

Bloom and Cuckoo Filters

9/29/26

Kahbod Aeini,
CS 848



“Have I seen x before?”

A set S of n items. One operation: $\text{contains}(x)$.

Exact set

- stores every key: $n \times \text{key size}$
- 1M URLs at 80 bytes $\approx$ 80 MB
- or keep it on disk $\rightarrow$ one I/O per question

always right

Filter

- stores no keys at all
- same 1M URLs $\approx$ 1.2 MB
- answers from memory, ~ 100 ns

occasionally wrong — in one direction only

The question is asked far more often than the answer is yes.

Approximate Membership Query

Methods: `insert(x)` · `contains(x)` · `sometimes delete(x)`

	says “probably in S”	says “definitely not”
x IS in S	correct	NEVER HAPPENS
x is NOT in S	false positive — probability ϵ	correct

One-sided error. No false negatives, ever.

Five things to compare

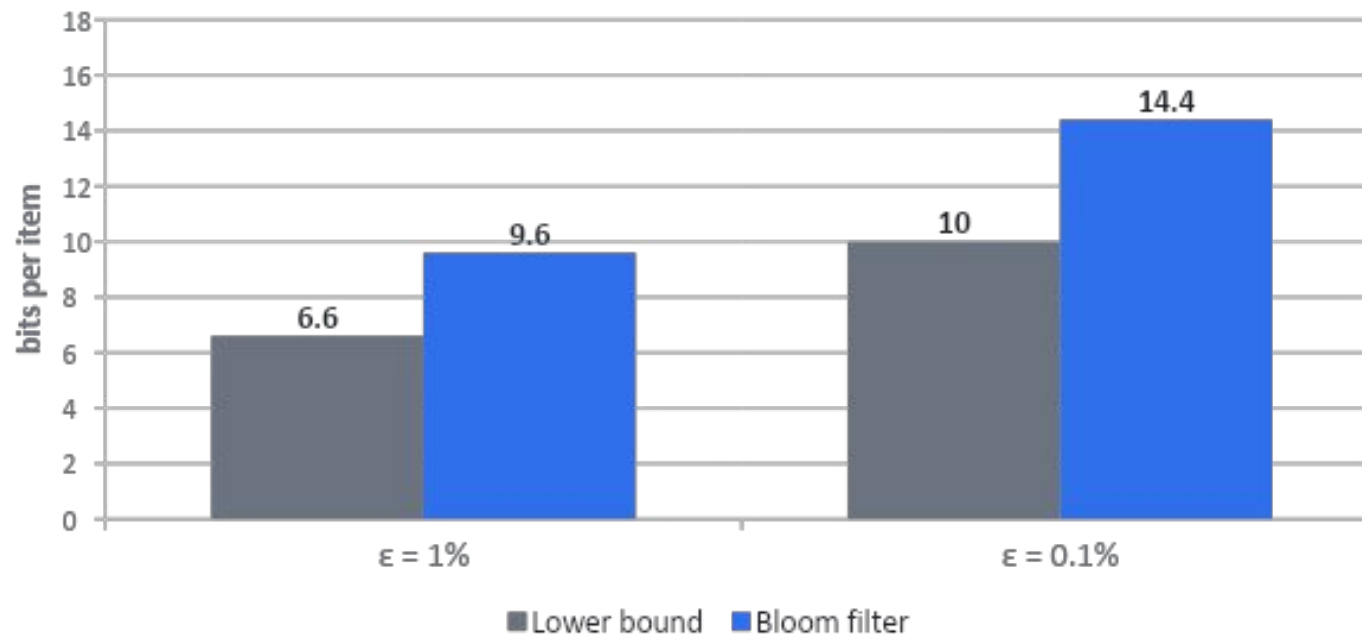
the rest of the talk is these five

- 1 ϵ false positive rate
- 2 bits / item space, at a given ϵ
- 3 lookup cost random memory accesses (cache misses), not instructions
- 4 delete supported, or not
- 5 when full does it degrade, or refuse?

How small can any filter be?

$\log_2(1/\epsilon)$ bits per item

information-theoretic bound, Carter et al. 1978



1.44 ×

A Bloom filter always costs 1.44 × the bound — at every ϵ .

Where these actually run

1 LSM-trees

RocksDB · Cassandra · HBase

A `get(key)` may have to check many SST files. One filter per file answers “not here” in ~ 100 ns and skips a ~ 100 μ s SSD read.

Filters are why an LSM read is not $O(\text{\#files})$ I/Os.

2 CDN caches

Akamai

~ 75 % of objects are requested exactly once. Admit to cache only on the second request — the filter remembers “seen once” without storing URLs.

Cuts disk writes dramatically.

3 Safe Browsing

Chrome

Millions of malicious URLs, checked on every navigation, locally. A false positive costs one server round trip.

Missing a bad URL is not an option.

Roadmap

- 1 Bloom filter** how it works · live demo · what it can't do
- 2 Bloom trade-offs** why m , k and n interact
- 3 Cuckoo hashing → Cuckoo filter** the partial-key trick · live demo
- 4 Cuckoo trade-offs** bucket size, fingerprint size, load
- 5 Head to head** when is “better than Bloom” actually true?

The Bloom filter (1970)

Burton H. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors”, CACM 13(7), 1970

His actual problem: automatic hyphenation

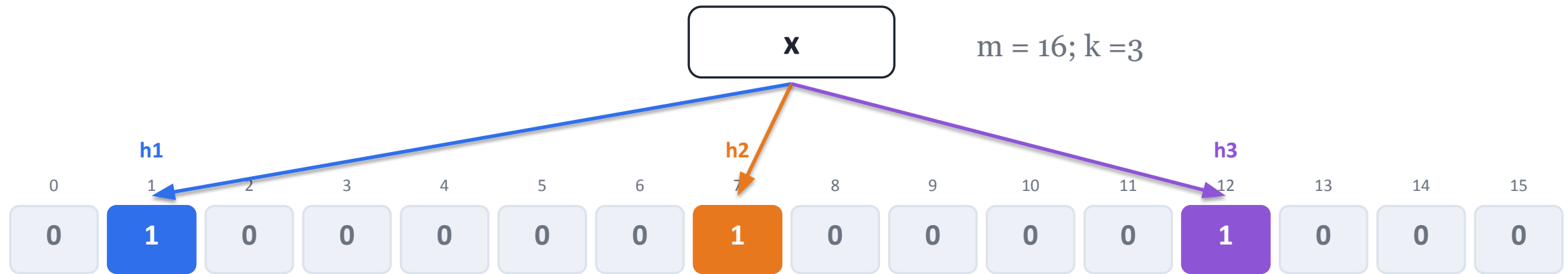
- 90 % of English words hyphenate by simple rules
- the other 10 % need a dictionary lookup — and the dictionary lived on disk
- checking the disk for every word was the bottleneck

Keep a small in-memory structure that answers “this word definitely follows the rules” for most words — and pay for a disk read only for the rest.

55 years old. Still in RocksDB, Cassandra, HBase, Chrome, Squid, Bitcoin.

An array of m bits. k hash functions.

That is the whole structure.



insert(x):

set the bits at $h_1(x), h_2(x), \dots, h_k(x)$ — all mod m

contains(x):

any bit is 0 → definitely not in S

all bits are 1 → probably in S

No keys are stored. Nothing is ever compared. The item is gone the moment you insert it.

Pseudocode

insert(x) :

```
for i in 0 .. k-1:
    bits[  $h_i(x) \bmod m$  ] = 1
```

contains(x) :

```
for i in 0 .. k-1:
    if bits[  $h_i(x) \bmod m$  ] == 0:
        return DEFINITELY_NOT
return PROBABLY_YES
```

Cost per operation

k hash computations

k random memory accesses

False Positive depends on

1. Number of buckets — m
2. Number of hash functions — k
3. Load factor — n/m

No delete.

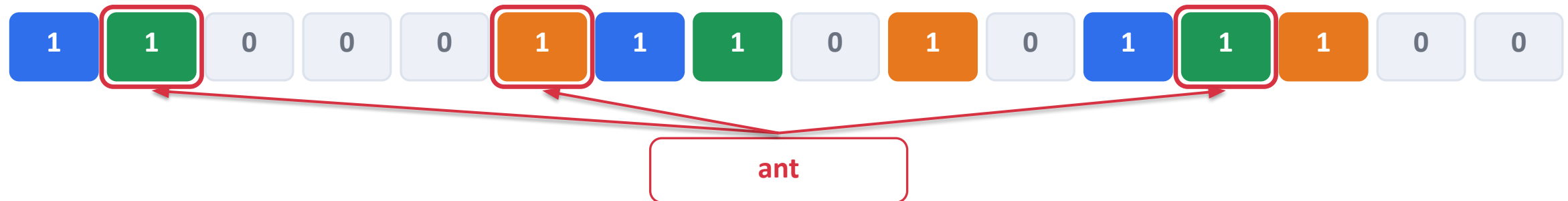
No way to determine *because of which input* a given bit is 1.
Clearing a bit would break every other item that shares it.

Worked example — real values, $m = 16$, $k = 3$

murmur3, seeds 0/1/2, mod 16



contains("cow") → bits 1, 6, 10 → bit 10 is 0 → definitely not present



"probably in S" — but "ant" was never inserted. Its three bits were set by three different words.

Let's see a live demo

Insert and Contains

False Positive Probability

Given a pair-wise independent hash-family $H = \{h_1, \dots, h_k\}$, after n insertions:

P(a given bit is still 0)

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}}$$

P(a given bit is 1)

$$1 - e^{-\frac{kn}{m}}$$

A false positive
i.e. all k queried bits are 1

$$\varepsilon \approx \left(1 - e^{-\frac{kn}{m}}\right)^k$$

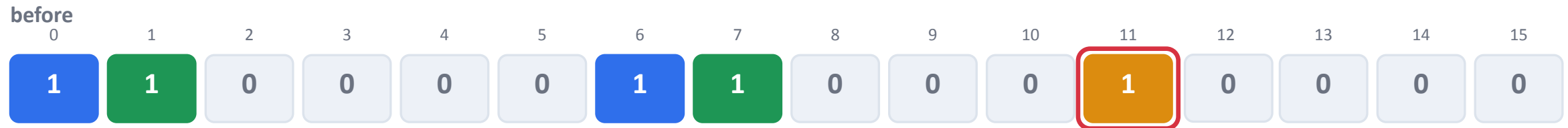
You cannot delete from a Bloom filter

Why not just clear the k bits?

dog → 0, 6, 11

hen → 1, 7, 11

they share bit 11



after deleting "dog" — bits 0, 6, 11 cleared



"hen" is now a false negative — the one guarantee is gone.

The filter never recorded that bit 11 had two owners. Real systems hit this constantly: cache eviction, a deleted row, an expired URL.

Counting Bloom filter: delete, at 4× the space

Fan, Cao, Almeida, Broder — “Summary Cache” (SIGCOMM 1998)

- insert → increment k counters
- delete → decrement k counters
- lookup → are all k counters > 0 ?

4 bits per counter is enough

at the optimal load each counter is Poisson($\ln 2$), so $P(\text{counter} \geq 16) \approx 7 \times 10^{-17}$

target ϵ	Bloom	Counting Bloom
1 %	9.6 bits/item	38.3 bits/item
0.1 %	14.4 bits/item	57.5 bits/item

4 ×

Deletion costs four times the memory.

This table is the Cuckoo filter's best argument.

Live demo

Delete in Bloom and Count Bloom

The rest of the limitations

Can't resize

m and k are fixed at build time. To grow you must re-insert every item — and you threw the items away.

Can't enumerate

You cannot ask a Bloom filter what is in it.

Can't count

No “how many times did I see x ?”

k random accesses = up to k cache misses

The bits are scattered by design. At $k = 7$, one lookup can touch 7 cache lines.

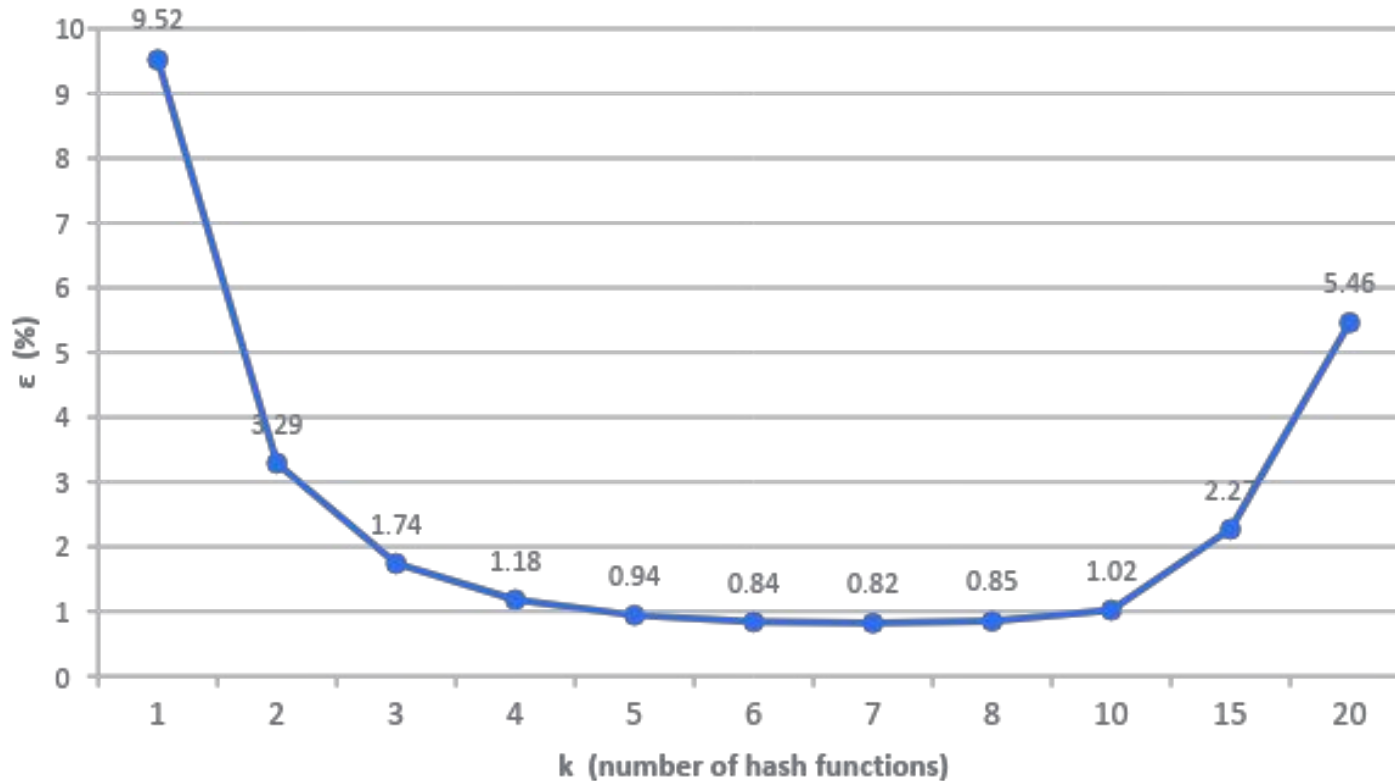
ϵ is fixed by the n you designed for

Insert more than n and it degrades.

Blocked Bloom filters (Putze et al. 2007) put all k bits in one cache line: ~ 1 miss, slightly worse ϵ .

Choosing k: more hashes is not better

at $m/n = 10$ bits per item - i.e. the filter is given 10 bits of memory for each item stored (1M items $\rightarrow$ 1.2 MB)



$$k^* = (m/n) \cdot \ln 2$$

$$\approx 0.693 \cdot m/n \rightarrow k = 7 \text{ here}$$

Too few hashes: too few bits to check.
Too many: you fill the array with 1s.

At the optimum, exactly half the bits are 1.

Deriving the space cost, line by line

m bits · k independent, uniform hash functions · n items inserted · write $c = n/m$

1 One hash write misses a given bit

$$\Pr[\text{one write misses bit } j] = 1 - \frac{1}{m}$$

2 n inserts perform kn independent writes

$$\Pr[\text{bit } j = 0] = \left(1 - \frac{1}{m}\right)^{kn}$$

3 Let m grow: $(1 - 1/m)^m \rightarrow e^{-1}$

$$\left(1 - \frac{1}{m}\right)^{kn} = \left[\left(1 - \frac{1}{m}\right)^m\right]^{kn/m} \approx e^{-kn/m} \equiv u$$

4 A false positive needs all k probed bits = 1

$$\varepsilon(k) = (1 - u)^k = (1 - e^{-kc})^k, \quad c = n/m$$

5 Reparametrise by u, the chance a bit is still 0

$$k = -\frac{\ln u}{c}, \quad \ln \varepsilon = k \ln(1 - u) = -\frac{\ln u \cdot \ln(1 - u)}{c} = -\frac{h(u)}{c}$$

6 h is symmetric, vanishes at both ends → interior max

$$h(u) = h(1 - u), \quad h'\left(\frac{1}{2}\right) = 0 \Rightarrow u^* = \frac{1}{2}, \quad h\left(\frac{1}{2}\right) = (\ln 2)^2$$

7 So half the bits are set, and k follows

$$e^{-k^*c} = \frac{1}{2} \Rightarrow k^* = \frac{m}{n} \ln 2$$

8 Substitute the optimum back into ε

$$\ln \varepsilon^* = -\frac{h(1/2)}{c} = -\frac{m}{n} (\ln 2)^2 \Rightarrow \varepsilon^* = e^{-\frac{m}{n} (\ln 2)^2} = 0.6185^{m/n}$$

Given a target error rate, how much memory and how many hashes?

$$\frac{m}{n} = \frac{\log_2(1/\varepsilon)}{\ln 2} = 1.4427 \log_2 \frac{1}{\varepsilon} \quad k^* = \log_2 \frac{1}{\varepsilon}$$

the 1.44 is exactly $1 / \ln 2$ — and it appears at the optimum, so no tuning removes it

Choosing m: bits per item buys accuracy

at the optimal k: $\epsilon = 0.6185^{(m/n)}$ i.e. $m/n = 1.44 \cdot \log_2(1/\epsilon)$

bits / item	k*	ϵ
4.8	3	10 %
9.6	7	1 %
14.4	10	0.1 %
19.2	13	0.01 %

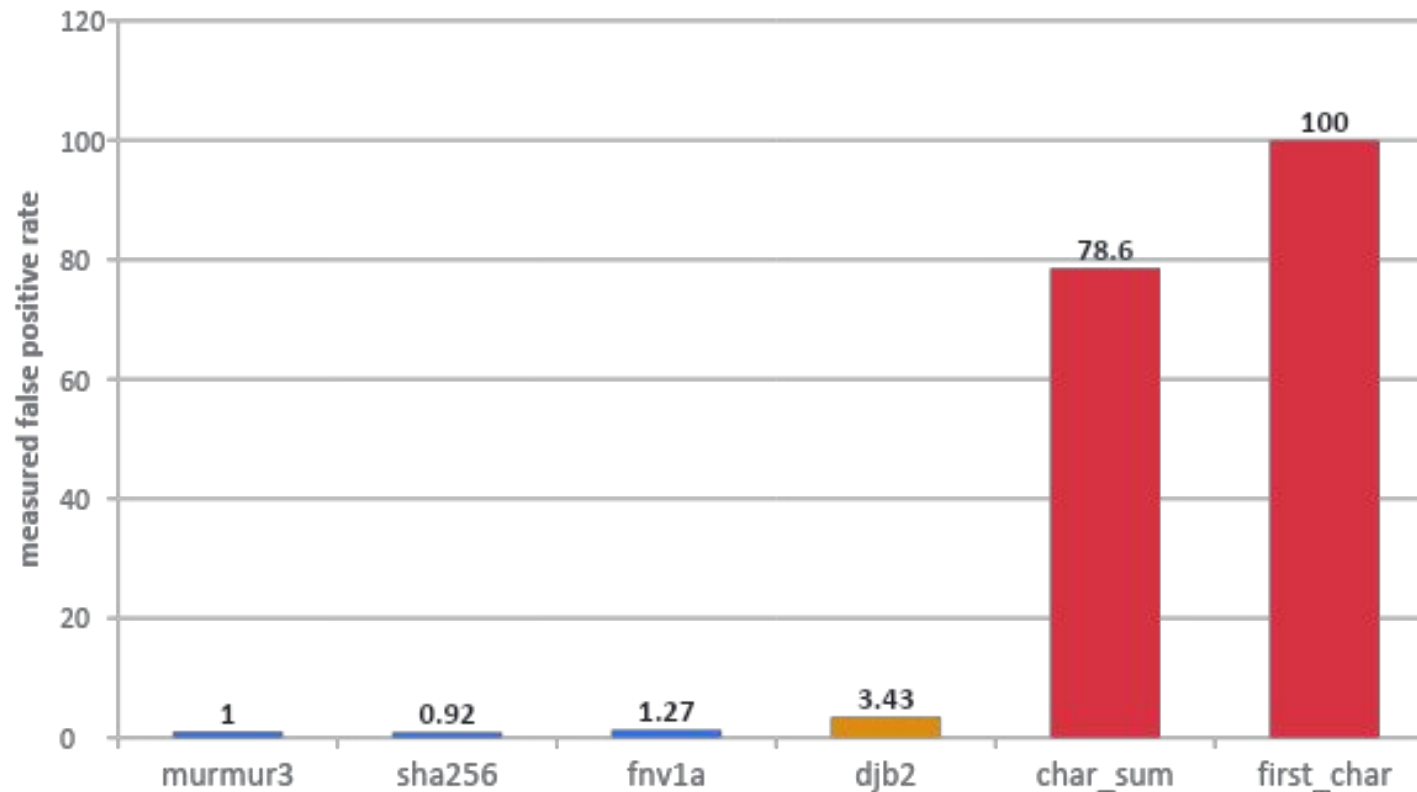
**Every extra 4.8 bits per item
divides ϵ by 10.**

- Independent of n — 1000 items or 10 billion, the per-item cost is the same
- Independent of key size — an 80-byte URL and a 4-byte integer both cost 9.6 bits at 1 %
- Always $1.44 \times$ the information-theoretic floor

This is why “10 bits per key” is the folklore default in RocksDB — it is the 1 % row.

The analysis assumes perfect hashes

1000 items · $m = 9586$ bits · $k = 7$ · theory says 1.00 % — measured with Bloom.py



**Same structure.
Same m , k , words.
Only the hash changed.**

Live

Ctrl+H walks the hash down to char_sum. Every word is re-inserted, and the measured line jumps far above the dashed theory line.

Ctrl+H back to murmur3 before moving on.

variant	fixes	how	space	cache misses	delete
Counting Bloom L. Fan, P. Cao, J. Almeida, and A. Z. Broder 1998	no delete	every bit becomes a 4-bit counter ; insert increments, delete decrements	3–4×	k	✓
d-left counting Bloom F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese 2006	counting Bloom's 4×	d-left hashing: store <i>fingerprints</i> in d sub-tables instead of counters; delete = remove a fingerprint	1.5–2×	d	✓
Blocked Bloom F. Putze, P. Sanders, and S. Johannes 2007	k scattered probes	the array is a row of small Bloom filters, each one cache line ; an item uses exactly one block	1×	1	✗
Quotient filter M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok 2012	delete <i>and</i> locality	compact hash table storing fingerprint quotients plus 3 metadata bits per slot; linear-probing-style clusters	1–1.2×	≥ 1	✓
Cuckoo filter L. Fan, D. Andersen, M. Kaminsky and M. Mitzenmacher 2014	all of the above	(next section)	$\leq 1\times$	2	✓

Cuckoo Filter

better than Bloom

Cuckoo hashing (Pagh & Rodler, 2001)

A hash table where every key has exactly two possible homes.

lookup(x)

check bucket $h_1(x)$, then $h_2(x)$. Two probes, worst case. Always.

insert(x)

either home free → done. Both full → evict an occupant and re-insert it into ITS other home.

1. both of x's homes are full

bucket 1

a

c

bucket 5

b

d

x wants bucket 1 or 5 — no room

2. x evicts a

bucket 1

x

c

bucket 5

b

d

a must move to its other home

3. a lands in a free slot

bucket 1

x

c

bucket 5

b

d

bucket 7

e

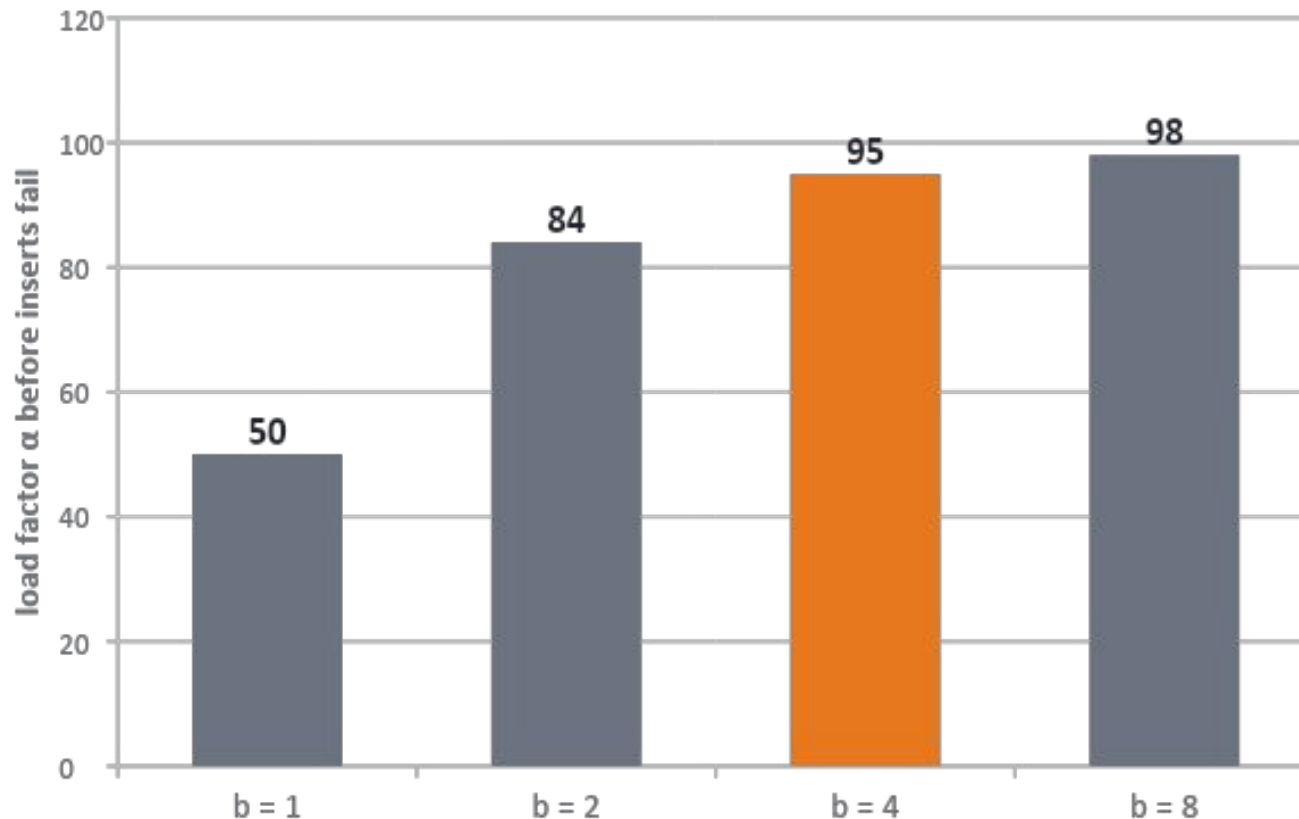
a

the chain ends

The name: the cuckoo chick hatches in another bird's nest and shoves the other eggs out.

Buckets of more than one item

Let each bucket hold b entries. Now an item has $2b$ places it could live.



Cuckoo filter default: $b = 4$

95 % occupancy — and the 4 entries sit in the same cache line, so it is still one memory access.

Why this matters

Plain cuckoo hashing with one item per bucket wastes half the table — which is why the original was never attractive for filters.

Keep $\alpha = 95$ %: it appears in every space formula from here on.

From hash table to filter: the problem

A cuckoo hash table stores keys. A filter must not.

So store a fingerprint instead: f bits of hash of x . i.e. a bit string derived from the item using a hash function

Lookup is fine

check the 2^b slots for a matching fingerprint — $fp(x)$

But insert is broken

To relocate an occupant you must compute ITS other bucket = $h_2(\text{key})$.

You don't have the key. You threw it away.

Reading the key back from disk on every eviction defeats the entire point.

bucket 4

231

16

250

113

which bucket does 231 belong to?



The idea: partial-key cuckoo hashing

$$i_1 = \text{hash}(x) \bmod m$$

$$i_2 = i_1 \text{ XOR } (\text{hash}(\text{fp}) \bmod m)$$

hash of the FINGERPRINT — not of the key

XOR is its own inverse

$$i_2 = i_1 \text{ XOR } \text{hash}(\text{fp}) \quad 7 \text{ XOR } 6 = 1$$

$$i_1 = i_2 \text{ XOR } \text{hash}(\text{fp}) \quad 1 \text{ XOR } 6 = 7$$

**Relocation needs no key.
The filter is self-sufficient.**

What it costs

- m must be a power of two, so the XOR stays in range
- i_1 and i_2 are no longer independent — the standard cuckoo hashing analysis doesn't directly apply
- a bucket has at most 2^f possible partners: too small an f and items can't spread, so the achievable load factor drops

Pseudocode

```
fingerprint(x):  f bits of hash(x)      # never 0: 0 = empty
```

lookup(x):

```
fp = fingerprint(x)
i1 = hash(x) mod m
i2 = i1 XOR (hash(fp) mod m)
return fp in bucket[i1] or fp in bucket[i2]
```

delete(x): # same start, then:

```
remove one copy of fp from bucket[i1] or bucket[i2]
```

insert(x):

```
if bucket[i1] or bucket[i2] has a free slot:
    put fp there; return OK
i = random(i1, i2)          # both full: start kicking
repeat MaxKicks (500) times:
    swap fp with a random entry of bucket[i]
    i = i XOR (hash(fp) mod m) # evicted one's other bucket
    if bucket[i] has a free slot:
        put fp there; return OK
return TABLE_FULL
```

2

memory accesses per lookup or delete —
regardless of ϵ

Same trick, twice

The purple lines are the same expression: once to find the second bucket, once to re-home an evicted fingerprint.

Delete is three lines

...because the fingerprint identifies the entry.
Compare with the Bloom filter, where it is impossible.

Live demo

Cuckoo Filter

The three knobs



slots per bucket

sets the achievable load factor α ($b = 4 \rightarrow 95\%$)



fingerprint bits

sets the false positive rate



how full you let it get

sets the current space cost

$$\text{space per item} = f / \alpha \text{ bits}$$

Bigger b packs the table better — but needs a longer fingerprint for the same ϵ .

Fingerprint size sets ϵ

A lookup compares against $2b$ stored fingerprints, each a random f -bit value:

$$\epsilon \leq 2b / 2^f$$

→

$$f \geq \log_2(2b / \epsilon)$$

target ϵ	f needed	space = $f / 0.95$
1 %	10 bits	10.5 bits/item
0.1 %	13 bits	13.7 bits/item
0.01 %	17 bits	17.9 bits/item

Why b cuts both ways

Doubling b ($4 \rightarrow 8$) lifts α from 95 % to 98 % — but costs one more bit of fingerprint for the same ϵ .

$b = 4$ is where it balances.

In a Bloom filter ϵ depends on how full the array is, so it degrades as you insert. Here ϵ is fixed by f the moment you choose it — inserting more only brings you closer to insert failure.

The load factor wall

Bloom filter, past capacity

keeps accepting

ϵ climbs silently toward 100 %

Cuckoo filter, when full

ϵ stays put

inserts start to fail — at $b = 4$, around 95 % load the chain hits $\text{MaxKicks} = 500$

Which failure do you prefer — a structure that lies more and more, or one that refuses your data?

LIVE — Ctrl+A

“The cuckoo filter is full at $n = 32$ (100 % load) with FPR 3.5 %. The Bloom filter never refuses an insert, but the FPR is now 90 %.”

The cuckoo filter's knobs, derived

m buckets · b entries per bucket · f -bit fingerprints · n items · $\alpha = n/(mb)$ · C = bits per item

- 1 Each of the $\leq 2b$ stored fingerprints matches with prob. 2^{-f}

$$\varepsilon = 1 - (1 - 2^{-f})^{2b} \approx \frac{2b}{2^f}$$

- 2 Demand a target ε and solve for the fingerprint size

$$f \geq \log_2 \frac{2b}{\varepsilon} = \log_2 \frac{1}{\varepsilon} + \log_2 2b$$

- 3 Space per item = table bits ÷ items stored

$$C = \frac{\text{bits}}{\text{items}} = \frac{f m b}{\alpha m b} = \frac{f}{\alpha}$$

α = the load reached before inserts fail ($b = 4 \rightarrow 95\%$)

- 4 Substitute f into C

$$C \leq \frac{\log_2 (1/\varepsilon) + \log_2 2b}{\alpha(b)}$$

- 5 Read it as: the information bound, plus an overhead

$$\log_2 \frac{1}{\varepsilon} + \log_2 2b \quad \text{i.e.} \quad \text{bound} + 3 \text{ bits when } b = 4$$

At load α only $2b\alpha$ slots are occupied, so the honest figure is

$$\varepsilon \approx \frac{2b\alpha}{2^f} \quad \text{— using } 2b \text{ (a full table) is the paper's conservative choice.}$$

Fix two, the rest follow:

$$f = \left\lceil \log_2 \frac{2b}{\varepsilon} \right\rceil \quad \alpha = \alpha(b) \quad C = \frac{f}{\alpha} \quad \varepsilon = \frac{2b}{2^{\alpha C}}$$

$$C_{\text{cuckoo}} \approx \frac{\log_2 (1/\varepsilon) + 3}{0.95} \quad \text{vs.} \quad C_{\text{Bloom}} = 1.44 \log_2 \frac{1}{\varepsilon}$$

the +3 is fixed, Bloom's 44 % grows with $\log_2(1/\varepsilon)$ — which is why the winner depends on ε

Semi-sorting: one bit back, at a price

The idea

Order inside a bucket doesn't matter.

A lookup only asks “is fp in this bucket?” — so [231, 16, 250, 113] and [16, 113, 231, 250] are the same bucket.

So store the sorted multiset, not the sequence.

With $b = 4$ and $f = 4$ (empty slots count as fingerprint 0), how many distinct sorted buckets are there?

$$\binom{16+4-1}{4} = \binom{19}{4} = 3876 < 4096 = 2^{12}$$

$4 \times 4 = 16$ bits $\rightarrow$ 12 bits i.e. 1 bit saved per fingerprint

Precompute those 3876 buckets once; store a 12-bit index into that table instead of the bucket itself.

The price

	CF	ss-CF
bits per item	12.60	12.58
false positive rate	0.19 %	0.09 %
build (M keys/s)	5.00	3.13
lookup (MOPS)	≈ 11	≈ 6
delete (MOPS)	≈ 9	≈ 4.5

Table 3 · Figures 5 and 8

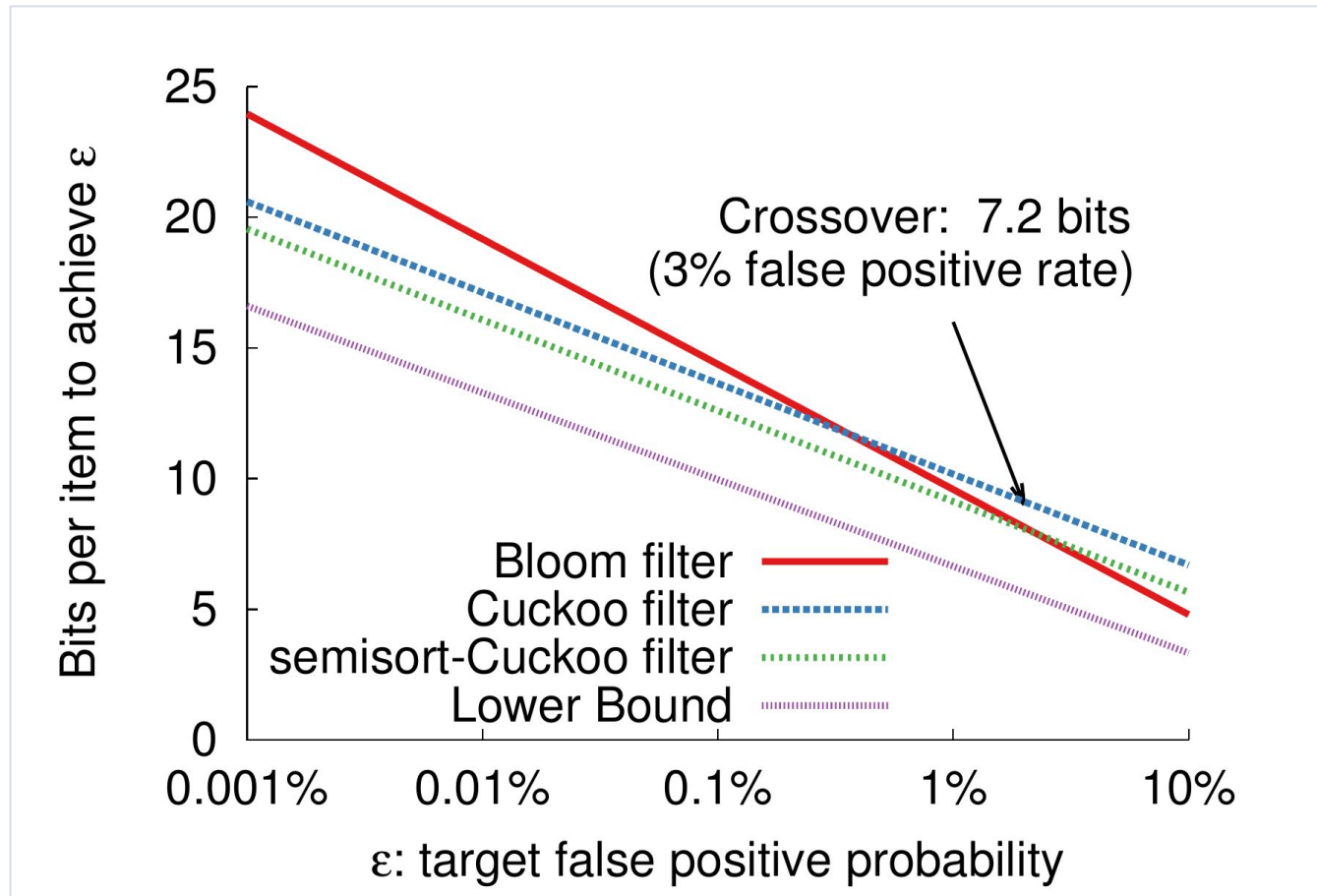
The saved bit is spent on a longer fingerprint: same space, half the false positive rate.

Only for $b = 4$ · with $f > 4$ only the top 4 bits are encoded, the rest stored directly · the encode/decode table must fit in cache

$$C_{\text{ss-CF}} \approx \frac{\log_2(1/\varepsilon) + 2}{\alpha} \quad \text{vs.} \quad C_{\text{CF}} \approx \frac{\log_2(1/\varepsilon) + 3}{\alpha}$$

this is the version that beats Bloom below $\varepsilon \approx 3\%$ — plain cuckoo only wins below $\approx 0.35\%$

The paper's own space comparison



Crossover: 7.2 bits per item

= ϵ of 3 %.

To the left (lower ϵ) the cuckoo filter wins.
To the right the 1970 Bloom filter is smaller.

Read the four lines

Lower Bound = $\log_2(1/\epsilon)$

Bloom = $1.44 \times$ that

Cuckoo = $(\log_2(1/\epsilon) + 3) / \alpha$

semisort-Cuckoo = one bit less

Speed: the argument that doesn't depend on ϵ

Bloom lookup

k

probes at k random locations $\rightarrow$ up to k cache misses ($k = 7$ at $\epsilon = 1\%$)

Cuckoo lookup

2

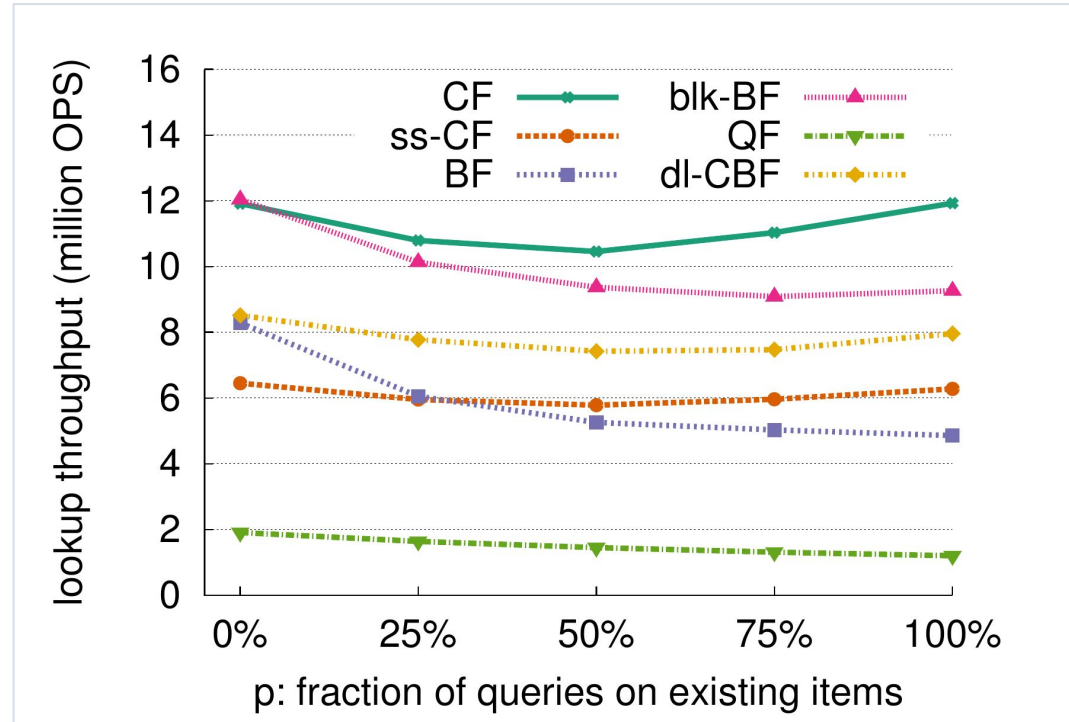
buckets, each contiguous $\rightarrow$ 2 cache misses, whatever ϵ is

Demand a lower ϵ and Bloom needs more probes. Cuckoo just needs a longer fingerprint — the number of memory accesses never changes.

But: they compare against a standard Bloom filter. A blocked Bloom filter also touches one cache line and is often faster than both.

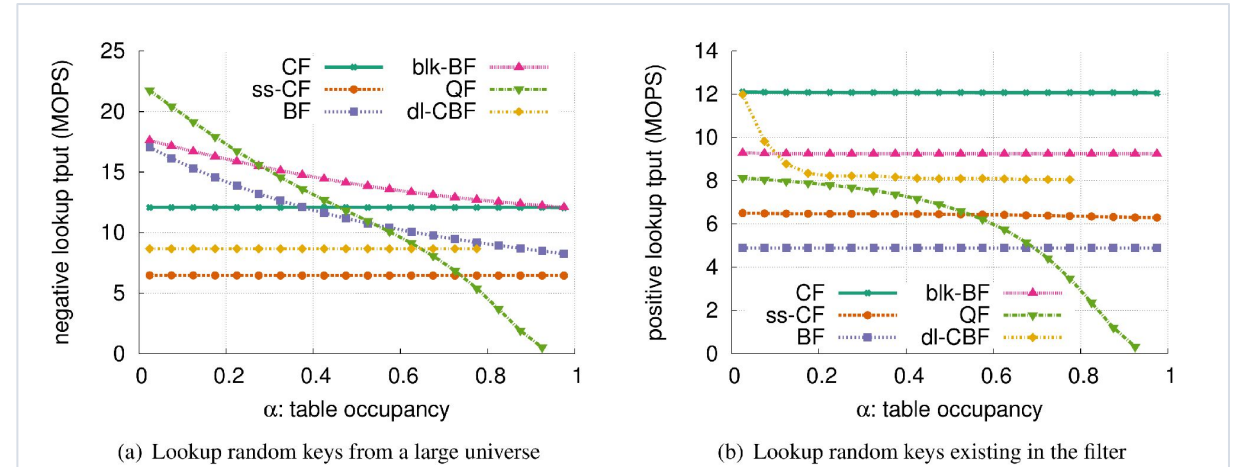
Measured lookup throughput

Figure 5 — filters full, varying the share of queries that hit



CF holds 10.4–12 MOPS across every workload mix.

Figure 6 — negative (left) and positive (right) lookups as the table fills



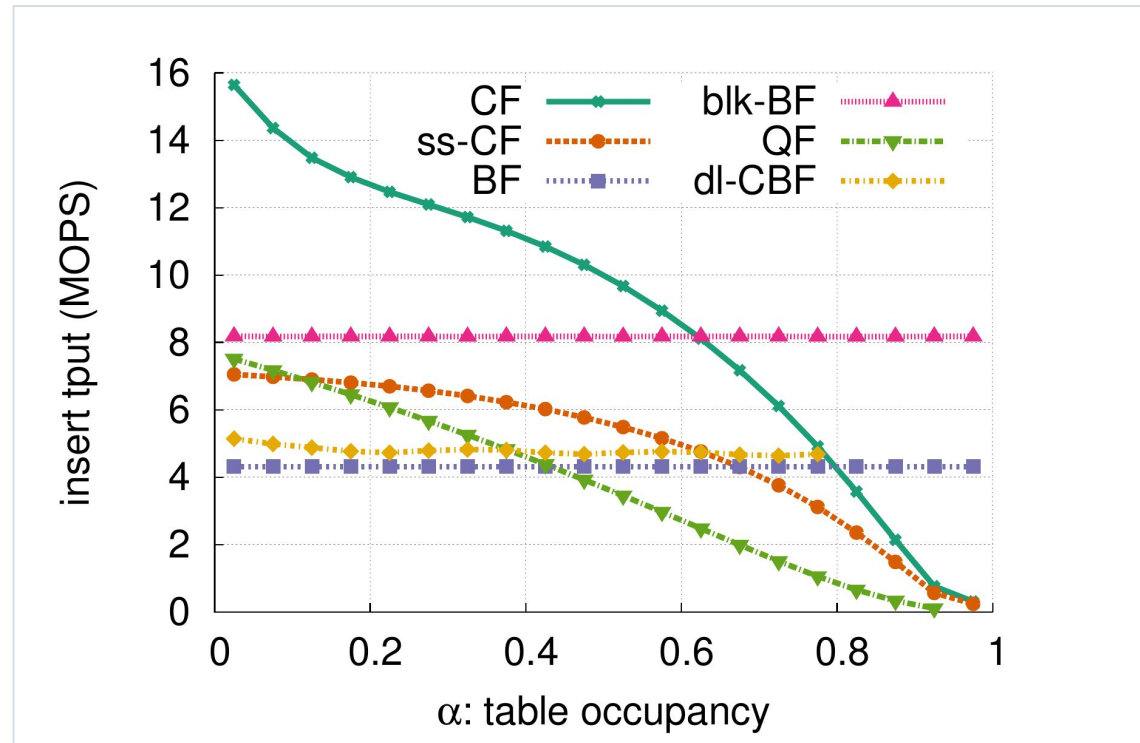
CF is the flat line.

Two buckets, always — so throughput barely moves with the workload mix or the load factor (the dip at $p = 50\%$ is branch misprediction).

BF drops as k bits must be read; QF collapses as its probe chains grow; ss-CF pays for decoding.

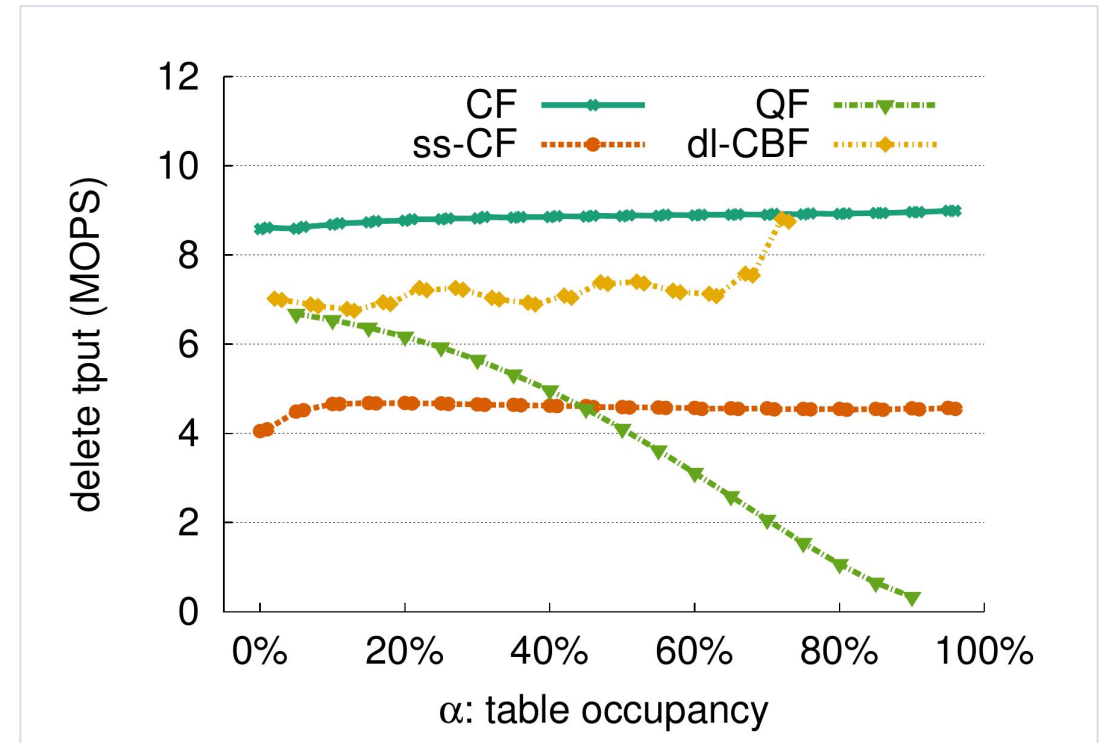
Where the cuckoo filter pays: inserts

Figure 7 — insert throughput as the table fills



15.6 → 0.5 MOPS as it fills. Kicking gets expensive; the Bloom filters are flat.

Figure 8 — delete-until-empty throughput



Deletes are the cuckoo filter's best case: highest and flat, ~9 MOPS.

The scorecard

the table from slide 4, completed

	Bloom	Counting Bloom	Cuckoo (b = 4)
bits/item @ $\epsilon = 1\%$	9.6	38.3	10.2 (9.1 semi-sorted)
bits/item @ $\epsilon = 0.1\%$	14.4	57.5	13.6 (12.6)
memory accesses / lookup	k (7 at 1 %)	k	2
delete	no	yes — with care	yes — with care
when full	ϵ climbs	ϵ climbs	insert refused
false negatives	never	possible after a bad delete	possible after a bad delete
resize	no	no	no

Need deletes, or ϵ below ~1 %? Cuckoo. Tolerate ϵ above 3 % and want simplicity? Bloom, still, after 55 years.

What people use in 2026



Still Bloom

RocksDB · Cassandra · HBase — blocked / partitioned variants, ~10 bits per key



Cuckoo

RedisBloom ships one; picked where deletes matter



Ribbon filters (2021)

RocksDB's newer option — closer to the space bound than Bloom, at more CPU



XOR & binary fuse filters (2020, 2022)

~1.08–1.23 × the bound and faster than both — but static: build once, no inserts, no deletes

The dividing line today is mutability, not Bloom vs Cuckoo: static set → XOR / ribbon · mutable set → Bloom or Cuckoo

Takeaways

- 1 A filter trades a bounded, one-sided error for a huge space saving. The error must be one-sided to be useful.
- 2 Bloom: k bits, k probes, no delete, $1.44 \times$ the space bound. Degrades silently when overfull.
- 3 Cuckoo: fingerprints in a cuckoo hash table. Delete works, lookups are always 2 probes.
- 4 The enabling trick is one line: $i2 = i1 \text{ XOR hash}(\text{fingerprint})$ — relocate without the key.
- 5 “Better than Bloom” is true below $\sim 3\%$ false positive rate, and the paper's own numbers say so.

References

Bloom, B. H.	Space/Time Trade-offs in Hash Coding with Allowable Errors. CACM 13(7), 1970.
Fan, L., Cao, P., Almeida, J., Broder, A.	Summary Cache. SIGCOMM 1998 / IEEE-ACM ToN 2000. (counting Bloom)
Pagh, R., Rodler, F.	Cuckoo Hashing. ESA 2001 / J. Algorithms 2004.
Putze, F., Sanders, P., Singler, J.	Cache-, Hash- and Space-Efficient Bloom Filters. WEA 2007. (blocked)
Kirsch, A., Mitzenmacher, M.	Less Hashing, Same Performance. ESA 2006.
Fan, B., Andersen, D. G., Kaminsky, M., Mitzenmacher, M. D.	Cuckoo Filter: Practically Better Than Bloom. CoNEXT 2014.
Graf, T. M., Lemire, D.	XOR Filters. ACM JEA, 2020.
Dillinger, P. C., Walzer, S.	Ribbon Filter. 2021.

Demo code: [Hash.py](#) · [Bloom.py](#) · [Cuckoo.py](#) · [Demo.py](#) — every number on these slides is computed by that code