CS 6530: Advanced Database Systems Fall 2024

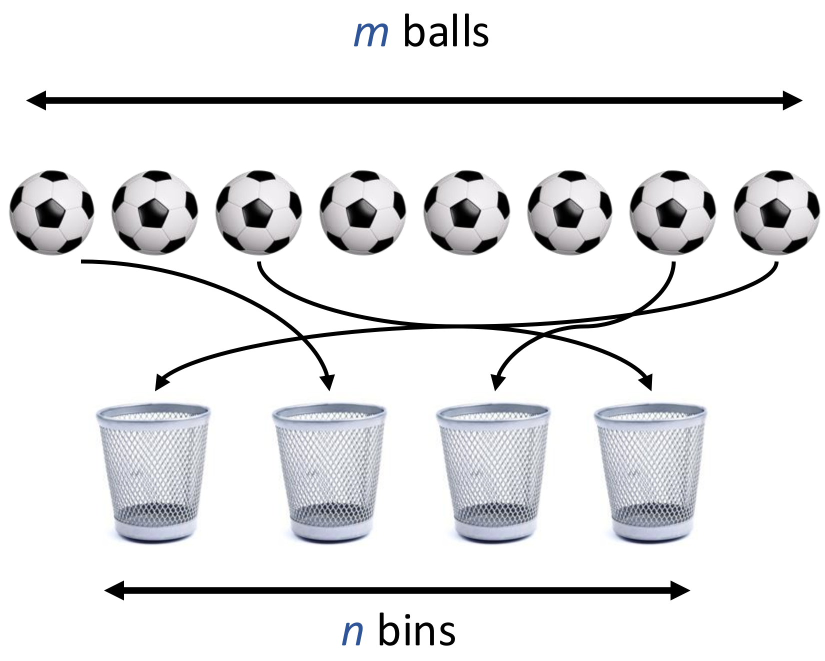
Lecture 12
Filters

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	$(1 + o(1))\frac{\log n}{\log \log n}$	$\frac{m}{n} + \sqrt{\frac{m \log n}{n}}$

Acknowledgement: Udi Wieder

The multiple choice paradigm

- Throw *m* balls into *n* bins:
  - Pick *d* bins uniformly at random (*d* >= 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

Acknowledgement: Udi Wieder

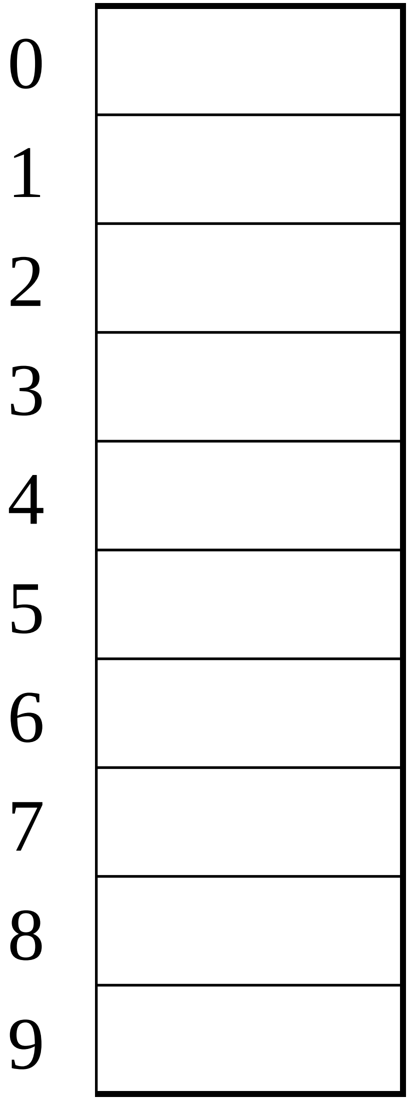
Collision Resolution

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)

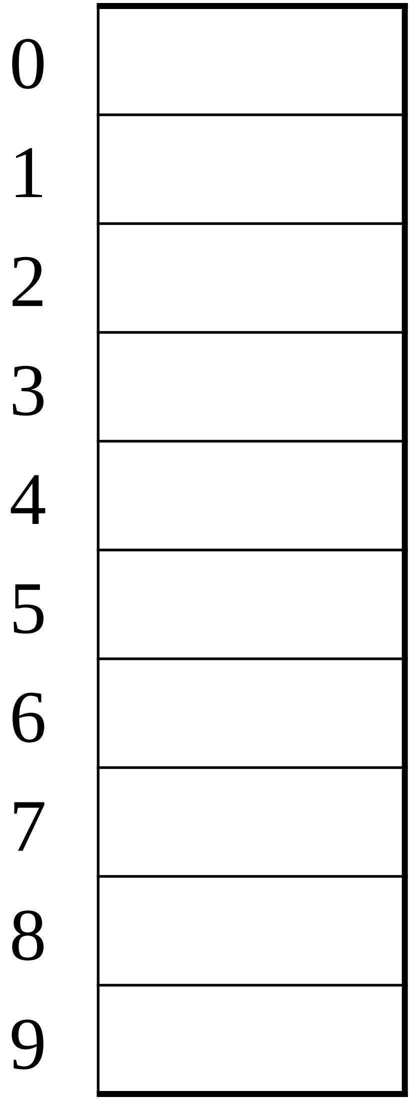
Separate Chaining



Insert:
10
22
107
12
42

- **<u>Separate chaining</u>**: All keys that map to the same hash value are kept in a list (or "bucket").

Open Addressing



Insert:
38
19
8
109
10

- **<u>Linear Probing</u>**: 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

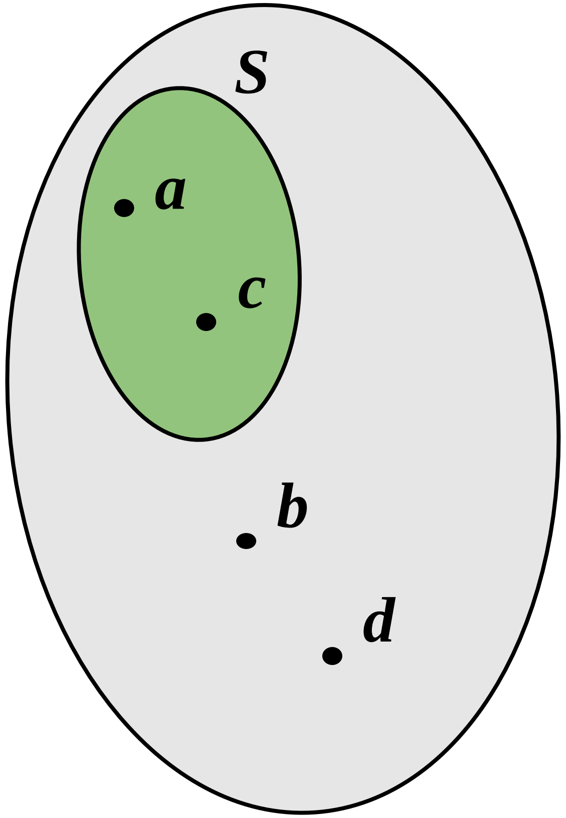
Open addressing

- 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***

Dictionary data structure

A dictionary maintains a set S from universe U.



membership(a): ✓

membership(b): ✗

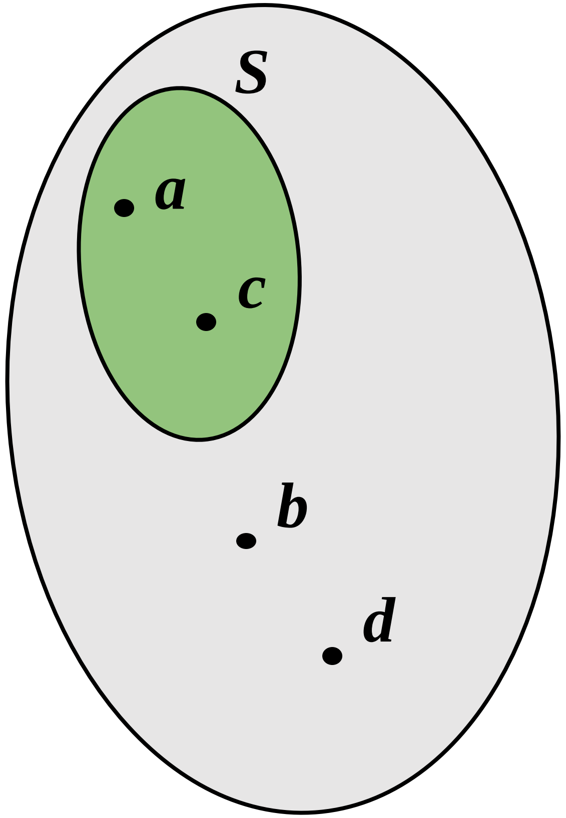
membership(c): ✓

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:

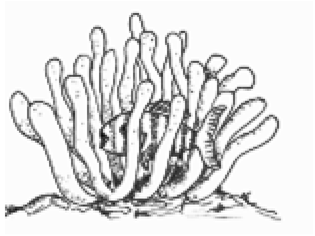
- ✗ with probability $> 1 - \varepsilon$ — **true negative**
- ✓ with probability $\leq \varepsilon$ — **false positive**

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

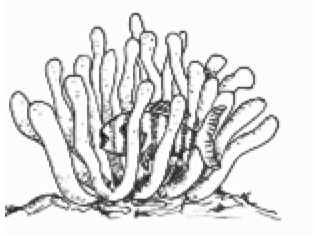
False-positive rate enables filters to be compact

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

Small

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

Large



Filter

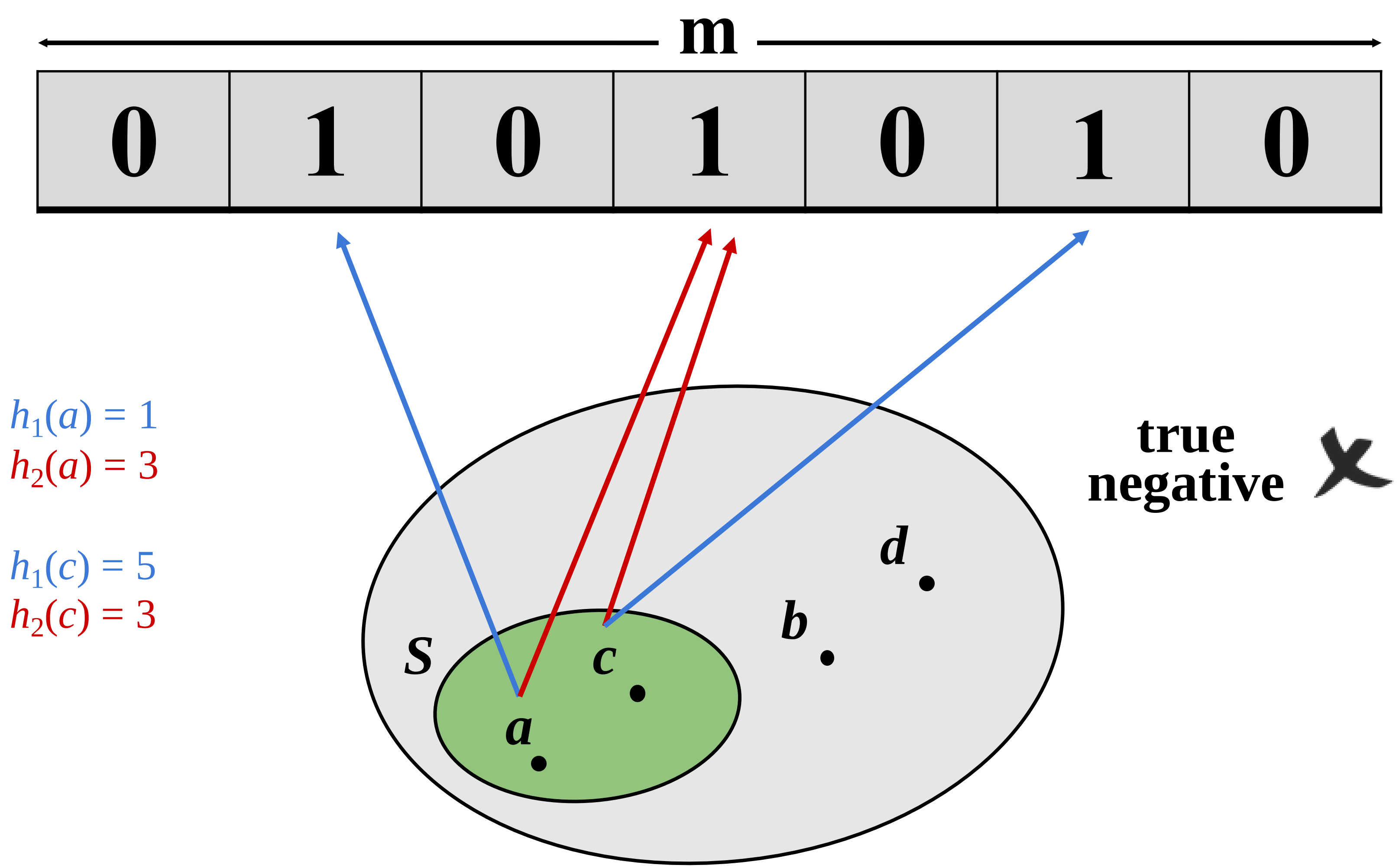
Dictionary

For most practical purposes:

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

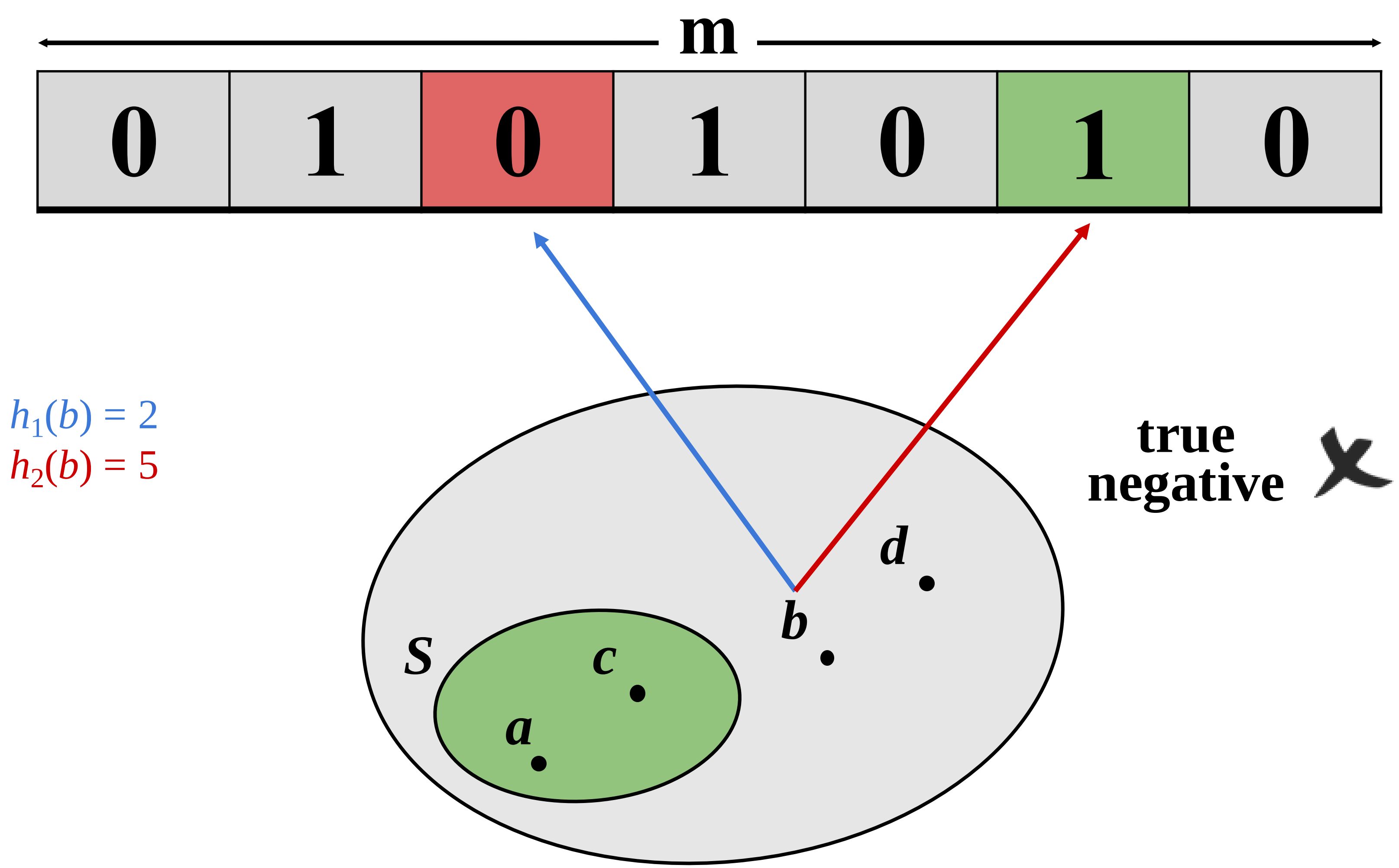
Classic filter: The Bloom filter [Bloom '70]

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



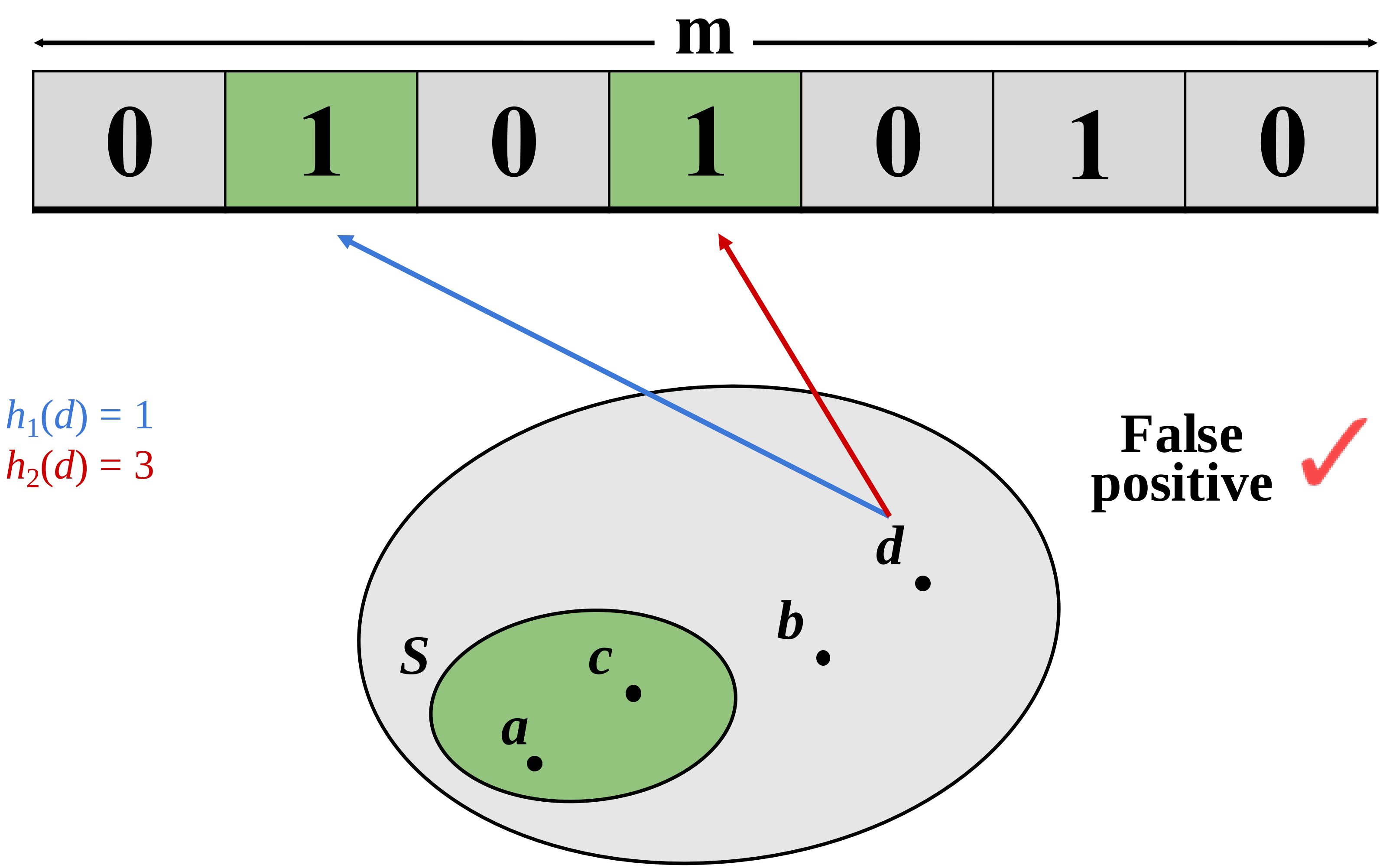
Classic filter: The Bloom filter [Bloom '70]

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 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

Bloom filters are ubiquitous (> 10K citations)

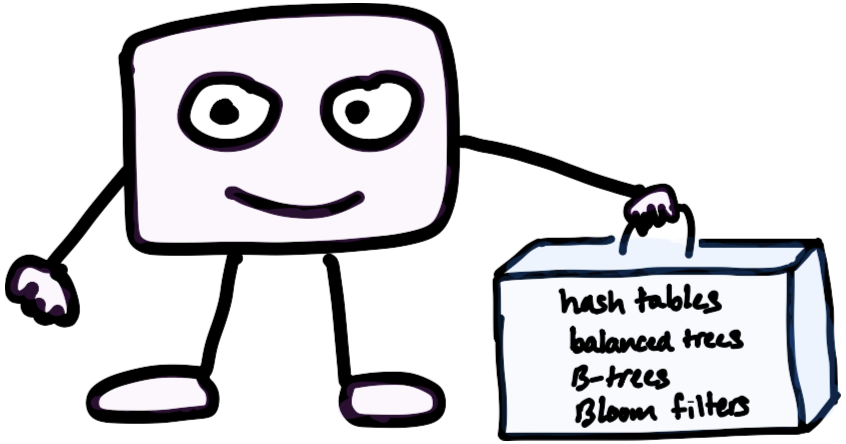
Computational biology

Databases

Networking

Storage systems

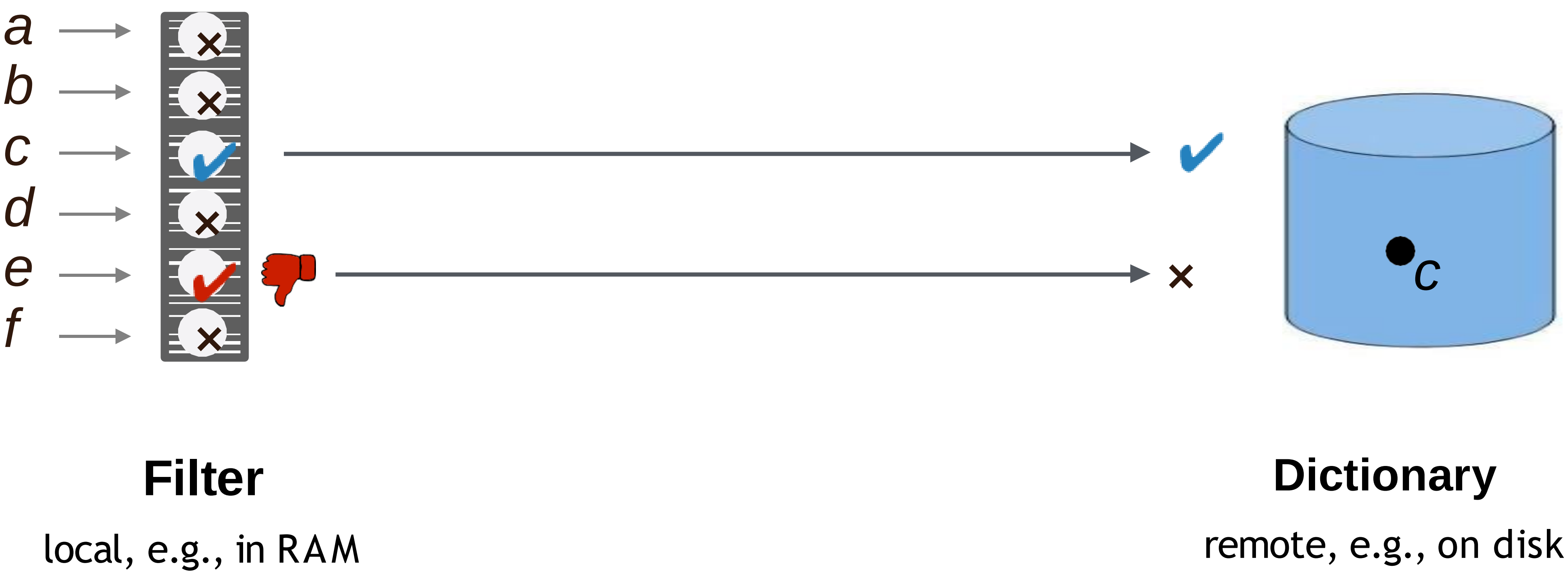
Streaming applications



Most common filter use

Filter out queries to a large remote dictionary.

Only an ε-fraction of negative queries don't get filtered out.



Speed up from filter use

Workload has *P* positive and *N* negative queries.

Dictionaries w/o Bloom Filters	Dictionaries w/ Bloom Filters
$P+N$	$P+\varepsilon N$

Remote Accesses of Dictionary

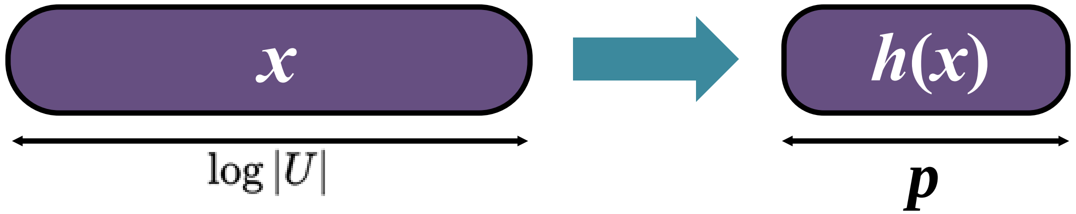
Applications often work around Bloom filter limitations

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]

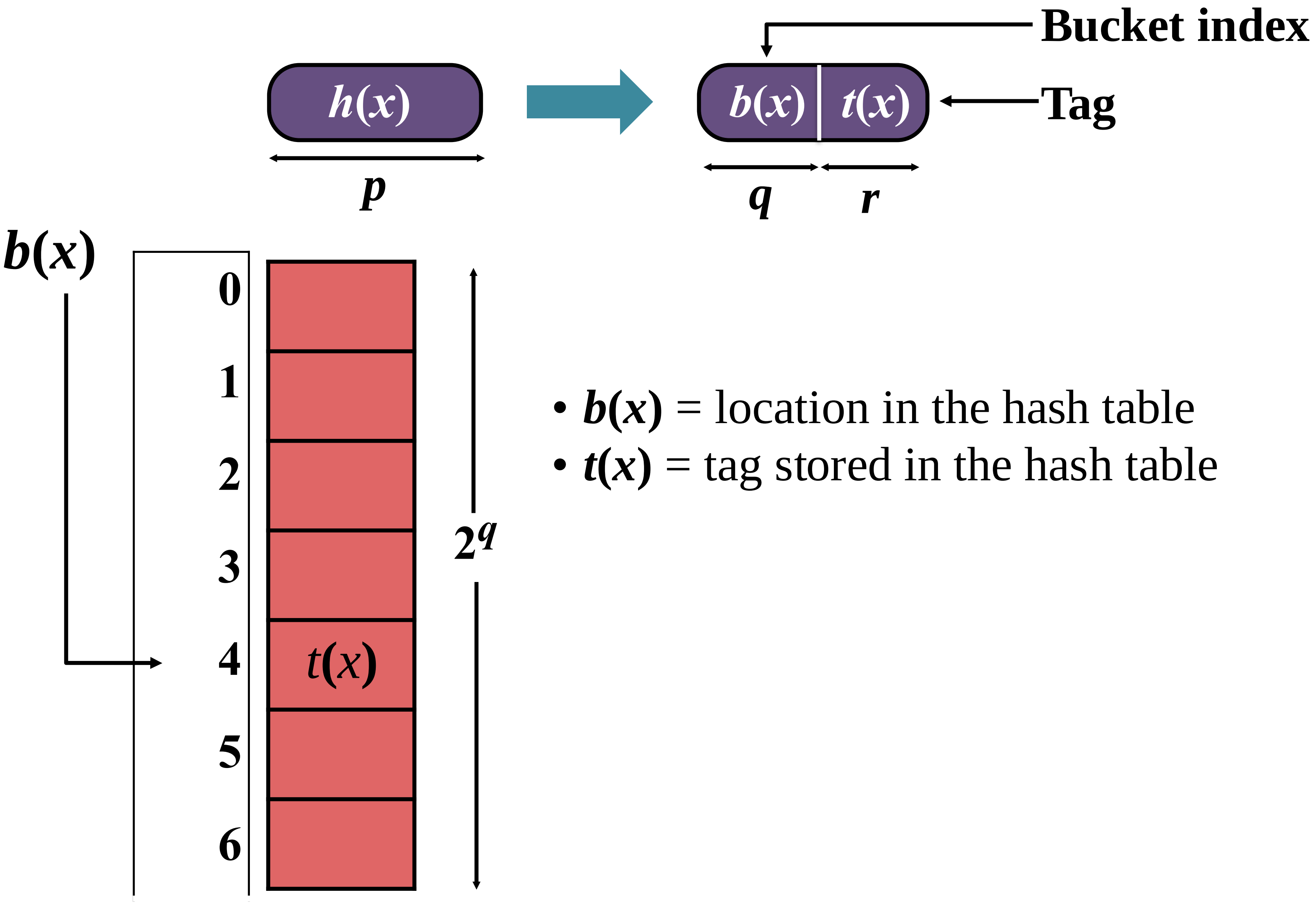
- **Store fingerprints compactly in a hash table.**
  - Take a fingerprint $h(x)$ for each element x.



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

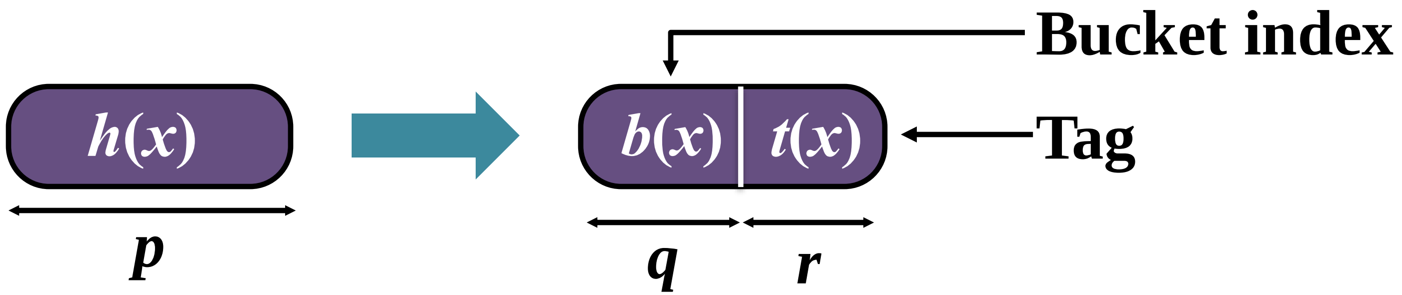
$$\Pr[\, x \text{ and } y \text{ collide}] = \frac{1}{2^p}$$

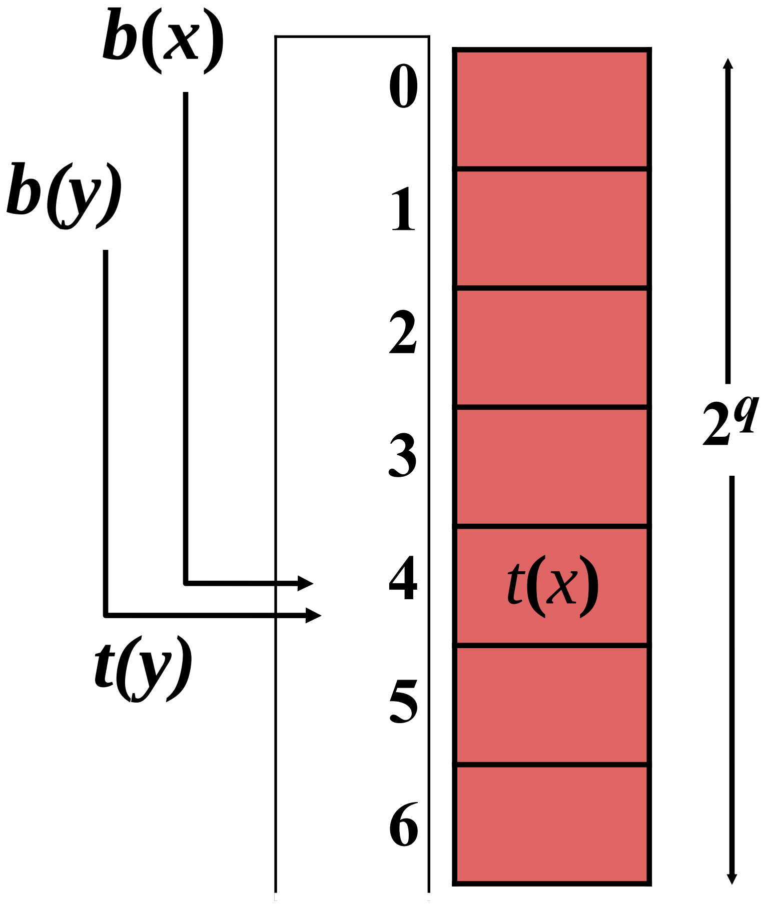
Storing fingerprints compactly



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

Storing fingerprints compactly

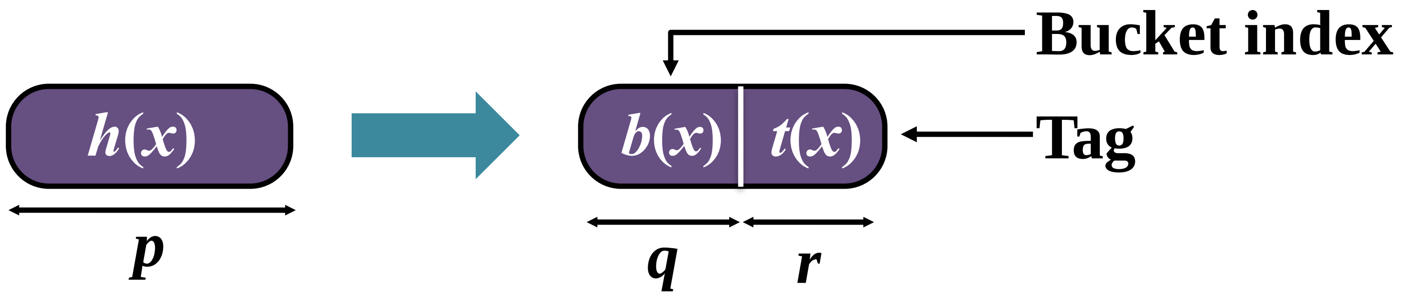


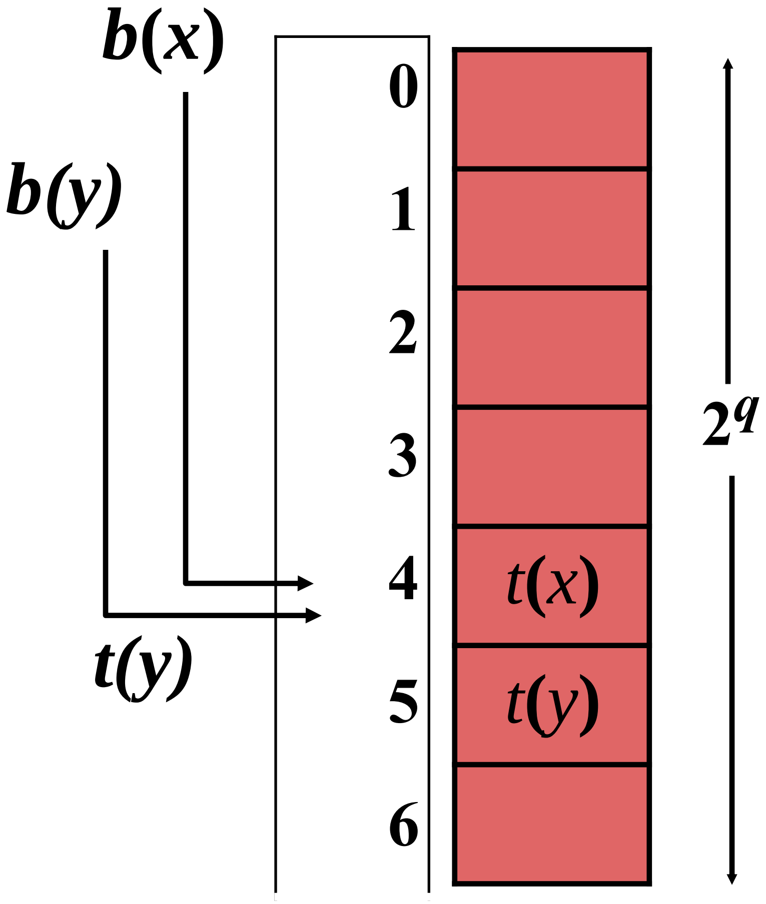


- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

Collisions in the hash table?

Storing fingerprints compactly



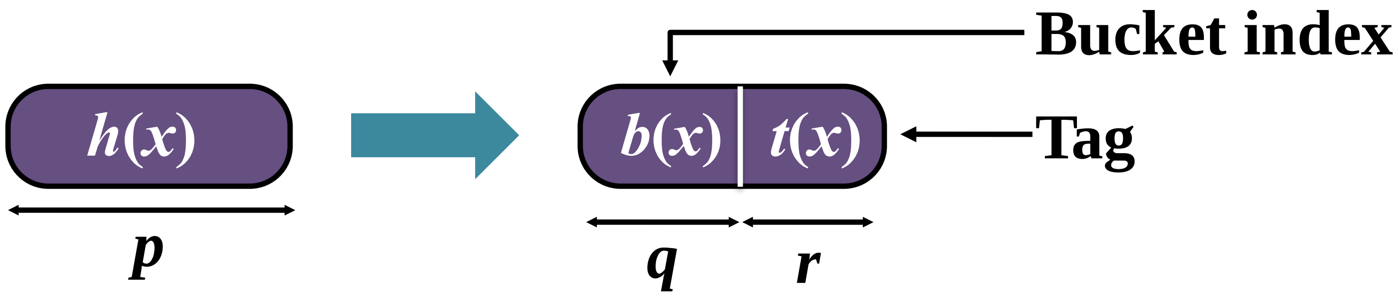


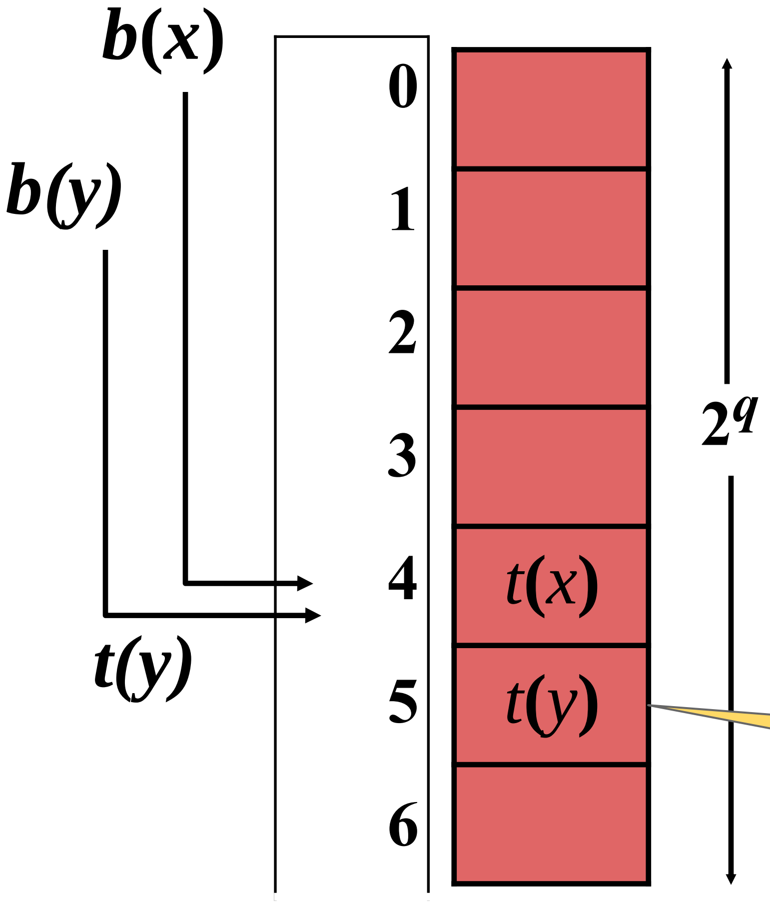
- ***b*(*x*)** = location in the hash table
- ***t*(*x*)** = tag stored in the hash table

Collisions in the hash table?

- Linear probing
- Robin Hood hashing

Storing fingerprints compactly





- ***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

$t(y)$ belongs to slots 4 or 5?

Resolving collisions in the QF

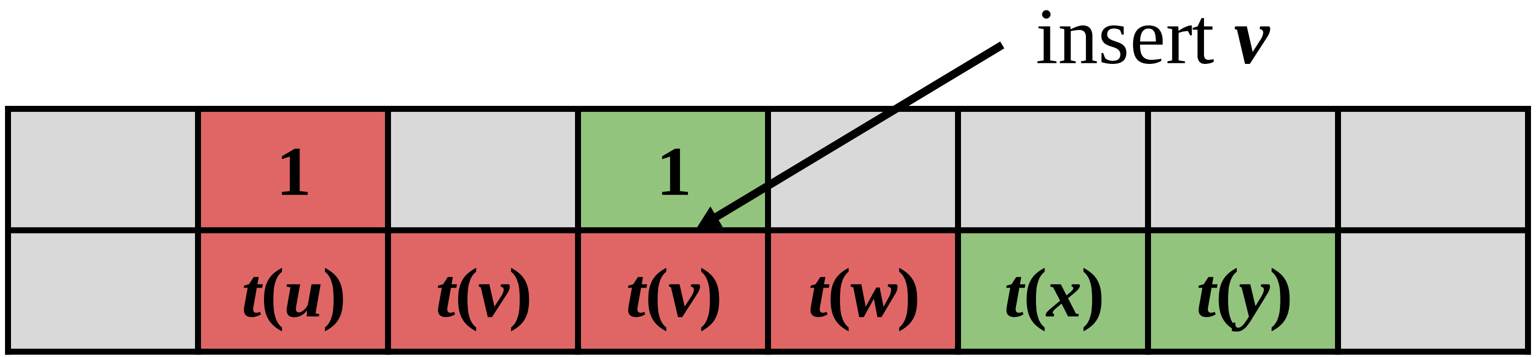
- QF uses two metadata bits to resolve collisions and identify home bucket

	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

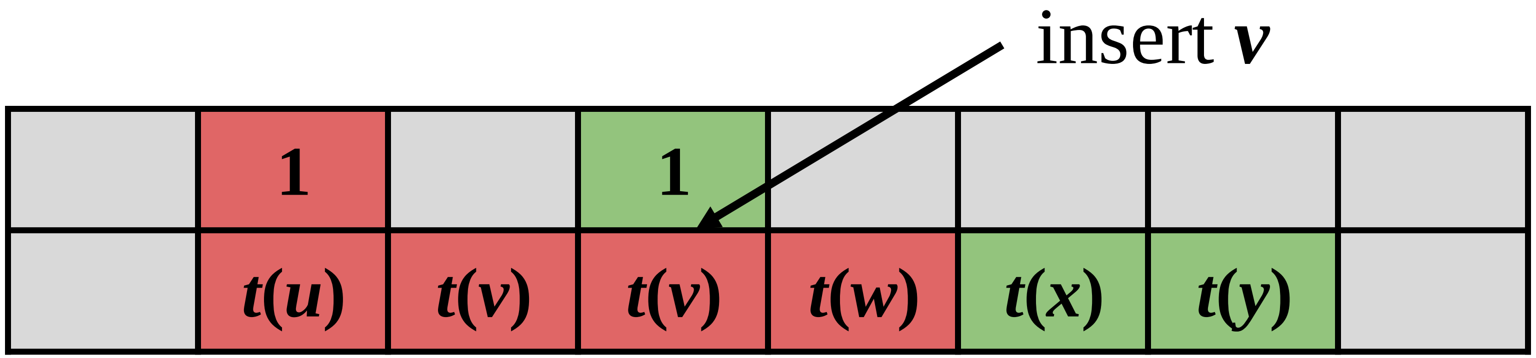
- QF uses two metadata bits to resolve collisions and identify home bucket



- The metadata bits group tags by their home bucket

Resolving collisions in the QF

- 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

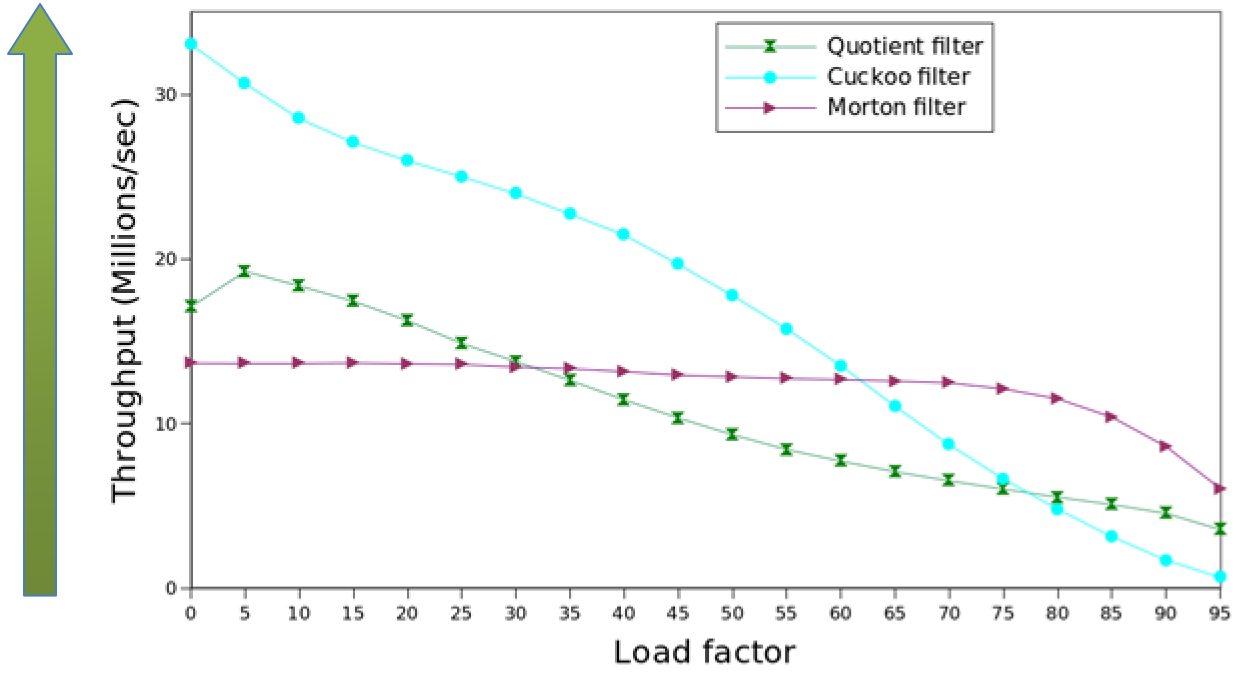
Types of filters

- 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..

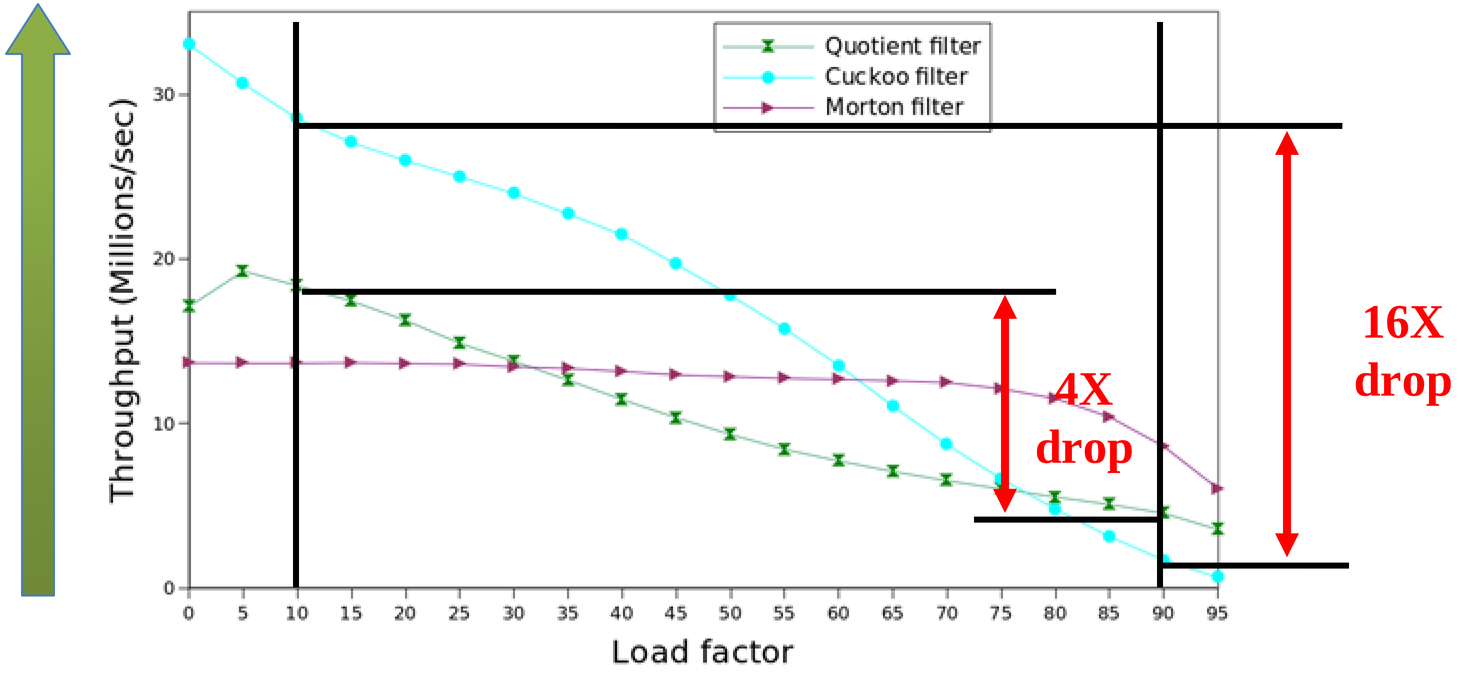
Performance suffers due to high-overhead of ***collision resolution***



Applications must choose between space and speed.

Current filters have a problem..

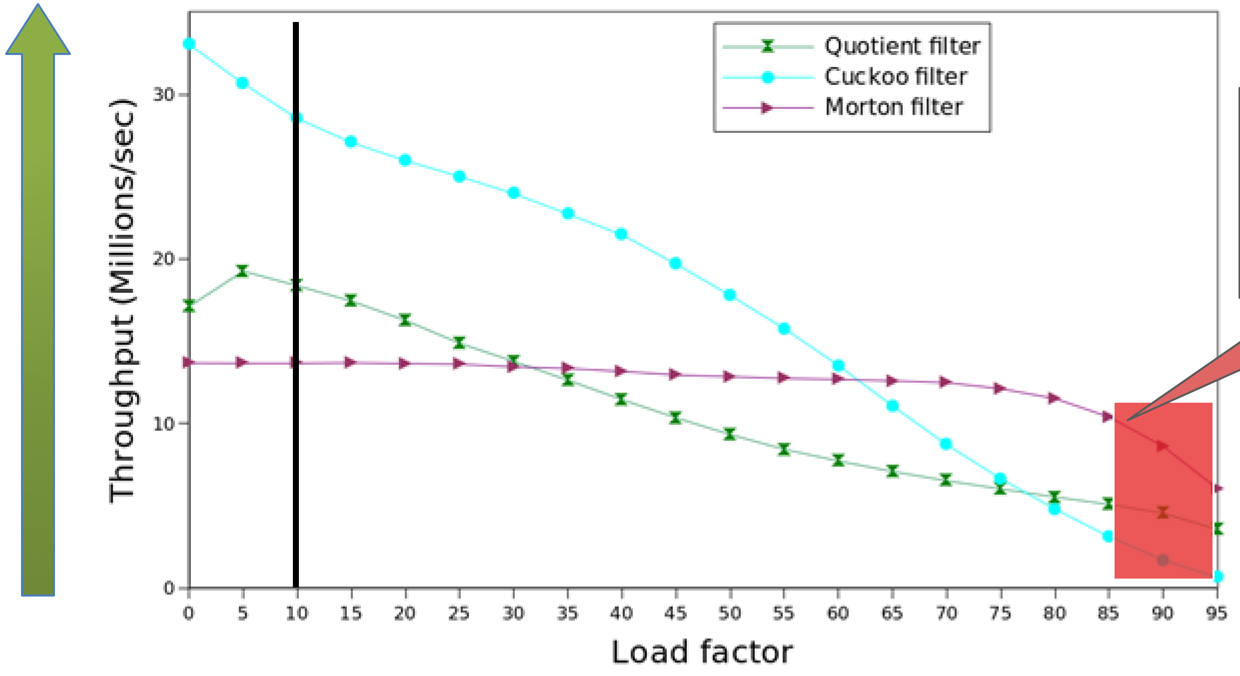
Performance suffers due to high-overhead of ***collision resolution***



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.

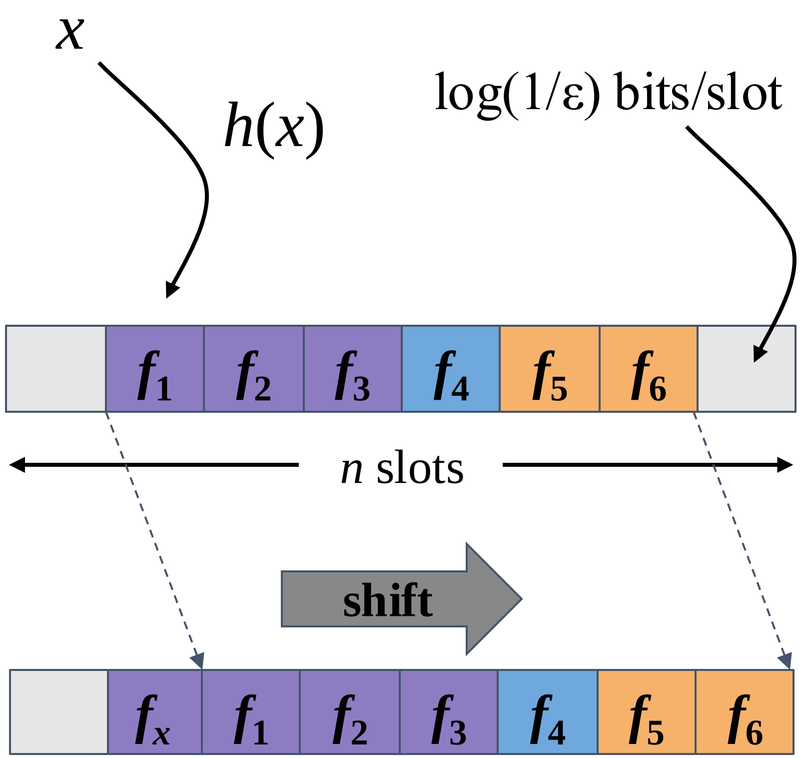
Why quotient filters slow down

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

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

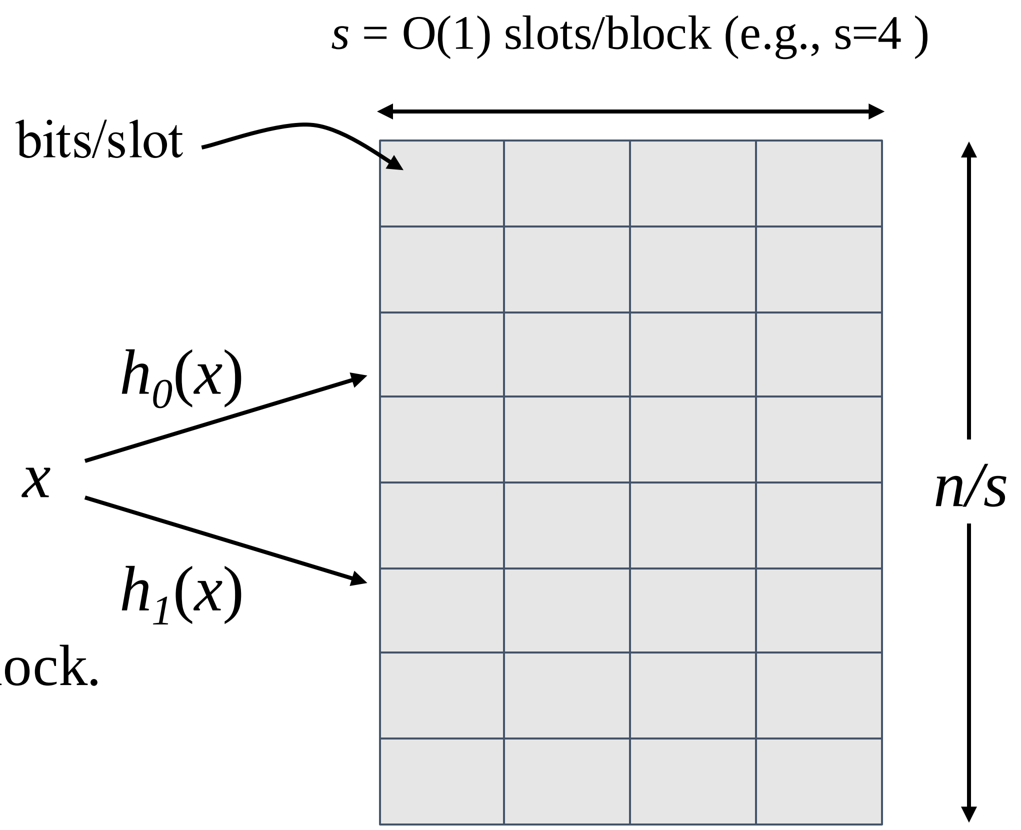
To insert item *x*:

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.

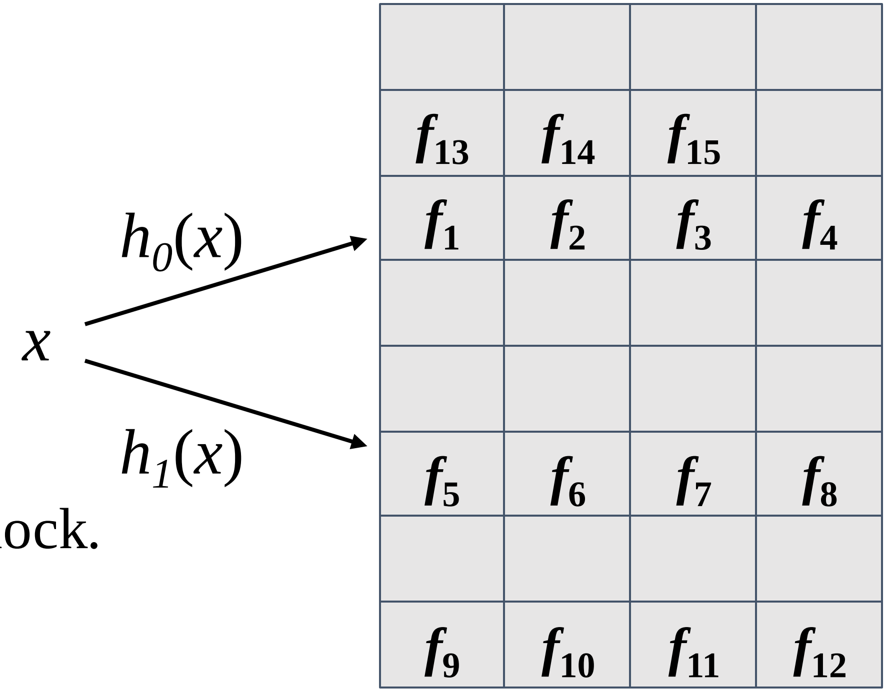
Why cuckoo filters slow down



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.

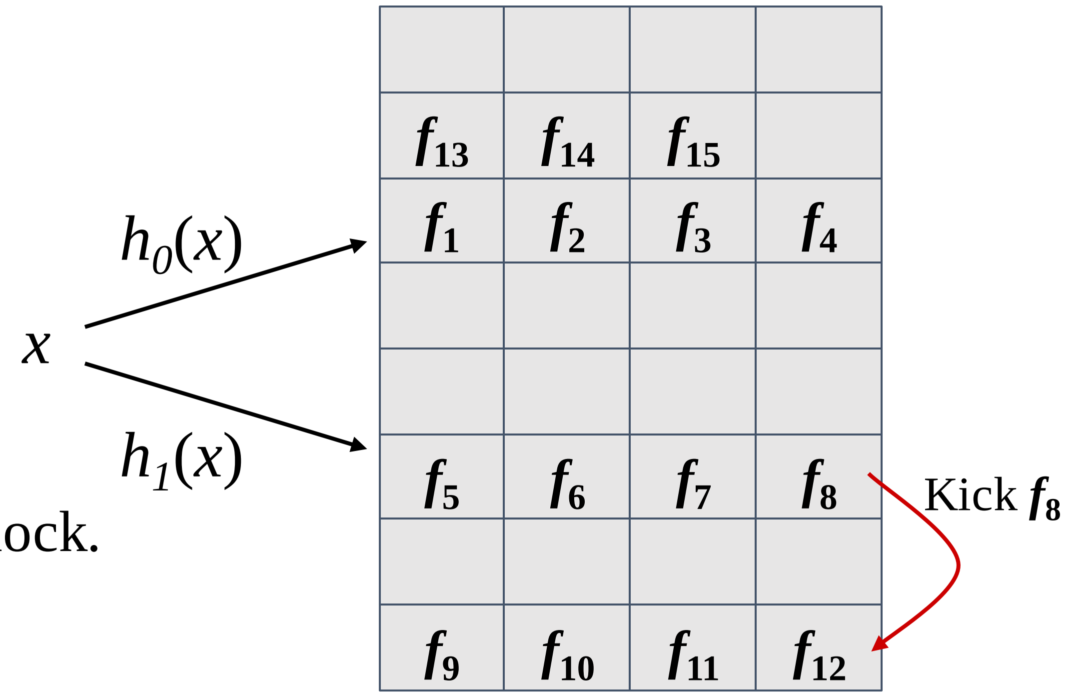
Why cuckoo filters slow down



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



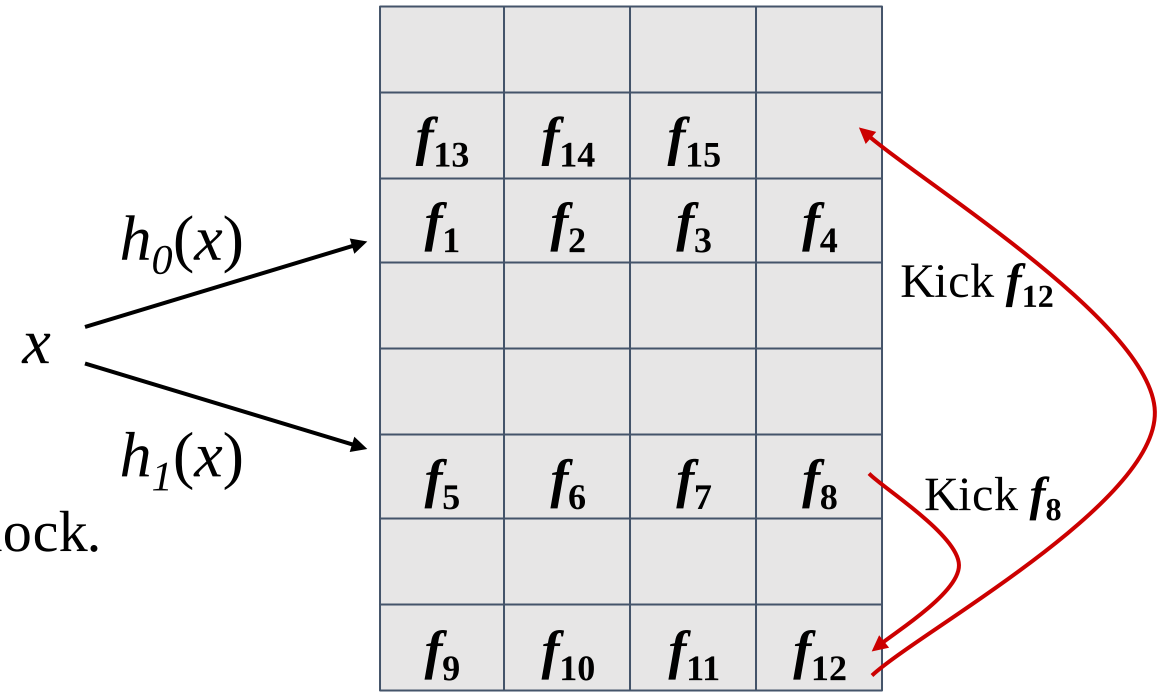
To insert item x:

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

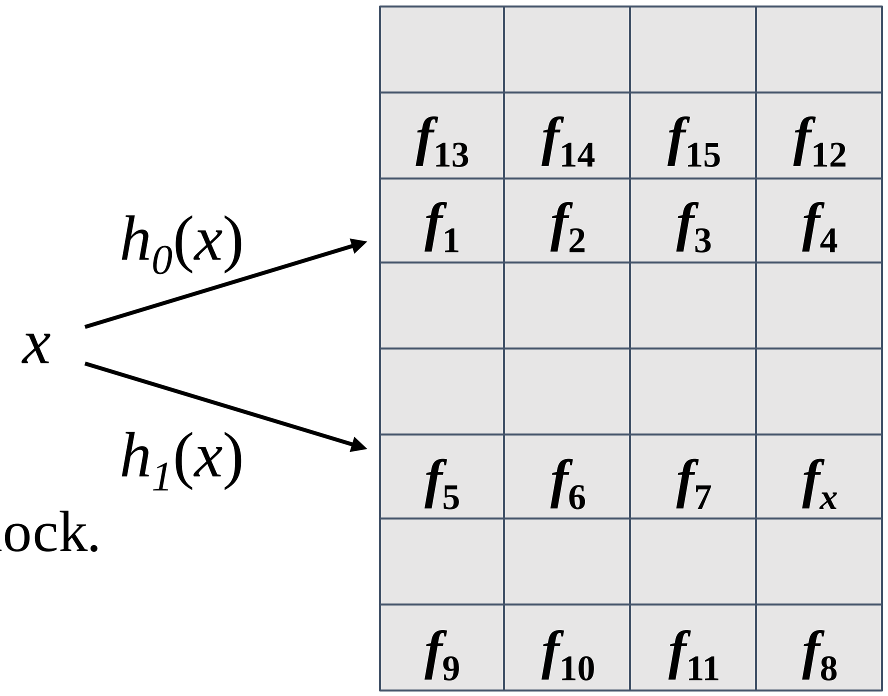
To insert item x:

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.

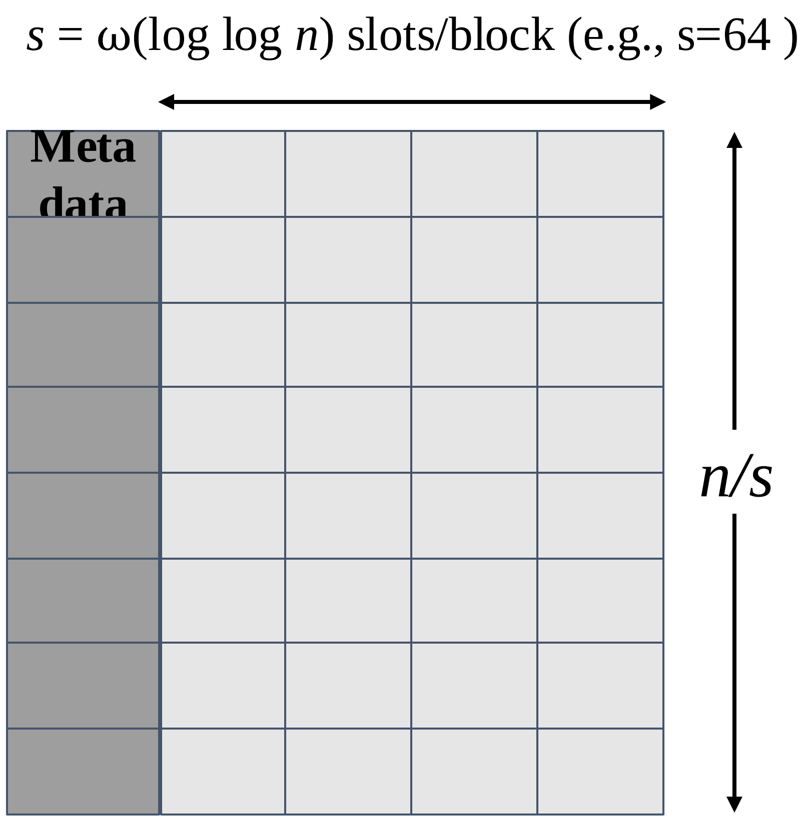
As the CF fills, inserts have to do more kicking.

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

Cuckoo filter performance

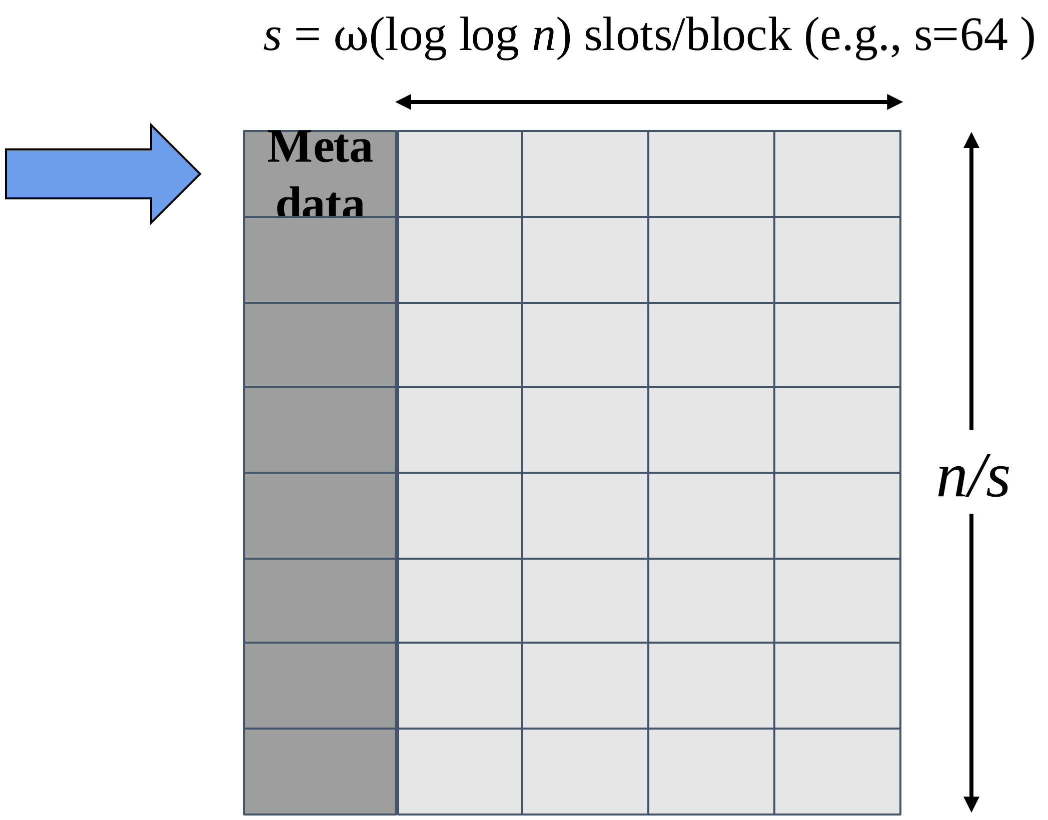
	Optimal	Cuckoo filter
Space (bits)	$\approx n \ \log(1/\epsilon) + \Omega(n)$	$\approx n \ \log(1/\epsilon) + 3n$
CPU cost	$O(1)$	up to 500
Data locality	$O(1)$ probes	random probes

Vector quotient filter design



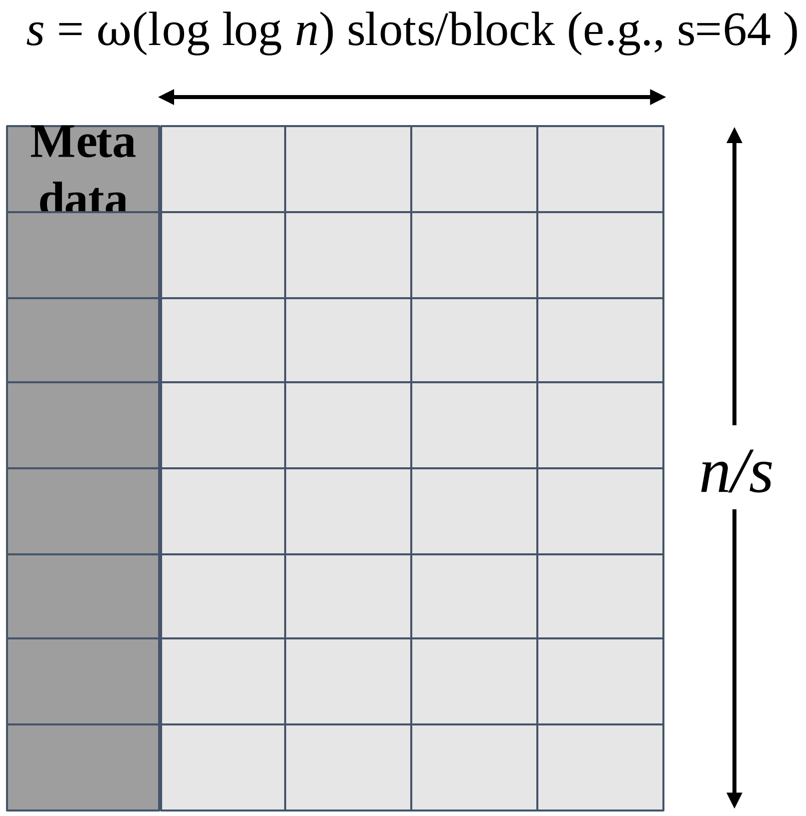
Vector quotient filter design

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*.

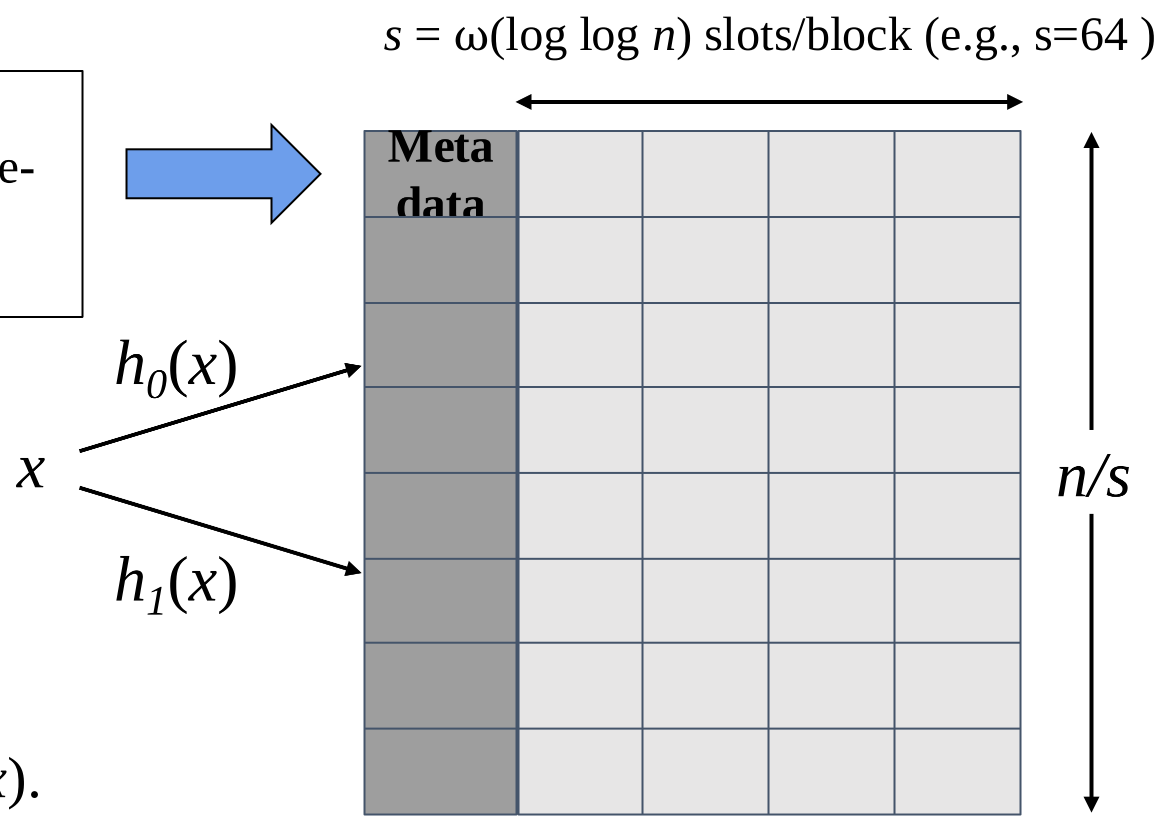


To insert item *x*:

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*.

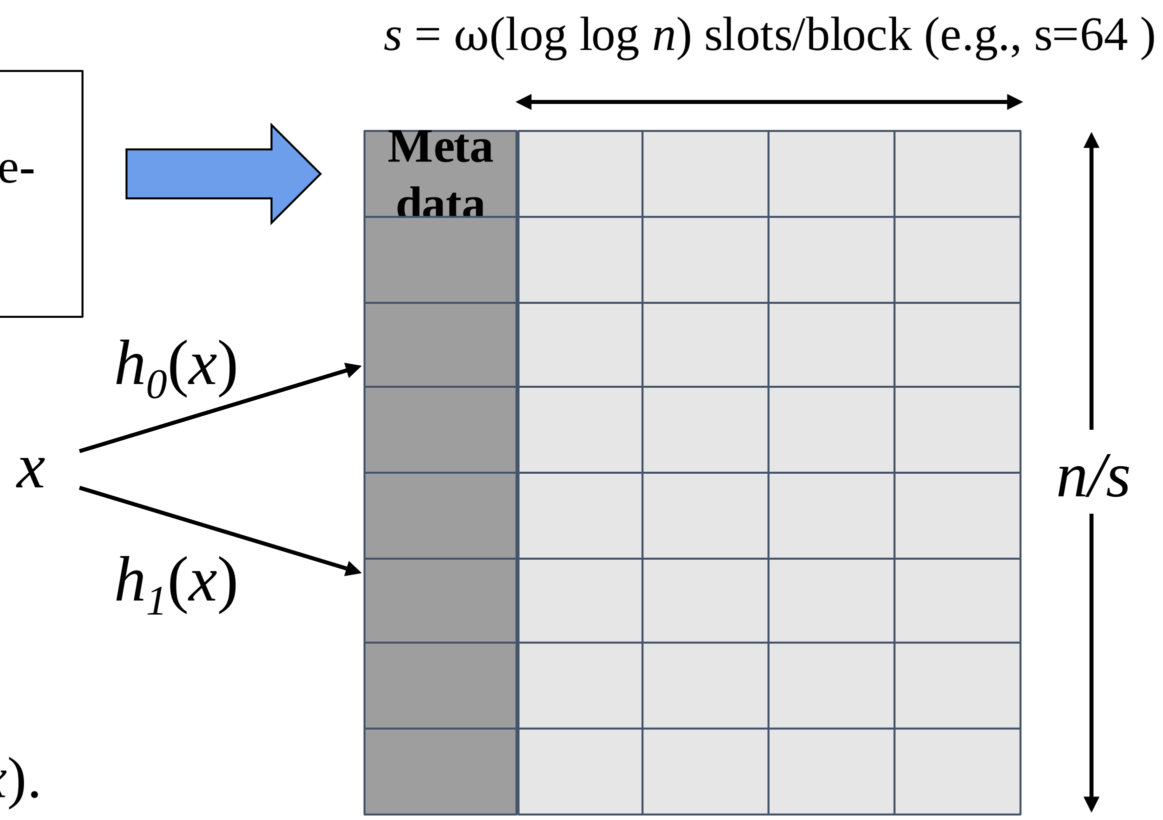


To insert item x:

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*.



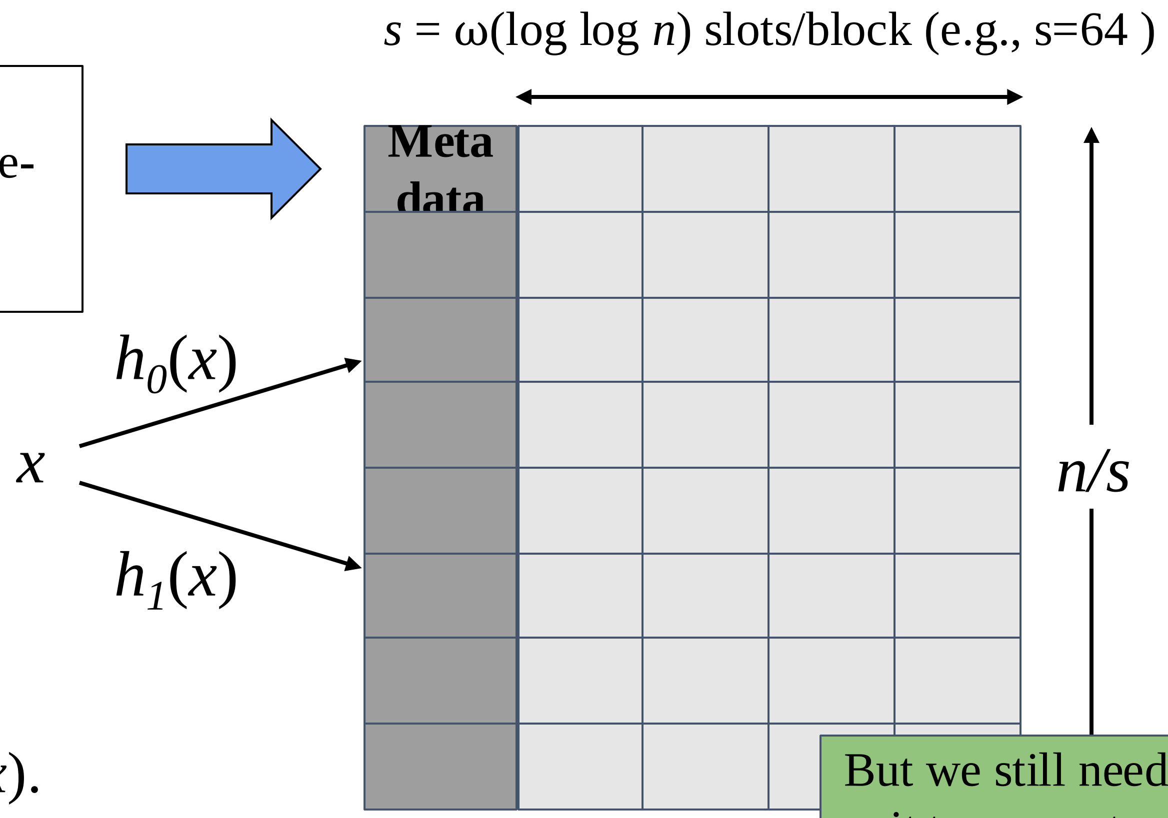
To insert item *x*:

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:

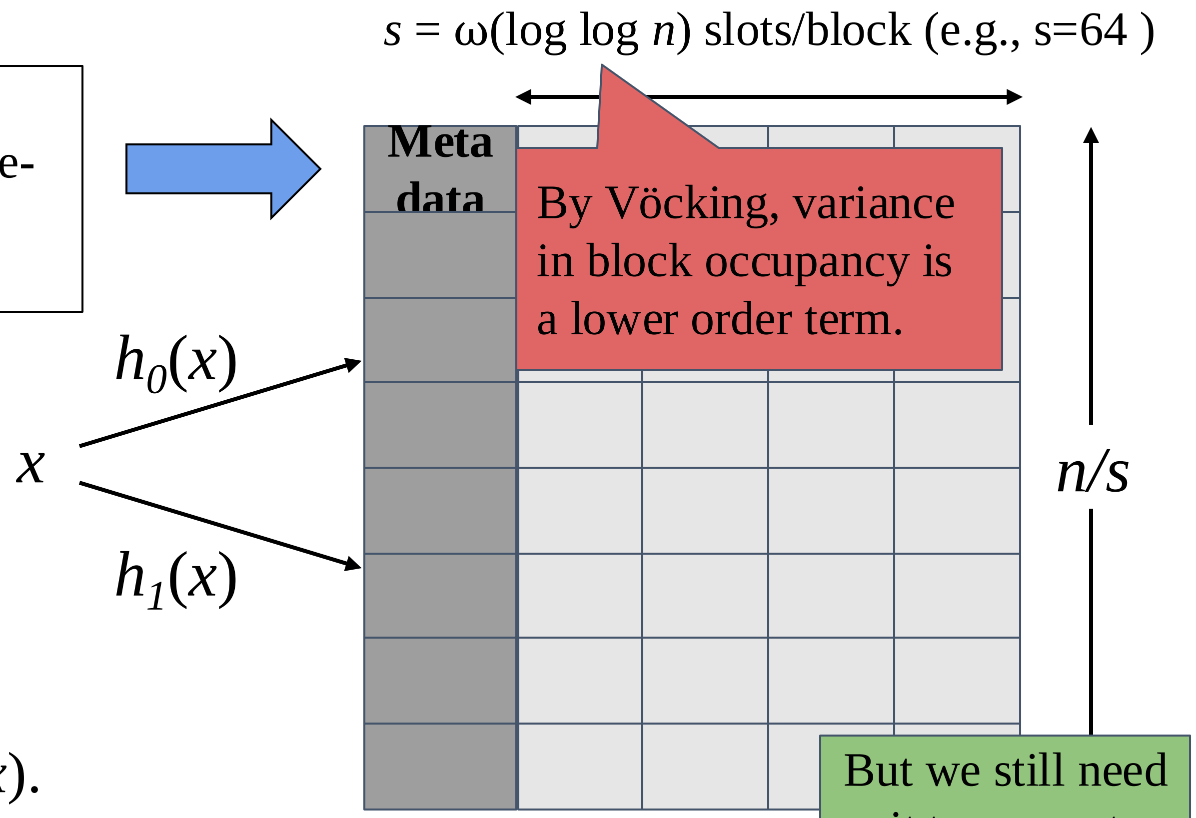
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

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.

To insert item *x*:

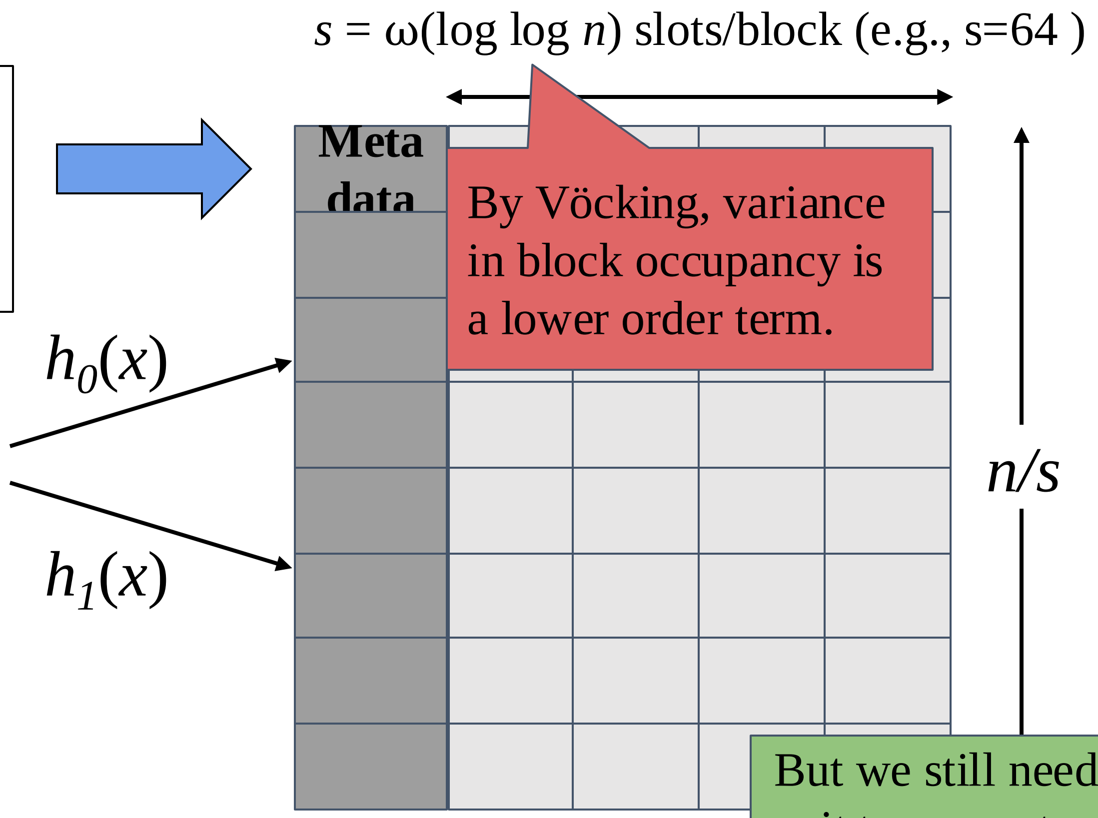
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. ~~Kick an item if needed.~~

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.

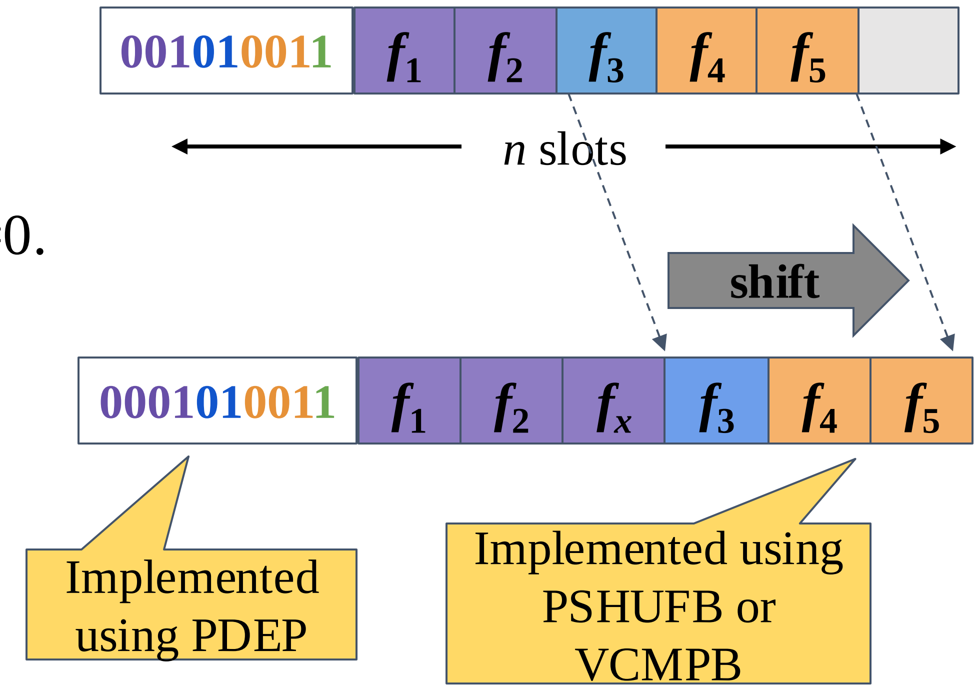
A vectorizable mini quotient filter

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 $b=s/\ln 2$.

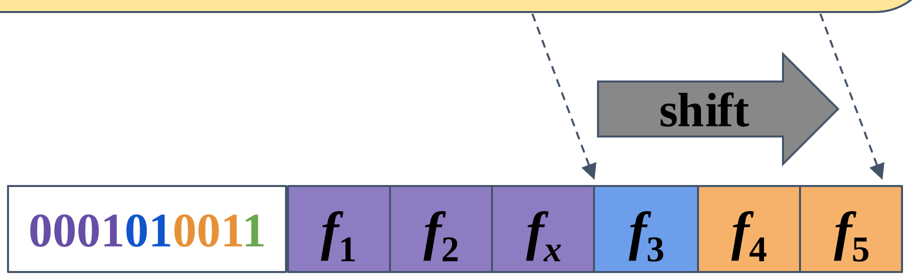
A vectorizable mini quotient filter

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 $\beta(x)$=0.



Implemented using PDEP

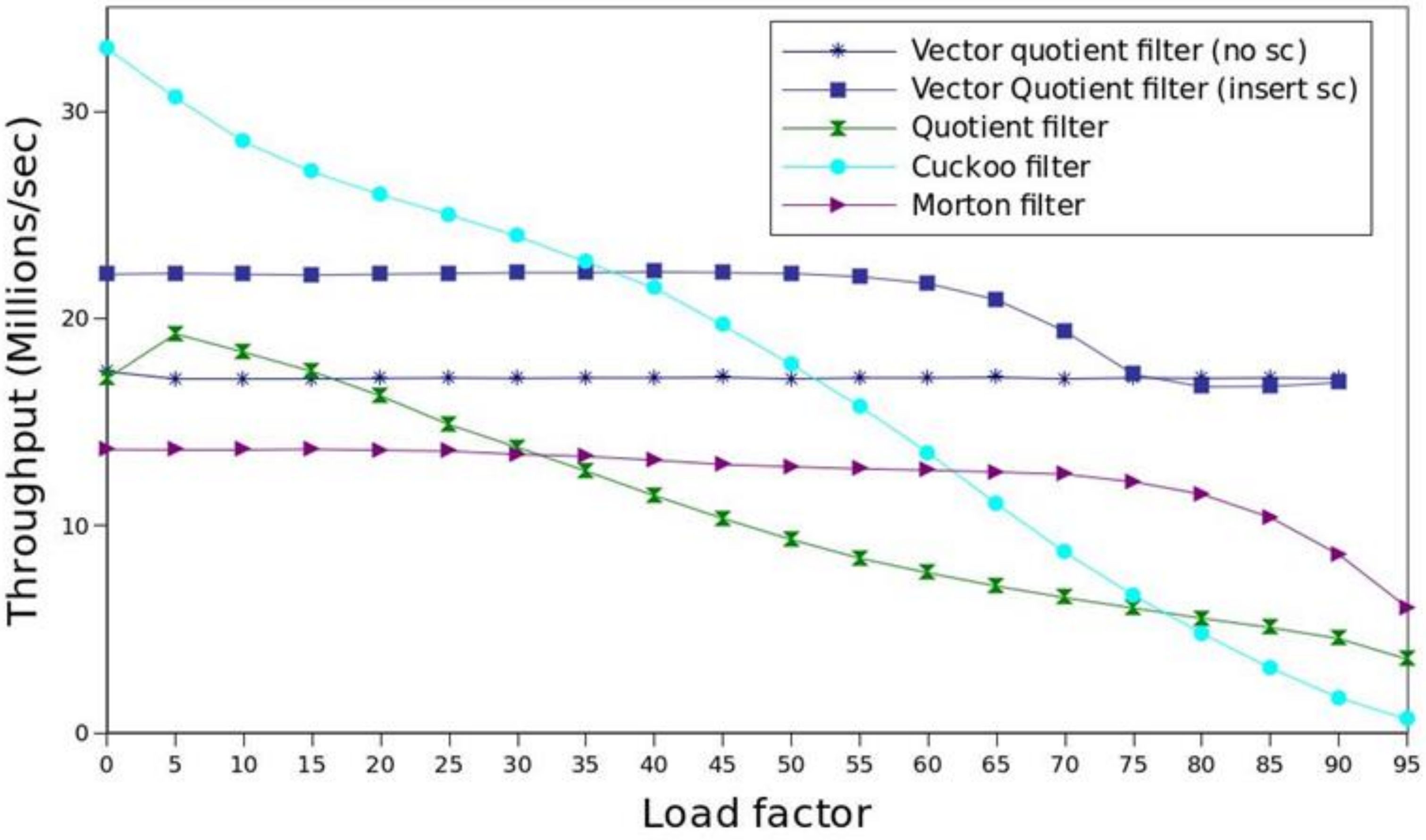
Implemented using PSHUFB or VCMPB

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

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