An Optimal Bloom Filter Replacement [a]

Rasmus Pagh, IT University of Copenhagen

Joint work with

Anna Pagh, IT University of Copenhagen

S. Srinivasa Rao, University of Waterloo

[a] To appear in *SODA 2005*

Outline

- Bloom filters
- Applications of Bloom filters
- Our replacement for Bloom filters
- Improvements over some extensions
- Conclusions and open problems

Bloom filter – abstract data structure

A randomized data structure for approximate membership queries.

Store $S \subseteq U$ efficiently to answer:

Given $x \in U$, 'is $x \in S$?' correctly with high probability

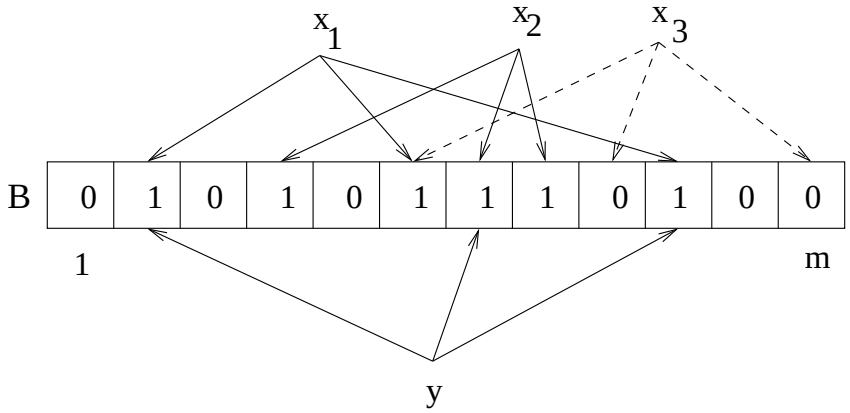
- For $x \in U$, if $x \in S$ answer YES
- if $x \notin S$ answer NO with probability $\geq 1 - \epsilon$

I.e., false positives are allowed, but not false negatives.

Bloom filter

Let $h_1, h_2, \ldots, h_k : U \to \{1, \ldots, m\}$ be truly random functions

[Bloom, *CACM '70*]



Storage scheme: Bit vector where $B[h_i(x)] = 1$ for $x \in S$, $1 \leq i \leq k$

Query scheme: answer YES iff $B[h_1(y)] = \ldots = B[h_k(y)] = 1$

Insertion: straightfoward; Deletions: not supported

Applications of Bloom filters

Used in early UNIX spell-checkers to save space

To store a dictionary of unsuitable passwords

Differential file for a database

- store the updates to a database in a differential file (and periodically merge with the database)
- store the primary keys of the updated records using a Bloom filter

To speed up semijoin operations in distributed databases
(to compute the intersection of two sets)

Applications

Web cache sharing

Longest prefix matching (IP lookup)

Network traffic flow measurement - Multi-resolution Space-code Bloom filters

Cryptography - Secure indexes, Encrypted Bloom filters; history independent

Bloom filter principle [Broder & Mitzenmacher, '02]: Whenever a list or set is used, and space is a consideration, a Bloom filter should be considered. When using a Bloom filter, consider the potential effects of false positives.

Bloom filter space and time

Space: m bits (plus the space for the hash functions)

Query time: O(k)

Smallest ϵ for $k \approx \ln 2 \cdot (m/n)$, namely $\epsilon \approx 2^{-k}$.

Equivalently: $m = n \log(1/\epsilon)/\ln 2 \approx 1.44\, n \log(1/\epsilon)$.

Best possible space is around $n \log(1/\epsilon)$.

Can it be achieved by an efficient data structure?

Shortcomings of Bloom filter

1. **Dependence on ϵ:** query time $k = \lg(1/\epsilon)$ grows as the false positive rate ϵ decreases
2. **Suboptimal space:** space usage is a factor 1.44 from optimal
3. **Lack of hash functions:** there is no known way of choosing the hash functions that can be shown to work
4. **No deletions:** deletions are not supported (unless using asymptotically more space)

Some solutions

Single hash function: time - $O(1)$; but space - (n/ϵ) (1 & 3)
[Carter et. al., STOC '78]

Compression: by compressing the Bloom filter, space can be reduced to the optimum (2)
[Mitzenmacher, IEEE Transactions on Networking '02]

Counting Bloom filters: by storing the multiplicities of the hashed locations, one can support deletions (4), but increases the space asymptotically
[Fan et al., IEEE Transactions on Networking '00]

Our solution

- Use a single hash function, $h : U \to [n/\epsilon]$ to map the elements of S into a bit vector B of size n/ϵ
- Store the bit vector efficiently

B is a bit vector of size n/ϵ with at most n 1s

We can represent B using $\lg \binom{n/\epsilon}{n} + o(n) \approx n \lg(1/\epsilon) + O(n)$ bits

Queries take $O(1)$ time [Pagh, ICALP '99]

Resolves 1, 2 and 3 – need to dynamize

Dynamization

We can store B using a succinct dynamic set structure to support insertions [Raman & Rao, ICALP '03]

To support deletions, we store $\{h(x)|x \in S\}$ as a *multi*set

Insertions and deletions correspond to incrementing and decrementing the multiplicities of the hashed values

Need: *Succinct dynamic multiset* representation that supports lookup, insert/delete queries

Succinct dynamic multiset

Theorem: A dynamic multiset of n elements from $[m]$ can be maintained using $B + o(B) + O(n)$ bits, where $B = \lg \binom{m+n}{n}$, while supporting lookups in $O(1)$ time, insert/delete in $O(1)$ expected amortized time.

The proof uses a reduction from a multiset to a collection of set representations, a solution to maintaining binary counters in the bit probe model, and some memory management techniques

Main result

Theorem: Given a positive constant $\epsilon < 1$, a dynamic multiset M of size at most n, with elements from $\{0,1\}^w$ can be maintained such that:

- (approximate) checking whether a given $x \in U$ belongs to M can be done in $O(1)$ time. If $x \in M$, the answer will be YES. If $x \notin M$, the answer is NO with probability at least $1 - \epsilon$
- insertions and deletions to M can be done in $O(1)$ expected amortized time. (Deletions are not 'verified')
- the space usage is at most $(1 + o(1))n \lg(1/\epsilon) + O(n + w)$ bits.

A practical variant

Replace the succinct dynamic dictionary structure with a simple dynamic hashing scheme by [Cleary, IEEE Trans. on Computers '84]

Space - $n \lg(1/\epsilon) + O(n)$
Query time - $O(\lg(1/\epsilon))$ (word probes)

Memory accesses are sequential - better cache performance than Bloom filters

Spectral Bloom filter

[Cohen & Matias, SIGMOD '03]

Generalizes a Bloom filter to store an approximate multiset.

Membership query is generalized to a multiplicity query.

Space usage is same as a Bloom filter; query time is $\Theta(\lg(1/\epsilon))$.

Using our structure space can be made optimal, while the query time is $O(\lg c)$ for a query element with multiplicity c

Bloomier filter

[Chazelle et.al., SODA '04]

An element x has satellite information $f(x) \in [2^s]$ associated with it.

For $x \in S$, we need to return $f(x)$;
for a false-positive, we can return $f(x)$ for an arbitrary $x \in S$

Space: $O(n \log(1/\epsilon) + ns)$; query time: $O(1)$

Our improvement: Space: $n \lg(1/\epsilon) + O(n + \lg w)$; Query time $O(1)$

Lossy dictionary

[Pagh & Rodler, ESA '01]

Set representation with both false positives and false negatives

A lossy dictionary with δn false negatives requires space that is $(1 - \delta)$ times that of one without false negatives

Static case: optimal space is obtained by omitting a δ fraction of the keys in our data structure.

We get optimal space (+ lower order terms) even in the dynamic case.

Conclusions

- *space and time optimal* approximate dictionary using *explicit* hash function families that supports insertions and *deletions*.
- A practical variant and improvements over some extensions of Bloom filters.

Practical impact?

It would be nice to see if our "practical variant" beats Bloom filters for small ϵ. A great student project! (But don't use Cleary's algorithm directly.)