Binary Fuse Filters: Fast and Tiny Immutable Filters

Daniel Lemire

professor, Data Science Research Center

Université du Québec (TÉLUQ)

Montreal

blog: https://lemire.me

twitter: @lemire

GitHub: https://github.com/lemire/

Probabilistic filters?

- Is x in the set S?

- Maybe or *definitively not*

Usage scenario?

- We have this expensive *database*. Querying it cost you.

- Most queries should not end up in the data.

- We want a small 'filter' that can prune out queries.

Theoretical bound

- Given N elements in the set

- Spend k bits per element

- Get a false positive rate of $1/2^k$

Usual constraints

- Fixed initial capacity

- Difficult to update safely without access to the set

- To get a 1% false-positive rate: ≈ 8 bits?

Hash function

- From any objet in the *universe* to a *word* (e.g., 64-bit word)

- Result looks random

```c
uint64_t murmur64(uint64_t h) {
  h ^= h >> 33;
  h *= UINT64_C(0xff51afd7ed558ccd);
  h ^= h >> 33;
  h *= UINT64_C(0xc4ceb9fe1a85ec53);
  h ^= h >> 33;
  return h;
}
```

Conventional Bloom filter

- Start with a bitset B.

- Using `k` hash functions $f_1, f_2, \ldots$

Adding an element

- Given an object x from the set, set up to `k` bits to 1
- $B[f_1(x)] \leftarrow 1, B[f_2(x)] \leftarrow 1, \ldots$

Checking an element

- Given an object x from the universe, set up to `k` bits to 1
- $(B[f_1(x)] = 1)$ AND $(B[f_2(x)] = 1)$ AND $\ldots$

Checking an element: implementation

- Typical implementation is *branchy*

- If not $(B[f_1(x)] = 1)$, return false

- If not $(B[f_2(x)] = 1)$, return false

- ...

- return true

```c
uint64_t hash = hasher(key);
uint64_t a = (hash >> 32) | (hash << 32);
uint64_t b = hash;
for (int i = 0; i < k; i++) {
  if ((data[reduce(a, length)] & getBit(a)) == 0) {
    return NotFound;
  }
  a += b;
}
return Found;
```
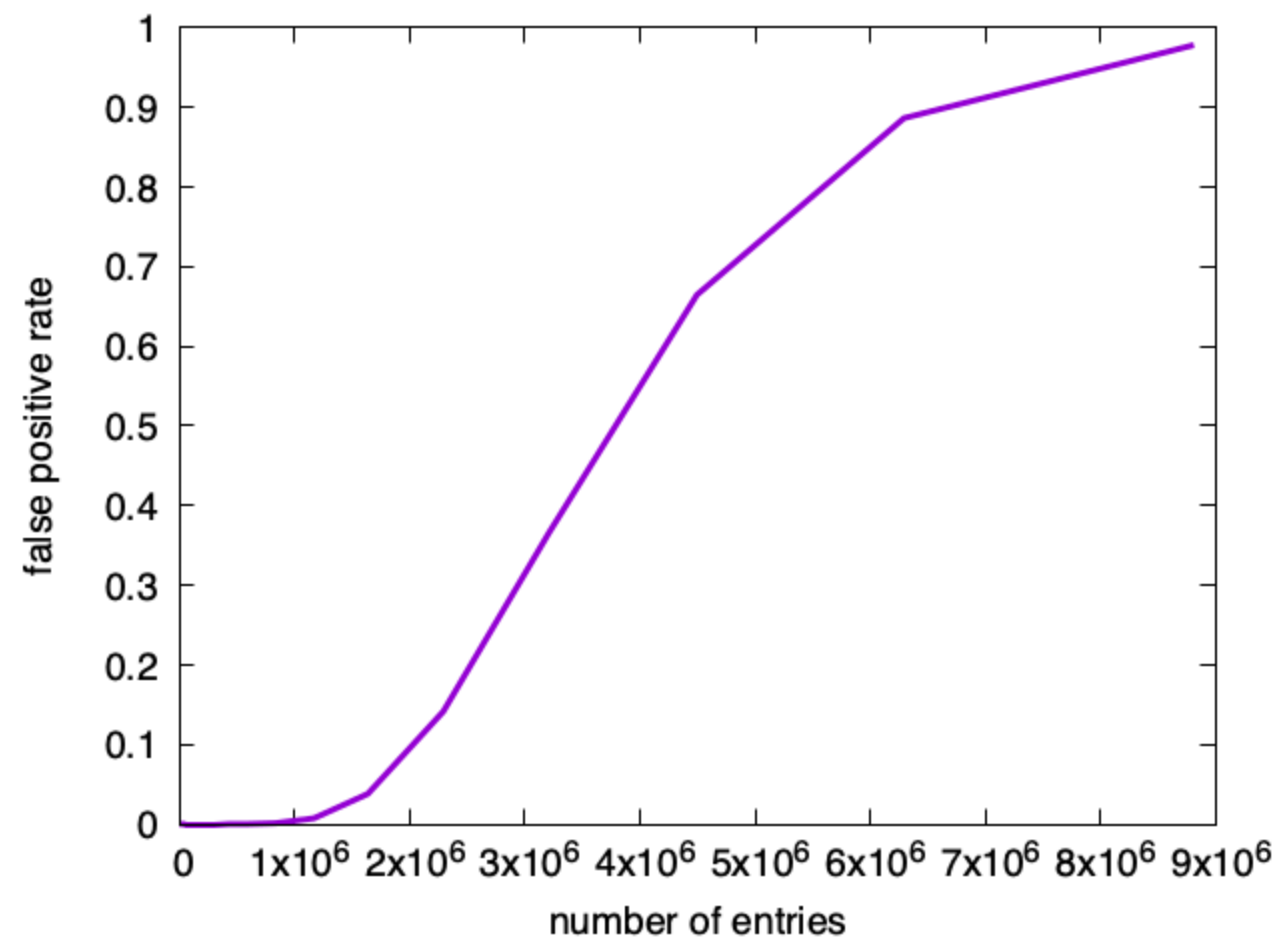
False positive rate

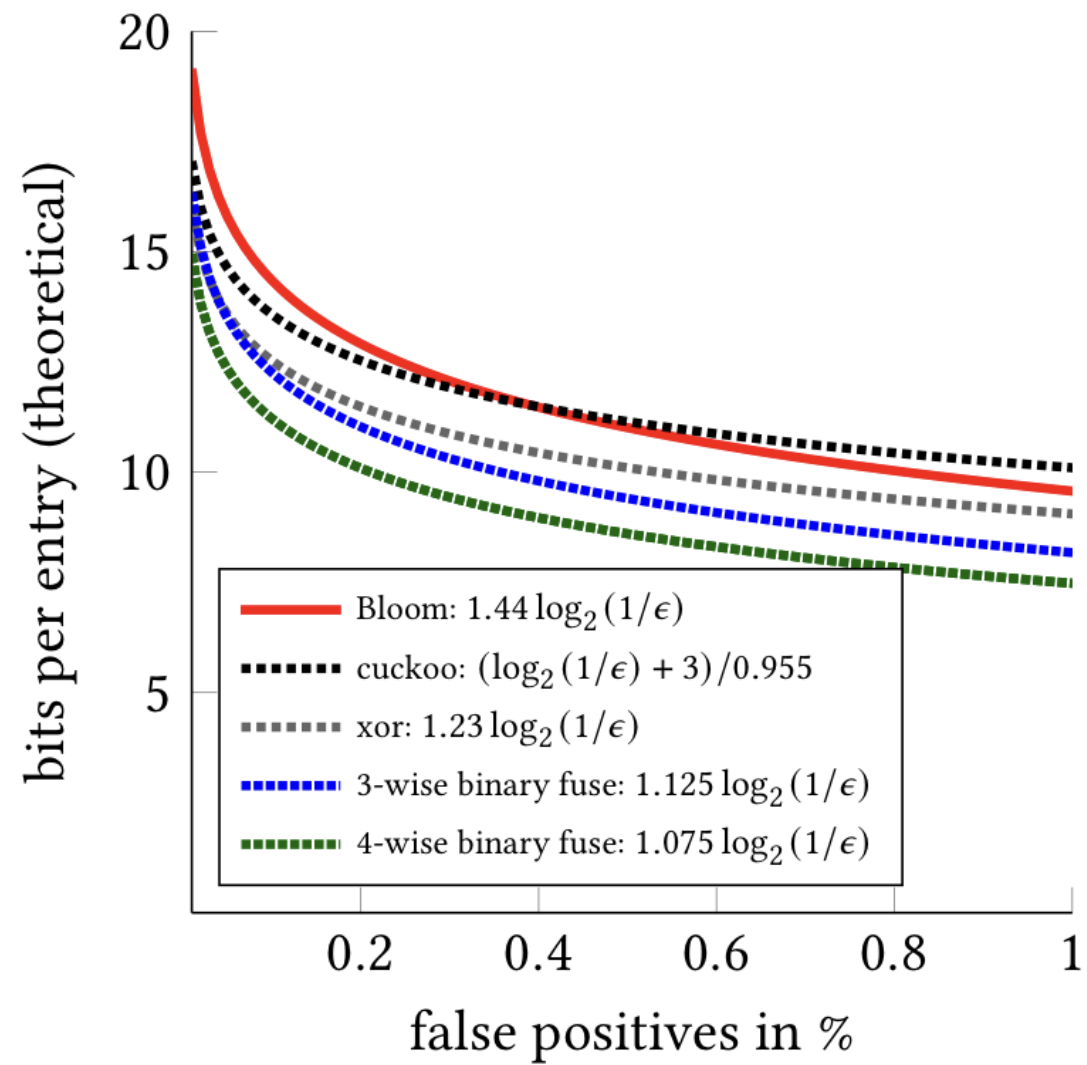| bits per element	hash functions	fpp
9	6	1.3%
10	7	0.8%
12	8	0.3%
13	9	0.2%
15	10	0.07%
16	11	0.04%

Bloom filters: upsides

- Fast construction

- Flexible: excess capacity translates into lower false positive rate

- Degrades smoothly to a useless but 'correct' filter
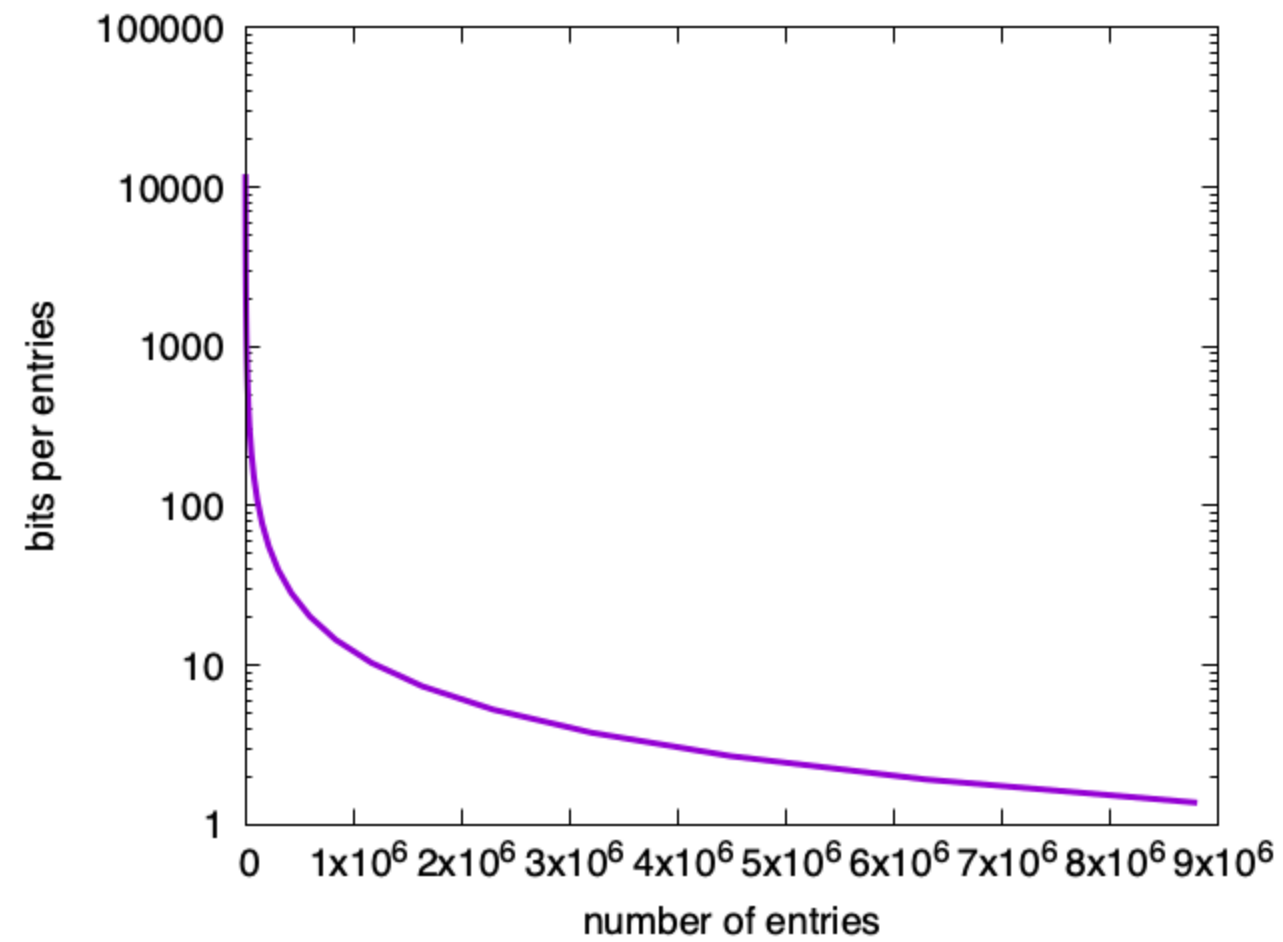
Bloom filters: downsides

- 44% above the theoretical minimum in storage

- Slower than alternatives (lots of memory accesses)

Legend:
- Bloom: $1.44 \log_2(1/\epsilon)$
- cuckoo: $(\log_2(1/\epsilon) + 3)/0.955$
- xor: $1.23 \log_2(1/\epsilon)$
- 3-wise binary fuse: $1.125 \log_2(1/\epsilon)$
- 4-wise binary fuse: $1.075 \log_2(1/\epsilon)$

x-axis: false positives in %

y-axis: bits per entry (theoretical)

Memory accesses

number of hash functions	cache misses (miss)	cache misses (hit)
8	3.5	7.5
11	3.8	10.5

(Intel Ice Lake processor, out-of-cache filter)

Mispredicted branches

number of hash functions	all out	all in
8	0.95	0.0
11	0.95	0.0

(Intel Ice Lake processor, out-of-cache filter)

Performance

number of hash functions	always out (cycles/entry)	always in (cycles/entry)
8	135	170
11	140	230

(Intel Ice Lake processor, out-of-cache filter)

Blocked Bloom filters

- Same as a Bloom filters, but for a given object, put all bits in one cache line

- Optional: Use SIMD instructions to reduce instruction count

Blocked Bloom filters: pros/cons

- Stupidly fast in both construction and queries

- ~56% above the theoretical minimum in storage

```cpp
auto hash = hasher_(key);
uint32_t bucket_idx = reduce(rotl64(hash, 32), bucketCount);
__m256i mask = MakeMask(hash);
__m256i bucket = directory[bucket_idx];
return _mm256_testc_si256(bucket, mask);
```

Binary fuse filters

- Based on theoretical work by Dietzfelbinger and Walzer

- Immutable datastructure: build it once

- Fill it to capacity

- Fast construction

- Fast and simple queries

Arity : 3-wise, 4-wise

- 3-wise version has three hits, 12% overhead

- 4-wise version has four hits, 8% overhead

Queries are silly

- Have an array of *fingerprints* (e.g., 8-bit words)

- Compute 3 (or 4) hash functions: $f_1(x), f_2(x), f_3(x)$

- Compute fingerprint function ($f(x) \to$ 8-bit word)

- Compute XOR and compare with fingerprint:
  $$(B[f_1(x)] = 1) \text{ XOR } (B[f_2(x)] = 1) \text{ XOR } (B[f_3(x)] = 1) = f(x)$$

```
bool contain(uint64_t key, const binary_fuse_t *filter) {
  uint64_t hash = mix_split(key, filter->Seed);
  uint8_t f = fingerprint(hash);
  binary_hashes_t hashes = hash_batch(hash, filter);
  f ^= filter->Fingerprints[hashes.h0] ^ filter->Fingerprints[hashes.h1] ^
       filter->Fingerprints[hashes.h2];
  return f == 0;
}
```

	cache misses	mispredictions
3-wise binary fuse	2.8	0.0
4-wise binary fuse	3.7	0.0

(Intel Ice Lake processor, out-of-cache filter)

	always out (cycles/entry)	always in (cycles/entry)	bits per entry
Bloom $k = 8$	135	170	12
3-wise bin. fuse	85	85	9.0
4-wise bin. fuse	100	100	8.6

(Intel Ice Lake processor, out-of-cache filter)
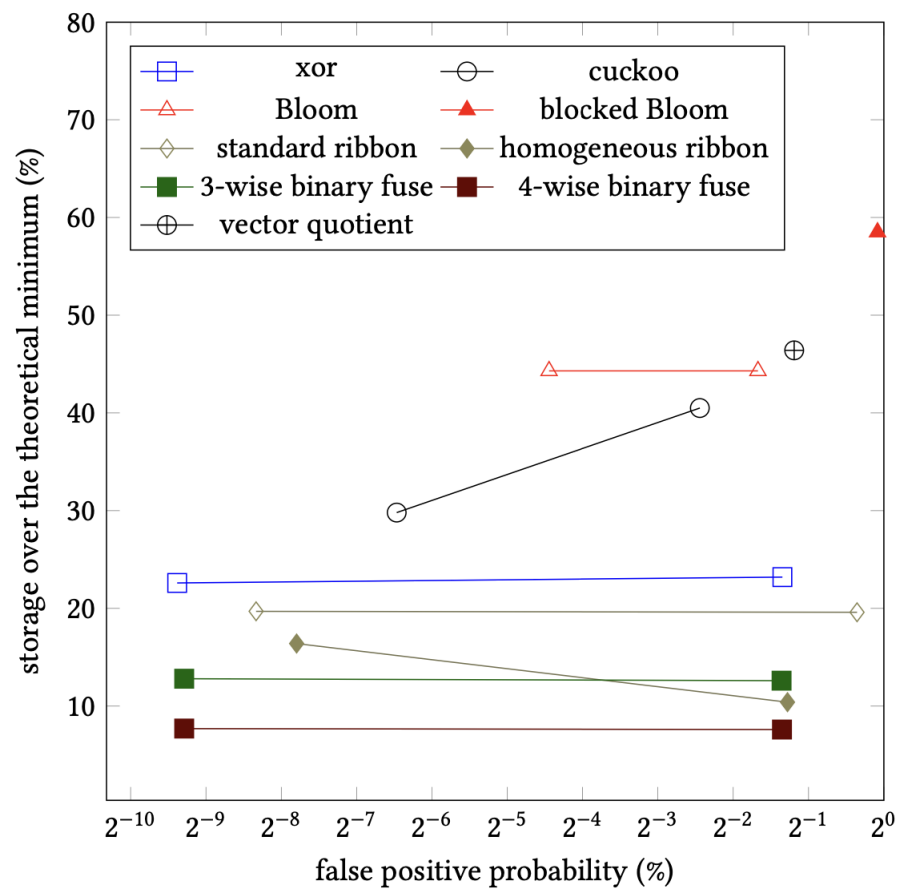
(a) Relative space usage
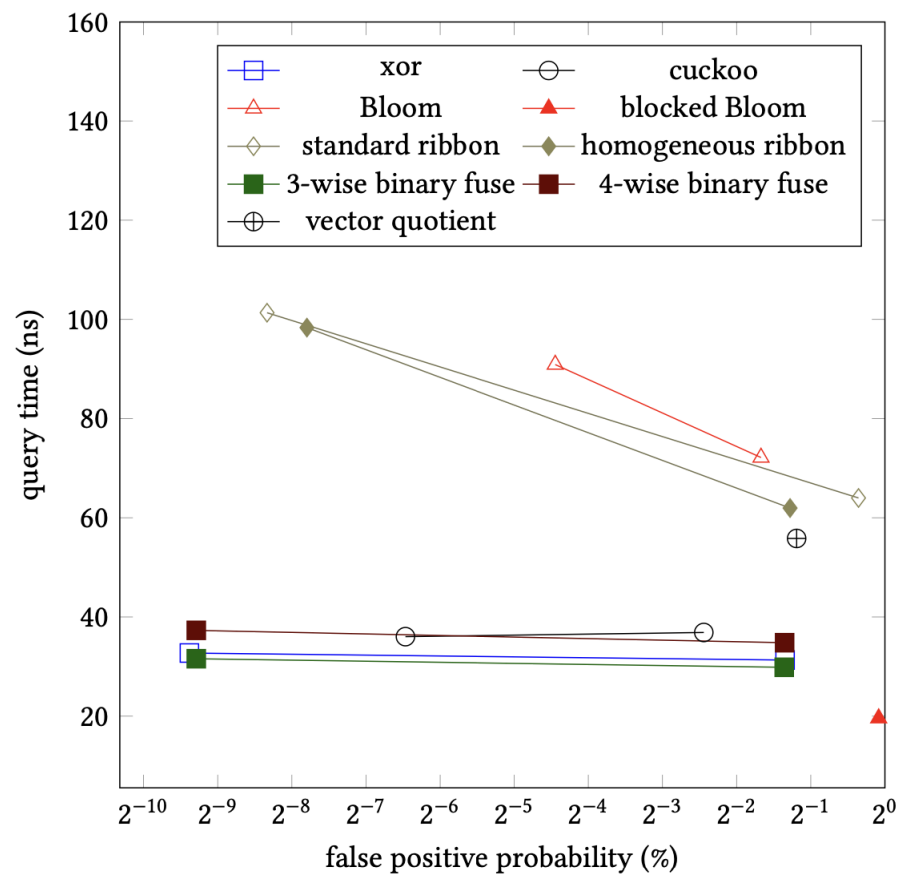
(b) Query time

Construction 1

- Start with array for fingerprints containing slightly more fingerprints than you have elements in the set

- Divide the array into segments (e.g., 300 disjoint)

- Number of fingerprints in segment: power of two (hence *binary*)

Construction 2

- Map each object x in set, to locations $B[f_1(x)], B[f_2(x)], B[f_3(x)]$
- The locations should be in three consecutive segments (so relatively nearby in memory).

Construction 3

- At the end, each location $B[i]$ is associated with some number of objects from the set

Construction 4

- Find a location mapped from a single set element x, e.g., $B[f_1(x)]$

- Record this location which is owned by x

- Remove the mapping of x to locations $B[f_1(x)], B[f_2(x)], B[f_3(x)]$

- Repeat

Construction 5

- Almost always, the construction terminates after one trial
- Go through the matched keys, in reverse order, adn set (e.,g.)
$$B[f_1(x)] = f(x) \text{ XOR } B[f_2(x)] \text{ XOR } B[f_3(x)]$$

Construction: Performance

- Implemented naively: terrible performance (random access!!!)

- Before the construction begins, sort the elements of the sets according to the segments they are mapped to.

- This greatly accelerates the construction
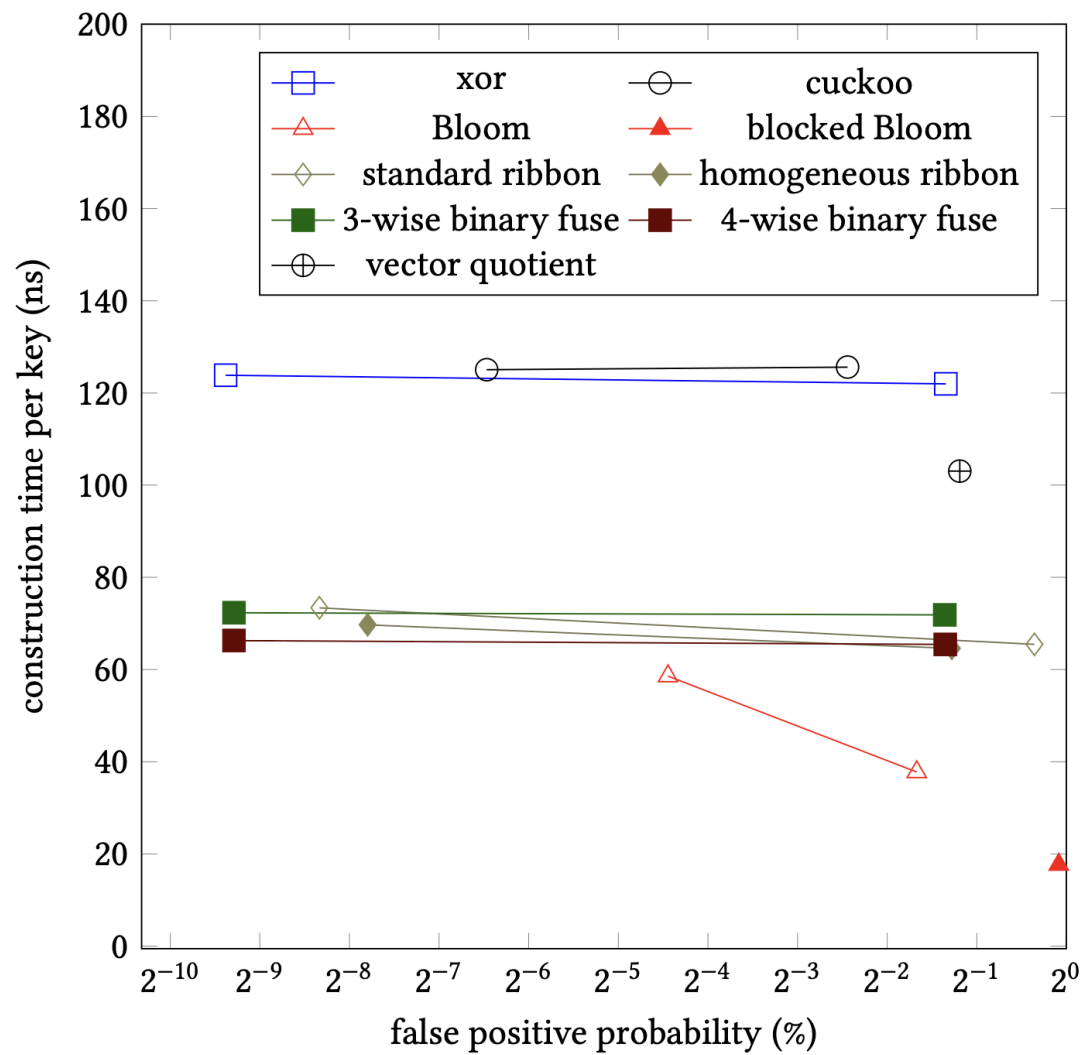
How does the performance scale with size?

For warm small filters, number of access is less important.
Becomes more computational.

For large cold filters, accesses are costly.

10M entries

	ns/query (all out)	ns/query (all in)	fpp	bits per entry
Bloom	17	14	0.32%	12.0
Blocked Bloom (NEON)	3.8	3.8	0.6%	12.8
3-wise bin. fuse	3.5	3.5	0.39%	9.0
4-wise bin. fuse	4.0	4.0	0.39%	8.6

(Apple M2)

100M entries

	ns/query (all out)	ns/query (all in)	fpp	bits per entry
Bloom	38	33	0.32%	12.0
Blocked Bloom (NEON)	11	11	0.6%	12.8
4-wise bin. fuse	17	17	0.39%	9.0
4-wise bin. fuse	20	20	0.39%	8.6

(Apple M2)

Compressibility (zstd)

	bits per entry (raw)	bits per entry (zstd)
Bloom $k = 8$	12.0	12.0
3-wise bin. fuse	9.0	8.59
4-wise bin. fuse	8.60	8.39
theory	8.0	8.0

Sending compressed filters

Compressed (zstd) binary fuse filters can be within 5% of the theoretical minimum.

Some links

- Bloom filters in Go: https://github.com/bits-and-blooms/bloom

- Binary fuse filters in Go: https://github.com/FastFilter/xorfilter

- Binary fuse filters in C: https://github.com/FastFilter/xor_singleheader

- Binary fuse filters in Java: https://github.com/FastFilter/fastfilter_java

- Giant benchmarking platform: https://github.com/FastFilter/fastfilter_cpp

Other Links

- Blog https://lemire.me/blog/
- Twitter: @lemire
- GitHub: https://github.com/lemire