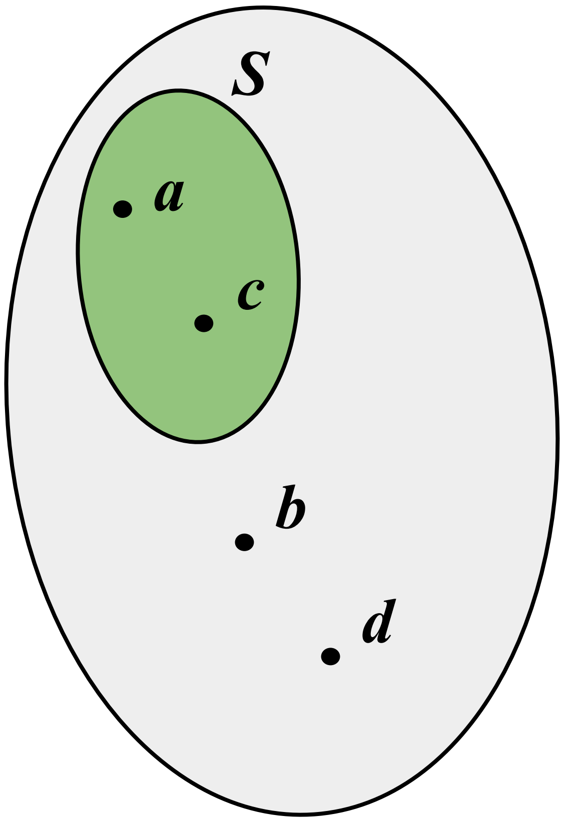
Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design

Prashant Pandey, Alex Conway, Joe Durie, Michael A. Bender,
Martin Farach-Colton, Rob Johnson

Filter data structure

A filter is an ***approximate*** representation of a set.



membership(a): ✓

membership(b): ✗

membership(c): ✓

membership(d): ✓ **false positive**

A filter supports ***<u>approximate</u>*** membership queries on S.

A filter guarantees a false-positive rate ε

if $q \in$ S, return ✓ with probability 1 **true positive**

if $q \notin$ S, return

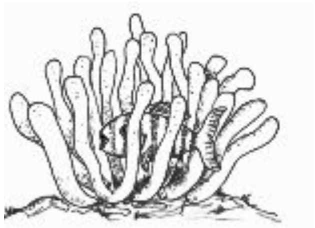
- ✗ with probability □1 - ε **true negative**
- ✓ with probability $\leq \varepsilon$ **false positive**

one-sided errors

False-positive rate enables filters to be compact

$$\text{space} \geq n \log(1/\epsilon)$$

$$\text{space} = \Omega(n \log |U|)$$



Filter

Dictionary

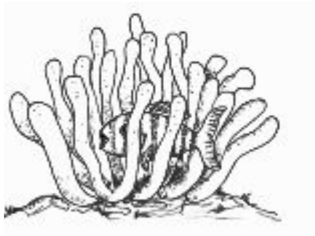
False-positive rate enables filters to be compact

$$\text{space} \geq n \log(1/\epsilon)$$

Small

$$\text{space} = \Omega(n \log |U|)$$

Large



Filter

Dictionary

For most practical purposes:

ε = 2%, a filter requires ≈ 8 bits/item

Filters are ubiquitous

Streaming applications

Networking

Databases

Computational biology

Storage systems

Filter design objectives

	Optimal
Space (bits)	$\approx n \ \log(1/\epsilon) + \Omega(n)$
CPU cost	$O(1)$
Data locality	$O(1)$ probes

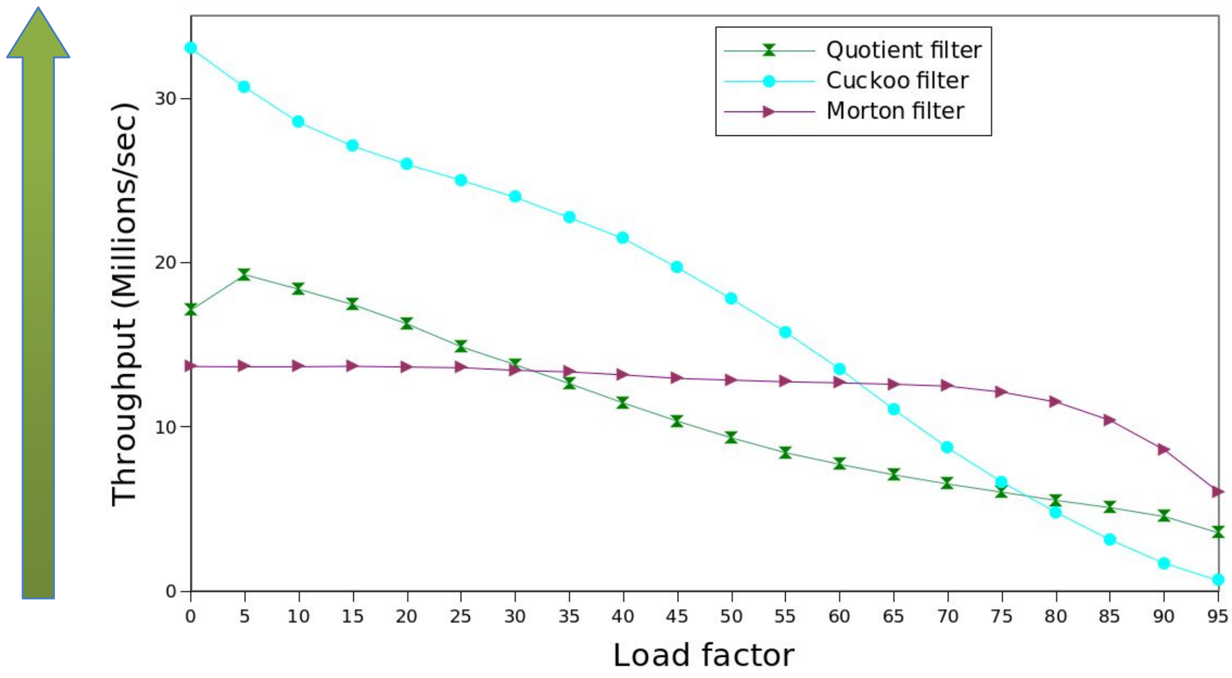
Types of filters

- Bloom filters [Bloom ‘70]
- Quotient filters [Pagh et al. ‘05, Dillinger et al. ‘09, Bender et al. ‘12, Einziger et al. ‘15, Pandey et al. ‘17]
- Cuckoo/Morton filters [Fan et al. ‘14, Breslow & Jayasena ‘18]
- Others
    - Mostly based on perfect hashing and/or linear algebra
    - Mostly static
    - e.g., Xor filters [Graf & Lemire ‘20]

State of the art in practical dynamic filters.

Current filters have a problem..

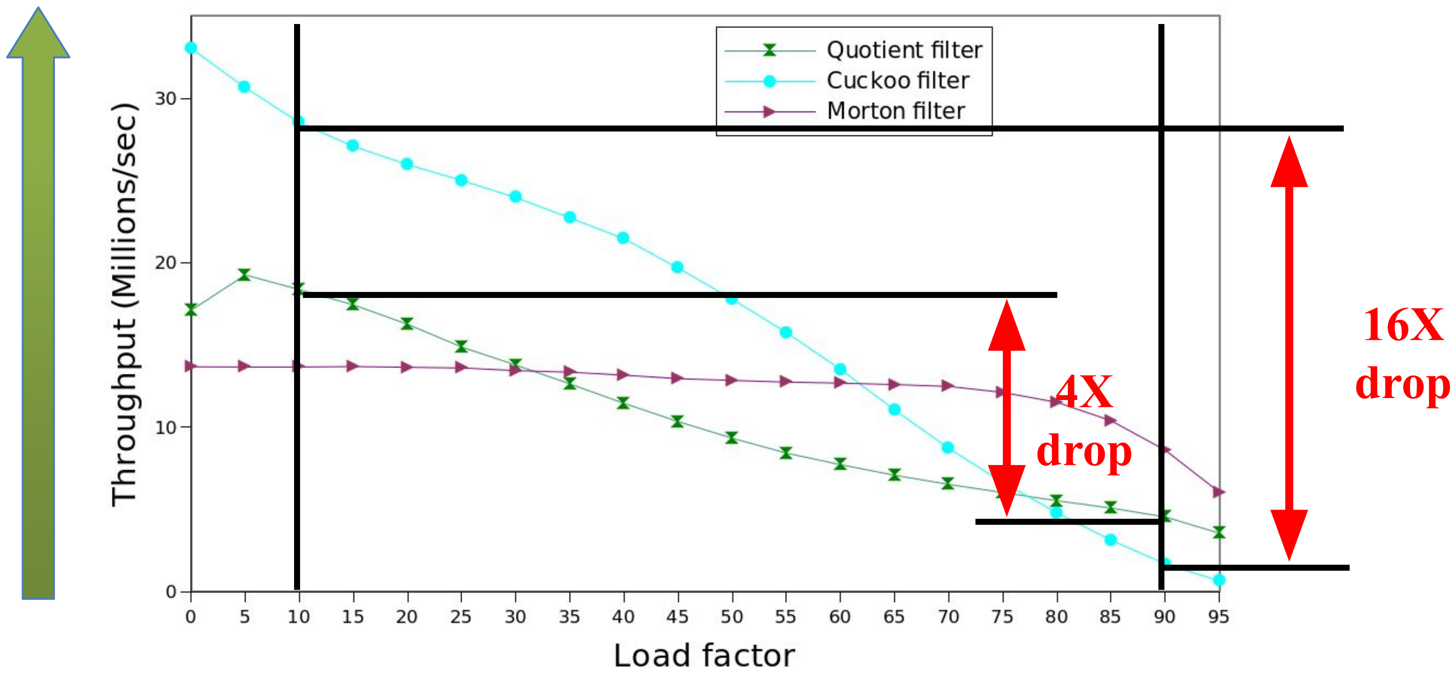
Performance suffers due to high-overhead of ***collision resolution***



Applications must choose between space and speed.

Current filters have a problem..

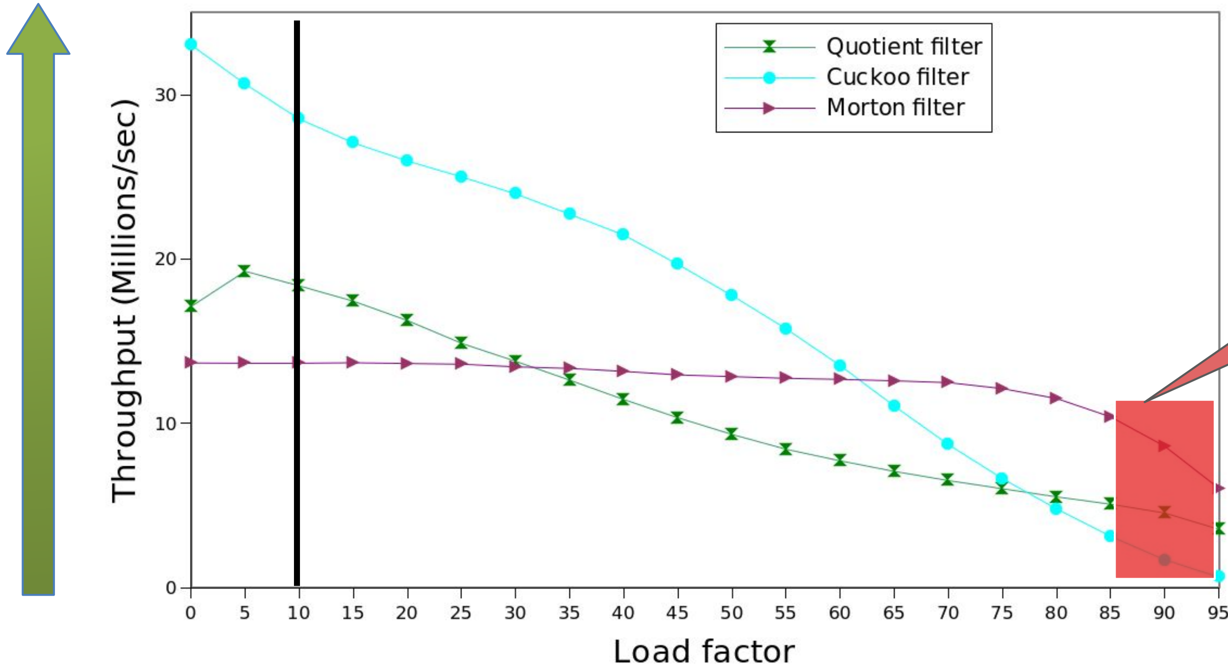
Performance suffers due to high-overhead of ***collision resolution***



Applications must choose between space and speed.

Current filters have a problem..

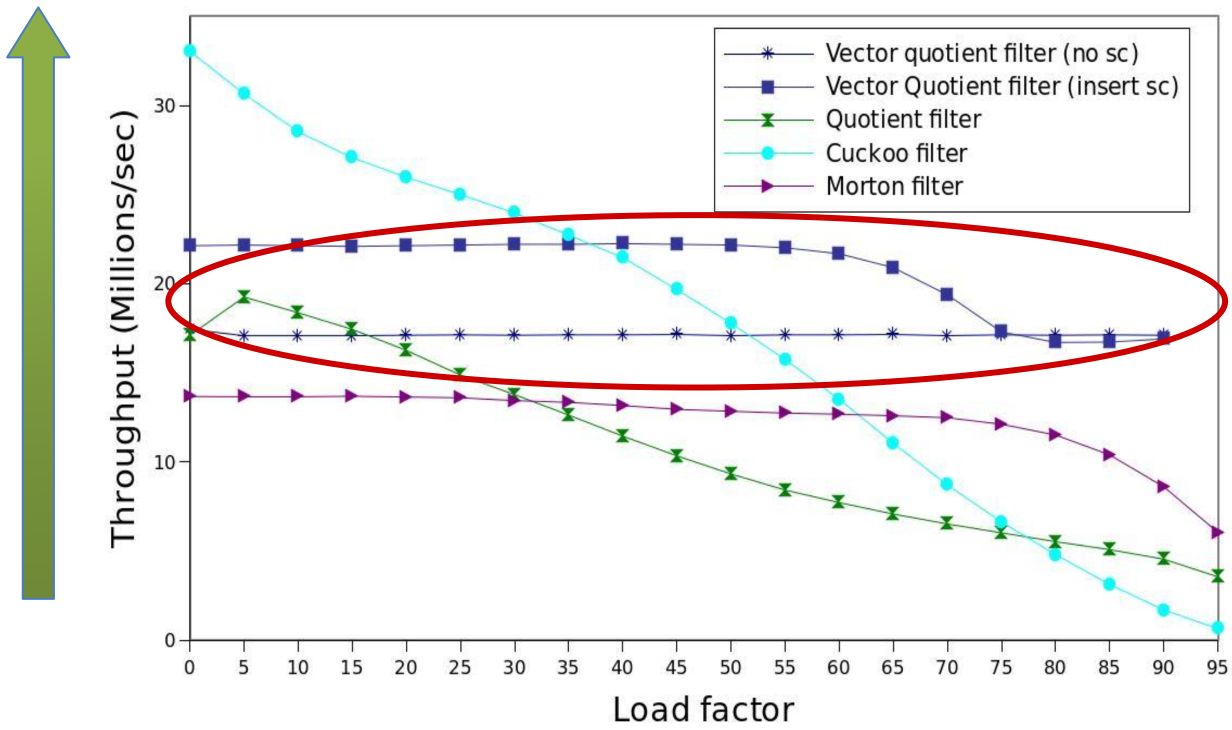
Performance suffers due to high-overhead of ***collision resolution***



Performance only matters at high load factors

Update intensive applications maintain filters close to full.

In this talk



The vector quotient filter offers high performance at all load factors.

Quotient filter performance [Pandey et al. '17]

	Optimal	Quotient filter
Space (bits)	$\approx n \log(1/\epsilon) + \Omega(n)$	$\approx n \log(1/\epsilon) + 2.125n$
CPU cost	$O(1)$	$O(1)$ expected
Data locality	$O(1)$ probes	1 probe + scan

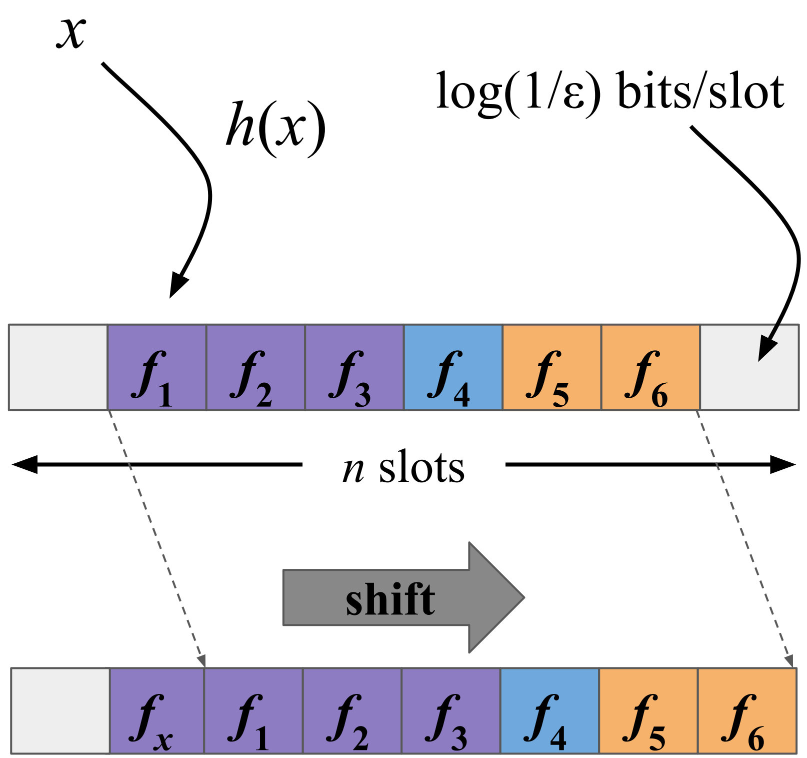
Why quotient filters slow down

Quotient filters use Robin-Hood hashing (a variant of linear probing)

QFs use 2 bits/slot to keep track of runs.

To insert item x:

1. Find its run.
2. Shift other items down by 1 slot.
3. Store $f(x)$.

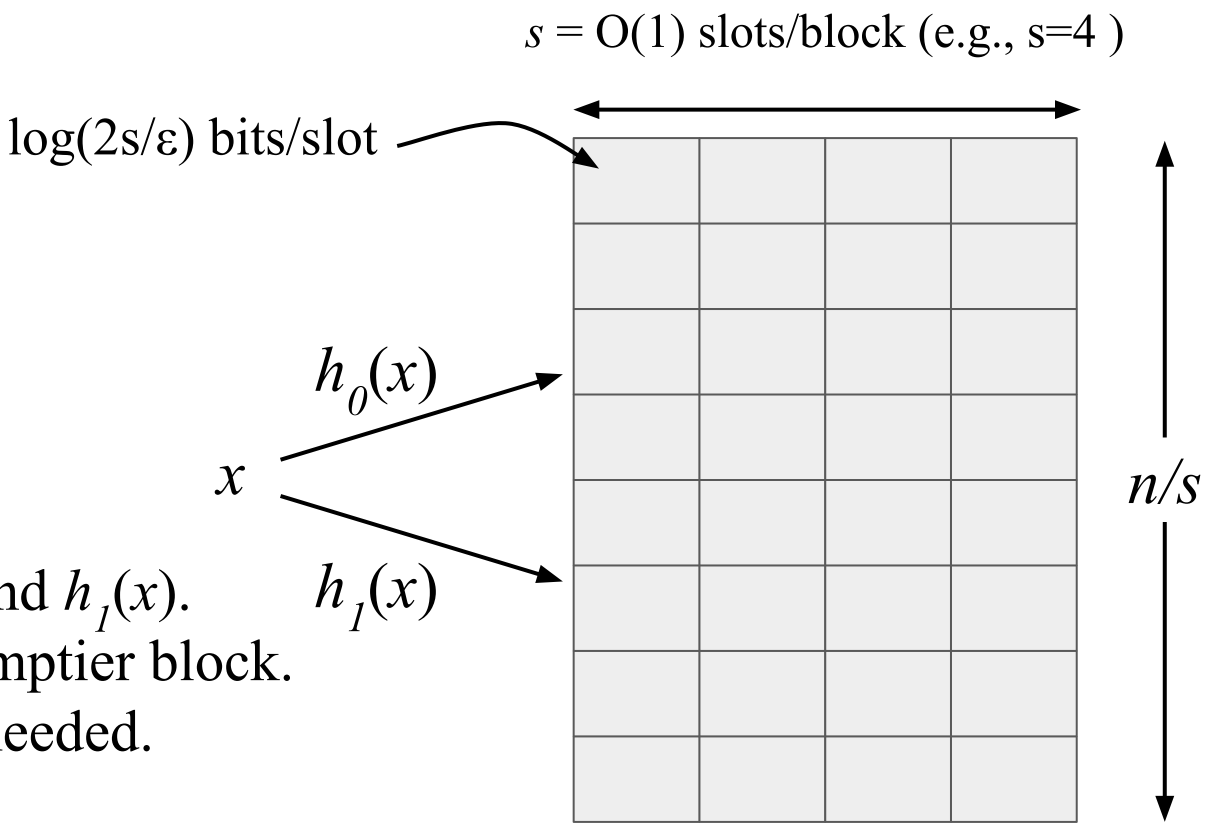


As the QF fills, inserts have to do more shifting.

Cuckoo filter performance [Fan et al. '14]

	Optimal	Cuckoo filter
Space (bits)	$\approx n \ \log(1/\epsilon) + \Omega(n)$	$\approx n \ \log(1/\epsilon) + 3n$
CPU cost	$O(1)$	up to 500
Data locality	$O(1)$ probes	random probes

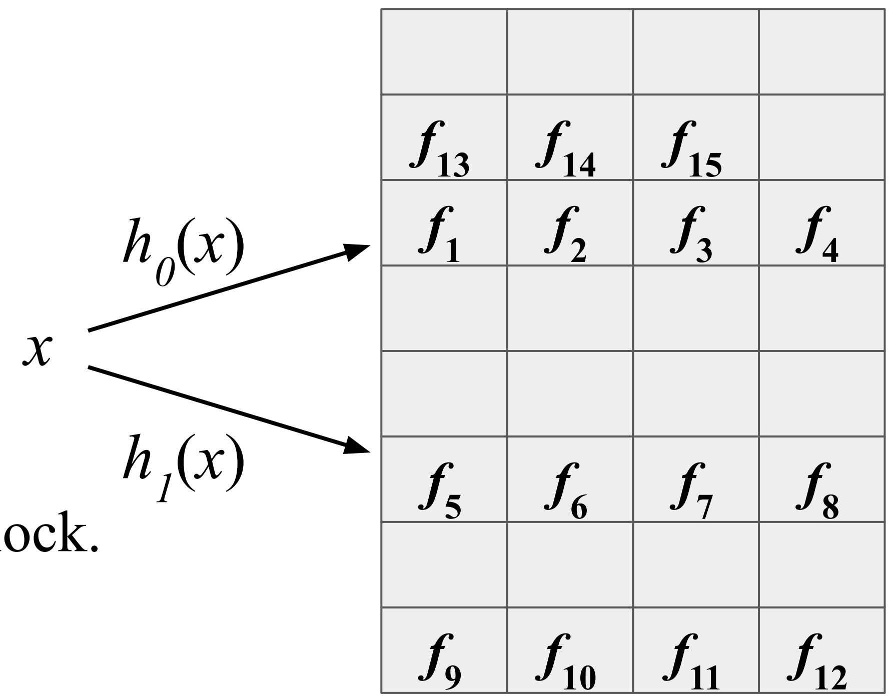
Why cuckoo filters slow down



To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.

Why cuckoo filters slow down



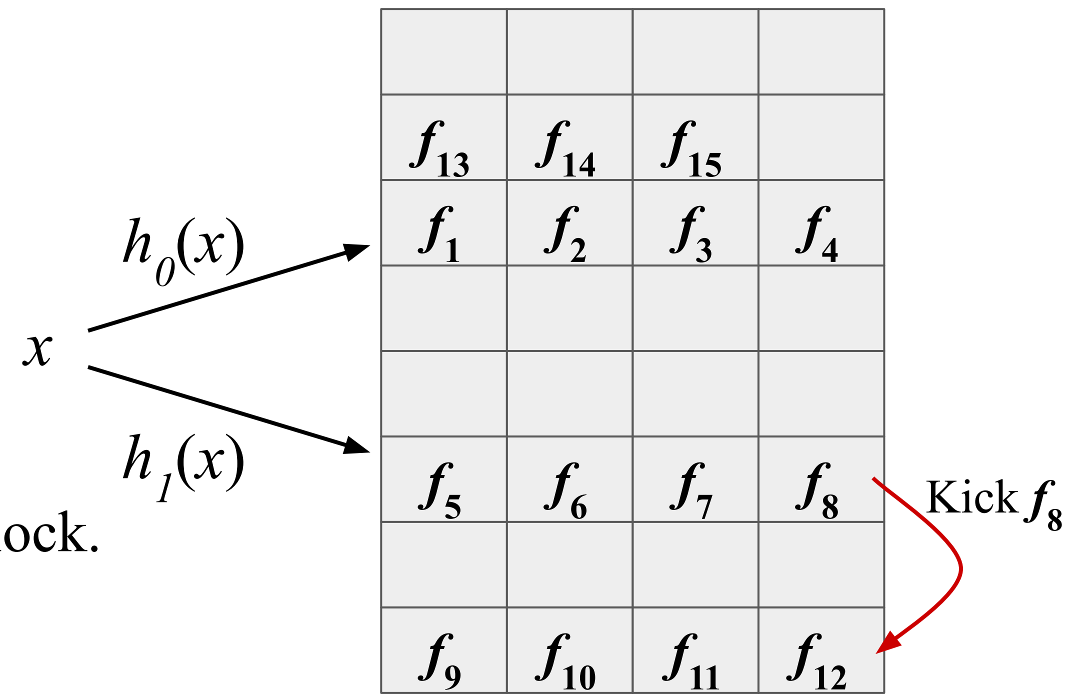
To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.

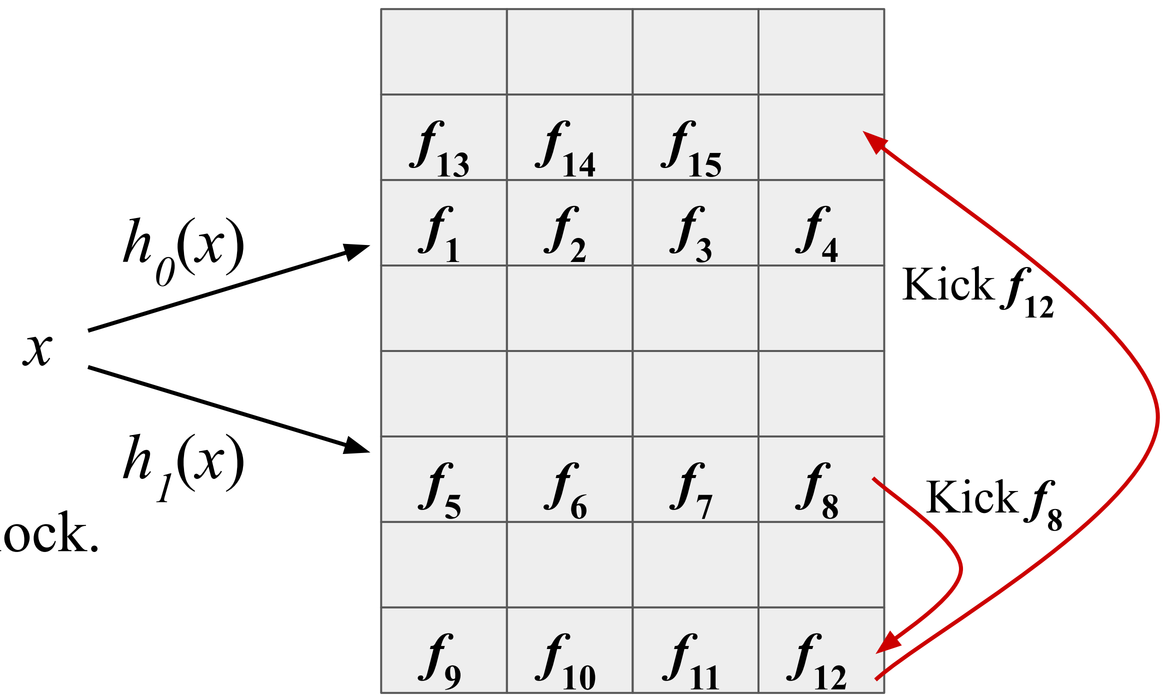
Why cuckoo filters slow down

To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.



Why cuckoo filters slow down

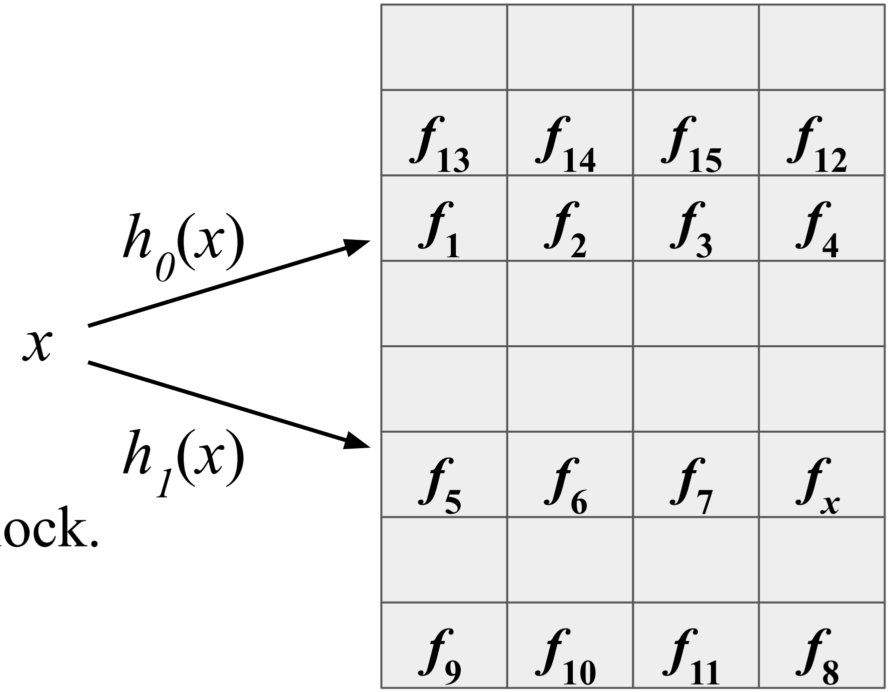


To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.

Note: $h_0(x)$ and $h_1(x)$ need to be dependent to support kicking.

Why cuckoo filters slow down



To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.

As the CF fills, inserts have to do more kicking.

Note: $h_0(x)$ and $h_1(x)$ need to be dependent to support kicking.

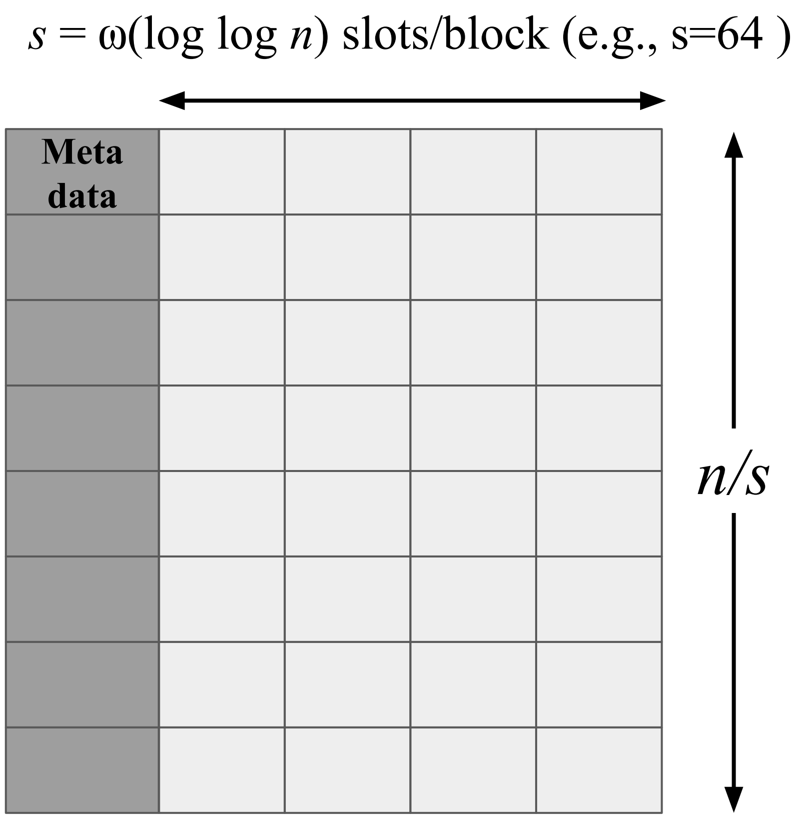
Vector quotient filter [Pandey et al. ‘21]

2-choice hashing

AVX-512

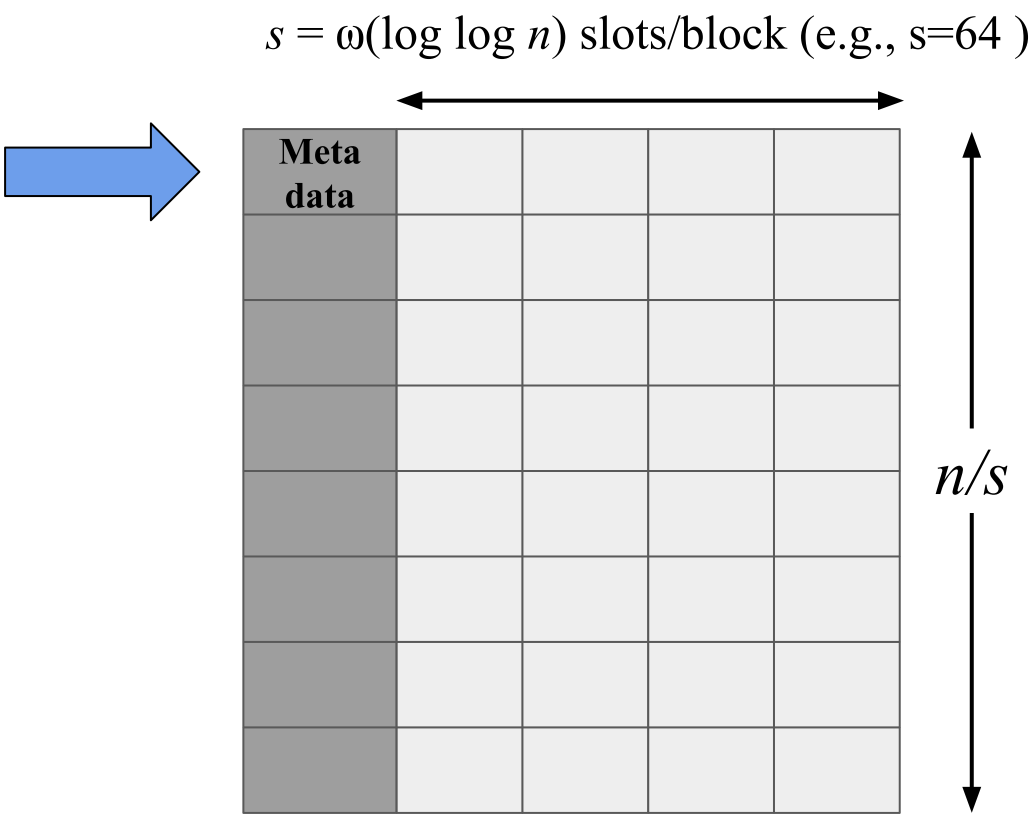
Robin Hood hashing

Vector quotient filter design



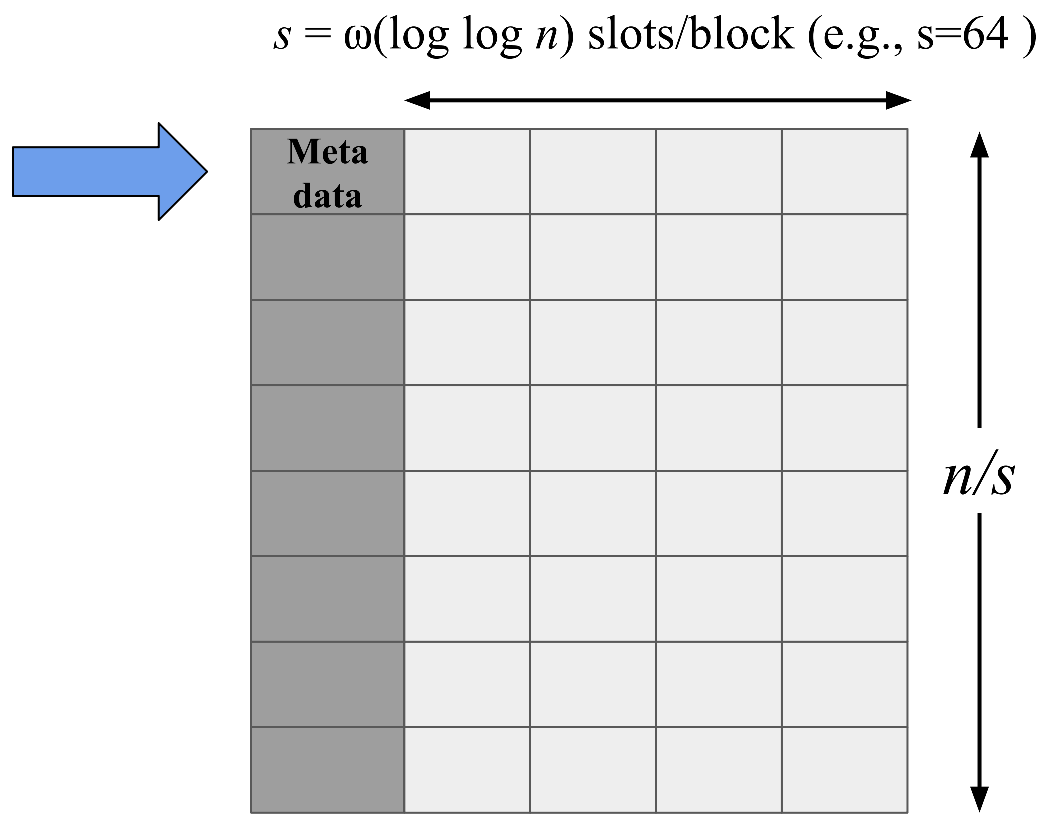
Vector quotient filter design

Each block is a small quotient filter with false-positive rate ε/2 and capacity *s*.



Vector quotient filter design

Each block is a small quotient filter with false-positive rate ε/2 and capacity *s*.

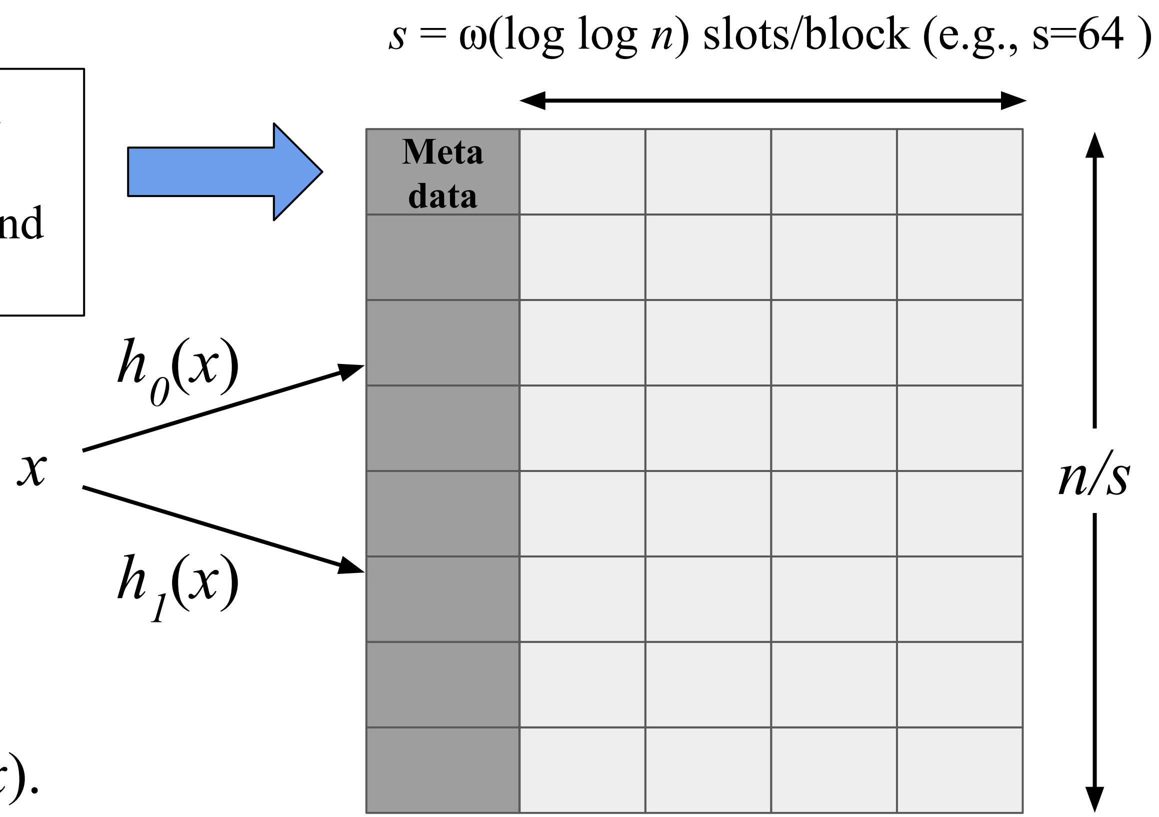


To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

Vector quotient filter design

Each block is a small quotient filter with false-positive rate ε/2 and capacity *s*.

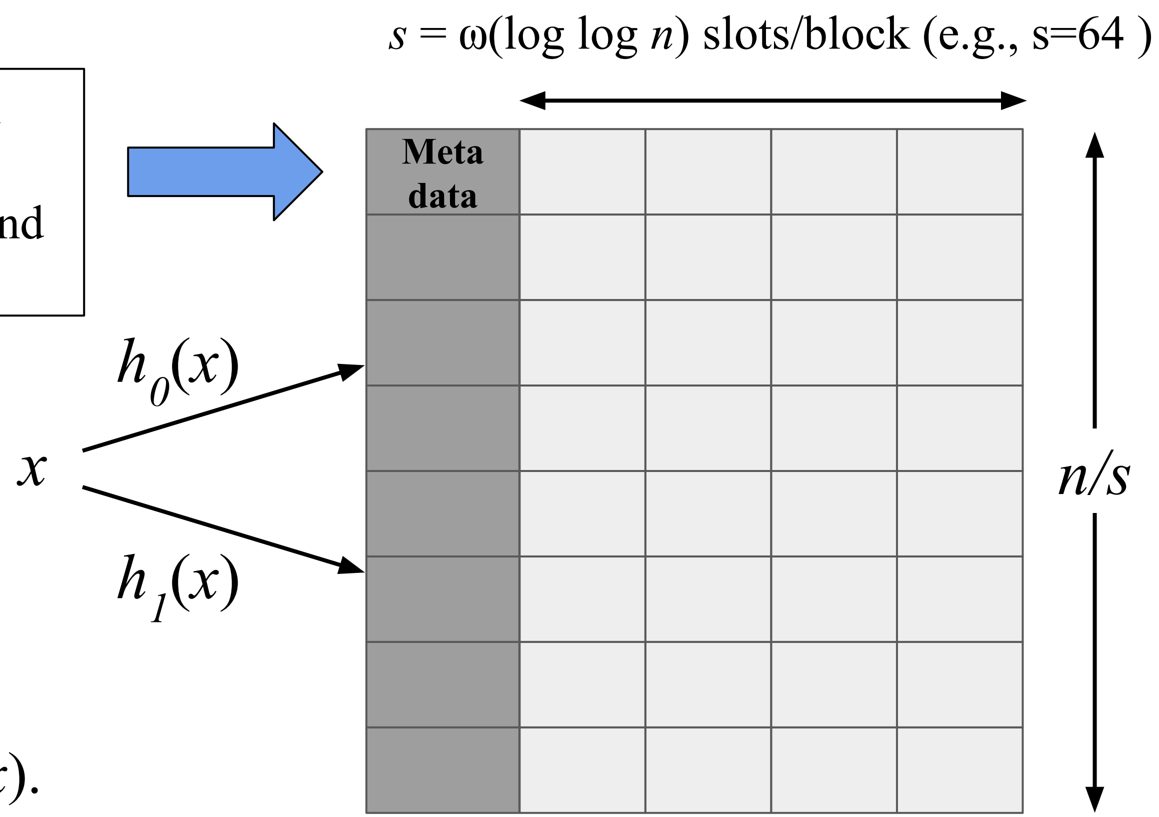


To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

Vector quotient filter design

Each block is a small quotient filter with false-positive rate ε/2 and capacity *s*.



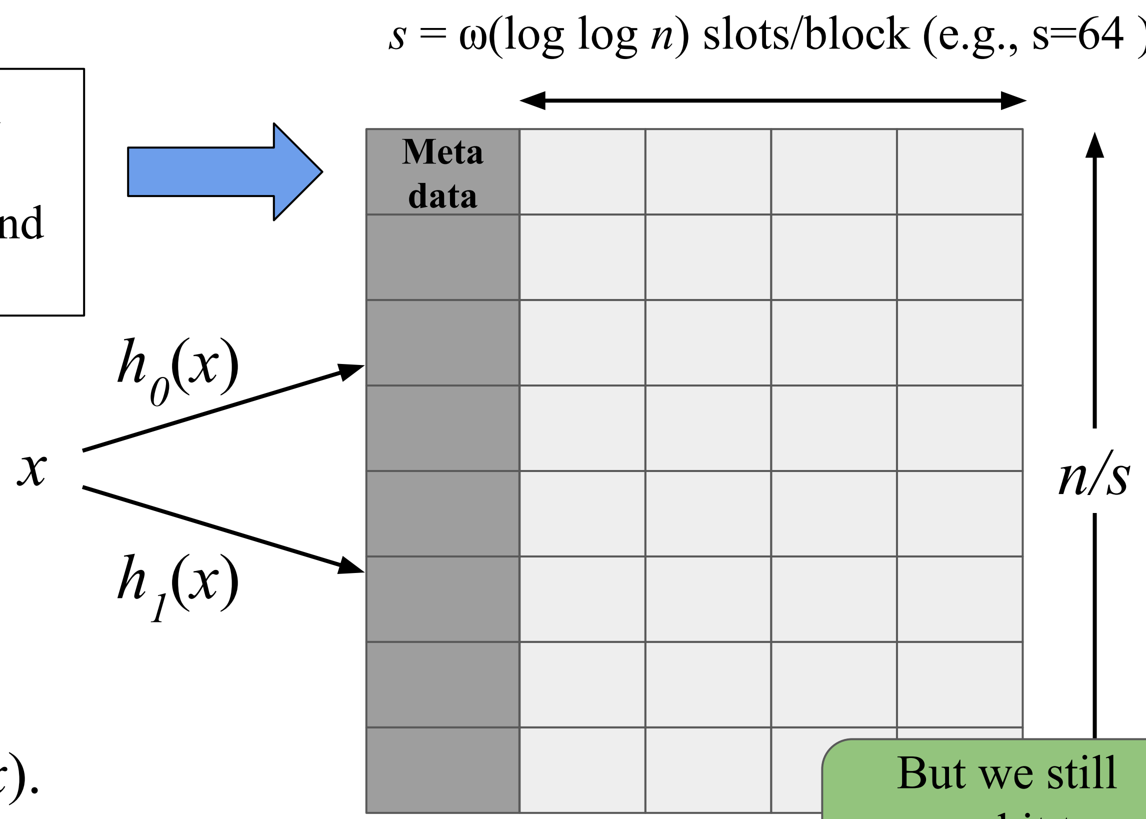
To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

No kicking ⇒ $h_0(x)$ and $h_1(x)$ can be independent for insert-only workload.

Vector quotient filter design

Each block is a small quotient filter with false-positive rate ε/2 and capacity *s*.



To insert item x:

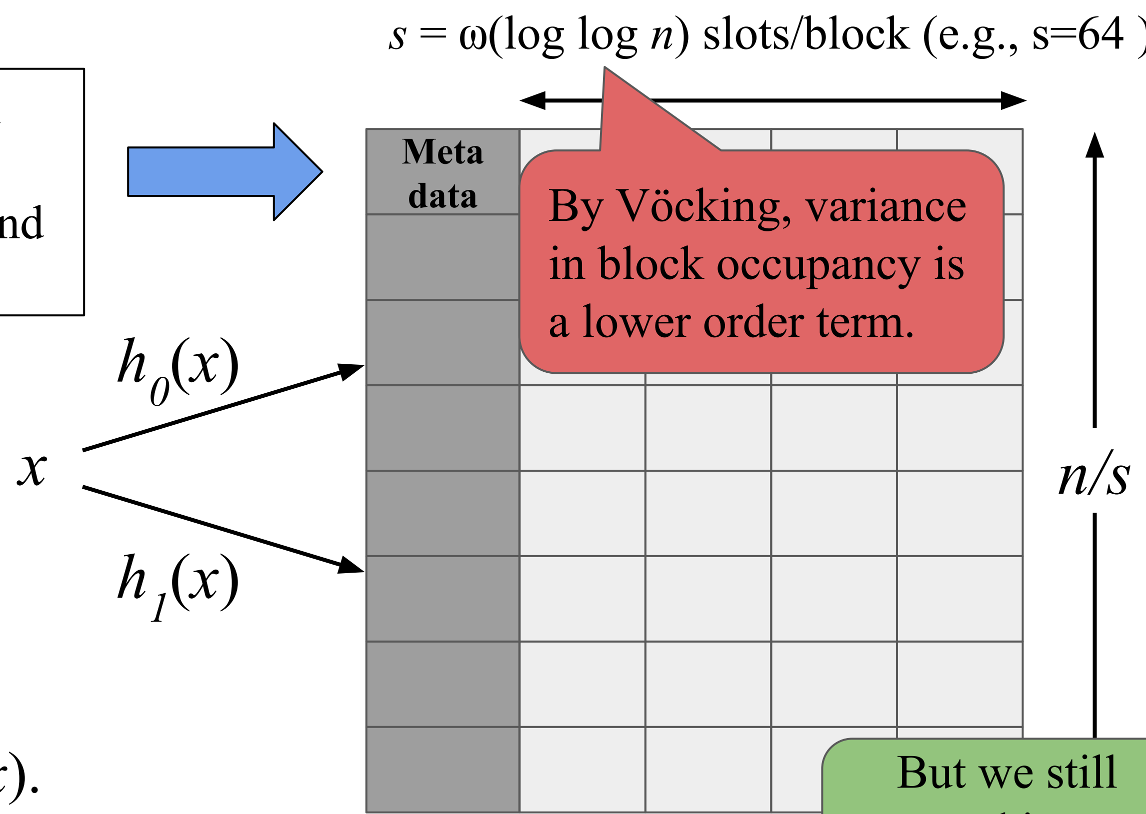
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

But we still need it to support deletes.

No kicking ⇒ $h_0(x)$ and $h_1(x)$ can be independent for insert-only workload.

Vector quotient filter design

Each block is a small quotient filter with false-positive rate ε/2 and capacity *s*.



To insert item x:

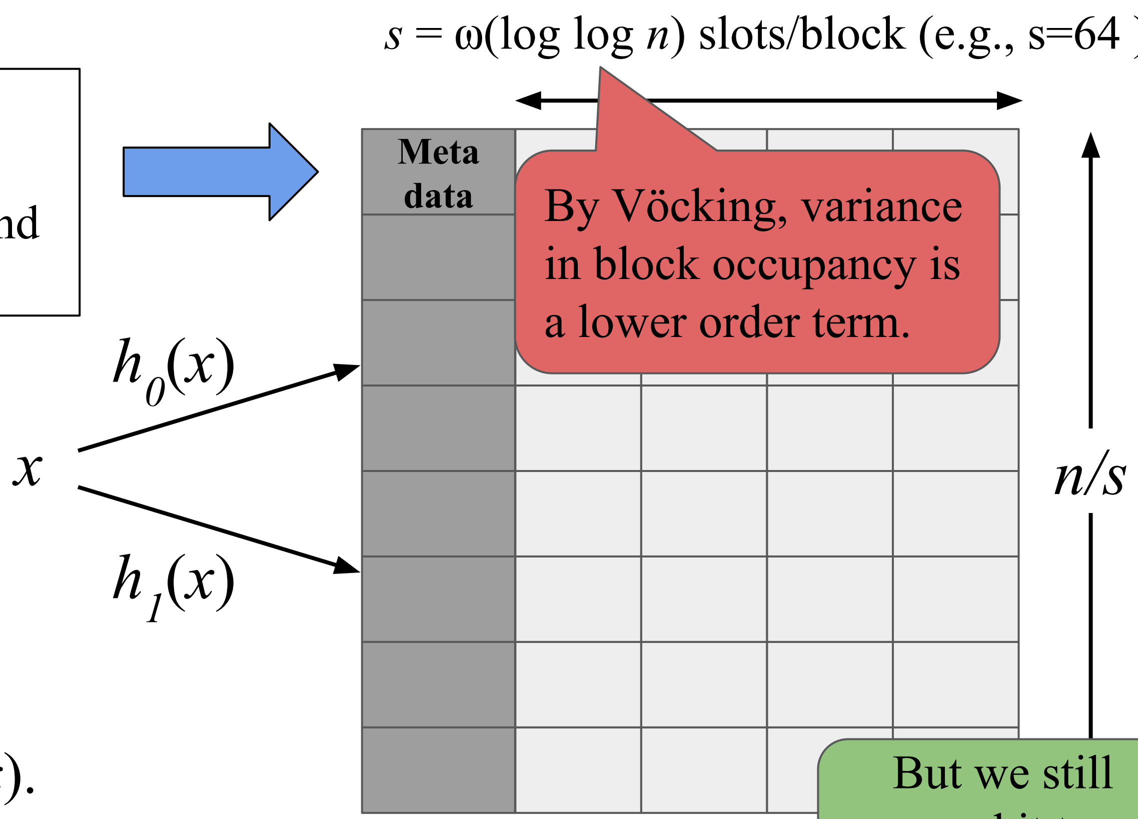
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

No kicking ⇒ $h_0(x)$ and $h_1(x)$ can be independent for insert-only workload.

But we still need it to support deletes.

Vector quotient filter design

Each block is a small quotient filter with false-positive rate ε/2 and capacity *s*.



By Vöcking, variance in block occupancy is a lower order term.

No kicking ⇒ easier concurrency

To insert item x:

1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

No kicking ⇒ $h_0(x)$ and $h_1(x)$ can be independent for insert-only workload.

But we still need it to support deletes.

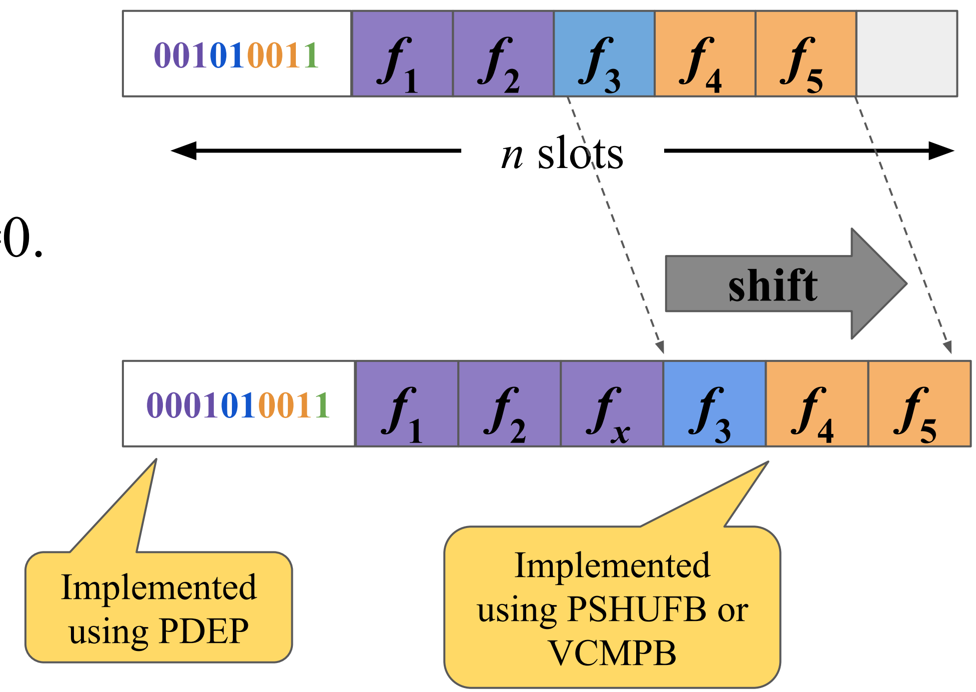
A vectorizable mini quotient filter

Each block has b logical buckets.

Fingerprints of each bucket are stored together.

We keep a bit vector of bucket boundaries.

Insert x, where $\beta(x)=0$.



Space efficiency is maximized when $b=s/\ln 2$.

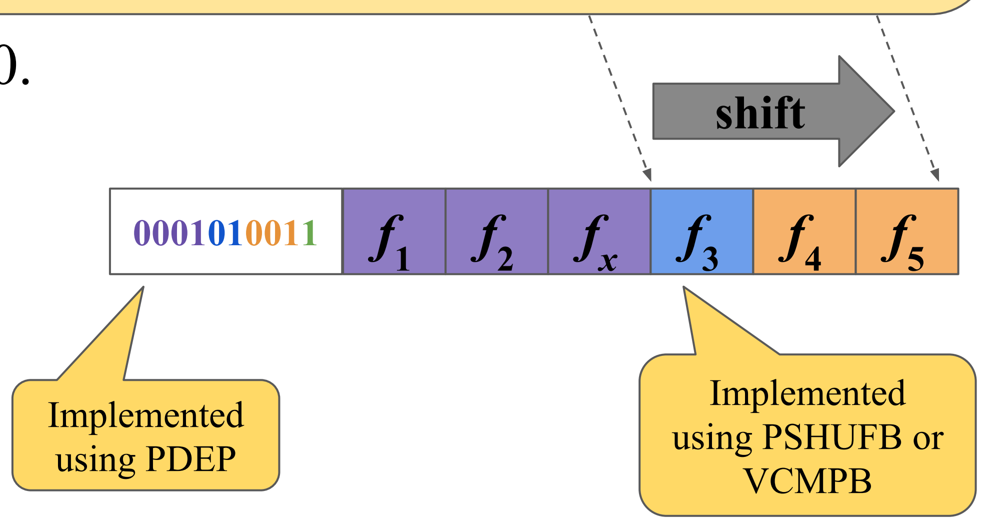
A vectorizable mini quotient filter

Each block has *b* logical buckets.

Fingerprints of each bucket are stored together.

Operations take constant time in a vector model of computation for vectors of size ω(log log n) [Bellloch '90]. Example, using AVX-512 instructions.

Insert *x*, where *β*(*x*)=0.

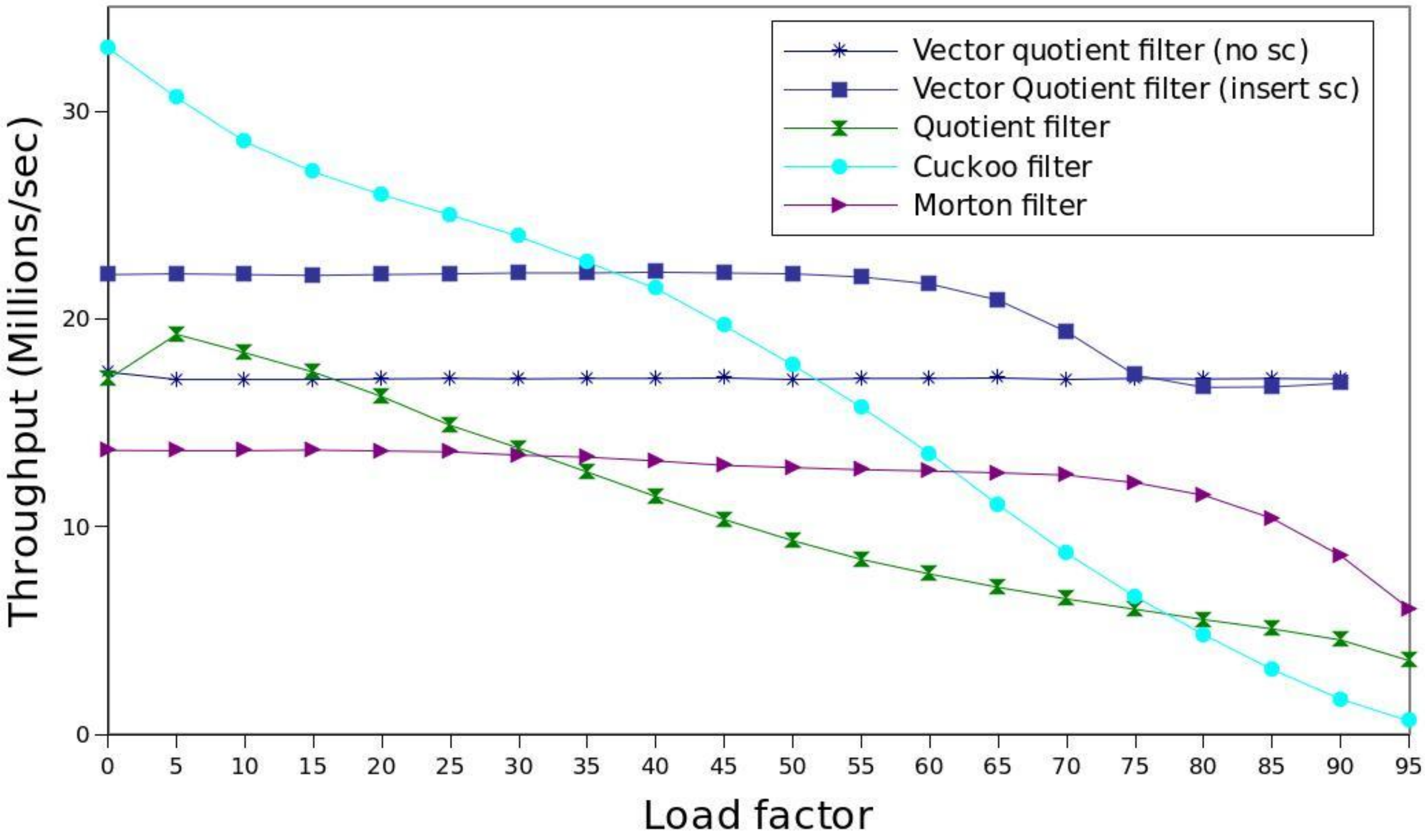


Space efficiency is maximized when *b*=*s*/ln2.

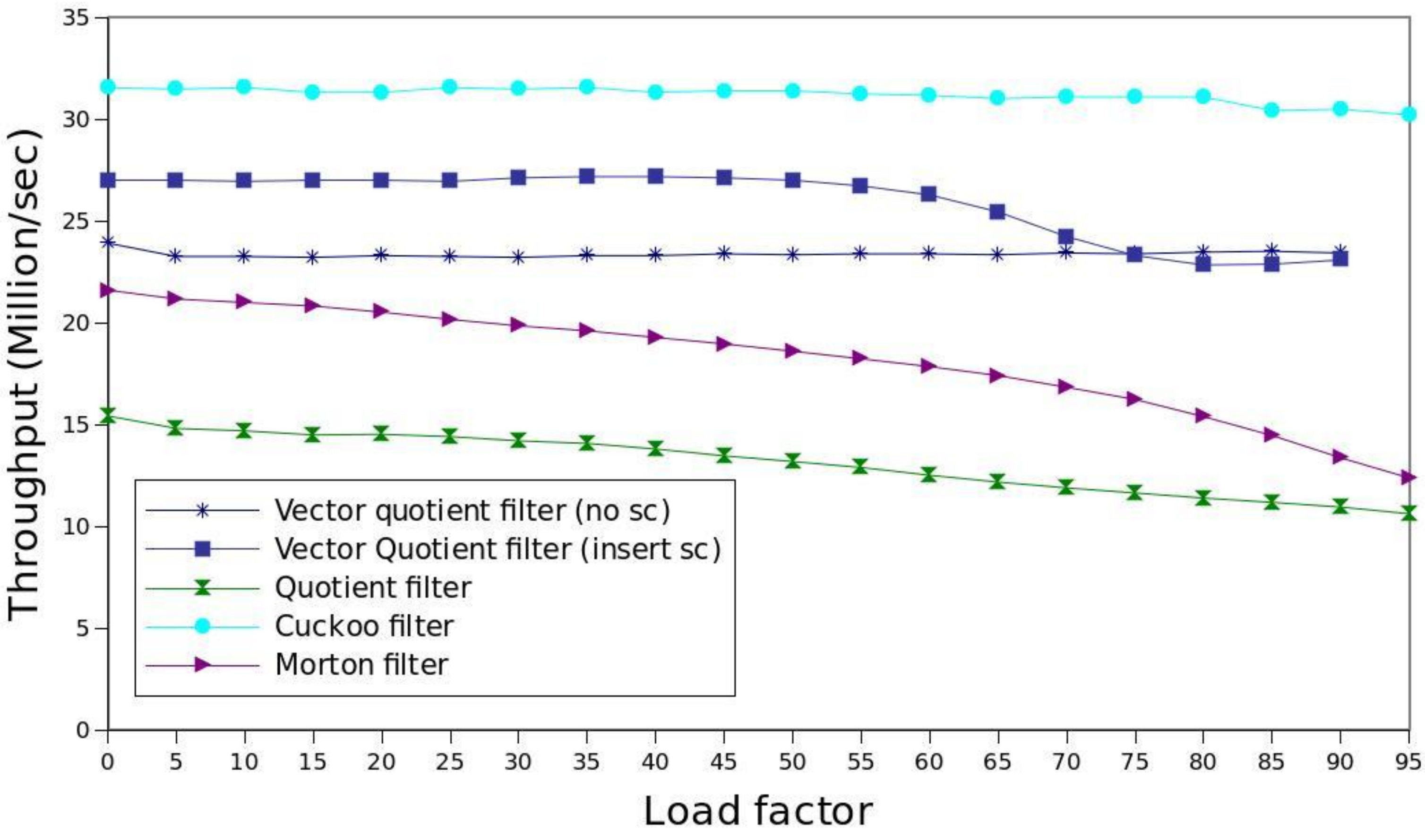
Vector quotient filter (VQF) performance

	Optimal	VQF
Space (bits)	$\approx n \ \log(1/\epsilon) + \Omega(n)$	$\approx n \ \log(1/\epsilon) + 2.91n$
CPU cost	$O(1)$	$O(1)$
Data locality	$O(1)$ probes	2 probes

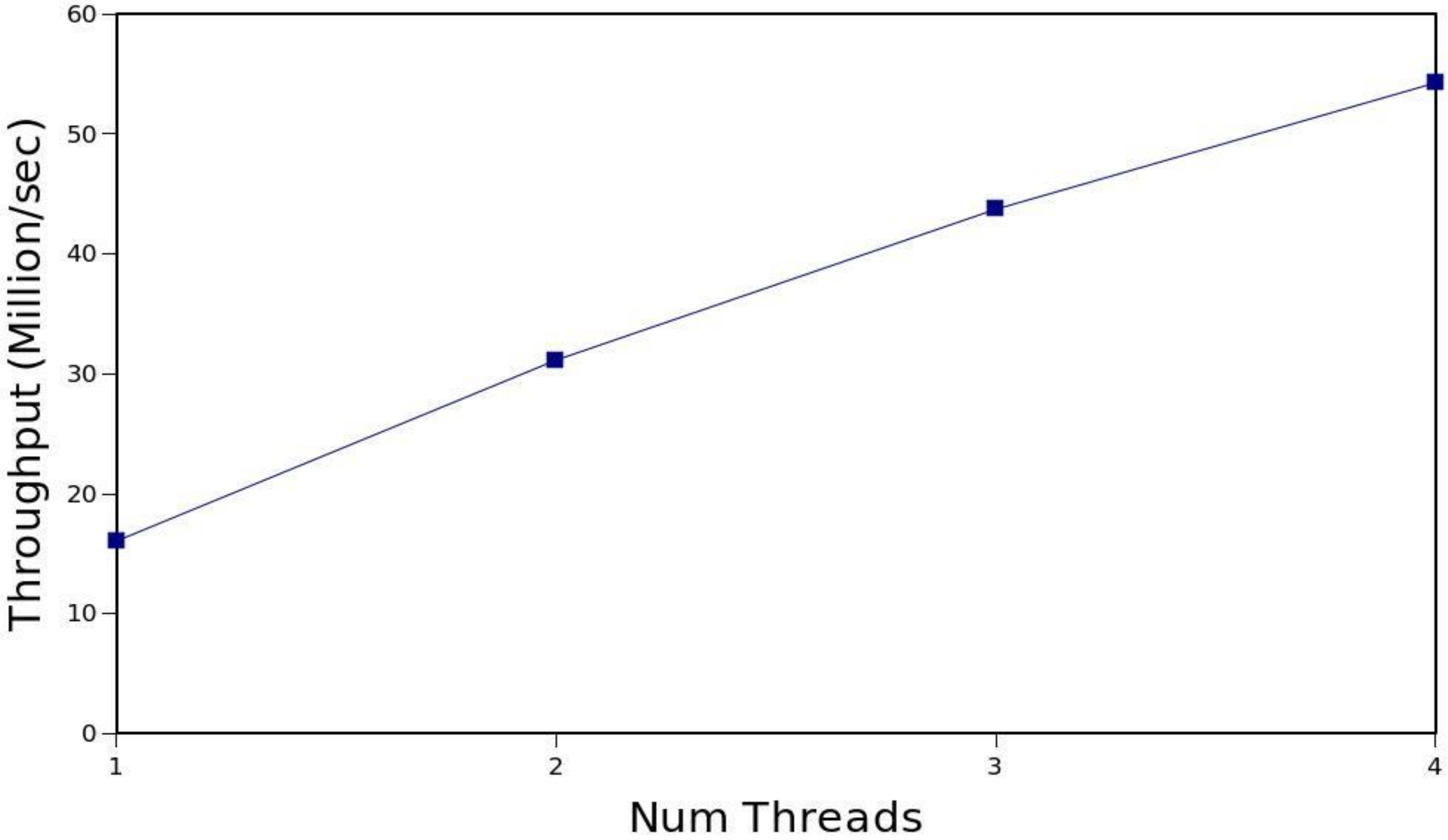
Evaluation: insertion



Evaluation: lookups



Evaluation: concurrency



Conclusion

The vector quotient filter outperforms current state of the art.

VQFs don't have time/space tradeoff.

https://github.com/splatlab/vqf