# FASTER: A Concurrent Key-Value Store with In-Place Updates

Badrish Chandramouli[†], Guna Prasaad[‡*], Donald Kossmann[†], Justin Levandoski[†], James Hunter[†], Mike Barnett[†]

[†]Microsoft Research [‡]University of Washington

badrishc@microsoft.com, guna@cs.washington.edu, {donaldk, justin.levandoski, jahunter, mbarnett}@microsoft.com

## ABSTRACT

Over the last decade, there has been a tremendous growth in data-intensive applications and services in the cloud. Data is created on a variety of edge sources, e.g., devices, browsers, and servers, and processed by cloud applications to gain insights or take decisions. Applications and services either work on collected data, or monitor and process data in real time. These applications are typically update intensive and involve a large amount of state beyond what can fit in main memory. However, they display significant temporal locality in their access pattern. This paper presents FASTER, a new key-value store for point read, blind update, and read-modify-write operations. FASTER combines a highly cache-optimized concurrent hash index with a *hybrid log*: a concurrent log-structured record store that spans main memory and storage, while supporting fast in-place updates of the hot set in memory. Experiments show that FASTER achieves orders-of-magnitude better throughput – up to 160M operations per second on a single machine – than alternative systems deployed widely today, and exceeds the performance of pure in-memory data structures when the workload fits in memory.

**ACM Reference Format:**
Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196898

## 1 INTRODUCTION

There has recently been a tremendous growth in data-intensive applications and services in the cloud. Data created on a variety of edge sources such as Internet-of-Things devices, browsers, and servers is processed by applications in the cloud to gain insights. Applications, platforms, and services may work offline on collected data (e.g., in Hadoop [13] or Spark [42]), monitor and process data in real time as it arrives (e.g., in streaming dataflows [12, 42] and actor-based application frameworks [2]), or accept a mix of record inserts, updates, and reads (e.g., in object stores [30] and web caches [36]).

[*]Work performed during internship at Microsoft Research.

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196898

### 1.1 Challenges and Existing Solutions

State management is an important problem for all these applications and services. It exhibits several unique characteristics:

- ***Large State***: The amount of state accessed by some applications can be very large, far exceeding the capacity of main memory. For example, a targeted search ads provider may maintain per-user, per-ad and clickthrough-rate statistics for billions of users. Also, it is often cheaper to retain state that is infrequently accessed on secondary storage [18], even when it fits in memory.
- ***Update Intensity***: While reads and inserts are common, there are applications with significant update traffic. For example, a monitoring application receiving millions of CPU readings every second from sensors and devices may need to update a per-device aggregate for each reading.
- ***Locality***: Even though billions of state objects maybe alive at any given point, only a small fraction is typically "hot" and accessed or updated frequently with a strong temporal locality. For instance, a search engine that tracks per-user statistics (averaged over one week) may have a billion users "alive" in the system, but only have a million users actively surfing at a given instant. Further, the hot set may drift over time; in our example, as users start and stop browsing sessions.
- ***Point Operations***: Given that state consists of a large number of independent objects that are inserted, updated, and queried, a system tuned for (hash-based) point operations is often sufficient. If range queries are infrequent, they can be served with simple workarounds such as indexing histograms of key ranges.
- ***Analytics Readiness***: Updates to state should be readily available for subsequent offline analytics; for e.g., to compute average ad clickthrough-rate drift over time.

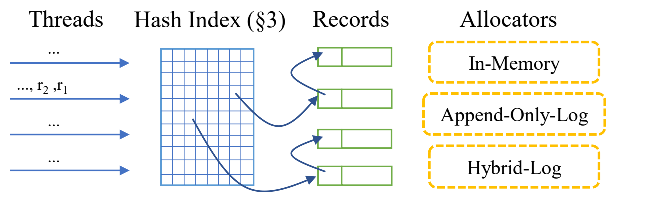
A simple solution adopted by many systems is to partition the state across multiple machines, and use pure in-memory data structures such as the Intel TBB Hash Map [5], that are optimized for concurrency and support *in-place updates* – where data is modified at its current memory location without creating a new copy elsewhere – to achieve high performance. However, the overall solution is expensive and often severely under-utilizes the resources on each machine. For example, the ad serving platform of a search engine may partition its state across the main memory of hundreds of machines, resulting in a low per-machine request rate and poorly utilized compute resources. Further, pure in-memory data structures make recovery from failures complicated.

Key-value stores are a popular alternative for state management. A key-value store is designed to handle *larger-than-memory* [28] data, and supports failure recovery by storing data on secondary storage. Many such key-value stores [25, 34, 40] have been proposed in the past. However, these systems are usually optimized for blind updates, reads, and range scans, rather than point operations and *read-modify-writes* (e.g., for updating aggregates). Hence, these systems do not scale to more than a few million updates per second [8], even when the hot-set fits entirely in main memory. Caching systems such as Redis [36] and Memcached [30] are optimized for point operations, but are slow and depend on an external system such as a database or key-value store for storage and/or failure recovery. The combination of concurrency, in-place updates (in memory), and ability to handle data larger than memory is important in our target applications; but these features are not simultaneously met by existing systems. We discuss related work in detail in Sec. 8.

### 1.2 Introducing FASTER

This paper describes a new concurrent key-value store called FASTER, designed to serve applications that involve update-intensive state management. The FASTER interface (Sec. 2.2 and Appendix E) supports, in addition to reads, two types of state updates seen in practice: *blind updates*, where an old value is replaced by a new value blindly; and *read-modify-writes* (*RMWs*), where the value is atomically updated based on the current value and an input (optional). RMW updates, in particular, enable us to support partial updates (e.g., updating a single field in the value) as well as mergeable aggregates [39] (e.g., sum, count). Being a point-operations store, FASTER achieves an in-memory throughput of 100s of million operations per second. Retaining such high performance while supporting data larger than memory required a careful design and architecture. We make the following contributions:

- (Sec. 2) Towards a scalable threading model, we augment standard epoch-based synchronization into a generic framework that facilitates lazy propagation of global changes to all threads via trigger actions. This framework provides FASTER threads with unrestricted access to memory under the safety of epoch protection. Throughout the paper, we highlight instances where this generalization helped simplify our scalable concurrent design.
- (Sec. 3 and 4) We present the design of a concurrent latch-free resizable cache-friendly hash index for FASTER. When coupled with a standard in-memory record allocator, it serves as an in-memory key-value store, which we found to be more performant and scalable than popular pure in-memory data structures.
- (Sec. 5 and 6) Log-structuring [24, 38] is a well-known technique for handling data larger than memory and supporting easy failure recovery. It is based on the *read-copy-update* strategy, in which updates to a record are made on a new copy on the log. We find that such a design could limit throughput and scalability in FASTER. As noted earlier, in-place updates are critical for performance. In this respect, we propose `HybridLog`: a new hybrid log that seamlessly combines in-place updates with a traditional append-only log. The `HybridLog` organization allows FASTER to perform in-place updates of "hot" records and use read-copy-updates for colder records. Further, it acts as an efficient cache by shaping what resides in memory without any per-record or per-page statistics. Doing so concurrently required us to solve novel technical challenges. Sec. 6.5 sketches the consistency guarantees of FASTER in the presence of failures.



| | Record Allocator | Concurrent & Latch-free | Larger-Than-Memory | In-Place Updates |
|---|---|---|---|---|
| §4 | In-Memory | ✓ | | ✓ |
| §5 | Append-Only-Log | ✓ | ✓ | |
| §6 | Hybrid-Log | ✓ | ✓ | ✓ |

**Figure 1: Overall FASTER architecture.**

- (Sec. 7) We perform a detailed evaluation using the YCSB [4] benchmark, evaluate the "up to orders-of-magnitude" better performance and near-linear scalability of FASTER with `HybridLog`. Further, we use simulations to better understand the natural caching behavior of `HybridLog` in comparison to more expensive, state-of-the-art caching techniques.

FASTER follows the design philosophy of *making the common case fast*. By carefully (1) providing fast point-access to records using a cache-efficient concurrent latch-free hash index; (2) choosing when and how, expensive or uncommon activities (such as index resizing, checkpointing, and evicting data from memory) are performed; and (3) allowing threads to perform in-place updates most of the time, FASTER exceeds the throughput of pure in-memory systems for in-memory workloads, while supporting data larger than memory and adapting to a changing hot set.

Implemented as a high-level-language component using dynamic code generation, FASTER blurs the line between traditional key-value stores and update-only "state stores" used in streaming systems (e.g., Spark State Store [14] and Flink [12]). Further, `HybridLog` is record-oriented and approximately time-ordered, and can be fed into scan-based log analytics systems (cf. Appendix F).

The main result shown in this paper with the help of FASTER and `HybridLog` is that it is possible to have it all: high update rates, low cost by limiting the memory footprint, support for larger-than-memory data, and performance that exceeds pure in-memory data structures when the working-set fits in memory.

## 2 SYSTEM OVERVIEW

FASTER is a concurrent latch-free key-value store that is designed for high performance and scalability across threads. We use latch-free atomic operations such as *compare-and-swap*, *fetch-and-add*, and *fetch-and-increment* (cf. Appendix A) extensively in our design, and heavily leverage an extended epoch-based synchronization framework (Sec. 2.3) to help us support in-place updates.

### 2.1 FASTER Architecture

Fig. 1 shows the overall architecture of FASTER. It consists of a *hash index* that holds pointers to key-value records and a *record allocator* that allocates and manages individual records. The index (Sec. 3) provides very efficient hash-based access to hash buckets. The hash bucket is a cache-line sized array of hash bucket entries. Each entry includes some metadata and an address (either logical or physical) provided by a record allocator. The record allocator stores and manages individual records. Hash collisions that are not resolved at the index level are handled by organizing records as a linked-list. Each record consists of a record header, key, and value. Keys and values may be fixed or variable-sized. The header contains some metadata and a pointer to the previous record in the linked-list. Note that keys are not part of the FASTER hash index, unlike many traditional designs, which provides two benefits:

- It reduces the in-memory footprint of the hash index, allowing us to retain it entirely in memory.
- It separates user data and index metadata, which allows us to mix and match the hash index with different record allocators.

We describe three allocators in this paper (the table in Fig. 1 summarizes their capabilities): an *in-memory* allocator (Sec. 4) that enables latch-free access and in-place updates to records; an *append-only* log-structured allocator (Sec. 5) that provides latch-free access and can handle data larger than main-memory, but without in-place updates; and finally a novel *hybrid-log* allocator (Sec. 6) that combines latch-free concurrent access with in-place updates and the ability to handle larger-than-memory data.

## 2.2 User Interface

FASTER supports reads, advanced user-defined updates, and deletes. We use dynamic code generation to integrate the update logic provided as user-defined delegates during compile time into the store, resulting in a highly efficient store with native support for advanced updates. We cover code generation in Appendix E. The generated FASTER runtime interface consists of the following operations:

- `Read`: Read the value corresponding to a key.
- `Upsert`: Replace the value corresponding to a key with a new value *blindly* (i.e. regardless of the existing value). Insert as new, if the key does not exist.
- `RMW`: Update the value of a key based on the existing value and an input (optional) using the update logic provided by the user during compile-time. We call this a *Read-Modify-Write (RMW)* operation. The user also provides an initial value for the update, which is used when a key does not exist in the store. Additionally, users can indicate that an `RMW` operation is *mergeable*, also called a *CRDT* [39] (for *conflict-free replicated datatype*) during compile time. Such a data type can be computed as partial values that can later be merged to obtain the final value. For example, a summation-based update can be computed as partial sums and these can be summed up for the final value.
- `Delete`: Delete a key from the store.

Further, some operations may go *pending* in FASTER due to several reasons covered in this paper. FASTER returns a `PENDING` status in such cases; threads issue a `CompletePending` request periodically to process outstanding pending operations related to that thread.

## 2.3 Epoch Protection Framework

FASTER is based on a key design principle aimed at scalability: *avoid expensive coordination between threads in the common fast access path.* FASTER threads perform operations independently with no synchronization most of the time. At the same time, they need to agree on a common mechanism to synchronize on shared system state. To achieve these goals, we extend the idea of *multi-threaded epoch protection* [23] into a framework enabling lazy synchronization over arbitrary global actions. While systems like Silo [41], Masstree [29] and Bw-Tree [25] have used epochs for specific purposes, we extend it to a generic framework that can serve as a building block for FASTER and other parallel systems. We describe epoch protection next, and depict its use as a framework in Sec. 2.4.

*Epoch Basics.* The system maintains a shared atomic counter $\mathbf{E}$, called the current epoch, that can be incremented by any thread. Every thread $T$ has a thread-local version of $\mathbf{E}$, denoted by $E_T$. Threads refresh their local epoch values periodically. All thread-local epoch values $E_T$ are stored in a shared *epoch table*, with one cache-line per thread. An epoch $c$ is said to be *safe*, if all threads have a strictly higher thread-local value than $c$, i.e., $\forall\, T:\ E_T > c$. Note that if epoch $c$ is safe, all epochs less than $c$ are safe as well. We additionally maintain a global counter $\mathbf{E_s}$, which tracks the current *maximal safe epoch*. $\mathbf{E_s}$ is computed by scanning all entries in the epoch table and is updated whenever a thread refreshes its epoch. The system maintains the following invariant: $\forall\, T:\ \mathbf{E_s} < E_T \leq \mathbf{E}$.

*Trigger Actions.* We augment the basic epoch framework with the ability to execute arbitrary global actions when an epoch becomes safe using *trigger actions*. When incrementing the current epoch, say from $c$ to $c+1$, threads can additionally associate an action that will be triggered by the system at a future instant of time when epoch $c$ is safe. This is enabled using the *drain-list*, a list of ⟨`epoch`, `action`⟩ pairs, where `action` is the callback code fragment that must be invoked after `epoch` is safe. It is implemented using a small array that is scanned for actions ready to be triggered whenever $\mathbf{E_s}$ is updated. We use atomic *compare-and-swap* on the array to ensure an action is executed exactly once. We recompute $\mathbf{E_s}$ and scan through the drain-list only when there is a change in current epoch, to enhance scalability.

## 2.4 Using the Epoch Framework

We expose the epoch protection framework using the following four operations that can be invoked by any thread $T$:

- `Acquire`: Reserve an entry for $T$ and set $E_T$ to $\mathbf{E}$
- `Refresh`: Update $E_T$ to $\mathbf{E}$, $\mathbf{E_s}$ to current maximal safe epoch and trigger any ready actions in the drain-list
- `BumpEpoch(Action)`: Increment counter $\mathbf{E}$ from current value $c$ to $(c+1)$ and add ⟨`c`, `Action`⟩ to drain-list
- `Release`: Remove entry for $T$ from epoch table

Epochs with trigger actions can be used to simplify lazy synchronization in parallel systems. Consider a canonical example, where a function `active-now` must be invoked when a shared variable `status` is updated to `active`. A thread updates `status` to `active` atomically and bumps the epoch with `active-now` as the trigger action. Not all threads will observe this change in `status` immediately. However, all of them are guaranteed to have observed it when they refresh their epochs (due to sequential memory consistency using memory fences). Thus, `active-now` will be invoked only after all threads see the status to be active and hence is safe.

We use the epoch framework in FASTER to coordinate system operations such as memory-safe garbage collection (Sec. 4), index resizing (Appendix B), circular buffer maintenance and page flushing (Sec. 5), shared log page boundary maintenance (Sec. 6.2), and

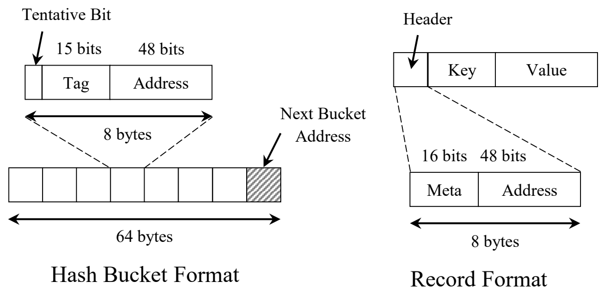


**Figure 2: Detailed FASTER index and record format.**

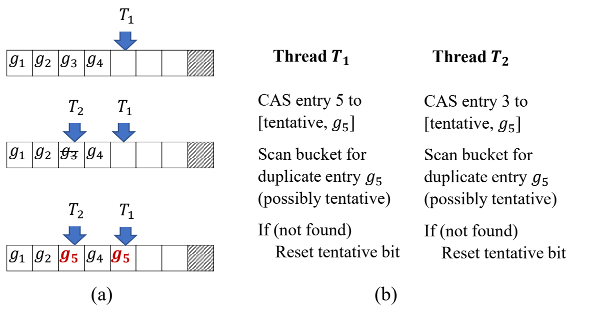

$T_1$
$g_1$ $g_2$ $g_3$ $g_4$
$T_2$ $T_1$
$g_1$ $g_2$ ~~$g_3$~~ $g_4$
$T_2$ $T_1$
$g_1$ $g_2$ $g_5$ $g_4$ $g_5$
(a)
Thread $T_1$
CAS entry 5 to [tentative, $g_5$]
Scan bucket for duplicate entry $g_5$ (possibly tentative)
If (not found) Reset tentative bit
Thread $T_2$
CAS entry 3 to [tentative, $g_5$]
Scan bucket for duplicate entry $g_5$ (possibly tentative)
If (not found) Reset tentative bit
(b)

**Figure 3: (a) Insert bug; (b) Thread ordering in our solution.**

checkpointing (Sec. 6.5), while at the same time providing FASTER threads unrestricted latch-free access to shared memory locations in short bursts for user operations such as reads and updates.

## 2.5 Lifecycle of a FASTER Thread

As our running example in the paper, we use FASTER to implement a *count store*, in which a set of FASTER user threads increment the counter associated with incoming key requests. A thread calls `Acquire` to register itself with the epoch mechanism. Next, it issues a sequence of user operations, along with periodic invocations of `Refresh` (e.g., every 256 operations) to move the thread to current epoch, and `CompletePending` (e.g., every 64K operations) to handle any prior pending operations. Finally, the thread calls `Release` to deregister itself from using FASTER.

## 3 THE FASTER HASH INDEX

A key building block of FASTER is the *hash index*, a concurrent, latch-free, scalable, and resizable hash-based index. As described in Sec. 2, the index works with a record allocator that returns logical or physical memory pointers. For ease of exposition, we assume a 64-bit machine with 64-byte cache lines. On larger architectures, we expect to have larger atomic operations [19], allowing our design to scale. In Sec. 4, 5, and 6, we will pair this index with different allocators to create key-value stores with increasing capabilities.

## 3.1 Index Organization

The FASTER index is a cache-aligned array of $2^k$ *hash buckets*, where each bucket has the size and alignment of a cache line (Fig. 2). Thus, a 64-byte bucket consists of seven 8-byte hash bucket *entries* and one 8-byte entry to serve as an overflow bucket pointer. Each overflow bucket has the size and alignment of a cache line as well, and is allocated on demand using an in-memory allocator.

The choice of 8-byte entries is critical, as it allows us to operate latch-free on the entries using 64-bit atomic compare-and-swap operations. On a 64-bit machine, physical addresses typically take up fewer than 64 bits; e.g., Intel machines use 48-bit pointers. Thus, we can *steal* the additional bits for index operations (at least one bit is required for FASTER). We use 48-bit pointers in the rest of the paper, but note that we can support pointers up to 63 bits long.

Each hash bucket entry (Fig. 2) consists of three parts: a *tag* (15 bits), a *tentative bit*, and the *address* (48 bits). An entry with value 0 (zero) indicates an empty slot. In an index with $2^k$ hash buckets, the tag is used to increase the effective hashing resolution of the index from $k$ bits to $k + 15$ bits, which improves performance by reducing hash collisions. The hash bucket for a key with hash value $h$ is first identified using the first $k$ bits of $h$, called the offset of $h$. The next 15 bits of $h$ are called the tag of $h$. Tags only serve to increase the hashing resolution and may be smaller, or removed entirely, depending on the size of the address. The tentative bit is necessary for insert, and will be covered shortly.

## 3.2 Index Operations

The FASTER index is based on the invariant that *each (offset, tag) has a unique index entry*, which points to the set of records whose keys hash to the same offset and tag. Ensuring this invariant while supporting concurrent latch-free reads, inserts and deletes of index entries is challenging.

*Finding and Deleting an Entry.* Locating the entry corresponding to a key is straightforward: we identify the hash-bucket using $k$ hash bits, scan through the bucket to find an entry that matches the tag. Deleting an entry from the index is also easy: we use compare-and-swap to replace the matching entry (if any) with zero.

*Inserting an Entry.* Consider the case where a tag does not exist in the bucket, and a new entry has to be inserted. A naive approach is to look for an empty entry and insert the tag using a compare-and-swap. However, two threads could concurrently insert the same tag at two *different* empty slots in the bucket, breaking our invariant. As a workaround, consider a solution where every thread scans the bucket from left to right, and deterministically chooses the first empty entry as the target. They will compete for the insert using compare-and-swap and only one will succeed. Even this approach violates the invariant in presence of deletes, as shown in Fig. 3a. A thread $T_1$ scans the bucket from left to right and chooses slot 5 for inserting tag $g_5$. Another thread $T_2$ deletes tag $g_3$ from slot 3 in the same bucket, and then tries to insert a key with the same tag $g_5$. Scanning left to right will cause thread $T_2$ to choose the first empty entry 3 for this tag. It can be shown that this problem exists with any algorithm that independently chooses a slot and inserts directly: to see why, note that just before thread $T_1$ does a compare-and-swap, it may get swapped out and the database state may change arbitrarily, including another slot with the same tag.

While locking the bucket is a possible (but heavy) solution, FASTER uses a latch-free two-phase insert algorithm that leverages the *tentative* bit entry. A thread finds an empty slot and inserts the record with the tentative bit set. Entries with a set tentative bit are deemed invisible to concurrent reads and updates. We then re-scan the bucket (note that it already exists in our cache) to check if there is another tentative entry for the same tag; if yes, we back off and retry. Otherwise, we reset the tentative bit to finalize the insert. Since every thread follows this two-phase approach, we are guaranteed to maintain our index invariant. To see why, Fig. 3b shows the ordering of operations by two threads: there exists no interleaving that could result in duplicate non-tentative tags.

### 3.3 Resizing and Checkpointing the Index

For applications where the number of keys may vary significantly over time, we support resizing the index on-the-fly. We leverage epoch protection and a state machine of phases to perform resizing at low overhead (cf. Appendix B). Interestingly, the use of latch-free operations always maintains the index in a consistent state even in the presence of concurrent operations. This allows us to perform an asynchronous fuzzy checkpoint of the index without obtaining read locks, greatly simplifying recovery (cf. Sec. 6.5).

## 4 AN IN-MEMORY KEY-VALUE STORE

We now build a complete in-memory key-value store using the Faster hash index from Sec. 3, along with a simple in-memory allocator such as `jemalloc` [1]. Records with the same (offset, tag) value are organized as a reverse singly-linked-list. The hash bucket entry points to the tail (most recent record) in the list, which in turn points to the previous record, and so on (see Fig. 1). Each record may be fixed- or variable-sized, and consists of a 64-bit record header, the key, and the value. The record header is shown in Fig. 2. Apart from the previous pointer, we use a few bits (invalid and tombstone) to keep track of information that is necessary for log-structured allocators (cf. Sec. 5 and 6). These bits are stored as part of the address word, but may be stored separately as well, if necessary.

### Operations with In-Memory Allocator

In Faster, user threads read and modify record values in the safety of epoch protection, with record-level concurrency handled by the user's read or update logic. For example, one could use fetch-and-add for counters, take a record-level lock, or leverage application-level knowledge of partitioning for latch-free updates. Operations on the store are described next.

*Reads.* We find the matching tag entry from the index as described in Sec. 3.2, and then traverse the linked-list for that entry to find a record with the matching key.

*Updates and Inserts.* Both blind updates (upserts) and RMW updates begin by finding the hash bucket entry for the key. If the entry does not exist, we use the two-phase algorithm described in Sec. 3.2 to insert the tag along with the address of the new record, into an empty slot in the hash bucket. If the entry exists, we scan the linked-list to find a record with a matching key. If such a record exists, we perform the operation in-place. A thread has guaranteed access to the memory location of a record, as long as it does not refresh its epoch. This property allows threads to update a value in-place without worrying about memory safety. If such a record does not exist, we splice in the new record into the tail of the list using a compare-and-swap. In our count store example, we increment the counter value for an existing key, using either

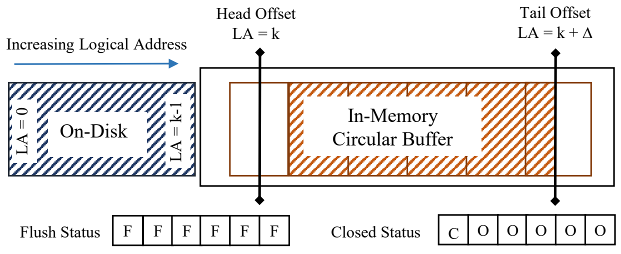


**Figure 4: Tail Portion of the Log-Structured Allocator**

a fetch-and-increment or a normal in-place increment (if keys are partitioned). The initial value for the insert of a new key is set to 0.

*Deletes.* We delete a record by atomically splicing it out of the linked-list using a compare-and-swap on either a record header or hash bucket entry (for the first record). When deleting the record from a singleton linked-list, the entry is set to 0, making it available for future inserts. A deleted record cannot be immediately returned to the memory allocator because of concurrent updates at the same location. We use our epoch protection framework to solve this problem: each thread maintains a thread-local (to avoid a concurrency bottleneck) free-list of (epoch, address) pairs. When the epochs become safe, we can safely return them to the allocator.

## 5 HANDLING LARGER DATA IN FASTER

As a strawman solution, we transform our in-memory Faster from Sec. 4 into a full-fledged key-value store that can handle data larger than memory by building a log-structured record allocator. Log structuring is a well researched area [24, 25, 34], and our approach is a straightforward adaptation of existing techniques, augmented with epoch protection for lower synchronization overhead. Being append-only, such a system does not perform well; we will show in Sec. 7.4.1 that it achieves a throughput of no more than 20 million operations per second, and does not scale with the number of threads. However, this design is useful as a building block to help explain our main contribution in Sec. 6, where we will get back scalable performance using a novel *hybrid log* allocator.

### 5.1 Logical Address Space

We start by defining a global logical address space that spans main memory and secondary storage. The record allocator allocates and returns 48-bit logical addresses corresponding to locations in this address space. Unlike our pure in-memory version, the Faster hash index now stores the logical address of a record instead of its physical address. The logical address space is maintained using a *tail offset*, which points to the next free address at the tail of the log. An additional offset, called the *head offset*, tracks the lowest logical address that is available in memory. The head offset is maintained at an approximately constant lag from the tail offset, equal to the memory available for the log. In order to minimize overhead, we update it only when the tail offset crosses page boundaries.

The contiguous address space between the current head and tail offsets (i.e., the tail portion of the log) is present in a bounded in-memory circular buffer, as shown in Fig. 4. The circular buffer is a linear array of fixed-size page frames, each of size $2^F$ bytes, that are each allocated sector-aligned with the underlying storage device,

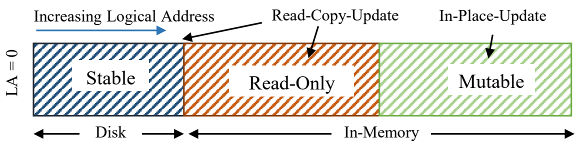


**Figure 5: Logical Address Space in `HybridLog`**

| Logical Address | Action |
|---|---|
| `Invalid` | Make a new record at tail-end |
| < `HeadOffset` | Issue Async IO Request |
| < `ReadOnlyOffset` | Make a mutable copy at tail-end |
| $< \infty$ | Update in-place |

**Table 1: Update scheme with Read-Only Marker**

in order to allow unbuffered reads and writes without additional memory copies. A logical address $L$ greater than the head address resides in main memory at offset equal to the last $F$ bits of $L$, in the page frame with position equal to $L \gg F$ in the circular array.

New record allocation always happens at the tail. We maintain the tail offset as two values in one word – a page number and an offset. For efficiency, a thread allocates memory using a fetch-and-add on the offset; if the offset corresponds to an allocation that would not fit on the current page, it increments the page number and resets the offset. Other threads that see a new offset greater than page size wait for the offset to become valid, and retry.

## 5.2 Circular Buffer Maintenance

We need to manage the off-loading of log records to secondary storage in a latch-free manner, as threads perform unrestricted memory accesses between epoch boundaries. To achieve this, we maintain two status arrays: a *flush-status* array tracks if the current page has been flushed to secondary storage, and a *closed-status* array determines if a page frame can be evicted for reuse. Since we always append to the log, a record is *immutable* once created. When the tail enters a new page $p + 1$, we bump the epoch with a flush trigger action that issues an asynchronous I/O request to flush page $p$ to secondary storage. This action is invoked only when the epoch becomes safe – because threads refresh epochs at operation boundaries, we are guaranteed that all threads would have completed writing to addresses in page $p$, and the flush is safe. When the asynchronous flush operation completes, the flush-status of the page is set to *flushed*.

As the tail grows, an existing page frame may need to be evicted from memory, but we first need to ensure that no thread is accessing the page. Traditional databases use a latch to pin pages in the buffer pool before every access so that it is not evicted when in use. For high performance, however, we leverage epochs to manage eviction. Recall that the head offset determines if a record is available in memory. To evict pages from memory, we increment the head offset and bump the current epoch with a trigger action to set the closed-status array entry for the older page frame. When this epoch is safe, we know that all threads would have seen the updated head offset value and hence would not be accessing those memory addresses. Note that we must ensure that the to-be-evicted page is completely flushed before updating the head offset, so that threads that need those records can retrieve it from storage.

## 5.3 Operations with Append-Only Allocator

Blind updates simply append a new record to the tail of the log and update the hash index using a compare-and-swap as before. If the operation fails, we simply mark the log record as invalid (using a header bit) and retry the operation. Deletes insert a tombstone record (again, using a header bit), and require log garbage collection (cf. Appendix C). Read and RMW operations are similar to their in-memory counterparts described in Sec. 4. However, updates are always appended to the tail of the log, and linked to the previous record. Further, logical addresses are handled differently. For a retrieved logical address, we first check if the address is more than the current head offset. If yes, the record is in memory and we can proceed as before. If not, we issue an asynchronous read request for the record to storage. Being a record log, we retrieve only the record and not the entire logical page. In our count store example, every counter increment results in appending the new counter to the tail of the log (reading the older value from storage if necessary), followed by a compare-and-swap to update the index entry.

Every user operation is associated with a context that is used to continue the operation when the I/O completes. Each Faster thread has a thread-local pending queue of contexts of all completed asynchronous requests issued by that thread. Periodically, the thread invokes a `CompletePending` function to dequeue these contexts and process the continuations. Note that the continuation may need to issue further I/O operations, e.g., for a previous logical address in the linked-list of records.

## 6 ENABLING IN-PLACE UPDATES IN FASTER

The log allocator design presented in the previous section, in addition to handling data larger than memory, enables a latch-free access path for updates due to its append-only nature. But this comes at a cost: every update involves atomic increment of the tail offset to create a new record, copying data from previous location and atomic replace of the logical address in the hash index. Further, an append-only log grows fast, particularly with update-intensive workloads, quickly making disk I/O a bottleneck.

On the other hand, in-place updates have several advantages in such workloads: (1) frequently accessed records are likely to be available in higher levels of cache; (2) access paths for keys of different hash buckets do not collide; (3) updating parts of a larger value is efficient as it avoids copying the entire record or maintaining expensive delta chains that require compaction; and (4) most updates do not need to modify the Faster hash index.

## 6.1 Introducing `HybridLog`

`HybridLog` is a novel data structure that combines in-place updates (in memory) and log-structured organization (on disk) while providing latch-free concurrent access to records. `HybridLog` spans memory and secondary storage, where the in-memory portion acts as a cache for hot records and adapts to a changing hot set.

In `HybridLog` the logical address space is divided into 3 contiguous regions: (1) *stable region* (2) *read-only region* and (3) *mutable region* as shown in Fig. 5. The stable region portion is on secondary storage. The in-memory portion is composed of read-only and

| T1 | T2 | T3 |
|---|---|---|
| Obtain L |  | Obtain L |
| L < R1 |  |  |
|  | R1 → R2 |  |
|  |  | L < R2 |
|  |  | L' : 5 |
| L: 4 → 5 |  |  |

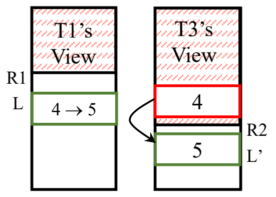


Figure 6: Lost Update Anomaly

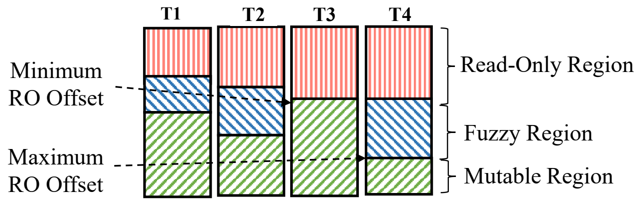


Figure 7: Thread Local View of Hybrid Log Regions

mutable regions. Records in the mutable region can be modified in-place, while records in the read-only region cannot. In order to update a record currently in the read-only region, we follow the Read-Copy-Update (RCU) strategy: a new copy is created in the mutable region and then updated. Further updates to such a record are performed in-place, as long as it stays in the mutable region.

We implement `HybridLog` on the log allocator from Sec. 5 using an additional marker called the *read-only offset*, that corresponds to a logical address residing in the in-memory circular buffer. The region between head-offset and read-only offset is the read-only region and the region after read-only offset is the mutable region. If a record is at a logical address more than read-only offset, it is updated in-place. If the address is between read-only and head offset, we create an updated copy at the end of tail and update the hash index to point to the new location; if the address is less than head-offset, it is not available in memory and hence an asynchronous IO request is issued to retrieve the record from secondary storage. Once it is obtained, a new updated copy is created at the end of tail followed by updating the hash index. This update scheme is summarized in Table 1.

The read-only offset is maintained at a constant lag from tail-offset and is updated only at page boundaries similar to the head-offset. Since none of the pages with logical address less than the read-only offset are being updated concurrently, it is safe to flush them to secondary storage. As tail-offset grows, read-only offset shifts along making pages ready to be flushed. Once they are safely offloaded to disk, they can be evicted from the circular buffer (when necessary) using the head-offset and closed-status array like in Sec. 5. Thus, the read-only offset serves as a light-weight indicator of pages that are ready to be flushed to disk. Note that the read-only offset in `HybridLog` enables latch-free access to records in the mutable region, whereas in traditional designs, records (or pages) must be pinned in the buffer pool before updating it to prevent concurrent updates while flushing them to disk.

The lag between read-only and tail offsets determines the division of main memory buffer capacity into fast in-place updatable and immutable read-only regions. In addition to helping flush pages safely to secondary storage, the read-only region also acts as a second-chance cache for records before being off-loaded to disk. We discuss the caching behavior and impact of sizing the hybrid log regions on FASTER performance in Sec. 6.4.

## 6.2 Lost-Update Anomaly

The read-only offset is updated and read atomically. However, it is still possible that a thread decides on the update scheme based on a stale value of the offset, leading to incorrect execution. Consider the scenario shown in Fig. 6, from our count store example. Threads $T_1$ and $T_3$ obtain the same logical address $L$ from the FASTER hash index. $T_1$ decides to do an in-place update as $L$ is more than the current read-only offset $R_1$. Meanwhile, a thread $T_2$ updates the read-only offset from $R_1$ to $R_2$ due to shifting of tail-offset. Now, thread $T_3$ compares $L$ with $R_2$ and decides to create a new record at $L'$ with the updated value of 5. However, thread $T_1$ updates the value to 5 at $L$. All future accesses will use the value at $L'$ and hence we have lost the update by $T_1$.

The above anomaly occurs because a thread $T_2$ updates read-only offset, while $T_1$ is acting based on the current value. We could prevent this by obtaining a read lock on read-only offset for the entire duration of $T_1$'s operation. However, such a locking scheme is expensive and unnecessarily delays shifting of read-only offset, which is integral to maintaining the circular buffer. On the other hand, even if the read-only offset has shifted, the anomaly occurs because one thread ($T_1$) makes an update decision based on a stale value and another ($T_2$) based on the new value of the offset.

We use another marker called the *safe read-only offset* to eliminate such incorrect executions. This marker tracks the read-only offset that has been seen by all the threads. It is designed based on the following invariant: safe read-only offset is the minimum value of read-only offset seen by any active FASTER thread. We maintain this using the epoch-trigger action mechanism as follows: whenever the read-only offset is updated, we bump the current epoch along with a trigger action that updates the safe read-only offset to the new value. This epoch-based update for safe read-only offset satisfies the invariant because all threads that crossed the current epoch must have seen the new value of read-only offset.

With an additional marker, the safe read-only offset, `HybridLog` is divided into 4 regions. We call the region between safe read-only and read-only offset as *fuzzy region*, because some threads might see it as after the read-only offset while some others may see it as before. Threads are guaranteed to obtain the latest values of safe read-only and read-only offsets only when they refresh their epochs. As a result, each thread might have a thread-local view of these markers as shown in Fig. 7. Thread $T_4$ has the highest value of read-only offset because it has refreshed its epoch recently, while $T_3$ has stale values as it has not refreshed recently. However, note that the safe read-only offset for any thread is at most the minimum read-only offset (thread $T_3$) and this is ensured by our epoch protection framework. When the logical address of a record is less than safe read-only, threads may try to create a new record concurrently and only one will succeed due to the atomic compare-and-swap operation on the FASTER hash index.

## 6.3 Fuzzy Region

When a record falls in the fuzzy region, interestingly, different types of updates can be handled differently. Here, we classify the types

| **Logical Address** | **Read-Modify-Write** | **CRDT Update** | **Blind Update** |
|---|---|---|---|
| `Invalid` | Create a new record at tail-end | Create a new record at tail-end | Create a new record at tail-end |
| < `HeadAddress` | Issue Async IO Request | Create a delta record at tail-end | |
| < `SafeReadOnlyAddress` | Add to pending list | | |
| < `ReadOnlyAddress` | Create an updated record at tail-end | | |
| < $\infty$ | Update in-place concurrently | Update in-place concurrently | Update in-place concurrently |

**Table 2: Update scheme for different types of updates**

of updates into three, namely *blind update, read-modify-write* and *CRDT update*. The update scheme for each of these update types is summarized in Table 2.

*Blind Update.* This update does not read the old value of a key. Even if one thread is updating a previous location in-place, another thread can create a new record at the end of tail with the new value. Since the updates are issued concurrently, semantics of the application must allow all possible serial orders. Further, we can avoid an expensive retrieval from the disk in case the record is not available in memory, as we do not need the old value.

*Read-Modify-Write.* This kind of update first reads and then updates a record based on the current value. Since we cannot be entirely sure that no other thread is updating a value concurrently, we cannot create a new copy at the end of tail precisely to avoid the lost-update anomaly discussed earlier. So, we defer the update by placing the context in a pending queue to be processed later, similar to how records on storage are handled.

*CRDTs.* CRDT updates are RMWs, but present an interesting middle-ground between blind updates and RMWs. Recall that CRDTs can be computed as independent partial values that can later be merged to obtain the final value. Our running example (count store) is a CRDT, as multiple partial counts can be summed to obtain the overall count value. With CRDT updates, we can handle the fuzzy region similar to blind updates. When a record is in the fuzzy region (or on disk), we simply create and link a new delta record at the tail, with the update performed on the initial (empty) value. A read has to reconcile all delta records to obtain the final converged value. One can imagine a scheme that periodically collapses deltas to maintain a bound on the length of delta chains.

## 6.4 Analysis of the Hybrid Log

*Cache Behavior and Shaping of the Log.* The in-memory portion of a key-value store acts like a cache and so performance heavily depends on its efficiency. Several caching protocols have been proposed in the context of buffer pool management in databases and virtual memory management in operating systems such as First-In First-Out (FIFO), CLOCK, Least Recently Used (LRU) and an extended version of LRU, the LRU-K[33] Protocol. All of them (except FIFO) require fine-grained per-page (or per-record) statistics to work efficiently. Interestingly, Faster is achieves a good caching behavior at a per-record granularity without any such overheads, by virtue of the access pattern. The hybrid in-place and copy update scheme of Faster results in efficient caching, quite similar to a Second-Chance FIFO protocol. We compare these protocols using a simulation in Sec. 7.5.

Faster shapes the log based on the access pattern and helps keep the hot items in memory. Consider a write-heavy workload on our count store example (Appendix D addresses other kinds of workloads). When a record is retrieved from disk for update, the new record with updated count is created at the end of tail. The record stays in memory and is available for in-place updates, until it enters the read-only region of the hybrid log. If a key is hot, it is likely that there is a subsequent request before it is evicted from memory resulting in a new mutable record. This serves as a second chance for the key to remain cached in memory. Otherwise, it is evicted to disk, making space for hotter keys in memory.

*Sizing the Hybrid Log Regions.* Sizing the mutable and read-only regions in the hybrid log allocator is important. One extreme (lag = 0) is an append-only store, while the other extreme (lag = *buffer-size*) is an in-memory store when data fits in memory. The size of read-only region determines the degree of *second chance* provided to a record to stay cached in memory. A smaller read-only (or larger mutable) region results in better in-memory performance due to in-place updates. However, a hot record might be evicted to disk simply because there was no access to that key for a very short time. A larger read-only region, on the other hand, results in expensive append-only updates, causing the log to grow faster. Further, it causes a replication of records in the read-only and mutable region effectively reducing the in-memory cache size. We observe that, in practice, an $90 : 10$ division of buffer size for the mutable and read-only regions result in good performance. We evaluate the performance impact of `HybridLog` region sizes in Sec. 7.4.2.

## 6.5 Recovery and Consistency in Faster

In the event of a failure, the unflushed tail of `HybridLog` is lost. However, Faster can recover to a database state that is consistent with respect to the *monotonicity* property: for any two update requests $r_1$ and $r_2$ issued (in order) by a thread, the state after recovery includes the effects of (1) none; (2) only $r_1$; or (3) both $r_1$ and $r_2$. In other words, the state after recovery cannot include the effects of $r_2$ without also including $r_1$. We can achieve this property using a *Write-Ahead-Log* (*WAL*) that logs all the modifications due to a request, similar to traditional databases and modern key-value stores such as RocksDB. Applications can periodically obtain a *fuzzy checkpoint* of Faster in memory, which can then be used in combination with the WAL to recover to a consistent state. Recovering from a fuzzy checkpoint using a WAL is a well-studied problem [32, 37], and hence we do not cover it in this paper.

*Eliminating the WAL.* Having a separate WAL could introduce a bottleneck for update-intensive workloads, so we have designed a recovery scheme for Faster that does not require a WAL. We sketch our solution briefly below, but leave a detailed treatment of recovery to future work. The basic idea is that we can treat `HybridLog` as our WAL, and delay commit in order to allow in-place updates within a limited time window.

*Checkpointing Faster.* While technically we can rebuild the entire hash-index from the HybridLog, checkpointing the index periodically allows faster recovery. All operations on the Faster index are performed using atomic compare-and-swap instructions. So, the checkpointing thread can read the index asynchronously without acquiring any read locks. However, since the hash index is being updated concurrently, such a checkpoint is fuzzy, and may not be consistent with respect to a location on the HybridLog. However, we can use the HybridLog to recover a consistent version of the hash index from this fuzzy checkpoint, as described next.

We record the tail-offset of the HybridLog before starting ($t_1$) and after completing ($t_2$) the fuzzy checkpoint. All updates to the hash index during this interval correspond only to records between $t_1$ and $t_2$ on the log, because in-place updates do not modify the index. However, some of these updates may be part of the fuzzy checkpoint and some may not. During recovery, we scan through the records between $t_1$ and $t_2$ on the HybridLog in order, and update the recovered fuzzy index wherever necessary. The resulting index is a consistent hash index that corresponds to HybridLog until $t_2$, because all updates to hash index entries after completing the fuzzy checkpoint (and recording the tail-offset $t_2$) correspond only to records after $t_2$ on the log.

Finally, by moving the read-only offset of the HybridLog to $t_2$, we get a checkpoint corresponding to location $t_2$ in the log, after the corresponding flush to disk is complete. Note that our checkpointing algorithm can be performed in the background without quiescing the database. Every such checkpoint in Faster is *incremental*, as we offload only data modified since the last checkpoint. Incremental checkpointing usually requires a separate bitmap-like data structure to identify data that needs to be flushed, whereas Faster achieves this by organizing data differently.

*Discussion.* The state after recovery using the above technique may violate monotonicity due to in-place updates: update $r_1$ can modify a location $l_1 \geq t_2$, whereas a later update $r_2$ may modify a location $l_2 < t_2$. Our checkpoint until $t_2$, that includes $l_2$ but not $l_1$, violates monotonicity. Interestingly, we can restore monotonicity by using epochs and triggers so that threads can collaboratively switch over to a new version of the database, as identified by a location on HybridLog – the details are left as future work.

## 7 EVALUATION

We evaluate Faster in four ways. First, we compare overall throughput and multi-thread scalability of Faster to leading key-value stores when the dataset fits in memory. Next, we perform experiments with data larger than main-memory, by varying the memory budget. Third, we perform micro-benchmarks to illustrate how the specific design choices of Faster and HybridLog. Finally, we use simulations to evaluate the caching behavior of Faster.

### 7.1 Implementation, Setup, Workloads

*Implementation.* We implemented Faster in C# as an embedded component that can be used with any application, and uses code generation to inline user functions for performance. Threads issue a sequence of operations for 30 secs, and we measure the number of operations completed during that period. We point Faster to a file on SSD to store the log. We assume an expiration-based garbage collection scheme (Appendix C) and do not include this cost in results. Costs related to checkpointing and recovery are not included as well. We size the in-place-updatable region at 90% of memory unless indicated otherwise. By default, we size the Faster index with #keys/2 hash bucket entries. *While Faster can be paired with in-memory allocators, all experiments in this paper use* HybridLog, *and thus represent the complete version of the Faster key-value store that can handle data larger than main-memory.*

*Setup.* Experiments are carried out on two identical machines, one running Windows Server 2018 (for Faster) and another running Ubuntu Linux (for other systems, since they are optimized for Linux). Both are Dell PowerEdge R730 machines with 2.60GHz Intel Xeon CPU E5-2690 v4 CPUs. They have 2 sockets and 14 cores (28 hyperthreads) per socket. They have 256GB RAM and a 3.2TB FusionIO NVMe SSD drive for the log. We pin threads to hardware cores where possible. The two-socket experiments shard threads across sockets for a smoother gradient with an increasing number of threads. We preload input datasets into memory and run each test for 30secs, except for RocksDB, which is described below.

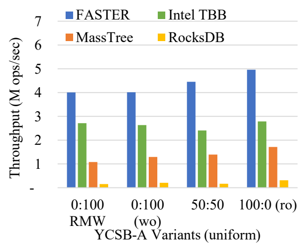
*Workloads.* We use an extended version of the YCSB-A workload from the Yahoo Cloud Serving Benchmark [4], with 250 million distinct 8 byte keys, and value sizes of 8 bytes and 100 bytes. Workloads are described as R:BU for the fraction of reads and blind updates in the workload. Further, we add *read-modify-write* (RMW) updates in addition to the blind updates supported by the benchmark. Such updates are denoted as 0:100 RMW in experiments (we only experiment with 100% RMW updates in this paper). RMW updates increment a value by a number from a user-provided input array with 8 entries, to model a running per-key "sum" operation.

Apart from the provided Uniform and Zipfian ($\theta = 0.99$) distributions, we add a new *hot set* distribution in some experiments, which models a hot and cold set of keys, with items moving from cold to hot, staying hot for a while, and then becoming cold. This distribution models users of a search engine, for example.

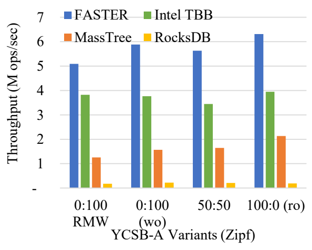
*Baseline Systems.* We compare Faster with two categories of systems: pure in-memory and larger-than-memory systems. In the pure in-memory category, we evaluate against Masstree [29], a high-performance pure in-memory range index, and Intel TBB concurrent hash map [5], a highly optimized pure in-memory hash index. In the category of systems that can handle large data, we evaluate against RocksDB [40] and Redis [36], two leading key-value stores. Note that we add comparisons to stores that employ range indices mainly because such systems are deployed widely even for point workloads, and are known to be highly optimized for main memory (Masstree) and larger-than-memory (RocksDB). We configured RocksDB with write-ahead logging and checksums disabled, with parameters as recommended on the RocksDB Wiki [9]. We used direct I/O for reads and writes, and ran each test for 10 minutes. With MassTree, we used the default configuration, pinning software threads to cores. With Intel TBB, we stored values in-line in the hash map. Redis is evaluated separately in Sec. 7.2.4.

### 7.2 Comparison to Existing Systems

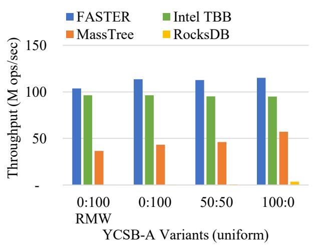
7.2.1 *Single Thread Performance.* For a single thread, we use YCSB (8 byte payloads) with a 100% RMW workload, as well as



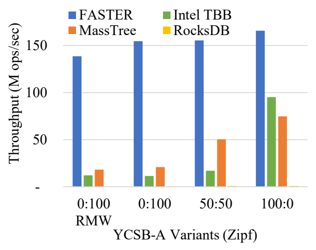
**(a) Single thread; uniform distr.**



**(b) Single thread; Zipf distr.**

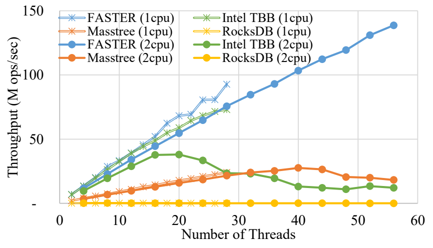


**(c) All threads; uniform distr.**

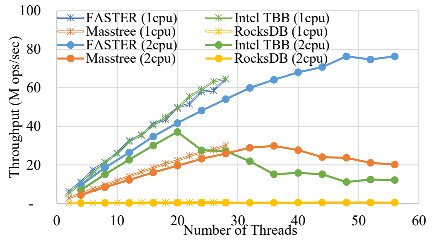


**(d) All threads; Zipf distr.**

**Figure 8: Throughput comparison of FASTER to other systems, YCSB dataset fitting in memory.**



**(a) RMW updates; 8-byte payloads.**



**(b) Blind updates; 100-byte payloads.**

**Figure 9: Scalability with increasing #threads, YCSB dataset fitting in memory.**

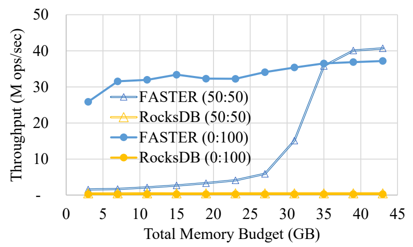


**Figure 10: Throughput with increasing memory budget, for 27GB dataset.**

varying read percentages with blind updates, with the dataset fitting in memory. The results for uniform and Zipf are shown in Fig. 8a and Fig. 8b respectively. We note that FASTER is able to achieve very high single-threaded throughput, which makes it a good fit in embedded environments. Further, FASTER, which handles data larger than memory, outperforms pure in-memory systems as well.

*7.2.2 All Threads Performance.* We use all 56 threads (on two sockets), and compare the systems with the same workload as before. The results for uniform and Zipf are shown in Fig. 8c and Fig. 8d respectively. FASTER is able to achieve a throughput of up to 115M ops/sec for uniform, and 165M ops/sec for a Zipfian workload. Interestingly, Intel TBB hash does well with uniform, but faces some contention with the Zipf distribution and is unable to scale. Other systems show much lower performance, as expected.

We also ran experiments varying the size of the tag in the index, to check its impact on throughput. Briefly, for the YCSB $50 : 50$ uniform workload on all threads, we found that even with a tag of just 1 bit (or 4 bits), performance decreased by less than 14% (or 5%), verifying that FASTER can robustly handle larger address sizes.

*7.2.3 Scalability.* We plot performance with a Zipf distribution, with an increasing number of threads, using one CPU and using both CPUs. We first evaluate a 100% RMW YCSB workload with 8 byte payloads in Fig. 9a. We see FASTER scales very well on both one CPU and two CPUs. Masstree (a pure in-memory range index) also scales well, but has much lower absolute performance. The Intel TBB hash map scales well within one CPU, but falls over at around 20 cores when running on two CPUs, likely because of locking contention with the Zipf workload. We next evaluate a $0 : 100$ blind upsert workload with 100 byte payloads in Fig. 9b. In this case, performance is linear until 48 threads on two CPUs, but levels off after this point because the larger 100 byte payloads cause the system to reach the maximum cross-socket available bandwidth.

*7.2.4 Comparison to Redis.* Redis is an in-memory key-value store (or cache) that differs from FASTER in three ways:

- Redis is not concurrent, and the user is expected to incur the overhead of hash partitioning the data and operations.
- While Redis offers optional recoverability, it expects that all data will fit in memory and snapshots the database using `fork`.
- Redis is designed to be accessed over a network, and as such is designed with this bottleneck in mind. Thus, its performance is expected to be lower than embedded systems such as FASTER.

We investigated the last point by running `redis-benchmark` on a single thread, and with the following parameters: `-d 8 -c 10 -P ${PIPELINE} -t set,get -r 1000000 -n 20000000`. The values are 8 bytes, we use 10 client threads, connected to `localhost` (to avoid network overhead), and ran 20M get and set operations, which access random records in the range between 0 and 1M. We varied the pipeline (batching) depth from 1 to 200. In this simplified scenario, we saw around 1.1M sets/sec and 1.4M gets/sec. For a key space of 250M (similar to our YCSB workload), we saw around 700K sets/sec and 900K gets/sec. These speeds are significantly lower than those for single-threaded FASTER.

## 7.3 Larger-than-Memory Experiments

We run YCSB with 100 byte payloads, for a core dataset size of 27GB, and set the number of threads to 14 (7 cores) on one socket. The memory budget for FASTER includes 2GB of space for the hash index, which is sized at #key/8 hash buckets for this experiment.

First, we use a $50 : 50$ Zipf workload, and plot throughput vs. RocksDB in Fig. 10. As expected, FASTER slows down with limited memory because of increased random reads from SSD, but quickly reaches in-memory performance levels once the entire dataset fits in memory. We believe that the steep performance drop-off as we reduce memory can be improved by optimizing the I/O path in FASTER; we plan to address this problem in future work.

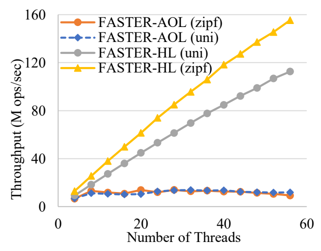
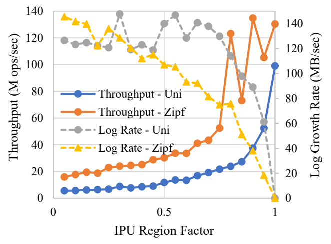
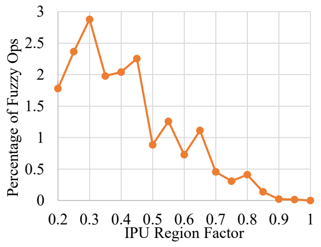


Figure 11: Throughput with append-only vs. hybrid logs.



(a) Throughput & log growth rate.



(b) Percentage of fuzzy ops.

Figure 12: Effect of increasing IPU region.

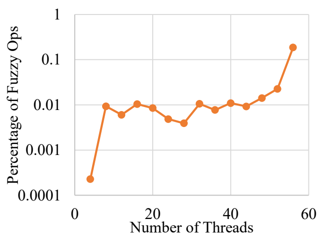


Figure 13: Percentage of fuzzy ops with increasing #threads.

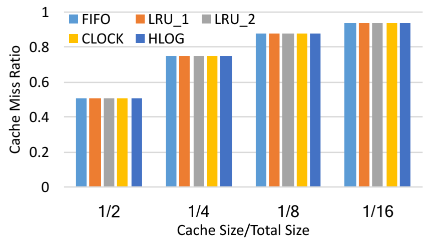


Figure 14: Cache miss ratio (Uniform).

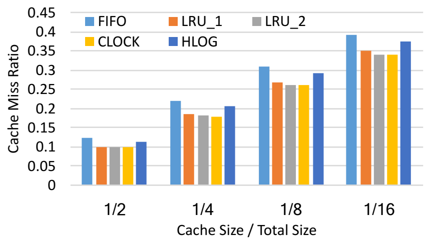


Figure 15: Cache miss ratio (Zipf).

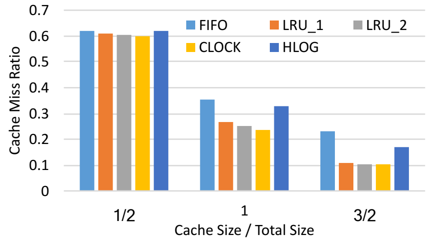


Figure 16: Cache miss ratio (Hot Set).

Next, with a 0 : 100 blind update workload, performance of FASTER is slightly lower with enough memory, as expected. As the memory budget drops, more data spills to storage, but throughput does not drop by as much as it does for reads, because of the higher efficiency of bulk sequential log writes (with no random reads) to SSD. We see that RocksDB achieves around 500K operations per second at best, for both these workloads.

Finally, we use a 0 : 100 workload with 80% read-only region and a uniform distribution. This stresses the write throughput of the system because of fewer in-place updates. We found FASTER to reach a sequential log write bandwidth of 1.74GB/sec, close to the theoretical maximum of 2GB/sec for our SSD drive.

## 7.4 Detailed Evaluation of FASTER

*7.4.1 Comparison to Append-Only.* We run YCSB-A 50 : 50 (50% reads and 50% blind updates) with the FASTER append-only log allocator. Our circular in-memory buffer has $2^{15}$ pages, and each page is 4MB in size. Fig. 11 shows the performance for uniform and Zipf distributions. We see that while performance scales linearly with the hybrid log, the creation of new entries in the log and the contention on the tail result in the append-only log being significantly slower. In case of hybrid log, Zipf performs much better than uniform, because the key skew results in better TLB and cache behavior. On the other hand, with append only, the performance benefit of a Zipf distribution is outweighed by more failed compare-and-swap operations, because of conflicting updates.

*7.4.2 Size of IPU region.* We use a YCSB-A with 100% RMW workload, and vary the *IPU Region Factor*, defined as the fraction of the dataset that is in the in-place-update (IPU) region of the log. Fig. 12a shows overall throughput achieved (on 56 threads) and the rate at which the log grows (on the secondary axis), as we increase the IPU Region Factor. With a uniform distribution of keys, we see that throughput increases as we increase the IPU region, because we get more opportunities to perform in-place updates. Further, the log grows more slowly, which is highly desirable. Note that we show log growth rate for 8 byte keys and values, for a total of 24 bytes per record, but the growth rate will multiply for larger payloads. Finally, the plots for a Zipfian key distribution indicate that due to the concentration of keys, throughput is high at even lower IPU region factors than that for uniform, due to the shaping effect of the hybrid log. The log growth rate declines more rapidly as well (with increasing IPU region size), due to the same reason.

*7.4.3 Fuzzy Region.* We execute a 100% RMW workload (uniform) on all 56 threads, and vary the IPU region size from 0.25 to 1.0 of the database, which causes the log to grow faster, and in turn results in the `ReadOnlyAddress` moving faster, which should result in more updates going pending in the fuzzy region. Fig. 12b shows the percentage of fuzzy updates with increasing IPU region size. Threads refresh their epochs every 256 operations. Even with a uniform key distribution and 56 threads, this percentage rises no more than 3%, and is higher than 0.5% only in the artificial scenario where less than 70% of memory is used for in-place updates.

Next, we keep the log growth rate fixed by fixing the IPU Region factor at 0.8, and vary the number of threads. As expected, Fig. 13 shows that the percentage of fuzzy operations increases with the number of threads, but stays below 1% even with all 56 threads.

## 7.5 Simulation of Caching Behavior

We perform a simulation study to compare well-known caching protocols (Sec. 6.4) with the caching behavior of `HybridLog` (HLOG). We maintain a constant-sized key buffer as a cache, and use each caching protocol to evict a key whenever an accessed key is not in the buffer. For HLOG, we have a read-only marker that is at a constant lag from the tail address; when a key is in read-only region, we copy it to end of tail like in FASTER. We vary the cache size and observe cache hit rate for 3 access patterns: uniform, zipfian ($\theta = 0.99$) and the *hot-set* distribution. The hot set distribution consists of a shifting hot-set (1/5th total size) uniformly accessed with 90% probability and the remaining cold-set accessed uniformly with a 10% probability. Our results are shown in Fig. 14, Fig. 15 and

Fig. 16. HLOG performs as well as other protocols for the uniform distribution. In case of zipfian and hot-set distributions, HLOG's cache miss rate is higher than LRU-1, LRU-2 and CLOCK protocols due to replication of keys in memory. The HLOG protocol results in two copies of hot keys one within the read-only region and one in the mutable region, thus reducing the effective cache size. However, HLOG is better than simple FIFO as it provides keys a second chance to stay in the cache. Overall HLOG's caching behavior is competent with other optimized algorithms without requiring to maintain extensive statistics, unlike other protocols (except FIFO), while still providing a latch-free fast-access path.

## 8 RELATED WORK

FASTER contributes to a richly researched space of concurrent system designs, data structures, key-value stores, file systems, and databases. We summarize the current state-of-the-art next.

*In-Memory Designs and Structures.* Many concurrent design patterns have been proposed in the past, such as hazard pointers [31] and the repeat offender problem [20]. Epoch-based designs [16, 23] have been used by many systems [25, 29, 41] to alleviate specific bottlenecks. However, we augment epoch protection with trigger actions, and design it as a generic framework to enable lazy synchronization. FASTER uses it as a building block at several instances in the paper (see Sec. 2.4). In-place updates are prevalent in pure in-memory data structures. Read-copy-update was proposed to allow readers to be unaffected by concurrent writers. Fast sequential writes in modern storage resulted in the popularity of log structuring , which applies read-copy-update to data on an append-only log. Log structuring was first applied to file systems [38], but was later used in key-value stores [24, 34, 35, 40]. The FASTER hybrid log combines these techniques to achieve the best of both worlds: high performance for hot data and fast sequential logging to storage.

*Hash Key-Value Stores.* In-memory caches and stores such as Redis [36], Memcached [30], and improvements such as MemC3 [10] and MICA [27] are used to speed-up web deployments and alleviate database load, but do not themselves handle data larger than memory (Redis supports a write-ahead log for recovery). Distributed systems such as RAMCloud [35] and FaRM [7] focus on scaling out the key-value store, e.g., by using partitioning, remote memory, or RDMA, rather than leveraging storage for cold data. Reported single-node performance of all these systems is lower than our target. In contrast, FASTER is a single-node concurrent high-level language component that exploits storage using a hybrid log-based data organization. Streaming state stores, such as the Spark State Store [14] and the Storm Trident State Store [15], use a simple partitioned in-memory hash table, synchronously checkpointed periodically. They do not support concurrent access and have low reported throughputs. Google Cloud Dataflow [17] stores recent state in memory, and offloads data periodically to BigTable [3], resulting in potentially high overhead due to the decoupling.

*Range Key-Value Stores.* Systems such as Masstree [29, 41] are pure in-memory range indices, and as such target different applications. Systems such as Cassandra [11], RocksDB [34, 40], and Bw-Tree [25] are key-value stores that can handle state larger than main memory. However, they are not a good fit for our target applications due to their key-ordered page format that is optimized for reads and range queries, at the cost of greater complexity. Further, their use of read-copy-update can be expensive for update-intensive workloads. RocksDB supports in-place updates in its in-memory component (level 0), but is unable to exploit it to get acceptable performance for in-memory workloads. RocksDB generally achieves throughput less than 1M ops/sec, and its "merge" operation is expensive for RMW workloads. In contrast, FASTER targets point operations and update-intensive workloads at very high performance.

*Databases.* In-memory databases are optimized for more general data processing needs than what FASTER is optimized for. They cannot handle data that spills to secondary storage. ERMIA [22] is a fast memory-optimized database that uses techniques such as latch-free structures, epoch protection, and log-structuring in its design. However, it is append-only, and targets fully serializable transactions instead of atomic point operations, leading to lower expected performance for our target applications. Traditional databases handle large data using a buffer pool. FASTER avoids a buffer-pool and page latching, but uses coarse-grained regions in the log to achieve high performance while adapting to a changing working set. H-Store [21] partitions the workload and avoids concurrency, but this strategy can create shuffle overheads, load imbalance, and skew issues. Deuteronomy [26] and SQL Server Hekaton [6] use hashing to index the recovery log and in-memory database respectively, but are based on read-copy-update.

## 9 CONCLUSIONS

We present FASTER, a new concurrent key-value store optimized for update-intensive applications. FASTER combines concurrent latch-free execution, seamless integration of secondary storage, and in-place update capabilities, to achieve very high throughput in a multi-threaded setting. FASTER is based on a latch-free index that works with `HybridLog`, a novel concurrent log that combines an in-place updatable region with a log-structured organization, to optimize for the hot set without any fine-grained caching statistics. Experiments show that FASTER achieves orders-of-magnitude better throughput – up to 160M operations per second on a single machine – than alternative systems deployed widely today, outperforms pure in-memory data structures when the workload fits in memory, and degrades gracefully as memory becomes limited.

*Acknowledgments.* We would like to thank David Lomet, Umar Minhas, Rahul Potharaju, Ryan Stutsman, Kai Zeng, and the anonymous reviewers for their detailed comments, help, and support.

## REFERENCES

[1] 2017. jemalloc. http://jemalloc.net/. (2017). [Online; accessed 30-Oct-2017].
[2] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/
[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. https://doi.org/10.1145/1365815.1365816
[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10).* ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[5] Intel Corporation. 2017. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/. (2017). [Online; accessed 30-Oct-2017].
[6] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710
[7] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory, In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014). https://www.microsoft.com/en-us/research/publication/farm-fast-remote-memory/
[8] Facebook. 2017. RocksDB Performance Evaluation. https://github.com/facebook/rocksdb/wiki/performance-benchmarks. (2017). [Online; accessed 30-Oct-2017].
[9] Facebook. 2017. RocksDB Wiki. https://github.com/facebook/rocksdb/wiki. (2017). [Online; accessed 30-Oct-2017].
[10] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan
[11] Apache Software Foundation. 2017. Apache Cassandra. http://cassandra.apache.org/. (2017). [Online; accessed 30-Oct-2017].
[12] Apache Software Foundation. 2017. Apache Flink. https://flink.apache.org/. (2017). [Online; accessed 30-Oct-2017].
[13] Apache Software Foundation. 2017. Apache Hadoop. http://hadoop.apache.org/. (2017). [Online; accessed 30-Oct-2017].
[14] Apache Software Foundation. 2017. Spark State Store: A new framework for state management for computing Streaming Aggregates. https://issues.apache.org/jira/browse/SPARK-13809. (2017). [Online; accessed 30-Oct-2017].
[15] Apache Software Foundation. 2017. Trident State Store. http://storm.apache.org/releases/current/Trident-state.html. (2017). [Online; accessed 30-Oct-2017].
[16] Keir Fraser. 2004. *Practical Lock-Freedom*. Technical Report.
[17] Google. 2017. Google Cloud Dataflow. https://cloud.google.com/dataflow/. (2017). [Online; accessed 30-Oct-2017].
[18] Jim Gray and Goetz Graefe. 1997. The Five-minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. *SIGMOD Rec.* 26, 4 (Dec. 1997), 63–68. https://doi.org/10.1145/271074.271094
[19] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-Word Compare-and-Swap Operation. In *In Proceedings of the 16th International Symposium on Distributed Computing*. Springer-Verlag, 265–279.
[20] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, London, UK, UK, 339–353. http://dl.acm.org/citation.cfm?id=645959.676129
[21] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211
[22] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1675–1687. https://doi.org/10.1145/2882903.2882905
[23] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (Sept. 1980), 354–382. https://doi.org/10.1145/320613.320619
[24] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 877–888. https://doi.org/10.14778/2536206.2536215
[25] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 302–313. https://doi.org/10.1109/ICDE.2013.6544834
[26] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper15.pdf
[27] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim
[28] Lin Ma, Joy Arulraj, Sam Zhao, Andrew Pavlo, Subramanya R. Dulloor, Michael J. Giardino, Jeff Parkhurst, Jason L. Gardner, Kshitij Doshi, and Stanley Zdonik. 2016. Larger-than-memory Data Management on Modern Storage Hardware for In-memory OLTP Database Systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. ACM, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2933349.2933358
[29] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855
[30] Memcached. 2017. https://memcached.org/. (2017). [Online; accessed 30-Oct-2017].
[31] Maged M. Michael. 2002. Safe Memory Reclamation for Dynamic Lock-free Objects Using Atomic Reads and Writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing (PODC '02)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/571825.571829
[32] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162. https://doi.org/10.1145/128765.128770
[33] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*. ACM, New York, NY, USA, 297–306. https://doi.org/10.1145/170035.170081
[34] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048
[35] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 29–41. https://doi.org/10.1145/2043556.2043560
[36] Redis. 2017. https://redis.io/. (2017). [Online; accessed 30-Oct-2017].
[37] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. 2016. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1539–1551. https://doi.org/10.1145/2882903.2915966
[38] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. https://doi.org/10.1145/146941.146943
[39] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642
[40] Facebook Open Source. 2017. RocksDB. http://rocksdb.org/. (2017). [Online; accessed 30-Oct-2017].
[41] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713
[42] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113

## A ATOMIC OPERATIONS

FASTER heavily leverages native latch-free atomic 64-bit operations such as compare-and-swap (CAS), fetch-and-add, and fetch-and-increment. CAS compares a given value to that at the location and swaps to a desired value atomically (all or nothing). Fetch-and-add adds a given value to the value at the location and returns the original value. Similarly, fetch-and-increment atomically increments the value at the given location.

## B RESIZING THE FASTER INDEX

The hash index may need to be resized over time as keys are inserted and removed from the store. Without resizing, bucket linked-lists could grow large and result in reduced performance, or there could be many wasted buckets, resulting in memory waste[1].

[1]Index resizing does not address the problem of many keys mapping to the same hash value, which is problematic for all hashing schemes. Standard techniques such

Recall that the index is sized in powers of 2. Logically, we have two versions of the hash index during resizing: one of the current size (*old*) and another (*new*) of double the size when growing, or half the size when shrinking. Further, resizing occurs in three phases: *prepare-to-resize*, *resizing*, and *stable*. We maintain both these values (version and phase) in a single byte called `ResizeStatus`. A thread reads `ResizeStatus` to determine what phase it is in. In the common *stable* phase, threads proceed directly with their operation on the active version of the table.

A hash index is logically divided into *n* contiguous chunks, where *n* is set to the smaller of the maximum concurrency and the number of hash buckets in the active version. Chunks serve as the granularity at which threads can independently perform resizing. There is a shared *pin array* of *n* counters, which are used only during resizing to indicate the number of threads updating buckets in a given chunk. When a thread wishes to resize, it allocates an index of double (or half) the size, and sets the phase to *prepare-to-resize*. It then bumps the current epoch with a future trigger action to atomically set the phase to *resizing* and version to *new*. Threads that are in the *prepare-to-resize* phase are aware that resizing is going to occur, but cannot start because other threads may not be aware of resizing yet. Therefore, they use *fetch-and-increment* to increment the pin count (if it is non-negative) in the pin array entry corresponding to the chunk (in the old version) that they are operating over. Similarly, they decrement the pin count after their operation.

Threads that are in the *resizing* phase know that all threads are using the pin array. Therefore, they compare-and-swap the pin count of the chunk from 0 to $-\infty$ (until successful) to indicate that they are starting to resize that chunk. Threads in the *prepare-to-resize* phase that see a negative pin count refresh their value of `ResizeStatus` to enter the *resizing* state immediately.

When splitting a chunk, a thread iterates over the records in each hash bucket and copies over entries to one of two destination hash buckets in the new index (merging works similarly). Finally, it increments a counter (*numChunks*) to indicate that the chunk is done. Threads co-operatively grab other chunks to resize if the chunk they are accessing is being resized by another thread (indicated by a pin count of $-\infty$). Finally, when *numChunks* reaches *n*, we are done with resizing, and can set `ResizeStatus` to *stable* in order to resume high-performance normal operation.

When using Faster with `HybridLog`, resizing leaves records on disk untouched. A split causes both new hash entries to point to the same disk record, whereas a merge creates a meta-record pointing to two disk records, in the two prior linked-lists, and adds this meta-record to the linked-list for the merged hash entry.

## C GARBAGE COLLECTION FOR `HYBRIDLOG`

`HybridLog` is a log-structured record store, and as such requires to be trimmed from the head of the log in order not to grow indefinitely on storage. Interestingly, `HybridLog` by its nature has lower garbage collection overhead than traditional logs because in-place updates significantly reduce the rate at which the tail of the log grows. We can garbage collect `HybridLog` in two ways:

- *Expiration*: Data stored in cloud providers often has a maximum time to live, after which it is deleted. We can use this property to periodically delete chunks of log prefixes from storage.
- *Roll To Tail*: We can roll forward a chunk of the log by scanning from the head and copying over live key-values to the tail.

We prefer to use the expiration-based garbage collection mechanism, as it reflects our use cases where the log is used for analytics, and expires based on data collection guidelines. The Faster index keeps track of the earliest valid logical address, and when a thread encounters an invalid address in a hash bucket, it simply deletes it. Further, any linked list traversal of log records is stopped when it encounters an invalid previous logical address.

*Identifying Live Values.* In the roll-to-tail approach, we need to identify whether a given key is live or not, in order to determine if we should copy it to the tail. We could traverse the linked-list for the corresponding hash entry, but this may be expensive. Instead, we can reserve an *overwrite* bit in the record header to indicate that the record has been overwritten by a subsequent operation. We can set this bit even if the record is in the read-only region (until it gets flushed to disk). On garbage collection, we perform the linked-list scan only for records that do not have this bit set. This captures the common case of a data item being hot and frequently updated, and then suddenly becoming cold — all earlier versions of the record would have the overwrite bit set, assuming that the record was hot enough to get copied over to the tail before being flushed. The final version of the record (now cold) likely has an entry in the in-memory index, allowing us to avoid a random seek into the log.

## D HANDLING READ-HOT RECORDS

The single `HybridLog` design we present in the paper works well for update-mostly workloads. Reads are simply treated as updates and copied over to the tail of `HybridLog`. Interestingly, this is a good solution for read-mostly workloads where the working set fits in memory as well, because the read-hot records get clustered into the tail of `HybridLog` in memory, and provide good in-memory performance without significant log growth.

For a mixed workload with a non-trivial number of read-hot records, our design can accommodate a separate read cache. In fact, we can simply create a new instance of `HybridLog` for this purpose. The only difference between this log and the primary `HybridLog` is that there is no flush to disk on page eviction. Record headers in these read-only records point to the corresponding records in the primary log. As in normal `HybridLog`, the size of the "read-only" region controls the degree of "second chance" that records get (to move back to the tail) before being evicted from the read-only cache.

For the hash index, we have two options: (1) The hash index can use an additional bit to identify which log the index address points to. When a read-only record is evicted, the index entry needs to be updated with the original pointer to the record on the primary log. Index checkpoints need to overwrite these addresses with addresses on the primary log. (2) We can keep a separate read-only hash index to lookup the read-only `HybridLog`. Read or update operations on the main index that point to addresses on disk first check this index before issuing an I/O operation. This approach provides clean separation, at the cost of an additional cache miss for read-hot

as adding key suffixes to spread the load of colliding keys are applicable, but are orthogonal to our proposal.

objects. A detailed evaluation of these techniques is outside the scope of this paper.

## E INTERFACE AND CODE GENERATION

Faster separates a *compile-time interface*, which accepts user-defined read and update logic in the form of functions; and a customized *runtime interface*, whose code is generated for an application for the required read, upsert, and RMW operations.

The user-defined functions are defined over five types: Key, Value, Input, Output, and Context. The first two types represent the data stored in Faster. The *Input* type is used to update or read a value in the store. For instance, we may have a sequence of CPU readings used to update a per-device average: here, the key is a device-id (`long`), input is the reading (`int`) and the value is the average CPU utilization (`float`). The *Output* type is for the output read (or computed) from the value and an (optional) input. For example, input could be a field id to select a field to be copied from the value on a read. Finally, the Context type represents user state that is used to relate asynchronous callbacks with their corresponding original user operation.

```
void CompletionCallback(Context*);

// Read functions
void SingleReader(Key*, Input*, Value*, Output*);
void ConcurrentReader(Key*, Input*, Value*, Output*);

// Upsert functions
void SingleWriter(Key*, Value*, Value*);
void ConcurrentWriter(Key*, Value*, Value*);

// RMW functions
void InitialUpdater(Key*, Input*, Value*);
void InPlaceUpdater(Key, Input*, Value*);
void CopyUpdater(Key*, Input*, Value*, Value*);
```

For functions that have two parameters of type `Value`, the first represents the old value and the second represents the new, updated, value. Faster invokes the `CompletionCallback` with a user-provided context associated with a pending operation, when completed. To support reads, the user defines two functions. The first, `SingleReader`, takes a key, an input, and the current value and allows the user to populate a pre-allocated output buffer. The system guarantees read-only access to the value during the operation. The second, `ConcurrentReader`, is similar, but may be invoked concurrently with updates or writes; the user is expected to handle concurrency (e.g., using an S-X lock).

Faster supports two kinds of updates: Upserts and RMWs. An upsert require two functions: `SingleWriter` overwrites the value with a new value, where Faster guarantees exclusive write access. `ConcurrentWriter` may be called (as its name implies) concurrently with other reads and writes. An RMW requires three update functions: an `InitialUpdater` to populate the initial value, an `InPlaceUpdater` to update an existing value in-place, and a `CopyUpdater` to write the updated value into a new location, based on existing value and the input. Initial and copy updaters are guaranteed exclusive access to the value, whereas in-place updaters may be invoked concurrently. Users can optionally indicate that an RMW is *mergeable*, which allows Faster to apply CRDT optimizations.

We use these functions to generate the Faster runtime interface:

```
Status Read(Key*, Input*, Output*, Context*);
Status Upsert(Key*, Value*, Context*);
Status RMW(Key*, Input*, Context*);
Status Delete(Key*, Context*);
void Acquire();  void Release();
void CompletePending(bool wait);
```

`Read` takes a key, an input, and a pre-allocated buffer for storing the output. `Upsert` and `RMW` take a key and value as parameters. Threads call `Acquire` and `Release` to register and deregister with Faster. They call `CompletePending` regularly to continue pending operations. A thread may optionally block (when `wait = true`), until all outstanding operations issued by the thread are completed.

While it is possible to implement these advanced operations on top of a simple key-value interface, such layering adds significant overheads to the end-to-end application performance. For example, one might choose to use an atomic fetch-and-add instead of latches to build a sum-based update store, use non-latched operations in `SingleReader` and `SingleWriter`, or even use non-latched operations everywhere if they know that their input arrives partitioned.

## F LOG ANALYTICS

The Faster record log is a sequence of updates to the state of the application. Such a log can be directly fed into a stream processing engine to analyze the application state across time. For example, one may measure the rate at which values grow over time, or produce hourly dashboards of the hottest keys in the application. The size of read-only and in-place updatable regions in `HybridLog` controls the frequency of updates to values present in the log. Further, one may handle point-in-time queries by scanning the log, or query historical values of a given key (since our record versions are linked in the log). Investigating these possibilities in a *hybrid updates and analytics* system is an interesting direction for future work.

## G ALGORITHMS FOR HYBRIDLOG AND FASTER

We present algorithms for `HybridLog` in Alg. 1. `Allocate` is invoked by a thread when it wishes to allocate a new record. New records are allocated at the tail using fetch-and-add. If the address is within a logical page (common case), we simply return the logical address. The first thread whose `Allocate` overflows the page handles buffer maintenance and resets the offset for the new page. Other threads spin-wait for this thread to reset the offset.

Next, we present the *Read*, *Upsert*, and *RMW* algorithms for Faster using `HybridLog`, in Algs. 2, 3, and 4 respectively. The `find_tag` procedure finds an existing (non-tentative) entry in the index, while `find_or_create_tag` returns an existing (non-tentative) entry, or creates a new one, using the two-phase insert algorithm described in Sec. 3. The `trace_back_until` procedure traverses the record linked-list that is present in memory to obtain the logical address of the record that corresponds to the key or first on-disk record (obtained from the last record in memory).

A read operation issues a read request to disk if logical address is less than head offset, reads using the single reader if record is in the safe-read-only region, or the concurrent reader if it is in the fuzzy or mutable region. Upsert updates in-place if the record is in the mutable region, and creates a new copy at the tail otherwise. RMW issues a read request if logical address is less than head offset; creates a new record at the tail if it is in the safe-read-only region;

puts an operation into a pending list for processing later, if it is in the fuzzy region; and updates it in-place if it is in the mutable region. For Reads and RMW, the operation context is enqueued into a pending queue when the asynchronous operation is complete. These operations continue processing (using their saved contexts) when the user invokes `CompletePending`.

```
function Allocate(int size)
    offset = toffset.fetch_and_add(size);
    if offset + size < page_size then
        return (tpage ≪ page_size_bits) | offset;
    else if offset < page_size then
        buffer_maintenance (tpage++);
        index = tpage % buffer_size;
        spin-wait until closed-status[index] == 'C';
        flush-status[index] = 'D'; closed-status[index] = 'O';
        allocate (or clean) frame[index] for tpage;
        if (offset + size) == page_size then
            toffset.store(0);
            return ((tpage - 1) ≪ page_size_bits) | offset;
        else
            toffset.store(size);
            return (tpage ≪ page_size_bits) | 0;
    else
        spin-wait until toffset < page_size;
        return Allocate(size);
procedure buffer_maintenance(long tpage)
    index = tpage % buffer_size;
    if head_offset is lagging then
        new_head = (tpage - buffer_size) ≪ page_size_bits;
        adjust new_head based on flush-status;
        old_head = head_offset;
        head_offset.store(new_head);
        bump_epoch(( ) ⇒
            { close pages in range [old_head, new_head); });
    if ro_offset is lagging then
        ro_offset.store((tpage - ro_lag) ≪ page_size_bits);
        bump_epoch(( ) ⇒ { update_safe_ro(ro_offset)});
procedure update_safe_ro(long new)
    old = safe_ro_offset;
    safe_ro_offset = new;
    foreach page ∈ [old, new) do
        async_flush_page(page, callback : ( ) ⇒ {flush-status[index] = 'F';});
```

**Algorithm 1:** Algorithms for `HybridLog`

```
function Status Read(key, input, output, ctx)
    entry = find_tag(key);
    laddr = trace_back_until(key, entry.address, head_offset);
    if laddr is invalid then
        return Status.Error;
    else if laddr < head_offset then
        Create context and issue async IO request to disk;
        return Status.Pending;
    else
        paddr = get_physical_address(laddr);
        record = obtain record at paddr;
        if laddr < safe_ro_offset then
            SingleReader(key, input, record.value, output);
        else
            ConcurrentReader(key, input, record.value, output);
        return Status.OK;
```

**Algorithm 2:** Read algorithm in FASTER

```
function Status Upsert(key, value, ctx)
    entry = find_or_create_tag(key);
    laddr = trace_back_until(key, entry.address, head_offset);
    if laddr is valid and laddr > read_only_offset then
        paddr = get_physical_address(laddr);
        record = Obtain record at paddr;
        ConcurrentWriter(key, record.value, value);
        return Status.OK;
    else
        new_laddr = Allocate (size);
        new_record = record at new_paddr;
        write key, entry.address at new_record;
        SingleWriter(key, new_record.value, value);
        updated_entry = unset tentative bit, tag, new_laddr;
        if entry.CAS(entry, updated_entry) then
            return Status.OK;
        else
            new_record.invalid = true;
            return Upsert(key, value, ctx);
```

**Algorithm 3:** Upsert Algorithm in FASTER

```
function Status RMW(key, input, ctx)
    entry = find_or_create_tag(key);
    laddr = trace_back_until(key, entry.address, head_offset);
    if laddr is invalid then
        goto CREATE_RECORD;
    else if laddr < head_offset then
        create context and issue async IO request to disk;
        return Status.Pending;
    else
        paddr = get_physical_address(laddr);
        record = Obtain record at paddr;
        if laddr < safe_ro_offset then
            goto CREATE_RECORD;
        else if laddr < ro_offset then
            add context to pending list;
            return Status.Pending;
        else
            InplaceUpdater(key, input, record.value);
            return Status.OK;
    CREATE_RECORD:
    new_laddr = Allocate (size);
    new_paddr = get_physical_address(new_laddr);
    new_record = record at new_paddr;
    write key, entry.address at new_record;
    if laddr is invalid then
        InitialUpdater(key, input, new_record.value);
    else
        CopyUpdater(key, input, record.value, new_record.value);
    updated_entry = unset tentative bit, tag, new_laddr;
    if not entry.CAS(entry, updated_entry) then
        new_record.invalid = true;
        return RMW(key, input, ctx);
    return Status.OK;
```

**Algorithm 4:** RMW Algorithm in FASTER