Making Cache Monotonic and Consistent

Making Cache Monotonic and Consistent

Shuai An Yang Cao
University of Edinburgh
{shuai.an, yang.cao}@ed.ac.uk

Abstract

We propose monotonic consistent caching (MCC), a cache scheme for applications that demand consistency and monotonicity. MCC warrants that a transaction-like request always sees a consistent view of the backend database and observed writes over the cache will not be lost. We show that the complexity of MCC ranges from PTIME to NP-COMPLETE. We characterize MCC via a notion of obsolete items, based on which we abstract a principle for designing competitive MCC policies. By applying the principle, we develop an optimal MCC policy for the batch model, where requests in a batch are known in advance. For the online and semi-online models, we develop ML-augmented policies that benefit from blackbox ML models for classifying obsolete items, while being provably competitive even if the ML is arbitrarily bad. Using benchmark and real-life traces, we show that MCC policies reduce 39.09% of database reads for Redis atop HBase and improve their throughput by 77.15%.

1 Introduction

It has been a common practice to augment databases with external data caches, *e.g.,* Memcached [6] and Redis [7], to support data intensive Web applications [20, 59, 61, 62, 75]. By redirecting data access requests away from the database, data caches can reduce the load on the backend database, improving the overall system throughput.

Example 1: Consider a social media application in Fig. 1, where Alice successively accesses Bob's profile via read requests R_1, R_3 and R_4, during which Bob modifies his profile via write W_2, between R_1 and R_3. Initially Bob's profile in the database D is (name:Bob, region:US, phone:111, address:California). The application server M uses a Redis cache C with "infinitely" large space and is initially empty; following look-aside caching [40, 59, 65], M bridges D and C.

Upon R_1, the application server M makes C fetch and cache name:Bob, region:US, address:California from the database D. It then receives a cache invalidation message for W_2, *e.g.,* ban [3], which notifies that the cached region and address are stale. Following the lazy eviction strategy [7] in Redis or the ban protocol, C still keeps them as it is not running out of space. Then name and region of R_3 are both cache hits while phone is a miss. Hence M fetches phone:222 from D for R_3 and caches it in C. This seems to have made R_3 a cache hit as each requested item is a hit in C. It, however, may not pass the application logic since phone:222 is not *consistent* with name:Bob, region:US in C, *i.e.,* there is never a time that (Bob, US, 222) exists in D. To this end, M re-fetches region:UK from D and adds it to C, to make R_3 a consistent cache hit, *i.e.,* (Bob, UK, 222).

After that, R_4 could have a consistent cache hit (Bob, US, California) over C. However, it may not pass the application logic as Alice already observes region:UK in R_3 but this would be lost in R_4, *i.e.,* Alice's view on region would go back in time if M used this consistent hit for R_4. Hence, M fetches address:London and caches it in C, to make R_4 see a "consistent" and "monotonic" view of D. □

As shown in Example 1, a cache request may access a set of items.

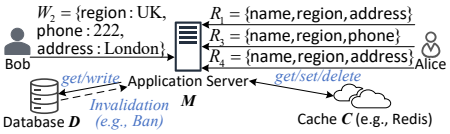


Figure 1: Application-level data caching in Example 1

For such requests, even a cache hit could still be useless if it sees inconsistent (*e.g.,* R_3) or non-monotonic (*e.g.,* R_4) items. Indeed, consistency assures that the application always sees a consistent view of the database at certain point of time, while monotonicity prevents it from losing observed writes. They are the reported desiderata of applications in, *e.g.,* social network [20, 24, 59, 69, 75], e-commerce [17, 25, 27, 73] and streaming services [2, 14, 26, 33, 47, 68].

While previous studies [27, 37, 38, 40, 42, 52, 56, 62, 65] have developed customized systems and cache schemes that help track writes and inform the applications about inconsistent cache hits, they rely on application logic to specify the actions for such "invalid" hits or use default handling rules, *e.g.,* treating them as cache miss. This however leads to two *different* and *independent* forces in maintaining the cache content: (a) the cache replacement policy that decides which cached items to evict when cache overflow occurs (*e.g.,* LRU), and (b) the logic that handles invalid cache hits that do not conform to the desired semantic properties. This impairs the performance guarantees of existing cache replacement policies and leads to suboptimal performance. Indeed, we will see shortly that traditional optimal policies are not ideal anymore due to nontrivial interference with the handling of invalid cache hits (Section 4). In addition to this, monotonicity has not yet been addressed in previous schemes, in particular for requests accessing a set of items.

Contributions. We fill the void by making two contributions:

(1) We develop cache policies that holistically account for both traditional cache overflows and invalid cache hits caused by violation of consistency and monotonicity. We prove that they are *theoretically competitive* and even *optimal*, which are beyond the reach of conventional policies that are optimized only for cache overflows.

(2) To make practical use of the policies, we develop MCCache, a tool that deploys them over existing caches, *e.g.,* Redis and Memcached, without changing their internal implementations; this improves their throughput atop HBase by 77.15% on average.

Below we elaborate these in more detail.

MCC policies. We first formulate *monotonic consistent caching* (MCC), a scheme that uniformly captures cache overflows and invalid cache hits that violate monotonicity and consistency. MCC allows us to characterize the effectiveness of cache policies in the presence of monotonicity and consistency. We show that traditional optimal policies are not competitive anymore and formally study cache policy design under MCC. We consider all three input models that have ever been used in caches: batch, semi-online and online.

The batch model abstracts cases where we have a high volume of requests that are processed in batches. This has been used in *e.g.,*

Facebook's Memcached clusters [59] and transaction systems [9], where requests are buffered before being served. The semi-online model assumes that read requests are known as a batch while write requests are not. A typical example of the semi-online model is secondary nodes in Redis [8], as writes are propagated from the primary node via cache invalidation while reads are batched as usual in local buffers. The online model has the least restriction and is the mostly perceived: both read and write requests are revealed to the cache policy online at runtime, one after another [20, 66, 67]. An online cache policy has to make the cache decision for each request on-the-fly, without any knowledge of subsequent requests.

Complexity. We investigate the complexity of MCC. We prove that in general optimal MCC cache policy is NP-Complete, as opposed to conventional caching that is trivially in Ptime [18]. Unlike conventional cache policies that only need to decide which cached items to evict upon cache overflows caused by cache misses, MCC policies also have to deal with inconsistent or non-monotonic cache hits, and decide which version of the items to cache.

We consider two *version selection strategies*: `Eager` and `Lazy`. Informally, `Lazy` allows to cache items with bounded staleness, in line with the lazy eviction of Redis, while `Eager` always fetches the most current items and automatically warrants monotonicity.

Surprisingly, we find that the complexity of MCC significantly differs *w.r.t.* `Eager` and `Lazy`. Indeed, we show that optimal MCC policy remains NP-Hard with `Eager`, while it becomes Ptime with `Lazy`.

Characterization. We dig deeper and characterize the impact of monotonicity and consistency on caching. We identify a class of cached data items, which we refer to as *obsolete* items, that can explain why the hardness of MCC policies varies with `Eager` and `Lazy`: with `Lazy` it is Ptime to decide whether a cached item is obsolete while it becomes coNP-Complete with `Eager`.

Based on the characterization, we develop a principled approach to the design of MCC policies. It builds upon the following proposition (informal): if a cache policy $\mathcal{P}$ is c-competitive [12] for MCC, then $\mathcal{P}^o$ is at least c-competitive, where P^o evicts obsolete items first and then acts exactly the same as $\mathcal{P}$ does upon cache overflows.

Optimal policies. As applications of the principle, we develop

(a) an MCC policy for the batch model that is *optimal* with `Lazy`.

(b) MCC policies for the semi-online and online models that can incorporate *blackbox* ML classifier $\mathcal{M}$ for deciding obsolete items:
   - they can benefit from accurate classifications from $\mathcal{M}$ and be provably *competitive* (online) and *optimal* (semi-online);
   - they are *ML-robust*, *i.e.,* they remain competitive even when $\mathcal{M}$ is adversarial and produces arbitrarily bad predictions.

MCCache. We develop MCCache[1], a pluggable system that adds the support of MCC to existing data caches in a non-intrusive way. Specifically, MCCache bypasses the internal cache policy of data caches, mostly some LRU variant [6, 7], by running a dedicated MCC cache policy atop them and injecting MCC cache actions via eviction operators provided by the underlying data caches. Our current implementation of MCCache provides built-in connectors for the most popular data caches including Redis and Memcached.

Using both YCSB benchmark and real-life traces, we evaluate the effectiveness of our cache policies for Redis and Memcached. We find the following. (1) MCCache effectively empowers Redis and Memcached with the capacity of upholding monotonicity and consistency. (2) Under the batch model, our MCC policies in MCCache reduce 42.76% of the cost (number of database reads) of MCC policies adopted from conventional caching, up to 61.79%. (3) Under the semi-online and online model, on average our policies have costs 41.28% and 20.01% lower than competitor policies, respectively, with ML predictions that have 95% of accuracy; they remain 28.02% and 12.16% better even when the classification accuracy of the ML oracle is as low as 80%. (4) As a proof of concept, we train a simple classification model $\mathcal{M}$ using LightGBM [45] as the ML oracle to predict obsolete items for our semi-online and online policies in MCCache for Redis with HBase as the backend database. We find that on average with MCCache, Redis achieves 85.23%, 68.17% and 35.51% higher throughput than with competitor policies, under the batch, semi-online and online models, respectively; similarly for Memcached.

Summary & organization. In summary, we deliver the following.

- We propose monotonic consistent caching (MCC) for caches that uphold consistency and monotonicity (Section 3). We develop MCCache, a lightweight tool that enables existing caches, *e.g.,* Redis and Memcached, to benefit from MCC policies (Section 3.2).
- We study the complexity of MCC policies. We show it is NP-Complete with `Eager`, and becomes Ptime with `Lazy` (Section 4.1).
- We develop a principle for competitive MCC policies by characterizing MCC with a notion of obsolete items (Section 4.2).
- Following the principle, we develop an MCC policy for the batch model, and prove that it is optimal with `Lazy` (Section 5).
- We design ML-augmented semi-online and online MCC policies that are both competitive and ML robust (Section 6).
- Using YCSB benchmark and three real-life cache traces, we evaluated the effectiveness of our MCC policies. On average MCC policies reduce the database access of Redis/Memcached over HBase by 39.09%, improving its throughput by 77.15% (Section 7).

(See [4] for full proofs of all the results and additional experiments.)

Related Work. We categorize the related work as follows.

Cache systems and schemes. Data caches such as Redis [7] and Memcached [6] have been well established in practice [20, 49, 59, 75], to improve system throughput by reducing database load. There has also been effort to customize data caches with semantic guarantees according to the application logic, *e.g.,* consistency [15, 35, 36, 38, 42, 49, 57, 62], read-your-wirte [20, 65] and cache serializability [27].

These systems develop cache protocols for various trade-offs between consistency guarantees and performance. In contrast, (1) we rethink the design of cache policies for these cache systems, by holistically taking into account traditional cache overflows and cache hits that are invalid due to violation of semantic properties imposed by the applications. (2) We also develop tool to make these caches monotonic and consistent, and benefit from MCC policies.

Monotonicity and consistency. Monotonicity and consistency are two fundamental desiderata of streaming [2, 5, 26, 33, 47, 54], distributed [39, 48, 68, 72] and transaction [15, 27, 38, 49, 62] systems. Consistency applies to scenarios where multiple copies of the same

[1] https://github.com/jiayouanan/mccache

database item in different versions co-exist and applications want to see a consistent view of the items. Monotonicity asserts that for consecutive reads to a data item, the latter one never sees a version older than the earlier one, which is more of a concern in, *e.g.,* streaming applications [2, 5, 33, 54] where updates are frequent.

In contrast to distributed consistency that studies whether different copies of the *same* item reflect the same value, we study whether *multiple* items in a cache hit of a request that accesses a set of items are consistent, *i.e.,* whether the cache hit reflects a consistent view of the database and hence is useful to the applications. Similarly, the impact of monotonicity in caching has not yet been addressed.

Cache policies. There has been a host of work on the design and analysis of cache eviction policies that decide which cached items to evict upon cache overflows. This includes offline caching where the sequence of the read requests are known in advance [13, 18, 21, 22, 31, 43], and online caching where read requests arrival online one by one [12, 19, 29, 30, 44, 55, 60]. In both cases, the analysis focuses on the competitiveness of the policies for singleton read-only requests. In particular, it has been shown that for paging, the Belady's rule is optimal [18]. For online policies, LRU and FIFO are k-competitive deterministic policies and are widely used in practice.

There has also been recent work on ML-augmented cache policies for the paging problem [51, 64, 66, 67, 74]. The idea is to exploit ML models that predict some information about the read sequence, *i.e.,* the next arrival time of a read request, to improve cache hit rate.

Our work differs from existing work in the following. (1) Instead of conventional caching that requests one item at a time, we consider transaction-like requests that access a set of data items. (2) In MCC, only monotonic and consistent cache hits are valid. In contrast, these properties do not exist in the context of conventional caching, whose analysis even does not consider write operations. (3) In contrast to ML-assisted cache policies [51, 64, 66, 67, 74] that predict the sequence of future requested items, which is often difficult if not infeasible in practice, our online policies incorporates simple binary classification models which are much easier than predicting workload sequence. This is made possible by characterizing the impact of the monotonicity and consistency on caching, which is beyond the scope of conventional caching and paging.

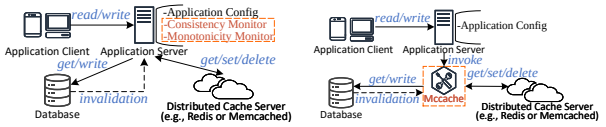
Closer to this work is [15], which applies consistent caching to improving the throughput of deterministic transaction databases. It studies caching with request reordering for offline transactions over a single-version database. Different from the study, we consider caching with both monotonicity and consistency over multi-version databases with different version selection strategies, *e.g.,* `Eager` and `Lazy`. In addition, we study caching under the batch, semi-online and online input models, instead of for offline transactions only.

2 Preliminary

We review the basics of caching and conventional cache policies.

Database versions. We model a database D simply as a set of data items $\{d_1, \ldots, d_n\}$; in practice D could be *e.g.,* a relational database or a key-value store. In this work, we focus on the case where all data items are of the same size, *e.g.,* tuples of the same relation, or values of the same column family in key-value stores; our results can be readily extended to cases where items are of varying sizes.

We consider both read and write operations to D. A write $\mathtt{W}[d_i]$ updates d_i in D with a new value, while a read $\mathtt{R}[d_i]$ returns d_i. Conceptually, each write generates a new version of D. We denote by $D[i]$ the (logical) snapshot of D in *database version* i. Similarly, each data item d in D also evolves into a new version once it is updated by a write $\mathtt{W}[d]$. We denote by $d[j]$ the value of d in *item version* j. Note that, $D[i+1]$ and $D[i]$ differ in at least one data item, *i.e.,* there exists $d \in D$ such that $d[j] \in D[i]$ and $d[j+1] \in D[i+1]$.



(a) *existing architecture* (b) *architecture with* MCCache

Figure 2: Application-level data caching architectures

Cache basics. We consider databases augmented with an external application-level data cache C, as adopted by *e.g.,* Facebook [59] and Twitter [75]. As shown in Fig. 2a, writes are committed to the database and are propagated to C via cache invalidation governed by application logic [40, 62, 65]. By logging the invalidation message, *e.g.,* ban [3], C (via application servers) knows the difference between the item version of d in C and that of d in the latest D, referred to as the *staleness* of d in C and denoted by $\mathrm{stale}(d)$. Intuitively, one may want to use cached items with bounded staleness [62, 65].

Consider a read request $\mathtt{R}[d]$. If d is not in C, then $\mathtt{R}[d]$ is called a cache miss. In such a case, the cache fetches d from D to C so that $\mathtt{R}[d]$ is made into a cache hit. A *cache overflow* occurs when C has no room for storing the newly fetched d; the cache then needs to evict some cached items from C, so that d could be brought into C.

Cache policy. For a sequence ℓ of requests, a *cache schedule* P for ℓ is a list of eviction actions for cache overflows occurred when serving ℓ over cache C. More specifically, for each read $\mathtt{R}[d]$ in ℓ, if it is a cache miss that inflicts a cache overflow, then the entry for $\mathtt{R}[d]$ in P is the items to be evicted from C so that d can be brought into C from D; otherwise, it is an empty entry. A *cache policy* is an algorithm $\mathcal{P}$ that, given any sequence ℓ, generates a cache schedule for ℓ.

Many cache policies have been developed. For example, LRU and LRU-k [60] are online policies for cases when ℓ consists of reads revealed one by one, and Belady [18, 21] tackles the case when ℓ is known in advance. Both Memcached and Redis use LRU by default.

3 Caching with Monotonicity and Consistency

We first formulate monotonic consistent caching (MCC) in Section 3.1. In Section 3.2, we then present MCCache, a lightweight pluggable tool that deploys MCC policies over existing caches .

3.1 Monotonic Consistent Caching

Requests. A *set-based request* R is of the form $\{d_1, \ldots, d_m\}$, where each d_i is referred to as a read *query*. Over a database D, each d_i retrieves d_i from D if it exists, or returns 'miss' otherwise (to reduce notation, we use d interchangeably for items in D and queries in R). Similarly, a set-based write request W updates multiple data items in one go. We often refer to set-based requests simply as requests.

In practice, R could be a read-only transaction in large-scale distributed databases [1, 20, 50, 59, 62, 71, 75], where d_i is a read operation. For streaming query serving systems [10, 58], R could be

a query and d_i is a cached view [5]. For Web applications, R could be an HTTP request that fetches tens or even hundreds of data elements that could be in arbitrary types of values, including *e.g.,* tuples, strings, dates or integers, from the backend database [49, 50].

Properties. In all these applications, there are two fundamental desiderata: *consistency* and *monotonicity*, which we formalize below.

Consistency. Consider a database D that changes over time by write requests. A read request $R = \{d_1, \ldots, d_m\}$ is *consistent over* D if there exists a database version l such that the retrieved items d_1, $\ldots$, d_m all exist in $D[l]$, *i.e., R* sees a consistent view of D.

Monotonicity. For two read requests R and R', we say that R *antecedes* R' *over* D if for any query d that appears in both R and R' such that d reads $d[i]$ from D for R and reads $d[j]$ for R', we have $i \leq j$. For a sequence ℓ of read requests $R_1, \ldots, R_n$, we say that R_i is *monotonic in* ℓ *over* D if for any $R_l (l < i)$, R_l antecedes R_i over D.

In addition to these, applications may also impose conditions on data freshness to avoid reading data that are too stale.

Monotonic consistent caching. We next study caching with monotonicity and consistency imposed by the applications, referred to as *monotonic consistent caching* (MCC).

Consider a database D augmented with cache C via an application server, as shown in Fig. 2a. Consider a sequence ℓ of requests $T_1, \ldots, T_n$, where each T_i is either a read request R or write request W that accesses to multiple data items in D. Let the cache C be of limited size much smaller than that of D. Assume that only items with staleness no larger than s can be used in C. Under MCC, a read request $R = \{d_1, \ldots, d_m\}$ in ℓ has three possible states over C:

(1) Monotonic consistent cache (MCC) hit. Informally, R is an *MCC hit* over C for D if (i) R is a cache hit over C, *i.e.,* each d_i in R is a cache hit; (ii) R is consistent over D via the items read from C; (iii) R is monotonic in ℓ over D; and (iv) items read by R are not too stale. Formally, R is an MCC hit, if C contains $d_1[v_1], \ldots, d_m[v_m]$, such that

(a) there exists a database version $D[l]$ that contains all of $d_i[v_i]$ for $i \in [1, m]$, *i.e., R* sees a consistent view of D via C;

(b) for each d_i of R, there exists no request R' that precedes R in ℓ and sees a version of d_i newer than $d_i[v_i]$ read by R; and

(c) each cached d_i has $\text{stale}(d_i[v_i]) \leq s$, a staleness bound.

Intuitively, condition (a) ensures that R is answered consistently even when C caches data items from multiple versions of D, (b) further asserts that accesses to data items via C are monotonic, and (c) enforces a data freshness condition on the answers to R.

(2) Non-MCC hit. R is a *non-MCC hit* over C if (a) for each d_i, there exists a copy of d_i in C, however, (b) the cached copies of $d_i (i \in [1, m])$ in C cannot form an MCC hit for R. That is, the cached data items for R are either not consistent, older than the versions that have been seen by some request R' that precedes R in ℓ, or not fresh enough.

(3) Cache miss. R is a cache miss if C does not contain some d_i of R.

Under MCC, only an MCC hit is valid for the applications. Previous studies provide effective methods to identify cache hits that are inconsistent [27, 37, 38, 40, 42, 52, 56, 62, 65]. However, they use conventional cache policies, *e.g.,* LRU to manage cache content by deciding cache evictions upon overflows, while invoking application logic to deal with invalid cache hits by, *e.g.,* re-fetching the items in the cache hits. This, however, leads to two separate algorithmic logics that manage C: (a) cache policies for cache overflows and (b) rules for handling cache hits that violate consistency. This voids the competitiveness of cache policies [18, 53, 60] due to the foreign operations on the cache content from the applications.

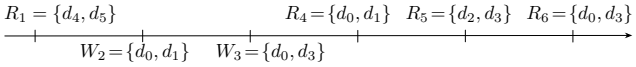


Figure 3: Sequence ℓ_1 in Example 2

We put them into the same space for MCC, via the notion of *MCC schedules* below. This allows us to formally study MCC policy design.

MCC schedules. Similar to conventional cache schedules, an MCC schedule decides, for a sequence ℓ of requests over cache C, the actions to C upon each read request in ℓ. Different from conventional cache policies that contain only cache eviction operations for cache overflows caused by cache misses, an MCC schedule also needs to address non-MCC hits: which stale items in C to update and which version should be re-fetched in order to make them into MCC hits.

Consider a sequence ℓ of requests over database D and a cache C. When processing requests of ℓ one by one under MCC, an *MCC schedule for* ℓ *over* C is a list P of updating operations to C, one for each read request R in ℓ, such that the following holds:

(a) $P[R]$ is one of following: (i) fetching a set of data items from D to C with some version selection strategy (more below) when R is a non-MCC hit or cache miss, and evicting sufficiently many items from C first if it has no room to hold the fetched items, or (ii) *nil* if R is already an MCC hit, *i.e., C* does not need updates to serve R;

(b) After applying $P[R]$ to C, R becomes an MCC hit over C; and

(c) At all times the size of items in C does not exceed its capacity k.

An *MCC policy* is an algorithm that, given any sequence ℓ of requests, generates an MCC schedule for ℓ over C.

Version selection. We consider two natural version selection strategies in MCC schedules upon R that is a non-MCC hit or a cache miss: `Eager` and `Lazy`. (a) With `Eager`, schedules eagerly update cached items to the latest version of fetching them from D if they are not cached; note that with `Eager` it is always feasible to make R an MCC. (b) With `Lazy`, schedules instead try to make use of stale cached items in a bounded way, while conforming to consistency, monotonicity and staleness bound; it does this by fetching data items from the oldest version l of D that can make R an MCC hit. That is, for any $l' < l$, fetching items from $D[l']$ cannot make R a consistent cache hit that satisfies both monotonicity and staleness.

Intuitively, `Eager` enforces that items are current when they are brought into C. In contrast, `Lazy` allows C to use slightly less fresh items to answer read requests as long as they have bounded staleness. Both `Eager` and `Lazy` have to conform to the staleness bound upon MCC hits. In particular, `Lazy` and `Eager` converge when $s = 0$.

Example 2: Consider a sequence ℓ_1 of read and write requests as shown in Fig. 3. Assume that the cache C initially contains $\{d_0, d_1, d_2, d_3\}$ and is full. Let staleness bound $s = 1$. Then ℓ_1 generates 3 database versions, say $D[0]$ prior to W_2, $D[1]$ after W_2 and $D[2]$ after W_3; similarly, d_0 has three item versions while d_1 and d_3 have two. R_1 is a cache miss as C does not have d_4 or d_5. Hence, one has to

evict two items from C to cache R_1. Assume that we evict d_0 and d_1; then upon R_4, we have a cache miss as C contains neither d_0 nor d_1.

With Eager, we fetch $d_0[2]$ and $d_1[1]$ from $D[2]$ to answer R_4, making it into an MCC hit (note that R_4 causes an overflow; hence we evict d_4 and d_5 first). Then R_5 is an MCC hit, while R_6 is a non-MCC hit as $d_0[2]$ and $d_3[0]$ are not consistent. Hence we need to further fetch $d_3[1]$ from $D[2]$ for R_6. On the other hand, if we use Lazy, we fetch from $D[1]$ for R_4 as it contains $d_0[1]$ and $d_1[1]$, conforming to staleness bound $s = 1$. After that, both R_5 and R_6 are MCC hits. □

3.2 Making Data Caches Monotonic and Consistent

We present MCCache, a tool that adds MCC policies to data caches.

MCCache. As shown in Fig. 2b, MCCache is a middleware built atop existing data caches. It applies to the look-aside caching (Fig. 2a) that is widely adopted by Web applications, *e.g.,* Facebook [59] and Twitter [75], where writes commit to the database and then propagated to the application server and cache via invalidation [3].

MCCache intercepts only the cache replacement process of the data caches, by evicting cached items according to MCC schedules generated by MCC policies in MCCache, bypassing their default cache policies, *e.g.,* LRU and LRU-k [60], that are oblivious to monotonicity and consistency. This is made possible since MCC schedules deal with both cache overflows and non-MCC cache hits, before passing the read items to the application server. Hence underlying the cache replacement logic never faces cache overflows; as a result, existing cache policies, *e.g.,* LRU, never kick in.

MCCache provides built-in connectors to Redis and Memcached. (See more details about the design of MCCache in [4]).

Input models. MCCache supports three input models, depending on the knowledge that the MCC policies have about the requests ℓ.

(a) Batch model. Under the batch model, the sequence ℓ of read/write requests of the current batch is known to the MCC policy in advance when it generates MCC schedule for ℓ. The batch model abstracts applications in *e.g.,* deterministic databases [9] where concurrent transactions are batched via a global append-only log [70] first before they are executed, or streaming applications where requests are processed in epochs that are grouped by sessions [11, 46].

(b) Semi-online model. Under the semi-online model, read requests are known to the MCC policies while write requests are not. It models the scenario where the cache buffer receives possibly out-of-order writes (cache invalidation) while reads are batched. It also applies to secondary Redis nodes, which receive read requests only in their local buffers while writes are handled by the primary Redis node and are propagated over time via cache invalidation [8].

(c) Online model. Requests in ℓ are coming online and are revealed to $\mathcal{P}$ one after another. When the policy $\mathcal{P}$ decides the schedule for a request R, it has *no* knowledge about any of the subsequent requests in ℓ. This is the most common one and is widely used when the arrival rate of the requests is not high enough for batching.

Example 3: Consider the sequence ℓ of (R_1, W_2, R_3, R_4) in Example 1. Under the batch model, MCC policies decide actions on C for each R_i with ℓ known in advance (*i.e.,* batched). Under the semi-online model, only R_1, R_3 and R_4 are known while W_2 becomes visible only after R_1. For the online model, MCC policies decide actions for each R_i with the knowledge of only R_j and W_h with $j, h \leq i$. □

4 Implications of Monotonicity and Consistency

To start with, we first formulate the problem of MCC policies design and analyze its complexity (Section 4.1). We then abstract a principle for designing competitive MCC policies for the MCCPolicy module, by developing characterizations of MCC (section 4.2).

4.1 Complexity of MCC Policies

Problem. A central task in caching is the problem of cache policy design. Below we formulate the task for MCC policies, referred to as the *monotonic consistent caching problem* (MCCP), as follows:

Input	A cache buffer C of capacity k, a sequence ℓ of read and write requests over D, and a staleness bound s.
Output	An MCC schedule P for ℓ over C.
Objective	Minimize cost(P), the total number of items to be fetched from D when serving ℓ over C with P.

Intuitively, the MCCP problem is to design an MCC policy that minimizes the caching cost for any input sequence of requests. In particular, we will study MCCP in all three input models, depending on how we know about ℓ when generating cache schedules.

Complexity. To understand the intricacy inherent to MCC, we first investigate the complexity of MCCP under the batch model below. As will be seen shortly, this allows us to develop a coherent principled approach to MCCP for all three input models.

Unlike conventional caching which is trivially in PTIME [18], MCC is much more complicated, as illustrated below.

Example 4: Continuing with Example 2, we show that the optimal policy for conventional caching under the batch model, namely the Belady's rule [18], is no longer optimal when adopted for MCC. The idea of Belady's rule is to evict, upon a cache overflow, the item in the cache whose next read request time is the furthest in ℓ.

Assume that we use Lazy for version selection. Upon R_1, Belady evicts d_2 and d_3 as the nearest next request containing them is R_5, which is the furthest in ℓ. Then R_4 is a cache hit but a non-MCC one. This is because C contains $d_0[0]$, exceeding the staleness bound $s = 1$ as the latest D (*i.e.,* $D[2]$) contains $d_0[2]$. Thus, C has to refetch $d_0[1]$ from $D[1]$. To make R_4 an MCC hit, d_1 has to be further updated to $d_1[1]$ as C contains $d_1[0]$ which is not consistent with $d_0[1]$. R_5 is a cache miss, which requires to fetch $d_2[0]$ and $d_3[0]$ from $D[0]$. R_6 is MCC hit. Hence, Belady has a total cost of 6 reads.

Now consider the schedule in Example 2, *i.e.,* evicting d_0 and d_1 upon R_1. One can readily verify that its cost is 4 reads with Lazy, as both R_5 and R_6 become MCC hit, better than Belady.

Similarly, one can verify that, with Eager Belady's rule incurs a cost of 6 reads, while the schedule of Example 2 has a cost of 5. □

As shown in Example 4, under MCC a read request can be a cache hit that is inconsistent or exceeds the staleness bound. Hence, conventionally performant or even optimal cache policies do not work well when they are adopted and extended to uphold consistency and monotonicity. Indeed, MCC policies are much harder to design.

Theorem 1: (1) *The decision version of* MCCP *is* NP-COMPLETE.

(2) *It remains* NP-HARD *with* Eager *as the version selection strategy.*

(3) *It becomes* PTIME *when used with* Lazy. □

Below we sketch a proof of Theorem 1(1) and (2). We will give a constructive proof of Theorem 1(3) in Section 5. (See Appendix A of the full version [4] for a complete proof).

Proof sketch: We only need to show that MCCP is in NP and it is NP-HARD when used with Eager. An NP algorithm for MCCP works as follows: (a) guess a schedule P of length the same as that of ℓ, and (b) check whether P is a valid MCC schedule for ℓ over C, in PTIME.

We show that MCCP is NP-HARD with Eager by reduction from the maximum coverage problem (MCP) which is NP-COMPLETE [34]. Given two integer k and h, and n sets $\Gamma = \{S_1, \ldots, S_n\}$, MCP is to decide whether there exists a k-subset Γ' of Γ such that $\bigcup_{S\in\Gamma'} S$ contains at least h elements. The reduction uses s=1, a sequence ℓ of $M + 2m + 4$ read requests and 2 write requests, and C of capacity $(2m + 1)M + m + n + 1$, where $m = |\bigcup_{S\in\Gamma} S|$ and $M = \sum_{S\in\Gamma} |S|$. □

4.2 Characterizations

We next characterize the root cause of the complexity of MCC and develop a principled approach to designing MCC policies.

We first take a closer look at MCC policies, and see why conventional cache policies do not work when adopted for MCC.

Example 5: There are three scenarios where cached items can be obsolete, *i.e.,* useless. Recall Example 2. One can verify that, initially, d_0 (*i.e.,* $d_0[0]$) in C cannot be used to answer R_4 or any other requests in ℓ_1 due to the data freshness restriction: upon R_4, $d_0[0]$ is not fresh enough *w.r.t.* the staleness bound $s = 1$ for R_4. As a result, d_1 ($d_1[0]$) in C is also obsolete as it cannot make a consistent cache hit with the updated d_0, *i.e.,* $d_0[1]$ with Lazy and $d_0[2]$ with Eager.

Consider another case where we have a cache C' that contains both $d_0[0]$ and $d_0[1]$. Then we can tell that $d_0[0]$ is obsolete for sure with both Lazy and Eager: when $d_0[1]$ is fetched to C', $d_0[0]$ cannot be used anymore due to monotonicity on d_0. □

As shown in Example 5, a cached item can be in a state that it could never be used to answer read requests, due to monotonicity, consistency and data freshness in MCC. They are naturally key to a cache policy to be optimal as in case of cache overflows, such items could be safely evicted without impairing any potential MCC hits.

Obsolete items. Consider a sequence ℓ of requests to be processed over cache C. At any time t, an item d in cache C is *obsolete* for ℓ if for any read request $R \in \ell$ to be processed at or after time t, d can never be used by any MCC schedule to answer R over C even $d \in R$.

Note that, obsolete items are a concept over time. Indeed, d can be obsolete in C upon the arrival of request $R \in \ell$, while it however could still be used in an MCC hit for a read request prior to R.

A principle. The concept of obsolete items naturally gives us a principled approach to the design of MCC policies: when a cache overflow occurs, *i.e.,* C is short of space for newly fetched items, it is natural to evict items in C that are obsolete at the time first.

Indeed, for any MCC schedule P for ℓ over C, let P' be a schedule derived from P as follows: for each read request $R \in \ell$, $P'[R]$ first evicts all items in C that are obsolete at the time when R is processed, and then follows $P[R]$. Then the cost of P', *i.e.,* the total number of items fetched from the database, is no larger than that of P over ℓ, and moreover, P' is a valid MCC schedule for ℓ as long as P is.

However, in order to understand and make full use of the idea, there are two questions to answer. (1) How effective is evicting obsolete items in reducing the cost of cache schedules? (2) What is the complexity of identifying obsolete items?

Effectiveness. We first study the effectiveness of evicting obsolete items, by examining the extent that obsolete items can help reduce non-MCC hits. Intuitively, one of the main differences between MCC and conventional caching is that a cache hit under MCC can be a non-MCC hit and hence is invalid. We want to know whether by evicting obsolete items we can eliminate non-MCC hits and narrow down the gap between MCC and conventional caching.

We show that obsolete items indeed capture all non-MCC hits with Lazy. For convenience, assume *w.l.o.g.* that we evict obsolete items after answering each read request R in ℓ over C. Note that this is not a restriction as obsolete items at the time right after answering R are exactly those right before R', where R' is the read request immediately next to R in ℓ. Consider sequence ℓ over cache C.

We say that a read request R is an *MCC miss* over C if one can make R into an MCC hit over C by fetching items in $R \setminus C$ only. Then by induction on the length of ℓ, one can readily verify the following.

Proposition 2: *For any MCC schedule P for ℓ over C, if P evicts all obsolete items right after each read request, then with both* Lazy *and* Eager *each read request R in ℓ is either an MCC hit or MCC miss.* □

Proposition 2 tells us that the principle of evicting obsolete items helps eliminate all non-MCC hits and non-MCC misses. This justifies the effectiveness of the principle for MCC cache policies.

Complexity. To use the principle of evicting obsolete items in practice, it is necessary to understand the complexity of obsolete items.

Theorem 3: (1) *It is in* PTIME *to decide whether an item in C is obsolete with* Lazy *at the time when a read request R in ℓ is processed.*

(2) *It becomes* coNP-COMPLETE *when* Eager *is used. It remains* coNP-HARD *even staleness bound s is a fixed constant no smaller than 1.* □

Proof sketch: We first sketch a proof of Theorem 3(2). To see that it is in coNP, observe that an item d in C is not obsolete at time j iff there exists an MCC schedule that uses d to answer a request at or after j, and it is in PTIME to check whether a schedule witnesses this. We show that it is coNP-HARD by proving it is NP-HARD to check whether d is not obsolete, via a reduction from the 3SAT problem, which is NP-COMPLETE [34]. Given a proposition formula ψ, 3SAT decides whether ψ is satisfiable. The reduction uses a cache of size $5n + m + 3$, ℓ with $6n + 3m + 3$ read and 2 write requests, and s=1, where m (resp. n) is the number of clauses (resp. variables) in ψ. □

We will constructively prove Theorem 3(1) in Section 5 by developing a PTIME algorithm for finding obsolete items.

5 An Optimal Policy for the Batch Model

Using the principle developed in Section 4, we present bMCP, a "small-but-sweet" MCC policy for the batch model that works with both Lazy and Eager, and is proven optimal with Lazy.

Conceptually, the basic idea of bMCP is simple: it combines the principle we developed in Section 4.2 with the Belady's rule. When a cache overflow occurs upon a read request R in a sequence ℓ, it first evicts all obsolete items in the cache at the time, to see whether R can be made a cache hit; if not it further evicts cached items of

ALGORITHM 1: The bMCP policy

Input: Cache C, sequence ℓ, staleness bound s.
Output: MCC cache schedule P for ℓ over C.

Upon each read request R_i in ℓ:
1 **if** R_i is an MCC hit over C **then** $P[R_i] \leftarrow nil$; *// no need to update cache for R_i*
2 **else** *// R_i is a non-MCC hit or cache miss*
3 $l \leftarrow$ decideDBv(R_i, C, s); *// decide the database version that R_i should see*
4 update cached copies of items of R by reading from $D[l]$ if they are older;
5 **if** $R_i \nsubseteq C$ and C has no room for $R_i \setminus C$ **then** *// R_i is a cache miss*
6 $O \leftarrow$ OB(C, ℓ_i, s); *// ℓ_i: the suffix of ℓ starting from R_i*
7 evict all items in O from C ; *// evicting obsolete items*
8 **foreach** item d of R_i that is not yet cached in C **do**
9 **if** C is full **then** *// obsolete items do not suffice; further apply Belady's rule*
10 evict from C the item with next read time furthest in ℓ;
11 read d from version $D[l]$ and cache it in C;
12 $P[R_i] \leftarrow$ all evictions and reads for R_i; *// cache actions to make R_i MCC hit*

Procedure decideDBv(R, C, s) *// for Lazy*
1 **foreach** $d \in R \cap C$ **do** $l_d \leftarrow \max(d.\text{minV}, d.\text{lastV})$;
2 **return** $l \leftarrow \max_{d \in R \cap C} l_d$

Procedure OB(C, ℓ_i, s)
1 do a dry run of bMCP(C, ℓ_i, s) by assuming C has an infinite capacity;
2 **return** the set of items initially in C that do not appear in ℓ_i or are updated by line 4 of bMCP in the dry run;

which the next read time in ℓ is the furthest away from R, until C has enough room to hold items requested by R.

There are however two challenges regarding the design and analysis of bMCP. (1) How to deal with non-MCC cache hit or cache miss of which the hit part is inconsistent? That is, why bMCP is an MCC policy? (2) Why is this embarrassingly simple policy optimal with Lazy? Below we formally address the challenges.

The bMCP policy. As shown in Algorithm 1, given a cache C (of bounded capacity), sequence ℓ of requests, and a staleness bound s that imposes the data freshness requirement, bMCP generates a cache schedule P for ℓ over C by deciding the cache update actions $P[R_i]$ for each read request R_i in ℓ, one after another.

Specifically, if R_i is an MCC hit, $P[R_i]$ is simply *nil* as no cache update is needed to serve R_i over C (line 1). Otherwise, R_i is either a cache miss or non-MCC hit; in both cases C needs to be updated to accommodate R_i. To this end, bMCP first decides whether the cached part of R_i satisfies monotonicity, consistency and staleness in C. It does this by deciding the database version l that R_i should see according to the MCC scheme, via decideDBv (line 3; more below). If the cached copies of items requested by R_i do not agree with $D[l]$, it updates them by re-fetching them from $D[l]$ (line 4).

If R_i still misses items in C, *i.e.,* R_i is a cache miss, bMCP first evicts all obsolete items from C via OB (lines 5-7; more below). It then fetches items of R_i that are not cached in C from $D[l]$ one by one (lines 8-11). If C still has no sufficient space to hold all the missing items for R_i, it then applies the Belady's rule, *i.e.,* evicting the cached items of which the next request time is the furthest in ℓ, until all items of R_i are brought in C. These eviction and fetching operations form the cache actions for R_i in P (line 12).

(1) Deciding database version (decideDBv). We next describe how bMCP decides the database version l for each read request R. For Eager, this is straightforward as l is simply the latest database version. The algorithmic logic for Lazy is also shown in Algorithm 1. Intuitively, for each cached item d, we record two database versions, (a) d.minV for the minimum database version that contains d while conforming to staleness bound s, and (b) d.lastV keeps the lowest database version that contains d in the same version as the cached d that was lastly used by some read request R' preceding R in ℓ.

With this, decideDBv decides, for each d of R that is also cached in C, a version l_d that conforms to both staleness and monotonicity by taking the maximum of d.minV and d.lastV (line 1). It then picks the maximum l_d among all items d of R that are in C as the database version l for R (line 2). This further warrants consistency for R.

(2) Identifying obsolete items (OB). As also shown in Algorithm 1, procedure OB identifies obsolete items for bMCP via a dry run of bMCP over C for ℓ_i, by assuming C has an infinite capacity. Hence, the conditions in line 5 and line 9 never hold, *i.e.,* no eviction occurs in the dry run (which also ensures that OB is never invoked in the dry run). It returns all items that are in C at the start of dry run but are never used to answer some read request. That is, all items that are initially in C but are (a) updated by line 4 of bMCP or (b) not requested by ℓ_i are returned as obsolete items.

Example 6: Continue with Example 2. We show how bMCP generates the cache schedules. Consider Lazy first. bMCP decides that R_1 is a cache miss and fetches d_4 and d_5 for it from $D[0]$, which causes a cache overflow. It then evicts all obsolete items, *i.e.,* d_0 and d_1, from C to make R_1 into an MCC hit. Again, R_4 is a cache miss and cache overflow; however, there exists no obsolete item in C. Hence, bMCP evicts d_4 and d_5 for R_4 and caches d_0 and d_1 from $D[1]$. After that, it decides that R_5 and R_6 are both MCC hit. This is exactly the cache schedule in Example 2. The case with Eager is similar. □

Analysis. We next show that OB and bMCP are constructive proofs of Theorem 3(1) and Theorem 1(3), respectively.

Complexity. We start with complexity analysis. For each read request R in ℓ, bMCP can decide cache actions $P[R]$ in $O(|R|) + T_{\text{ob}}(R)$ time, where $|R|$ is the number of items requested by R and $T_{\text{ob}}(R)$ is the time for identifying obsolete items via OB for R if R causes a cache overflow. In theory, $T_{\text{ob}}(R)$ is $O(|\ell_R|)$, where ℓ_R is the suffix of ℓ starting from R and $|\ell_R|$ is the total length of ℓ_R. However, in practice, the dry run of OB does not need to examine all requests in ℓ_R. As will be practiced in Section 7, by scanning as few as 10 requests OB can typically identify sufficiently many obsolete items.

Properties. bMCP guarantees monotonicity and consistency, and is theoretically optimal when used with Lazy.

Theorem 4: *Under the batch model,*

(1) bMCP *is monotonic and consistent with both* Lazy *and* Eager*;*

(2) OB *finds all the obsolete items with* Lazy*; and*

(3) bMCP *is optimal with* Lazy. □

Proof sketch: For (1), consistency with both Lazy and Eager is warranted by lines 4 and 11 of bMCP, which assure that each R sees exactly the database version l decided by decideDBv(R). Monotonicity holds automatically with Eager. It is guaranteed for Lazy by line 1 of decideDBv that ensures read versions of an item never decrease.

For (2), the crux is a lemma: with Lazy for any MCC schedule P for ℓ over C, if an item $d[i]$ is obsolete at time j by P, then for any

MCC schedule P' for ℓ that keeps $d[i]$ in C at time j, $d[i]$ must also be an obsolete item in P'. We prove (3) by induction on the length of ℓ, using a lemma: an item $d[i]$ is obsolete for R iff the database version that `decideDBv`(R) identifies does not contain $d[i]$. □

6 Competitive Online Policies with ML Oracles

We next extend our study of MCC policies to the semi-online model (Section 6.1) and online model (Section 6.2), by again applying the principle of evicting obsolete items. However, the challenge is, in both models, the policies cannot decide obsolete items as they have no knowledge of future (write) requests. Nonetheless, we propose ML-augmented MCC policies that can incorporate *any blackbox binary classifier* as an oracle that predicts whether a cached item is obsolete. Moreover, we show that the policies are robust: (a) they benefit from accurate classification and can even be optimal; and (b) they are still theoretically competitive even with adversarial predictions.

6.1 A Robust Competitive Semi-Online MCC Policy

We first present `sMCP`, an MCC policy for the semi-online model that incorporates only a binary classifier $\mathcal{M}$ for deciding obsolete items.

The `sMCP` policy. The design of `sMCP` is simple: we simply replace line 6 of `bMCP` in Algorithm 1 by invoking a plugged-in ML classifier $\mathcal{M}$ that predicts obsolete items in C (more below). All the other steps of `sMCP` remain exactly the same as `bMCP`.

ML oracle. `sMCP` can be used with any blackbox ML classifier $\mathcal{M}$ that, given a cached item d in C, predicts whether d is obsolete. It uses $\mathcal{M}$ as an oracle and does not make any assumption on its design or what guarantees it must have on the generalization error. We will give a proof-of-concept design of $\mathcal{M}$ in Section 7 shortly and show that `sMCP` can already benefit from models with moderate accuracy.

Properties. `sMCP` inherits monotonicity and consistency from `bMCP` since it differs from `bMCP` only in the way of identifying obsolete items. Hence, the proof of Theorem 4(1) applies to `sMCP` as well.

While `sMCP` is simple, it has some interesting properties. In particular, we show its *competitive ratio* can be quantified by the classification accuracy of $\mathcal{M}$ over ℓ, which verifies that it can benefit from good predictions from $\mathcal{M}$ and even be optimal. Furthermore, we also show that `sMCP` can be made robust against arbitrarily bad $\mathcal{M}$.

Competitiveness. We first review the notion of competitiveness [53], which has been widely used to analyze online algorithms [12, 30].

In the context of MCC, a policy $\mathcal{P}$ is *c-competitive against* policy $\mathcal{P}'$ if, for every sequence ℓ of requests, $\text{cost}(\mathcal{P}, \ell) \leq c \cdot \text{cost}(\mathcal{P}', \ell) + O(1)$, where $\text{cost}(\mathcal{P}, \ell)$ is the cost of the schedule P that $\mathcal{P}$ generates for ℓ, and c is called a competitive ratio. $\mathcal{P}$ is *c-competitive* if $\mathcal{P}$ is c-competitive against OPT, the optimal offline policy, *i.e.,* `bMCP` for MCC. Here c could be a function over ℓ, to express instance-dependent competitiveness. In particular, $\mathcal{P}$ is *optimal* if it is 1-competitive, *i.e.,* it has the lowest cost on every ℓ among all policies.

We say that the ML oracle $\mathcal{M}$ makes a mis-classification if it predicts that an item d is obsolete while it is not or vice versa. Denote by $\eta(\ell)$ the accumulated absolute error that $\mathcal{M}$ makes when `sMCP` generates schedules for ℓ with $\mathcal{M}$, which is the total number of mis-classifications $\mathcal{M}$ makes. Let $\epsilon(\mathcal{M}) = \frac{\eta(\ell)}{\text{cost}(\text{OPT},\ell)}$. Intuitively, $\epsilon(\mathcal{M})$ measures the quality of the ML oracle $\mathcal{M}$ by comparing its accumulated absolute error with the total cost that OPT (*i.e.,* `bMCP`) incurs over the same input ℓ. Then below we show that the competitive ratio of `sMCP`, denoted by CR(`sMCP`), is a function over $\epsilon(\mathcal{M})$. (See Appendix A of full version [4] for a formal proof.)

ALGORITHM 2: The `oMCP` policy (online model)

Input: Cache C, sequence ℓ revealed one by one online, staleness bound s.
Output: MCC cache schedule P for ℓ over C.

Upon the arrival of R_i:
1 **if** R_i is an MCC hit over C **then**
2 $P[R_i] \leftarrow nil$;
3 *mark all items requested by R_i in C*;
4 **else** *// R_i is a non-MCC hit or cache miss*
5 $l \leftarrow$ `decideDBv`(R_i, C, s); *// recall from Algorithm 1*
6 **foreach** $d \in R_i \cap C$ **do**
7 **if** the cached version of d in C is older than the one in $D[l]$ **then**
8 update d in C by re-fetching it from $D[l]$;
9 *mark d in C*;
10 **foreach** item d of R_i that is not yet cached in C **do**
11 **if** C is full **then** *// C has no room for d*
12 **if** there exists no *unmarked item* in C **then**
13 *unmark all the items in C and start a new phase*;
14 **foreach** $d_o \in C$ **do**
15 **if** $\mathcal{M}$ predicts that d_o is obsolete **then**
16 evict d_o from C
17 **if** C still has no room for q **then**
18 evict the least recently used item in C ; *// use LRU*
19 read d from version $D[l]$ and cache it in C;
20 *mark d in C*;
21 $P[R_i] \leftarrow$ all evictions and reads for R_i; *// cache actions to make R_i MCC hit*

Lemma 5: *With* `Lazy`, CR(`sMCP`) $\leq 1 + \epsilon(\mathcal{M})$. □

Proof sketch: An error of $\mathcal{M}$ can either be (a) an FP error if it predicts d obsolete while it is not, or (b) an FN error if it tells that d is not obsolete while it indeed is. We then show that each FP and FN error, respectively, causes at most 1 more cost when compared to `bMCP` with `Lazy`. Hence, cost(`sMCP`) $\leq$ cost(`bMCP`) $+ \eta_1 + \eta_2$, where η_1 and η_2 are the accumulated absolute FP and FN errors, respectively. This gives an upper bound of $1 + \epsilon(\mathcal{M})$ on CR(`sMCP`) with `Lazy`. □

Lemma 5 verifies the following about policy `sMCP`.

(1) `sMCP` benefits from accurate predictions from $\mathcal{M}$. It gets strictly better performance (competitiveness) with ML oracles of higher accuracy. In particular, when $\mathcal{M}$ produces absolutely accurate predictions, `sMCP` becomes optimal as CR(`sMCP`) = 1 with `Lazy`.

(2) `sMCP` has decent performance even with moderate $\mathcal{M}$. Indeed, $\epsilon(\mathcal{M})$ is easily much smaller than 1 as the absolute error $\mathcal{M}$ makes over ℓ is typically smaller than the total cost of OPT over ℓ. This is because each item incurs 1 cost when it is brought into the cache C by OPT, while not all items in C of `sMCP` are mis-classified by $\mathcal{M}$.

ML robustness. We say that a policy $\mathcal{P}$ is *ML-robust* if its competitive ratio is no larger than a number that is *independent* of the length $|\ell|$ of ℓ, no matter how bad the ML oracle $\mathcal{M}$ becomes. Note that, Lemma 5 does not show that `sMCP` is ML-robust as we have not derived an upper bound for $\epsilon(\mathcal{M})$ independent of $|\ell|$.

Instead of upper bounding $\epsilon(\mathcal{M})$, we show that one can make `sMCP` robust by further combining it with another variant of `bMCP`. Denote by $\texttt{bMCP}^0$ the variant of `bMCP` without lines 5-7 in Algo-

rithm 1, and by sMCP* the below combination of sMCP and bMCP^0:

(a) Run both sMCP and bMCP^0 independently on input ℓ in parallel.

(b) sMCP* switches between following the actions of sMCP and the actions of bMCP^0 (it starts with sMCP initially). It switches from sMCP to bMCP^0 upon R of ℓ if the cost of sMCP so far is at least twice as that of bMCP^0; similarly when it switches from bMCP^0 to sMCP. When switching from one to another, sMCP* reconciles its cache content to that of the one it switches to.

Note that sMCP* does not actually operate the cache and read the database when executing sMCP and bMCP^0 as it only simulates their cache decisions. The combination method has been developed to combine multiple online algorithms for the paging problem [16, 29, 30]. It ensures that the combined algorithm is competitive against each individual component algorithm. Below we show that the idea also applies to MCC, and moreover, sMCP* is ML-robust.

Theorem 6: *With* Lazy*, (1)* sMCP* *is 18-competitive even when* $\mathcal{M}$ *produces arbitrarily bad classification; and (2)* sMCP* *is optimal when* $\mathcal{M}$ *produces no mis-classification.* □

Proof sketch: We prove (1) by using the following lemmas: (a) bMCP^0 is 2-competitive with Lazy under the semi-online model, and (b) sMCP* is 9-competitive to both bMCP^0 and sMCP. Here Lemma (b) is verified by proving that the total cost of switching from bMCP^0 to sMCP (resp. sMCP to bMCP^0) is bounded by the cost of bMCP^0 (resp. sMCP), and then applying the online algorithm combiner of [16, 30]. For (2), it follows from that sMCP reduces to the bMCP with perfect $\mathcal{M}$ and sMCP* starts with copying sMCP and never switches to bMCP^0. □

6.2 Extending to the Online Model

We further extend our study to the online model. Along the same lines as sMCP, we present oMCP, an online MCC policy that incorporates any blackbox binary classifier $\mathcal{M}$ for deciding obsolete items. We show that oMCP is both competitive and ML-robust.

Similar to bMCP and sMCP, oMCP also applies the principle of evicting obsolete items with a plugged-in ML oracle $\mathcal{M}$. However, there are two new challenges under the online model. (1) It cannot employ the Belady's rule to further evict items when cache overflow occurs even when all obsolete items have been evicted (line 10 of Algorithm 1), as it does not have access to subsequent read requests in ℓ. (2) As a result of (1), the consequence of mis-classification by $\mathcal{M}$ on the competitiveness of the policy could be amplified unboundedly by non-optimal eviction choices made in line 10 of Algorithm 1.

Hence, one cannot simply replace OB and Belady's rule in line 10 of bMCP with ML oracle $\mathcal{M}$ and some alternative eviction strategy *e.g.,* LRU, respectively, as this would yield an uncompetitive policy.

Nonetheless, below we show that both competitiveness and ML-robustness are attainable for MCC under the online model, by applying the ML oracle $\mathcal{M}$ in a controlled way in oMCP.

The oMCP **policy**. In contrast to sMCP that evicts obsolete items upon each overflow, oMCP invokes $\mathcal{M}$ and evicts obsolete items in phases to bound the affected scope of mis-classifications by $\mathcal{M}$.

More specifically, as shown in Algorithm 2, oMCP decides the cache update action $P[R_i]$ upon the arrival of a read request R_i in a way that is similar to sMCP, except that it divides its run over ℓ in *phases* by marking items in the cache C. An item d in C is marked at the time when it is brought into C (lines 9 and 20), or when it is part of an MCC hit for R_i (line 3). A phase ends when all items in C are marked (line 12). In such case, a new phase starts by unmarking all items in C (line 13). Marking and unmarking are logical operations for partitioning the execution trace of oMCP over ℓ into phases; they are not part of cache actions in the schedule.

Policy oMCP only invokes the ML oracle $\mathcal{M}$ and evicts items that are predicted obsolete by $\mathcal{M}$ at the beginning of each phase (lines 12-16). If C still has no room to hold new items requested by a cache miss R_i after evicting the predicted obsolete items or in the middle of a phase where no obsolete items are evicted (line 17), oMCP falls back to use LRU as the eviction strategy as an alternative to the Belady's rule used by bMCP and sMCP (line 18).

Analysis. Similar to sMCP, oMCP guarantees monotonicity and consistency since (a) bMCP does and (b) oMCP differs from bMCP only in their identified obsolete items, which only affect the performance, *i.e.,* competitiveness, of the cache policies. We next show that oMCP is both competitive and ML-robust. Let k be the size of C.

Theorem 7: *(1)* oMCP *is* k*-competitive even when* $\mathcal{M}$ *is arbitrarily bad,* e.g., $\mathcal{M}$ *mis-classifies each and every item. (2) There exists no deterministic online MCC policy that is* k'*-competitive, for any* $k' < k$. □

By Theorem 7(1), we know that oMCP is competitive and ML-robust against mis-classifications from arbitrarily bad $\mathcal{M}$. Moreover, Theorem 7(2) confirms that oMCP is as good as any deterministic online MCC policy without using ML, no matter how bad $\mathcal{M}$ can be.

Proof sketch: We verify (1) by proving the following lemmas, with Lazy. (a) With adversarial $\mathcal{M}$, the cost of oMCP is at most $k + N_i$ in phase i, where N_i is the number of obsolete items arrived in phase i. (b) The optimal bMCP incurs at least $1 + N_i$ costs in phase i. For (2), observe that traditional caching is a special case of MCC and the best deterministic traditional online policy is k-competitive. □

7 Experimental Study

Findings. Using YCSB benchmark and real-life traces, we experimentally evaluated the effectiveness of cache policies bMCP, sMCP and oMCP using Memcached and Redis under the batch, semi-online and online models, respectively. Our main finding is that they clearly outperform existing cache policies. More specifically:

- Under the batch model, bMCP outperforms MCC variants of conventional policies in cost (number of database reads) by 42.76% on average; in particular, it improves the optimal conventional policy, Belady [18], by 30.61% on average, up to 45.12%.
- Under the semi-online model, sMCP outperforms conventional policies by 41.28% on average with an ML oracle $\mathcal{M}$ that has an accuracy of 95%; it has a lower cost than the optimal conventional policy Belady even when the accuracy of $\mathcal{M}$ is below 75%.
- For the online model, the cost of oMCP is on average 20.01% lower than that of the conventional policies with ML $\mathcal{M}$ of accuracy of 95%. Similar to sMCP, oMCP is also robust: it has lower cost than competitors even if $\mathcal{M}$ has an accuracy below 75%.
- We developed a simple proof-of-concept ML model $\mathcal{M}$ for predicting obsolete items. By deploying MCCache atop Redis and Memcached with HBase as the backend database and $\mathcal{M}$ as the

ML module, we find that on average, the throughput of Redis is 85.23%, 68.17% and 35.51% higher with bMCP, sMCP and oMCP than with the competitor policies, respectively.

Below we first specify the settings (Section 7.1). We then present our evaluation results on the costs of all cache policies (Section 7.2). Finally, we discuss our proof-of-concept ML model for sMCP and oMCP, and system throughput evaluation of Redis and Memcached as caches with MCCache and HBase as the database (Section 7.3).

7.1 Experimental Setup

Datasets. We used both benchmark transactions and three real-life access traces as sources of set-based requests that MCC targets.

YCSB *benchmark*. We used YCSB [23] to generate read and write transactions conforming to YCSB core A workloads, where each item is a value; the tasks are transactions that access the items via keys and serve as the MCC requests. It has the below parameters:

- θ: the built-in Zipfian distribution parameter in YCSB to control the skewness of reads and writes; larger θ means more skewed read/write accesses. We varied θ from 0.8 to 1.2 (1.2 by default).
- write%: the percentage of write requests in the YCSB trace; larger write% indicates higher rate of updates to the YCSB database. We varied write% from 30% to 50% and set it to 50% by default.
- #-queries: the total number of read/write operations in a transaction, ranging from 4 to 12, and is set to 8 by default.
- p_{write}: the percentage of operations in a (hybrid) request that are write, which varies from 12.5% to 62.5% (50% by default).
- dbsize: the size of the YCSB database, ranging from 10GB to 30GB, with the number of keys varied from 10M to 30M.
- vsize: the size of an item (value) in YCSB database. We varied vsize in [1KB, 1024KB] (1KB by default), following [23, 41, 63].

Distributions. In addition to the default Zipfian, we also tested with alternative YCSB built-in distributions [23]: (a) *Uniform*: each item has an equal probability to be read/written; (b) *Exponential*: reads/writes follow an exponential distribution; and (c) *Latest*: similar to Zipfian but tends to access significantly more new items.

Real-life traces. We also used real-life traces from three applications.

(a) Wiki: a slice of real-life web access trace collected on a CDN node serving media content for Wikipedia [67]. Wiki consists of 10^8 items that are grouped into requests. The number of operations (#-queries) in a Wiki request, the write percentage (write%) and the percentage of write operations in a hybrid request (p_{write}) are varied in the same way as YCSB does.

(b) Twitter: a one-week-long user requests trace from Twitter's in-memory caching clusters collected in March 2020 [75], where each record has explicit reads and writes. We tested with its cluster 14 and used the successive reads and writes as set-based requests.

(c) Ibm: a single week cloud-based key-value dataset with 99 traces collected from IBM Cloud Object Store service [28]. Each trace includes 22 thousand to 187 million read/write requests. We used trace 1 and varied requests in the same way as YCSB and Wiki do.

Given a configuration of the parameters, we generated sequences of requests as workloads, each consisting of 5000 requests.

Cache policies. We implemented all our MCC policies, *i.e.,* bMCP, sMCP and oMCP for the batch, semi-online and online policies, respectively. In addition, we also developed the following policies as baselines that are adopted from conventional cache policies:

(a) mcBelady: a simplified version of bMCP that does not evict obsolete items, *i.e.,* Algorithm 1 without lines 5-7. It treats each request as multiple singleton reads and processes them one by one, using Belady [18] for eviction upon cache overflows.

(b) LRU: the mostly used cache policy (default in Redis and Memcached); we tailored it for MCC similar to how we derived mcBelady.

(c) LRU-k [60]: a popular variant of LRU adopted for MCC.

(d) BeladySet: a direct application of Belady's rule that processes one (set-)request at a time without serializing it first.

(e) LRUSet: a direct application of LRU similar to BeladySet.

Among them, mcBelady and BeladySet work for the batch and semi-online models only, while others work for all input models. We injected cache schedules of all policies into Redis/Memcached via MCCache, so that they are compared in exactly the same fair setup.

System deployment. We deployed MCCache atop Redis v7.0.2 and Memcached v1.5.6, with HBase v2.2.4 as the backend database that stores all the four datasets. For sMCP and oMCP, our implementation either (a) accepts a classification on obsolete items with controlled accuracy so that we can evaluate the impact of the classification accuracy on the cost of sMCP and oMCP schedules; or (b) directly uses a plugged-in ML classification model (see Section 7.3 and Appendix C of full version [4]) that predicts obsolete items on-the-fly, so that we can evaluate the impact of the cache policies on the overall system performance, *e.g.,* throughput.

Configuration. The experiments were run on AWS EC2, where we used the m5.24xlarge instance for HBase and m5.8xlarge for Redis and Memcached. All instances are in the same region connected by 10 Gigabit intranet. The cache size was set to a p_{csize} fraction of the database. We varied p_{csize} from 10% to 30% (20% by default). We also varied the staleness bound s of MCC from 0 to 10 (10 by default). When varying a parameter, all other parameters were set to their default. Each experiment was run 3 times. The average is reported. Due to space limit, we report key results below; see Appendix B of full version [4] for a complete report and additional experiments.

7.2 Cost of Cache Policies with Varying Predictions

We first evaluated the effectiveness of all cache policies in reducing database read load. To do this, we tested the cost of the policies, measured as #dbread, the total number of reads to the database when serving YCSB and Wiki workloads. Note that, #dbread is not exactly consistent with cache hit rate, since a non-MCC hit or an MCC miss could also save reads to the database.

Effectiveness under the batch model. We first report the cost (#dbread) of all cache policies under the batch model. The results over all the four datasets are reported in Figures 4a-4h.

(1) bMCP is consistently the best among all policies in all cases with both Lazy and Eager. On average, the #dbread of bMCP is 30.83%, 44.63%, 44.96%, 51.73% and 50.98% lower than that of mcBelady, LRU, LRU-k, BeladySet and LRUSet, respectively, over YCSB with Lazy, up to 45.12%, 52.25%, 52.97%, 61.79% and 61.08%; similarly for other traces. This also confirms the optimality of bMCP (Theorem 4).

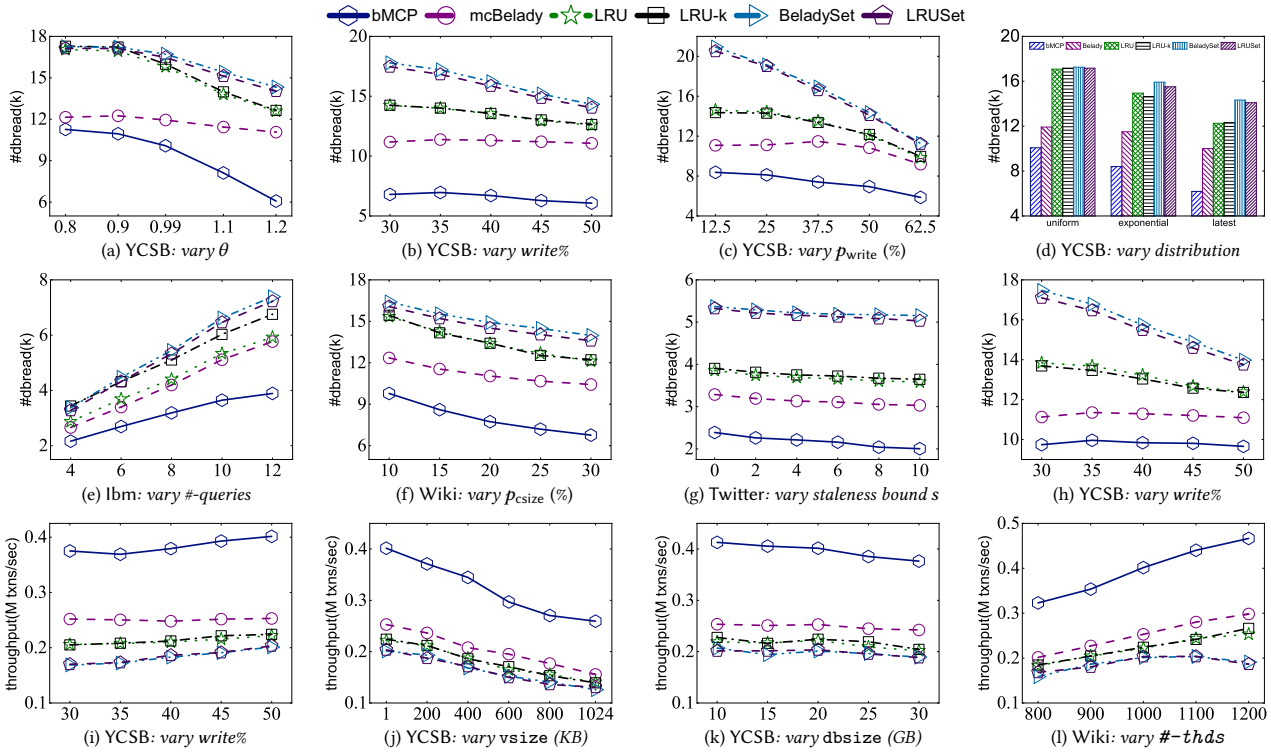


Figure 4: Experimental results under the batch model: *(a)-(g) and (i)-(l) with* Lazy*; (h) with* Eager

(2) As shown in Figures 4a-4e, the benefit of bMCP is even more evident when the workloads are skewed or with larger write%, *i.e.,* higher update rate. Over YCSB with Lazy, when θ varies from 0.8 to 1.2, the gap between bMCP and the second-best policy, mcBelady, increases from 7.21% to 45.12%; similarly, when write% varies from 30% to 50%, the gap increases from 39.20% to 45.12%. The results are also consistent when we varied the data distributions (Fig. 4d): the gap is 26.93% and 38.16% over the Exponential and Latest distributions, respectively, while it is 15.50% over Uniform that has no skewness.

This is because with more skewed accesses or higher update rates, cached items are more likely obsolete. It verifies the effectiveness of the principle of obsolete items underlying all of our policies.

(3) As shown in Fig. 4f, the cost of all cache policies reduces with larger cache (p_{csize}), while the gaps between bMCP and other policies are stable. On the other hand, the gaps grow when the staleness bound s increases (Fig. 4g), which shows that bMCP can capitalize boundedly stale cached items in a consistent and monotonic way.

(4) As shown in Figures 4b and 4h, the cost of all policies becomes higher with Eager than with Lazy. In particular, bMCP benefits from Lazy the most. This verifies that the use of slightly stale cached items in a bounded, monotonic and consistent way does help with cache performance, and bMCP best exploits stale data in such a safe way.

Semi-online and online: effectiveness and ML-robustness. We also evaluated our semi-online policy sMCP and online policy oMCP by comparing their #dbread with that of all other policies over YCSB and Wiki. To assess their ML-robustness, we fed sMCP and oMCP with classification predictions of varying accuracy. In particular, we denote by sMCP(α%) the sMCP with classification on obsolete items that has α% of probability being correct; similarly for oMCP(α%). Key results under semi-online and online models are shown in Figs 5a-5c and 6a-6c respectively. For clarity, we only plot the best competitor *i.e.,* mcBelady for semi-online and LRU for online model.

(1) Both sMCP and oMCP consistently have much lower #dbread than other policies over varying workloads when provided with accurate classification. For instance, the #dbread of sMCP(100%) is on average 41.54%, 51.22%, 51.94%, 59.24%, 58.36% lower than that of mcBelady, LRU, LRU-k, BeladySet and LRUSet over YCSB with Lazy, respectively, when varying write% from 30% to 50% (Fig. 5b). The results are consistent when varying skewness θ of YCSB (Fig. 5a), over Wiki (Fig. 5c), or under the online model (Figures 6a-6c).

(2) Both sMCP and oMCP are robust against classification errors. For instance, sMCP still outperforms mcBelady, the second-best policy, by 13.89% with classification accuracy as low as 80% when write% is 50% over Wiki with Lazy; similarly for other cases. Furthermore, achieving 80% accuracy is not difficult for a binary classification ML model in caching tasks [66, 67] (also in Section 7.3). On the other hand, we also noticed that both sMCP and oMCP with 75% ML accuracy sometimes perform worse than best competitors when the workloads are less skewed or with fewer writes. This is because in these cases, cached items are less likely obsolete, which in turn makes false positive classifications more likely to happen. Nevertheless, the ML-robustness of sMCP and oMCP assures us that their performance does not degrade unboundedly even *each and every* classification made by the ML model is incorrect (Theorems 6 and 7).

(3) We also find that Eager is more sensitive to classification accuracy and relies more on accurate predictions than Lazy. This shows that Lazy is superior for caching as it allows better use of boundedly stale items than Eager does. It is also consistent with the theoretical properties that sMCP and oMCP hold with Lazy.

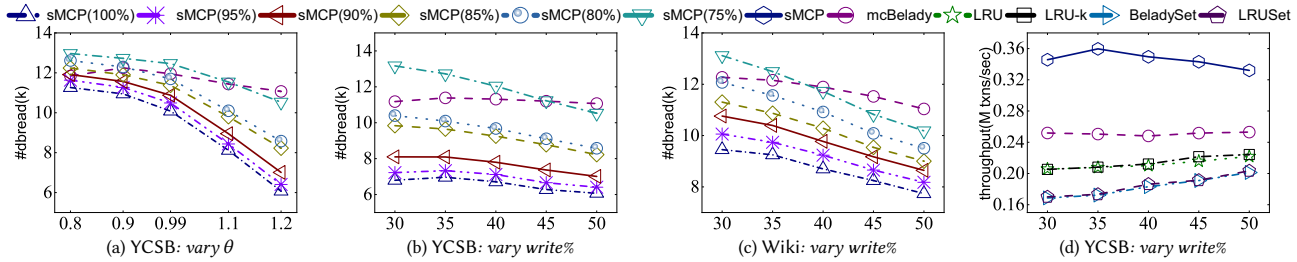


Figure 5: Experimental results under the semi-online model

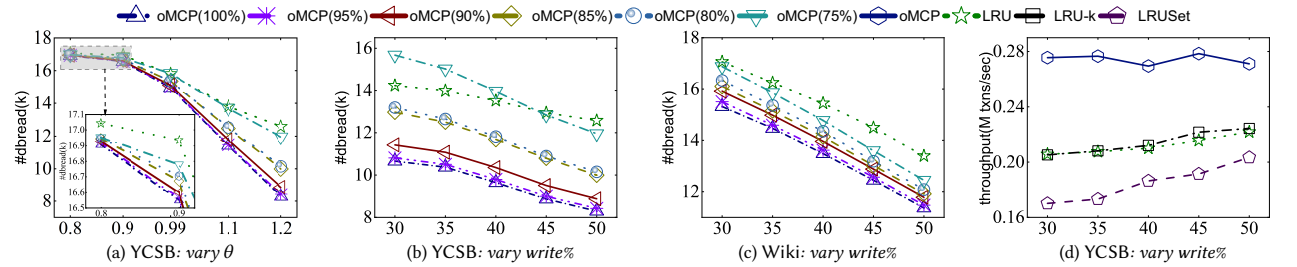


Figure 6: Experimental results under the online model

7.3 System Evaluation: A Proof of Concept

As a proof of concept, we developed a simple binary classification model $\mathcal{M}_o$ for the ML module of MCCache to decide obsolete items, and deployed `sMCP` and `oMCP` with $\mathcal{M}_o$, as well as `bMCP`. Using HBase as the backend database, we evaluated the throughput of MCCache atop Redis and Memcached (Redis by default), with different cache policies over YCSB and Wiki under all three input models. Below we first sketch the design of $\mathcal{M}_o$ (see Appendix C of full version [4] for more details). We then present our findings.

A classification model $\mathcal{M}_o$. $\mathcal{M}_o$ uses LightGBM [45], which is an implementation of the Gradient Boosting Decision Tree (GBDT) framework [32] with strong generalization properties over tabular data. $\mathcal{M}_o$ uses three types of features: (a) delta features that record, for each data item d, the distance between consecutive requests containing d; (b) set-request features that encode the data items of each request; and (c) frequency features that keep track of the occurrence frequency of the data items in the requests. All the features are computed using a sliding window of 2000 requests over the training set, of which the labels are computed via `OB` of Algorithm 1.

Throughput. To quantify the effectiveness of all cache policies via throughput, we run multiple cache threads of requests from the Redis nodes to keep the HBase node saturated. We varied the number of threads (`#-thds`) from 800 to 1200 (1000 by default) by controlling the number of cache nodes. Each request thread was run for 1 hour and the throughput of the overall system, *i.e.,* the number of requests processed per second, was calculated. The throughput is based on the end-to-end processing time, including the overhead of classification by $\mathcal{M}_o$. Key results under the batch, semi-online and online models are reported in Figures 4i-4l, Fig. 5d and Fig. 6d, respectively.

(1) With `bMCP` the system throughput is consistently the highest among all policies. For instance, over YCSB with `Lazy`, the throughput of `bMCP` is on average 52.80%, 80.81%, 79.06%, 109.82% and 107.97% higher than mcBelady, LRU-k, LRU, BeladySet and LRUSet, respectively, when write% varies from 30% to 50% (Fig. 4i).

(2) The throughput of all cache policies decreases when `vsize` increases (*e.g.,* Fig. 4j), due to higher cost per read/write operation. However, the improvement of our policies over competitors remains stable, *e.g.,* consistently around 83.01% over YCSB when `vsize` varies from 1KB to 1024KB. In contrast, cache policies are insensitive to `dbsize` (*e.g.,* Fig. 4k), since the number of reads to HBase depends on workloads and cache size, independent of `dbsize`.

(3) As shown in Fig. 4l, most cache policies benefit from increased threads (larger `#-thds`) and `bMCP` has the highest throughput in all cases. However, the throughput of BeladySet and LRUSet degrades when `#-thds` is larger than 1100. This is due to their worse performance (larger #dbread) than the others, which leads to a large number of read shifted to the database, causing higher contention.

(4) Our example classification model $\mathcal{M}_o$ has an average prediction accuracy of 92.35%. Nonetheless, we find that `sMCP` and `oMCP` with $\mathcal{M}_o$ still outperform their semi-online and online competitors, *e.g.,* by 68.17% and 35.51%, respectively, on average over YCSB with `Lazy`, up to 108.77% and 61.99% (Figures 5d and 6d). This shows that `sMCP` and `oMCP` can easily incorporate plugged-in coarsely designed classifiers and achieve higher system throughput than other policies.

8 Conclusion

We have proposed MCC, a cache scheme for set-based requests from applications that demand consistency and monotonicity. We have formulated the problem of MCC policy design, settled its complexity, and characterized optimal MCC policies. Based on the characterization, we have developed an optimal MCC policy under the batch model, and ML-augmented MCC policies for the semi-online and online models with provably competitiveness and ML-robustness guarantees. We have developed MCCache, a tool that adds MCC policies to existing caches, *e.g.,* Redis and Memcached. Our experimental study has verified that the policies are effective in reducing reads to the database and improving system throughput.

One topic for future work is to incorporate ML predictors on request sequences [16, 51, 64, 67, 74] to further improve cache performance. Another topic is to consider the per-client monotonicity.

References

[1] Mysql cluster cge. https://www.mysql.com/products/cluster/, 2016.

[2] What consistency guarantees should you expect from your streaming data platform? https://materialize.com/blog-consistency/, 2020.

[3] Foshttpcache: An introduction to cache invalidation. https://foshttpcache.readthedocs.io/en/stable/invalidation-introduction.html, 2021.

[4] Full version. https://homepages.inf.ed.ac.uk/ycao/MCCfull.pdf, 2022.

[5] Materialize. https://materialize.com/, 2022.

[6] Memcached. https://memcached.org/, 2022.

[7] Redis. https://redis.io/, 2022.

[8] Scaling with redis cluster. https://redis.io/docs/manual/scaling/, 2022.

[9] D. J. Abadi and J. M. Faleiro. An overview of deterministic database systems. *Commun. ACM*, 61(9):78–88, 2018.

[10] A. Agiwal and K. L. et al. Napa: Powering scalable data warehousing with robust query performance at google. *Proc. VLDB Endow.*, 14(12):2986–2998, 2021.

[11] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, 2013.

[12] S. Albers. Online algorithms: a survey. *Mathematical Programming*, 97(1):3–26, 2003.

[13] S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *SODA*, pages 31–40, 1999.

[14] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *SOCC 2013*, pages 23:1–23:10, 2013.

[15] S. An, Y. Cao, and W. Zhao. Competitive consistent caching for transactions. In *ICDE*, 2022.

[16] A. Antoniadis, C. Coester, M. Eliás, A. Polak, and B. Simon. Online metric algorithms with untrusted predictions. In *ICML*, 2020.

[17] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *EuroSys 2015*, pages 6:1–6:16. ACM, 2015.

[18] L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, 1966.

[19] D. Berend, S. Dolev, and M. Kogan-Sadetsky. Adaptiveclimb: adaptive policy for cache replacement. In *SYSTOR*, 2019.

[20] N. Bronson, Z. Amsden, and G. C. et al. TAO: facebook's distributed data store for the social graph. In *ATC*, pages 49–60, 2013.

[21] M. Chrobak, H. J. Karloff, T. H. Payne, and S. Vishwanathan. New results on server problems. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1990.

[22] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu. Caching is hard - even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, editors, *SoCC*, pages 143–154. ACM, 2010.

[24] A. Davoudian, L. Chen, and M. Liu. A survey on nosql stores. *ACM Comput. Surv.*, 51(2):40:1–40:43, 2018.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP, 2007*, pages 205–220. ACM, 2007.

[26] S. Dernbach, N. Taft, J. Kurose, U. Weinsberg, C. Diot, and A. Ashkan. Cache content-selection policies for streaming video services. In *INFOCOM 2016*, pages 1–9. IEEE, 2016.

[27] I. Eyal, K. Birman, and R. van Renesse. Cache serializability: Reducing inconsistency in edge transactions. In *ICDCS*, pages 686–695, 2015.

[28] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. I. Kat. It's time to revisit LRU vs. FIFO. In *HotStorage 2020*, 2020.

[29] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[30] A. Fiat, Y. Rabani, and Y. Ravid. Competitive k-server algorithms. *J. Comput. Syst. Sci.*, 48(3):410–428, 1994.

[31] L. Folwarczný and J. Sgall. General caching is hard: Even with small pages. *Algorithmica*, 79(2):319–339, 2017.

[32] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[33] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Strong and efficient consistency with consistency-aware durability. In S. H. Noh and B. Welch, editors, *FAST, 2020*, pages 323–337.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[35] S. Ghandeharizadeh, Y. Alabdulkarim, and H. Nguyen. Rangeqc: A framework for caching range predicate query results. In *SoCC*, 2018.

[36] S. Ghandeharizadeh and J. Yap. Gumball: a race condition prevention technique for cache augmented SQL database management systems. In D. Barbosa, K. LeFevre, and E. Terzi, editors, *DBSocial@SIGMOD*, 2012.

[37] S. Ghandeharizadeh and J. Yap. Cache augmented database management systems. In *DBSocial@SIGMOD*, 2013.

[38] S. Ghandeharizadeh, J. Yap, and H. Nguyen. Strong consistency in cache augmented SQL systems. In *Middleware*, pages 181–192. ACM, 2014.

[39] A. Girault, G. Gößler, R. Guerraoui, J. Hamza, and D. Seredinschi. Monotonic prefix consistency in distributed systems. In *FORTE*, 2018.

[40] P. Gupta, N. Zeldovich, and S. Madden. A trigger-based middleware cache for orms. In *Middleware*, 2011.

[41] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, 2017.

[42] V. Holmqvist, J. Nilsfors, and P. Leitner. Cachematic - automatic invalidation in application-level caching systems. In *ICPE*, 2019.

[43] S. Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.

[44] T. Johnson and D. E. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, 1994.

[45] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NeuIPS*, pages 3146–3154, 2017.

[46] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *2015 IEEE International Conference on Big Data*, pages 2785–2792, 2015.

[47] Z. Li, A. C. Begen, J. Gahm, Y. Shan, B. Osler, and D. Oran. Streaming video over HTTP with consistent quality. In R. Zimmermann, editor, *Multimedia Systems Conference 2014*, pages 248–258. ACM, 2014.

[48] Q. Liu, G. Wang, and J. Wu. Consistency as a service: Auditing cloud consistency. *IEEE Trans. Netw. Serv. Manag.*, 11(1):25–35, 2014.

[49] H. Lu, C. Hodsdon, K. Ngo, S. Mu, and W. Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *OSDI*, pages 135–150, 2016.

[50] H. Lu, S. Sen, and W. Lloyd. Performance-optimal read-only transactions. In *OSDI*, pages 333–349, 2020.

[51] T. Lykouris and S. Vassilvitskii. Competitive caching with machine learned advice. *J. ACM*, 68(4):24:1–24:25, 2021.

[52] K. Ma and B. Yang. Access-aware in-memory data cache middleware for relational databases. In *HPCC*, 2015.

[53] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990.

[54] F. McSherry, A. Lattuada, M. Schwarzkopf, and T. Roscoe. Shared arrangements: practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, 2020.

[55] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*. USENIX, 2003.

[56] R. M. Meloca and I. Nunes. A comparative study of application-level caching recommendations at the method level. *Empir. Softw. Eng.*, 27(4):88, 2022.

[57] J. Mertz, I. Nunes, L. D. Toffola, M. Selakovic, and M. Pradel. Satisfying increasing performance requirements with caching at the application level. *IEEE Softw.*, 2021.

[58] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SIGOPS*. ACM, 2013.

[59] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *NSDI 13*, pages 385–398, 2013.

[60] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In P. Buneman and S. Jajodia, editors, *SIGMOD*, pages 297–306. ACM Press, 1993.

[61] L. Phillips and B. Fitzpatrick. Livejournal's backend and memcached: Past, present, and future. In *LISA*. USENIX, 2004.

[62] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, pages 279–292. USENIX Association, 2010.

[63] G. Prasaad, A. Cheung, and D. Suciu. Handling highly contended OLTP workloads using fast dynamic partitioning. In *SIGMOD*, pages 527–542, 2020.

[64] D. Rohatgi. Near-optimal bounds for online caching with machine learned advice. In *SODA*, 2020.

[65] X. Shi, S. Pruett, K. Doherty, J. Han, D. Petrov, J. Carrig, J. Hugg, and N. Bronson. Flighttracker: Consistency across read-optimized online stores at facebook. In *OSDI*, 2020.

[66] Z. Shi, X. Huang, A. Jain, and C. Lin. Applying deep learning to the cache replacement problem. In *MICRO*, 2019.

[67] Z. Song, D. S. Berger, K. Li, and W. Lloyd. Learning relaxed belady for content distribution network caching. In *NSDI*, 2020.

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

[69] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS 94*, pages 140–149. IEEE Computer Society, 1994.

[70] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Fast dis-

tributed transactions and strongly consistent replication for OLTP database systems. *ACM Trans. Database Syst.*, 39(2):11:1–11:39, 2014.

[71] A. Z. Tomsic, M. Bravo, and M. Shapiro. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Middleware*, pages 120–133, 2018.

[72] M. Van Steen and A. S. Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[73] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[74] A. Wei. Better and simpler learning-augmented online caching. In *APPROX/RANDOM*, 2020.

[75] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *OSDI*, pages 191–208. USENIX Association, 2020.