Faster: A Concurrent Key Value Store with In-Place Updates

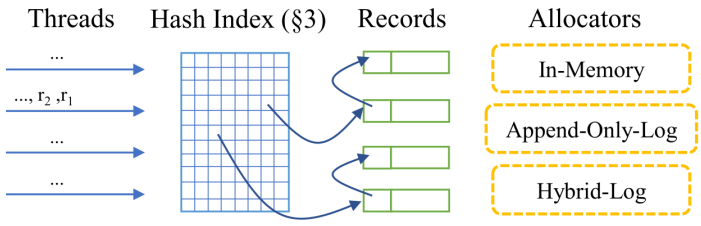
September 12, 2025

1 Introduction

- Augment standard epoch-based synchronization into a generic framework that facilitates lazy propagation of global changes to all threads via trigger actions.
- Concurrent latch-free resizable cache-friendly hash index
- log-structuring

2 System Overview

2.1 Architecture



	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.

The hash bucket is a cache-line sized array of hash bucket entries. Each entry includes some metadata and an address 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.

2.2 User Interface

- Read
- Upsert
- RMW
- Delete

2.3 Epoch Protection Framework

We extend the idea of multi-threaded epoch protection into a framework enabling lazy synchronization over arbitrary global actions. While systems like Silo, Masstree and Bw-Tree 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.

Epoch Basics. The system maintains a shared atomic counter **E**, called the **current epoch**, that can be incremented by any thread. Every thread T has a thread-local version of **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:

$$\forall T : \mathbf{E}_s < E_T < \mathbf{E}$$

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 $\langle epoch, action \rangle$ 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 $\langle c, Action \rangle$ 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, index resizing, circular buffer maintenance and page flushing, shared log page boundary maintenance, and checkpointing.

2.5 Lifecycle of a Faster Thread

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

We assume a 64-bit machine with 64-byte cache lines.

3.1 Index Organization

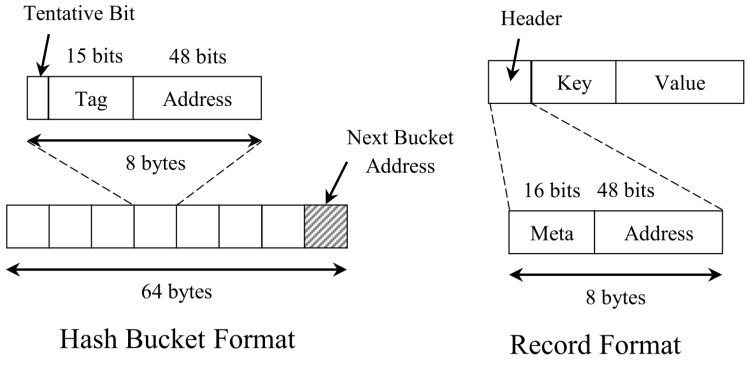


Figure 2: Detailed FASTER index and record format.

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. 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 (3.1) 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 2k 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

Consider the case where a tag does not exist in the bucket, and a new entry has to be inserted. However, two threads could concurrently insert the same tag at two *different* empty slots in the bucket.

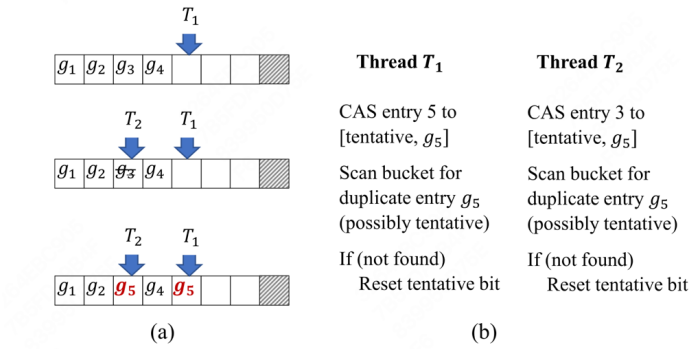


Figure 3: (a) Insert bug; (b) Thread ordering in our solution.

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. 3.2a. 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 T1 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. 3.2b shows the ordering of operations by two threads: there exists no interleaving that could result in duplicate non-tentative tags.

Consider only two slots in a bucket

1. find a new slot
2. (atomic) cas
3. (atomic) get another slot's value
4. check ifs the same key
5. (atomic) cas

Now suppose the key has two slots resulted by two inserts i_1 and i_2 and $i_1.5 \rightarrow i2.5$

Because $i_1.3 \rightarrow i_1.5$, we have $i_1.3 \rightarrow i_2.2 \rightarrow i_2.4$. But now i_2 can get i_1's result and cannot succeed.