

CS 6530: Advanced Database Systems Fall 2022

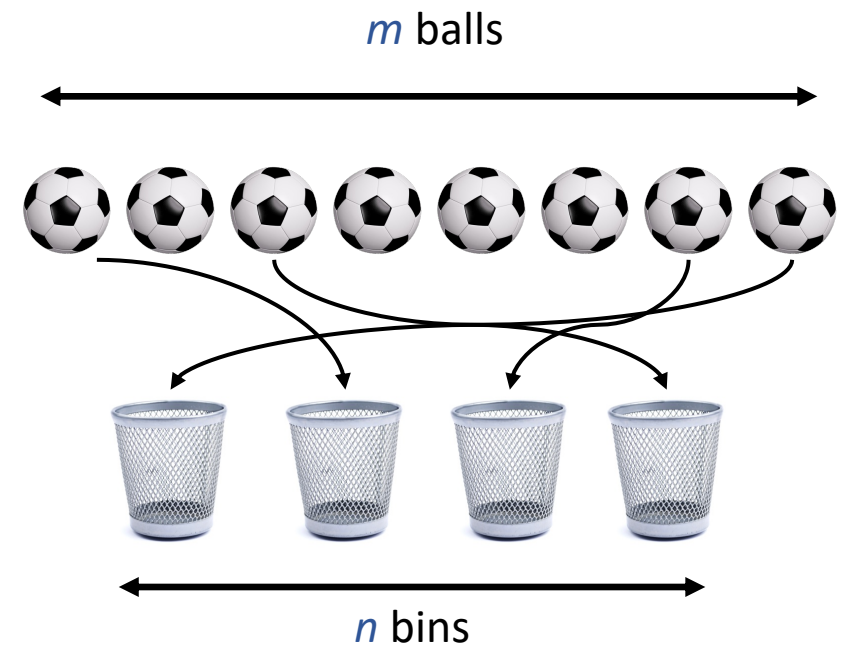
Lecture 04
In-memory indexing
(Hash tables, filters)

Prashant Pandey

prashant.pandey@utah.edu

The balls and bin model

- Resource load balancing is often modeled by the task of throwing balls into bins
 - Hashing, distributed storage, online load balancing, etc.
- Throw m balls into n bins:
 - Pick a bin uniformly at random
 - Insert a ball into the bin
 - Repeat m times.



The single choice paradigm

- Throw m balls into n bins:
 - Pick a bin uniformly at random
 - Insert a ball into the bin
 - Repeat m times.

Number of Balls	$m = n$	$m \geq n \log n$
Max Load	$(1 + o(1)) \frac{\log n}{\log \log n}$	$\frac{m}{n} + \sqrt{\frac{m \log n}{n}}$

The multiple choice paradigm

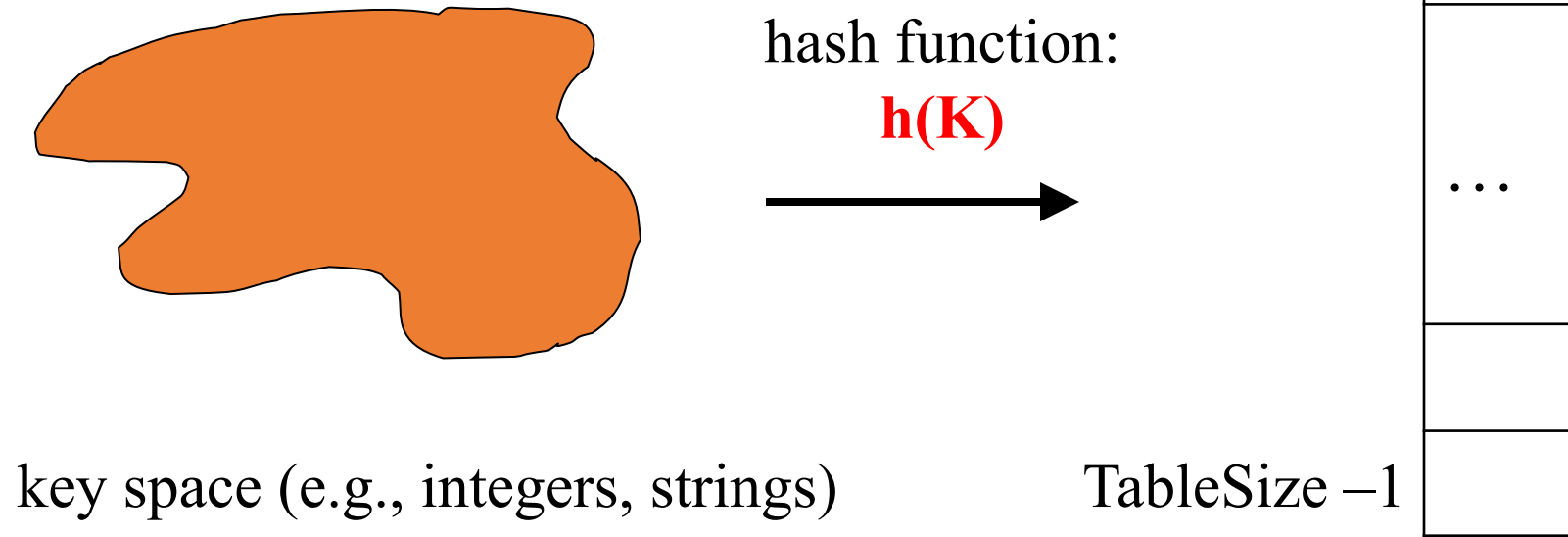
- Throw m balls into n bins:
 - Pick d bins uniformly at random ($d \geq 2$)
 - Insert the ball into the less loaded bin
 - Repeat m times.

Number of Balls	$m = n$	$m \geq n \log n$
Max Load with prob. $1 - \frac{1}{n}$	$\frac{\log \log n}{\log d}$ [ABKU94]	$\frac{m}{n} + \frac{\log \log n}{\log d}$ [BCSV00]

independent of m

Hash Tables

- Constant time $O(1)$ accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:



Example

- key space = integers
- TableSize = 10
- $h(K) = K \bmod 10$
- **Insert: 7, 18, 41, 94**

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions

1. **simple/fast** to compute,
2. Avoid **collisions**
3. have keys distributed **evenly** among cells.

Perfect Hash function:

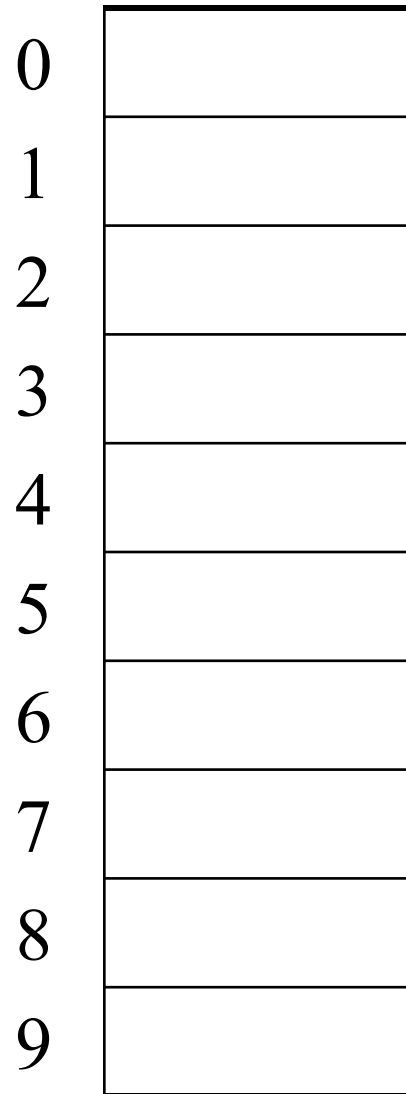
Collision Resolution

Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

Separate Chaining



Insert:

10

22

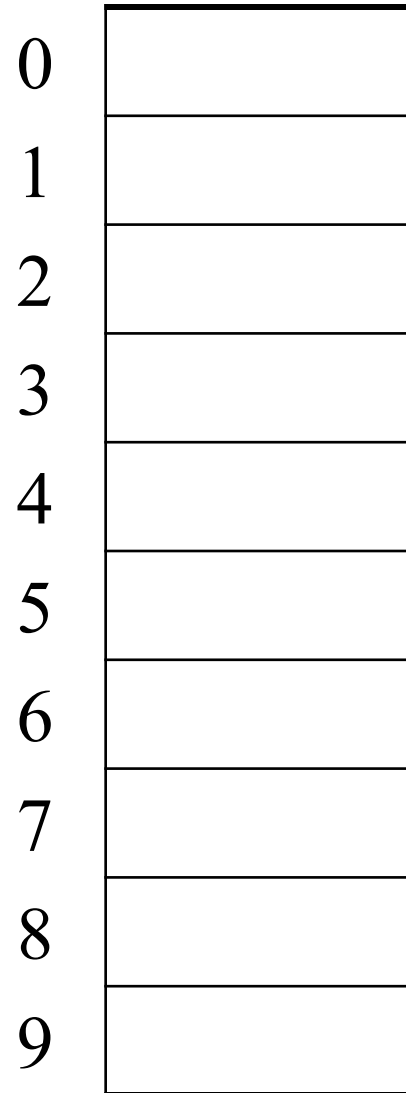
107

12

42

- **Separate chaining:** All keys that map to the same hash value are kept in a list (or “bucket”).

Open Addressing



Insert:

38

19

8

109

10

- **Linear Probing:** after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Existing hash table techniques

Separate chaining

- Chaining with linked-list
- Chaining with binary tree

Open addressing

- Linear probing
- Coalesced chaining
- Double hashing
- Cuckoo hashing
- Hopscotch hashing
- Robin Hood hashing
- 2-choice hashing
- d-left hashing

- Cuckoo hashing suffers from *random hopping*
- Linear probing/Robin Hood hashing suffer from *long chains*
- 2-choice/d-left hashing suffer from *multiple probes*

Desirable features in a hash table

- High write performance
- High read performance
- Space efficiency

Desirable features in a hash table

- High write performance
- High read performance
- Space efficiency
- High concurrency
- Efficient resizing
- Crash-safety (PMEM)

Three design objectives for hash tables

- **Stability**
 - The position where an element is stored is guaranteed not to change except during a resize
- **Low associativity**
 - The number of locations where an element is allowed to be stored is small
- **Space efficiency**
 - The ratio of the data size and the hash table size

Design objectives for hash tables

- Stability

- Low write amp → high write performance
- Simplified concurrency & crash consistency

- Low associativity

Low read amp → high read performance

- Space efficiency

Current hash table space

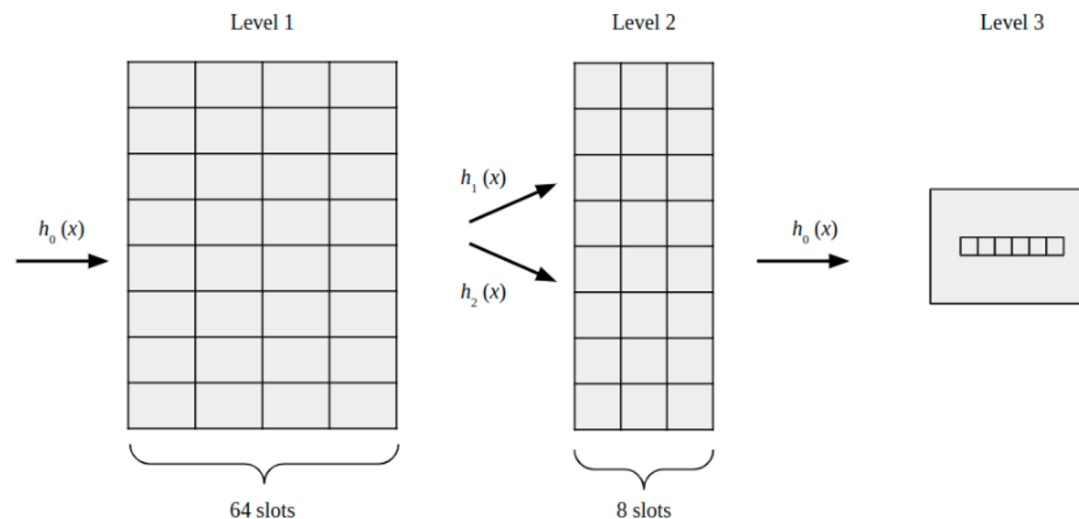
	Type	Stability	Low associativity	Space efficiency
Cuckoo HT	Cuckoo	✗	✓	✓
Intel TBB	Chaining	✓	✗	✗
Dash	Extendible	✗	✓	✗
CLHT (PMEM)	Chaining	✓	✗	✗
Folklore	Linear probing	✗	??	✓

Inherent tension between stability, low associativity, and space efficiency.

Iceberg HT simultaneously achieves all three design objectives

Three levels

- Level 1 uses single hashing
- Level 2 used two-choice hashing
- Level 3 is chaining

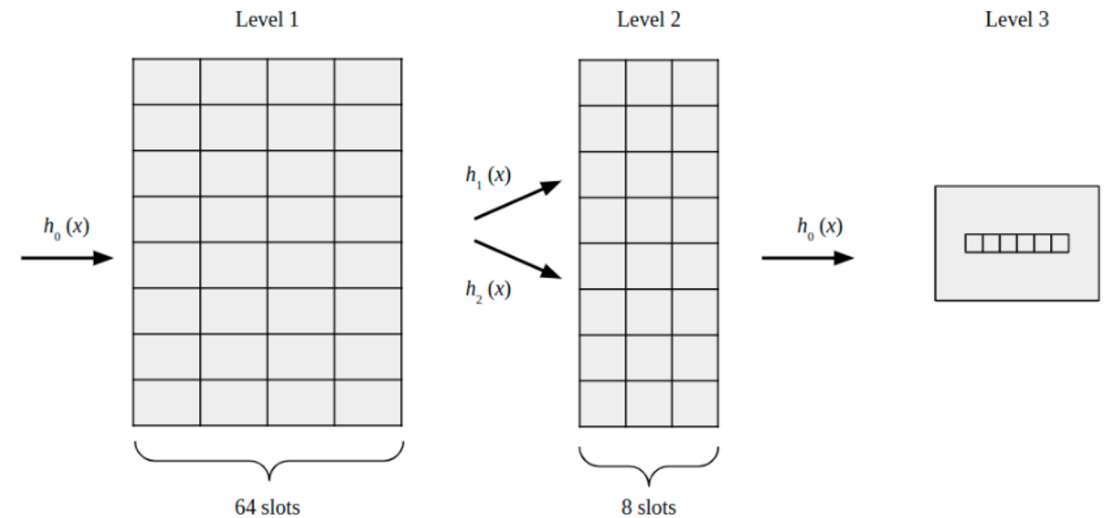


Level 1 blocks contain 64 slots

Level 2 is configured to be 10% the size of level 1

Iceberg HT simultaneously achieves all three design objectives

- Items do not move unless the hash table is resized
- Associativity is low
- Large blocks in level 1 and two-choice hashing in level 2 enables high space efficiency



Iceberg HT design (metadata)

- We use **8 bit fingerprint** for each key in the metadata
- Metadata for each block in level 1 (and 2) fits inside **1 cache line**
- Metadata helps to:
 - **locate empty slots** during insertion
 - **filter out slots** to probe during a query
- Metadata fingerprint is also used for synchronization
- Metadata is transient is only stored in DRAM
- We rebuild the metadata during recovery

Insert, query, and delete operations take 2 cache line access for most items

Iceberg HT features

- High performance and space efficiency
- Non-blocking and in-place resizing
- Lock-free operations
- Fenceless crash safety

Iceberg HT performance (DRAM)

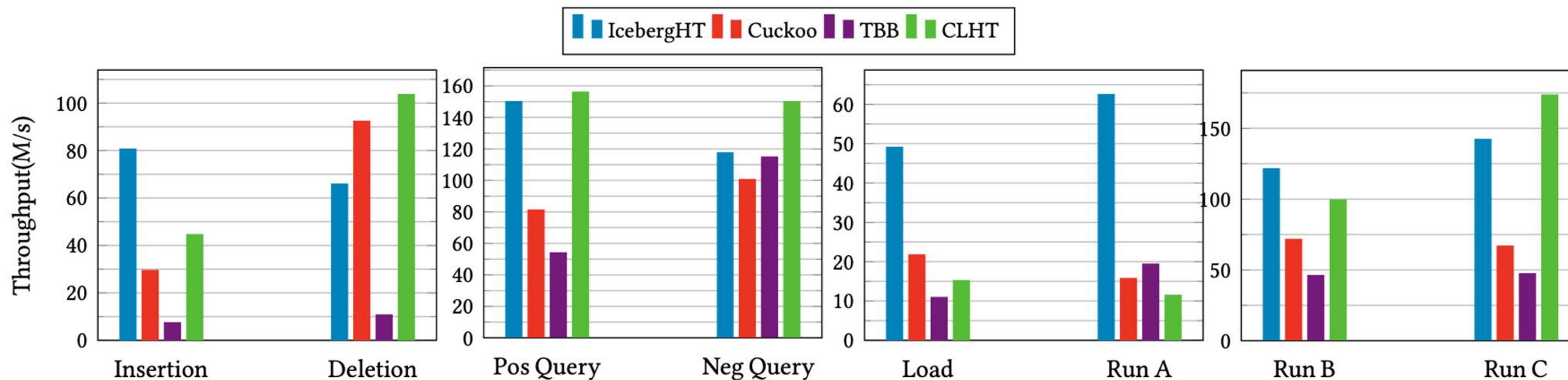


Figure 6: Throughput for insertions, deletions, and queries (positive and negative) using 16 threads for DRAM hash tables. The throughput is computed by inserting $0.95N$ keys-value pairs where N is the initial capacity of the hash table. (Throughput is Million ops/second)

Iceberg HT performance (DRAM)

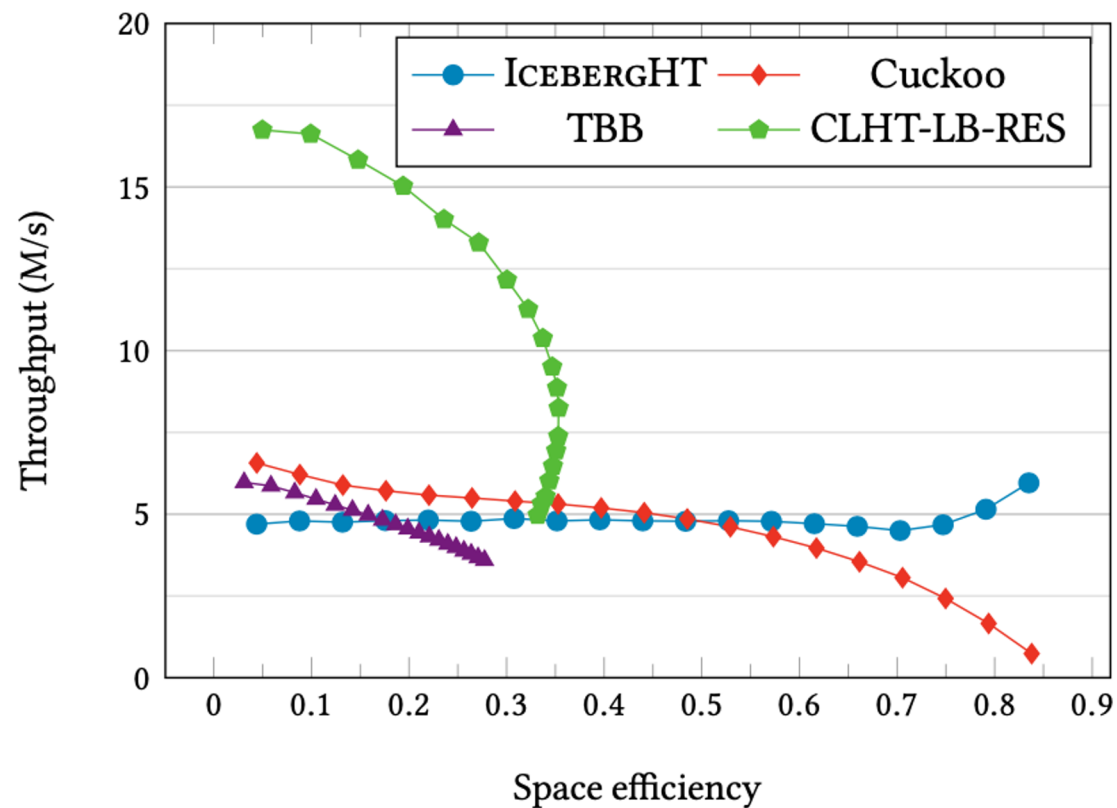


Figure 7: Insertion throughput and space efficiency performance of hash tables in DRAM. (Throughput is Million ops/second)

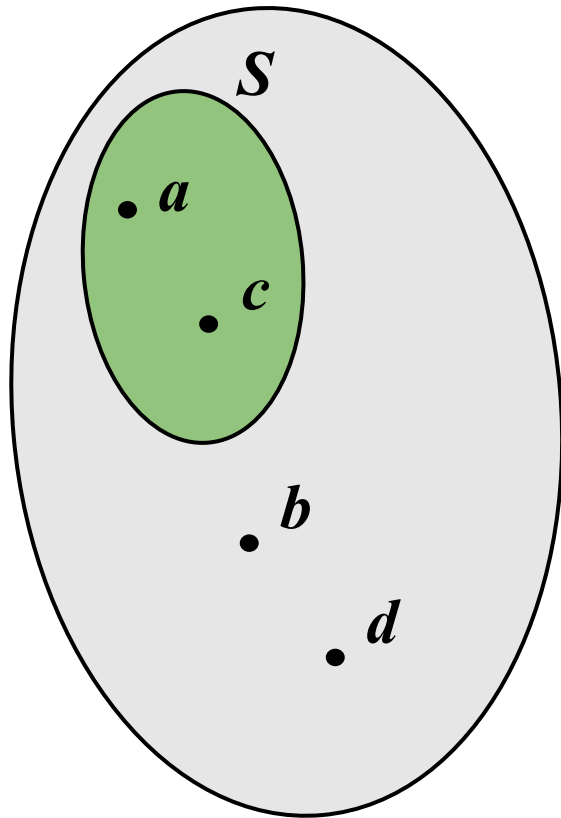
Iceberg HT performance (DRAM)

	DRAM					
	Insertions			Positive Queries		
Percentile	ICEBERGHT	libcuckoo	TBB	ICEBERGHT	libcuckoo	TBB
50	299 ns	264 ns	819 ns	286 ns	198 ns	494 ns
95	624 ns	2.02 μ s	1.59 μ s	629 ns	429 ns	955 ns
99	1.21 μ s	5.99 μ s	2.24 μ s	994 ns	562 ns	1.22 μ s
99.9	32.1 μ s	19.8 μ s	6.52 μ s	1.60 μ s	836 ns	1.57 μ s
99.99	41.7 μ s	219 μ s	9.27 μ s	2.92 μ s	218 μ s	4.97 μ s
max	8.62 ms	2.05 s	734 ms	8.54 ms	1.01 s	42.8 μ s

Table 4: Percentile latencies in ICEBERGHT, libcuckoo and TBB on DRAM for YCSB workload A run.

Dictionary data structure

A dictionary maintains a set S from universe U .



membership(a): ✓

membership(b): ✗

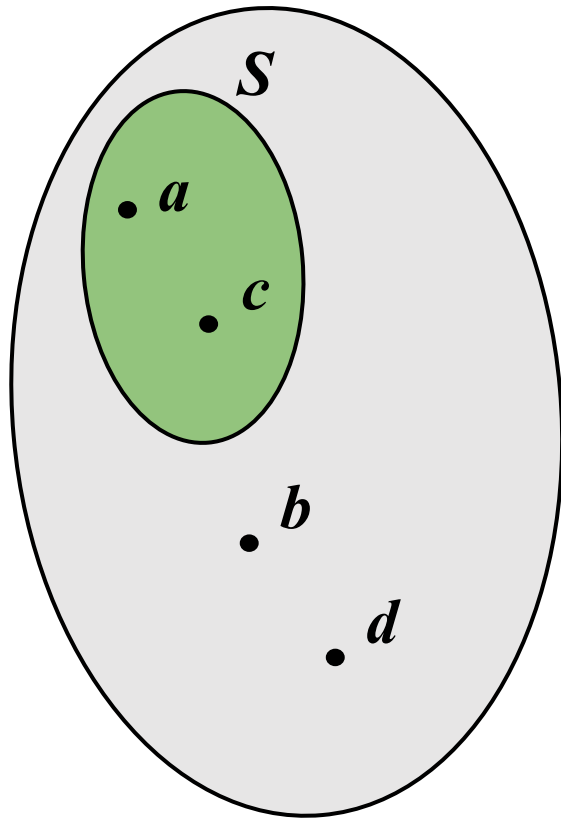
membership(c): ✓

membership(d): ✗

A dictionary supports membership queries on S .

Filter data structure

A filter is an *approximate* dictionary.



membership(a): ✓


membership(b): ✗

membership(c): ✓

membership(d): ✓ 🙅 **false positive**

A filter supports approximate membership queries on S .

A filter guarantees a false-positive rate ε

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

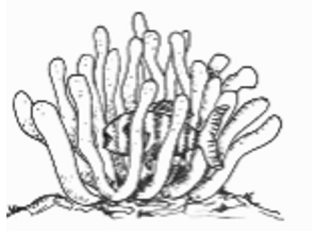
if $q \notin S$, return $\left\{ \begin{array}{l} \times \text{ with probability } > \square 1 - \varepsilon \text{ **true negative**} \\ \checkmark \text{ with probability } \leq \varepsilon \text{ **false positive**} \end{array} \right.$

one-sided
errors

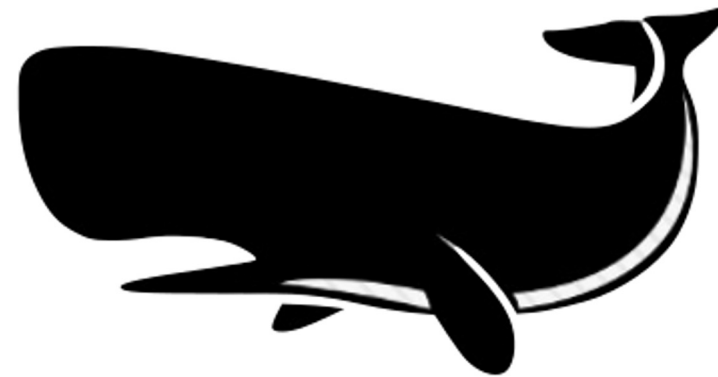
False-positive rate enables filters to be compact

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

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



Filter

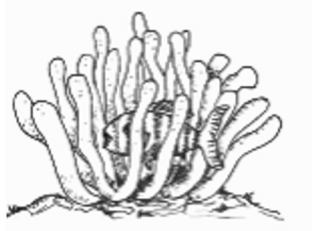


Dictionary

False-positive rate enables filters to be compact

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

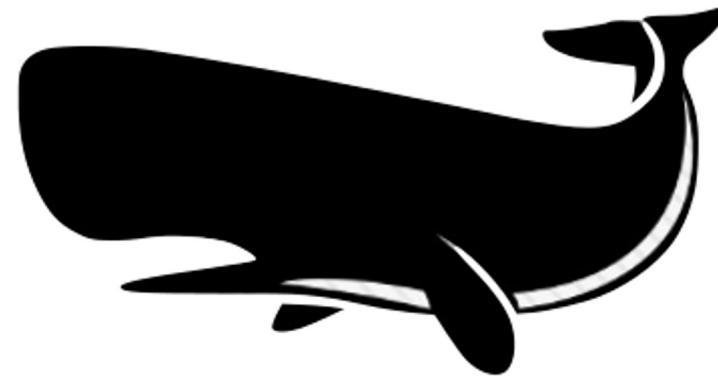
Small



Filter

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

Large



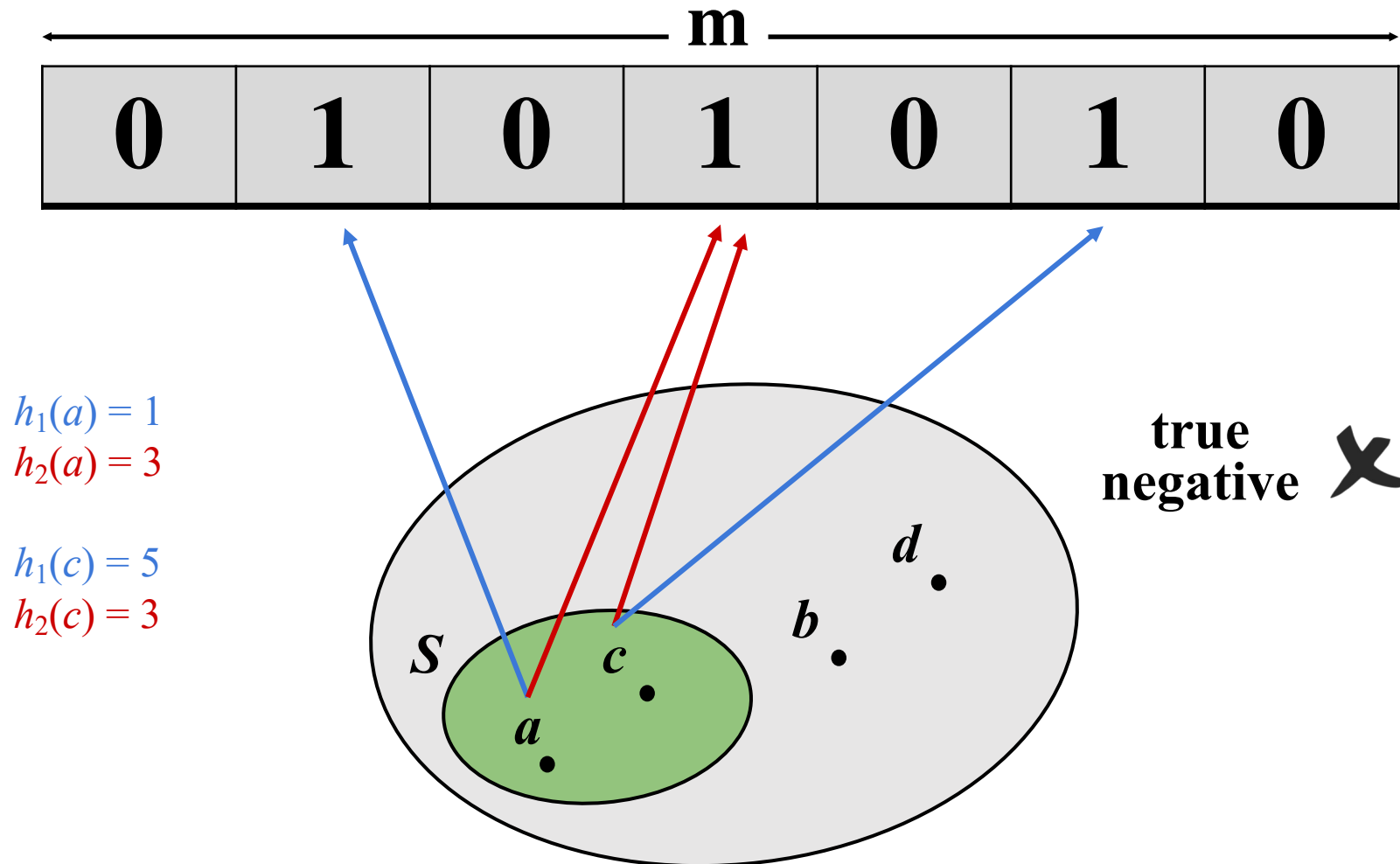
Dictionary

For most practical purposes:

$\epsilon = 2\%$, a Bloom filter requires ≈ 8 bits/item

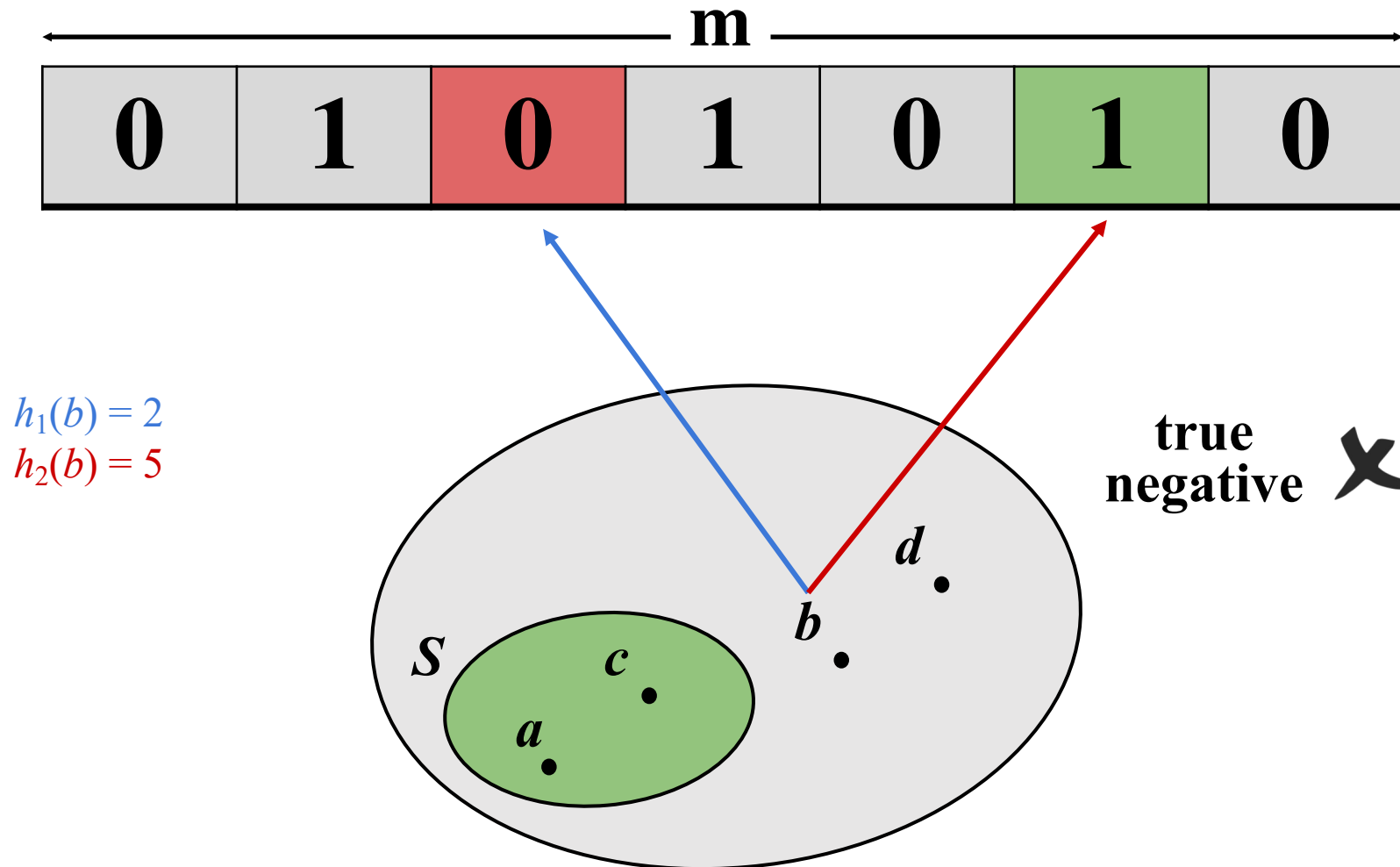
Classic filter: The Bloom filter [Bloom '70]

Bloom filter: a bit array + k hash functions (here $k=2$)



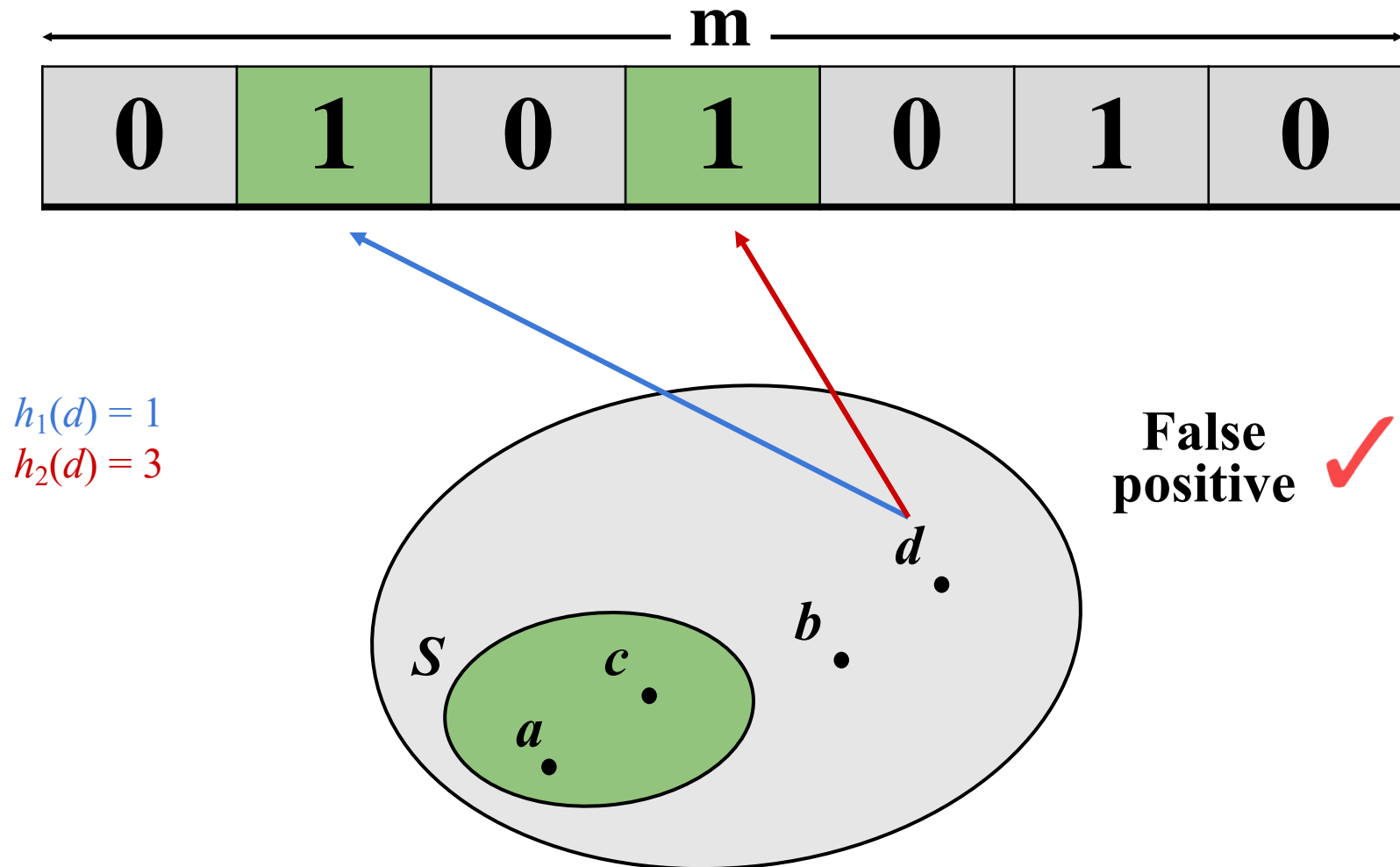
Classic filter: The Bloom filter [Bloom '70]

Bloom filter: a bit array + k hash functions (here $k=2$)



Classic filter: The Bloom filter [Bloom '70]

Bloom filter: a bit array + k hash functions (here $k=2$)



Bloom filters are ubiquitous (> 4300 citations)

Streaming applications



Networking



Databases



Computational biology



Storage systems



Bloom filters have suboptimal performance

	Bloom filter	Optimal
Space (bits)	$\approx 1.44 n \log(1/\epsilon)$	$\approx n \log(1/\epsilon) + \Omega(n)$
CPU cost	$\Omega(1/\epsilon)$	$O(1)$
Data locality	$\Omega(1/\epsilon)$ probes	$O(1)$ probes

Applications often work around Bloom filter limitations

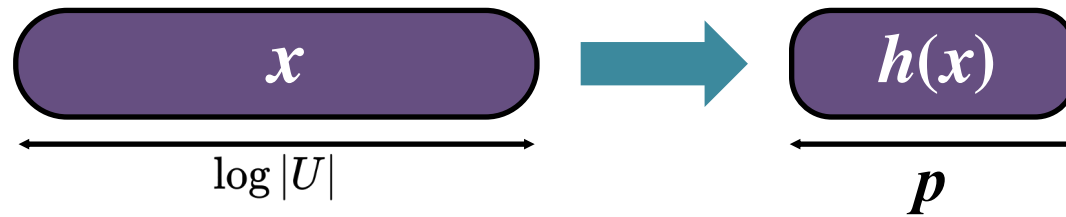
Limitations	Workarounds
No deletes	Rebuild
No resizes	Guess N , and rebuild if wrong
No filter merging or enumeration	???
No values associated with keys	Combine with another data structure

Bloom filter limitations increase system complexity, waste space, and slow down application performance

Quotienting is an alternative to Bloom filters

[Knuth. Searching and Sorting Vol. 3, '97]

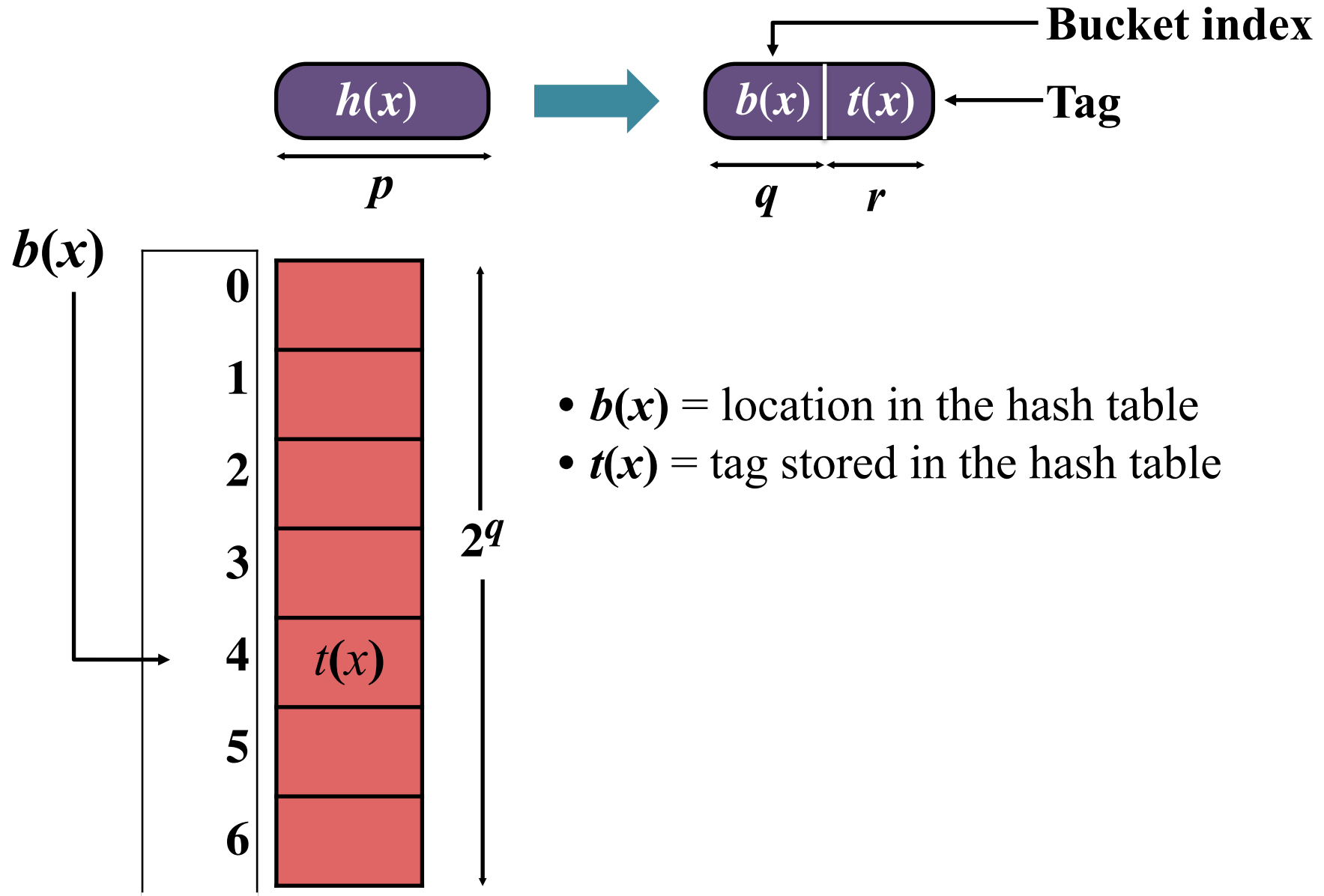
- **Store fingerprints compactly in a hash table.**
 - Take a fingerprint $h(x)$ for each element x .



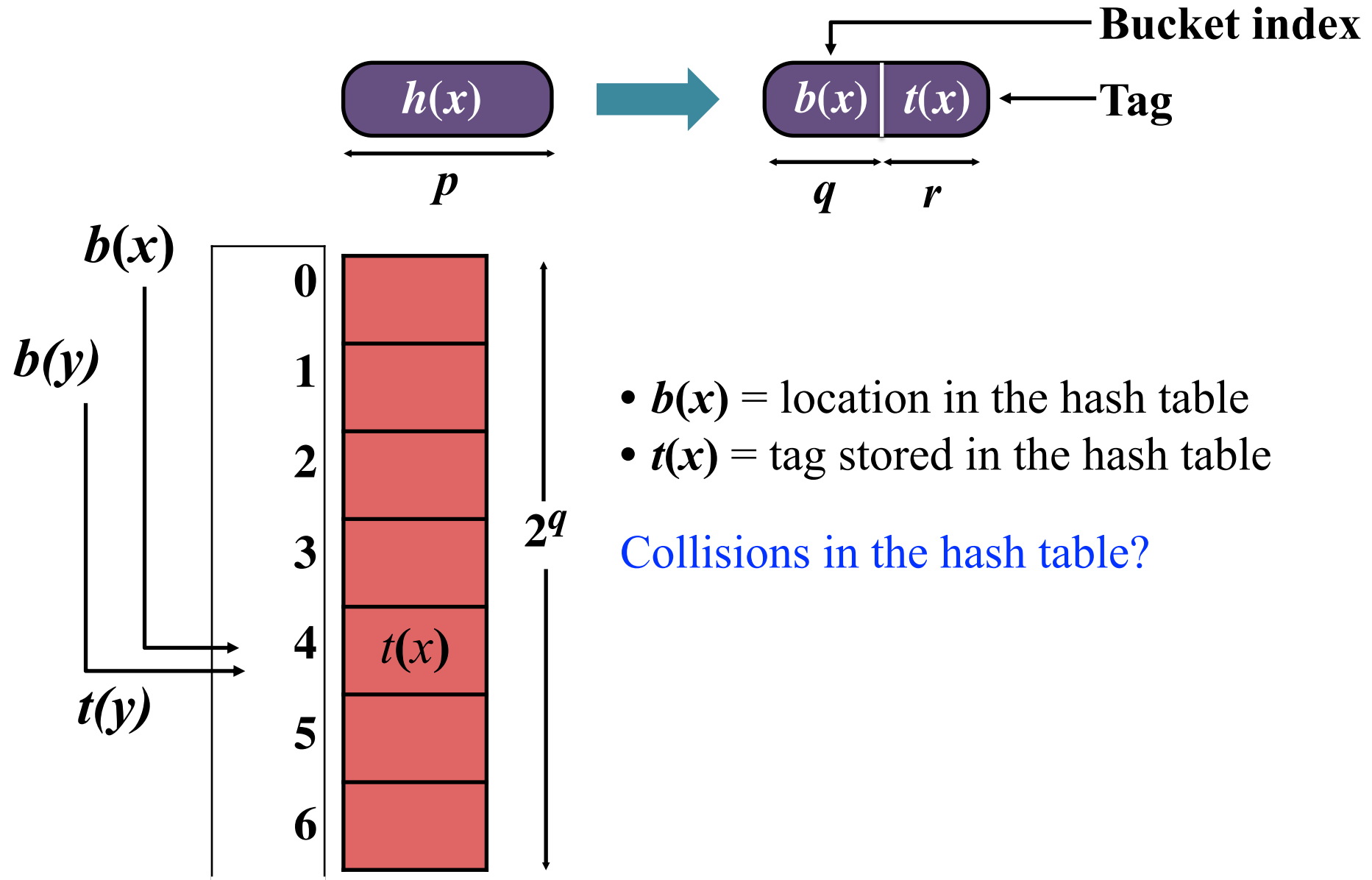
- **Only source of false positives:**
 - Two distinct elements x and y , where $h(x) = h(y)$
 - If x is stored and y isn't, $\text{query}(y)$ gives a false positives

$$\Pr[x \text{ and } y \text{ collide}] = \frac{1}{2^p}$$

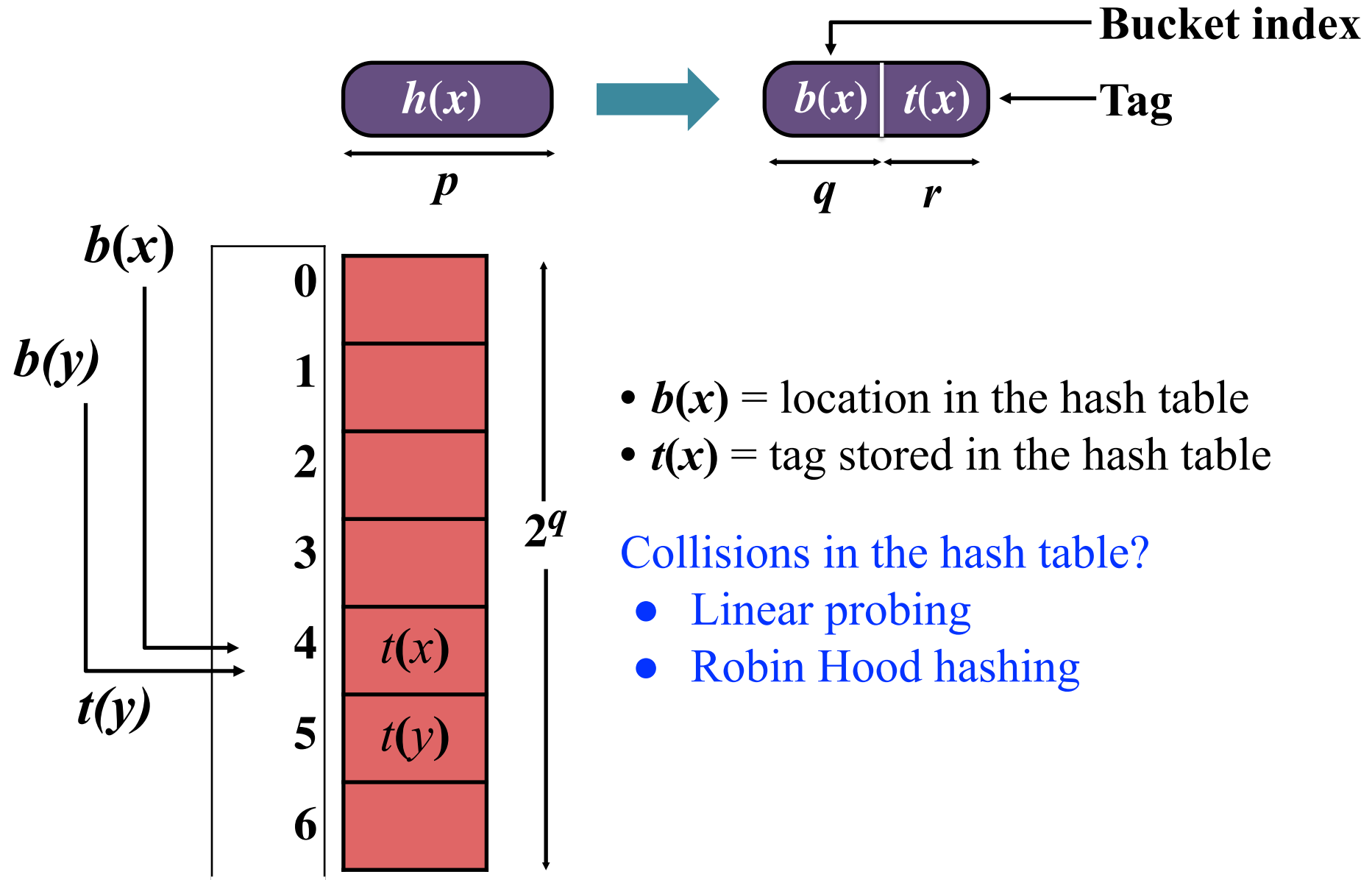
Storing fingerprints compactly



Storing fingerprints compactly



Storing fingerprints compactly

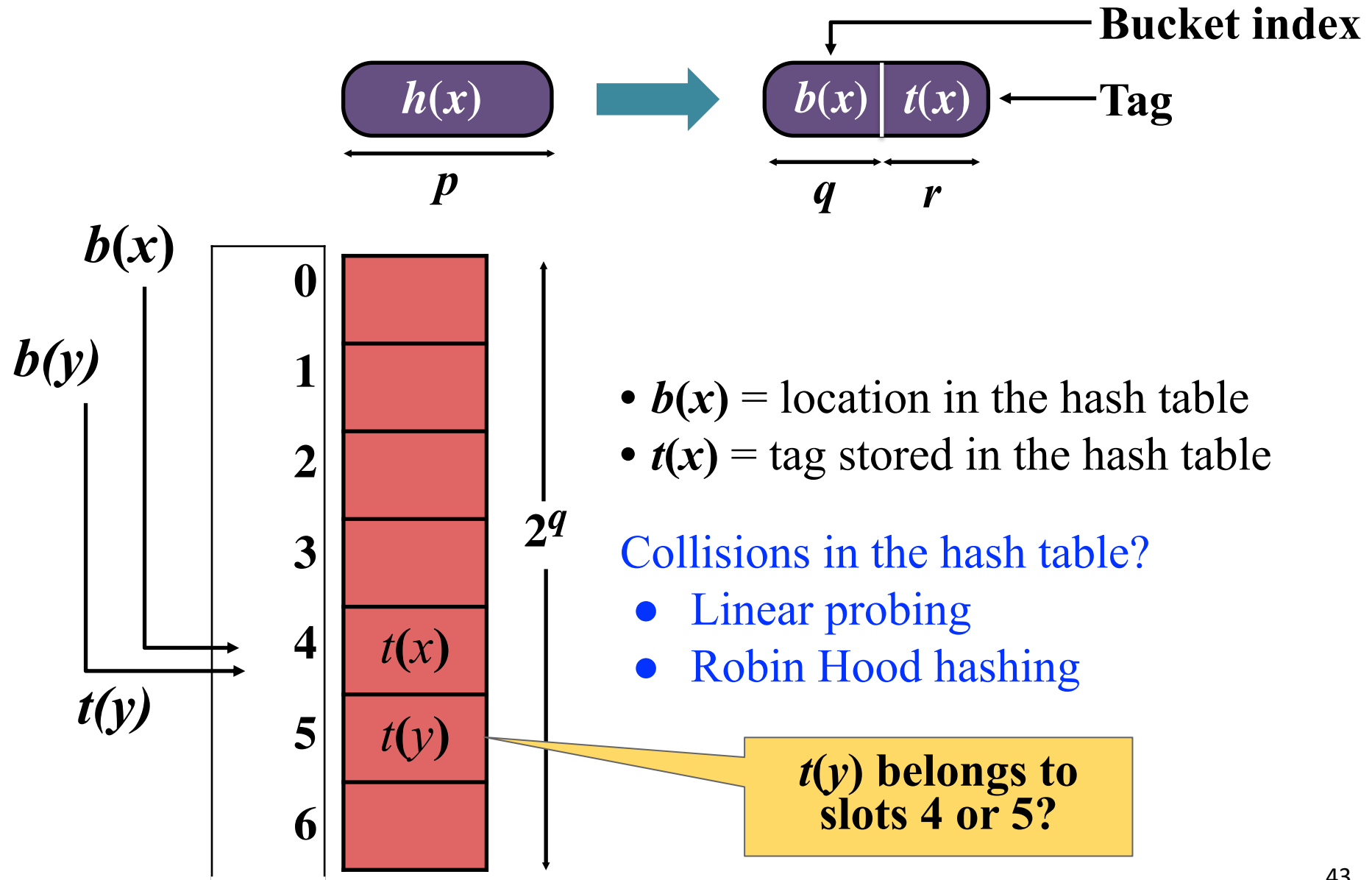


- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

Collisions in the hash table?

- Linear probing
- Robin Hood hashing

Storing fingerprints compactly



Resolving collisions in the QF [Pandey et al. SIGMOD '17]

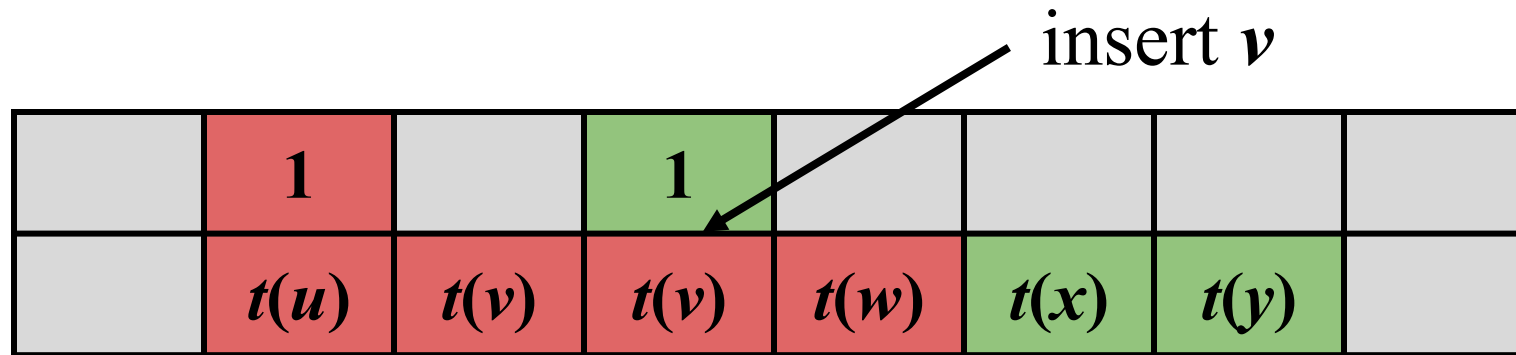
- QF uses two metadata bits to resolve collisions and identify home bucket

	1		1				
	$t(u)$	$t(v)$	$t(w)$	$t(x)$	$t(y)$		

- The metadata bits group tags by their home bucket

Resolving collisions in the QF [Pandey et al. SIGMOD '17]

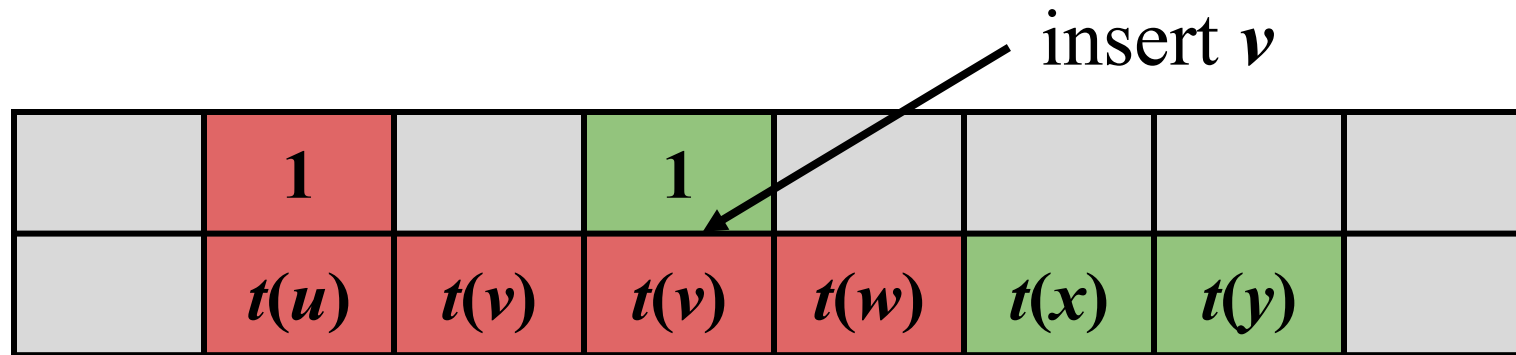
- QF uses two metadata bits to resolve collisions and identify home bucket



- The metadata bits group tags by their home bucket

Resolving collisions in the QF [Pandey et al. SIGMOD '17]

- QF uses two metadata bits to resolve collisions and identify home bucket



- The metadata bits group tags by their home bucket

The metadata bits enable us to identify the slots holding the contents of each bucket.

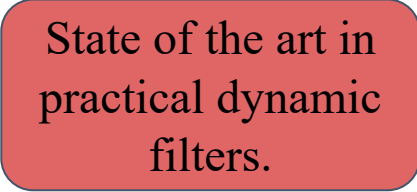
Quotient filters use less space than Bloom filters for all practical configurations

	Quotient filter	Bloom filter	Optimal
Space (bits)	$\approx n \log(1/\epsilon) + 2.125n$	$\approx 1.44 n \log(1/\epsilon)$	$\approx n \log(1/\epsilon) + \Omega(n)$
CPU cost	$O(1)$ expected	$\Omega(1/\epsilon)$	$O(1)$
Data locality	1 probe + scan	$\Omega(1/\epsilon)$ probes	$O(1)$ probes

The quotient filter has theoretical advantages over the Bloom filter

Types of filters

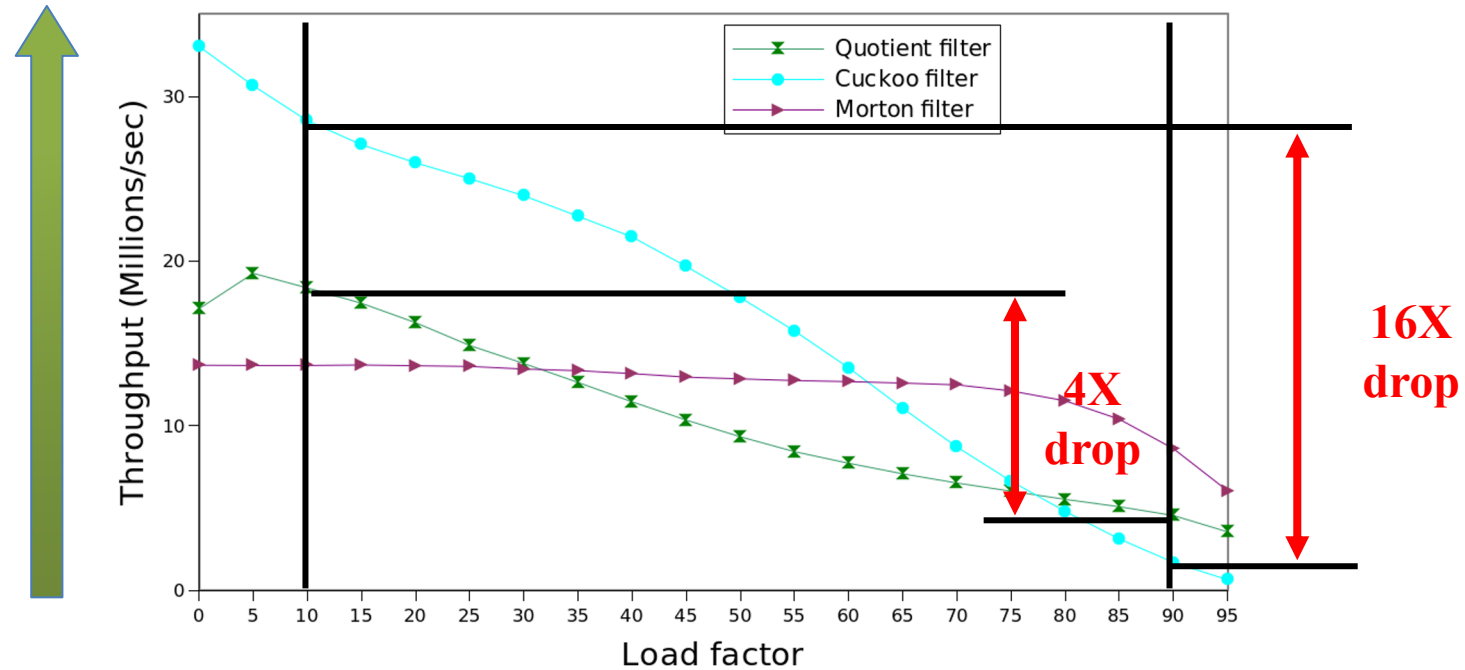
- Bloom filters [Bloom '70]
- Quotient filters [Pagh et al. '05, Dillinger et al. '09, Bender et al. '12, Einziger et al. '15, Pandey et al. '17]
- Cuckoo/Morton filters [Fan et al. '14, Breslow & Jayasena '18]
- Others
 - Mostly based on perfect hashing and/or linear algebra
 - Mostly static
 - e.g.,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



State of the art in practical dynamic filters.

Current filters have a problem..

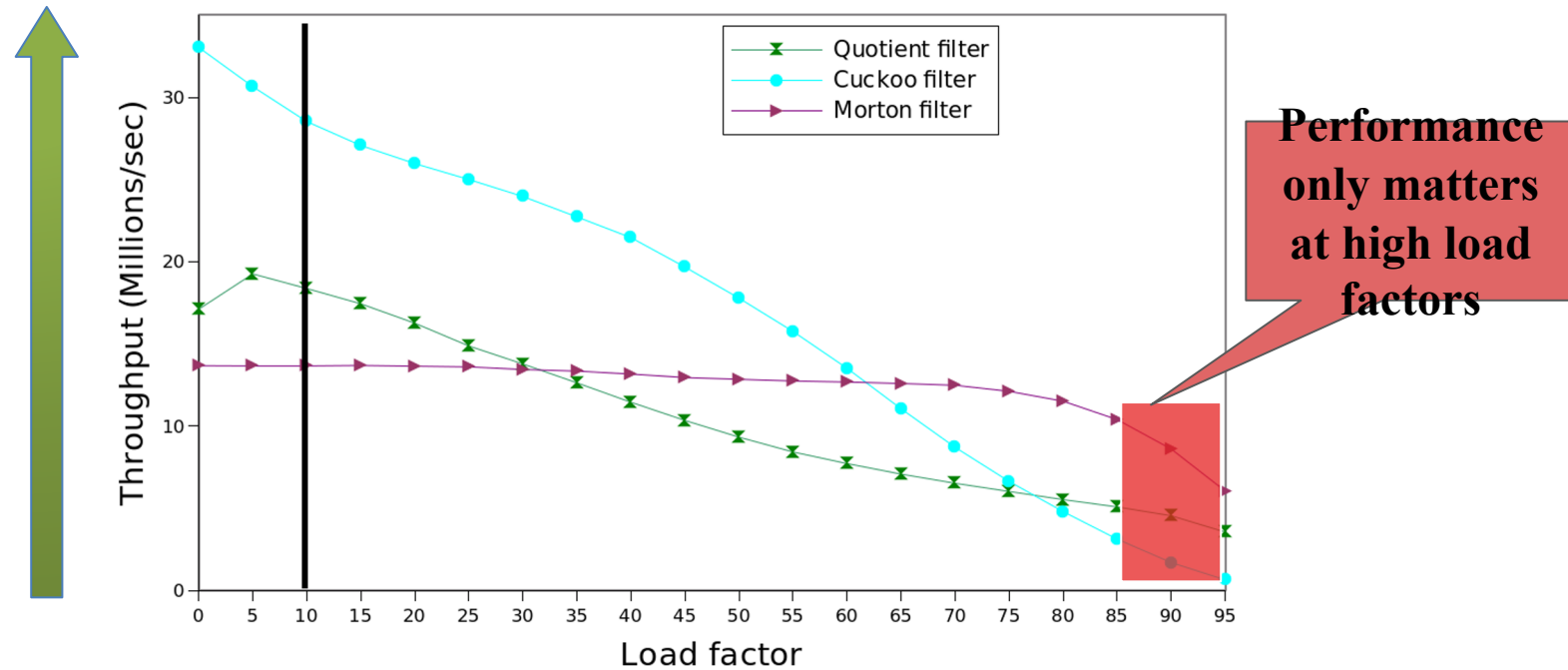
Performance suffers due to high-overhead of *collision resolution*



Applications must choose between space and speed.

Current filters have a problem..

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



Update intensive applications maintain filters close to full.

Why quotient filters slow down

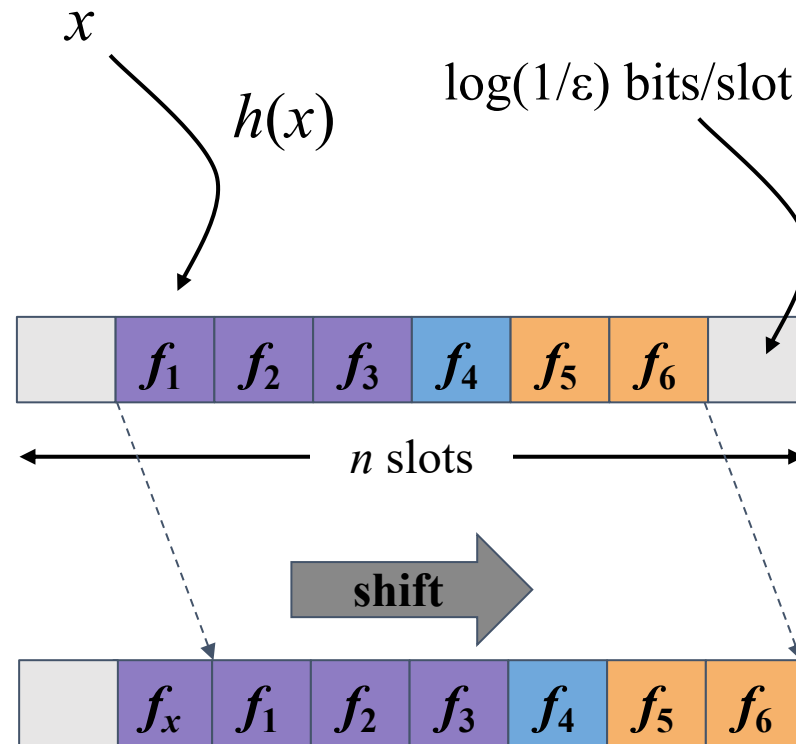
Quotient filters use Robin-Hood hashing (a variant of linear probing)

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 $f(x)$.

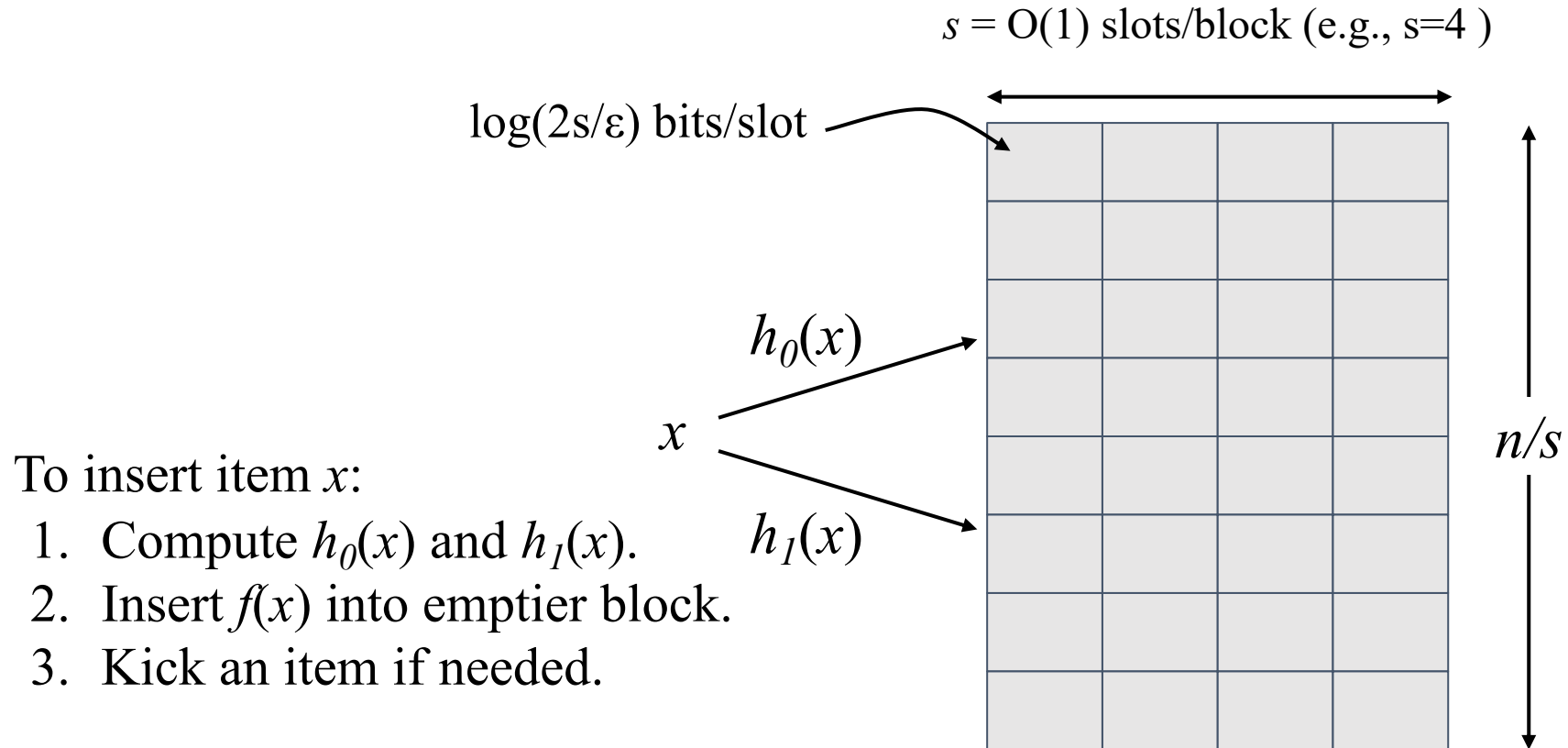
As the QF fills, inserts have to do more shifting.



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

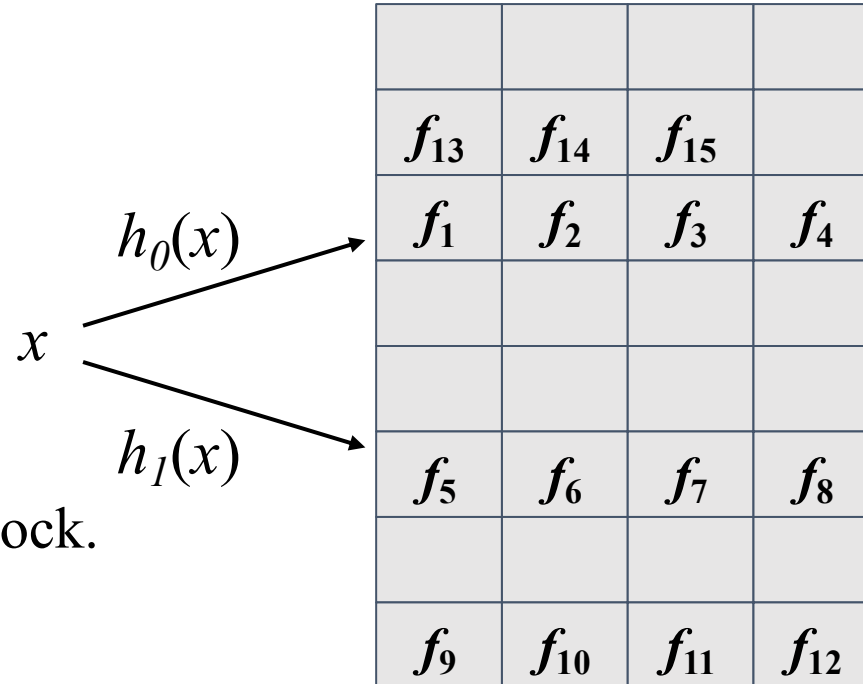
Why cuckoo filters slow down



Why cuckoo filters slow down

To insert item x :

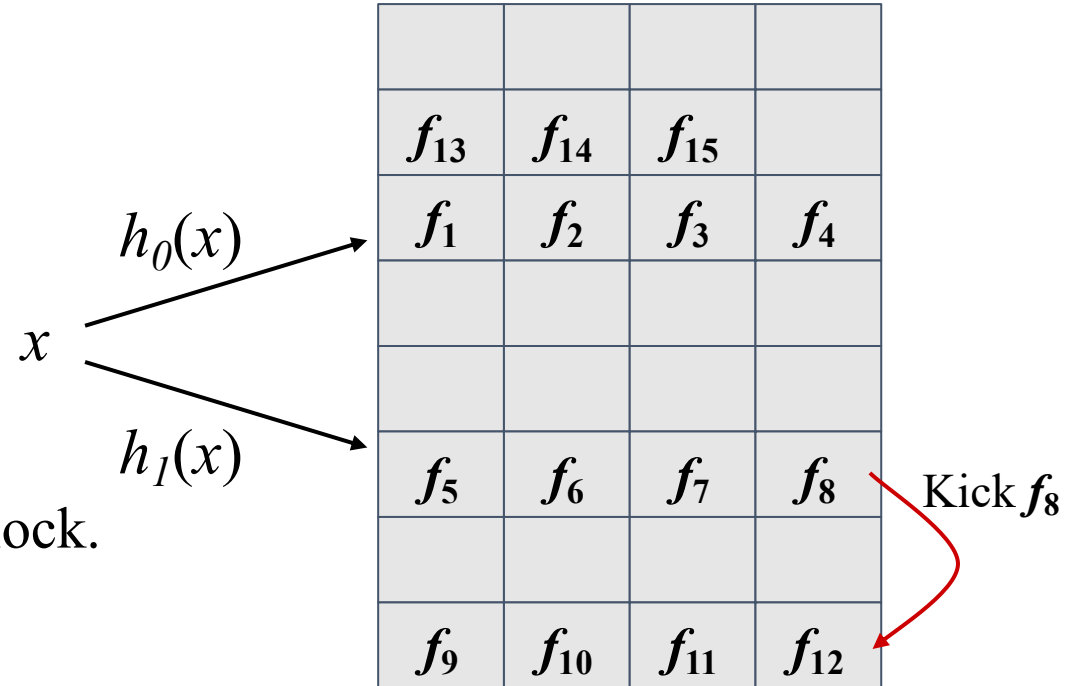
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.



Why cuckoo filters slow down

To insert item x :

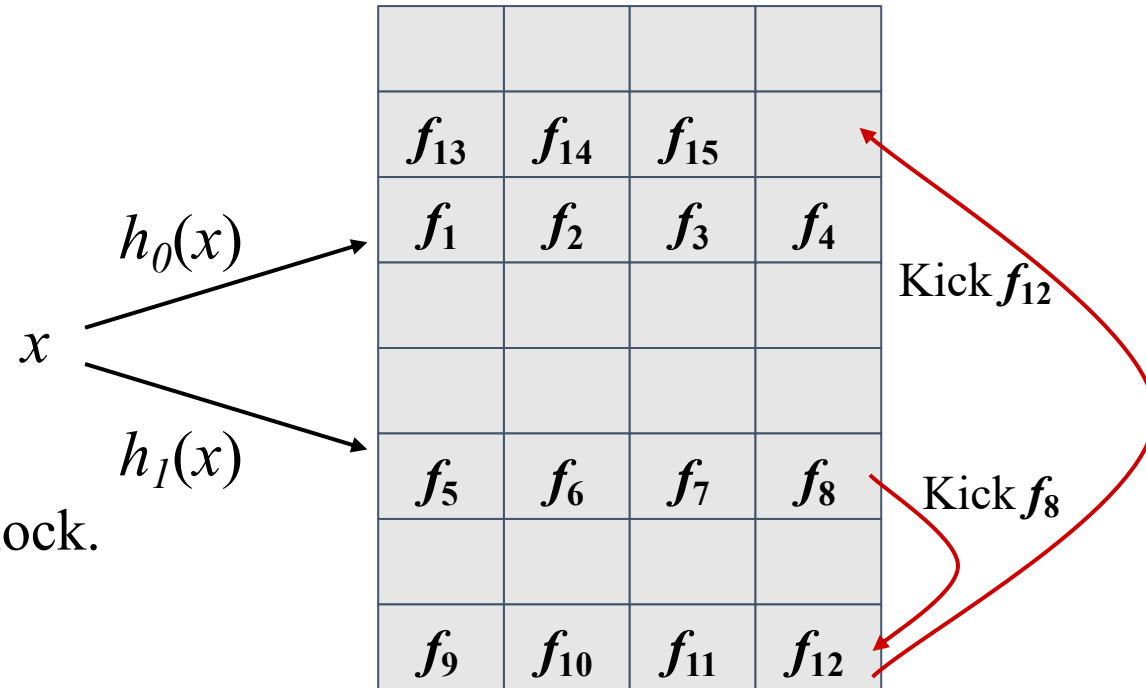
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.



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.

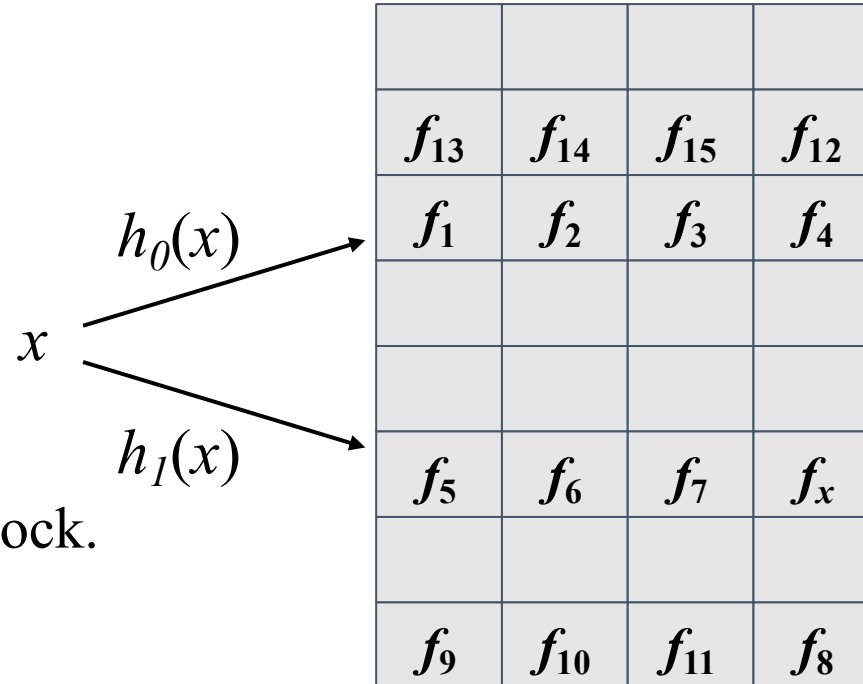


Note: $h_0(x)$ and $h_1(x)$ need to be dependent to support kicking.

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.

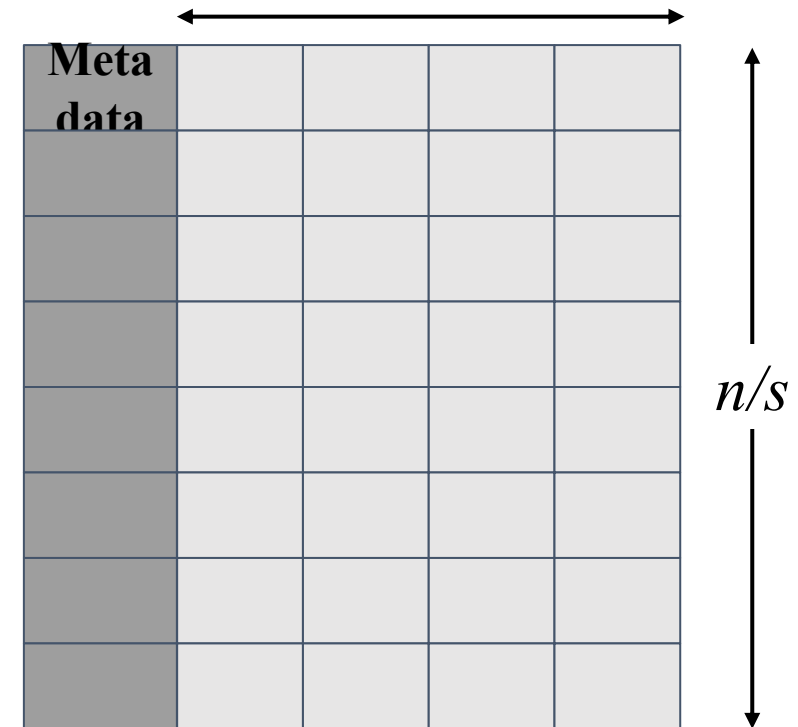
Note: $h_0(x)$ and $h_1(x)$ need to be dependent to support kicking.

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

Vector quotient filter design

$s = \omega(\log \log n)$ slots/block (e.g., $s=64$)

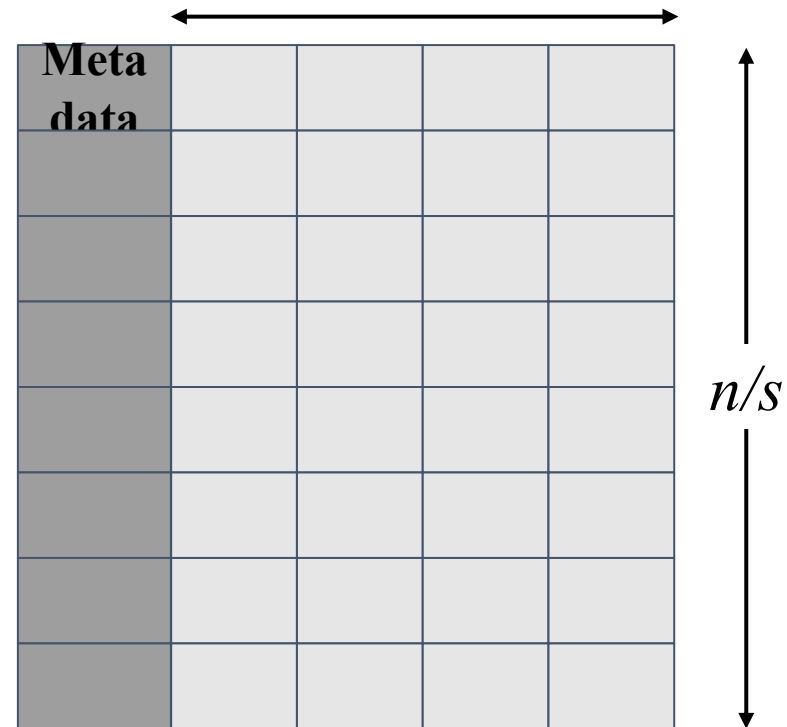


Vector quotient filter design

Each block is a small quotient filter with false-positive rate $\epsilon/2$ and capacity s .

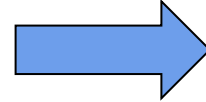


$s = \omega(\log \log n)$ slots/block (e.g., $s=64$)

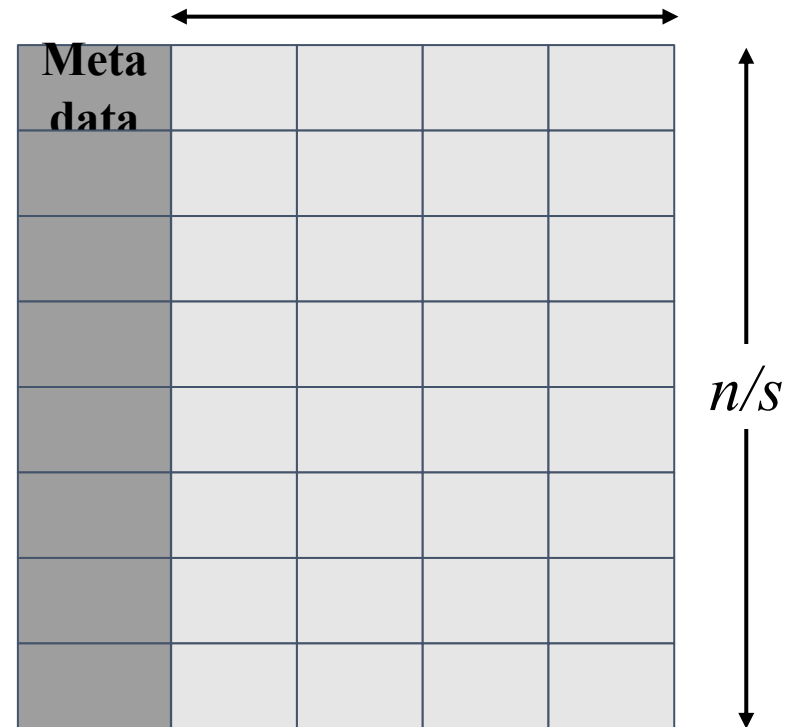


Vector quotient filter design

Each block is a small quotient filter with false-positive rate $\varepsilon/2$ and capacity s .



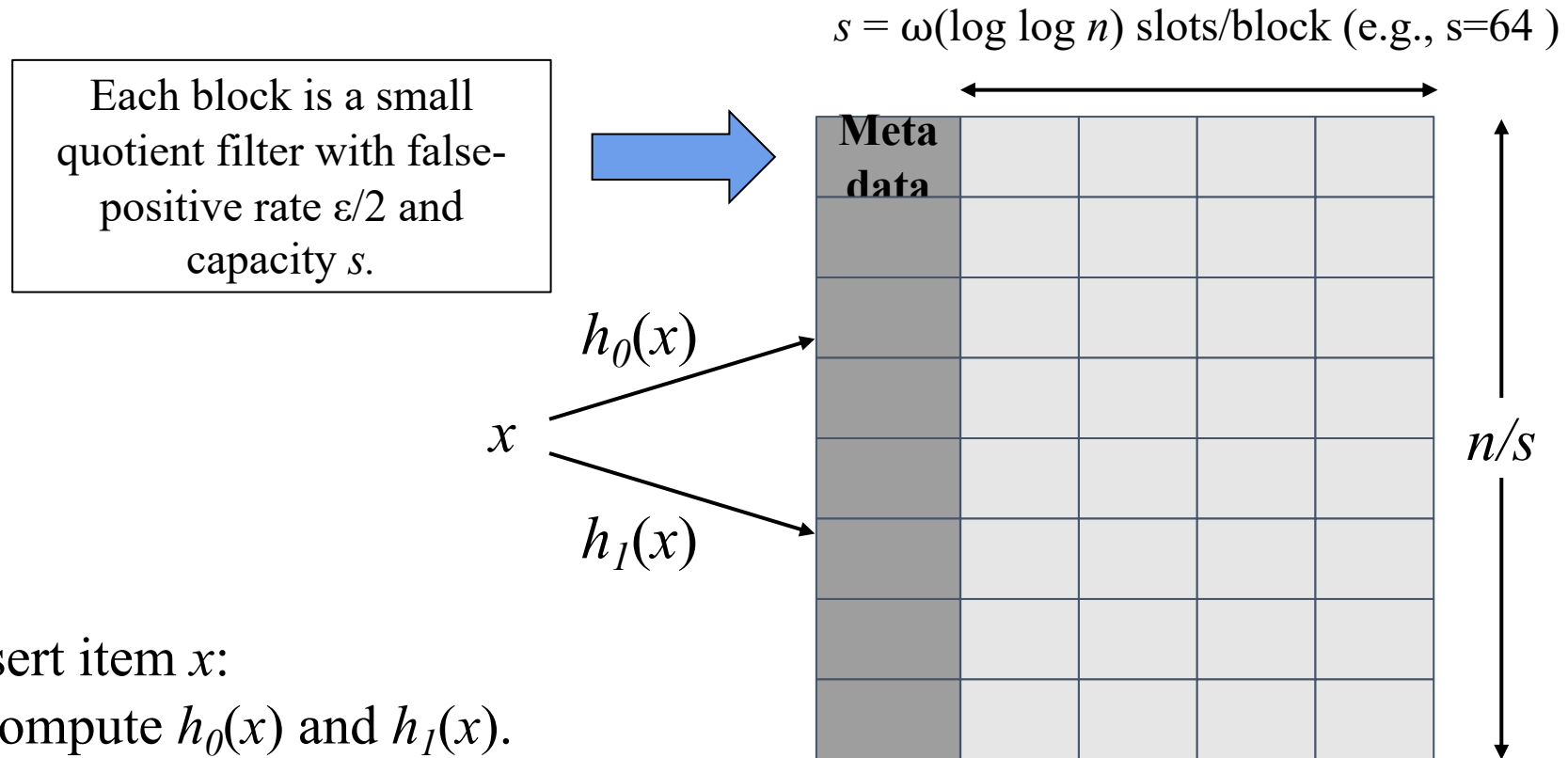
$s = \omega(\log \log n)$ slots/block (e.g., $s=64$)



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

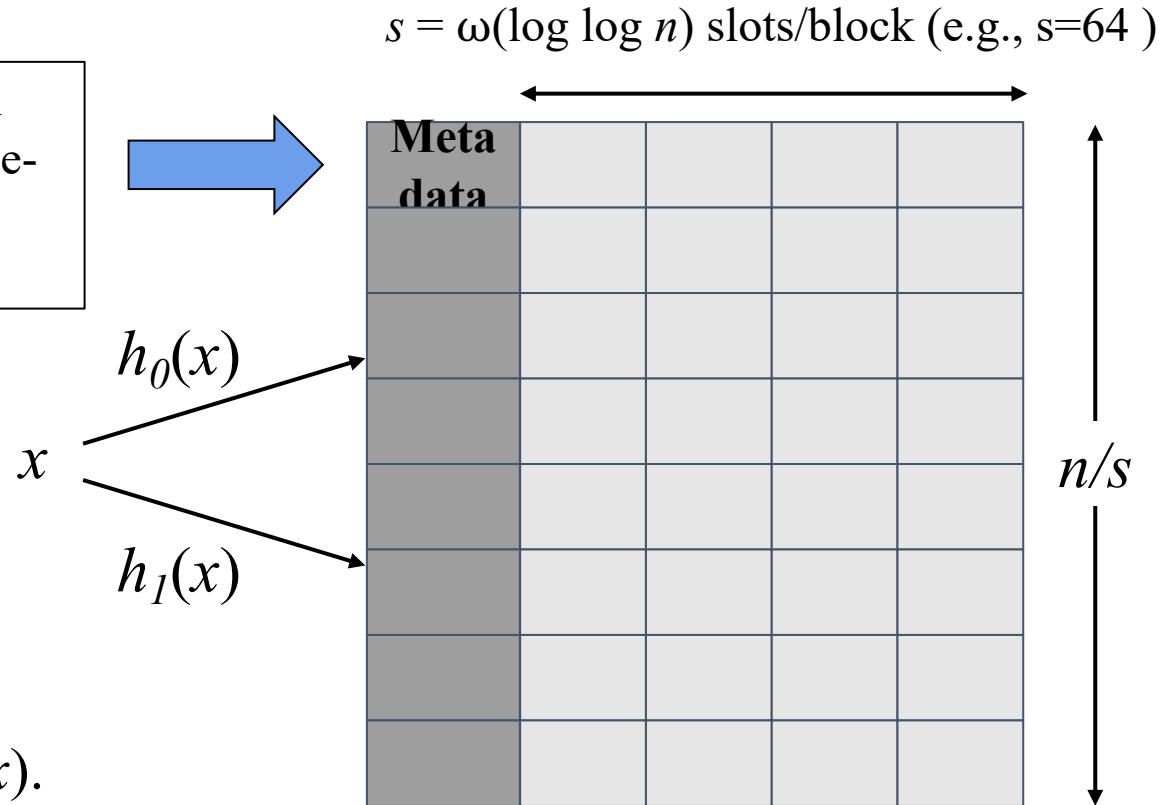


To insert item x :

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 $\epsilon/2$ and capacity s .



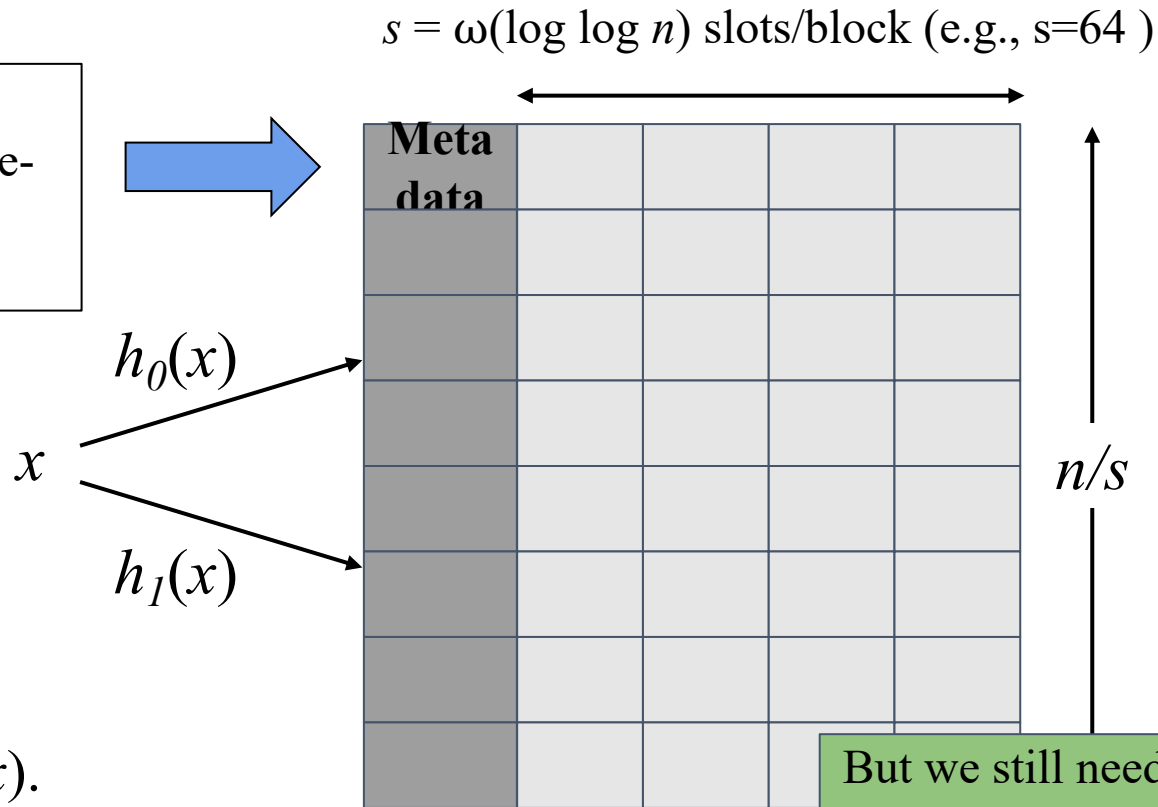
To insert item x :

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

No kicking $\Rightarrow h_0(x)$ 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 $\epsilon/2$ and capacity s .



To insert item x :

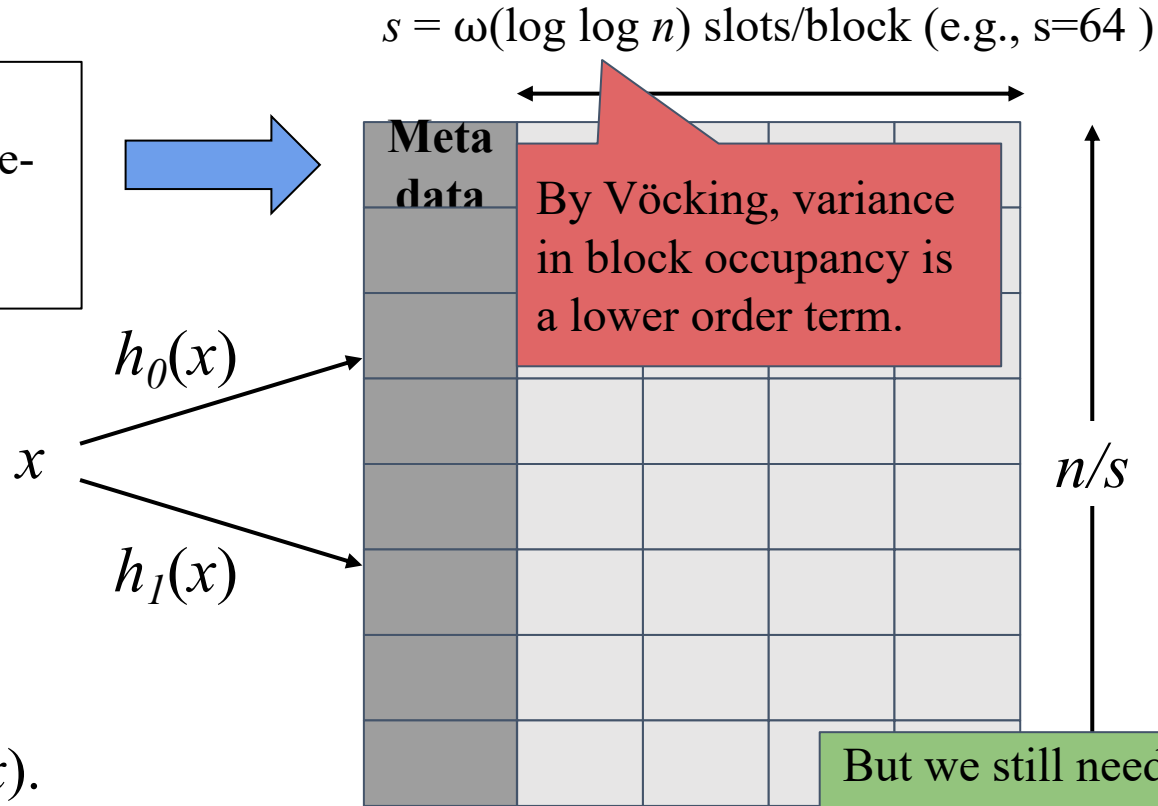
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

No kicking $\Rightarrow h_0(x)$ 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 $\epsilon/2$ 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 $\Rightarrow h_0(x)$ and $h_1(x)$ can be independent for insert-only workload.

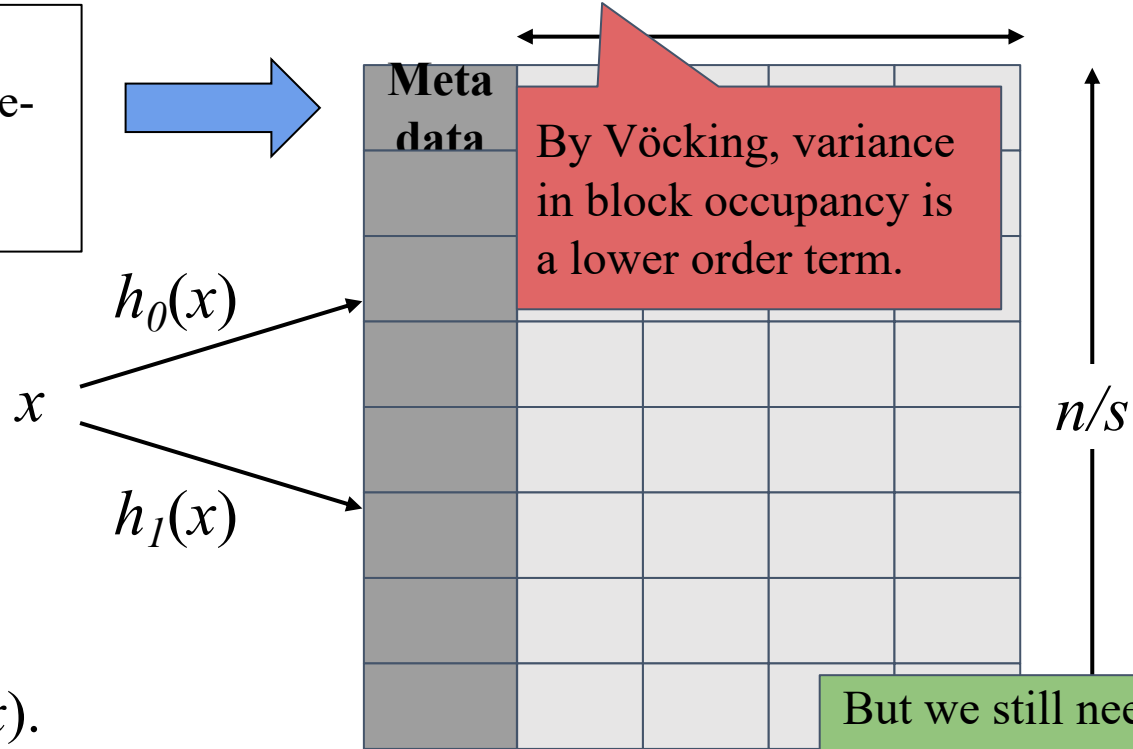
But we still need it to support deletes.

Vector quotient filter design

Each block is a small quotient filter with false-positive rate $\epsilon/2$ and capacity s .

No kicking \Rightarrow easier concurrency

$s = \omega(\log \log n)$ slots/block (e.g., $s=64$)



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 $\Rightarrow h_0(x)$ and $h_1(x)$ can be independent for insert-only workload.

But we still need it to support deletes.

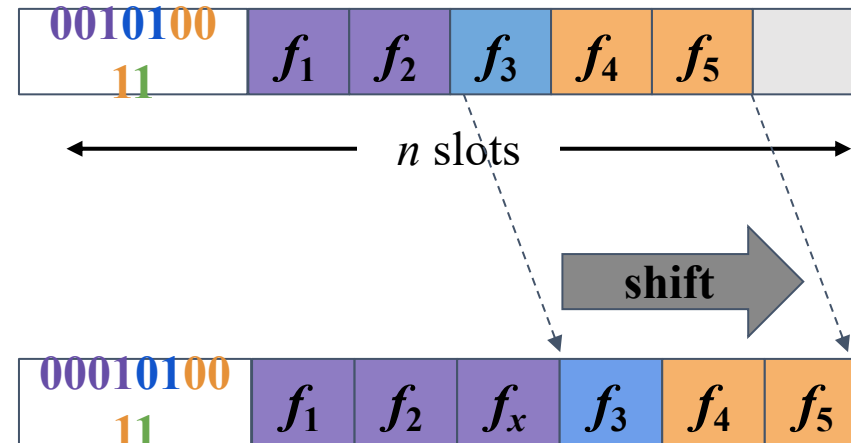
A vectorizable mini quotient filter

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 $b=s/\ln 2$.

Implemented using PDEP

Implemented using PSHUFB or VCMPPB

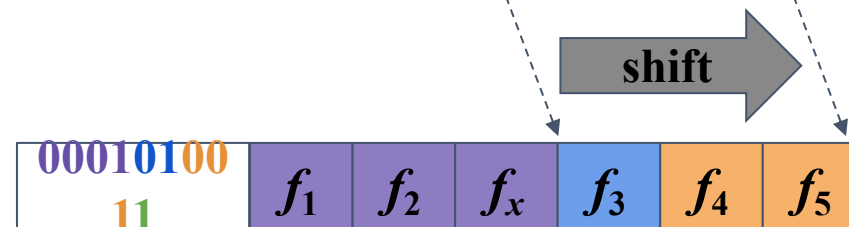
A vectorizable mini quotient filter

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 $\omega(\log \log n)$ [Belloch '90].
Example, using AVX-512 instructions.**

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



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

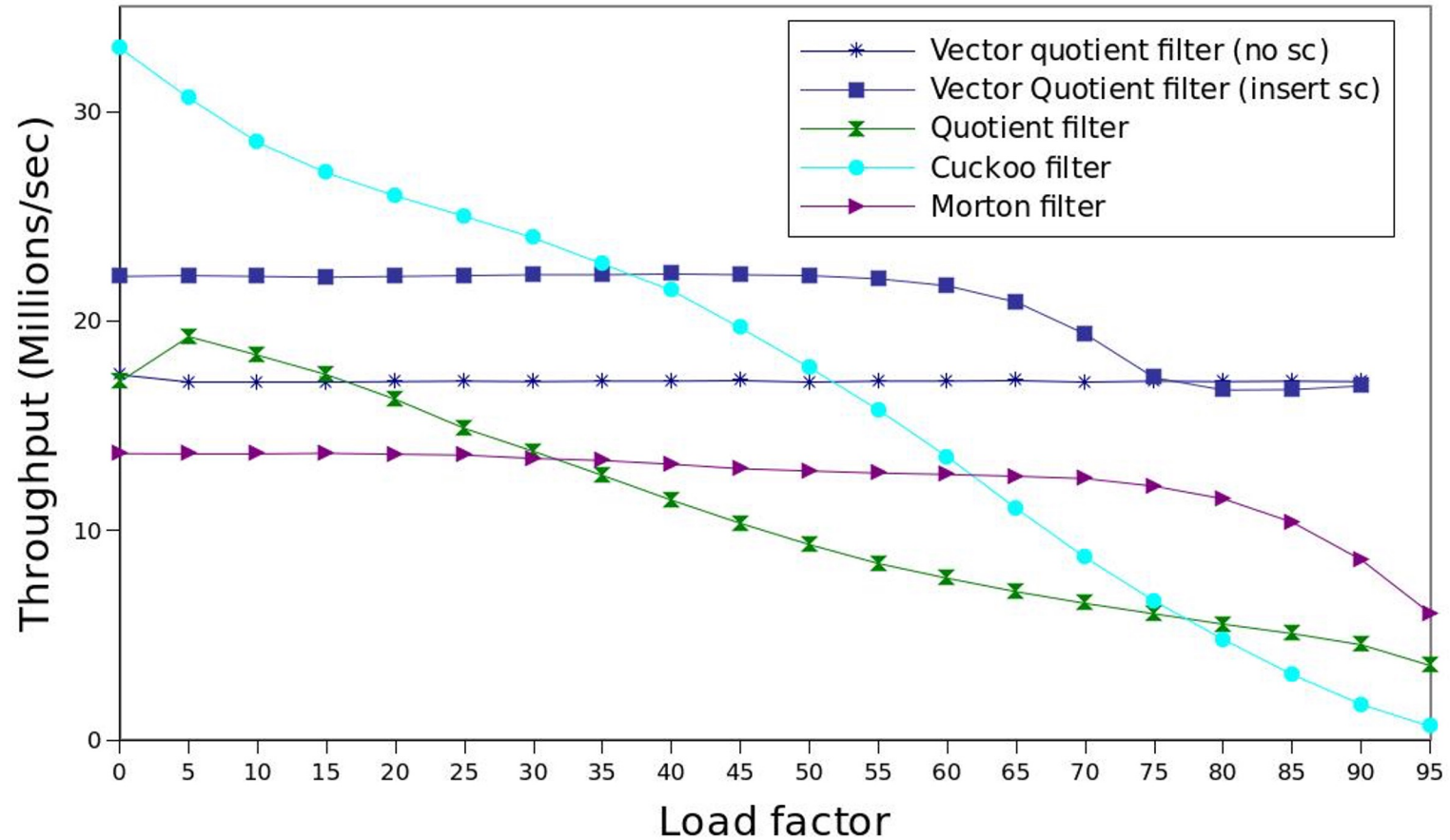
Implemented using PDEP

Implemented using PSHUFB or VCMPPB

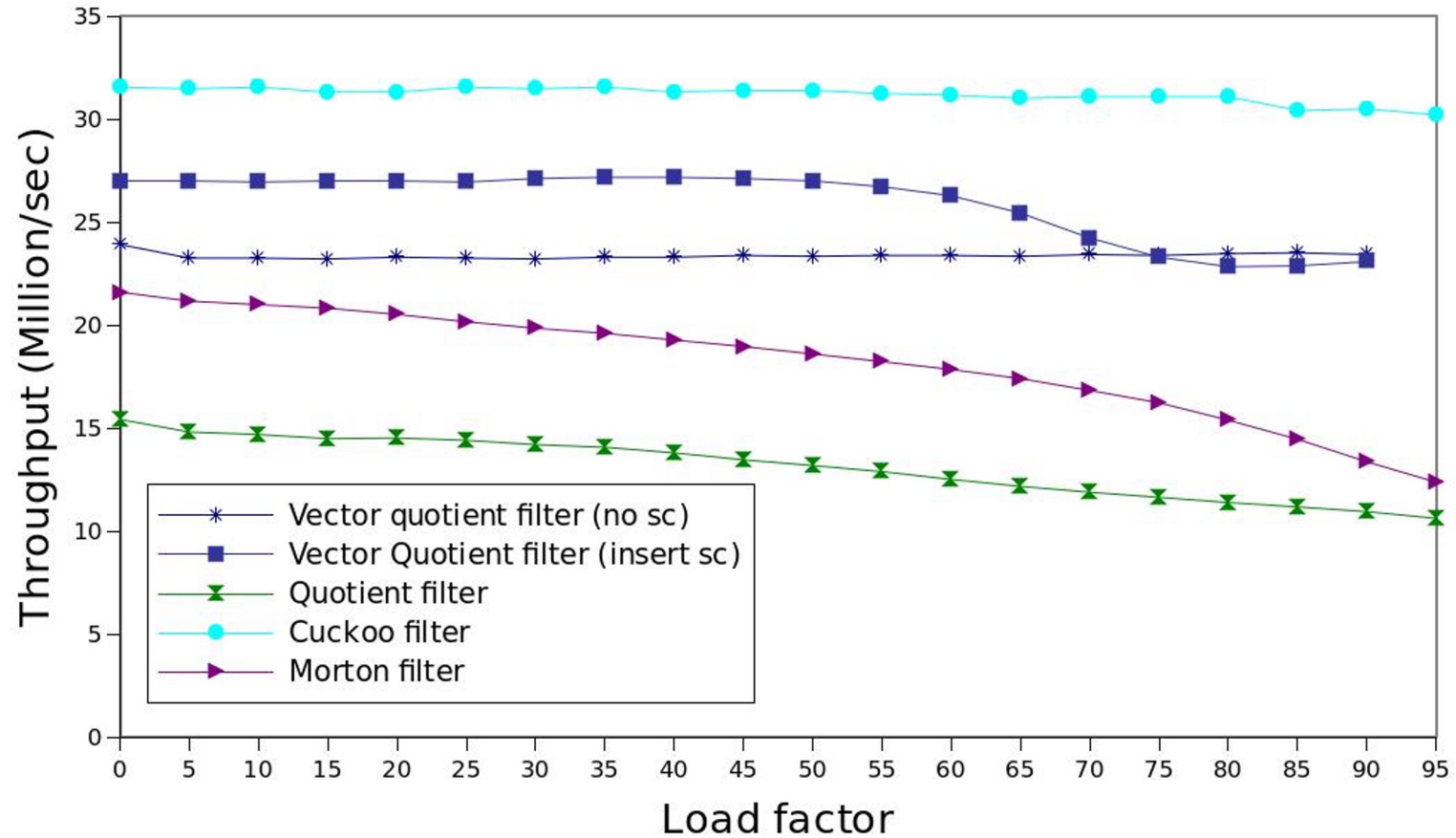
Vector quotient filter (VQF) performance

	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

