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 $N = \text{max element}$

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 $\varepsilon$ 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, CACM 1970; \( \approx \) 25k other publications

Widely implemented, practical

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

Membership testing: \( O(1/\varepsilon) \) time

Can add but not remove elements
Comparison of filters: Cuckoo filter

Fan et al, CoNEXT ’14; ≈ 500 other publications

Implemented and practical,
better in practice than Bloom

Storage: \((1 + o(1))n \log_2 \frac{1}{\epsilon}\) 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: Xor filter

Graf and Lemire, JEA 2020, only one publication
For details, see https://r-libre.teluq.ca/1857/

Implemented and practical,
better in practice than Bloom
often better than cuckoo

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

Membership testing: \(O(1)\) time

Can handle constant error rates, unlike cuckoo

Cannot handle additions or removals
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 $a \rightarrow 0, 3, 4; b \rightarrow 1, 5, 7; c \rightarrow 2, 3, 5; d \rightarrow 1, 4, 8; e \rightarrow 0, 3, 5$

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 $0, 3, 4$: $B = 000 011 001$

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

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

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}]

\[
Pr[\text{bit is 1}] = 1 - Pr[\text{same bit is 0}]
= 1 - Pr[\text{all hashes of elements miss that bit}]
= 1 - \left(1 - \frac{1}{N}\right)^{kn}
= 1 - \left(\left(1 - \frac{1}{N}\right)^N\right)^{kn/N}
\approx 1 - \left(\frac{1}{e}\right)^{kn/N}
\]
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 $\varepsilon$?

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)

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

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

For sets of size \( n \), with desired false positive rate \( \varepsilon \):

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

Choose bit array size \( N \approx \frac{n \log_2 \frac{1}{\varepsilon}}{\log 2} \approx 1.44n \log_2 \frac{1}{\varepsilon} \)

Store bitmap set of hashes of elements

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

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 $(h_1(x) \ XOR \ h_2(f(x)))$

When we see fingerprint $f$ in cell with index $i$
it's other home cell has index $(i \ XOR \ h_2(f))$

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, SWAT 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

$$\Pr[\text{false positive}] \leq (\# \text{ elements that could collide}) \times \Pr[\text{collision}]$$

$$= n \times \Pr[\text{same } h_1(x)] \times \Pr[\text{same } f(x)]$$

$$= n \times O\left(\frac{1}{n}\right) \times \frac{1}{\# \text{ fingerprints}}$$

$$= O\left(\frac{1}{2^k}\right).$$

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

Insertion analysis needs $k$ to be nonconstant ($\varepsilon = o(1)$)

$\Rightarrow$ 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