# The Cache Sketch: Revisiting Expiration-based Caching in the Age of Cloud Data Management

Felix Gessert, Michael Schaarschmidt, Wolfram Wingerath,
Steffen Friedrich, Norbert Ritter

Database and Information Systems Group
University of Hamburg
{gessert, xschaars, wingerath, friedrich, ritter}@informatik.uni-hamburg.de

**Abstract:** The expiration-based caching model of the web is generally considered irreconcilable with the dynamic workloads of cloud database services, where expiration dates are not known in advance. In this paper, we present the *Cache Sketch* data structure which makes expiration-based caching of database records feasible with rich tunable consistency guarantees. The Cache Sketch enables database services to leverage the large existing caching infrastructure of content delivery networks, browser caches and web caches to provide low latency and high scalability. The Cache Sketch employs Bloom filters to create compact representations of potentially stale records to transfer the task of cache coherence to clients. Furthermore, it also minimizes the number of invalidations the service has to perform on caches that support them (e.g., CDNs). With different age-control policies the Cache Sketch achieves very high cache hit ratios with arbitrarily low stale read probabilities. We present the *Constrained Adaptive TTL Estimator* to provide cache expiration dates that optimize the performance of the Cache Sketch and invalidations. To quantify the performance gains and to derive workload-optimal Cache Sketch parameters, we introduce the YCSB Monte-Carlo Caching Simulator (*YMCA*), a generic framework for simulating the performance and consistency characteristics of any caching and replication topology. We also provide empirical evidence for the efficiency of the Cache Sketch construction and the real-world latency reductions of database workloads under CDN-caching.

## 1 Introduction

In today's cloud data management, most database-as-a-service (DBaaS) systems are exposed through REST/HTTP interfaces. REST APIs make it easy for applications to interact with database services and allow service providers and application designers to leverage the mature, well-researched HTTP protocol and infrastructure. This is particularly true for "NoSQL" systems where the central operations create, read, update, delete (CRUD) map well to REST and HTTP semantics. Yet today, to the best of our knowledge, no DBaaS is capable of exploiting the expiration-based HTTP caching model and its rich, globally distributed content-delivery infrastructure. The reason lies in the impossibility to predict the correct expiration date for database records - any unexpected write operation would entail reads from stale cached copies that did not yet expire. The key insight to the Cache Sketch solution presented in this paper is that the task of cache invalidation can be

shifted from the server to the client using an appropriate data structure.

Caching support in database REST APIs is not only critical for applications that are geographically distributed from the physical location of the database service. It is even more important for mobile and web applications whose performance is governed almost exclusively by latency. Large-scale web sites therefore allocate significant resources to manually optimize caching of static content like images, scripts and style sheets. However, since the recent shift to smarter clients and single-page applications, dynamic database content is increasingly requested in end devices directly, either through data APIs of custom application servers or "backend-as-a-service" systems. This makes data requests extremely latency critical, as they block the user experience. Various studies have quantified the dramatic effects of latency on user satisfaction. For instance, Amazon has found that 100ms of additional latency decrease revenue by 1%. Google similarly discovered that 500ms of additional page load time decrease traffic by 20%[1].

To tackle this problem of significant practical relevance, we introduce a DBaaS caching methodology that employs automatic caching of database records requested through a REST/HTTP API. Cache consistency is ensured using a dual approach: *expiration-based* web caches (browser/device caches, forward proxy caches, ISP caches) are kept coherent through client-side revalidations enabled by the Cache Sketch data structure, whereas *invalidation-based* web caches are invalidated by purge requests issued by the database service. The proposed caching methodology is applicable to any data-serving cloud service, but particularly well-suited for database- and backend-as-a-service systems. The *client Cache Sketch* is a Bloom filter of potentially stale records maintained in the database service. To determine whether a record can safely be fetched from caches, clients query the Cache Sketch before reads. If the record's id is contained in the cache sketch, a revalidation request is sent, as intermediate caches might hold a stale copy. The issued HTTP revalidation request instructs caches to check, whether the database record has a different version than the locally cached copy. If a false positive occurs, a harmless revalidation on a non-stale record is performed, which is similar to a cache miss.

Clients leverage the Cache Sketch for three different goals: fast application and session starts (*cached initialization*), cached reads with consistency guarantees (*bounded staleness*) and low-latency transactions (*conflict-avoidant optimistic transactions*). For *cached initialization*, clients transparently store every fetched record in the client cache (usually the browser cache). At the begin of a new session or page visit, the Cache Sketch is transferred, so clients can check which cached copies from the last session are still up-to-date. The number of necessary requests is thus reduced to the cache miss ratio of intermediate caches. To maintain $\Delta$-*bounded staleness*, the Cache Sketch is refreshed in intervals of $\Delta$. The interval constitutes a controllable upper bound on the staleness of loaded records. Similarly, *conflict-avoidant optimistic transactions* load the Cache Sketch at transaction begin. Subsequent transactional reads exploit cached records, reducing the overall duration and associated abort probability of the transaction.

By optimistically caching all records and employing the Cache Sketch to only revalidate stale records, the same cache hit ratio is achieved as if the time to the next write was known

[1] see http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx (12/12/2014) for both claims

in advance. A record can only be removed from the Cache Sketch once it is sure to have been expired in all caches. Thus, precise estimations of expiration times impact cache hit ratios after writes as well as the number of necessary invalidations. To tune the inherent trade-off between cache hits, stale reads, invalidations and false positives towards a given preference, we present the *Constrained Adaptive TTL Estimator* (CATE) that complements the Cache Sketch by adjusting cache expiration to optimize the trade-off.

The contributions of this paper are threefold:

- We propose the **Cache Sketch** as a data structure to enable the use of expiration- and invalidation-based web caching for cloud data management systems to combine the latency benefits of caching with rich consistency guarantees.
- We describe the Constrained Adapative TTL Estimation (**CATE**) algorithm that computes record expiration dates to minimize stale read probabilities and invalidations while maximizing cache hits.
- We present the Monte-Carlo caching simulation Framework **YMCA** that allows to analyze and optimize caching strategies and Cache Sketch parameters for pluggable network, database and caching topologies.

The paper is structured as follows. Section 2 presents the Cache Sketch and its properties and effects. Section 3 outlines the TTL estimation problem and a possible solution. Section 4 introduces the YMCA simulation frameworks and presents simulated and real empirical results for the proposed combination of web caching and cloud data management. Sections 5 examines related work and Section 6 concludes.

## 2 Staleness Avoidance through Cache Sketches

The expiration-based caching model of HTTP was deliberately designed for scalability and simplicity. It therefore lacks cache coherence protocols and assumes a static TTL (time to live) indicating the time span for which a resource is valid, allowing every cache to keep a copy. This model works well for immutable content, for example a particular version of JavaScript library. With the rise of REST APIs for cloud services however, this model fails in its naive form - TTLs of dynamic content, in particular database records and query results are not known in advance. This has lead to database interfaces that forbid caching in the first place as otherwise staleness would be uncontrollable.

Figure 1 shows an architectural overview of how our Cache Sketch approach addresses this problem. Every cache in the request path serves cached database records requested by their key to the client, which can either be an end-user's browser, a mobile application or an application server. The Bloom filter of the client Cache Sketch is queried to send a request either as normal request (record not contained) or a revalidation (record contained). The revalidation forces caches to update their copy using an HTTP request conditioned over the record's version (*Etag*).

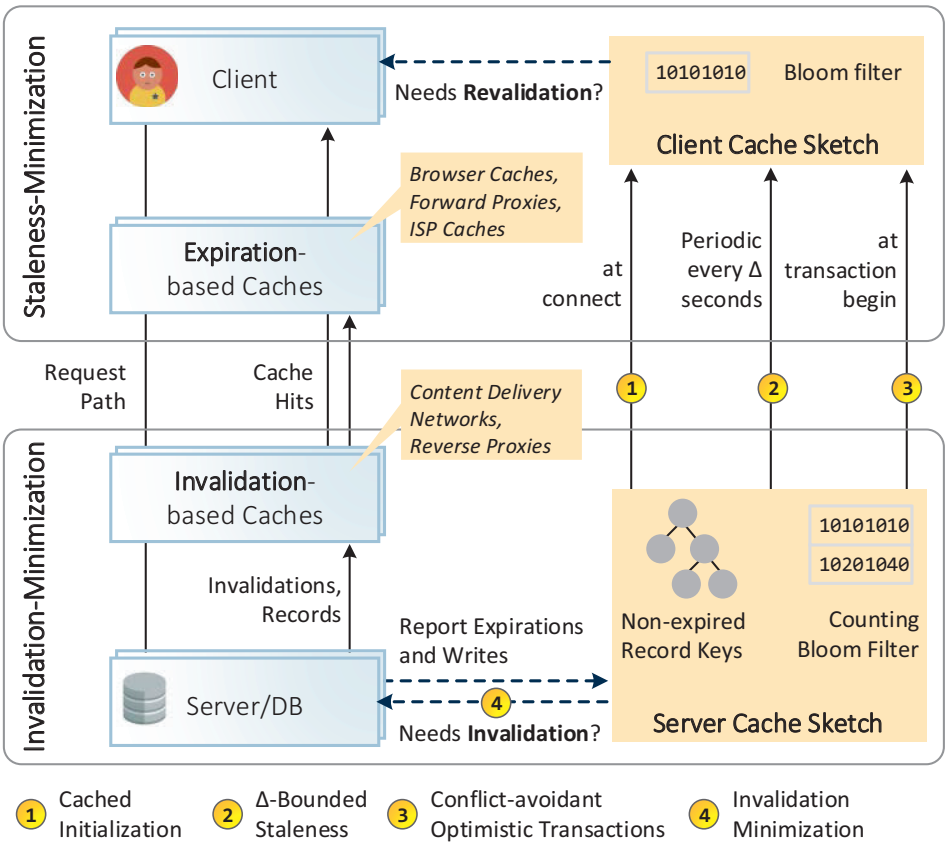



Figure 1: Architectural overview of the client and server Cache Sketch.

The database service tracks the highest TTLs provided at a cache miss. On a subsequent write, the record is added to the Counting Bloom filter of the server Cache Sketch and removed when the record is expired. The database service is furthermore responsible for purging records from invalidation-based caches (CDNs and reverse proxy caches), which allows them to answer revalidations. To minimize invalidation broadcasts, purges are only sent, if the server Cache Sketch reports a record as non-expired. It is important to note, that this scheme does not require any modifications of the HTTP protocol or web cache behavior. The proposed Cache Sketch approach is not specific to a particular database service architecture and can be realized either directly in the nodes of database system or as a tier of stateless REST servers exposing the database. We chose the latter approach, building on the database-independent ORESTES middleware [GBR14].

There are several advantages of caching database records close to clients. First, cache hits have lower latency and higher throughput than uncached requests, as TCP throughput is inversely proportional to the round-trip time [Gri13]. Second, the database service is

under lower load, as it only has to handle write requests and cache misses. Third, clients profit from requests of other clients, as all caches except the browser/device cache are shared. Fourth, flash crowds, i.e. load spikes caused by unexpected and sudden popularity, are mitigated by caching and do not bring down the database service [FFM04]. Fifth, temporary unavailability of the database service can be hidden for reads by letting CDNs and reverse proxies serve cached records while the service is unreachable.

### 2.1 The Client Cache Sketch

For each potentially non-expired record $x$, the client Cache Sketch has to contain its key $key_x$. For now, we will only consider key/id lookups - the most common access pattern in key-value, document and wide-column stores - and discuss how the scheme can be extended to query results, later. As shown in Figure 2, a read on a key is performed by querying the Bloom filter $b_t$ of the client Cache Sketch $c_t^c$ that was generated at time $t$. The key is hashed using $k$ independent uniformly distributed hash functions that map from the key domain to $[1, m]$, where $m$ is the bit array size of $b_t$. If all bits $h_1(key), \ldots, h_k(key)$ equal 1, the record is contained and has to be considered stale.

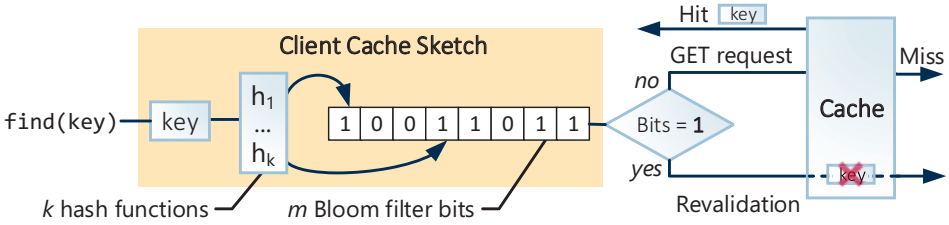



Figure 2: Database read using the Cache Sketch.

Theorem 1 deducts the central guarantee offered by the Cache Sketch using the time-based consistency property $\Delta$-atomicity [GLS11]. $\Delta$-atomic semantics assert that every value becomes visible during the first $\Delta$ time units after the acknowledgement of its write.

**Theorem 1.** *Let $c_t^c$ be the client Cache Sketch generated at time $t$, containing the key $key_x$ of every record $x$ that was written before it expired in all caches, i.e. every $x$ for which holds that $\exists\, r(x, t_r, TTL), w(x, t_w) : t_r + TTL > t > t_w > t_r$ where $r(x, t_r, TTL)$ is a read (cache miss) on $x$ at time $t_r$, $TTL$ is the TTL provided for that read and $w(x, t_w)$ is a write on $x$ at time $t_w$.*

*A read on record $x$ performed at time $t_x$ using $c_t^c$ satisfies $\Delta$-atomicity with $\Delta = t_x - t$, i.e. the read is guaranteed to see only records that are at most $\Delta = t_x - t$ time units stale.*

*Proof.* Consider there was a read issued at time $t_x$ using $c_t^c$ that returned a record $x$ that was stale for $\Delta > t_x - t$. This implies that $x$ must have been written at a time $t_w < t$ as otherwise $\Delta < t_x - t$. Now, if there has been no previous read $r(x, t_r, TTL)$ with

$t_r + TTL > t_x$ then $x$ could not have been stale for at least $t_x - t$ time units. If there has been such a read, by the construction of $c_t^c$ the record's key would have been contained in it and the read (a revalidation) could not have been stale. □

Theorem 1 states, that clients can tune the desired degree of consistency by controlling the age $\Delta$ of the Cache Sketch: the age directly defines $\Delta$-atomicity. This guarantee relies on the linearizability of the underlying database system, i.e. writes are assumed to be directly visible to uncached reads. If the database is only eventually consistent with $\Delta_{db}$-atomicity, the guarantee is weakened to $(\Delta + \Delta_{db})$-atomicity[2]. Similarly, if the invalidation-based caches only support asynchronous invalidations (which is typical for real-world CDNs [PB08]) with $\Delta_c$-atomicity, the consistency guarantee becomes $(\Delta + \Delta_c)$-atomicity[3]. If $\Delta_c$ is an undesired source of uncertainty, $\Delta$-atomicity can be established in two ways. First, invalidation-based caches can be treated as pure expiration-based caches by not letting them answer revalidation requests. The trade-off is that this increases read latency and the load on the database service. Second, invalidations can be performed synchronously. This is a good option for reverse-proxy caches located in the network of the database service. Here, the trade-off is that cache misses have higher latency and can be blocked by the unavailability of a cache node (no partition tolerance).

**Definition 1.** *A client follows **cached initialization**, if all initial reads are performed using the freshly loaded Cache Sketch $c_t^c$. A read at $t_{now}$ follows **$\Delta$-bounded staleness**, if it only uses $c_t^c$ if $t_{now} < t + \Delta$. A **conflict-avoidant optimistic transaction** started at $t_s$ uses $c_{t_s}^c$ for transactional reads.*

Definition 1 introduces three client-driven age-control techniques for the Cache Sketch. *Cached initialization* builds on the insight, that initially $\Delta = 0$ for a Cache Sketch piggybacked upon connection. This implies, that every cached record can be used without degrading consistency, i.e. loading the Cache Sketch is equivalent to loading all initially required records in bulk, which may also include all static resources (images, scripts, etc.) of the application or website.
$\Delta$-*Bounded staleness* guarantees $\Delta$-atomicity by not letting the age of the Cache Sketch exceed $\Delta$. Updates may be performed *eagerly* or *lazily*. With eager updates, the client updates $c_t^c$ in intervals of $\Delta$. As this may incur updates despite the absence of an actual workload, lazy updates only fetch a new Cache Sketch on demand. To this end, if a read request is issued at $t_{now} > t + \Delta$, the request is turned into a revalidation instructing the service to append the Cache Sketch to the result. Hence, at the mild cost of a cache miss every $\Delta$ time units, $c_t^c$ is updated without additional requests.
Similar to cached initialization, a *conflict-avoidant optimistic transaction* (COT) is started by loading the Cache Sketch. The caching model is only compatible with optimistic transactions as reads are performed in caches which cannot participate in a lock-based concurrency control scheme. By having clients collect the read-sets of their transactions consisting of record ids and version numbers, the database service can realize the transaction

[2] Bailis et al. [BVF+12] have extensively studied the staleness of Dynamo-style systems. They found, that with high probability $\Delta_{db}$ is very low and for many configurations not perceivable at all.

[3] We are not aware of any scientific studies on CDN purge latencies. Anecdotally, the Fastly CDN used in our evaluations employs the bimodal multicast protocol for invalidations with measured latencies typically much lower than 200ms: fastly.com/blog/building-fast-and-reliable-purging-system

validation using BOCC+, as shown in [GBR14]. The important alteration that COTs bring to this scheme is that cached reads can drastically reduce the duration $T$ of the transaction, while the Cache Sketch limits staleness to $T$. Since the abort probability of optimistic transactions has been shown to grow exponentially with $T$ [Tho98], lowering $T$ through cache hits can greatly reduce abort rates[4].

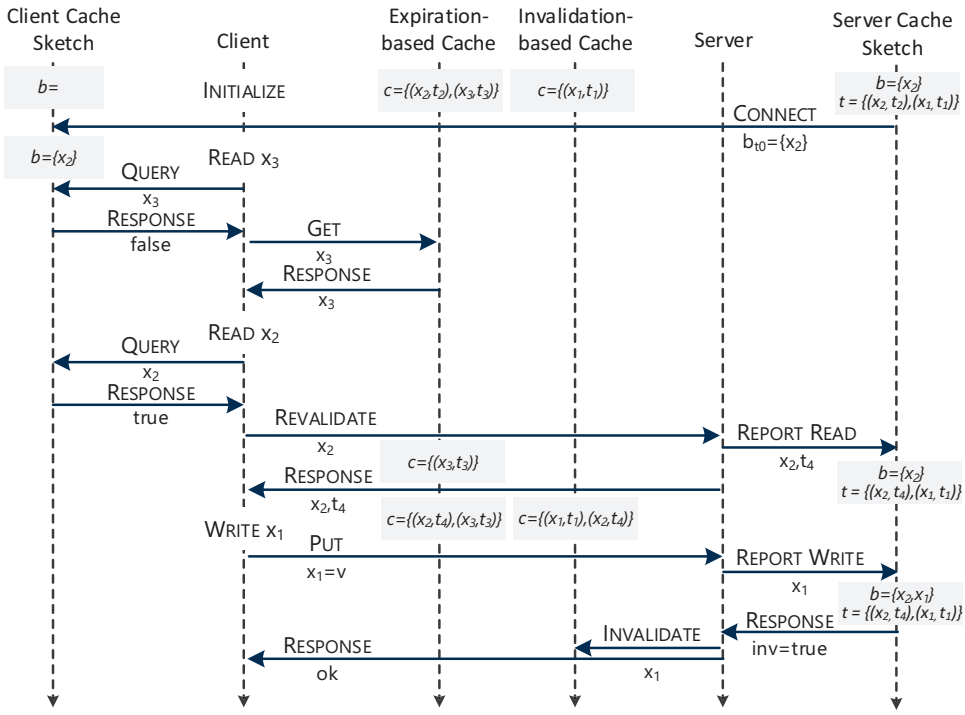



Figure 3: An end-to-end example of the proposed Cache Sketch methodology.

Figure 3 shows an end-to-end example of Cache Sketch usage. First, the client fetches the Cache Sketch. As $x_3$ is not contained in it, the record is fresh and hence requested normally, resulting in a cache hit. The next record $x_2$ is contained and hence a revalidation is sent, causing the expiration-based cache to evict its cached copy. The server returns $x_2$ with a new TTL/expiration date $t_4$, which is saved in both caches. Additionally, the new expiration date is also reported to the server Cache Sketch, where expiration state is tracked. On the subsequent write on $x_1$, the server Cache Sketch adds $x_1$ to the Bloom filter, since its expiration date $t_1$ has not yet passed. This also tells the server, that invalidation-based caches need to be purged. Any later readers are therefore able to revalidate $x_1$ from an invalidation-based cache.

[4]We skip many details here. An extensive quantitative investigation is an important part of our future work.

### 2.2 The Server Cache Sketch

The purpose of the server Cache Sketch $c_t^s$ is the efficient and correct generation of client Cache Sketch defined in Theorem 1. This requires two important capabilities the client Cache Sketch lacks. First, the server Cache Sketch must support *removal* of keys in order to evict expired items. Second, it must support *invalidation queries* which report, whether a write has to be propagated as an invalidation.

**Definition 2.** *The server Cache Sketch $c_t^s$ consists of a Counting Bloom filter $cb_t$ containing all elements of $c_t^c$ and a mapping of keys to their maximum expiration date $e = \{(k_i, t_i) | max_{t_i = t_r + TTL}(r(x, t_r, TTL) \wedge t_i > t_{now})\}$. When $x$ is updated or deleted, $k_x$ is added to $cb_t$ iff $k_x \in e$. Similarly, an invalidation is only necessary, if $k_x \in e$.*

The employed Counting Bloom filter [BMM02] is an extension of the Bloom filter that allows removals and can be implemented to materialize the corresponding normal Bloom filter, so retrievals of $c_t^c$ do not require any computation. To make the retrieval of the Cache Sketch efficient, the size $m$ of the Bloom filter must be chosen carefully. The false positive rate $p$ is determined by the size $m$ of the bit vector, the number of inserted elements $n$ and the number of hash functions $k$: $p \approx (1 - exp(-kn/m))^k$. The optimal number of hash functions is $k = \lceil ln(2) \cdot (n/m) \rceil$, giving the size as $m = -n \cdot ln(p)/ln(2)^2$.

A simple model is to choose $m$ so that transferring $c_t^c$ only requires a single round-trip, even at connection startup. This is achieved, if the message size of $m$ bits (and some HTTP metadata) measured in TCP segments of 1,460 bytes does no exceed the initial TCP congestion window size 10, i.e. $m \approx 10 \cdot 1460\, byte = 116800\, bit$. For a false positive rate $p \leq 0.05$, the filter could hence contain up to $n \approx 18732$ distinct records. If $n$ increased to 50000, $p$ would grow logarithmically to $p = 0.326$. If the Bloom filter is only transferred over an already established connection (e.g., after loading an HTML page), it can be significantly larger without incurring an additional round-trip[5].

The server Cache Sketch represents shared state between all server nodes. It lies in the critical request path as read, update and delete operations require modifying it. Previously, we assumed a single $c_t^s$ of every tenant's database. As a generalization, $c_t^s$ can be partitioned and replicated based on tables (resp. buckets, collections, classes) by maintaining a separate $c_t^s$ for each table. This solves two problems. First, updates to the Cache Sketch scale horizontally, mitigating potential write bottlenecks. Second, if an aggregate Cache Sketch for all tables is too large, clients can opt to fetch the Cache Sketch only for the required tables. To expose the aggregate Cache Sketch, the database service assembles the Cache Sketch by performing a union over the respective Bloom filters, which is a simple bitwise OR over their respective bit vectors [BMM02].

To extend the Cache Sketch approach from cached records to cached query results, each cacheable query $q$ has to be identified by a key $k_q$ (e.g., a concatenation of the query and its parameters), so it can be tracked in $c_t^s$. The database service then evaluates for each record update, whether a matching query $q$ exists and treats it like an update to $q$ itself[6]. An

[5] This is an effect of the TCP slow-start algorithm which continuously increases the congestion window.

[6] From an implementation perspective, this could for instance be achieved through distributed real-time processing frameworks (e.g., Storm, S4, Samza) or well-known techniques for materialized view maintenance.

alternative approach to query caching is to represent query results as lists of record keys, which can then individually profit from caching. The trade-off between both approaches are lower latency of cached results opposed to higher overall hit rates and reuse between queries. Combining both approaches in a single method is part of our future work.

### 2.3 Quantifying $(\Delta, p)$-Atomicity for the Web Caching Model

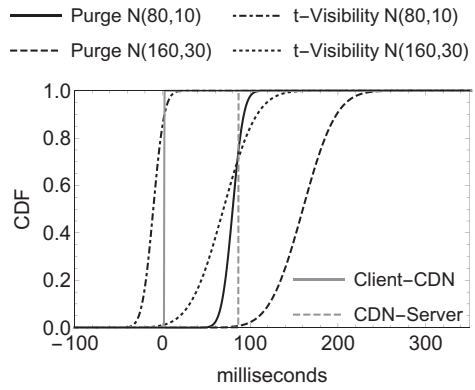
For consistent databases, the Cache Sketch guarantees $(\Delta+\Delta_c)$-atomicity, where $\Delta_c$ is the upper bound for the staleness of records read from invalidation-based caches. $\Delta_c$ largely overestimates staleness, since access is often local to geographic regions and seldomly governed by worst-case delays. We therefore refine $\Delta_c$ to $(\Delta_c, p)$-atomicity[7], which a read satisfies if it is $\Delta_c$-atomic with probability $p$ [BVF+12]. The probability $p$ for $(\Delta_c, p)$-atomic semantics can be expressed through the round-trip latencies $T_{cc}$ (client-cache), $T_{sc}$ (server-cache) and $T_i$ (invalidation). A revalidation or cache miss hitting an invalidation-based cache is $\Delta_c$-atomic, if the time for the corresponding write acknowledgement to travel back to the sender plus the time for the read to reach the cache subtracted from the invalidation latency is smaller than $\Delta_c$:

$$p = Pr[T_i - (T_{sc}/2 + T_{cc}/2 + T_{cc}/2) \leq \Delta_c] \tag{1}$$

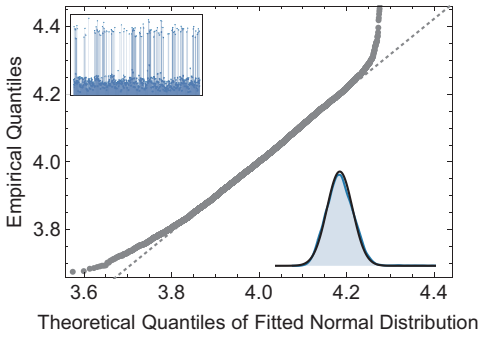
We gathered real-world latency traces to quantify $(\Delta_c, p)$-atomicity and to feed our later simulations with realistic assumptions. The setup consists of a client located in the Amazon EC2 California region, a server in EC2 Ireland and the Fastly CDN as an example of an invalidation-based web caching system. We derived maximum-likelihood distribution fits for $T_{cc}$ and $T_{sc}$ for different continuous distribution families as shown in Figure 4b and 4d, after applying the Tukey-outlier criterion to account for measurement noise, such as the the noisy-neighbor problem. Though there is consensus in the networking literature that in the general case, network delays cannot be modeled using a single distribution [VM06], the normal and Gamma distribution yield good fits for the described setup (goodness-of-fit p-values 0.21 and 0.68 with the Cramér-von Mises test). This is illustrated in the QQ-plot in Figure 4c, which shows that apart from the tails of the raw data (with outliers), the normal distribution describes $T_{cc}$ very accurately.

Based on this data, $(\Delta_c, p)$-atomicity can be computed as shown in Figure 4a with $T_{cc}/2 \sim N(2.00, 0.06)$ and $T_{sc}/2 \sim N(86.54, 0.06)$ for two $T_i$ distributions. For $T_i \sim N(80, 10)$, which we found to be a good upper bound in our experiments, the probability of reading a fresh value starts high and quickly converges to 1. For caches located nearer to the server, $p$ would converge even faster. In conclusion, with asynchronous invalidations exhibiting $(\Delta_c, p)$-atomicity, the Cache Sketch guarantees $(\Delta+\Delta_c, p)$-atomicity. This allows precise reasoning about the consistency trade-off for a given scenario of latency distributions and eases the decision on whether invalidations should be allowed to be asynchronous.

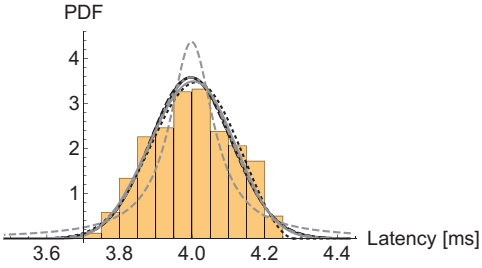
[7] $(\Delta, p)$-atomicity is also referred to as $t$-Visibility.



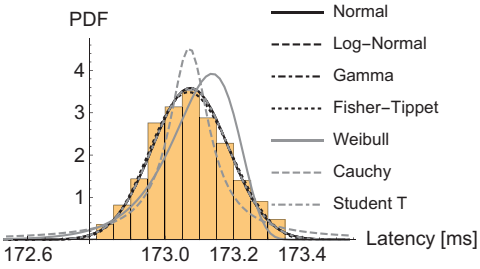

(a) $(\Delta_c, p)$-atomicity in the web caching model.




(c) QQ-Plot for the normal fit of $T_{cc}$.




(b) Distribution of client-cache latency $T_{cc}$.




(d) Distribution of server-cache latency $T_{sc}$

Figure 4: Analysis of exemplary latencies and their effect on $(\Delta_c, p)$-atomicity.

## 3 Optimal Cache Expiration

The TTL for which caches are allowed to store records significantly affects cache hits, stale reads, invalidations and false positives in the Cache Sketch. For instance, records that experience a write-only workload but are cached with large TTLs would hurt performance, as each write would entail an unnecessary invalidation. Likewise, read-heavy records would suffer from small cache hit ratios, if assigned TTLs are too small. The usefulness of the Cache Sketch depends on its false positive rate. Therefore, we introduce the concept of *TTL estimators* which try to minimize costs.

**Definition 3.** *A TTL estimator* $E(id, \lambda_m^{id}, \lambda_w^{id}) \rightarrow TTL_{id}$ *maps a record's historic cache miss rate* $\lambda_m^{id}$ *and write rate* $\lambda_w^{id}$ *to a TTL that minimizes the cost function:*

$$cost = w_1 \cdot \frac{\#cachemisses}{\#ops} + w_2 \cdot \frac{\#invalidations}{\#ops} + w_3 \cdot \frac{\#stalereads}{\#ops} + w_4 \cdot p$$

The cost function is parameterized by weights $w_i$ that express the relative severity of each condition. For example, in a setup with a slow single server, many invalidation-based

caches and an application with low consistency requirements, $w_1$ and $w_2$ would be large to protect the server, while $w_3$ and $w_4$ would be smaller.

The estimator is invoked for every cache miss to decide on the next TTL. As a baseline, we propose the *Static Estimator* $E_{static}(id) = TTL_{max}$ that always minimizes cache miss costs through a high static TTL. The trade-off is, that every write on record $x$ happening $t$ seconds before the expiration causes an invalidation, opens the possibility of a stale read caused by the asynchronous invalidation and forces the cache sketch to contain $x$ for the remaining $t$ seconds, increasing its false positive rate. This implies, that the static estimator should only be employed if the Cache Sketch is large enough to hold all records that might be updated in a time window of $TTL_{max}$. A straight-forward improvement is thus obvious: instead of always estimating very large TTLs, TTLs should rather be correlated to the expected time to the next write on a record. Furthermore, TTLs should also be lower if the workload is write-dominant and higher if it is read-dominant.

To make the improved TTL estimation feasible, some assumptions have to be made. First, we assume, that the per-record workloads are readily available to estimators in the form of cache-miss rates $\lambda_m^{id}$ and write rates $\lambda_w^{id}$. Second, to estimate the probability of writes in certain time intervals, a continuous-time stochastic process of writes $\{W(t), t \in T\}$ is assumed where the random variables $X(t)$ model the amount of writes seen until time $t$. Intuitively, given that exactly one write happened in the interval $[0, t]$, the time of occurence should be uniformly distributed over $[0, t]$. This requirement is met by the *Poisson process*, which is the most commonly used stochastic process for modeling arrival processes. It is characterized by increments that follow a Poisson distribution $Pr[W(t+s) - W(s) = k] = (\lambda_w t)^k / k! e^{-\lambda_w t}$, where $\lambda_w$ is the write rate, i.e. the expected amount of writes in a time interval of length $t$ is $E[W(t)] = \lambda_w t$. A very central property for our TTL estimation problem is, that inter-arrival times between writes $T_w$ are exponentially distributed with mean $1/\lambda_w$, i.e. $Pr[T_w < TTL] = 1 - e^{-\lambda_w TTL}$. This implies, that knowing an record's write rate is sufficient information to derive the expected time of the next write $E[T_w] = 1/\lambda_w$ and the quantiles $Q(p, \lambda_w) = -ln(1-p)/\lambda_w$. As the stochastic process of reads is unobservable (hidden through caches), we specifically do neither require knowledge about the *workload mix*, i.e. record-specific read-write ratios nor the *popularity distribution*, i.e. the underlying distribution of record access frequencies. Instead, the TTL estimator implicitly adapts to these conditions.

### 3.1 Constrained Adaptive TTL Estimation

The goal of the constrained adaptive TTL Estimator (CATE) is to minimize the cost function, while constraining the size of the Cache Sketch to meet a good false positive rate. To this end, CATE adapts TTLs to the cache miss rate $\lambda_r$ and write rate $\lambda_w$ instead of merely estimating the time to the next write. The estimation approach is illustrated in Figure 5a: write and cache miss metrics are aggregated in the server and fed into the estimator for each cache miss to retrieve a new TTL. The algorithm is based on four design choices:

1. *Read-only* records yield $TTL_{max}$ and *write-only* records are not cached.

2. If the miss rate $\lambda_m$ equals the write rate $\lambda_w$, the record should be cached for its expected lifetime expressed by the *interarrival time median* of writes $Q(0.5, \lambda_w)$, i.e. the TTL is chosen so that the probability of a write before expiration is 50%.

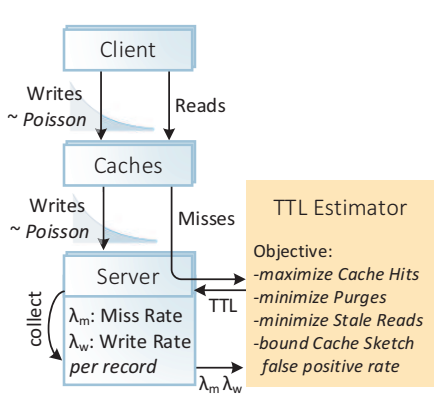
3. A *ratio function* $f\colon \mathbb{R} \to [0,1]$ expresses, how the miss-write ratio impacts the estimated TTLs. It maps the imbalance between misses and writes to $p_{target}$ which gives the TTL as the quantile $Q(p_{target}, \lambda_w)$. If for instance misses dominate writes, $p = 0.9$ would allow a 90% chance of a write before expiration, in order to increase cache hits. Using quantiles over TTLs for the ratio function has two advantages. First, the probability of a write happening before the expiration is easier to interpret than an abstract TTL. Second, the quantile scales with the write rate. The ratio function and its parameters can be tuned to reflect the weights in the cost function.

4. *Constraints* on the false positive rate of the Cache Sketch and the number of invalidations per time period are satisfied by lowering TTLs.

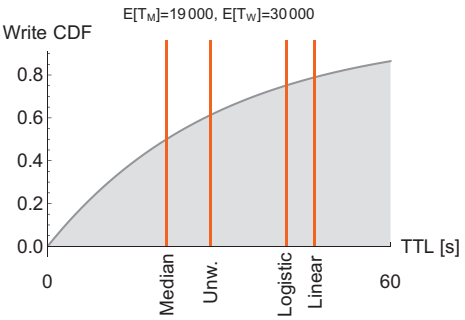
**Algorithm 1** Constrained Adaptive TTL Estimation (CATE)

**procedure** ESTIMATE($\lambda_m$ : miss rate, $\lambda_w$ : write rate) $\rightarrow TTL$
  **constants:** $TTL_{max}$, $slope$, $f$ : ratio function
  **if** $\lambda_w = NIL$ **then return** $TTL_{max}$
  $$imbalance = \begin{cases} \lambda_m/\lambda_w - 1 & \text{if } \lambda_m \geq \lambda_w \\ -(\lambda_r/\lambda_w - 1) & \text{else} \end{cases}$$
  $p_{max} \leftarrow Pr[T_w < TTL_{max}] = (1 - e^{-\lambda_w TTL_{max}})$
  **if** $f$ **is** linear **then** $p_{target} \leftarrow 0.5 + slope \cdot imbalance$
  **else if** $f$ **is** logistic **then** $p_{target} \leftarrow p_{max}/(2p_{max} \cdot e^{-slope \cdot imbalance})$
  **else if** $f$ **is** unweighted **then** $p_{target} \leftarrow \lambda_m/(\lambda_m + \lambda_w)$
  **if** Cache Sketch capacity exceeded **then**
    Decrease $p_{target}$ by a penalty proportional to false positive rate
  **if** Invalidation budget exceeded **then**
    Decrease $p_{target}$
  $$TTL = \begin{cases} 0 & \text{if } p_{target} \leq 0 \\ TTL_{max} & \text{if } p_{target} \geq p_{max} \\ Q(p_{target}, \lambda_w) & \text{else} \end{cases}$$
  **return** $TTL$

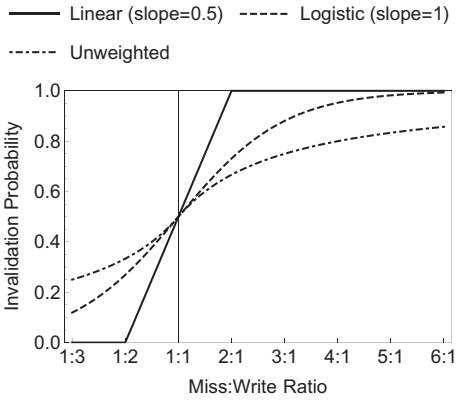
Algorithm 1 describes CATE. The ESTIMATE procedure is invoked for each cache miss. It requires three constants: the maximum TTL $TTL_{max}$, the ratio function $f$ and the $slope$ which defines how strongly $f$ translates the imbalance between misses and writes into smaller or greater TTLs. First, the miss-write $imbalance$ is calculated. We define it to be 0 if $\lambda_m = \lambda_w$, $x$ if $\lambda_m$ is $x$ times greater than $\lambda_w$ and $-x$ if $\lambda_w$ is $x$ times greater than $\lambda_m$. Next, the ratio function maps imbalance to the allowed probability $p_{target}$ of a write (and invalidation) before the expiration date. $p_{target}$ is capped at $p_{max} = Pr[T_w < TTL_{max}]$,



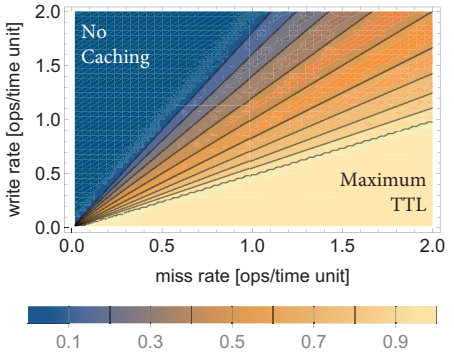

(a) The TTL estimation process.




(b) TTL estimations for an example workload.




(c) Ratio functions.




(d) $p_{target}$ contour plot for linear ratio function.

Figure 5: Constrained Adaptive TTL Estimation.

so that the estimated TTL never gets larger than $TTL_{max}$. We consider three types of ratio functions shown in Figure 5c: a *linear* and a *logistic* function of the imbalance, as well as the *unweighted* fraction of misses in all operations.

In order not to overfill the Cache Sketch, its current false positive rate is considered. If it exceeds a defined threshold, $p_{target}$ is decreased to trade invalidations on non-expired records against revalidations on expired records. By lowering the probability of writes on non-expired records, Cache Sketch additions decrease, too. Invalidations are treated similarly: if the budget of allowed invalidations is exceeded, $p_{target}$ is decreased. This way, Cache Sketch additions and invalidations are effectively rate-limited. Optimal decrements depend on the severity $a$ of a violation and can be computed as $p_{target} = p_{target} * (1-f)^a$, where $f$ is the degree of violation, for example the difference between the allowed and actual false positive rate. Last, the TTL derived as the quantile $Q(p, \lambda_w)$ is returned.

Figure 5b gives an example of estimated TTLs for a read-heavy scenario, as well as the corresponding probability $Pr[T_w < TTL]$ of a write before expiration. By construction,

all three ratio functions yield a TTL that is higher than the median time between two writes in order to drive cache misses down. The magnitude of this TTL correction is determined by the ratio function and its slope. This makes it obvious, that minimizing the cost function requires tuning of the ratio function in order to meet the relative weights between misses, invalidations, stale reads and false positives. As finding the right $TTL_{max}$ and $slope$ in running system is a cumbersome, manual and error-prone process, we introduce a framework in Section 4 that chooses parameters using Monte-Carlo simulations to find the best solution under a given workload and error function. Figure 5d shows the effect of different miss- and write rates as a contour plot of the linear ratio function. In the upper left area, writes clearly dominate misses, so the estimator opts to not cache the record at all - frequent invalidations would clearly outweigh seldom cache hits. In the bottom right area on the other hand, misses dominate writes, so the record is cached for $TTL_{max}$. The area in between gradually shifts to higher TTLs (values of $p_{target}$), with the steepness of the ascent varying with the slope.

As explained above, estimating TTLs requires each database service node to have approximations of write and miss *access rates* for each record. To this end, inter-arrival times are monitored and averaged over a time window using a simple moving average (SMA) or exponentially-weighted moving average (EWMA). The space requirements of the SMA are high, as potentially many arrival times for each record have to be tracked, whereas the EWMA only requires a single value. If the space requirements are still too high, sampling is applied. More specifically, exponentially-biased reservoir sampling is an appropriate stream sampling method that prefers newly streamed values over older ones. The reservoir is a fixed-size stream sample, i.e. in this case a map of record ids to their write and miss moving averages. In the approach of load-balanced middleware service nodes, every server already sees an unbiased sample of operations, whereas in the case that Cache Sketch maintenance is co-located with each partitioned database node, only local records have to be tracked, mitigating the space requirements.

## 4 Evaluation

We have implemented a Yahoo! Cloud Serving Benchmark (YCSB) wrapper for Monte-Carlo simulation (*YMCA*) for arbitrary caching architectures, which runs completely in memory. YCSB [CST+10] is a widely-adopted standard benchmark for CRUD data stores. As shown in Figure 7, YMCA consists of a client that implements the YCSB interface for basic CRUD operations, an arbitrary number of cache layers and additional modules for collecting metrics, in particular stale reads, cache misses and invalidations. Cache layers are stacked onto each other and can model any caching topology (e.g., a CDN or a reverse proxy). Latencies between layers are drawn from pluggable distributions, assuming symmetric latencies.

Overall, YMCA provides a toolbox to analyze caching behavior of multi-layered database infrastructures. The YMCA client tracks and reports stale reads. A read is considered stale, if there was an acknowledged write with a version that is newer than the version the client read and the timestamp from the begin of the read is newer than the write (i.e., the

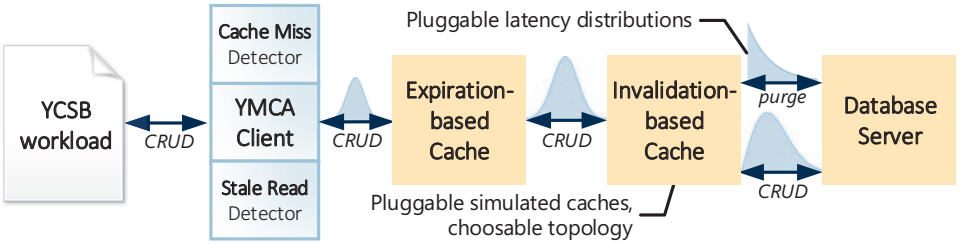


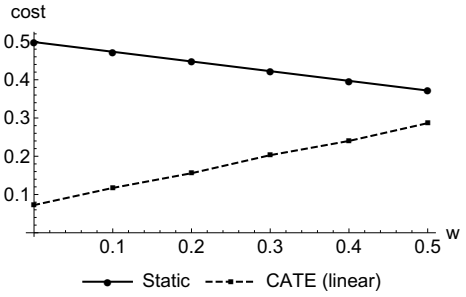

Figure 6: Concept of the extensible YMCA.

read started after the write was acknowledged to the client). Apart from stale reads and invalidations, YMCA also keeps track of cache hits and misses reported by each cache. In order to simulate long durations, YMCA implements a time scaling mechanism: all latencies and TTL estimations can be scaled by a defined factor. In the following, we assume the setting from Section 2.3 that includes an infrastructure consisting of a client, a CDN and the database service. The database service employs the server Cache Sketch to decide, if an update requires an invalidation and passes cache misses to the TTL estimator to assign the record-specific TTL.

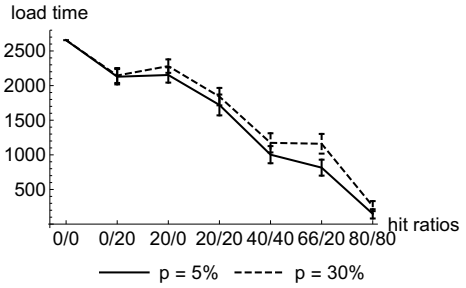
### 4.1 Parameter Optimization for the TTL Estimators

As discussed above, adaptive TTL estimation depends on the $slope$ of the ratio function, as well as $TTL_{max}$. In order to optimize these parameters, we use a variation of *maximum descent hill climbing*. Initial slopes of the ratio function are drawn uniformly at random in the $[0, 1]$ range. The algorithm then tests if increasing or decreasing the slope provides an improvement of the simplified cost function $(w \cdot \#cachemisses + (1 - w) \cdot \#invalidations)/\#ops$ that is to be minimized. Since the number of invalidations is an approximate indicator of stale reads as well as a measure of the Bloom filter population, we have found the simplified cost function to be a well-working simplification of the original cost function. Depending on the cost of cache misses compared to invalidations (and the subsumed false positive and stale read rate), terms are weighted with $w \in [0, 1]$.

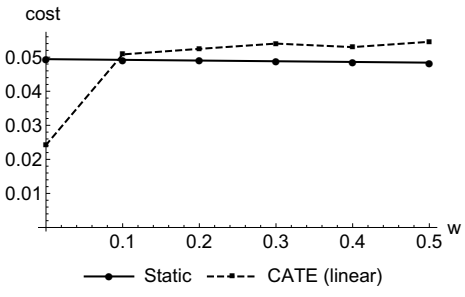
Testing directions of $TTL_{max}$ and $slope$ constitutes a super-step, which concludes by persisting the maximum direction of change (towards a lower cost) for the next super-step to start with. The algorithm terminates after maximum number of super-steps (50) or when it cannot improve costs. Optimizations were performed for YCSB workloads A (write-heavy; read/write balance 50%/50%) and B (read-heavy; read/write balance 95%/5%) with a Zipfian popularity distribution. Each simulation step was run on 100,000 operations for 10 threads, with a targeted throughput of 200 and a time scaling factor of 50, on the default amount of 1000 records. We ran the hill climbing algorithm from 25 starting points. Figure 7a and 7c show the resulting costs as a function of $w$ for the optimized parameters of CATE, with a linear ration function compared to the static TTL estimation



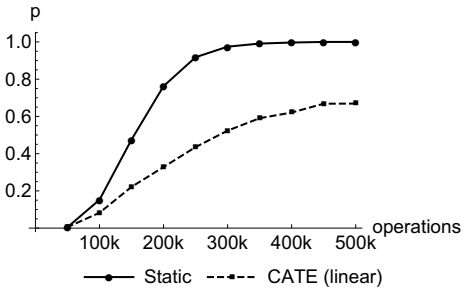

(a) Costs for Workload A.




(b) Page load time improvement through the cached initialization model.




(c) Costs for Workload B.

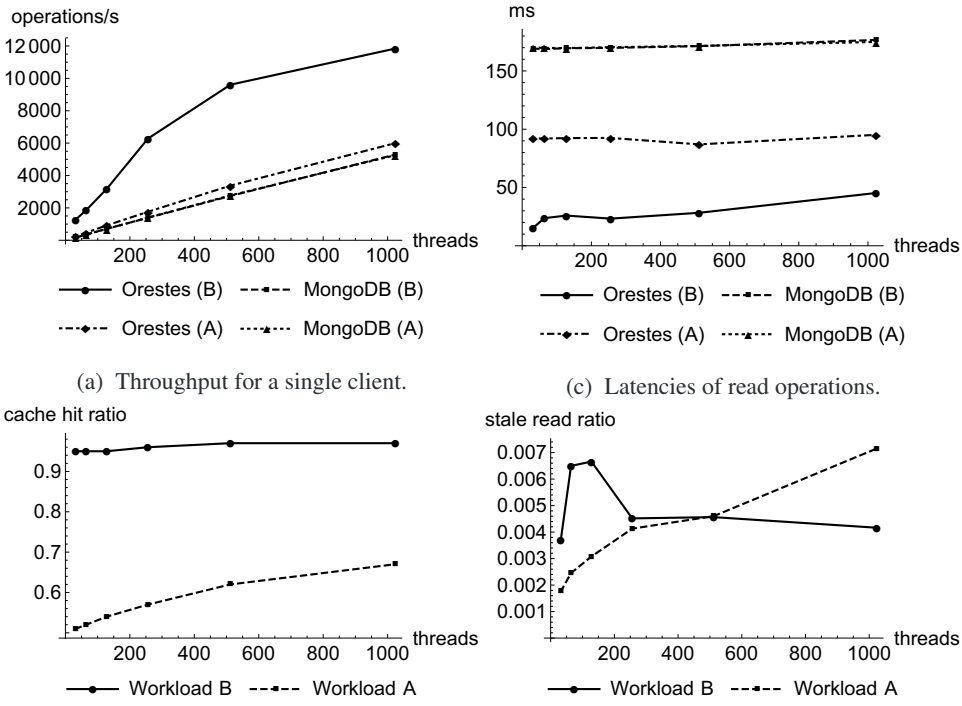



(d) False positive rate for a n=1k, p=0.05 Cache Sketch under Zipf-distributed operations.

Figure 7: YMCA simulation results.

with a high $TTL_{max}$. The results demonstrate, that CATE performs significantly better than static estimation for applications that do prefer high cache hit rates (workload A). Unsurprisingly, read-heavy workloads leading to many cache hits perform slightly better with a static (maximum) TTL estimation (unless cache misses are weighted very low).

As page load time is arguably the most important web performance metric, we analyzed the gains of *cached initialization* for different browser cache/CDN cache hit rates and two Cache Sketch false positive rates, assuming an average web page with 90 resources using 6 connections [GBR14] and that the Cache Sketch is used for every resource. The results shown in Figure 7b are as drastic as expected: for instance, for the 66%/20% cache hit rate described for Facebook photos [HBvR$^+$13], the speedup is over 320% for $p = 0.05$. The development of the Cache Sketch false positive rate is shown in Figure 7d for 100k records, workload B, a slope optimized for 100k operations and the Bloom filter configured to contain 1k elements with $p = 0.05$. As expected, CATE achieves lower false positive rates by decreasing TTLs, when $p$ grows too large. Even though the Cache Sketch is provisioned to only hold 1/100 of all records, the static estimator performs surprisingly well, as long as the number of operations is smaller than the number of total records.




(a) Throughput for a single client.

(c) Latencies of read operations.

(b) Cache hit ratios.

(d) Stale read ratios.

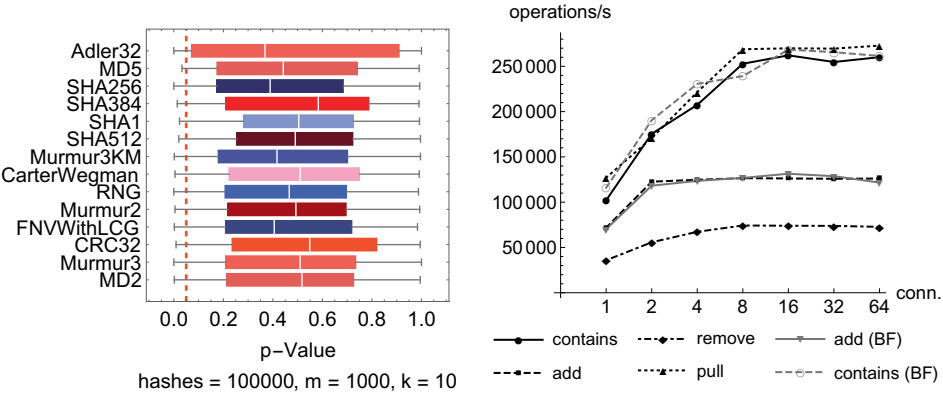
Figure 8: Performance and consistency metrics for YCSB with CDN-caching.

### 4.2 YCSB Results for CDN-Cached Database Workloads

To validate the results in a real-world setup, we conducted the YCSB benchmark for the described setup on Amazon EC2, using c3.8xlarge instances for the client (*northern California region*) and server (*Ireland*), while caching in the Fastly CDN. We took the document store MongoDB as a baseline and compared it to an ORESTES server running on the MongoDB machine to add the Cache Sketch and the REST API. The benchmark was performed with the same configuration as the simulation, but using the static TTL estimator. Figure 8 shows latency, throughput, cache hit ratios and stale reads for 32 to 1024 threads (i.e. YCSB clients). The results reveal the expected behavior: latency and throughput are considerably improved in both workloads, although a slight non-linearity between 512 and 1024 threads occurs, caused by thread scheduling overhead of the limited single-machine design of YCSB. MongoDB achieves the same latency and throughput in both workloads, since all operations are bound by network latency. The very few stale reads show considerable variance and were largely independent from the number of threads, as seen in Figure 8d, supporting our argument that $(\Delta_c, p)$-atomicity is an appropriate consistency measure and CDNs well-suited to answer Cache Sketch-triggered revalidations.

### 4.3 Efficient Bloom filter maintenance

The server Cache Sketch requires an efficient underlying Counting Bloom filter. For this purpose, we developed a Bloom filter framework available as an already widely-used open-source project[8]. It supports normal and Counting Bloom filters as in-memory data structures as well as shared filters backed by the in-memory key-value store Redis. The library supports the table-based sharding and replication introduced in Section 3 for high-throughput workloads. The Redis Bloom filter uses the capabilities of Redis to maintain an efficient bit vector for the materialized Bloom filter and relies on pipelining and batch transactions to ensure performance and consistency. The choice of Redis is motivated by its tunable persistence complemented with very low latency.




(a) Quality of Bloom filter hashes for random words (Box-Whisker plot of p-values).

(b) Throughput of the Redis-backed Counting Bloom filter and Bloom Filter (BF).

Figure 9: Analysis of the Redis-backed Bloom filters.

Figure 9 shows selected performance characteristics of the Redis Bloom filters. The uniformity of implemented hash functions for random Strings is evaluated in Figure 9a using the p-values for 100 $\chi^2$-goodness-of-fit tests. For random inputs (e.g., UUID record keys) all hash functions perform reasonably well - including simple checksums. However, for keys exhibiting structure, the best trade-off between speed of computation and uniformity is reached by Murmur 3. Figure 9b plots the throughput of the unpartitioned, non-replicated Redis Bloom filters for a growing amount of connections with $m = 100000$ and $k = 5$ on an Intel quad-core server with 16GB RAM. Read operations (querying, pulling the complete filter) achieve roughly 250k ops/s, while write operations (adding, removing) that require some overhead for counter maintenance and concurrency still achieve over 50k ops/s resp. 100k ops/s. This illustrates, that even a single-server Redis Cache Sketch is not likely to become a bottleneck in a database service.

[8] Available at https://github.com/Baqend/Orestes-Bloomfilter along with much more detailed results.

## 5 Related Work

Expiration-based web caching of static content has been researched from many perspectives. Huang et al. give an up-to-date analysis of the Facebook photo serving architecture which includes browsers caches, CDNs, custom edge caches and data store-level caching [HBvR+13]. The Summary Cache project [FCAB00] is another example for the use of Bloom filters in caching, where they are employed as metadata digests in cooperative web caches. Candan et al. [CLL+01] pioneered the idea of exploiting invalidation-based web caching for databases with the CachePortal system that detects changes of HTML pages based on underlying SQL queries and triggers corresponding invalidations.

An alternative approach to low latency applications are geo-replicated database systems, where a wealth of new systems and protocols have recently been proposed, including PNUTS, Walter, COPS, Megastore, Spanner, F1 and MDCC [KPF+13]. Some of the earlier approaches like DBCache and DBProxy [APTP03] also relied on caching, however in the form of dedicated database proxies. Geo-replicated approaches explore different positions in the consistency vs. performance trade-off-space, but usually require multiple synchronous wide-area round-trips for a consistency guarantee. Consistency in distributed and replicated storage systems has been studied in both theory [GLS11] and practice: PBS [BVF+12] has a similar approach to YMCA, using Monte-Carlo simulation to determine average staleness of reads in Dynamo-style quorum systems.

Our focus in this paper is different from previous work on web caching and geo-replication, since we aim to enable the use of expiration-based caching for database workloads with tunable $\Delta$-atomicity, relying only on readily available infrastructure and client capabilities.

## 6 Conclusions

In this paper, we addressed the problem of enabling database services to serve data from the globally-distributed caching infrastructure of the web. The problem is motivated by the observation, that web performance and user-perceived latency are a key differentiator for cloud service and application providers. More specifically, we designed the *Cache Sketch*, a data structure that allows clients to control their desired degree of consistency in the form of $\Delta$-atomicity, while being able to read every non-stale record from expiration-based caches (e.g., browser caches). To this end, the database service maintains the Cache Sketch as a Bloom filter of potentially stale records, while additionally employing it to decide, whether an update operation requires purging of invalidation-based caches (e.g., CDNs). To minimize the Cache Sketch size, invalidation costs and cache misses, we proposed the concept of *TTL Estimators* that produce access-dependent expiration dates. To reason about the resulting performance and consistency, the YCSB Monte-Carlo Caching Simulator offers a generic framework for analyzing different workloads, caching architectures and Cache Sketch parameters. The evaluation of CDN-cached workloads and Counting Bloom filter performance supports our claim, that the Cache Sketch is a feasible approach for large latency reductions in database services.

Our work leaves a number of questions for future investigation. One important area is the combination of the Cache Sketch with optimistic transactions, in particular a quantitative analysis of the abort rate reduction that can be achieved. Another important area is improving the efficiency of the Cache Sketch even further by designing the TTL estimator to learn optimal decisions online, without previous training in simulations, perhaps through techniques of time-series analysis and reinforcement learning.

## References


[APTP03] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proceedings of the ICDE*, pages 821–831, 2003.

[BMM02] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Math.*, 2002.

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[CLL+01] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling Dynamic Content Caching for Database-driven Web Sites. In *SIGMOD*, pages 532–543, New York, NY, USA, 2001. ACM.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, pages 143–154, 2010.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM TON*, 8(3):281–293, 2000.

[FFM04] Michael J. Freedman, Eric Freudenthal, and David Mazieres. Democratizing Content Publication with Coral. In *NSDI*, volume 4, pages 18–18, 2004.

[GBR14] Felix Gessert, Florian Bücklers, and Norbert Ritter. ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency. In *CloudDB 2014*, 2014.

[GLS11] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. Analyzing consistency properties for fun and profit. In *ACM PODC*, pages 197–206. ACM, 2011.

[Gri13] Ilya Grigorik. *High performance browser networking*. O'Reilly Media, [S.l.], 2013.

[HBvR+13] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *SOSP*, pages 167–181, 2013.

[KPF+13] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *EuroSys*, pages 113–126. ACM, 2013.

[PB08] M. Pathan and R. Buyya. A taxonomy of CDNs. *Content delivery networks*, pages 33–77, 2008.

[Tho98] A. Thomasian. Concurrency control: methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, 1998.

[VM06] Piet Van Mieghem. *Performance analysis of communications networks and systems*. Cambridge University Press, 2006.