Software Cache

Juncheng Yang

March 4

Harvard John A. Paulson
School of Engineering
and Applied Sciences

Agenda

- Software cache basics
- Different types of caches
- Miss ratio curve
- Eviction algorithm
- Admission algorithm
- Prefetch algorithm

Key Questions to Think About After Class

- What is a cache and why would it work?
- What is a miss ratio curve and how can I construct one efficiently?
- What are some eviction, admission and prefetching algorithms? How do they work?

Cache basics

overview

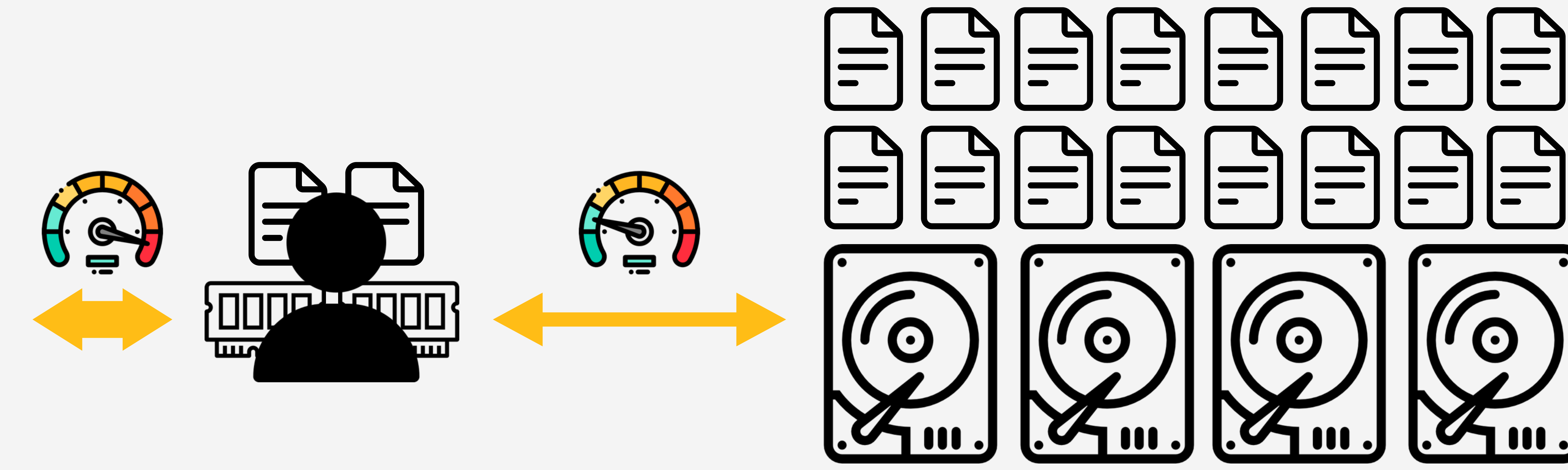
five-min rule

What is a cache?

- Storage device latency: 10s μs to 10 ms
- Cache: a ***fast but small*** storage storing a portion of the dataset to speed up data access

$$\text{Latency} = \text{Hit Latency} + (\text{Miss Ratio} \times \text{Miss Latency})$$

10% miss ratio: reduce latency by ~90%



Software cache

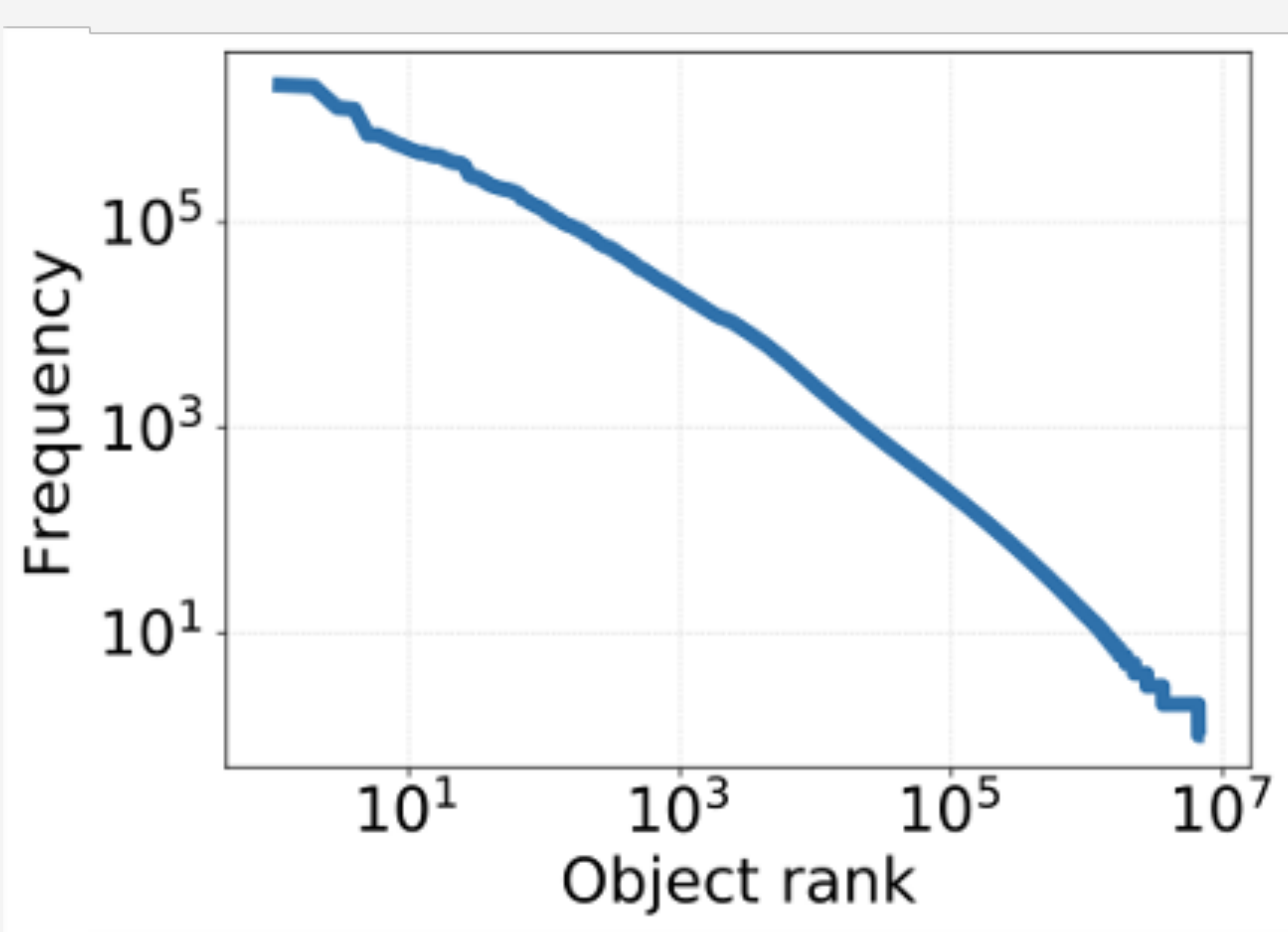
- Cache: A ***fast but small*** storage storing a portion of the dataset to speed up data access
- Software cache: all decisions made in software

- Google CliqueMap: PBs of DRAM[1]
- Twitter: 100s clusters, 100s TB of DRAM, 100,000s cores[2]

[1] CliqueMap: Productionizing an RMA-Based Distributed Caching System, SIGCOMM'21
[2] A large scale analysis of hundreds of in-memory cache clusters at Twitter, OSDI'20

Locality

- Why would a cache work?
- Temporal locality
  - recently accessed data are likely to be reused
- Skewed popularity
  - data popularity follow power-law (80-20 rule)
  - Zipf's law:
    - frequency $\propto \dfrac{1}{(rank + b)^{\alpha}}$
    - a small set of objects are accessed more than others
- What happens if there is no locality and skewed popularity?



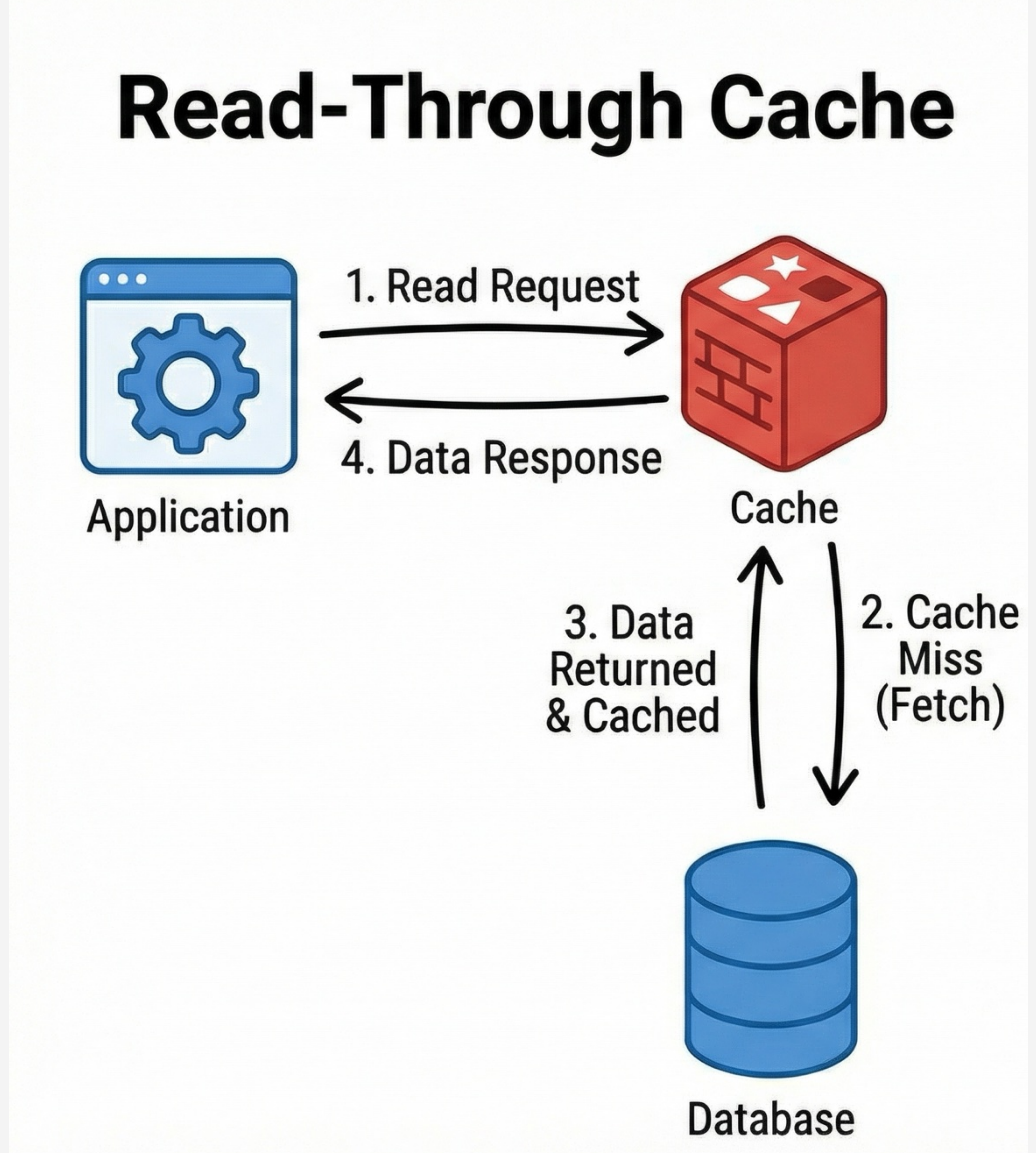
Metrics

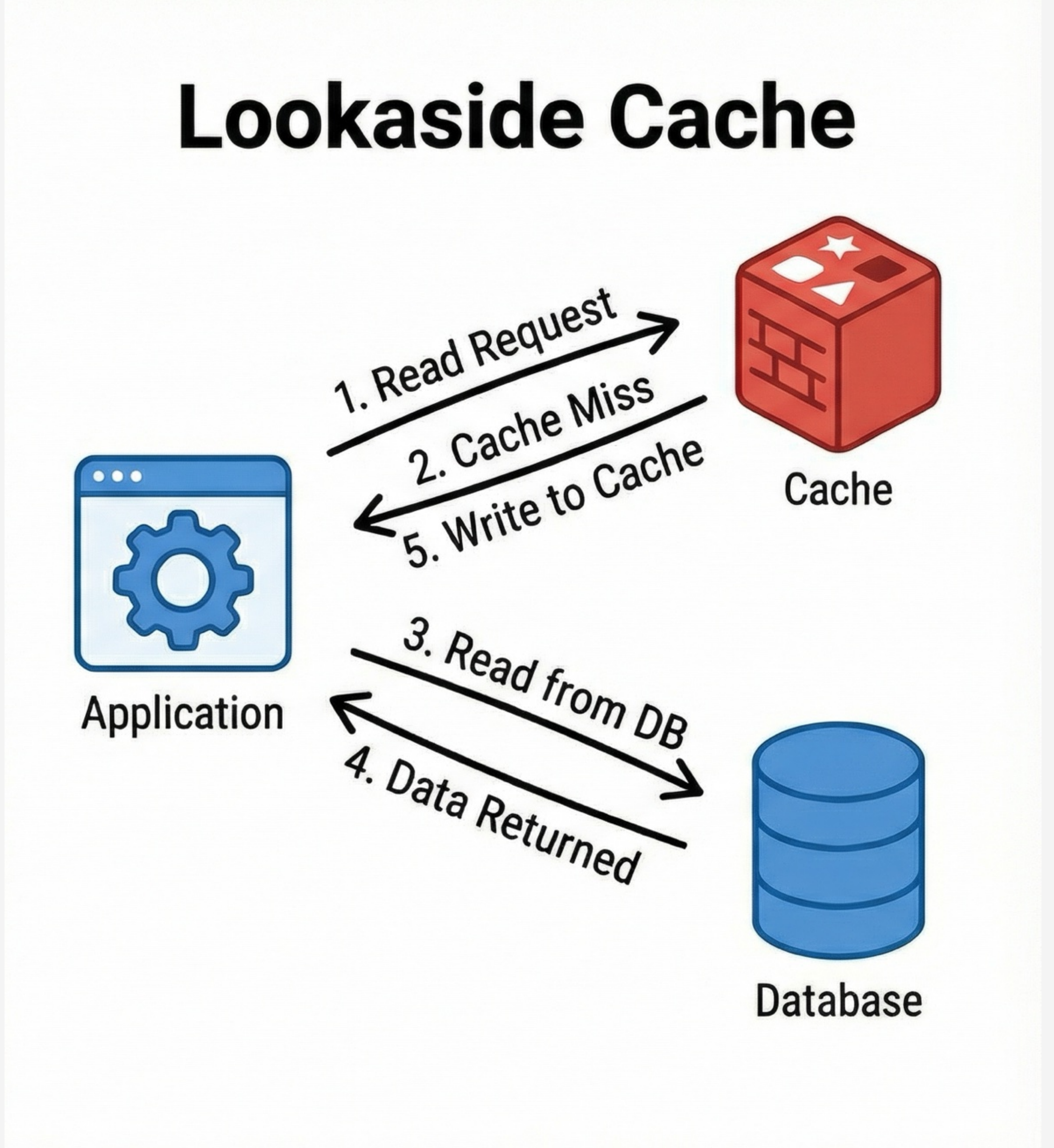
- Efficiency
  - measured using miss ratio and byte miss ratio
  - different from *miss rate*
- Performance
  - measured using throughput (requests/sec)
  - scalability measures how throughput changes with #cores
- More?
  - better efficiency metrics
  - robustness
  - simplicity
  - fairness

Cache miss

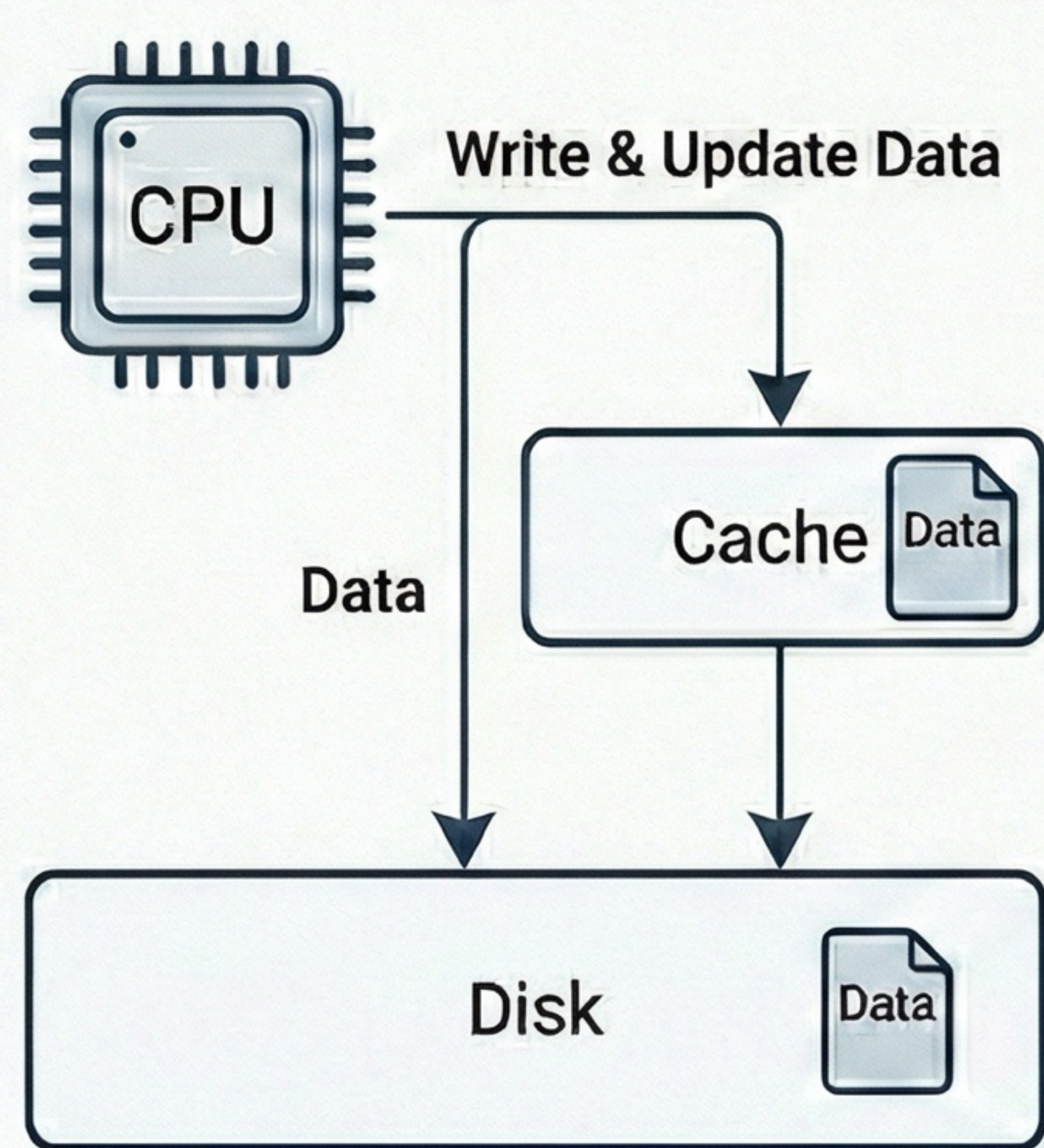
- Compulsory miss
  - first seen data
  - also called cold miss
- Capacity miss
  - most common
  - highly depends on eviction algorithm
- ~~Conflict miss~~
  - rare in software cache
  - not set-associative

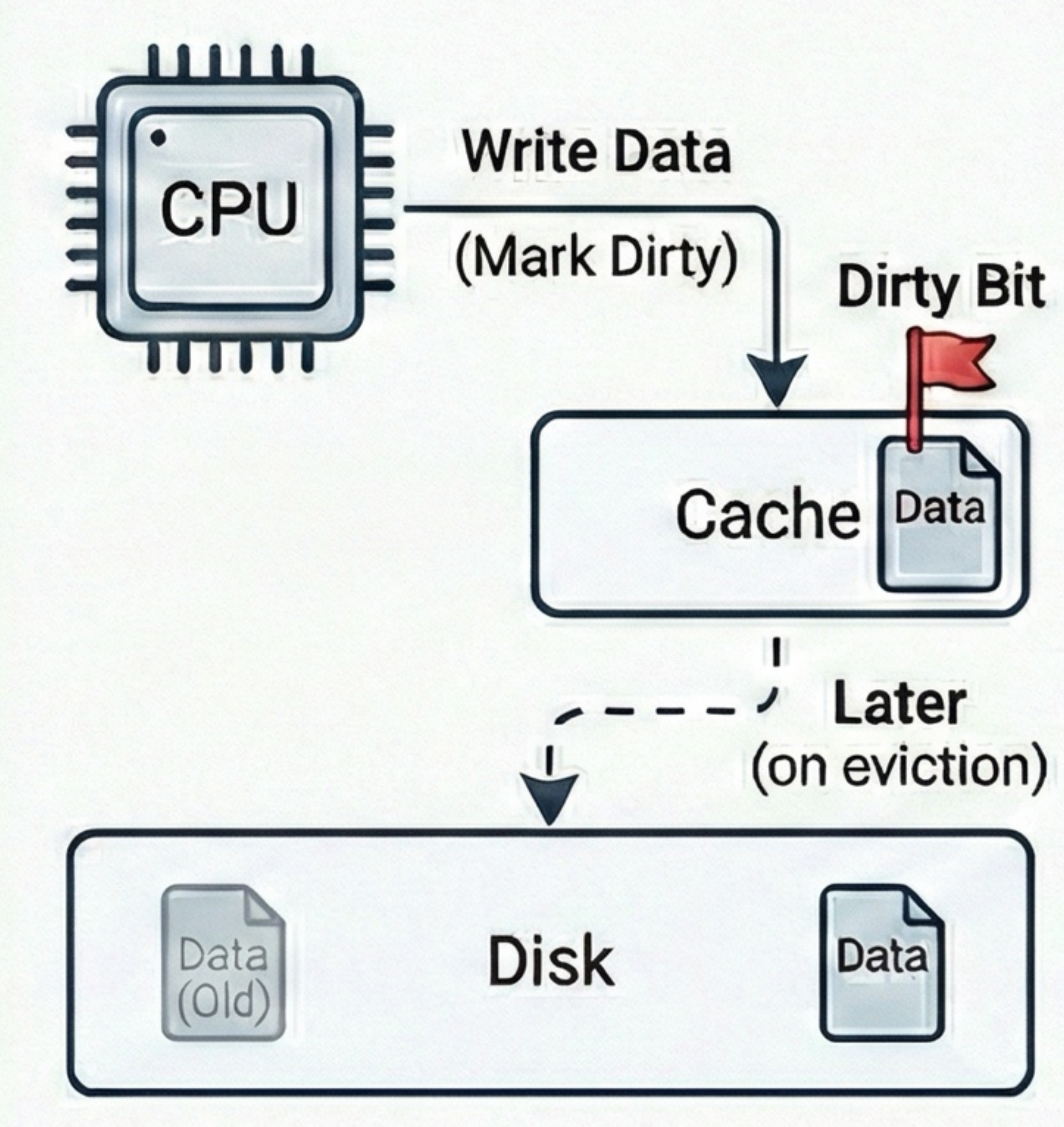
Cache policies

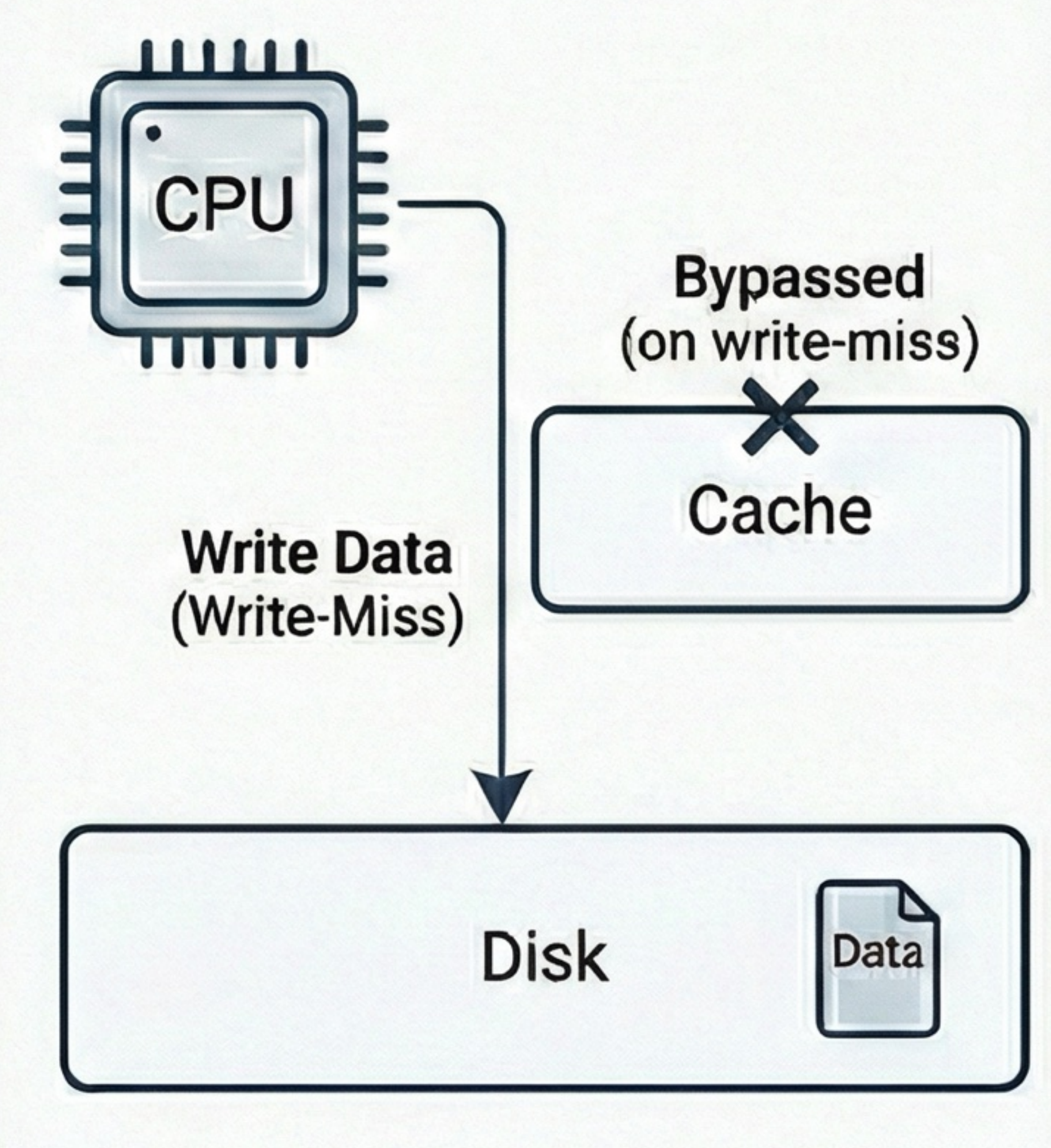




Cache policies





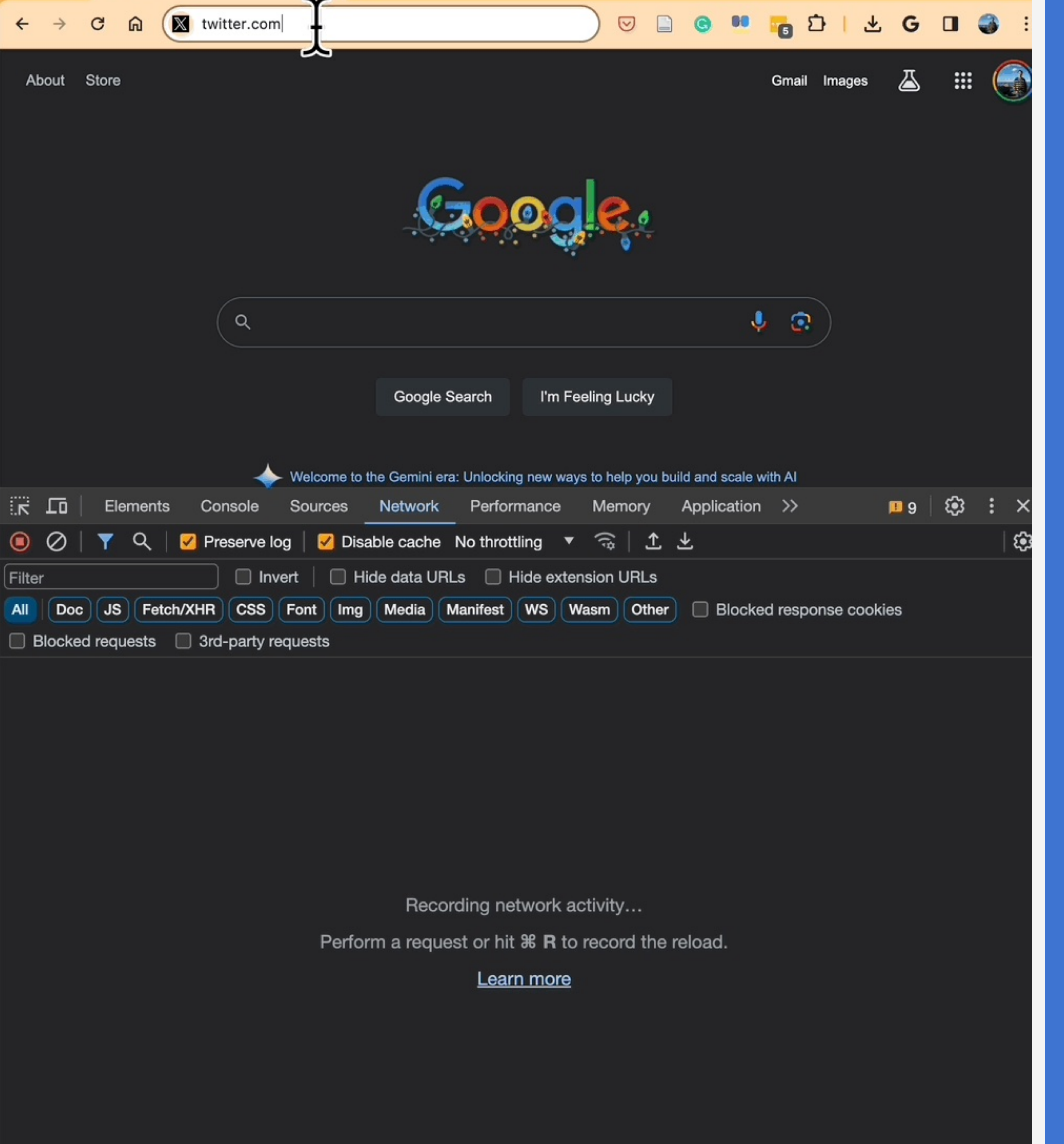


Five-minute rule

- Jim Gray and Franco Putzolu (1986)
- Pages referenced every five minutes should be memory resident
  - if you access some data more than once every 5 minutes, keep in memory
  - $\text{Break-even Interval (sec)} = \dfrac{\text{Price of Disk / Disk IOPS}}{\text{Price of RAM}}$
- 10-year later
  - disk capacity exploded, but speed did not improve much
  - memory price also dropped exponentially
  - the five-minute rule held! (but it is more like 5-10 minutes)

Different types of caches

Page cache
Key-value cache
Object cache



How many requests are served by caches?

1000s

A typical web application

API
gate
way

CDN

Service

Service

Service

Service

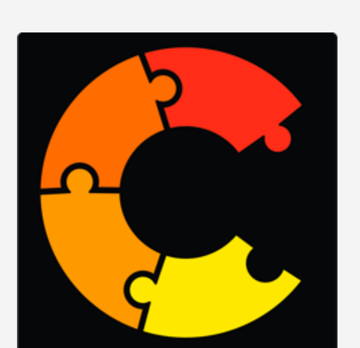
distributed
key-value
caches

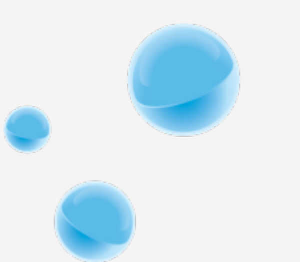
A typical web application

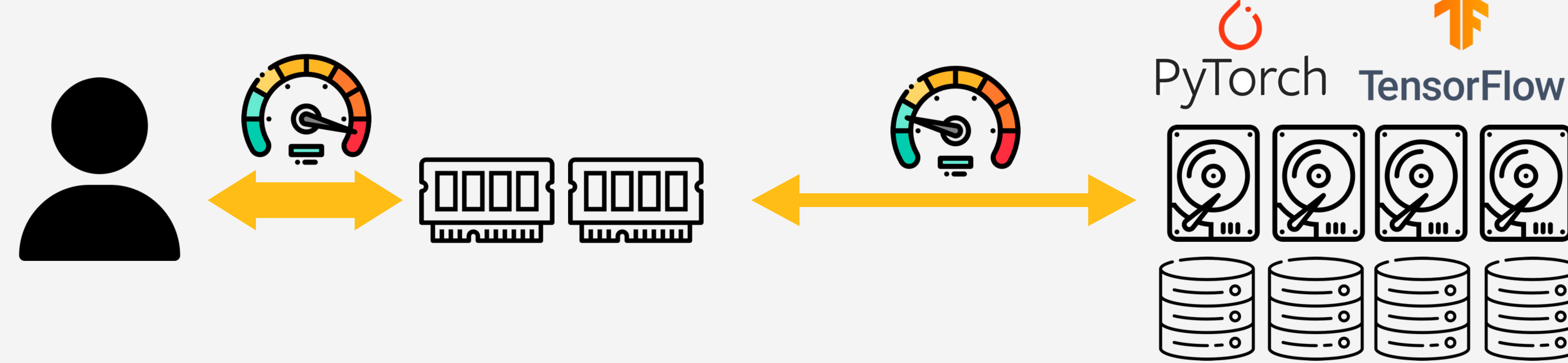
What the caches used for?

Ubiquitous deployment

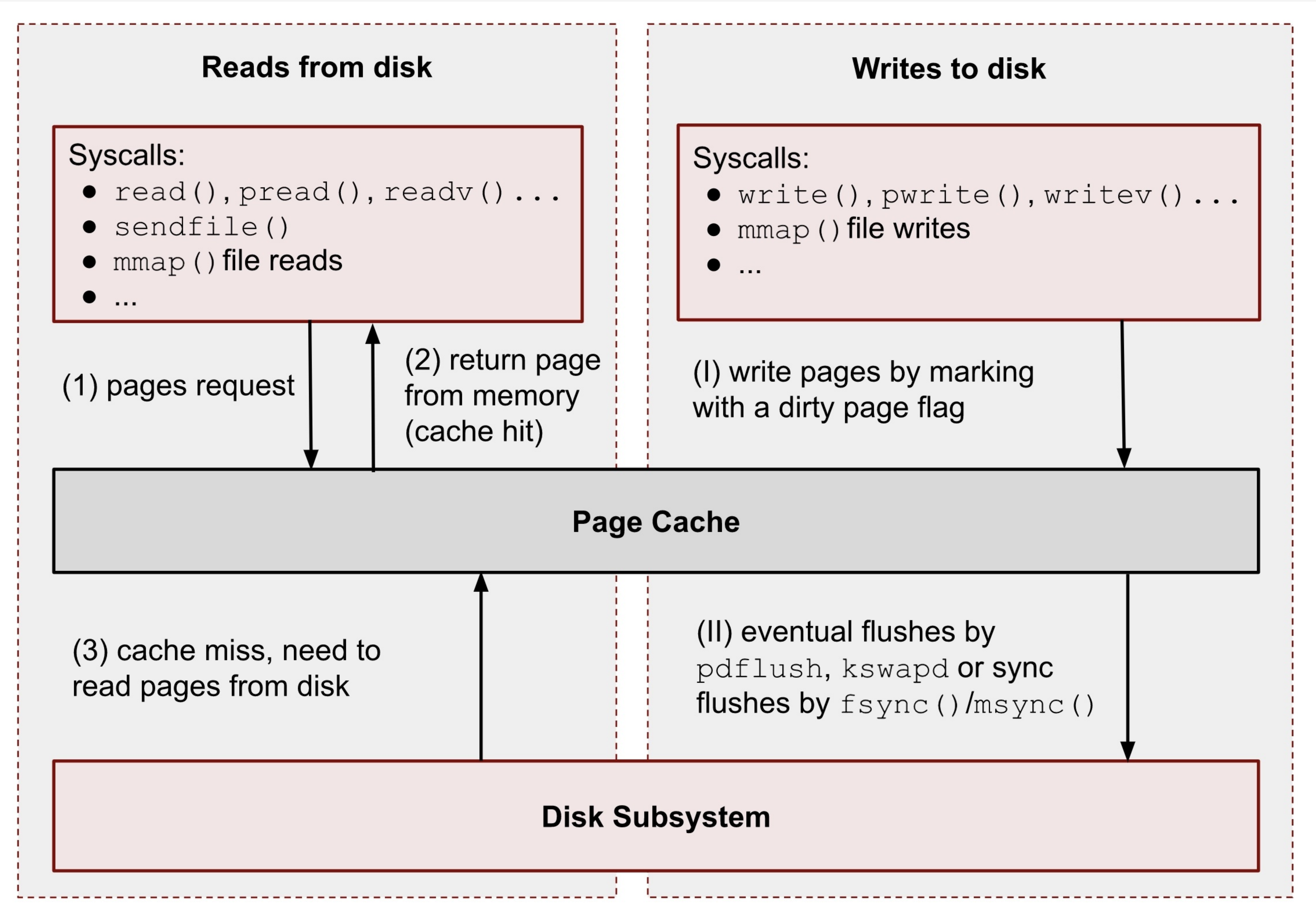
- speed up data access
- reduce data movement
- reduce repeated computation







Page cache

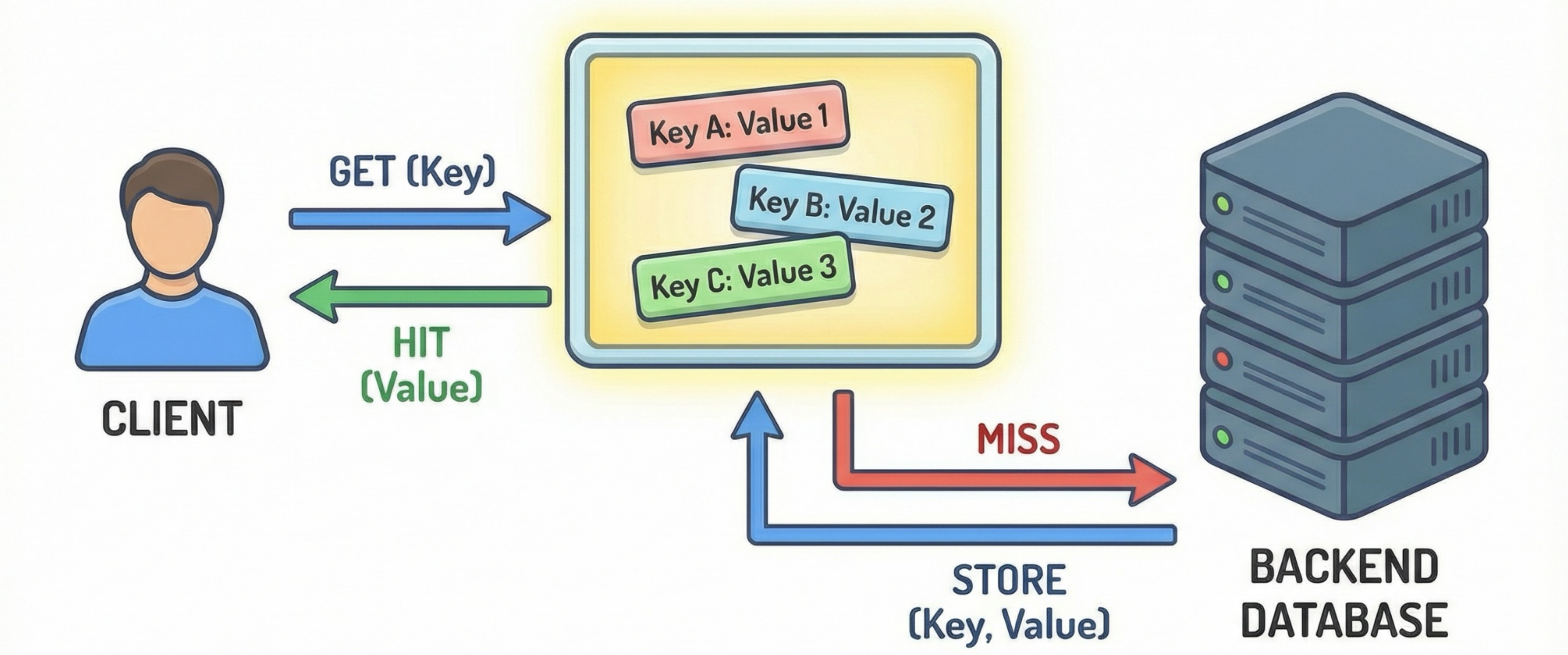


Page cache

- Unique characteristics
  - spatial locality
    - block i, i+1, i+2…
  - scan (streaming) access pattern
    - sequentially access a large mount of data
    - e.g., database table scan, back up, media streaming
  - loop
    - multiple rounds of scan

Key-value cache

- Interface: key-value
- Used to store temporary data
  - database queries
  - pre-computed results
- Most commonly in-memory and distributed
  - consistent hashing
  - lookaside cache
- Miss ratio is often low
  - 0.1%-10%

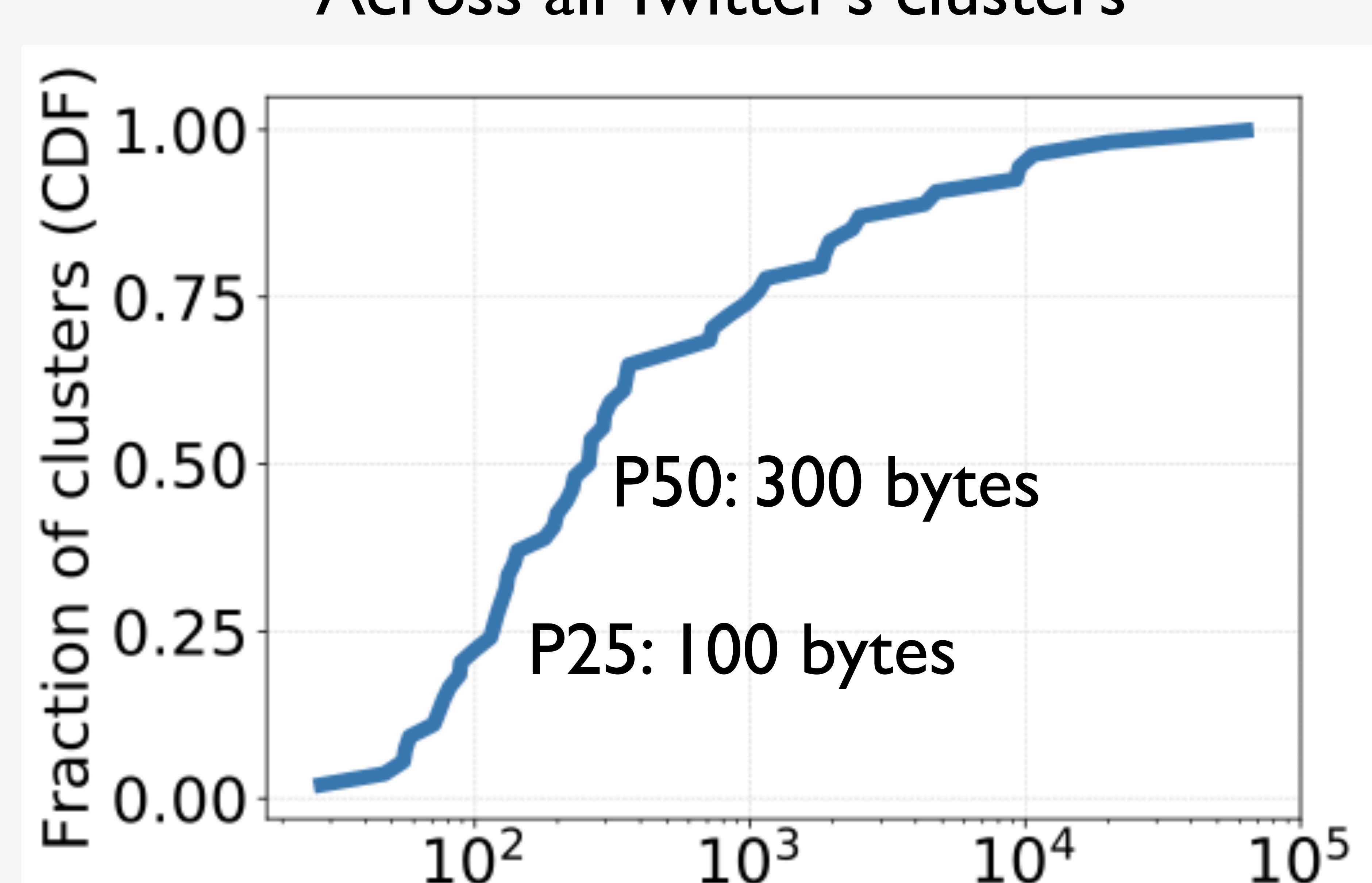


This is different from the KV-cache in LLM

Key-value cache

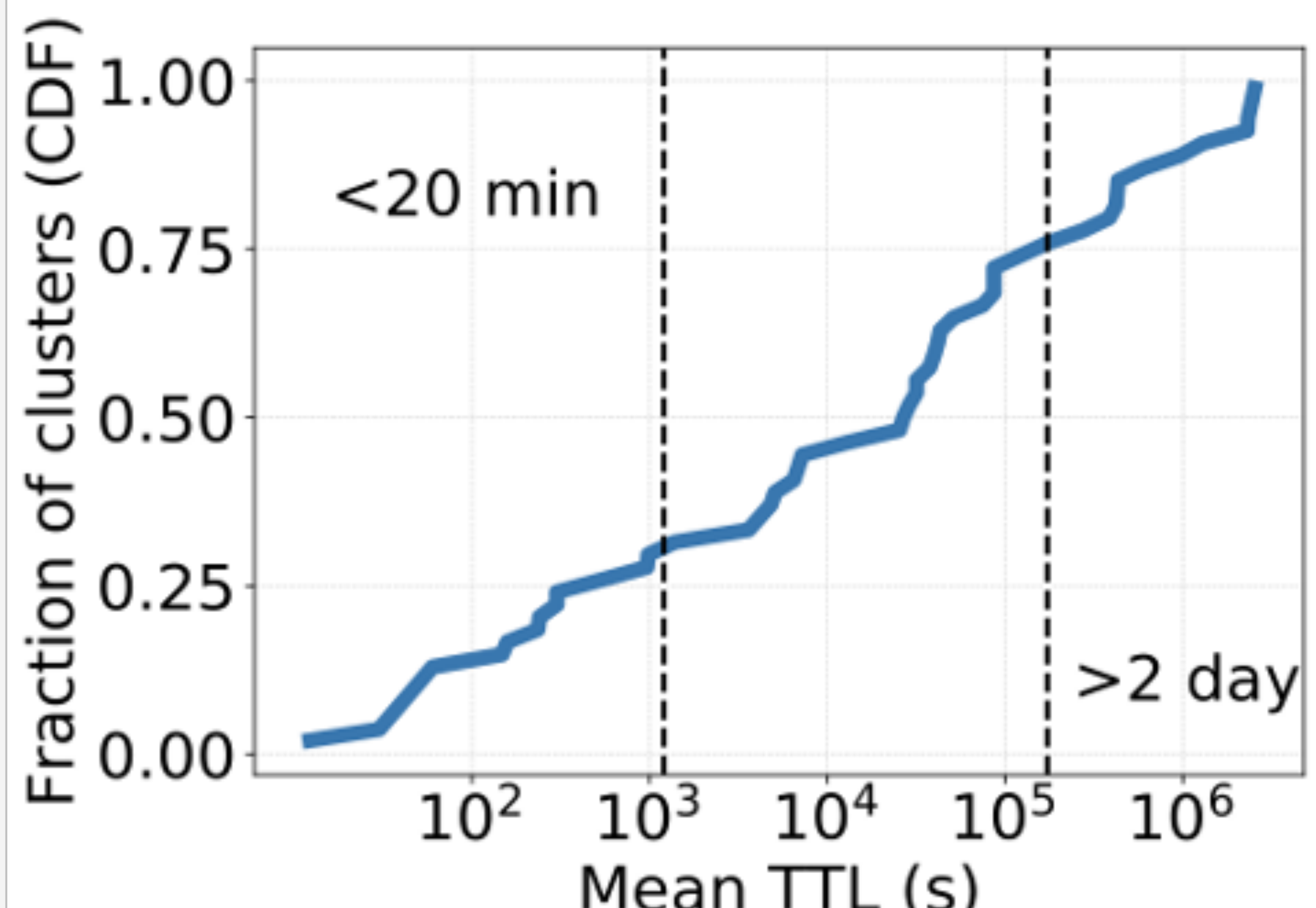
- Object size is small
  - Twitter's largest clusters
    - mean: ~200, ~50, ~200, ~70, ~200 bytes
  - design needs to consider metadata size
  - sensitive to overhead

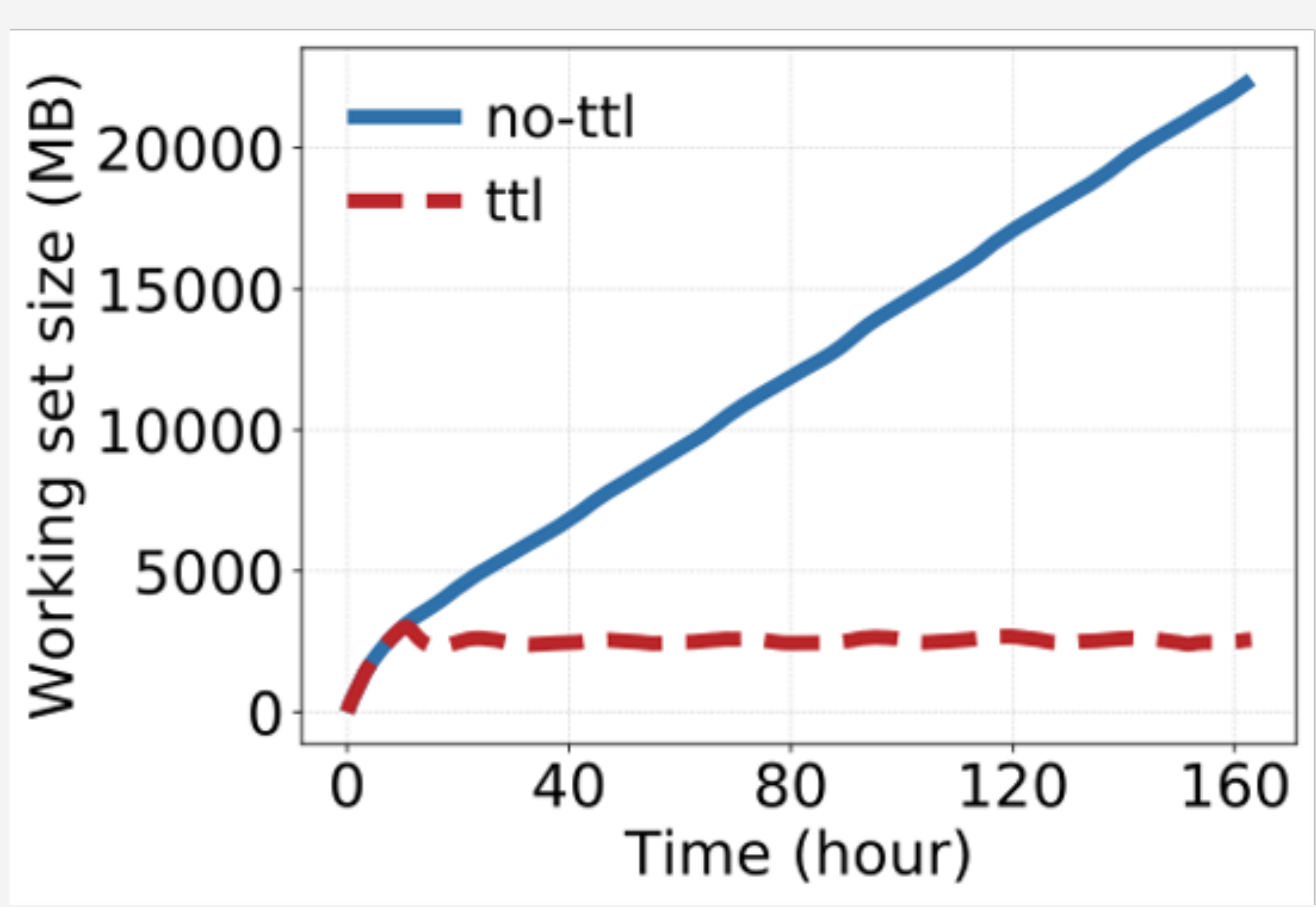
Across all Twitter's clusters



Key-value cache

- TTL is widely used
  - bound staleness: update are best-effort
  - periodic refresh
  - implicit deletion: GDPR
- how to quickly remove expired objects efficiently?
  - check multiple eviction candidates
  - scan
  - sample
  - TTL-friendly layout: Segcache[1]

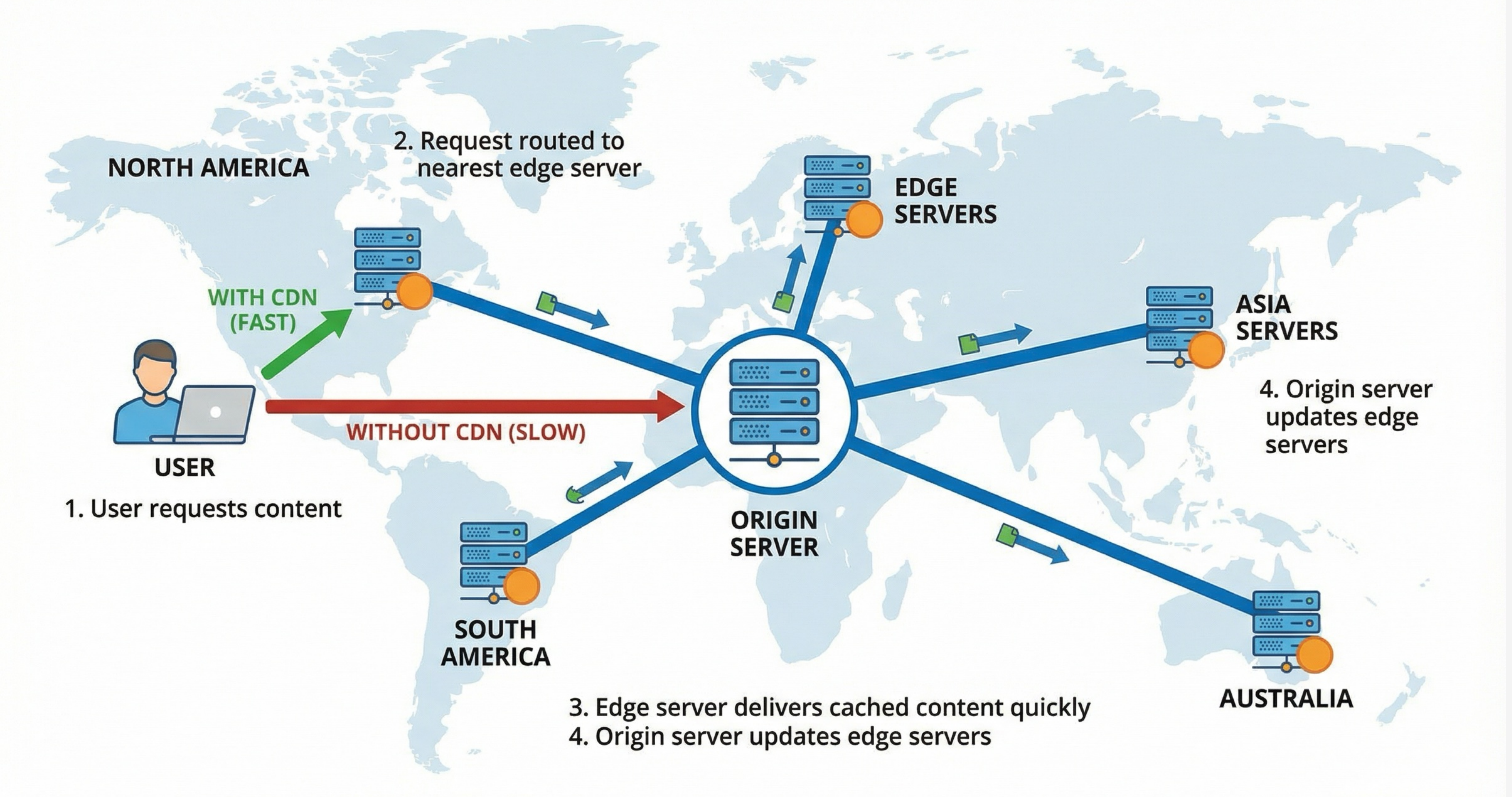




[1] Segcache: a memory-efficient and scalable in-memory key-value cache for small objects, NSDI'21

Object cache

- Most common: Content Delivery Networks (CDN)
- Objects are large and have large variance (KBs-GBs)



Object cache

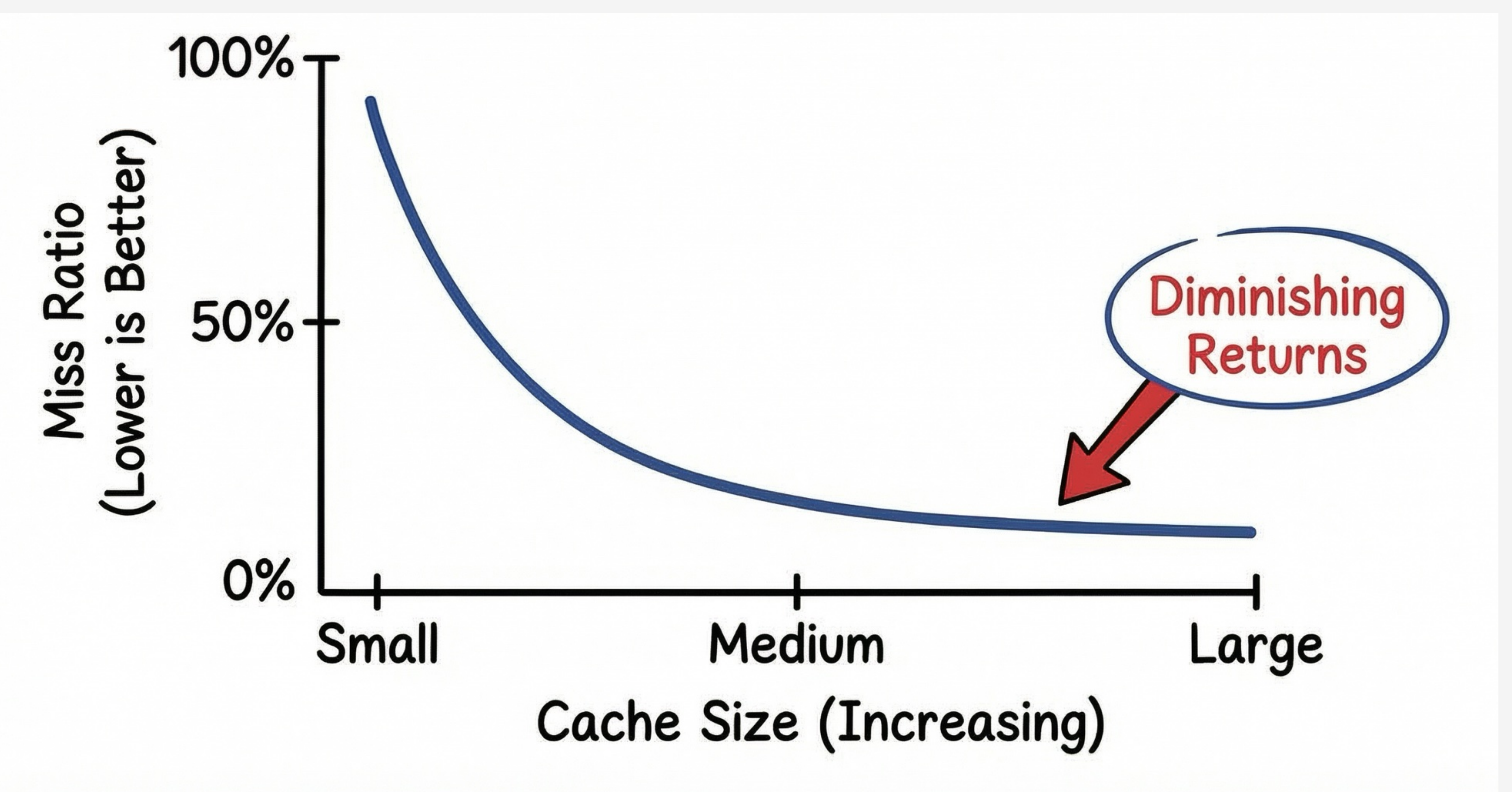
- Most common: Content Delivery Networks (CDN)
- Accessed via HTTP (often over the Internet)
  - cache-control headers
    - max-age, expires
    - no-cache, no-store, public vs private
    - ETag, last-modified
    - must-revalidate, stale-while-revalidate
  - many layers: browser, CDN layer 1, CDN layer 2, origin cache
- Miss ratio
  - 30-90% depends on layers and workloads

Miss Ratio Curve

LRU and stack property
SHARDS: spatial sampling

Miss ratio curve

- What is a miss ratio curve
  - miss ratio vs size
- Why do we need a miss ratio curve
  - cache sizing
  - cache partitioning
  - compare algorithms
  - detecting scans



Constructing miss ratio curves

- Brute force: simulate a cache at each size
  - slow (time complexity)
  - huge memory (space complexity)
    - simulate a cache 40byte / object * 100M object = 4 GB

Constructing miss ratio curves

- SHARDS: spatial sampling (sample 1% of objects)
  - compensate request for hot objects, i.e., expected #req = 1% #req
- Temporal sampling SHOULD NOT be used for caching / storage
  - more popular objects will be sampled and causes bias

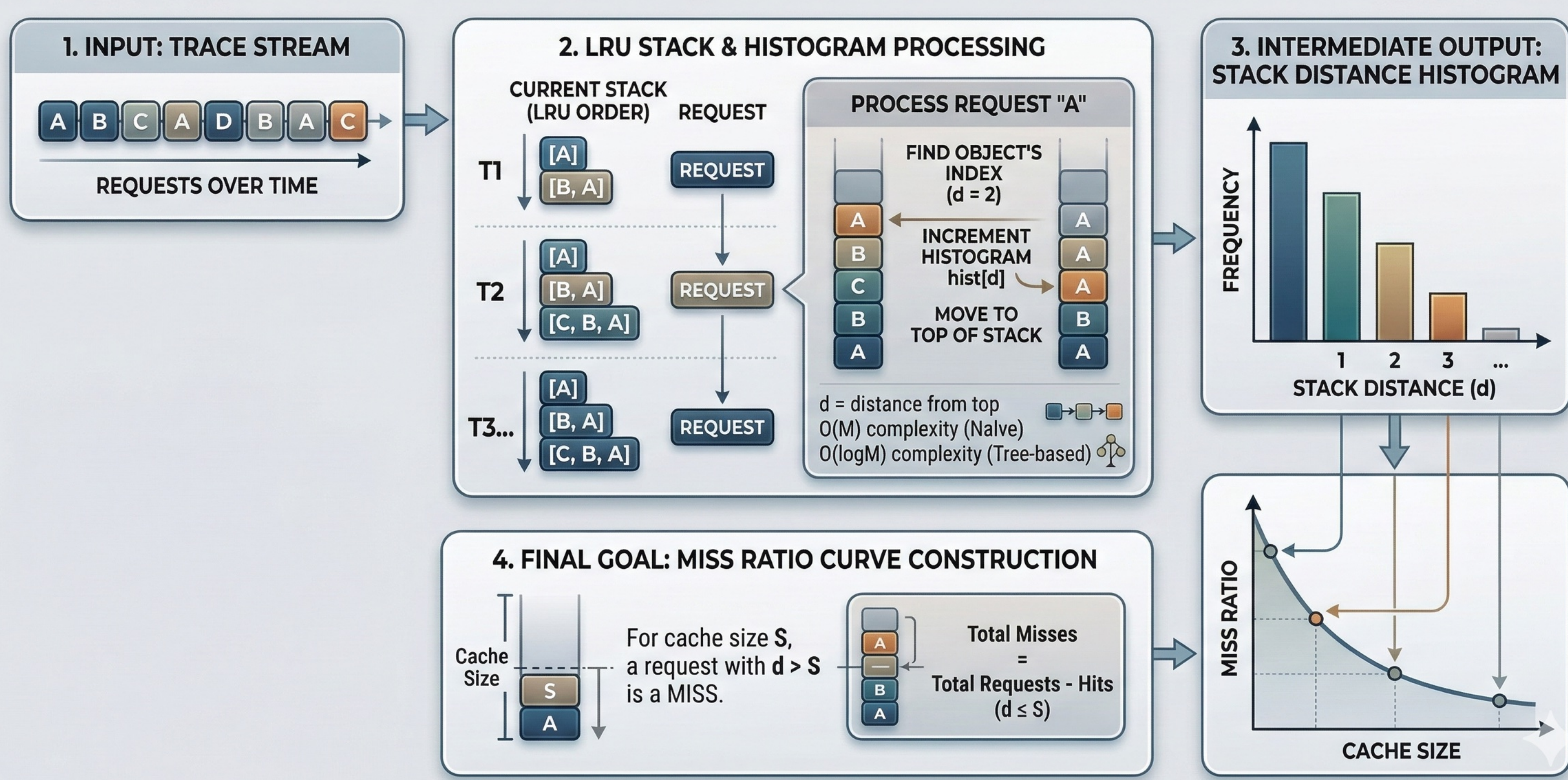
Constructing miss ratio curve for LRU: Stack Property

- Stack property (inclusion property)
  - the set of pages in a cache of size *s* is always a subset of the pages in a cache size *s+1*
  - implications: increasing cache size NEVER increases the miss ratio
- Stack algorithms
  - LRU, LFU, Belady
  - most algorithms are not
- Stack distance (reuse distance)
  - #objects between two requests
  - `A B C B A`: stack distance between two A is 2
  - if cache size >= 2, `A` will be a hit

Constructing miss ratio curve for LRU

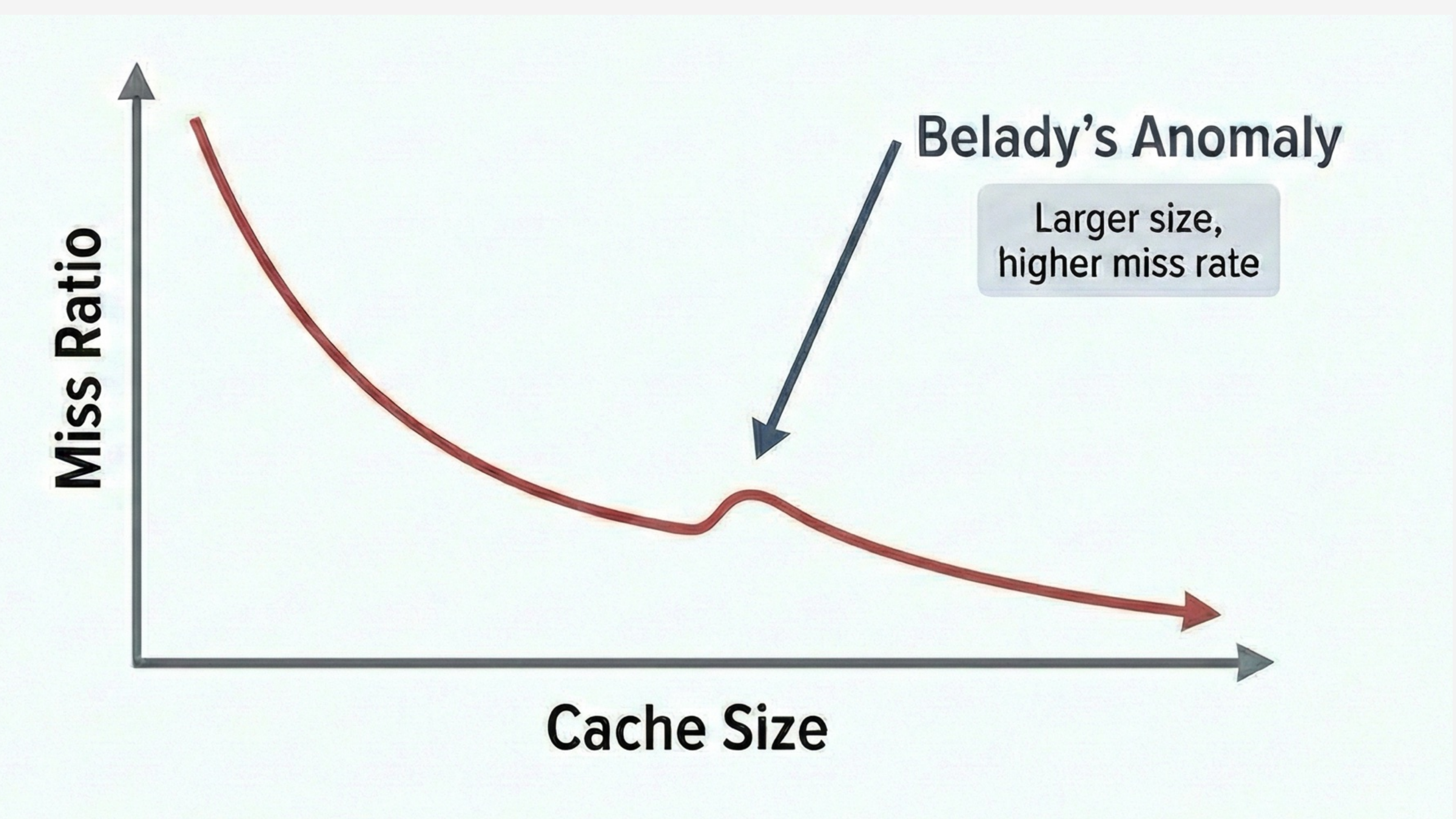
- Construct miss ratio curve for LRU
  - calculate stack distance histogram *hist*
  - $\#hits = \sum_{i=0}^{size} hist[i]$
- Calculate stack distance histogram
  - maintain a stack
  - process trace
    - find item's current index in the stack (*d*)
    - increment histogram *hist[d]*
    - move item to the top of stack
  - naive implementation: using linked list for stack, *O(M)* for finding index (M is #object)
  - efficient implementation: using a tree, *O(logM)* time complexity

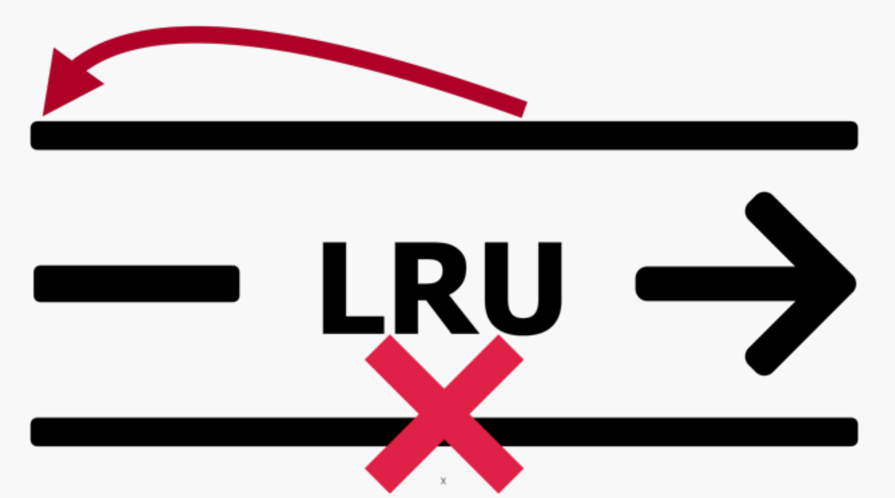
CONSTRUCTING LRU MISS RATIO CURVE



Belady's anomaly

- Belady's anomaly
  - increasing cache size leads to increasing miss ratio
- Cliffs
  - a tiny increase in cache size leads to a sharp miss ratio decrease
  - scan resistance





Eviction algorithm

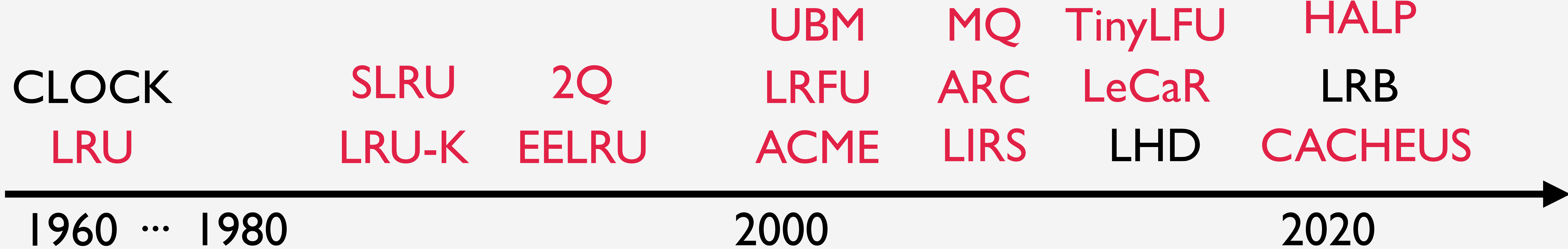
First-in-first-out
Least-recently-used
CLOCK
S3-FIFO
SIEVE

Cache eviction algorithm basics

- Key assumption
  - temporal locality
- Least-recently-used (LRU)
- First-in-first-out (FIFO)
- Belady
  - offline optimal for uniform-sized items
  - evict the object that will be requested the furthest in the future

The need for simple and scalable cache eviction algorithm

- 60+ years of research on designing eviction algorithms: most are LRU-based
  - not scalable
  - increasingly complex



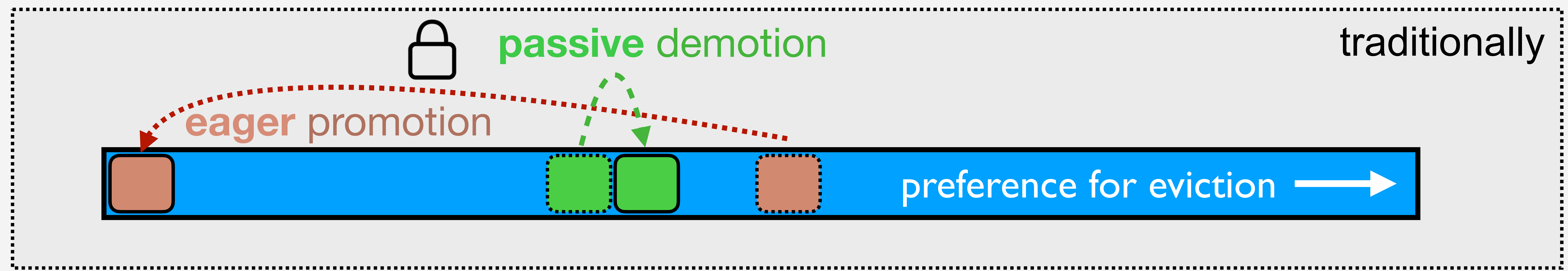
A simple algorithm: FIFO eviction algorithm

- First-in-first-out (FIFO)
  - simpler than LRU
  - fewer metadata
  - no computation
  - more scalable
  - flash-friendly

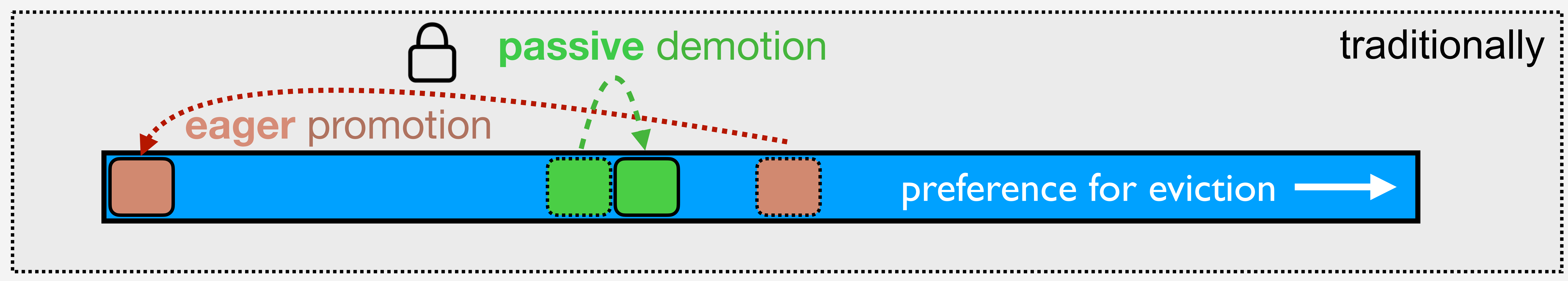
The only drawback:

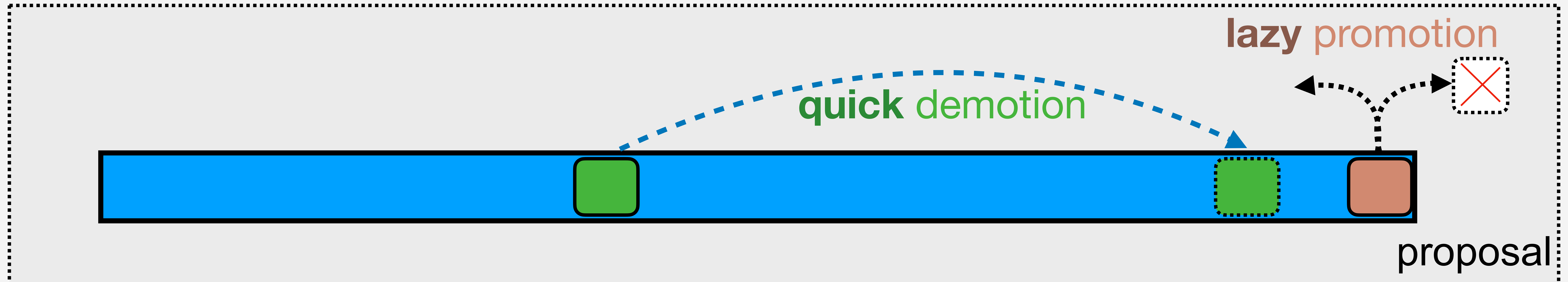
FIFO has a high miss ratio

Existing eviction algorithms focus on promotion

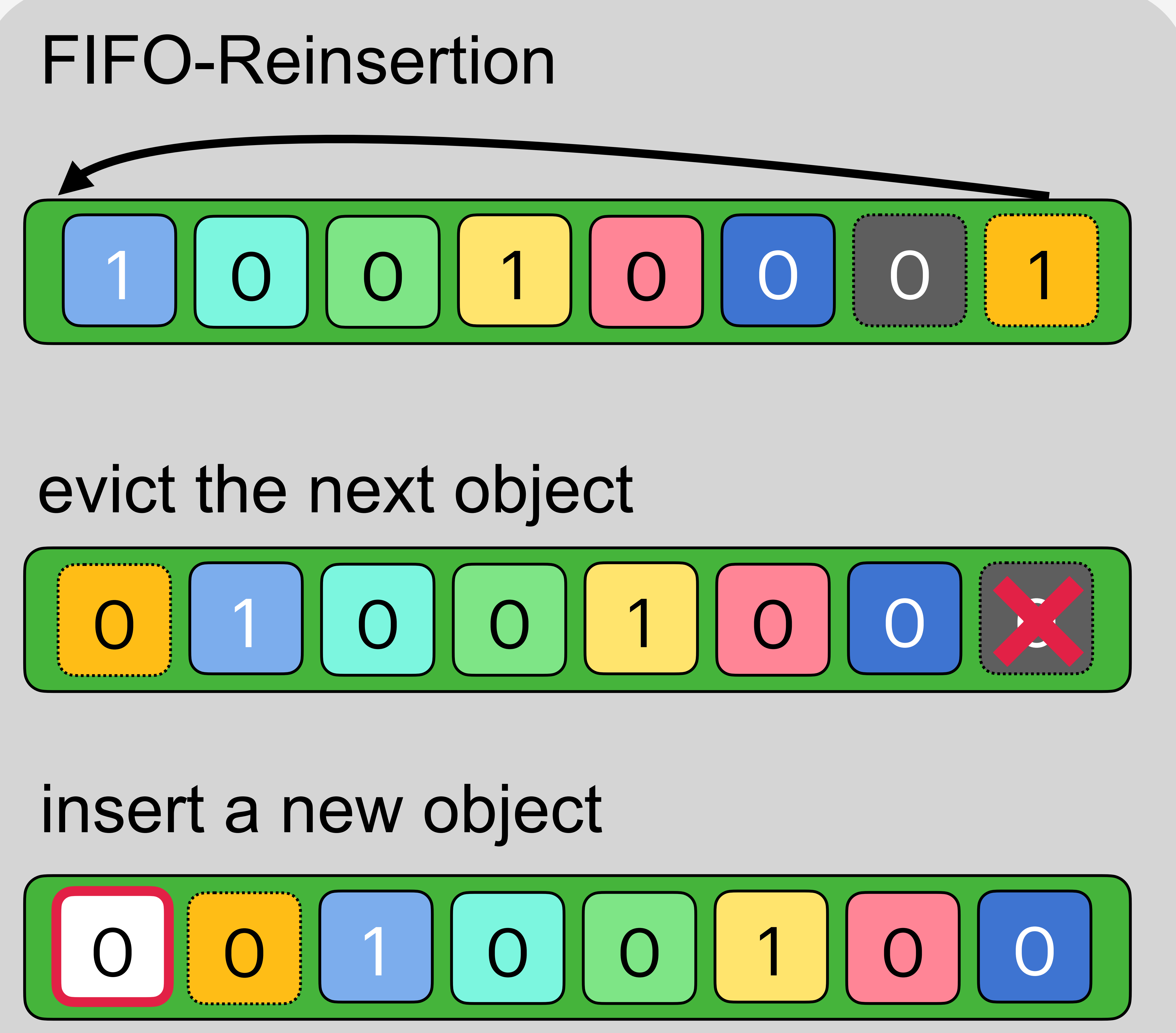


Demotion should be the first-class citizen





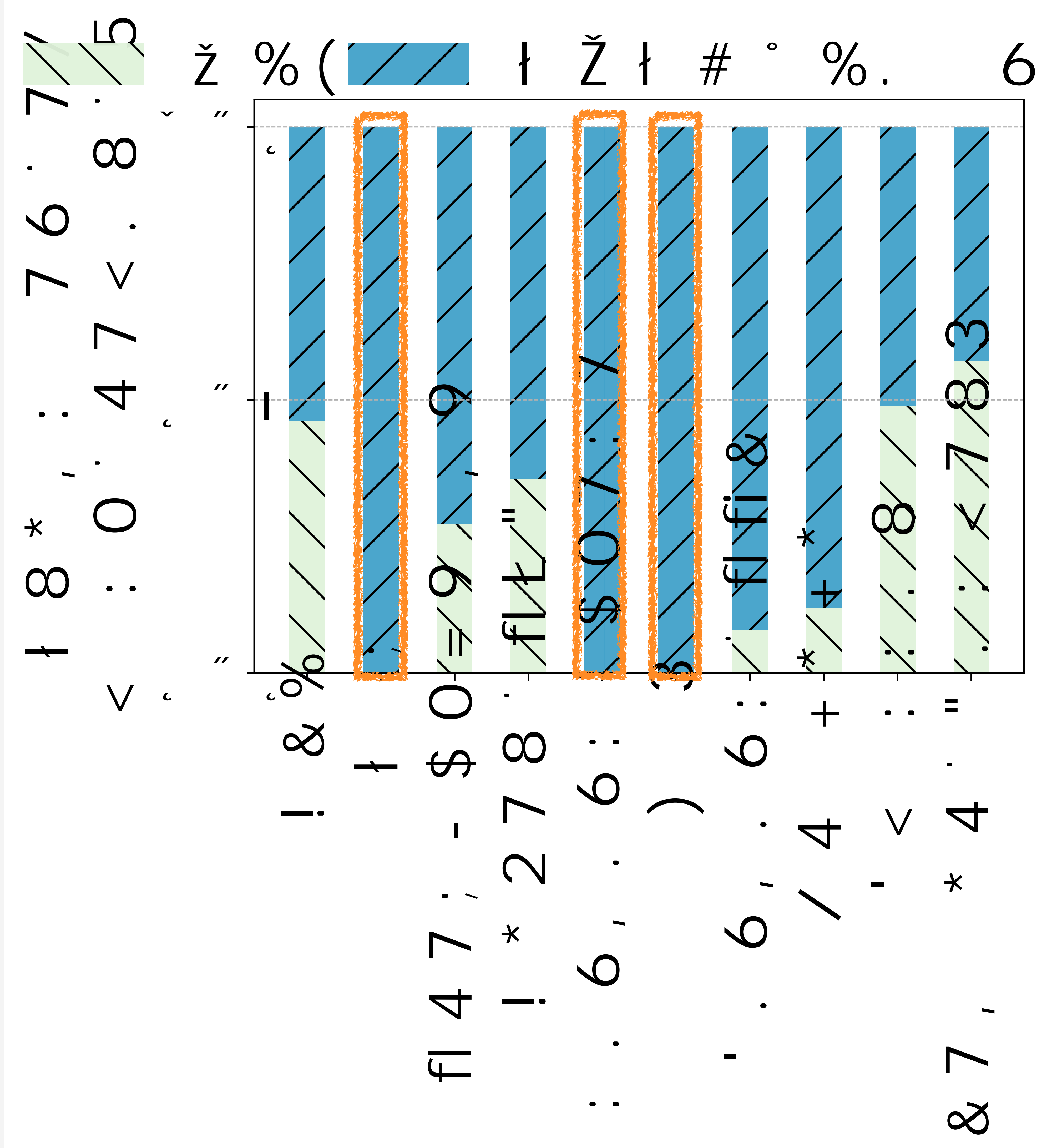
Lazy promotion: only promote during eviction



Benefits of lazy promotion

- less computation
- better decision
- more efficient (lower miss ratio)

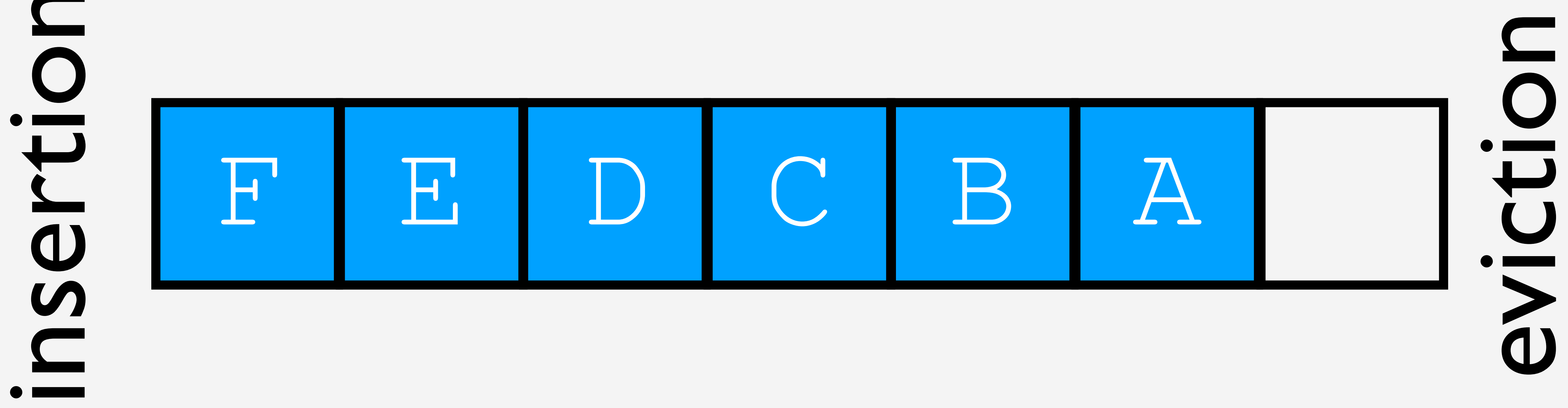
Lazy promotion: only promote during eviction



Why is FIFO-Reinsertion more efficient?

evict new objects faster!

Requests: B N A X



Objects push N towards eviction	
LRU	A X
FIFO-Reinsertion	A X B

Quick demotion: quickly evict new objects

- One-hit wonders: objects appeared once in the sequence
- Cache workloads: *shorter request sequences have larger one-hit-wonder ratios*

time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
request	A	B	A	C	B	A	D	A	B	C	B	A	E	C	A	B	D

start time	end time	sequence length (# objects)	# one-hit wonder	one-hit wonder ratio
1	17	5	1 (E)	20%
1	7	4	2 (C, D)	50%

Quick demotion: quickly evict new objects

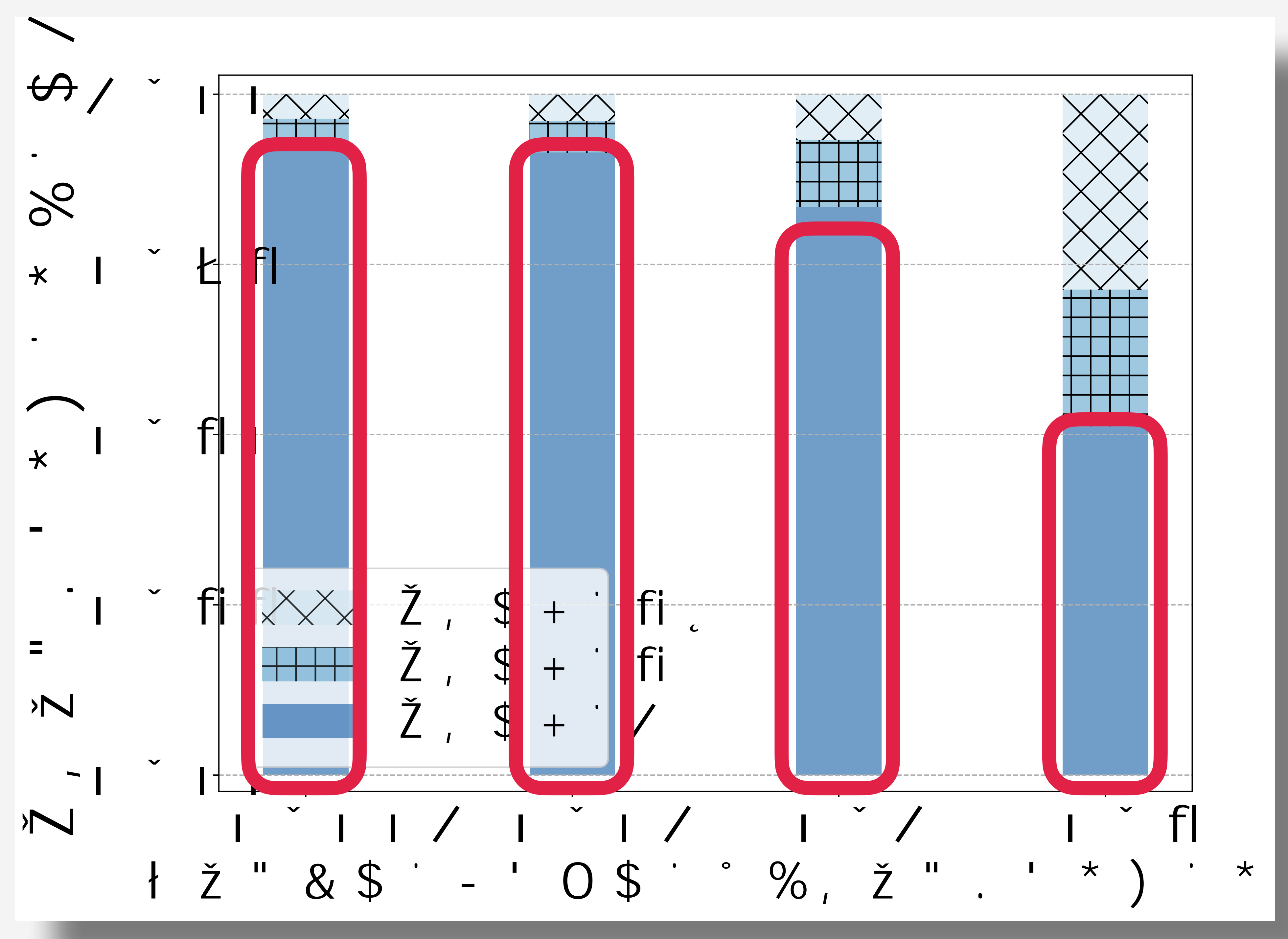
- One-hit wonders: objects appeared once in the sequence
- Cache workloads: *shorter request sequences have larger one-hit-wonder ratios*

One-hit-wonder ratio of week-long traces at 10% length:
72% (mean on **6594** traces)

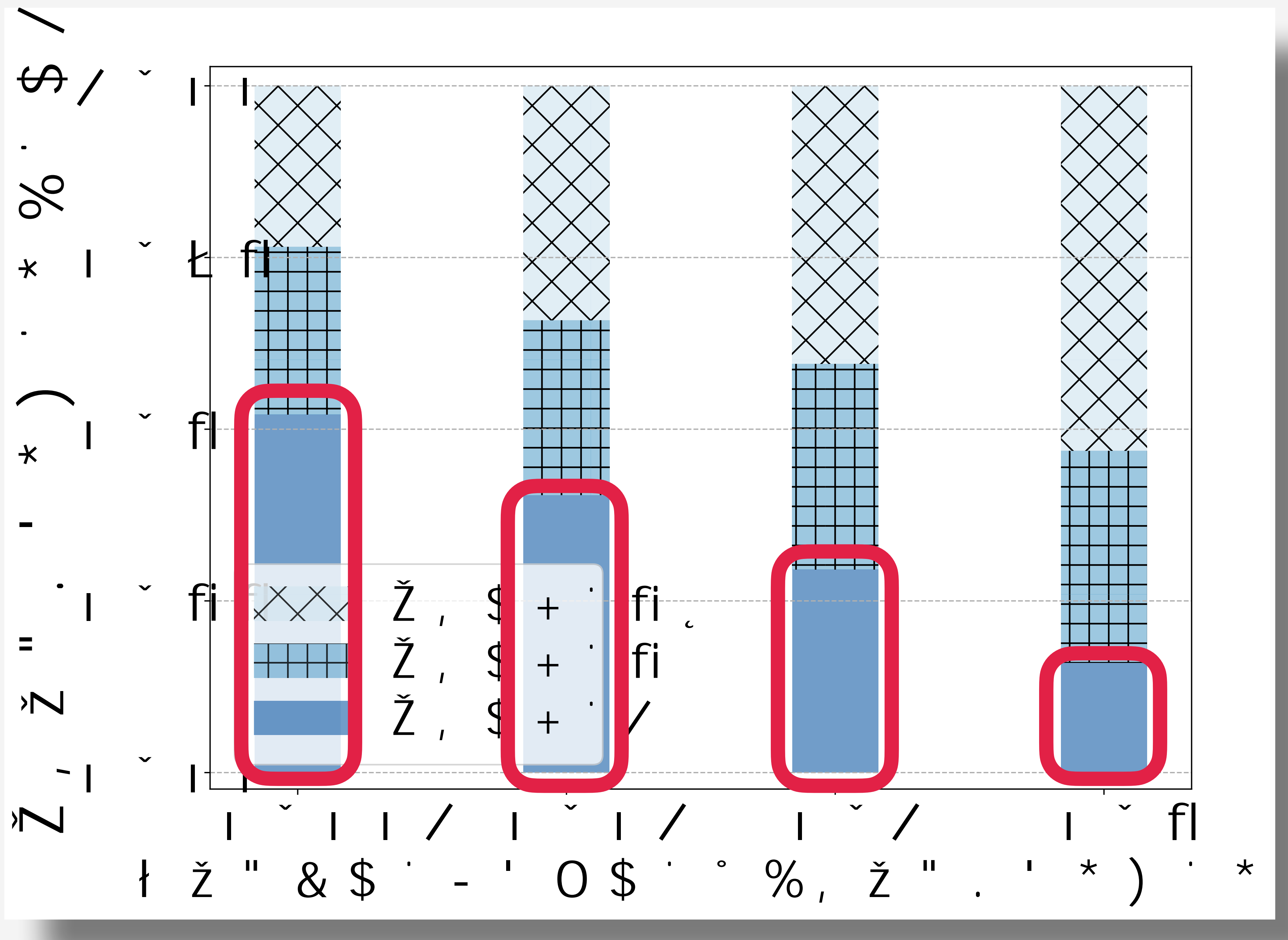
Implication: most objects are not used before evicted

Observation

Most objects are not reused before evicted



LRU cache running MSR workload



LRU cache running Twitter workload

Quick demotion: quickly evict new objects

ARC: A SELF-TUNING, LOW OVERHEAD REPLACEMENT CACHE

Nimrod Megiddo and Dharmendra S. Modha
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

four LRU queues
+ adaptive algorithm

LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance *

three LRU queues
+ a new metric

Cliffhanger: Scaling Performance Cliffs in Web Memory Caches

Asaf Cidon[1], Assaf Eisenman[1], Mohammad Alizadeh[2], and Sachin Katti[1]

LRU + partitioning

TinyLFU: A Highly Efficient Cache Admission Policy

LHD: Improving Cache Hit Rate by Maximizing Hit Density

Nathan Beckmann, Haoxian Chen, Asaf Cidon

HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network

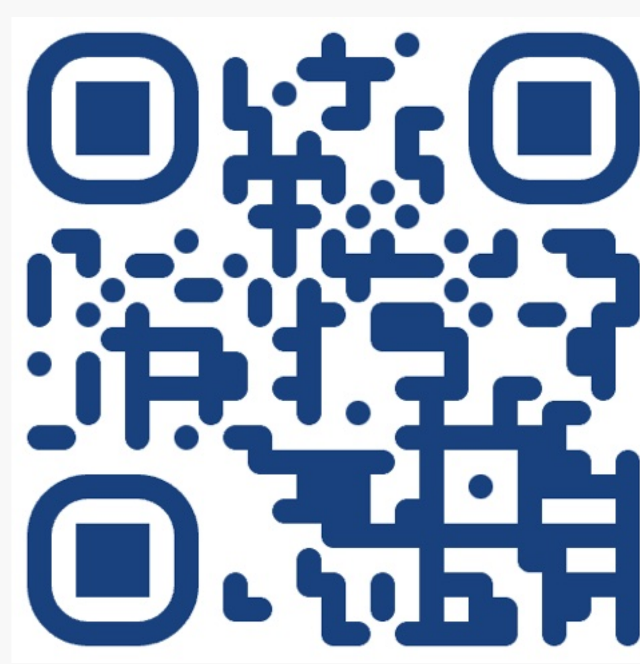
The secret sauce of state-of-the-art algorithms:

evicting new objects very aggressively

S3-FIFO Design

<u>S</u>imple, <u>S</u>calable caching with <u>three</u> <u>S</u>tatic <u>FIFO</u> queues

https://s3fifo.com

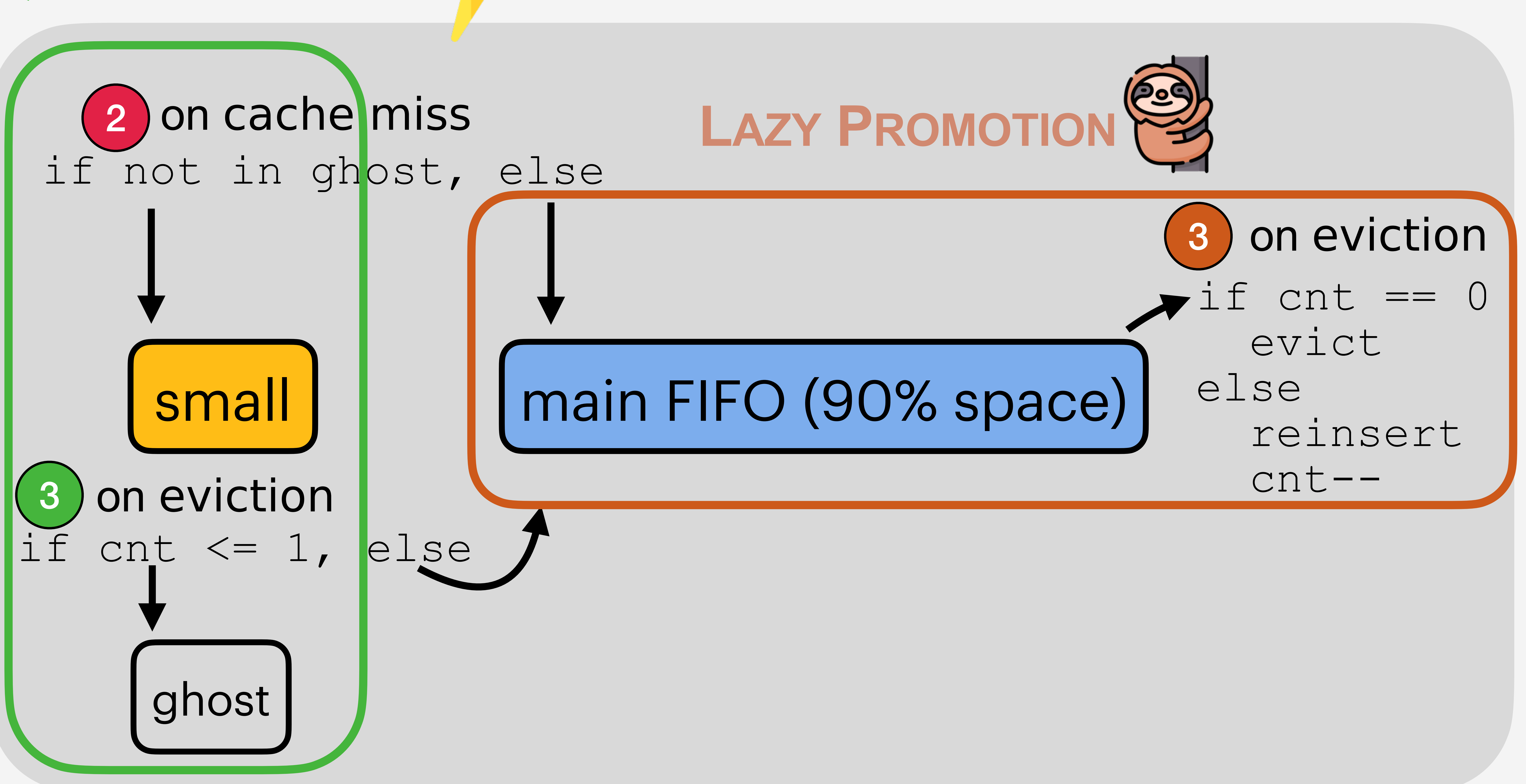


S3-FIFO design

```
struct object {
  …
  uint8_t cnt:2;
}
```

1 on cache hit

```
cnt++
```

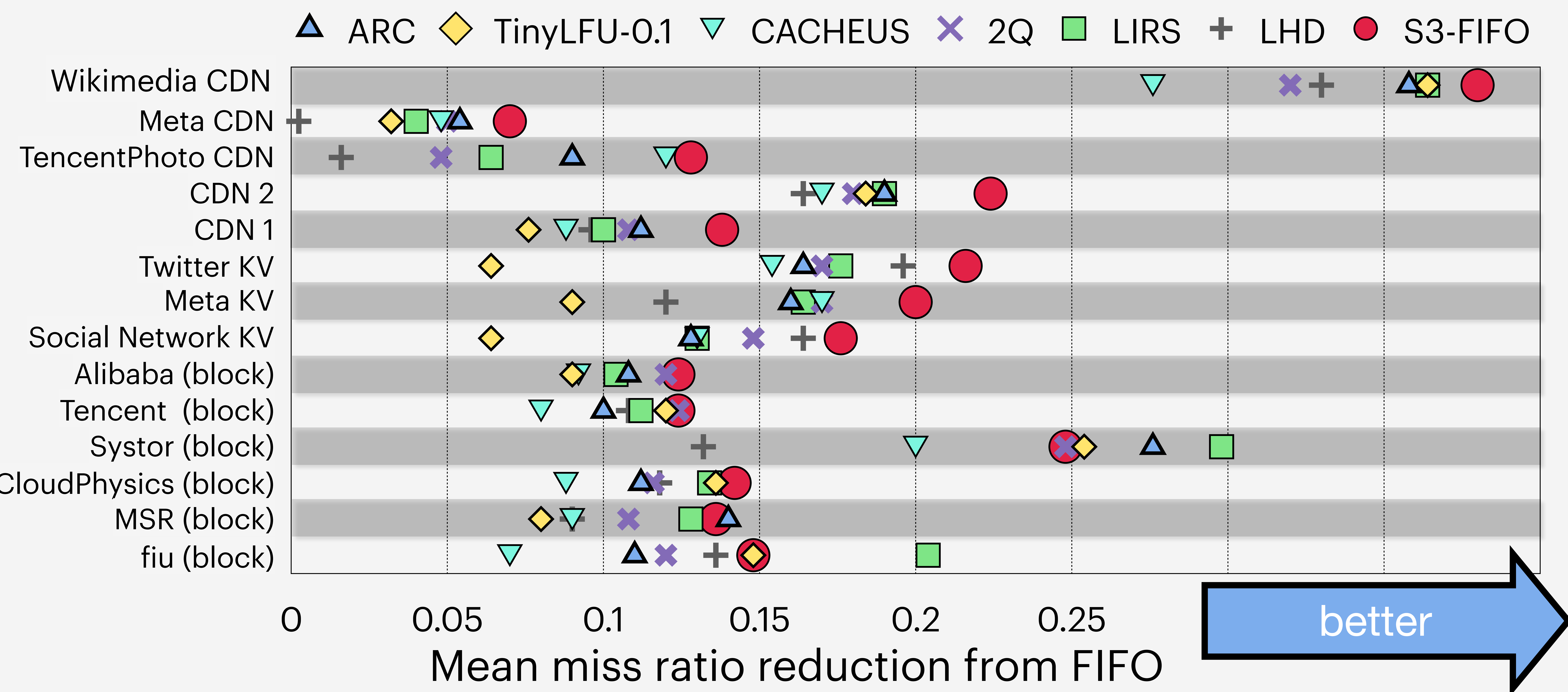


S3-FIFO features

- **Simple and robust**: static queues
- **Fast**: no metadata update for most requests
- **Scalable**: no lock
- **Tiny metadata**: 2 bits
- **Flash-friendly**: sequential writes

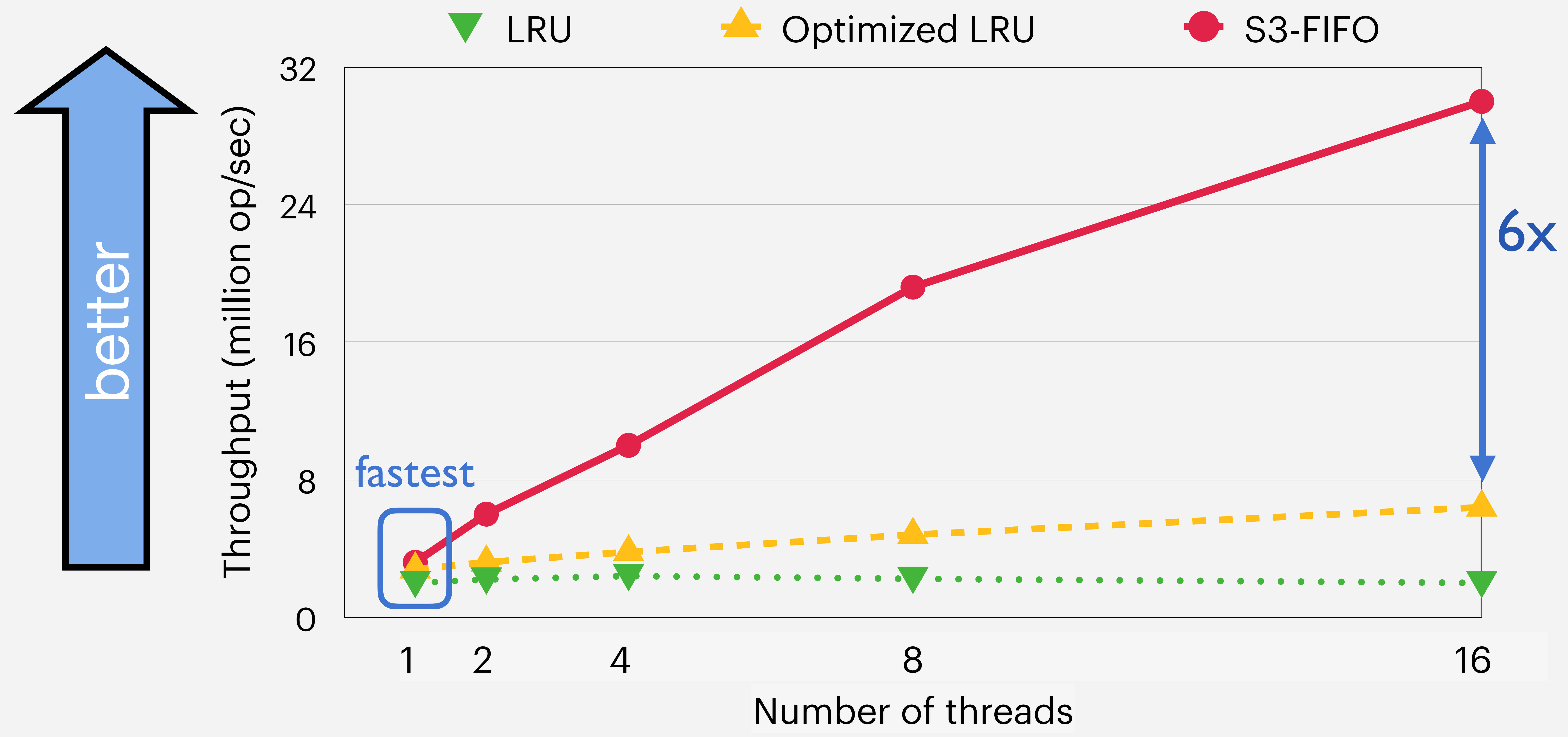
Can be implemented using one, two or three FIFO queues

S3-FIFO is efficient across datasets



Evaluated on 6594 traces with 848 billion requests from 12 sources, collected between 2007 and 2023
This evaluation is million times larger than previous works

Throughput scales with number of threads

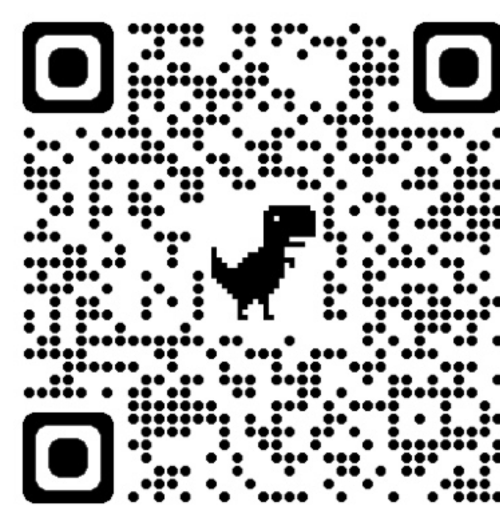


Implemented in Meta CacheLib, synthetic Zipf 1.0 workload, benchmarked on Cloudlab r6420 node

SIEVE

An eviction algorithm simpler than LRU

https://sieve-cache.com



The secret to designing efficient eviction algorithms

Lazy Promotion

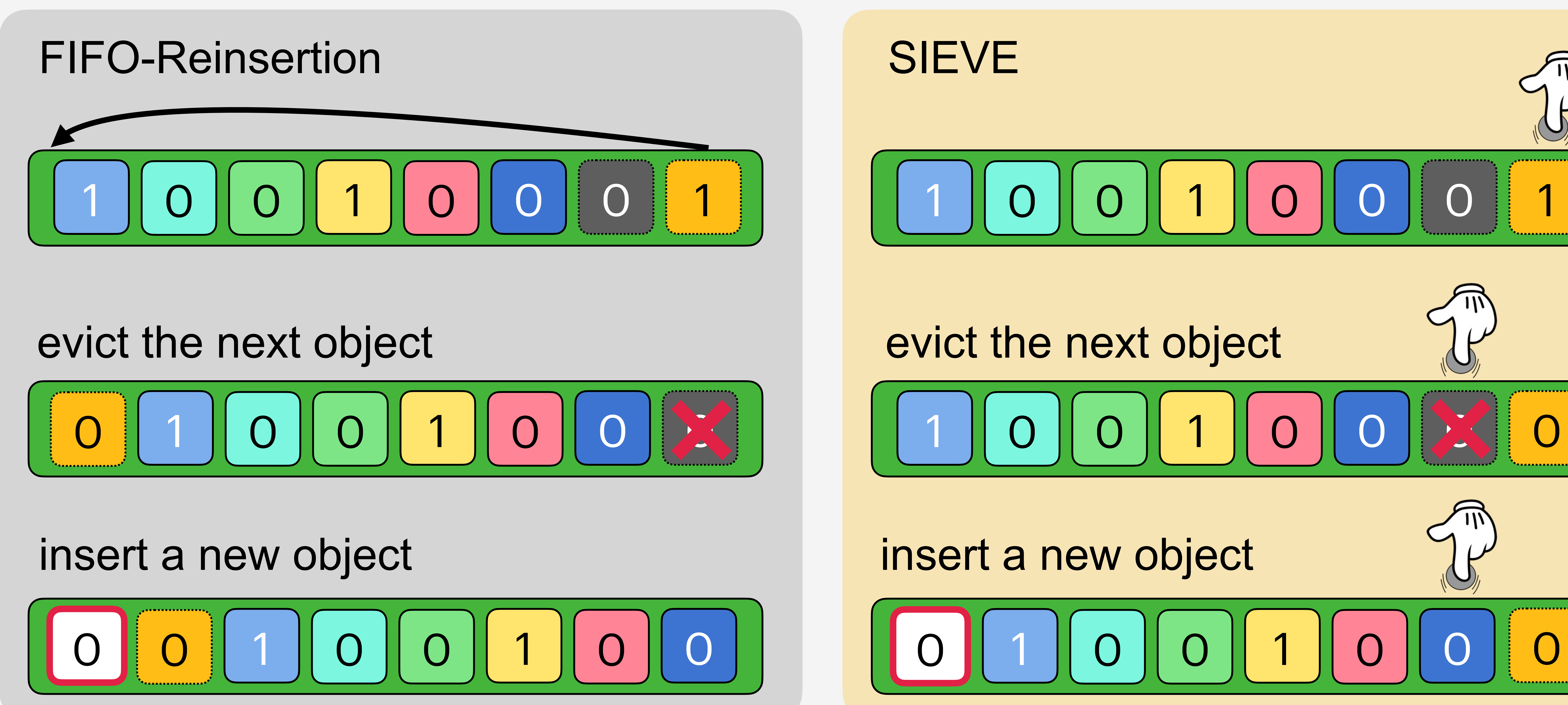
Retain popular objects with minimal effort

Quick Demotion

Remove unpopular objects fast, such as one-hit-wonders

SIEVE: combining lazy promotion and quick demotion

A small change turn FIFO-Reinsertion to SIEVE



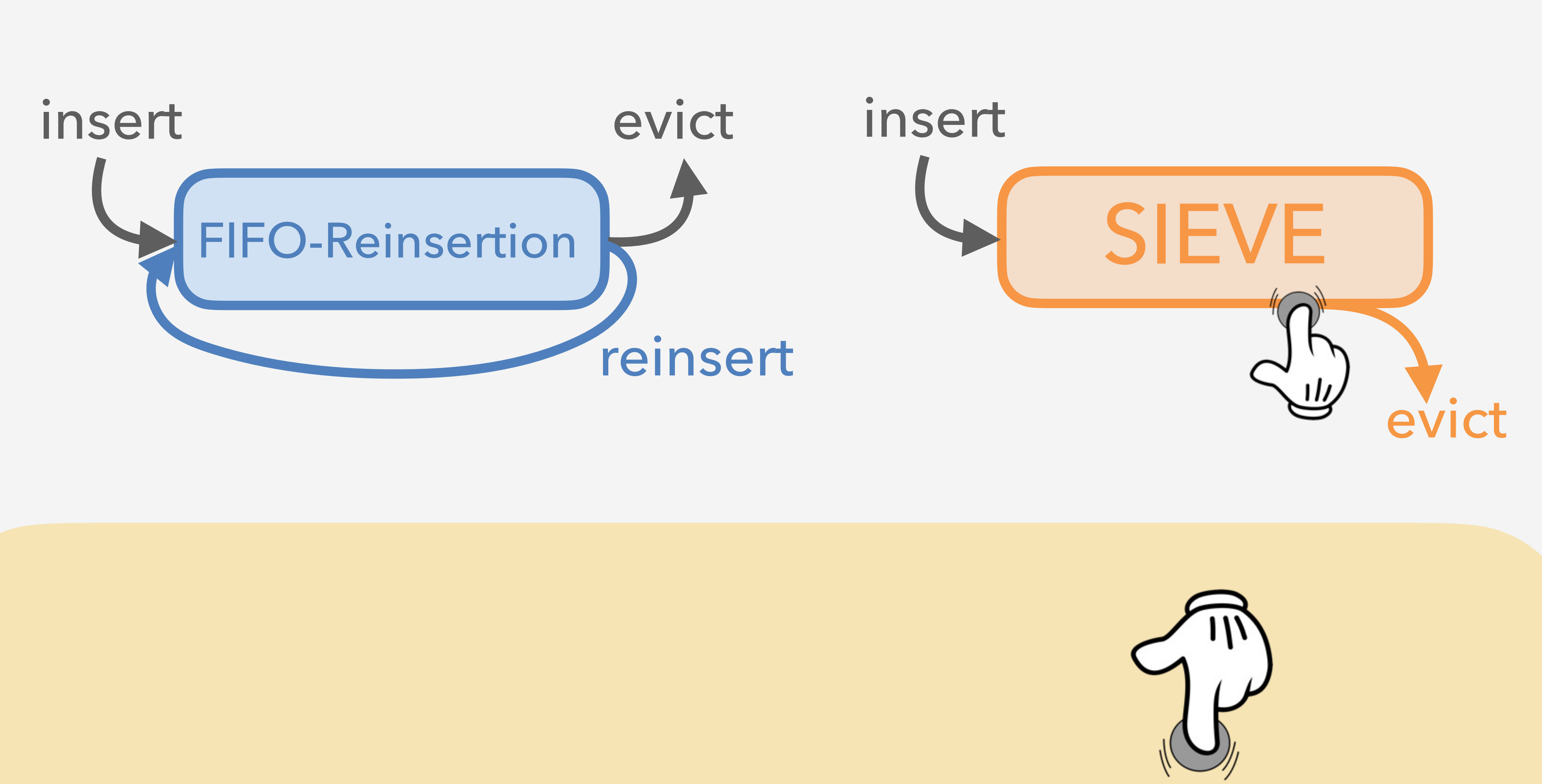
0: not accessed, 1: has accessed

SIEVE features

- Extremely simple
- ZERO parameter
- Fast and scalable
- Small per-object metadata
- TTL-friendly

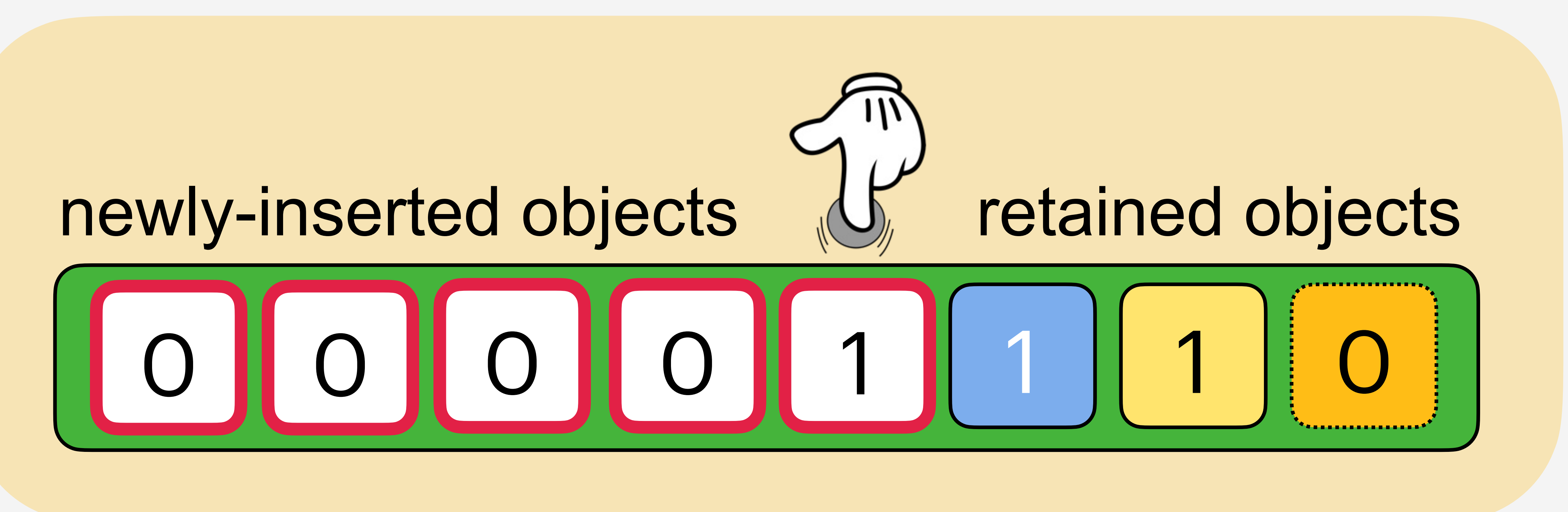
Why does SIEVE work?

- Retain popular objects effectively



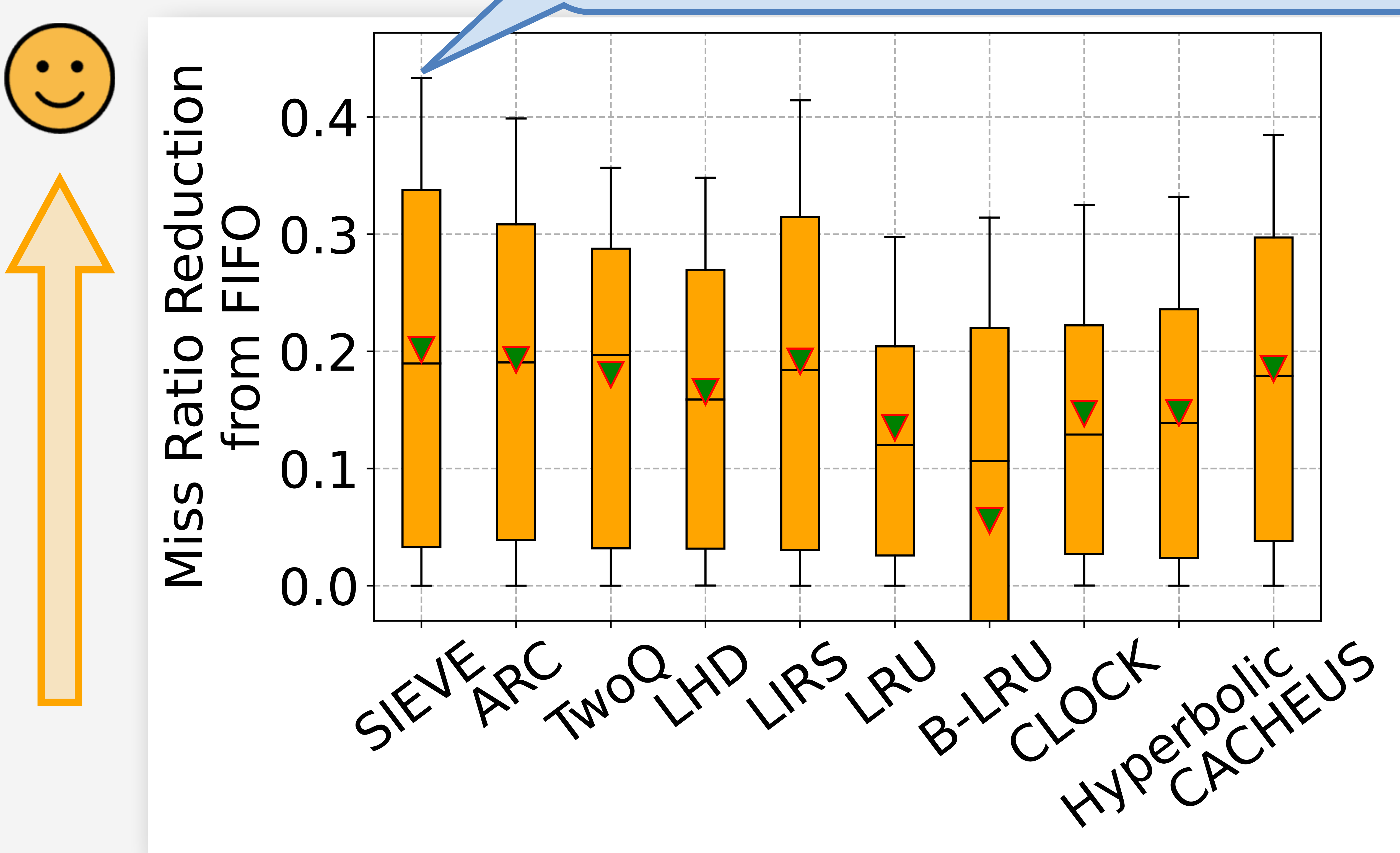
- Evict new objects quickly

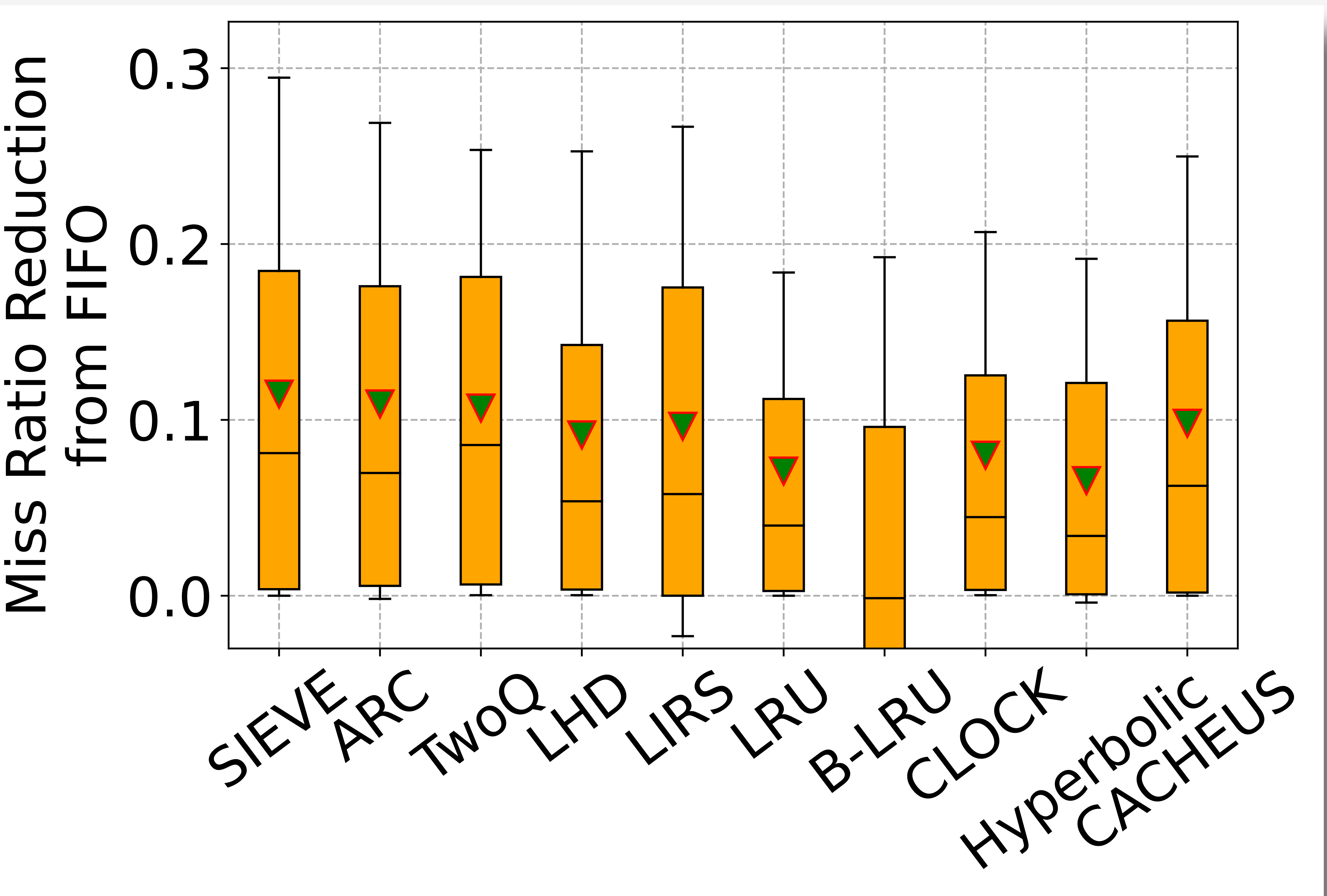
- Separate new and old objects



SIEVE achie… efficiency

SIEVE reduces FIFO's miss ratio by more than 42% on 10% of the traces (top whisker) with a mean of 21%

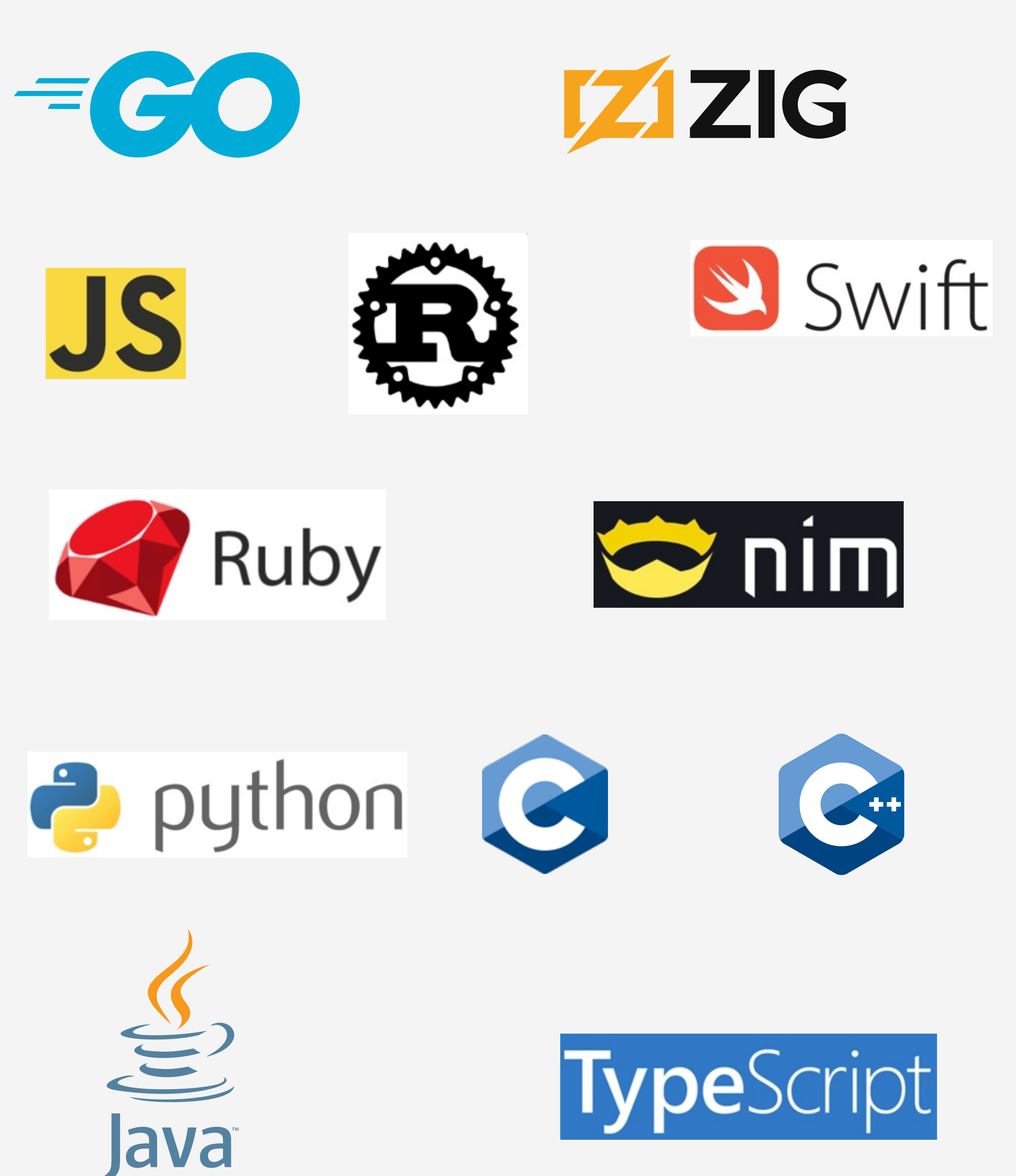




SIEVE also achieves the lowest miss ratio on the well-studied Zipfian workloads

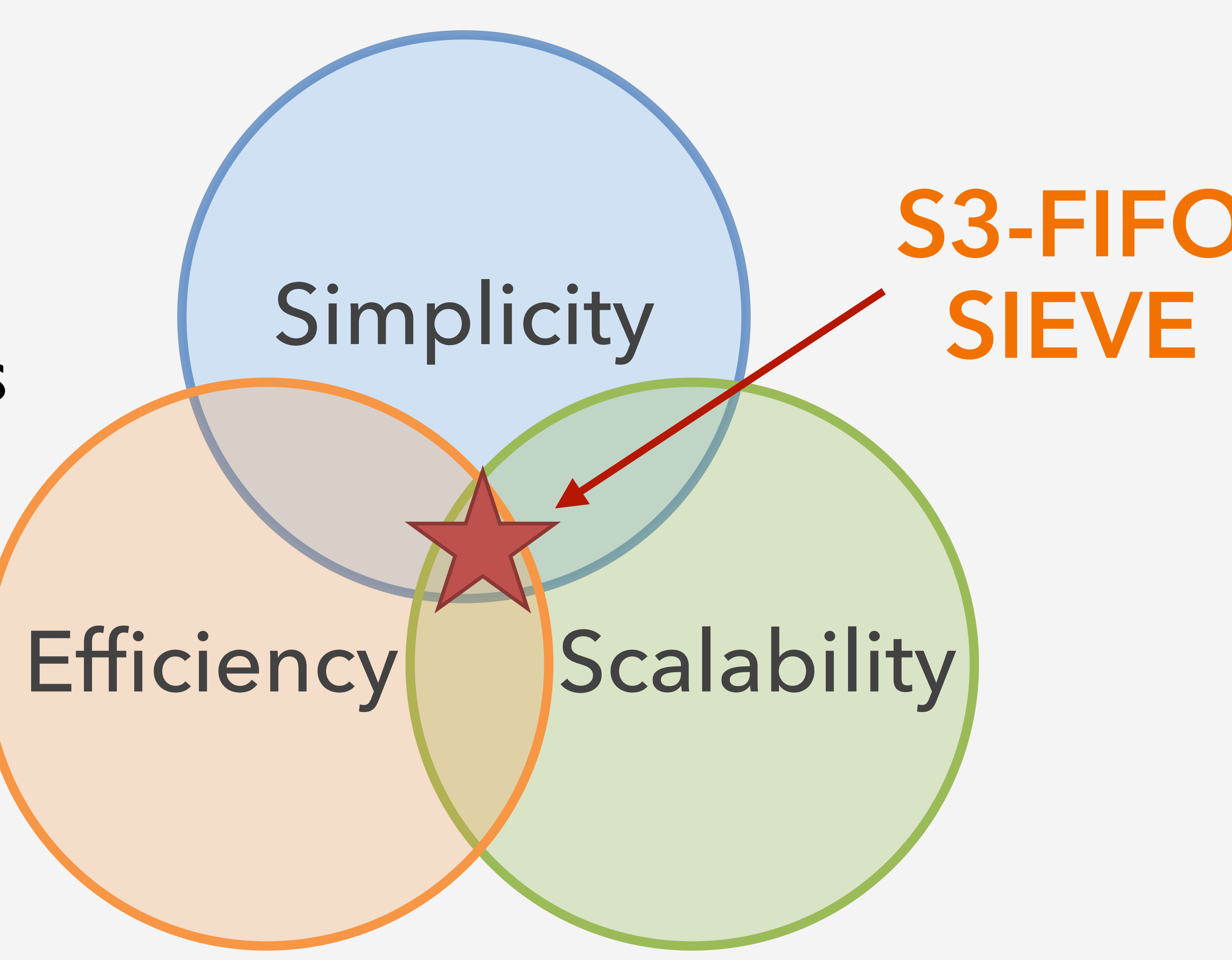
SIEVE has been widely used

- SIEVE is available in **40+** cache libraries with **16** programming languages
- Production systems integrated SIEVE: Android, ImmuDB, SkiftOS, DragonFly…



Takeaway

- Two techniques
  - Lazy promotion
  - Quick demotion
- S3-FIFO: simple, scalable caching with three static FIFO queues
  - small queue for filtering
  - main queue with reinsertion
- SIEVE: the simplest algorithm combining lazy promotion and quick demotion
- **FIFO queues are all you need for caching**



https://s3fifo.com https://sievecache.com

Admission algorithm

Size
Frequency (Bloom Filter)
Probabilistic
ML-based

Admission Algorithms

- Why do we need an admission algorithm?
  - one-hit wonders
  - large objects
  - flash cache (limited endurance)
- Common admission
  - size: reject object sizes larger than threshold
  - bloom filter: admit upon second request
  - probabilistic: simple, achieve required write rate on flash
  - ML-based: Flashield (predict reuse on flash)
- Increasingly important
  - increasing amount of data, many of which do not have many reuses
  - however, it is hard—how do you know whether an item is useful before they are reused?

Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification, NSDI'18
CacheSack: Admission Optimization for Google Datacenter Flash Caches, ATC'23

Prefetching algorithm

Spatial-locality-based
History-based

Prefetching

- Sequential prefetching (readahead)
  - the next *i* pages: c+1, c+2…
    - *n* can be adaptive based on how many of the previously prefetched pages are used
  - stride prefetching: c+8, c+16…
- Correlation-based prefetching (history-based)
  - if every time page c is accessed, page b is also accessed => prefetch b if c is accessed
  - Markov prefetcher, Mithril
- Potential caveats
  - cache pollution
  - waste limited bandwidth

Mithril: mining sporadic associations for cache prefetching, SoCC'17
Reducing file system latency using a predictive approach, USENIX'94

Summary

- Software cache basics
- Different types of caches
- Miss ratio curve
- Eviction algorithm
- Admission algorithm
- Prefetch algorithm

Backup

Key-value cache management system