Faster: A Concurrent KV Store with In-Place Updates

By: Phillip Tran, Vineet Raju, Arkash Jain

Introduction, Motivation & Solution

Background

Edge devices produce vast amounts of data that are update intensive and require large amounts of state

Large Amounts of Data Produced

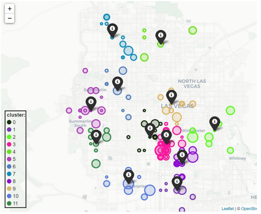
- Data has been produced at a large scale from edge sources like browsers, devices, servers, and processed by cloud applications for analytics.
- Hadoop and Spark need to process data as it arrives.

Update Intensity

- There is significant update traffic.
- Updates of states should be readily available for offline analytics

Locality & Point Operations

- A search engine may have billion users alive in a system but only a million are currently actively surfing in the alive.
- Range queries if infrequent, can be solved



Background Questions?

- Can you think of examples where tons of data is produced which may need to be processed?
- What value lies in optimizing for point queries?

Motivation

State exceeds main memory while constant updates temporal locality of objects are cumbersome to deal with

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

In-Memory Data Stores

Systems partition the state across multiple machines and use pure in-memory data structures

Solutions are expensive and severely under-utilized

Key-Value Stores

KV store is designed to handle larger than memory data and support failure recovery by storing data on a secondary storage.

Do not scale over a million updates per second.

FASTER

- New concurrent key-value store, designed to serve applications that involve update-intensive state management supporting data larger than memory

What are the major technical contributions of the paper?

FASTER

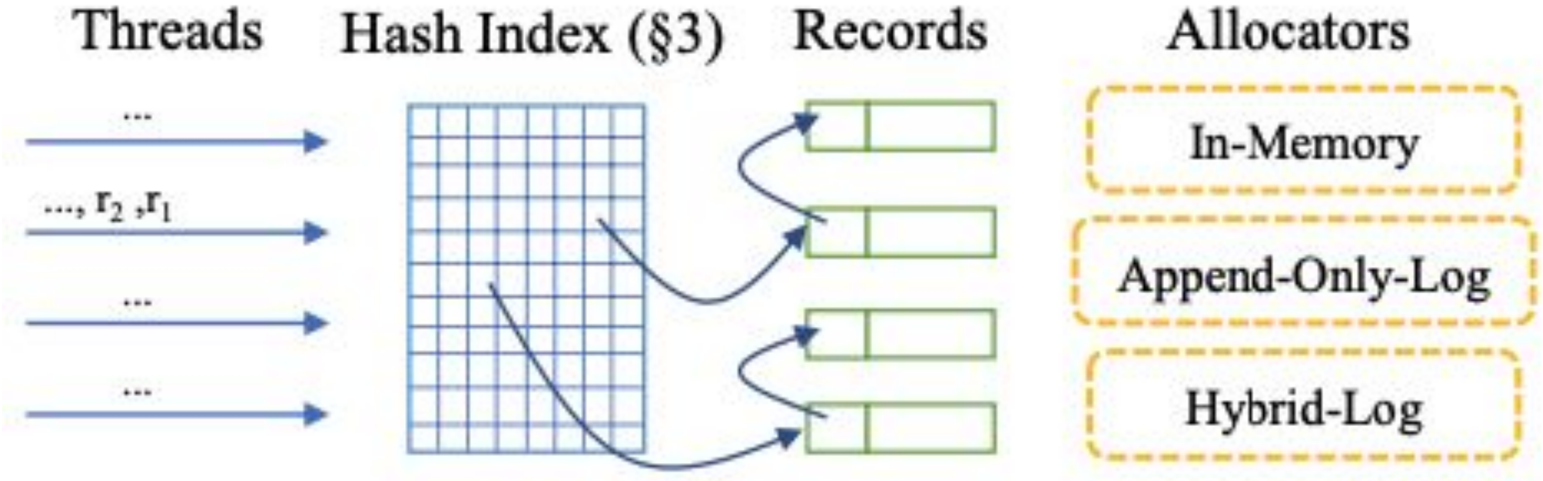
Major Contributions:

(1) Threading: epoch protection with trigger actions

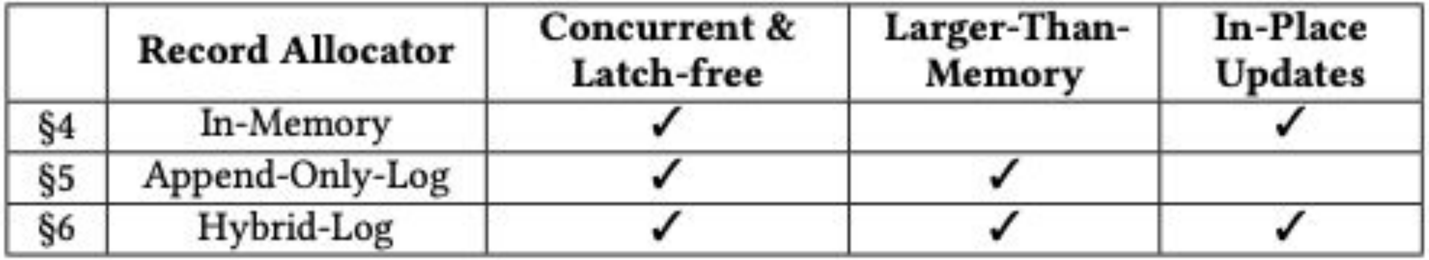
(2) Indexing: concurrent hash index

(3) Record Storage: *Hybrid Log* Record Allocator

System Architecture



System Architecture



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

Background Questions?

- What’s the benefit of in-place updates in a hybrid log?
- Why can you not use in-place updates in append-only logs?

System Architecture: User Interface

- Read, Blind updates
- Atomic read-modify-write (RMW) for running aggregates, partial field updates, …

read()

upsert()

rmw()

delete()

System Architecture: Epoch Protection Framework

- avoiding thread synchronization in the common fast path
- we still need a mechanism to agree on shared system state

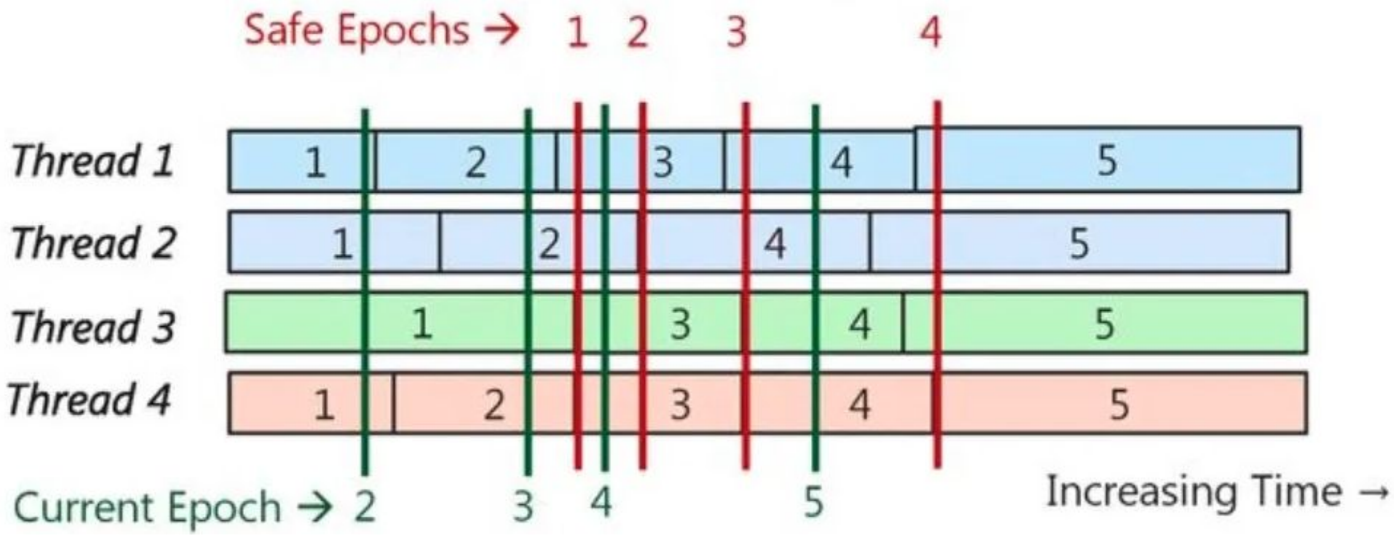
Epoch Protection

- system maintains a shared, atomic counter, **E** (current epoch) - can be incremented by any thread
- each thread keeps a (stale) local epoch counter copied from **E** (refresh periodically)
- epoch, **c** is safe if all threads have a strictly higher local thread-local value than **c**

System Architecture: Epoch Protection Framework

Epoch Protection

- system maintains a shared, atomic counter, **E** (current epoch) - can be incremented by any thread
- each thread keeps a (stale) local epoch counter copied from **E** (refresh periodically)
- epoch, **c** is safe if all threads have a strictly higher local thread-local value than **c**



Extending Epoch Protection: Adding Trigger Actions

How did the authors extend Epoch Protection to make it a more general framework?

Extending Epoch Protection: Adding Trigger Actions

- adds a primitive to epoch protection: function callbacks (trigger) with epoch increment from **c** to **c+1**
- trigger action specified with be executed later when **c** becomes safe
- simplifies lazy synchronization in multi-threaded systems

- function, **active_now()** must be invoked when a shared variable, ***status*** is updated to *active*
- thread updates shared status to active
- increment current epoch with trigger **active_now()**
- now we are guaranteed that all threads have seen the active status before **active_now()** is invoked

- used extensively throughout the system: memory safety, non-blocking index resizing, log buffer maintenance, recovery

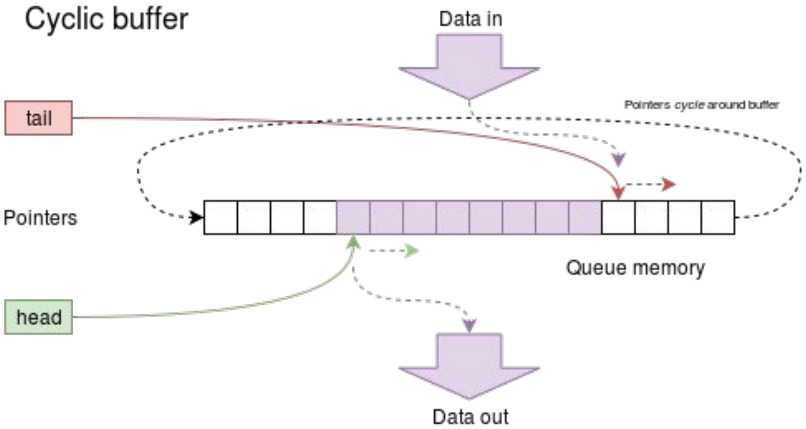
Applications

Handling Large Data: Logical Address Space

Adapting Log Structuring along with an epoch protection framework for lower synchronization overhead.

- Spans primary and secondary storage where record allocator returns 48-bit logical addresses of spaces in memory instead of the physical address.

- Tail offset finds the next free space in the tail of the log and head offset finds lowest logical address available at a lag from the tail present in a contiguous address space using an in-memory circular buffer



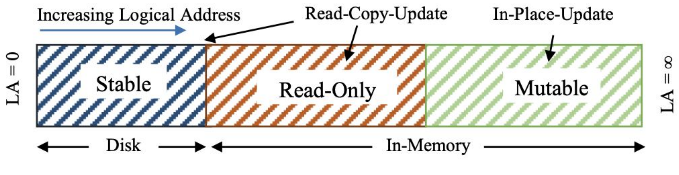
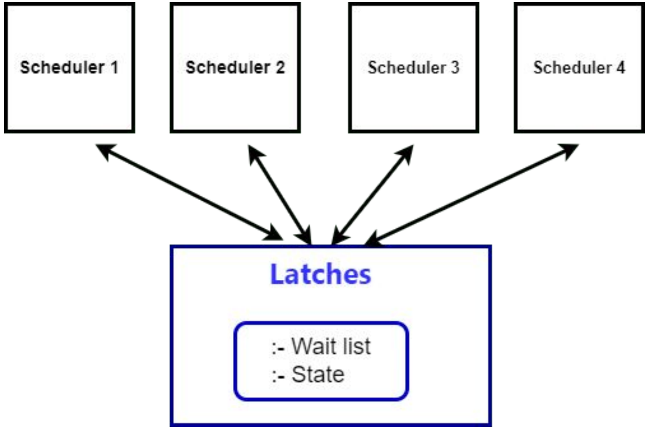


Figure 5: Logical Address Space in HybridLog

Handling Large Data: Circular Buffer Maintenance

Epochs allow latch free eviction of data from storage for efficiency.

- 2 arrays are maintained:
  - Flush array - tracks the status of the page to check if its flushed to secondary storage or not.
  - Closed array - determines whether the page can be evicted for reuse or not.
- The logs are immutable so when the page number is changed from *p* to *p+1* there is a trigger action to flush this page to secondary storage via asynchronous I/O calls, done when the epoch is safe.
- Status of the page is set to flush once done.



Background Questions?

- How do traditional databases flush pages?
- What makes FASTER so different then?

Handling Large Data: Append-Only Allocator

Adapting Log Structuring along with an epoch protection framework for lower synchronization overhead.

- 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 operation fails we mark it as invalid and retry

- Deletes insert a tombstone record (again, using a header bit), and require log garbage collection
- Read and RMW operations are similar to their in-memory counterparts

- Updates are appended to tail

- Retrieve only the record not the entire logical page

Background Questions?

- How do in-memory databases read?
- Is RMW atomic and why are they used?

Features

Background Questions?

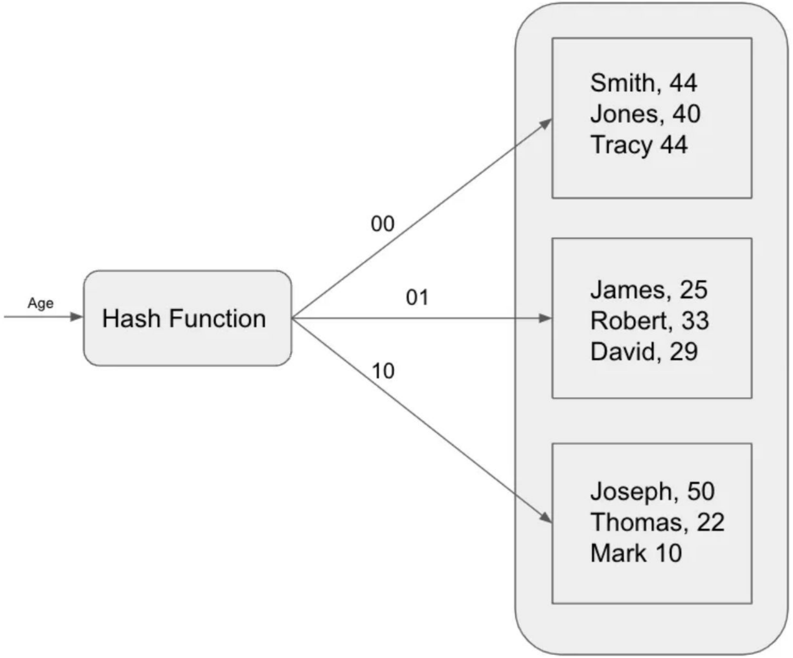
- What is a hash-based index?
- Define all of the bolded terms: “A **concurrent**, **latch-free**, **scalable**, **resizable** hash-based index.”

Features: The Faster Hash Index

“A **concurrent, latch-free, scalable, resizable** hash-based index.”

- **Concurrent:** Supports multiple threads at the same time.
- **Latch-Free:** Doesn’t require locking, which will reduce performance.
- **Scalable:** Able to handle large amounts of data.
- **Resizeable:** The size of the index change be changed after the fact.

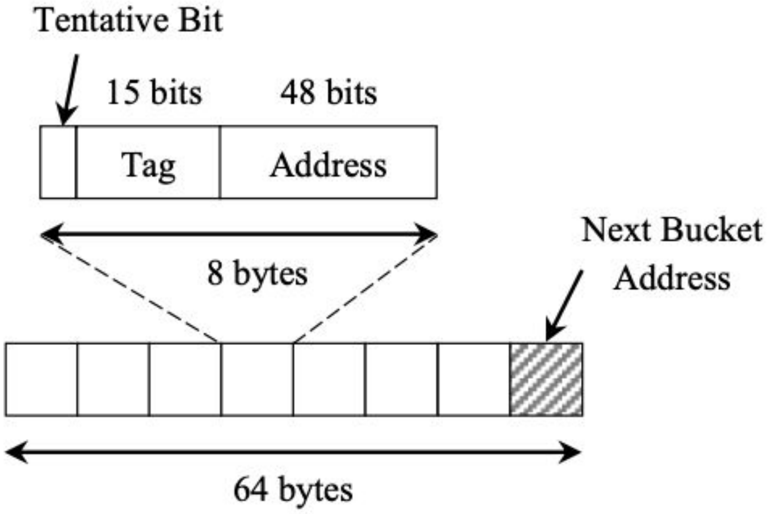
Hash-Based Index: allows us to quickly find data given a a search key value.



Hash indexing structure

Features: The Faster Hash Index

- 2^k hash buckets (k = keys)
- 8-byte entries = 64-bit atomic compare-and-swap operations
- Tag increases hashing resolution to k + 15 bits; reduces collisions
- (offset, tag)
  - Key with hash value h
  - First k bits of h (offset)
  - Next 15 bits of h (tag)
- Invariant
  - Each (offset, tag) has unique index entry



Hash Bucket Format

Features: The Faster Hash Index (cont.)

Find Operation

1. Use k hash bits: find the hash bucket
2. Scan through bucket, find entry matching the tag

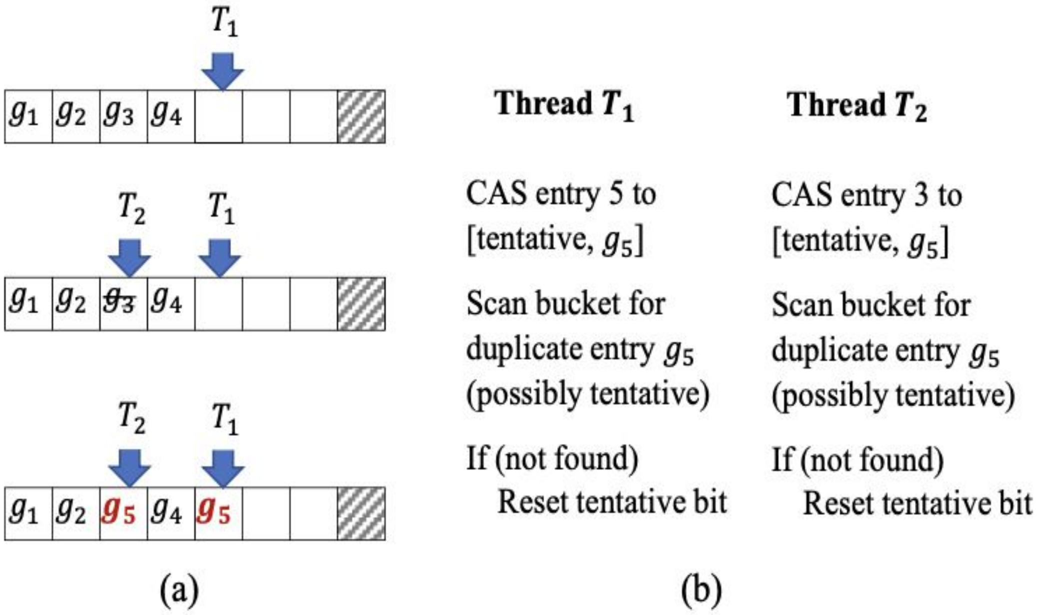
Delete Operation

1. Perform the find operation.
2. Use compare-and-swap, replace the matching entry with 0.

Features: The Faster Hash Index (cont.)

Insert Operation

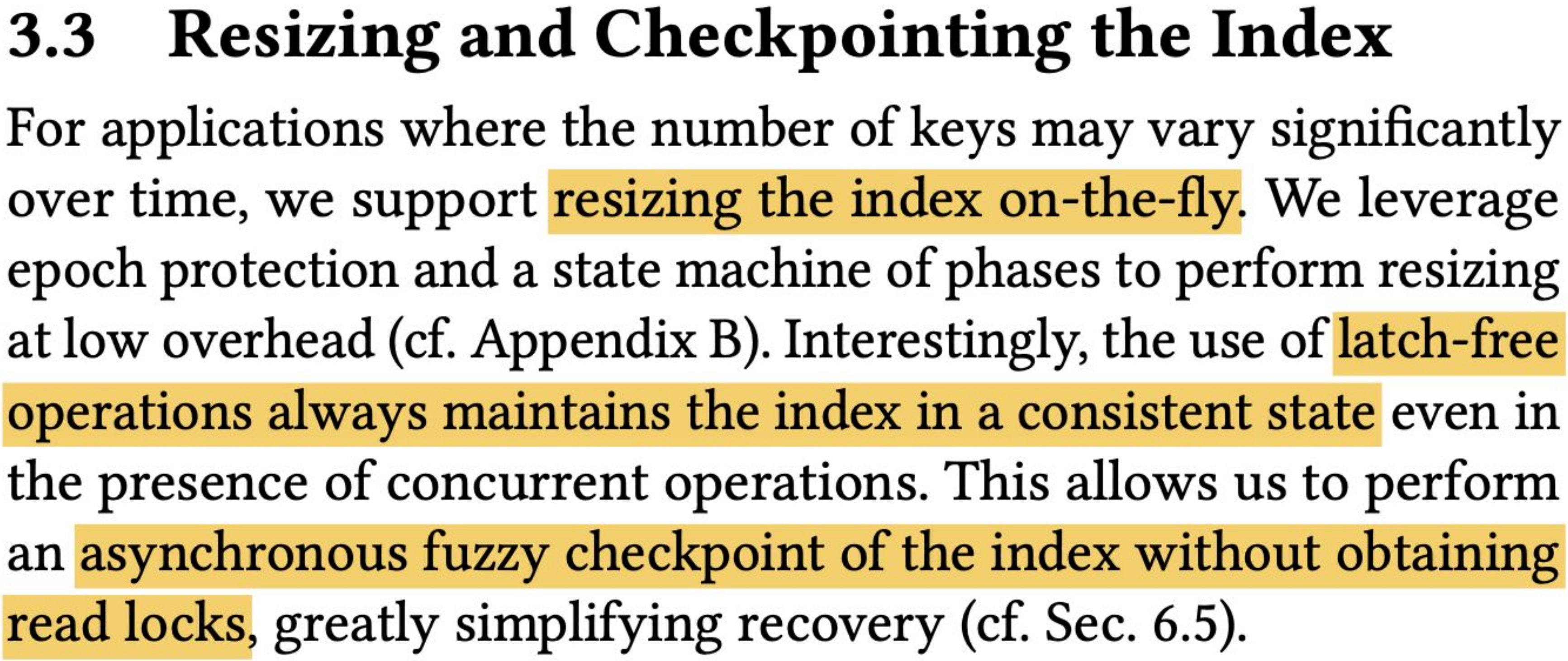
1. Find empty slot; insert record with tentative bit = 1
   a. This slot is now invisible to concurrent read/update operations
2. Scan bucket to check if another tentative entry for the tag exists; if so, try again
3. Else reset bit to 0 to finalize insert operation



Thread T_1	Thread T_2
CAS entry 5 to [tentative, g_5]	CAS entry 3 to [tentative, g_5]
Scan bucket for duplicate entry g_5 (possibly tentative)	Scan bucket for duplicate entry g_5 (possibly tentative)
If (not found) Reset tentative bit	If (not found) Reset tentative bit

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

Features: The Faster Hash Index (cont.)



3.3 Resizing and Checkpointing the Index

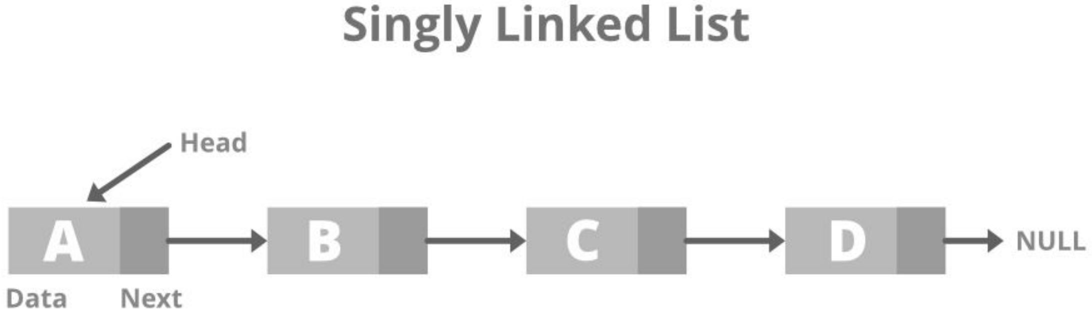
For applications where the number of keys may vary significantly over time, we support resizing the index on-the-fly. We leverage epoch protection and a state machine of phases to perform resizing at low overhead (cf. Appendix B). Interestingly, the use of latch-free operations always maintains the index 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).

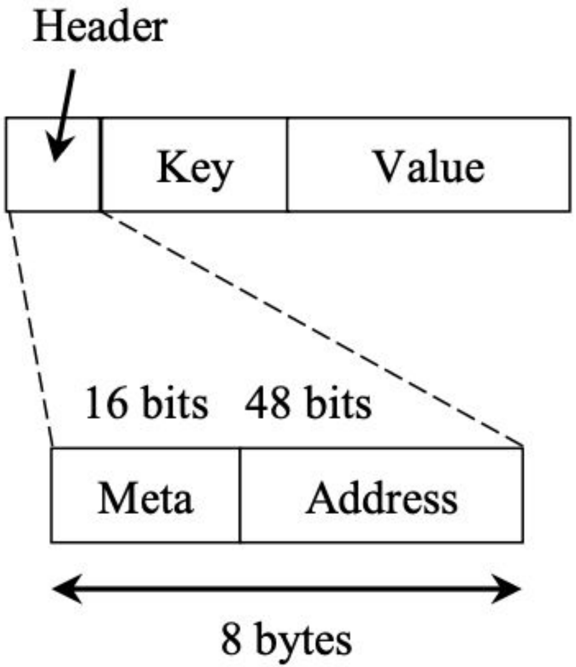
Background Questions?

- What is a linked list?

Features: In Memory KV Store

- Combine hash index with simple in-memory allocator to create complete in-memory KV store.
- Records with same (offset, tag) organized in singly-linked-list.
- Hash bucket entry points to tail (most recent entry), which points to previous record, etc.





Record Format

Features: In Memory KV Store (cont.)

Read Operations:

1. Find tag entry from index.
2. Traverse linked-list for entry to find record with matching key.

Delete Operations:

1. Compare-and-swap on record header or (for the first record) hash bucket entry.
2. Set entry to 0, which makes it available for future inserts.
3. Not immediately returned to memory allocator, only does so when epoch becomes safe.

Update/Insert Operations:

1. Find hash bucket entry for the key…
2. If doesn’t exist, use 2 phase algorithm to insert it as described previously.
3. If exists, scan linked-list to find record with matching key and insert in-place.
4. If match key doesn’t exist, splice new record into tail of list using compare-and-swap.

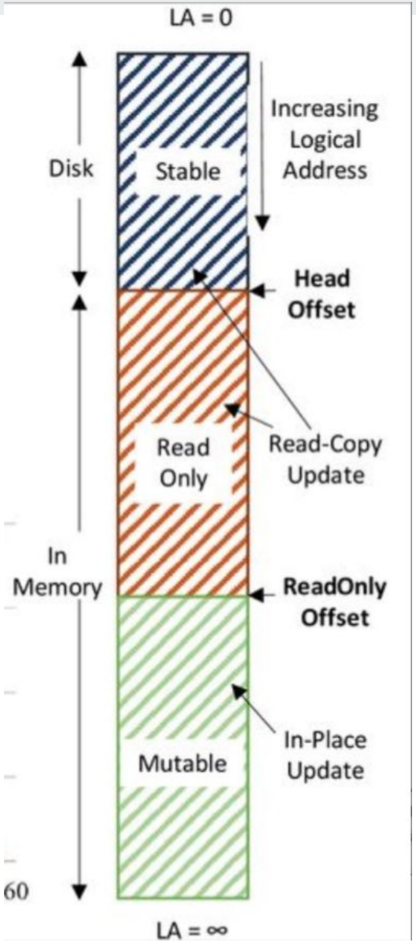
Features: *HybridLog*

- memory is divided into three regions:
    - stable (on disk): read-copy-update
    - mutable (in memory): in-place update
    - read-only (in memory): read-copy-update

- hybrid concurrency model:
    - read-copy-update on index
    - in-place update for record-level concurrency

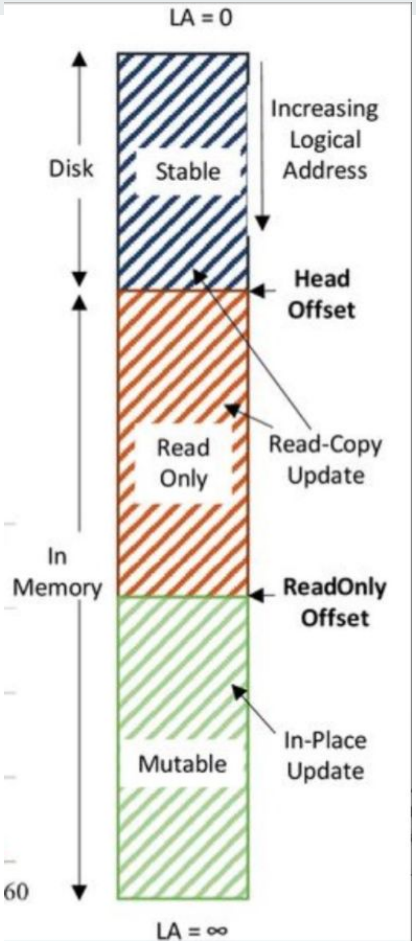
basic read-write-modify algorithm:

```
if < head_offset: issue async io request

if < ReadOnly_offset: copy to tail, CAS update hash
index

if < infinity: update in-place

new_record: add to tail, update hash table
```



Features: *HybridLog*

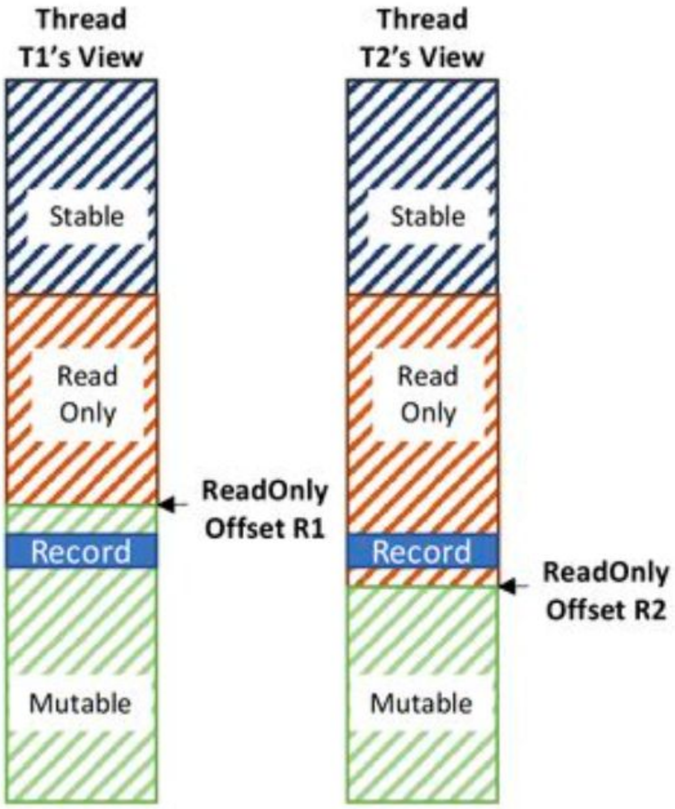
Any issues with this design?



HybridLog- Lost Update Anomaly

- threads guaranteed to read new offsets only at epoch boundaries

- example:
  - thread 1 sees only ReadOnly offset = R1
  - thread 2 sees only ReadOnly offset = R2
  - update by thread 1 is lost => BAD



HybridLog- Fuzzy Region

- fuzzy region: mutability status is not agreed upon by all threads
- `safe readonly offset`

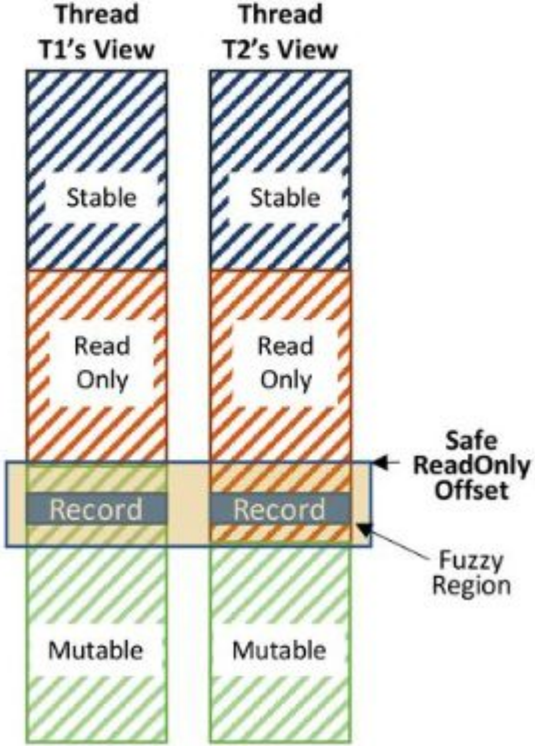
Epoch protection with trigger action:

- `ReadOnlyOffset = k;`

  `BumpEpoch( () => {SafeReadOnlyOffset = K });`

modified read-write-modify algorithm:

```
if < head_offset: issue async io request

if < Safe ReadOnly_offset: copy to tail, update hash table

if < ReadOnly_offset: go pending

if < infinity: update in-place

new_record: add to tail, update hash table
```



HybridLog- Caching, Recovery and Consistency

caching:

- good caching behavior at a per-record granularity without overheads of typical fine-grained caching algorithms
- similar to the second-chance FIFO protocol
- hybrid log *size matters*
  - 90 : 10 division of buffer size for the mutable and read-only regions result in good performance

recovery and consistency:

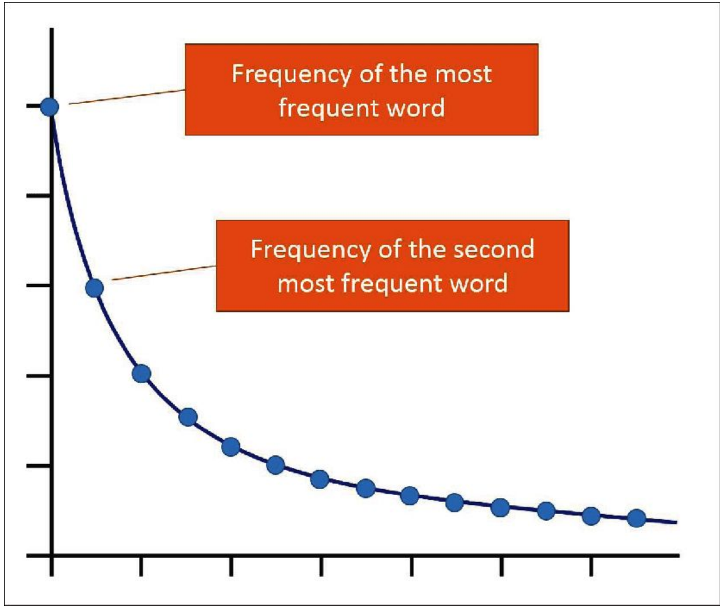
- on failure, unflushed tail is lost
- Consistent with the monotonicity property
  - typically, achieved with **write-ahead-log** (WAL)
- treat hybrid log as a WAL and delay commit to allow in-place updates within a limited time window

Evaluation & Conclusion

Evaluation

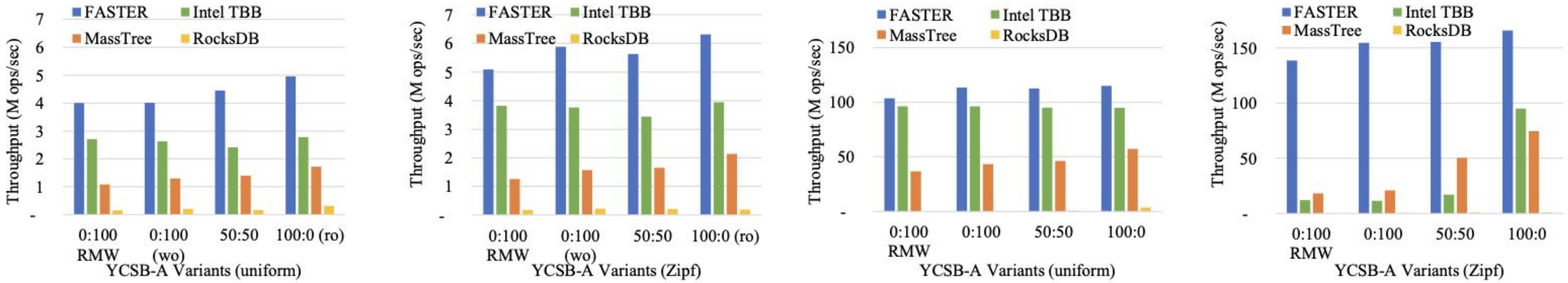
Experimental Setup

1. Implemented in C#
2. When applicable: HybridLog only
3. Threads continuously generate operations for 30 seconds, measure # of operations
4. Two Identical machines (one Windows for Faster, one Ubuntu for others)
   a. 2x Xeon E5-2690 v4, 256GB RAM, 3.2TB NVMe SSD
5. YCSB-A workload
   a. 250M 8-byte keys, values from 8-bytes to 100-bytes
6. Compare against: in-memory, larger-than-memory systems



Zipf Distribution: Does anyone know what this is?

Evaluation (cont.)

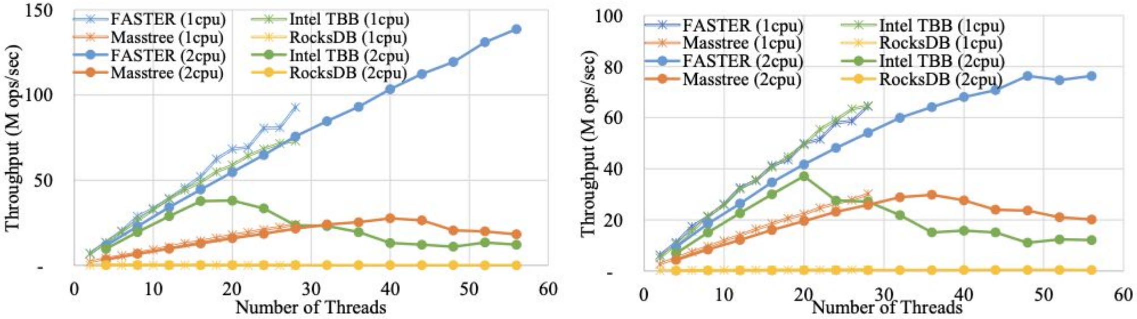


(a) Single thread; uniform distr. **(b) Single thread; Zipf distr.** **(c) All threads; uniform distr.** **(d) All threads; Zipf distr.**

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

Benchmark: threading capabilities

Evaluation (cont.)



(a) RMW updates; 8-byte payloads. **(b) Blind updates; 100-byte payloads.**

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

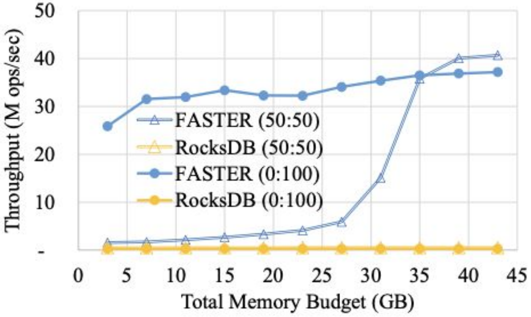


Figure 10: Throughput with increasing memory budget, for 27GB dataset.

Benchmark: Throughput capabilities

Evaluation (cont.)

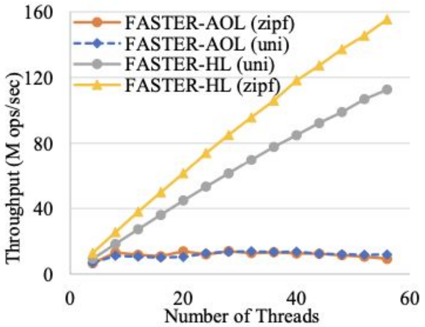
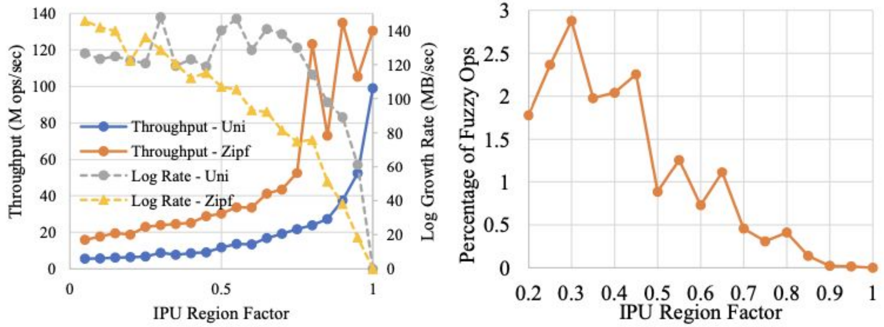


Figure 11: Throughput with append-only vs. hybrid logs.



(a) Throughput & log growth rate.

(b) Percentage of fuzzy ops.

Figure 12: Effect of increasing IPU region.

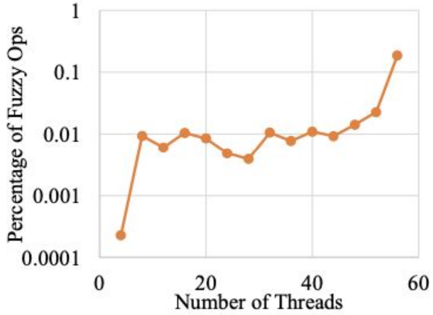


Figure 13: Percentage of fuzzy ops with increasing #threads.

Benchmarks: HybridLog

Evaluation (cont.)

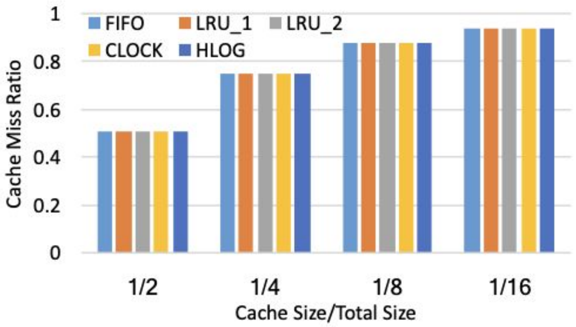


Figure 14: Cache miss ratio (Uniform).

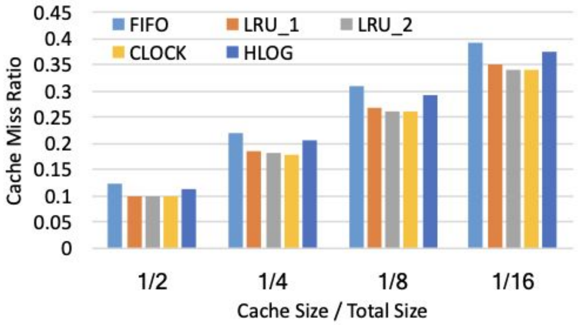


Figure 15: Cache miss ratio (Zipf).

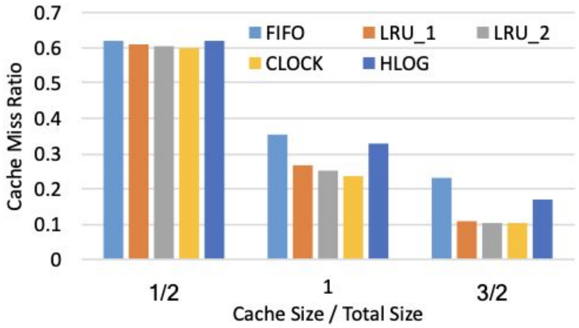


Figure 16: Cache miss ratio (Hot Set).

Benchmarks: Caching Behavior

Conclusions

- FASTER is a high-performance concurrent KV store optimized for update-intensive applications
    - C# implementation provided
- demonstrated that following properties are achievable in the same system:
    - heavy update workload with larger-than-memory data
    - exceed pure in-memory performance when workload fits in memory
    - optimize for moving hot set without any fine-grained caching statistics

Possible further areas of exploration:

- detailed exploration of optimal recovery strategies
- issues with monotonicity in fuzzy checkpointing
- optimizing I/O path to improve performance degradation characteristics in larger-than-memory experiments
- hybrid log analytics
- could this design work on read-only workload variants?