot of messages
- tradeoff in **Bloom filters** between # of bits and # of messages
- false hit ratio is small enough that interproxy messages are dominated by remote cache hits and remote stale hits
  - little different between **Bloom-16** and **Bloom-32**
- **Bloom filters** reduced # of messages by 25-60 times over ICP
Summary Representations

- **Bloom filters** improve over ICP by 55% to 64% in terms of message bytes sent
- **Bloom filters** send a few large messages, ICP sends many small messages
Prototype

- Prototype built upon Squid
- tested using trace-replay and synthetic experiments
- reduces CPU overhead between 30% and 95%
- improves client latency