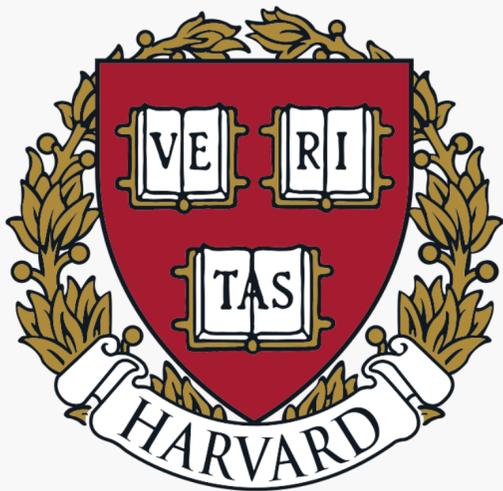
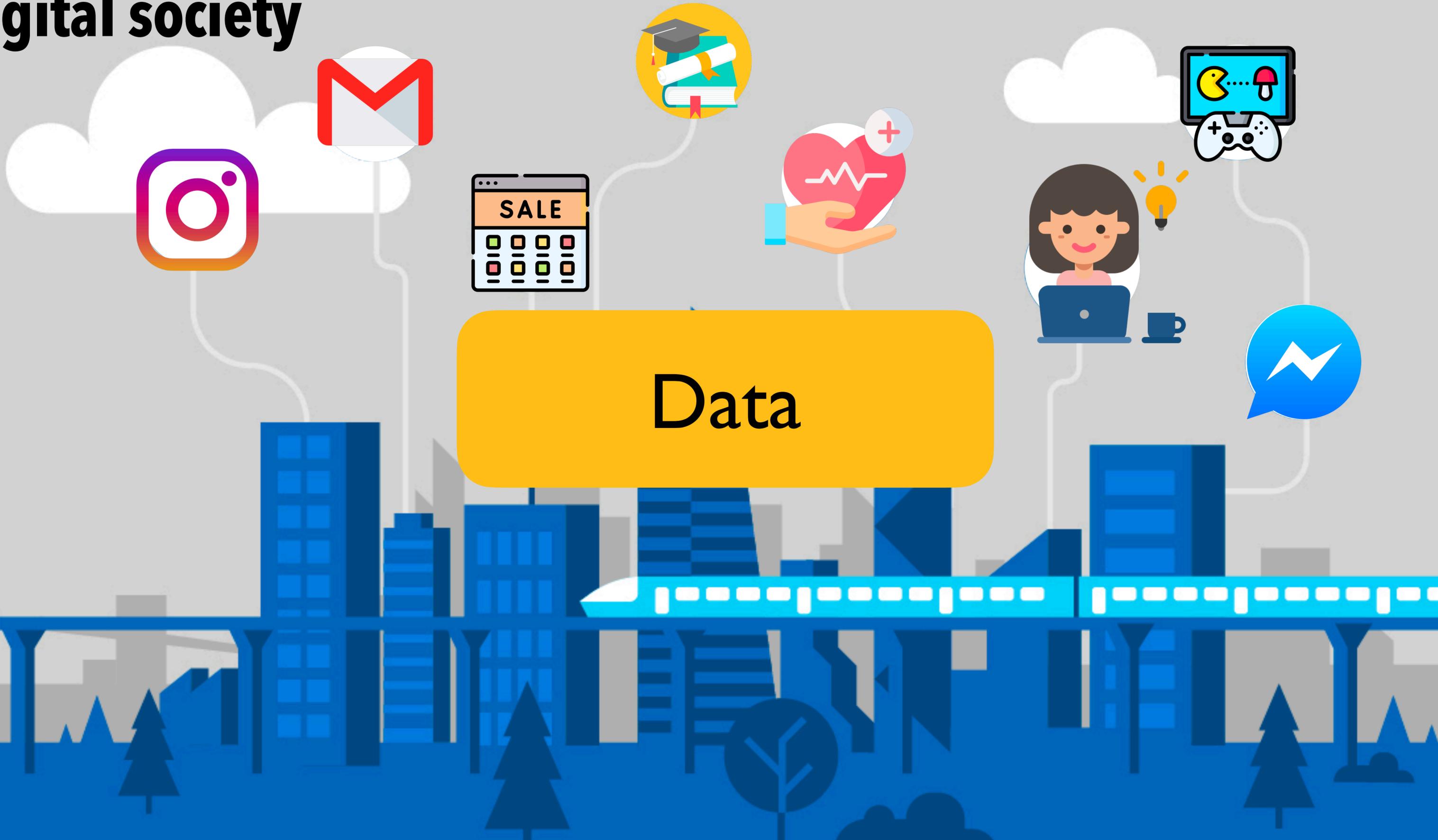


No more LRU, Simple Scalable Caching with only FIFO Queues

Juncheng Yang
Harvard University

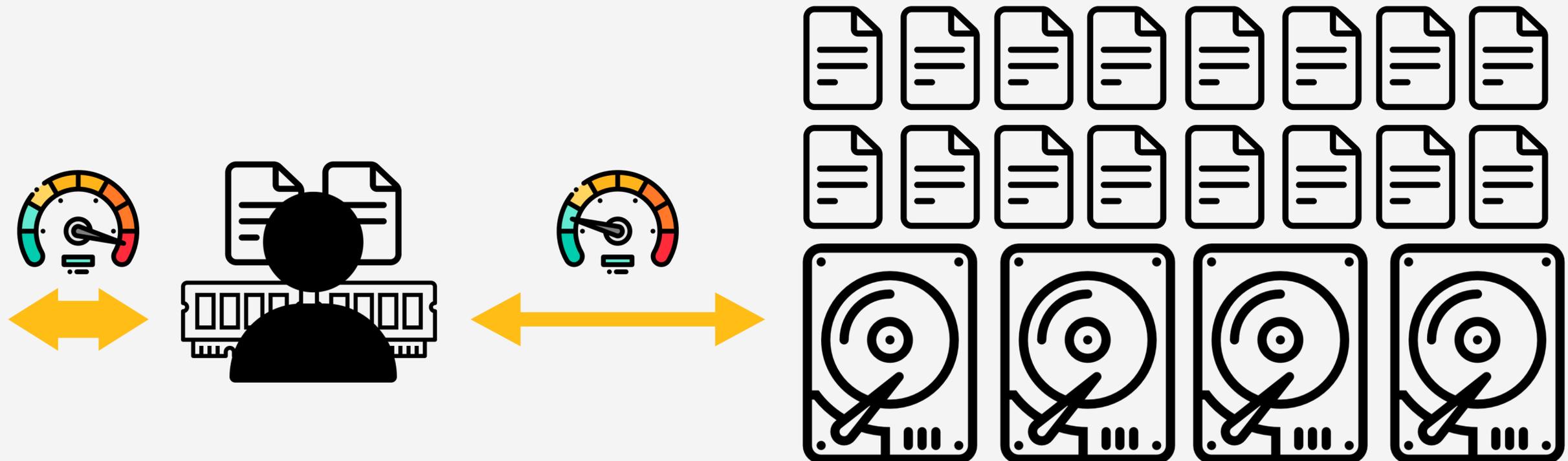


Digital society



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



Software cache is deployed everywhere

Ubiquitous deployment

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

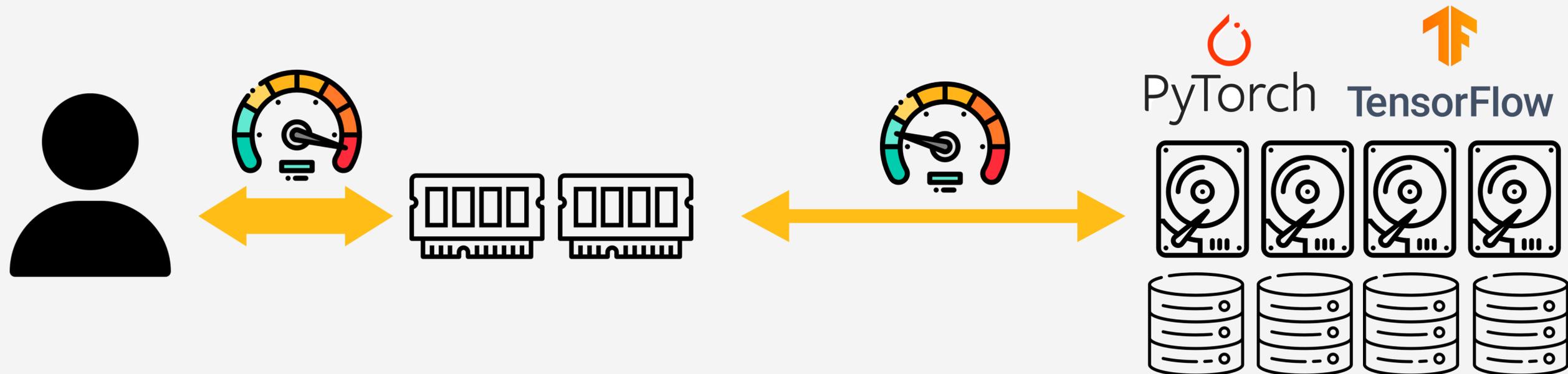


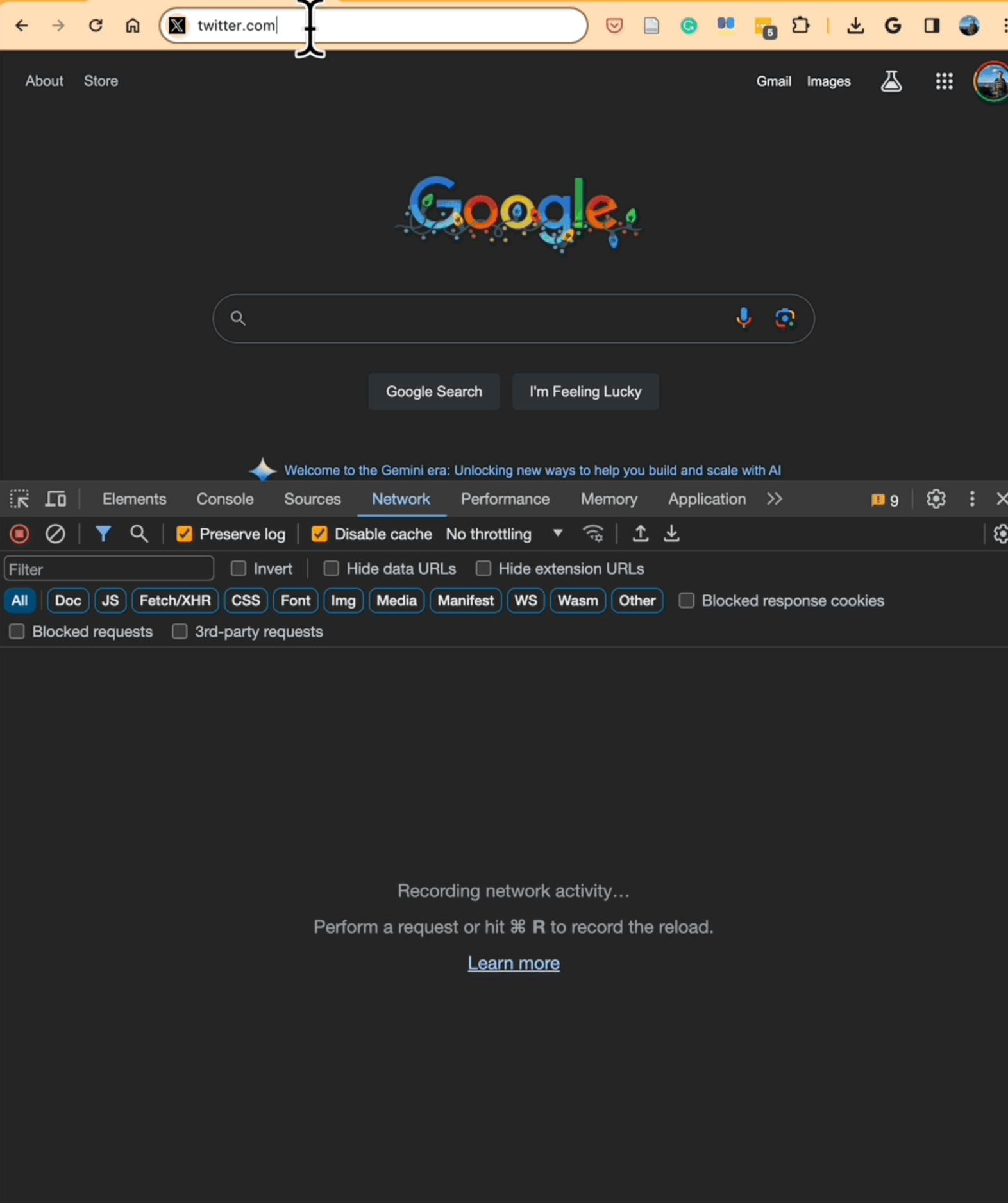
redis

traffic server™



VARNISH
CACHE



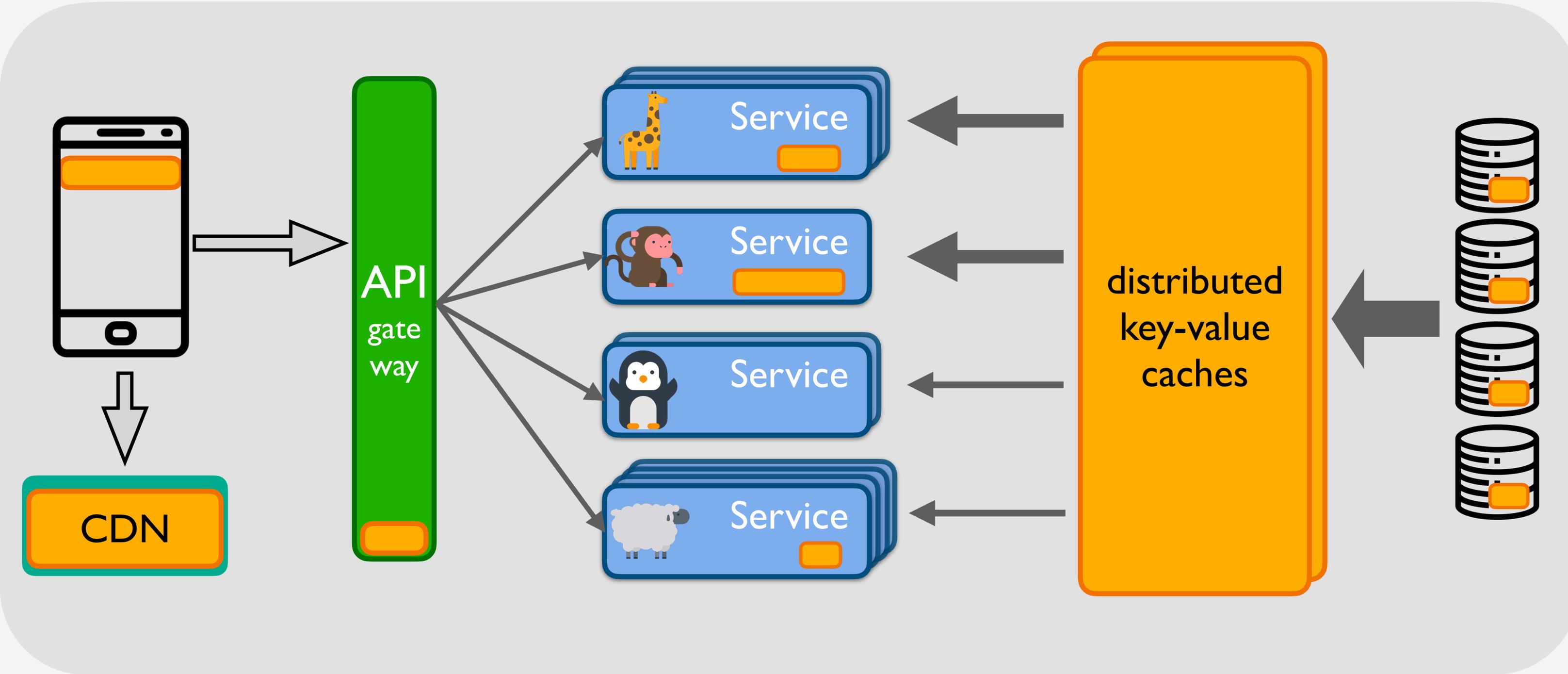


How many requests are served by caches?

10000s

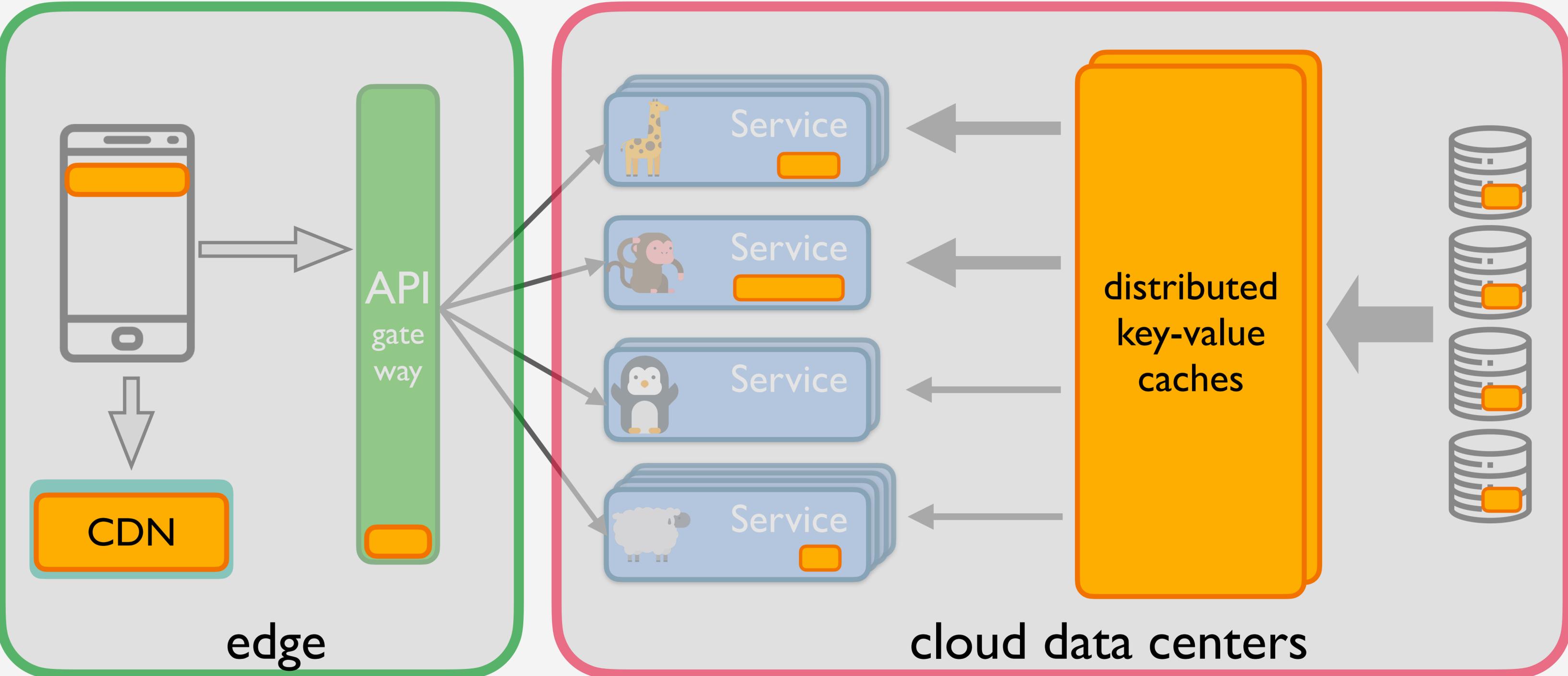
A typical web application

cache



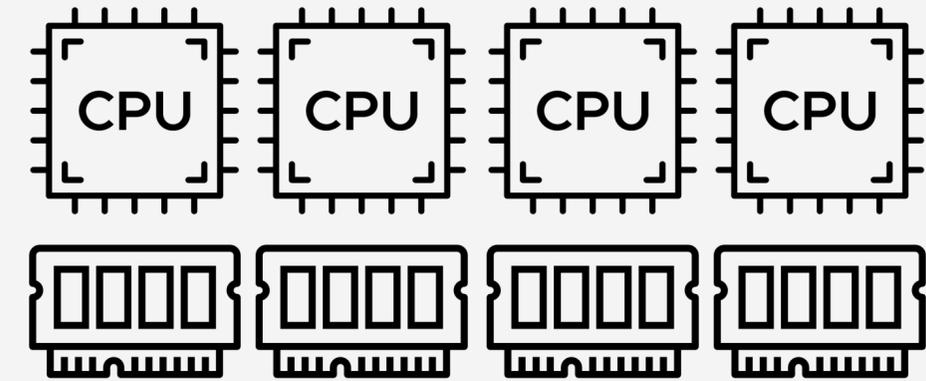
A typical web application

cache



Software cache consumes a huge amount of resources

- Google CliqueMap: **PBs of DRAM**^[1]
- Twitter: 100s clusters, **100s TB of DRAM, 100,000s cores**^[2]
- Meta, Pinterest...



How to make caching more **sustainable**?

- **Less** DRAM: caching more useful objects



efficient caching

- **Fewer** CPU cores: faster and concurrent cache hits



fast and scalable caching

[1] CliqueMap: Productionizing an RMA-Based Distributed Caching System, SIGCOMM'21

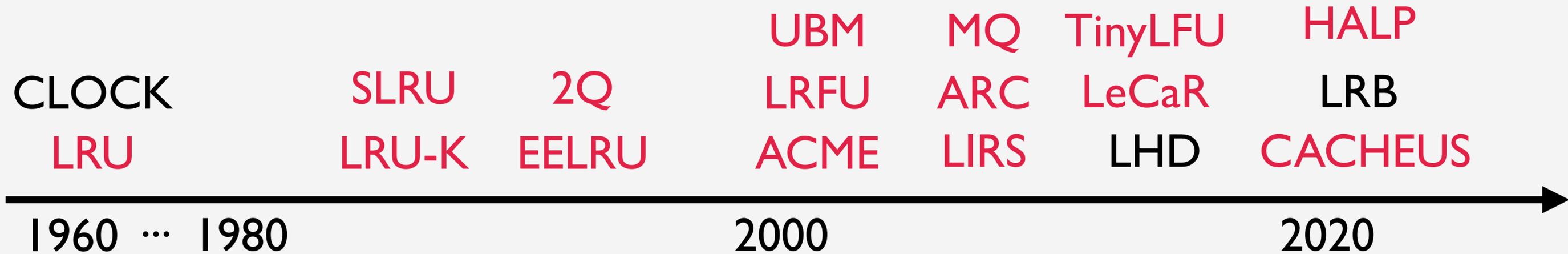
[2] A large scale analysis of hundreds of in-memory cache clusters at Twitter, OSDI'20

The core of a cache

eviction algorithm

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



"Predicting which pages will be accessed in the near future is tricky, and the kernel has evolved a number of mechanisms to improve its chances of guessing right. But **the kernel not only often gets it wrong, it also spend a lot of CPU time to make the incorrect decision.**"

— kernel developer

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

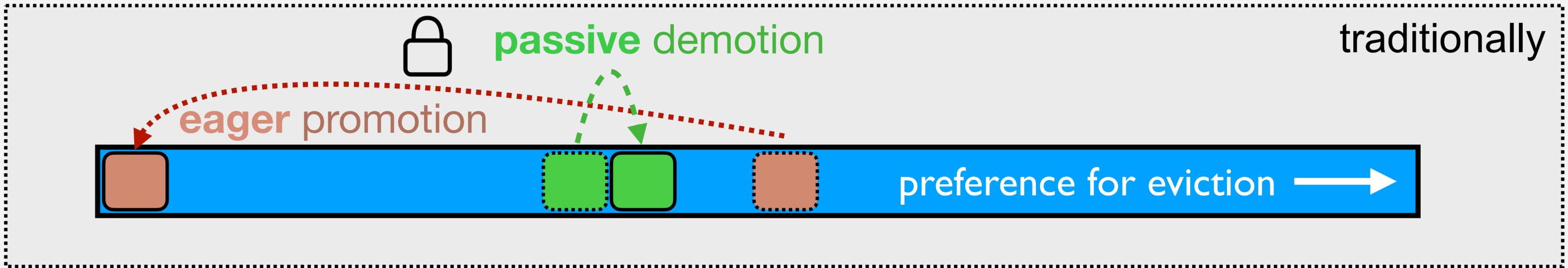
LAZY PROMOTION



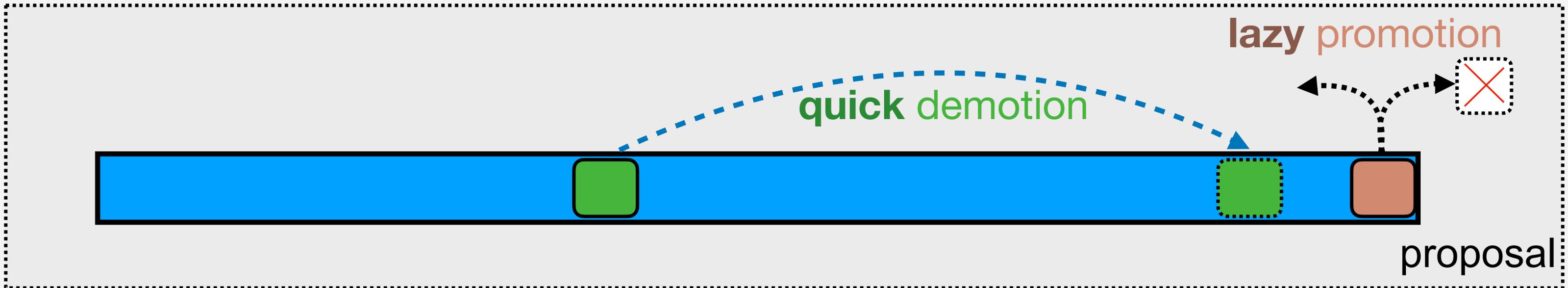
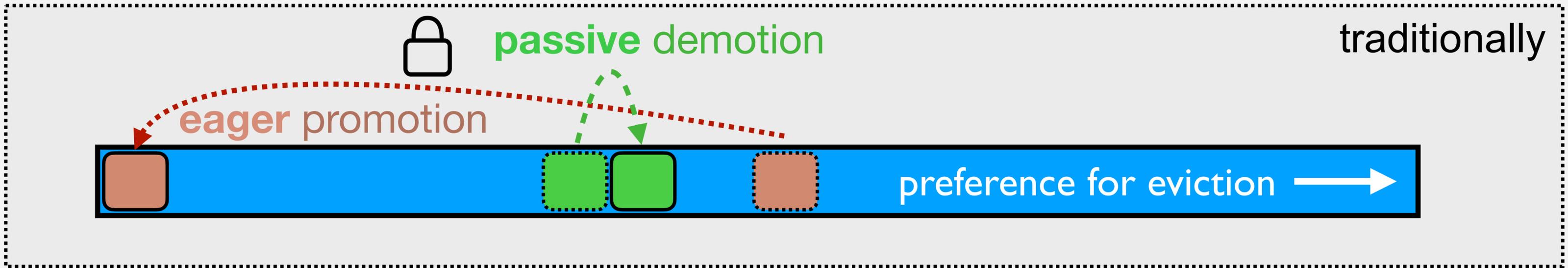
QUICK DEMOTION



Existing eviction algorithms focus on promotion



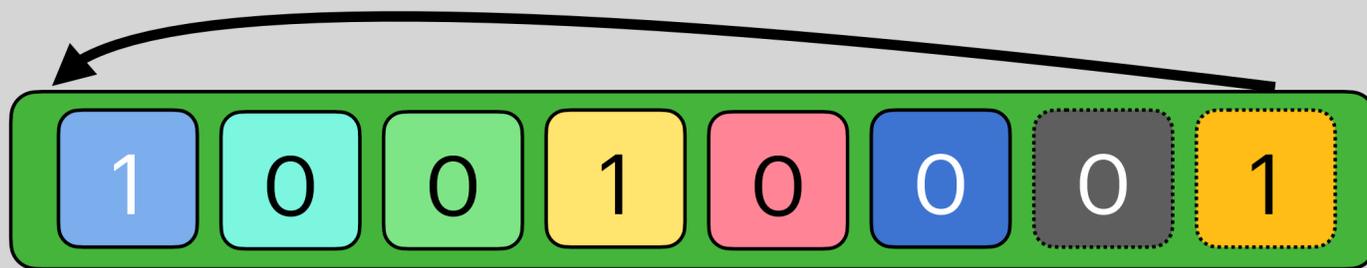
Demotion should be the first-class citizen



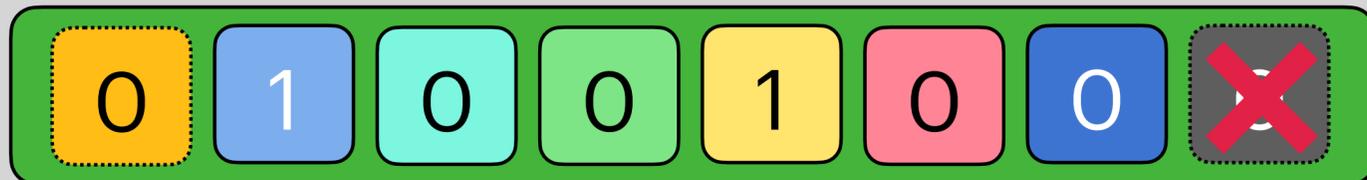
Lazy promotion: only promote during eviction

0: not accessed, 1: has accessed

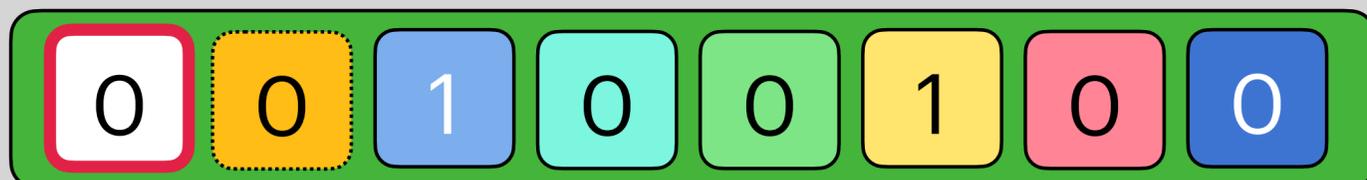
FIFO-Reinsertion



evict the next object



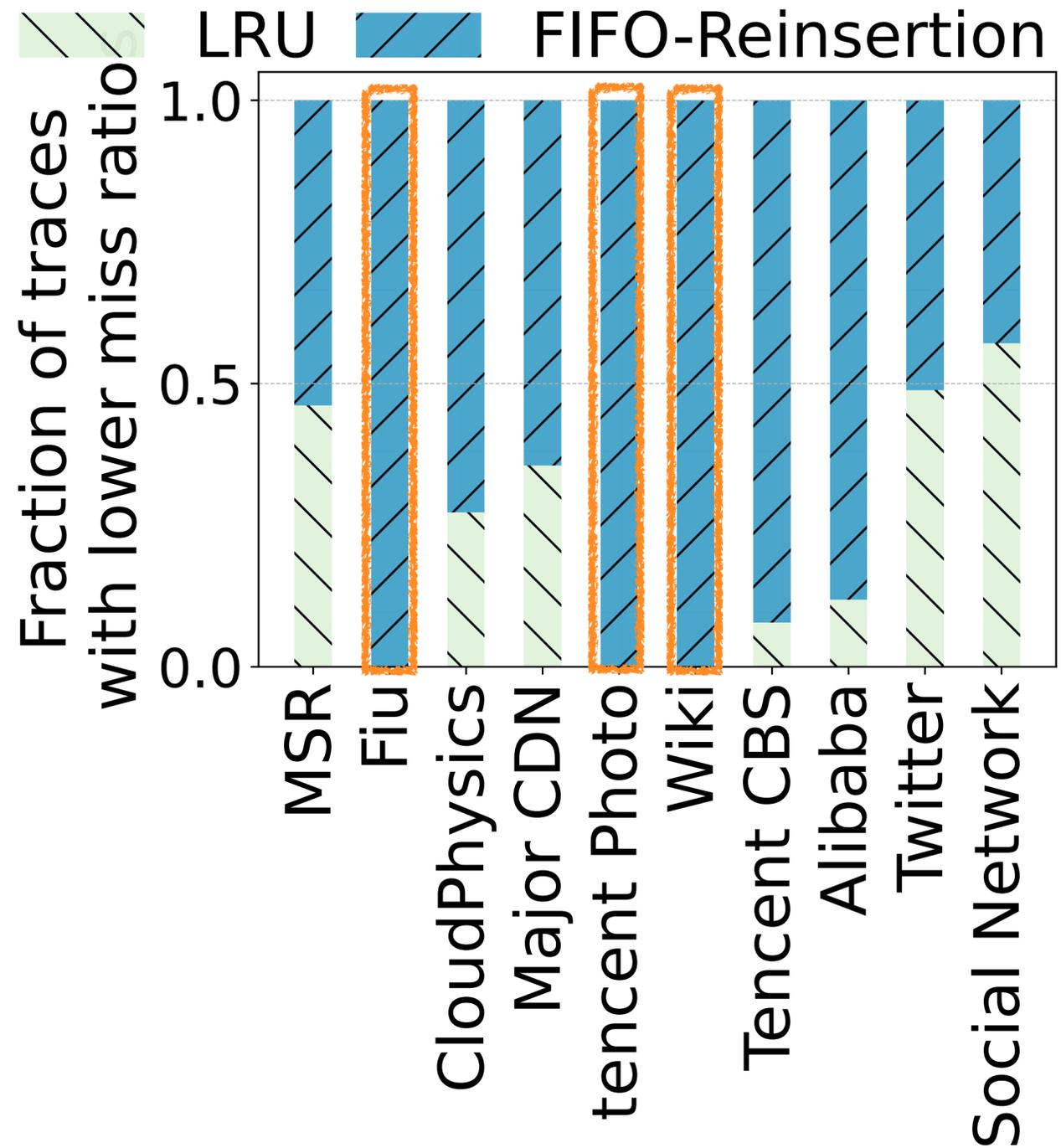
insert a new object



Benefits of lazy promotion

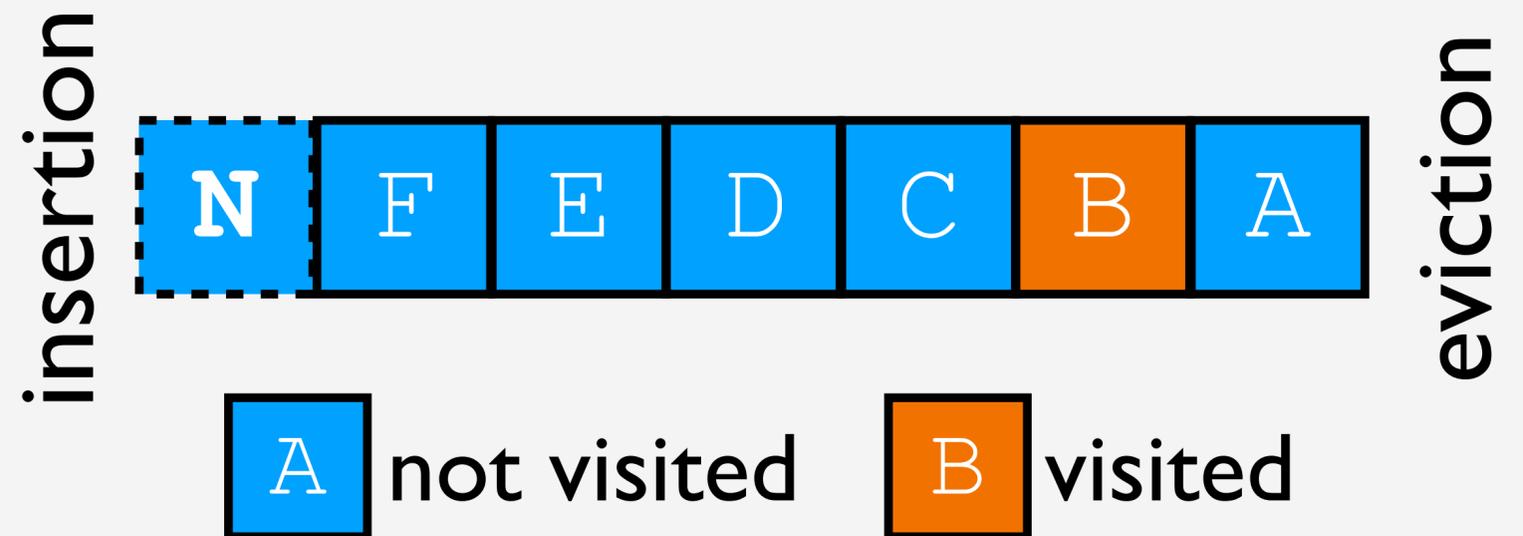
- less computation
- better decision
- more efficient (lower miss ratio)

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



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

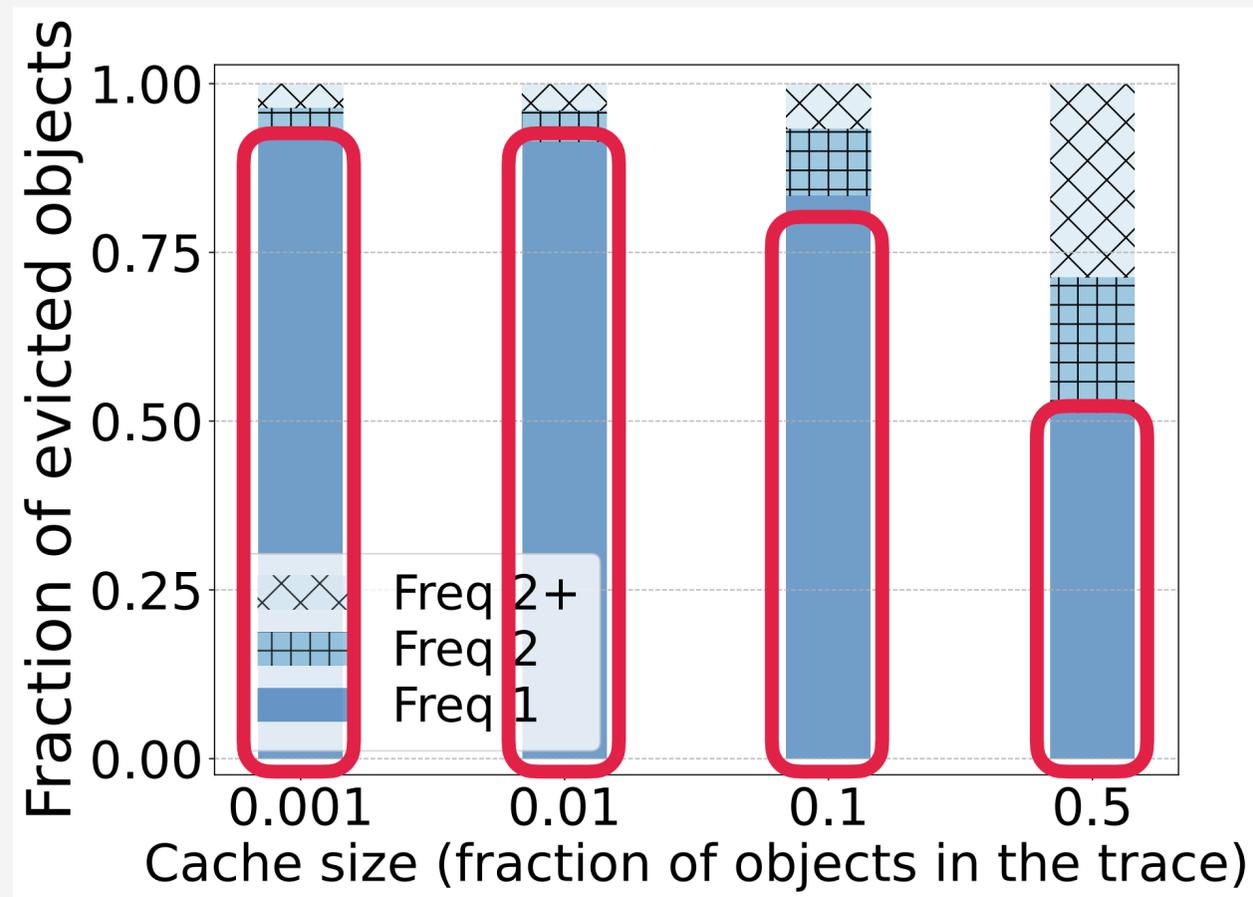
- One-hit wonders: objects appeared once in the sequence
- Cache workloads: *shorter* request sequences have *larger* one-hit-wonder ratios

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

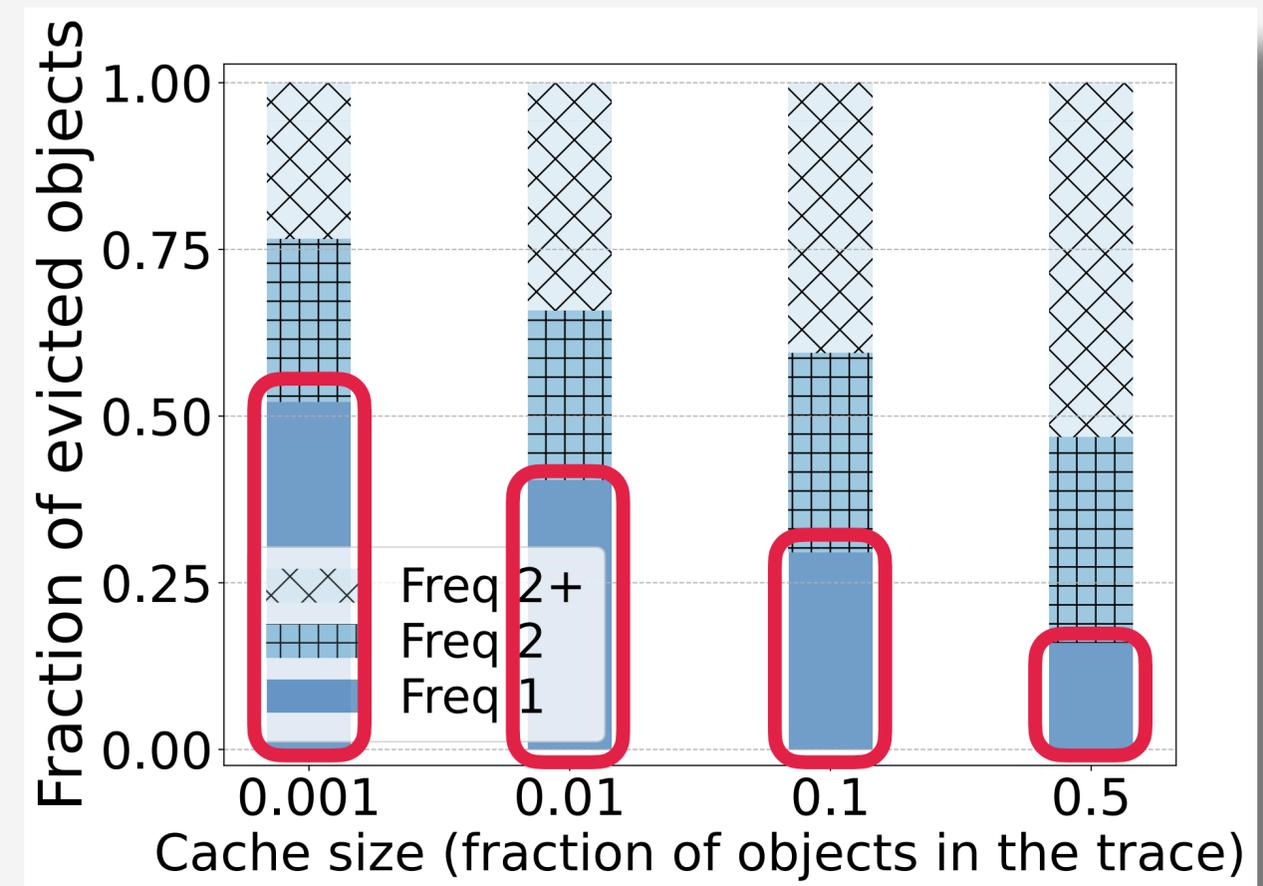
Implication: most objects are not used before evicted

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

Abstract
In a demand-driven system, cache replacement is a substantial problem. We propose a new replacement policy, ARC, that dynamically balances between the recency and frequency components in an *online* and *self-tuning* fashion. The policy ARC uses a learning rule to adaptively and continually revise its assumptions about the workload. The policy ARC is *empirically universal*, that is, it empirically outperforms all other replacement policies on a wide range of workloads.

TinyLFU: A Highly Efficient Cache Admission Policy

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

three LRU queues
+ a new metric

ABSTRACT
Although LRU replacement policy has been commonly used in the buffer cache management, it is well known for its inability to cope with access patterns with weak locality. Previous work, such as LRU-K and 2Q, attempts to enhance LRU capacity by making use of additional history information of previous block references other than only the recency information used in LRU. These algorithms greatly

1.1 The Problems of LRU Replacement Policy
The effectiveness of cache block replacement algorithms is critical to the performance stability of I/O systems. The LRU (Least Recently Used) replacement is widely used to manage buffer cache due to its simplicity, but many anomalous behaviors have been found with some typical workloads, where the hit rates of LRU may only slightly increase with

LHD: Improving Cache Hit Rate by Maximizing Hit Density

Nathan Beckmann, Haoxian Chen, Asaf Cidon

Cliffhanger: Scaling Performance Cliffs in Web Memory Caches

Asaf Cidon¹, Assaf Eisenman¹, Mohammad Alizadeh², and Sachin Katti¹

LRU + partitioning

ABSTRACT
Web caches are a critical component of the Internet. They help to reduce user latency. Small performance improvements in these systems can result in large end-to-end gains. For example, a marginal increase in hit rate of 1% can reduce the application layer latency by over 35%. However, existing web cache resource allocation policies are workload oblivious and first-come-first-serve. By an-
hit rate by just 1% would reduce the read latency by over 35% (from 376μs at 98.2% hit rate to 278μs at 99.2% hit rate). The end-to-end speedup is even greater for user queries, which often wait on hundreds of reads [26]. Web caching systems are generally simple: they have

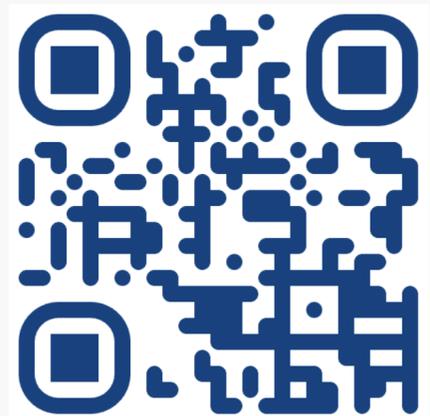
HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network

The **secret sauce** of state-of-the-art algorithms:
evicting new objects very aggressively

S3-FIFO Design

Simple, Scalable caching with three Static FIFO queues

<https://s3fifo.com>



S3-FIFO design

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

1 on *cache hit*
cnt++

QUICK DEMOTION ⚡

2 on *cache miss*
if not in ghost, else

small

3 on *eviction*
if cnt <= 1, else

ghost

LAZY PROMOTION 🐼

main FIFO (90% space)

3 on *eviction*
if cnt == 0
evict
else
reinsert
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

Evaluation setup

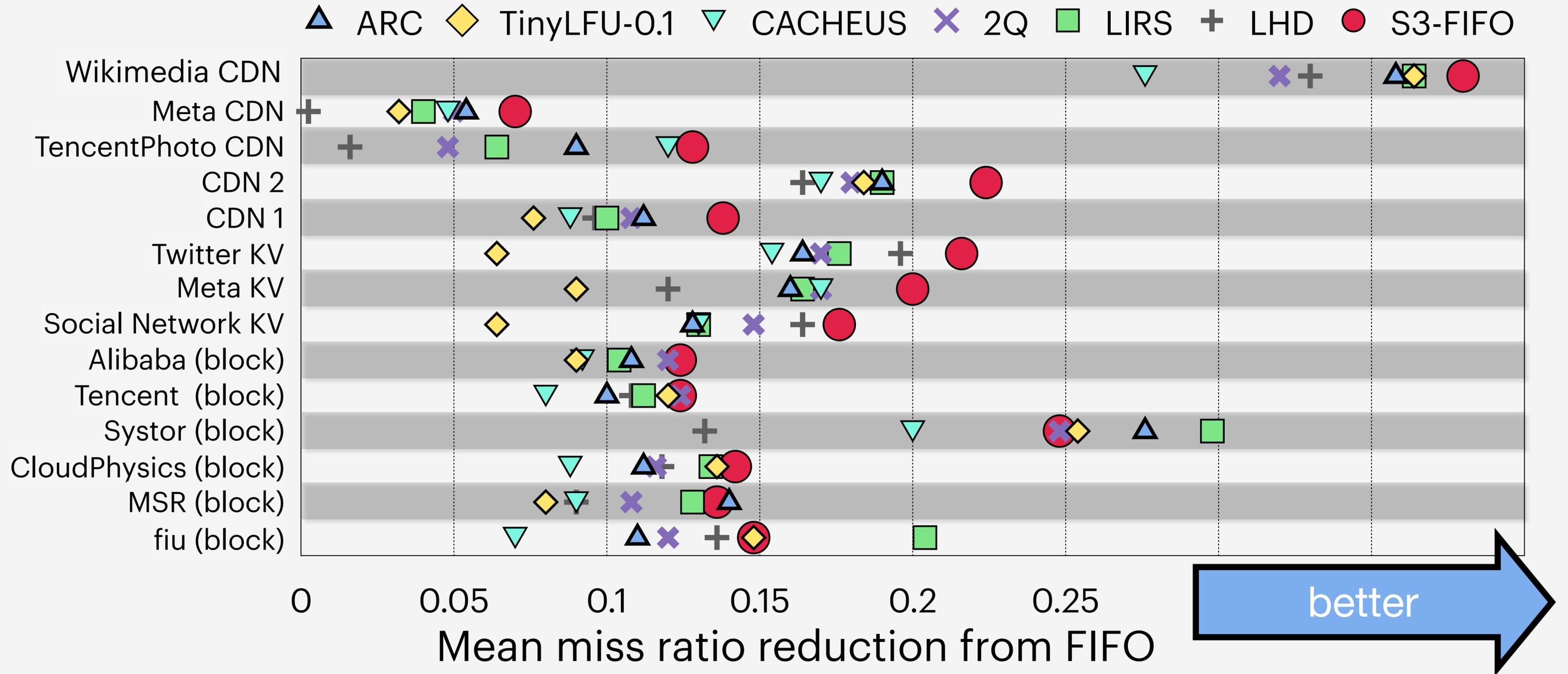
- Data
 - 14 datasets, **6594** traces from Twitter, Meta, Microsoft, Wikimedia, Tencent, Alibaba, major CDNs...
 - **848 billion** requests, **60 billion** objects
 - collected between 2007 and 2023
 - block, key-value, object caches
- Platform
 - libCacheSim, cachelib
 - PDL cluster and CloudLab with 1 million core-hours
- Metric
 - miss ratio reduction from FIFO
 - throughput in Mops/sec

Data and software are open-sourced



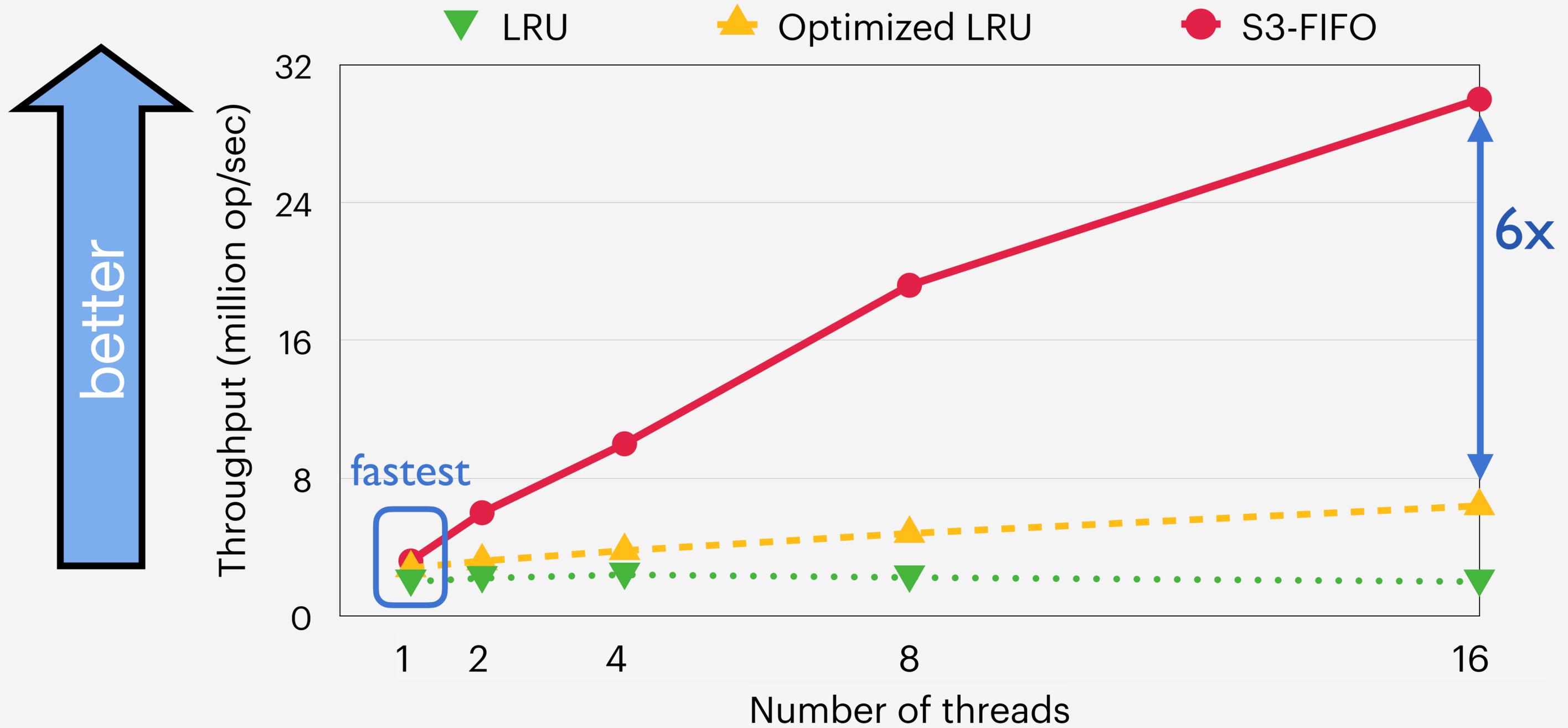
CloudLab

S3-FIFO is efficient across datasets

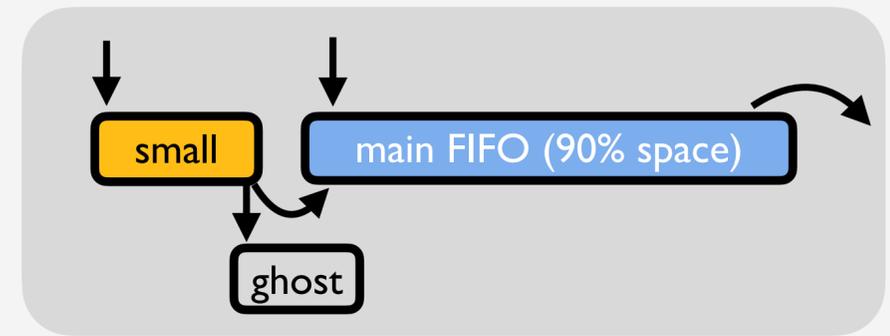


Evaluated on 6594 traces with 848 billion requests from 12 sources, collected between 2007 and 2023
This evaluation is milliontimes larger than previous works

Throughput scales with number of threads



Recap



<https://s3fifo.com>

- S3-FIFO: simple, scalable caching with three static FIFO queues
 - prevalence of one-hit wonders
 - small FIFO queue: quickly evict one-hit wonders, reinsertion: keep popular objects
- The first work showing that **FIFO queue is sufficient to design efficient algorithms**
- S3-FIFO recognition and impact
 - Covered by many blogs, newsletters, meetups in English, Chinese, Korean, Japanese...
 - **Deployed** at Google, VMware, Redpanda...
 - More than ten **open-source libraries** implemented S3-FIFO in Rust, C, C++, JavaScript, Python...



SIEVE

An eviction algorithm simpler than LRU



The secret to designing efficient eviction algorithms



LAZY PROMOTION

Retain popular objects with minimal effort



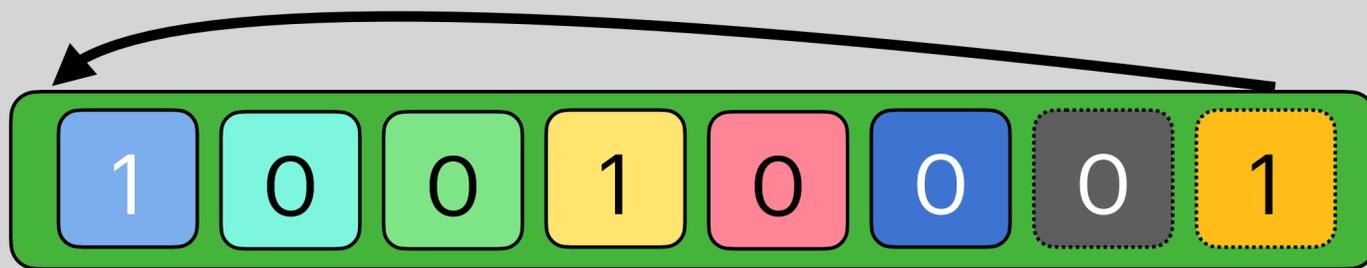
QUICK DEMOTION

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

SIEVE: combining lazy promotion and quick demotion

A small change turn FIFO-Reinsertion to SIEVE

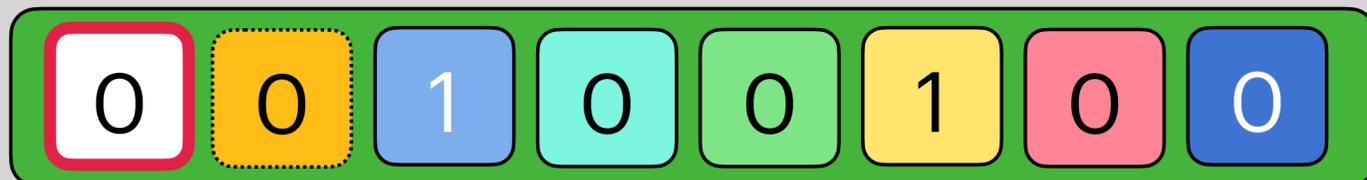
FIFO-Reinsertion



evict the next object



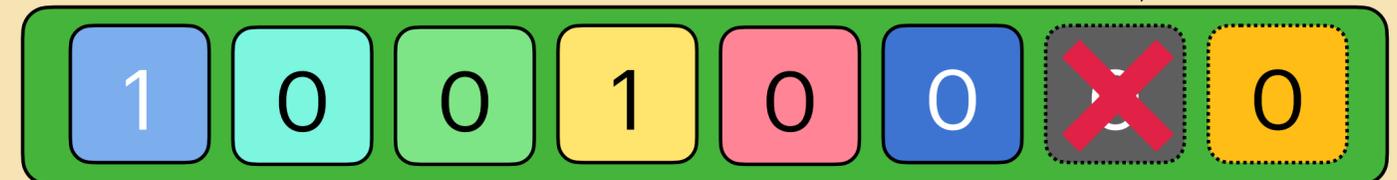
insert a new object



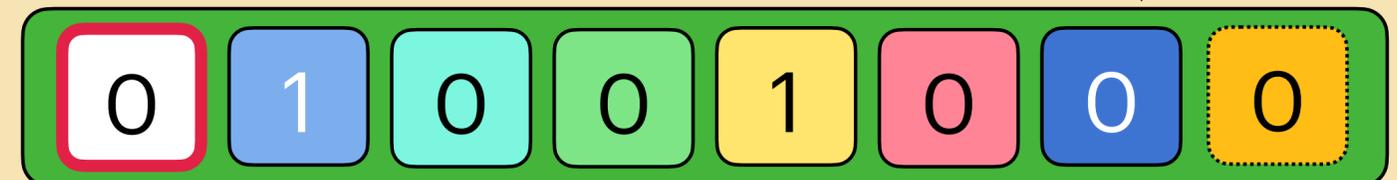
SIEVE



evict the next object



insert a new object



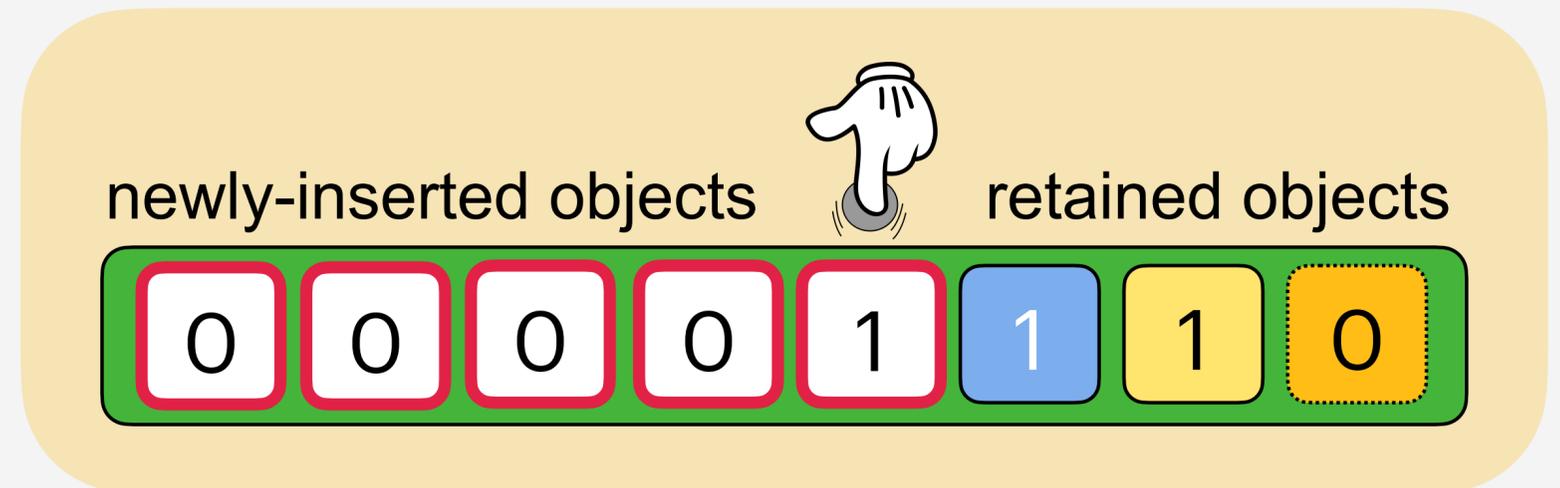
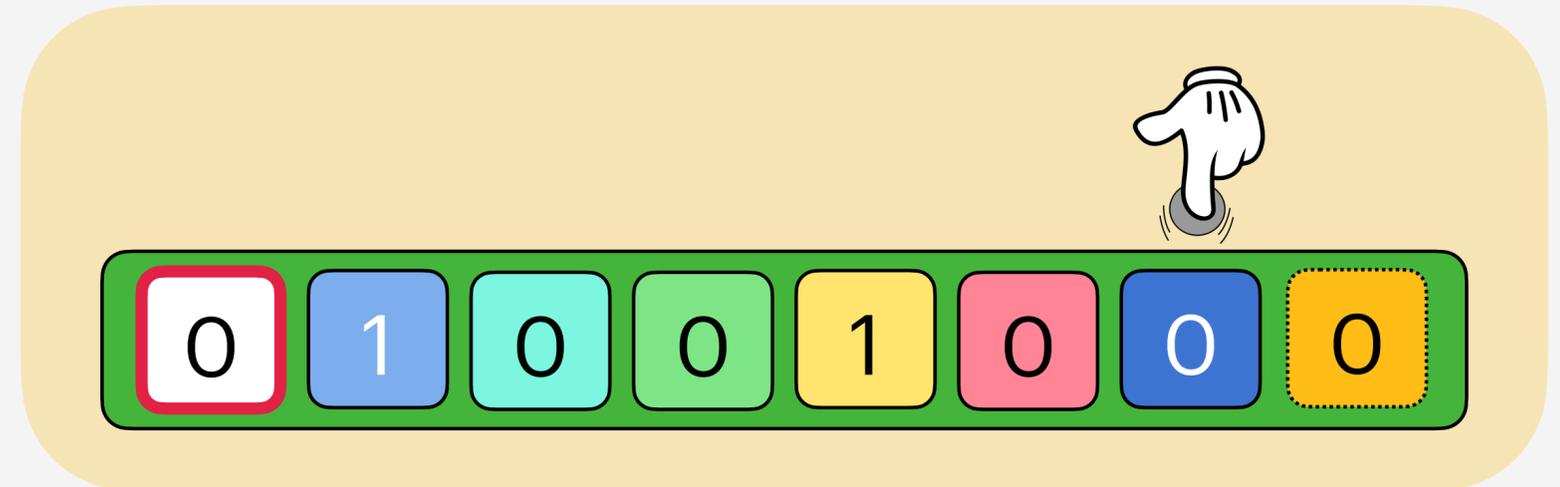
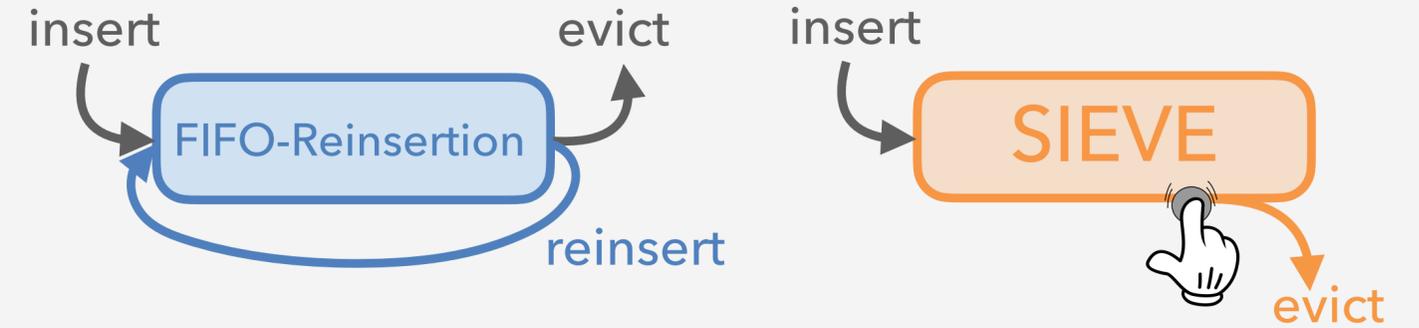
0: not accessed, 1: has accessed

SIEVE features

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

Why does SIEVE work?

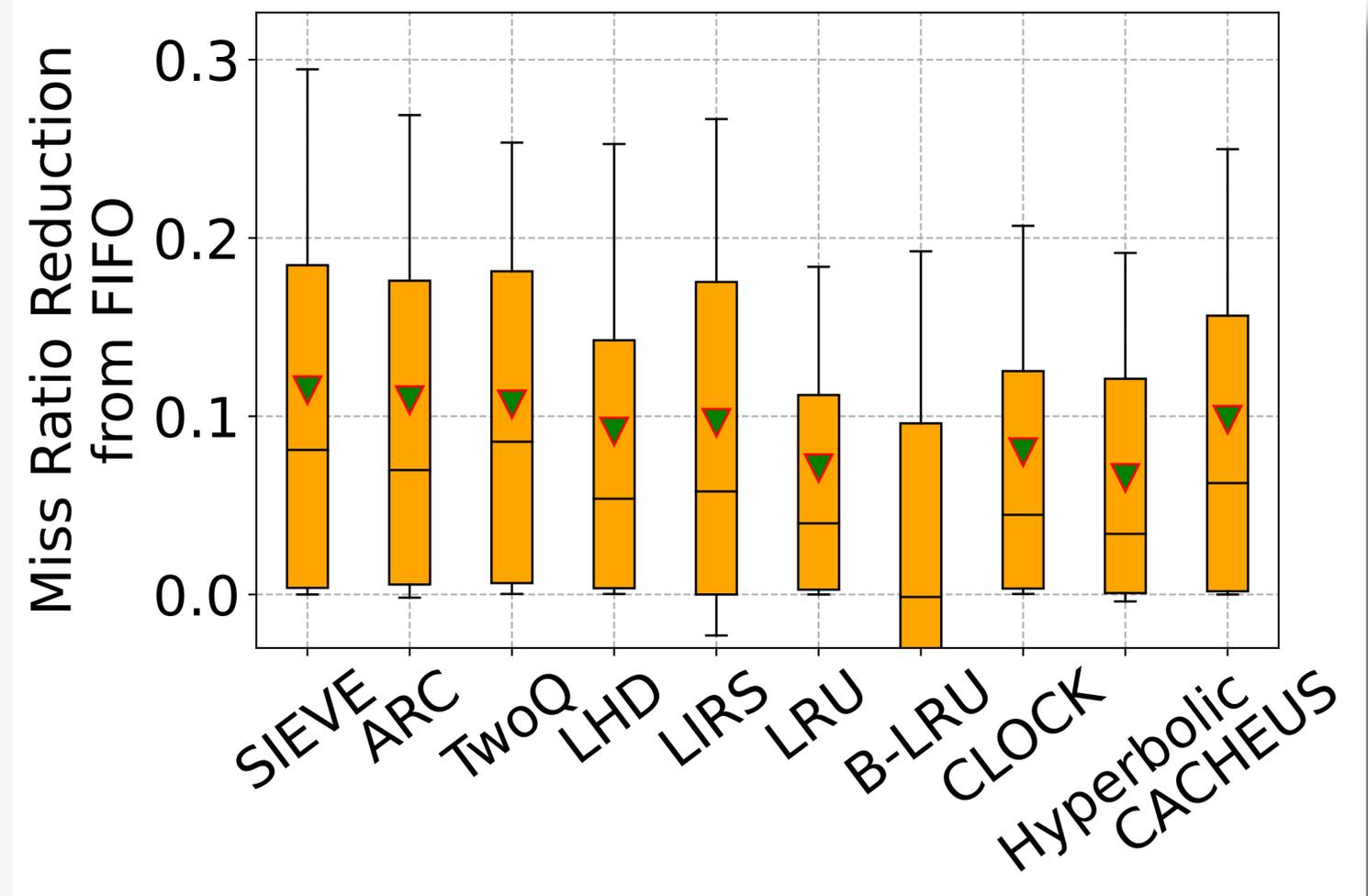
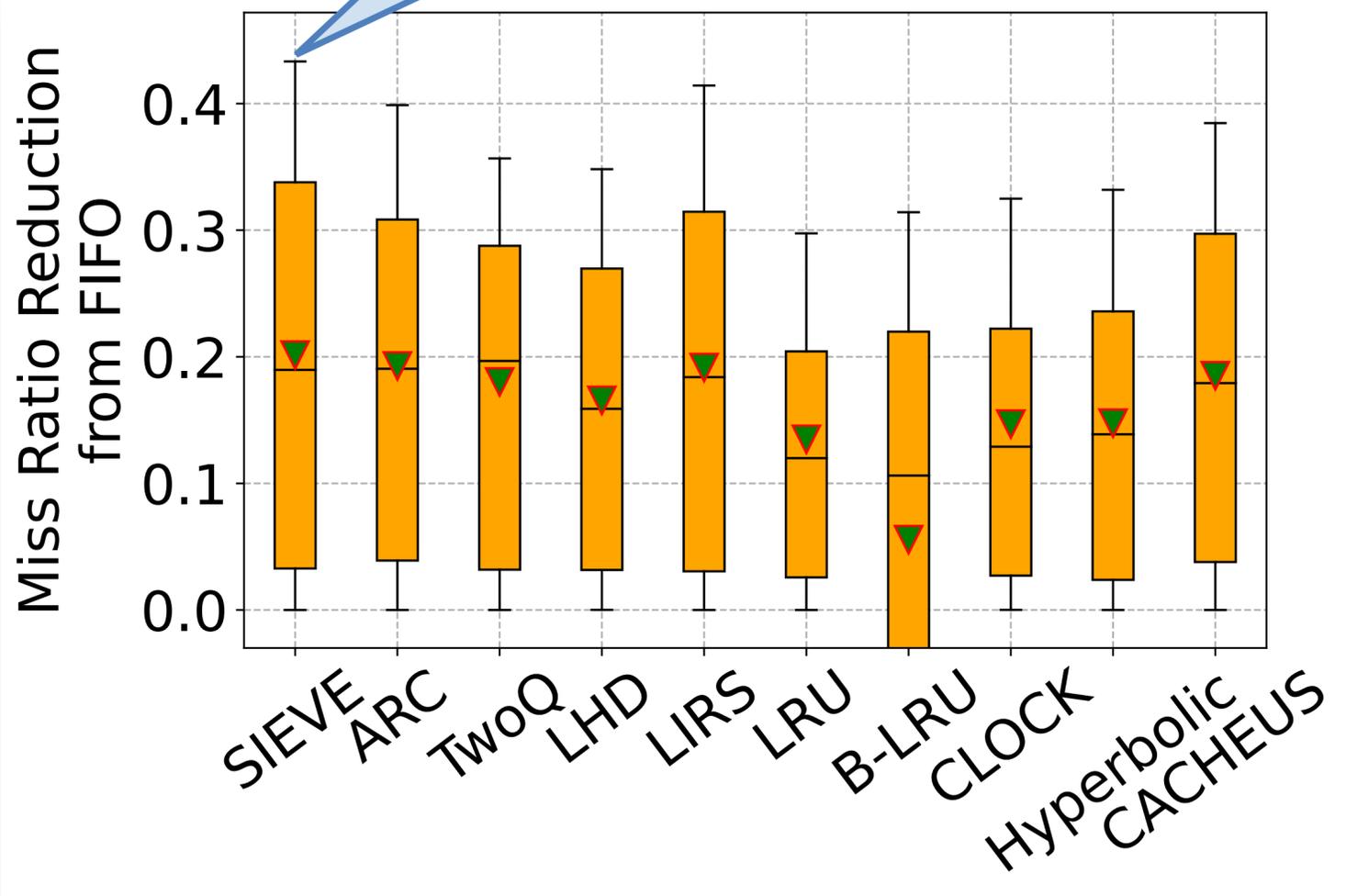
- Retain popular objects effectively
- Evict new objects quickly
- Separate new and old objects



SIEVE achieves

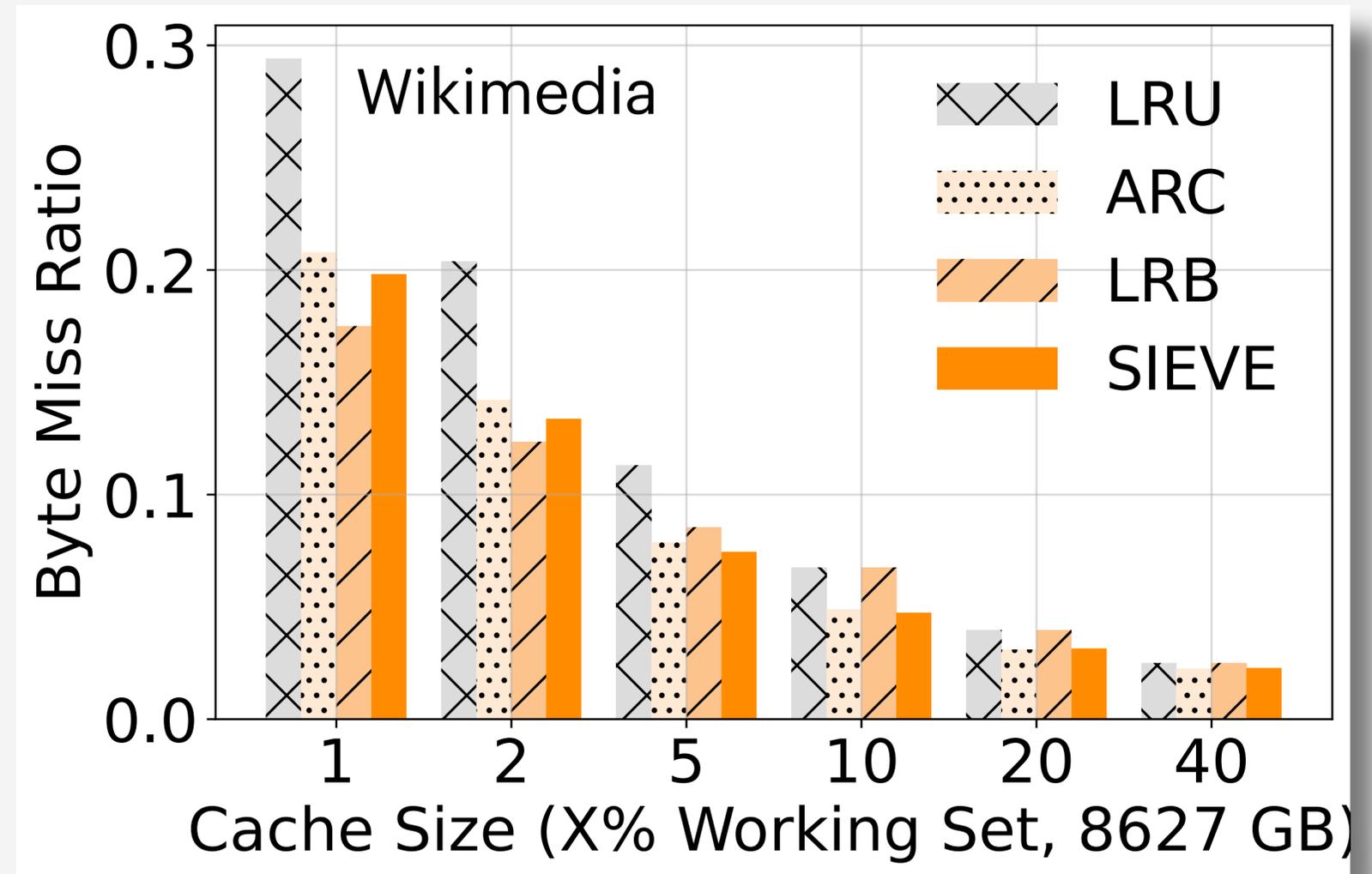
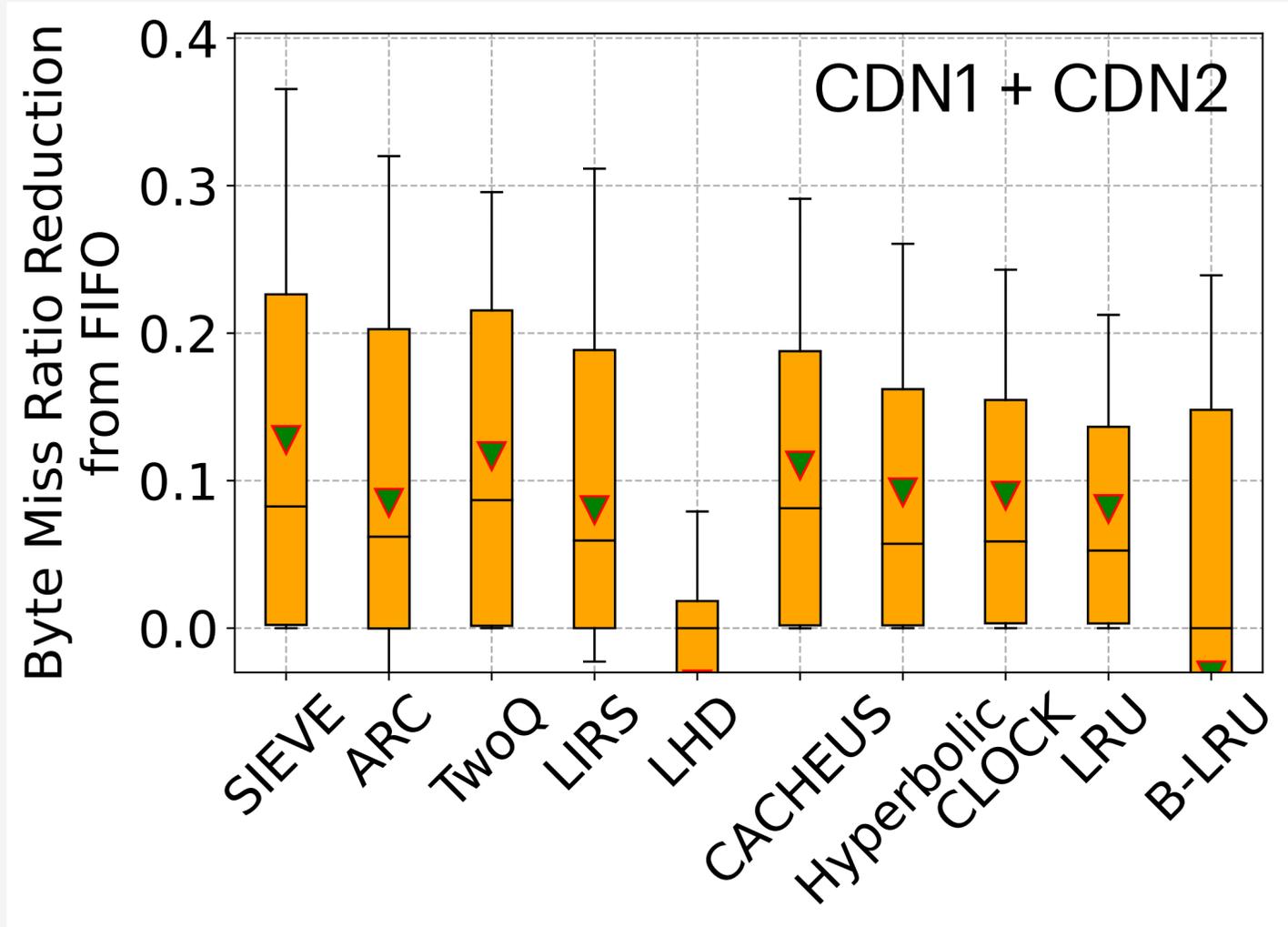
SIEVE reduces FIFO's miss ratio by more than 42% on 10% of the traces (top whisker) with a mean of 21%

efficiency



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

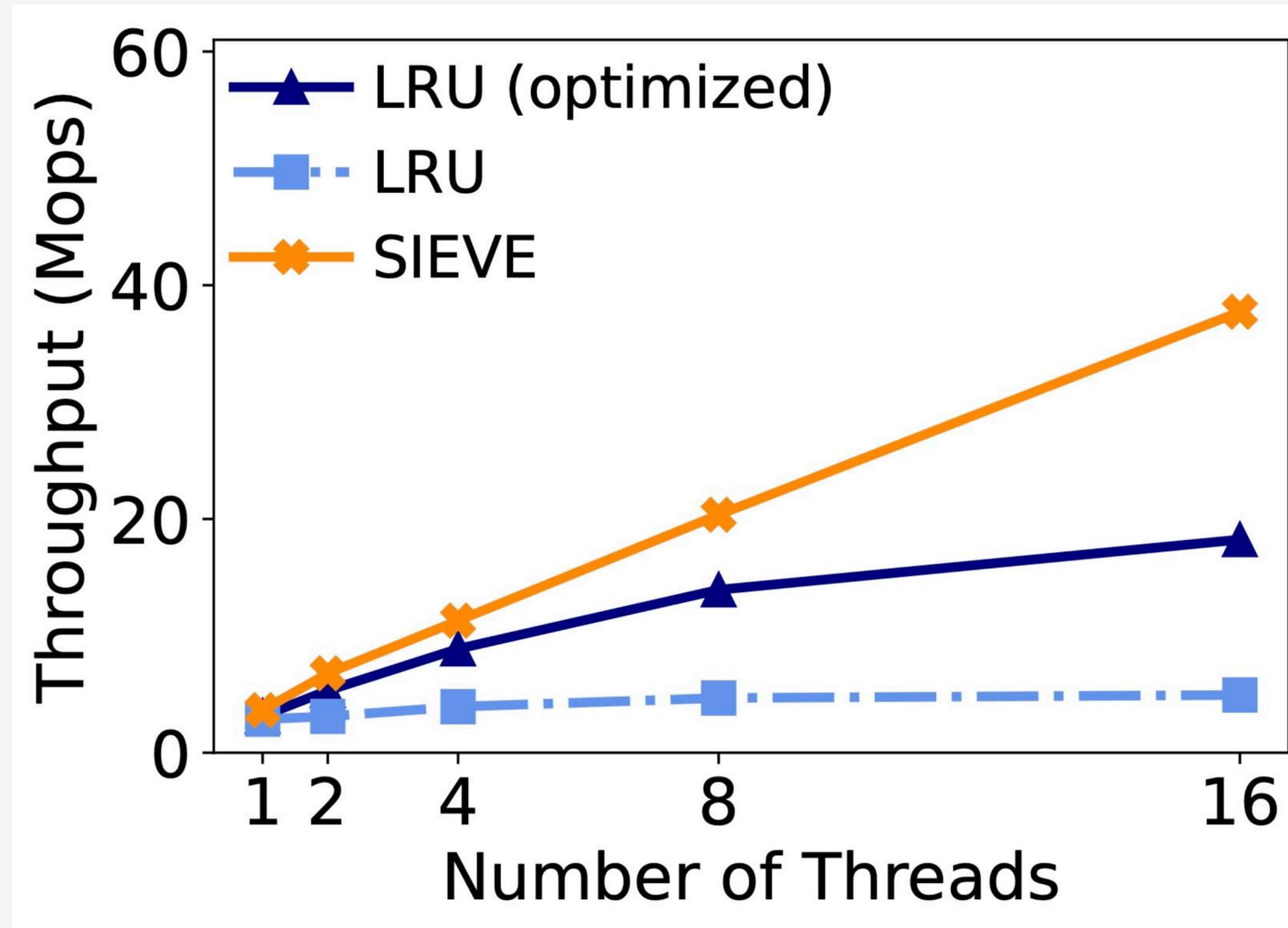
SIEVE achieves the lowest byte miss ratio



Better than all state-of-the-art algorithms

Small cache: better than ARC, close to LRB
Large cache: better than LRB

SIEVE throughput scales with the number of threads



compared to optimized LRU:
16% **faster** with a single thread, 2x **faster** with 16 threads

SIEVE is simple to implement

Cache library	Language	Eviction algorithm	Lines of change
groupcache	Golang	LRU	21
mnemonist	Javascript	LRU	12
lru-rs	Rust	LRU	16
lru-dict	Python + C	LRU	21

Adoption

Large systems:  Pelikan  Nyrkiö  SkiftOS  DragonFly

 DNSCrypt-proxy  encrypted-dns-resolver

Cache libraries:  golang-fifo  js-sieve  rust-sieve-cache  go-sieve

 sieve_cache (Ruby)  zig-sieve (Zig)  sieve (Swift)

 sieve (JavaScript)  sieve (Elixir)  sieve (Nim)

 sieve-cache (Java)  sieve (Python)  sieve-cache-in-rust

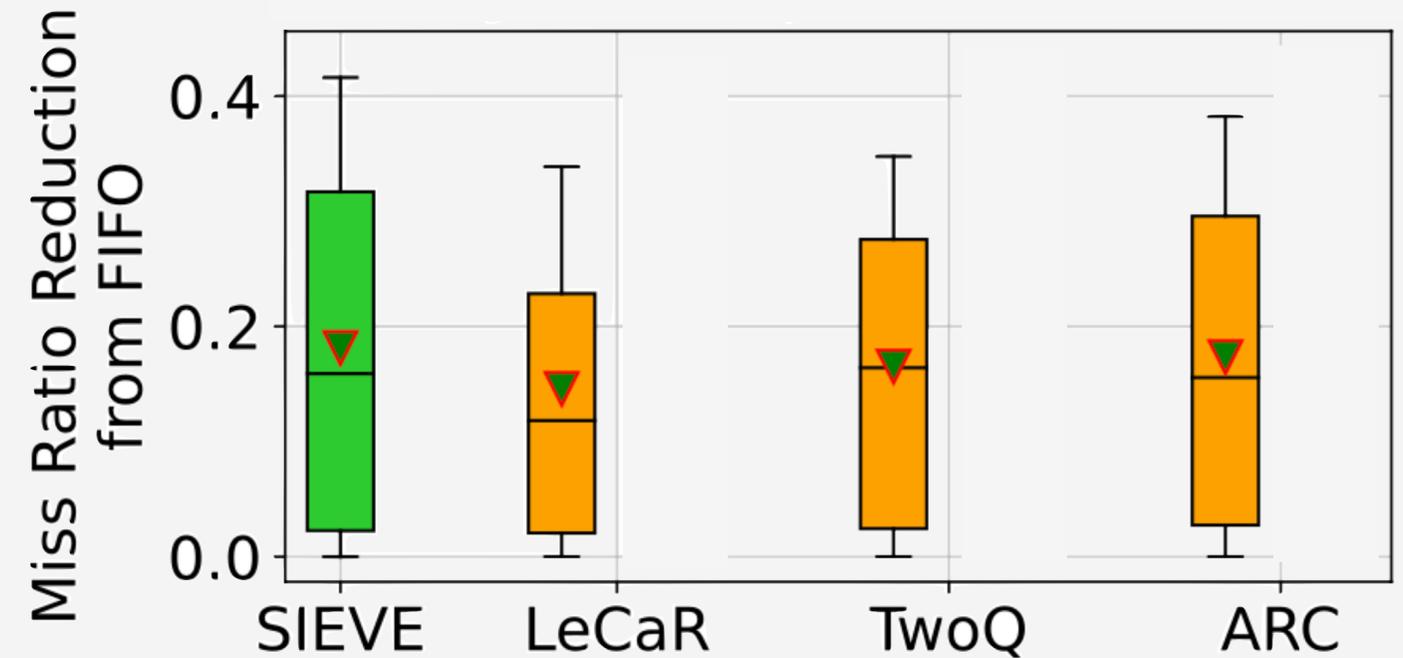
 sieve-cache (JavaScript)  gosieve,  sieve (typescript)

SIEVE can be used as a cache primitive

LeCaR: LRU + LFU + ML

TwoQ: LRU + FIFO

ARC: LRU + LRU + 2 ghost queues



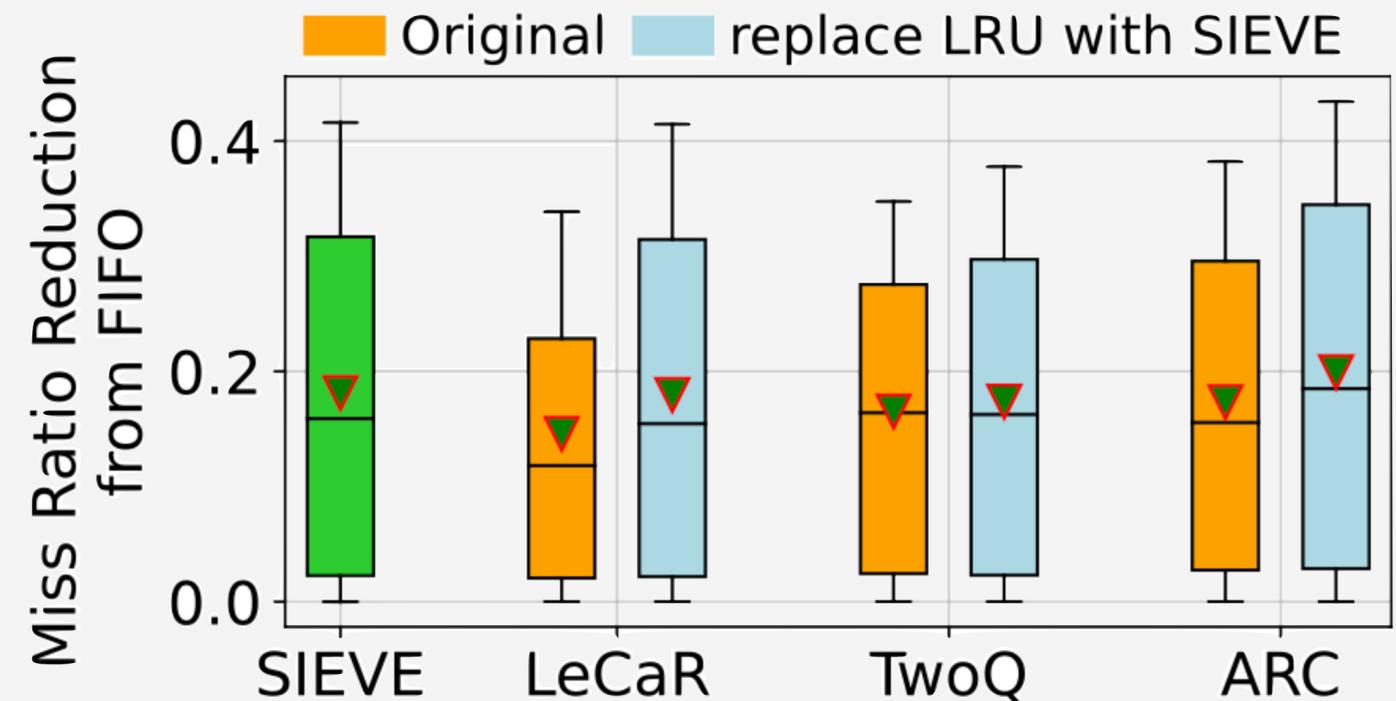
SIEVE can be used as a cache primitive

LeCaR: LRU + LFU + ML

TwoQ: LRU + FIFO

ARC: LRU + LRU + 2 ghost queues

Replace LRU with SIEVE



SIEVE has been widely used

- SIEVE is available in **20+** cache libraries with **12** programming languages
- Production systems integrated SIEVE: Pelican, SkiftOS, DragonFly...



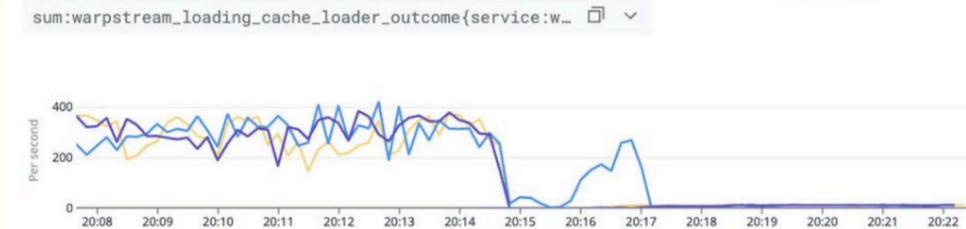
Richard Artoul 
@richardartoul

Turns out Ristretto cache is **async**... I switched WarpStream's footer cache from Ristretto to golang-fifo (Sieve algo) and got a 33x reduction in cache misses and 16% CPU savings...

Cache Loads

Richard Artoul | Updated 4 minutes ago

sum:warpstream_loading_cache_loader_outcome{service:w...    

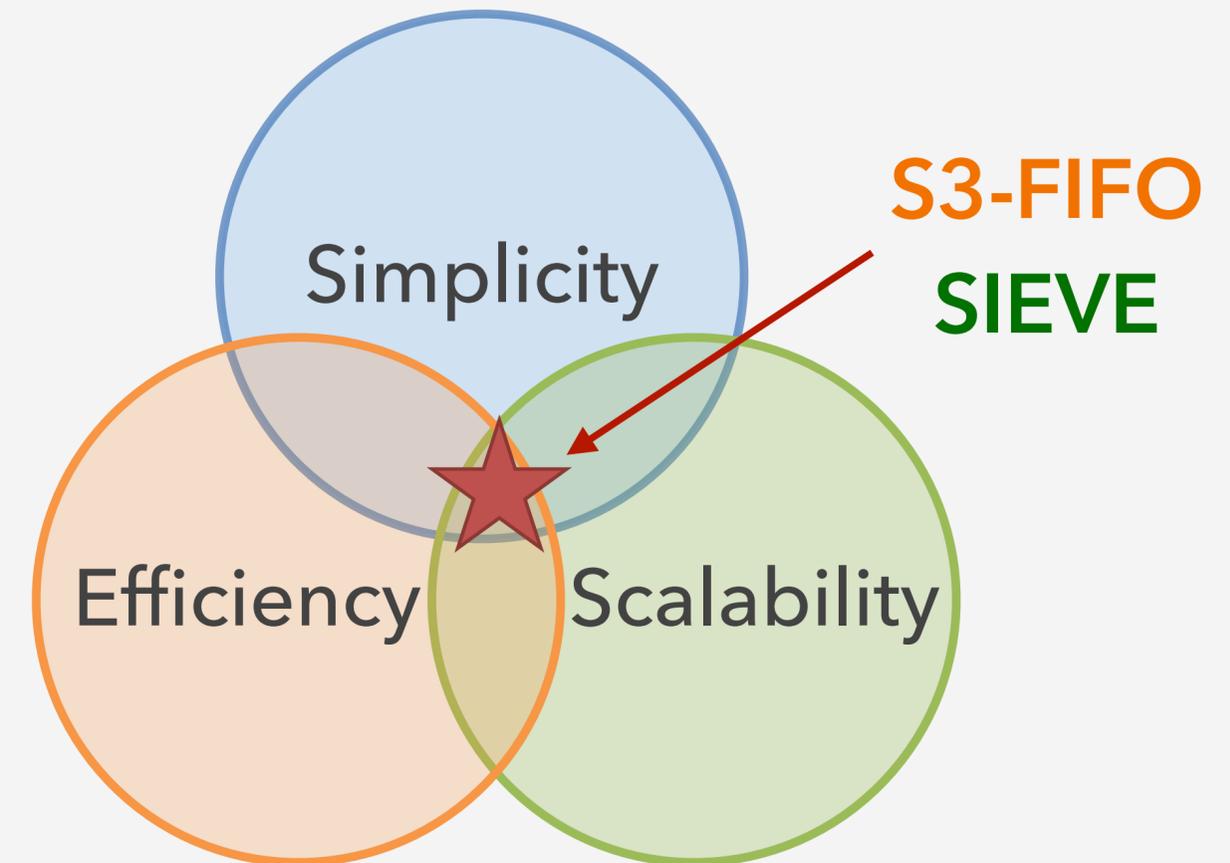


Tags in sum:warpstream_loading_cache_loader_outcome{service:warp-agent,env...	Avg	Min	Max	Sum	Value
cache_name:acls,host:i-0cc491918ccbaf6e7	6e-3 /s	0 /s	0.10 /s	0.20 /s	—
cache_name:acls,host:i-0ddd25eb52b97b6f	0.01 /s	0 /s	0.30 /s	0.50 /s	—
cache_name:acls,host:i-0ee0b6128679455ed	0.013 /s	0 /s	0.40 /s	0.40 /s	—
cache_name:cluster,host:i-0ddd25eb52b97b6f	4e-3 /s	0 /s	0.10 /s	0.10 /s	—

9:35 PM · Jan 20, 2024 · 17.3K Views

Summary

- 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



<https://s3fifo.com>

<https://sieve-cache.com>

How we can build **better** storage systems **together**



Juncheng Yang
<https://junchengyang.com>