CS 261: Data Structures

Week 3: Sets

Lecture 3b: Filters

David Eppstein
University of California, Irvine

Spring Quarter, 2025

This work is licensed under a Creative Commons Attribution 4.0 International License

Filters

Main idea of filters

Represent n-element sets using only $O(n)$ bits

Better than hash tables, $O(n)$ words

Better than bitmaps, $O(N)$ bits where $N =$ max element

What do we have to pay to get this savings?

Answers are approximate

If $x \in S$, filter will always say that $x \in S$
(cannot have "false negatives")

But if $x \notin S$, it might incorrectly say $x \in S$
(can have "false positives")

False positive rate

Choose a random x that is not in your set S

What is the probability that your filter incorrectly says $x \in S$?

Called the "false positive rate"

We want it to be small, so we will use ε as notation

Typically known when we initialize filter structure,
used to determine its structural parameters

Often (but not always) ok to assume constant, e.g. $\varepsilon = 0.1$

When are filters useful?

If processing non-members is easier and you expect many of them

Filter can be small enough to fit in cache $\Rightarrow$ fast

Use slower exact set data structure to check matched elements

Few false positives $\Rightarrow$ few unnecessary calls to exact structure

When are filters useful?

If memory is limited and some false positives are harmless

Example: Access control for private internet server

Use filter on firewall to only allow whitelisted clients through

Firewall needs only small memory for filter

Server can handle smaller volume of non-clients that get through

Comparison of filters: Bloom filter

[Bloom 1970]; $\approx$ 28k other publications

Widely implemented, practical

Storage: $1.44 n \log_2 \frac{1}{\varepsilon}$ bits
larger than optimal by the 1.44 factor

Membership testing: $O(\log 1/\epsilon)$ time

Can add but not remove elements

Comparison of filters: Cuckoo filter

[Fan et al. 2014]; $\approx$ 1600 other publications

Implemented and practical,
better in practice than Bloom

Storage: $(1 + o(1))n \log_2 \frac{1}{\varepsilon}$ bits, optimal!

Membership testing: $O(1)$ time
(with good locality of reference: works well with cache)

Can add and remove elements

Storage bound requires $\epsilon = o(1)$
bigger sets need to have smaller false positive rates

(Some sources exaggerate this requirement by saying that
"in theory, Cuckoo filters do not work")

Comparison of filters: Recent alternatives

Xor filters: [Graf and Lemire 2020]
Binary fuse filters: [Graf and Lemire 2022]

Fast, optimal storage for constant error rates, not dynamic

Quotient filters: [Pandey et al. 2017]
Morton filters: [Breslow and Jayasena 2020]
Vector quotient filters: [Pandey et al. 2021]

Similar design and performance to cuckoo filters

Quotient has least space; vector quotient is fastest

Bloom filters

Main idea of Bloom filters

Two parameters, N and k, to be chosen later

Store a table B of N bits, initially all zero

Construct k hash functions $h_1(x), \ldots h_k(x)$

To add x to the set, set its bits to one:

$$B[h_1(x)] = B[h_2(x)] = \cdots = B[h_k(x)] = 1$$

To test membership, check that all bits are one:

for $i = 1, 2, \ldots k$:
    if $B[h_i(x)] = 0$:
        return False
return True

B is just the bitmap representation of the set of hashes of elements!

Example of Bloom filter

Suppose $N = 9$ and $k = 3$ with hash functions mapping $a \to 0, 3, 4$; $b \to 1, 5, 7$; $c \to 2, 3, 5$; $d \to 1, 4, 8$; $e \to 0, 3, 5$

Initially $B = b_8 b_7 b_6\ b_5 b_4 b_3\ b_2 b_1 b_0 = 000\,000\,000$

Add a, setting bits $0, 3, 4$: $\quad B = 000\,011\,001$

Add b, setting bits $1, 5, 7$: $\quad B = 010\,111\,011$

Add c, setting bits $2, 3, 5$: $\quad B = 010\,111\,111$

Test membership for d: $b_1 = b_4 = 1$, $b_8 = 0 \Rightarrow$ return False

Test membership for e: $b_0 = b_3 = b_5 = 1 \Rightarrow$ return True
This is a false positive!

Bloom filter analysis

Let f be the fraction of bits that are one $\Rightarrow$

(by random hash assumption) false positive rate $\varepsilon = f^k$

Can't use Chernoff bound (bits are not independent of each other)
but related Azuma–Hoeffding inequality $\Rightarrow f \approx E[f]$

Linearity of expectation $\Rightarrow E[f] = \Pr[\text{any given bit is one}]$

$$\begin{aligned}
\Pr[\text{bit is 1}] &= 1 - \Pr[\text{same bit is 0}] \\
&= 1 - \Pr[\text{all hashes of elements miss that bit}] \\
&= 1 - \left(1 - \tfrac{1}{N}\right)^{kn} \\
&= 1 - \left(\left(1 - \tfrac{1}{N}\right)^{N}\right)^{kn/N} \\
&\approx 1 - \left(\tfrac{1}{e}\right)^{kn/N}
\end{aligned}$$

Bloom filter analysis (continued)

Simplifying assumptions: Suppose we already know N

Let's try plugging fractional values of k into the calculation
(even though in the actual data structure it must be an integer)

What choice of k gives the best false positive rate ε?

Turns out to be: k that makes fraction of ones be $f = 1/2$

(Can prove by calculus, but intuitive reason: because then the Bloom filter has the highest possible information content)

$$f = \frac{1}{2} \quad \Rightarrow \quad 1 - \left(\tfrac{1}{e}\right)^{kn/N} = \frac{1}{2} \quad \Rightarrow \quad N = \frac{kn}{\log 2}$$

With $f = 1/2$, $\varepsilon = 1/2^k$ giving $k = \log_2 \frac{1}{\varepsilon}$ and $N = \dfrac{n \log_2 1/\varepsilon}{\log 2}$

Bloom filter summary

For sets of size n, with desired false positive rate ε:

Choose number of hash functions $k \approx \log_2 \frac{1}{\varepsilon}$

Choose bit array size $N \approx \dfrac{n \log_2 1/\varepsilon}{\log 2} \approx 1.44 n \log_2 \dfrac{1}{\varepsilon}$

Store bitmap set of hashes of elements

Additions and membership tests take time $O(k)$,
which is $O(1)$ for $\varepsilon =$ constant

Can't remove any element because we don't know which of its bits are shared with other elements and which are used only by it

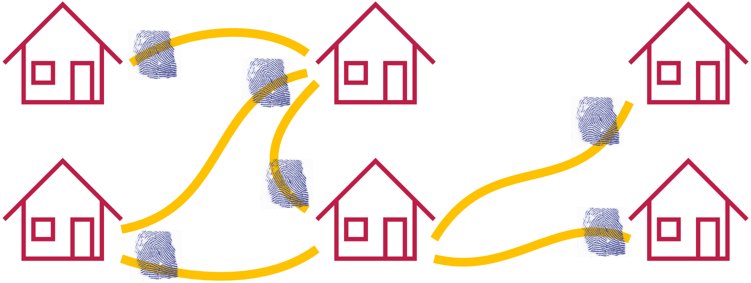
Cuckoo filters

Main idea

Use a hash function f to compute a short "fingerprint" $f(x)$ for each element x

Store fingerprints, not key-value pairs, in a cuckoo hash table

(each fingerprint can go in one of two possible home cells)



Saves space because fingerprints use fewer bits than full elements

Basic operations

Test if x is in set:
Check whether either of the two cells for x contains $f(x)$

False positive:
Some other element collides with x in both location and fingerprint

Insert x:
(Allowing > 1 fingerprint/cell to get load factor near one)

Add fingerprint $f(x)$ to home cell for x
If fingerprints overflow, insert recursively to second home cells

Delete x:
Remove fingerprint from one of its two homes

Difficulties

When we move a fingerprint $f(x)$ to its other cell,
we don't know which element x generated it
$\Rightarrow$ compute new cell using only current cell and $f(x)$

Fingerprints in any one cell can only go to a small number of other cells (as many as the number of different fingerprints)
$\Rightarrow$ the two cells for x cannot be chosen independently

Cuckoo hashing analysis depends on independence of pairs of cells
$\Rightarrow$ we need to prove that this works (all fingerprints can be inserted) all over again, without using independence

How to find the two homes for a fingerprint

Original version:

Choose three hash functions h_1, h_2, and f

Map each element x to fingerprint $f(x)$
with two homes $h_1(x)$ and $(h_1(x) \text{ xor } h_2(f(x)))$

When we see fingerprint f in cell with index i
its other home cell has index $(i \text{ xor } h_2(f))$

We don't need to know the x that generated it!

Works well in practice (up to same load factor as cuckoo hash)

No mathematical proof that it works!

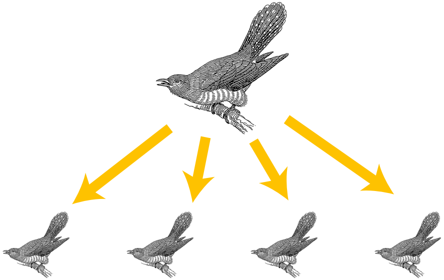
How to find the two homes for a fingerprint

Simplified version [Eppstein 2016]:

Choose two hash functions h_1 and f

Map x to fingerprint $f(x)$ with homes $h_1(x)$ and $(h_1(x) \text{ xor } f(x))$

Effectively partitions big cuckoo hash table into many smaller ones, within which pairs of home cells are chosen independently



Can reuse random-graph analysis from cuckoo hashing!

How much space do we need?

Assume k bits per fingerprint, then

$$\begin{aligned}
\Pr[\text{false positive}] &\le (\#\text{ elements that could collide}) \times \Pr[\text{collision}] \\
&= n \times \Pr[\text{same } h_1(x)] \times \Pr[\text{same } f(x)] \\
&= n \times O\left(\frac{1}{n}\right) \times \frac{1}{\#\text{ fingerprints}} \\
&= O\left(\frac{1}{2^k}\right).
\end{aligned}$$

Invert this: false positive rate ε needs $k = \log_2 \frac{1}{\varepsilon} + O(1)$

Insertion analysis needs k to be nonconstant ($\epsilon = o(1)$)
$\Rightarrow$ can replace $+O(1)$ in formula for k by $\times(1 + o(1))$

Cuckoo load factor near one $\Rightarrow$ multiply space by $(1 + o(1))$

So for false positive rate $\varepsilon = o(1)$, need $(1 + o(1))n \log_2 \frac{1}{\varepsilon}$ bits

Summary

Summary

- Set operations and their implementation in Python and Java
- How to combine sets using single-element operations
- Exact representations of sets using hash tables
- Exact representations of sets using bitmaps
- Filters: approximate representations of sets
- False positives versus false negatives
- Bloom filters and cuckoo filters
- Nonexistence of good data structures for disjointness

References and image credits, I

Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. doi: 10.1145/362686.362692.

Alex D. Breslow and Nuwan Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *VLDB Journal*, 29(2-3):731–754, 2020. doi: 10.1007/S00778-019-00561-0.

David Eppstein. Cuckoo filter: simplification and analysis. In Rasmus Pagh, editor, *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22–24, 2016, Reykjavik, Iceland*, volume 53 of *LIPIcs*, pages 8:1–8:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPICS.SWAT.2016.8.

Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: practically better than Bloom. In Aruna Seneviratne, Christophe Diot, Jim Kurose, Augustin Chaintreau, and Luigi Rizzo, editors, *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2–5, 2014*, pages 75–88, 2014. doi: 10.1145/2674005.2674994. URL `https://www.eecs.harvard.edu/~michaelm/postscripts/cuckoo-conext2014.pdf`.

Thomas Mueller Graf and Daniel Lemire. Xor filters: faster and Smaller Than Bloom and cuckoo filters. *ACM Journal of Experimental Algorithmics*, 25:1–16, 2020. doi: 10.1145/3376122.

References and image credits, II

Thomas Mueller Graf and Daniel Lemire. Binary fuse filters: fast and smaller than xor filters. *ACM Journal of Experimental Algorithmics*, 27:1.5:1–1.5:15, 2022. doi: 10.1145/3510449.

Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: making every bit count. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14–19, 2017*, pages 775–787, 2017. doi: 10.1145/3035918.3035963.

Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender, Martin Farach-Colton, and Rob Johnson. Vector quotient filters: overcoming the time/space trade-off in filter design. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20–25, 2021*, pages 1386–1399. ACM, 2021. doi: 10.1145/3448016.3452841.