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

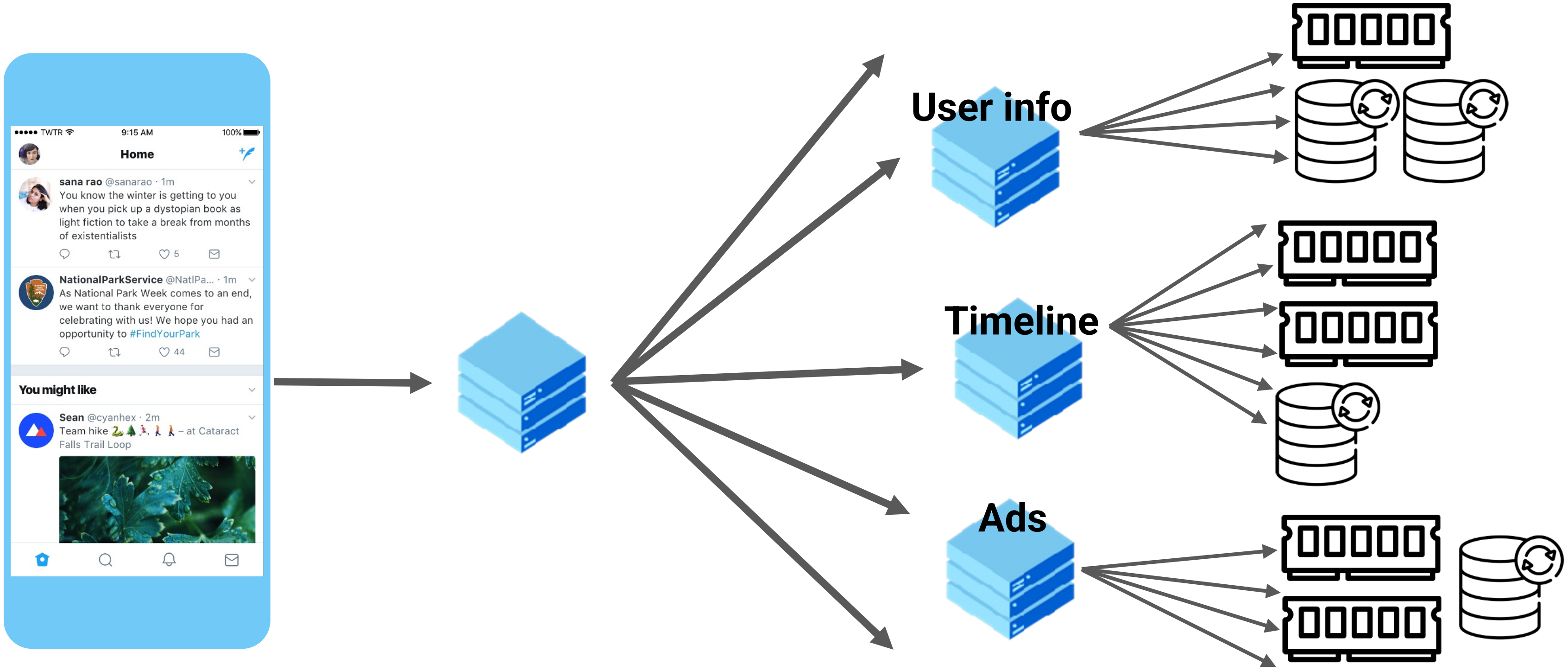
Juncheng Yang, Yao Yue, Rashmi Vinayak
Carnegie Mellon University & Twitter

Carnegie Mellon University
School of Computer Science

Background

In-memory caching is ubiquitous in the modern web services

To reduce latency, increase throughput, reduce backend load



How are in-memory caches used?

Do existing assumptions still hold?

Cache use cases

Types of operations

Object size distribution and evolution

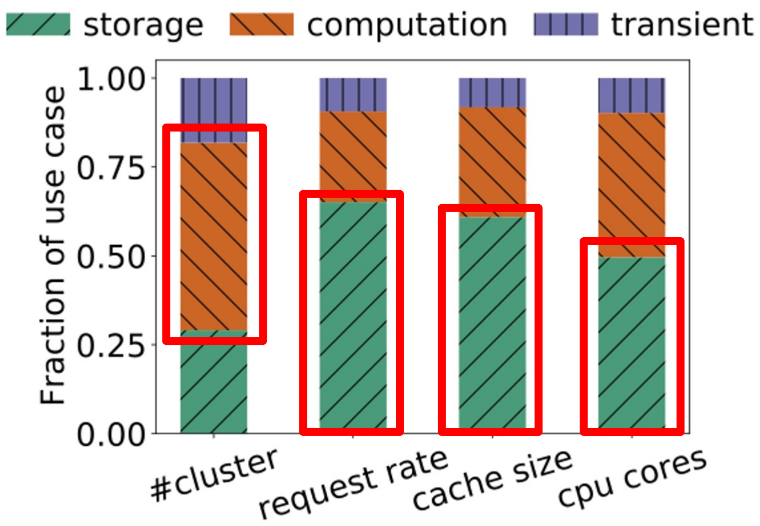
Time-to-live (TTL) and working set

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

Cache use cases

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



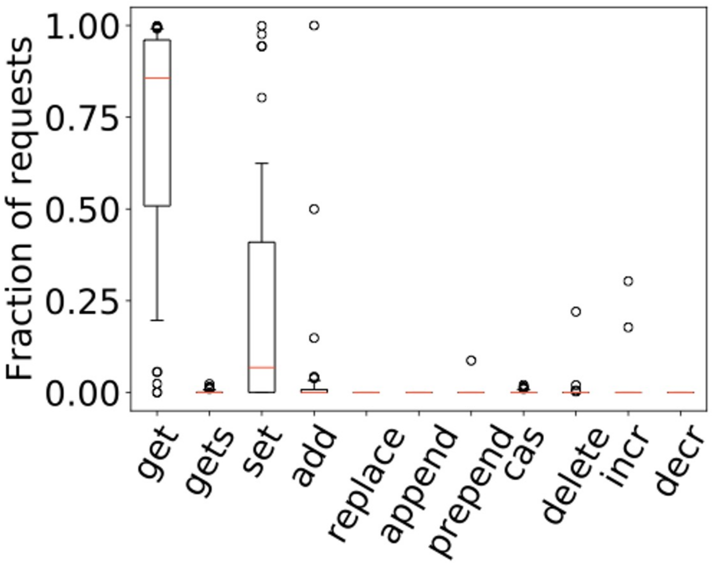
Trace collection and open source

- 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

Types of operations

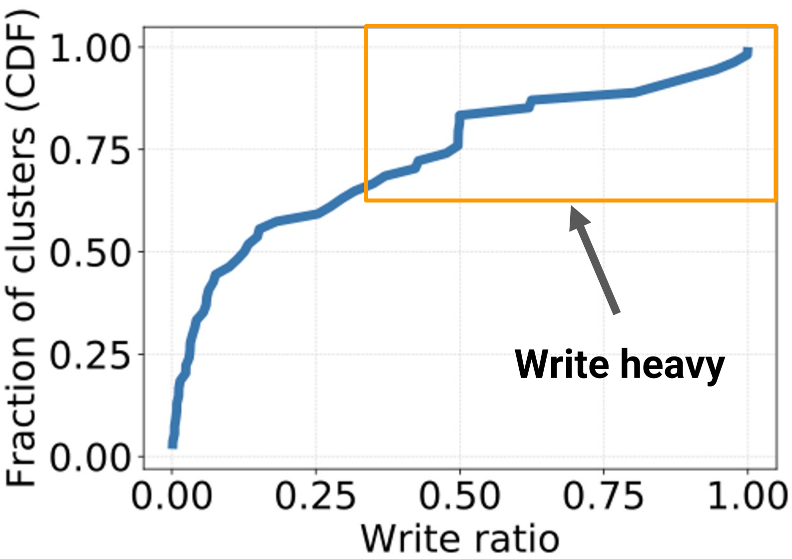
Types of operations

`get`/`set` are most common



Optimize for less frequent operations

35% of clusters are write-heavy (30%)



Optimize for write-heavy workloads

- Challenging: scalability, tail latency

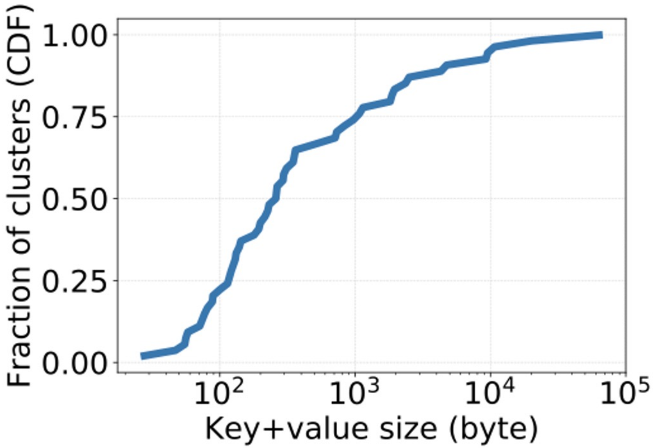
`cas`: compare and set

Object size

Object size

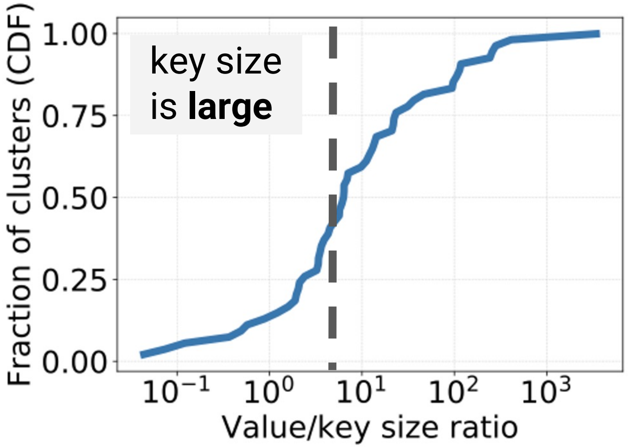
Object sizes are small

- 25% cluster mean object size < 100 bytes
- Median 230 bytes



Value/key size ratio can be small

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



Overhead of metadata

- Memcached uses 56 bytes per-obj metadata
- Research systems often add more metadata

Small value/key size ratio

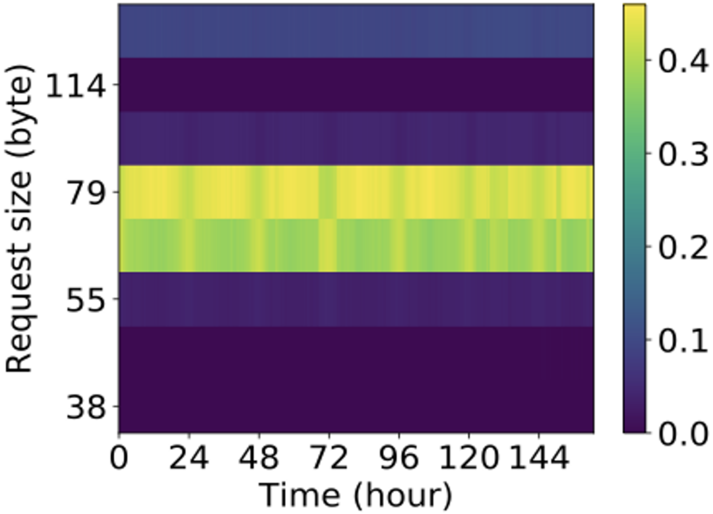
- Name spaces are part of keys
  - `Ns1:ns2:obj` or `obj/ns1/ns2`
- Robust and lightweight key compression

Dynamic size distribution

Size distribution over time

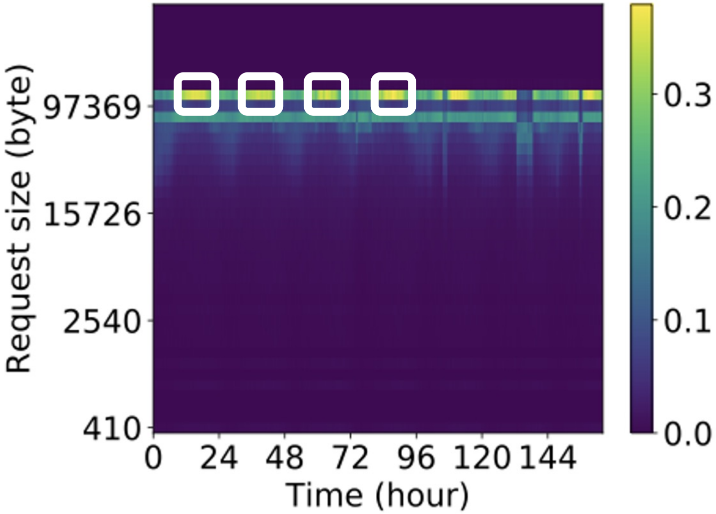
Size distribution can be static

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



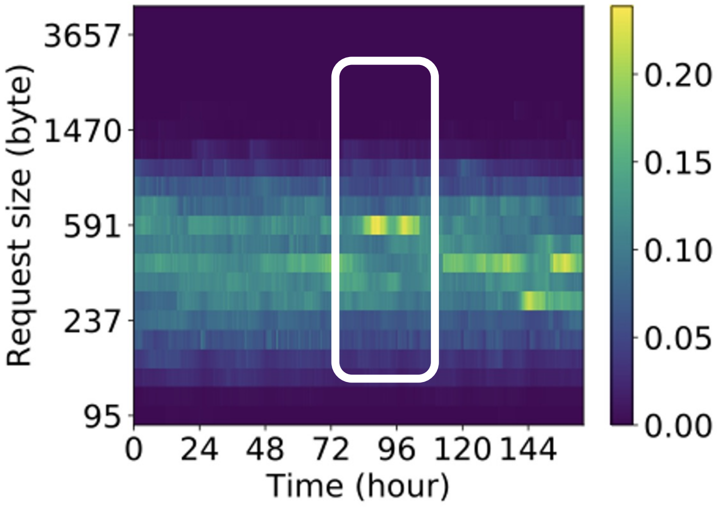
Most of the time, it is not static

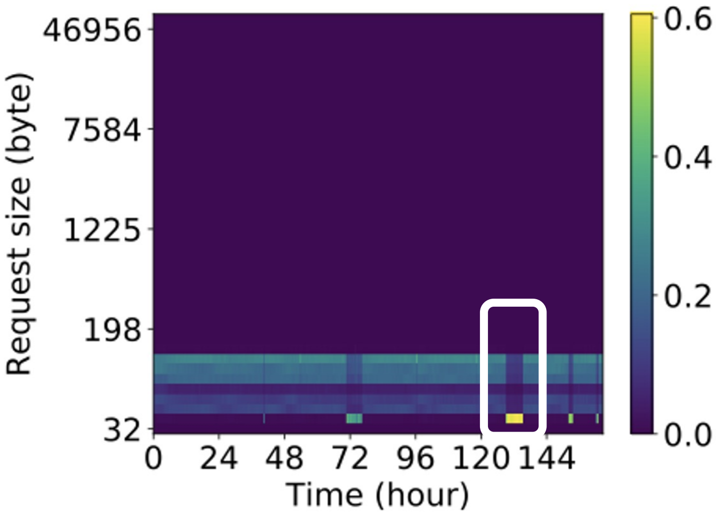
The workload below shows a diurnal patterns



Size distribution over time

Sudden changes are not rare





Size distribution changes make memory management difficult

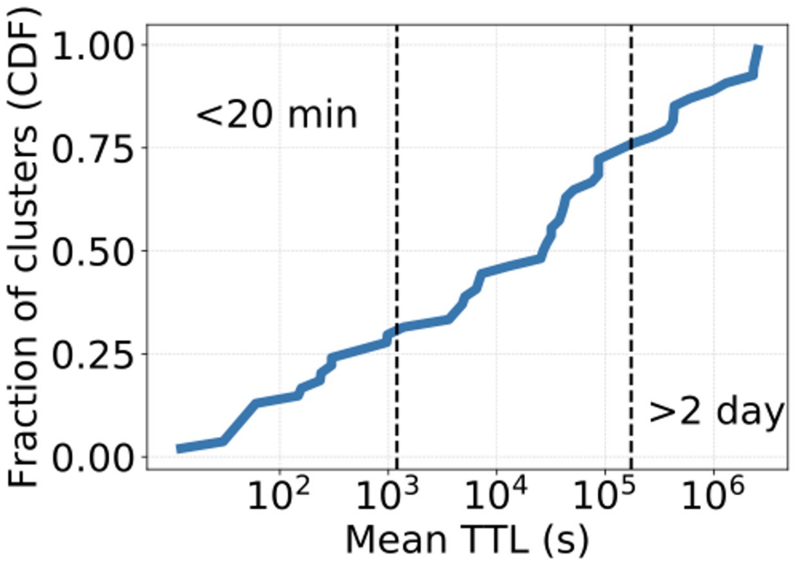
- Sub-optimal slab migration
- Innovations needed on better strategies

Time-to-live (TTL)

- TTLs are set during writes
- Expired objects cannot be served

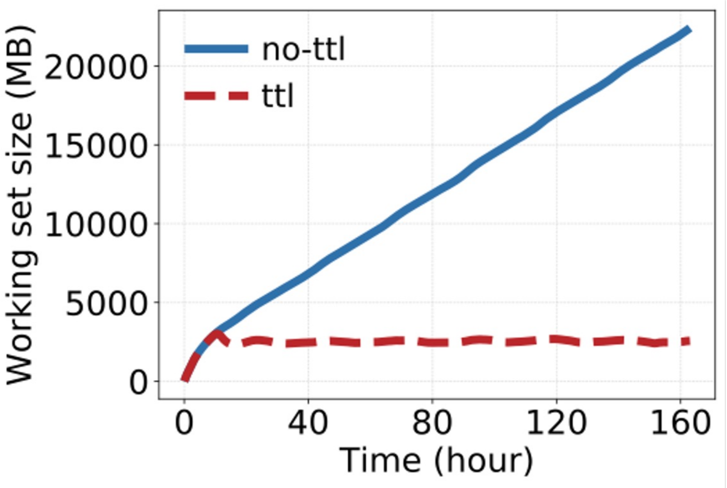
TTL use cases and usages

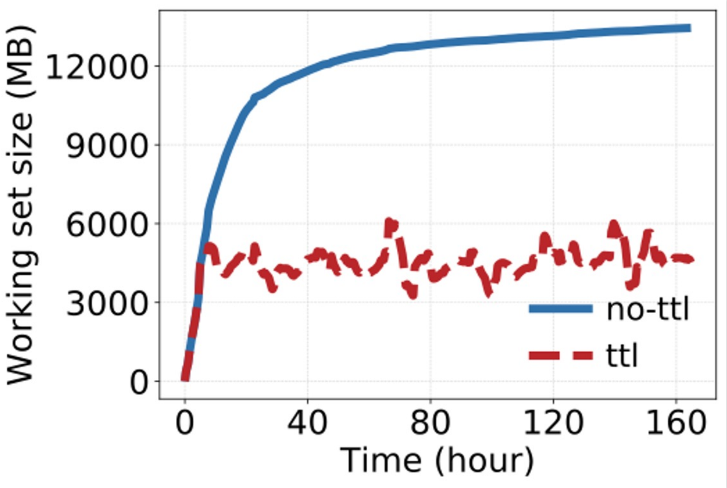
- Bounding inconsistency
  - Cache updates are best-effort
- Periodic refresh
  - Computation
- Implicit deletion
  - Rate limiter
  - GDPR



TTLs are usually short

Short TTLs lead to bounded working set sizes

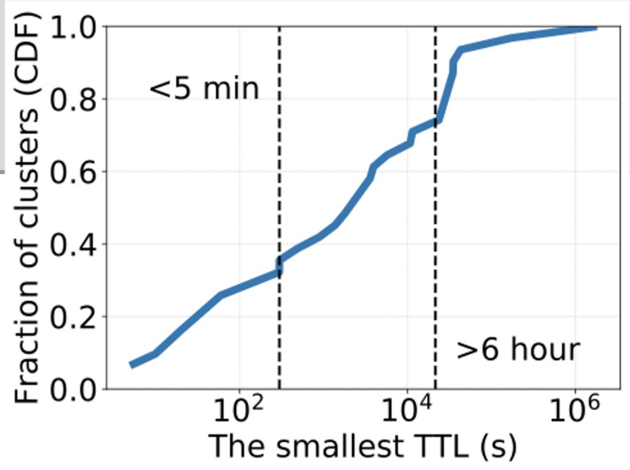


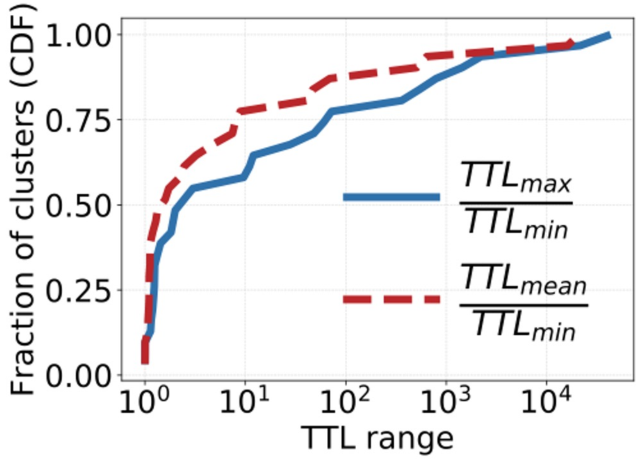


There is no need for a huge cache size if expired objects can be removed in time

Implications of short TTLs

- Existing TTL expiration approaches
  - Remove upon next access
  - Transient object pool
  - Scanning full cache
  - Sampling
- Existing approaches are not sufficient
- Innovation needed on efficient TTL expiration





More in the paper

Production statistics

- Small miss ratio and small variations
- Request spikes are not always caused by hot keys

Object popularity

- Mostly Zipfian with large parameter alpha
- Small deviations

Eviction algorithms

- Highly workload dependent
- Four types of results
- FIFO achieves similar miss ratios as LRU

Non-trivial fraction of write-heavy workloads

Small objects, expensive metadata

Dynamic object size distribution

Wide TTL usage, proactive expiration > eviction

Traces are available at
https://github.com/twitter/cache-trace
https://github.com/Thesys-lab/cacheWorkloadAnalysisOSDI20

Contact: juncheny@cs.cmu.edu