# Data Systems at Scale:
## Scaling Up by Scaling Down and Out

Prashant Pandey
VMware Research

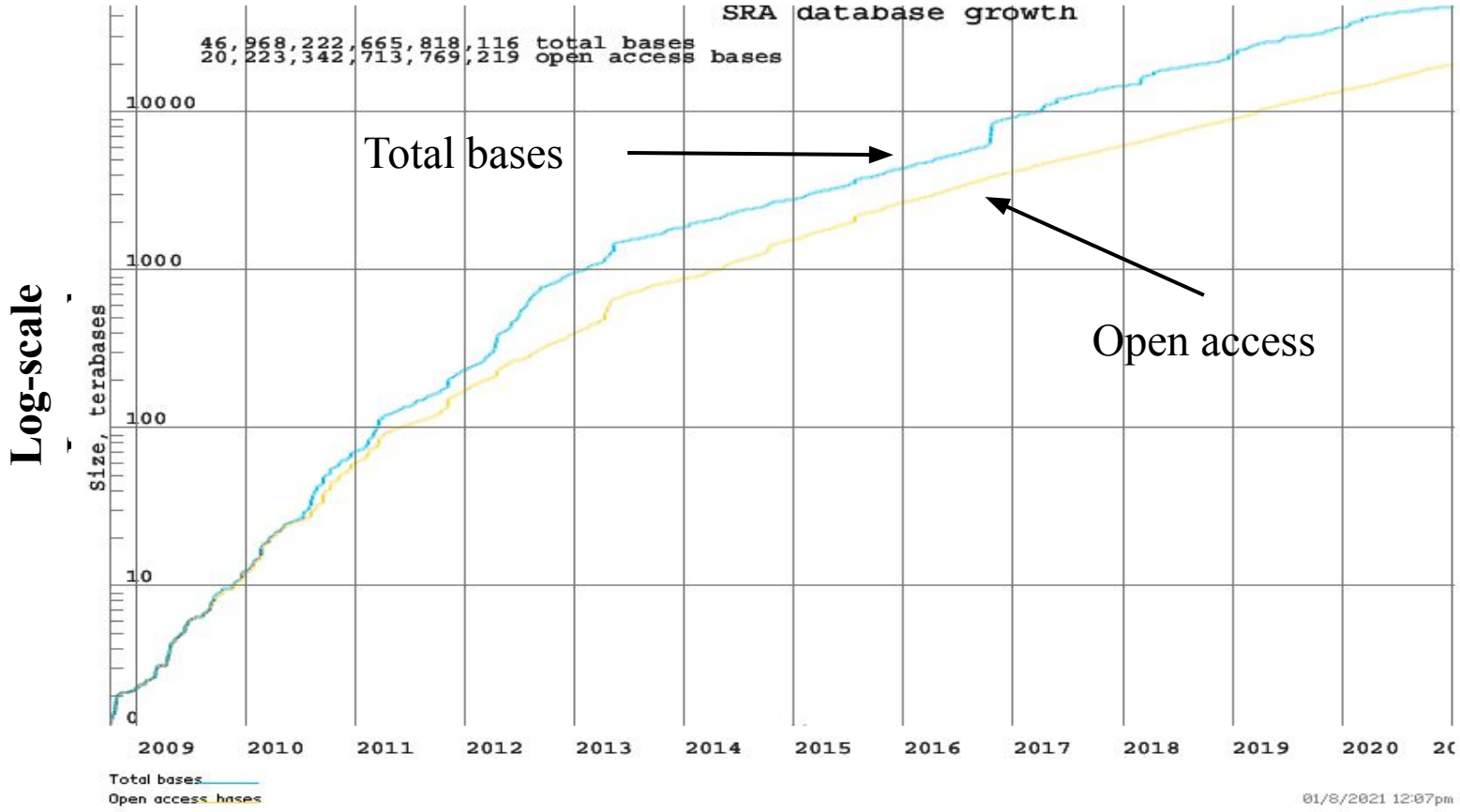
# Scalability challenge in a tweet


Professor Bioinformatics and Computational Biology
The University of Edinburgh

Associate Professor Computational Biology
Johns Hopkins University

# Sequence Read Archive (SRA) growth

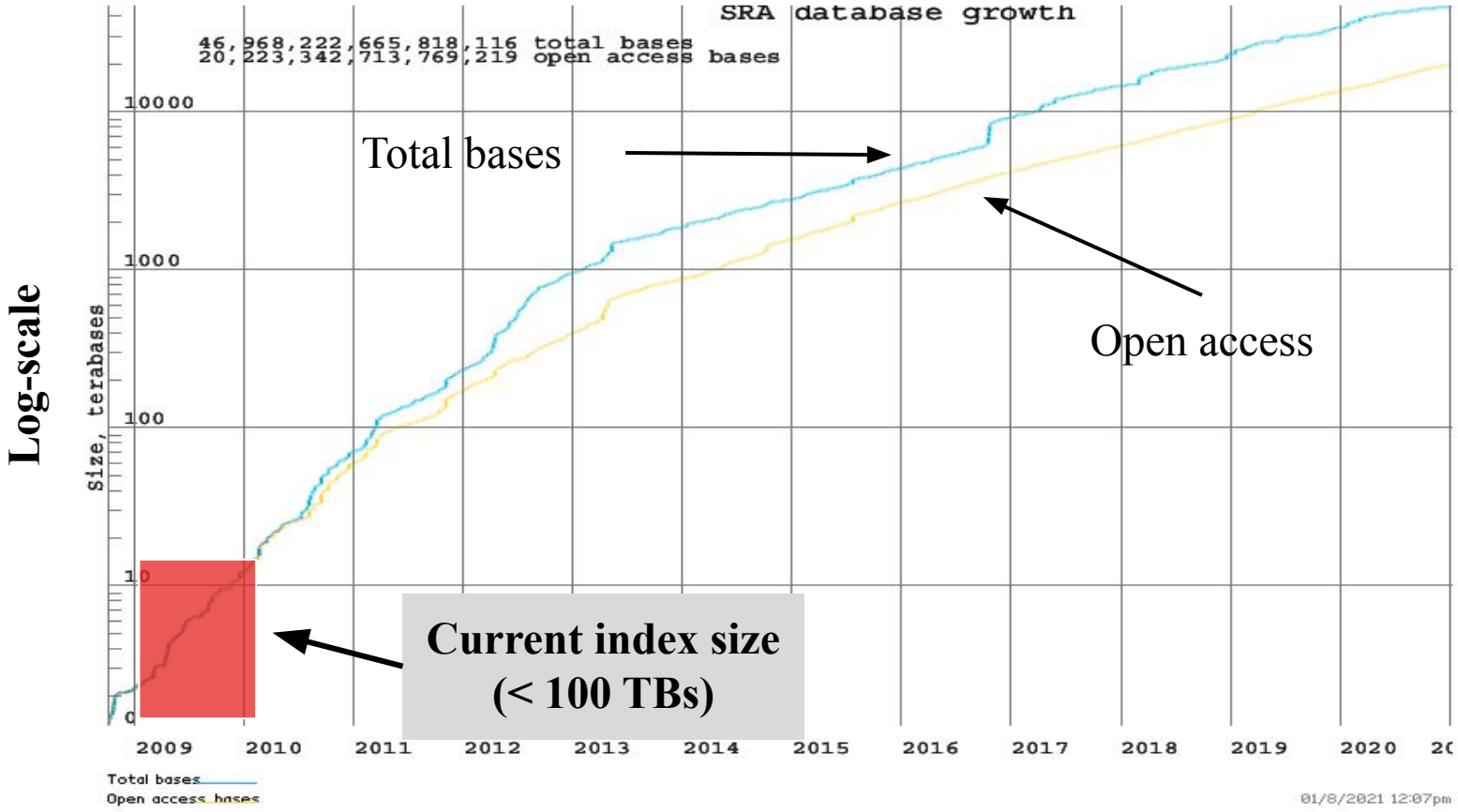
SRA contains a lot of ***diversity information***




**Q**: What if I find e.g., a new disease-related gene, and want to see if it appeared in other experiments?

# Scalability is the bottleneck for data science

SRA contains a lot of ***diversity information***




This renders what is otherwise an immensely valuable public resource ***largely inert***

# Scalability is a ubiquitous challenge

Cyber monitoring

Internet of Things (IoT)

Financial tech

Social networks

AstroPhysics

Chemistry

Environmental science

- People generate 2.5 quintillion bytes of data each day. (IBM, 2016)
- More than 150 zettabytes (150 trillion gigabytes) of data will need analysis by 2025. (Forbes, 2019)
- 90 percent of the world's data was created between 2015 and 2016 alone. (IBM, 2016)

https://learn.g2.com/big-data-statistics

- 
- 
- 
- 
- 

**24. 88% of data is ignored by companies.**

https://leftronic.com/big-data-statistics/

(Forrester Research)

A widely-quoted figure from a 2012 paper from Forrester Research says that, on average, companies analyze only 12% of the available data. Reasons for this include a lack of analytics tools, repressive data silos, and the difficulty in knowing which information is valuable and which is worth leaving.

# Three approaches to handle massive data

# Three approaches to handle massive data

## Shrink it

**Goal**: make data smaller to fit in RAM

**Techniques**:

- Compact & succinct data structures
- Filters, e.g., Bloom, quotient, etc.

# Three approaches to handle massive data

## Shrink it

**Goal**: make data smaller to fit in RAM

**Techniques**:

- Compact & succinct data structures
- Filters, e.g., Bloom, quotient, etc.

## Organize it

**Goal**: organize data in a disk-friendly way

**Techniques**:

- B-tree
- $B^{\varepsilon}$-tree
- LSM-tree

# Three approaches to handle massive data

## Shrink it

**Goal**: make data smaller to fit in RAM

**Techniques**:

- Compact & succinct data structures
- Filters, e.g., Bloom, quotient, etc.

## Organize it

**Goal**: organize data in a disk-friendly way

**Techniques**:

- B-tree
- $B^{\varepsilon}$-tree
- LSM-tree

## Distribute it

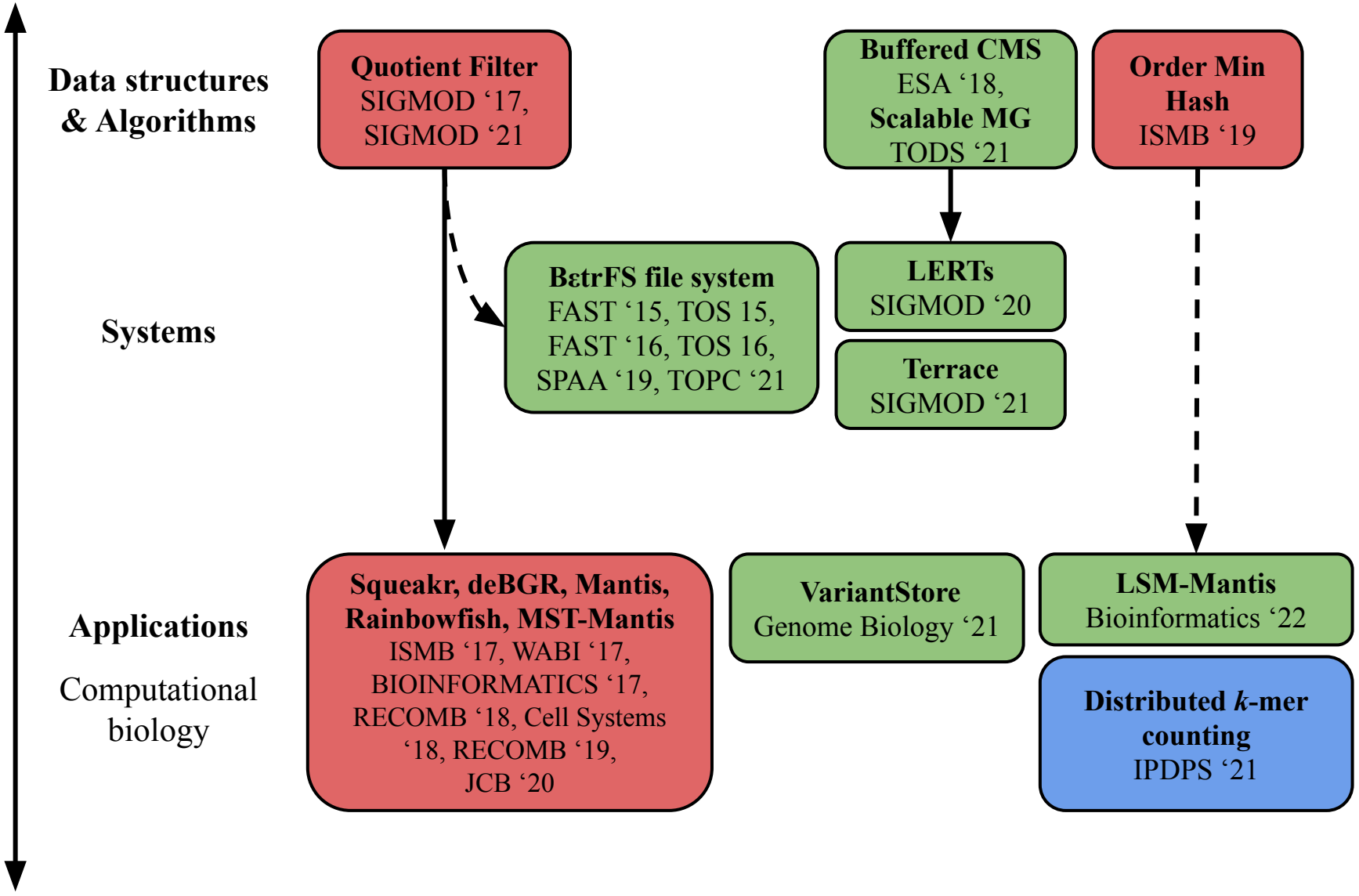
**Goal**: partition and distribute data on multiple nodes

**Techniques**:

- Distributed hash table
- Distributed key-value store

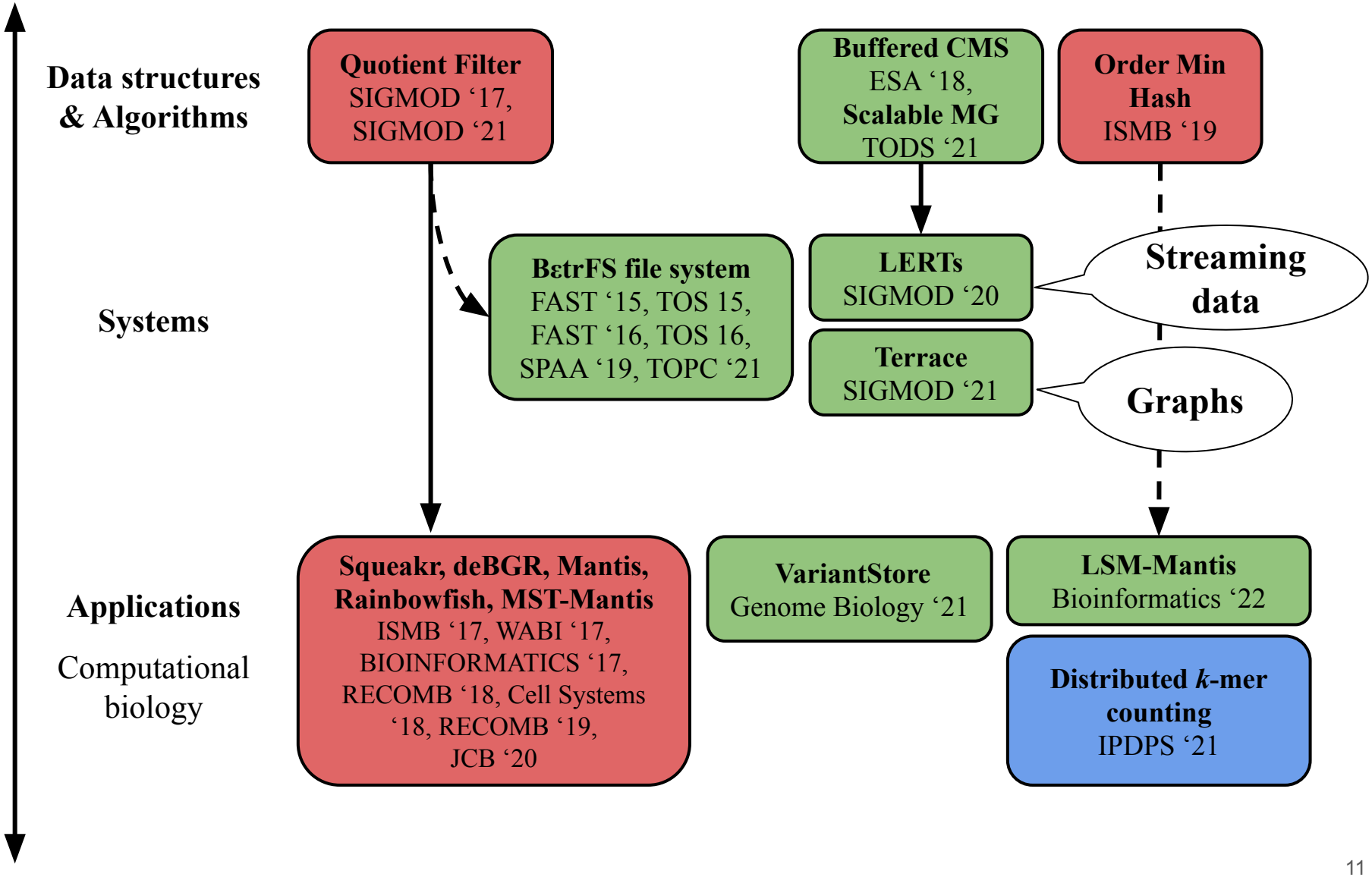
# Research output

Shrink | Organize | Distribute



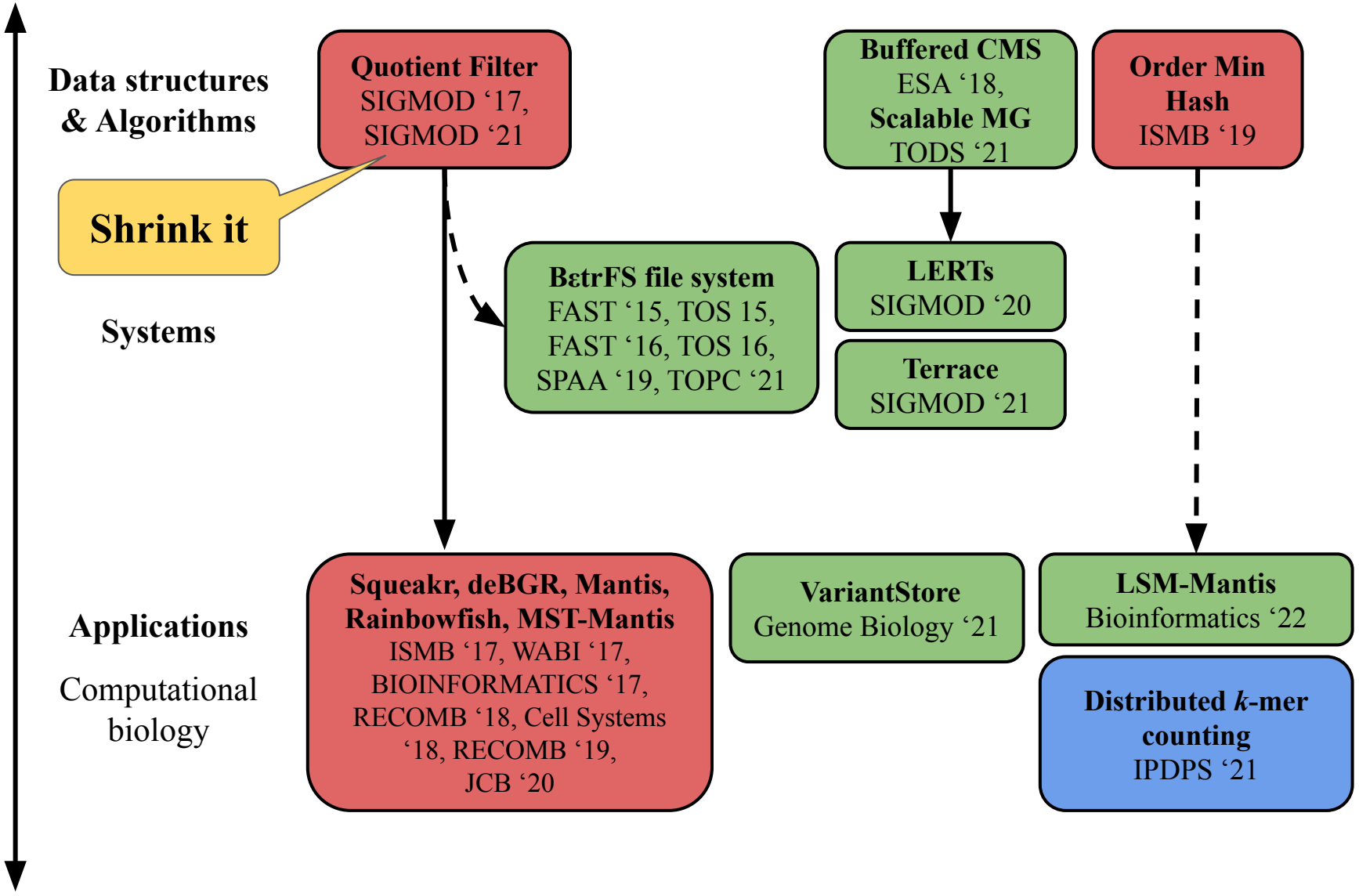
# In this talk:

Shrink | Organize | Distribute



# In this talk:

Shrink | Organize | Distribute



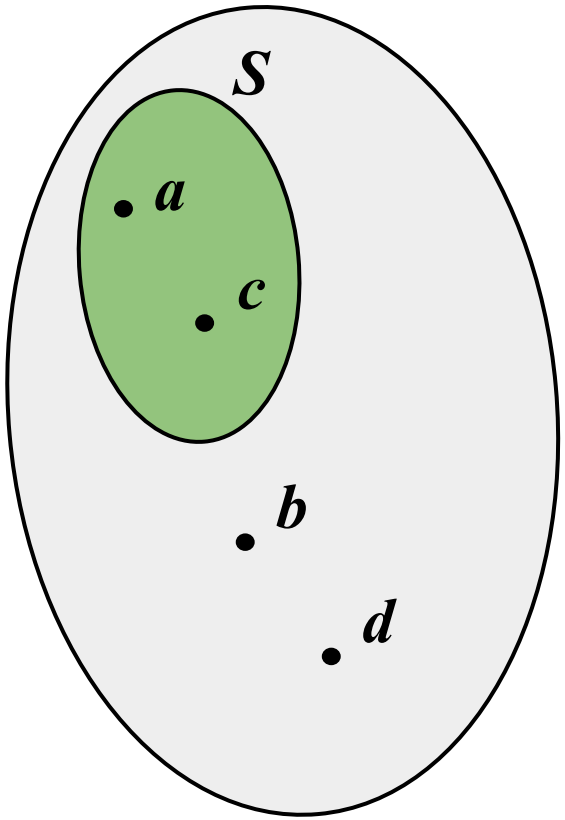
# Filters

~~Filters~~

Time to change your filter

# Dictionary data structure

A dictionary maintains a set $S$ from universe $U$.




membership($a$): ✓

membership($b$): ✗

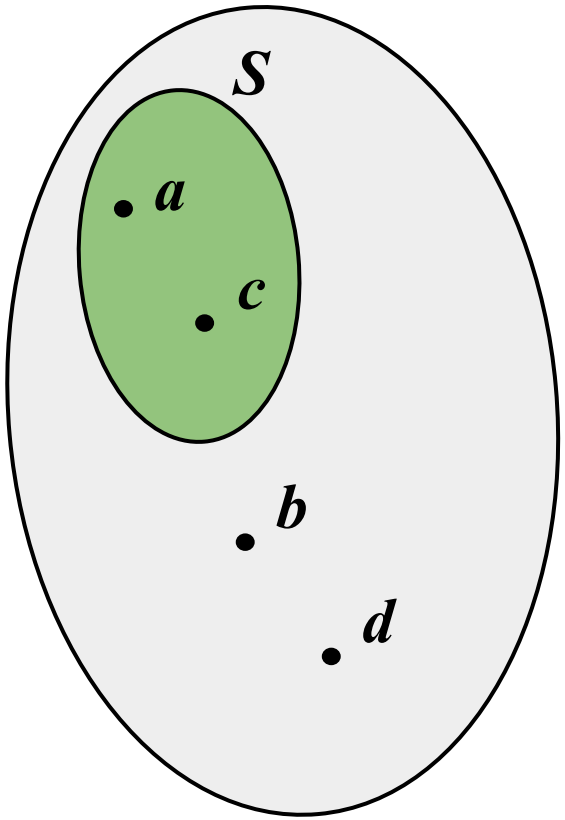
membership($c$): ✓

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**

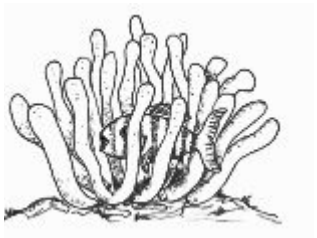
if $q \notin$ S, return
- ✗ with probability $\geq 1 - \varepsilon$ — **true negative**
- ✓ with probability $\leq \varepsilon$ — **false positive**

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**

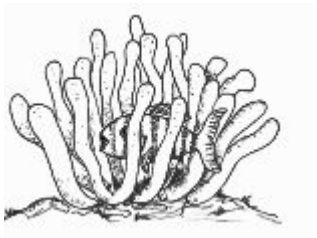
# False-positive rate enables filters to be compact

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

Small

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

Large



**Filter**

**Dictionary**

**For most practical purposes:**

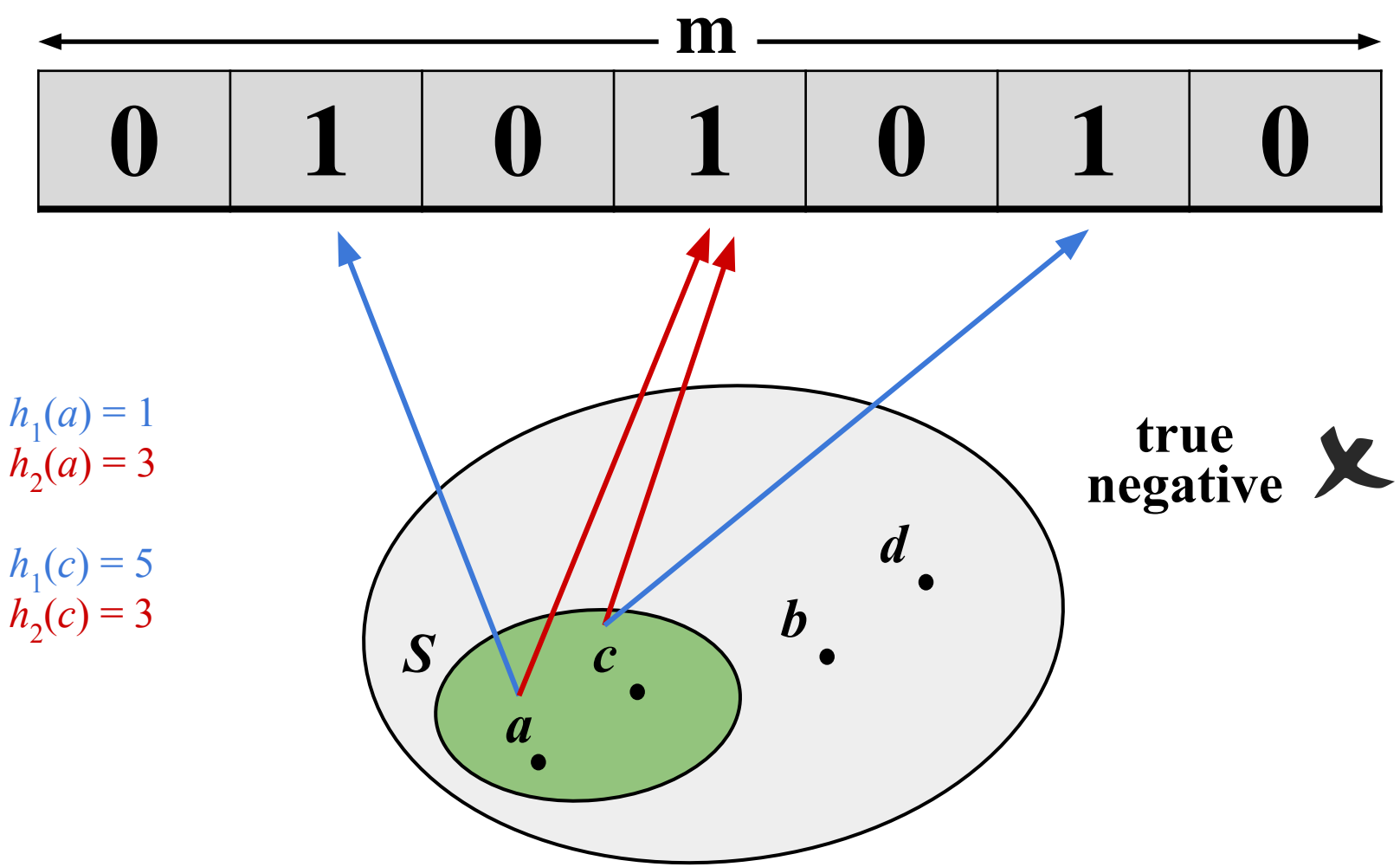
**ε = 2%, a filter requires ≈ 8 bits/item**

# Filter design objectives

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

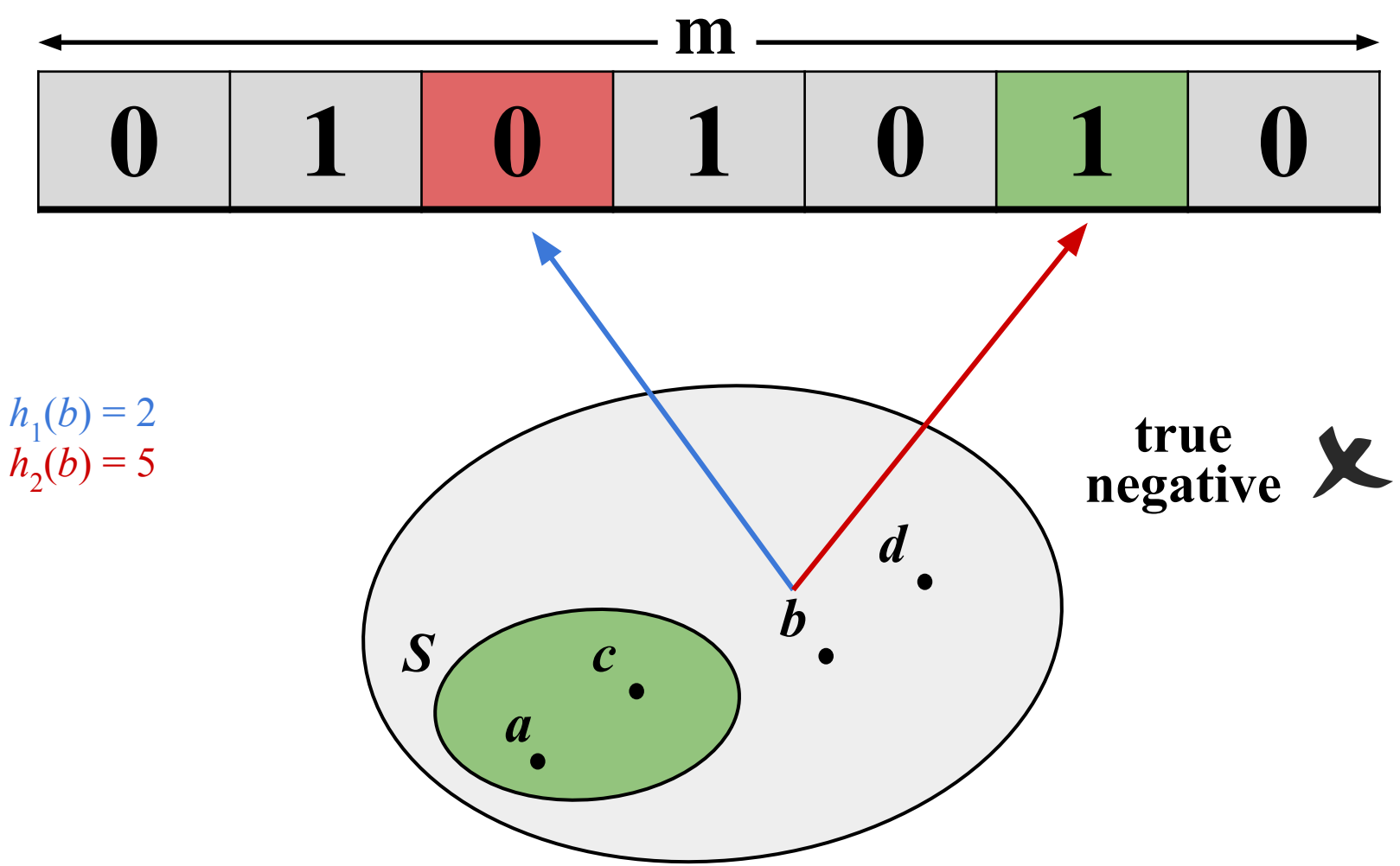
# Classic filter: The Bloom filter [Bloom '70]

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



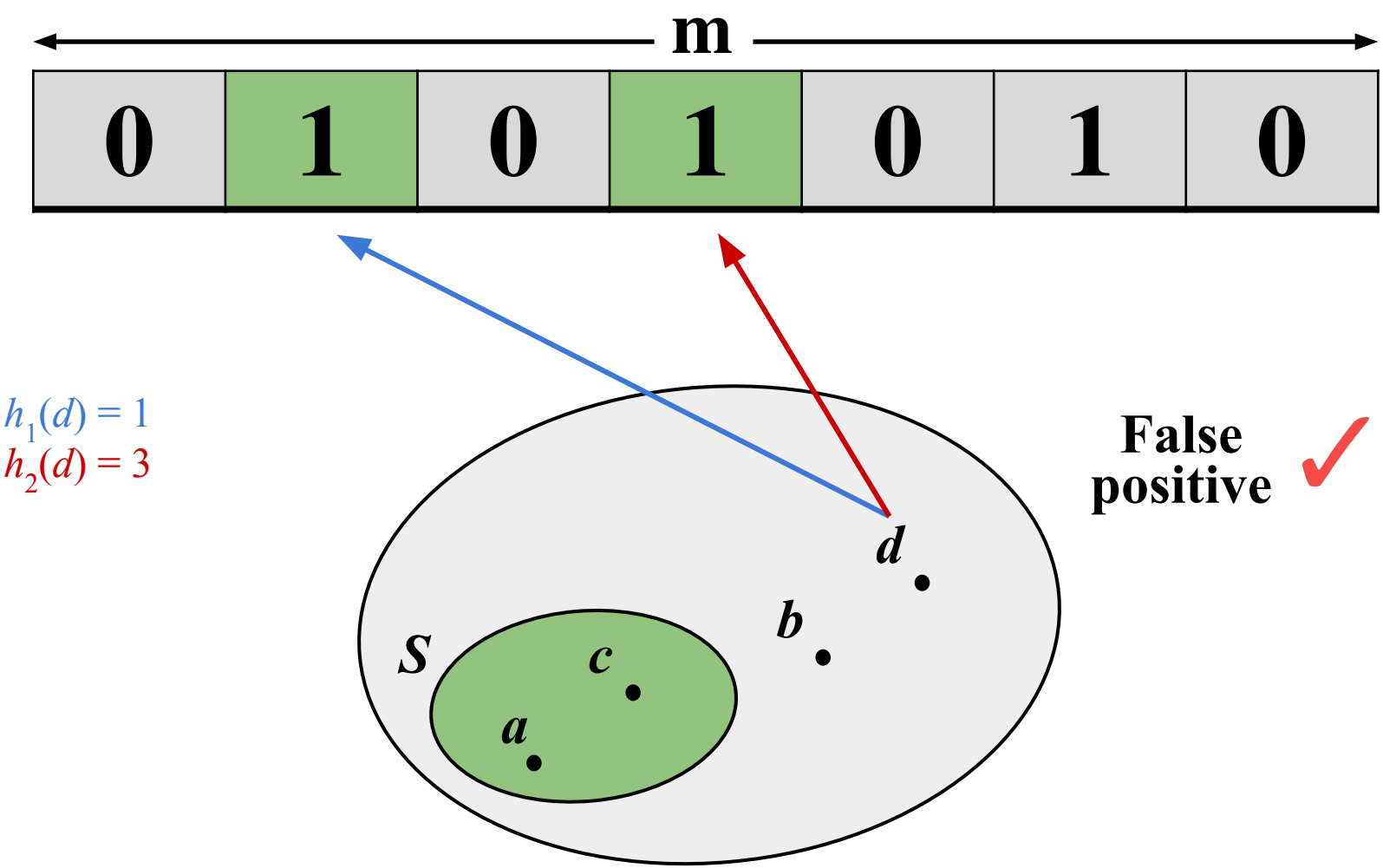
# Classic filter: The Bloom filter [Bloom '70]

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

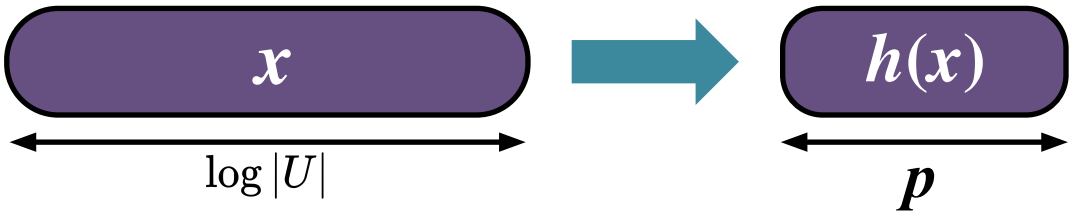
| Limitations | Workarounds |
|---|---|
| No deletes | Rebuild |
| No resizes | Guess *N*, and rebuild if wrong |
| No filter merging or enumeration | ??? |
| No values associated with keys | Combine with another data structure |

**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]

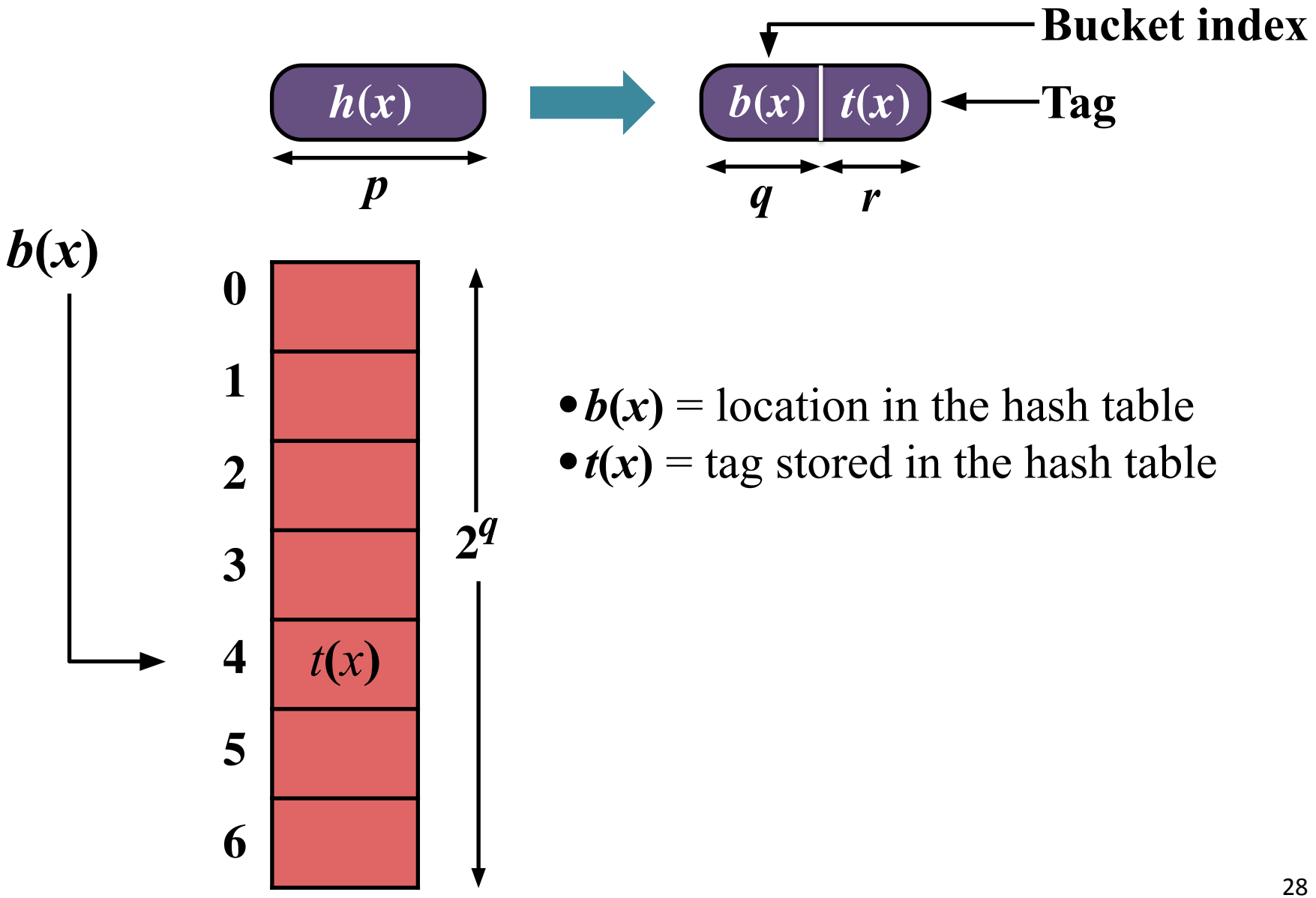
- **Store fingerprints compactly in a hash table.**
  - Take a fingerprint $h(x)$ for each element $x$.




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

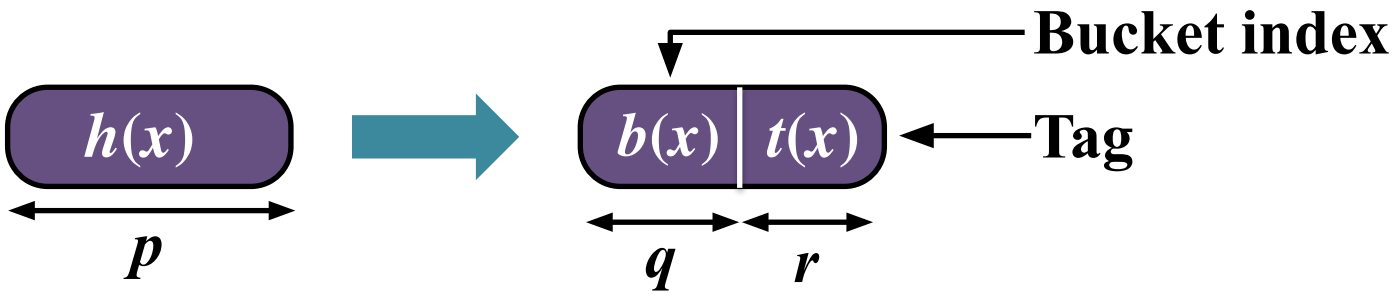
$$\Pr[x \text{ and } y \text{ collide}] = \frac{1}{2^p}$$

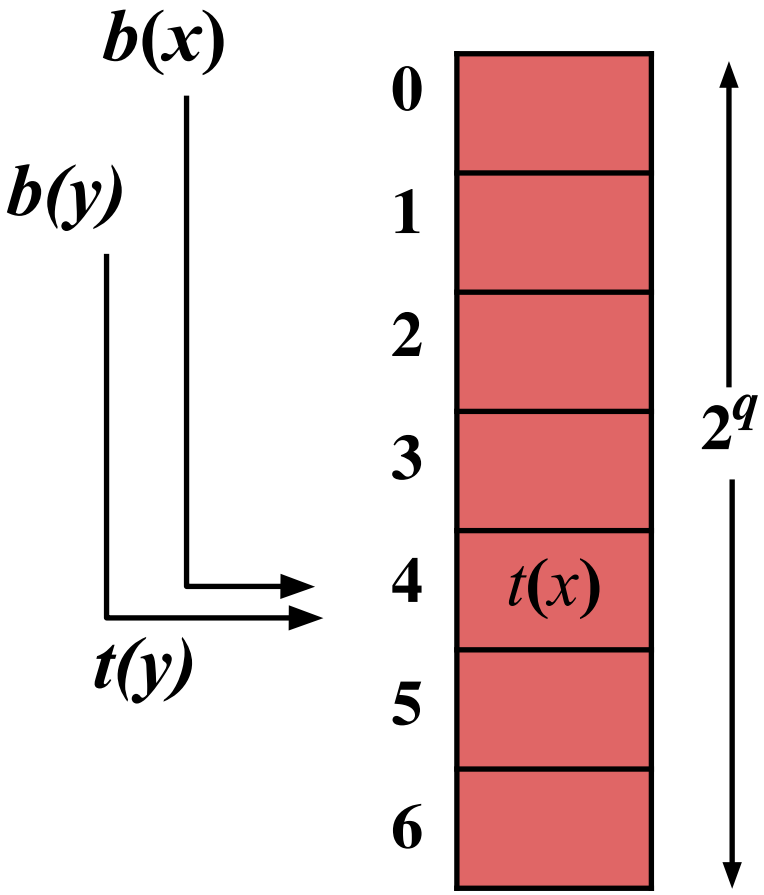
# Storing fingerprints compactly




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

# Storing fingerprints compactly

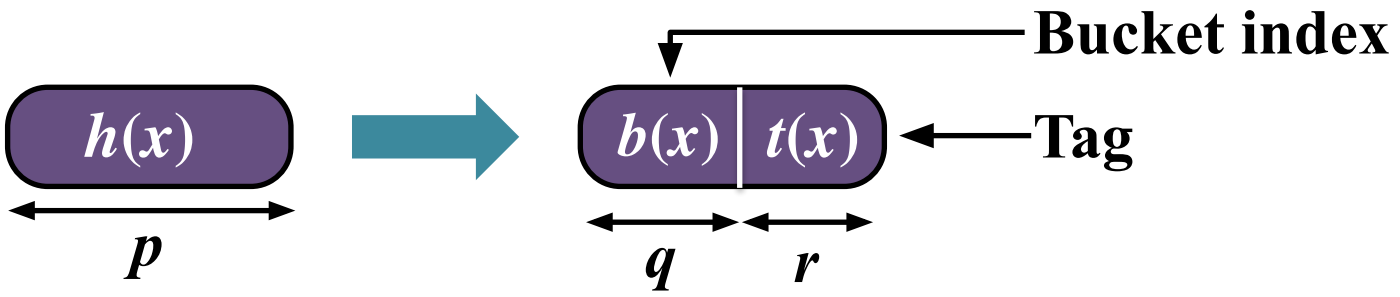


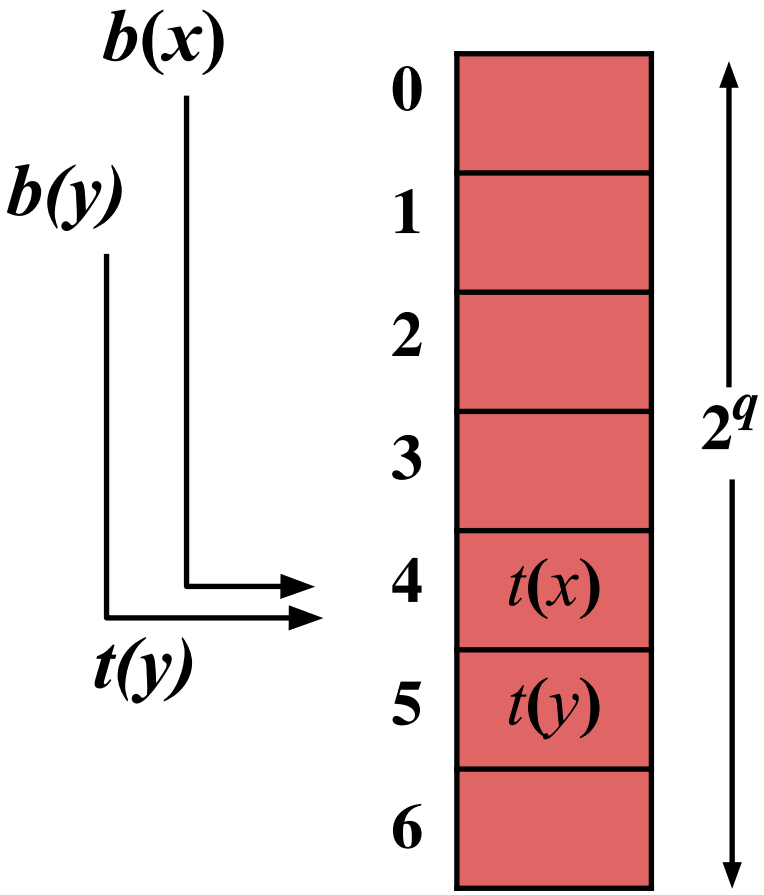




- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

Collisions in the hash table?

# Storing fingerprints compactly



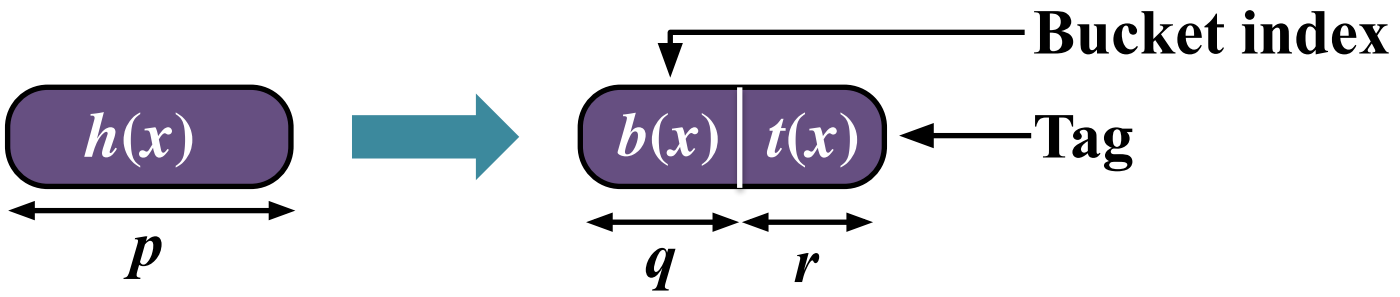



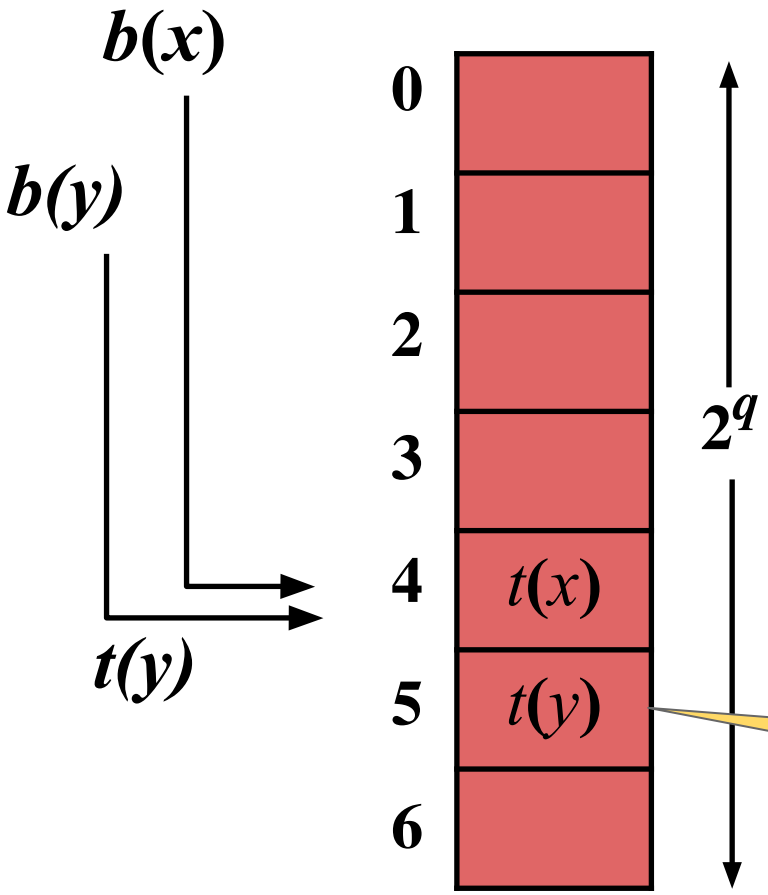

- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

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

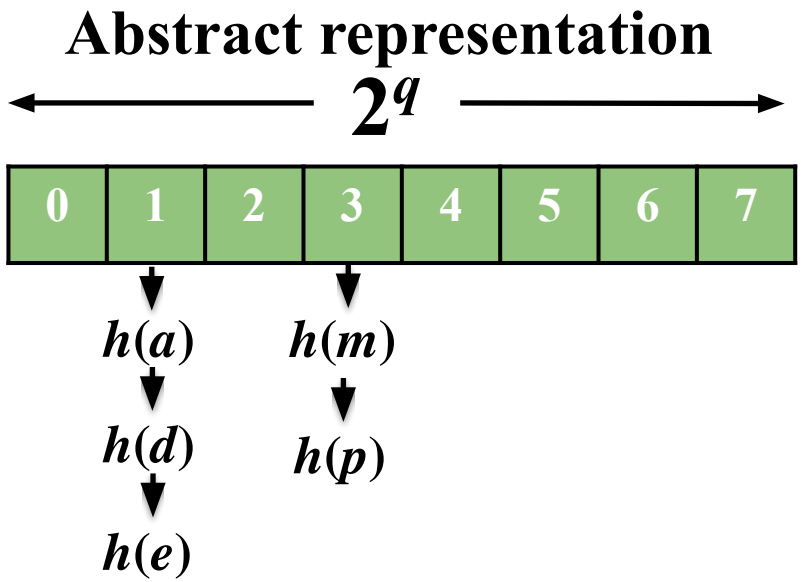
**$t(y)$ belongs to slots 4 or 5?**

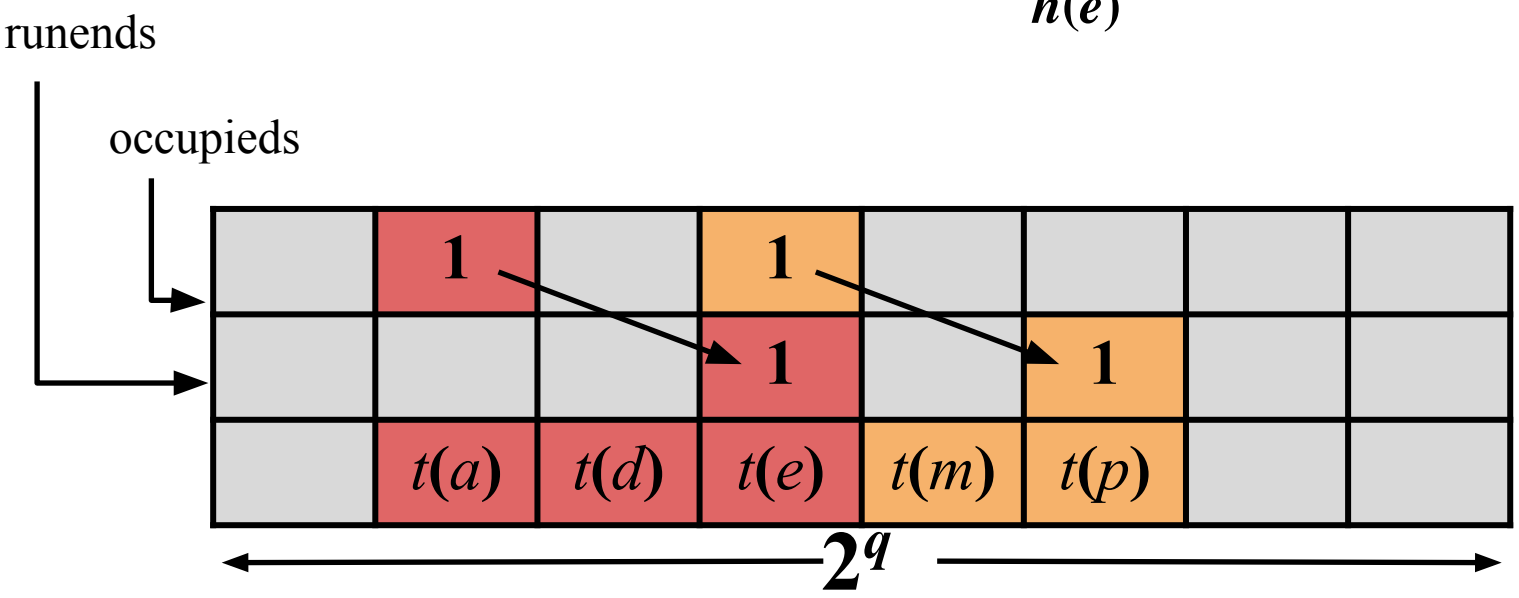
# Resolving collisions in the QF [Pandey et al. SIGMOD '17]

**Implementation:**

**2 meta-bits per slot.**

$h(x) \dashrightarrow h_0(x) \,||\, h_1(x)$



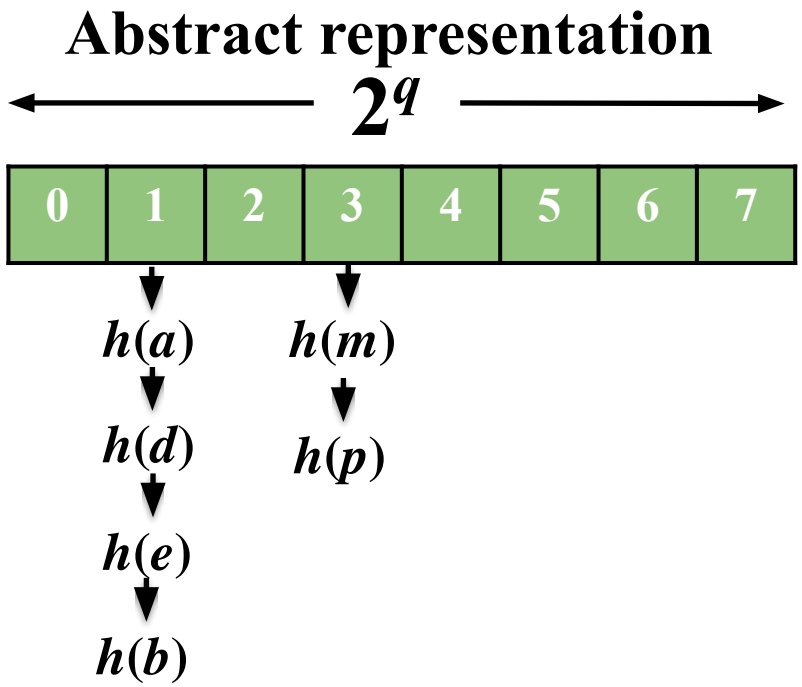



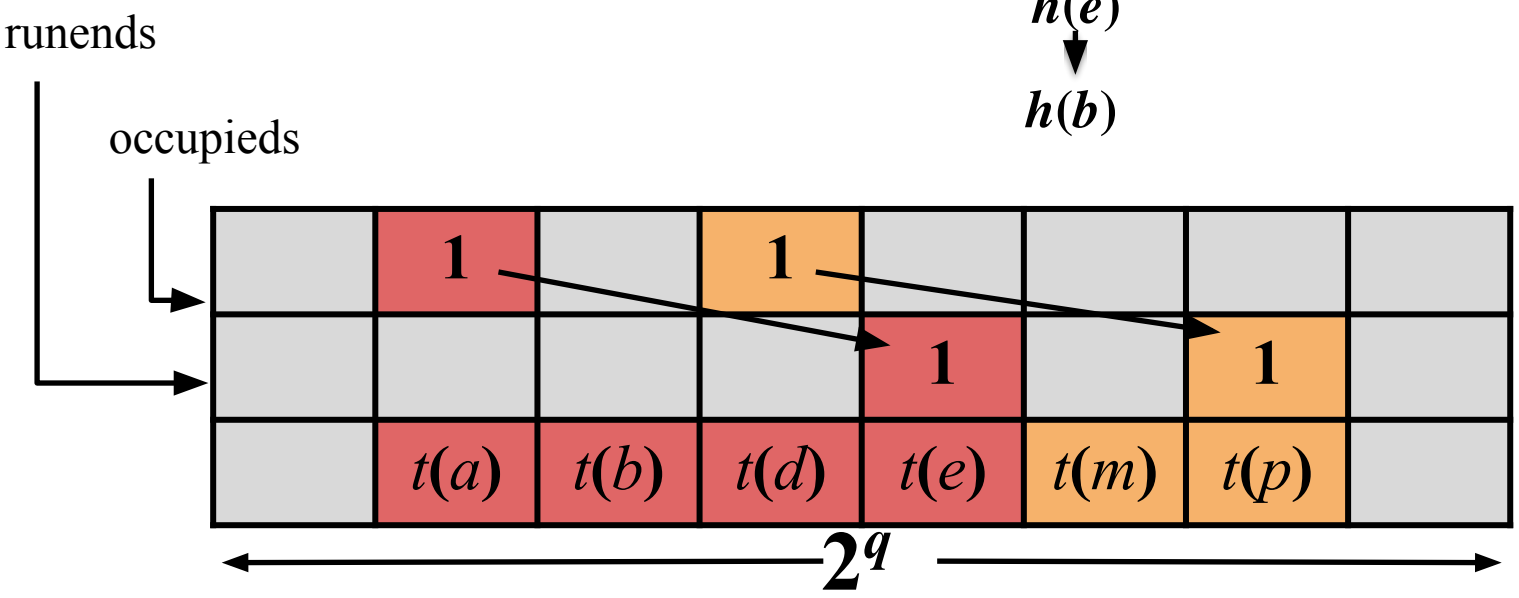
# Resolving collisions in the QF [Pandey et al. SIGMOD '17]

**Implementation:**

**2 meta-bits per slot.**

$h(x) \dashrightarrow h_0(x) \,||\, h_1(x)$



# Quotienting enables many features in the QF

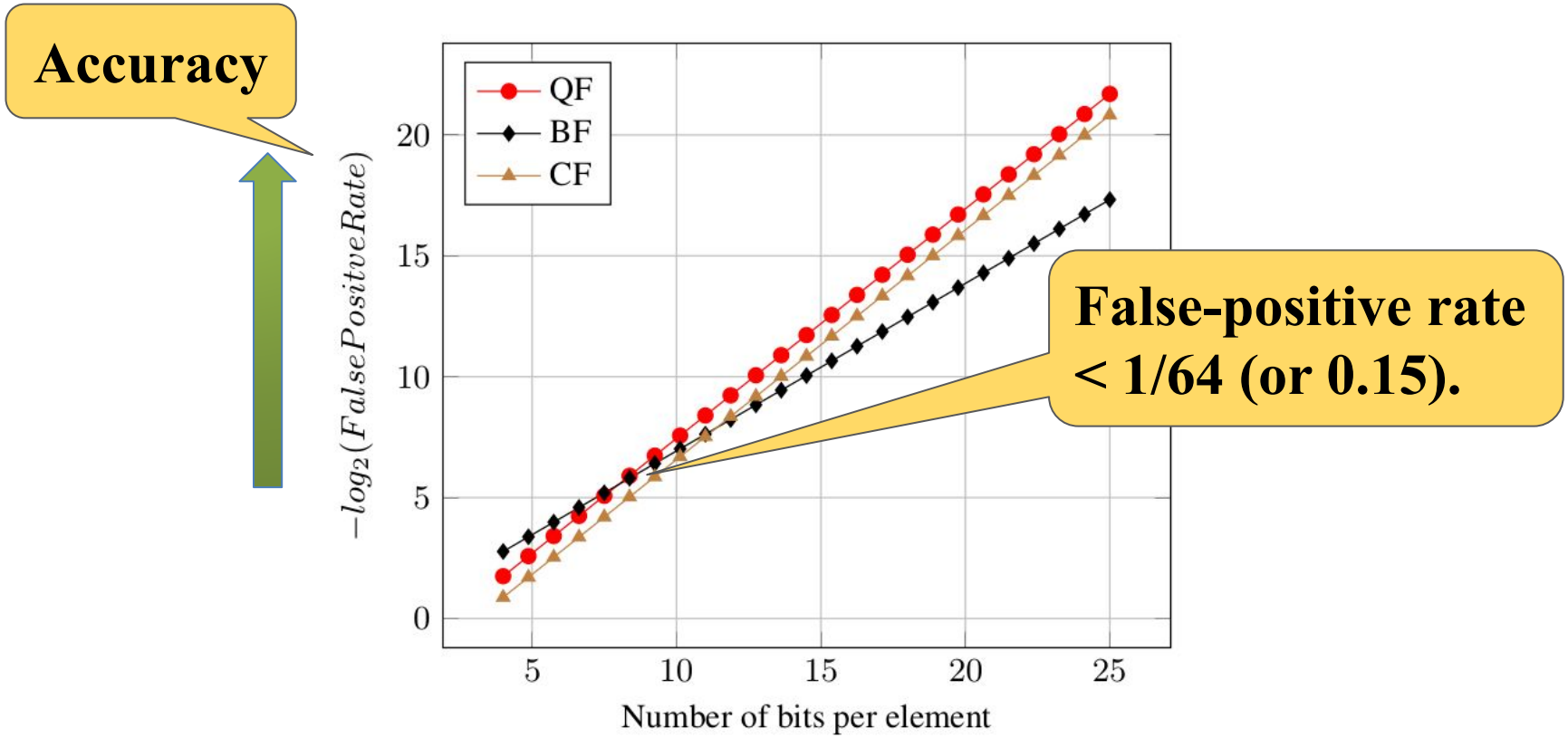
- Good cache locality
- Efficient scaling out-of-RAM
- Deletions
- Enumerability/Mergeability
- Resizing
- Maintains count estimates or associate values
- Uses variable-sized encoding for counts **[Counting quotient filter]**
  - **Asymptotically optimal space: $O(\sum |C(x)|)$**

# 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

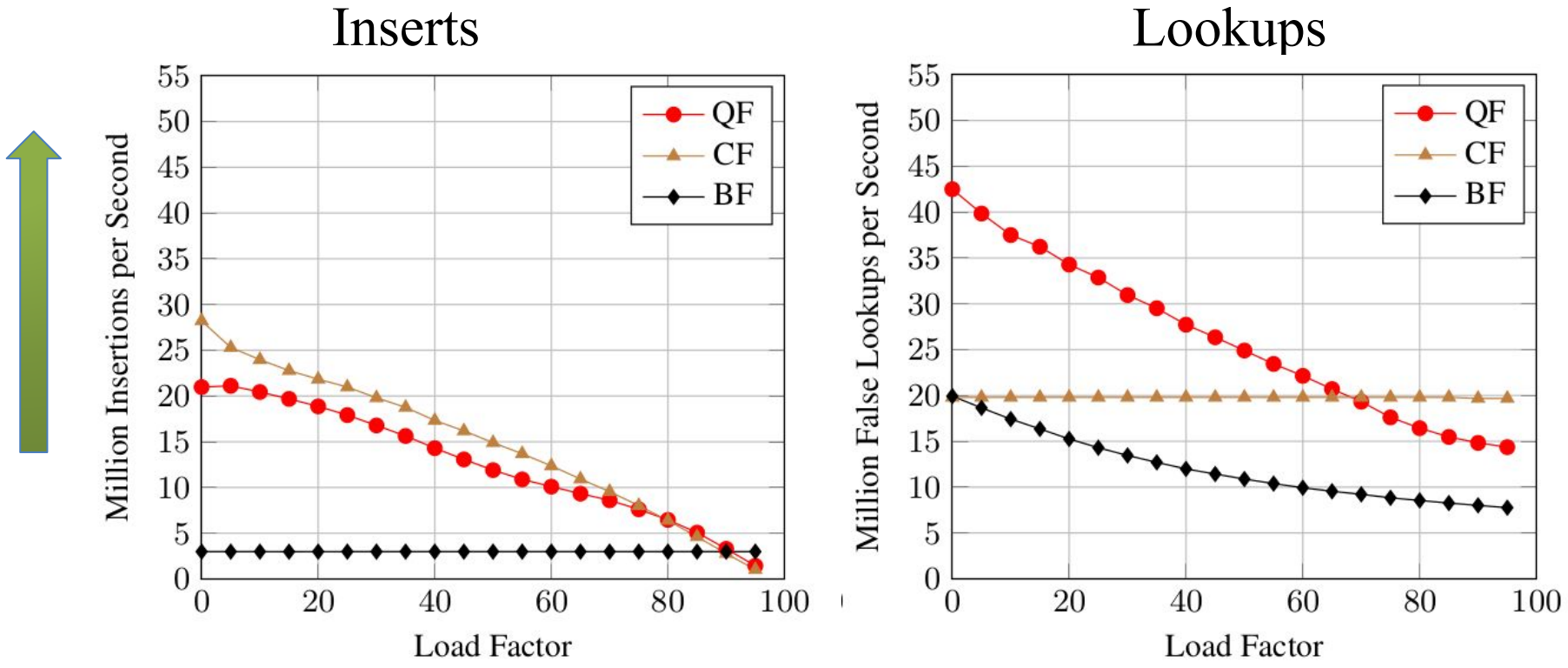
# Quotient filters use less space than Bloom filters for all practical configurations




**Bloom filter: *~1.44 log(1/ε)* bits/element.**

**Quotient filter: *~2.125 + log(1/ε)* 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 fast at low load-factors and slightly slower at higher load-factors

# Summary of filters

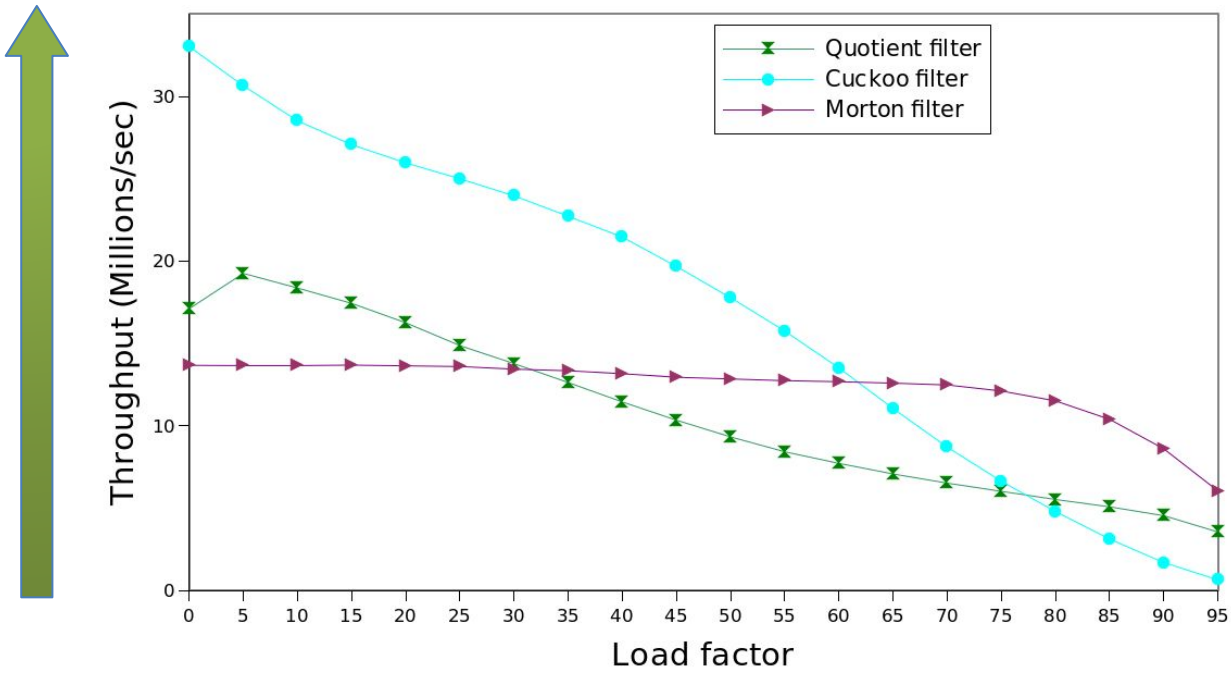
- Bloom filters [Bloom ‘70]
- Quotient filters [Pagh et al. ‘05, Dillinger et al. ‘09, Bender et al. ‘12, Einziger et al. ‘15, Pandey et al. ‘17]
- Cuckoo/Morton filters [Fan et al. ‘14, Breslow & Jayasena ‘18]

State of the art in practical dynamic filters.

- Others
  - Mostly based on perfect hashing and/or linear algebra
  - Mostly static
  - e.g., Xor filters [Graf & Lemire ‘20]

# Current filter performance

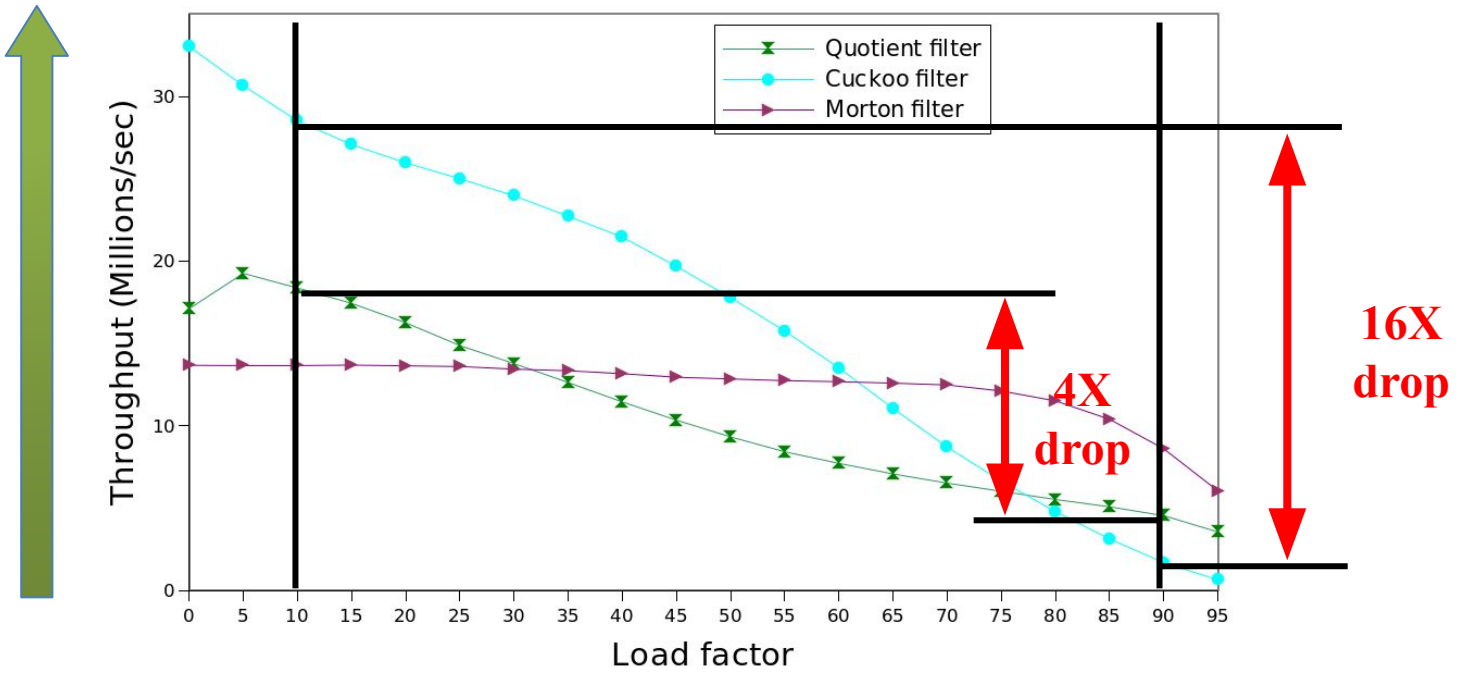
Performance suffers due to high-overhead of ***collision resolution***




**Question: do you see a problem here??**

# Current filters have a problem..

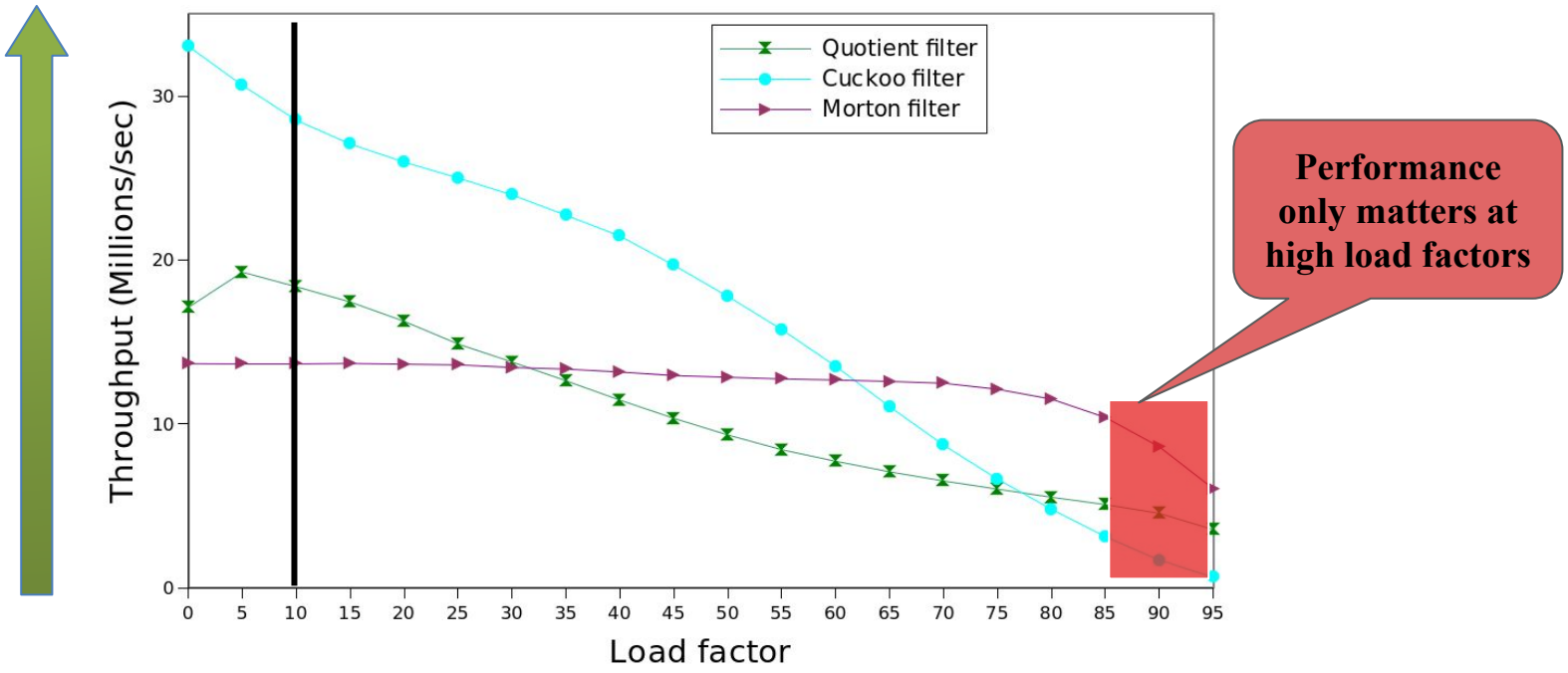
Performance suffers due to high-overhead of ***collision resolution***




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.

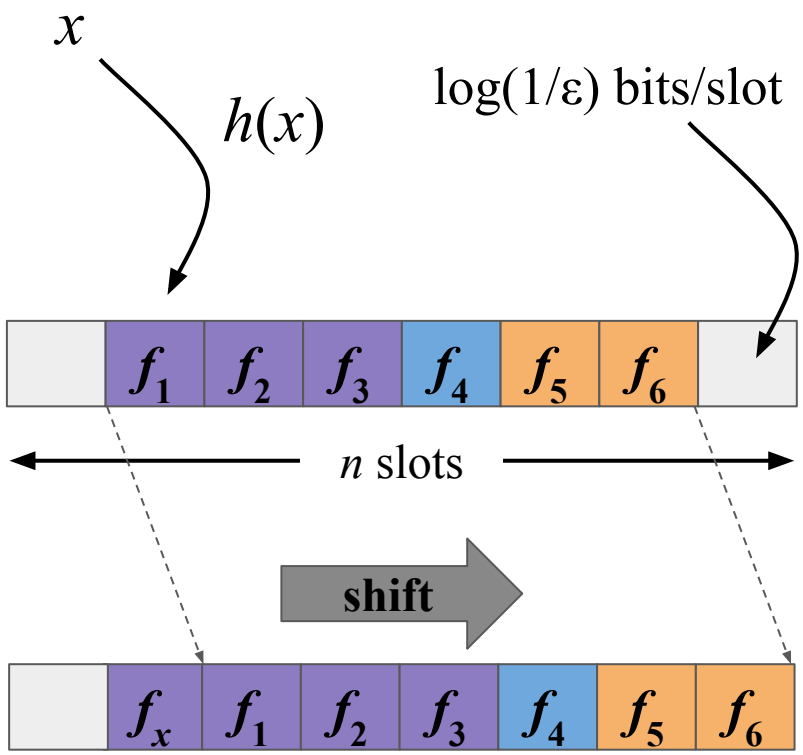
# Why quotient filters slow down

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

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

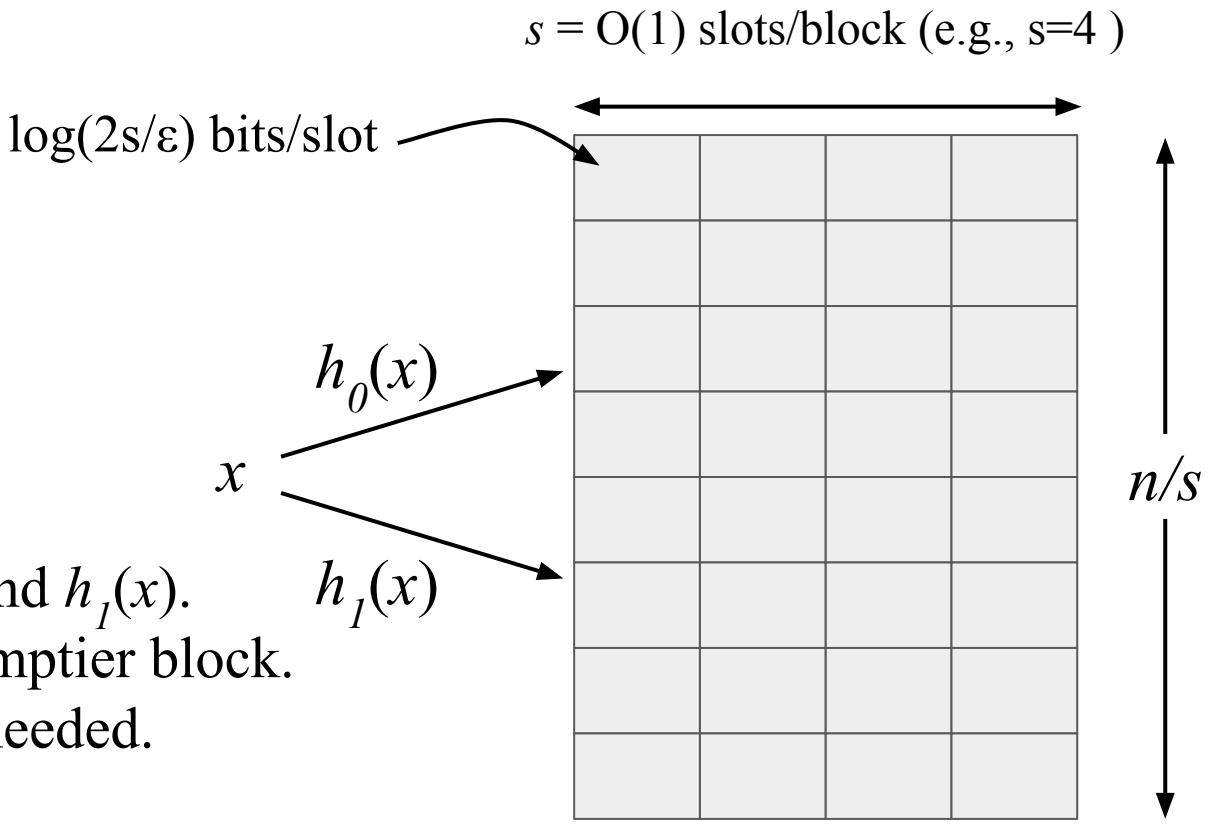
To insert item $x$:

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.

# Why cuckoo filters slow down



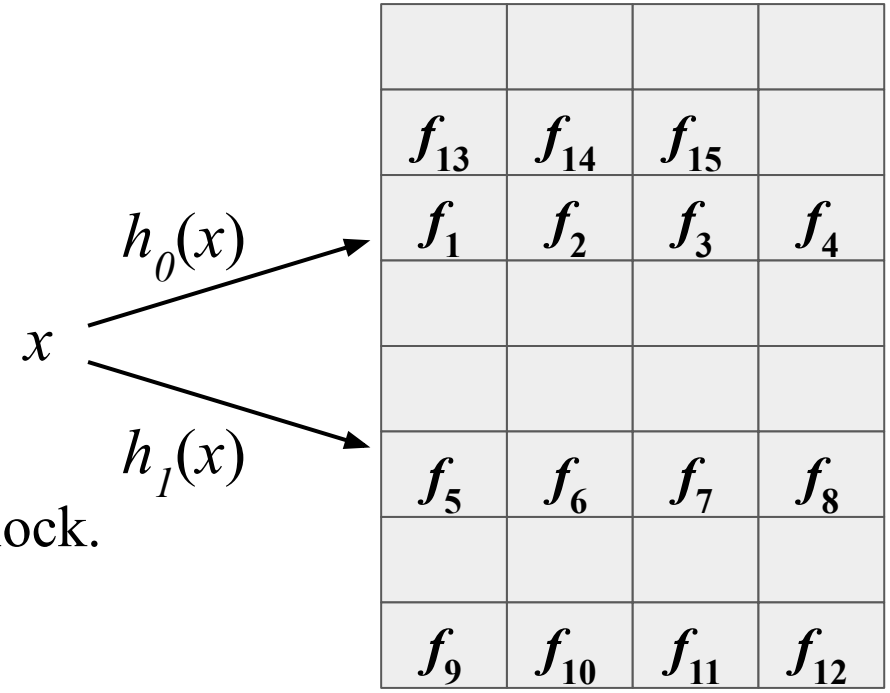

To insert item $x$:

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 $x$:

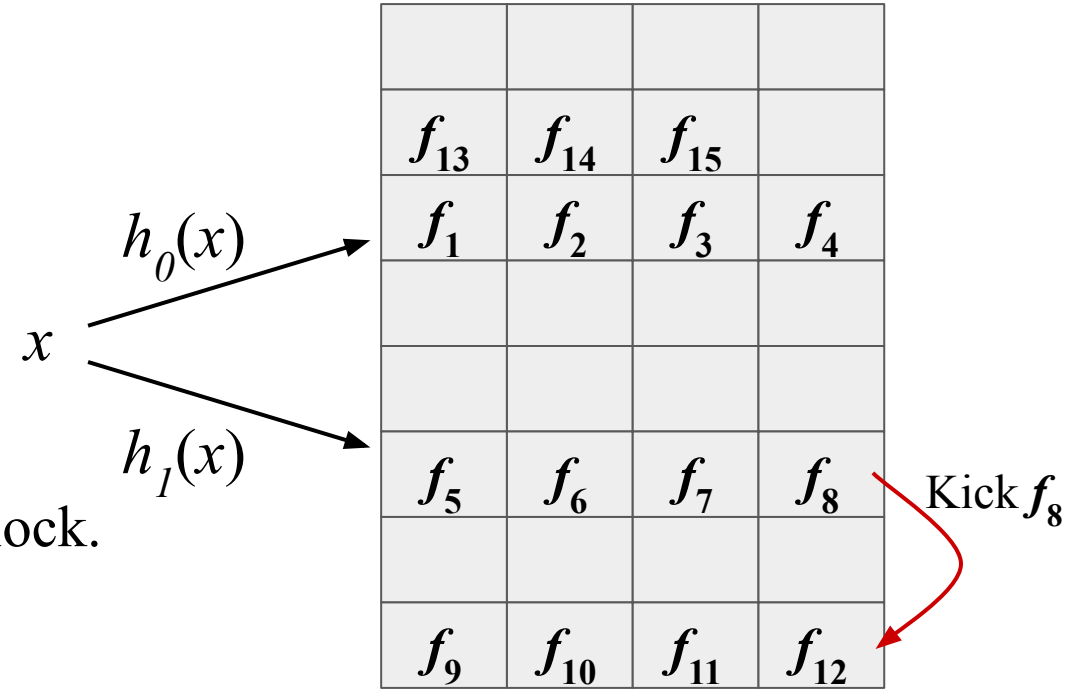
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.



# Why cuckoo filters slow down

To insert item $x$:

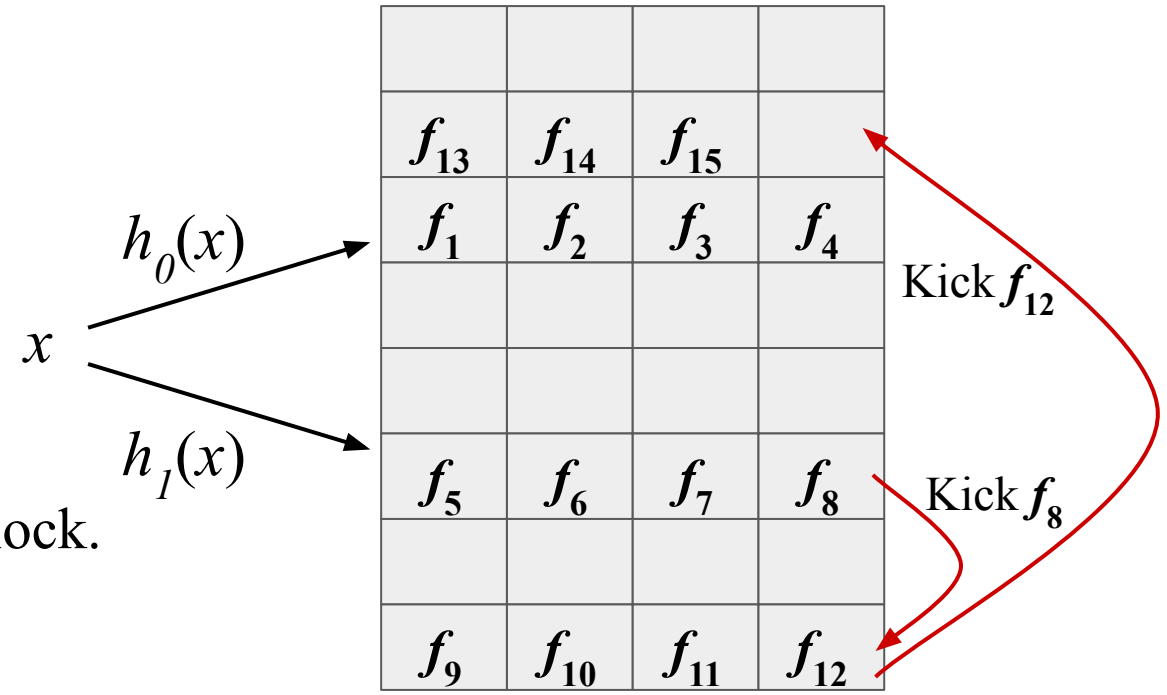
1. Compute $h_0(x)$ and $h_1(x)$.
2. Insert $f(x)$ into emptier block.
3. Kick an item if needed.



# 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.

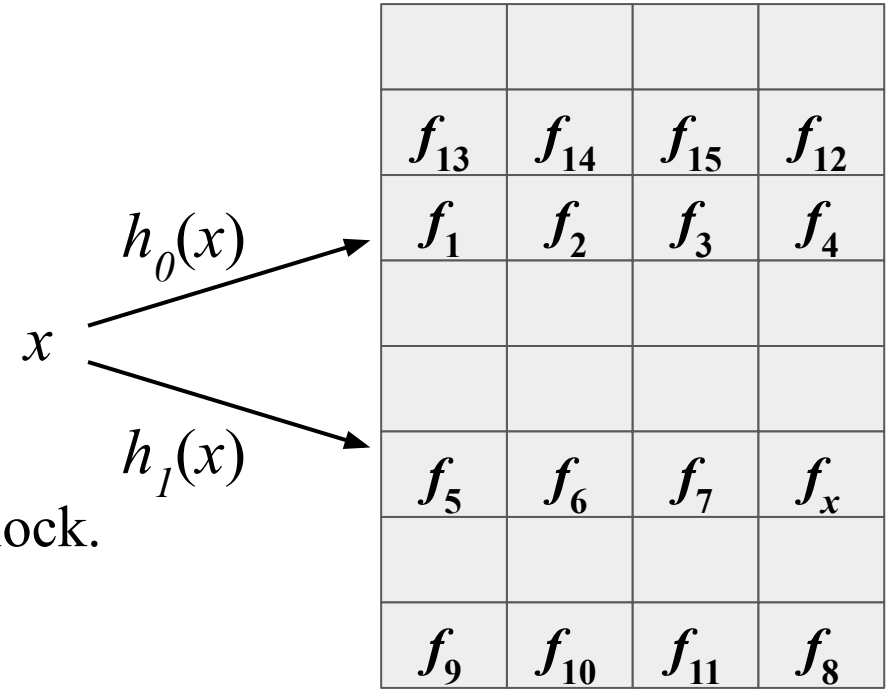



**Note:** $h_0(x)$ and $h_1(x)$ need to be dependent to support kicking.

# 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.



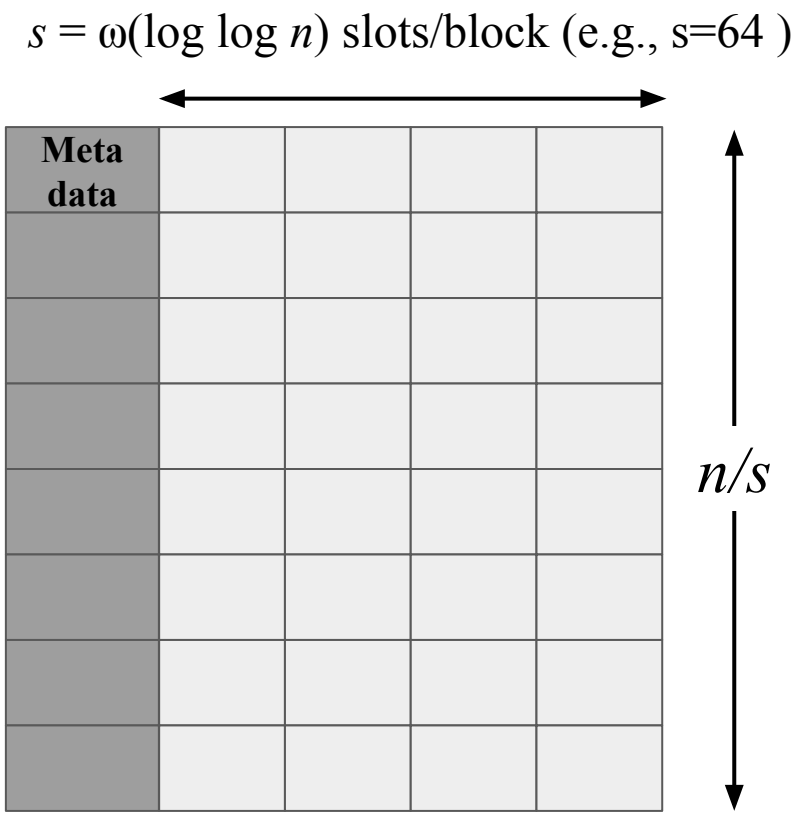

As the CF fills, inserts have to do more kicking.

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

# Cuckoo filter performance [Fan et al. '14]

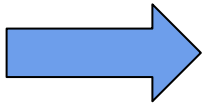
| | Optimal | Cuckoo filter |
|---|---|---|
| Space (bits) | $\approx n \log(1/\epsilon) + \Omega(n)$ | $\approx n \log(1/\epsilon) + 3n$ |
| CPU cost | $O(1)$ | up to 500 |
| Data locality | $O(1)$ probes | random probes |

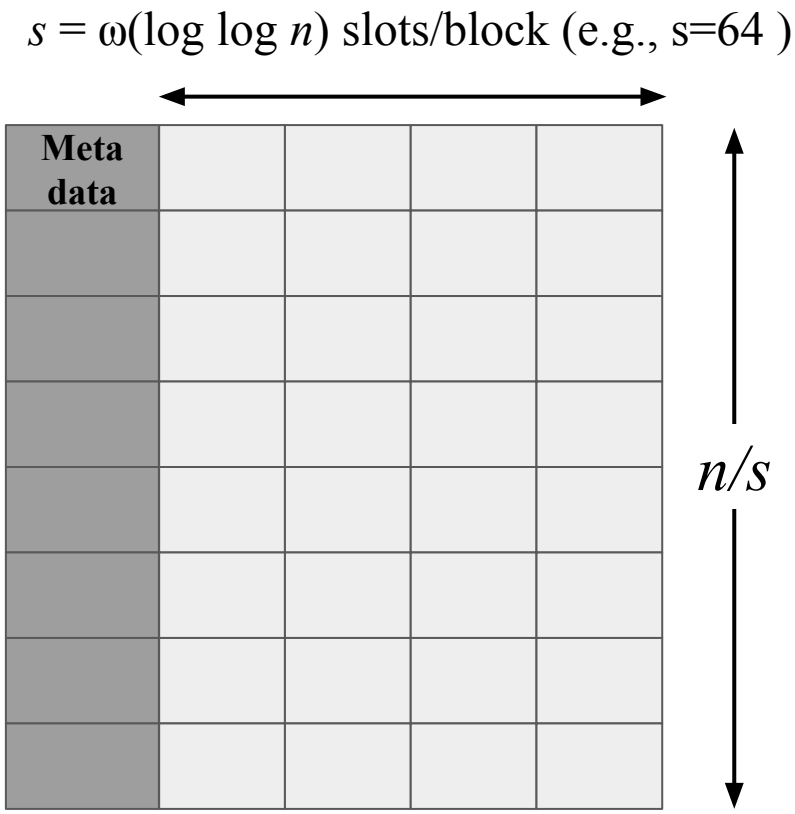
# Vector quotient filter design [Pandey et al. SIGMOD '21]



# Vector quotient filter design [Pandey et al. SIGMOD '21]

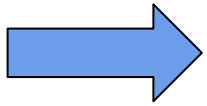
Each block is a small quotient filter with false-positive rate ε/2 and capacity *s*.

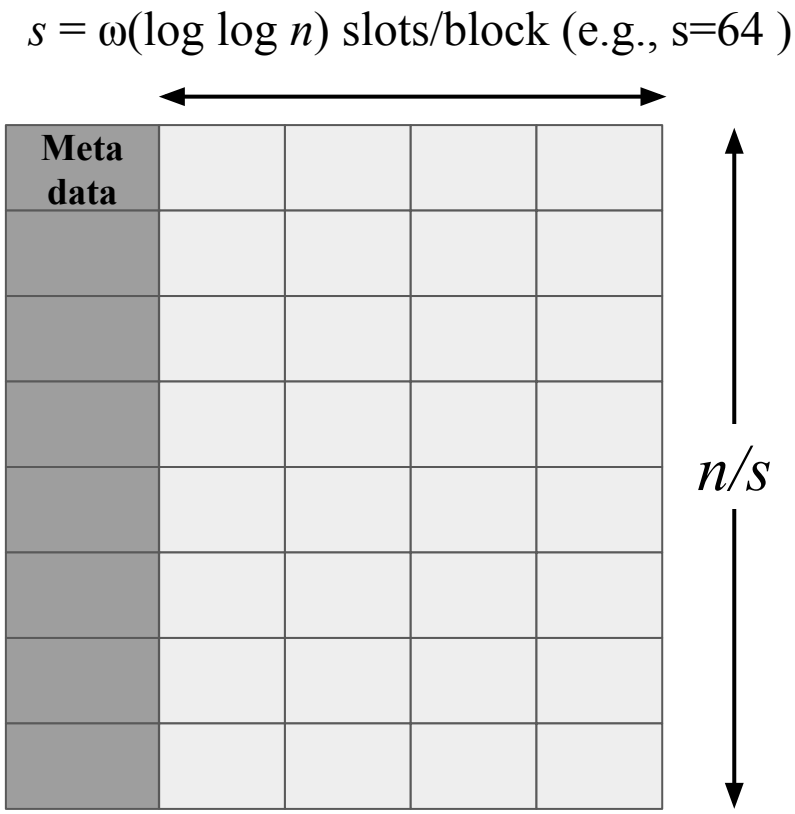




# Vector quotient filter design [Pandey et al. SIGMOD '21]

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

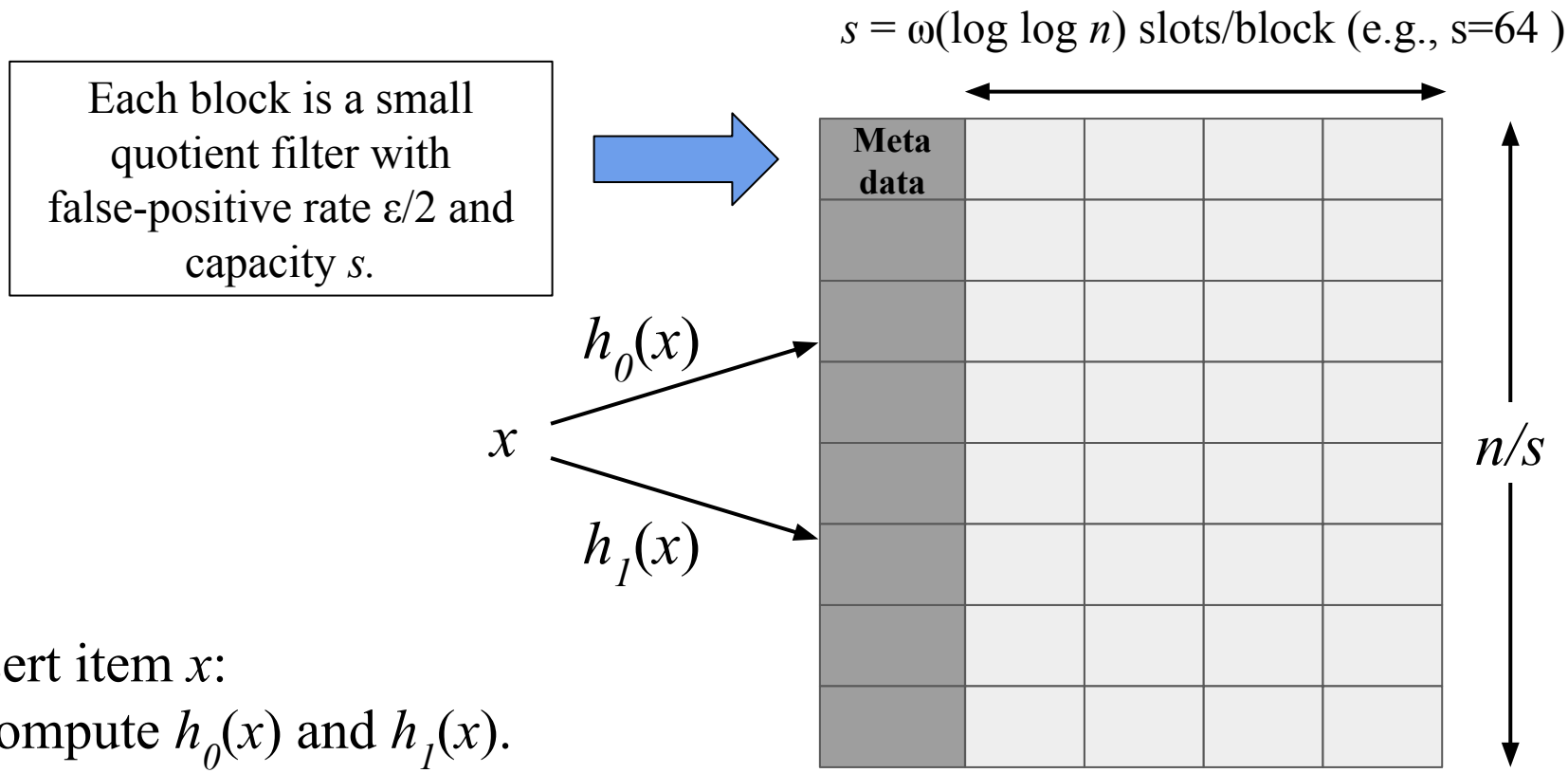





To insert item *x*:

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 [Pandey et al. SIGMOD '21]

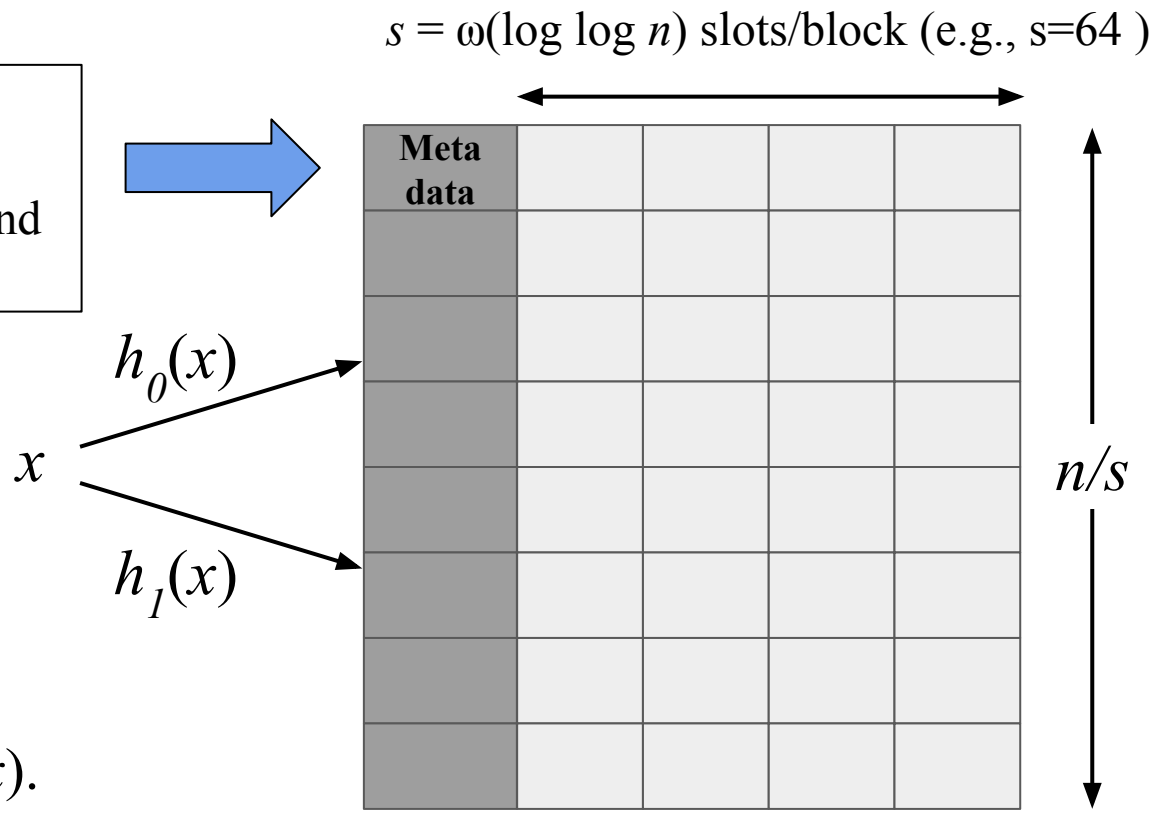



To insert item $x$:

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 [Pandey et al. SIGMOD '21]

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

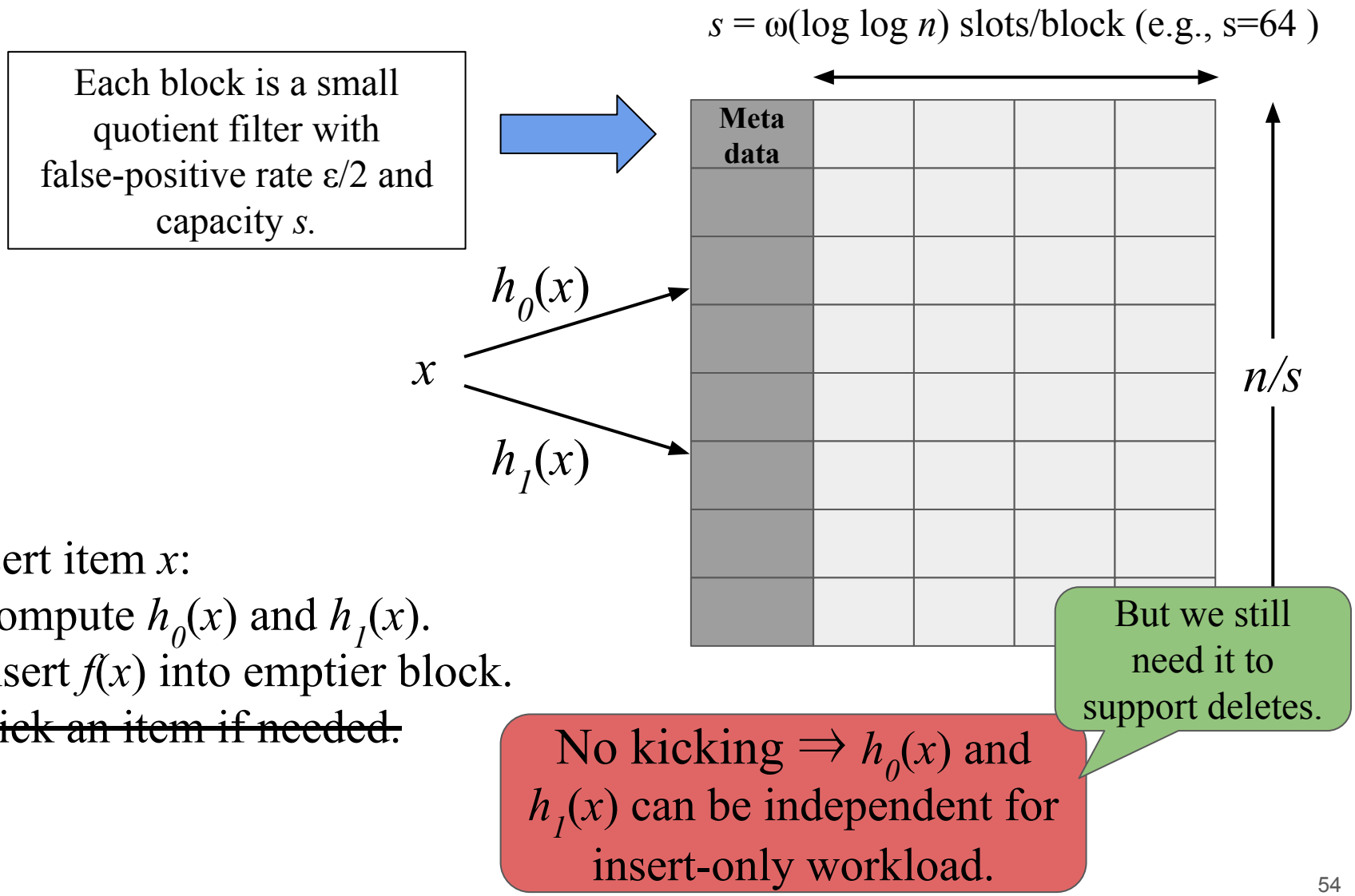



To insert item $x$:

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 [Pandey et al. SIGMOD '21]

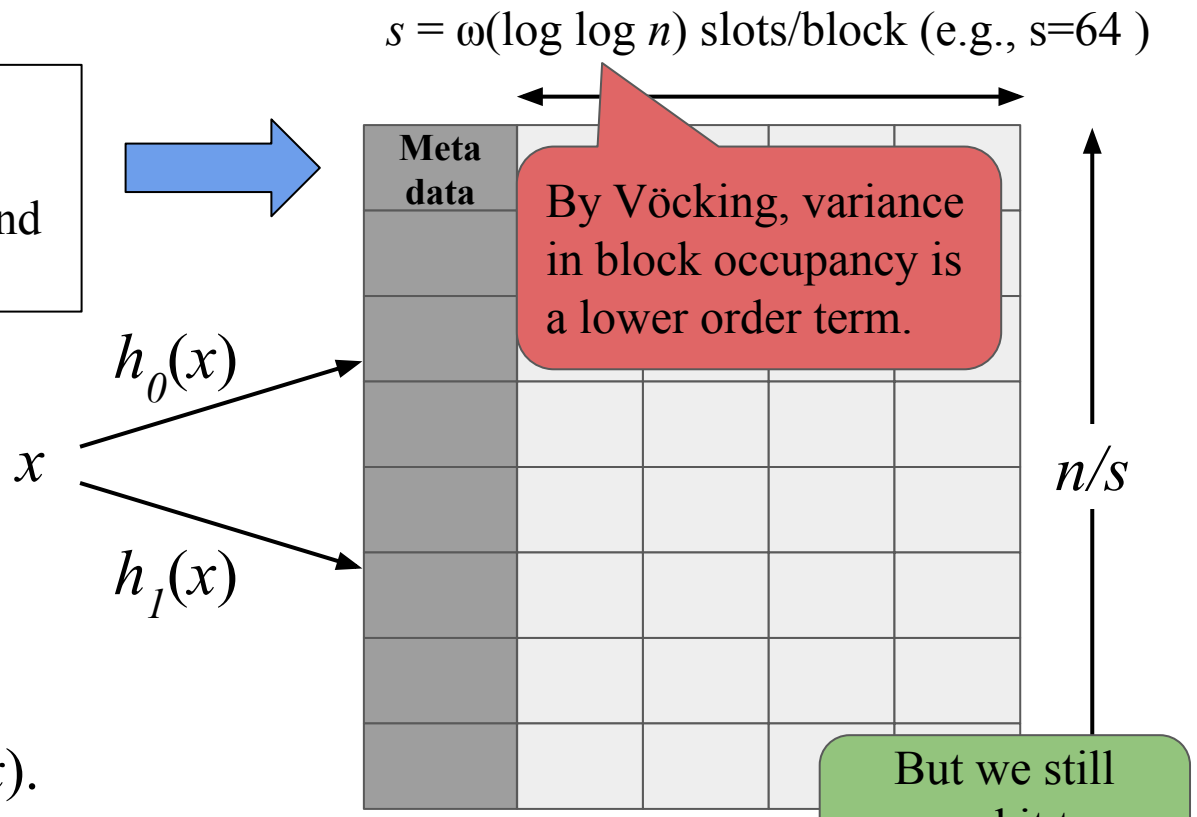



To insert item $x$:

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 [Pandey et al. SIGMOD '21]

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



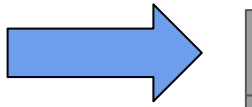

To insert item $x$:

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

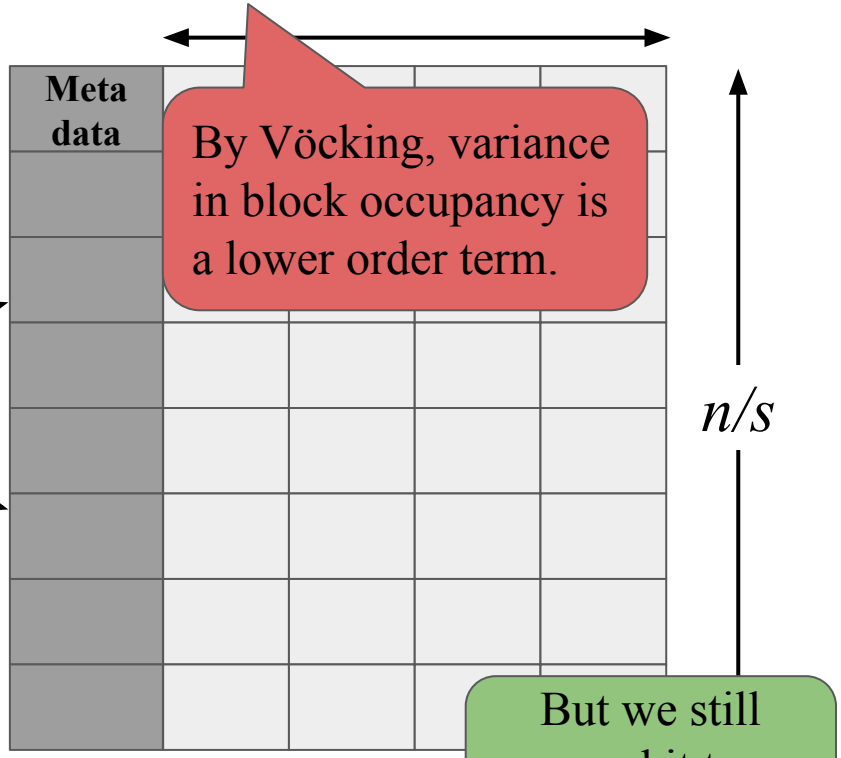
No kicking ⇒ $h_0(x)$ and $h_1(x)$ can be independent for insert-only workload.

# Vector quotient filter design [Pandey et al. SIGMOD '21]

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



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




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

No kicking ⇒ easier concurrency

$h_0(x)$

$x$

$h_1(x)$

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.

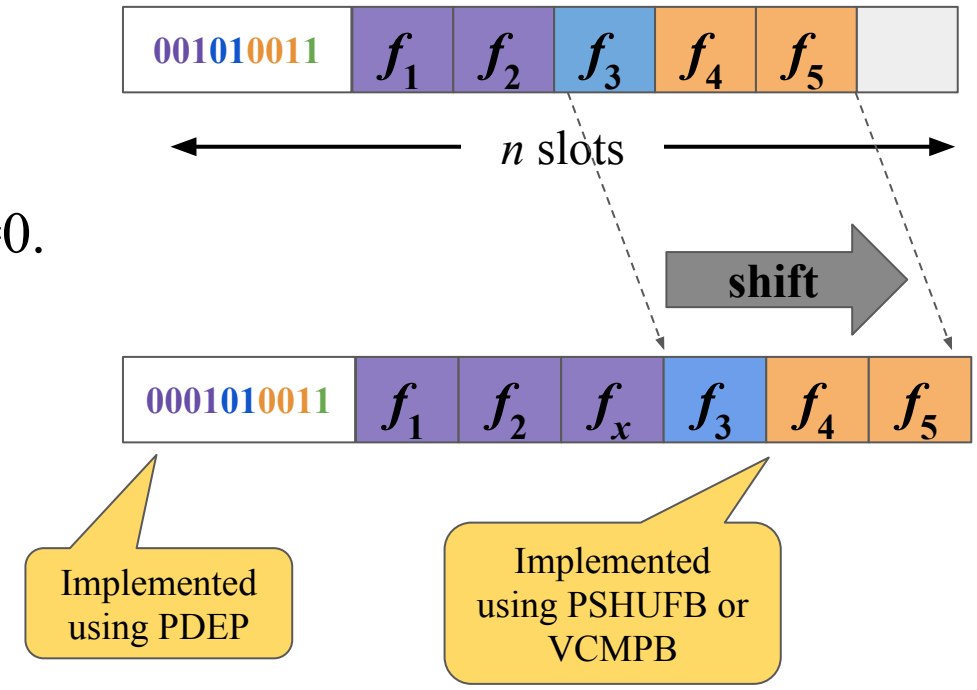
# A vectorizable mini quotient filter

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 $b=s/\ln 2$.

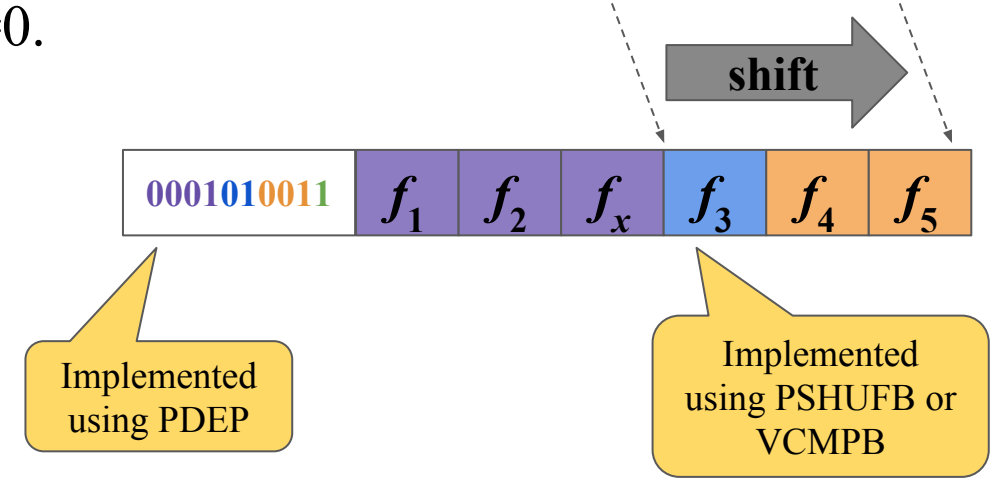
# A vectorizable mini quotient filter

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 $\beta(x)=0$.

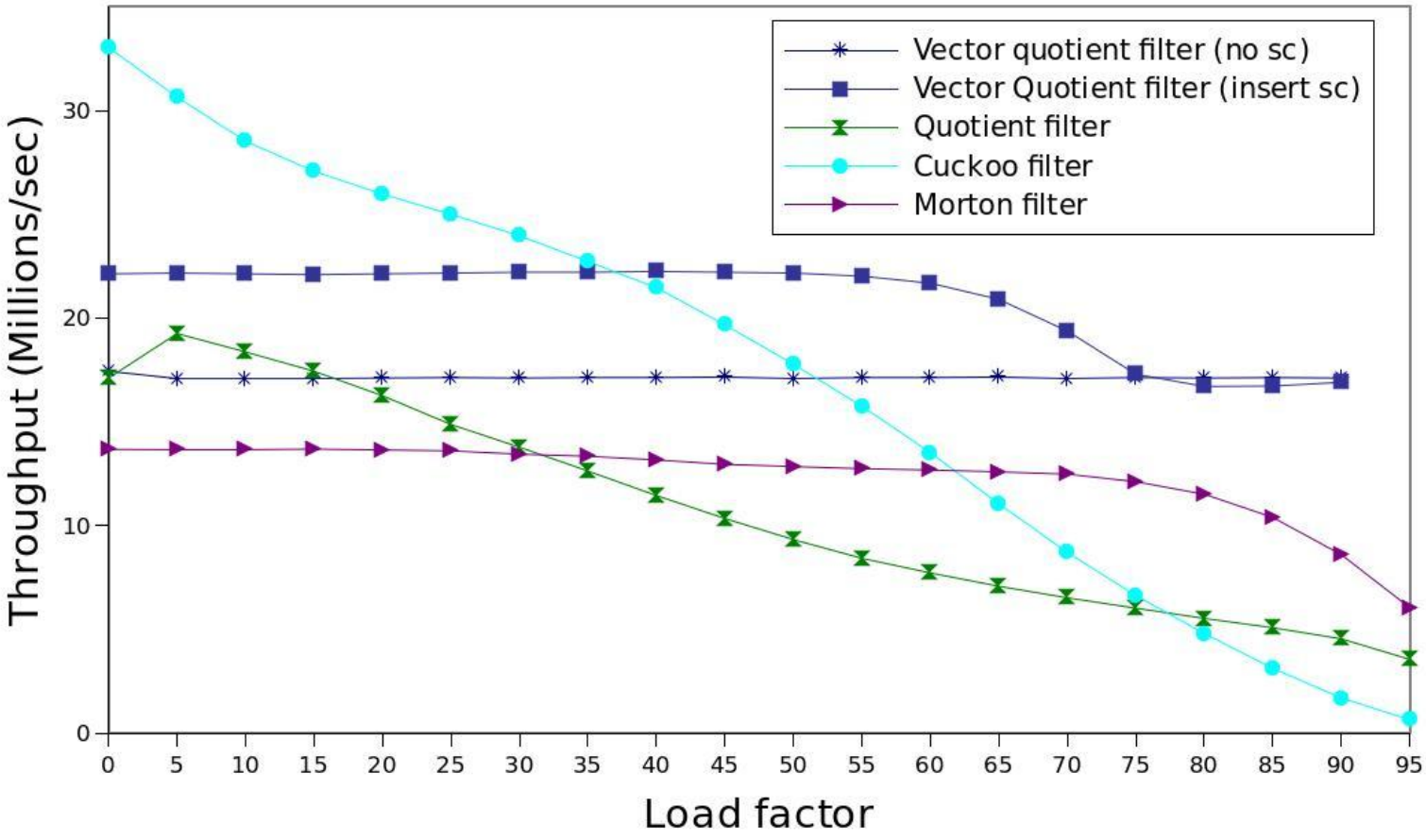



Space efficiency is maximized when $b=s/\ln 2$.

# Vector quotient filter (VQF) performance

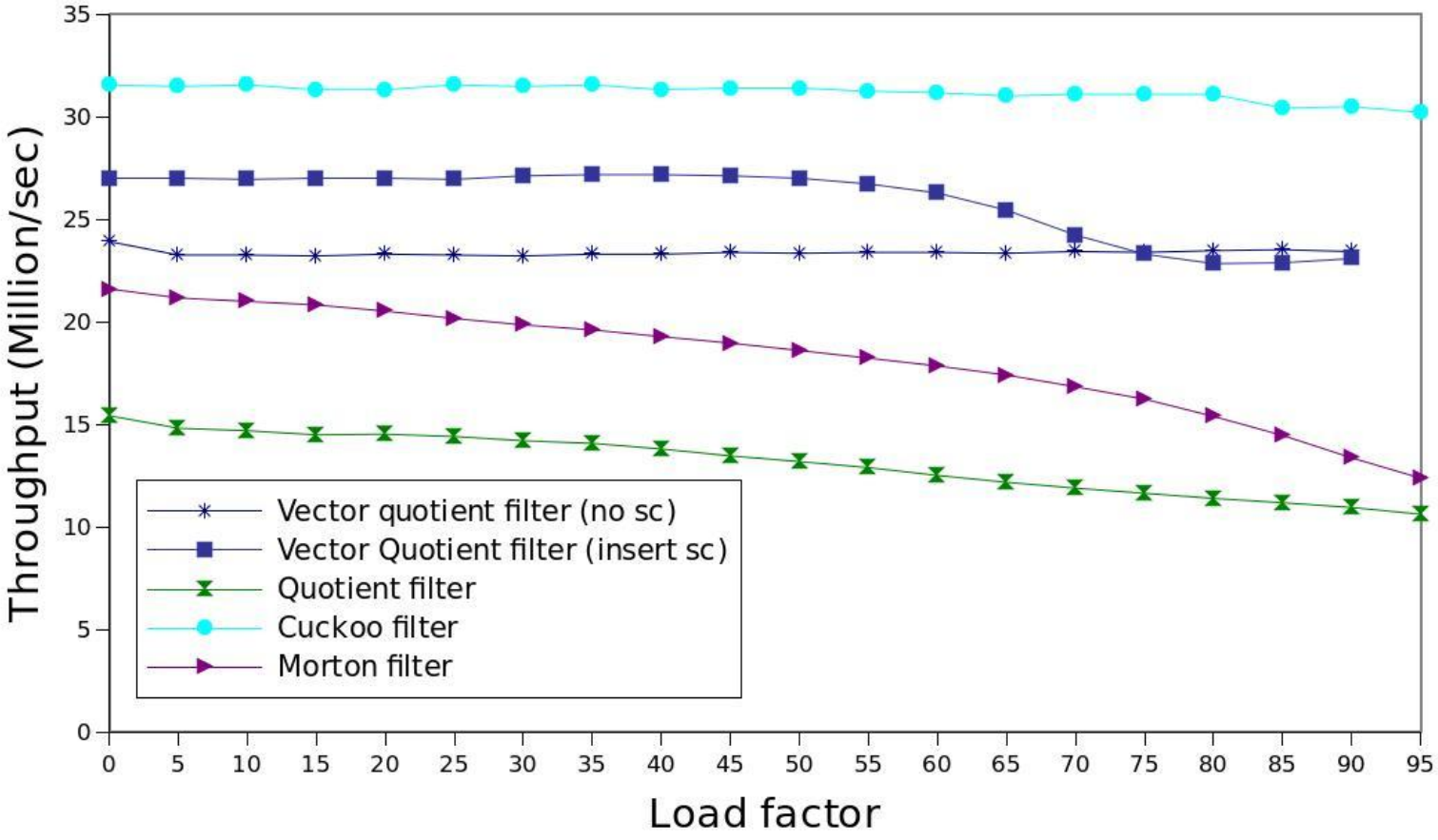
| | Optimal | VQF |
|---|---|---|
| Space (bits) | $\approx n \log(1/\epsilon) + \Omega(n)$ | $\approx n \log(1/\epsilon) + 2.91n$ |
| CPU cost | $O(1)$ | $O(1)$ |
| Data locality | $O(1)$ probes | 2 probes |

# Evaluation: insertion

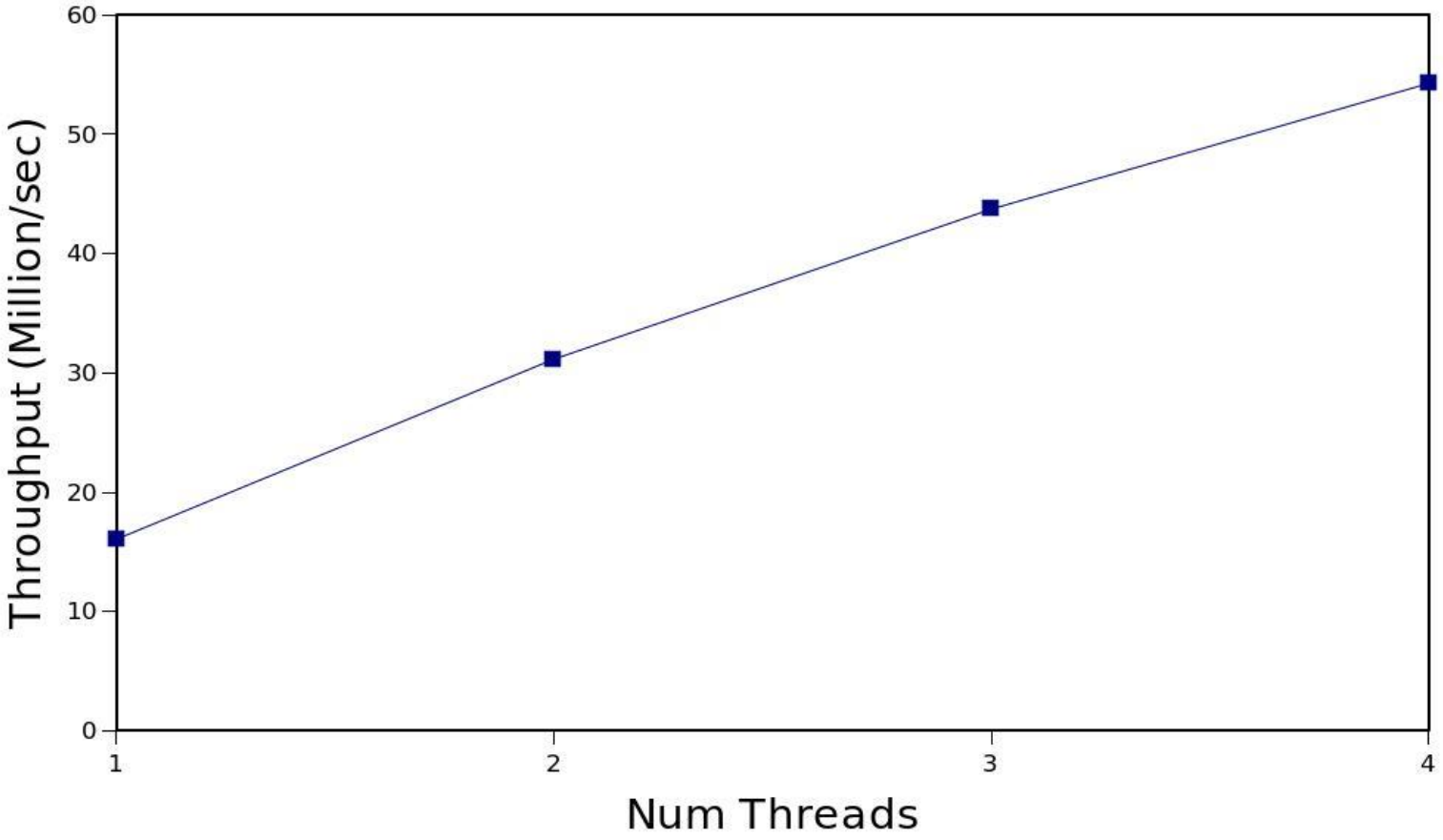



The vector quotient filter offers high performance at all load factors.

# Evaluation: lookups



# Evaluation: concurrency



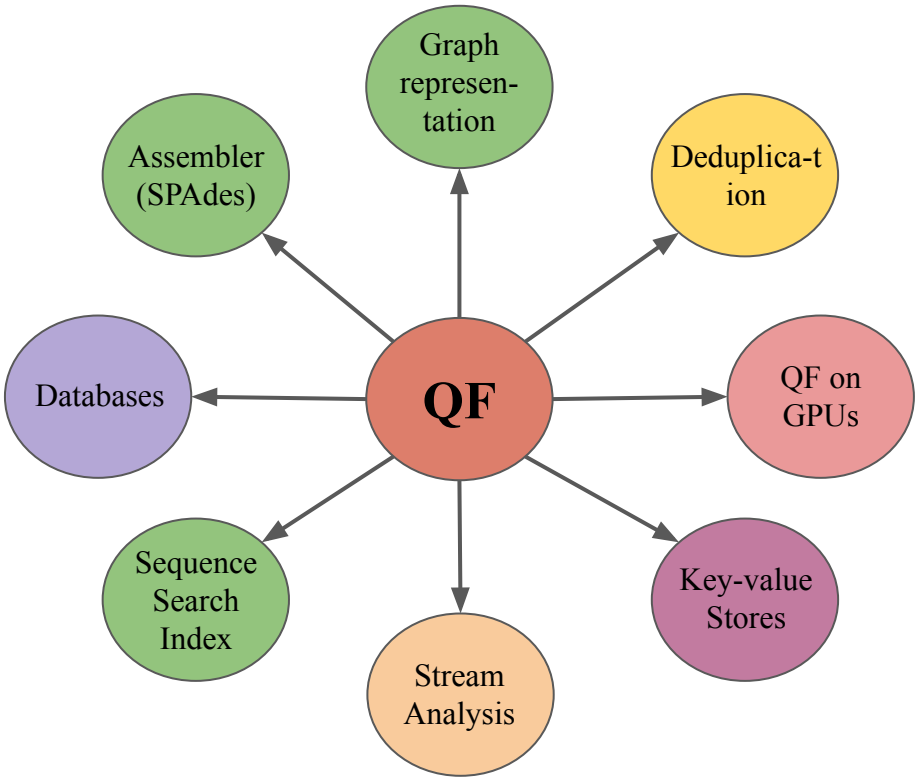
# Quotient filter's impact in computer science

**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***

# Quotient filter's impact in computer science

**Computational biology**

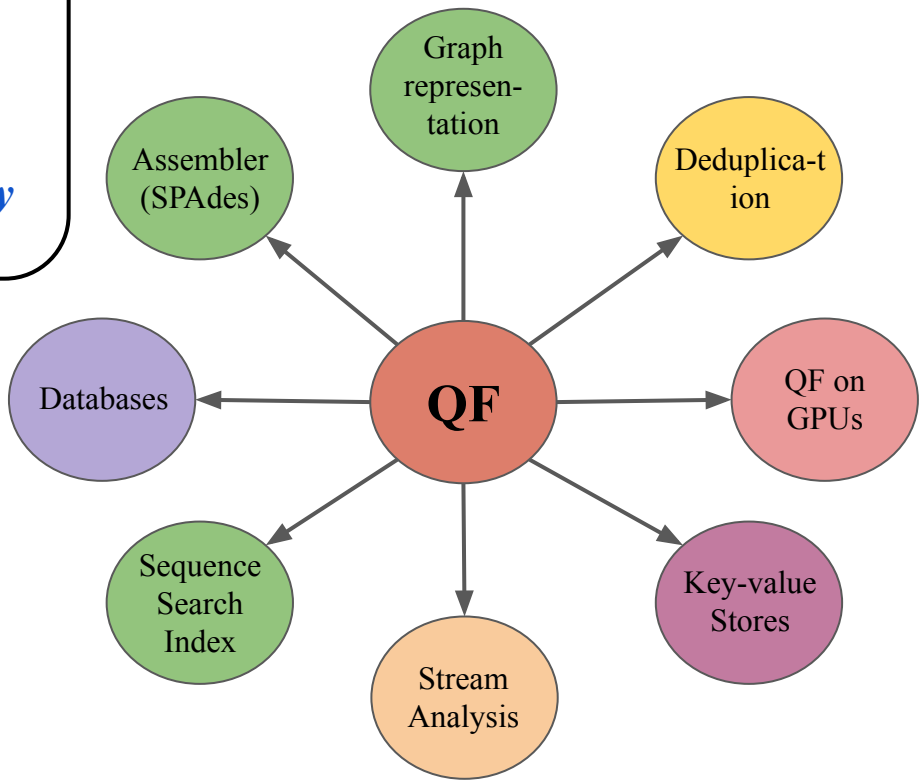
1. Squeakr
2. deBGR
3. Mantis
4. VariantStore
5. MetaHipmer
6. SPAdes assembler
7. Khmer software
8. MQF

**Industry**

1. VMware
2. Nutanix
3. Apocrypha
4. Hyrise
5. ***A data security startup***

**Databases/Systems**

1. Anomaly detection
2. BetrFS file system
3. Graph representation
4. Counting on GPUs
5. Concurrent filters




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