



+



# Lecture 24: Filters

Helen Xu

[hxu615@gatech.edu](mailto:hxu615@gatech.edu)

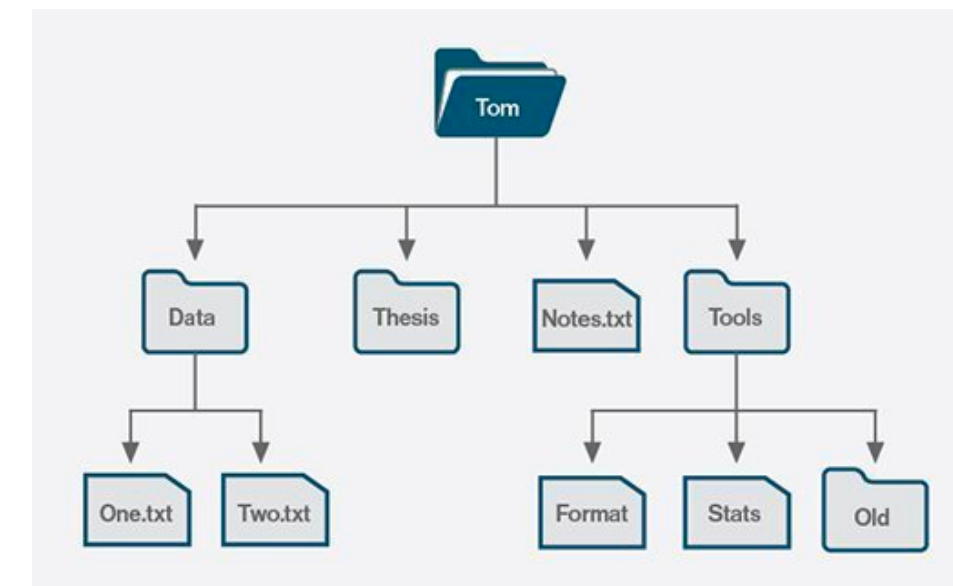


Georgia Tech College of Computing  
School of Computational  
Science and Engineering

# Motivation: Filters are ubiquitous



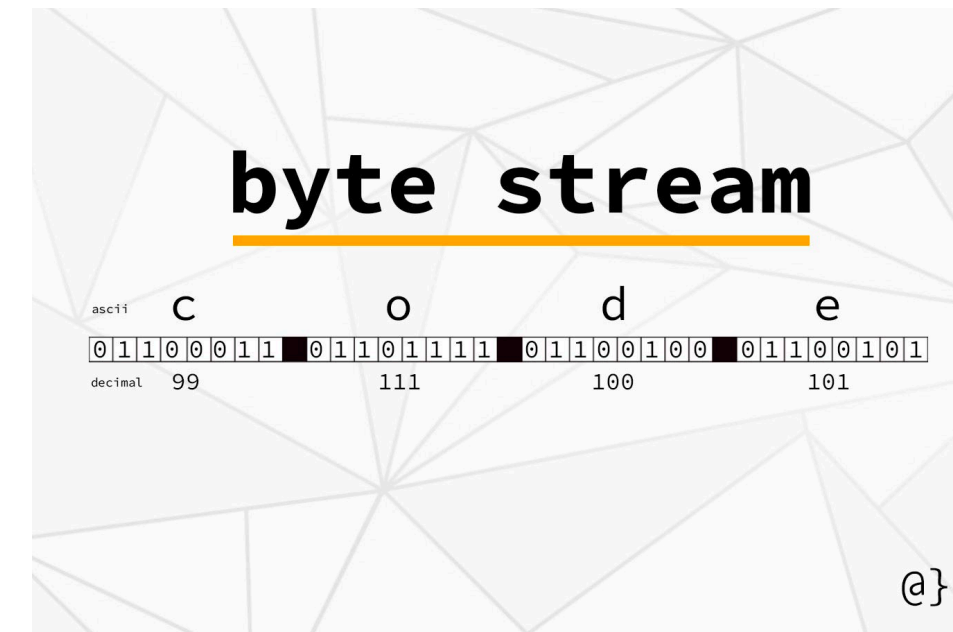
Computational biology



Storage systems



Databases



Streaming applications

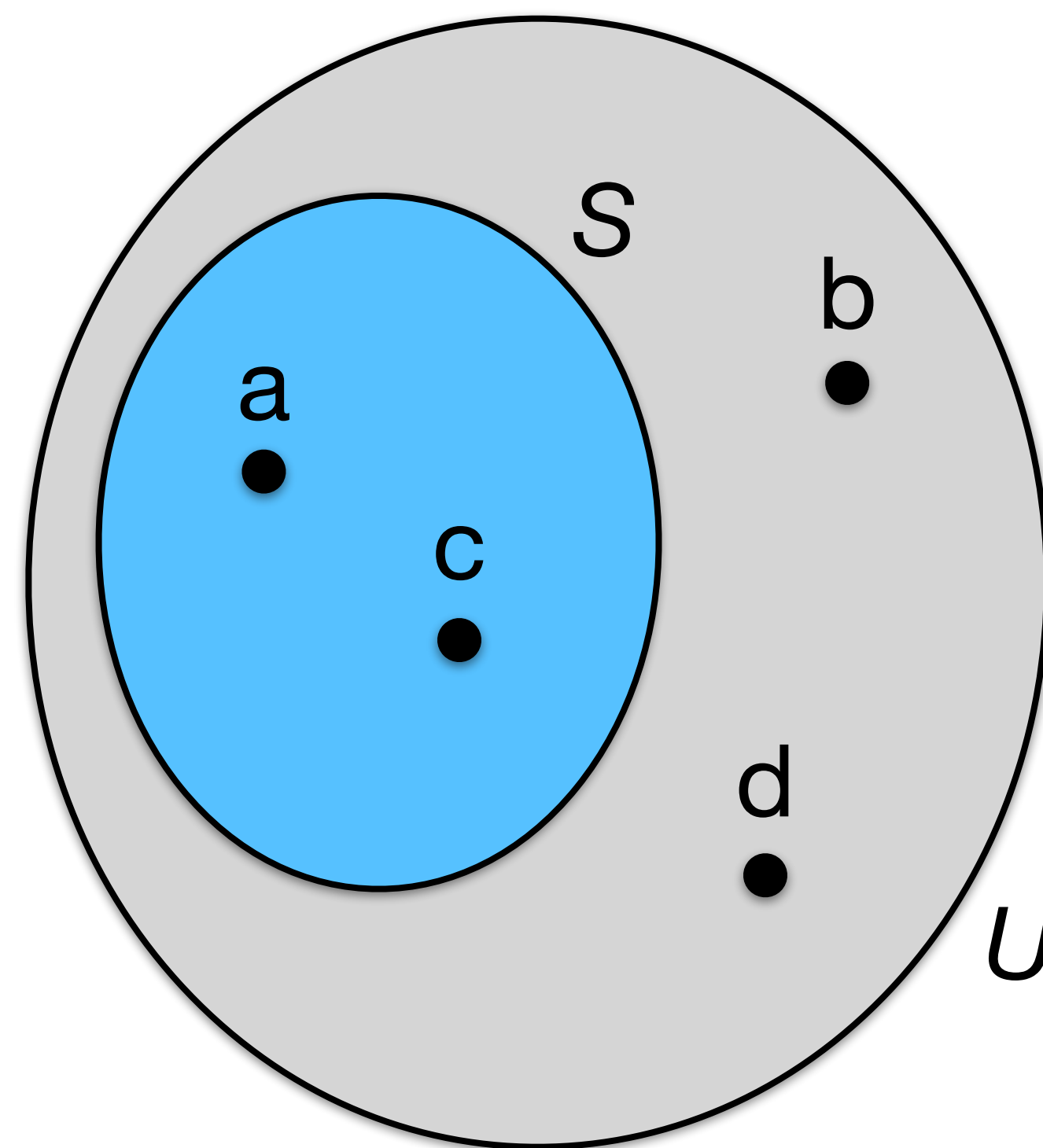



Networking


e.g., Bloom filter


# Recap: Filter Data Structure


A filter is an **approximate** dictionary.



member(a): **yes** 

member(b): **no** 

member(c): **yes** 

member(d): **yes** 

False positive

A filter supports **approximate membership queries** on  $S$ .

# Recap: A Filter Guarantees a False-Positive Rate $\epsilon$

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

If  $q \notin S$ , return **no** with probability  $> 1 - \epsilon$  **true negative**  
**yes** with probability  $\leq \epsilon$  **false positive**

One-sided error (no false negatives)



# Recap: False-positive rate enables filters to be compact

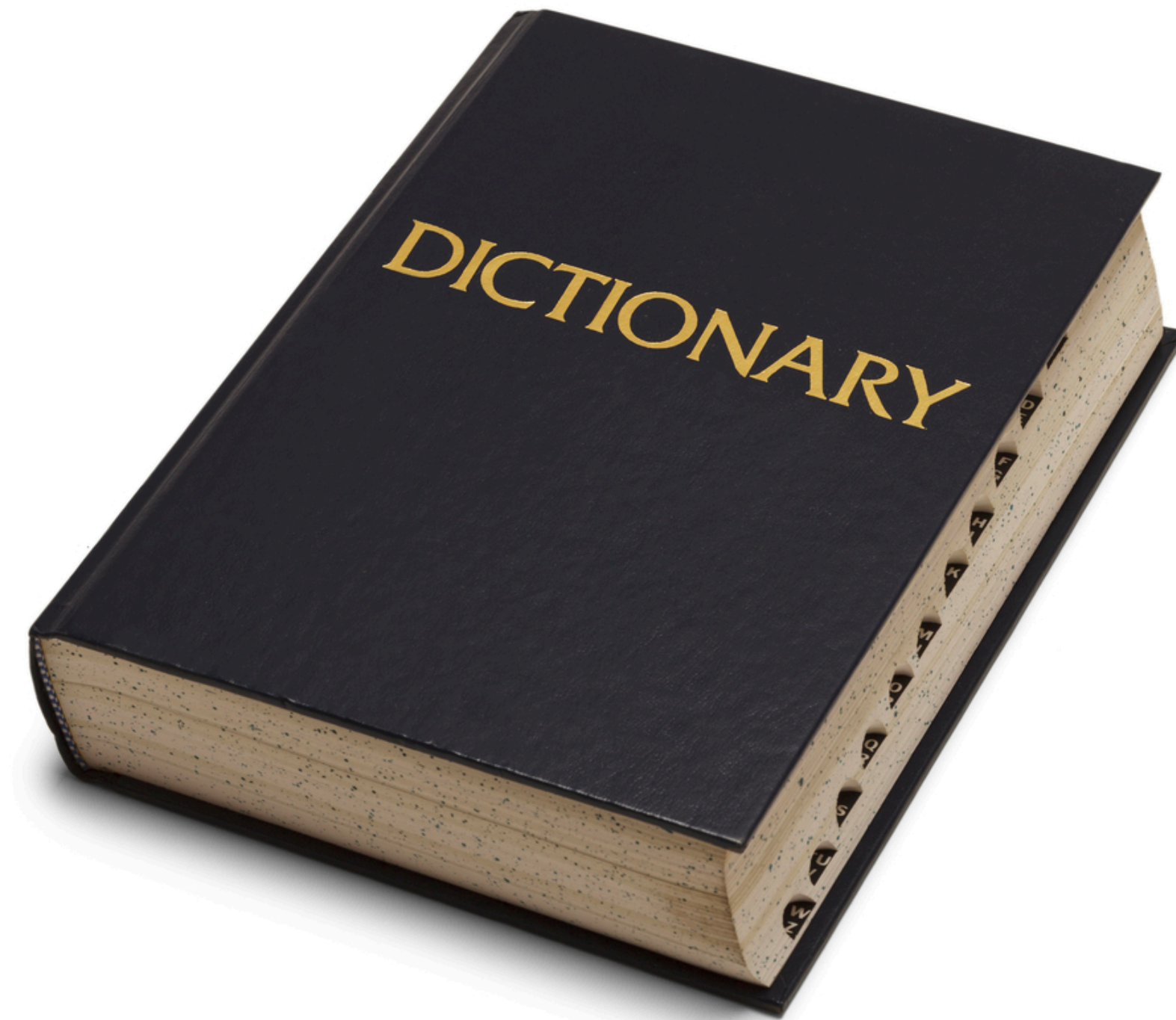
space of filter  $\geq n \log(1/\epsilon)$

small



space of dictionary  $= \Omega(n \log |U|)$

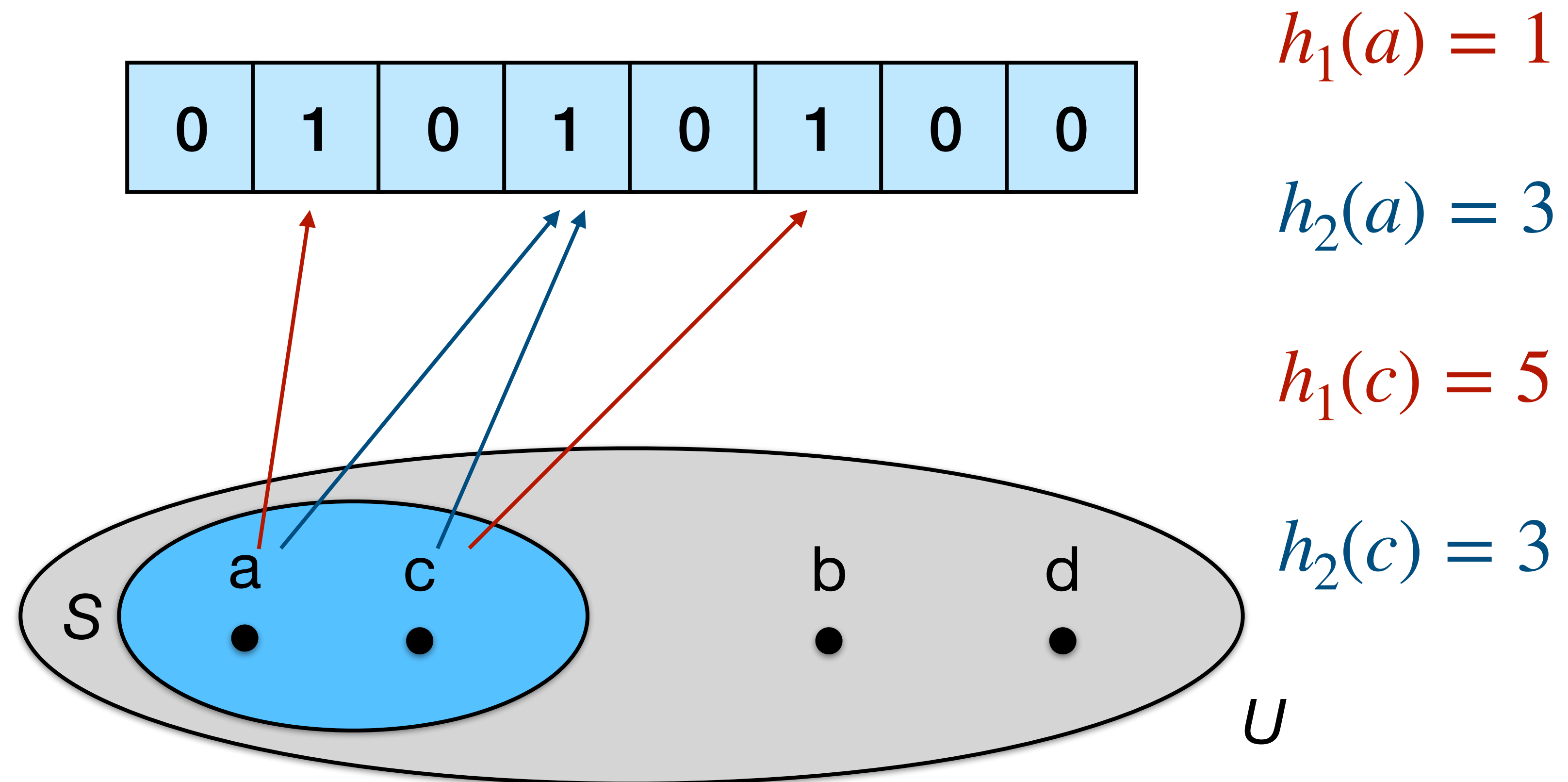
large



For most practical purposes:  $\epsilon = 2\%$ , so a filter requires  $\sim 8$  bits per item

# Recap: The Bloom Filter [Bloom '70]

Bloom filter: a bit array +  $k$  hash functions ( $k=2$  in this example)



# 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, 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.



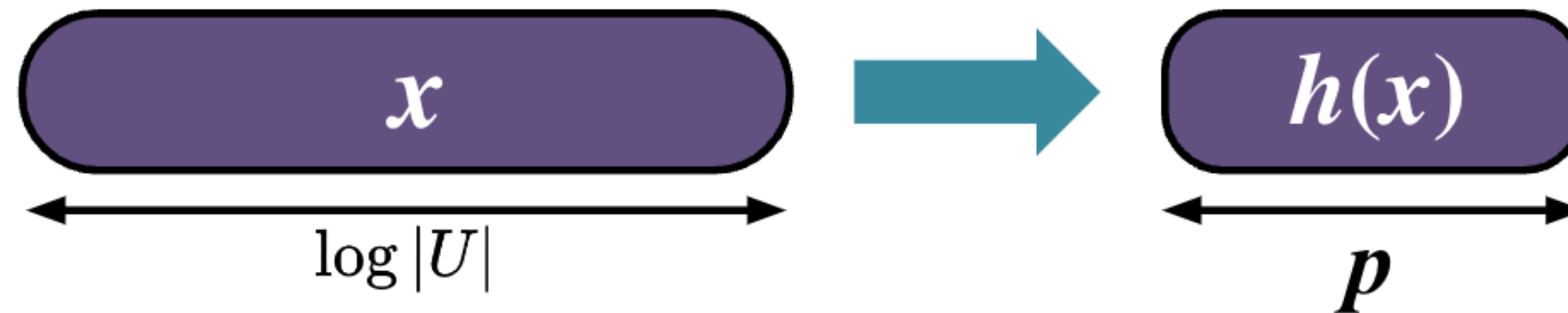
# Quotient Filters

# Quotienting: 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  $x$ .

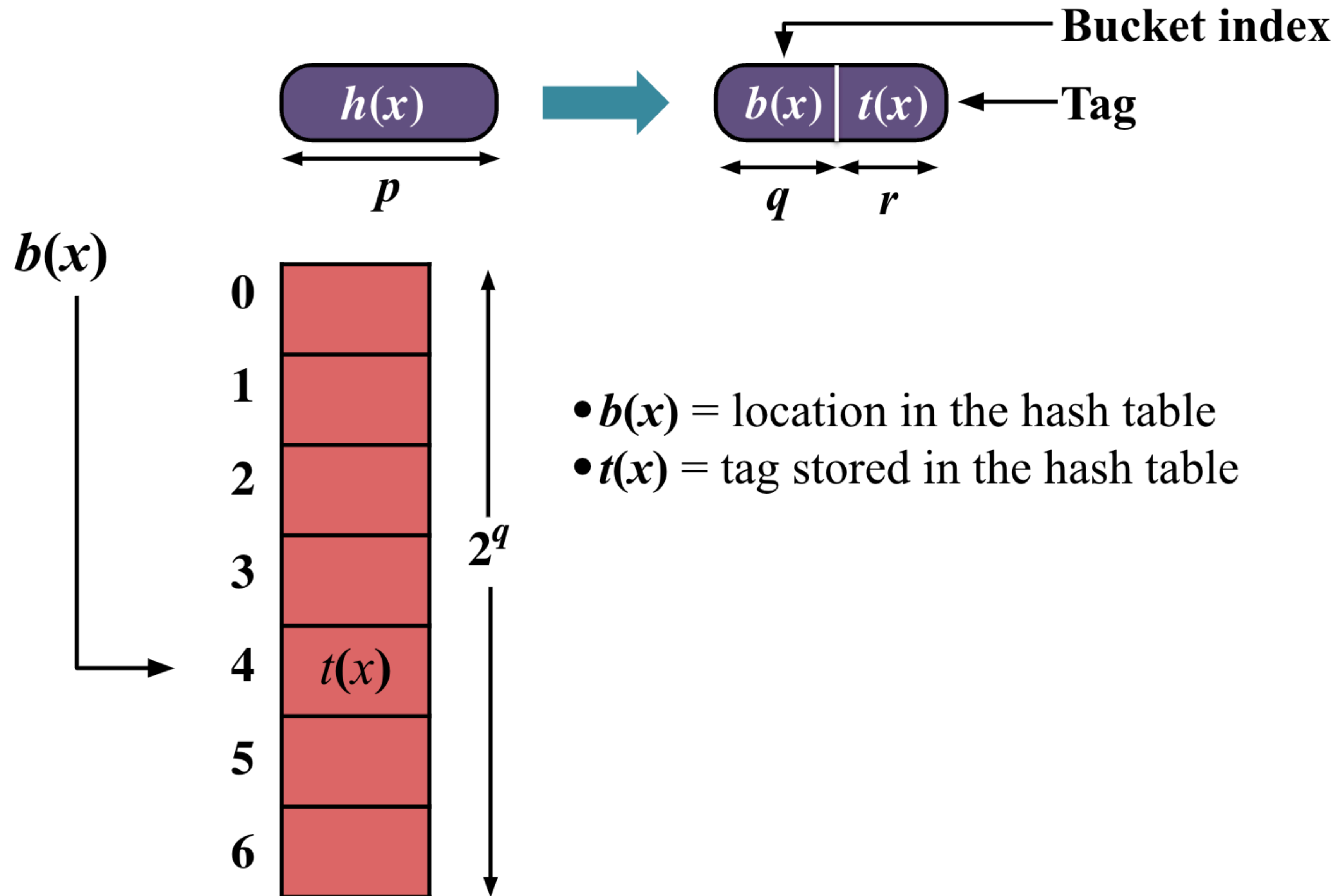


Only source of false positives:

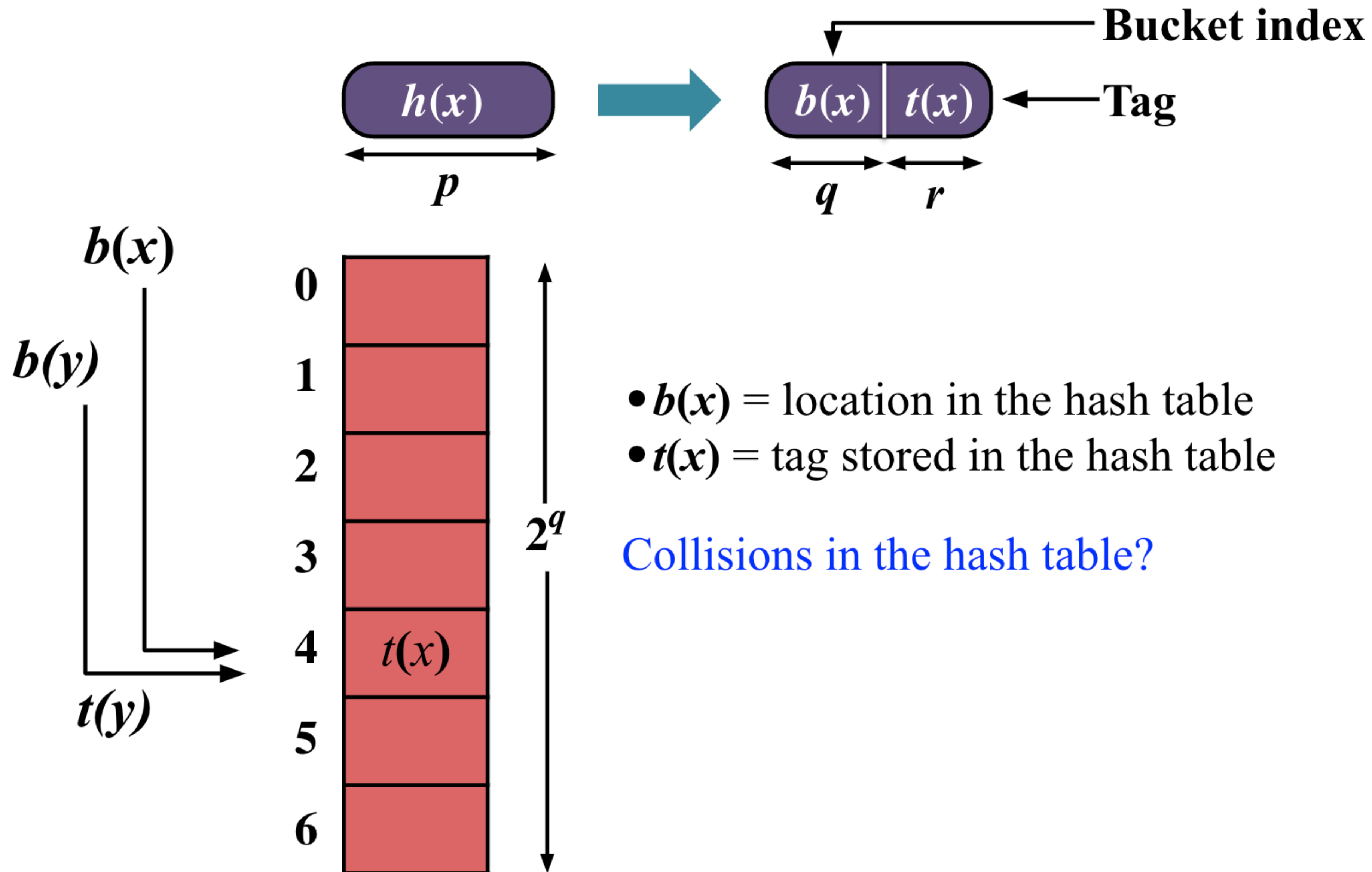
- Two distinct elements  $x$  and  $y$ , where  $h(x) = h(y)$
- If  $x$  is stored and  $y$  isn't,  $\text{query}(y)$  gives a false positive

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

# Storing fingerprints compactly

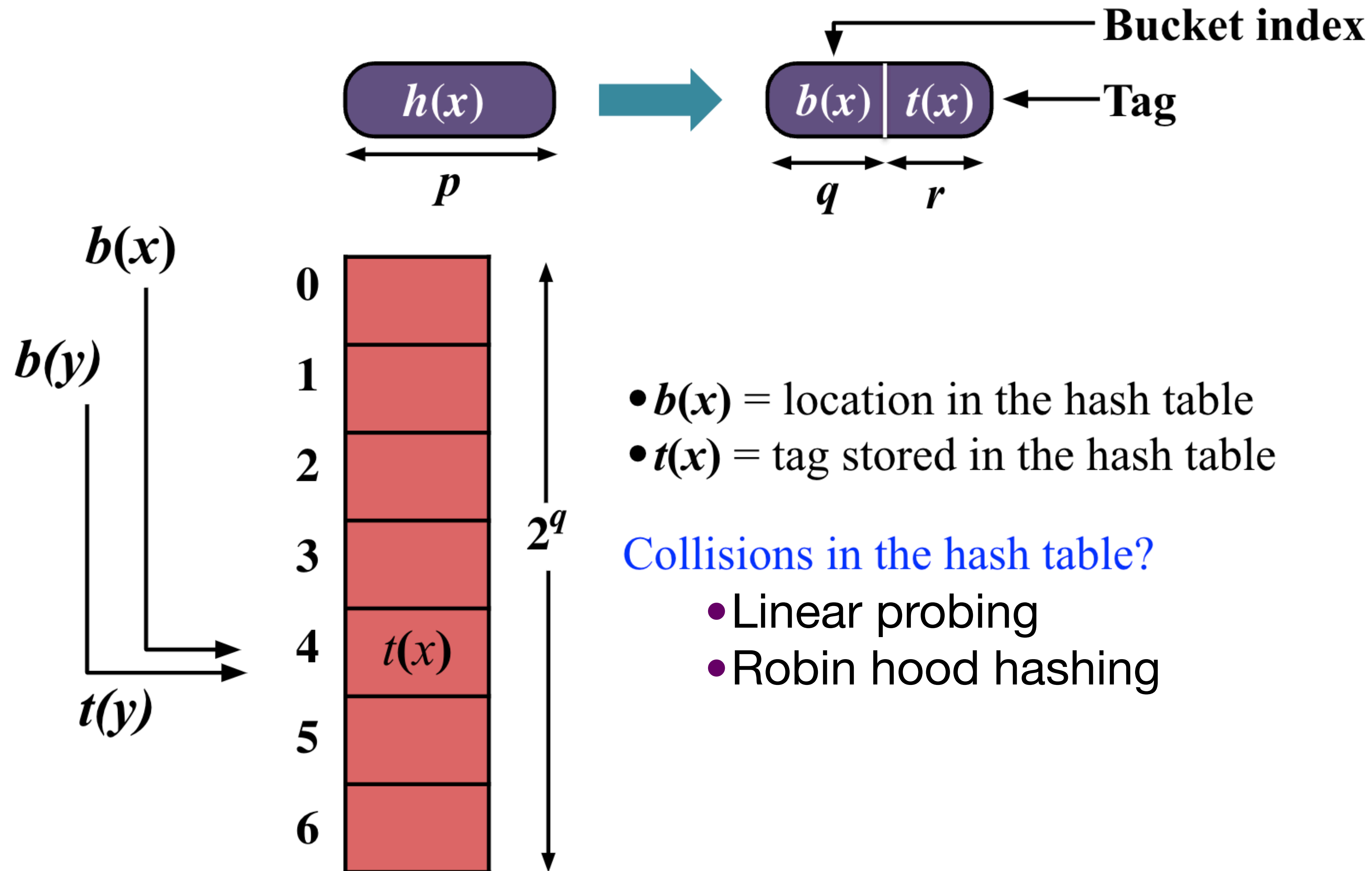


# Storing fingerprints compactly

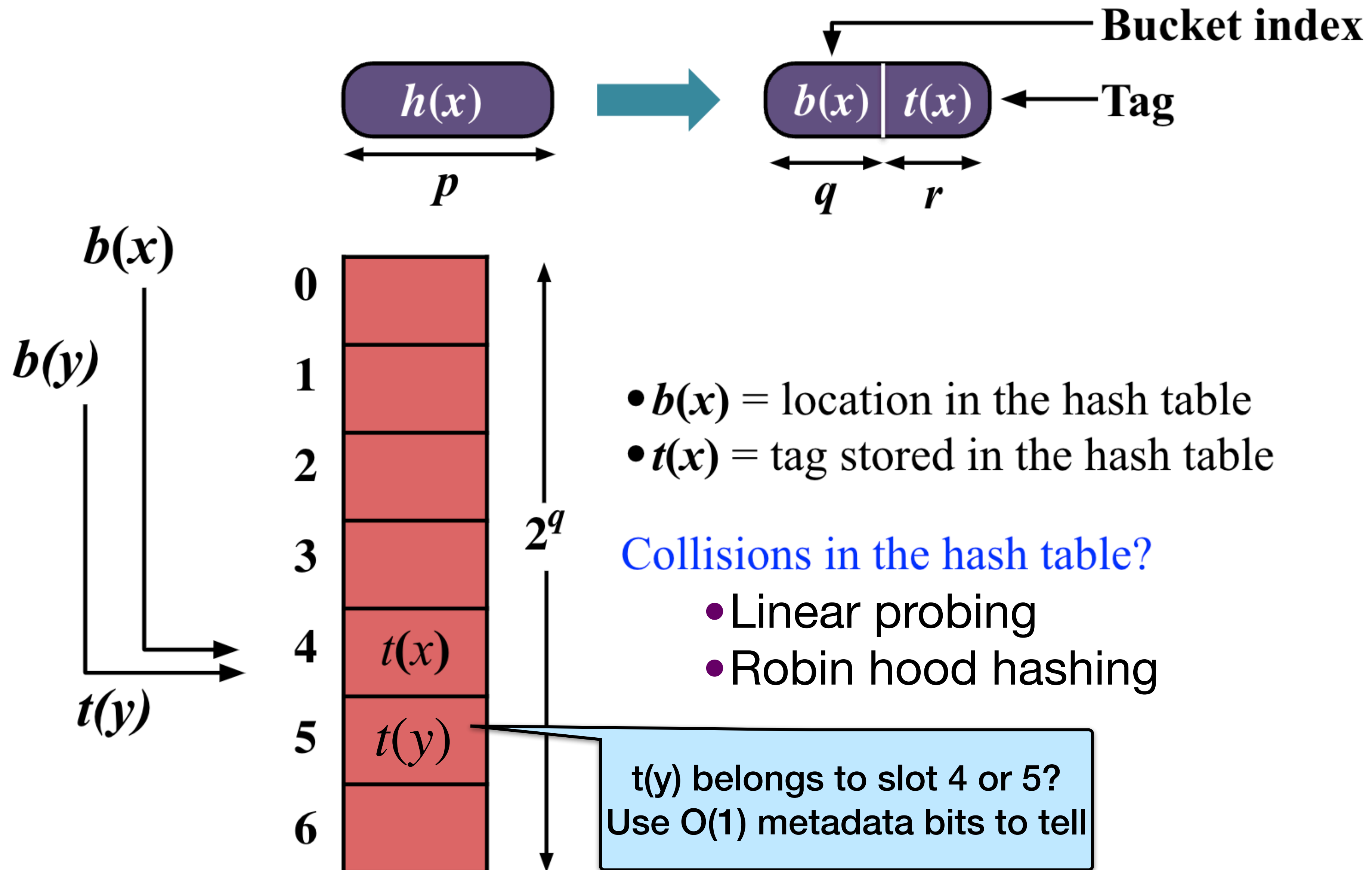




# Storing fingerprints compactly



# Storing fingerprints compactly

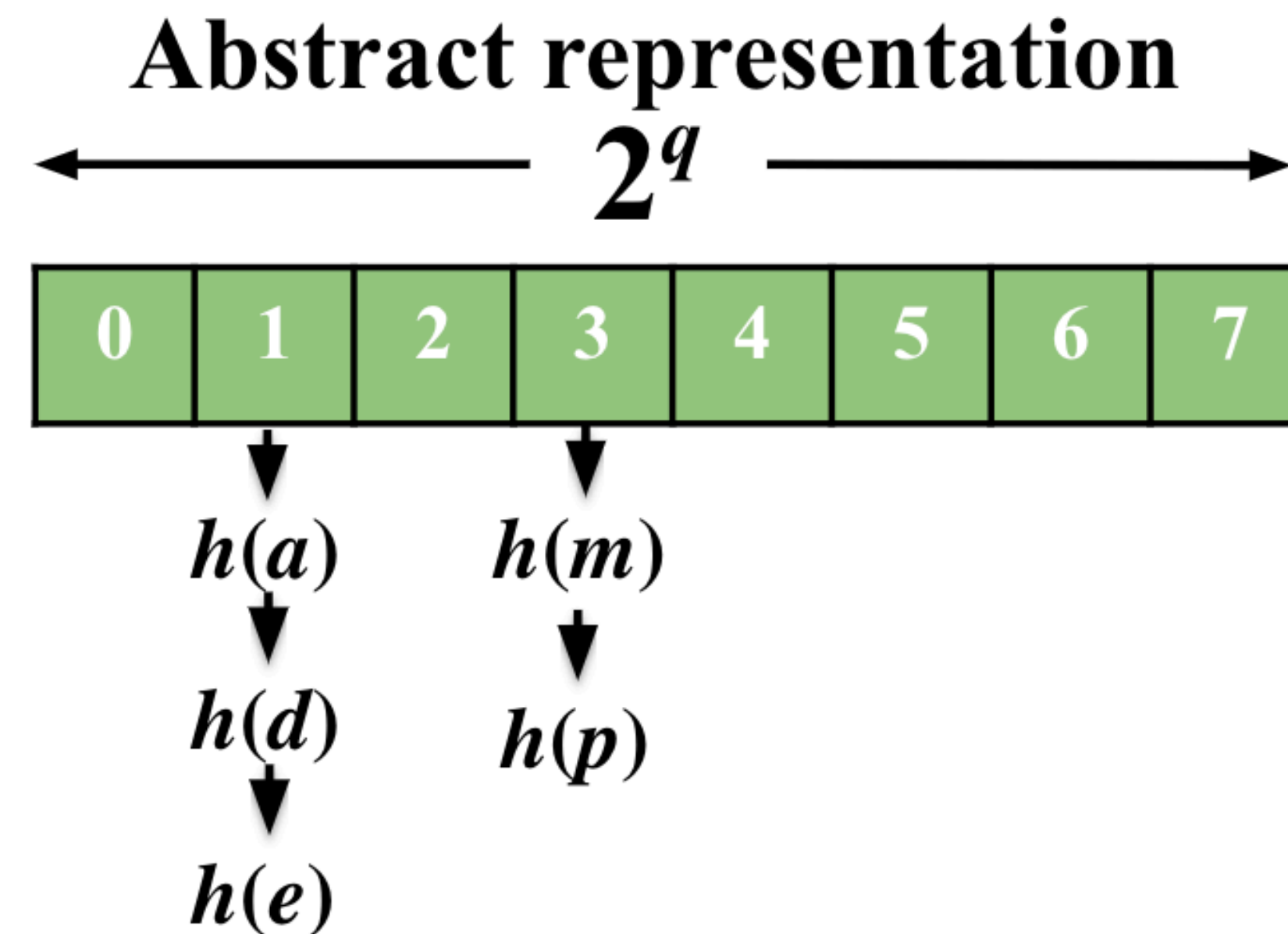


# Resolving collisions in the QF

**Implementation:**

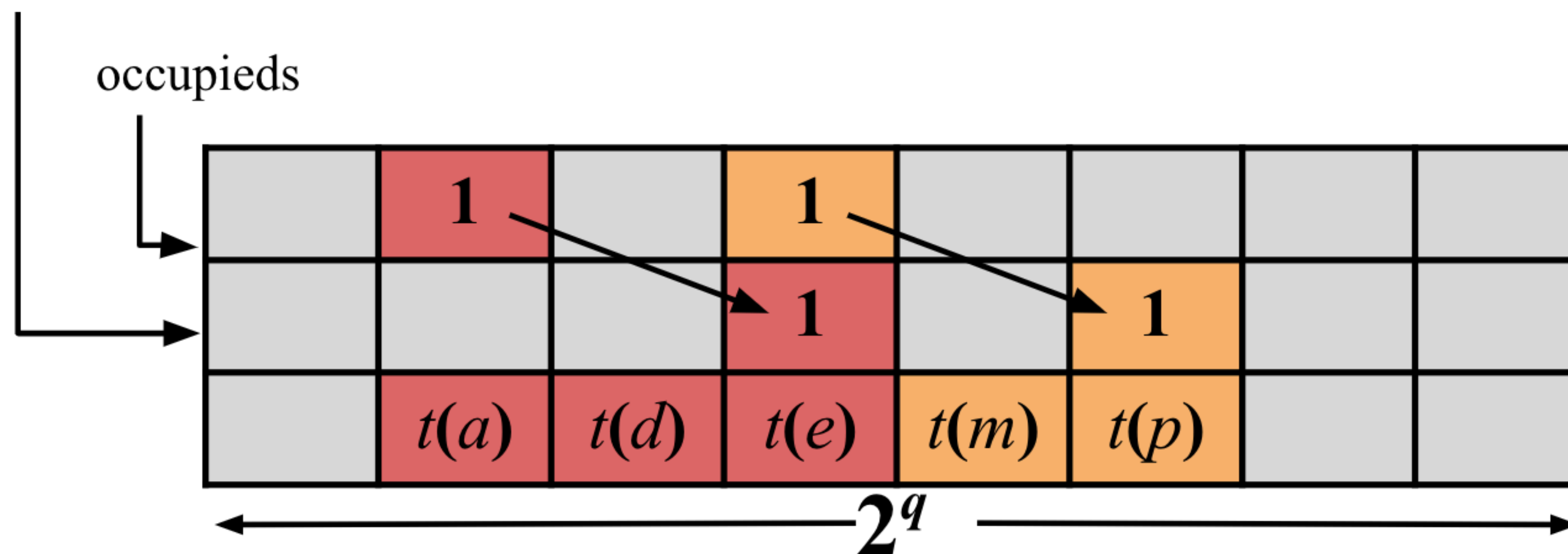
**2 meta-bits per slot.**

$h(x) \rightarrow h_0(x) \parallel h_1(x)$



runends

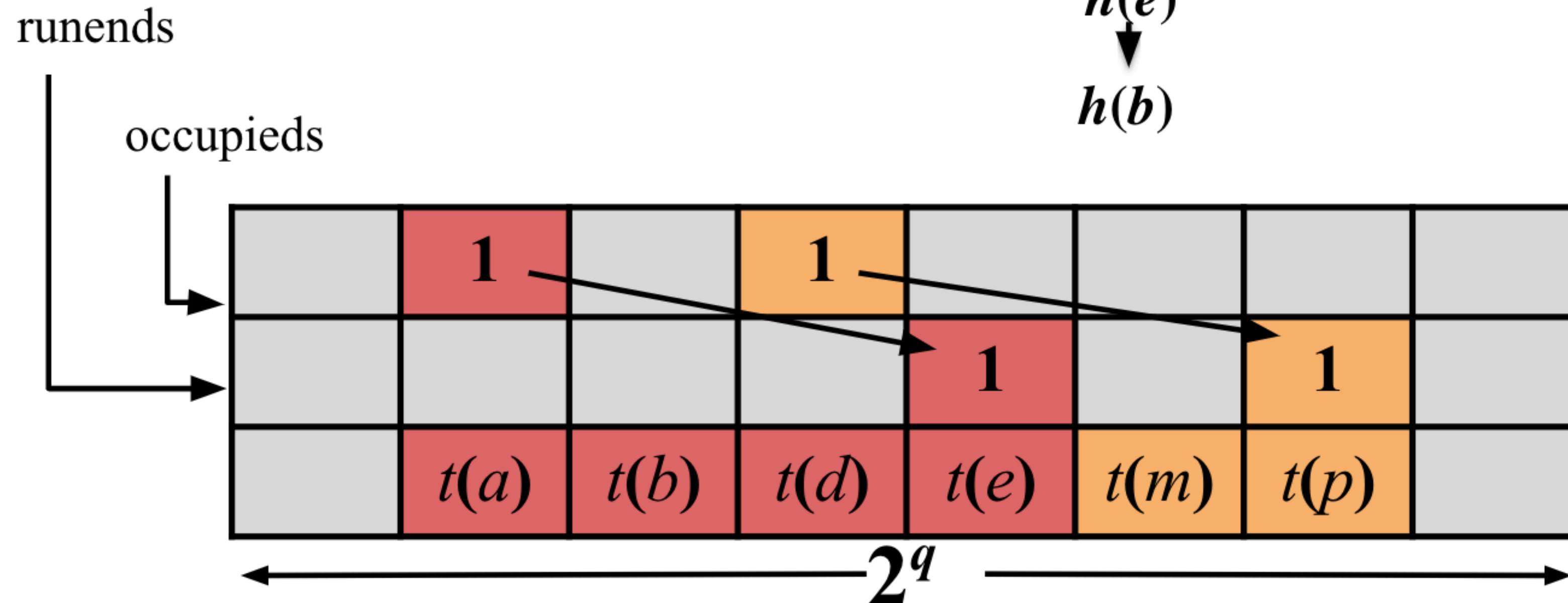
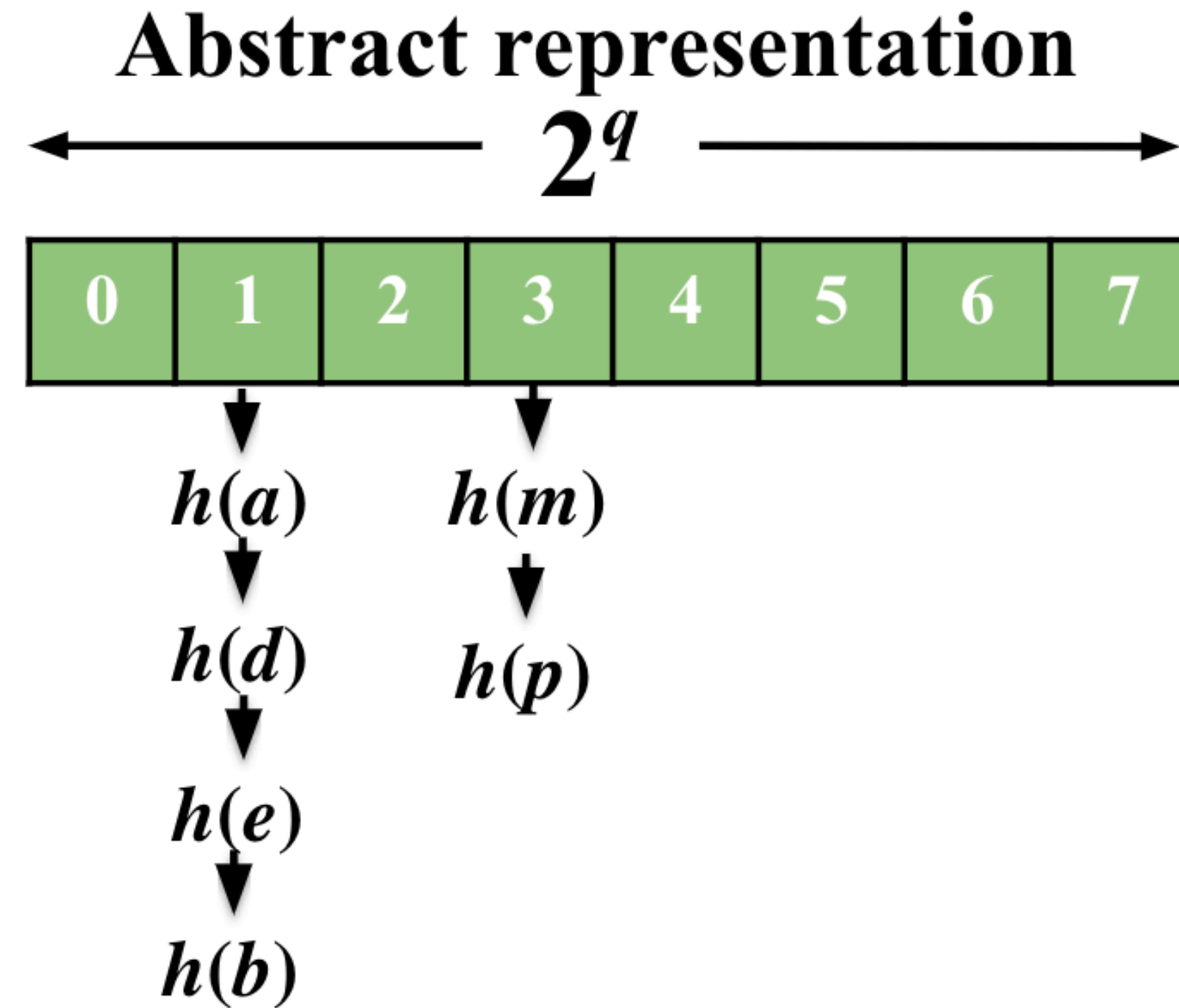
occupieds



# Resolving collisions in the QF

**Implementation:**  
**2 meta-bits per slot.**

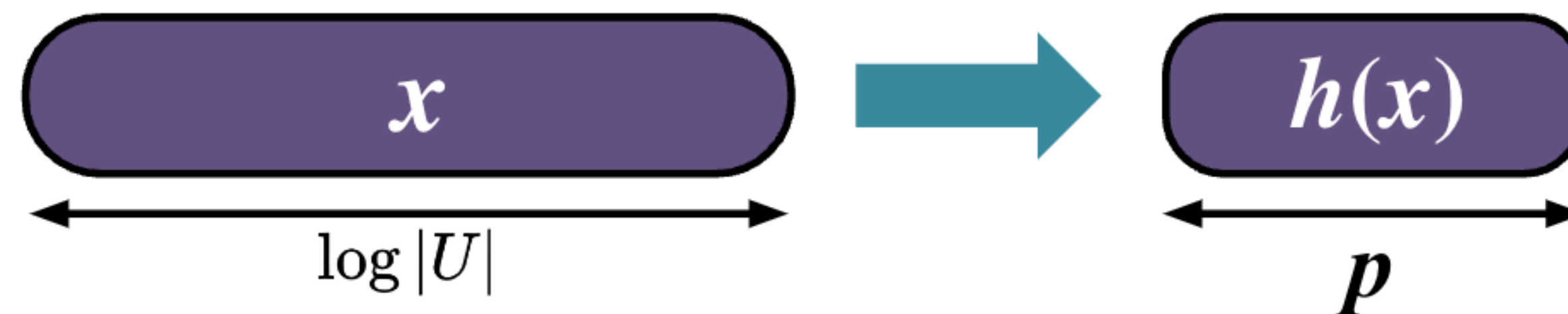
$$h(x) \dashrightarrow h_0(x) \parallel h_1(x)$$





# Quotienting enables many features in the QF

- Good cache locality
- Efficient scaling out-of-RAM
- Deletions
- Enumerability/Mergeability
- Resizing
- Maintains count estimates or associate values
- Uses variable-sized encoding for counts **[Counting quotient filter]**

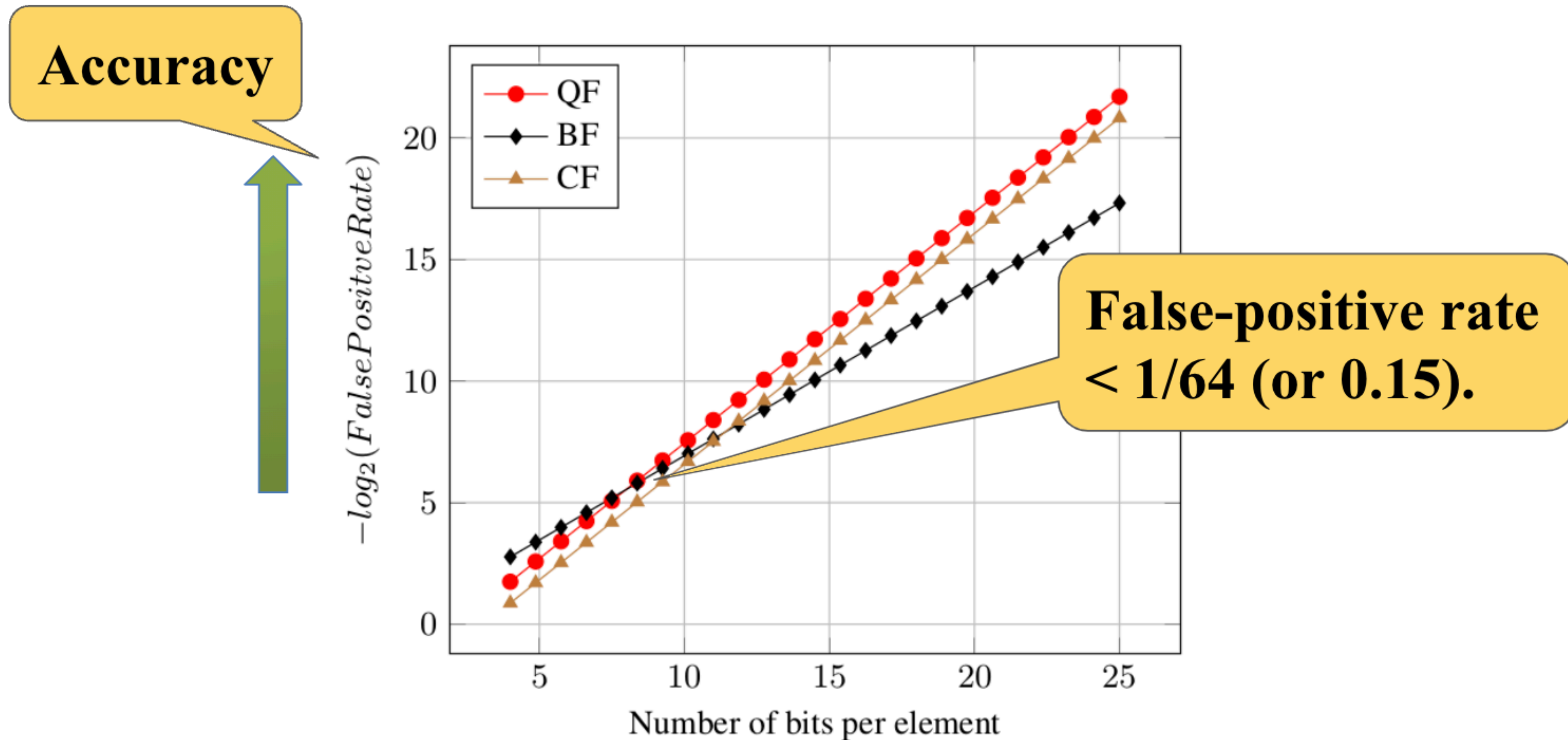


# Quotient filters use less space than Bloom filters for all practical configurations

	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

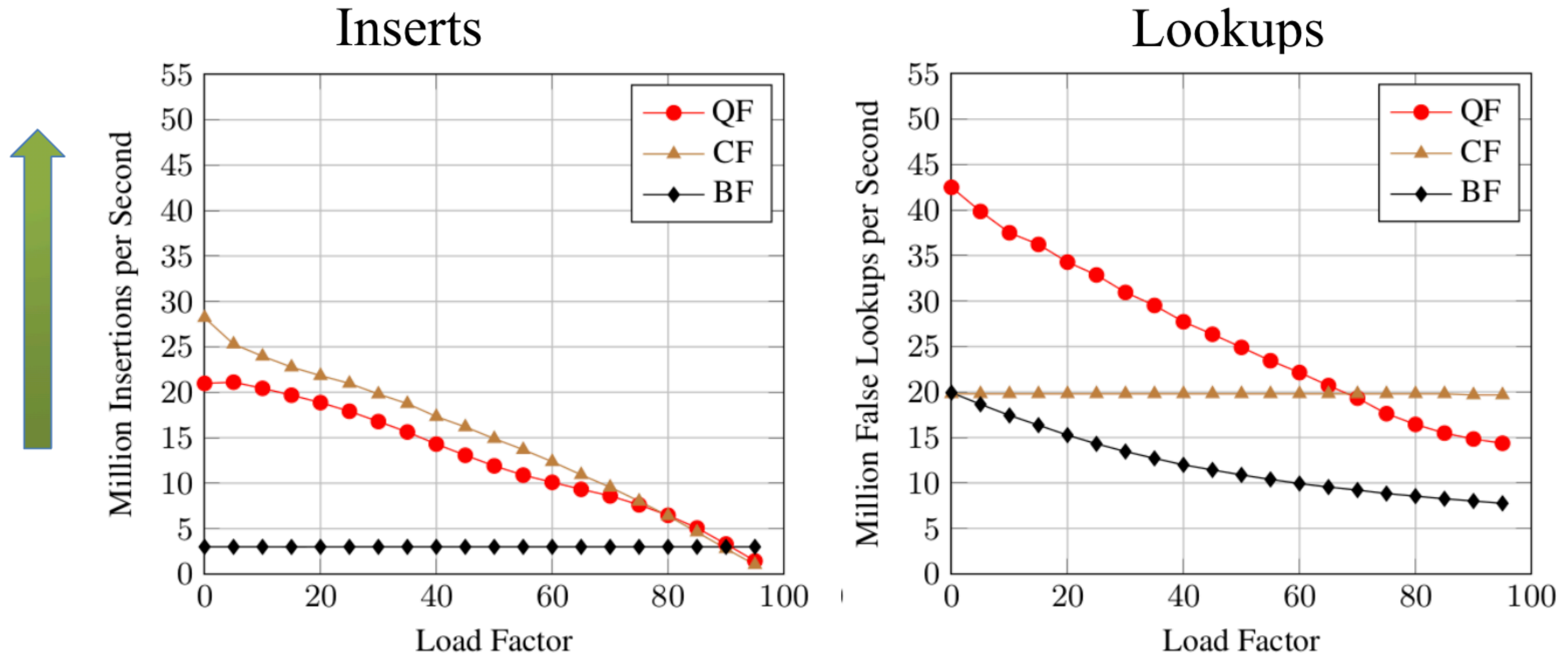
# Quotient filters use less space than Bloom filters for all practical configurations



**Bloom filter:  $\sim 1.44 \log(1/\epsilon)$  bits/element.**

**Quotient filter:  $\sim 2.125 + \log(1/\epsilon)$  bits/element.**

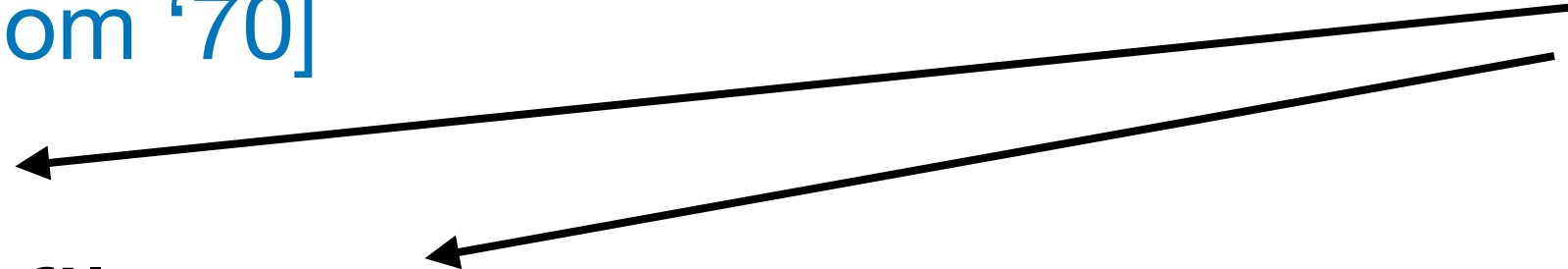
# Quotient filters perform better (or similar) to other non-counting filters



- Insert performance is similar to the state-of-the-art non-counting filters
- Query performance is significantly faster at low load-factors and slightly slower at higher load-factors

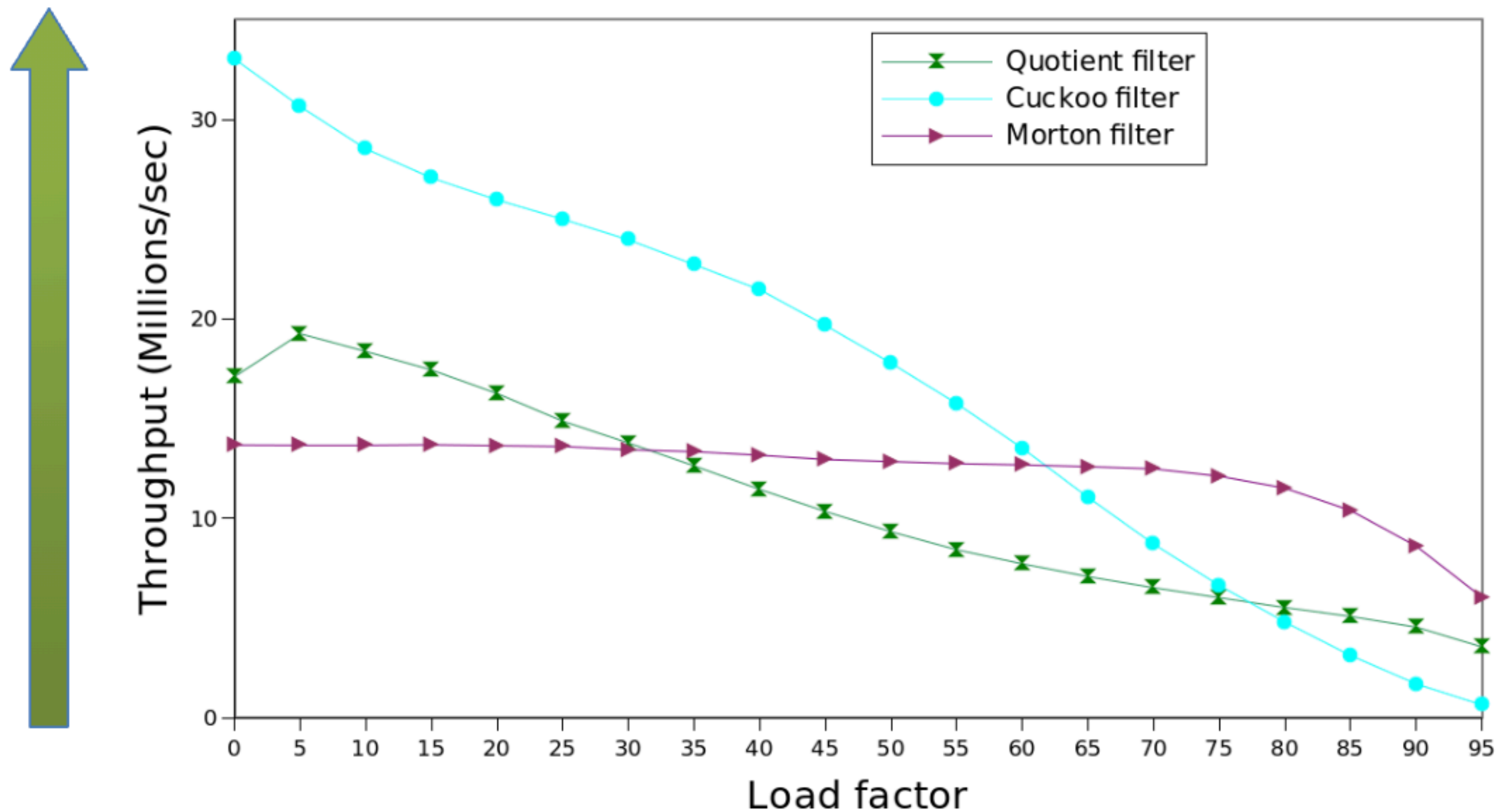


# Summary of filters

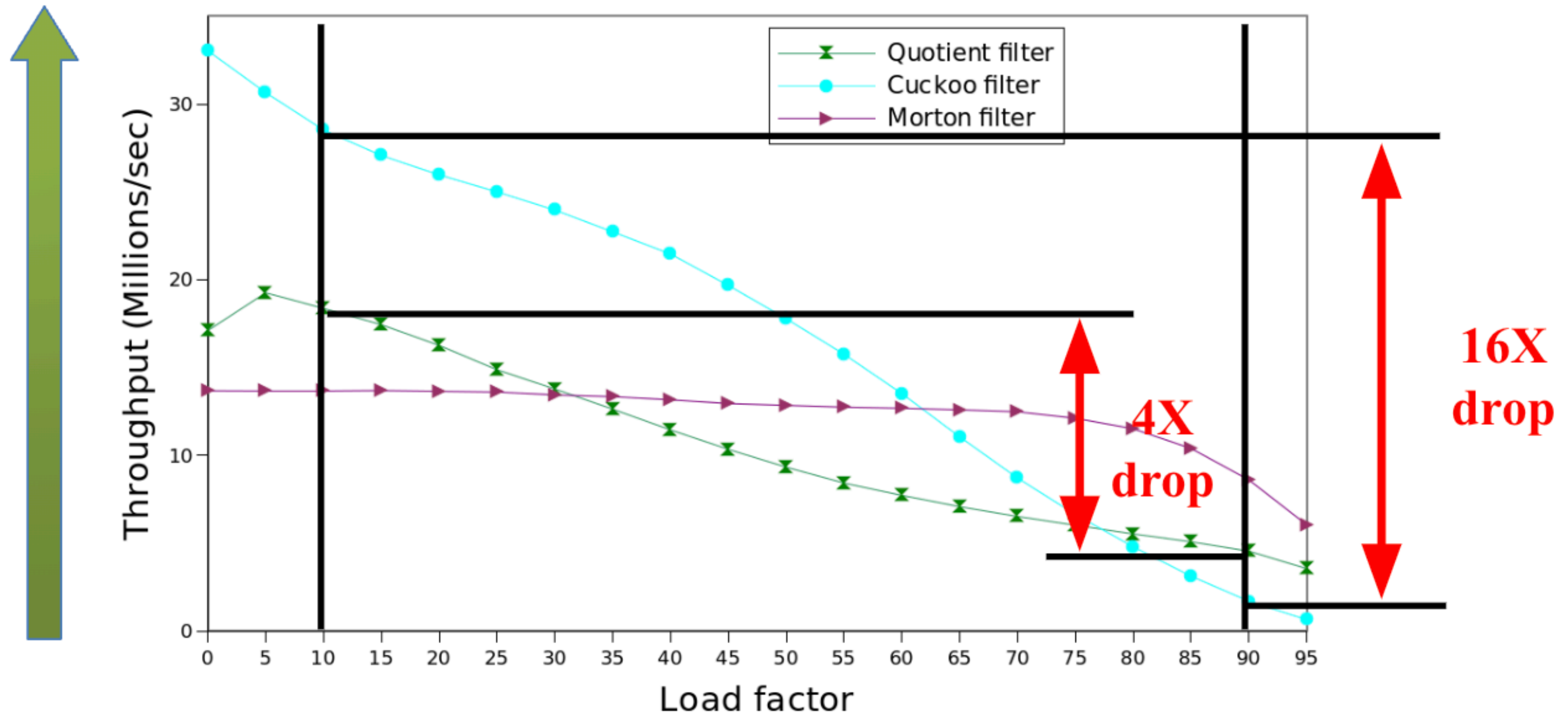
- Bloom filters [[Bloom '70](#)]
  - Quotient filters
  - Cuckoo/Morton filters [[Fan et al. '14](#), [Breslow & Jayasena '18](#)]
  - Others
    - Mostly based on perfect hashing and/or linear algebra
    - Mostly static
- e.g., Xor filters [[Graf & Lemire '20](#)]
- State of the art in  
practical dynamic filters**
- 

# Current filter performance

- Performance suffers due to **high overhead of collision resolution** at high load factors
- Problem: many applications are write-heavy and maintain hash tables at **high load factors.**

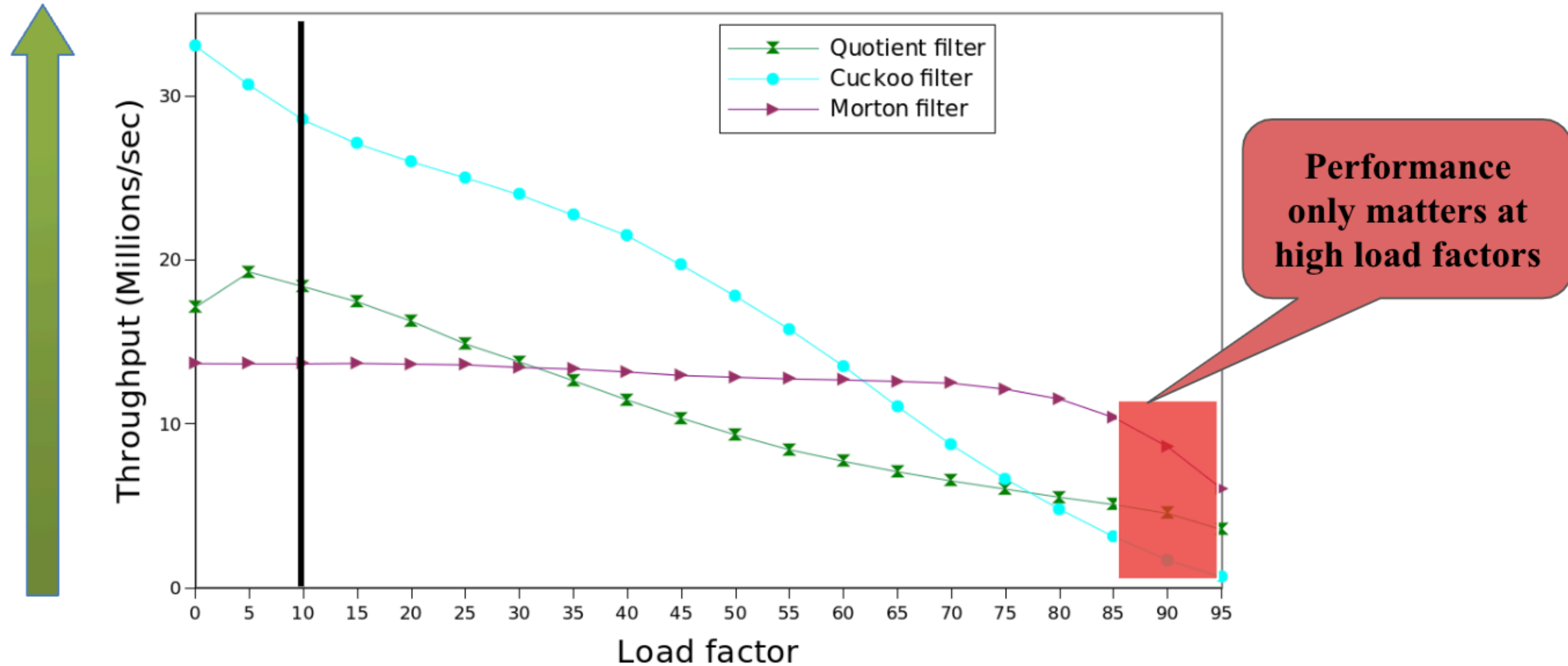


# Space and speed tradeoff in current filters



Applications must choose between space and speed.

# Space and speed tradeoff in current filters



Update-intensive applications maintain filters close to full.

# Why quotient filters slow down

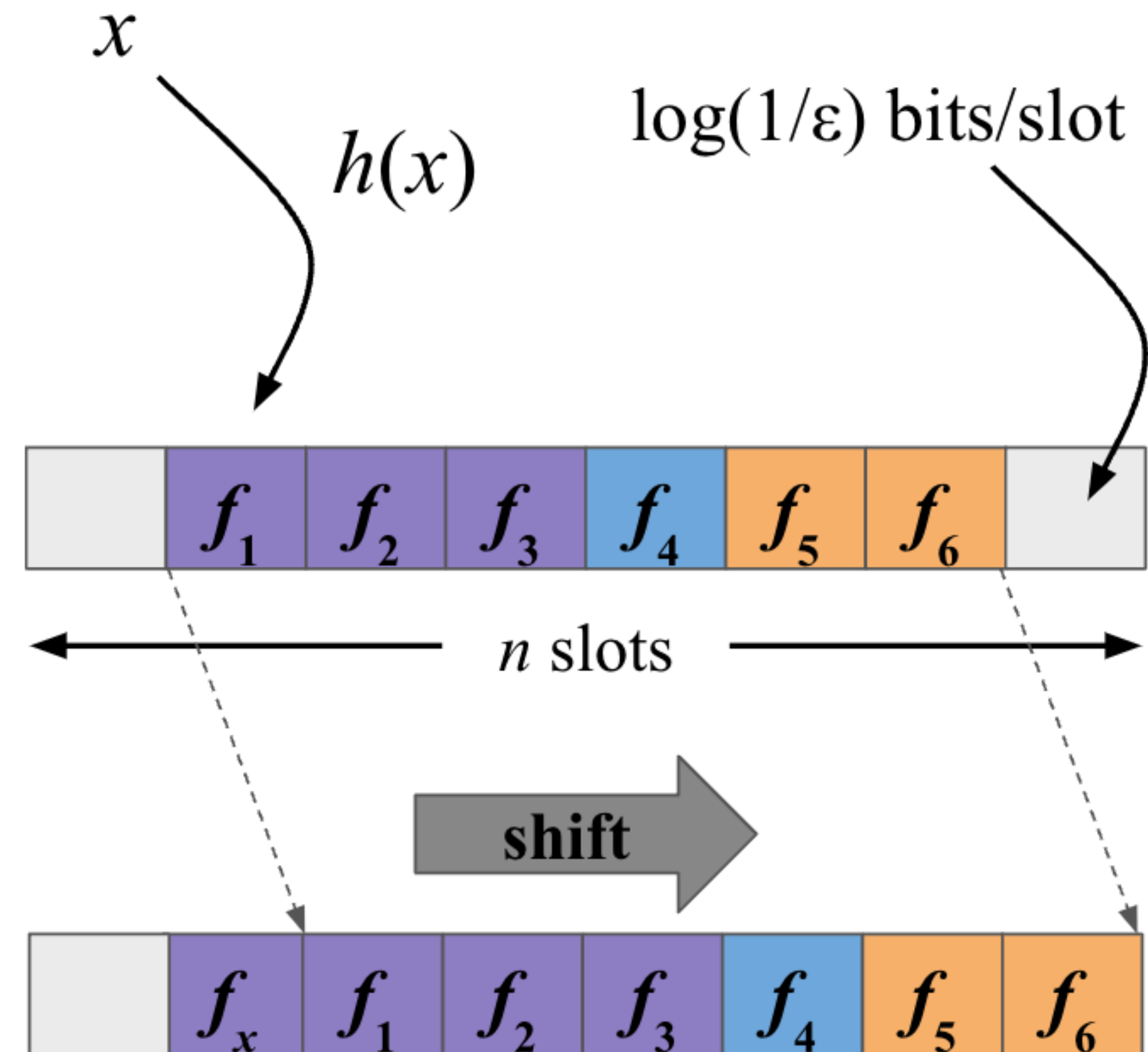
Quotient filters use Robin-Hood hashing (a variant of linear probing)

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.

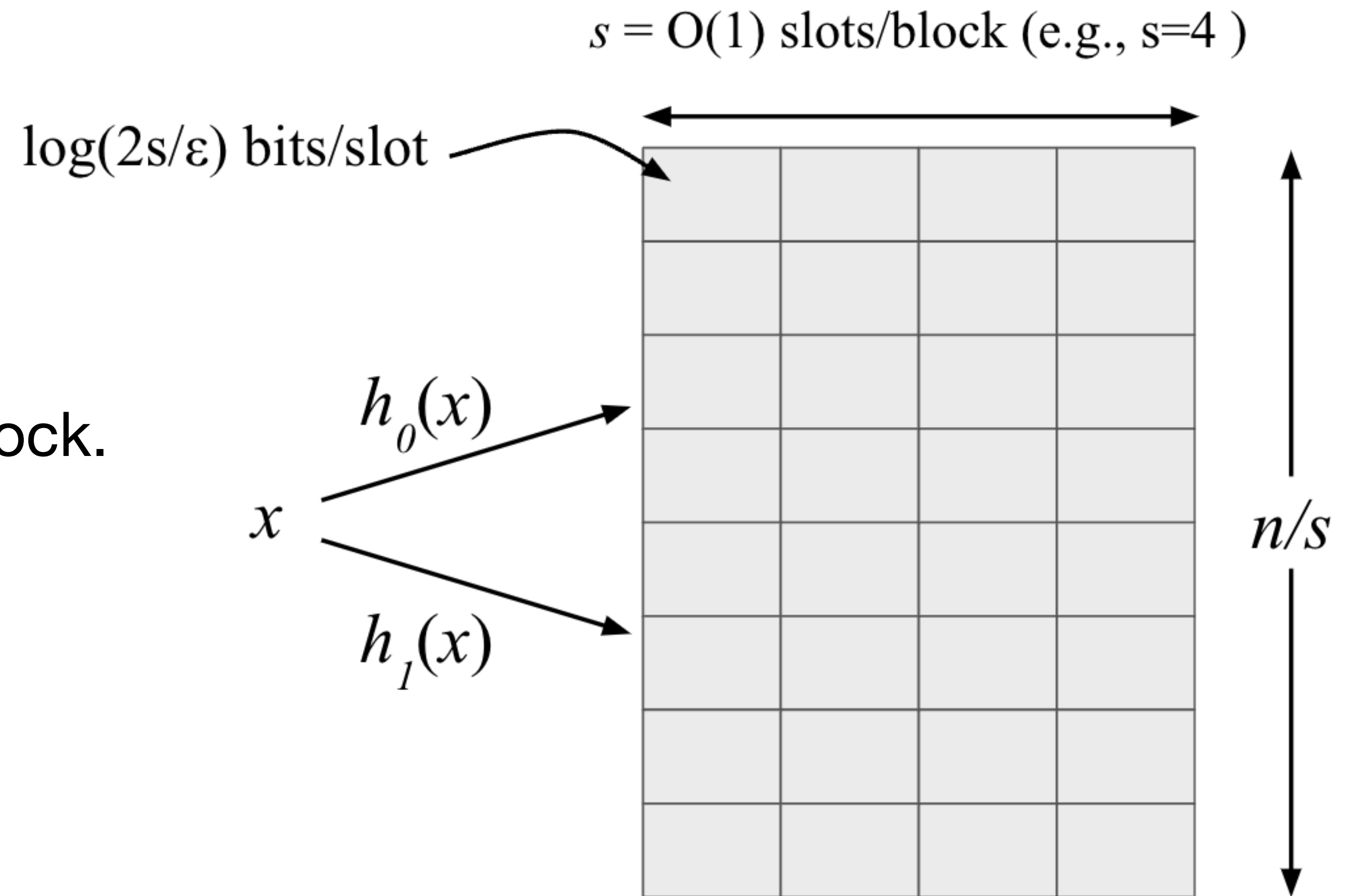




# Why cuckoo filters slow down

To insert item  $x$ :

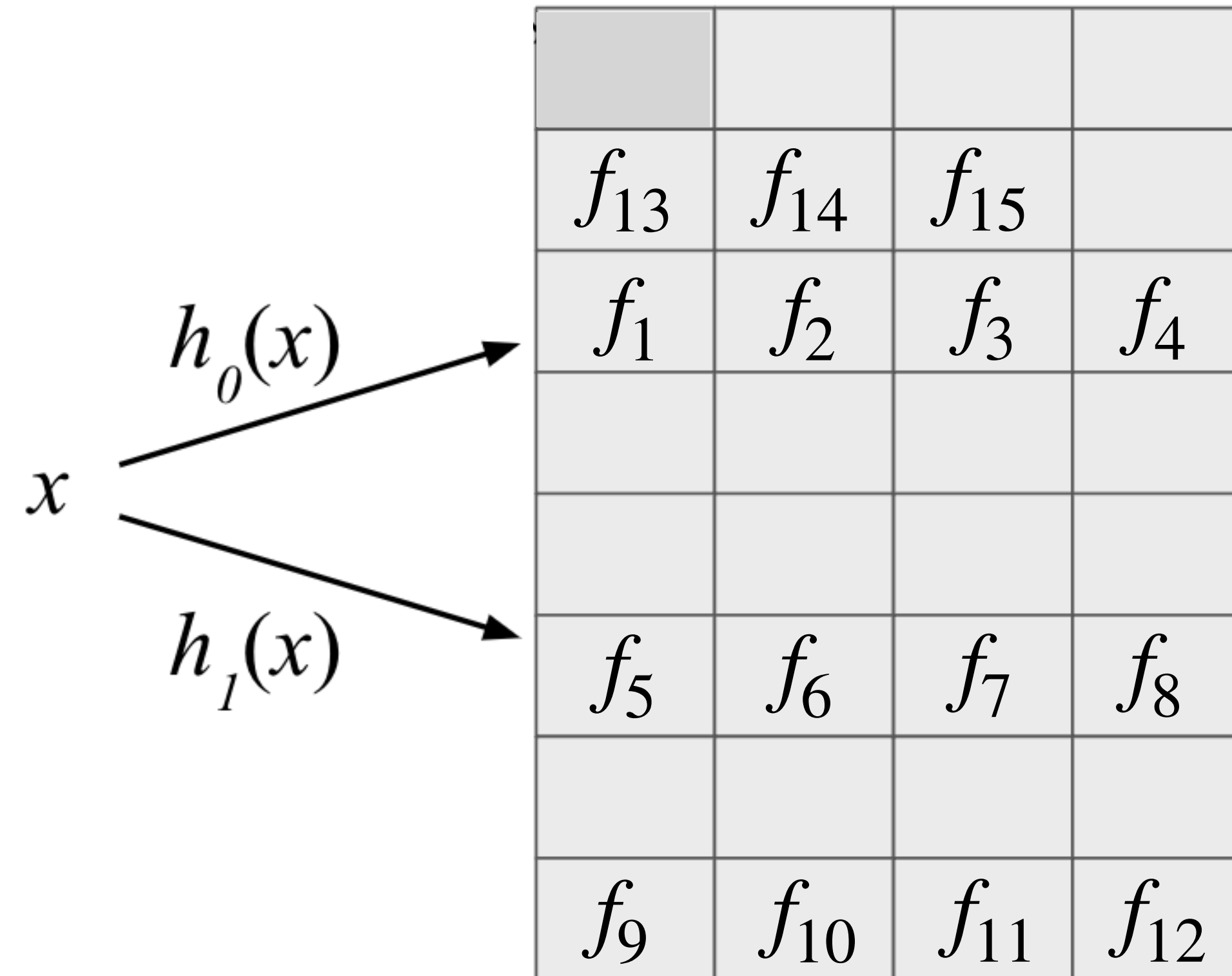
1. Compute  $h_0(x)$  and  $h_1(x)$ .
2. Insert  $f(x)$  into emptier block.
3. Kick an item if needed.



# Why cuckoo filters slow down

To insert item  $x$ :

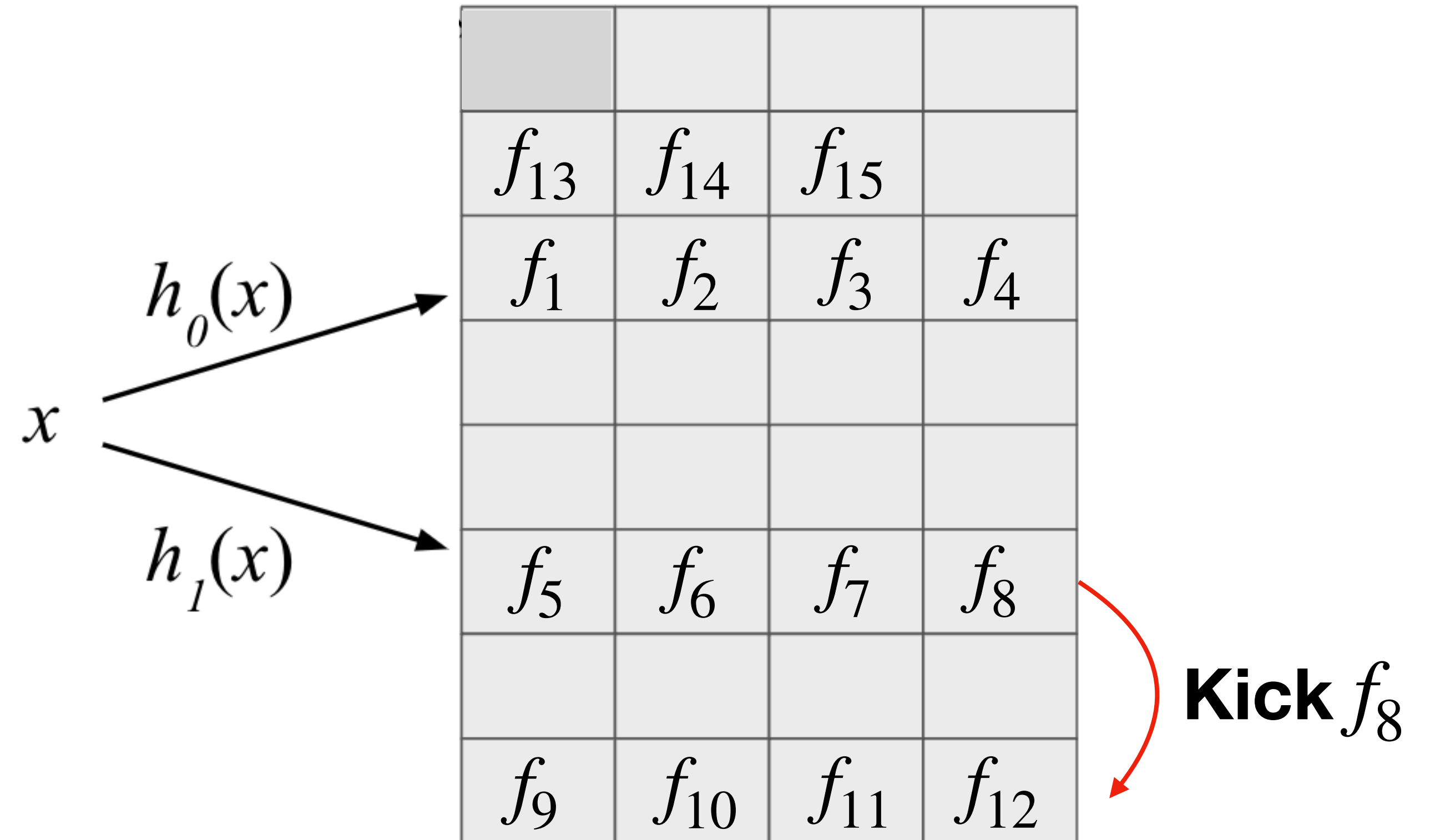
1. Compute  $h_0(x)$  and  $h_1(x)$ .
2. Insert  $f(x)$  into emptier block.
3. Kick an item if needed.



# Why cuckoo filters slow down

To insert item  $x$ :

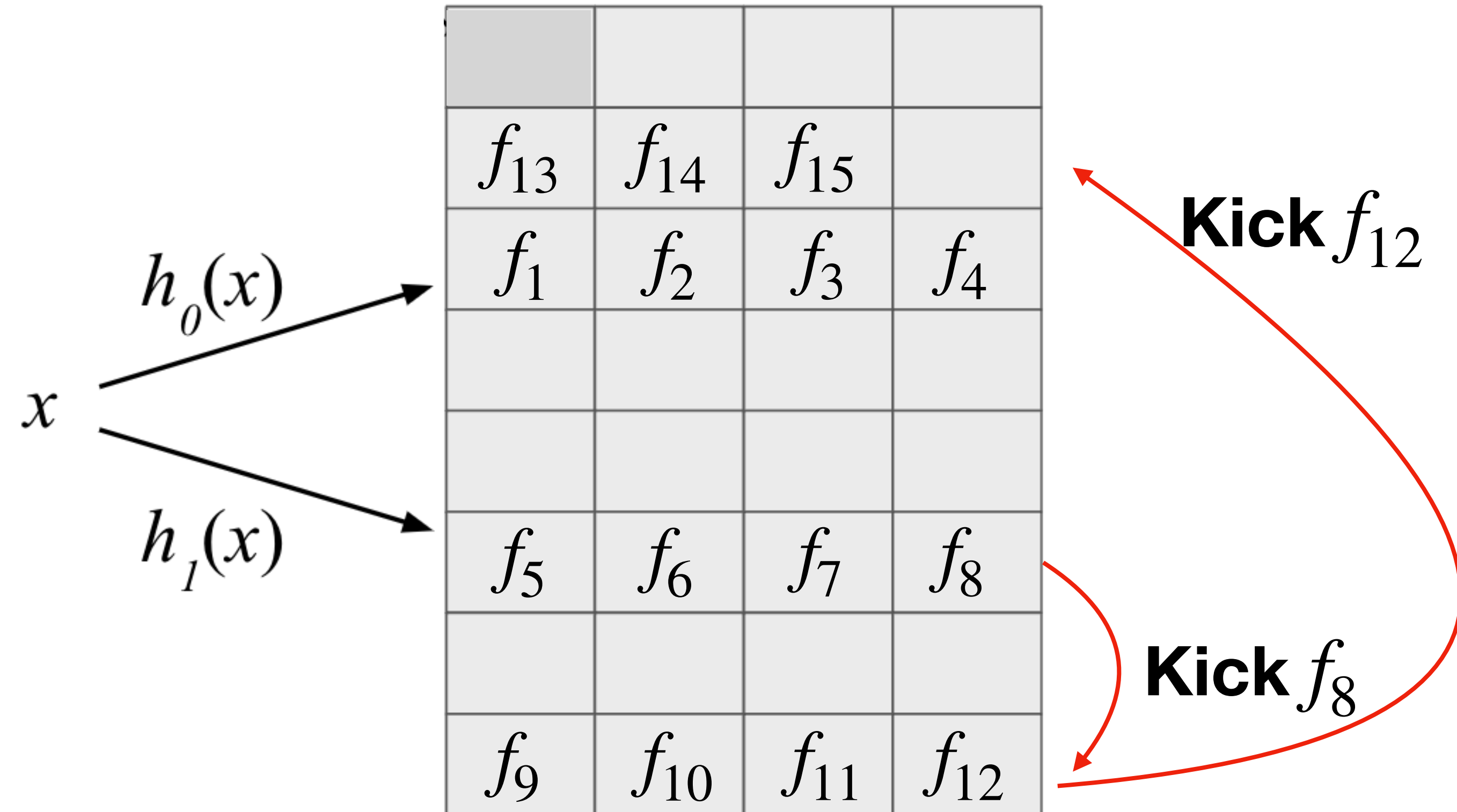
1. Compute  $h_0(x)$  and  $h_1(x)$ .
2. Insert  $f(x)$  into emptier block.
3. Kick an item if needed.



# 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.

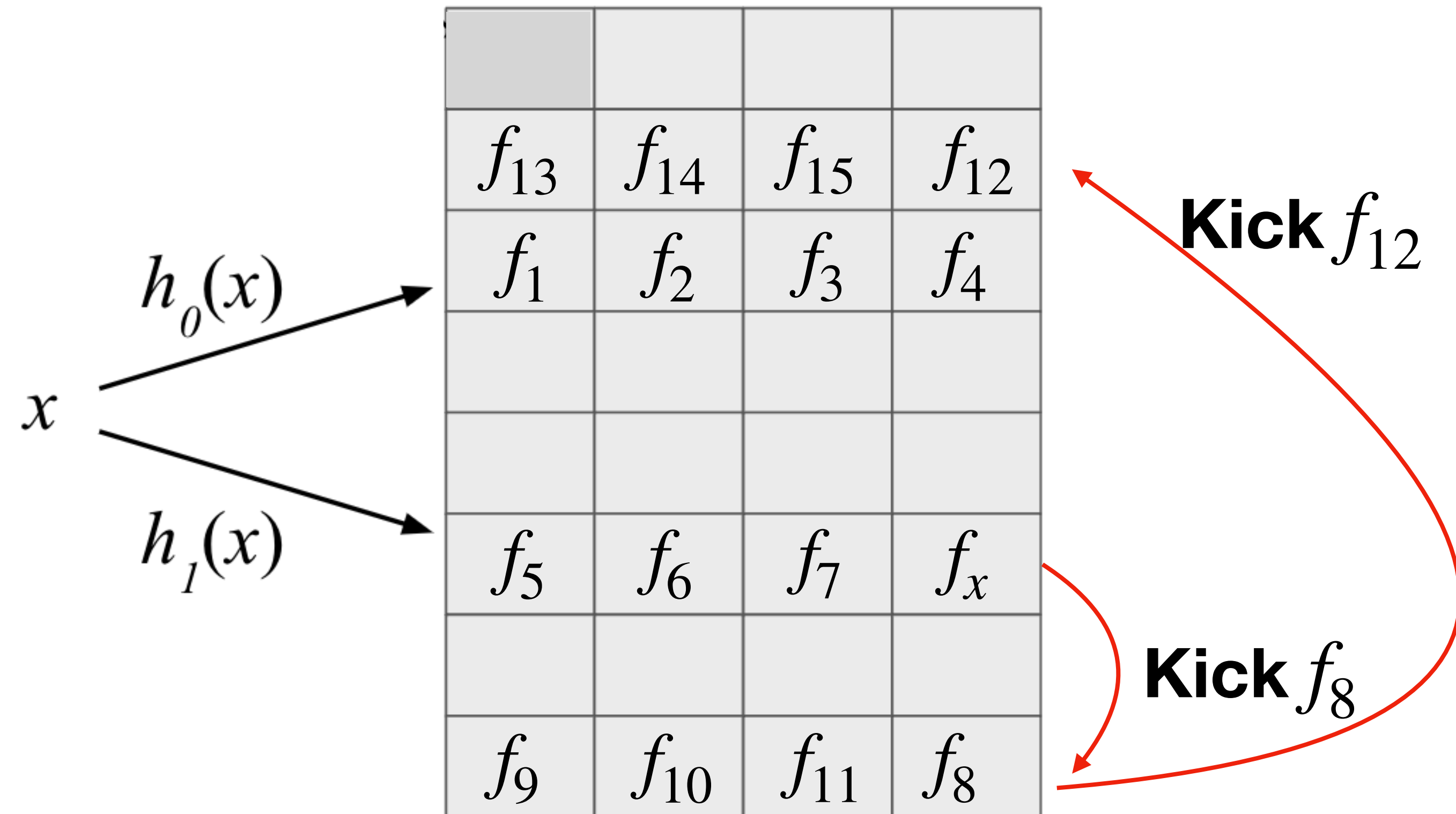


# Why cuckoo filters slow down

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)



# Aside: Power of two choices

Suppose that  $n$  balls are placed into  $n$  bins. Let the **load** of a bin be the number of balls in that bin after all the balls have been thrown. What is the **maximum load** over all bins once the process terminates?

Theorem: If the balls are thrown into bins **independently** and uniformly at random, the maximum load is  $O(\log(n)/\log \log(n))$ .

Theorem: For each ball, if we **choose 2 bins** independently and uniformly at random and place the ball into the less full one, the maximum load is  $O(\log \log(n))$ .

“Expected length of the longest probe sequence in hash code searching,” Gonnet. JACM '81.

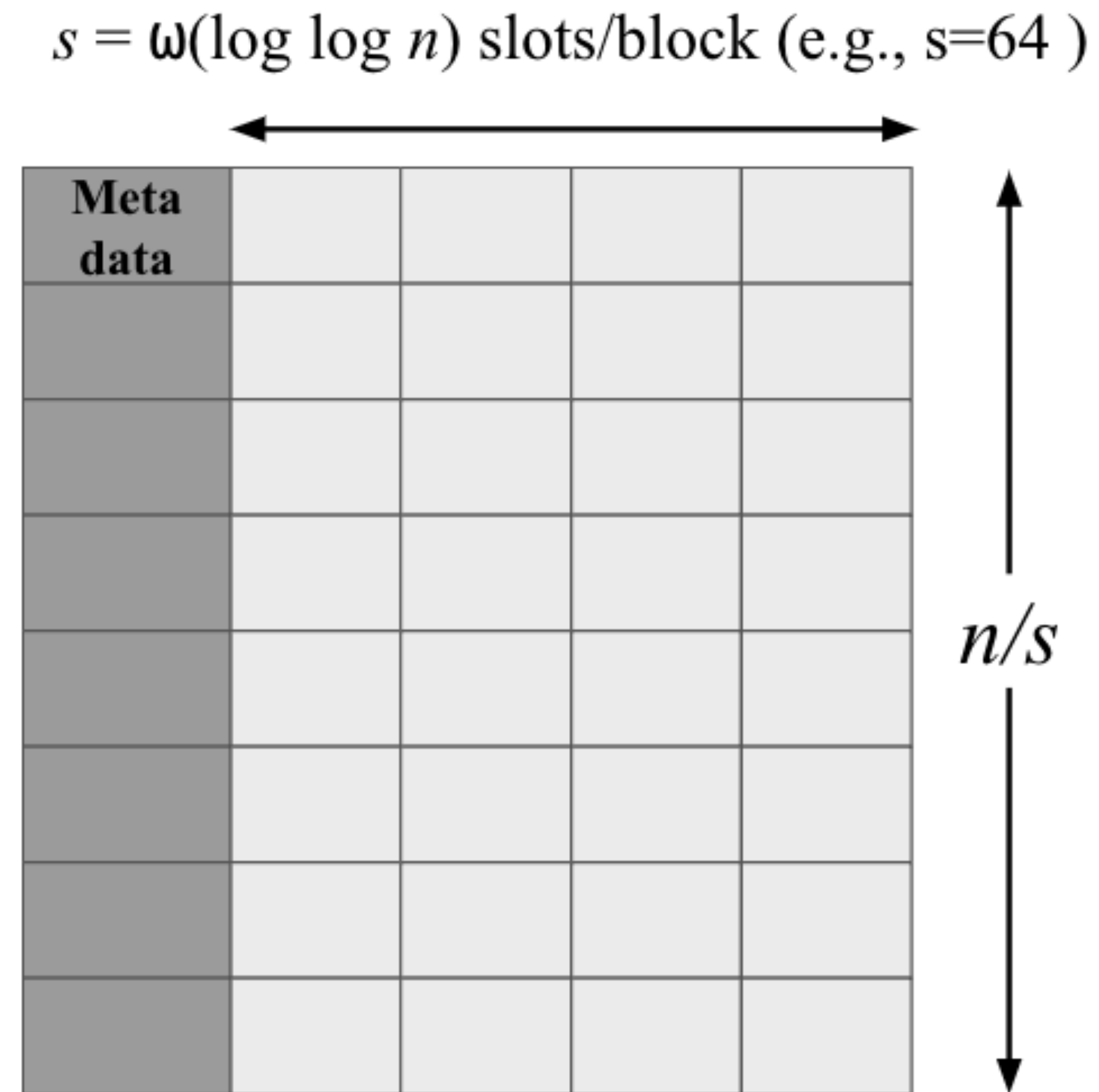
“Balanced allocations,” Azar, Broder, Karlin, Upfal. STOC '94.

More results - <https://www.eecs.harvard.edu/~michaelm/postscripts/mythesis.pdf>

# Cuckoo filter performance

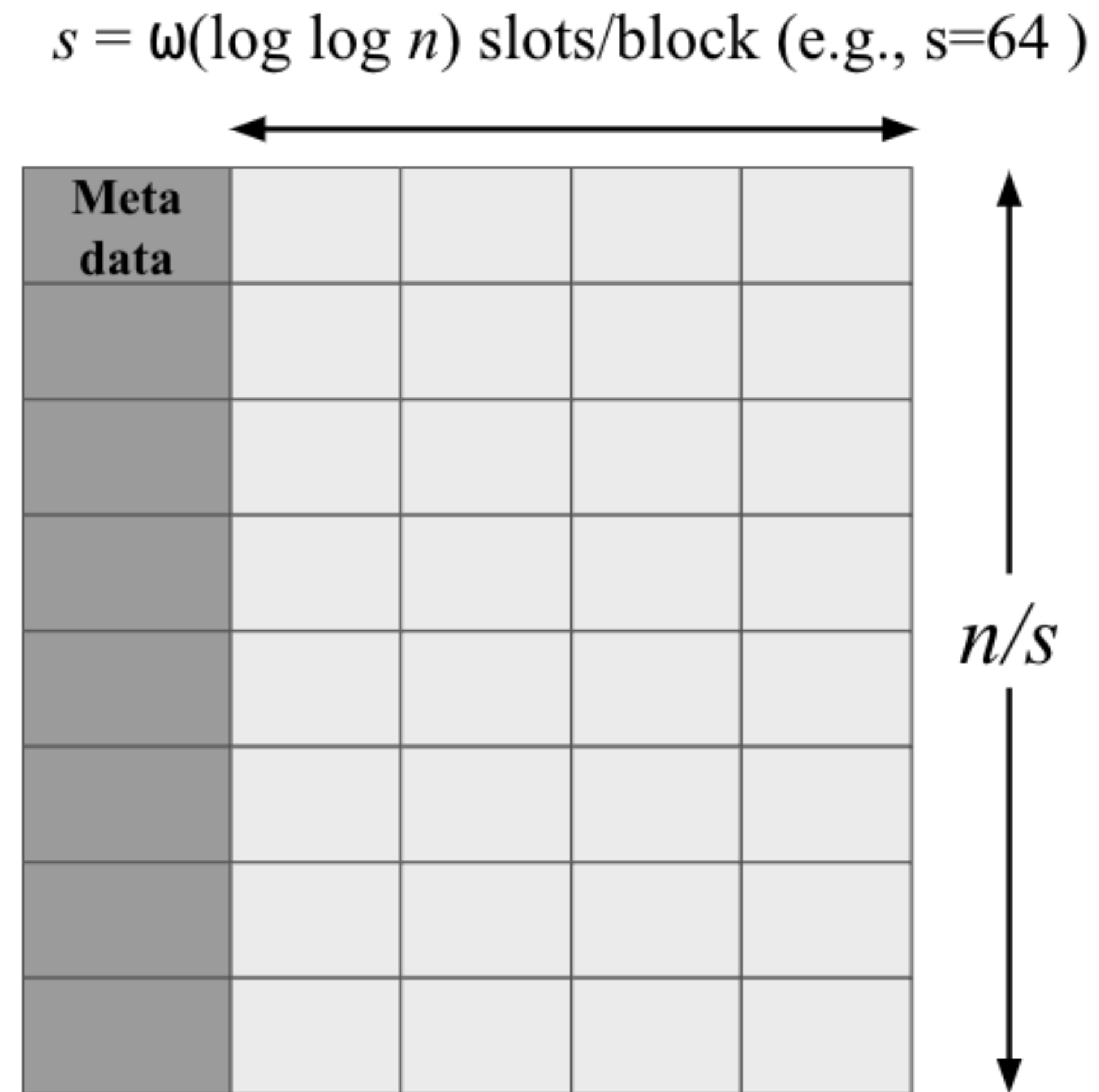
	Optimal	Cuckoo filter
Space (bits)	$\approx n \log(1/\epsilon) + \Omega(n)$	$\approx n \log(1/\epsilon) + 3n$
CPU cost	$O(1)$	up to 500
Data locality	$O(1)$ probes	random probes

# Vector quotient filter design



# Vector quotient filter design

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

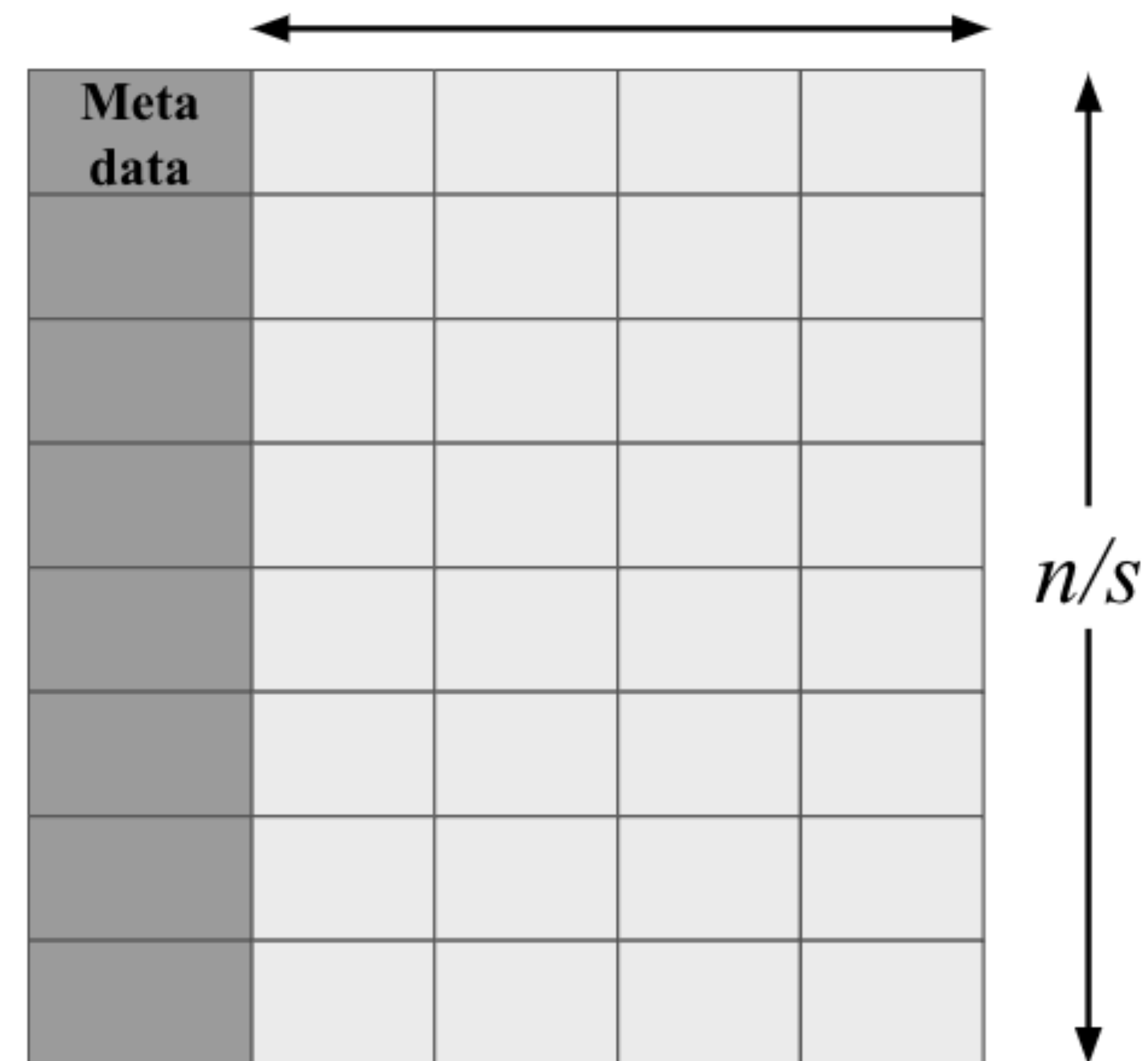


# Vector quotient filter design

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.,  $s=64$ )



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.~~



# How to avoid kicking

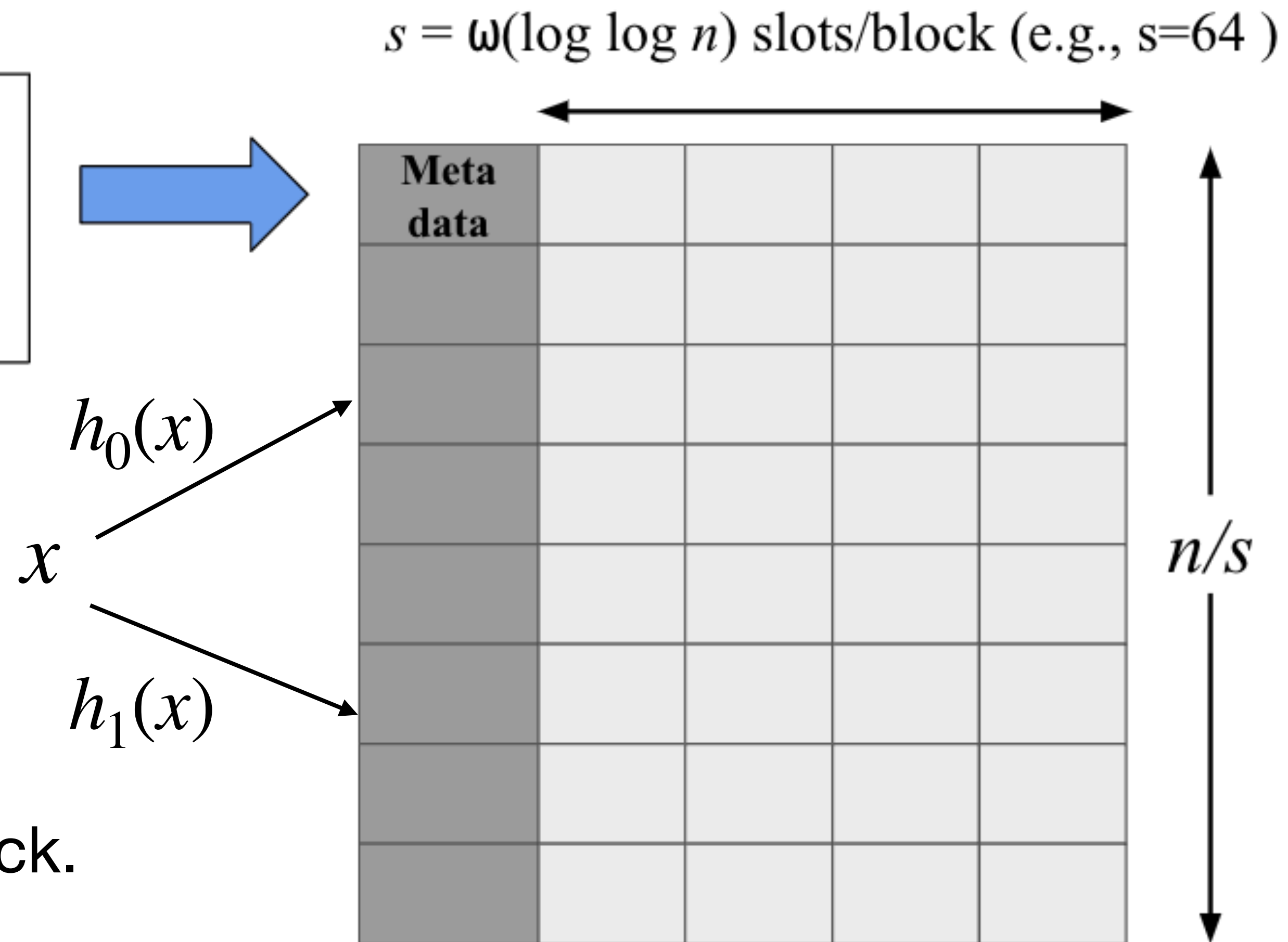
In a VQF, an insertion fails if both blocks are full via power-of-two-choices.

Theorem [Berenbrink et al]: If we toss  $m$  balls into  $n$  bins using the power-of-two-choices, then, with high probability, the maximum load of any bin is  $m/n + O(\ln \ln n)$ .

Therefore, to create a VQF for  $n$  items, we allocate  $k = O(n \ln n)$  **blocks**, each with capacity  $s = n/k + \Theta(\ln \ln n)$  **items** and false positive rate  $\epsilon/2$ . By the theorem, all items can be inserted into the filter without any block reaching maximum capacity, and hence all insertions succeed whp.

# Vector quotient filter design

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

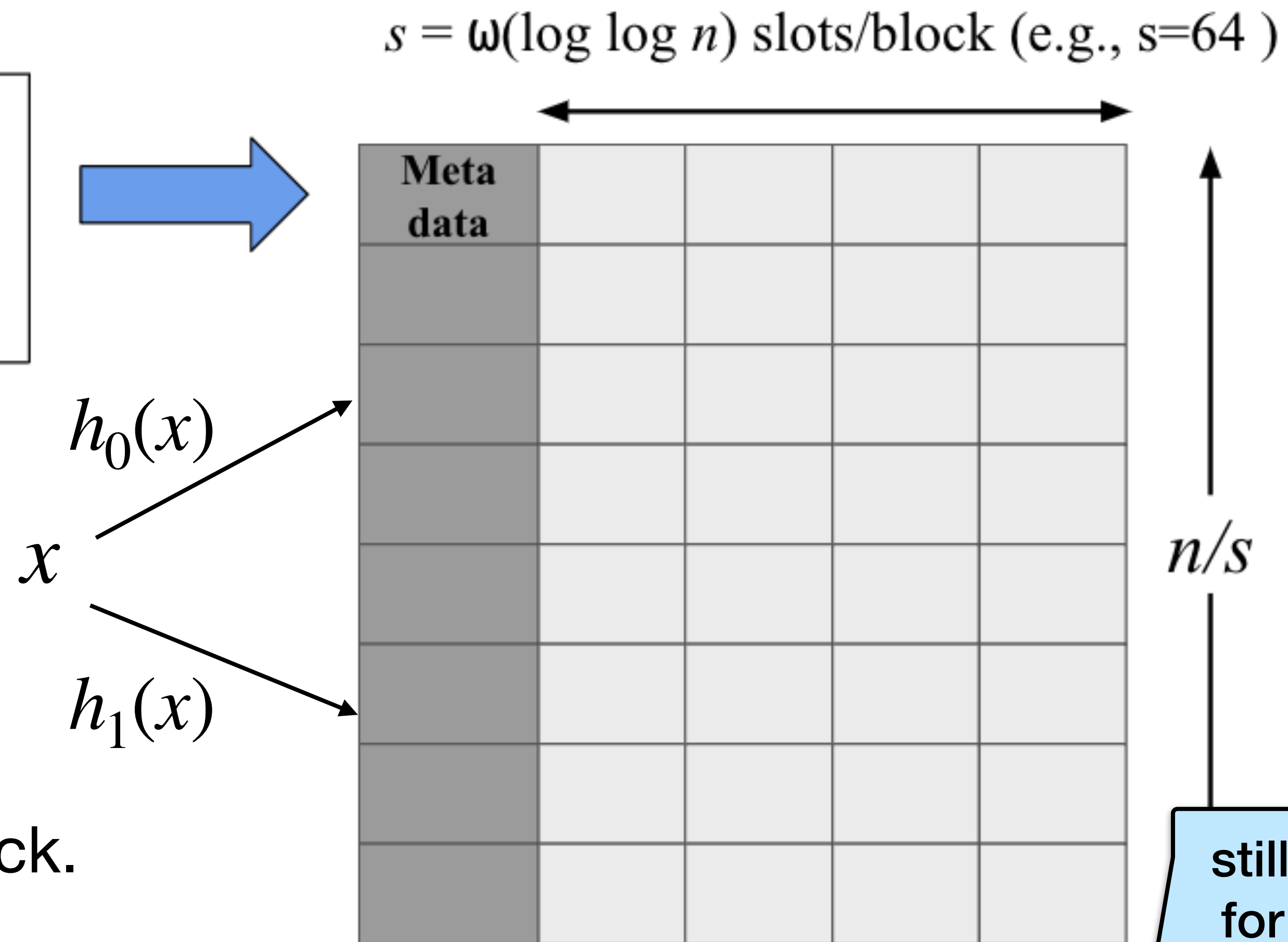


To insert item  $x$ :

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

# 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

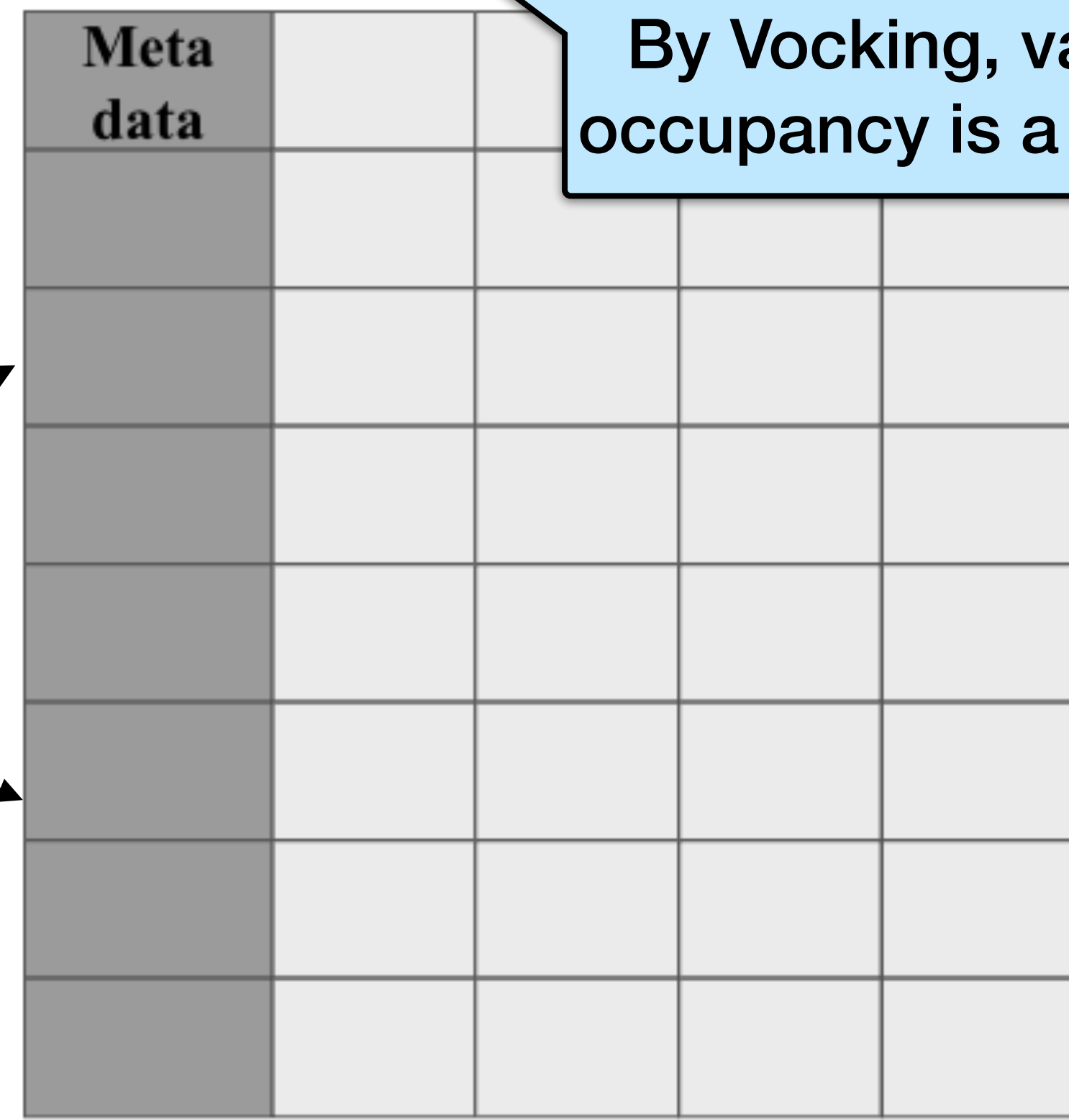
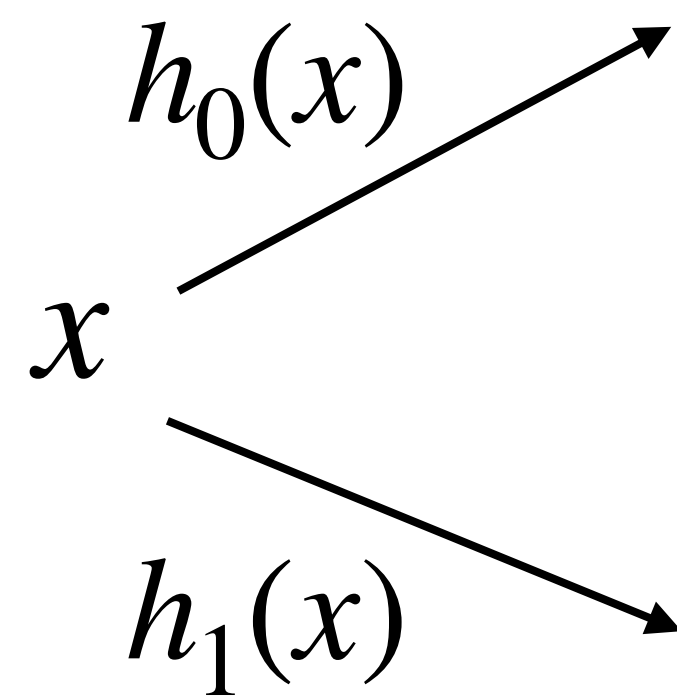
# Vector quotient filter design

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



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

By Vocking, variance in block occupancy is a lower-order term



$n/s$

still needed for deletes

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.~~

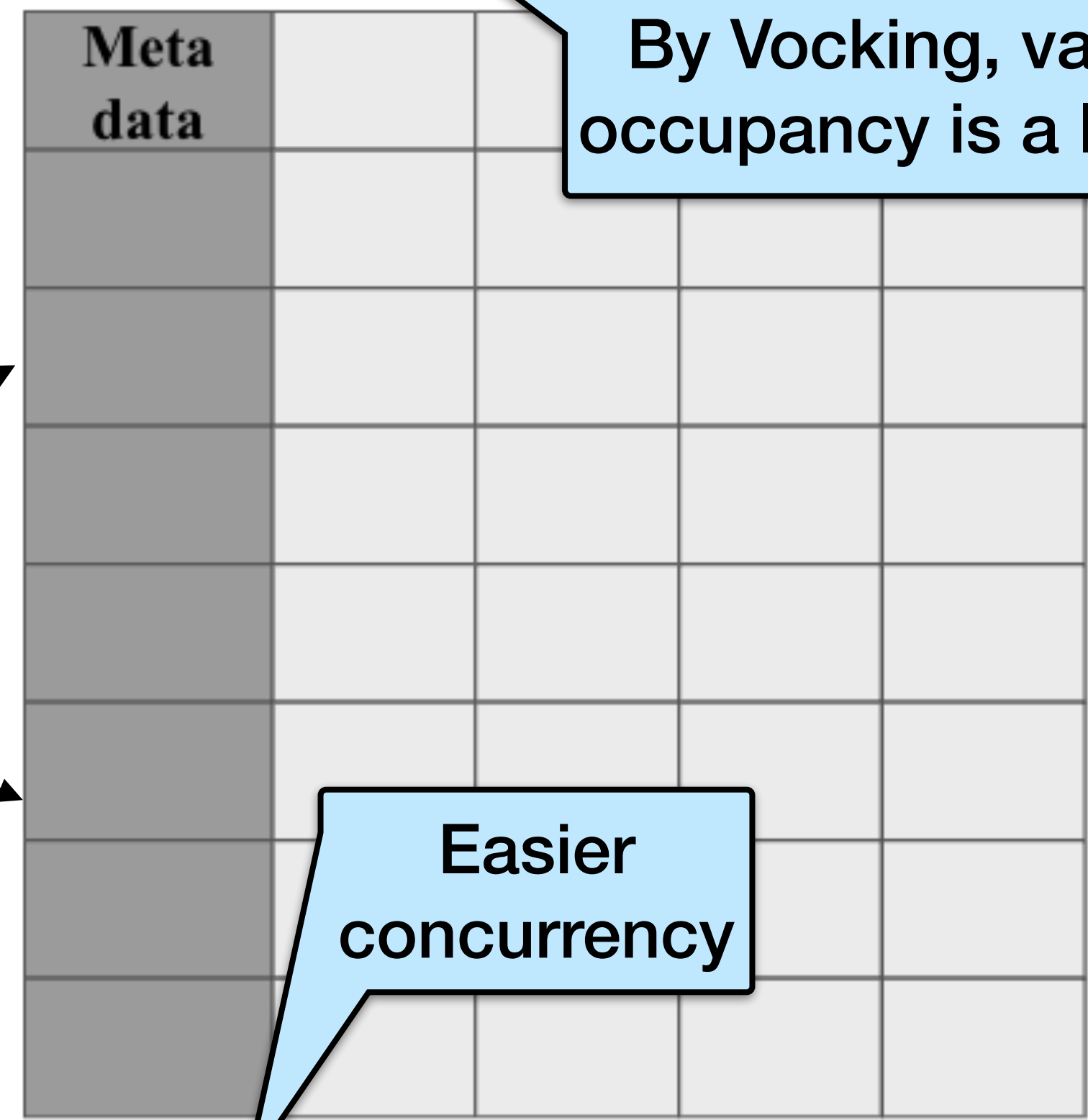
No kicking  $\rightarrow h_0(x)$  and  $h_1(x)$  can be independent for insert-only workload

# Vector quotient filter design

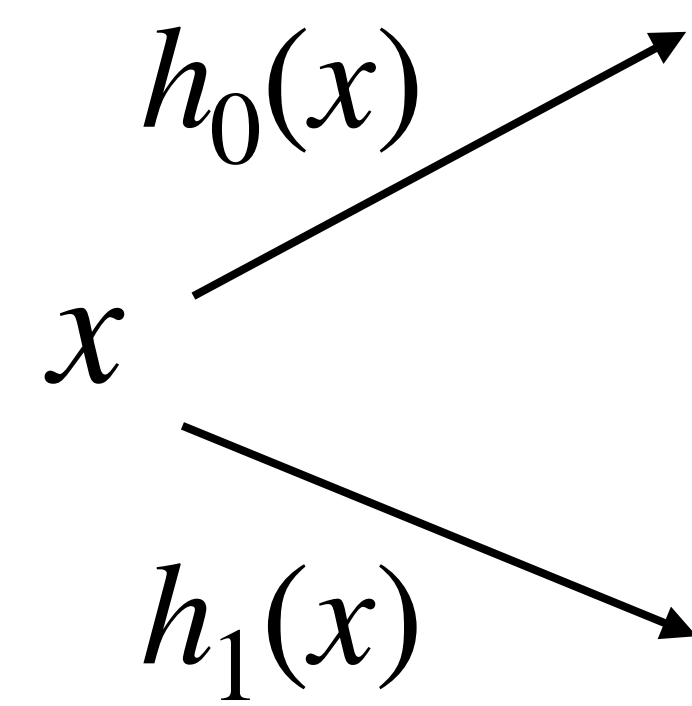
Each block is a small quotient filter with false-positive rate  $\epsilon/2$  and capacity  $s$ .



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



By Vocking, 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.~~

Easier concurrency

still needed for deletes

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



# A vectorizable mini quotient filter

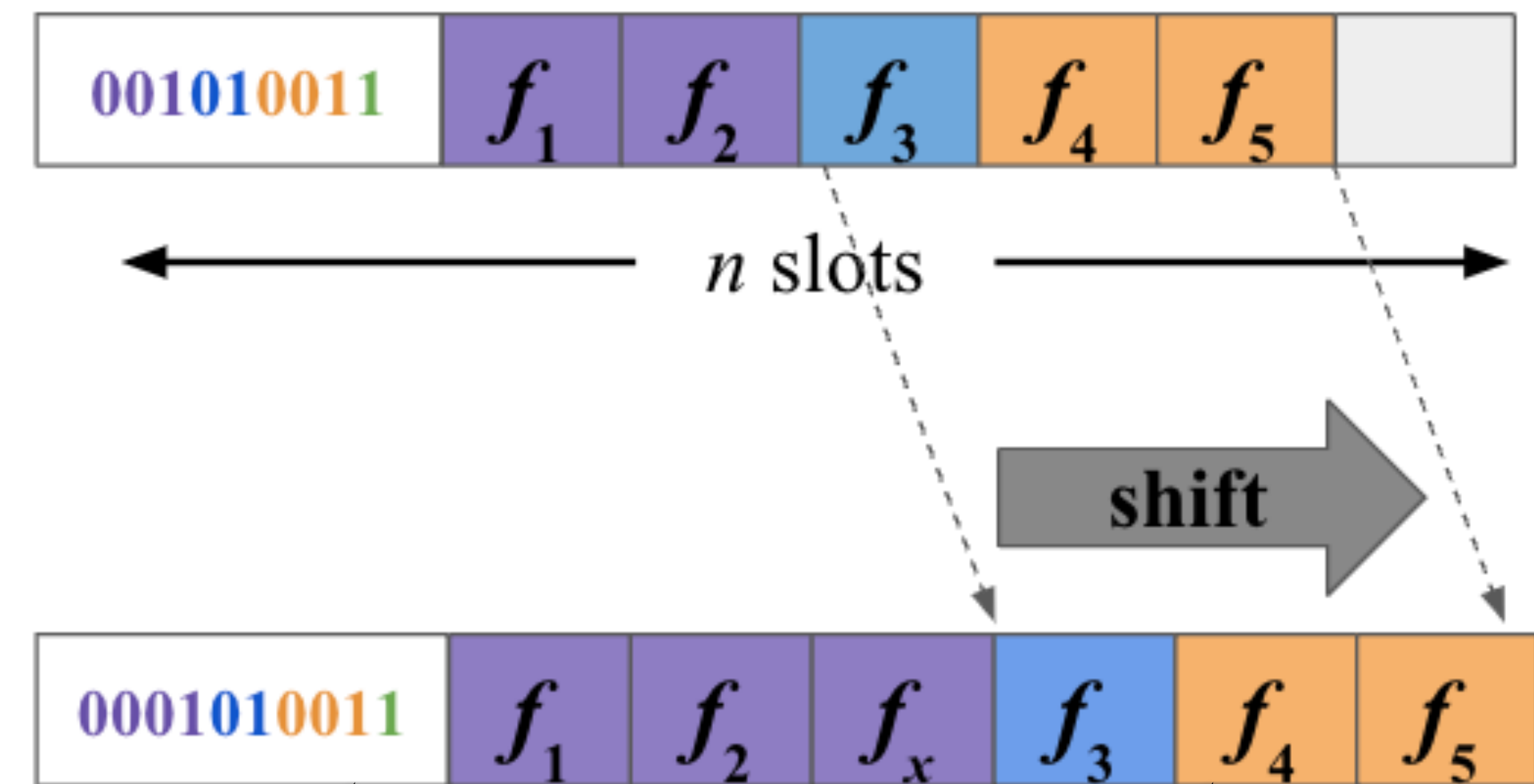
Each block has  $b$  logical buckets.

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  
with  $b = s / \ln 2$



Implemented  
with PDEP

Implemented with  
PSHUFB or VCMPPB

# A vectorizable mini quotient filter

Each block has  $b$  logical buckets.

Fingerprints of each bucket are stored together.

We  
bo

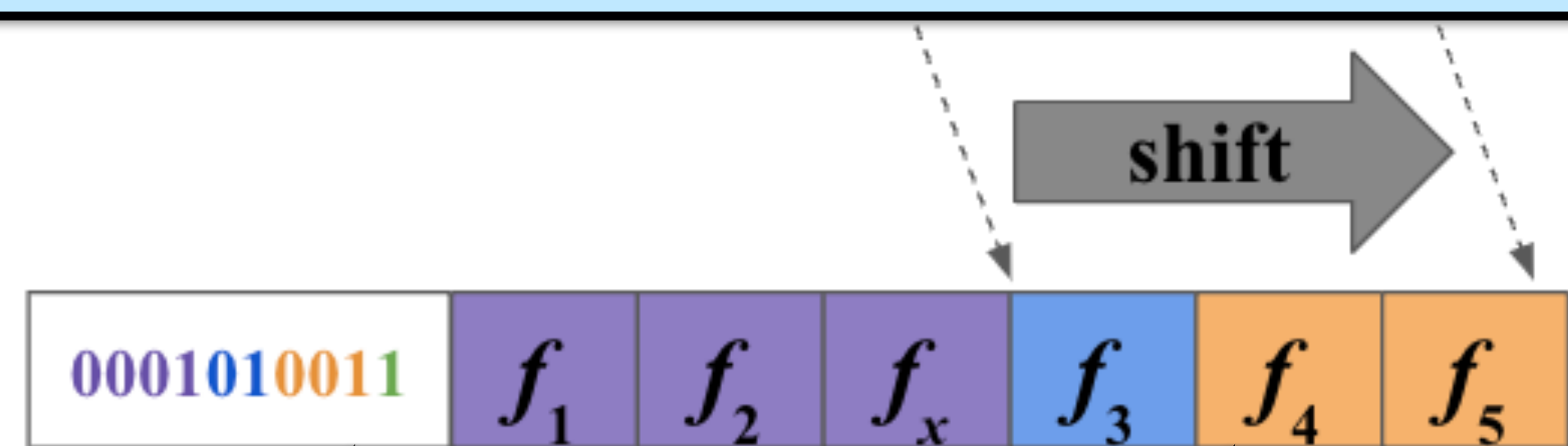
Operations take constant time in a vector model of computation for vectors of size  $\omega(\log \log n)$  [Blelloch 90]. e.g., using AVX-512 instructions.

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

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

Implemented with PDEP

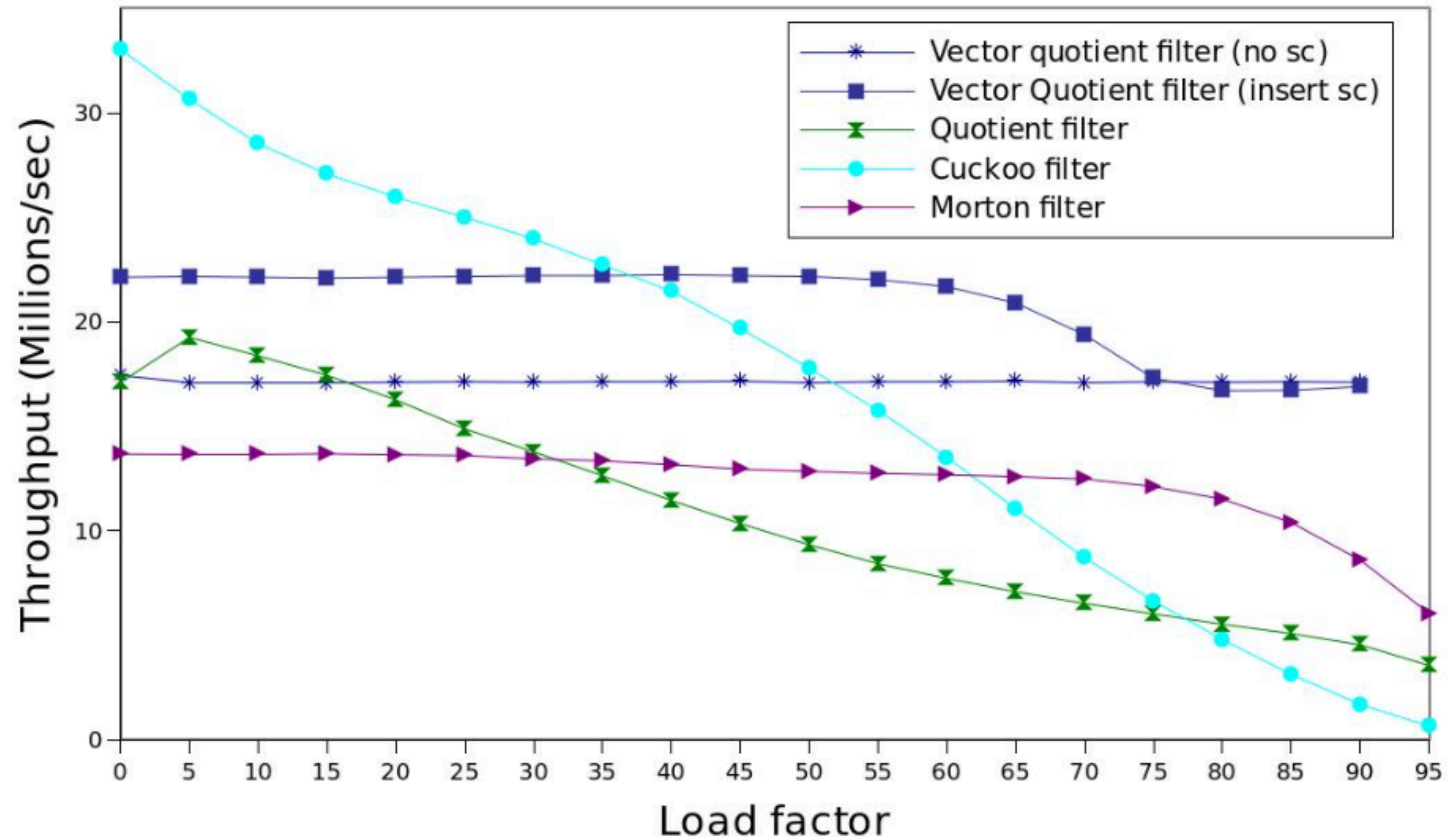
Implemented with PSHUFB or VCMPPB



# 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

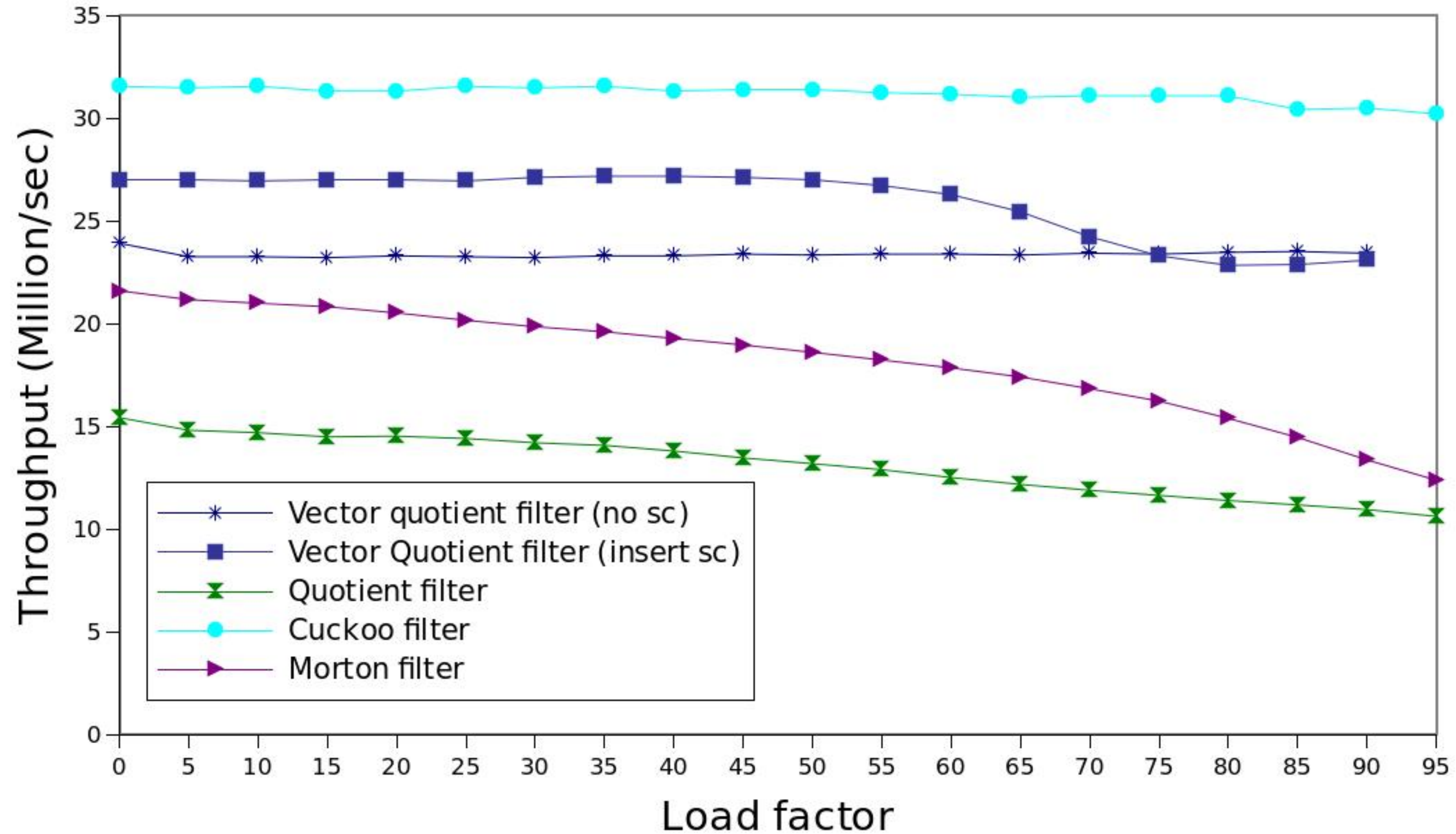
# Evaluation - Insertion



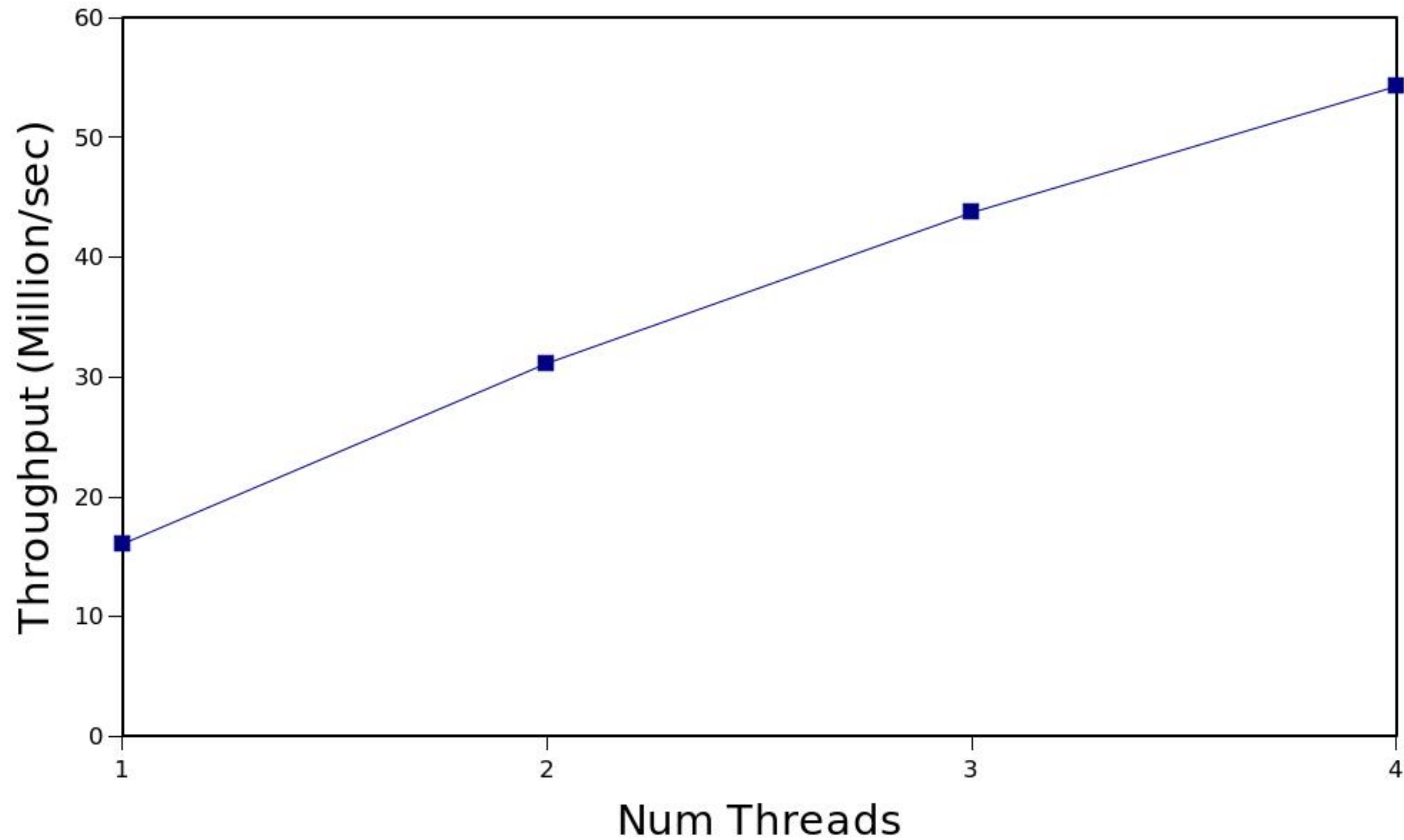
The VQF offers high performance at all load factors



# Evaluation - lookups



# Evaluation - concurrency





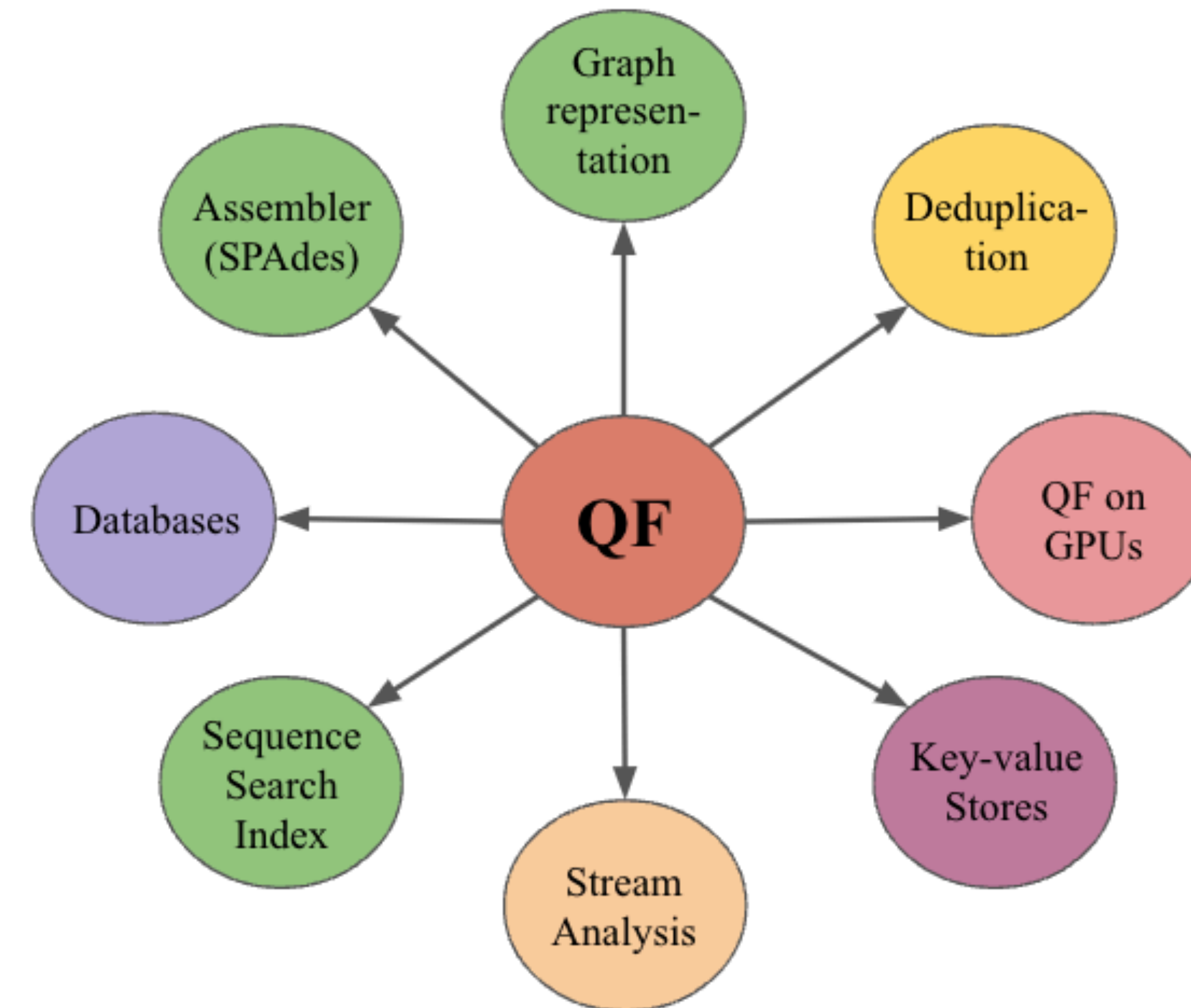
# Quotient filter's impact in computer science

## Computational biology

1. Squeakr
2. deBGR
3. Mantis
4. VariantStore
5. MetaHipMer

## Databases/Systems

1. Anomaly detection
2. BetrFS file system
3. Graph representation



*Theoretically well-founded* data structures can have a *big impact* on multiple subfields across *academia and industry*

**High Performance Filters For GPUs**  
**McCoy, Hofmeyr, Yelick, Pandey -**  
**PPoPP 23, ACDA 23**

# Applications in an exascale world

- High Performance Data Analytics (HPDA) is the intersection of **High Performance Computing (HPC)** and **Big Data**
- HPDA applications run on massive systems like **supercomputers**
- **GPUs** power these supercomputers



#1: Frontier  
9408 nodes, 37,632 GPUs  
1,685.65 PFlop/s Peak



# Metagenomics



Soil sample



Water sample



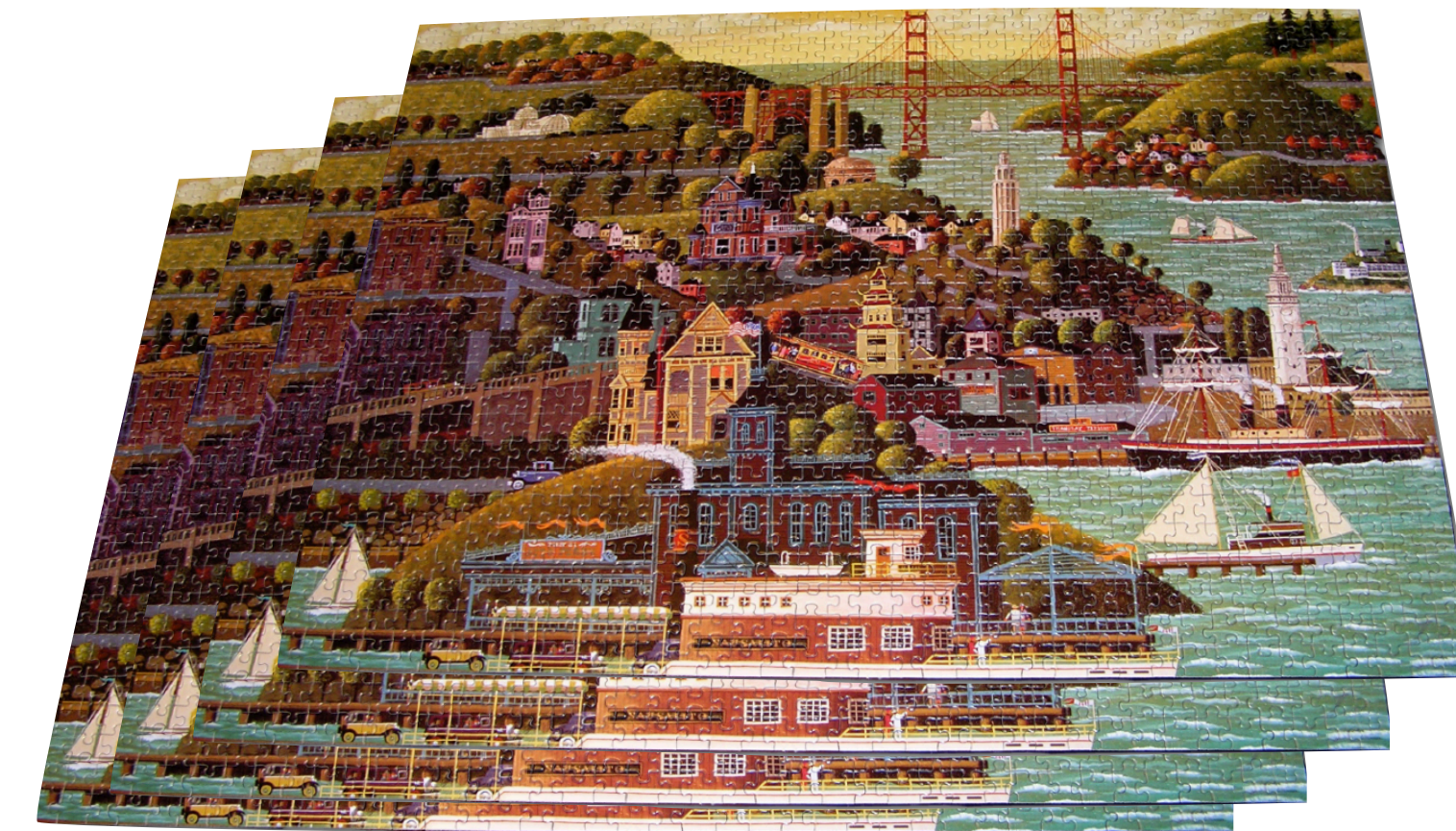
Human gut

Metagenomics is the study of microbes that inhabit an environment and their interactions.



# Metagenomic Assembly

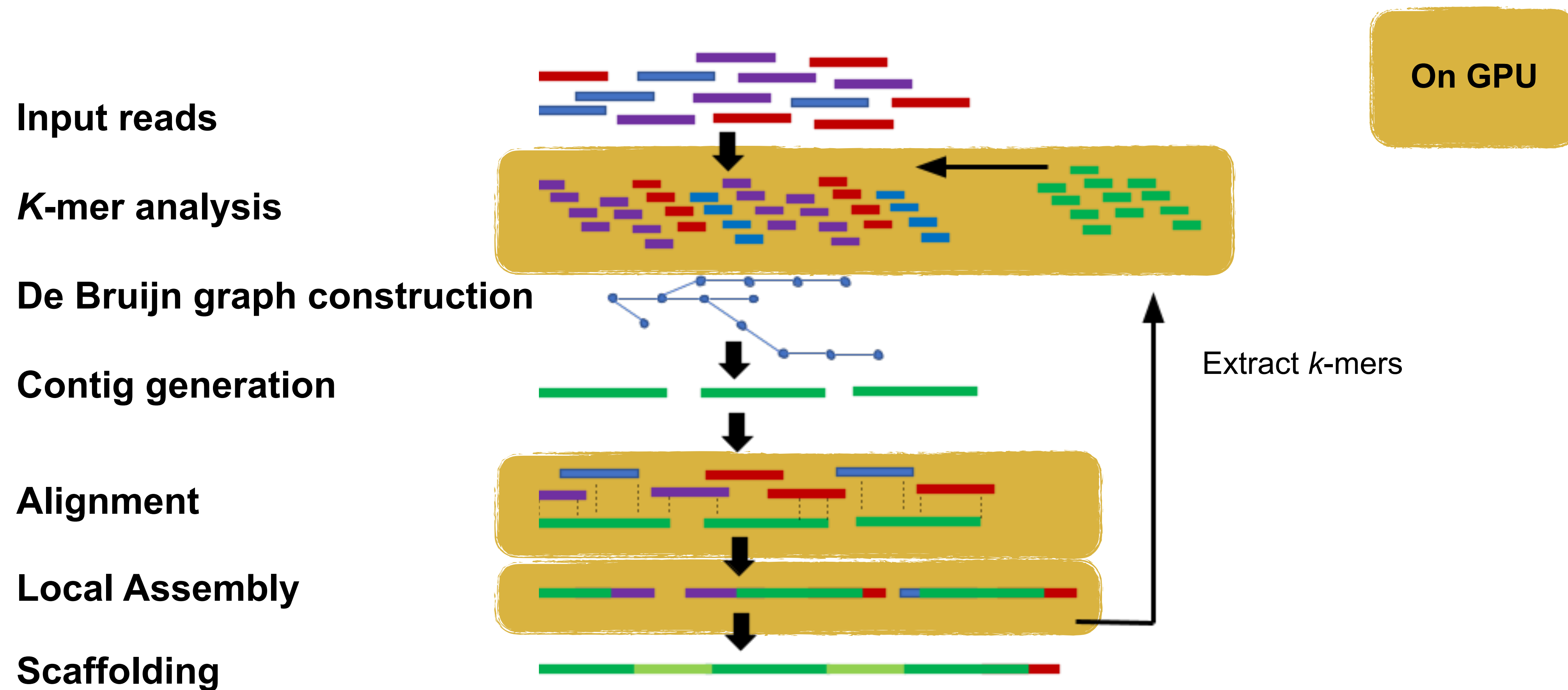
- Sequences are generated as fragments called **reads**
- Rebuilding DNA strands from the reads is compute/memory intensive



It's like building the puzzle without the picture on the box and there are multiple different puzzles in the same box!



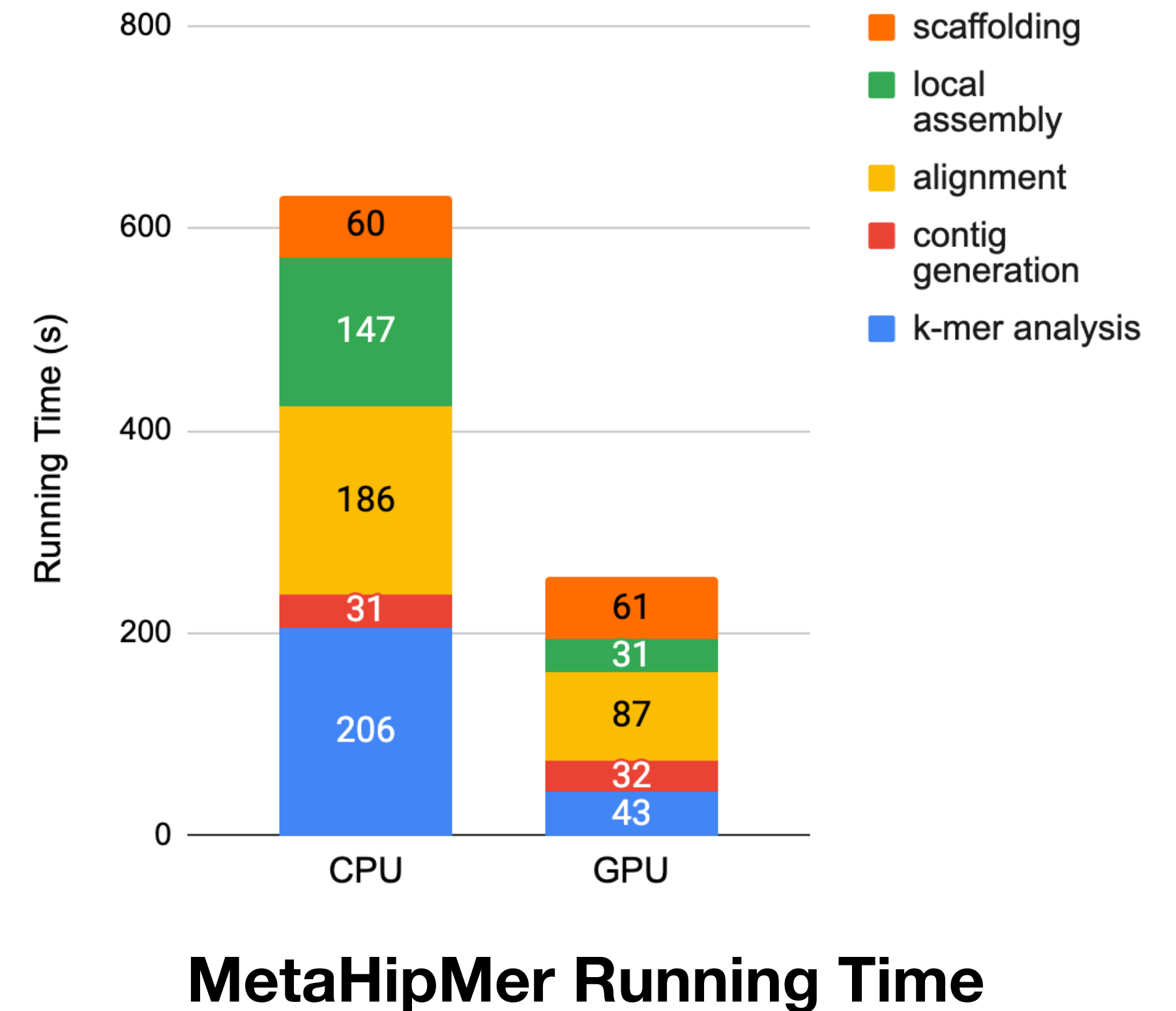
# MetaHipMer: an exascale metagenomic assembler





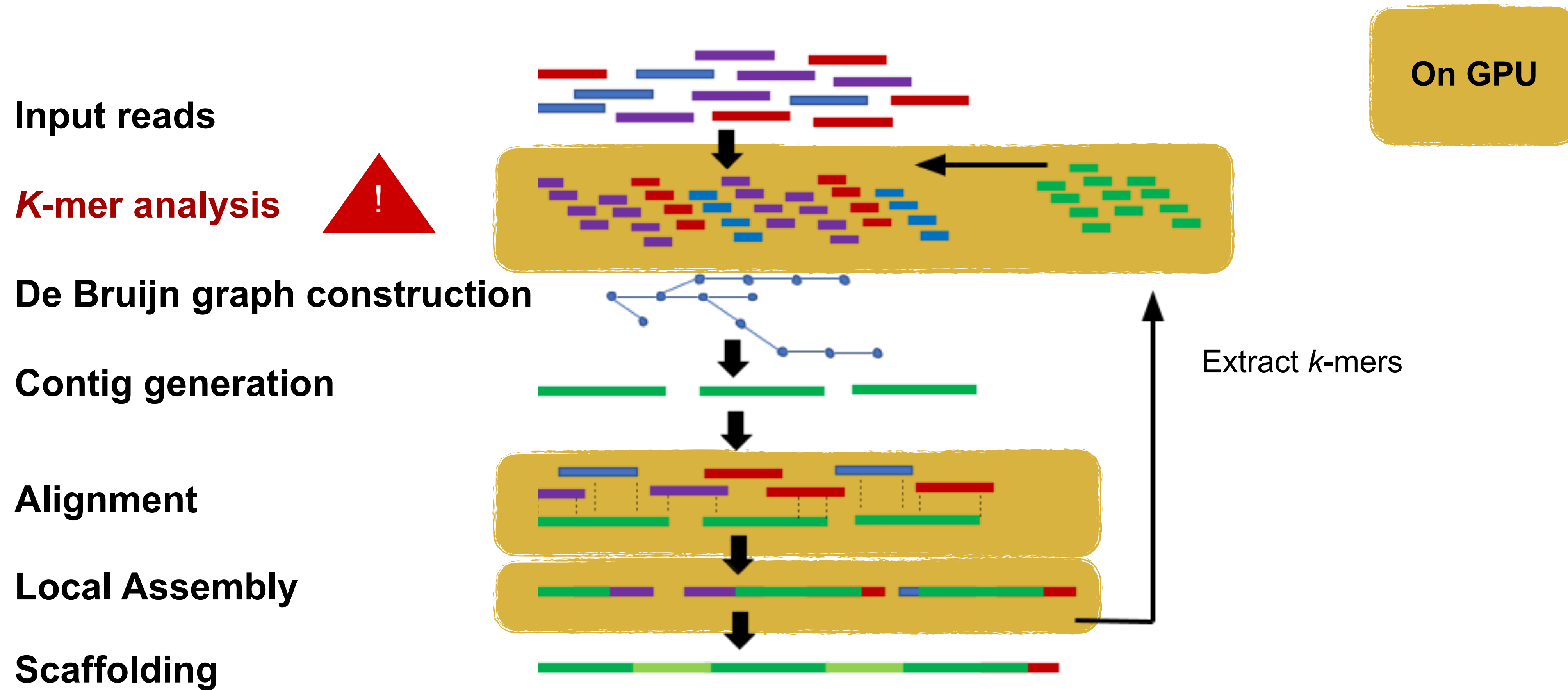
# GPUs accelerate metagenomic assembly

- MHM recently completed the largest co-assembly ever
  - **9,400 nodes** on Frontier
  - **37,000 GPUs**
  - **71.6 terabyte** assembly of Tara Oceans dataset



**105 terabyte Human Microbiome dataset not assembled yet!**

# GPUs are the memory bottleneck



**! Peak memory usage in *k*-mer analysis!**

# Tradeoff in GPU-enabled k-mer analysis

Speed

Memory



Faster compute

Low device memory

# Tradeoff in GPU-enabled k-mer analysis

Speed

Memory



**Filters can help overcome the memory-speed tradeoff in GPUs!**



Faster compute

Low device memory



# K-mer analysis requires filters with:



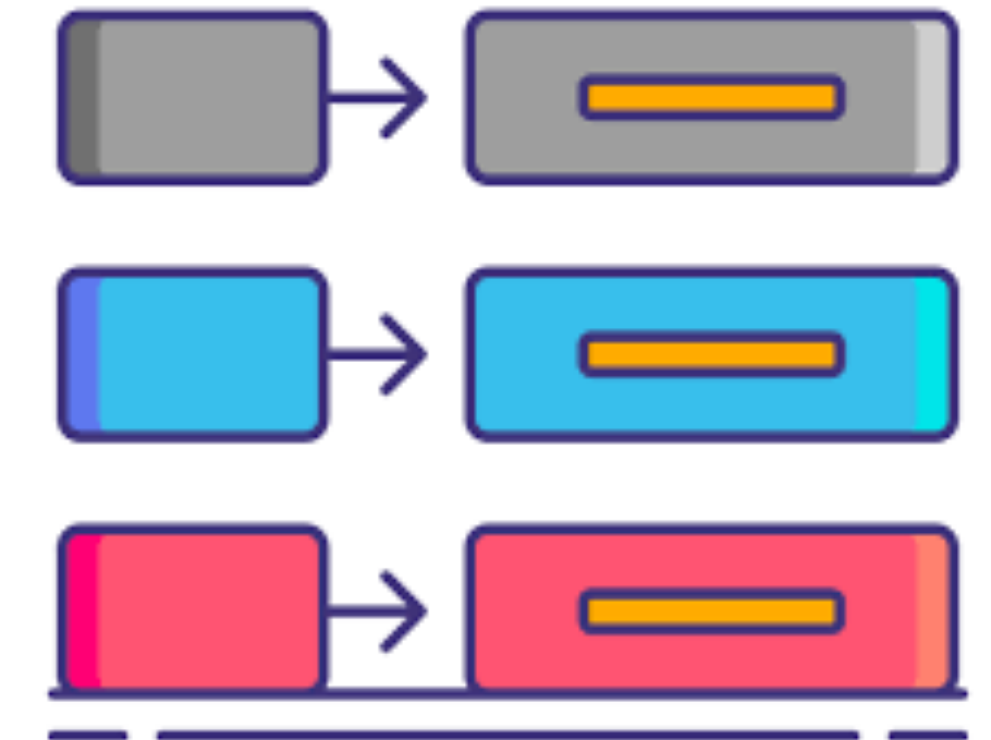
**High performance**



**Space efficiency**



**Deletions**



**Key-value support**

# Existing GPU filters lack critical features

	Inserts	Queries	Deletions	Counting	Key-Value Association
Bloom Filter	✓	✓			
Blocked Bloom Filter <sup>[1]</sup>	✓	✓			
RSQF <sup>[2]</sup>	✓	✓	✓*		✓*
SQF <sup>[2]</sup>	✓	✓	✓		✓*

[1] Junger et al. 2020  
[2] Geil et al. 2018

\* Not supported in implementation, could be supported in theory



# Existing GPU filters lack critical features

	Inserts	Queries	Deletions	Counting	Key-Value Association	Performance
Bloom Filter	✓	✓				
Blocked Bloom Filter <sup>[1]</sup>	✓	✓				✓
RSQF <sup>[2]</sup>	✓	✓	✓*		✓*	
SQF <sup>[2]</sup>	✓	✓	✓		✓*	

[1] Junger et al. 2020  
[2] Geil et al. 2018

\* Not supported in implementation, could be supported in theory

# Existing GPU filters lack critical features

	Inserts	Queries	Deletions	Counting	Key-Value Association	Performance
Bloom Filter	✓	✓				
Blocked Bloom Filter <sup>[1]</sup>	✓	✓				✓
RSQF <sup>[2]</sup>	✓	✓	✓*		✓*	
SQF <sup>[2]</sup>	✓	✓	✓		✓*	

Goal: To build a GPU filter that can achieve high-performance and supports different features (eg counting, values)

[1] Junger et al. 2020  
[2] Geil et al. 2018

\* Not supported in implementation, could be supported in theory

# Proposed solution: TCF and GQF

	Inserts	Queries	Deletions	Counting	Key-Value Association	Performance
Bloom Filter	✓	✓				
Blocked Bloom Filter <sup>[1]</sup>	✓	✓				✓
RSQF <sup>[2]</sup>	✓	✓	✓*		✓*	
SQF <sup>[2]</sup>	✓	✓	✓		✓*	
<b>TCF</b>	✓	✓	✓		✓	✓
<b>GQF</b>	✓	✓	✓	✓	✓	✓

[1] Junger et al. 2020  
 [2] Geil et al. 2018

\* Not supported in implementation, could be supported in theory

# Results

- Present new GPU filter designs:
  - Two-Choice Filter (TCF)
    - Stable filter with key-value association/deletion
  - GPU Quotient Filter (GQF)
    - Filter with key-value association/deletion/dynamic counters
- Up to **4.4x faster** than previous GPU filters
- Thread-level point API and host bulk API for easy integration
- **43% reduction in overall peak memory** usage in MetaHipMer

# Results

- Present new GPU filter designs:
  - **Two-Choice Filter (TCF)**
    - Stable filter with key-value association/deletion
  - GPU Quotient Filter (GQF)
    - Filter with key-value association/deletion/dynamic counters
- Up to **4.4x faster** than previous GPU filters
- Thread-level point API and host bulk API for easy integration
- **43% reduction in overall peak memory** usage in MetaHipMer



# GPU Challenges

## 1. Thread divergence

- Warps diverge and slow down if threads perform different operations

## 2. Memory coherence

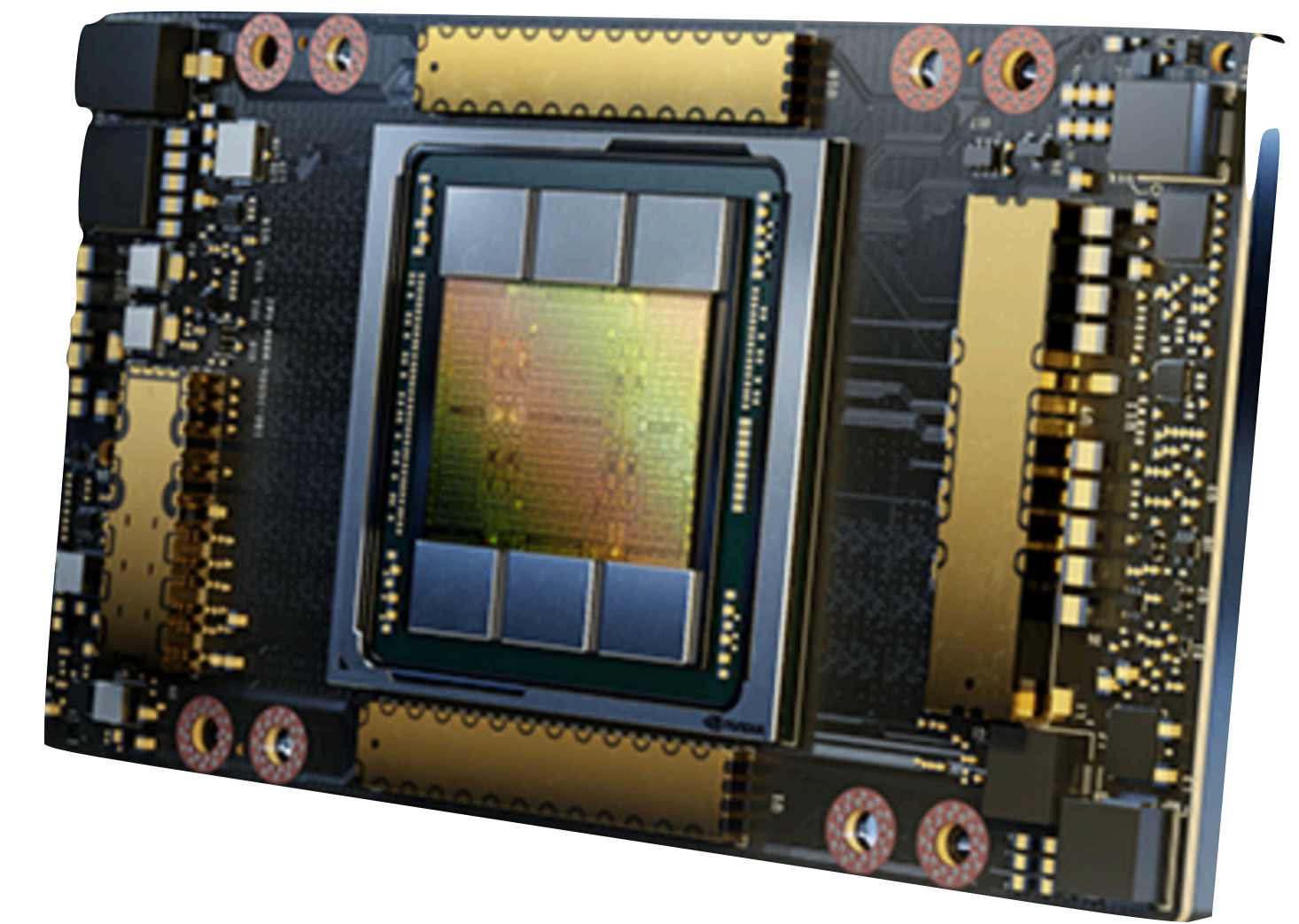
- Warps slow down if threads read from different cache lines

## 3. Limited memory

- 80 GB vs 1 TB - GPU memory can't fall back to disk

## 4. Massive parallelism

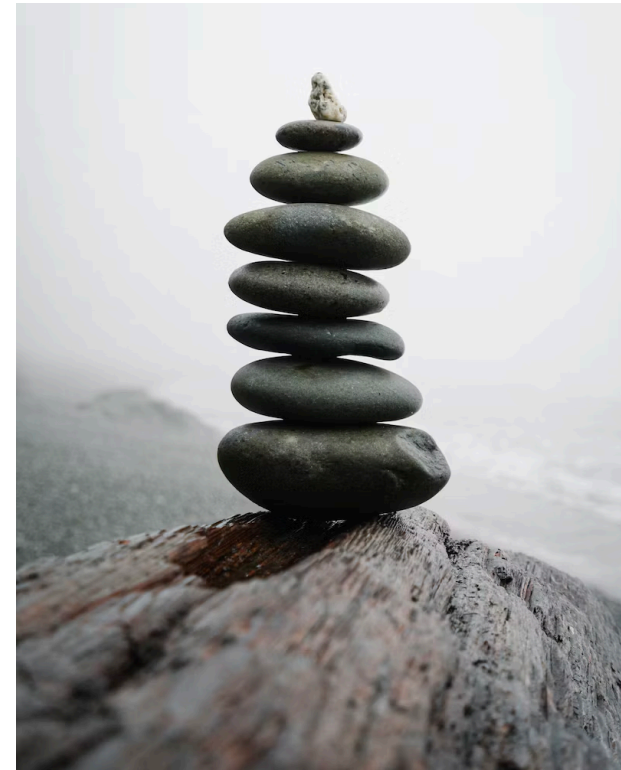
- ~80,000-160,000 simultaneous threads



Nvidia A100 Tensor GPU

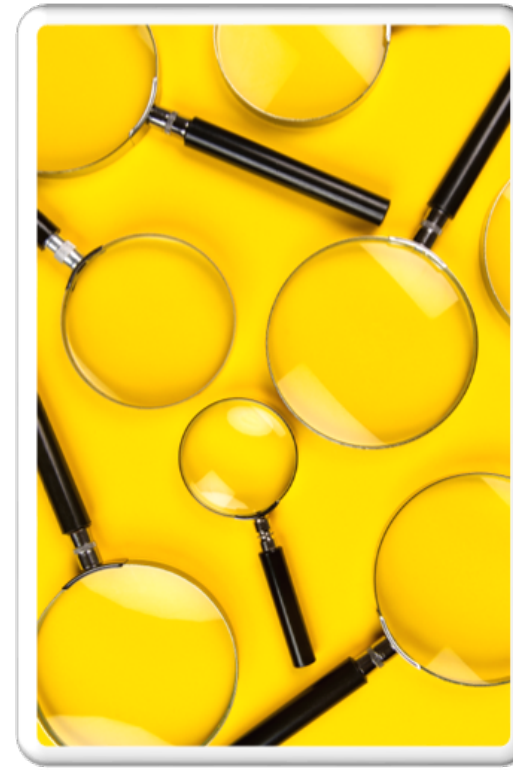


# Design Goals for GPU Filters



## Stability

Items don't move after insertion



## Low associativity

Map each item to one or a small number of locations



## Space efficiency

Minimum overhead from pointers or over provisioning

# Mapping GPU challenges to filter design goals

## Filter design goal

## GPU challenge

Low associativity



Thread divergence and memory coherence

Stability



High degree of parallelism

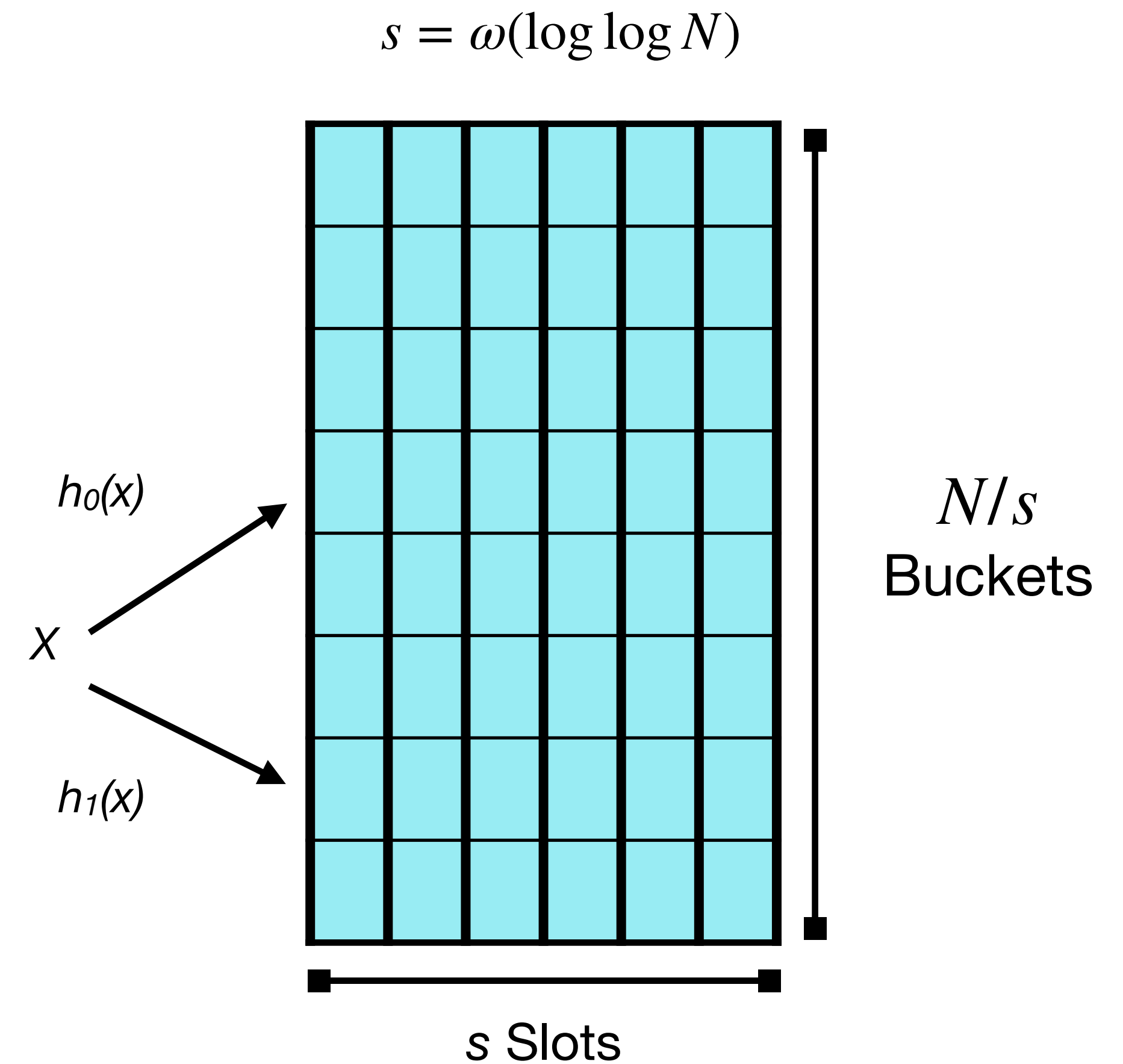
Space efficiency



Limited memory

# Two-choice filter on GPUs

- $s = \omega(\log \log N) \approx 48$ 
  - No drops up to 90% load
  - Strategy used by VQF
- **Slow on GPUs** — too many slots to probe with 1 warp
- **Not stable** — tags move inside buckets
- Can increase throughput by setting  $s$  to a smaller value
  - However, can't reach high **space efficiency**



# Choosing the optimal bucket size

Can we efficiently use warps with bucket sizes less than 32?



# Choosing the optimal bucket size

Can we efficiently use warps with bucket sizes less than 32?

Yes, with Cooperative Groups

# With small bucket sizes, warps may not be fully utilized

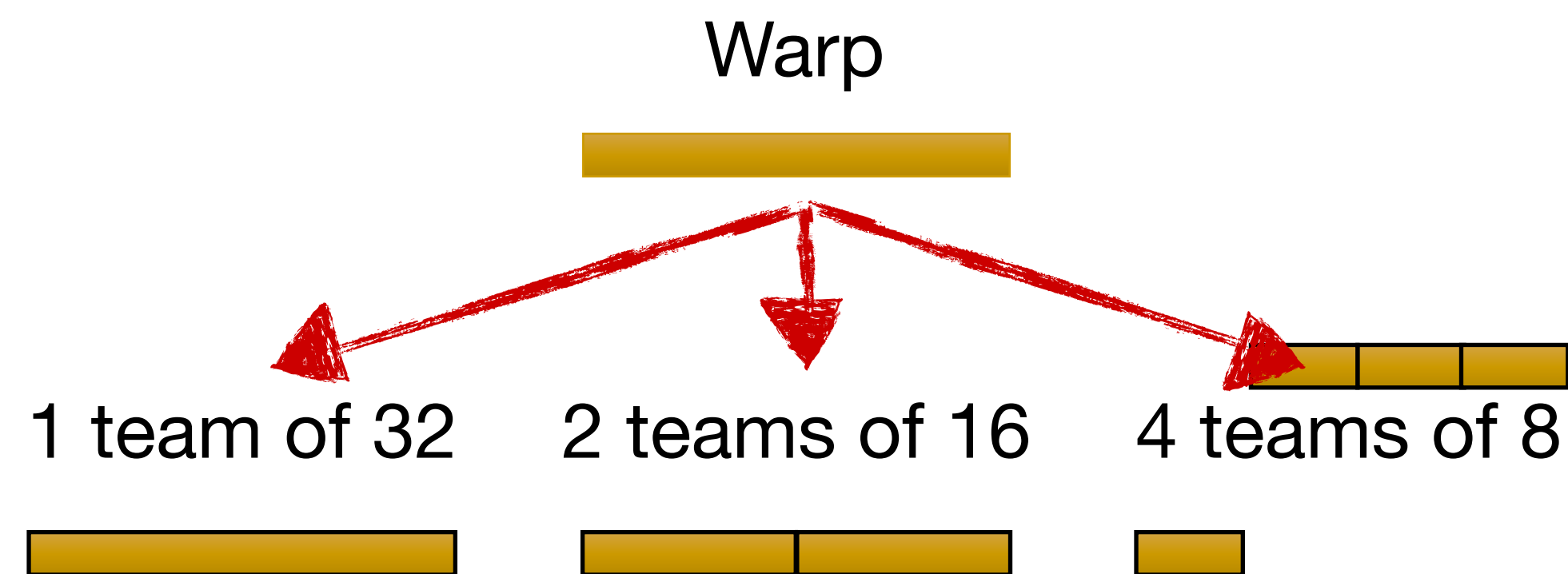
The cooperative groups API lets us split warps into smaller teams called

## Cooperative Groups

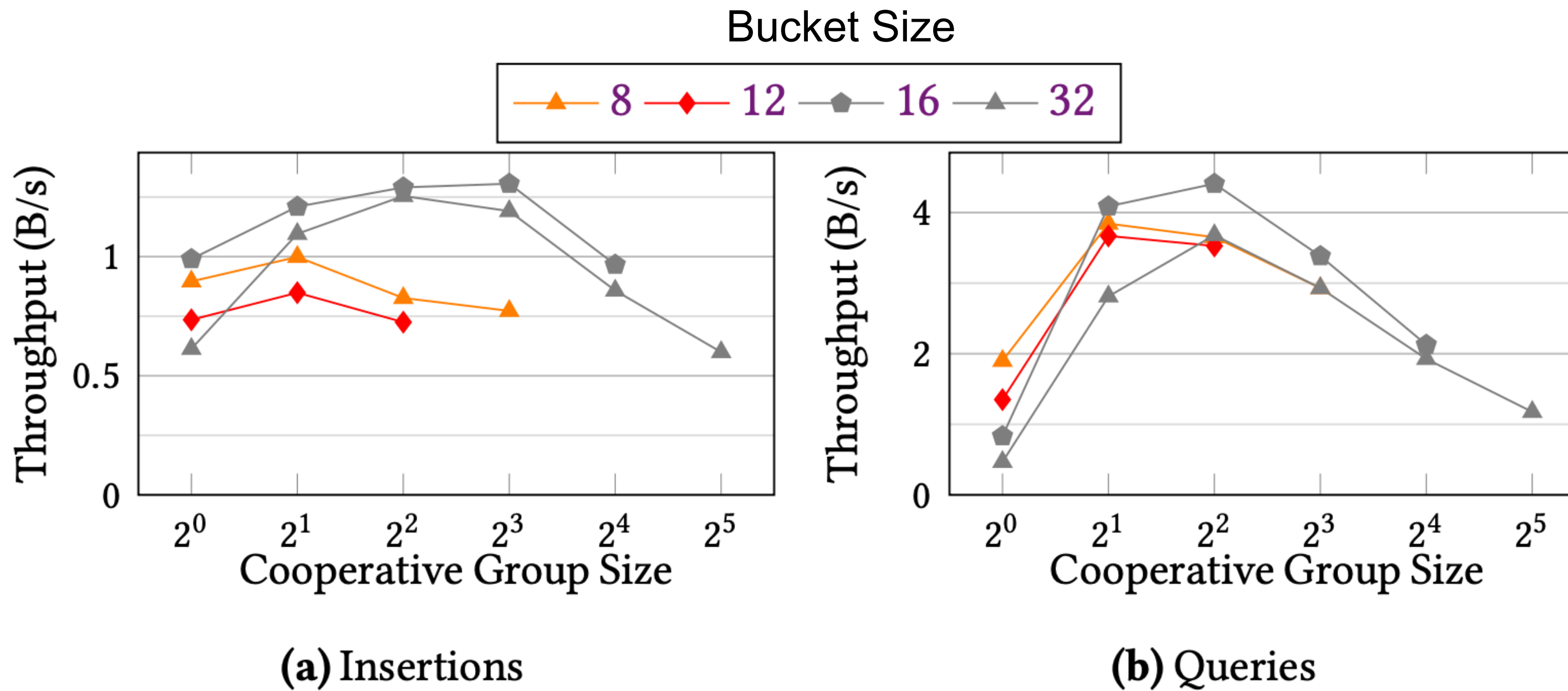
This is a logical partition: underlying hardware has not changed

Cooperative groups let us **trade computation for memory:**

- Less compute per group, but we can amortize cost of loading buckets

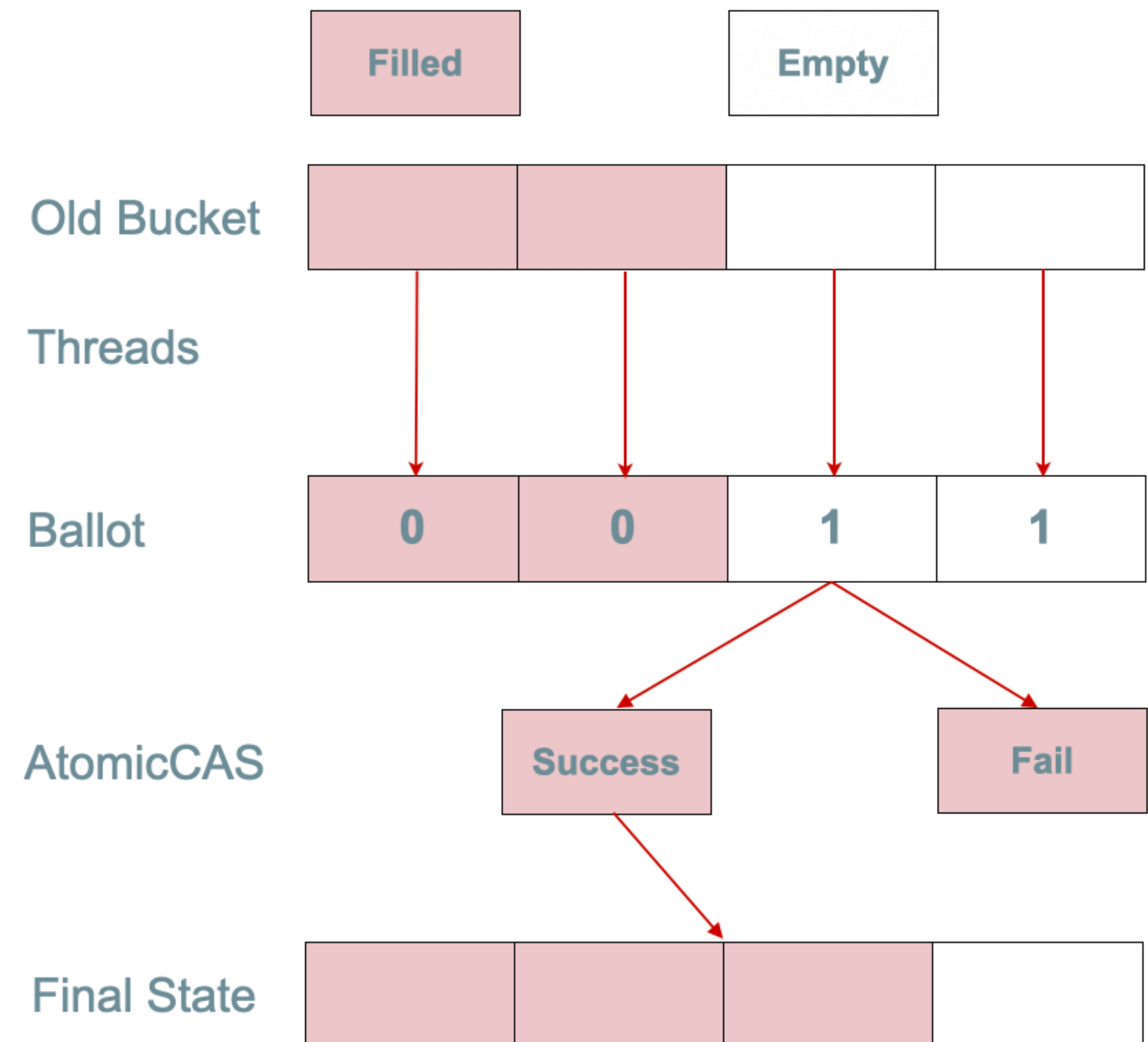


# Evaluation - Optimal bucket size



# Buckets are modified atomically

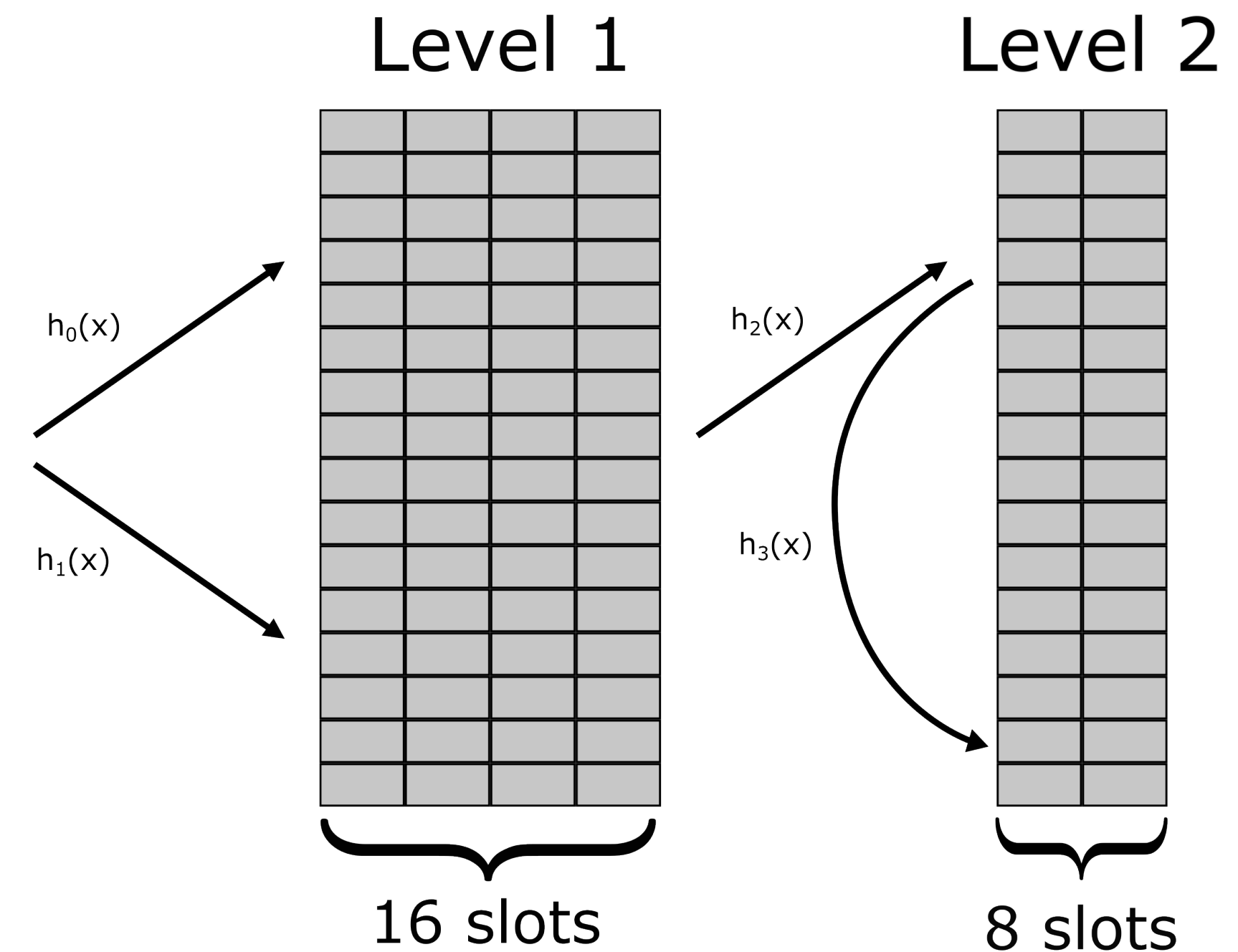
- **CUDA coherence is weak** - no guarantee that changes will be observed in other blocks without thread fencing / atomics
- Cache old state - verify with atomicCAS
- All insertions done atomically, all queries done lazily





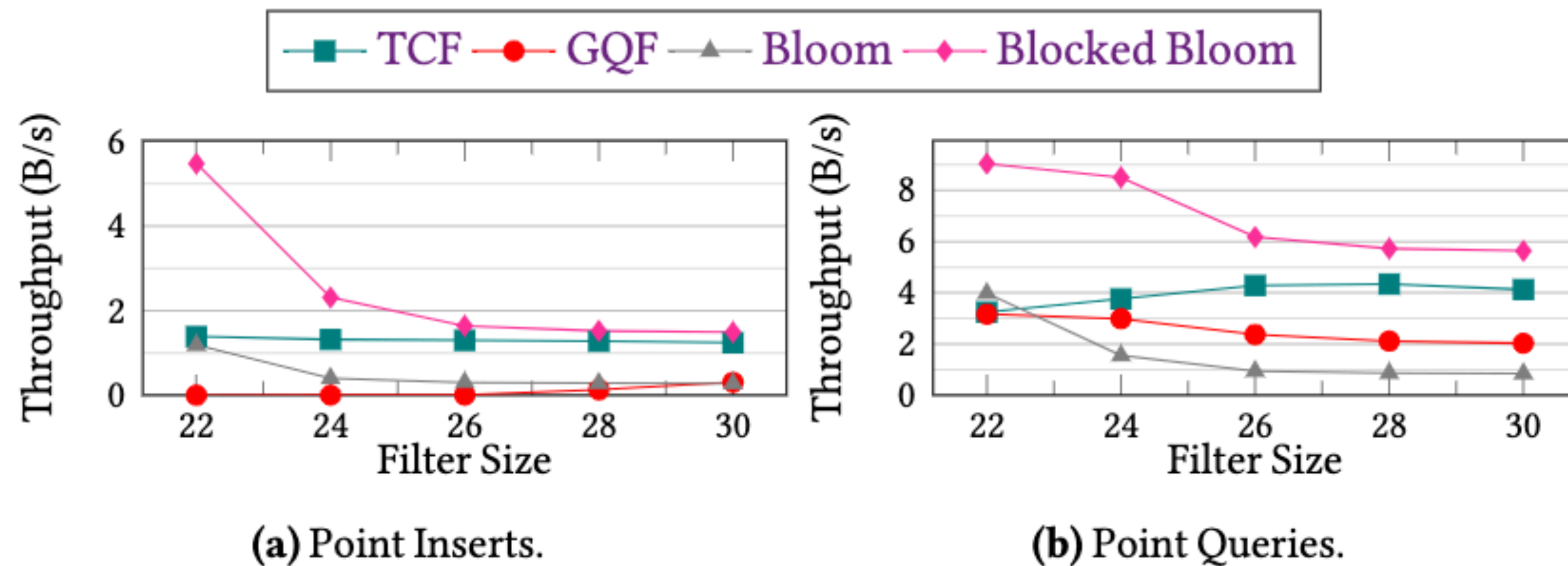
# Frontyard-backyard hashing

- Bucket size is chosen to be 16
- Items drop around 70% load
- Small backing table catches drops, allows scaling to 90% load
- Backing table is ~1-2% of the total filter size.
- Uses linear probing to traverse buckets

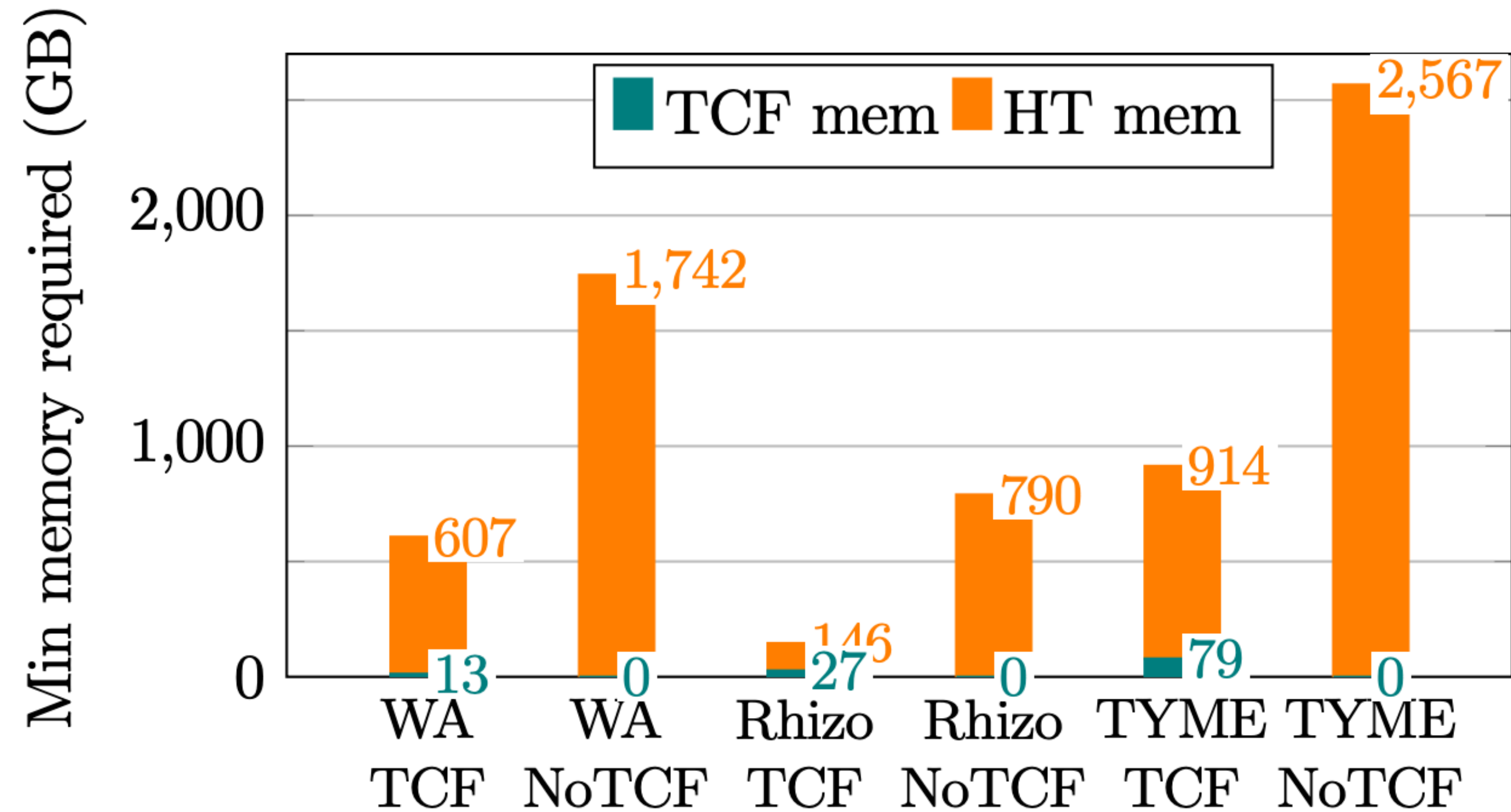


# Results

	BF	Blocked BF	SQF	RSQF	TCF	GQF
<b>False Positive (%)</b>	0.15	0.71	1.17	1.55	0.024	0.19
<b>Bits Per Item</b>	10.10	9.73	9.7	7.87	16	10.68



# Aggregate savings



Peak memory use in  $k$ -mer analysis is reduced by 2.8 - 5.4x!

**This results in a 43% reduction in peak memory use in the assembly pipeline**

# Takeaways

- The two-choice filter overcomes the feature-performance tradeoff of previous GPU Filters
- Simple design with strong theoretical foundation results in practical data structures
- Using a GPU filter can **vastly reduce memory use** of  $k$ -mer analysis
  - No measured decrease in assembly quality
  - No measured increase in overall runtime
- Filters with advanced features **simplify** the pipeline

Github and lab page:

Libraries: <https://github.com/saltsystemslab/gpu-filters>

UtahDB: <http://mod.cs.utah.edu/>

