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

Juncheng Yang, *Carnegie Mellon University;* Yao Yue, *Twitter;* K. V. Rashmi, *Carnegie Mellon University*

https://www.usenix.org/conference/osdi20/presentation/yang

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

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

Juncheng Yang
Carnegie Mellon University

Yao Yue
Twitter

K. V. Rashmi
Carnegie Mellon University

Abstract

Modern web services use in-memory caching extensively to increase throughput and reduce latency. There have been several workload analyses of production systems that have fueled research in improving the effectiveness of in-memory caching systems. However, the coverage is still sparse considering the wide spectrum of industrial cache use cases. In this work, we significantly further the understanding of real-world cache workloads by collecting production traces from 153 in-memory cache clusters at Twitter, sifting through over 80 TB of data, and sometimes interpreting the workloads in the context of the business logic behind them. We perform a comprehensive analysis to characterize cache workloads based on traffic pattern, time-to-live (TTL), popularity distribution, and size distribution. A fine-grained view of different workloads uncover the diversity of use cases: many are far more write-heavy or more skewed than previously shown and some display unique temporal patterns. We also observe that TTL is an important and sometimes defining parameter of cache working sets. Our simulations show that ideal replacement strategy in production caches can be surprising, for example, FIFO works the best for a large number of workloads.

1 Introduction

In-memory caching systems such as Memcached [14] and Redis [16] are heavily used by modern web applications to reduce accesses to storage and avoid repeated computations. Their popularity has sparked a lot of research, such as reducing miss ratio [26, 28, 36, 37], or increasing throughput and reducing latency [43, 52, 53, 56]. On the other hand, the effectiveness and performance of in-memory caching can be workload dependent. And several important workload analyses against production systems [24, 59] have guided the explorations of performance improvements with the right context and tradeoffs in the past decade [43, 56].

Nonetheless, there remains a significant gap in the understanding of current in-memory caching workloads. Firstly, there has been a lack of comprehensive studies covering the wide range of use cases in today's production systems. Secondly, there have been new trends in in-memory caching usage since the publication of previous work [24]. Thirdly, some aspects of in-memory caching received little attention in the existing studies, but are known as critical to practitioners. For example, TTL is an important aspect of configuring in-memory caching, but it has largely been overlooked in research. Last but not least, unlike other areas where open-source traces [62, 63, 68, 70] or benchmarks [38] are available, there has been a lack of open-source in-memory caching traces. Researchers have to rely on storage caching traces [26], key-value database benchmarks [43, 56] or synthetic workloads [39, 57] to evaluate in-memory caching systems. Such sources either have different characteristics or do not capture all the characteristics of production in-memory caching workloads. For example, key-value database benchmarks and synthetic workloads don't consider how object size distribution changes over time, which impacts both miss ratio and throughput of in-memory caching systems.

In this work, we bridge this gap by collecting and analyzing workload traces from 153 Twemcache [6] clusters at Twitter, one of the most influential social media companies known for its real-time content. Our analysis sheds light on several vital aspects of in-memory caching overlooked in existing studies and identifies areas that need further innovations. The traces used in this paper are made available to the research community [1]. To the best of our knowledge, this is the first work that studied over 100 different cache workloads covering a wide range of use cases. We believe these workloads are representative of cache usage at social media companies and beyond, and hopefully provide a foundation for future caching system designs. Here's a summary of our discoveries:

1. In-memory caching does not always serve read-heavy workloads, write-heavy (defined as write ratio > 30%) workloads are very common, occurring in more than 35% of the 153 cache clusters we studied.
2. TTL must be considered in in-memory caching because it limits the effective (unexpired) working set size. Efficiently removing expired objects from cache needs to be prioritized over cache eviction.
3. In-memory caching workloads follow approximate Zipfian popularity distribution, sometimes with very high skew. The workloads that show the most deviations tend to be write-heavy workloads.
4. The object size distribution is not static over time. Some workloads show both diurnal patterns and experience sudden, short-lived changes, which pose challenges for slab-based caching systems such as Memcached.

5. Under reasonable cache sizes, FIFO often shows similar performance as LRU, and LRU often exhibits advantages only when the cache size is severely limited.

These findings provide a detailed new look into production in-memory caching systems, while unearthing some surprising aspects not conforming to the folklore and to the commonly used assumptions.

2 In-memory Caching at Twitter

2.1 Service Architecture and Caching

Twitter started its migration to a service-oriented architecture, also known as microservices, in 2011 [8]. Around the same time, Twitter started developing its container solution [2, 3] to support the impending wave of services. Fast forward to 2020, the real-time serving stack is mostly service-oriented, with hundreds of services running inside containers in production. As a core component of Twitter's infrastructure, in-memory caching has grown alongside this transition. Petabytes of DRAM and hundreds of thousands of cores are provisioned for caching clusters, which are containerized.

At Twitter, in-memory caching is a managed service, and new clusters are provisioned semi-automatically to be used as look-aside cache [59] upon request. There are two in-memory caching solutions deployed in production, Twemcache, a fork of Memcached [14], is a key-value cache providing high throughput and low latency. The other solution, named Nighthawk, is Redis-based and supports rich data structures and replication for data availability. In this work, we focus on Twemcache because it serves the majority of cache traffic.

Cache clusters at Twitter are considered *single-tenant*[1] based on the service team requesting them. This setup is very beneficial to workload analysis, because it allows us to tag use cases, collect traces, and study the properties of workloads individually. A multi-tenant setup will make similar study extremely difficult, as researchers have to tease out individual workloads from the mixture, and somehow connect them to their use cases. In addition, smaller but distinct workloads can easily be overlooked or mis-characterized due to low traffic.

Unlike other cache cluster deployments, such as social graph caching [19, 30] or CDN caching [47, 69], Twemcache is mostly deployed as a single-layer cache, which allows us to analyze the requests directly from clients without being filtered by other caches. Previous work [47] has shown that layering has an impact on properties of caching workloads, such as popularity distribution. This single-tenant, single-layer design provides us the perfect opportunity to study the properties of the workloads.

2.2 Twemcache Provisioning

There are close to 200 Twemcache clusters in each data center as of writing. Twemcache containers are highly homogeneous and typically small, and a single host can run many of them. The number of instances provisioned for each cache cluster is computed from user inputs including throughput, estimated dataset sizes, and fault tolerance. The number of instances of each cluster is automatically calculated first by identifying the correct bottleneck and then applying other constraints, such as number of connections to support. Size of production cache clusters ranges from 20 to thousands of instances.

[1] Although each cluster is single-tenant, each tenant might cache multiple types of objects of different characteristics.

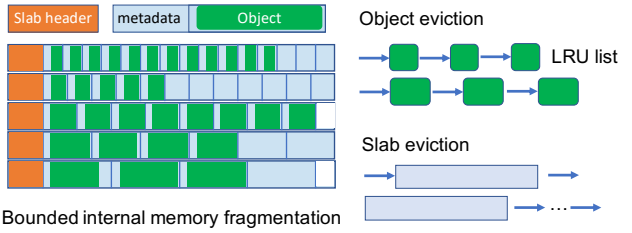


Figure 1: Slab-based memory management for bounded memory fragmentation. While Memcached uses object eviction, Twemcache uses slab eviction, which evicts all objects in one slab and returns the slab to global pool.

2.3 Overview of Twemcache

Twemcache forked an earlier version of Memcached with some customized features. In this section, we briefly describe some of the key aspects of its designs.

Slab-based memory management Twemcache often stores small and variable-sized objects in the range of a few bytes to 10s of KB. On-demand heap memory allocators such as ptmalloc [45], jemalloc [13] can cause large and unbounded external memory fragmentation in such a scenario, which is highly undesirable in production environment, especially when using smaller containers. To avoid this, Twemcache inherits the slab-based memory management from Memcached (Figure 1). Memory is allocated as fixed size chunks called *slabs*, which default to 1 MB. Each slab is then evenly divided into smaller chunks called *items*. The *class* of each slab decides the size of its items. By default, Twemcache grows item size from a configurable minimum (default to 88 bytes) to just under a whole slab. The growth is typically exponential, controlled by a floating point number called growth factor (default to 1.25), though Twemcache also allows precise configuration of specific item sizes. Higher slab classes correspond to larger items. An object is mapped to the slab class that best fits it, including metadata. In Twemcache, this per-object metadata is 49 bytes. By default, a slab of class 12 has 891 items of 1176 bytes each, and each item stores up to 1127 bytes of key plus value. Slab-based allocator eliminates external memory fragmentation at the cost of bounded internal memory fragmentation.

Eviction in slab-based cache To store a new object, Twemcache first computes the slab class by object size. If there is a slab with at least one free item in this slab class, Twemcache uses the free item. Otherwise, Twemcache tries to allocate a new slab into this class. When memory is full,

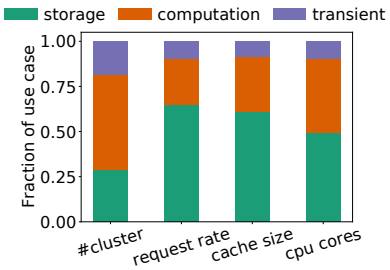


Figure 2: Resources consumed for the three cache use cases.

slab eviction is needed for allocation.

Some caching systems such as Memcached primarily performs item-level eviction, which happens in the same slab class as the new object. Memcached uses an approximate LRU queue per slab class to track and evict the least recently used item. This works well as long as object size distribution remains static. However, this is often not true in reality. For example, if all keys start with small values that grow over time, new writes will eventually require objects to be stored in a higher slab class. However, if all memory has been allocated when this happens, there will be effectively no memory to give out. This problem is called *slab calcification* and is further explored in Section 4.6.2. Memcached developed a series of heuristics to move memory between slab classes, and yet they have been shown as non-optimal [10, 11, 17, 46] and error prone [9].

To avoid slab calcification, Twemcache uses slab eviction only (Figure 1). This allows the evicted slab to transition into any other slab class. There are three approaches to choose the slab to evict: choosing a slab randomly (random slab), choosing the least recently used slab (slabLRU), and choosing the least recently created slab (slabLRC). In addition to avoiding slab calcification, slab-only eviction removes two pointers from object metadata compared to Memcached. We further compare object eviction and slab eviction in Section 6.

2.4 Cache Use Cases

At Twitter, it is generally recognized that there are three main use cases of Twemcache: caching for storage, caching for computation, and caching for transient data. We remark that there is no strict boundary between the three categories, and production clusters are not explicitly labeled. Thus the percentages given below are rough estimates based on our understanding of each cache cluster and their corresponding application.

2.4.1 Caching for Storage

Using cache to facilitate reading from storage is the most common use case. Backend storage such as databases usually has a longer latency and a lower bandwidth than in-memory cache. Therefore, caching these objects reduce access latency, increases throughput, and shelters the backend from excessive read traffic. This use case has received the most attention in research. Several efforts have been devoted to reducing miss ratio [26–28, 36, 37, 41, 47, 72] , redesigning for a denser storage device to fit larger working sets [19, 42, 65], improving load balancing [33, 34, 39] and increasing throughput [43, 56].

As shown in Figure 2, although only 30% of the clusters fall into this category, they account for 65% of the requests served by Twemcache, 60% of the total DRAM used, and 50% of all CPU cores provisioned.

2.4.2 Caching for Computation

Caching for computation is not new — using DRAM to cache query results has been studied and used since more than two decades ago [20, 58]. As real-time stream processing and machine learning (ML) become increasingly popular, an increasing number of cache clusters are devoted to caching computation related data, such as features, intermediate and final results of ML prediction, and so-called object hydration, — populating objects with additional data, which often combines storage access and computation.

Overall, caching for computation accounts for 50% of all Twemcache clusters in cluster count, 26%, 31% and 40% of request rate, cache sizes and CPU cores.

2.4.3 Transient data with no backing store

The third typical cache usage evolves around objects that only live in cache, often for short periods of time. It is not caching in the strict sense, and therefore has received little attention. Nonetheless, in-memory caching is often the only production solution that meets both the performance and scalability requirements of such use cases. While data loss is still undesirable, these use cases really prize speed, and tolerate occasional data loss well enough to work without a fallback.

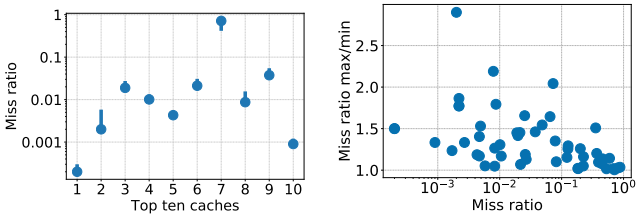
Some notable examples are rate limiters, deduplication caches, and negative result caches. Rate limiters are counters associated with user activities. They track and cap user requests in a given time window and prevent denial-of-service attacks. Deduplication caches are a special case of rate limiters, where the cap is 1. Negative result caches store keys from a larger database that are known to be misses against a smaller, sparsely populated database. These caches short-circuit most queries with negative results, and drastically reduce the traffic targeting the smaller database.

In our measurements, 20% of Twemcache clusters are under this category. Their request rates and cache sizes account for 9% and 8% of all Twemcache request rates and cache sizes, meanwhile, they account for 10% of all CPU cores of Twemcache clusters.

3 Methodology

3.1 Log Collection

Twemcache has a built-in non-blocking request logging utility called `klog` that can keep up with designed throughput in production. While it logs one out of every 100 requests by default, we dynamically changed the sampling ratio to 100% and collected week-long *unsampled* traces from two instances of each Twemcache cluster. Collecting unsampled



(a) Production miss ratio **(b)** Miss ratio variation

Figure 3: a) Production miss ratio of the top ten Twemcache clusters ranked by request rates, the bar shows the max and min miss ratio across one week. Note that the Y-axis is in log scale. b) The ratio between max and min miss ratio is small for most caches.

traces allows us to avoid drawing potentially biased conclusions caused by sampling. Moreover, we chose to collect traces from two instances instead of one to prevent possible cache failure during log collection and to compare results between instances for higher fidelity. Barring cache failures, the two instances have no overlapping keys.

3.2 Log Overview

We collected around 700 billion requests (80 TB in raw file size) from 306 instances of 153 Twemcache clusters, which include all clusters with per-instance request rate more than 1000 queries-per-sec (QPS) at the time of collection. To simplify our analysis and presentation, we focused on the 54 largest caches, which account for 90% of aggregated QPS and 76% of allocated memory. In the following sections, we use Twemcache workloads to refer to the workloads from these 54 Twemcache clusters. Although we only present the results of these 54 caches, we did perform the same analysis on the smaller caches, and they don't change our conclusions.

4 Production Stats and Workload Analysis

In this section, we start by describing some common production metrics to provide a foundation for our discussion, and then move on to workload analyses that can only be performed with detailed traces.

4.1 Miss Ratio

Miss ratio is one of the key metrics that indicate the effectiveness of a cache. Production in-memory caches usually operate at a low miss ratio with small miss ratio variation.

We present the miss ratios of the top ten Twemcache clusters ranked by request rates in Figure 3a where the dot shows the mean miss ratio over a week, and the error bars show the minimum and maximum miss ratio. Eight out of the ten Twemcache clusters have a miss ratio lower than 5%, and six of them have a miss ratio close to or lower than 1%. The only exception is a write-heavy cache cluster, which has a miss ratio of around 70% (see Section 4.3.2 for details about write-heavy workloads). Compared to CDN caching [47], in-memory caching usually has a lower miss ratio.

Besides a low miss ratio, miss ratio stability is also very important. In production, it is the highest miss ratio (and

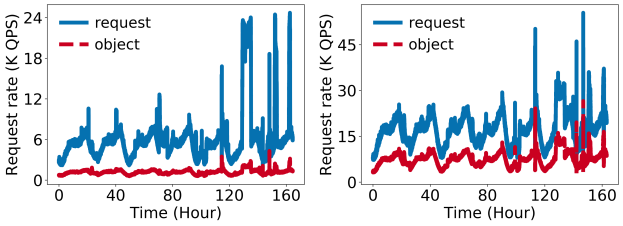


Figure 4: The number of requests and objects being accessed every second for two cache nodes.

request rate) that decides the QPS requirement of the backend. Therefore, a cache with a low miss ratio most of the time, but sometimes a high miss ratio is less useful than a cache with a slightly higher but stable miss ratio. Figure 3b shows the ratios of $\frac{mr_{max}}{mr_{min}}$ over the course of a week for different caches, where mr stands for miss ratio. We observe that most caches have this ratio lower than 1.5. In addition, the caches that have larger ratios usually have a very low miss ratio.

Low miss ratios and high stability in general illustrate the effectiveness of production caches. However, extremely low miss ratios tend to be less robust, which means the corresponding backends have to be provisioned with more margins. Moreover, cache maintenance and failures become a major source of disruption for caches with extremely low miss ratios. The combination of these factors indicate there's typically a limit to how much cache can reduce read traffic or how little traffic backends need to provision for.

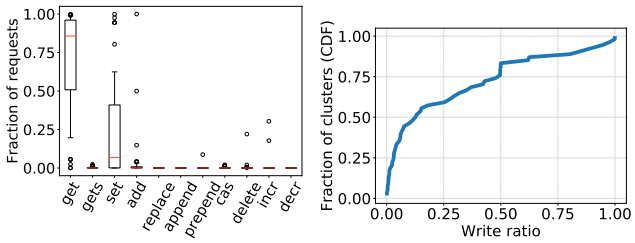
4.2 Request Rate and Hot Keys

Similar to previously observed [24], request rates show diurnal patterns (Figure 4). Besides, spikes in request rate are also very common because cache is the first responder to any change from the frontend services and end users.

When a request rate spike happens, a common belief is that hot keys cause the spikes [33, 48]. Indeed, load spikes often are the results of hot keys. However, we notice it is not always true. As shown in Figure 4, at times, when the request rate (top blue curve) spikes, the number of objects accessed in the same time interval (bottom red curve) also has a spike, indicating that the spikes are triggered by factors other than hot keys. Such factors include client retry requests, external traffic surges, scan-like accesses, and periodic tasks.

In addition to request rate spikes, caches often show other irregularities. For example, in Section 4.6.2, we show that it is common to see sudden changes in object size distribution. These irregularities can happen for various reasons. For instance, users change their behavior due to a social event, the frontend service adds a new feature (or bug), or an internal load test is started.

As a critical component in the infrastructure, caches stop most of the requests from hitting the backend, and they should be designed to tolerate these workload changes to absorb the impact.



(a) Relative use of each operation **(b)** Write ratio

Figure 5: a) Ratio of operation in each Twemcache cluster, box shows the 25^{th} and 75^{th} percentile, red bar inside the box shows the mean ratio, and whiskers are 10^{th} and 90^{th} percentile. b) write ratio distribution CDF across Twemcache clusters.

4.3 Types of Operations

Twemcache supports eleven different operations, of which `get` and `set` are the most heavily used by far. In addition, write-heavy cache workloads are very common at Twitter.

4.3.1 Relative usage comparison

We begin from the operations used by Twemcache workloads. Twemcache supports eleven operations `get`, `gets`, `set`, `add`, `cas` (check-and-set), `replace`, `append`, `prepend`, `delete`, `incr` and `decr`[2]. As shown in Figure 5a, `get` and `set` are the two most common operations, and average `get` ratio is close to 90% indicating most of the caches are serving read-heavy workloads. Apart from `get` and `set`, operations `gets`, `add`, `cas`, `delete`, `incr` are also frequently used in Twemcache clusters. However, compared to `get` and `set`, these operations usually account for a smaller percentage of all requests. Nonetheless, these operations serve important roles in in-memory caching. Therefore, as suggested by the author of Memcached, they should not be ignored [15].

4.3.2 Write ratio

Although most caches are read dominant, Figure 5a shows that both `get` and `set` ratios have a large range across caches. We define a workload as write-heavy if the percentage sum of `set`, `add`, `cas`, `replace`, `append`, `prepend`, `incr` and `decr` operations exceeds 30%. Figure 5b shows the distribution of write ratio across caches. More than 35% of all Twemcache clusters are write-heavy, and more than 20% have a write ratio higher than 50%. In other words, in addition to the well-known use case of serving read-heavy workloads, a substantial number of Twemcache clusters are used to serve write-heavy workloads. We identify the main use cases of write-heavy caches below.

Frequently updated data Caches under this category mostly belong to cache for computation or transient data (Section 2.4.2 & 2.4.3). Updates are accumulated in cache before they get persisted, or the keys eventually expire.

[2]See https://github.com/memcached/memcached/wiki/Commands for details about each command.

Opportunistic pre-computation Some services continuously generate data for potential consumption by itself or other services. One example is the caches storing recent user activities, and the cached data are read when a query asks for recent events from a particular user. Many services choose not to fetch relevant data on demand, but instead opportunistically pre-compute them for a much larger set of users. This is feasible because pre-computation often has a bounded cost, and in exchange read queries can be quickly fulfilled by pre-computed results partially or completely. Since this is a trade-off mainly for user experience, the caches under this category see objects with fewer reuse. Therefore, the write ratio is often higher (>80%), and object access (read+write) frequency is often lower. In one case, we saw one cluster with a mean object frequency close to 1.

4.4 TTL

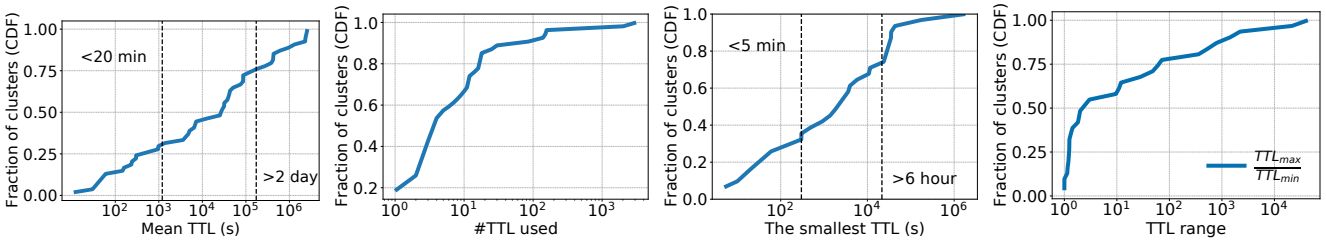
Two important features that distinguish in-memory caching from a persistent key-value store are TTL and cache eviction. While evictions have been widely studied [26, 28], TTL is often overlooked. Nonetheless, TTL has been routinely used in production. Moreover, as a response to GDPR [5], the usage of caching TTL has become mandatory at Twitter to enforce data retention policies. TTL is set when an object is first created in Twemcache, and decides its expiration time. Request attempts to access an expired object will be treated as misses, so keeping expired objects in the cache is not useful.

We observe that in-memory caching workloads often use short TTLs. This usage comes from the dynamic nature of cached objects and the usage for implicit deletion. Under this condition, effectively and efficiently removing expired objects from the cache becomes necessary and important, which provides an alternative to eviction in achieving low miss ratios.

4.4.1 TTL Usages

We measure the mean TTLs used in each Twemcache cluster and show the TTL distribution in Figure 6a. The figure shows that TTL ranges from minutes to days. More than 25% of the workloads use a mean TTL shorter than twenty minutes, and less than 25% of the workloads have a mean TTL longer than two days. Such a TTL range is longer than DNS caching (minutes) [51], but shorter than common CDN object caching (days to weeks). If we divide caches into *short-TTL caches* (TTL $\leq$ 12 hours) and *long-TTL caches* (TTL > 12 hours). Figure 6a shows 66% of all Twemcache clusters have a short mean TTL.

In addition to mean TTL distribution, we have also measured the number of TTL used in each cache. Figure 6b shows that only 20% of the Twemcache workloads use a single TTL, while the rest majority use more than one TTL. In addition, we observe that over 30% of the workloads use more than ten TTLs and there are a few workloads using more than 1000 TTLs. In the last case, some clients intentionally scatter TTLs over a pre-defined time range to avoid objects expiring at the



(a) Mean TTL distribution **(b)** Number of TTL in each cache **(c)** The smallest TTL distribution **(d)** TTL range distribution

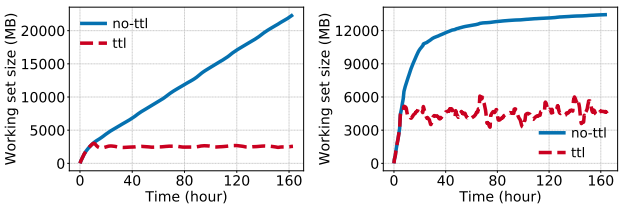
Figure 6: a) More than half of caches have mean TTL shorter than one day. b) Only 20% of caches use single TTL. c) The smallest TTL in each cache can be very long. d) TTLs ranges in workloads are often large.

same time. This technique is called *TTL jitter*. In another case, the clients seek the opposite effect — computing TTLs so that a group of objects will expire at the same, predetermined time.

Besides the number of TTLs used, the smallest TTL and the TTL range, defined as the ratio between TTL_{max} and TTL_{min}, are also important for designing algorithms that remove expired objects (see Section 7). Figure 6c shows that the smallest TTL in each cache varies from 10s of seconds to more than half day. In detail, around 30 to 35% of the caches have their smallest TTL shorter than 300 seconds, and over 25% of caches have the smallest TTL longer than 6 hours. Figure 6d shows the CDF of each workload's TTL range. We observe that fewer than 40% of the workloads have a relatively small TTL range ($< 2\times$ difference), while almost 25% of the caches have $\frac{TTL_{max}}{TTL_{min}}$ over 100.

Below we present the three main purposes of TTL to better explain how TTL settings relate to the usages of the caches.

Bounding inconsistency Objects stored in Twemcache can be highly dynamic. Because cache updates are best-effort, and failed cache writes are not always retried, it is possible that objects stored in in-memory cache are stale. Therefore, applications often use TTL to bound inconsistency, which is also suggested in the AWS Redis documentation [7]. TTLs for this purpose usually have relative large values, in the range of days. Some Twitter services further developed *soft TTL* to achieve a better tradeoff between data consistency and avaiilability. The main idea of soft TTL is to store an additional, often shorter TTL as part of the object value. When application decodes the value of a cached object and notices that the soft TTL has expired, it will refresh the cached value from its corresponding source of truth in the background. Meanwhile, the application continues to use the older value to fulfill current requests without waiting. Soft TTL is typically designed to increase with each background refresh, based on the assumption that newly created objects are more likely to see high volume of updates and therefore inconsistency.



(a) **(b)**

Figure 7: The working set size grows over time when TTL is not considered. However, when TTL is considered, the working set size is capped.

Implicit deletion In some caches, TTL reflects the intrinsic life span of stored objects. One example is the counters used for API rate limiting, which are declared as maximum number of requests allowed in a time window. These counters are typically stored in cache only, and their TTLs match the time windows declared in the API specification. In addition to rate limiters, GDPR required TTL would also fall into this category, so no data would live in cache beyond the duration permitted under the law.

Periodic refresh TTL is also used to promote data freshness. For example, a service that calculates how much a user's interest matches a cluster/community using ML models can make "who-to-follow" type of recommendations with the results. The results are cached for a while because user characteristics tend to be stable in the very short term, and the calculation is relatively expensive. Nonetheless, as users engage with the site, their portraits can change over time. Therefore such a service tends to recompute the results for each user periodically, using or adding the latest data since last update. In this case, TTL is used to pace a relatively expensive operation that should only be performed infrequently. The exact value of the TTL is the result of a balance between computational resources and data freshness, and can often be dynamically updated based on circumstances.

4.4.2 Working Set Size and TTL

Having the majority of caches use short TTLs indicate that the *effective working set size* (WSS_E) — the size of all unexpired objects should be loosely bounded. In contrast, the *total working set size* (WSS_T), the size of all active objects regardless of TTL, can be unbounded.

In our measurements, we identify two types of workloads

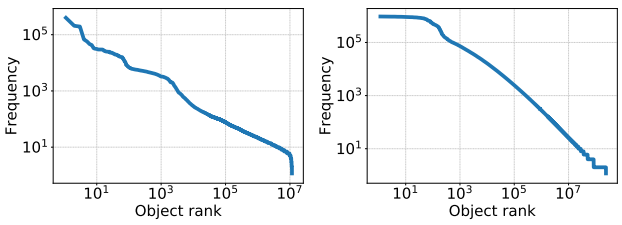
shown in Figure 7. The first type (Figure 7a) has a continuously growing WSS_T, and it is usually related to user-generated content. With new content being generated every second, the total working set size keeps growing. The second type of workload has a large growth rate in WSS_T at first, and then the growth rate decreases after this initial fast-growing period, as shown in Figure 7b. This type of workloads can be users related, the first quick increase corresponds to the most active users, the slow down corresponds to less active users. Although the two workloads show different growth patterns in total working set size, the effective working set size of both arrive at a plateau after reaching its TTL. Although the WSS_E may fluctuate and grow in the long term, the growth rate is much slower compared to WSS_T.

Bounded WSS_E means that, for many caches, there exists a cache size that the cache can achieve compulsory miss ratio, if an in-memory caching system can remove expired objects in time. This suggest the importance of quickly removing expired object from cache, especially for workloads using short TTLs. Unfortunately, while eviction has been widely studied [26, 28, 54], expiration has received little attention. And we will show in Section 7.2, existing solutions fall short on expiration.

4.5 Popularity Distribution

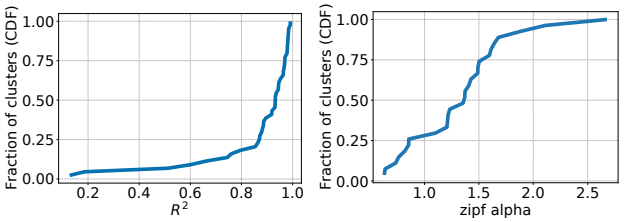
Object popularity is another important characteristic of a caching workload. Popularity distribution is often used to describe the cachebility of a workload. A popular assumption is that cache workloads follow Zipfian distribution [29], and the frequency-rank curve plotted in log-log scale is linear. A large body of work optimizes system performance under this assumption [33, 39, 44, 50, 57, 61]. However, a recent work from Facebook [19] suggested that in-memory caching workloads may not follow Zipfian distribution. Here we present the popularity of the caching workloads at Twitter.

Measuring all Twemcache workloads, we observe majority of the cache workloads still follow Zipfian distribution. However, some workloads show deviations in two ways. First, unpopular objects appear significantly less than expected (Figure 8a) or the most popular objects are less popular than expected (Figure 8b). The first deviation happens when objects



(a) **(b)**

Figure 8: Some workloads showing small deviations from Zipfian popularity. a) The least popular objects are less popular than expected. b) The most popular objects are less popular than expected.



(a) R^2 in Zipfian fitting **(b)** Zipfian parameter α

Figure 9: a) Most of workloads follow Zipfian popularity distribution with large confidence R^2. b) The parameter α in Zipfian distribution is large, and the popularity of most workloads are highly skewed ($\alpha > 1$).

are always accessed multiple times so that there are few objects with frequency smaller than some value. The second deviation happens when the client has an aggressive client-side caching strategy so that the most popular objects are often cached at client. In this case, the cache is no longer single-layer.

Although these deviations happen, they are rare, and we believe it is still reasonable to assume in-memory caching workloads follow Zipfian distribution. Since most part of the frequency-rank curves are linear in the log-log scale, we use linear fitting[3] confidence R^2 [12] as the metric for measuring the goodness of fit. Figure 9a shows the results of fitting. 80% of all workloads have R^2 larger than 0.8, and more than 50% of workloads have R^2 larger than 0.9. These results indicate that the popularity of most in-memory caching workloads at Twitter follows Zipfian distribution. We further measure the parameter α of the Zipfian distribution shown in Figure 9b. The figure shows that most of the α values are in the range from 1 to 2.5, indicating the workloads are highly skewed.

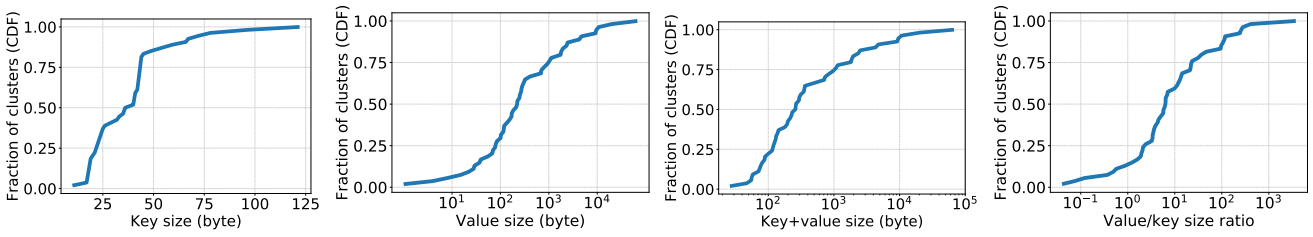
4.6 Object Size

One feature that distinguishes in-memory caching from other types of caching is the object size distribution. We observe that similar to previous observations [24], the majority of objects stored in Twemcache are small. In addition, size distribution is not static over time, and both periodic distribution shifts and sudden changes are observed in multiple workloads.

4.6.1 Size Distribution

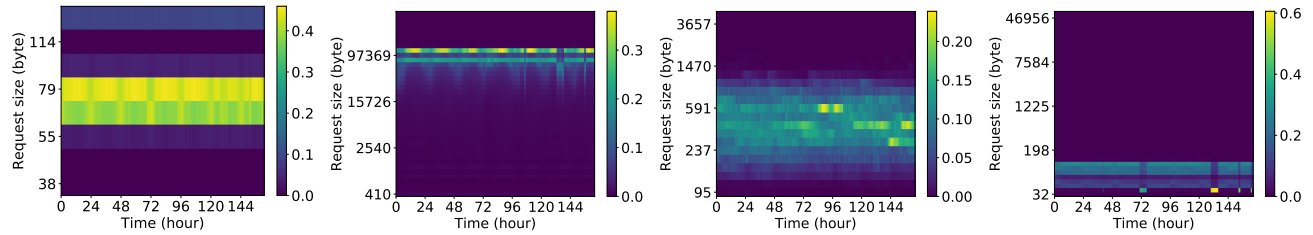
We measure the mean key size and value size in each Twemcache cluster, and present the CDF of the distributions in Figure 10. Figure 10a shows that around 85% of Twemcache clusters have a mean key size smaller than 50 bytes, with a median smaller than 38 bytes. Figure 10b shows that the mean value size falls in the range from 10 bytes to 10 KB, and 25% of workloads show value size smaller than 100 bytes, and median is around 230 bytes. Figure 10c shows that CDF

[3] We remark that linear regression is not the correct way to modelling Zipf distribution from the view of statistics, we perform this to align with existing works [29].



(a) Key size **(b)** Value size **(c)** Object size **(d)** Value/key size ratio

Figure 10: Mean key, value, object size distribution and mean $\frac{value}{key}$ size ratio across all caches.



(a) **(b)** **(c)** **(d)**

Figure 11: Heatmap showing request size distribution over time for four typical caches. X-axis is time, Y-axis is the object size using slab class size as bins, and the color shows the fraction of requests that fall into a slab class in that time window.

distribution of the mean object size (key+value), which is very close to the value size distribution except at small sizes. Value size distribution starts at size 1, while object size distribution starts from size 16. This indicates that for some of the caches, value size is dramatically smaller than the key size. Figure 10d shows the ratio of mean value and key sizes. We observe that 15% of workloads have the mean value size smaller than or equal to the mean key size, and 50% of workloads have value size smaller than 5× key size.

4.6.2 Size Distribution Over Time

In the previous section, we investigated the static size distribution of all objects accessed in the one week's time of each Twemcache cluster. However, the object size distribution of workloads are usually not static over time. In Figure 11, we show how the size distribution changes over time. The X-axis shows the time, and the Y-axis shows the size of objects (using slab class size as bins), the color shows how much of the objects in one time window fall into each slab class. We observe that some of the workloads show diurnal patterns (Figure 11a, 11b), while others show changes without strict patterns.

Periodic/diurnal object size shifts can come from the following sources, a) value for the same key grows over time. and b) size distribution correlates with temporal aspects of key access. For example, text content generated by users in Japan are shorter/smaller than those by users in Germany. In this case, it is the geographical locality that drives the temporal pattern. On the other hand, we do not yet have a good understanding of how most sudden, non-recurring changes happen. Current guesses include user behavior changes during events, and a temporary change in production settings.

Both short-term and long-term size distribution shifts pose additional challenges to memory management in caching systems. They make it hard to control or predict external fragmentation in caches that use heap memory allocators directly, such as Redis. For slab-based caching systems, they can cause slab calcification. In Section 7.5, we discuss why existing techniques do not completely address the problem.

Table 1: Correlation between write ratio and other properties

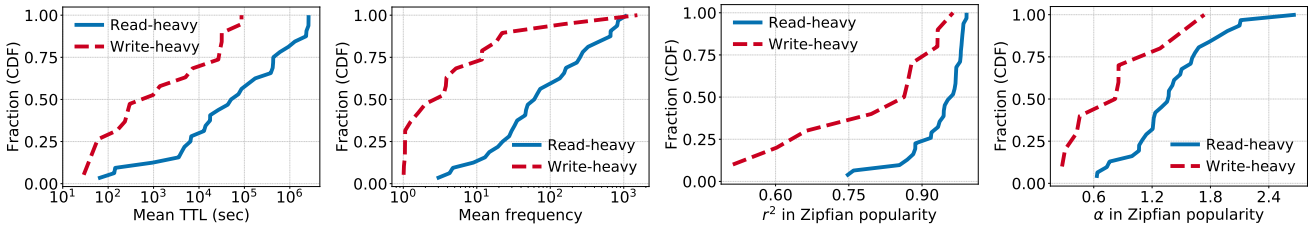
Property	Pearson coefficient with write ratio
log(TTL)	-0.6336
log(Frequency)	-0.7414
Zipf fitting R^2	-0.7690
Zipf alpha	-0.7329

5 Further Analysis of Workload Properties

We have shown the properties of the in-memory caching workloads at Twitter. In this section, we show the relationship between the properties, and how they relate to major caching use cases.

5.1 Correlations between Properties

Throughout the analysis in previous sections, we observe some workload characteristics have strong correlations with the write ratio. For example, write-heavy workloads usually use short TTLs. Presented in Figure 12a, the dashed red curve shows the mean TTL distribution of write-heavy workloads, and the solid blue curve shows the mean TTL distribution



(a) TTL distribution **(b)** Object frequency distribution **(c)** R^2 in fitting Zipfian **(d)** α of Zipfian distribution

Figure 12: Write-heavy workloads tend to show short TTL, small object access frequency, relatively large deviations from Zipfian popularity distribution and are usually less skewed (small α).

of read-heavy workloads. Around 50% of the write-heavy workloads have mean TTL shorter than 10 minutes, while for read-heavy workloads, this is 15 hours. Further, the Pearson coefficient between write ratio and log [4] of mean TTL (Table. 1) is -0.63 indicating a negative correlation, confirming that large write ratio workloads usually have short TTLs.

Besides TTL, write-heavy workloads also show low object frequencies. We present the mean object frequency (in terms of the number of accesses in the traces) of read-heavy and write-heavy workloads in Figure 12b. It shows that read-heavy workloads have a mean frequency mostly in the range from 6 to 1000, with 75% percentile above 200. Meanwhile, write-heavy workloads have a mean frequency mostly between 1 and 100, with 75% percentile below 10. We further confirm this relationship with the Pearson coefficient between write ratio and log of frequency, which is -0.7414 (Table. 1), suggesting the low object access frequency in write-heavy caches.

In addition, the popularity of write-heavy workloads has relatively larger deviations from Zipfian distribution, and the fitting confidence R^2 is usually much smaller than that of read-heavy workloads (Figure 12c). Moreover, the α parameter of Zipfian distribution in write-heavy workloads is usually small, as shown in Figure 12d. It shows the write-heavy workloads have a median α around 0.9, and the median of read-heavy workloads have an α around 1.4. This correlation is also backed up by the Pearson coefficient (Table 1).

5.2 Properties of Different Cache Use Cases

Here we further explore common properties exhibited by each of the three major caching use cases as described in Section 2.4.

5.2.1 Caching for Storage

Caches for storage usually serve ready-heavy workloads, and their popularity distributions typically follow Zipfian distribution with a large parameter α in the range of 1.2 to 2.2. While this type of workload is highly skewed, they are easier to cache, and in production, 95% of these clusters have miss ratios of around or less than 1%. Being more cacheable and having smaller miss ratios do not indicate they have small working set sizes. In our observation, 7 of the top 10 caches (ranked by cache size) belong to this category.

Because these caches store objects persisted in the backend storage, any modifications to the objects are explicitly written to both the backend and the cache. Therefore the TTLs used in these caches are usually large, in the range of days. There is no specific pattern about object size in this type of caches, and the value can be as large as tens of KB, or as small as a few bytes. For example, the number of favorites a tweet received is persisted in the backend database and sometimes cached.

5.2.2 Caching for Computation

Caches under this category serve both read-heavy and write-heavy traffic depending on the workloads. For example, machine learning feature workloads are usually read-heavy showing a good fit of Zipfian popularity distribution. While intermediate computation workloads are normally write-heavy and show deviations from Zipfian. Compared to caching for storage, workloads under this category use shorter TTLs, usually determined by the application requirement. For example, caches storing intermediate computation data usually have TTLs no more than minutes because other services will consume the data in a short time. For features and prediction results, the TTLs are usually in the range of minutes to hours (some up to days) depending on how fast the underlying data change and how expensive the computation is. The mean TTLs we observe for caches under this category is 9.6 hours. There are no particular patterns about object sizes in these caches.

Since objects stored in these caches are indirectly related to users and contents, the workloads usually have large key spaces and total working set sizes. For example, a cache storing the distance between two users will require a N^2 cache size where N denotes the number of users. However, because these caches have short TTLs, the effective working set sizes are usually much smaller. Thus removing expired objects can be more important than eviction for these caches.

As real-time stream processing becomes more popular, we envision there will be more caches being provisioned for caching computation results. Because the characteristics are different from caching for storage, they may not benefit equally from optimizations that only aim to make the

[4] We choose to use log of TTL and frequency because of their wide ranges in different workloads.

read path fast and scalable, such as optimistic cuckoo hashing [43]. Therefore, including evaluation against caching-for-computation workloads that are write-heavy and more ephemeral will paint a more complete picture of the capabilities of any caching system.

5.2.3 Transient Data with No Backing Store

There are two characteristics associated with this type of caches: Caches under this category usually have short TTLs, and the TTLs are often used to enforce implicit object deletion (Section 4.4). In addition, objects in these caches are usually tiny and we observe an average object size of 54 bytes. Although caches of this type only contribute 9% of total Twemcache cluster request rate and 8% of total cache sizes, they currently play an irreplaceable role in site operations.

6 Eviction Algorithms

We have shown the characteristics of in-memory cache workloads in the previous sections. In this section, we use the same cache traces to investigate the impact of eviction algorithms. This evaluation considers production algorithms offered by Twemcache and other production systems.

6.1 Eviction algorithm candidates

Object LRU and object FIFO LRU and FIFO are the most common algorithms used in production caching systems [4, 18]. However, they cannot be applied to systems using slab-based memory management such as Twemcache without modification. Therefore, we evaluate LRU and FIFO assuming the workloads are served using a non-slab based caching system, while ignoring memory inefficiency caused by external fragmentation. As a result, we expect that the results to have a bias toward the effectiveness of LRU and FIFO compared to the three slab-based algorithms. Production results for these two algorithms might be worse than what is suggested in this section, depending on the workloads.

slabLRU and slabLRC These two algorithms are part of eviction algorithms offered in Twemcache. slabLRU and slabLRC are equivalent to LRU and FIFO but executed at a level much coarser granularity of slabs rather than a single object. Twitter employs these algorithms to alleviate the effect of slab calcification and also to reduce the size of per-object metadata.

Random slab eviction Besides slabLRU and slabLRC, Twemcache also offers Random slab eviction, which globally picks a random slab to evict. This algorithm is workload-agnostic with robust behavior, and therefore used as the default policy in production. However, it is rarely the best of all algorithms and are non-deterministic, therefore we do not include it in comparison.

Memcached-LRU Memcached adapted LRU by creating one LRU queue per slab class. We call the resulted eviction algorithm Memcached-LRU, which does not enable Memcached's slab auto-move functionality. We did, however, evaluate Memcached-LRU with slab auto-move turned on, and most of the results are somewhere between LRU and slabLRU. The rest of the paper omits this combination.

6.2 Simulation Setup

We built an open-source simulator called libCacheSim [71] to study the steady-state miss ratio of the different eviction algorithms. Specifically, we use five-day traces to warm up the caches, then use one-day traces to evaluate cache miss ratios. Each algorithm is applied against all traces, and then grouped by results.

In terms of cache sizes, our simulation always starts with 64MB of DRAM, and chooses the maximum as 2× their current memory in production. We stop increasing the size for a particular workload when all algorithms have reached the compulsory miss ratio. Note that when plotting, the size range is truncated to better present the trend.

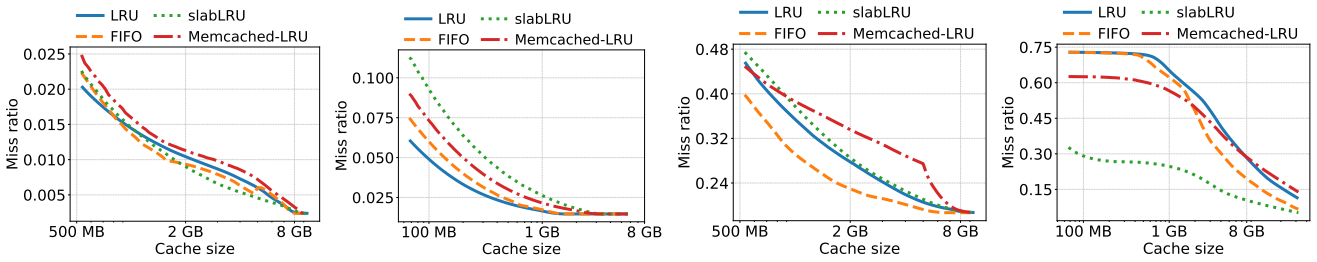
6.3 Miss Ratio Comparison

The outcome of our comparison can be grouped into four types, and representatives of each are shown in Figure 13.

The first group shows comparable miss ratios for all algorithms in the cache sizes we evaluated. For this type of workload, the choice of eviction algorithms has a limited impact on the miss ratio. Production deployments may very well favor simplicity or decide based on other operational considerations such as memory fragmentation. Twemcache uses random slab eviction by default because random eviction is simple and requires less metadata.

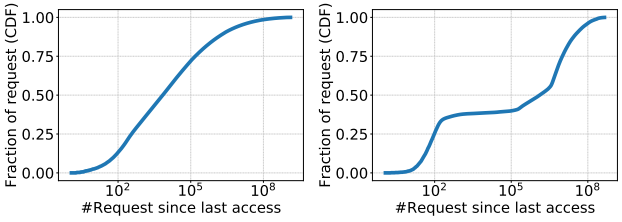
The second type of result shows that for some workloads LRU works better than others. Such a result is often expected because LRU protects recently accessed objects and is well-known for its miss ratio performance in workloads with strong temporal locality.

The third type of result shows that FIFO is the best eviction algorithm (Figure 13c). This result is somewhat surprising since it does not conform to what is typically observed in caching of other scenarios such as CDN caching. We give our suspected reasons below. Figure 14 shows the inter-arrival time distribution of the two workloads in Figure 13b and Figure 13c respectively. The inter-arrival time is the number of requests between two accesses to the same object. Figure 14a shows a smooth inter-arrival time curve, while Figure 14b shows a curve with multiple segments. For workloads with inter-arrival time like Figure 14a, LRU can work better than FIFO because it promotes recently accessed objects, which have a higher chance of being reused soon. This promotion protects the recently accessed objects but demotes other objects that are not reused recently. Demoting non-recently used objects can be an unwise decision if some of the demoted objects will be reused after 10^6 requests, such as the ones shown in Figure 14b. In contrast, FIFO treats each stored object equally; in other words, it protects the objects with a large inter-arrival gap. Therefore, for workloads similar to the one in Figure 14b, FIFO can perform better than LRU. Such



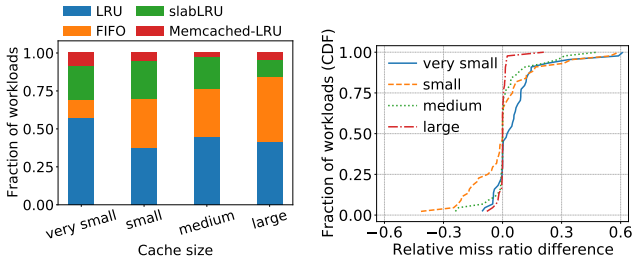
(a) Similar miss ratio **(b)** LRU is better **(c)** FIFO is better **(d)** slabLRU is better

Figure 13: Four typical miss ratio results: a) all algorithms have similar performance, b) LRU is slightly better than others, c) FIFO is better than others, d) slabLRU is much better than others.



(a) **(b)**

Figure 14: The inter-arrival gap distribution corresponding to the workloads in Figure 13b and Figure 13c respectively.



(a) **(b)**

Figure 15: a) The best eviction algorithms under different sizes. b) The relative miss ratio difference between FIFO and LRU under different sizes. Positive region shows FIFO is worse.

workloads may include scan type of requests such as a service that periodically sends emails.

The last type of result show that in some workloads, slabLRU performs much better than any other algorithms. The main reason is that the workloads showing this type of result have periodic/diurnal changes. Figure 11b shows the object size distribution over time of the workload corresponding to Figure 13d. We suspect this is due to the following reason, but we leave the verification as future work. Although LRU and FIFO are not affected by any change in object size distribution, they cannot respond to workload change instantly. In contrast, slabLRU can quickly adapt to a new workload when the new workload uses a different slab class because it prioritizes the slabs that have more recent access. From another view, slabLRU gives a larger usable cache size for the new workloads (slab class). Figure 13d shows that the difference between algorithms reduces at larger cache sizes, this is because the benefit of having a large usable cache size diminishes as cache size increases. Moreover, in these workloads, Memcached-LRU sometimes has better performance than LRU, but for most of the workloads, Memcached-LRU is worse (not shown in the figure) because of the missing capability of moving slabs. Thus it has a smaller usable cache size. When Memcached-LRU has better performance at small cache sizes, we suspect that the changing workloads cause thrashing for LRU and FIFO [27]. Since Memcached-LRU can only evict objects within from the same slab class as the new object, it protects the objects in other slab classes from thrashing, thus showing better performance.

In most cases, both miss ratio and the difference between algorithms decrease as cache capacity increases. We observe that within our simulation configuration, which stops at or before 2× current size, the difference between algorithms eventually disappears. This suggests that to achieve low miss ratio in real life, it can be quite effective to create implementations that increase the effective cache capacity, such as through metadata reduction, adopting higher capacity media, or data compression.

Given there are more than a couple of workloads showing each of the four result types, we would like to explore whether there is one algorithm that is often the best or close to the best most of the time.

In the next section, we explore how often each algorithm is the best with a special focus on LRU and FIFO.

6.4 Aggregated Statistics

In this section, we evaluate the same set of algorithms as in Section 6.3, focusing on four distinct cache sizes and present the aggregated statistics. Because different workloads have different working set sizes and compulsory miss ratios, we choose the four cache sizes in the following way. We define the *ultimate cache size* s_u to be the size where LRU achieves compulsory miss ratio for a workload. However, if LRU can not achieve compulsory miss ratio at 2× production cache size, we use 2× production cache size as s_u. We choose *large* cache size to be 90% of s_u, and *medium*, *small* and *very small*

cache sizes to be 60%, 20% and 5% of s_u respectively. We remark that, at Twitter, 76% of the caches have cache sizes larger than the *large* cache size category, and 34% of the rest have cache sizes within 10% of the *large* cache size.

We show the miss ratio comparison in Figure 15a, where each bar shows the fraction of workloads for which a particular algorithm is the best. We see that at the *large* cache size slabLRU is the best for around 10% of workloads, and this fraction gradually increases as we reduce cache size. This increase is because for smaller cache sizes, quickly adapting to workload change is more valuable. Besides this, FIFO has similar performance compared to LRU at *small, medium and large* size categories. And only at *very small* cache sizes, LRU becomes significantly better than FIFO. This is because at relatively large cache sizes, promoting recently accessed objects is less crucial. Instead, not demoting other objects is more helpful in improving the miss ratio, especially for workloads having multiple segments in inter-arrival time like the one shown in Figure 14b.

Figure 15a suggests that for close to half of the workloads, FIFO is as good as LRU at reasonably large cache sizes. Now we explore the magnitude by which FIFO is better or worse compared to LRU on each workload. Figure 15b shows the relative miss ratio difference between FIFO and LRU: $\left(\frac{mr_{\text{FIFO}} - mr_{\text{LRU}}}{mr_{\text{LRU}}}\right)$, where *mr* stands for miss ratio, for each workload at different cache sizes. When the value on X-axis is positive, it indicates that FIFO has a higher miss ratio, and LRU has better performance, while a negative value indicates the opposite. We observe that all the curves except the one for very small cache size are all close to being symmetric around x-axis value 0. This indicates that across workloads, FIFO and LRU have similar performance for small, medium and large cache sizes. For the very small size category, we observe LRU being significantly better than FIFO, this is because for workloads with temporal locality, promoting recently accessed objects becomes crucial at very small cache sizes. In production, most of the caches are running at cache sizes larger than or close to the *large* category. We believe that for most in-memory caching workloads, FIFO and LRU have a similar performance at reasonably large cache sizes.

The fact that FIFO and LRU often exhibit similar performance in production-like settings is important because using LRU usually incurs extra computational and memory overhead compared to FIFO [55, 56]. For example, implementing LRU in Memcached requires extra metadata and locks, some of which can be removed if FIFO is used.

7 Implications

In this section, we show how our observations differ from previous work, and what the takeaways are for informing future in-memory caching research.

7.1 Write-heavy Caches

Although 70% of the top twenty Twemcache clusters serve read-heavy workloads (Section 4.3.2), write-heavy workloads are also common for in-memory caching. This is not unique to Twitter. Previous work [24] from Facebook also pointed out the existence of write-heavy workloads, although the prevalence of them were not discussed due to the limited number of workloads. Furthermore, write-heavy workloads are expected to increase in prominence as the use case of caching for computation increases (Section 2.4.2). However, most of the existing systems, optimizations and research assume a read-heavy workload.

Write-heavy workloads in caching systems usually have lower throughput and higher latency, because the write path usually involves more work and can trigger more expensive events such as eviction. In Twitter's production, we observe that serving write-heavy workloads tend to have higher tail latencies. Scaling writes with many threads tends to be more challenging as well. In addition, as discussed in Section 5, write-heavy workloads have shorter TTLs with less skewed popularity, which are in sharp contrast to read-heavy workloads. This calls for future research on designing systems and solutions that consider performance on write-heavy workloads.

7.2 Short TTLs

In Section 4.4.1, we show that in-memory caching workloads frequently use short TTLs, and the usage of short TTLs reduces the effective working set size. Therefore, removing expired objects from the cache is far more important than evictions in some cases. In this section, we show that existing techniques for proactively removing expired objects (termed proactive expiration) are not sufficient. This calls for future work on better proactive expiration designs for in-memory caching systems.

Transient object cache An approach employed for proactive expiration (especially for handling short TTLs), proposed in the context of in-memory caches at Facebook [59], is to use a separate memory pool (called transient object pool) to store short-lived objects. The transient object cache consists of a circular buffer of size t with the element at index i being a linked list storing objects expiring after i seconds. Every second, all objects in the first linked list expire and are removed from the cache, then all other linked lists advance by one.

This approach is effective only when the cache user uses a mix of very short and long TTLs with the short TTL usually in the range of seconds. Since objects in the transient pool are never evicted before expiration, the size of transient pool can grow unbounded and cause objects in the normal pool to be evicted . In addition, the TTL threshold of admitting into transient object pool is non-trivial to optimize.

As we show in Figure 6b, 20% of the Twemcache workloads use a single TTL. For these workloads, transient object pool does not apply. For the workloads using multiple TTLs, we observe that fewer than 35% have their smallest TTL shorter than 300 seconds, and over 25% of caches have the smallest TTL longer than 6 hours (Figure 6c). This indicates

that the idea of transient object cache is not applicable to a large fraction of Twemcache clusters.

Background crawler Another approach for proactive expiration, which is employed in Memcached, is to use a background crawler that proactively removes expired objects by scanning all stored objects.

Using a background crawler is effective when TTLs used in the cache do not have a broad range. While scanning is effective, it is not efficient. If the cache scans all the objects every T_{pass}, an object of TTL t can be scanned up to $1+\lceil\frac{t}{T_{pass}}\rceil$ times before removal, and can overstay in the system by up to T_{pass}. The cache operator therefore has to make a tradeoff between wasted space and the additional CPU cycles and memory bandwidth needed for scanning. This tradeoff gets harder if a cache has a wide TTL range, which is common as observed in Section 4.4. While the Twemcache workloads are single tenant, wide TTL range issue would be further exacerbated for multi-tenant caches.

Figure 6d shows that TTLs used within each workload have a wide range. Close to 60% of workloads have the maximum TTL more than twice as long as the minimum, and 25% of workloads show a ratio at or above 100. This indicates that for the 25% of caches, if we want to ensure all objects are removed within 2× their TTLs, objects with the longest TTL will be scanned 100 times before expiration.

The combination of transient object cache with background crawler could extend the coverage of workloads that can be efficiently expired. However, the tradeoff between wasted space and the additional CPU cycles and memory bandwidth consumed for scanning would still remain. Hence, future innovation is necessary to fundamentally address use cases where TTLs exhibit a broad range.

7.3 Highly Skewed Object Popularity

Our work shows that the object popularity of in-memory caching can be far more skewed than previously shown [19], or compared to studies on web proxy workloads [29] and CDN workloads [47]. We suspect this has a lot to do with the nature of Twitter's product, which puts great emphasis on the timeliness of its content. It remains to be seen whether this is a widespread pattern or trend. Cache workloads are also more skewed compared to NoSQL database such as RocksDB [38], which is not surprising because database traffic is often already filtered by caches, and has the most skewed portion removed via cache hits. In other words, in-memory caching and NoSQL database often observe different traffic even for the same application. Besides these two reasons, sampling sometimes results in bias in the popularity modelling, and we avoid this by collecting unsampled traces. Our observation that the workloads still follow Zipfian distribution with large alpha value emphasizes the importance of addressing load imbalance [44, 57, 61].

7.4 Object Size

Similar to previously reported [24], we observe that objects cached in in-memory caching are often tiny (Section 4.6). As a result, in-memory caches are not always bound by memory size; instead, close to 20% of the Twemcache clusters are CPU-bound.

On the other hand, small objects signifies the relative large overhead of metadata. Memcached stores 56-byte with each object, and Twitter's current production cache uses 38-byte metadata with each object. Reducing object metadata further can yield substantial benefits for caching tiny objects.

In addition, we observe that compared to value size, the key size can be large in some workloads. For 60% of the workloads, the mean key size and mean value size are in the same order of magnitude. This indicates that reducing key size can be very important for these workloads. Many workloads we observed have namespaces as part of the object keys, such as `NS1:NS2:...:id`. This format is commonly used to mirror the naming in a multi-tenant database, which is also observed at Facebook [32]. Namespaces thus can occupy large fractions of precious cache space while being highly repetitive within a single cache cluster. However, there is no known techniques to "compress" the keys. To encourage and facilitate future research on this, we keep the original but anonymized namespace in our open sourced traces.

Several recent works [26, 28] on reducing miss ratio (improving memory efficiency) focused on improving eviction algorithms and often add more metadata. Given our observations here, we would like to call more attention to the optimization of cache metadata and object keys.

7.5 Dynamic Object Size Distribution

In Section 4.6.2, we show that the object size distribution is not static, and the distribution shifts over time can cause out-of-memory (OOM) exceptions for caching systems using external allocators, or slab calcification for those using slab-based memory management. In order to solve this problem, one solution, employed by Facebook, is to migrate slabs between slab classes by balancing the age of the oldest items in each class [59]. Earlier versions of Memcached approached this problem by balancing the eviction rate of each slab class. Since version 1.6.6, Memcached has also moved to using the solution of balancing the age as mentioned above.

Besides efforts in production systems, slab assignment and migration has also been a hot topic in recent research [31, 35, 36, 46]. However, to the best of our knowledge, the problem has only been studied under a "semi-static" request sequence. Specifically, the research so far assumes that the miss ratio curve or some other properties of each slab class hold steady for =certain amount of time, which often precludes periodic and sudden changes in object size distribution.

In general, the temporal properties of object sizes in cache are not well understood or quantified. As presented in Figure 11c and Figure 11d, it is not rare to see unexpected

changes in size distribution only lasting for a few hours. Sometimes it is hard to pinpoint the root cause of such changes. Nonetheless, we believe that temporal changes related to object size, whether recurring or as a one-off, usually have drivers with roots beyond the time dimension. For example, the tweet size drift throughout the day may very well depend on the locales or geo-location of active users. Some caches may be shared by datasets which differ in size distribution and access cycles, resulting in different distributions dominating the access pattern at different instants of the day. In this sense, studying the object size distribution over time could very well provide deeper insights into characteristics of the datasets being cached. Considering the increasing interest in using machine learning and other statistical tools to study and predict caching behavior, we think object size dynamics might provide a good proxy to evaluate the relationship between basic dataset attributes and their behavior in cache, allowing caching systems to make smarter decisions over time.

8 Related Work

Due to the nature of this work, we have discussed related works in detail throughout the paper.

Multiple caching and storage system traces were collected and analyzed in the past [24, 25, 32, 47, 48, 59]; however, only a limited number of reports focus on in-memory caching workloads [24,48,59]. The closet work to our analysis is Facebook's Memcached workload analysis [24], which examined five Memcached pools at Facebook. Similar to the observations in this work [24], we observe the sizes of objects stored in Twemcache are small, and diurnal patterns are common in multiple characteristics. After analyzing 153 Twemcache clusters at Twitter, in addition to previous observations [24], we show that write-heavy workloads are popular. Moreover, we focus on several aspects of in-memory caching which have not been studied to the best of our knowledge , including TTL and cache dynamics. Although previous work [24] proposed analytical models on the key size, value size, and inter-arrival gap distribution, the models do not fully capture all the dimensions of production caching workloads such as changing working set and dynamic object size distribution. Compared to synthetic workload models, the collection of real-world traces that we collected and open sourced provide a detailed picture of various aspects of the workloads of production in-memory caches. .

Besides workload analysis on Memcached, there have been several workload analysis on web proxy [21–23, 49, 64] and CDN caching [47, 67]. The photo caching and serving infrastructure at Facebook has been studied [47], with a focus on the effect of layering in caching along with the relationship between content popularity, age, and social-networking metrics.

In addition to caching in web proxies and CDNs, the effectiveness of caching is often discussed in workload studies [25,60,66] of file systems. However, these works primarily studied the cache to the extent that of its effectiveness in reducing traffic to the storage system rather than on aspects that affect the design of the cache itself. Besides, file system caching is different from distributed in-memory caches due to a variety of reasons. For example, file system caches usually stores objects of fixed-sized chunks (512 bytes, 4 KB or larger), while in-memory caches store objects of a much wider range (Section 4.6), and scan is common in file systems, while rare in in-memory caches.

Because of the similarities in the interface, in-memory caching is sometimes discussed together with key-value databases. Three different RocksDB workloads [32] at Facebook has been studied in depth, with a focus on the distribution of key and value sizes, locality, and diurnal patterns in different metrics. Although Twemcache and RocksDB have a similar key-value interface, they are fundamentally different because of their design and usage. RocksDB stores data for persistence, while Twemcache stores data to provide low latency and high throughput without persistence. In addition, compared to RocksDB, TTL and evictions are unique to in-memory caching.

9 Conclusion

We studied the workloads of 153 in-memory cache clusters at Twitter and discovered five important facts about in-memory caching. First, although read-heavy workloads account for more than half of the resource usages, write-heavy workloads are also common. Second, 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. Third, read-heavy in-memory caching workloads follow Zipfian popularity distribution with a large skew. Fourth, 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. Last, for a significant number of workloads, FIFO has similar or lower miss ratio performance as LRU for in-memory caching workloads. We have open sourced the traces collected at https://github.com/twitter/cache-trace.

Acknowledgements We thank our shepherd Andrea Arpaci-Dusseau and the anonymous reviewers for their valuable feedback. We thank our colleagues Jack Kosaian and Rebecca Isaacs for their extensive reviews and comments that improved this work. We also want to thank the Cache team and IOP team at Twitter for their support in collecting and analyzing the traces, and Daniel Berger for his comments in the early stage of the project. Moreover, we thank CloudLab [40] in helping us process the open-sourced traces, and Geoff Kuenning from SNIA in helping hosting and sharing the traces. This work was supported in part by NSF grants CNS 1901410 and CNS 1956271.

References

[1] Anonymized twitter production cache traces. https://github.com/twitter/cache-trace.

[2] Apache aurora. http://aurora.apache.org/. Accessed: 2020-05-06.

[3] Apache mesos. http://mesos.apache.org/. Accessed: 2020-05-06.

[4] Apache traffic server. https://trafficserver.apache.org/. Accessed: 2020-05-06.

[5] Art. 17 gdpr right to erasure ('right to be forgotten'). https://gdpr-info.eu/art-17-gdpr/. Accessed: 2020-05-06.

[6] Caching with twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache.html. Accessed: 2020-10-10.

[7] database caching strategy using redis. https://d0.awsstatic.com/whitepapers/Database/database-caching-strategies-using-redis.pdf. Accessed: 2020-05-06.

[8] Decomposing twitter: Adventures in service-oriented architecture. https://www.infoq.com/presentations/twitter-soa/. Accessed: 2020-09-25.

[9] Do not join lru and slab maintainer threads if they do not exist. https://github.com/memcached/memcached/pull/686. Accessed: 2020-08-06.

[10] Enhance slab reallocation for burst of evictions. https://github.com/memcached/memcached/pull/695. Accessed: 2020-08-06.

[11] Experiencing slab ooms after one week of uptime. https://github.com/memcached/memcached/issues/689. Accessed: 2020-08-06.

[12] How to interpret r-squared and goodness-of-fit in regression analysis. https://www.datasciencecentral.com/profiles/blogs/regression-analysis-\how-do-i-interpret-r-squared-and-assess-the. Accessed: 2020-09-28.

[13] jemalloc. http://jemalloc.net/. Accessed: 2020-05-06.

[14] memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2020-05-06.

[15] Paper review: Memc3. https://memcached.org/blog/paper-review-memc3/. Accessed: 2020-05-06.

[16] Redis. http://redis.io/. Accessed: 2020-05-06.

[17] slab auto-mover anti-favours slab 2. https://github.com/memcached/memcached/issues/677. Accessed: 2020-08-06.

[18] Varnish cache. https://varnish-cache.org/. Accessed: 2020-05-06.

[19] The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.

[20] Mehmet Altinel, Christof Bornhoevd, Chandrasekaran Mohan, Mir Hamid Pirahesh, Berthold Reinwald, and Saileshwar Krishnamurthy. System and method for adaptive database caching, July 1 2008. US Patent 7,395,258.

[21] Martin Arlitt, Rich Friedrich, and Tai Jin. Workload characterization of a web proxy in a cable modem environment. *ACM SIGMETRICS Performance Evaluation Review*, 27(2):25–36, 1999.

[22] Martin Arlitt and Tai Jin. A workload characterization study of the 1998 world cup web site. *IEEE network*, 14(3):30–37, 2000.

[23] Martin F Arlitt and Carey L Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on networking*, 5(5):631–645, 1997.

[24] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[25] Mary G Baker, John H Hartman, Michael D Kupfer, Ken W Shirriff, and John K Ousterhout. Measurements of a distributed file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 198–212, 1991.

[26] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd : Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[27] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75. IEEE, 2015.

[28] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.

[29] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99*, volume 1, pages 126–134. IEEE, 1999.

[30] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[31] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*, pages 353–362, 2019.

[32] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[33] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, February 2020. USENIX Association.

[34] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[36] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[37] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[38] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[39] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, 2019.

[40] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, jul 2019.

[41] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[42] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[43] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[44] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–12, 2011.

[45] Wolfram Gloger. ptmalloc. http://www.malloc.de/en/. Accessed: 2020-05-06.

[46] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, 2015.

[47] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of

facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181, 2013.

[48] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2014.

[49] Sunghwan Ihm and Vivek S Pai. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 295–312, 2011.

[50] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[51] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. Dns performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 153–167, 2001.

[52] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. Slik : Scalable low-latency indexes for a key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 57–70, 2016.

[53] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[54] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[55] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 476–488, 2015.

[56] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica : A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[57] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[58] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce G Lindsay, and Jeffrey F Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611, 2002.

[59] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[60] John K Ousterhout, Herve Da Costa, David Harrison, John A Kunze, Mike Kupfer, and James G Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, pages 15–24, 1985.

[61] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, Savannah, GA, November 2016. USENIX Association.

[62] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*, San Jose, CA, USA, October 2012.

[63] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, November 2011. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[64] Weisong Shi, Randy Wright, Eli Collins, and Vijay Karamcheti. Workload characterization of a personalized web site and its implications for dynamic content caching. In *Proceedings of the 7th International Workshop on Web Caching and Content Distribution (WCW'02)*. Citeseer, 2002.

[65] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq : Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[66] Werner Vogels. File system usage in windows nt 4.0. *ACM SIGOPS Operating Systems Review*, 33(5):93–109, 1999.

[67] Patrick Wendell and Michael J Freedman. Going viral: flash crowds in an open cdn. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 549–558, 2011.

[68] wikimedia. Analytics/data lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching. Accessed: 2020-05-06.

[69] Wikimedia. caching overview - wikitech. https://wikitech.wikimedia.org/wiki/Caching_overview. Accessed: 2020-05-06.

[70] John Wilkes. More Google cluster data. Google research blog, November 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[71] Juncheng Yang. libcachesim. https://github.com/1a1a11a/libCacheSim. Accessed: 2020-09-28.

[72] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 66–79, 2017.