# Data Systems at Scale

## Scaling Up by Scaling Down and Out

**Prashant Pandey, University of Utah**

**Northeastern University, Boston (Starting Spring 2025)**

My goal as a researcher is to build scalable data systems with strong theoretical guarantees

My goal as a researcher is to build scalable data systems with strong theoretical guarantees

↓

To scale and democratize next-generation data analyses

# Three approaches to build scalable data systems

## Compress it

Goal: make data smaller to fit inside fast memory

Filters, sketches, succinct data structures

## Organize it

Goal: organize data in a I/O friendly way

B-trees, LSM-trees, $B^e$-trees

## Distribute it

Goal: distribute data & reduce inter-node communication

Distributed hash tables

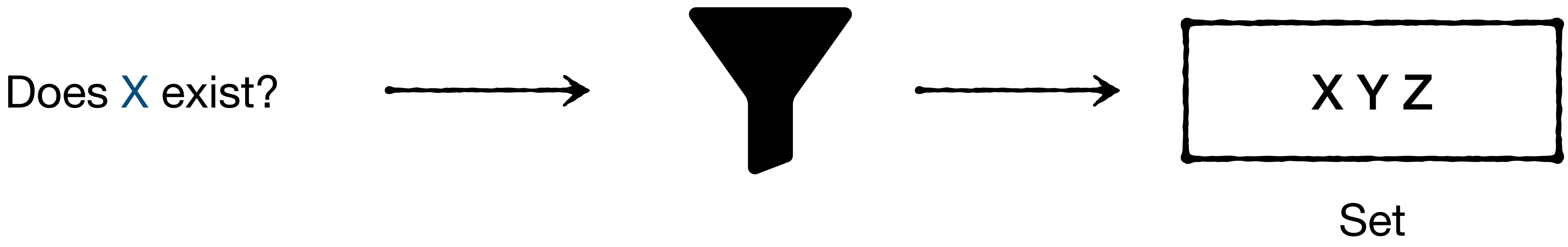
# In this talk:

**Compress it**

Goal: make data smaller to fit
inside fast memory

**Filters**

# What is a filter data structure?



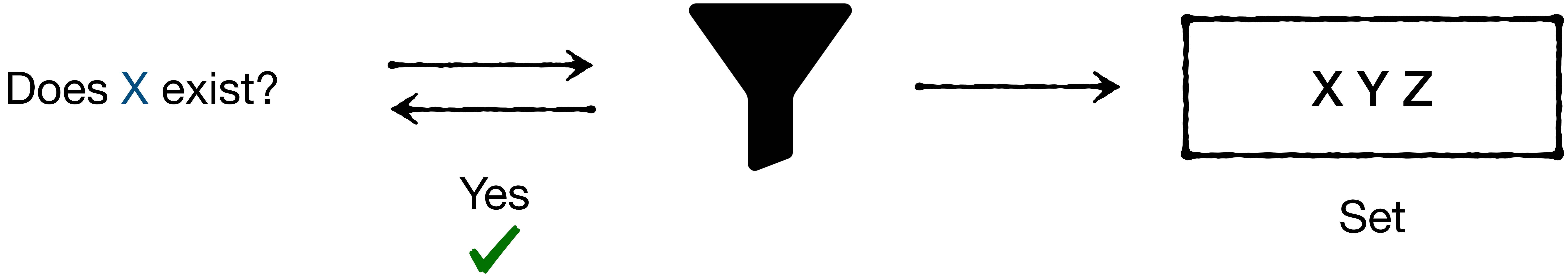
Does X exist?

X Y Z

Set

A filter compactly represents a set by trading off accuracy for space efficiency

# What is a filter data structure?



Does X exist?

Yes

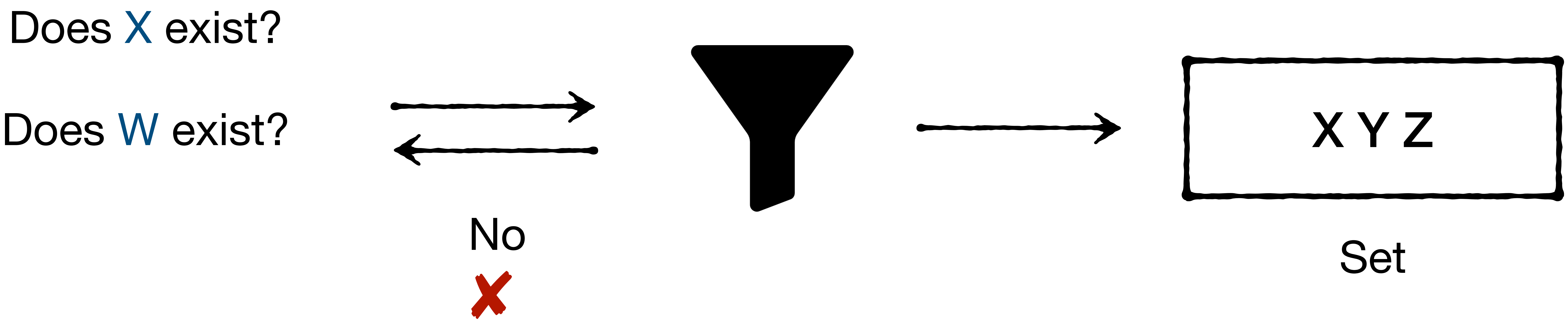
✓

X Y Z

Set

A filter compactly represents a set by trading off accuracy for space efficiency

# What is a filter data structure?



Does X exist?

Does W exist?

No ✘

X Y Z

Set

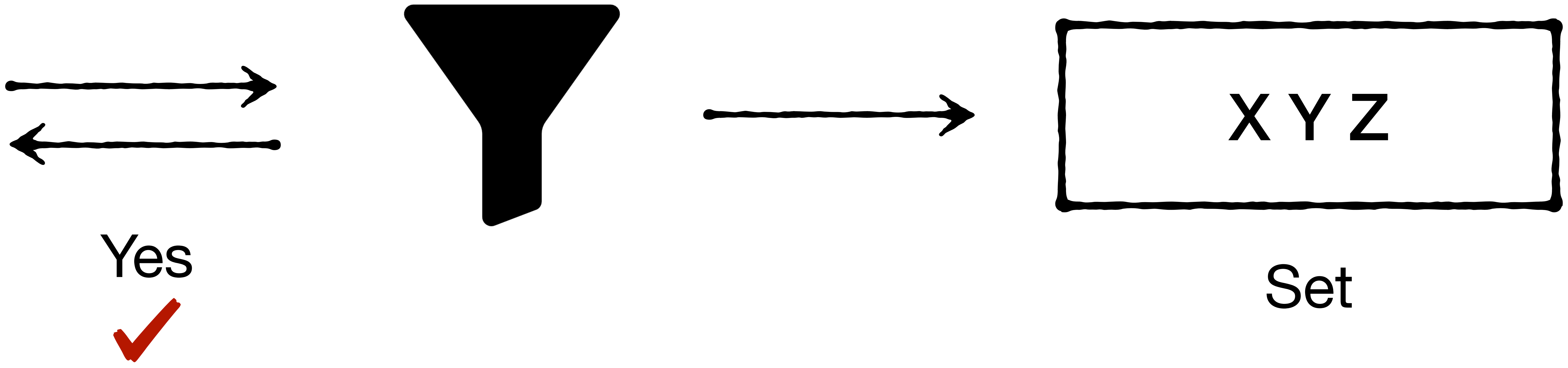
A filter compactly represents a set by trading off accuracy for space efficiency

# What is a filter data structure?

Does X exist?

Does W exist?

Does A exist?



Yes ✓

X Y Z

Set

A filter compactly represents a set by trading off accuracy for space efficiency

# A filter guarantees a false-positive rate $\epsilon$

$q =$ **query item** $S =$ **set of items**

| | | |
|---|---|---|
| if q ∈ S, return | **True** with probability 1 | true positive |
| if q ∉ S, return | **False** with probability $> 1 - \epsilon$ | true negative |
| | **True** with probability $\leq \epsilon$ | false positive |

One-sided errors

False positives with tunable probability

# False-positives enable filters to be compact

$n =$ **number of items** $U =$ **universe of items**

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

Small

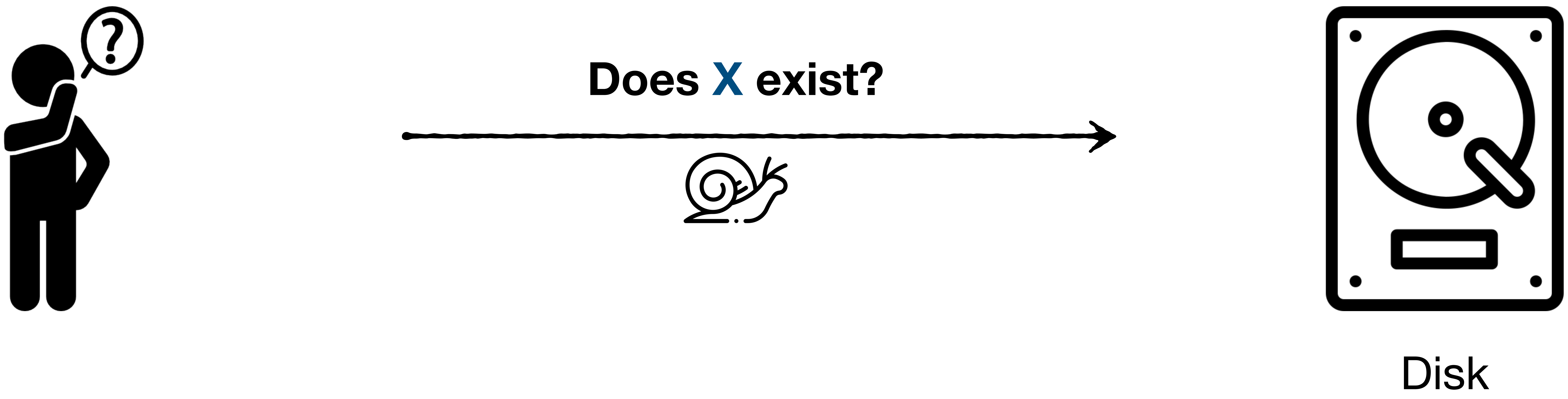
Filter

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

Large

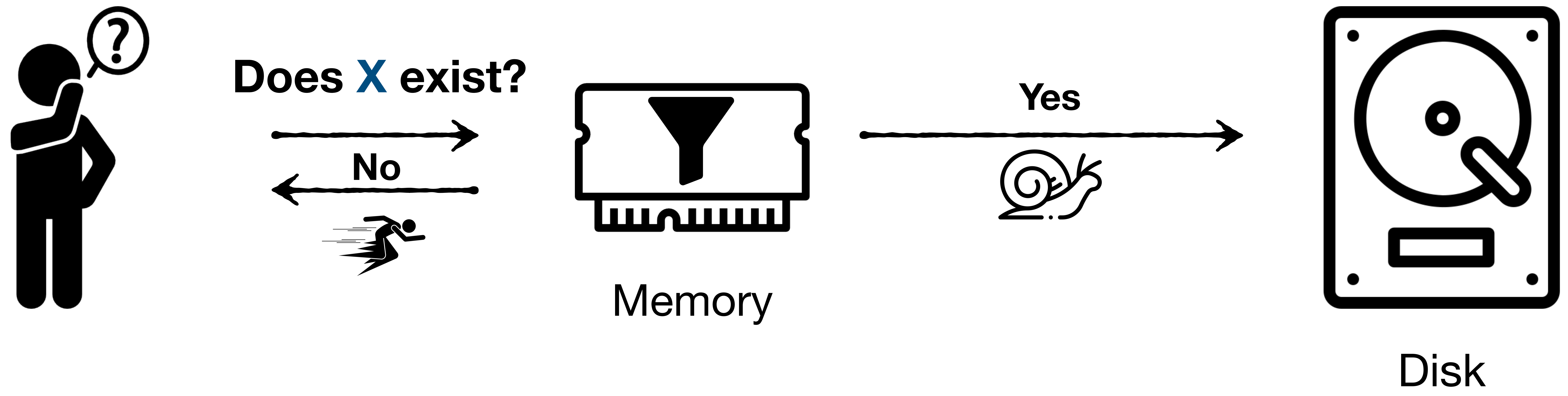
Hash table/Tree

For $\epsilon = 2\,\%$ , filters require ~1 Byte/item. Hash table/Tree can take >8-16 Byte/item.



Does X exist?

Disk



**Saves unnecessary disk accesses and network hops**

# Classic filter: The Bloom filter (BF) [Bloom 70]

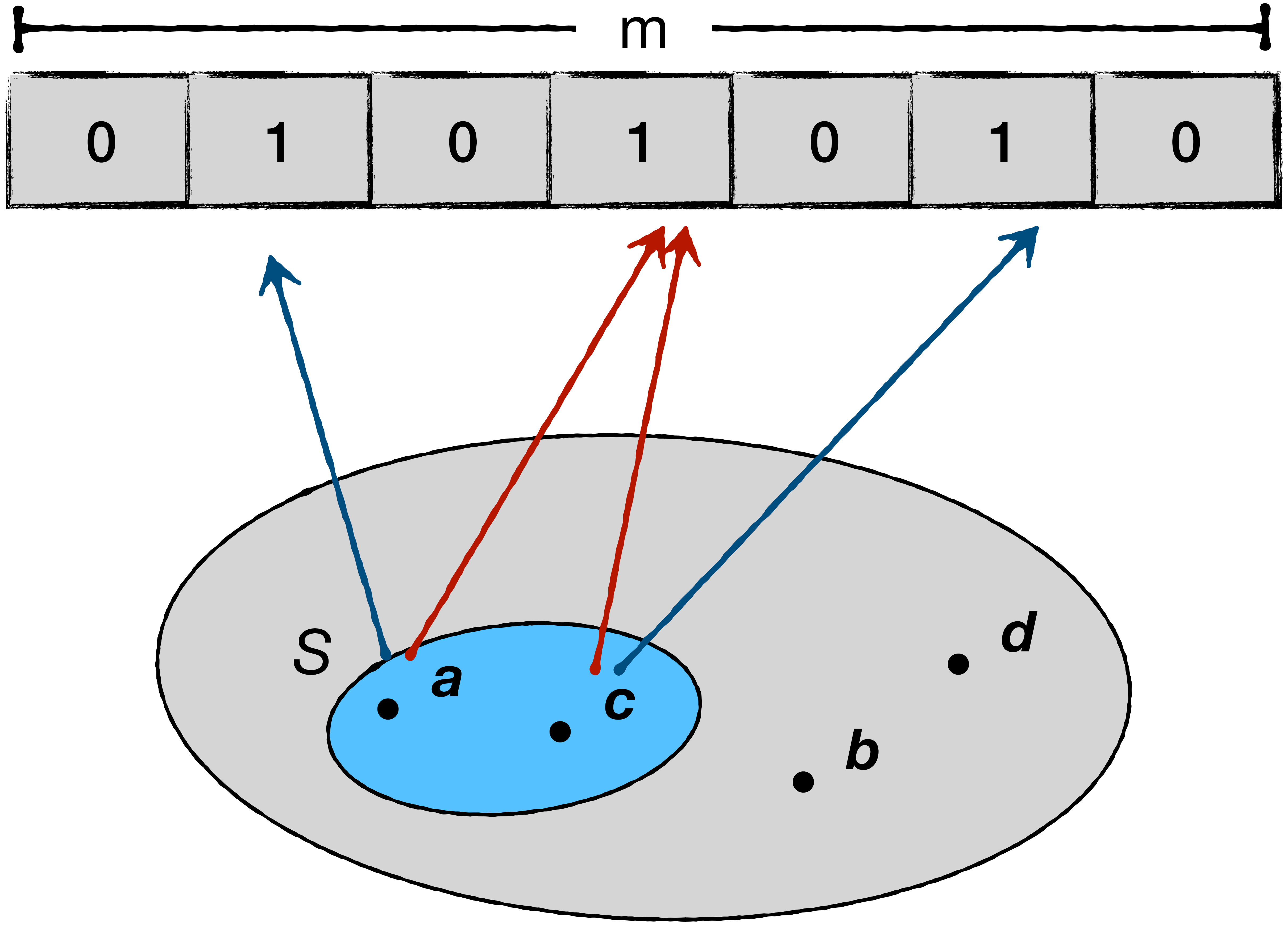
Bloom filter: *m* bit array + *k* hash functions (here *k*=2)

$h_1(a) = 1$

$h_2(a) = 3$

$h_1(c) = 5$

$h_2(c) = 3$

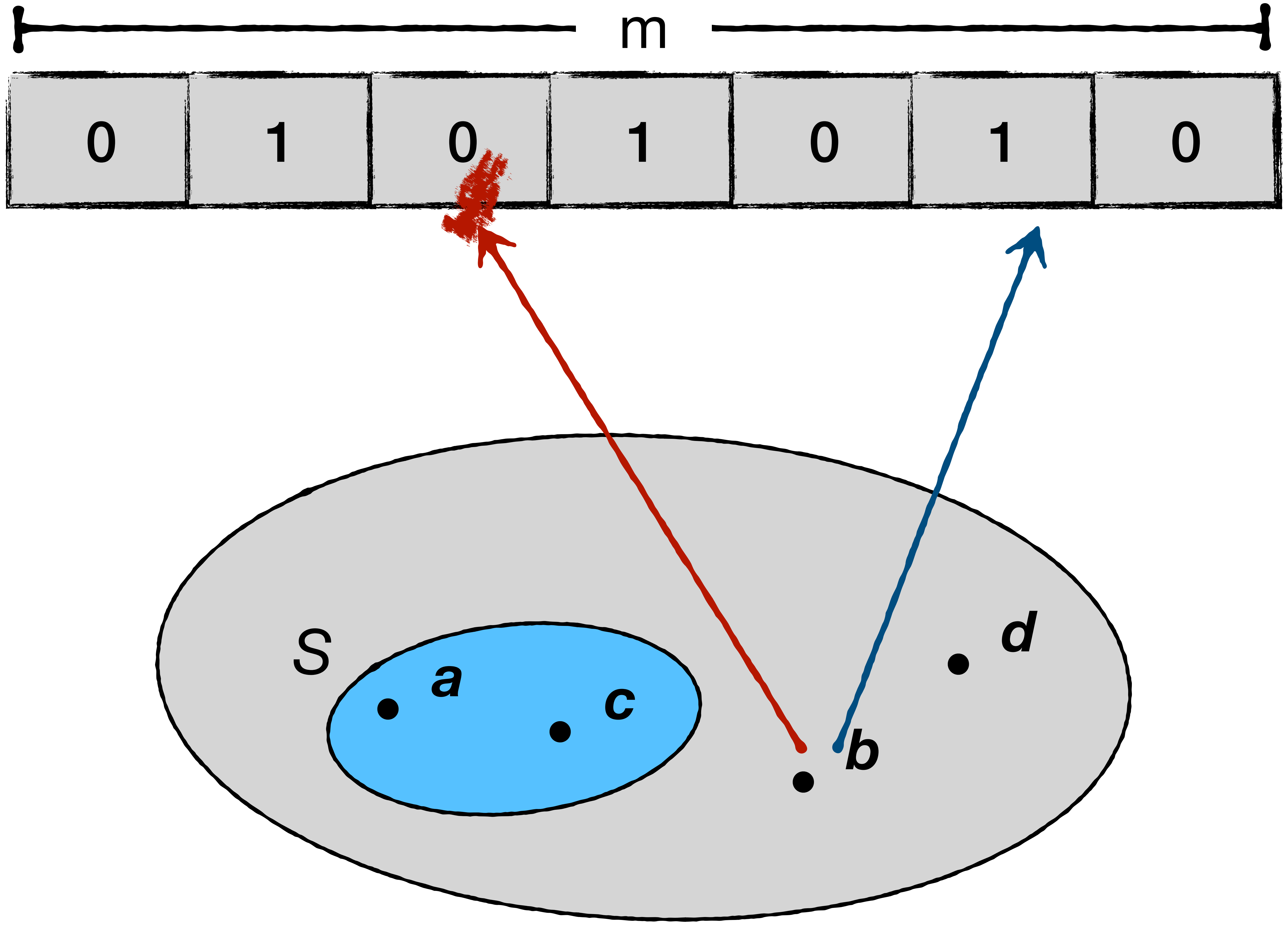


# Classic filter: The Bloom filter (BF) [Bloom 70]

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

$h_1(b) = 5$

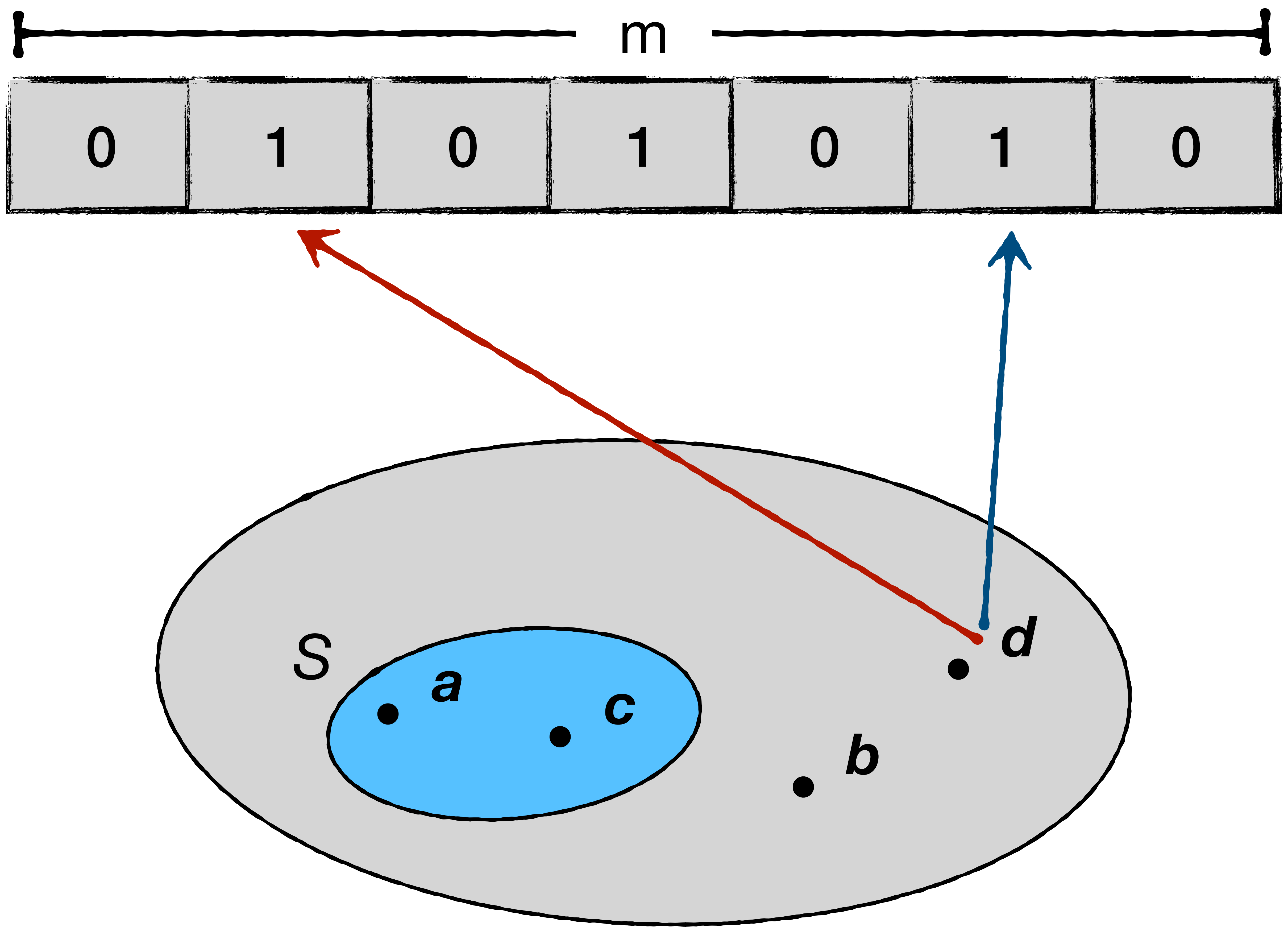
$h_2(b) = 2$



**True negative**

# Classic filter: The Bloom filter (BF) [Bloom 70]

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



$h_1(d) = 5$

$h_2(d) = 1$

**False positive**

# Bloom filters are ubiquitous (> 10K citations)

Computational biology

Databases

Networking

Stream processing

Storage systems

# Bloom filters have suboptimal performance

CFGMW 78: Optimal filter bound

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

# Applications workaround BF limitations

| **Limitation** | **Workaround** |
|---|---|
| No deletes | Rebuild |

# Applications workaround BF limitations

**Limitation**

No resizes

**Workaround**

Guess *N*,
Rebuild if wrong

# Applications workaround BF limitations

| **Limitation** | **Workaround** |
|---|---|
| No merging or enumeration | ??? |

# Applications workaround BF limitations

**Limitation**

No values
associated with keys

**Workaround**

Combine with other
data structures

# Bloom filters have several limitations

**No Deletes**

**No Resize**

**No Merging/ Enumeration**

**No value association**

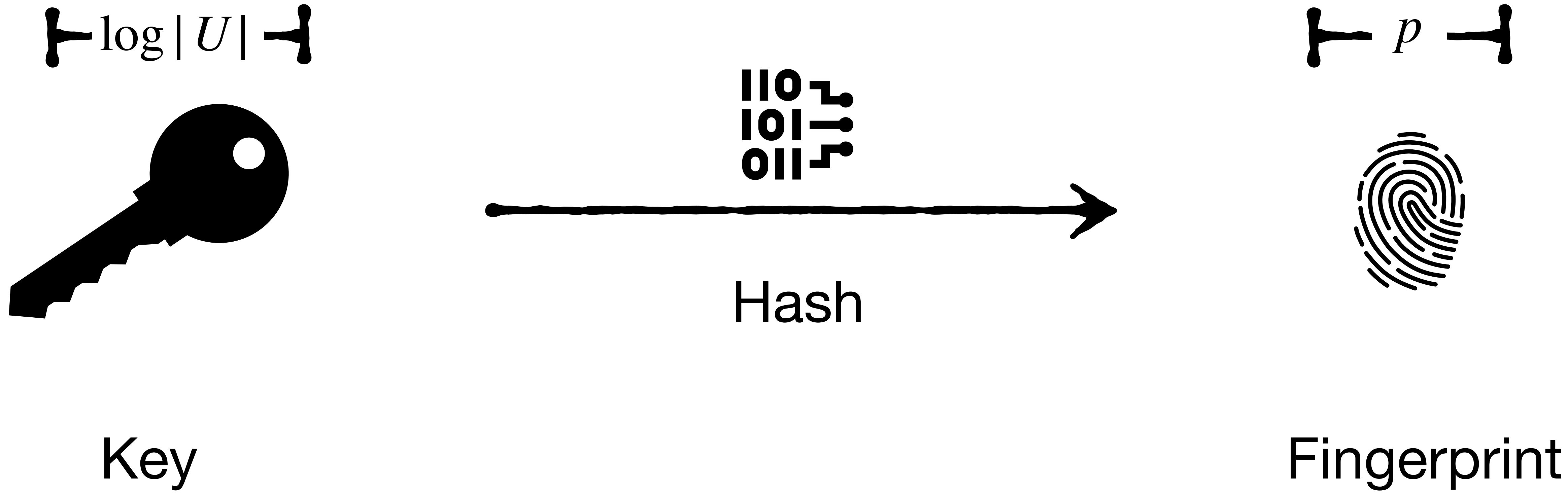
**No Counting**

**Poor cache locality**

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

# Fingerprinting is an alternative to Bloom filters

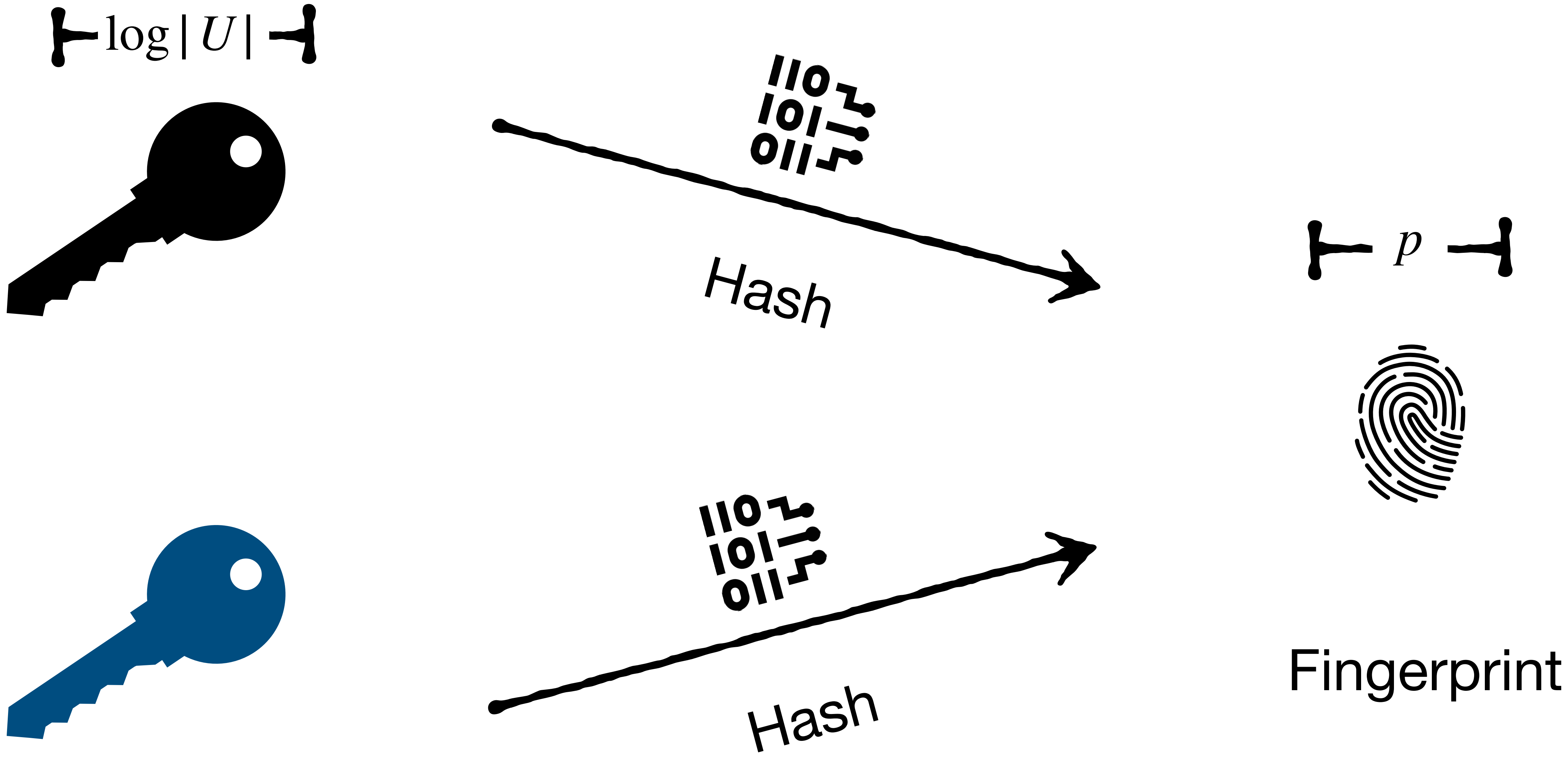
PPR05, DM09, BFJ+12, EF16, PBJ+17



**Store fingerprints compactly in a table**

# Fingerprinting is an alternative to Bloom filters

PPR05, DM09, BFJ+12, EF16, PBJ+17

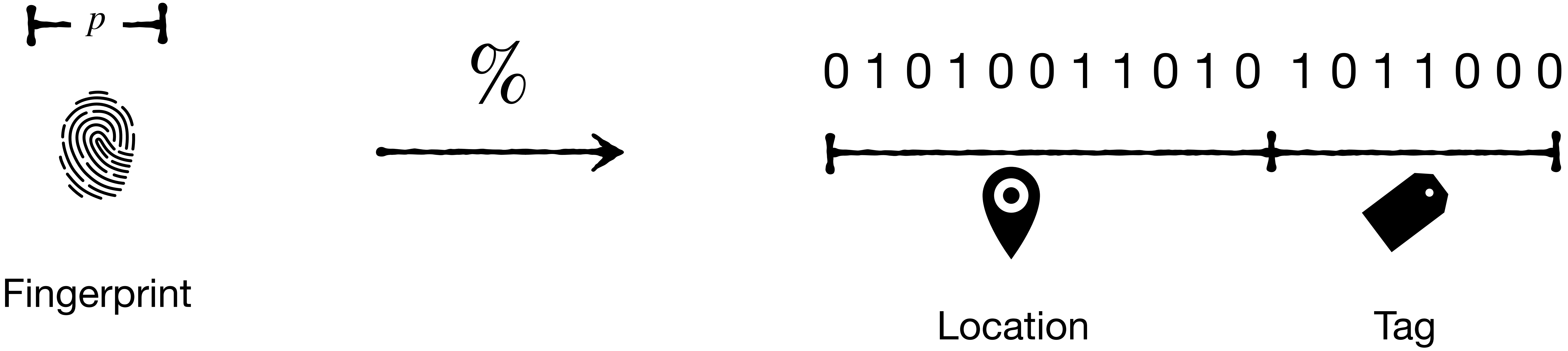


**False positives occur only when fingerprints collide**

$$\textbf{Pr [collision]} = \frac{1}{2^p}$$

# Storing fingerprints compactly using quotienting

Knuth 97



$p$

Fingerprint

$\xrightarrow{\%}$

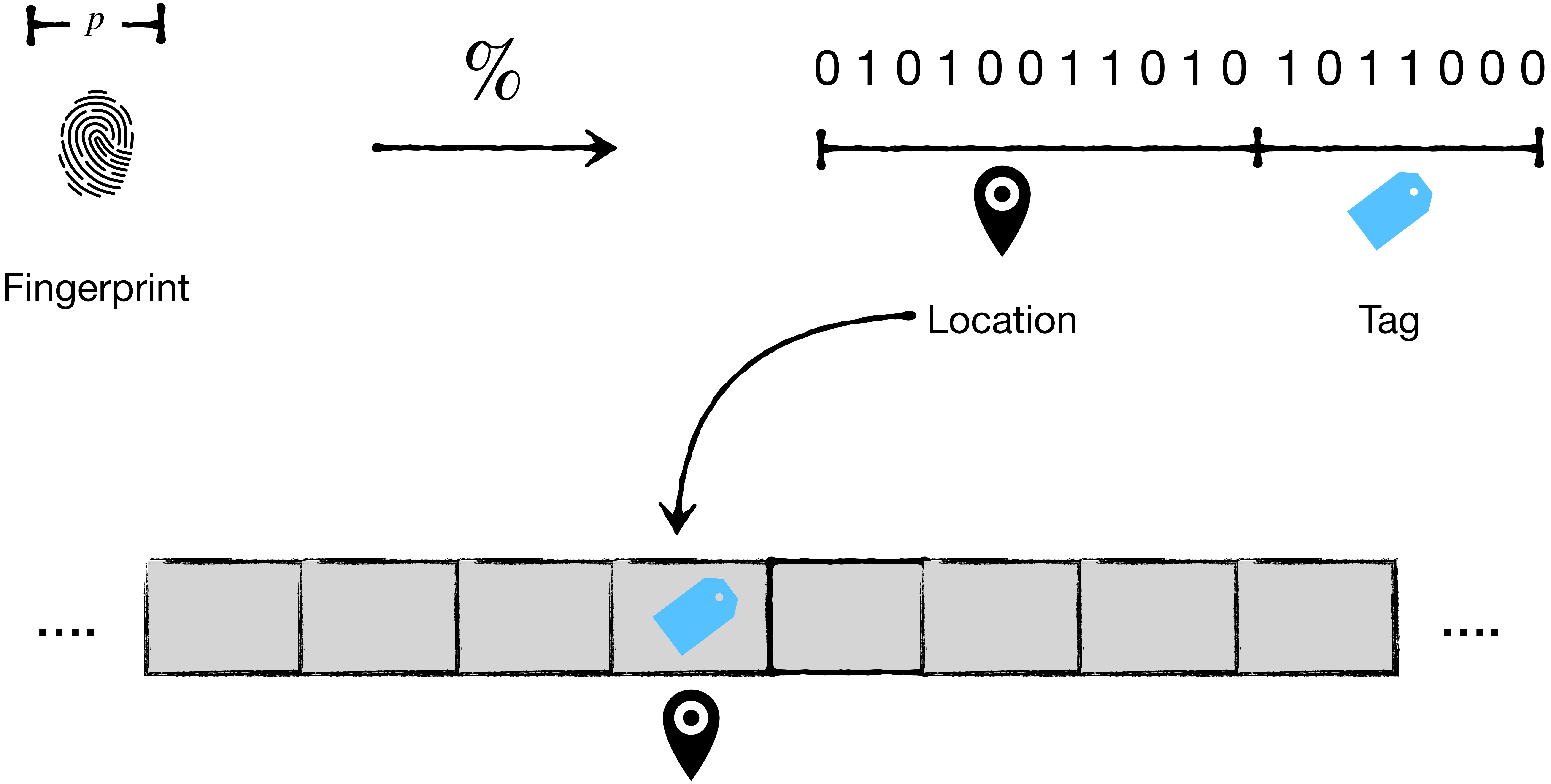
0 1 0 1 0 0 1 1 0 1 0 1 0 1 1 0 0 0

Location

Tag

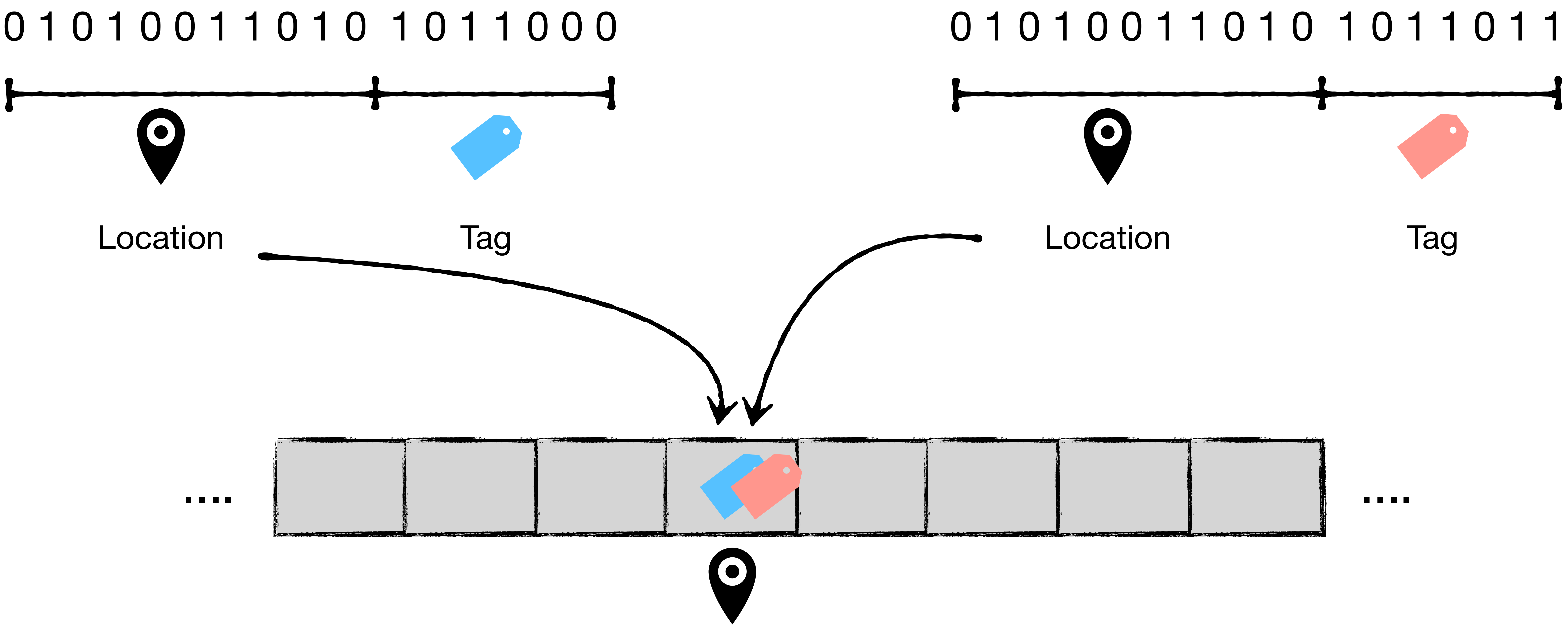
# Storing fingerprints compactly using quotienting

Knuth 97



# Storing fingerprints compactly using quotienting

Knuth 97



01010011010 1011000

Location

Tag

01010011010 1011011

Location

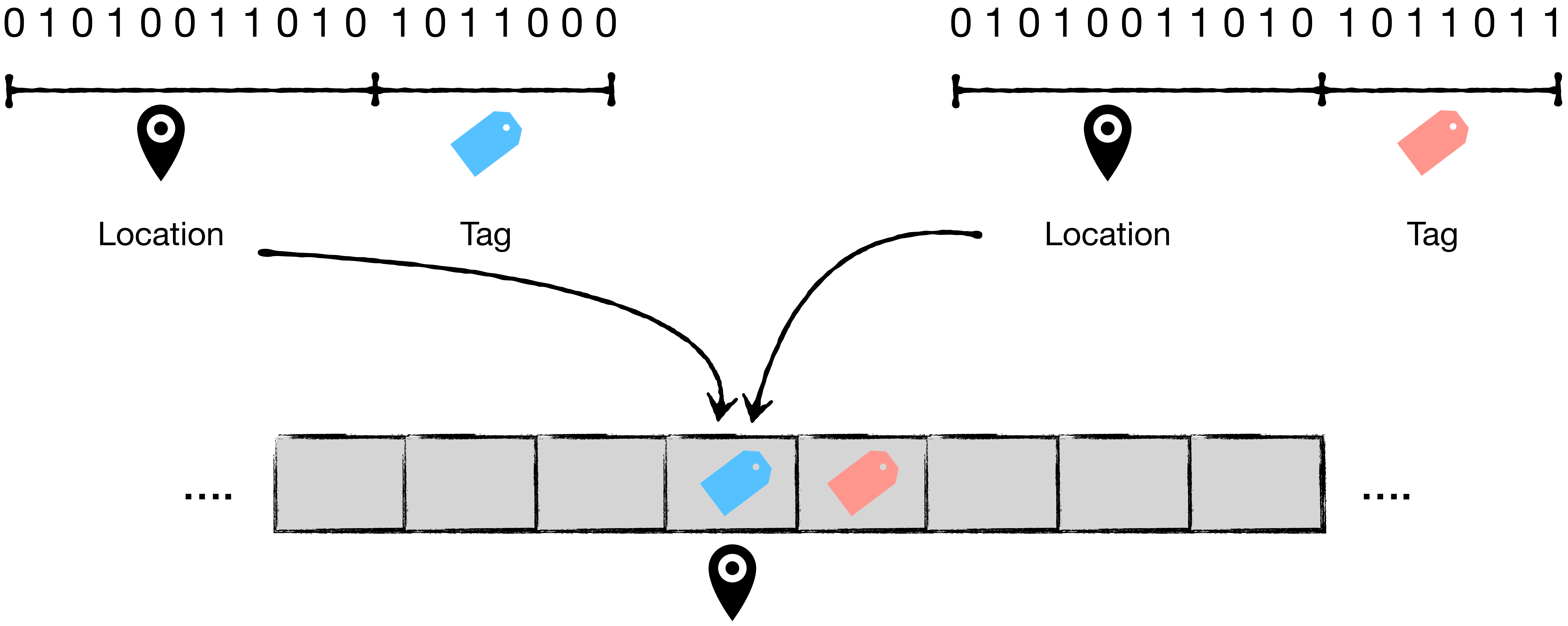
Tag

....

....

# Storing fingerprints compactly using quotienting

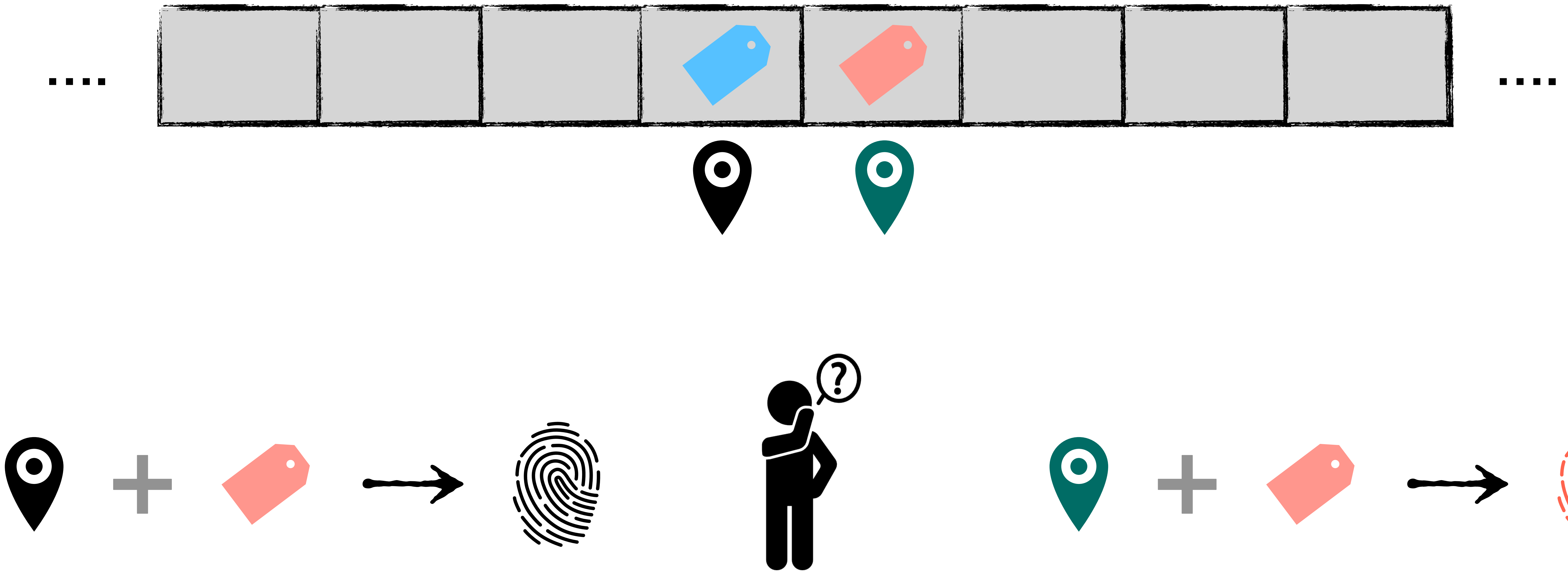
Knuth 97



**Use linear probing and Robinhood hashing**

# Resolving collisions in quotient filter (QF)

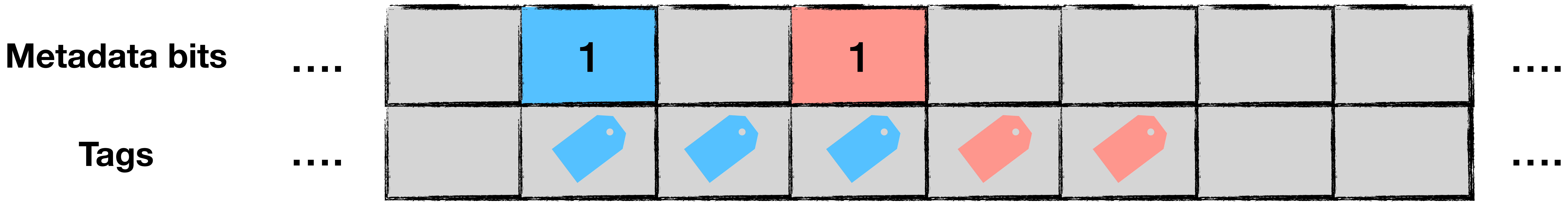
Pandey et al. SIGMOD 17



**How to identify the home slot of a given tag?**

# Resolving collisions in quotient filter (QF)

Pandey et al. SIGMOD 17



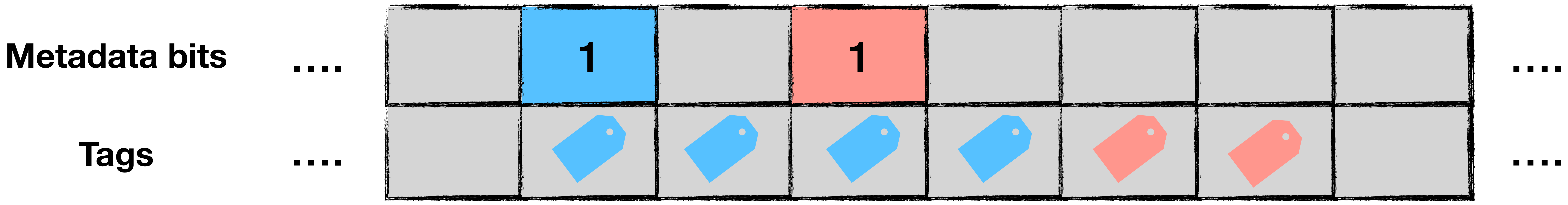
Metadata bits .... 1 1 ....

Tags .... ....

**Use two metadata bits/slot to group tags by their home slot**

# Resolving collisions in quotient filter (QF)

Pandey et al. SIGMOD 17



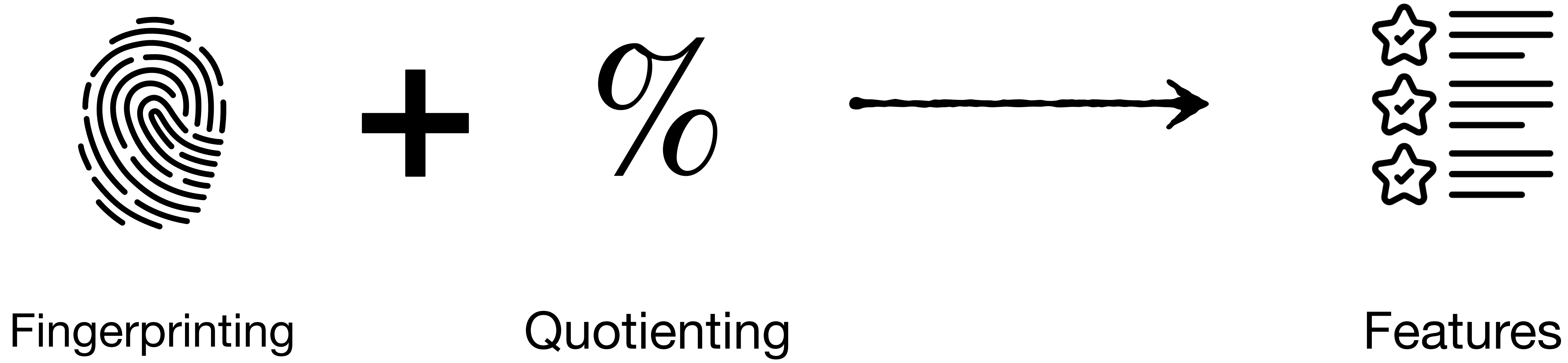
**Metadata bits help identify the home slot of each tag**

# Quotient filters offer better performance than BF

CFGMW 78: Optimal filter bound

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

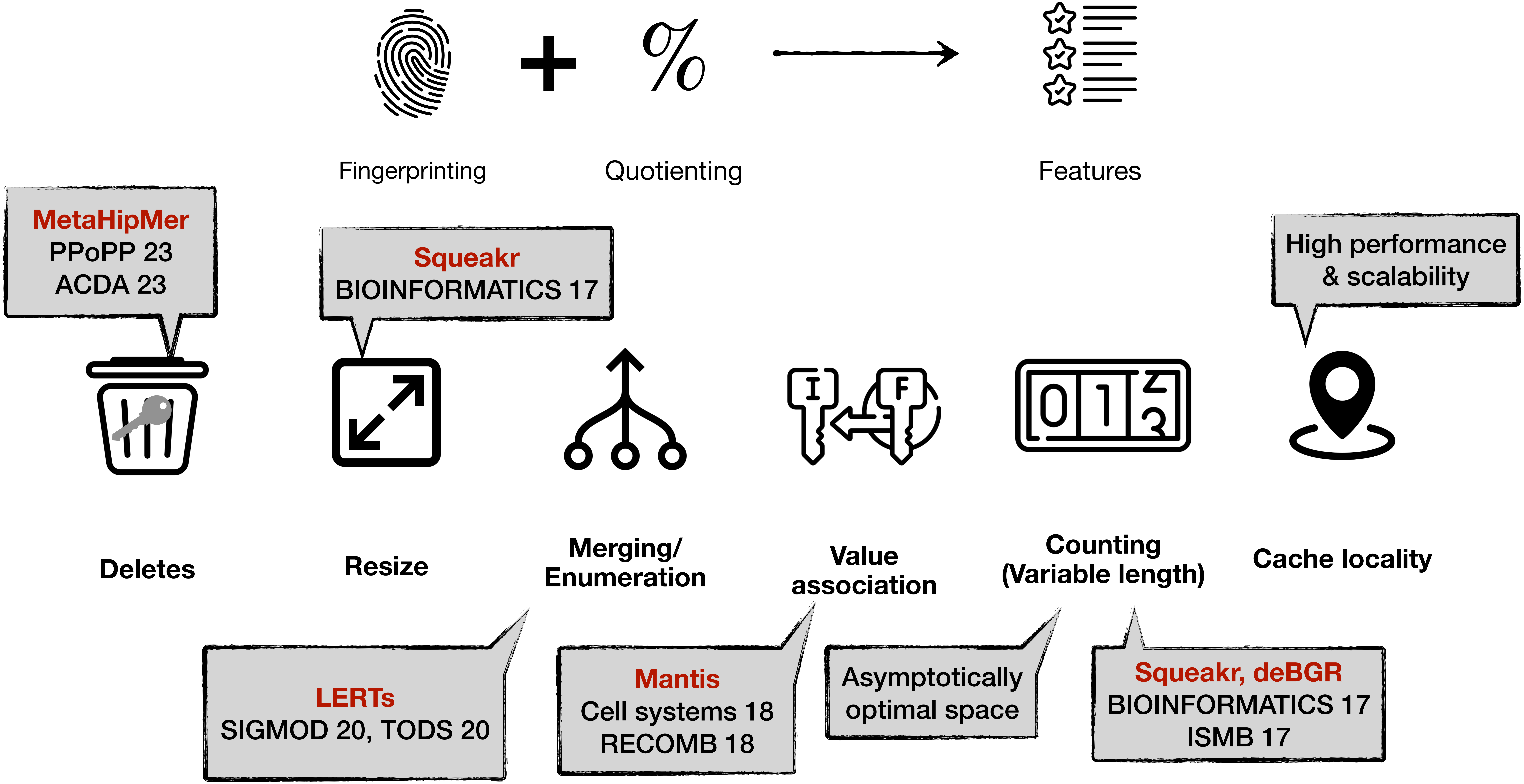
**Quotient filters have theoretical advantages over Bloom filters**



Fingerprinting

Quotienting

Features



Fingerprinting + Quotienting → Features

MetaHipMer
PPoPP 23
ACDA 23

Deletes

LERTs
SIGMOD 20, TODS 20

Squeakr
BIOINFORMATICS 17

Resize

Merging/
Enumeration

Mantis
Cell systems 18
RECOMB 18

Value
association

Asymptotically
optimal space

Counting
(Variable length)

Squeakr, deBGR
BIOINFORMATICS 17
ISMB 17

High performance
& scalability

Cache locality

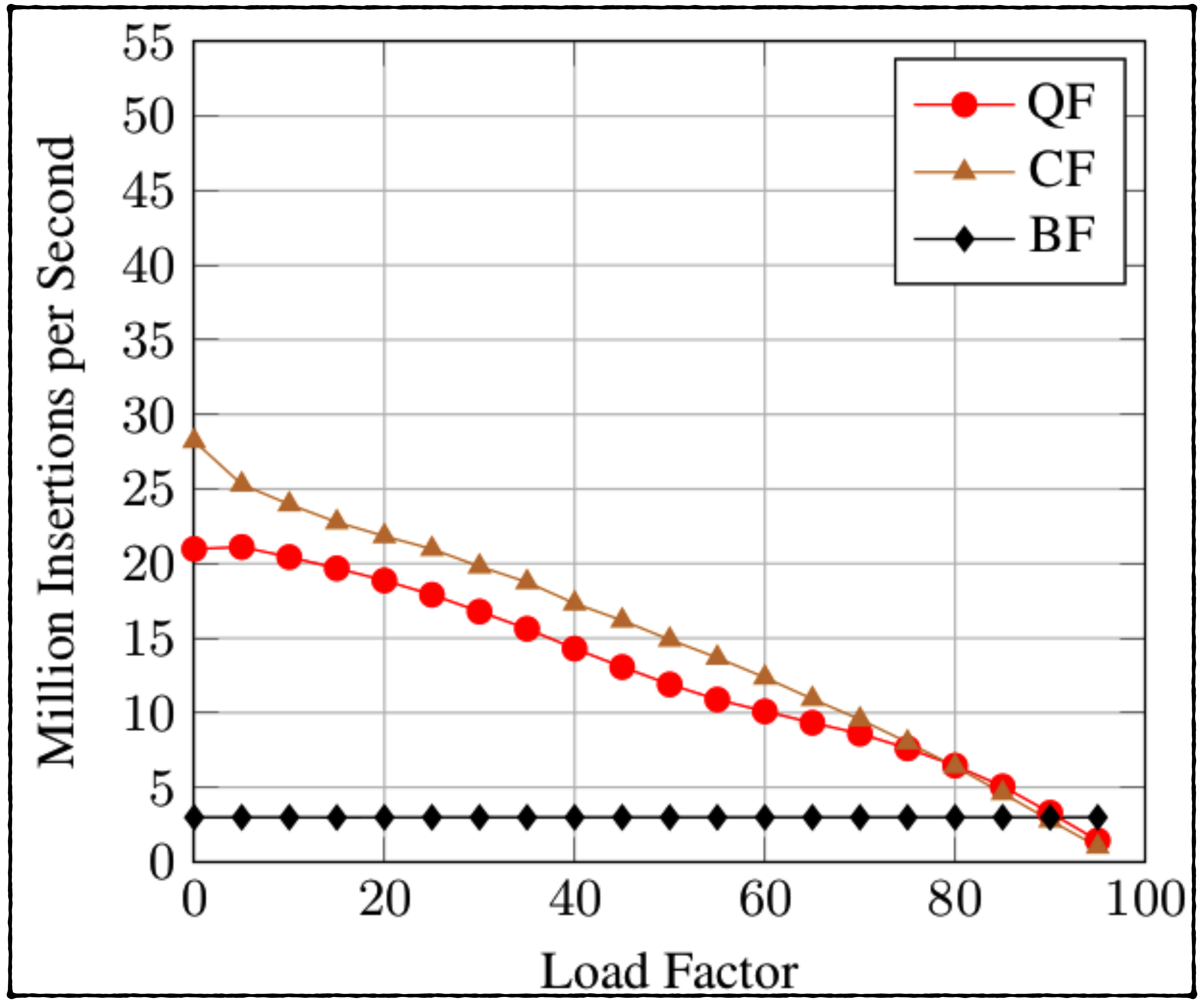
# Quotient filters empirical performance

QF: quotient filter

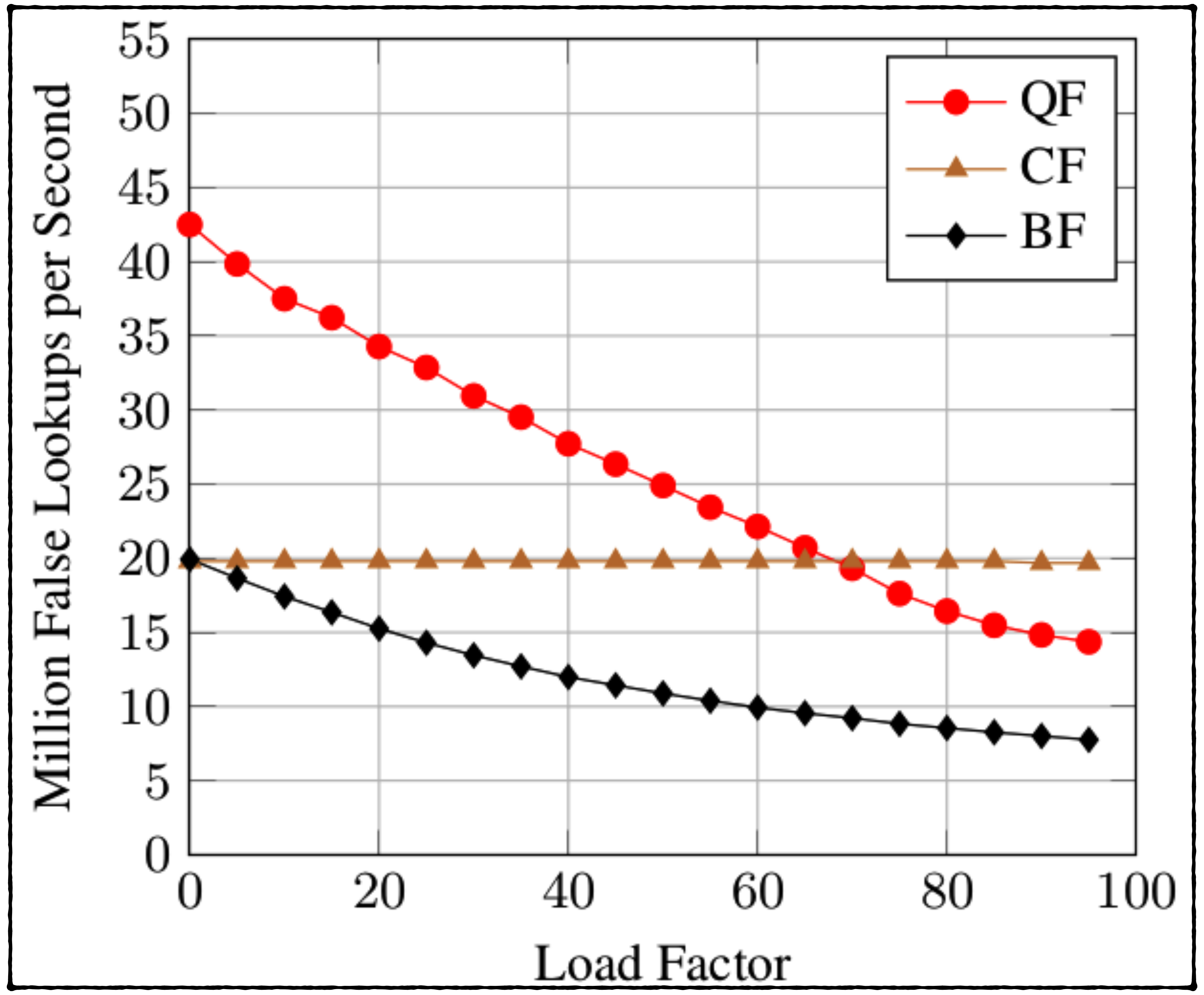
CF*: cuckoo filter [FAK+14]

BF*: Bloom filter

Inserts



Queries

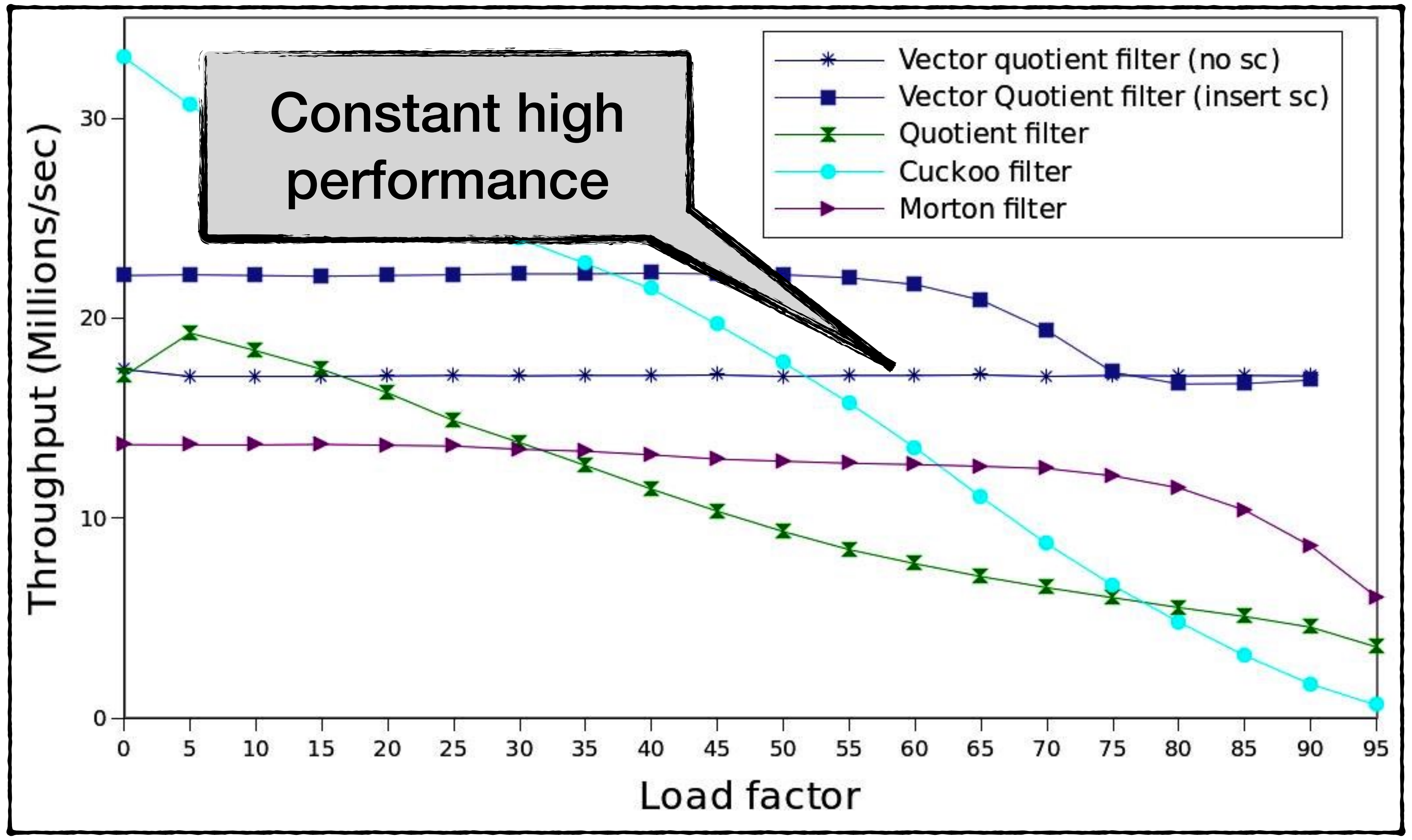


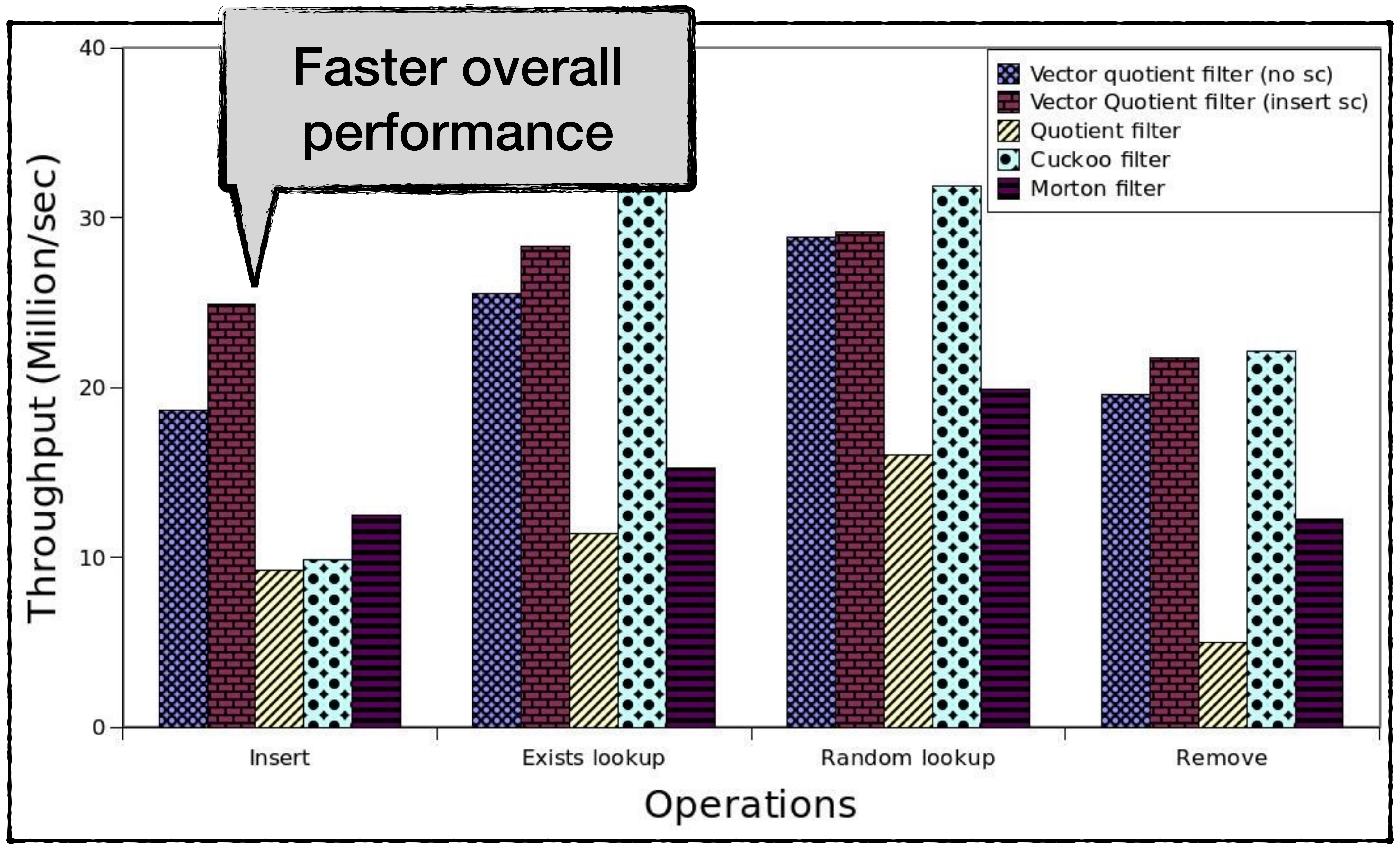
Insert performance is similar to the state-of-the-art non-counting filters

Query performance is significantly fast at low load factors and slightly slower at high load factors

# Vector quotient filters [SIGMOD 21]

Pandey et al. SIGMOD 21





Combining hashing techniques (Robinhood hashing + power of 2-choice hashing)

Using ultra-wide vector instructions (AVX-512)

# Quotient filter’s impact in computer science

Databases

Data security

Stream analysis

Storage systems

Sequence search

Genome assembly

Graph systems

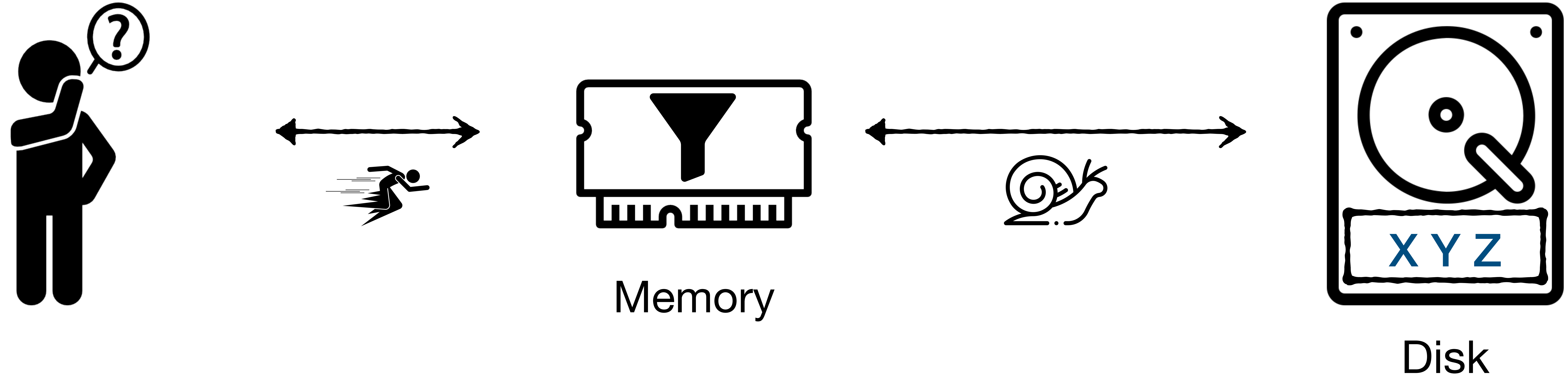
GPU data structure

# Takeaways

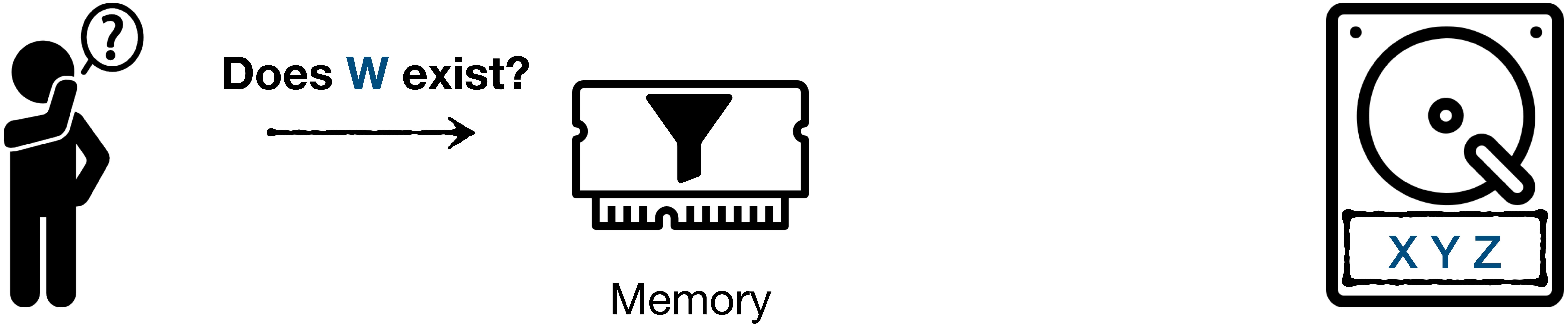
- Fingerprinting is powerful: provides deletions, enumerability, merging
- Quotienting complements fingerprinting: provides high cache locality, performance and compactness
- Quotient filter is a high-performance feature-full filter.

# Adaptivity

# Skewed workloads can make filters obsolete



# Skewed workloads can make filters obsolete



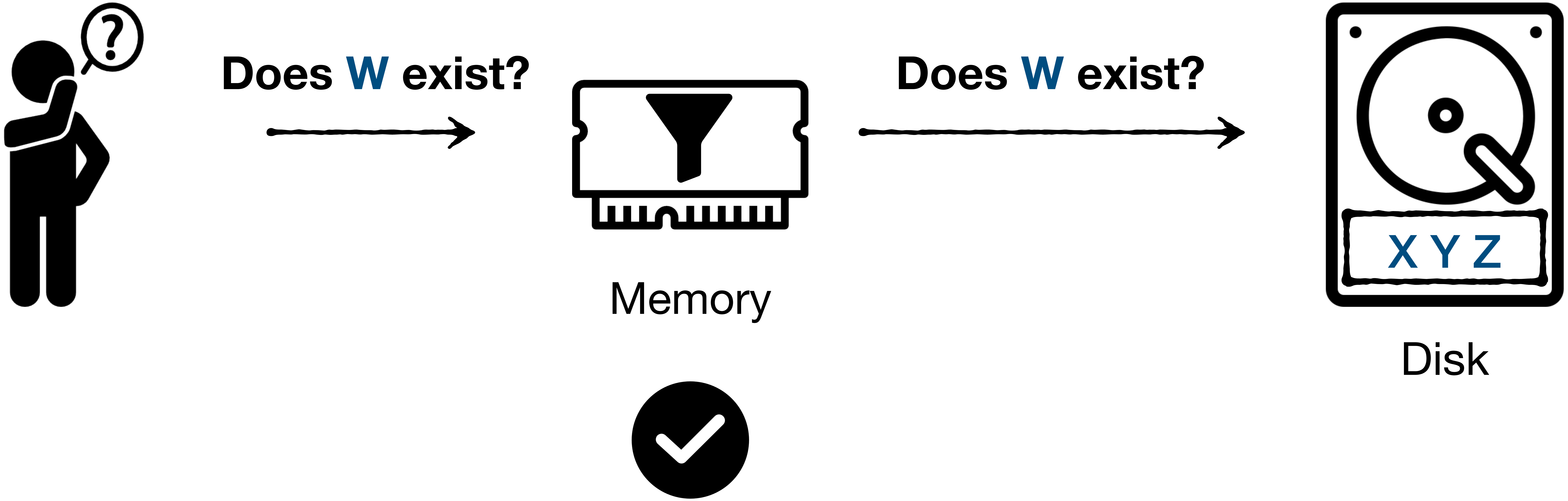
**Does W exist?**

Memory

X Y Z

Disk

# Skewed workloads can make filters obsolete



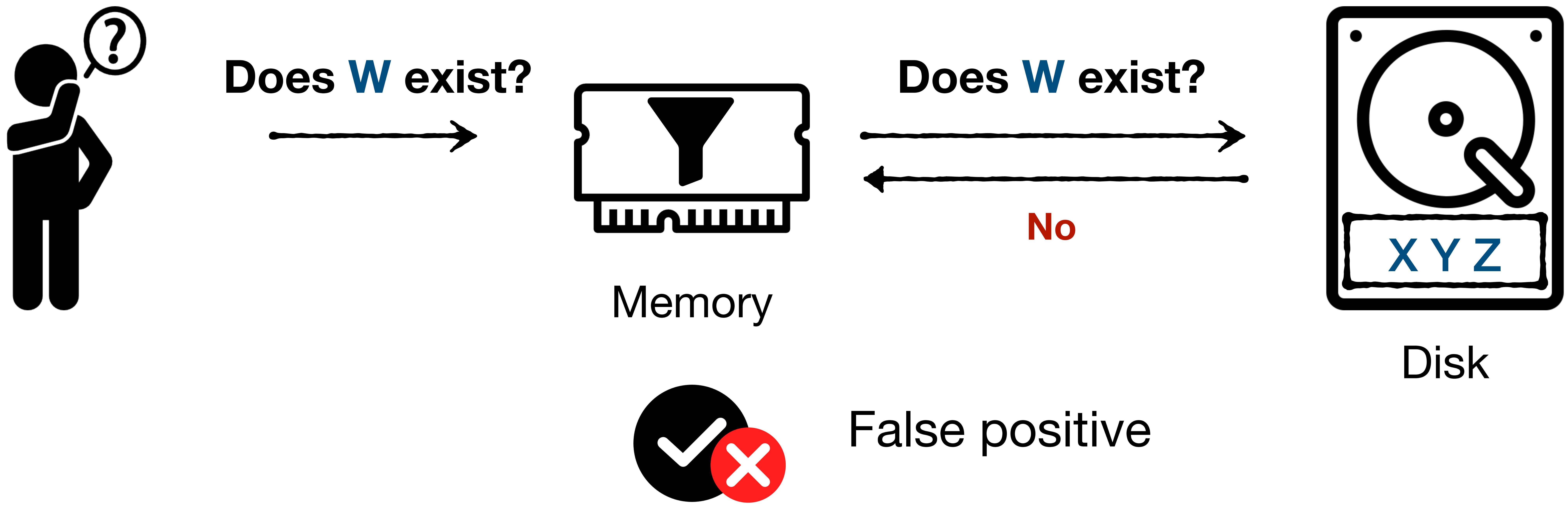
**Does W exist?**

**Does W exist?**

Memory

Disk

# Skewed workloads can make filters obsolete



Does **W** exist?

Memory

Does **W** exist?

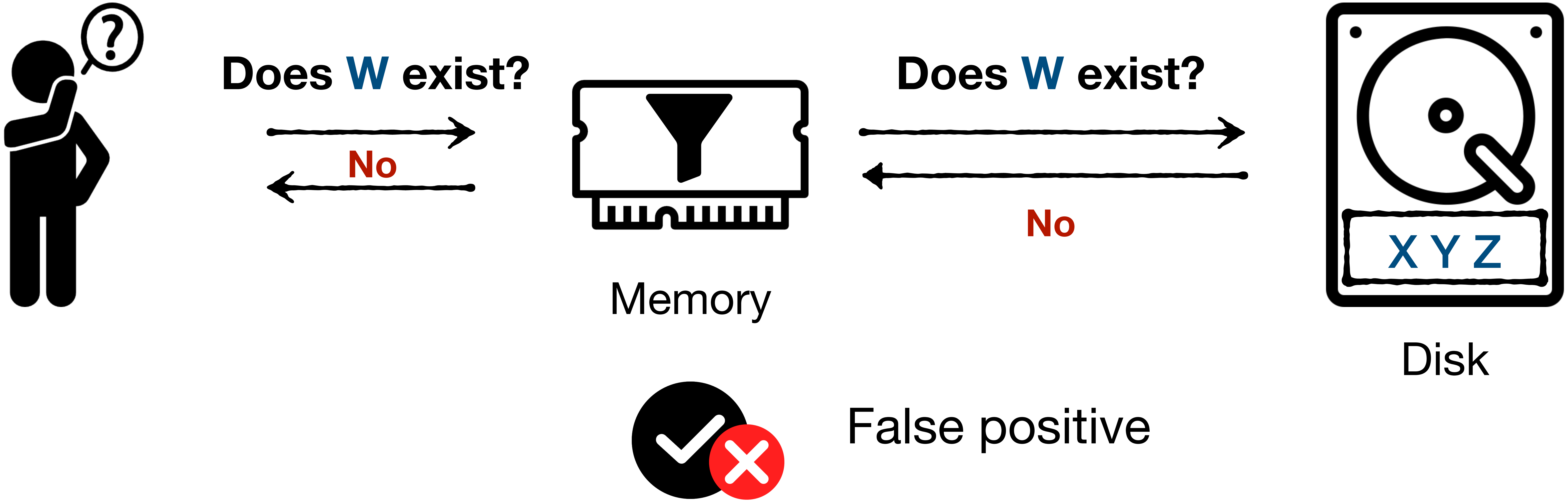
**No**

XYZ

Disk

False positive

# Skewed workloads can make filters obsolete



Does W exist?

No

Memory

Does W exist?

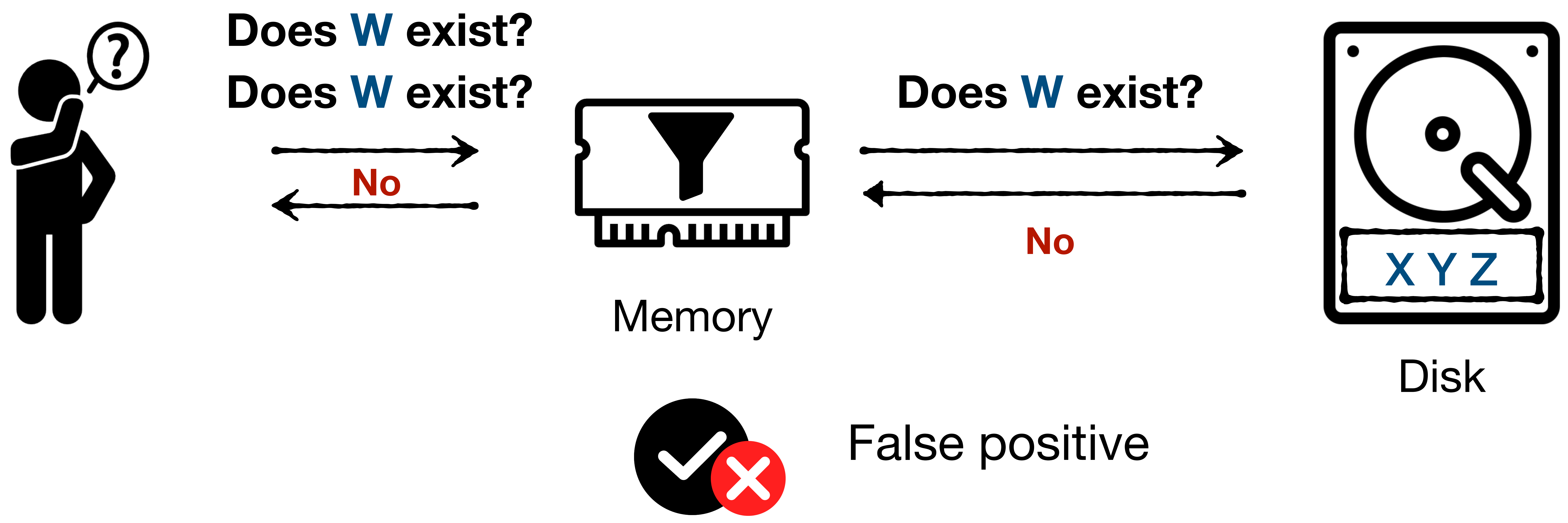
No

X Y Z

Disk

False positive

# Skewed workloads can make filters obsolete



Does W exist?

Does W exist?

No

Memory

Does W exist?

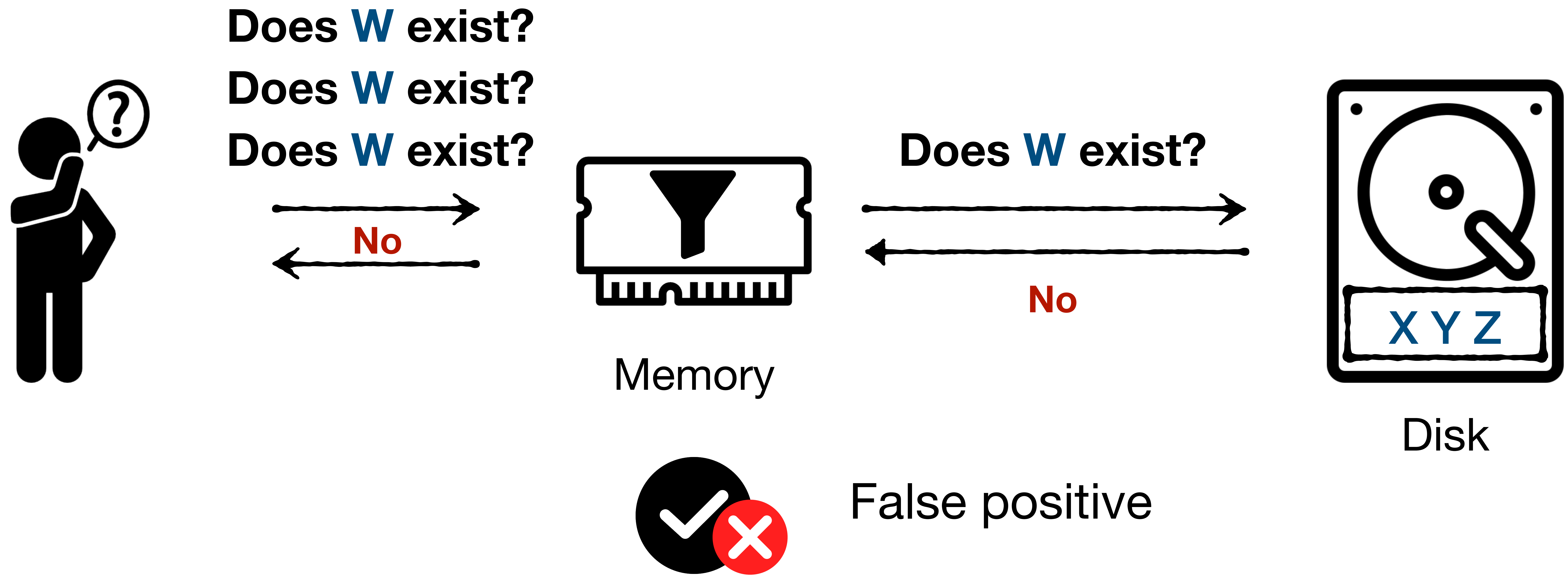
No

X Y Z

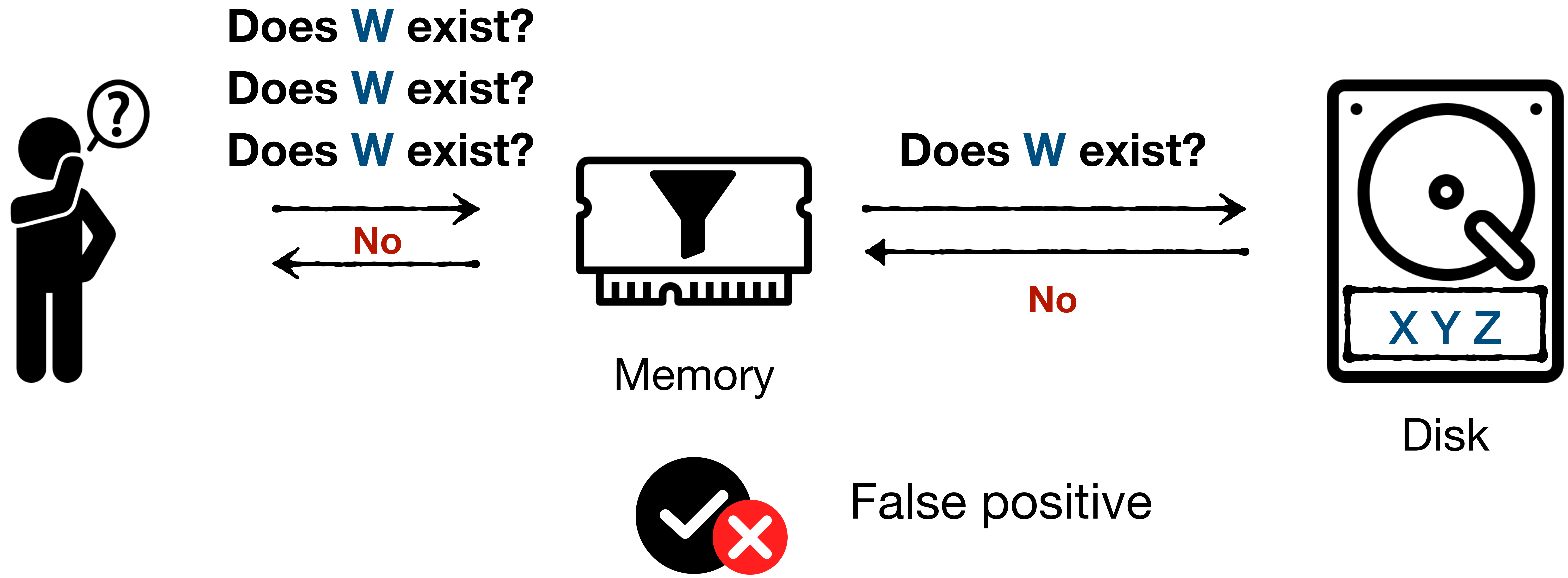
Disk

False positive

# Skewed workloads can make filters obsolete



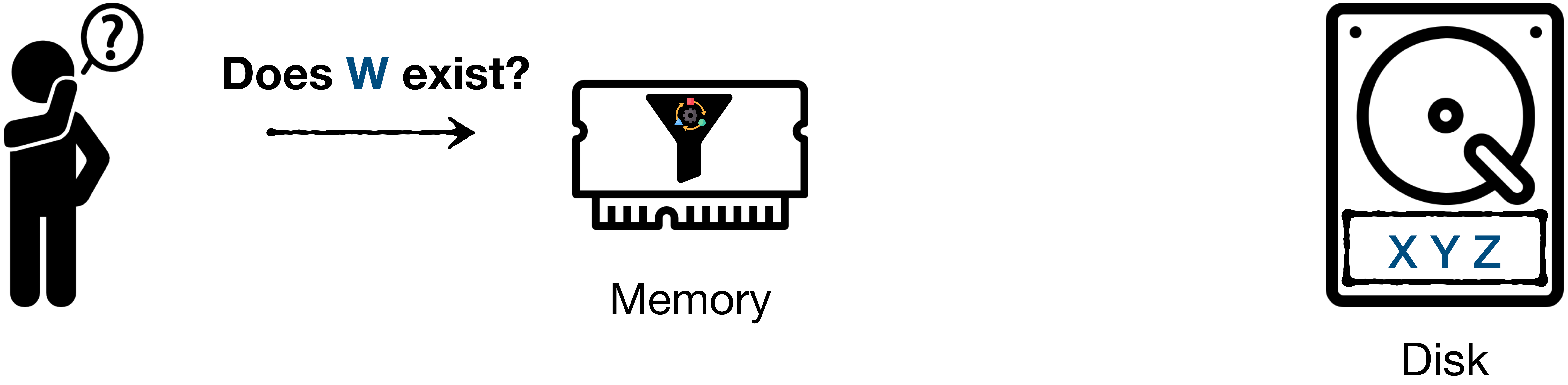
# Skewed workloads can make filters obsolete



**False-positive rate $\leq \epsilon$, only for a single query**

Can we learn from the feedback?

# Adaptive filters change their state upon feedback



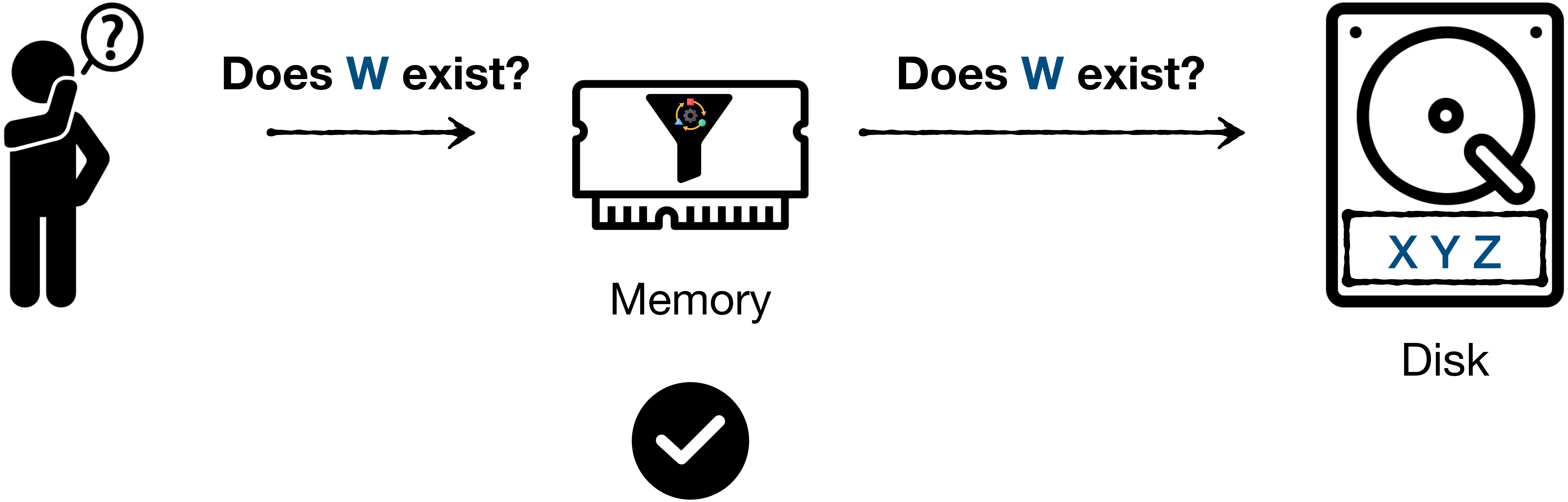
Does **W** exist?

Memory

XYZ

Disk

# Adaptive filters change their state upon feedback



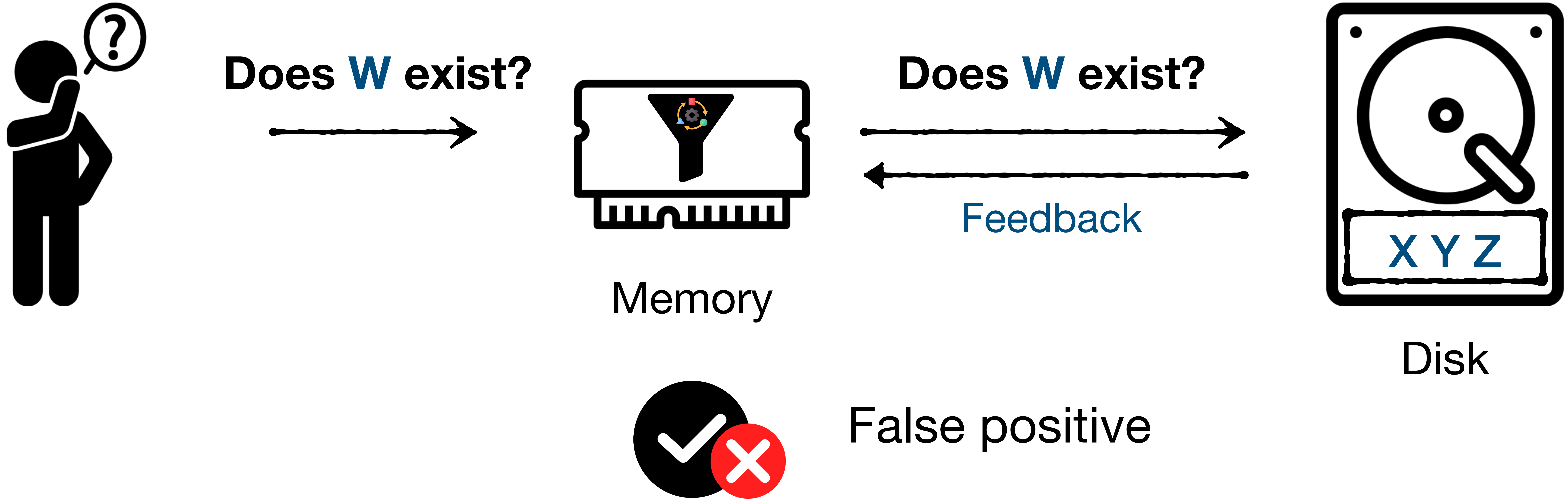
**Does W exist?**

Memory

**Does W exist?**

Disk

# Adaptive filters change their state upon feedback



**Does W exist?**

Memory

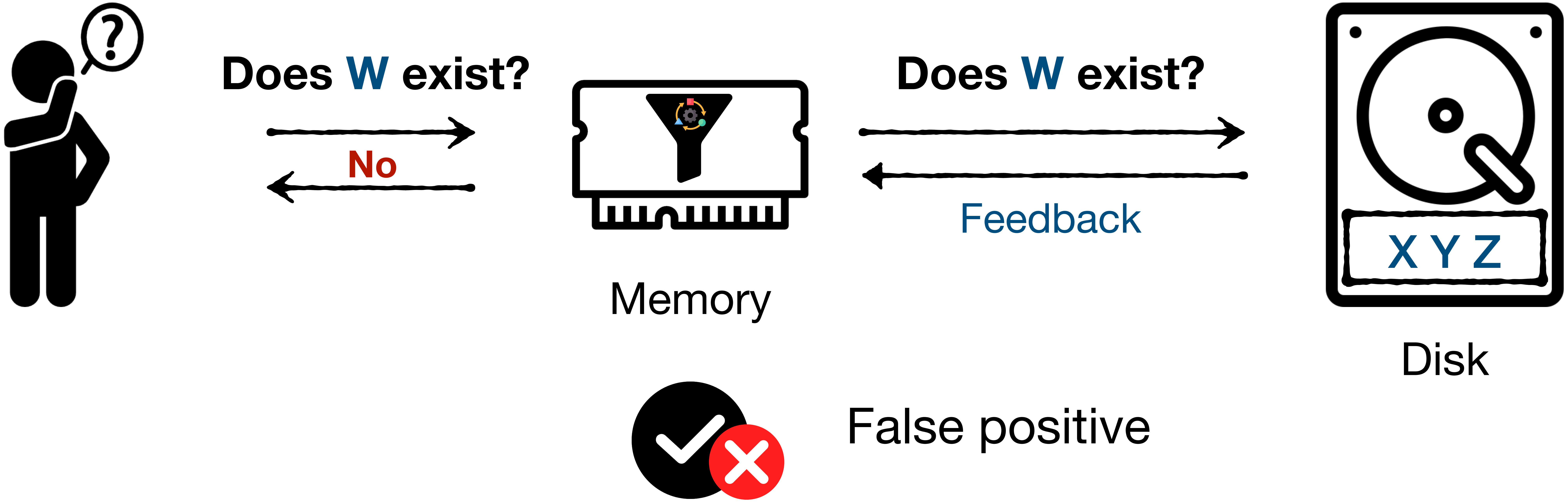
**Does W exist?**

Feedback

Disk

False positive

# Adaptive filters change their state upon feedback



Does **W** exist?

No

Does **W** exist?

Feedback

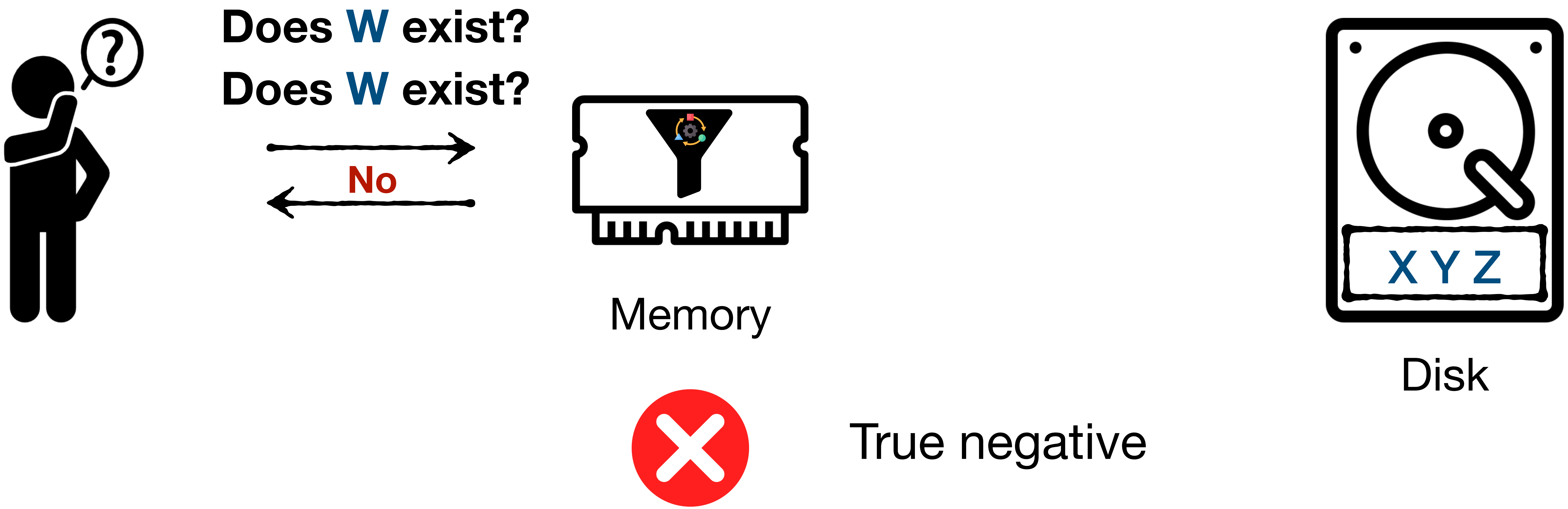
Memory

X Y Z

Disk

False positive

# Adaptive filters change their state upon feedback



**Does W exist?**

**Does W exist?**

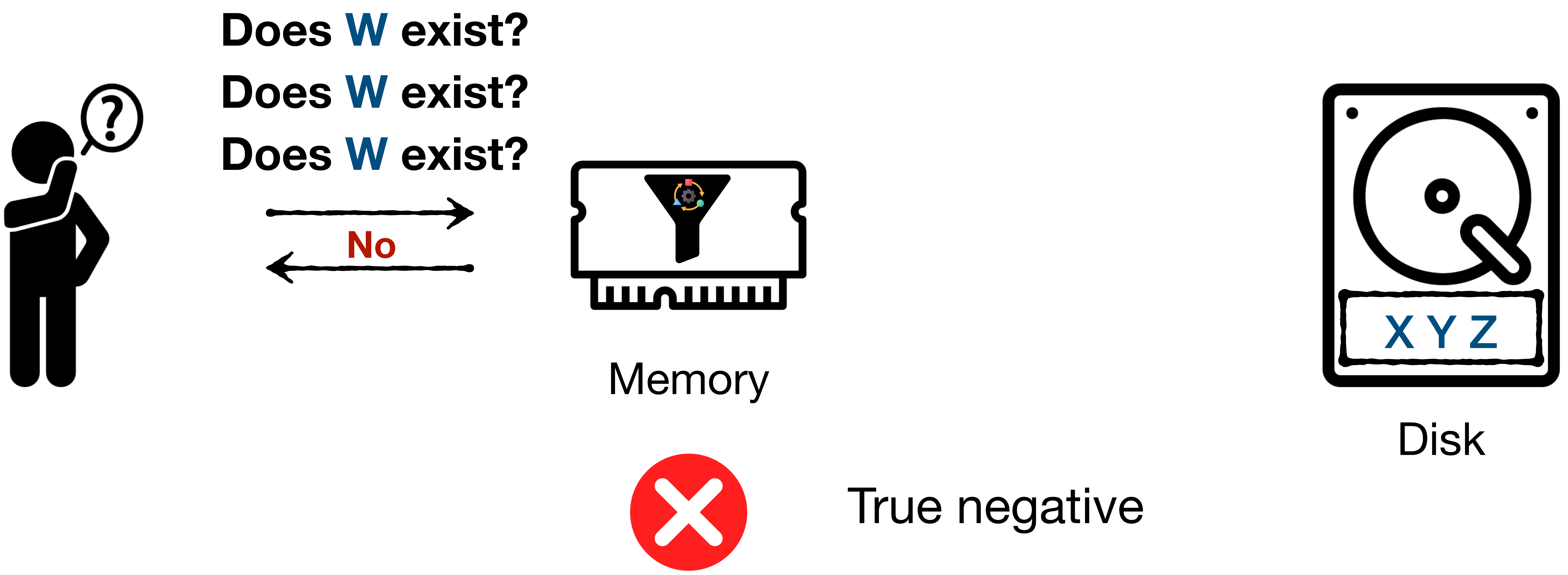
**No**

Memory

Disk

True negative

# Adaptive filters change their state upon feedback



Does **W** exist?

Does **W** exist?

Does **W** exist?

No

Memory

X Y Z

Disk

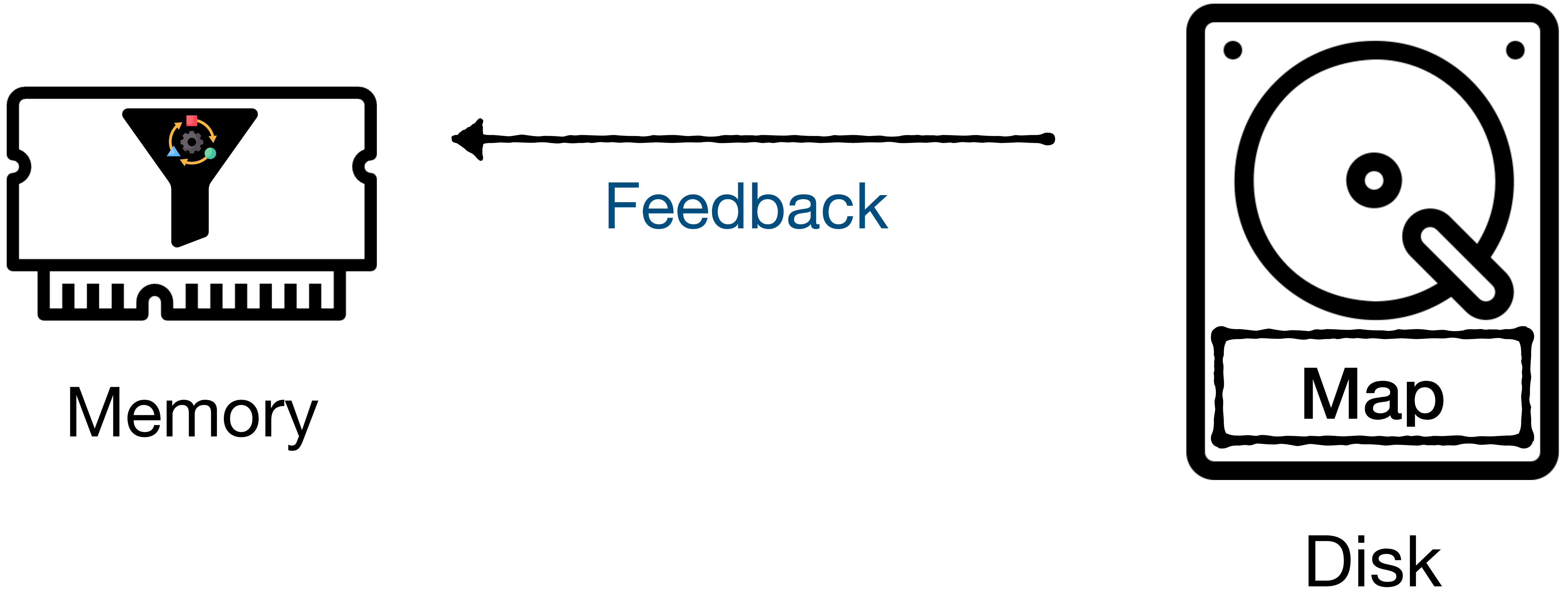
✗ True negative

# Adaptive filters [BFG+ 2018]

An adaptive filter modifies its state upon feedback and produces close to $O(\epsilon n)$ false positives for any sequence of $n$ queries

**False-positive rate $\leq \epsilon$, independent of the query distribution**

# Adaptive filter design has two parts [BFG+ 2018]



Memory

Feedback

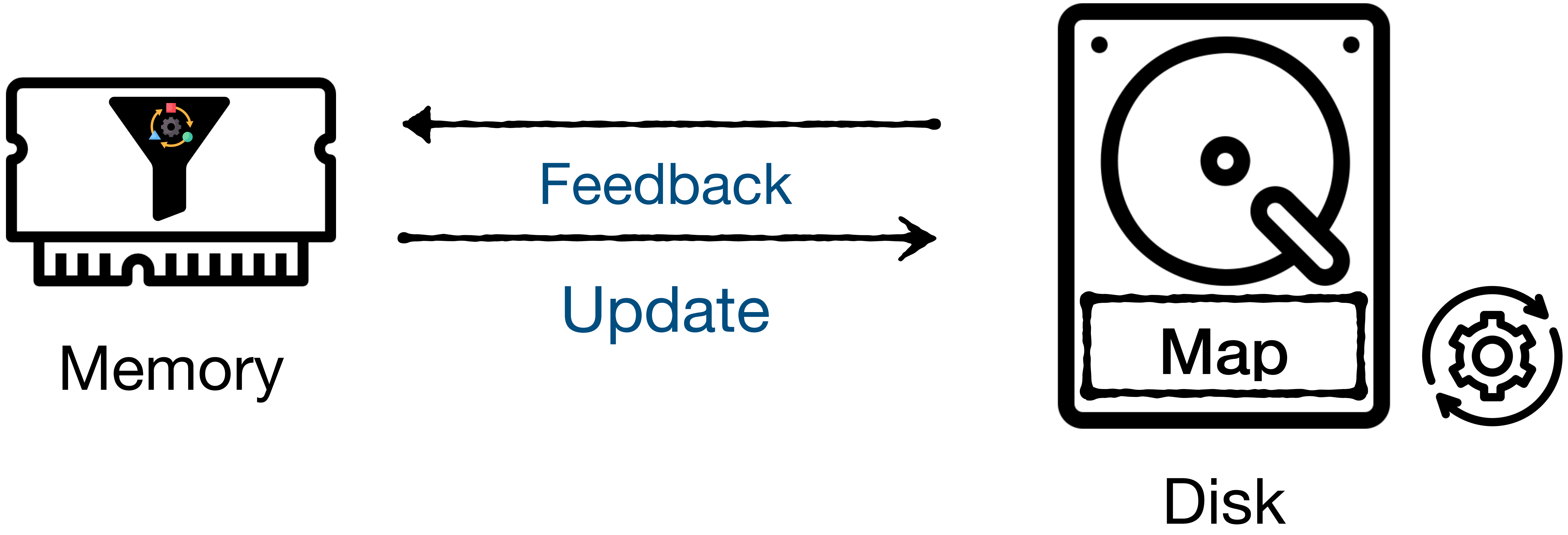
Map

Disk

Small in-memory filter
accessed on every query

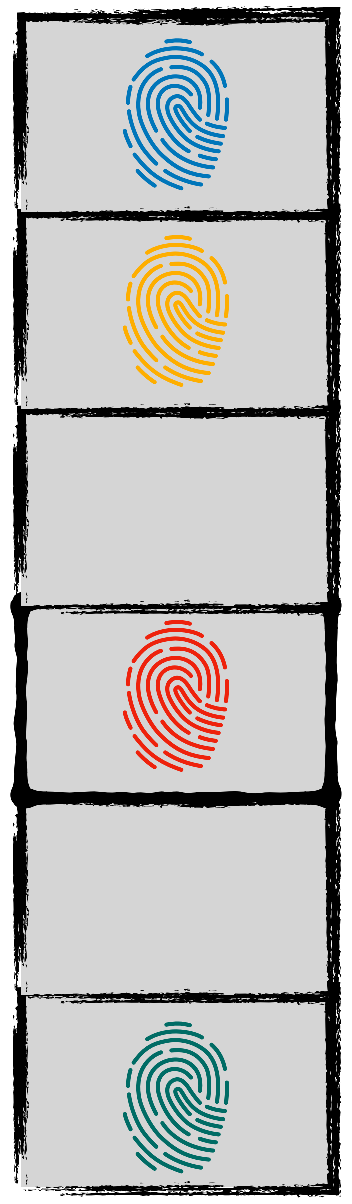
Large disk-resident map
accessed during adaptations

# Adaptive filter design has two parts [BFG+ 2018]

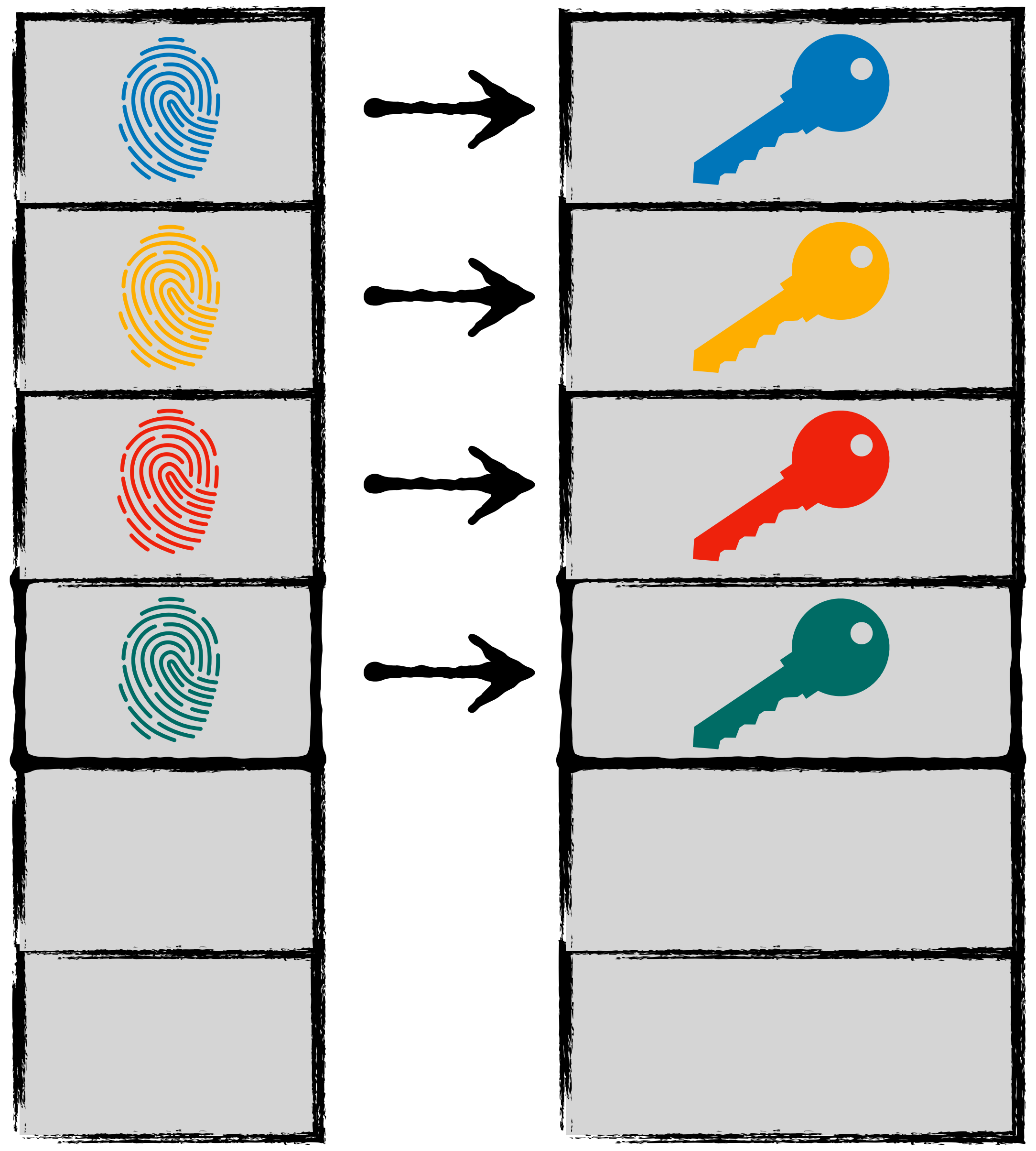


On-disk map enables adaptations and is updated to fix fingerprint collisions

# Adaptive filters employ variable-length fingerprints

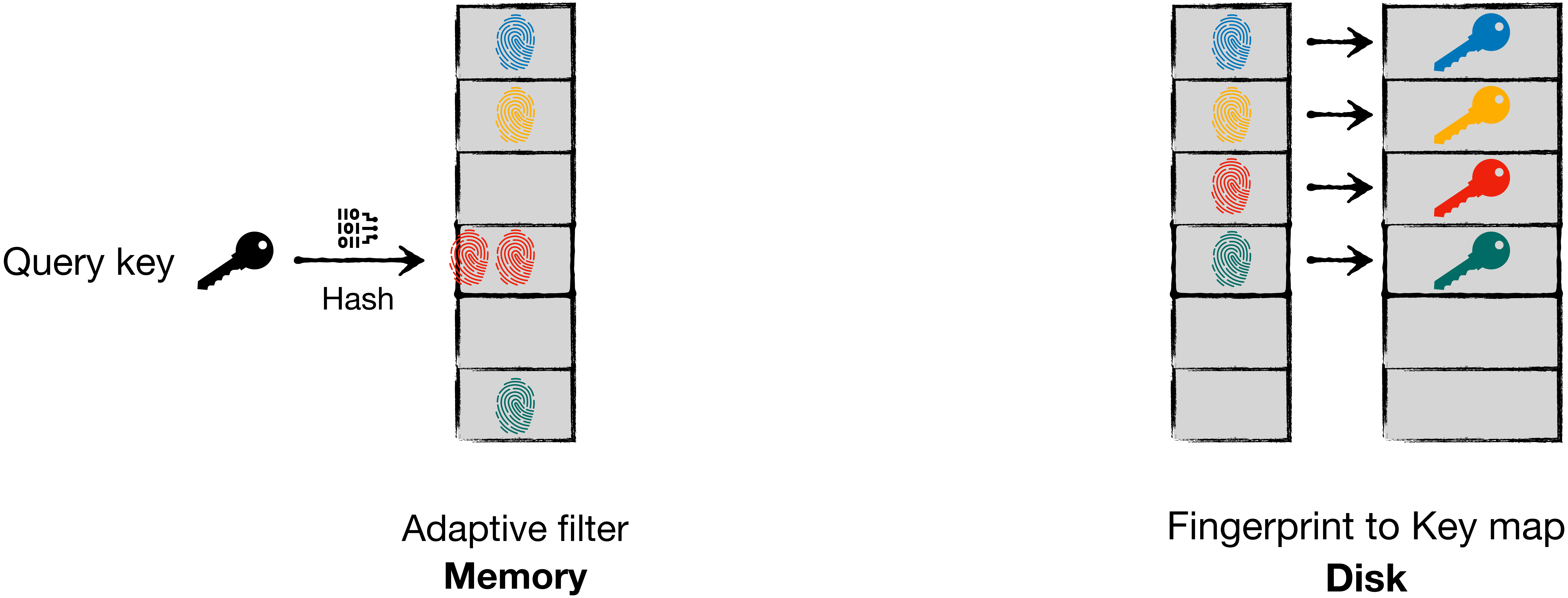


Adaptive filter
**Memory**



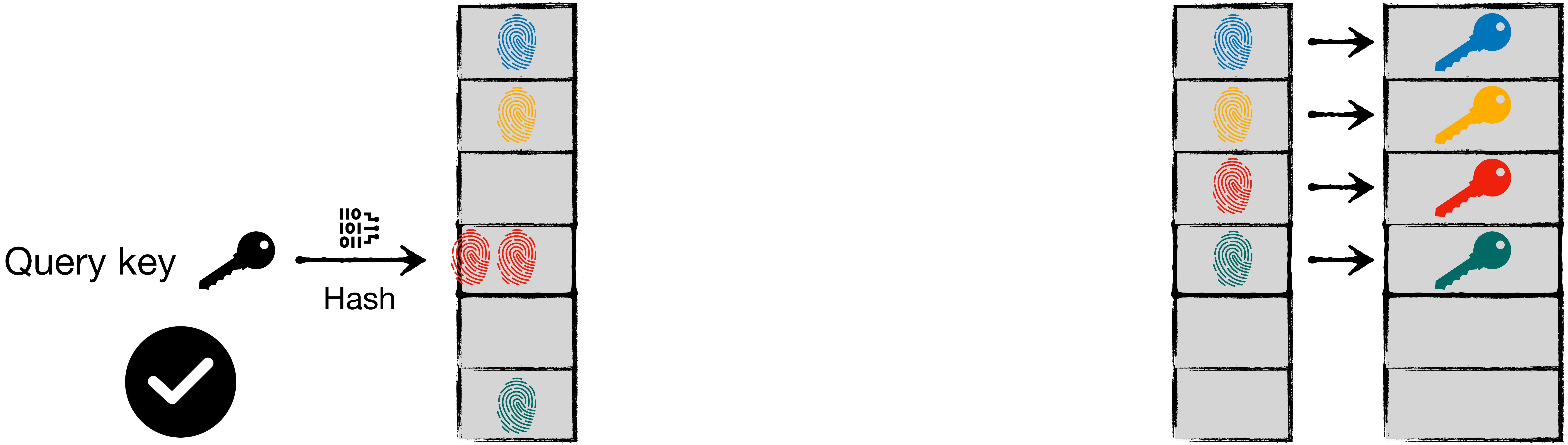
Fingerprint to Key map
**Disk**

# Adaptive filters employ variable-length fingerprints



Fingerprint collisions can cause false positives

# Adaptive filters employ variable-length fingerprints

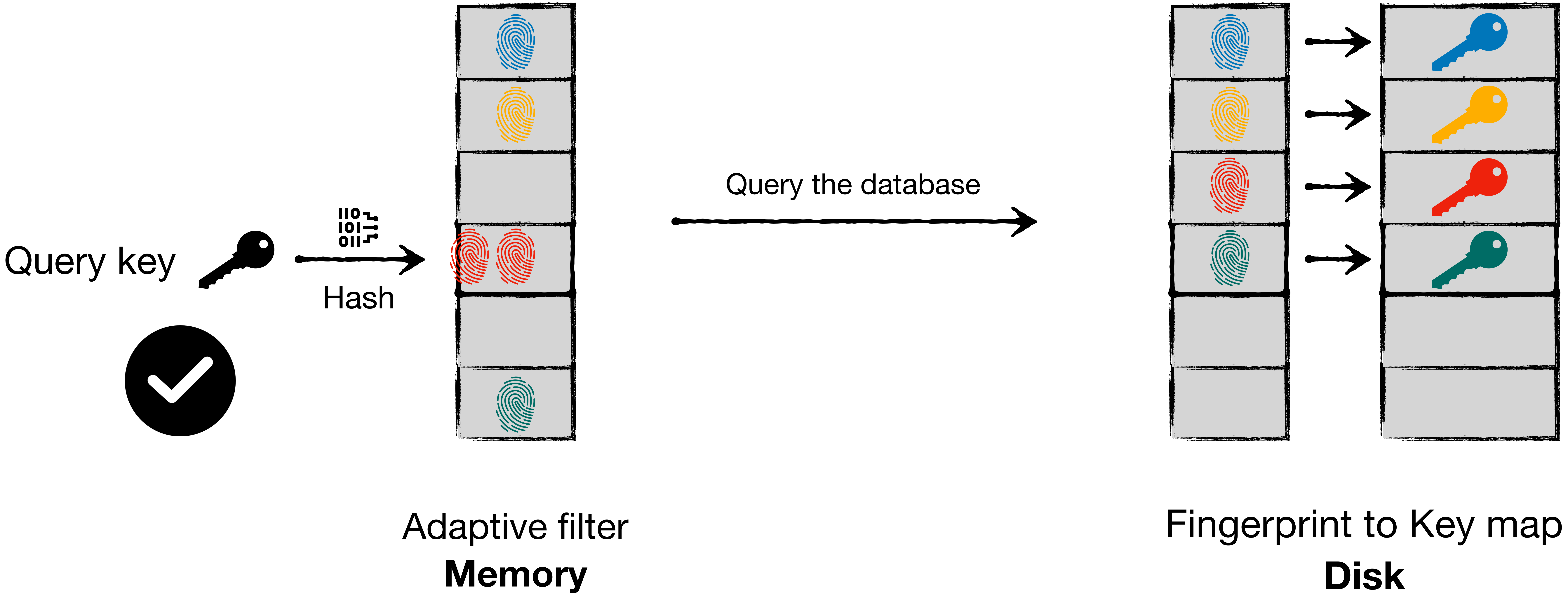


Adaptive filter
**Memory**

Fingerprint to Key map
**Disk**

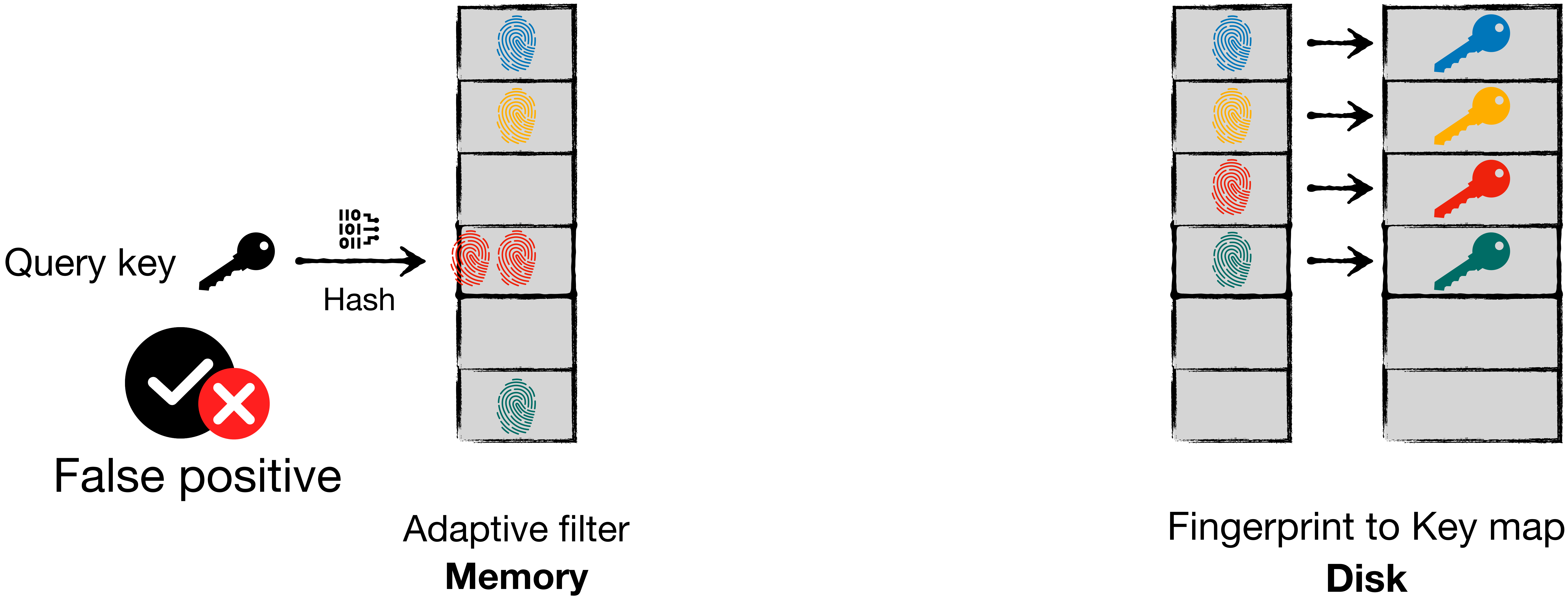
Fingerprint collisions can cause false positives

# Adaptive filters employ variable-length fingerprints



Fingerprint collisions can cause false positives

# Adaptive filters employ variable-length fingerprints



Query key

Hash

False positive

Adaptive filter

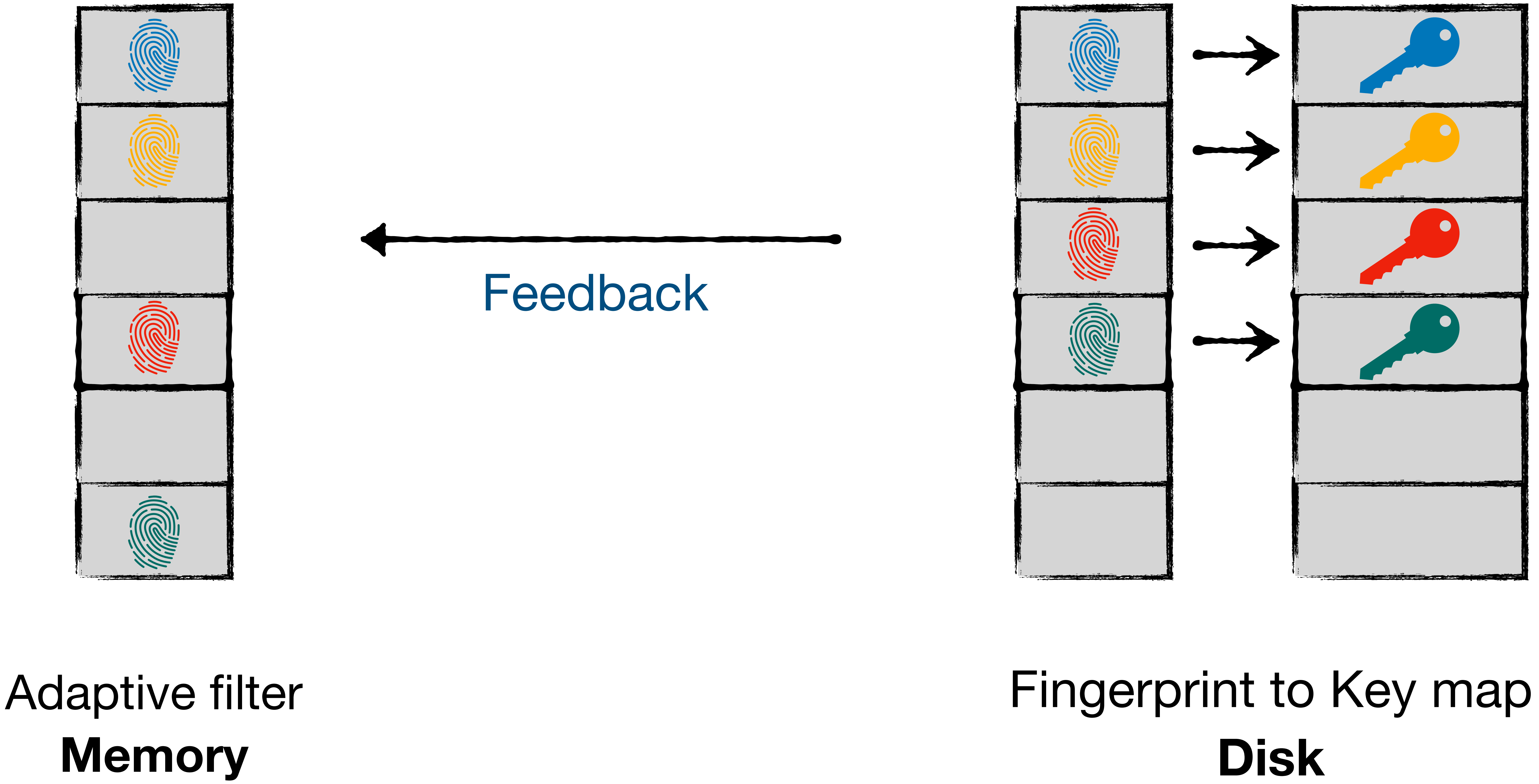
**Memory**

Fingerprint to Key map

**Disk**

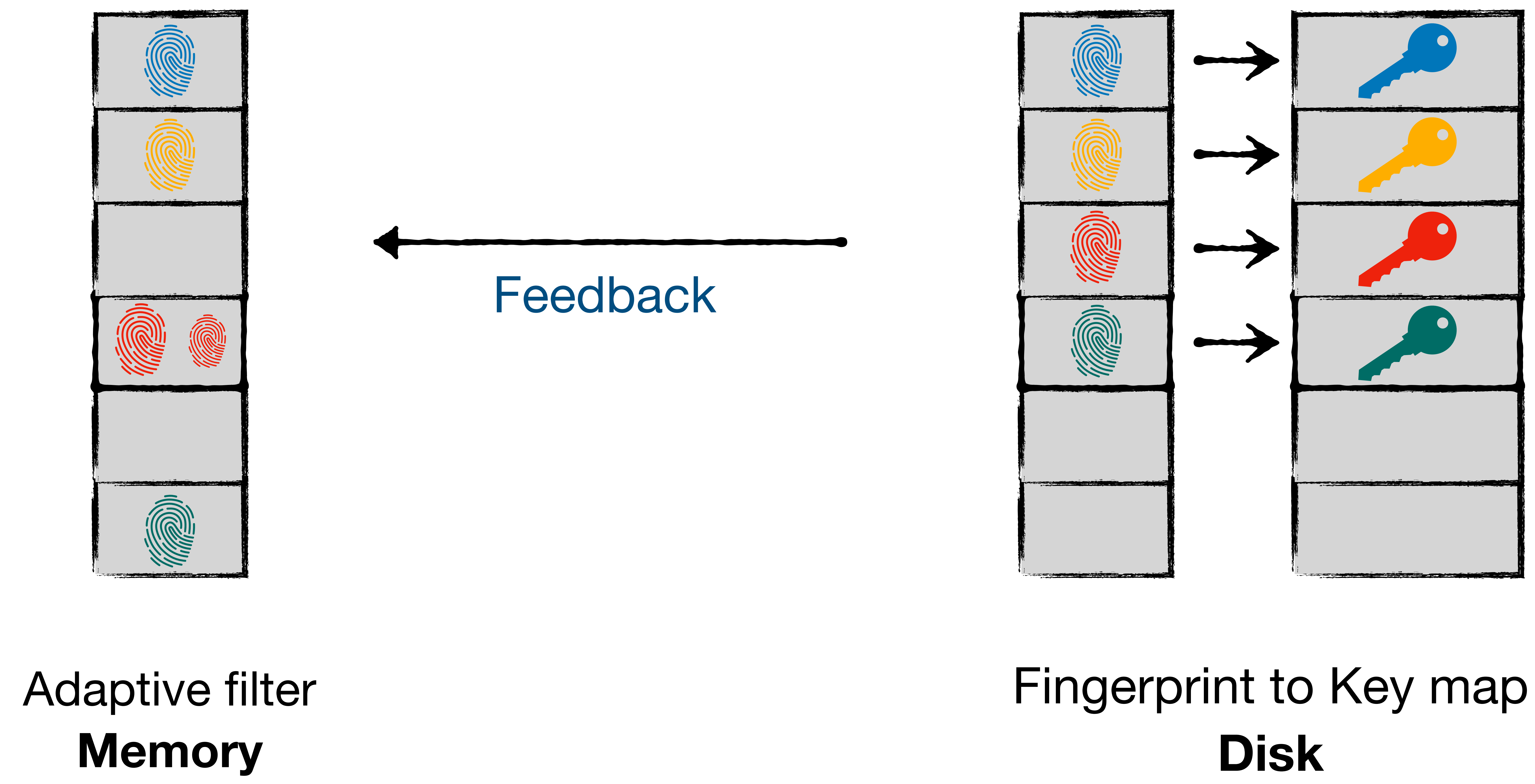
Fingerprint collisions can cause false positives

# Adaptive filters employ variable-length fingerprints



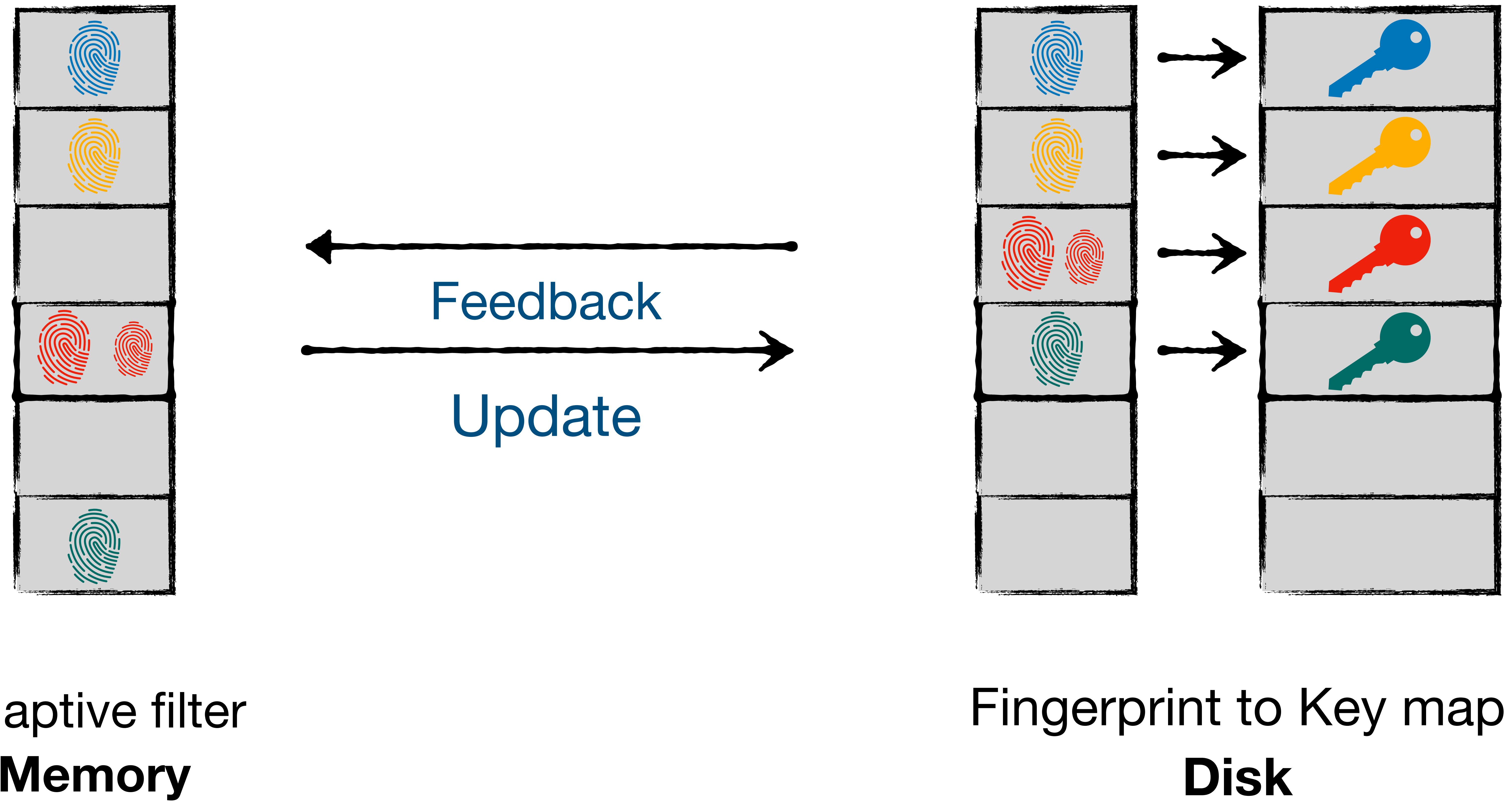
Feedback from the map can help fix the false positive

# Adaptive filters employ variable-length fingerprints



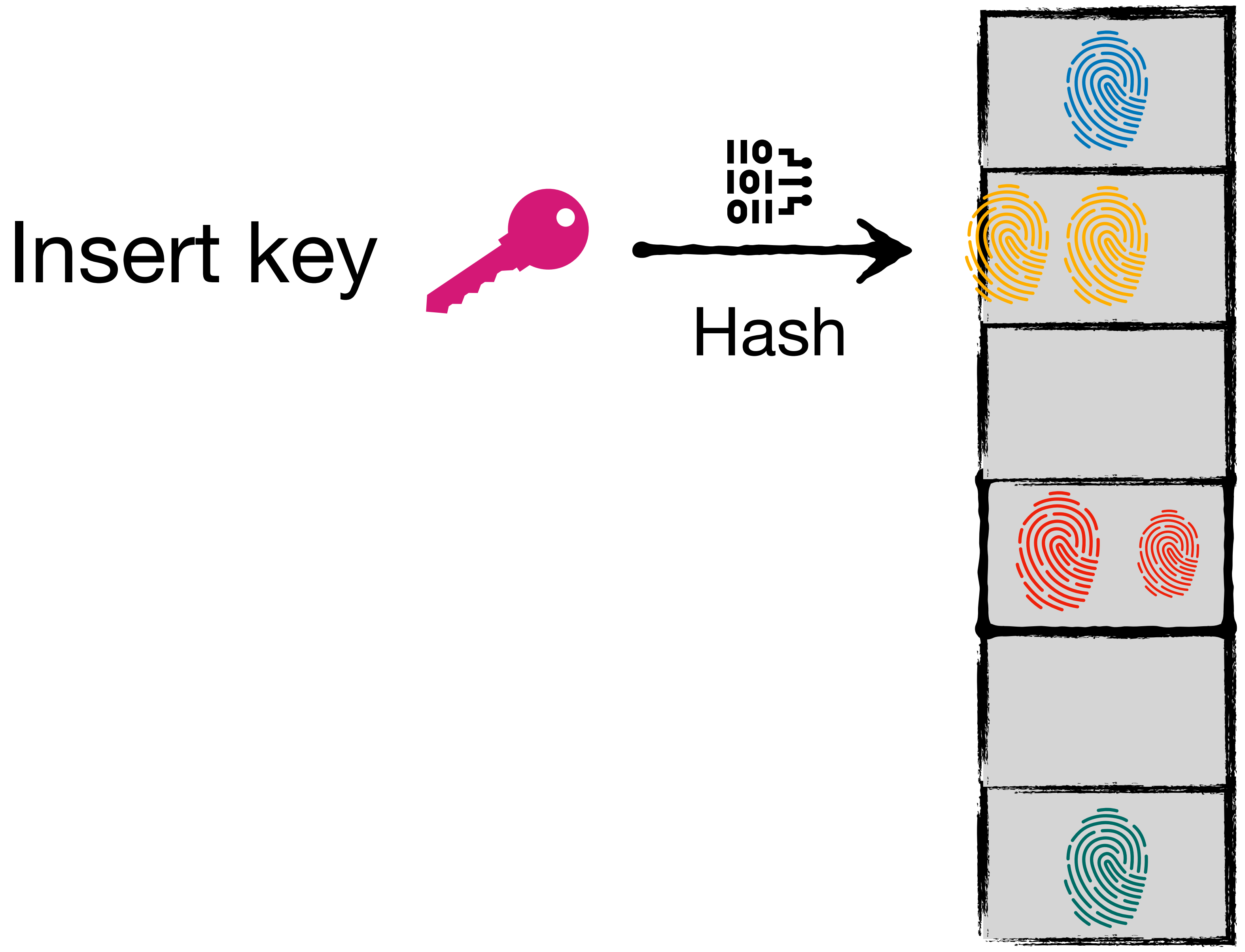
Extending the fingerprint of the existing key can avoid future false positives

# Adaptive filters employ variable-length fingerprints



Fingerprint map is updated accordingly

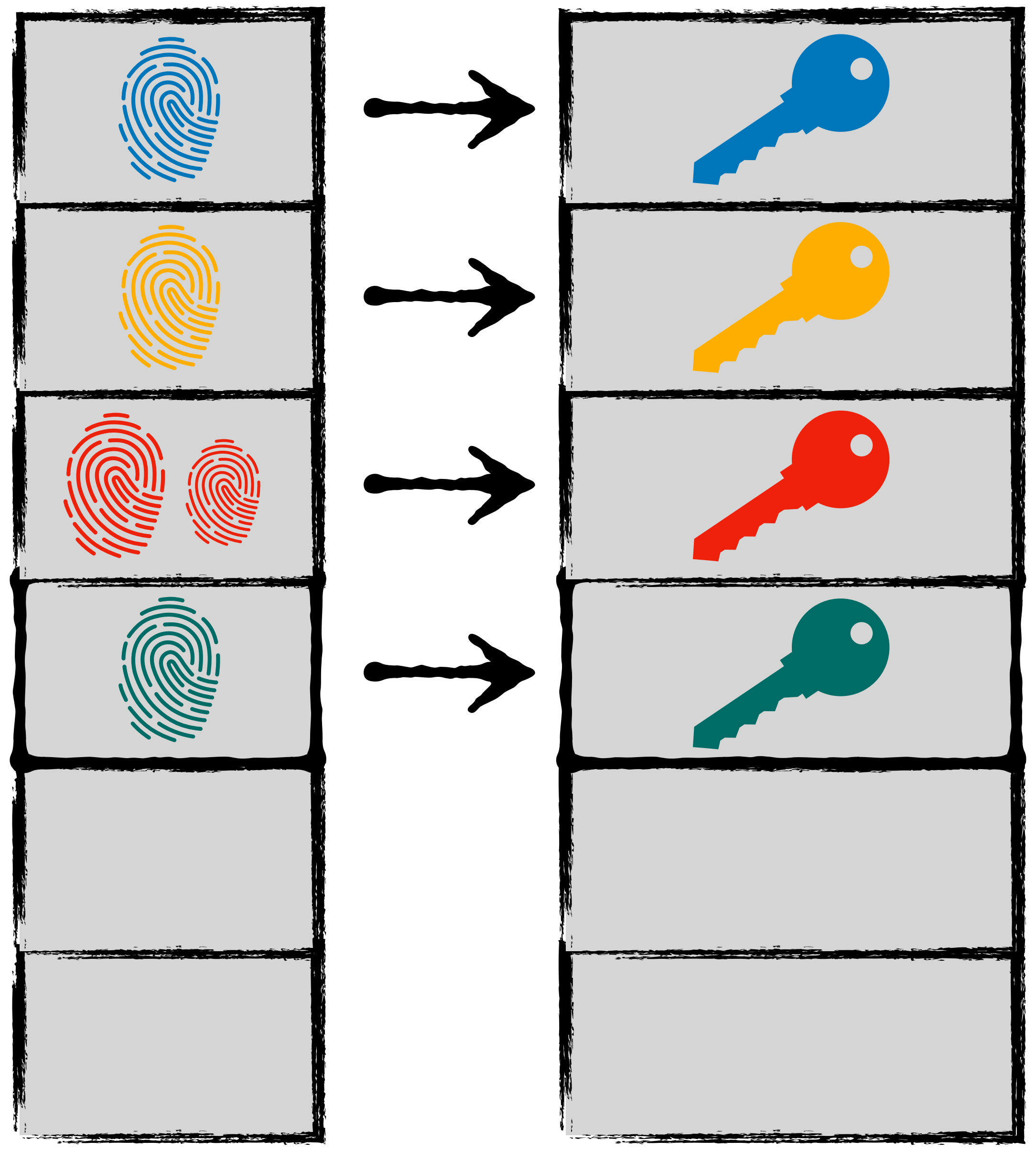
# Adaptive filters employ variable-length fingerprints



Insert key

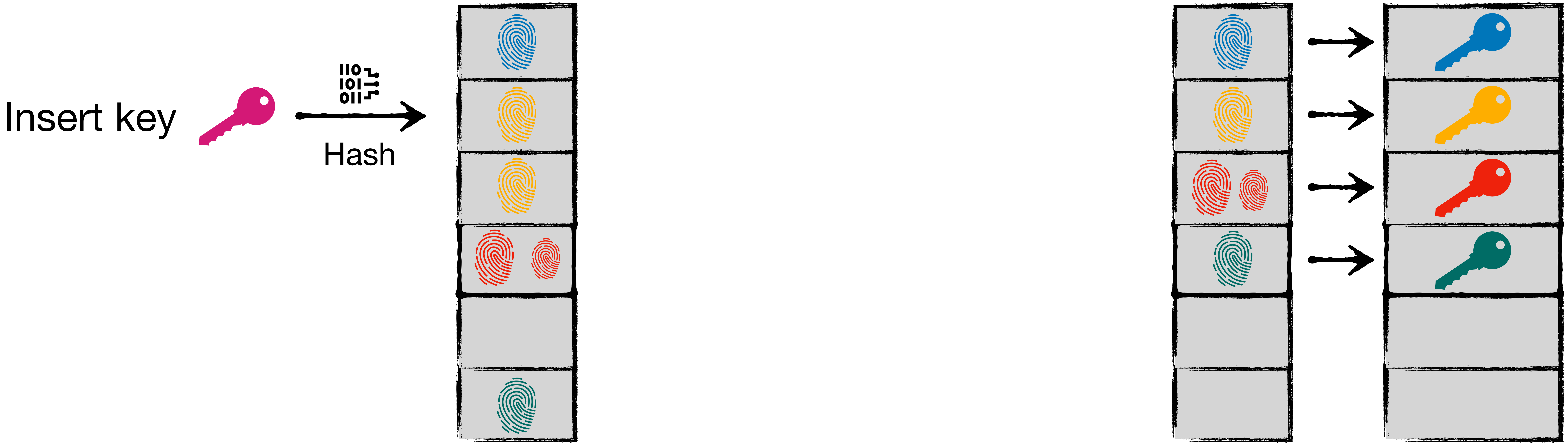
Hash

Adaptive filter
**Memory**



Fingerprint to Key map
**Disk**

# Adaptive filters employ variable-length fingerprints



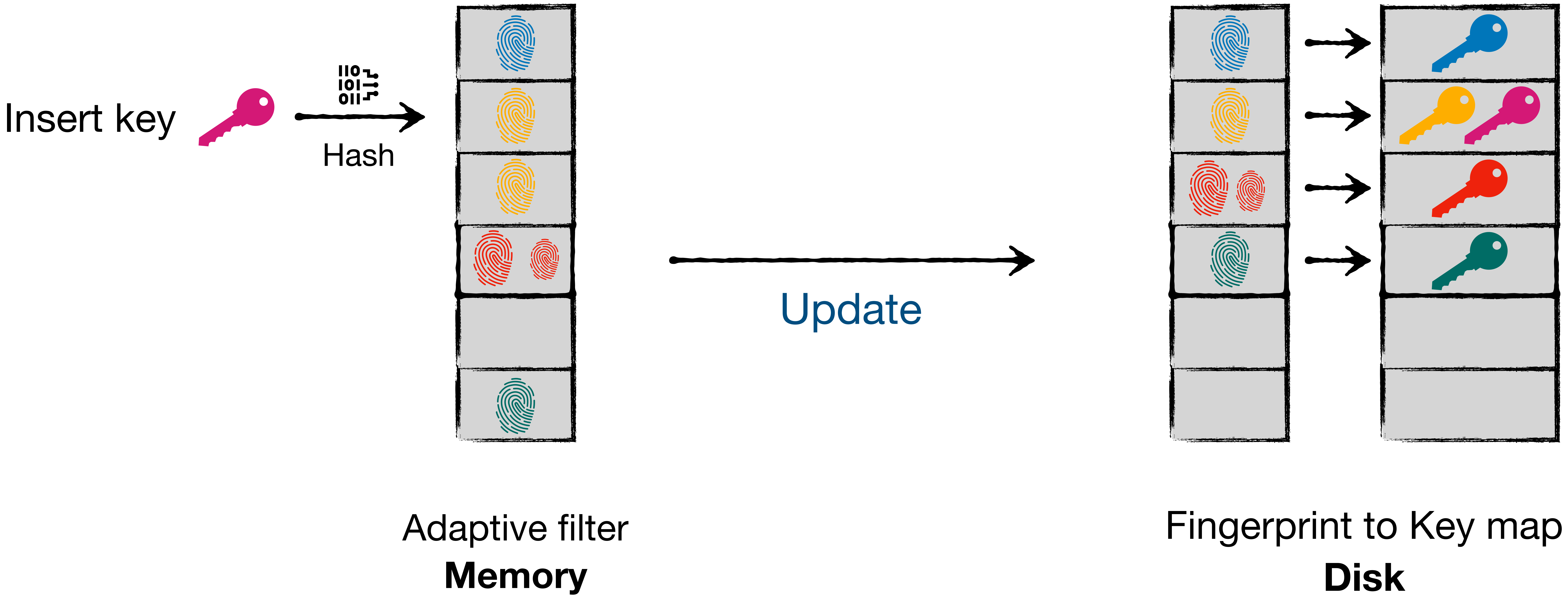
Insert key

Hash

Adaptive filter
**Memory**

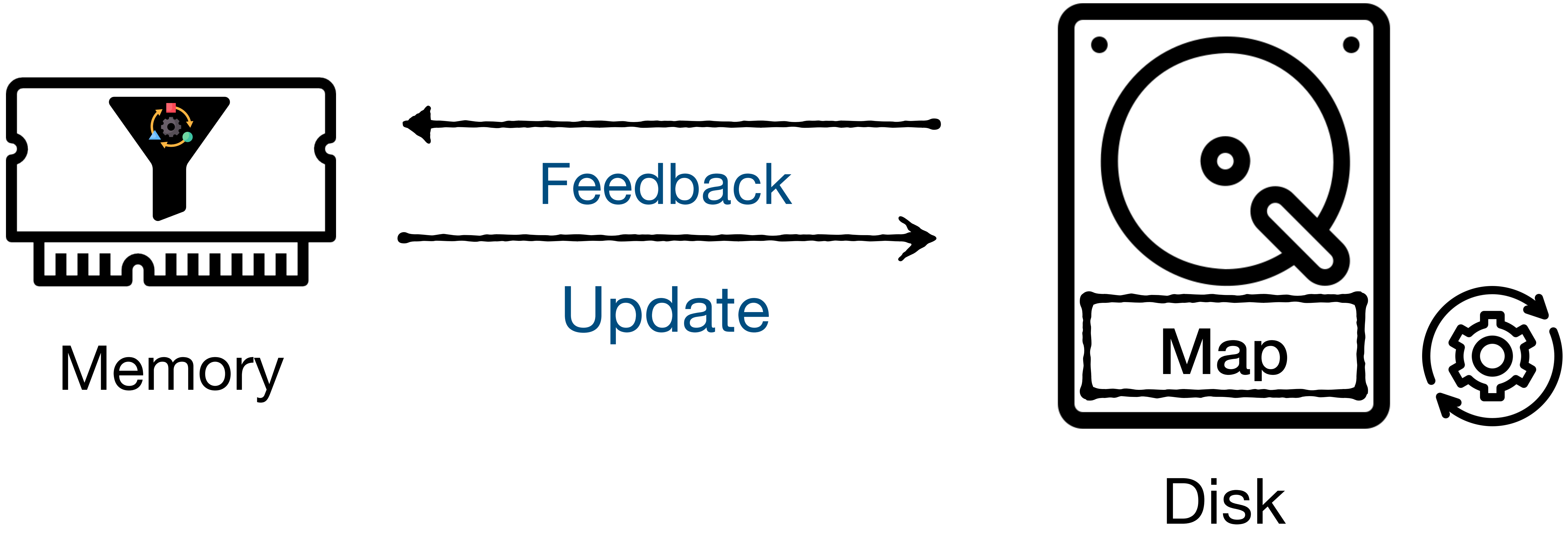
Fingerprint to Key map
**Disk**

# Adaptive filters employ variable-length fingerprints



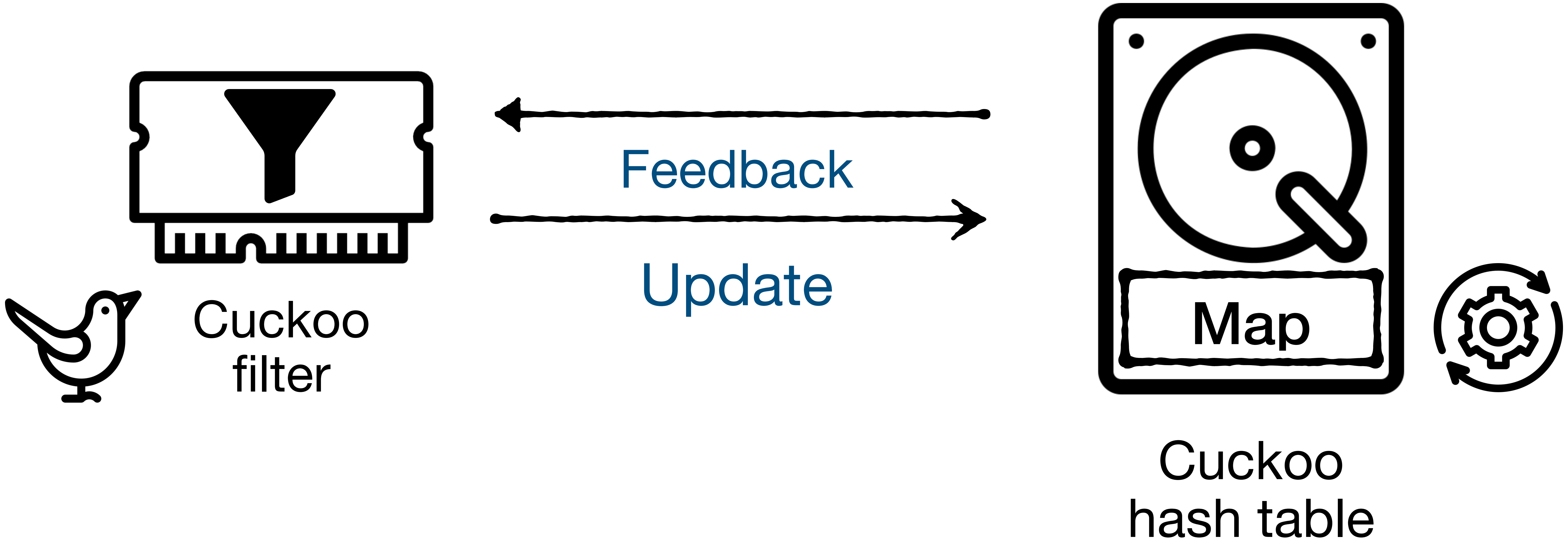
Fingerprint map is updated accordingly

# Fingerprint map updates dominate the performance



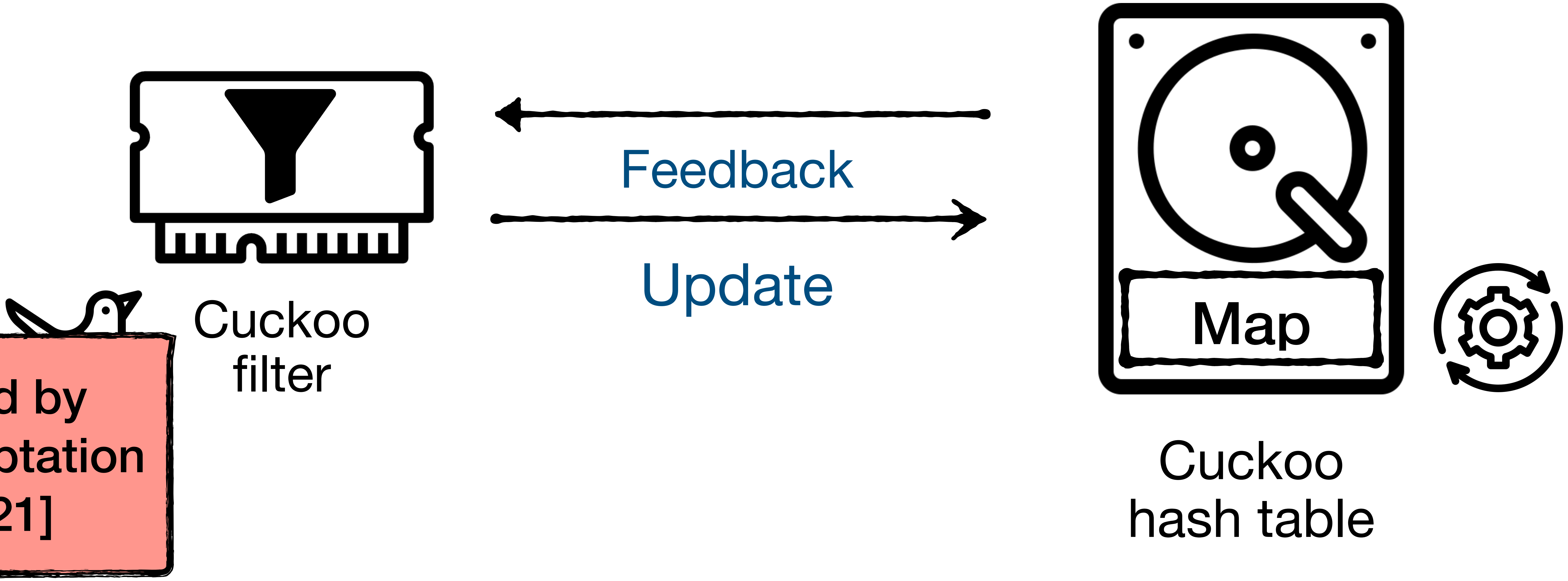
Minimizing the work in the map is crucial for the performance

# Adaptive cuckoo filters [MPR+ 2020]



Adaptivity by moving fingerprints around

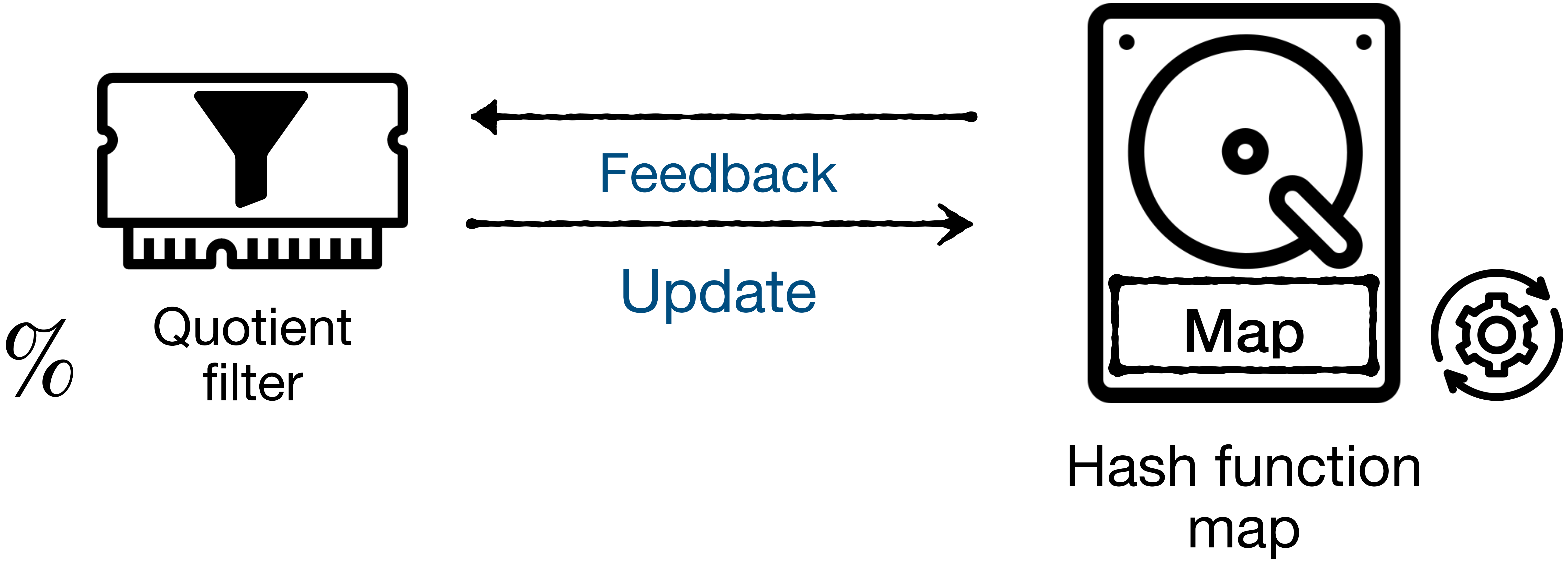
# Adaptive cuckoo filters offer weak adaptivity



Can be attacked by identifying an adaptation loop [KMP 2021]

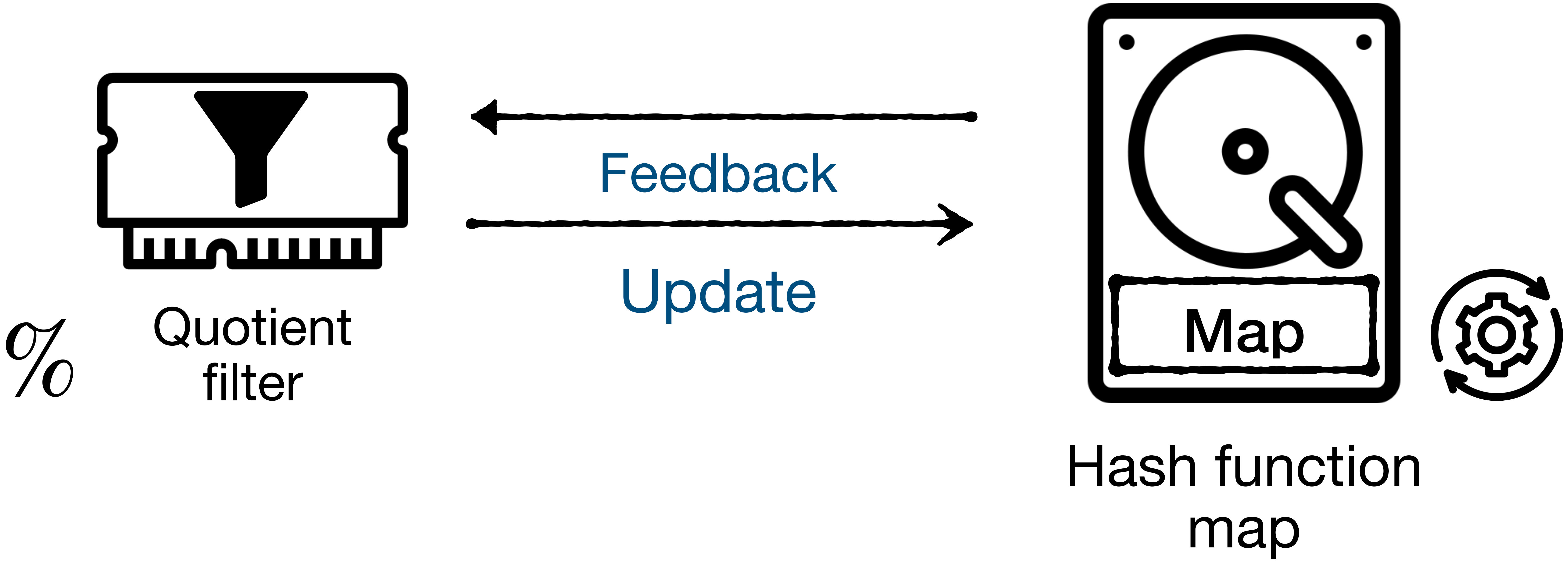
- ❗ Adaptivity by moving fingerprints around during insertions/queries
- ❗ Can forget previous false positives while adapting for new ones

# Telescoping filters [LMS+ 2021]



Adaptivity by changing hash function during insertions/queries

# Telescoping filters offer strong adaptivity



❗ Adaptivity by changing hash function during insertions/queries

Hash map grows during adaptations (variable-length fingerprints)

Does not forget previously learned fingerprints

# Adaptive quotient filter [WMT+ SIGMOD 2025]

- Adaptivity by using variable-length fingerprints to avoid collisions
- Based on the quotient filter (CQF) [PBJ+ 2017]
- Matches the space lower-bound to lower-order terms
- 10X—30X faster than other adaptive filters (ACF, TF) for disk-based database benchmarks
- Up to 6X faster performance than traditional filters (QF, CF) for disk-based database benchmarks

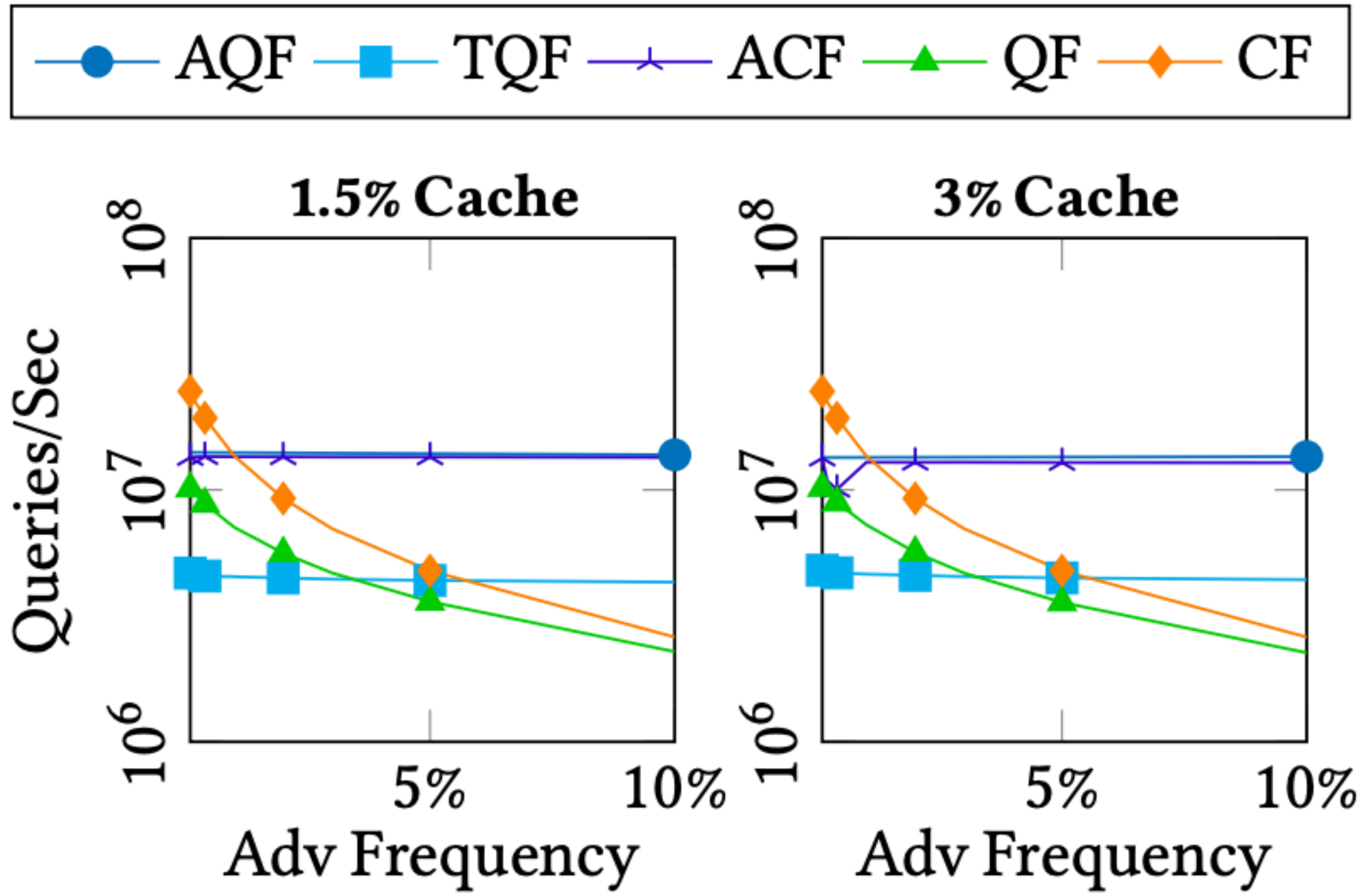
# Adaptive quotient filter design

Preserves CQF
performance and features

Stable reverse map
during insertions

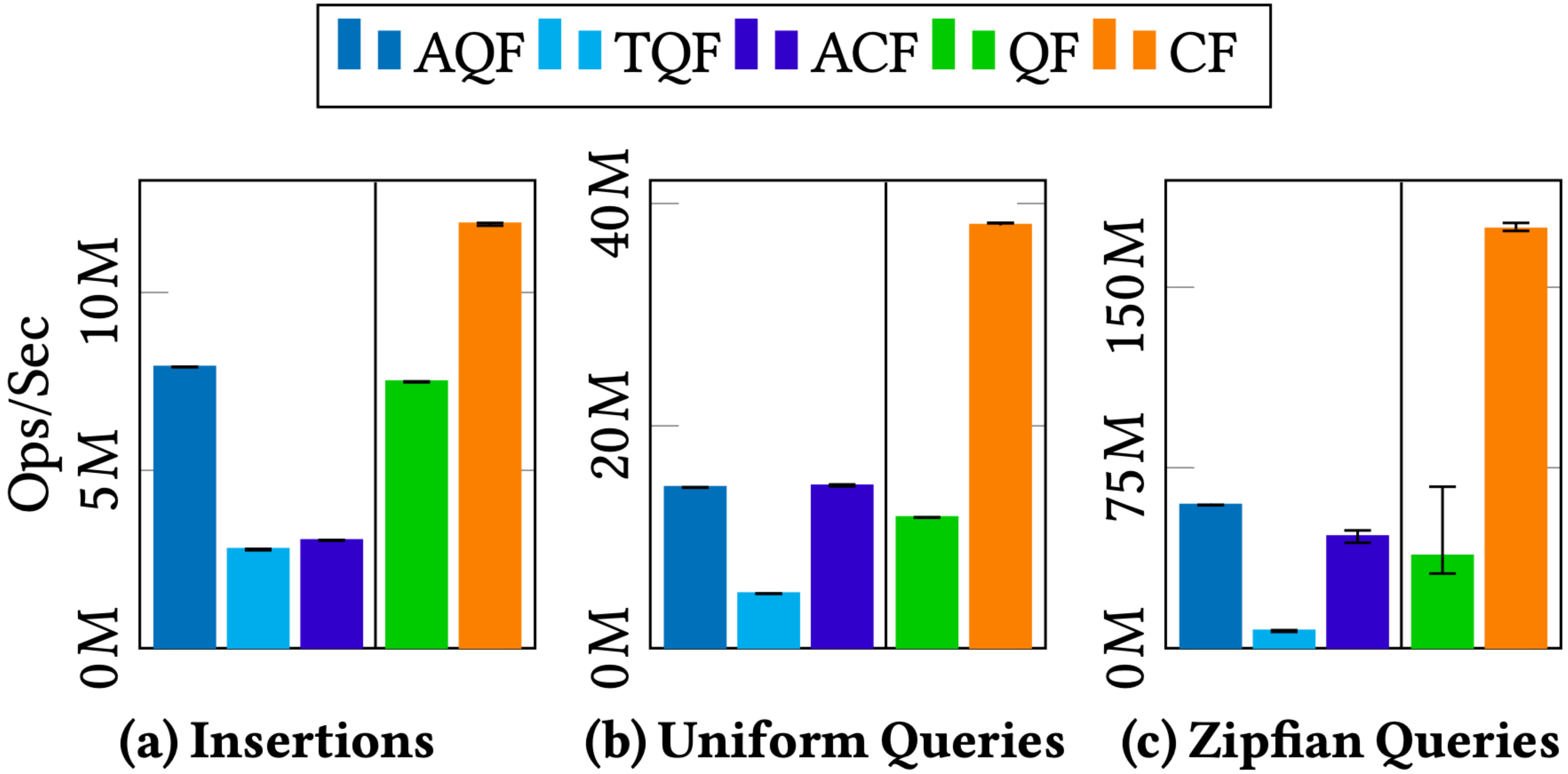
Supports dynamic
operations

# Database query performance



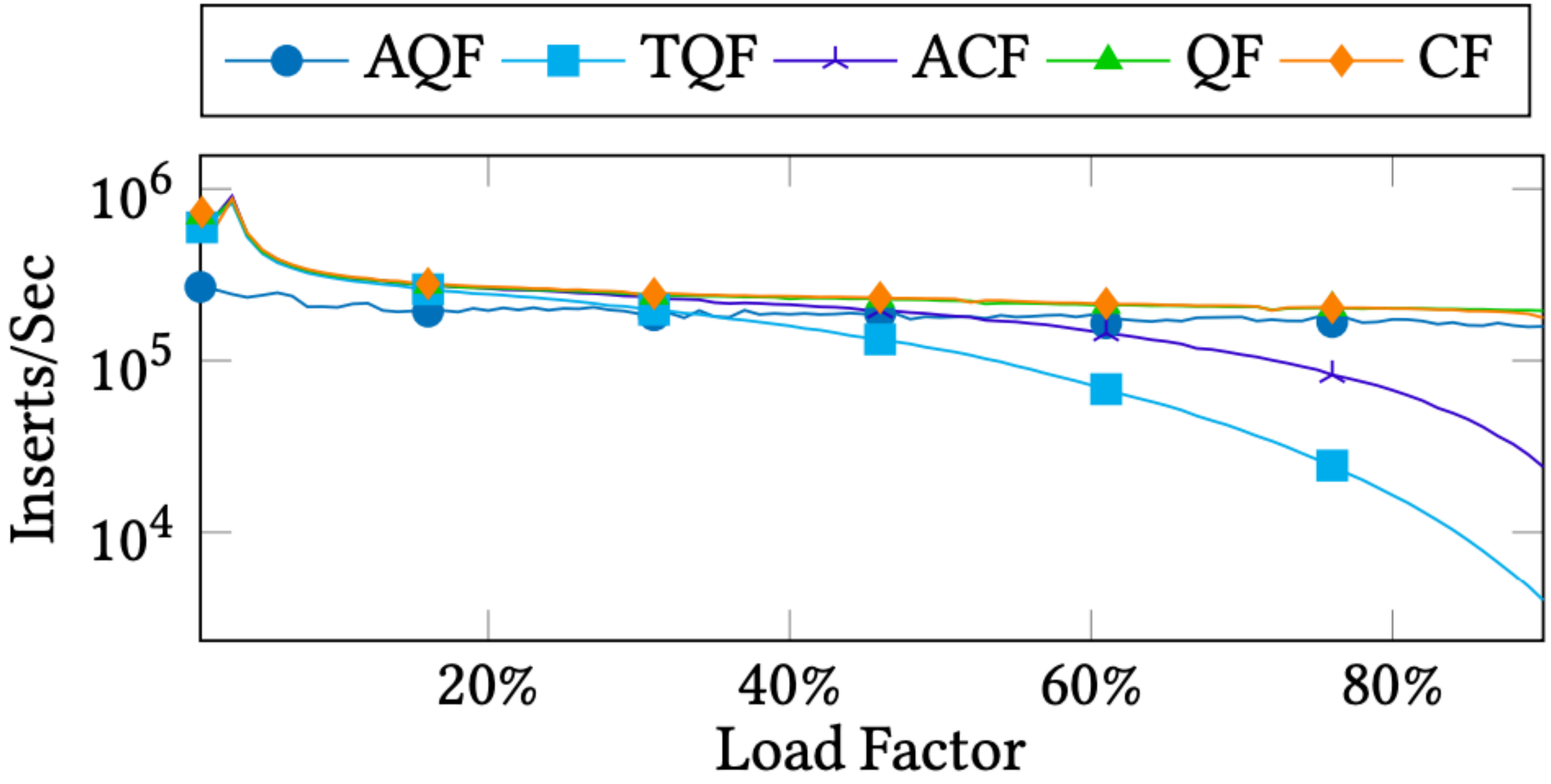
AQF up to 6X faster compared to QF/CF for database queries

# Micro-benchmark performance



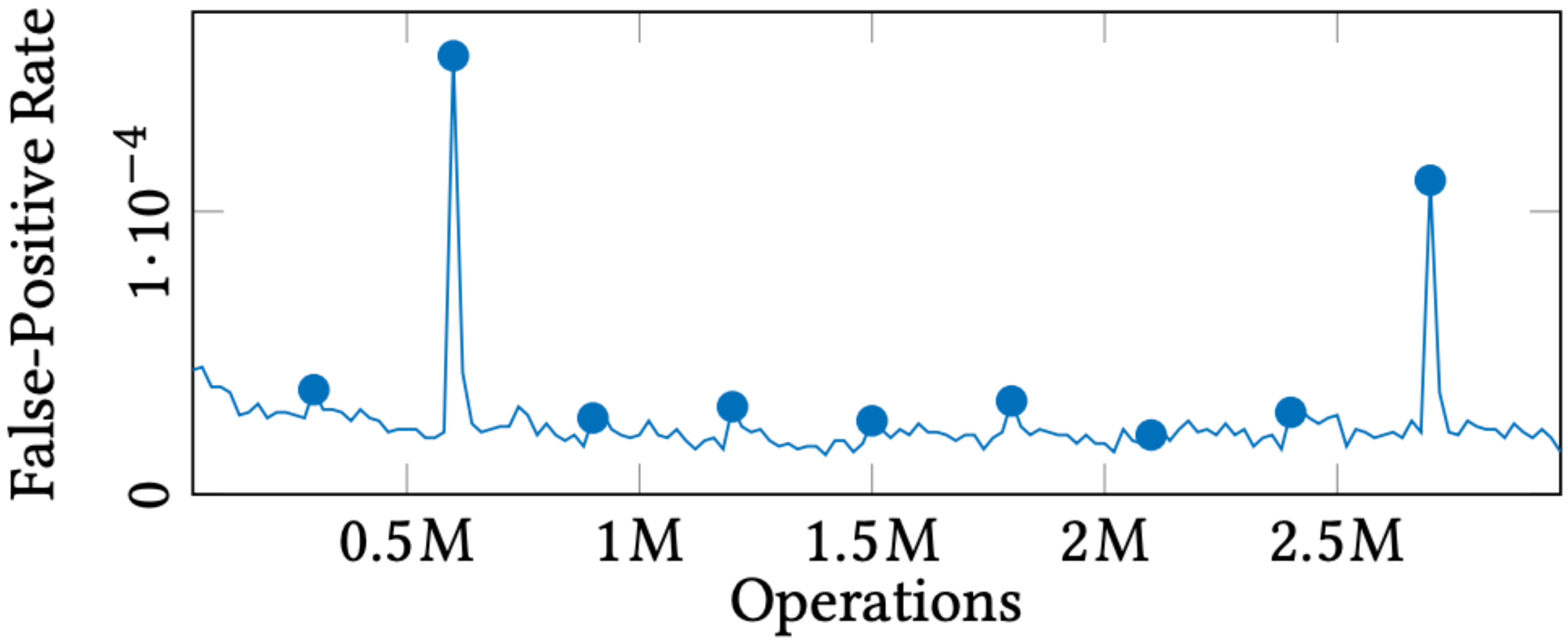
AQF has no overhead compared to the traditional CQF

# Database insertion performance



AQF performs similarly to QF/CF for database insertions
10X—30X faster than other adaptive filters

# Adaptivity rate on a churn workload



AQF adapts to new false positives almost immediately for churn workloads

AQF offers even stronger guarantees compared to the broom filter [BFG+ 2018]

# Monotonically adaptive filters [WMT+ SIGMOD 2025]

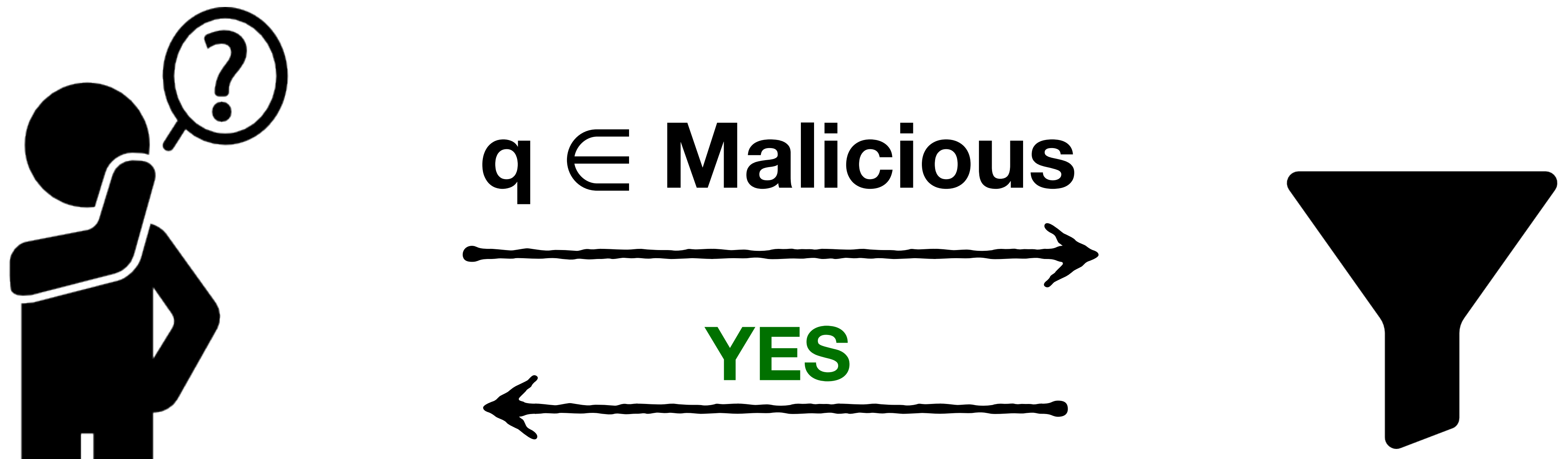
A filter that never forgets a false positive

We can use monotonicity to solve other problems; security

# False positives can be really expensive

Malicious URLs

Legitimate URLs



$q \in$ **Malicious**

**YES**

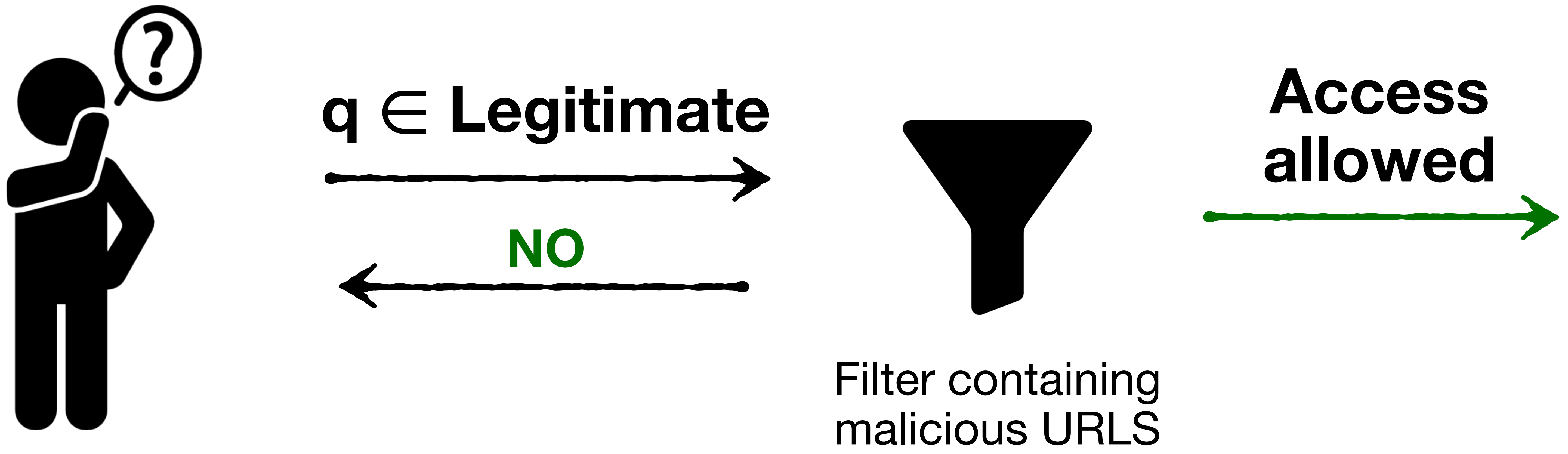
Filter containing
malicious URLS

Blocks malicious URLs

# False positives can be really expensive

Malicious URLs

Legitimate URLs



q ∈ **Legitimate**

**NO**

Filter containing
malicious URLS

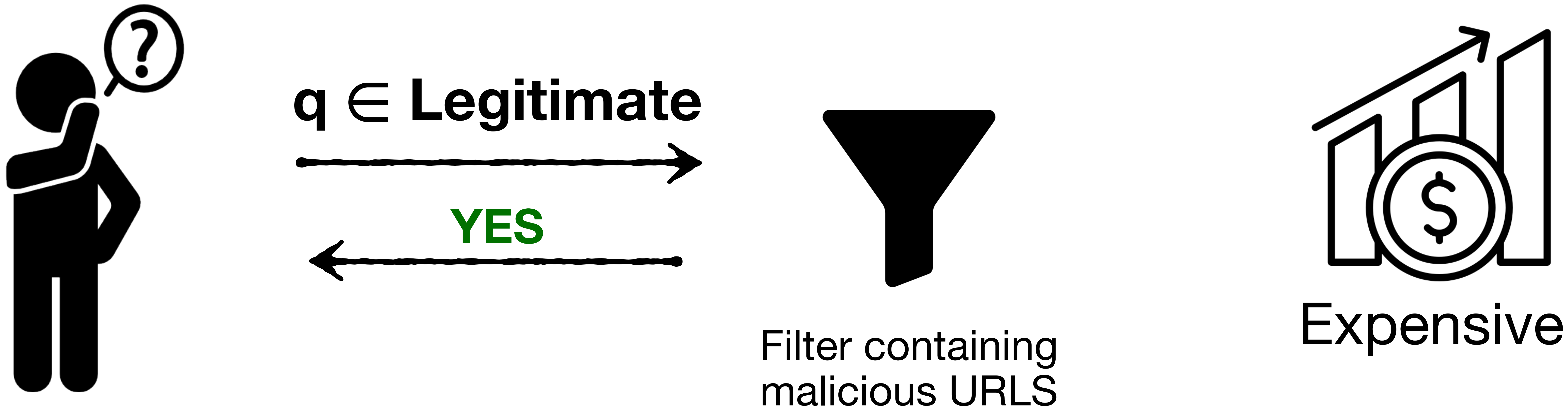
**Access allowed**

Allows legitimate URLs

# False positives can be really expensive

Malicious URLs

Legitimate URLs



**$q \in$ Legitimate**

**YES**

Filter containing
malicious URLS

Expensive

A false positive can **block critical URLs** such as a **voter registration webpage** or **emergency weather info**

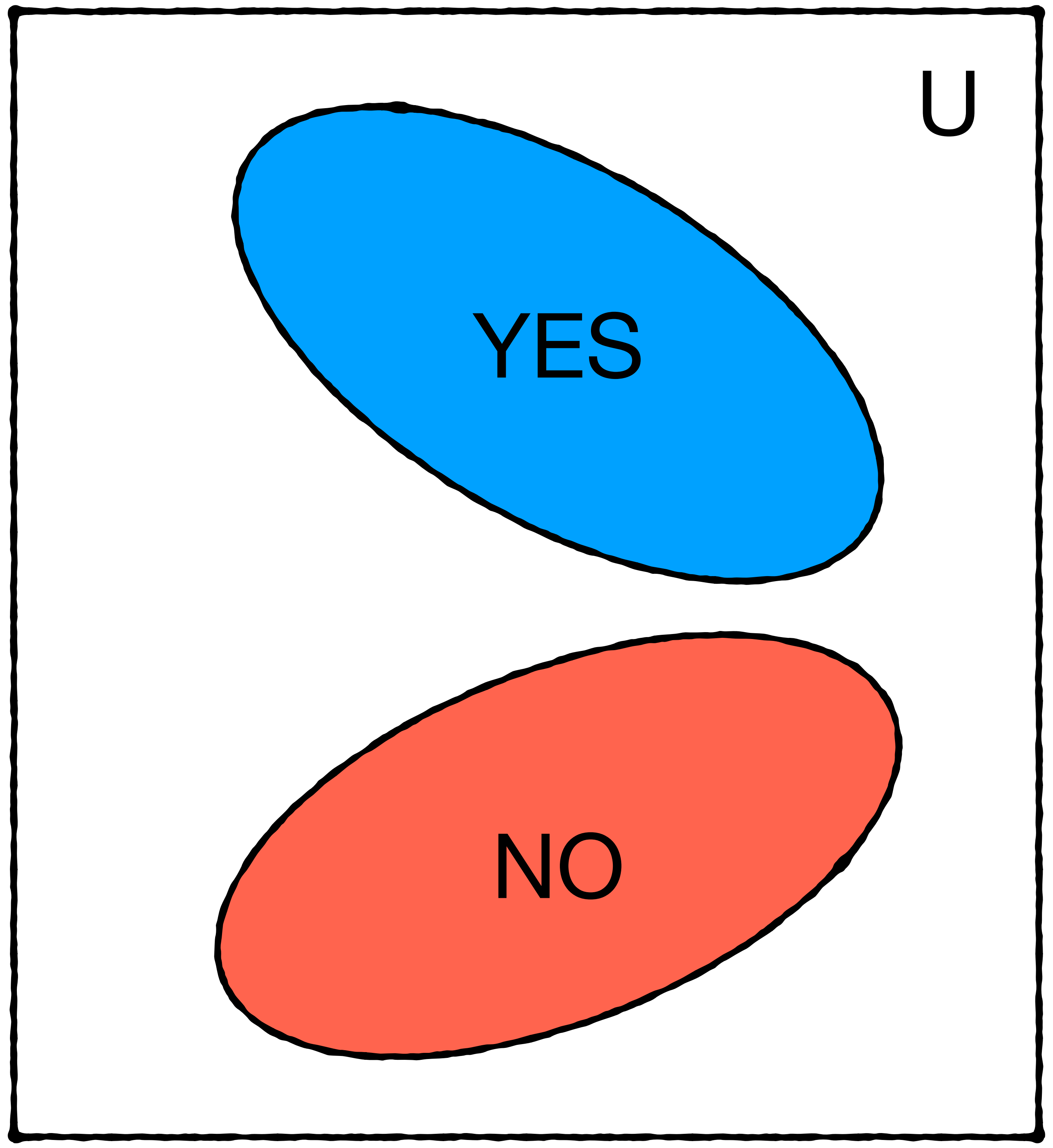
False positive

# YES/NO list problem

if q ∈ YES, return **True** with probability 1

if q ∈ NO, return **False** with probability 1

Otherwise **False** with probability $> 1 - \epsilon$



Applications:

- Detecting malicious URL
- Certificate revocation lists
- De Bruijn graph traversal

**Monotonicity is critical to support YES/NO List problem!**

# Prior work considered each problem separately

**Purpose-built solutions**

Bloomier filter [CKR+ 2004]

Cascading Bloom filter [TC 2009]

Static XOR filter [RSW+ 2021]

Seesaw counting filter [LCD+ 2022]

Complex design

Low performance

High space

# Monotonically adaptive filters solve many problems

- Security; avoiding DOS attacks
  - Static YES/NO list
  - Dynamic YES/NO list

- Robust performance guarantees
  - Skewed query distributions
  - Adversarial queries

Networking

Computational biology

Databases

# Takeaways

- Adaptability is a critical to achieve robust performance in the context of skewed/adversarial workloads

- Monotonically adaptive filters can help address challenges across applications

- We need to redesign traditional applications in the context of newer guarantees and API offered by adaptive filters

# Conclusion

- Data systems backed by strong theoretical guarantees are key to tackle future data analyses challenges
- We can efficiently employ modern hardware by developing new algorithmic paradigms
- Building open and scalable data systems is critical for democratizing data science

https://prashantpandey.github.io/

Acknowledgment: All icons in the talk are taken from https://www.flaticon.com/