# Workload Analysis of a Large-Scale Key-Value Store

Berk Atikoglu
Stanford, Facebook
atikoglu@stanford.edu

Yuehai Xu
Wayne State, Facebook
yhxu@wayne.edu

Eitan Frachtenberg*
Facebook
etc@fb.com

Song Jiang
Wayne State
sjiang@wayne.edu

Mike Paleczny
Facebook
mpal@fb.com

## ABSTRACT

Key-value stores are a vital component in many scale-out enterprises, including social networks, online retail, and risk analysis. Accordingly, they are receiving increased attention from the research community in an effort to improve their performance, scalability, reliability, cost, and power consumption. To be effective, such efforts require a detailed understanding of realistic key-value workloads. And yet little is known about these workloads outside of the companies that operate them. This paper aims to address this gap.

To this end, we have collected detailed traces from Facebook's Memcached deployment, arguably the world's largest. The traces capture over 284 billion requests from five different Memcached use cases over several days. We analyze the workloads from multiple angles, including: request composition, size, and rate; cache efficacy; temporal patterns; and application use cases. We also propose a simple model of the most representative trace to enable the generation of more realistic synthetic workloads by the community.

Our analysis details many characteristics of the caching workload. It also reveals a number of surprises: a GET/SET ratio of 30:1 that is higher than assumed in the literature; some applications of Memcached behave more like persistent storage than a cache; strong locality metrics, such as keys accessed many millions of times a day, do not always suffice for a high hit rate; and there is still room for efficiency and hit rate improvements in Memcached's implementation. Toward the last point, we make several suggestions that address the exposed deficiencies.

## Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Databases; D.4.8 [**Performance**]: Modeling and Prediction; D.4.2 [**Storage Management**]: Distributed Memories

*Corresponding author.


*SIGMETRICS'12,* June 11–15, 2012, London, England, UK.


## Keywords

Workload Analysis, Workload modeling, Key-Value Store

## 1. INTRODUCTION

Key-value (KV) stores play an important role in many large websites. Examples include: Dynamo at Amazon [15]; Redis at GitHub, Digg, and Blizzard Interactive [27]; Memcached at Facebook, Zynga and Twitter [18, 26]; and Voldemort at Linkedin [1]. All these systems store ordered $(key, value)$ pairs and are, in essence, a distributed hash table.

A common use case for these systems is as a layer in the data-retrieval hierarchy: a cache for expensive-to-obtain values, indexed by unique keys. These values can represent any data that is cheaper or faster to cache than re-obtain, such as commonly accessed results of database queries or the results of complex computations that require temporary storage and distribution.

Because of their key role in large website performance, KV stores are carefully tuned for low response times and high hit rates. But like all caching heuristics, a KV-store's performance is highly dependent on its workload. It is therefore imperative to understand the workload's characteristics. Additionally, analyzing and understanding large-scale cache workloads can also: provide insights into topics such as the role and effectiveness of memory-based caching in distributed website infrastructure; expose the underlying patterns of user behavior; and provide difficult-to-obtain data and statistical distributions for future studies.

In this paper, we analyze five workloads from Facebook's Memcached deployment. Aside from the sheer scale of the site and data (over 284 billion requests over a period of 58 sample days), this case study also introduces to the community several different usage scenarios for KV stores. This variability serves to explore the relationship between the cache and various data domains: where overall site patterns are adequately handled by a generalized caching infrastructure, and where specialization would help. In addition, this paper offers the following key contributions and findings:

1. A workload decomposition of the traces that shows how different applications of Memcached can have extreme variations in terms of read/write mix, request sizes and rates, and usage patterns (Sec. 3).

2. An analysis of the caching characteristics of the traces and the factors that determine hit rates. We found that different Memcached pools can vary significantly in their locality metrics, but surprisingly, the best predictor of hit rates is actually the pool's size (Sec. 6).

3. An examination of various performance metrics over time, showing diurnal and weekly patterns (Sec. 3.3, 4.2.2, 6).

4. An analytical model that can be used to generate more realistic synthetic workloads. We found that the salient size characteristics follow power-law distributions, similar to other storage and Web-serving systems (Sec. 5).

5. An exposition of a Memcached deployment that can shed light on real-world, large-scale production usage of KV-stores (Sec. 2.2, 8).

The rest of this paper is organized as follows. We begin by describing the architecture of Memcached, its deployment at Facebook, and how we analyzed its workload. Sec. 3 presents the observed experimental properties of the trace data (from the request point of view), while Sec. 4 describes the observed cache metrics (from the server point of view). Sec. 5 presents a simple analytical model of the most representative workload. The next section brings the data together in a discussion of our results, followed by a section surveying previous efforts on analyzing cache behavior and workload analysis.

# 2. MEMCACHED DESCRIPTION

## 2.1 Architecture

Memcached[1] is a simple, open-source software package that exposes data in RAM to clients over the network. As data size grows in the application, more RAM can be added to a server, or more servers can be added to the network. Additional servers generally only communicate with clients. Clients use consistent hashing [9] to select a unique server per *key*, requiring only the knowledge of the total number of servers and their IP addresses. This technique presents the entire aggregate data in the servers as a unified distributed hash table, keeps servers completely independent, and facilitates scaling as data size grows.

Memcached's interface provides the basic primitives that hash tables provide—insertion, deletion, and retrieval—as well as more complex operations built atop them.

Data are stored as individual items, each including a key, a value, and metadata. Item size can vary from a few bytes to over $100\,KB$, heavily skewed toward smaller items (Sec. 3). Consequently, a naïve memory allocation scheme could result in significant memory fragmentation. To address this issue, Memcached adopts a slab allocation technique, in which memory is divided into slabs of different sizes. The slabs in a class store items whose sizes are within the slab's specific range. A newly inserted item obtains its memory space by first searching the slab class corresponding to its size. If this search fails, a new slab of the class is allocated from the heap. Symmetrically, when an item is deleted from the cache, its space is returned to the appropriate slab, rather than the heap. Memory is allocated to slab classes based on the initial workload and its item sizes, until the heap is exhausted. Consequently, if the workload characteristics change significantly after this initial phase, we may find that the slab allocation is inappropriate for the workload, resulting in memory underutilization.

[1] `http://memcached.org/`

**Table 1: Memcached pools sampled (in one cluster). These pools do not match their UNIX namesakes, but are used for illustrative purposes here instead of their internal names.**

| Pool | Size | Description |
|---|---|---|
| USR | few | user-account status information |
| APP | dozens | object metadata of one application |
| ETC | hundreds | nonspecific, general-purpose |
| VAR | dozens | server-side browser information |
| SYS | few | system data on service location |

A new item arriving after the heap is exhausted requires the eviction of an older item in the appropriate slab. Memcached uses the Least-Recently-Used (LRU) algorithm to select the items for eviction. To this end, each slab class has an LRU queue maintaining access history on its items. Although LRU decrees that any accessed item be moved to the top of the queue, this version of Memcached coalesces repeated accesses of the same item within a short period (one minute by default) and only moves this item to the top the first time, to reduce overhead.

## 2.2 Deployment

Facebook relies on Memcached for fast access to frequently-accessed values. Web servers typically try to read persistent values from Memcached before trying the slower backend databases. In many cases, the caches are demand-filled, meaning that generally, data is added to the cache after a client has requested it and failed.

Modifications to persistent data in the database often propagate as deletions (invalidations) to the Memcached tier. Some cached data, however, is transient and not backed by persistent storage, requiring no invalidations.

Physically, Facebook deploys front-end servers in multiple datacenters, each containing one or more *clusters* of varying sizes. Front-end clusters consist of both Web servers, running primarily HipHop [31], and caching servers, running primarily Memcached. These servers are further subdivided based on the concept of *pools*. A pool is a partition of the entire key space, defined by a prefix of the key, and typically represents a separate application or data domain. The main reason for separate domains (as opposed to one all-encompassing cache) is to ensure adequate quality of service for each domain. For example, one application with high turnover rate could evict keys of another application that shares the same server, even if the latter has high temporal locality but lower access rates. Another reason to separate domains is to facilitate application-specific capacity planning and performance analysis.

In this paper, we describe traces from five separate pools—one trace from each pool (traces from separate machines in the same pool exhibit similar characteristics). These pools represent a varied spectrum of application domains and cache usage characteristics (Table 1). One pool in particular, ETC, represents general cache usage of multiple applications, and is also the largest of the pools; the data collected from this trace may be the most applicable to general-purpose KV-stores.

The focus of this paper is on workload characteristics, patterns, and relationships to social networking, so the exact details of server count and components have little relevance

here. It is important to note, however, that all Memcached instances in this study ran on identical hardware.

## 2.3 Tracing Methodology

Our analysis called for complete traces of traffic passing through Memcached servers for at least a week. This task is particularly challenging because it requires nonintrusive instrumentation of high-traffic volume production servers. Standard packet sniffers such as tcpdump[2] have too much overhead to run under heavy load. We therefore implemented an efficient packet sniffer called *mcap*. Implemented as a Linux kernel module, mcap has several advantages over standard packet sniffers: it accesses packet data in kernel space directly and avoids additional memory copying; it introduces only 3% performance overhead (as opposed to tcpdump's 30%); and unlike standard sniffers, it handles out-of-order packets correctly by capturing incoming traffic after all TCP processing is done. Consequently, mcap has a complete view of what the Memcached server sees, which eliminates the need for further processing of out-of-order packets. On the other hand, its packet parsing is optimized for Memcached packets, and would require adaptations for other applications.

The captured traces vary in size from $3TB$ to $7TB$ each. This data is too large to store locally on disk, adding another challenge: how to offload this much data (at an average rate of more than $80,000$ samples per second) without interfering with production traffic. We addressed this challenge by combining local disk buffering and dynamic offload throttling to take advantage of low-activity periods in the servers.

Finally, another challenge is this: how to effectively process these large data sets? We used Apache HIVE[3] to analyze Memcached traces. HIVE is part of the Hadoop framework that translates SQL-like queries into MapReduce jobs. We also used the Memcached "stats" command, as well as Facebook's production logs, to verify that the statistics we computed, such as hit rates, are consistent with the aggregated operational metrics collected by these tools.

# 3. WORKLOAD CHARACTERISTICS

This section describes the observed properties of each trace in terms of the requests that comprise it, their sizes, and their frequencies.

## 3.1 Request Composition

We begin by looking at the basic data that comprises the workload: the total number of requests in each server, broken down by request types (Fig. 1). Several observations delineate the different usage of each pool:

**USR** handles significantly more GET requests than any of the other pools. GET operations comprise over 99.8% of this pool's workload. One reason for this is that the pool is sized large enough to maximize hit rates, so refreshing values is rarely necessary. These values are also updated at a slower rate than some of the other pools. The overall effect is that USR is used more like RAM-based persistent storage than a cache.

**APP** has high GET rates too—owing to the popularity of this application—but also a large number of DELETE operations. DELETE operations occur when a cached database entry is modified (but not required to be set again in the cache). SET operations occur when the Web servers add a value to the cache. The relatively high number of DELETE operations show that this pool represents database-backed values that are affected by frequent user modifications.

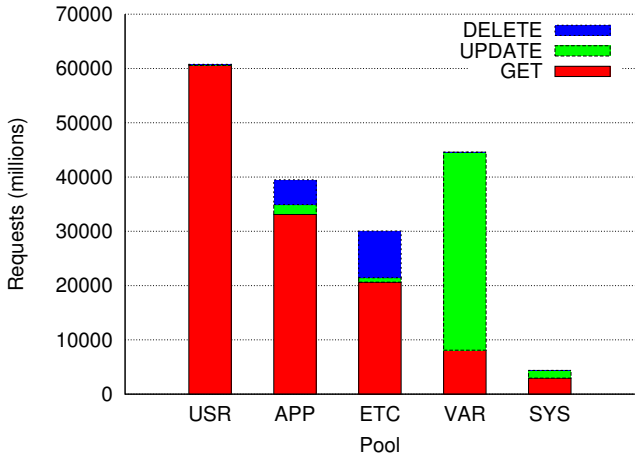


**Figure 1: Distribution of request types per pool, over exactly 7 days. UPDATE commands aggregate all non-DELETE writing operations, such as SET, REPLACE, etc.**

**ETC** has similar characteristics to APP, but with an even higher rate of DELETE requests (of which some may not be currently cached). ETC is the largest and least specific of the pools, so its workloads might be the most representative to emulate. Because it is such a large and heterogenous workload, we pay special attention to this workload throughout the paper.

**VAR** is the only pool sampled that is write-dominated. It stores short-term values such as browser-window size for opportunistic latency reduction. As such, these values are not backed by a database (hence, no invalidating DELETEs are required). But they change frequently, accounting for the high number of UPDATEs.

**SYS** is used to locate servers and services, not user data. As such, the number of requests scales with the number of servers, not the number of user requests, which is much larger. This explains why the total number of SYS requests is much smaller than the other pools'.

It is interesting to note that the ratio of GETs to UPDATEs in ETC (approximately $30:1$) is significantly higher than most synthetic workloads typically assume (Sec. 7). For demand-filled caches like USR, where each miss is followed by an UPDATE, the ratios of GET to UPDATE operations mentioned above are related to hit rate in general and the sizing of the cache to the data in particular. So in theory, one could justify any synthetic GET to UPDATE mix by controlling the cache size. But in practice, not all caches or keys are demand-filled, and these caches are already sized to fit a real-world workload in a way that successfully trades off hit rates to cost.

[2] `http://www.tcpdump.org/`
[3] `http://hive.apache.org/`

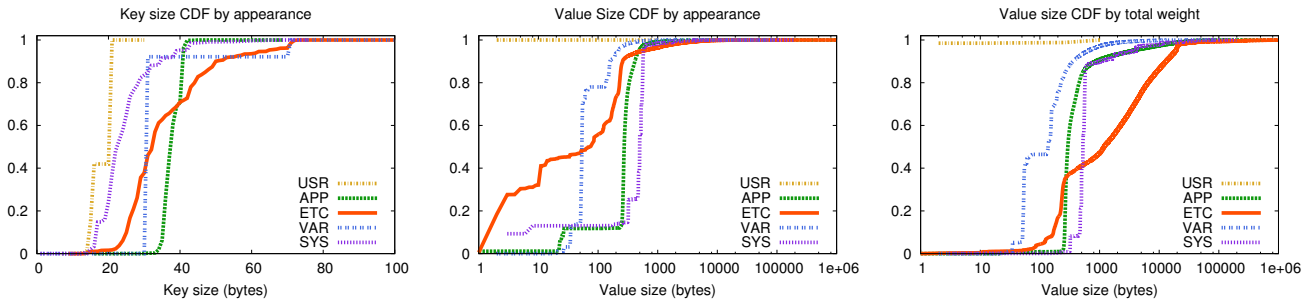


**Figure 2: Key and value size distributions for all traces. The leftmost CDF shows the sizes of keys, up to Memcached's limit of $250\,B$ (not shown). The center plot similarly shows how value sizes distribute. The rightmost CDF aggregates value sizes by the total amount of data they use in the cache, so for example, values under $320\,B$ or so in SYS use virtually no space in the cache; $320\,B$ values weigh around $8\%$ of the data, and values close to $500\,B$ take up nearly $80\%$ of the entire cache's allocation for values.**

## 3.2 Request Sizes

Next, we look at the sizes of keys and values in each pool (Fig. 2), based on SET requests. All distributions show strong modalities. For example, over 90% of APP's keys are 31 bytes long, and values sizes around $270\,B$ show up in more than 30% of SET requests. USR is the most extreme: it only has two key size values ($16\,B$ and $21\,B$) and virtually just one value size ($2\,B$). Even in ETC, the most heterogeneous of the pools, requests with 2-, 3-, or 11-byte values add up to 40% of the total requests. On the other hand, it also has a few very large values (around $1\,MB$) that skew the weight distribution (rightmost plot in Fig. 2), leaving less caching space for smaller values.

Small values dominate all workloads, not just in count, but especially in overall weight. Except for ETC, 90% of all cache space is allocated to values of less than $500\,B$. The implications for caching and system optimizations are significant. For example, network overhead in the processing of multiple small packets can be substantial, which explains why Facebook coalesces as many requests as possible in as few packets as possible [9]. Another example is memory fragmentation. The strong modality of each workload implies that different Memcached pools can optimize memory allocation by modifying the slab size constants to fit each distribution. In practice, this is an unmanageable and unscalable solution, so instead Memcached uses many (44) slab classes with exponentially growing sizes, in the hope of reducing allocation waste, especially for small sizes.

## 3.3 Temporal Patterns

To understand how production Memcached load varies over time, we look at each trace's transient request rate over its entire collection period (Fig. 3). All traces clearly show the expected diurnal pattern, but with different values and amplitudes. If we increase our zoom factor further (as in the last plot), we notice that traffic in ETC bottoms out around 08:00 and has two peaks around 17:00 and 03:00. Not surprisingly, the hours immediately preceding 08:00 UTC (midnight in Pacific Time) represent night time in the Western Hemisphere.

The first peak, on the other hand, occurs as North America starts its day, while it is evening in Europe, and continues until the later peak time for North America. Although different traces (and sometimes even different days in the same trace) differ in which of the two peaks is higher, the entire period between them, representing the Western Hemisphere day, sees the highest traffic volume. In terms of weekly patterns, we observe a small traffic drop on most Fridays and Saturdays, with traffic picking up again on Sundays and Mondays.

The diurnal cycle represents load variation on the order of $2\times$. We also observe the presence of traffic spikes. Typically, these can represent a swift surge in user interest on one topic, such as occur with major news or media events. Less frequently, these spikes stem from programmatic or operational causes. Either way, the implication for Memcached development and deployment is that one must budget individual node capacity to allow for these spikes, which can easily double or even triple the normal peak request rate. Although such budgeting underutilizes resources during normal traffic, it is nevertheless imperative; otherwise, the many Web servers that would take to this sudden traffic and fail to get a prompt response from Memcached, would all query the same database nodes. This scenario could be debilitating, so it must remain hypothetical.

# 4. CACHE BEHAVIOR

The main metric used in evaluating cache efficacy is hit rate: the percentage of GET requests that return a value. The overall hit rate of each server, as derived from the traces and verified with Memcached's own statistics, are shown in Table 2. This section takes a deeper look at the factors that influence these hit rates and how they relate to cache locality, user behavior, temporal patterns, and Memcached's design.

**Table 2: Mean cache hit rate over entire trace.**

| **Pool** | APP | VAR | SYS | USR | ETC |
|---|---|---|---|---|---|
| **Hit rate** | 92.9% | 93.7% | 98.7% | 98.2% | 81.4% |

## 4.1 Hit Rates over Time

When looking at how hit rates vary over time (Fig. 4), almost all traces show diurnal variance, within a small band of a few percentage points. USR's plot is curious: it appears to be monotonically increasing (with diurnal undulation). This behavior stems from the usage model for USR. Recall

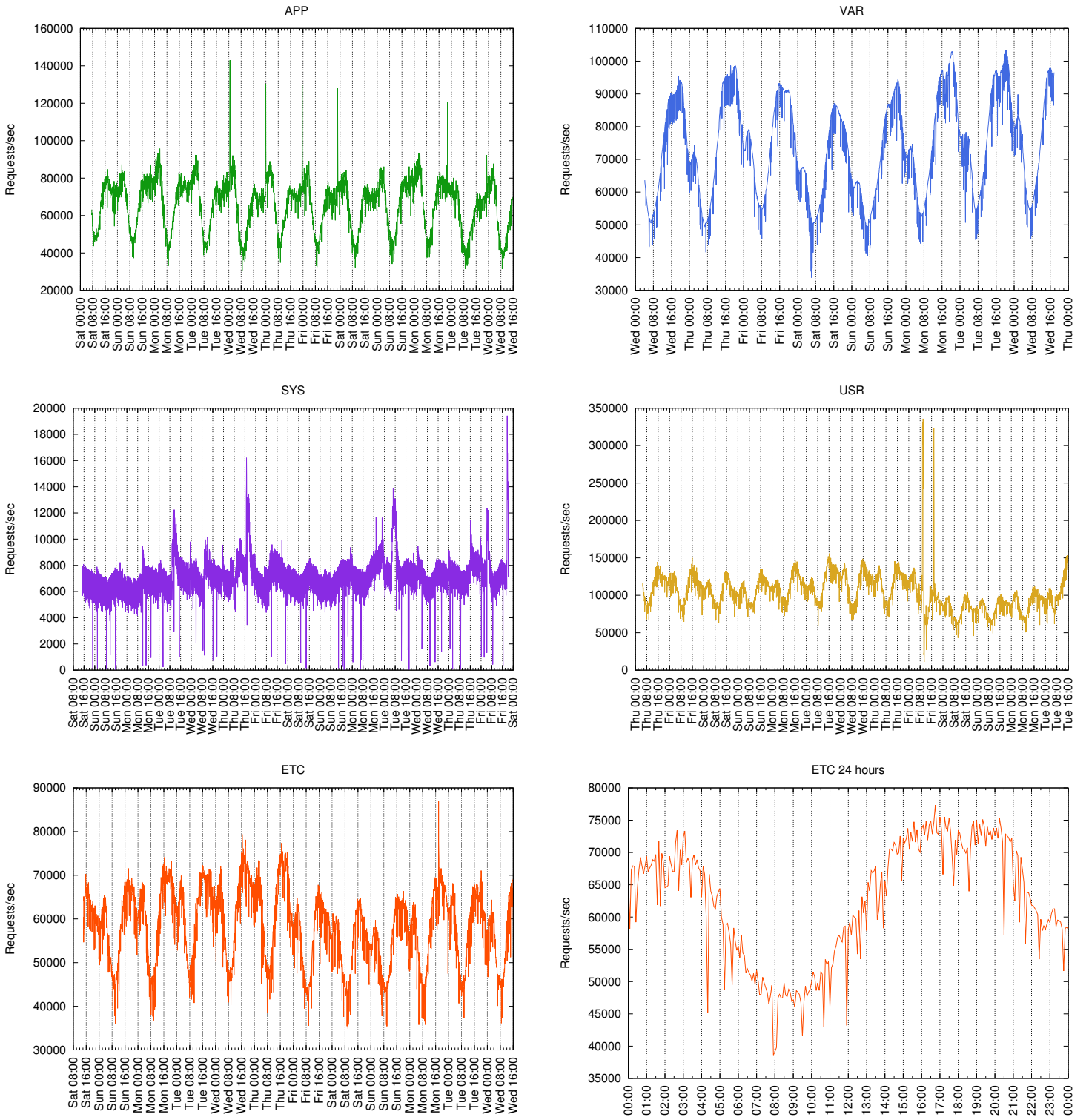


**Figure 3: Request rates at different dates and times of day, Coordinated Universal Time (UTC). Each data point counts the total number of requests in the preceding second. Except for USR and VAR, different traces were collected in different times. The last plot zooms in on a 24-hour period from the ETC trace for greater detail.**

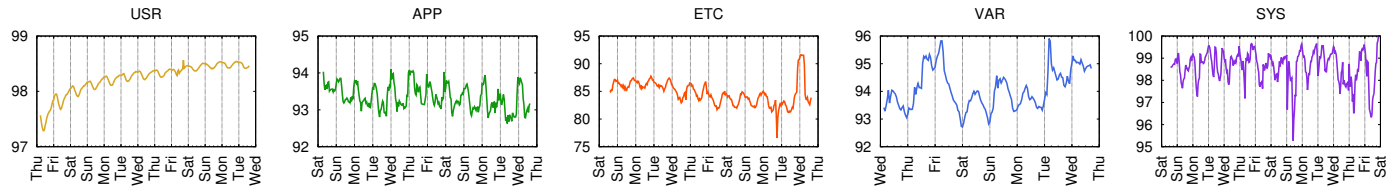


**Figure 4: GET hit rates over time for all pools (days start at midnight UTC).**

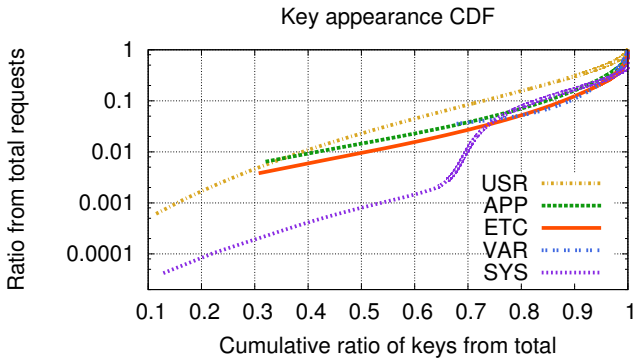


**Figure 5: CDFs of key appearances, depicting how many keys account for how many requests, in relative terms. Keys are ranked from least popular to most popular.**

from Sec. 2.2 that USR is sized large enough to minimize the number of misses—in other words, to contain almost all possible keys. When a USR Memcached server starts, it contains no data and misses on all requests. But over time, as clients add values to it while the pressure to evict is nonexistent, hit rates climb upwards. Thus, USR's transient hit rate is correlated not only with time of day, but primarily with the server's uptime, reaching 99.8% after several weeks.

Like USR, SYS has a relatively bounded data domain, so it can easily be sized to keep hit rates high and stable. But unlike the other four workloads, SYS does not react directly to user load, so its performance is less cyclical and regular.

## 4.2 Locality Metrics

This section looks at three ways to measure locality in GET requests: (1) how often and how much some keys repeat in requests; (2) the amount of unique keys and how it varies over time; and (3) reuse period, as a measure of temporal locality.

These metrics, unlike hit rates, are an inherent property of the request stream of each pool; changing the server's hardware or server count will not affect them. Consequently, this data could provide insights toward the workload, in isolation of implementation choices.

### 4.2.1 *Repeating Keys*

We start by looking at the distribution of key repeats (Fig. 5). All workloads exhibit the expected long-tail distributions, with a small percentage of keys appearing in most of the requests, and most keys repeating only a handful of times. So, for example, 50% of ETC's keys occur in only 1% of all requests, meaning they do not repeat many times, while a few popular keys repeat in millions of requests per day. This high concentration of repeating keys provides the justification for caching them in the first place.

All curves are remarkably similar, except for SYS's, which has two distinct sections. The first, up to about 65% of the keys, represents keys that are repeated infrequently—conceivably those that are retrieved when one or more clients start up and fill their local cache. The second part, representing the last 25% of keys and more than 90% of the requests, may account for the normal SYS scenario, when a value is added or updated in the cache and all the clients retrieve it.

### 4.2.2 *Locality over Time*

It is also interesting to examine how key uniqueness varies over time by counting how many keys *do not* repeat in close time proximity (Fig. 6). To interpret this data, note that a lower percentage indicates that fewer keys are unique, and therefore suggests a higher hit rate. Indeed, note that the diurnal dips correspond to increases in hit rates in Fig. 4.

An immediately apparent property is that in any given pool, this percentage remains relatively constant over time—especially with hour-long bins, with only small diurnal variations and few spikes. Data for 5-minute bins are naturally noisier, but even here most samples remain confined to a narrow range. This suggests that different pools have not only different traffic patterns, but also different caching properties that can benefit from different tuning, justifying the choice to segregate workloads to pools.

Each pool exhibits its characteristic locality band and average. SYS's low average rate of 3.3% unique keys per hour, for example, suggests that different clients request roughly the same service information. In contrast, USR's much higher average rate of 34.6% unique keys per hour, suggests that the per-user data it represents spans a much more disparate range . Generally, we would assume that lower bands translate to higher overall hit rates, all other things being equal. This turns out not to be the case. In fact, the Pearson correlation coefficient between average unique key ratios with 60-minute bins (taken from Fig. 6) and the average hit rates (Table 2) is negative as expected, but small: $-0.097$. Indeed not all other things are equal, as discussed in Sec. 4.1.

Comparing 5-minute bins to hour-long bins reveals that unique keys in the former appear in significantly higher concentrations than in the latter. This implies a rapid rate of decay in interest in most keys. But does this rate continue to drop very fast over a longer time window? The next section sets out to answer this question.

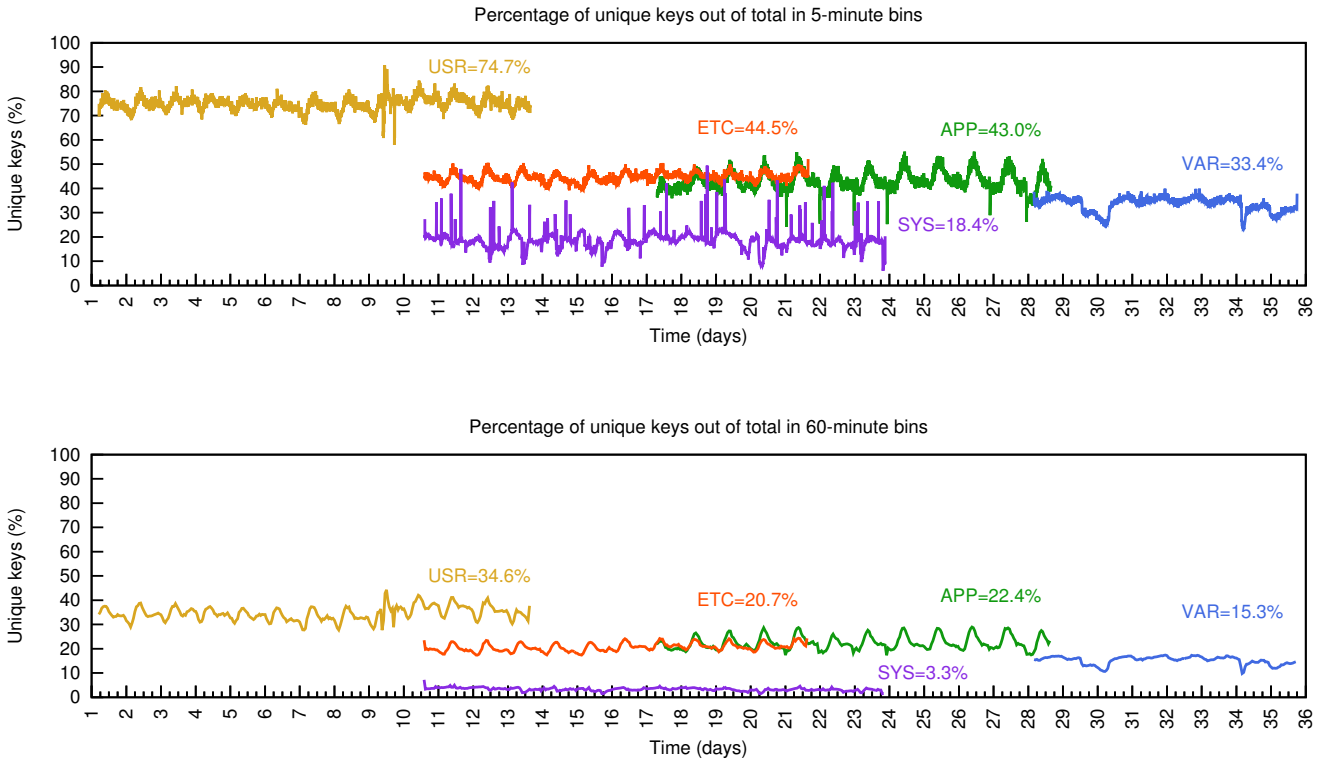


**Figure 6: Ratio of unique keys over time. Each data point on the top (bottom) plot shows how many unique keys were requested in the preceding 5 (60) minutes, as percentage of all keys. The label for each pool, at the top right corner of the data, also includes the average ratio throughout the entire pool's trace.**

### *4.2.3 Temporal Locality: Reuse Period*

Temporal locality refers to how often a key is re-accessed. One metric to quantify temporal locality of any given key is the reuse period—the time between consecutive accesses to the key. Fig. 7 counts all key accesses in the five traces, and bins them according to the time duration from the previous key's access. Unique keys (those that do not repeat at all within the trace period) are excluded from this count.

The answer to the question from the previous section is therefore positive: count of accesses in each reuse period continues to decay quickly after the first hour. For the ETC trace, for example, 88.5% of the keys are reused within an hour, but only 4% more within two, and within six hours, 96.4% of all nonunique keys have already repeated. It continues to decay at a slower rate. This access behavior suggests a pattern for Facebook's users as well, with some users visiting the site more frequently than others and reusing the keys associated with their accounts. Another interesting sub-pattern occurs every day. Note the periodic peaks on even 24 hours in four of the five traces, especially in the VAR pool that is associated with browser usage. These peaks suggest that a noteworthy number of users log in to the site at approximately the same time of day each time. Once more, these increased-locality indications also correspond to increased hit rates in Fig. 4.

As in the previous section, the SYS pool stands out. It does not show the same 24-hour periodicity, because its keys

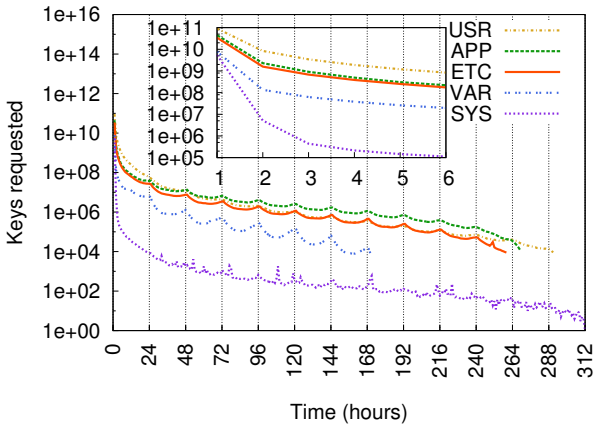


**Figure 7: Reuse period histogram per pool. Each hour-long bin $n$ counts keys that were first requested $n$ hours after their latest appearance. The inset zooms in on the five hours after the first.**

relate to servers and services, not users. It also decays precipitously compared to the others. As in Sec. 4.2.1, we find that since its data are cached locally by clients, it is likely that most of SYS's GET requests represent data that are newly available, updated or expired from the client cache; these are then requested by many clients concurrently. This would explain why 99.9% of GET requests are repeated within an hour of the first key access. Later, such keys would be cached locally and accessed rarely, perhaps when a newly added client needs to fill its own cache.

Nevertheless, there is still value in reuse period to predict hit rates. Since all pools have sufficient memory for over an hour of fresh data, the percentage of keys reused within an hour correlates positively with the overall hit rates in Table 2 (with a Pearson coefficient of 0.17). The correlation is stronger—with a coefficient of 0.44—if we omit USR and SYS, which have atypical cache behavior (minimum evictions in the former and local caching in the latter).

## 4.3 Case Study: ETC Hit Rates

We turn our attention to ETC's hit/miss rates, because frequent misses can noticeably hurt user experience. At this point, one might expect ETC's hit rate to exceed the 96% 6-hour key-reuse rate, since it is provisioned with more than enough RAM to store the fresh data of the preceding 6 hours. Unfortunately, this is not the case, and the observed hit rate is significantly lower at 81%. To understand why, we analyzed all the misses in the last 24 hours of the trace (Table. 3). The largest number of misses in ETC comes from keys that are accessed for the first time (at least in a 10-day period). This is the long tail of the locality metrics we analyzed before. Sec. 4.2.1 showed that $\approx 50\%$ of ETC's keys are accessed in only 1% of requests, and therefore benefit little or not at all from a demand-filled cache. The many deletions in the cache also hinder the cache's efficacy. ETC is a very diverse pool with many applications, some with limited reusability. But the other half of the keys that show up in 99% of the requests are so popular (some repeating millions of times) that Memcached can satisfy over 4 in 5 requests to the ETC pool.

**Table 3: Miss categories in last 24 hours of the ETC trace. Compulsory misses count GETs with no matching SET in the preceding 10 days (meaning, for all practical purposes, new keys to the cache). Invalidation misses count GETs preceded by a matching DELETE request. Eviction misses count all other missing GETs.**

| **Miss category** | Compulsory | Invalidation | Eviction |
|---|---|---|---|
| **Ratio of misses** | 70% | 8% | 22% |

# 5. STATISTICAL MODELING

This section describes the salient workload characteristics of the ETC trace using simple distribution models. The ETC trace was selected because it is both the most representative of large-scale, general-purpose KV stores, and the easiest to model, since it is not distorted by the idiosyncratic aberrations of application-specific pools. We also think that its mixed workload is easier to generalize to other general-purpose caches with a heterogeneous mix of requests and sizes. The more Facebook-specific workloads, such as USR or SYS, may be interesting as edge cases, but probably not so much as models for synthetic workloads.

The functional models presented here prioritize parsimonious characterization over fidelity. As such, they obviously do not capture all the nuances of the trace, such as its bursty nature or the inclusion of one-off events. But barring access to the actual trace, they can serve the community as a better basis for synthetic workload generation than assumptions based on guesswork or small-scale logs.

### *Methodology*

We modeled independently the three main performance properties that would enable simple emulation of this trace: key sizes, value sizes, and inter-arrival rates. The rate and ratio between GET/SET/DELETE requests can be derived from Sec. 3.1. For cache analysis, additional properties can be gleaned from Sec. 4.

To justify the assumption that the three properties are independent, we picked a sample of $1,000,000$ consecutive requests and measured the Pearson coefficient between each pair of variables. The pairwise correlations, as shown in Table 4, are indeed very low.

**Table 4: Pearson correlation coefficient between each two pair of modeled variables.**

| Variable pair | Correlation |
|---|---|
| Inter-arrival gap $\leftrightarrow$ Key size | $-0.0111$ |
| Inter-arrival gap $\leftrightarrow$ Value size | 0.0065 |
| Key size $\leftrightarrow$ Value size | $-0.0286$ |

We created functional models by fitting various distributions (such as Weibull, Gamma, Extreme Value, Normal, etc.) to each data set and choosing the distribution that minimizes the Kolmogorov-Smirnov distance. All our data resemble power-law distributions, and fit the selected models quite well, with the exception of a handful of points (see Fig. 8). To deal with these outliers and improve the fit, we removed these points as necessary and modeled the remaining samples. The few removed points are tabulated separately as a histogram, so a more accurate synthetic workload of the entire trace should combine the functional model with the short list of value-frequency outliers.

### *Key-Size Distribution*

We found the model that best fits key sizes in bytes (with a Kolmogorov-Smirnov distance of 10.5) to be Generalized Extreme Value distribution with parameters $\mu = 30.7984$, $\sigma = 8.20449$, $k = 0.078688$. We have verified that these parameters remain fairly constant, regardless of time of day.

### *Value-Size Distribution*

We found the model that best fits value sizes in bytes (with a Kolmogorov-Smirnov distance of 10.5), starting from 15 bytes, to be Generalized Pareto with parameters $\theta = 0$, $\sigma = 214.476$, $k = 0.348238$ (this distribution is also independent of the time of day). The first 15 values of length and probabilities can be modeled separately as a discrete probability distribution whose values are given in Table 5.

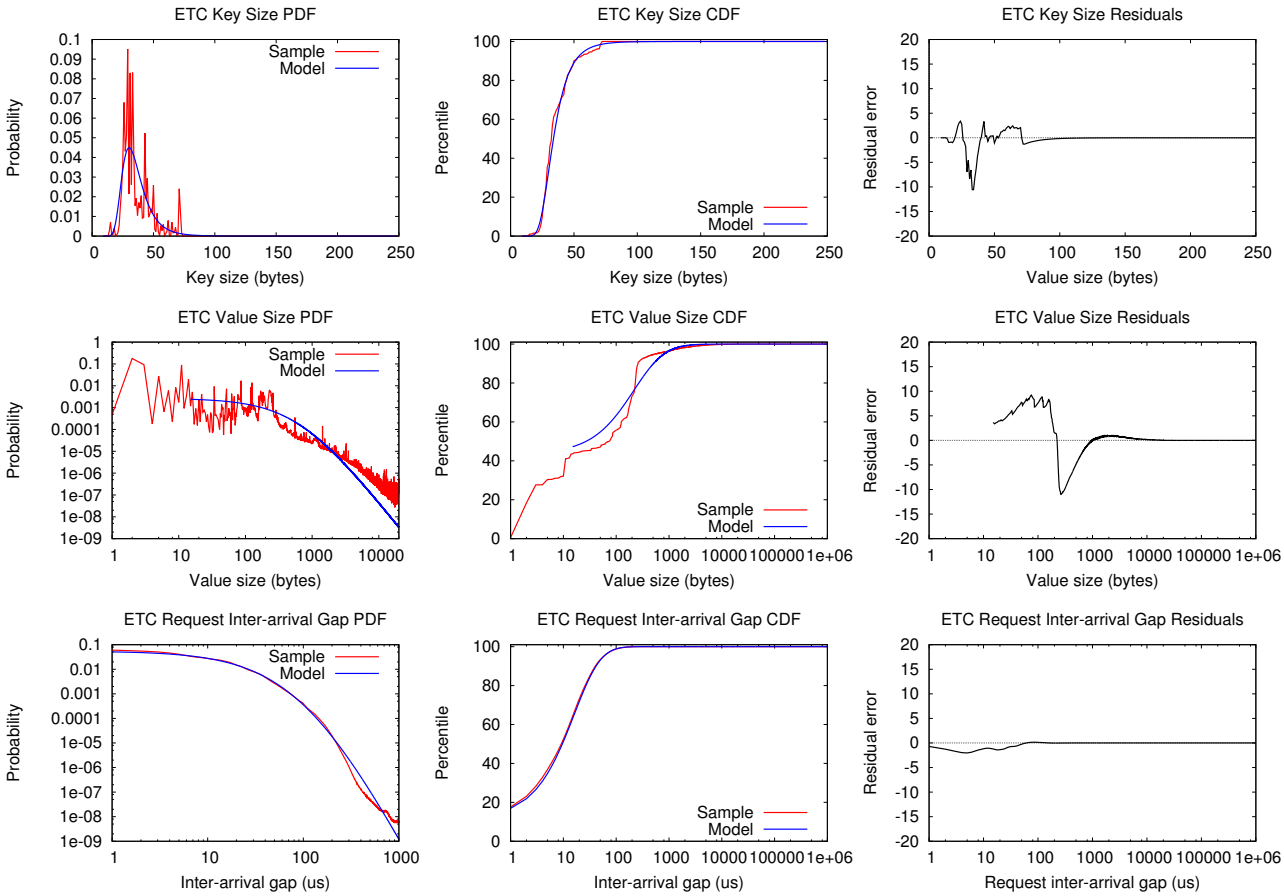


**Figure 8: PDF (left), CDF (middle), and CDF residuals (right) plots for the distribution of ETC's key size (top), value size (center). and inter-arrival gap (bottom). Note that some axes are logarithmic, and that PDF plots limit the X-axis to area of interest for greater detail.**

**Table 5: Probability distribution for first few value lengths, in bytes.**

| Value size | Probability |
|---|---|
| 0 | 0.00536 |
| 1 | 0.00047 |
| 2 | 0.17820 |
| 3 | 0.09239 |
| 4 | 0.00018 |
| 5 | 0.02740 |
| 6 | 0.00065 |
| 7 | 0.00606 |
| 8 | 0.00023 |
| 9 | 0.00837 |
| 10 | 0.00837 |
| 11 | 0.08989 |
| 12 | 0.00092 |
| 13 | 0.00326 |
| 14 | 0.01980 |

### *Inter-arrival Rate Distribution*

We found the model that best describes the time gap in microseconds between consecutive received requests (with a Kolmogorov-Smirnov distance of 2.0) to be Generalized Pareto with parameters $\theta = 0$, $\sigma = 16.0292$, $k = 0.154971$, starting from the second value.

One excluded point from this model is the first value, representing a gap of $0\,\mu sec$ (in other words, multiple requests at the same microsecond time slot), with a probability of 0.1159. This is likely an artifact of our measurement granularity and aggregation by the network stack, and not of concurrent requests, since they are all serialized by the single networking interface.

In addition, the model is most accurate up to about a gap of $1000\,\mu sec$. But the total number of sampled points not covered by this model (i.e., those requests that arrive more than $1msec$ after the previous request) represents less than 0.002% of the total samples and their residual error is negligible.

Unlike the previous two distributions, inter-arrival rate—the reciprocal function of offered load—is highly dependent on time of day, as evident in Fig. 3. For those wishing to capture this diurnal variation, this complete-trace model may be too coarse. To refine this distribution, we divided the

**Table 6: Hourly distributions for inter-arrival gap. The columns represent (in order): start time of each hourly bin (in UTC), the two Generalized Pareto parameters (with $\theta = 0$), the fraction of samples under $1\,\mu s$ gap , and the Kolmogorov-Smirnov distance of the fit.**

| Time | $\sigma$ | $k$ | $< 1\,\mu s$ | KS |
|---|---|---|---|---|
| 0:00 | 16.2868 | 0.155280 | 0.1158 | 2.18 |
| 1:00 | 15.8937 | 0.141368 | 0.1170 | 2.14 |
| 2:00 | 15.6345 | 0.137579 | 0.1174 | 2.09 |
| 3:00 | 15.7003 | 0.142382 | 0.1174 | 2.16 |
| 4:00 | 16.3231 | 0.160706 | 0.1176 | 2.32 |
| 5:00 | 17.5157 | 0.181278 | 0.1162 | 2.52 |
| 6:00 | 18.6748 | 0.196885 | 0.1146 | 2.64 |
| 7:00 | 19.5114 | 0.202396 | 0.1144 | 2.64 |
| 8:00 | 20.2050 | 0.201637 | 0.1123 | 2.58 |
| 9:00 | 20.2915 | 0.193764 | 0.1116 | 2.46 |
| 10:00 | 19.5577 | 0.178386 | 0.1122 | 2.35 |
| 11:00 | 18.2294 | 0.161636 | 0.1130 | 2.17 |
| 12:00 | 17.1879 | 0.140461 | 0.1138 | 2.00 |
| 13:00 | 16.2159 | 0.119242 | 0.1146 | 1.88 |
| 14:00 | 15.6716 | 0.104535 | 0.1152 | 1.76 |
| 15:00 | 15.2904 | 0.094286 | 0.1144 | 1.72 |
| 16:00 | 15.2033 | 0.096963 | 0.1136 | 1.72 |
| 17:00 | 14.9533 | 0.098510 | 0.1140 | 1.74 |
| 18:00 | 15.1381 | 0.096155 | 0.1128 | 1.67 |
| 19:00 | 15.3210 | 0.094156 | 0.1129 | 1.65 |
| 20:00 | 15.3848 | 0.100365 | 0.1128 | 1.68 |
| 21:00 | 15.7502 | 0.111921 | 0.1127 | 1.80 |
| 22:00 | 16.0205 | 0.131946 | 0.1129 | 1.96 |
| 23:00 | 16.3238 | 0.147258 | 0.1148 | 2.14 |

raw data into 24 hourly bins and modeled each separately. Fortunately, they all fit a Generalized Pareto distribution with $\theta = 0$ rather well. The remaining two parameters are distributed over time in Table. 6.

# 6. DISCUSSION

One pertinent question is, what are the factors that affect and predict hit rates? Since all hosts have the same amount of RAM, we should be able to easily explain the relative differences between different traces using the data we gathered so far on locality. But as Sec. 4 discusses, hit rates do not actually correlate very well with most locality metrics, but rather, correlates inversely with the size of the pool (compare Tables 1 and2). Does correlation imply causation in this case?

Probably not. A more likely explanation invokes a third, related parameter: the size of the application domain. Both in USR's and SYS's cases, these sizes are more or less capped, and the bound is small enough that a limited number of servers can cover virtually the entire domain, so locality no longer plays a factor. On the other extreme, ETC has a varying and growing number of applications using it, some with unbounded data. If any single application grows enough in importance to require a certain quality of service, and has the size limitations to enable this quality, given enough servers, then it is separated out of ETC to its own pool. So the applications that end up using ETC are precisely those that cannot or need not benefit from hit-rate guarantees.

Nevertheless, improving hit rates is important for these applications, or we would not need a cache in the first place. One way to improve ETC's hit rates, at least in theory, is to increase the total amount of RAM (or servers) in the pool so that we can keep a longer history. But in practice, beyond a couple of days' worth of history, the number of keys that would benefit from the longer memory is vanishingly small, as Fig. 7 shows. And of course, adding hardware adds cost.

A more fruitful direction may be to focus on the cache replacement policy. Several past studies demonstrated replacement policies with reduced eviction misses, compared to LRU, such as LIRS [19]. Table 3 puts an upper limit on the number of eviction misses that can be eliminated, at around 22%, meaning that eviction policy changes could improve hit rates by another $0.22 \times (1 - 0.814) = 4.1\%$. This may sound modest, but it represents over 120 million GET requests per day per server, with noticeable impact on service latency. Moreover, the current cache replacement scheme and its implementation are suboptimal when it comes to multithreaded performance [9], with its global lock protecting both hash table and slab LRUs. We therefore perceive great potential in alternative replacement policies, not only for better hit rates but also for better performance.

Another interesting question is whether we should optimize Memcached for hit rates or byte hit rates. To answer it, we estimated the penalty of each miss in the ETC workload, by measuring the duration between the missing GET and the subsequent SET that reinstates the value (presumably, the duration represents the time cost to recalculate the value, and is already highly optimized, emphasizing the importance of improved hit rates). We found that it is roughly proportional to the value size, so whether we fill the cache with few large items or many small items of the same aggregate size should not affect recalculation time much. On the other hand, frequent misses do noticeably increase the load on the back-end servers and hurt the user experience, which explains why this cache does not prioritize byte hit rate. Memcached optimizes for small values, because they are by far the most common values. It may even be worthwhile to investigate not caching large objects at all, to increase overall hit rates.

# 7. RELATED WORK

To the best of our knowledge, this is the first detailed description of a large-scale KV-store workload. Nevertheless, there are a number of related studies on other caching systems that can shed light on the relevance of this work and its methodology.

The design and implementation of any storage or caching system must be optimized for its workload to be effective. Accordingly, there is a large body of work on the collection, analysis, and characterization of the workloads on storage systems, including enterprise computing environments [2, 20, 21] and high-performance computing environments [11, 22, 30]. The observations can be of great importance to system design, engineering, and tuning. For example, in a study on file system workloads for large-scale scientific computing applications, Wang et. al. collected and analyzed file accesses on an 800-node cluster running the Lustre file system at Lawrence Livermore National Laboratory [30]. One of their findings is that in some workloads, small requests account for more than 90% of all requests, but almost all data are accessed by large requests. In a study on file sys-

tem workloads across different environments, Roselli et. al. found that even small caches can produce a high hit rate, but larger caches would have diminishing returns [28], similar to our conclusions on the ETC workload (Sec. 4.3).

In the work describing Facebook's photo storage system [8], the authors presented statistics of I/O requests for the photos, which exhibit clear diurnal patterns, consistent with our observations in this paper (Sec. 3.3).

Web caches are widely deployed as a caching infrastructure for speeding up Internet access. Their workloads have been collected and analyzed in Web servers [6, 25], proxies [5, 10, 16], and clients [7, 12]. In a study of requests received by Web servers, Arlitt and Williamson found that 80% of requested documents are smaller than $\approx 10KB$. However, requests to these documents generate only 26% of data bytes retrieved from the server [6]. This finding is consistent with the distribution we describe in Sec. 3.2

In an analysis of traces of client-side requests, Cunha et. al. show that many characteristics of Web use can be modeled using power-law distributions, including the distribution of document sizes, the popularity of documents, the distribution of user requests for documents, and the number of references to documents as a power law of their overall popularity rank (Zipf's law) [12]. Our modeling work on the ETC trace (Sec. 5) also shows power-law distributions in most request properties.

In light of the increasing popularity and deployment of KV-stores, several schemes were proposed to improve their performance, energy efficiency, and cost effectiveness [4, 9, 15, 26, 29]. Absent well-publicized workload traces, in particular large-scale production traces, many works used hypothetical or synthetic workloads [29]. For example, to evaluate SILT, a KV-store design that constructs a three-level store hierarchy for storage on flash memory with a memory based index, the authors assumed a workload of 10% PUT and 90% GET requests using $20B$ keys and $100B$ values, as well as a workload of 50% PUT and 50% GET requests for $64B$ KV pairs [23]. Andersen et. al. used queries of constant size ($256B$ keys and $1KB$ values) in the evaluation of FAWN, a KV-store designed for nodes consisting of low-power embedded CPUs and small amounts of flash storage [4]. In the evaluation of CLAM, a KV-store design that places both hash table and data items on flash, the authors used synthetic workloads that generate keys from a random distribution and a number of artificial workload mixes [3]. There are also some studies that used real workloads in KV-store evaluations. In two works on flash-based KV store-design, Debnath et. al. adopted workloads from online multi-player gaming and a storage de-duplication tool from Microsoft [13, 14]. Amazon's production workload was used to evaluate its Dynamo KV store, Dynamo [15]. However, these papers did not specifically disclose the workload characteristics.

Finally, there are multiple studies offering analytical models of observed, large-scale workloads. Of those, a good survey of the methods is presented in Lublin's and Feitelson's analysis of supercomputer workloads [17, 24].

# 8. CONCLUSION AND FUTURE WORK

This paper presented a dizzying number of views into a very large data set. Together, these views tell a coherent story of five different Memcached workloads at Facebook. We have ETC, the largest and most heterogeneous of the five. It has many keys that are requested millions of times a day, and yet its average hit rate is only 81.4% because half of its keys are accessed infrequently, and because a few large values take up a disproportionate amount of the storage. We have APP, which represents a single application and consequently has more uniform objects: 90% of them have roughly the same size. It also mirrors the interest of Facebook's users in specific popular objects, as evidenced in load spikes that are accompanied by improved locality metrics. We have VAR, a transient store for non-persistent performance data. It has three times as many writes as reads and 70% of its keys occur only once. But its 94% hit rate provides noticeable improvements to the user experience. We have USR, which is more like a RAM-based store for immutable two-byte values than a cache. It may not be the best fit for Memcached, but its overall data size is small enough that even a few Memcached servers can deliver a hit rate of 98.2%. And finally, we have SYS, another RAM-based storage that exhibits unique behavior, because its clients already cache its data. They only access SYS when new data or new clients show up, resulting in a low request rate and a nearly bimodal distribution of temporal locality: either a key is accessed many times in a short period, or virtually not at all.

This study has already answered pertinent questions to improve Facebook's Memcached usage. For example, Fig. 7 shows the relatively marginal benefit of significantly increasing the cache size for the ETC pool. As another example, the analysis in Sec. 6 demonstrated both the importance of increasing Memcached's hit rate, especially on larger data, as well as the upper bound on the potential increase.

The data presented here can also be used as a basis for new studies on key-value stores. We have also provided a simple analytical model of ETC's performance metrics to enable synthetic generation of more representative workloads. The treatment of workload modeling and synthetic load generation in this paper only scratches the surface of possibility, and deserves its own focus in a following publication. We plan to focus on this area and model the remaining workload parameters for ETC (such as key reuse), and other workloads as well. With these models, we would like to create representative synthetic load generators, and share those with the community.

We would also like to see improvements in the memory allocation model so that more room is saved for items in high demand. Areas of investigation include an adaptive slab allocation, using no slabs at all, or using prediction of item locality based on the analysis in this study.

Finally, we are looking into replacing Memcached's replacement policy. LRU is not optimal for all workloads, and can be quite slow. We have already started prototyping alternative replacement schemes, and the initial results are encouraging.


# Acknowledgements

We would like to thank Marc Kwiatkowski for spearheading this project and Mohan Srinivasan for helping with the kernel module. We are also grateful for the valuable feedback provided by the following: Goranka Bjedov, Rajiv Krishnamurthy, Rajesh Nishtala, Jay Parikh, and Balaji Prabhakar.

# 9. REFERENCES


[1] http://voldemort-project.com.

[2] Ahmad, I. Easy and efficient disk I/O workload characterization in VMware ESX server. In *Proceedings of IEEE International Symposium on Workload Characterization* (Sept. 2007).

[3] Anand, A., Muthukrishnan, C., Kappes, S., Akella, A., and Nath, S. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation* (Apr. 2010).

[4] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. FAWN: a fast array of wimpy nodes. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (Big Sky, Montana, 2009), ACM, pp. 1–14.

[5] Arlitt, M., Friedrich, R., and Jin, T. Workload characterization of a web proxy in a cable modem environment. *ACM SIGMETRICS - Performance Evaluation Review 27* (1999), 25–36.

[6] Arlitt, M. F., and Williamson, C. L. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking 5* (October 1997), 631–645.

[7] Barford, P., Bestavros, A., Bradley, A., and Crovella, M. Changes in web client access patterns. In *World Wide Web Journal, Special Issue on Characterization and Performance Evaluation* (1999).

[8] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (Oct. 2010).

[9] Berezecki, M., Frachtenberg, E., Paleczny, M., and Steele, K. Many-core key-value store. In *Proceedings of the Second International Green Computing Conference* (Orlando, FL, Aug. 2011).

[10] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the 18th Annual IEEE International Conference on Computer Communications* (1999).

[11] Carns, P., Latham, R., Ross, R., Kamil Iskra, S. L., and Riley, K. 24/7 characterization of petascale I/O workloads. In *Proceedings of the 4th Workshop on Interfaces and Architectures for Scientific Data Storage* (Nov. 2009).

[12] Cunha, C. R., Bestavros, A., and Crovella, M. E. Characteristics of WWW client-based traces. In *Technical Report TR-95-010, Boston University Department of Computer Science,* (July 1995).

[13] Debnath, B. K., Sengupta, S., and Li, J. Flashstore: High throughput persistent key-value store. *Proceedings of 36th International Conference on Very Large Data Bases (VLDB) 3*, 2 (2010).

[14] Debnath, B. K., Sengupta, S., and Li, J. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the Annual ACM SIGMOD Conference* (June 2010), pp. 25–36.

[15] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (Stevenson, WA, 2007), pp. 205–220.

[16] Duska, B. M., Marwood, D., and Feeley, M. J. The measured access characteristics of world-wide web client proxy caches. In *Proceedings of USENIX Symposium of Internet Technologies and Systems* (Dec. 1997).

[17] Feitelson, D. G. Workload modeling for performance evaluation. In *Performance Evaluation of Complex Systems: Techniques and Tools*, M. C. Calzarossa and S. Tucci, Eds., vol. 2459 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 2002, pp. 114–141. www.cs.huji.ac.il/~feit/papers/WorkloadModel02chap.ps.gz.

[18] Fitzpatrick, B. Distributed caching with memcached. *Linux Journal*, 124 (Aug. 2004), 72–78. www.linuxjournal.com/article/7451?page=0,0.

[19] Jiang, S., and Zhang, X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (2002), SIGMETRICS'02, ACM, pp. 31–42.

[20] Kavalanekar, S., Worthington, B., Zhang, Q., and Sharda, V. Characterization of storage workload traces from production windows servers. In *Proceedings of IEEE International Symposium on Workload Characterization* (Sept. 2008).

[21] Keeton, K., Alistair Veitch, D. O., and Wilkes, J. I/O characterization of commercial workloads. In *Proceedings of the 3rd Workshop on Computer Architecture Evaluation using Commercial Workloads* (Jan. 2000).

[22] Kim, Y., Gunasekaran, R., Shipman, G. M., Dillow, D. A., Zhang, Z., and Settlemyer, B. W. Workload characterization of a leadership class storage cluster. In *Proceedings of Petascale Data Storage Workshop* (Nov. 2010).

[23] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. Silt: A memory-eficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Oct. 2011).

[24] Lublin, U., and Feitelson, D. G. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing 63*, 11 (Nov. 2003), 1105–1122. www.cs.huji.ac.il/~feit/papers/Rigid01TR.ps.gz.

[25] Manley, S., and Seltzer, M. Web facts and fantasy. In *Proceedings of USENIX Symposium on Internet Technologies and Systems* (Dec. 1997).

[26] Petrovic, J. Using Memcached for data distribution in industrial environment. In *Proceedings of the Third International Conference on Systems* (Washington, DC, 2008), IEEE Computer Society, pp. 368–372.

[27] Reddi, V. J., Lee, B. C., Chilimbi, T., and Vaid, K. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)* (June 2010), ACM. portal.acm.org/citation.cfm?id=1815961.1816002.

[28] Roselli, D., Lorch, J. R., and Anderson, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference* (June 2000).

[29] Vasudevan, V. R. *Energy-Efficient Data-intensive Computing with a Fast Array of Wimpy Nodes*. PhD thesis, Carnegie Mellon University, Oct. 2011.

[30] Wang, F., Xin, Q., Hong, B., Miller, E. L., Long, D. D. E., Brandt, S. A., and McLarty, T. T. File system workload analysis for large scientific computing applications. In *Proceedings of 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies* (Apr. 2004).

[31] Zhao, H. HipHop for PHP: Move fast. https://developers.facebook.com/blog/post/358/, Feb. 2010.