Revisiting Cache Freshness for Emerging Real-Time Applications

Ziming Mao[†] Rishabh Iyer[†] Scott Shenker [†*] Ion Stoica [†]
[†]UC Berkeley [*]ICSI

Abstract

Caching is widely used in industry to improve application performance by reducing data-access latency and taking the load off the backend infrastructure. TTLs have become the de-facto mechanism used to keep cached data reasonably fresh (i.e., not too out of date with the backend). However, the emergence of real-time applications requires tighter data freshness, which is impractical to achieve with TTLs. We discuss why this is the case, and propose a simple yet effective adaptive policy to achieve the desired freshness.

CCS Concepts

• **Information systems → Information storage systems**; • **Computer systems organization**;

Keywords

Caching, Data Freshness, Time to Live (TTL), Cache Invalidation, Cache Update

ACM Reference Format:
Ziming Mao, Rishabh Iyer, Scott Shenker, Ion Stoica. 2024. Revisiting Cache Freshness for Emerging Real-Time Applications. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696348.3696858

1 Introduction

In-memory caching is widely used to improve application performance by reducing data-access latency and the load on the backend data store. The vast majority of these caches are deployed as lazy or cache-aside caches [4, 25, 26, 28] (shown in Figure 1). In such caches, reads are served from caches, writes are issued directly to the backend data store, and the caches are populated when read misses in the cache.

A key aspect of caching is *data freshness*. Data is fresh within a staleness bound T if a cached object reflects the state of the backend data store (the ground truth) at some point in the last T seconds [27].

The primary technique used to ensure data freshness in caches today is Time-To-Live (TTL) [5, 7, 8, 10, 11, 19–21, 26, 28, 29]. TTLs are typically on the order of minutes to hours [28], and work as follows: whenever a data object is brought into the cache from the data store, a timer of duration T is set. When the timer expires, the object is either (1) re-fetched from the data store or (2) expired and removed from the cache; both of these actions ensure that future reads see a fresh copy from the data store. The main advantage of TTLs is that they are easy to deploy because they need little coordination between the cache and the data store; all freshness decisions can be made using a simple timer local to the cache. This is a primary reason behind the popularity of TTL-based techniques to bound staleness for over two decades [14, 23].

However, the emergence of *real-time* applications with tighter freshness requirements demands new solutions for cache freshness [3]. For example, Databricks' Unity Catalog [16] stores metadata information that requires high data freshness, on the order of seconds. Other examples include serving dynamic web content [2], financial applications (e.g., viewing stock prices) [9, 13], ad bidding [31], and emergency response [9]. These applications have stringent freshness requirements since they typically entail real-time decision-making. For example, a service that provides stock information to analysts requires data to be as fresh as possible to enable real-time financial decisions. A service managing Access Control List (ACL) needs to be fresh to ensure that permissions can be added or revoked immediately.

TTLs introduce prohibitive overhead when applications require data freshness at real-time timescales. This is because, with TTLs, the rate at which additional read requests are made to the backend—either to re-fetch data or due to cache misses that occur when data is expired—is inversely proportional to T since these requests are made each time the timer with duration T expires. This leads to prohibitively high overheads when T is small. This overhead is so large that when designing systems for real-time applications, practitioners are forced to sacrifice caching (and its benefits) for data freshness, preferring to issue reads directly to the database instead.

HOTNETS '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696858

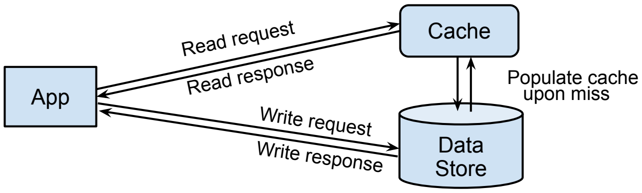


Figure 1: Lazy or cache-aside caches; the predominant way in which in-memory caches are deployed today. In such caches, data freshness is not guaranteed since writes bypass the cache. Servicing a miss can either be initiated by the cache or by the application.

So, we ask the question: *Is it feasible to efficiently provide cache freshness for real-time applications*?

To answer this question, we first develop a simple mathematical model that enables us to better understand the trade-offs presented by different techniques for ensuring cache freshness. We then use this model to show that, *at real-time timescales, making freshness decisions in response to incoming writes is more efficient than TTL-based policies*. Based on this observation, we develop a simple algorithm that adapts to the read-write characteristics of the incoming workload, and show, using simulations, that it has the potential to answer the above question in the affirmative. A salient benefit of our algorithm is that it makes freshness decisions on a *per-object* basis; this ensures that it can be implemented efficiently since it does not require coordinating states across objects.

While providing a theoretical model to reason about cache freshness, our work leaves several system-design questions unanswered. In particular, reacting to writes mandates active coordination between the backend and the cache, a topic that has received little attention due to the near-ubiquitous use of TTLs thus far. We conclude this paper with a set of open research questions that must be answered before real-time freshness can be realized in a practical system.

2 Reasoning About Freshness Quantitatively

We now introduce a simple mathematical model that enables us to quantitatively reason about the trade-offs presented by different techniques for ensuring cache freshness (§2.1). We then validate our model by showing how it can model the overheads of TTL-based policies at various timescales (§2.2).

2.1 The cost of serving fresh data

Since writes bypass caches, cached data is not guaranteed to be fresh. Thus, serving fresh data from the cache incurs cost (in terms of overhead on the infrastructure). We model this cost using two metrics: the freshness cost (C_F), and the staleness cost (C_S). C_F refers to the *throughput* overhead incurred to keep data fresh in the cache. C_F captures the overhead (*e.g.*, compute and network) of sending and receiving messages between the cache and the data store to keep data in the cache fresh: including backend invalidating or updating data in the cache upon write, or the cache fetching fresh data from the backend when a miss occurs due to stale data. C_F aggregates the overhead across different parts of the system into a single metric[1].

The staleness cost (C_S) refers to the *latency* overhead incurred when reading data in the cache that is *not* fresh (i.e., stale). This overhead manifests as increased end-to-end *latency* for clients since stale data causes a request to miss in the cache. As the precise latency is a function of the system implementation, we quantify C_S in terms of the *number of cache misses* that occur when the requested object was present in the cache, but could not be returned since it was stale. C_S is different from miss ratio, which *additionally* considers the misses as a result of reading un-cached data (data that was evicted or never brought into the cache).

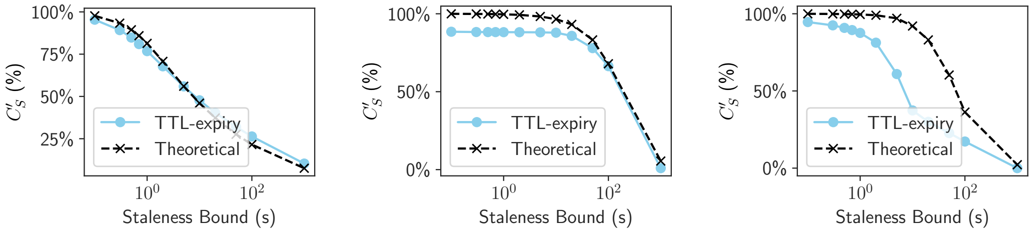
We use C_F and C_S to compare the throughput and latency overheads of different mechanisms to ensure freshness. To calculate C_F and C_S for entire workloads, we make a simplifying assumption that C_S and C_F for different data objects are *independent*, and so C_S and C_F for the entire workload is the sum of C_S and C_F for each object accessed in the workload. This assumption does not strictly hold; for instance, C_S is affected by whether the object is evicted from the cache (which is a function over all objects). However, we find that it allows for a simple formulation of C_S and C_F while providing results that closely match the simulations.

2.2 Why TTLs are no longer sufficient

TTLs are deployed in two forms: TTL-expiry and TTL-polling. In the former, when the TTL expires, the object is invalidated in the cache with the next read incurring a miss. In the latter, when the TTL expires, the object is re-fetched from the data store, to ensure that subsequent reads see fresh data.

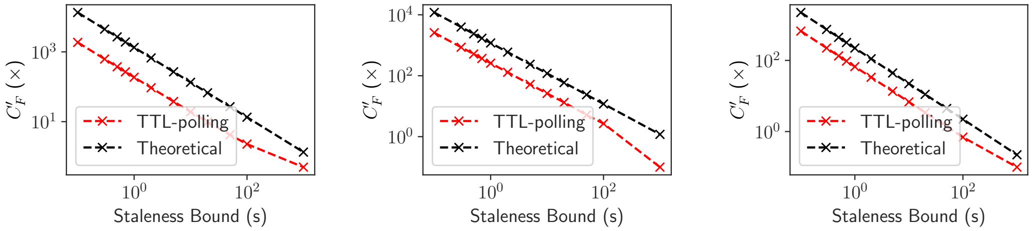
We now evaluate C_S and C_F for the above TTL-based policies. We use the "bounded staleness" definition of freshness introduced in §1: cached data is considered fresh if it reflects all writes made $\geq T$ time ago to the backend data store. Since we assume that C_S and C_F for different objects are independent and additive, we consider each object independently. Let $P_R(T)$, $P_W(T)$ be the probability that there exists at least one read, or one write over an interval T to that object. To calculate $P_R(T)$ and $P_W(T)$, one way is to model the request arrival as a Poisson process with an average rate λ. Like most prior work [2, 22, 30], we assume that individual requests to the object are independent and are reads with a probability of r and writes with a probability of $1 - r$. In this case, $P_R(T) = 1 - e^{-\lambda r T}$ and $P_W(T) = 1 - e^{-\lambda(1-r)T}$.

[1] Since the policy can be implemented differently and across various parts of the system (e.g., the cache, the backend, the load balancer, etc), we chose to aggregate them into a single metric for simplicity. We elaborate on this choice in §3.3.



(a) Poisson **(b) Meta** **(c) Twitter**

Figure 2: Effect of decreasing staleness bound on normalized staleness cost (C'_S, §2). The x-axis is in log scale.



(a) Poisson **(b) Meta** **(c) Twitter**

Figure 3: Effect of decreasing staleness bound on normalized freshness cost (C'_F, §2). Both the x and the y axis are in log scale.

TTL-expiry: To calculate C_S, we consider a time period of T'. Since data is expired every T, the number of misses incurred is 1 per interval T, if there was at least one read request during that interval. Therefore, C_S over a time interval T' is: $C_S = \frac{T'}{T} P_R(T)$. The miss ratio due to reading cached but stale data is the staleness cost divided by the total number of reads over T' (N_R, $N_R = \lambda r T'$ under Poisson): $\frac{C_S}{N_R}$. As $T \longrightarrow 0$, the miss ratio approaches 1. Since TTL-expiry does not need coordination with the backend to keep data in the cache fresh, the only overhead incurred as part of C_F is those to service misses due to stale data. So, $C_F = C_S \times c_m$, where c_m is the overhead incurred upon a miss.

TTL-polling: For this policy, the staleness cost (C_S) is zero. This is because TTL-polling proactively fetches data from the backend when the TTL expires, ensuring that any data present in the cache is never stale. However, this leads to a large C_F. Specifically, C_F over a time T' is $C_F = c_m \times \frac{T'}{T}$. This is because, at the end of each T, the cache must read the fresh value from the backend data store, just as it would during a miss. Once again, as $T \longrightarrow 0$, C_F increases significantly.

To demonstrate how large these overheads can get in practice, and also as a sanity check for our simple model, we perform simulations that measure the freshness and staleness costs. We simulate three workloads; all of which consist of multiple keys with limited cache capacity; to evaluate our assumption about C_S and C_F being additive. The three workloads are a synthetic Poisson workload with $\lambda = 10$ and Zipfian distribution ($s = 1.3$) across keys, and two production workloads from Meta [1, 7] and Twitter [28], respectively.

To give a better idea of how much these overheads matter, we normalize both C_F and C_S. C_F is normalized (C'_F) by the overhead incurred to serve all read requests in the system. Thus C'_F represents the ratio of the wasted cycles to the useful cycles spent serving data in the system. We normalize C_S (C'_S) by the number of reads for which the object requested was present in the cache. Thus C'_S represents the miss ratio caused solely due to reading stale data.

Figure 2 and Figure 3 illustrate the results for TTL-expiry and TTL-polling respectively[2], compared against our theoretical model. We see two clear takeaways: (1) for both policies, our model predicts the overhead with reasonable accuracy, despite our assumption of C_F and C_S being additive and independent (2) the overhead increases to prohibitive amounts as T shrink close to 0. Practitioners today are aware of the latter, and as a result, sacrifice caching (and its benefits) when building systems for applications that require real-time freshness.

3 Design

Our proposed approach is based on the observation that freshness decisions (e.g., whether to expire or re-fetch cached data) are only necessary when the system receives *write* requests. Specifically, to ensure bounded staleness of T for a particular object, the data store and the cache only need to coordinate once per T if one or more write requests to that object were received during the past T, and need not coordinate otherwise.

Based on this observation, we propose an approach that reacts to writes with either *updates* or *invalidates*. An update

[2] We show only C'_S for expiry since C'_F is a simple multiple with c_m. We plot only C'_F for polling since C'_S for polling is zero

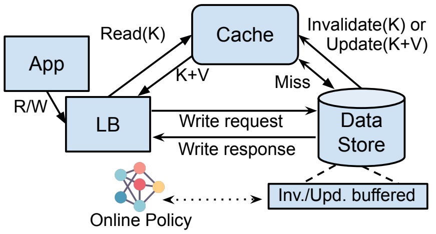


Figure 4: System Overview. Depending on the workload pattern, the policy dynamically decides between invalidation and updates. Invalidates or updates are buffered at the data store and batched over T.

is a message from the backend to the cache that *modifies* an object in the cache to reflect its latest state, and importantly *does nothing* if the object is not in the cache. An invalidate is a message from the backend to the cache that marks a cached object as stale or invalid, causing the following read to be treated as a miss. Updates and invalidates are counterparts to TTL-polling and TTL-expiry; both updates and TTL-polling refresh cached objects, while invalidates and TTL-expiry expire cached objects. The only difference is that updates and invalidates are performed when a write occurs, while the two TTL-based policies are used when the TTL expires.

Figure 4 shows our proposed system architecture. New invalidates or updates over T are buffered and batched at the data store. Depending on the policy (which we will discuss in the rest of the section), the backend either sends out invalidates or updates for the buffered keys every interval of T. Note, that this buffering of writes and sending of updates and invalidates can also be implemented at proxies, not just at the data store.

In the rest of the section, we first use our model to show that updates and invalidates typically lead to lower overheads than TTL-polling and TTL-expiry, respectively to make the case for reacting to writes and not TTLs (§3.1). We then explore the question of how to choose between sending an update or an invalidate upon receiving a write and show that different keys benefit from different decisions (§3.2).

3.1 Reacting to writes versus TTLs

We now calculate the freshness cost C_F and staleness cost C_S for updates and invalidates, and show that they are typically lower than the overheads for TTL-based policies at real-time timescales. To do so, we introduce two additional parameters in our model c_u and c_i, which refer to the overhead of updates and invalidates, respectively. We assume $c_u < c_m$ (i.e., it is cheaper to update than to incur a miss).

Updates are more efficient than TTL-polling. Consider a period T'. Our solution only requires sending one update every T in case of one or more writes during that duration. Hence, since the probability of at least one write over T is $P_W(T)$, the freshness cost of refreshing a single key over T' is $C_F = \frac{T'}{T} P_W(T) \times c_u$. In comparison, C_F for TTL-polling over T' is $c_m \times \frac{T'}{T}$. Since $c_m > c_u$ and $P_W(T) < 1$, we conclude that updates have lower throughput overhead. In terms of C_S, both the above policies proactively keep data fresh, and so $C_S = 0$, making updates more efficient than their TTL counterparts.

Invalidation is more efficient than TTL-expiry. Consider a time period T' broken into multiple intervals of duration T. Consider two consecutive intervals T_0 and T_1 under T', where T_1 follows T_0. Invalidates are batched and sent at the end of T_0. We assume that the backend can track keys that have been invalidated. We elaborate on this assumption in §3.3. This means that if a key k has been invalidated before the next write arrives at the backend, the backend does not need to send a second invalidate.

Let the probability that a key has been invalidated at the end of an interval be p. Under invalidation policy, the C_F is: $\frac{T'}{T}((1-p) \times P_W(T) \times c_i + p \times P_R(T) \times c_m)$. The first term is the expected overhead of an invalidate at the end of T_0 (probability of the key not being invalidated multiplied by the probability of a write multiplied by the cost of an invalidate). The second term is the expected overhead of a miss over T_1 (probability of the key being invalidated multiplied by the probability of a read and multiplied by the cost of a miss). To calculate p, if the key has been invalidated in T_0 and there is a read, the key will be brought into the cache. if the key has not been invalidated in T_0 and there is no write, the key will not be invalidated in T_1. Hence: $p = pP_R(T) + (1 - p)(1 - P_W(T))$. Solving: $p = \frac{P_W(T)}{P_R(T)+P_W(T)}$. If we substitute p into C_F and simplify, we get: $C_F = \frac{T'}{T} \frac{P_R(T)P_W(T)}{P_R(T)+P_W(T)}(c_m + c_i)$. C_S is $\frac{T'}{T} \frac{P_R(T)P_W(T)}{P_R(T)+P_W(T)}$.

We now compare these costs with those for TTL-expiry calculated in §2.2. We notice that C_S for invalidates is strictly lower than C_S for TTL-expiry: $\frac{T'}{T} P_R(T)$. Additionally, we note that for workloads that require real-time freshness, C_F of invalidation is also lower than C_F for TTL-expiry. For example, assuming request arrival is Poisson with $\lambda = 1$ and $r = 0.9$ and $T' = T$, C_F of invalidation is $0.00892(c_i + c_m)$ and C_F of TTL-expiry evaluates to $0.086c_m$, with the latter being significantly higher. However, if workloads consist of mostly writes and not many reads, TTL-miss might be cheaper than invalidation; such scenarios are unlikely as caches are useful for workloads that have reads.

In summary, reacting to writes enables real-time data freshness at lower overheads than TTL-based policies since invalidates and updates incur lower overheads than TTL-expiry and TTL-polling, respectively. However, we notice that invalidation is not *strictly* better than update or vice versa based on C_F and C_S. This raises an interesting question: When should the data store update and when should it invalidate? We provide an initial answer to the question in the following sections.

3.2 Picking between updates and invalidates

The key challenge in picking between updates and invalidates is that their relative costs not only depend on the values of the system parameters (e.g., c_m vs c_u) but also depend on the relative prioritization of the latency and throughput overheads (C_S vs C_F). While the answer is clear in simple scenarios — for instance, if one cares only about minimizing the latency (no matter the throughput cost), one would always send out updates, since they have $C_S = 0$ — it is less clear in more complex scenarios. We now seek to answer this question for two such scenarios—when one seeks to maximize throughput irrespective of latency cost, and when one seeks to maximize throughput for a given latency cost (e.g., an SLO).

Updates vs Invalidates when optimizing throughput. We now describe a simple formula that decides whether to send the cache an update or invalidate upon receiving a write request to minimize the throughput overhead of freshness. We formulate the problem of deriving this formula in the style of classic online algorithms [18]: we denote the gap between our online algorithm and the omniscient policy as G. The goal is for our policy to minimize G. Let k be the probability of an update. $1 - k$ is the probability of an invalidate. $k = 1$ indicates that the policy decides to always update. Again let T_0 and T_1 be two consecutive intervals. Assume invalidates or updates are batched and sent at the end of T_0, and read and write are independent. We have:

- **Interval T_1 has at least a read** (probability: $P_R(T)$). The optimal decision is to do an update with cost c_u.
- **Interval T_1 has no read but has at least a write.** (probability: $(1 - P_R(T))P_W(T)$). The optimal decision is to do nothing with cost 0.
- **If interval T_1 has neither read nor write, consider T_1 skipped.** The intervals (say T_2) following T_1 will incur the same *expected* gap G, down-weighted with probability $(1 - P_R(T))(1 - P_W(T))$.

Each component of G is the probability of an action (k or $1 - k$), times the probability of each of the three cases, and the cost difference to the optimal. Therefore, $G = (1 - k)P_R(T)(c_i + c_m - c_u) + k(1 - P_R(T))P_W(T)c_u + (1 - k)(1 - P_R(T))P_W(T)c_i + (1 - P_R(T))(1 - P_W(T))G$. G is minimized when the coefficient of k is negative: $c_u < \frac{P_R(T)}{P_R(T)+P_W(T)}(c_m + c_i)$. Intuitively, the policy should update if the cost of an update c_u is lower than the cost of an invalidate (right-hand side). If $T \longrightarrow 0$, the above formula reduces to $c_u < r(c_m + c_i)$. This result is surprisingly simple since it tells us that whether to update or invalidate depends only on the read/write ratio of requests to an object. It is *independent* of request rate λ and T when $T \longrightarrow 0$. At small timescales (T comparable to network delay), invalidates or updates have to be sent out immediately. Hence the decision should be independent of the exact values of T and λ.

Maximizing throughput for a latency SLO. System designers rarely optimize throughput in isolation; instead, they typically seek to maximize throughput while meeting a latency target (e.g., an SLO). To address such scenarios, we extend the formulation we just described with an additional constraint to respect a given latency SLO. Since latencies are functions of implementations, we instead use C_S (which represents the misses in the cache due to stale data) as a proxy for latency. Thus, we seek to minimize the throughput overhead (C_F) while meeting an upper-bound on the miss ratio due to staleness.

The staleness cost $C_S = \frac{T'}{T}\frac{P_R(T)P_W(T)}{P_R(T)+P_W(T)}$, the coefficient of c_m in the formula for C_F. The miss ratio due to reading stale data (C_S divided by the total number of reads N_r over T', or under Poisson, $\lambda rT'$) C'_S (first introduced in §2.2) is: $C'_S = \frac{1}{\lambda rT}\frac{P_R(T)P_W(T)}{P_R(T)+P_W(T)}$. If $T \longrightarrow 0$, C'_S reduces to: $1 - r$. Staleness cost can be applied as a constraint from the user. So, if C is the user-specified C'_S constraint: $C'_S \leq C$, the backend chooses to send updates if $(c_i + c_m) \times r > c_u$ or $1 - r > C$, and chooses to send invalidates if otherwise. Once again surprisingly, we see that the choice is *independent* of λ and T when $T \longrightarrow 0$.

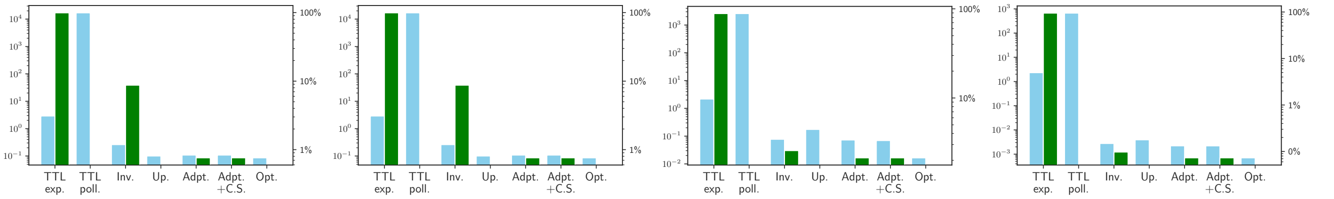
3.3 Realizing the policy in a system

So far the discussion has been over parameters (c_u, c_i, and c_m) – we next discuss preliminary ideas on how we can measure them in practice. Real workloads are often diverse and variable [28]; invalidates or updates likely work in some situations but fall short in others. Therefore, these parameters need to be set adaptively in response to system bottlenecks and different overheads of invalidates and updates per key.

Estimating c_u, c_i, c_m from systems bottlenecks.

To estimate c_u, c_i, c_m, the policy first detects system bottlenecks that may arise from various components such as backend CPUs, caches, and network bandwidth. The policy then decides the cost values to set given the system bottleneck.

We have developed tools to identify systems bottlenecks, such as by measuring backend CPU utilization from `/proc/stat`, network usage from `/proc/net/dev`, and disk I/O usage from `/proc/diskstats`. Users can also label a resource as the bottleneck based on offline profiling, which is often required before deployment. The optimal strategy depends on the nature of the bottleneck. For instance, if the backend CPU or the network bandwidth is the bottleneck, c_u, c_i, and c_m should be set based on either the CPU cycles needed for serialization, or message size. c_u, c_i and c_m should be scaled by the sizes of the actual keys and values. Table 1 illustrates one example of setting c_u, c_i, and c_m where either the cache or backend CPU is the bottleneck. In scenarios where the user prioritizes read latency over throughput or always overprovisions, the policy can set $c_m = \infty$ and only send updates.



(a) Poisson **(b) Poisson (Mix)** **(c) Meta** **(d) Twitter**

Figure 5: Comparison to baselines. Adpt. denotes our proposed adaptive policy. Adpt. + C.S. denotes data store knowing which keys are present in the cache (C.S.). Opt. denotes the optimal policy. The left x-axis is in the log scale. The blue bar indicates C'_F (left axis, in ×), the green bar indicates C'_S (right axis, in %), defined in §2.2. The y-axis is in the log scale.

Parameters	Breakdown
c_m: Miss	Cache: ser(K) + deser(K+V) + update Data Store: deser(K) + read + ser(K+V)
c_i: Invalidation	Cache: deser(K) + delete Data Store: ser(K)
c_u: Update	Cache: deser(K+V) + update Data Store: ser(K+V)

Table 1: An example of c_u, c_i, and c_m where either the compute at the cache or the backend is the bottleneck. `ser` and `deser` refer to serialization and deserialization respectively.

Approximation with $E[W]$, the expected number of writes between reads. From §3.2, while the overhead of invalidate and update depends on $P_R(T)$ and $P_R(T)$, We further introduce a pragmatic formula where we assume $T \rightarrow 0$. The formula in §3.2 can be approximated: we measure $E[W]$, the expected number of writes between reads, and pick invalidate if $E[W]c_u < c_m + c_i$, and update otherwise. To explain, consider a sequence of writes followed by a read. To ensure that the read retrieves the freshest data, an update policy needs to send $E[W]$ number of updates, while an invalidation-based policy only needs to send the first invalidate (by tracking previously invalidated keys), skip sending invalidates for subsequent writes, and incur a miss upon the read. Tracking invalidated keys is feasible because keys are much smaller in size compared to values. The backend can also just track hot keys or recent invalidations. This can be done by simply maintaining a hashmap or storing an extra field in the database. The decision to invalidate or update depends on the relative overhead of the two approaches as decided in the formula.

Estimating $E[W]$ per-key with sketches. We now discuss how $E[W]$ can be estimated per key. Exact $E[W]$ tracking requires three counters per key: C_1 stores the sum of $E[W]$ samples, and C_2 stores the number of $E[W]$ samples. C_3 stores the number of consecutive writes since the last read. To calculate the average $E[W]$, we divide C_1 by C_2. Upon write, we increment C_3. Upon read after a write, we add C_3 to C_1 and increment C_2 by 1. However, the overhead of exact tracking increases linearly with the number of keys and could become prohibitively expensive in practice.

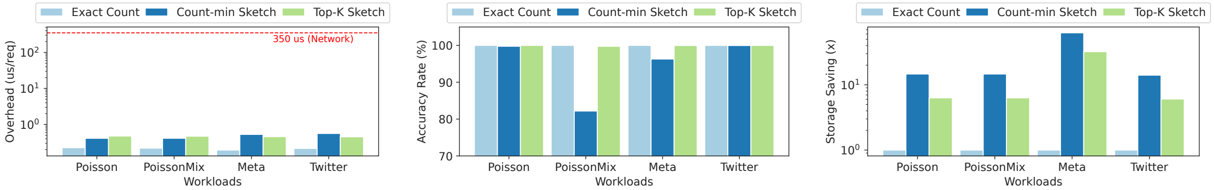
One can lower storage overhead by estimating $E[W]$ with Count-min sketch [15], which approximates read and write counters per key with a 2-D array. $E[W]$ can be estimated by dividing the number of writes by the number of reads. Upon reading or writing, the key passes multiple hash functions (one for each row) and is hashed into different columns of each row. To approximate the count for a key, we similarly hash the key multiple times and calculate the minimum of counters read from different columns that the key is hashed into. However, when the number of keys increases, one might obtain false positives due to hash collision.

To improve accuracy, we propose a modified Top-K sketch for better approximating the number of reads and writes. We keep the *exact count* for Top-K most accessed keys while using the Count-min sketch to *approximate* the count for the rest of the keys. This ensures that we get precise tracking for hot keys. A key can be promoted from Count-min sketch to Top-K if it becomes hot, or demoted from Top-K to Count-min sketch if it becomes cold.

3.4 Evaluation

How well does our policy perform? To evaluate our policy (Adpt.), we repeat the simulations performed in §2.2 with only throughput as the objective. We compare our policy against 6 baselines: TTL-expiry, TTL-polling, always-update (Up.), always-invalidate (Inv.), along with 2 hypothetical policies. Adpt.+C.S assumes that the data store has knowledge of which keys are present in the cache; this enables it to send updates and invalidates only to relevant data objects. Comparing Adpt. with Adpt.+C.S once again evaluates our assumption about being able to evaluate freshness for different keys individually and additively (§2.1). The second hypothetical policy (Opt.) is an omniscient policy that has complete knowledge of both the cache contents and future requests and is optimal.

We evaluate our policy on 4 workloads; 3 from §2.1, and a fourth that contains a 50-50 mix of two Poisson workloads, one that is read-heavy and another that is write-heavy. These



(a) Latency Overhead **(b) Prediction Accuracy** **(c) Storage Saving**

Figure 6: Comparison of Latency, Prediction Accuracy, and Storage Saving across multiple sketches.

workloads occur when sharing a cache across multiple applications, as is common practice today [12]. Figure 5 presents the results. We draw three conclusions: (1) reacting to writes provides significantly lower overheads than TTL-based policies, (2) our policy equals or outperforms naive update and invalidation-based policies. (3) while knowing the cache state can improve the overhead, our assumption about treating individual objects as independent and taking freshness decisions on a per-object basis is largely justified.

How well do sketches approximate $E[W]$ while lowering overhead? Overhead and accuracy of various sketches are presented in Figure 6. Importantly, sketches do not need to determine the precise value of $E[W]$; they only need to decide whether $E[W]c_u < c_i + c_m$ (so it tolerates some inaccuracies in $E[W]$ estimation). We draw three observations: (1) The overhead of looking up $E[W]$ and maintaining the sketches for Top-K sketch and Count-min Sketch is negligible compared to the network delay. (2) Top-K sketch leads to good accuracy in deciding whether to invalidate or update. Count-min sketch can sometimes make wrong predictions. (3) Count-min sketch leads to the largest space saving followed by Top-K sketch. We suggest using the Top-K sketch to track $E[W]$ as it has high accuracy with significant space savings.

4 Related Work

TTL has been widely used [5, 7, 8, 10, 11, 19–21, 26, 28, 29] and studied for in-memory caches: cache eviction [29], estimating MRC [26], modeling hit ratio [22], adaptive TTL for content delivery [6]. However, prior works do not specifically target data freshness, or provide a framework for understanding the overhead of maintaining freshness.

Cache invalidation has been explored: Meta [25] relies on invalidation to bound staleness at a larger timescale. MuCache [32] explores cache invalidation for microservice graphs without blocking other accesses. Recent blog [17] from Meta documents its system Polaris that detects and monitors inconsistencies introduced with cache invalidation. [24, 30] explores whether to invalidate or materialize cached views for web caches. However, to the best of our knowledge, none of the prior works explore a quantitative model of data freshness based on bounded staleness; or an adaptive algorithm deciding between invalidation and update to maintain data freshness with a staleness bound T.

5 Conclusion and Open Questions

In this paper, we conclude that the path to efficient real-time cache freshness is by reacting to writes using updates and invalidates. While we developed a theoretical model to show the potential of such an approach, several key questions remain:

Ensuring guaranteed delivery of updates and invalidates. For TTL, data is guaranteed to expire after a specified time. However, lost or re-ordered updates and invalidates may cause a cached object to remain in a stale state in the cache indefinitely [17]. This problem becomes more challenging in distributed and replicated caches since messages now have to be reliably multi-cast to the target caches. In the presence of resharding or node failures, ownership of keys can change. How to ensure that invalidation or update is propagated to the nodes that own the keys is itself a challenge.

Extending freshness formulation to many-to-many caching relationship. Our algorithm (§3.2) assumes that one cached object can be mapped to one data store object. While this covers many workloads, some cached objects come from multiple reads from the backend data store. For example, the client can cache a web page, which requires rendering multiple data objects from the backend data store, such as figures, HTML fragments, and tables. We believe we can extend our algorithm: a cached object has bounded staleness if its constituent parts satisfy the staleness bound, and C_F and C_S depend on the dependencies of the data read and written.

Combining freshness with eviction decisions. We believe renewed attention to cache freshness will also uncover interesting questions on how to factor freshness decisions into cache eviction algorithms. While prior works have explored leveraging TTLs in eviction [29], it is unclear how invalidation and updates can be co-designed with eviction since eviction algorithms can monitor the current value of the TTL timer, but cannot know when an invalidation or update is likely to arrive.

References

[1] 2024. Evaluating SSD hardware for Facebook workloads. https://cachelib.org/docs/Cache_Library_User_Guides/cachebench-fb-hw-eval. Accessed: 2024-06-22.

[2] Bahman Abolhassani, John Tadrous, Atilla Eryilmaz, and Edmund Yeh. 2021. Fresh caching for dynamic content. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.

[3] Atul Adya. 2023. Pulling Distributed Caches Out of the Dark Ages. https://sky.cs.berkeley.edu/events/sky-seminar-series-atul-adya-databricks-pulling-distributed-caches-out-of-the-dark-ages/. Accessed: 2024-06-22.

[4] Amazon Web Services, Inc. [n. d.]. Caching Best Practices. https://aws.amazon.com/caching/best-practices/. Accessed: 2024-06-14.

[5] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. 2018. Adaptive TTL-based caching for content delivery. *IEEE/ACM transactions on networking* 26, 3 (2018), 1063–1077.

[6] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. 2018. Adaptive TTL-based caching for content delivery. *IEEE/ACM transactions on networking* 26, 3 (2018), 1063–1077.

[7] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768.

[8] Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. 2014. Exact analysis of TTL cache networks. *Performance Evaluation* 79 (2014), 2–23.

[9] Laura Bright and Louiqa Raschid. 2002. Using latency-recency profiles for data delivery on the web. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 550–561.

[10] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. 2019. TTL-based cloud caches. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 685–693.

[11] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. 2020. Elastic provisioning of cloud caches: A cost-aware TTL approach. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1283–1296.

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

[13] James Cipar. 2014. *Trading freshness for performance in distributed systems*. Technical Report. Citeseer.

[14] Edith Cohen, Eran Halperin, and Haim Kaplan. 2005. Performance Aspects of Distributed Caches Using TTL-Based Consistency. *Theor. Comput. Sci.* (2005).

[15] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[16] Databricks. 2024. Unity Catalog: Fine-grained governance and security for your data lakehouse. https://www.databricks.com/product/unity-catalog. Accessed: 2024-09-19.

[17] Facebook Engineering. 2022. Cache Made Consistent: Improving Cache Efficiency and Data Freshness. *Facebook Engineering Blog* (June 2022). https://engineering.fb.com/2022/06/08/core-infra/cache-made-consistent/

[18] Amos Fiat and Gerhard J Woeginger. 1998. *Online algorithms: The state of the art*. Vol. 1442. Springer.

[19] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. 2014. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks* 65 (2014), 212–231.

[20] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 386–400.

[21] Jaeyeon Jung, Arthur W Berger, and Hari Balakrishnan. 2003. Modeling TTL-based Internet caches. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, Vol. 1. IEEE, 417–426.

[22] Jaeyeon Jung, Arthur W Berger, and Hari Balakrishnan. 2003. Modeling TTL-based Internet caches. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, Vol. 1. IEEE, 417–426.

[23] Jaeyeon Jung, Arthur W. Berger, and Hari Balakrishnan. 2003. Modelling TTL-based Internet Caches. In *Proceedings IEEE INFOCOM*.

[24] Alexandros Labrinidis and Nick Roussopoulos. 2003. Balancing performance and data freshness in web database servers. In *Proceedings 2003 VLDB Conference*. Elsevier, 393–404.

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

[26] Sari Sultan, Kia Shakiba, Albert Lee, Paul Chen, and Michael Stumm. 2024. TTLs matter: Efficient cache sizing with TTL-aware miss ratio curves and working set sizes. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 387–404.

[27] Doug Terry. 2013. Replicated data consistency explained through baseball. *Commun. ACM* 56, 12 (2013), 82–89.

[28] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.

[29] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 503–518.

[30] Haobo Yu, Lee Breslau, and Scott Shenker. 1999. A scalable web cache consistency architecture. *ACM SIGCOMM Computer Communication Review* 29, 4 (1999), 163–174.

[31] Shuai Yuan, Jun Wang, and Xiaoxue Zhao. 2013. Real-time bidding for online advertising: measurement and analysis. In *Proceedings of the seventh international workshop on data mining for online advertising*. 1–8.

[32] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2024. {MuCache}: A General Framework for Caching in Microservice Graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 221–238.