A large scale analysis of hundreds of in-memory cache clusters at Twitter

Juncheng Yang (Carnegie Mellon University)
Yao Yue (Twitter)
Rashmi Vinayak (Carnegie Mellon University)

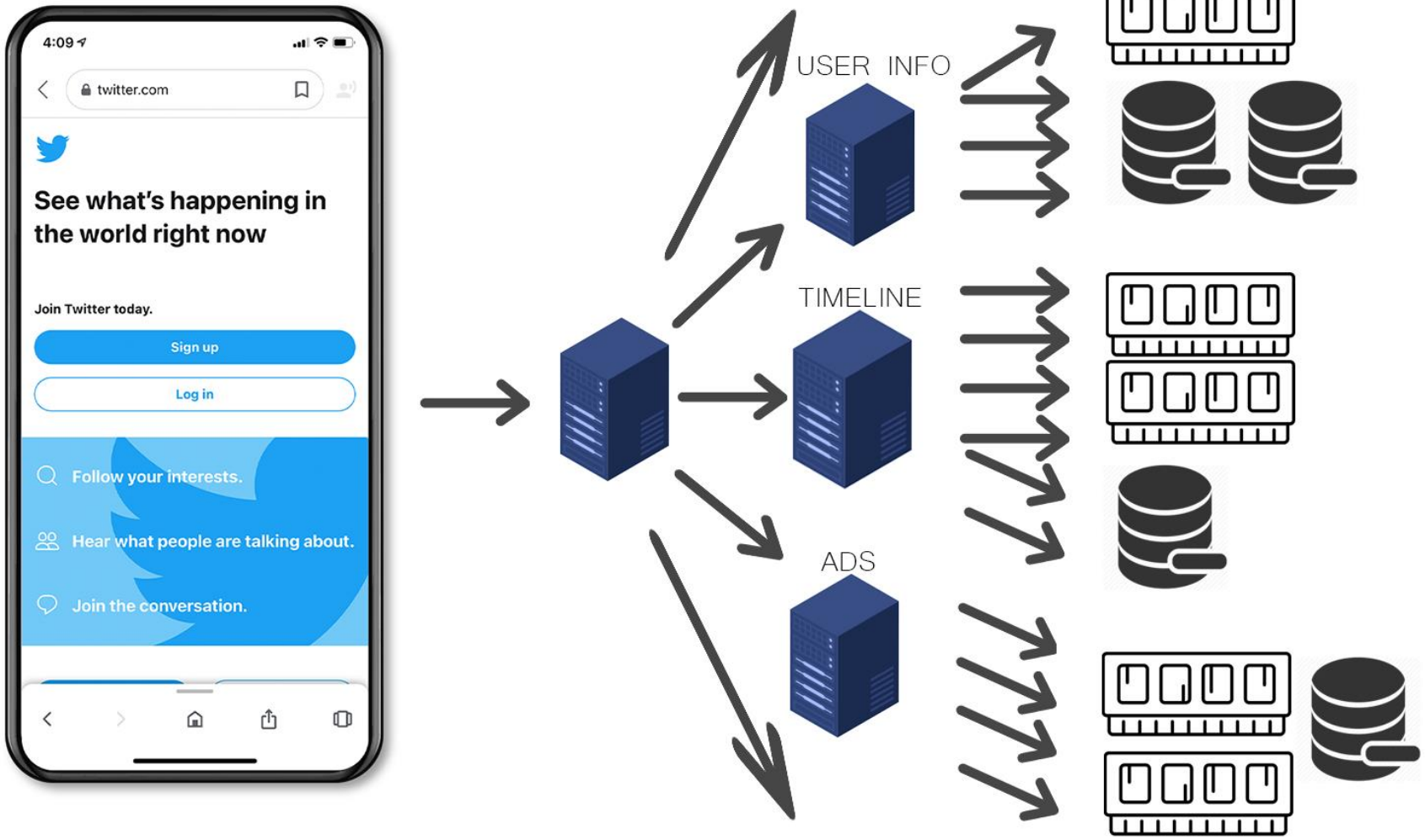
2021. 05. 12

Presentation by Agung Rahmat Ramadhan
agungnet@dankook.ac.kr

Table of contents

Background

In-memory caching is **ubiquitous** in the **modern web services**
To **reduce latency**, **increase throughput**, **reduce backend load**



How to use cache in memory?

Cache use cases

Write-heavy workloads

Object size distribution and evolution

Time-to-live (TTL) and working set

In-memory caches at Twitter

Timeline:

2011 – Microservices (Migration to a service-oriented arch)

2011 – Developing its container solution

2020 – Real-time service stack (hundreds of services running inside containers in production)

In-Memory Cache = A core component of Twitter's Infrastructure

- Grown alongside the transition above
- Petabytes of DRAM and hundreds of thousand of cores are provisioned for caching cluster

In-memory caches at Twitter

In-memory caching is a **Managed Service**

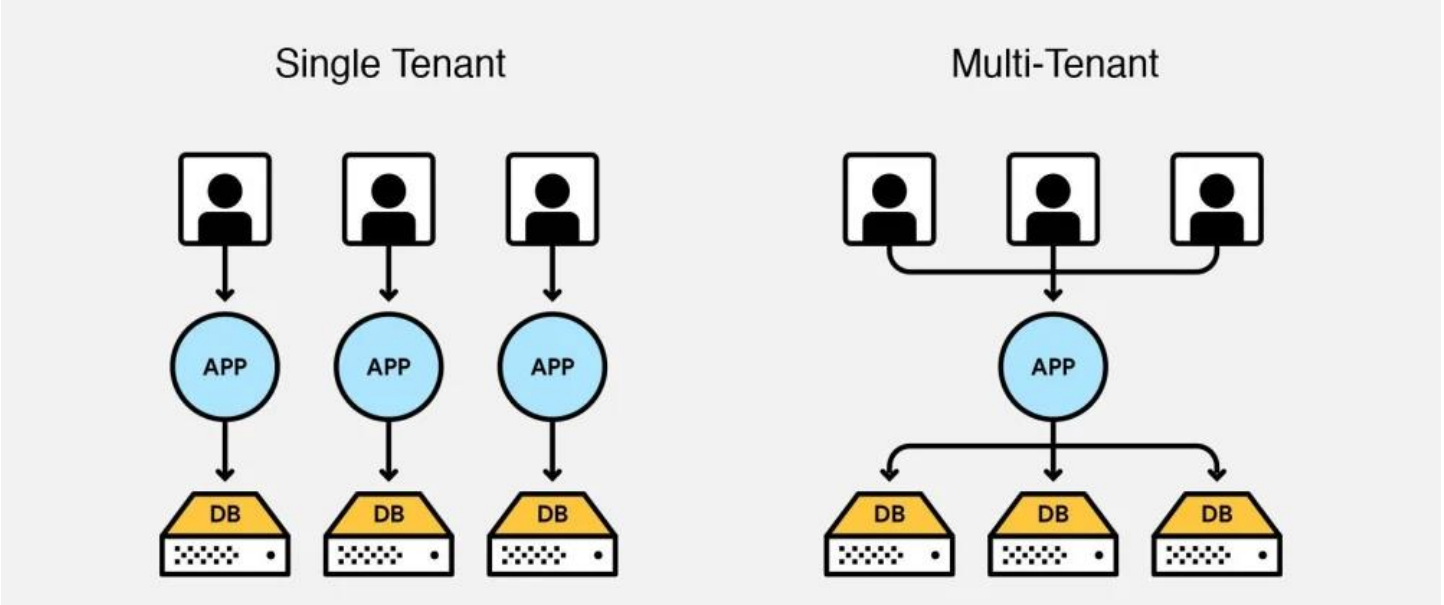
New Clusters are provisioned semi-automatic (look-aside cache)

Two in-memory caching solution:

1. **TwemCache** (Fork of Memcached) (Key-value cache)
2. Nighthawk (Redis-based)

In-memory caches at Twitter

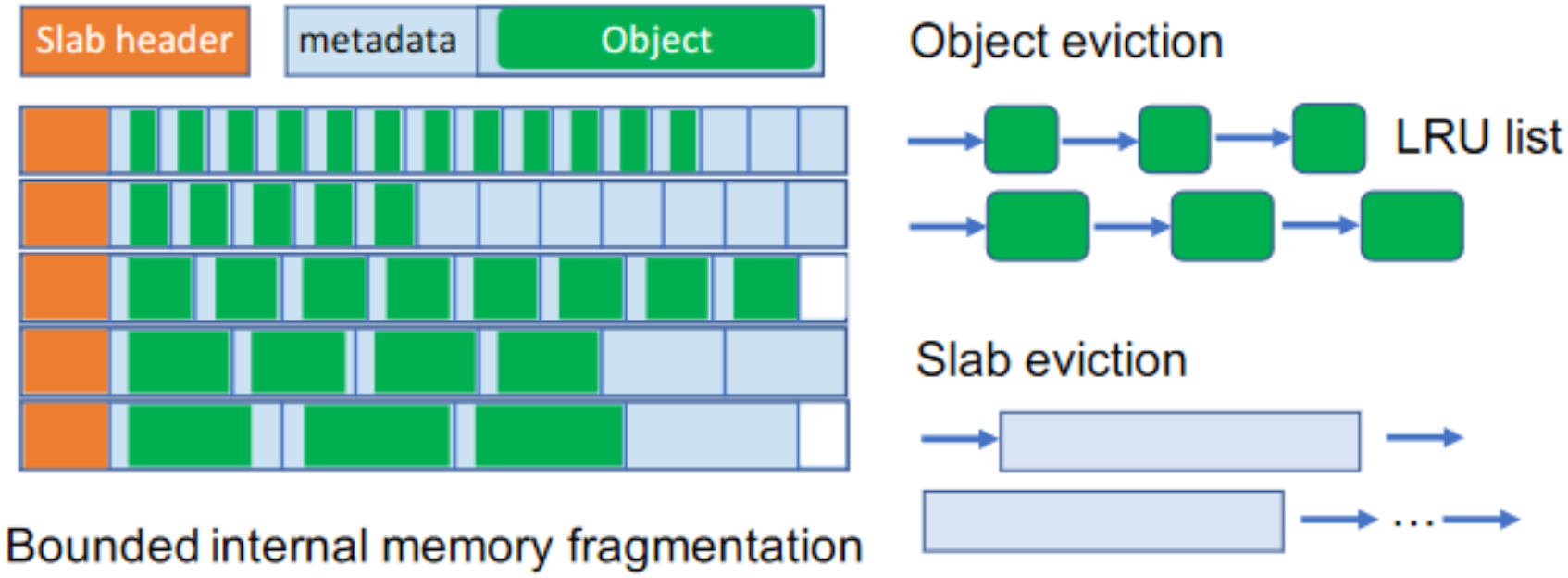
- Single tenant, single layer
  - Container-based deployment
- Large scale deployment
  - 100s cache clusters
  - 1s billion QPS
  - 100s TB DRAM
  - 100,000s CPU cores



Source: https://res.cloudinary.com/practicaldev/image

Overview of Twemcache

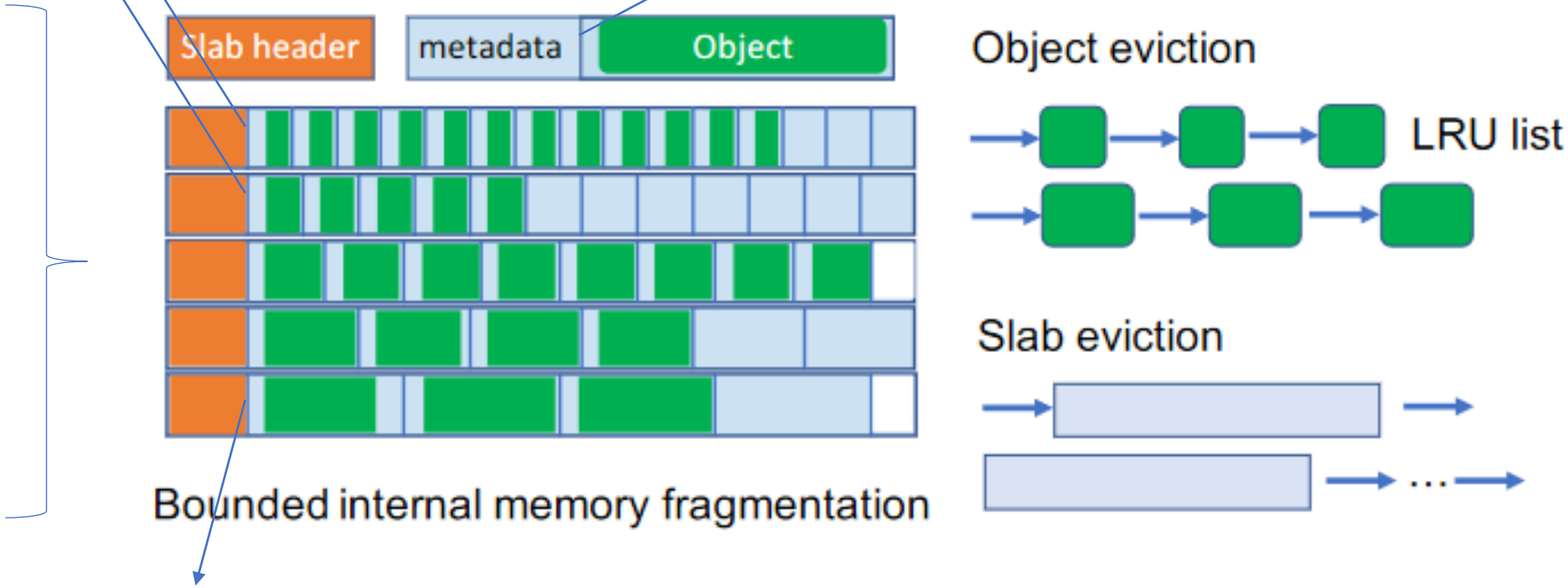
- Often stores small and variable-size objects (a few bytes ~ 10s of KB)
- Inherits the **Slab-based memory management**



Smaller chunks (**Items**)

49 bytes

Memory is allocated as fixed size chunks (**Slab**)

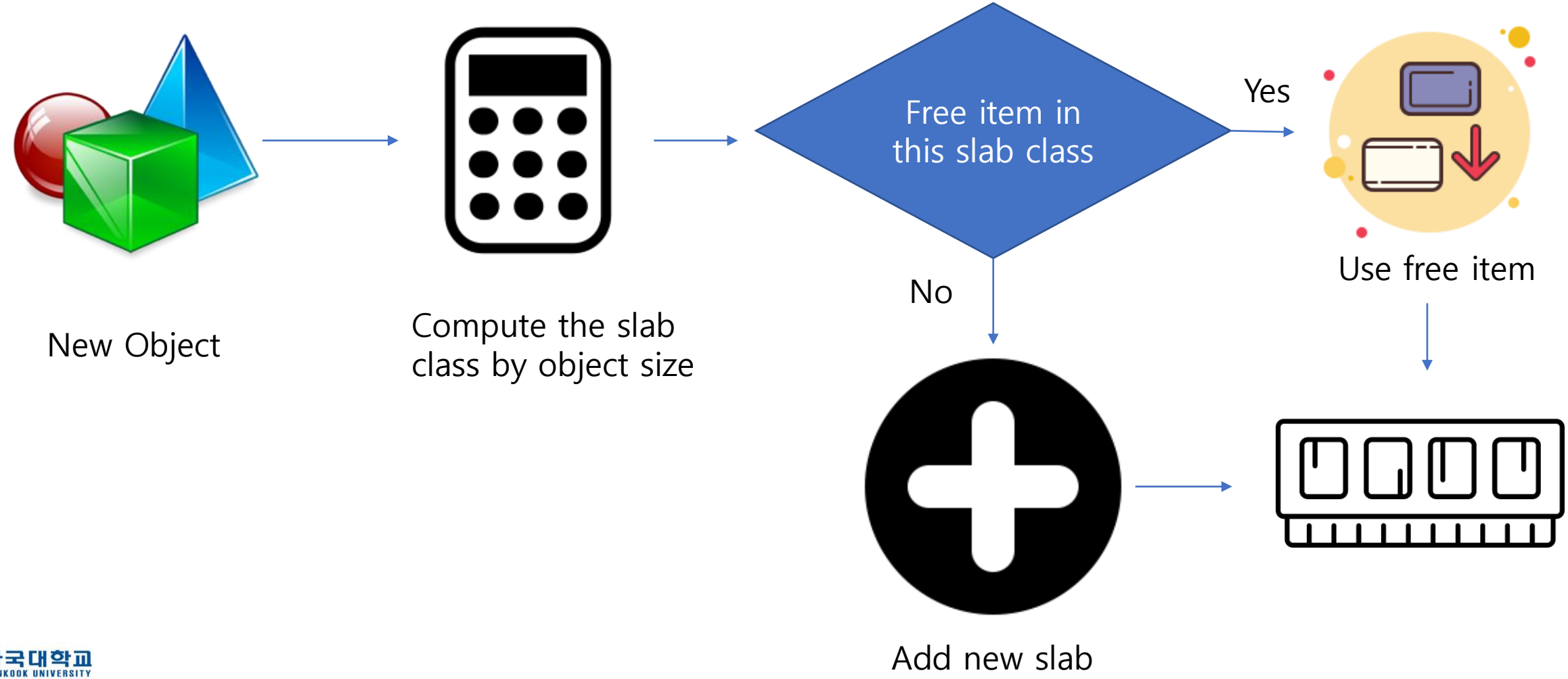


Def: 1MB

Default, Twemcache grows item size from **88 bytes** ~ **under whole slab**
The Growth controlled by Growth factor (1.25)

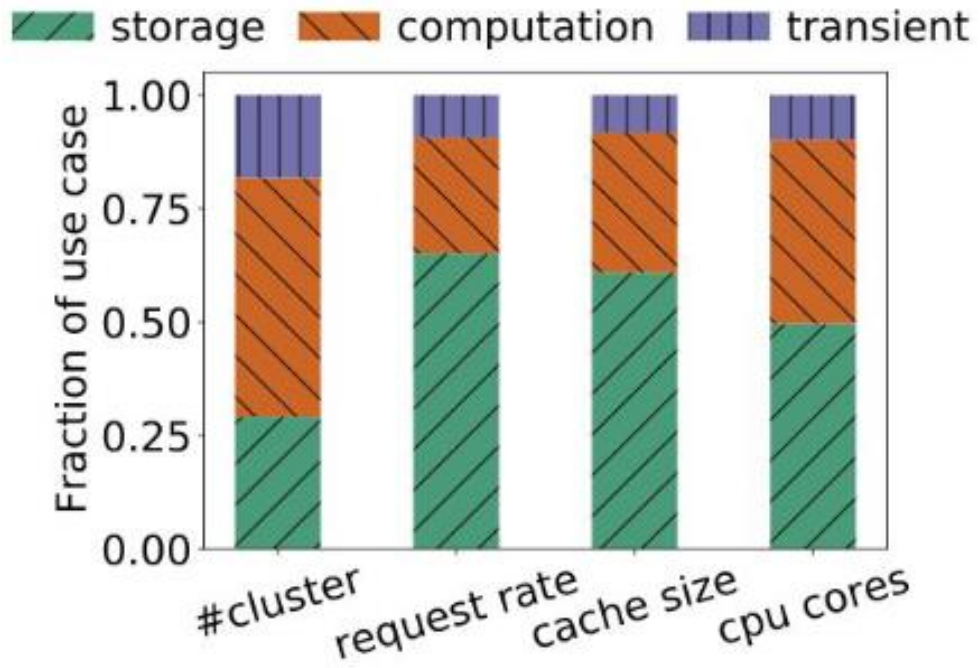
Class : **12**
Items : **891** items of **1176** bytes each
@ Item : **1127** bytes of key + val

Store new object in slab-based cache



Cache use cases

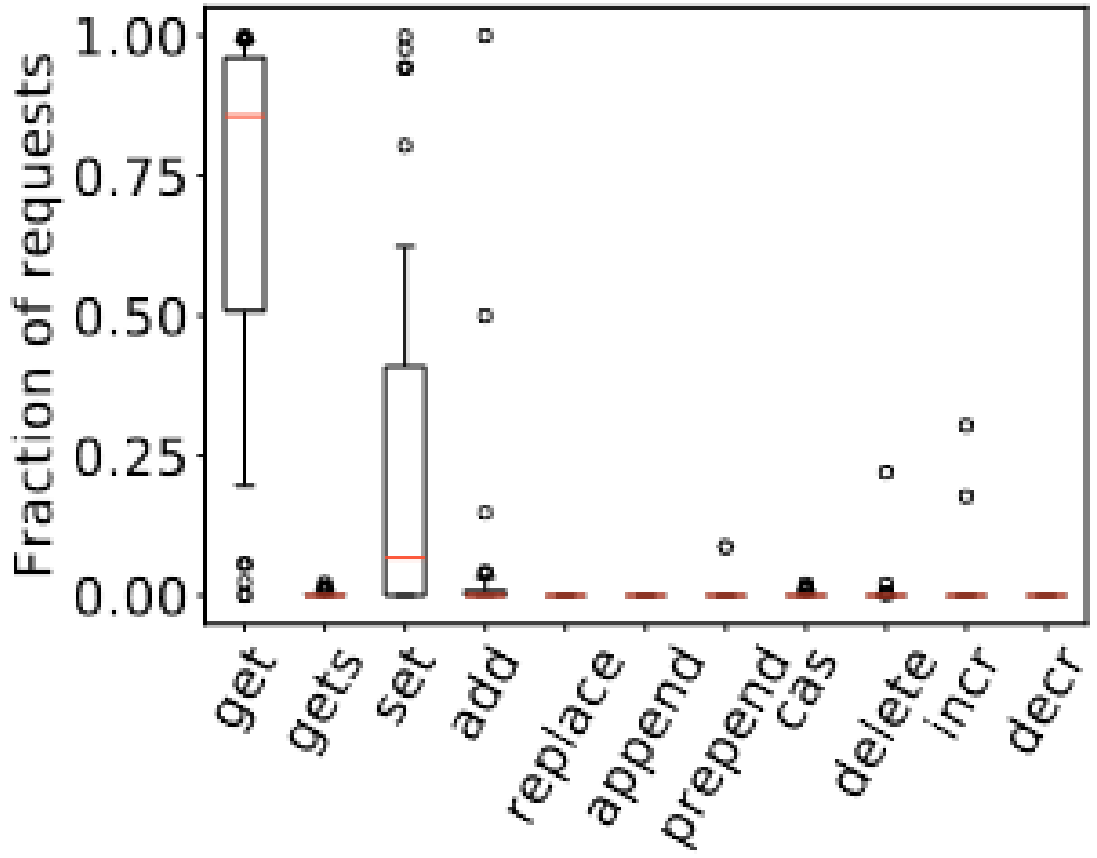
- Caching for storage
  - Most common and use most resources
- Caching for computation
  - Increasingly popular
  - Machine learning, stream processing
- Transient data with no backing store
  - Rate limiters
  - Negative caches



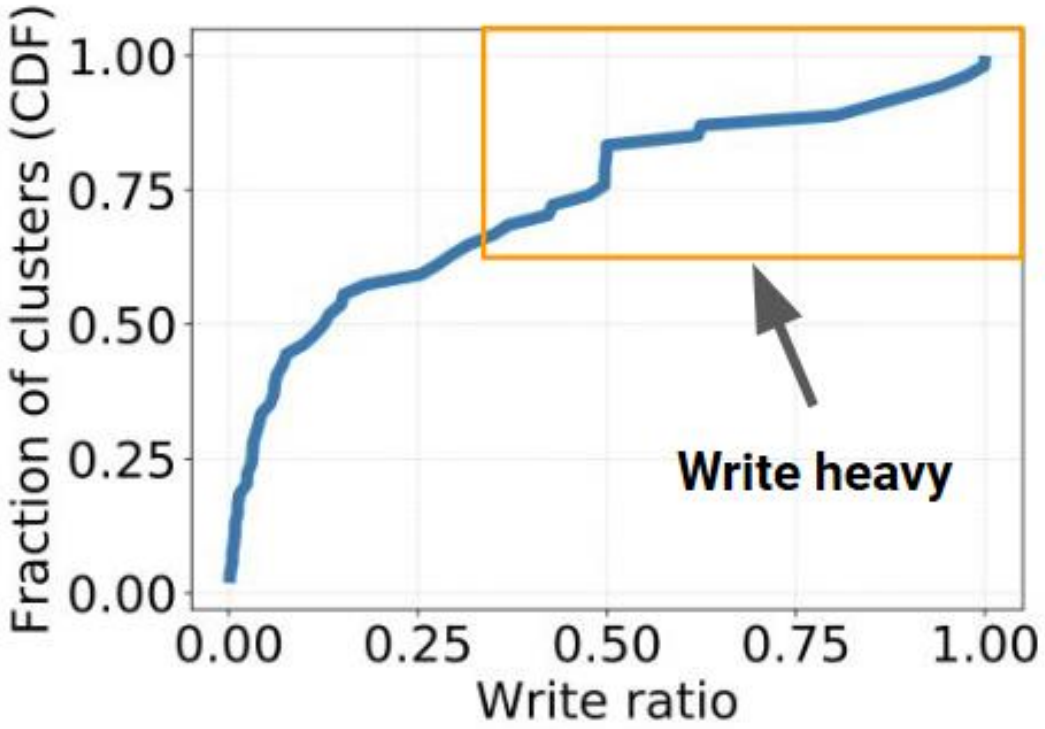
Trace collection and open source

- Source : Production traces from **153** in-memory cache clusters at **Twitter**
- Week-long unsampled traces from one instance of each Twemcache cluster
  - 700 billion requests, 80 TB in size
  - Focus on 54 representative clusters
- Traces are open source
  - https://github.com/twitter/cache-trace
  - https://github.com/Thesys-lab/cacheWorkloadAnalysisOSDI20

Relative Use of Each Operation



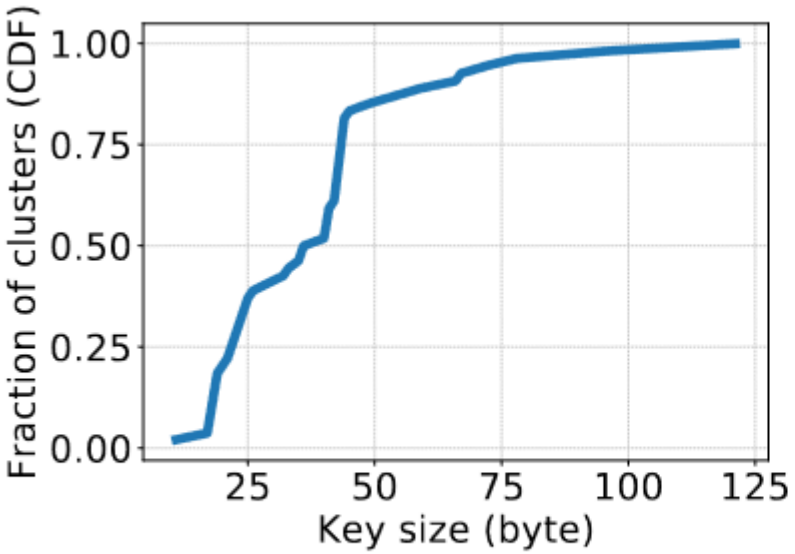
Write-heavy workloads analysis



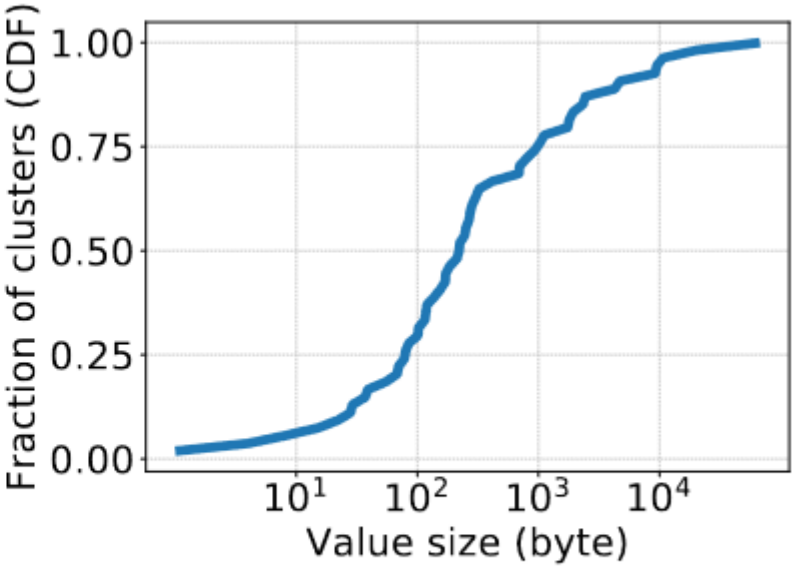
35% of clusters are write-heavy
(more than 30% writes)

Implication for future research:
- Optimization needed for write-heavy workloads
  - Challenges: scalability, tail latency

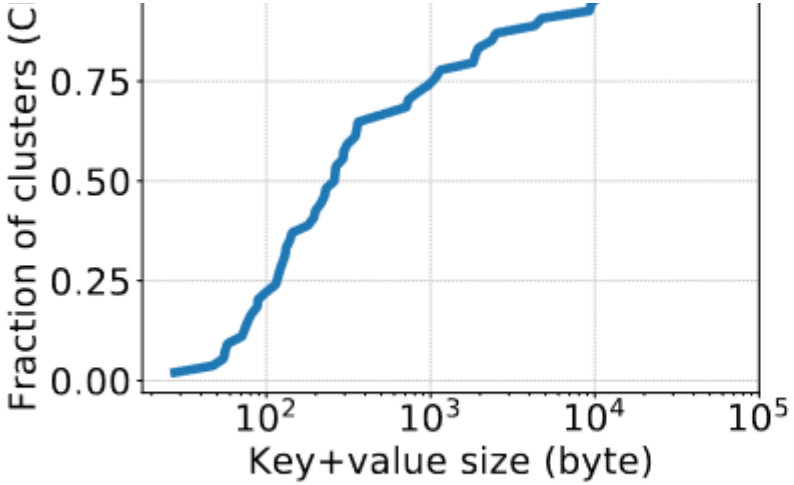
Object Size Analysis



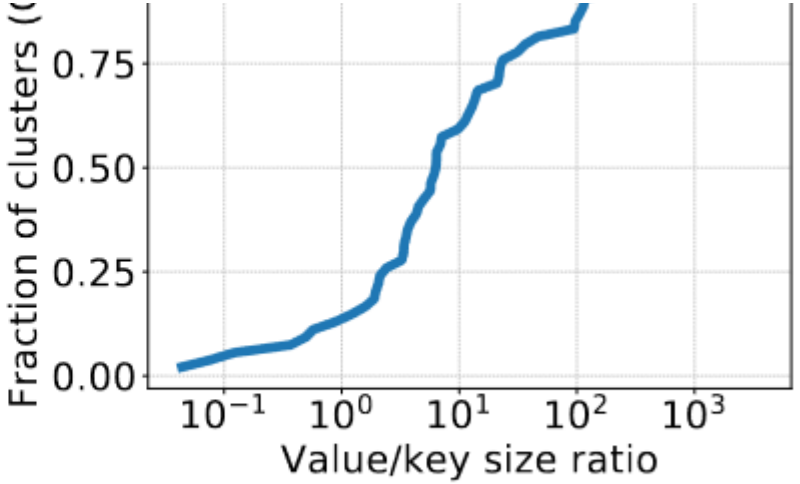
(a) Key size



(b) Value size

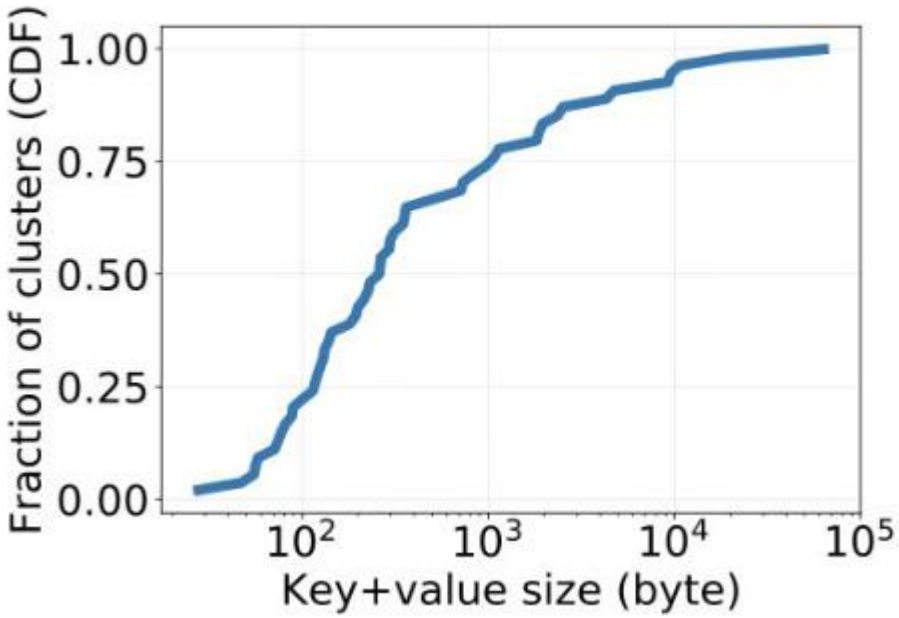


(c) Object size



(d) Value/key size ratio

Object size



Object sizes are small

- 24% cluster mean object size < 100 bytes
- Median 230 bytes

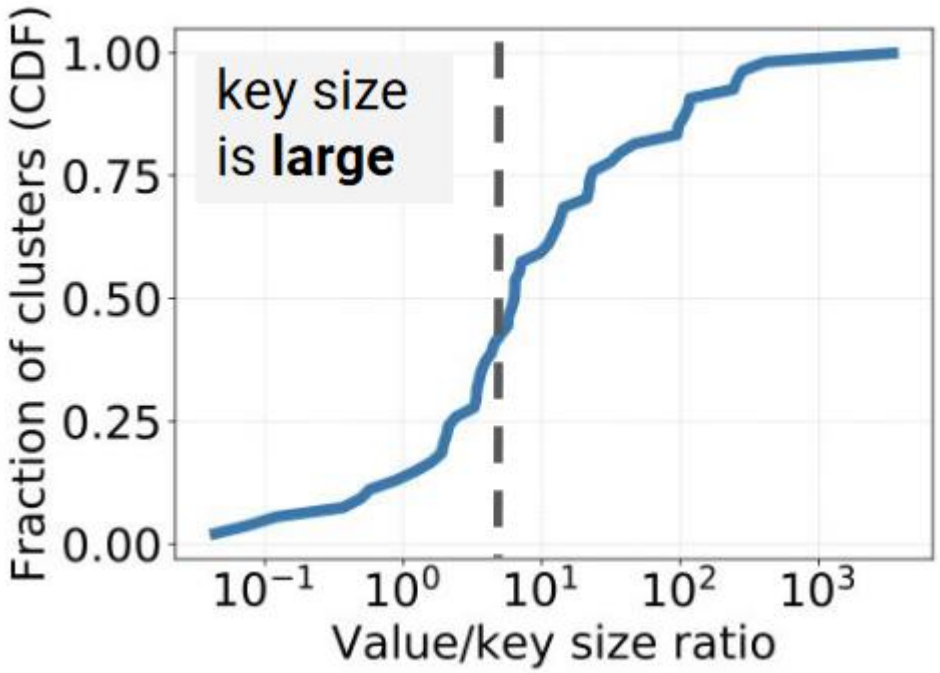
Metadata size is large

- Memcached uses 56 bytes per-obj metadata
- Research systems often add more metadata
- -> Reduce effective cache size

Implication for future research:

- Minimizing object metadata to increase effective cache size

Object size



Value/key size ratio can be small

- 15% cluster value size <= key size
- 50% cluster value size <= 5 x key size

Small value/key size ratio

- Name spaces are part of keys
  - Ns1:ns2:obj or obj/ns1/ns2

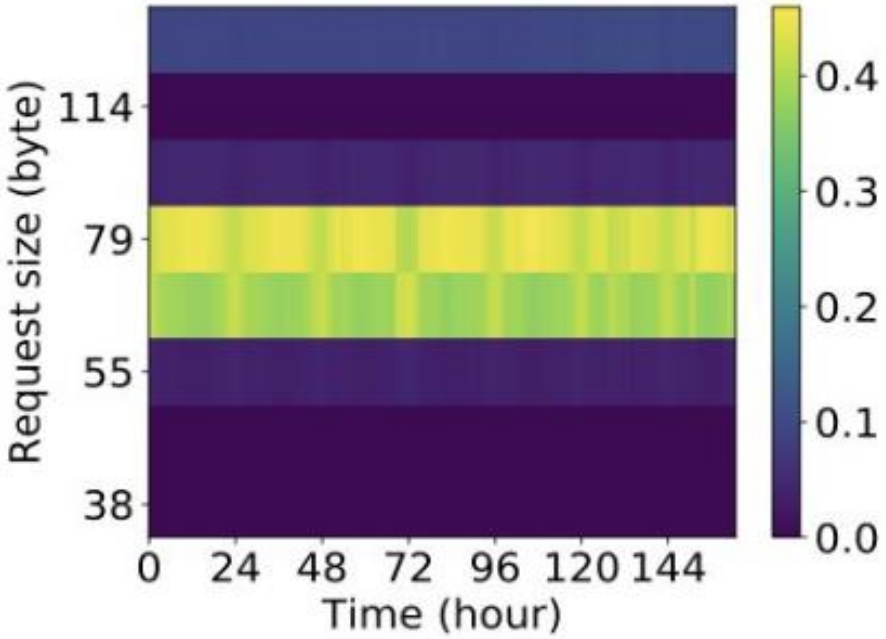
Implication for future research:

- A robust and lightweight key compression algorithm can increase effective cache size

Dynamic size distribution

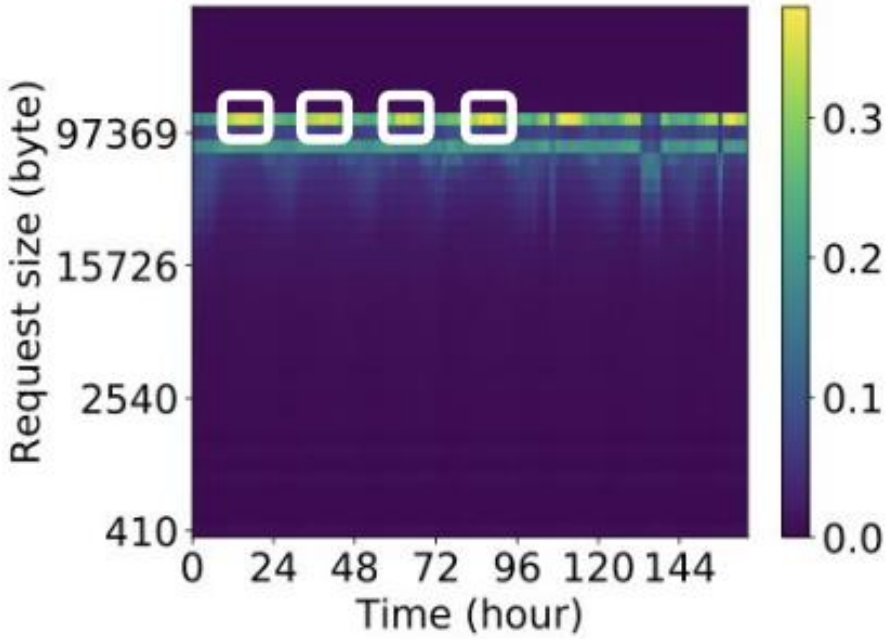
Size distribution can be static

Bright color: more requests are for objects of that size in the time window

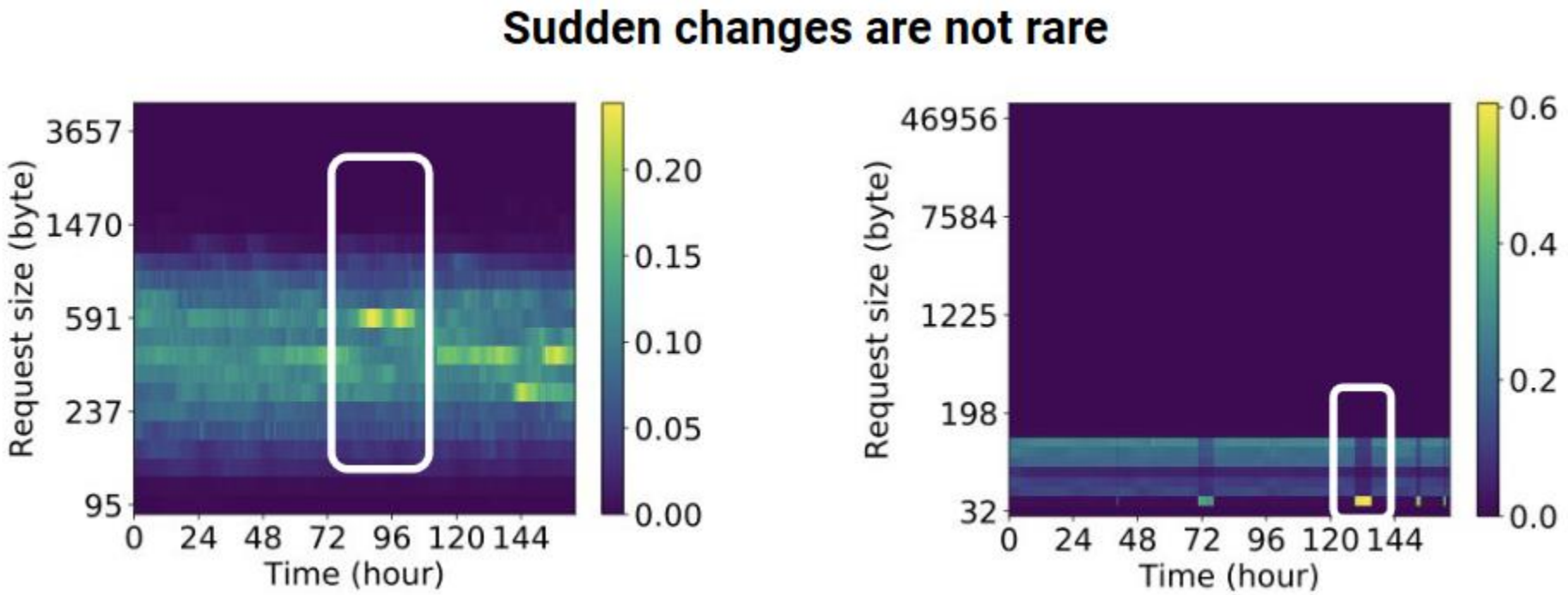


Most of the time, it is not static

The workload below shows a diurnal patterns



Size distribution over time



Implication for future research:

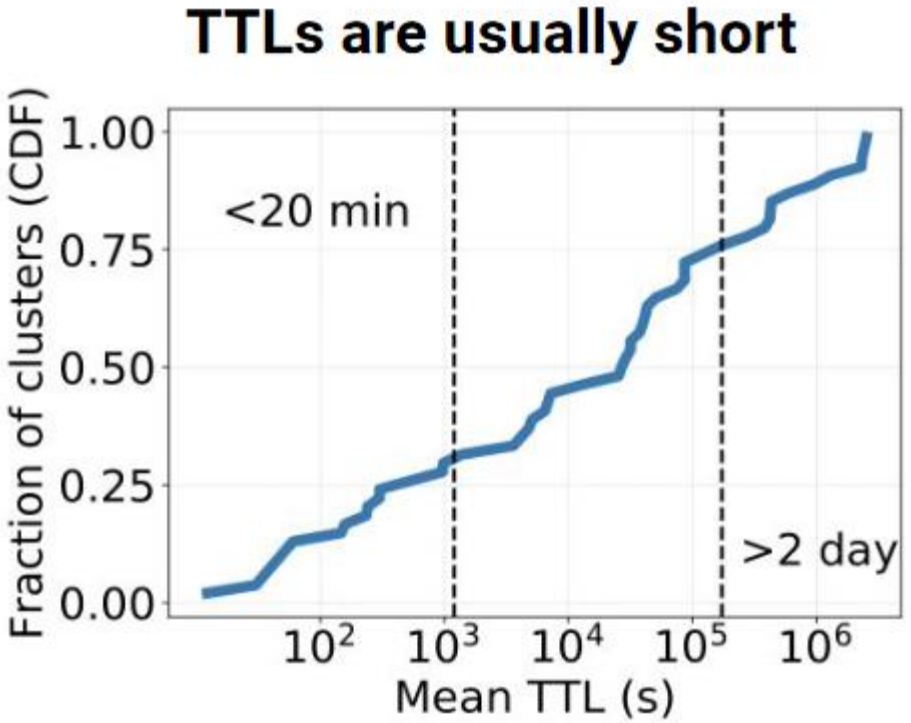
- Size distribution changes pose challenges to memory management
- Innovations needed on better memory management techniques

Time-to-live (TTL)

- How long an object can be used for serving requests
- Set during object writes
- Expired objects cannot be served

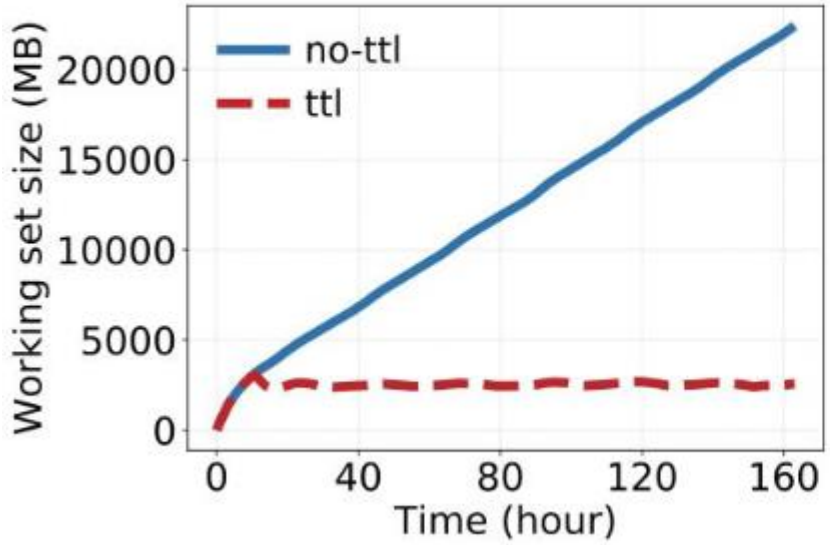
TTL use cases and usages

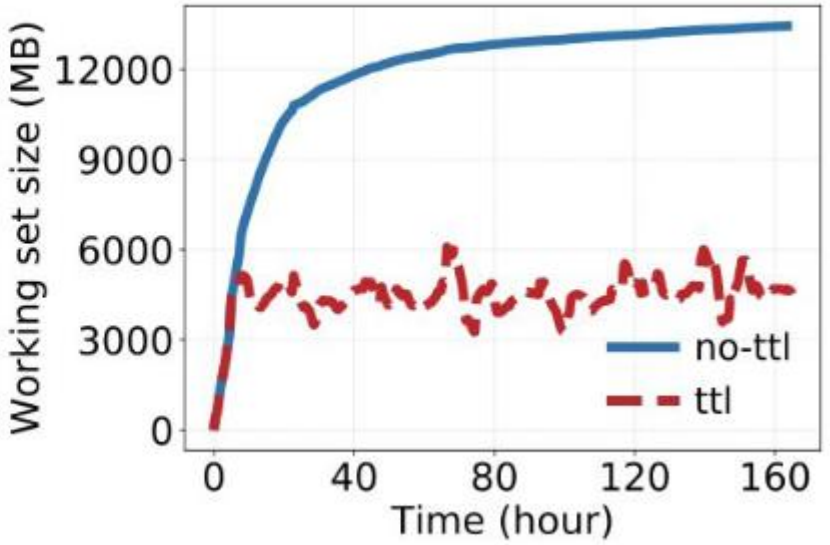
- Bounding inconsistency
  - Cache updates are best-effort
- Periodic refresh
  - Caches for computation store computation based on dynamic features
- Implicit deletion
  - Rate limiter
  - GDPR compliant



Short TTLs lead to bounded working set sizes

There is no need for a huge cache size if expired objects can be removed in time.

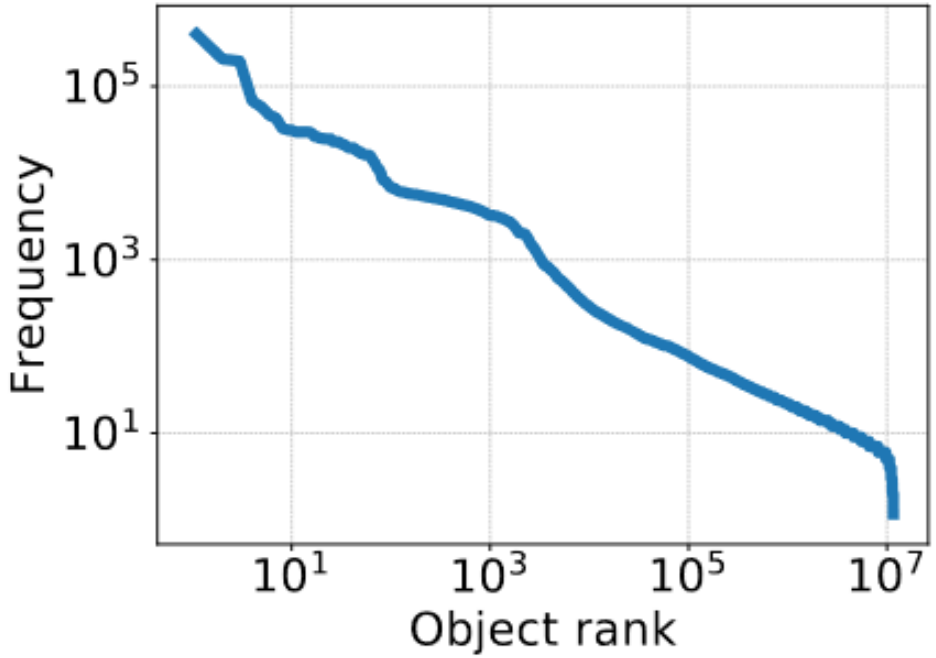




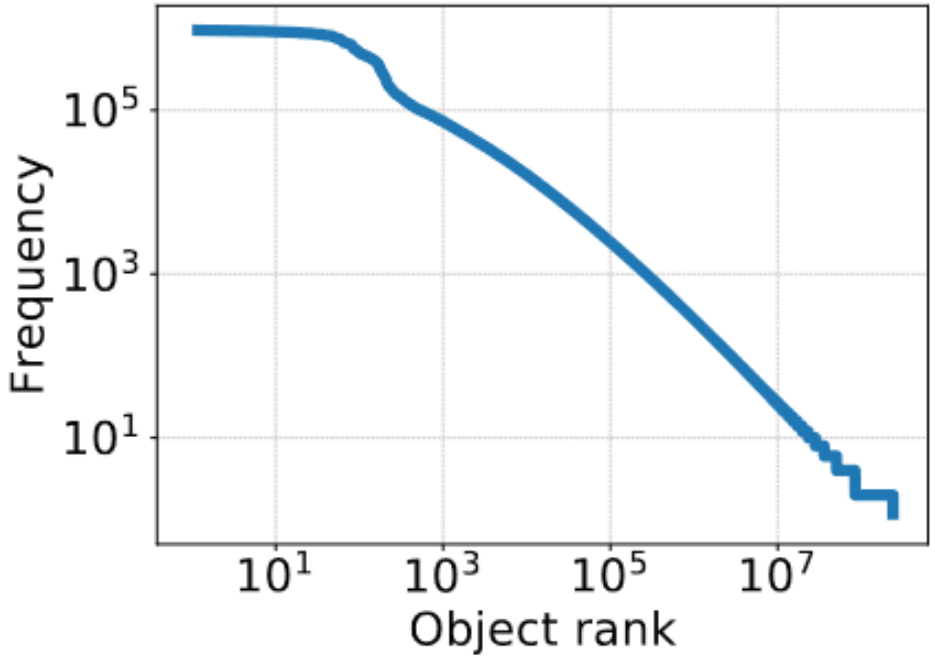
Implication for future research:

- Efficient proactive expiration techniques are more important than evictions
- Innovation needed on efficient TTL expiration

Cache workloads follow Zipfian distribution?

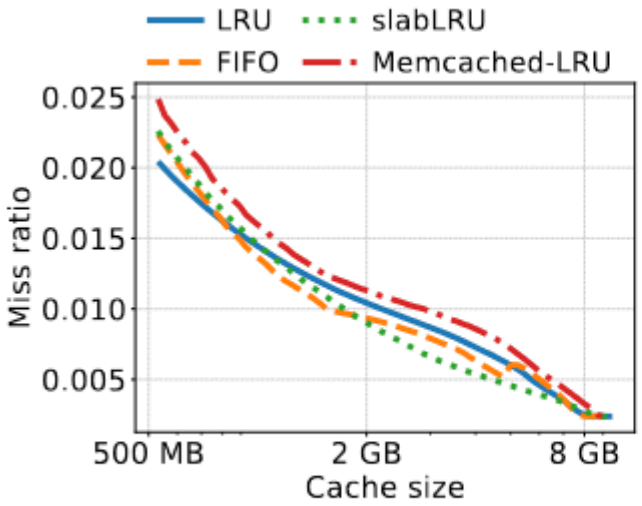


(a)

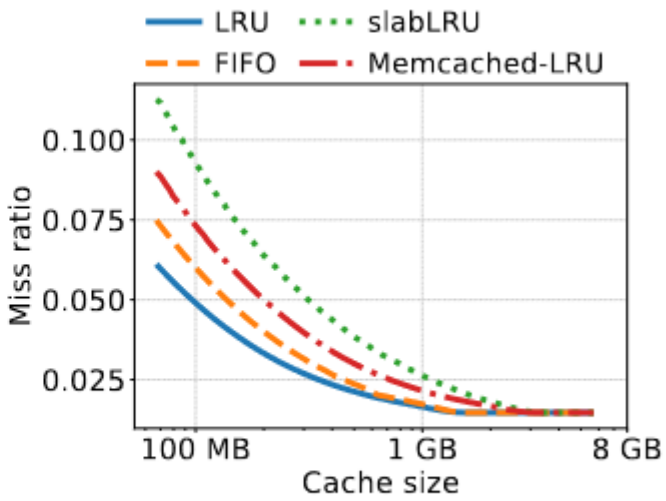


(b)

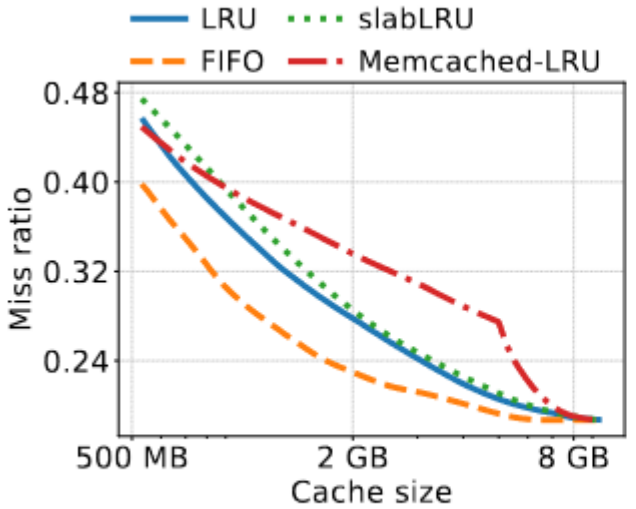
Eviction Algorithm Candidates (LRU vs FIFO)



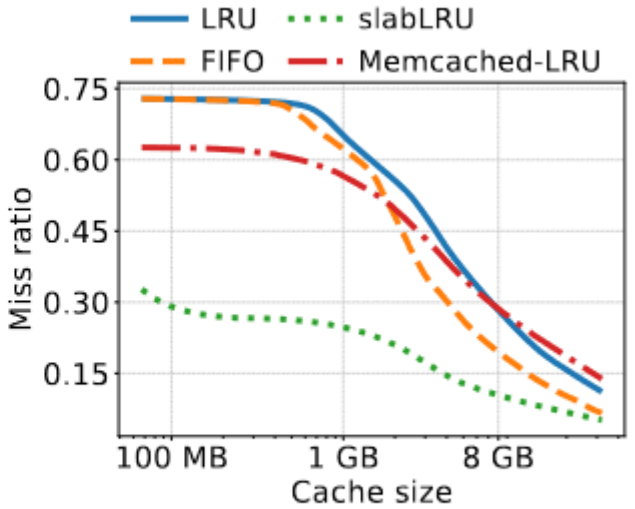
(a) Similar miss ratio



(b) LRU is better

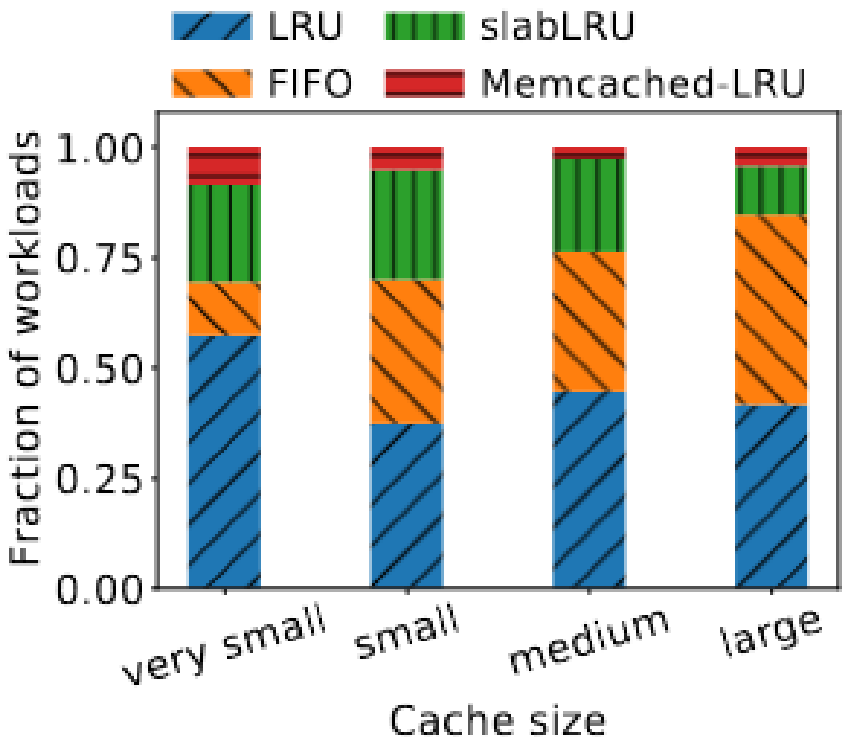


(c) FIFO is better

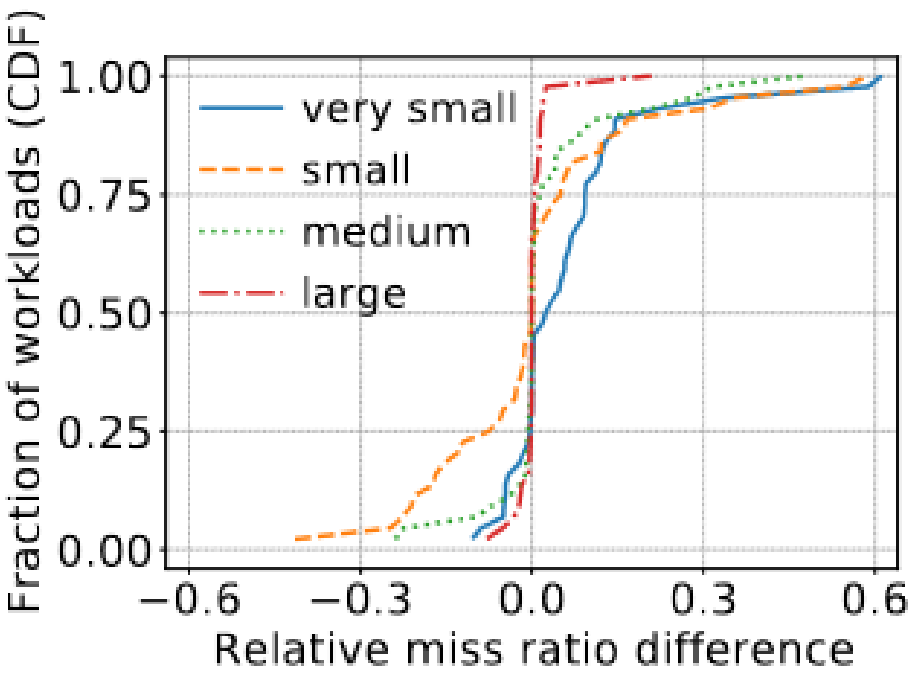


(d) slabLRU is better

Eviction Algorithm Candidates (LRU vs FIFO)



(a)



(b)

Conclusion

1. Although read-heavy workloads account for more than half of the resource usages, write-heavy workloads are also common.
2. In-memory caching clients often use short TTLs, which limits the effective working set size. Thus, removing expired objects needs to be prioritized before evictions.
3. Read-heavy in-memory caching workloads follow Zipfian popularity distribution with a large skew.
4. The object size distributions of most workloads are not static. Instead, it changes over time with both diurnal patterns and sudden changes, highlighting the importance of slab migration for slab-based in-memory caching systems.
5. For a significant number of workloads, FIFO has similar or lower miss ratio performance as LRU for in-memory caching workloads.