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

Authors: Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, Mike Barnett
University of Washington, Microsoft Research
Houston, TX 2018

Presented By: Alex Stevenson, Abbie Murphy

Outline

01 Introduction

02 Solution

03 Evaluation + Results

04 Conclusion

01

Introduction

- Handling large data in modern applications
- Addressing concurrency
- Current systems and key-value stores
- What is *FASTER* solving?

Data-Intensive Modern Applications

Large amounts of data created on edge devices

- *Large data sets*
- *Processed remotely by cloud applications*

Monitoring and processing data in real time

- Update-intensive data
- Larger-than-memory

Current Challenges

How can we manage large amounts of data at scale?

Advertising platform storing per-user statistics for billions of users

How can we optimize for fast point operations ?

Efficiently retrieve user-specific data without a range scan

How can we handle high update rates efficiently?

Monitoring systems updating per-device CPU metrics

How can we make data updates readily usable for analytics?

Offline analytics: calculating average clickthrough rate

How can we better serve data when queries are highly localized ?

Search engine actively processing data for fraction of billion of users

How can we support concurrency without hurting performance?

Multiple threads sharing data without slowing down performance

Concurrency in Applications

Systems must be capable of managing simultaneous access to states efficiently and reliably.

FOR THE CLASS:

What would you do to facilitate concurrency?

Concurrency in Applications

Systems must be capable of managing simultaneous access to states efficiently and reliably.

Key Concepts

- **Latch systems** use latches for thread-safe access

FOR THE CLASS:
What would you do to facilitate concurrency?
For thread safe access: Latches – exclusive access to protected data structures for threads

Concurrency in Applications

Systems must be capable of managing simultaneous access to states efficiently and reliably.

Key Concepts

- **Latch systems** use latches for thread-safe access

FOR THE CLASS:

What is one issue when trying to achieve synchronization using latch systems?

Concurrency in Applications

Systems must be capable of managing simultaneous access to states efficiently and reliably.

Key Concepts

- **Latch systems** use latches for thread-safe access
    - **Issue**: has delays + contention

FOR THE CLASS:
What is one issue when trying to achieve synchronization using latch systems?
Can increase waiting times!

Concurrency in Applications

Systems must be capable of managing simultaneous access to states efficiently and reliably.

Key Concepts

- **Latch systems** use latches for thread-safe access
    - **Issue**: has delays + contention

FOR THE CLASS:

What would you do if you wanted to ***avoid*** *latches?*

Concurrency in Applications

Systems must be capable of managing simultaneous access to states efficiently and reliably.

Key Concepts

- **Latch systems** use latches for thread-safe access
    - **Issue**: has delays + contention
- **Latch-free systems** avoids latches by using atomic operations or epoch-protection

FOR THE CLASS:

What would you do if you wanted to ***avoid*** *latches?*

Latch-free system examples:
Atomic operations or epoch-protection

Concurrency in Applications

Systems must be capable of managing simultaneous access to states efficiently and reliably.

Key Concepts

- **Latch systems** use latches for thread-safe access
    - **Issue**: has delays + contention
- **Latch-free systems** avoids latches by using atomic operations or epoch-protection

01

Scalability

scaling efficiently across threads and hardware

02

Contention

no more multiple threads competing for same resource

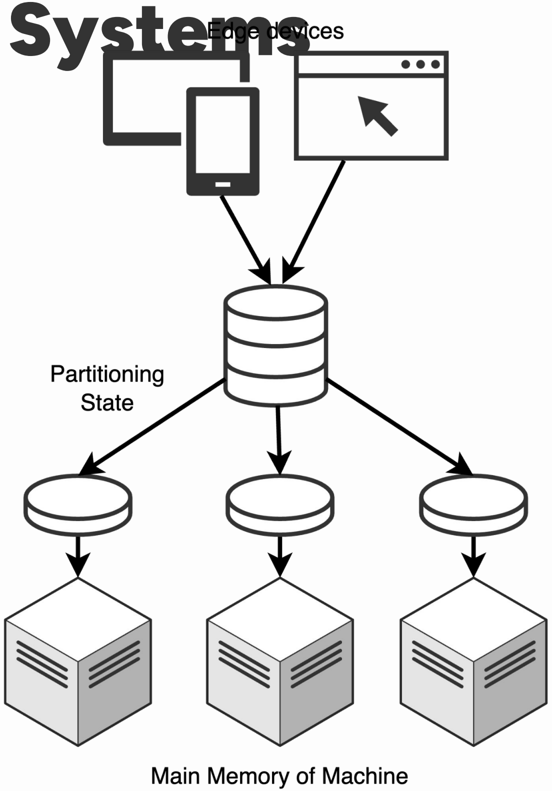
03

Synchronization

reducing waiting times and independent thread operation

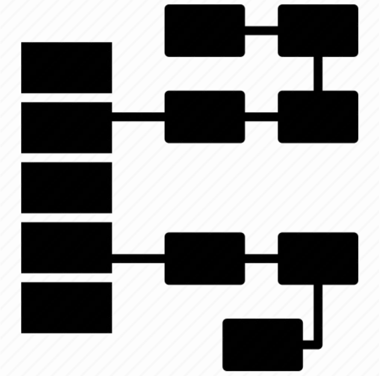
Challenges latch-free systems address

Current Systems



+

Pure In-Memory Data Structure



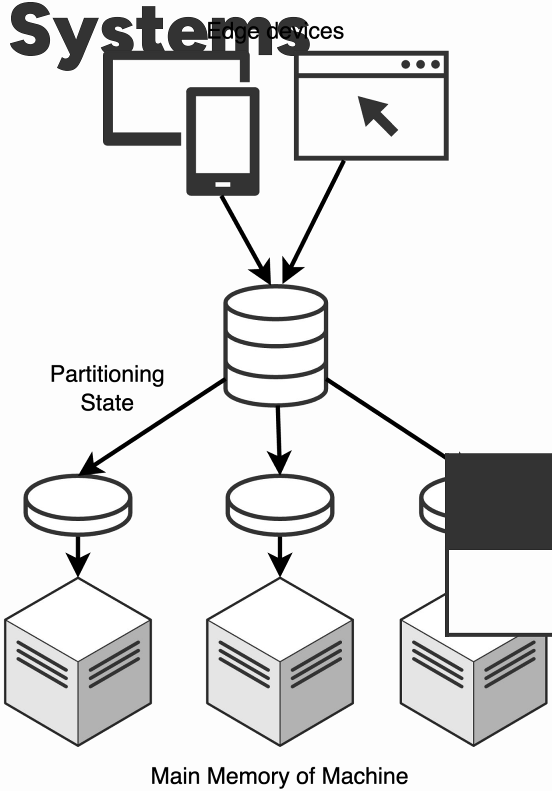
Example: Intel TBB Hash Map

\+ Optimized for concurrency

\+ Supports in-place updates

− Less efficient for managing larger-than-memory data

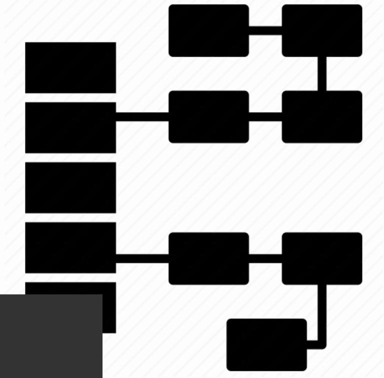
Current Systems



FOR THE CLASS:

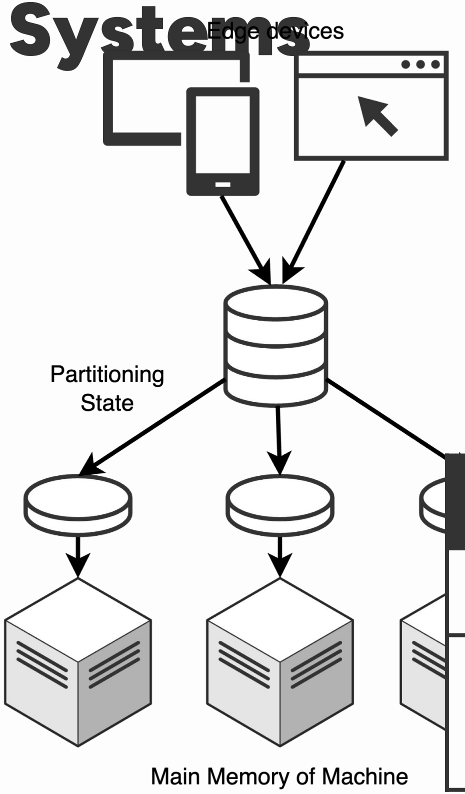
Why is this solution expensive?

Pure In-Memory Data Structure

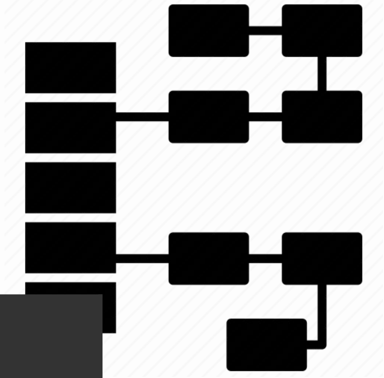


- ntel TBB Hash Map
- \+ Optimized for concurrency
- \+ Supports in-place updates
- − Less efficient for managing larger-than-memory data

Current Systems



Pure In-Memory Data Structure



ntel TBB Hash Map
timized for concurrency
pports in-place updates
s efficient for managing
ger-than-memory data

FOR THE CLASS:

Why is this solution expensive?

Under-utilization of machine resources
Need a structure that efficiently balance memory and storage use + balance larger than memory

What *FASTER* aims to solve

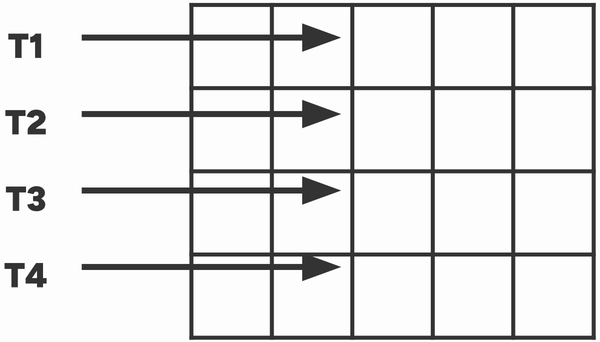
Concurrency

In-place updates

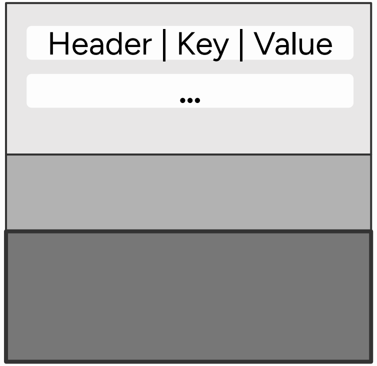
Larger than memory data

What is *FASTER* ?

Concurrent latch-free key-value store with in-place updates



Hash index



HybridLog

02

Solution

- Epoch-Protection Framework
- FASTER Architecture Overview
- FASTER's Hash Index
- In-Memory Key Value Stores
- HybridLog

Epoch-Protection Framework

Ensuring Efficient, Scalable Synchronization Across Threads

What is Epoch Protection?

Epoch Mechanism

Shared atomic counter (E)
Thread has local epoch
Lazy synchronization
(refreshed periodically)

Global Counter

Tracks maximal safe epoch
Updated when threads refresh

Trigger Actions

Drain-list holds <epoch, action> pairs
Actions triggered when epoch is safe
Executed using atomic operation

Why is this framework important?

Lazy Synchronization

Alleviates thread coordination cost

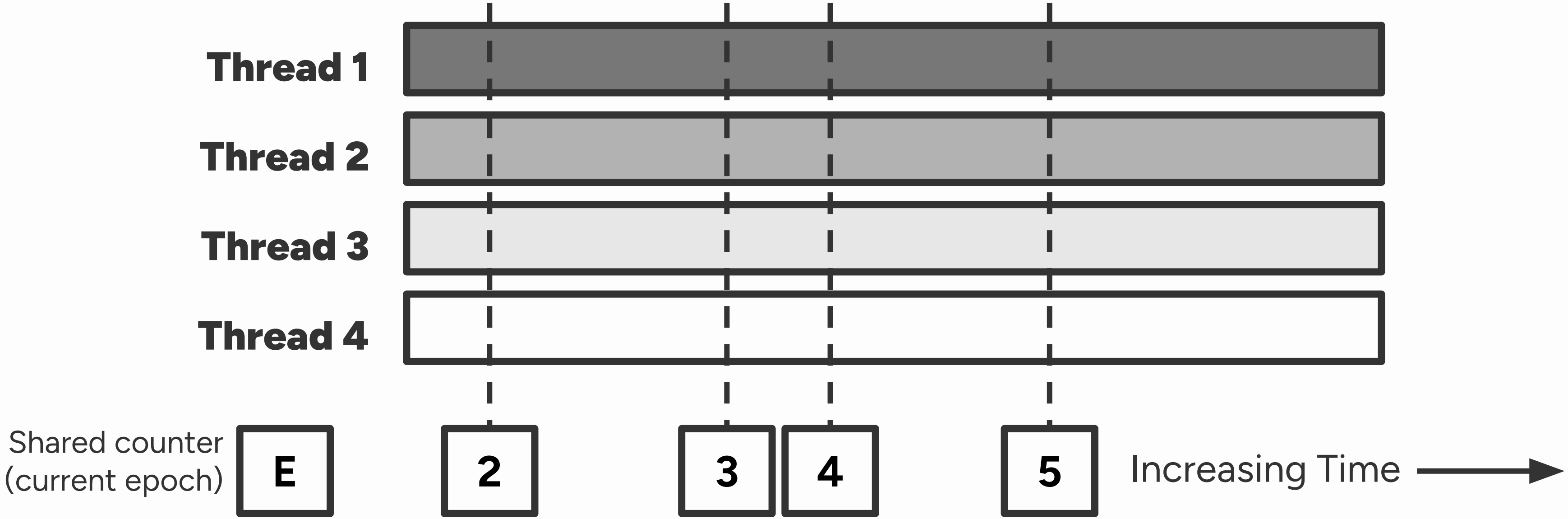
Efficient Concurrency

Independent thread operations
Maintains global consistency

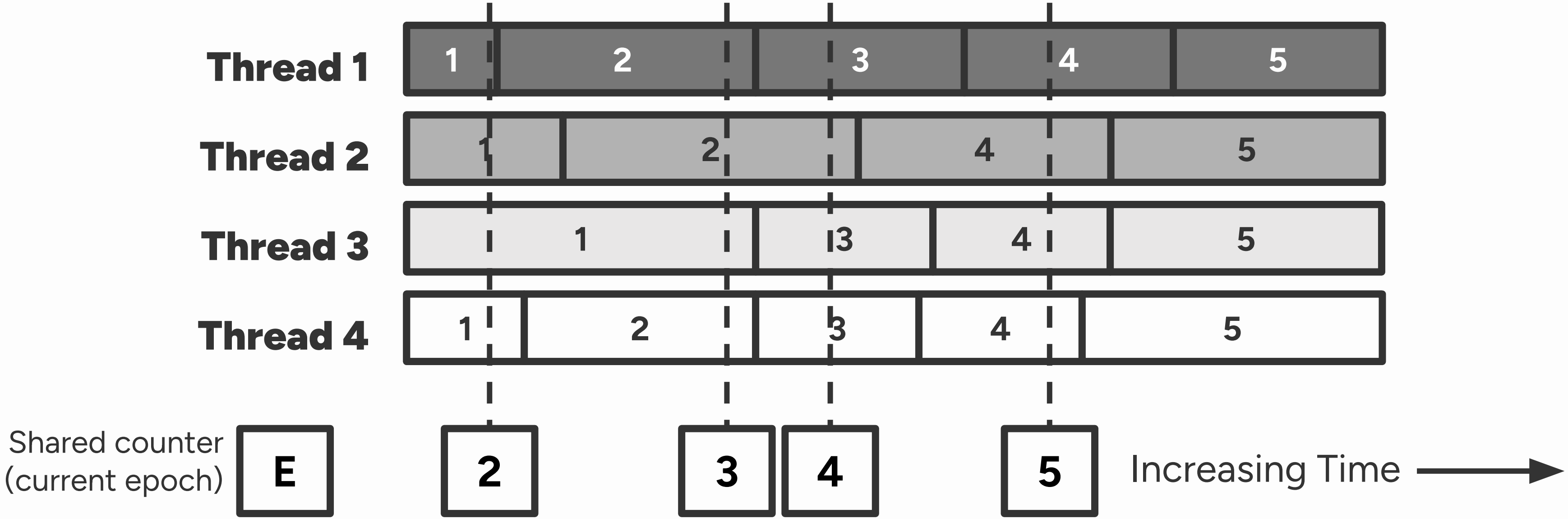
Latch-Free and Scalability

Avoids latches
Improves scalability using scalable thread model

Lazy Synchronization (Epoch-Protection)

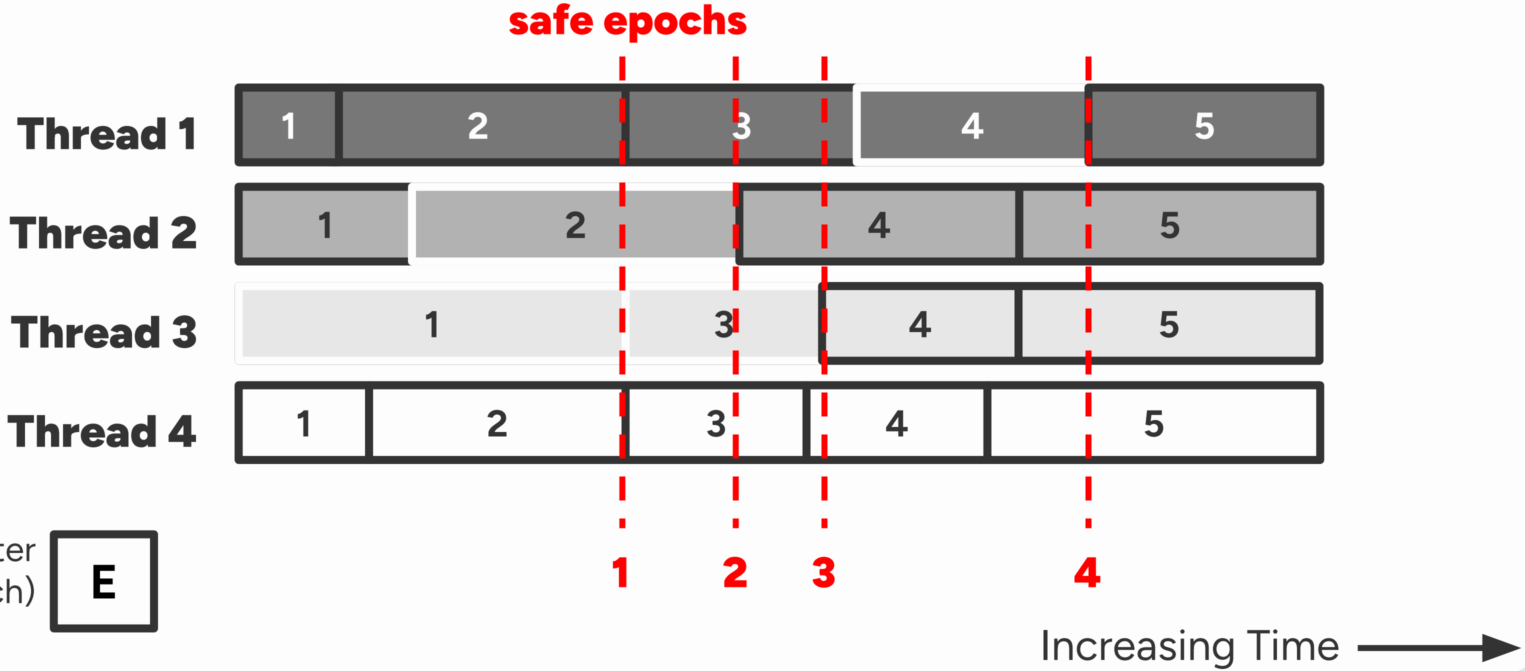


Lazy Synchronization (Epoch Protection)



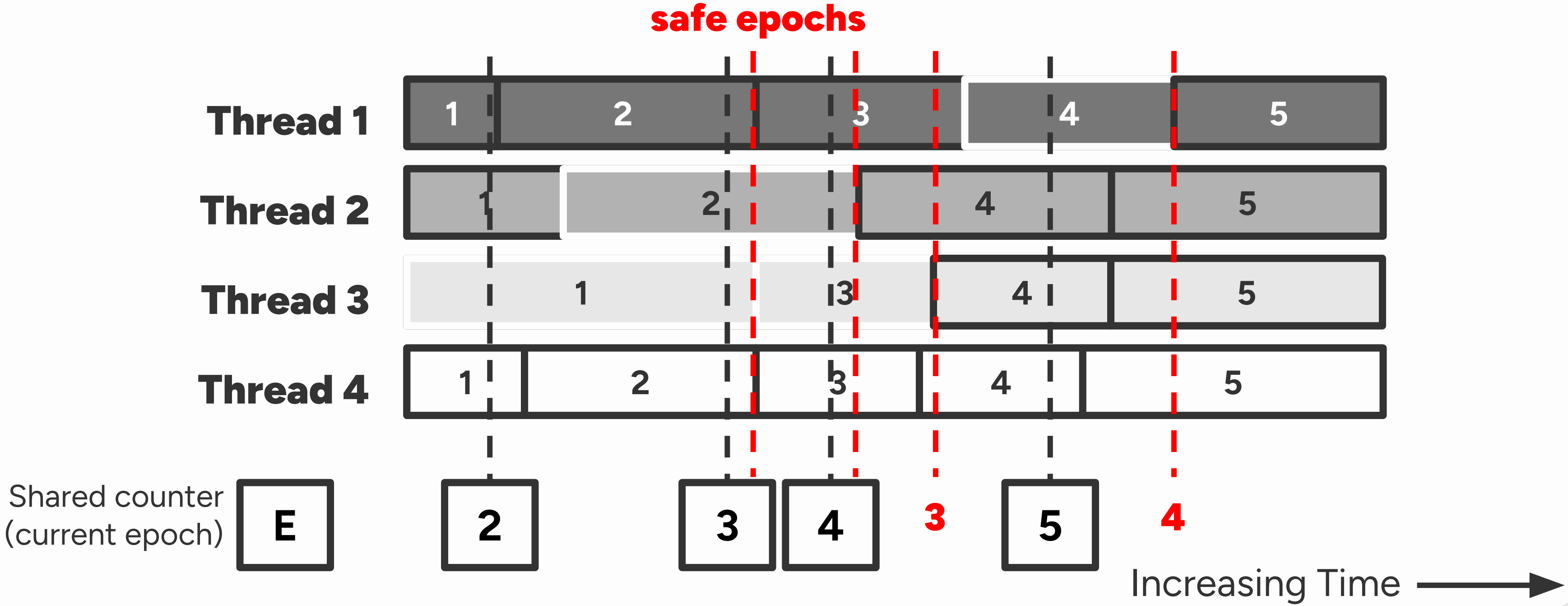
Each thread keeps stale local epoch counter copied from E

Lazy Synchronization (Epoch Protection)



Epoch c is considered safe if all thread-local values are greater than c

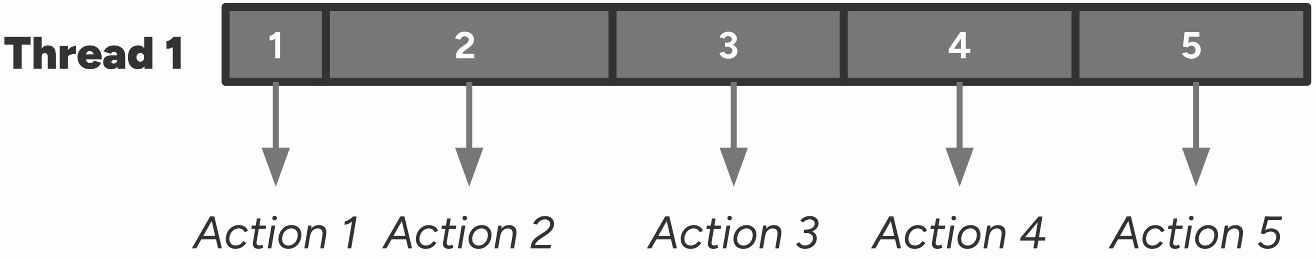
Lazy Synchronization (Epoch Protection)



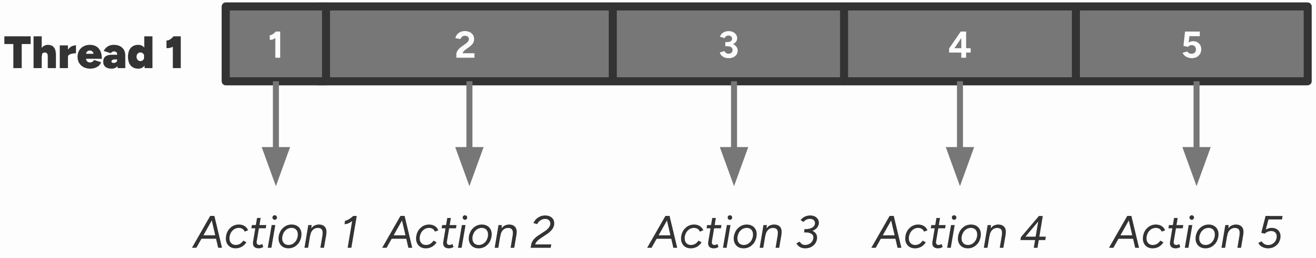
Epoch c is considered safe if all thread-local values are greater than c

Lazy Synchronization (Trigger Actions)

SImplifies synchronization in a multithreaded system



Lazy Synchronization (Trigger Actions)



Thread 1

1 2 3 4 5

Action 1 *Action 2* *Action 3* *Action 4* *Action 5*

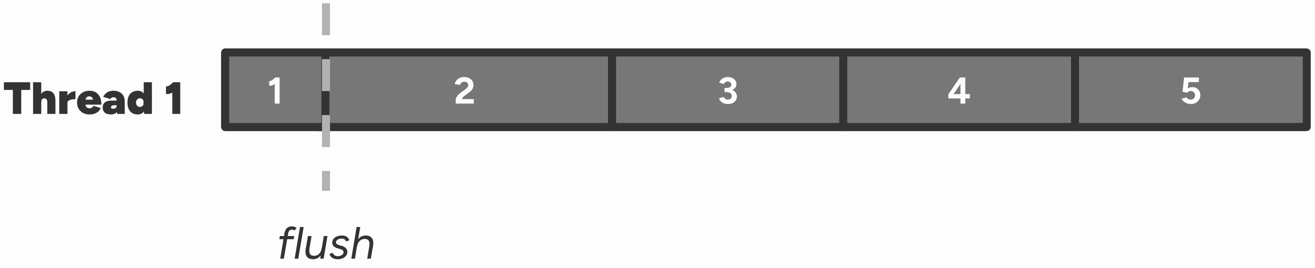
Drain List (epoch, action)

1, Action 1	2, Action 2	3, Action 3	4, Action 4	5, Action 5

Lazy Synchronization (Example)

Drain List (epoch, action)

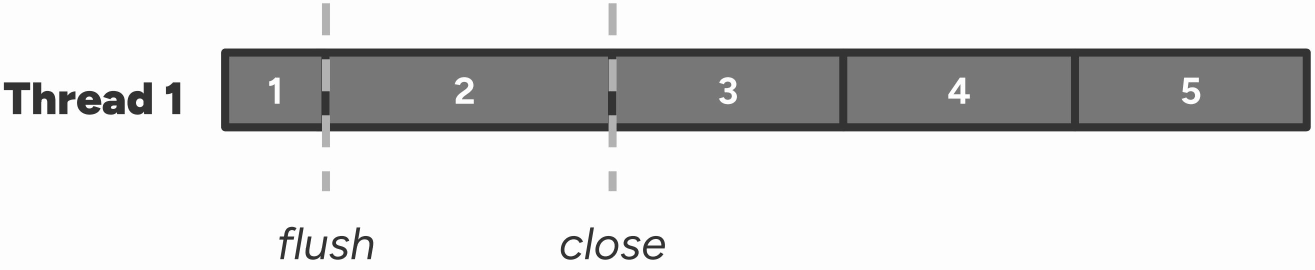
Lazy Synchronization (Example)



Drain List (epoch, action)

1, flush				

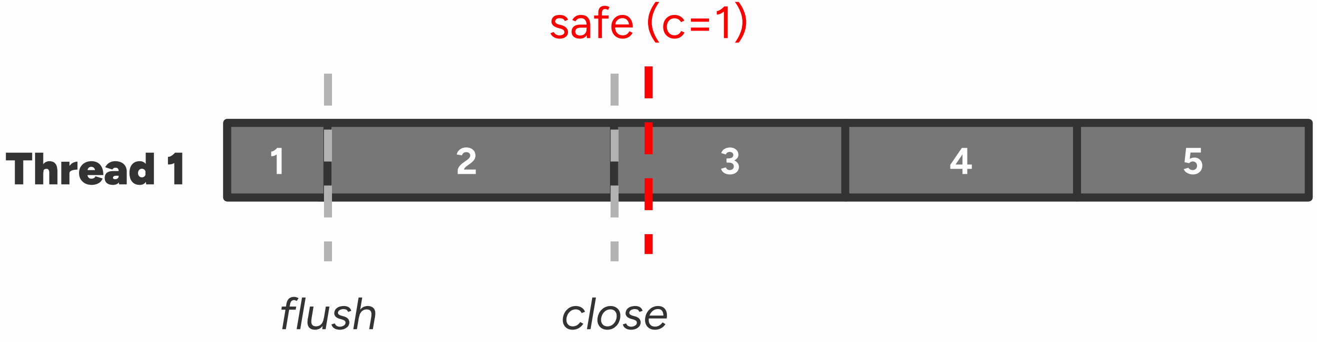
Lazy Synchronization (Example)



Drain List (epoch, action)

1, flush	2, close			

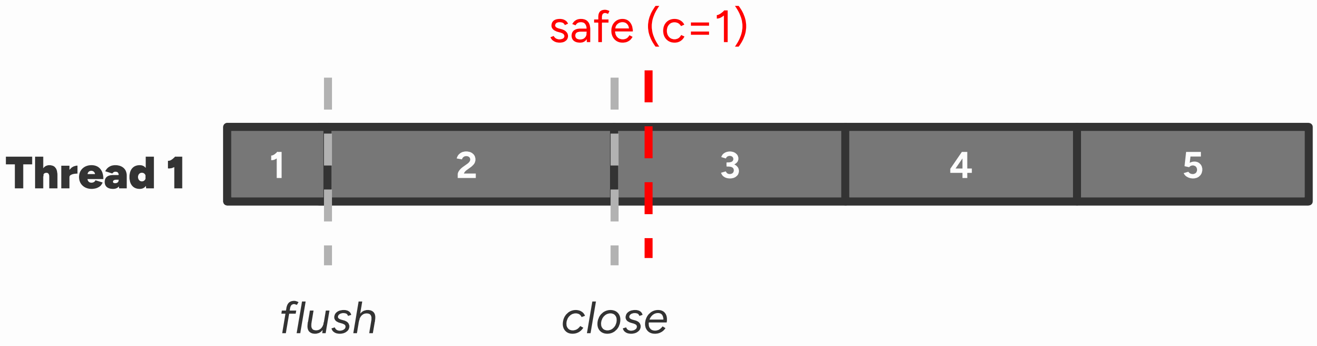
Lazy Synchronization (Example)



Drain List (epoch, action)

1, flush	2, close			

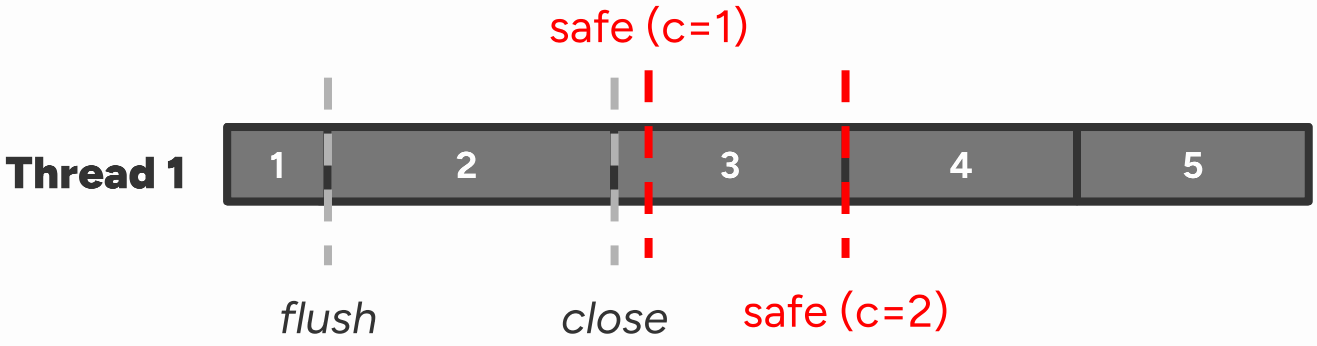
Lazy Synchronization (Example)



Drain List (epoch, action)

	2, close			

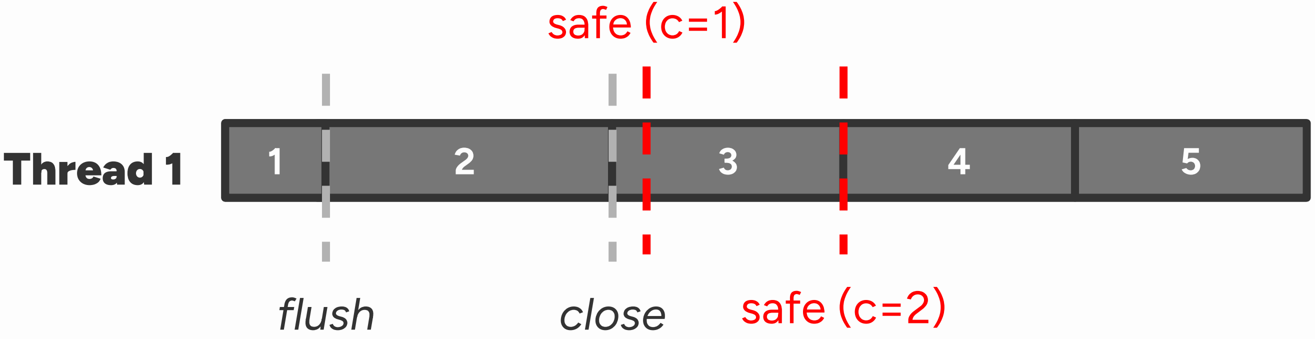
Lazy Synchronization (Example)



Drain List (epoch, action)

	2, close			

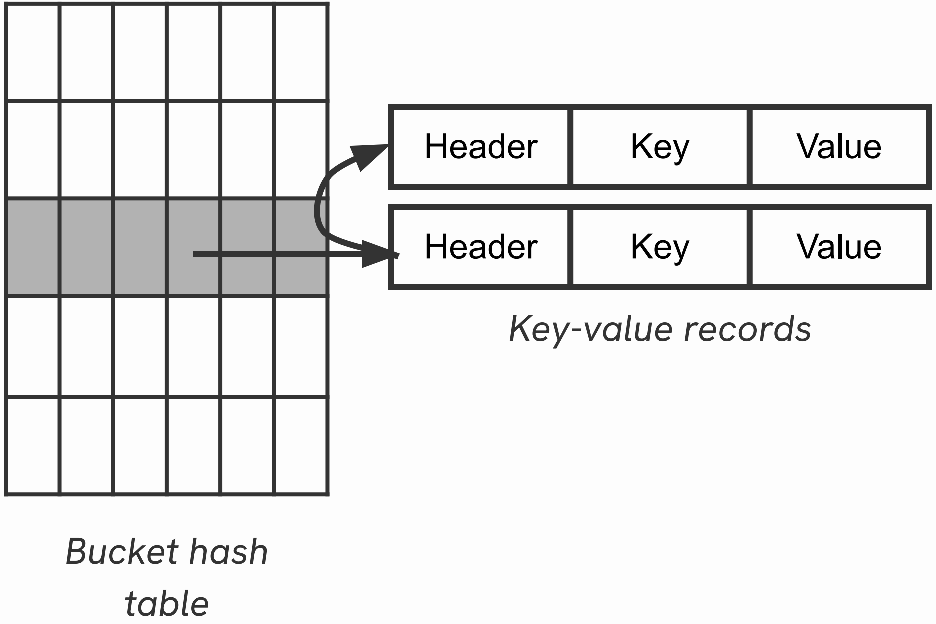
Lazy Synchronization (Example)



Drain List (epoch, action)

Architecture Overview

Hash Index

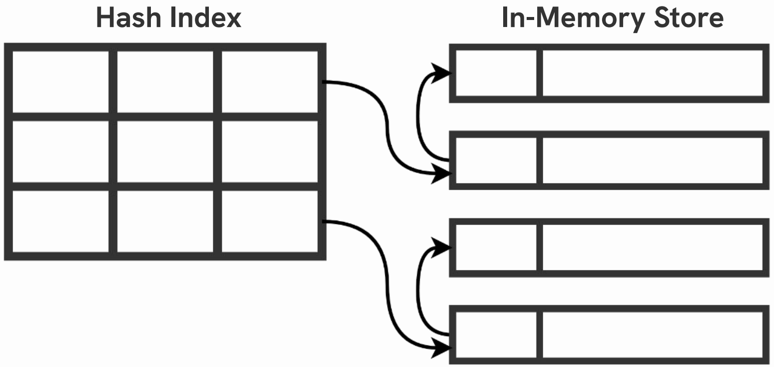


- ☑ **Efficient point queries**
- ☑ **Supports concurrency**
- ☑ **Scalability**

Allocators (In-Memory and Append-Only)

In-Memory Allocator

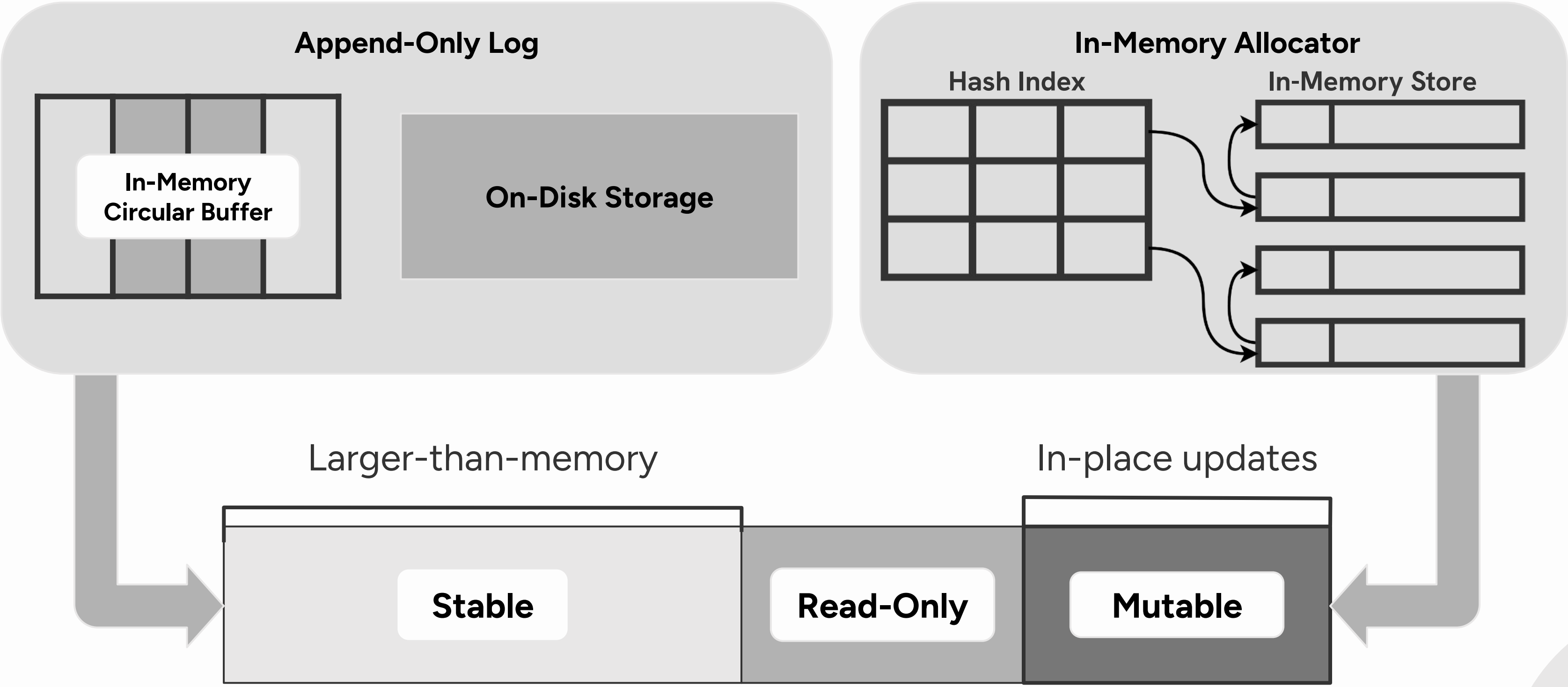
Features: In-place updates, latch-free access



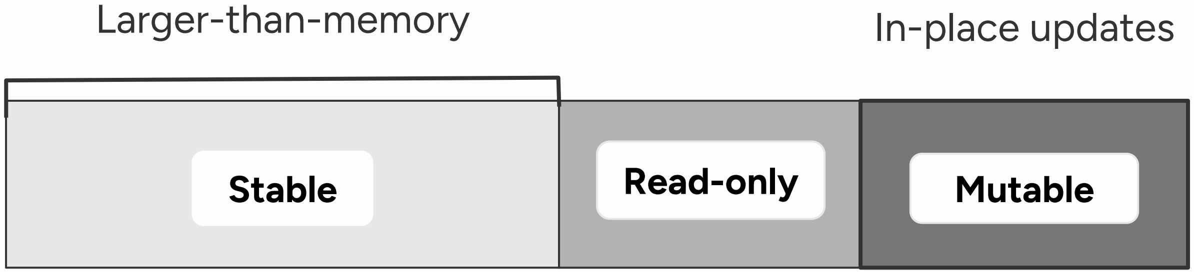
Append-Only Log

Features: Larger-than-memory, latch-free access

Allocators (HybridLog)

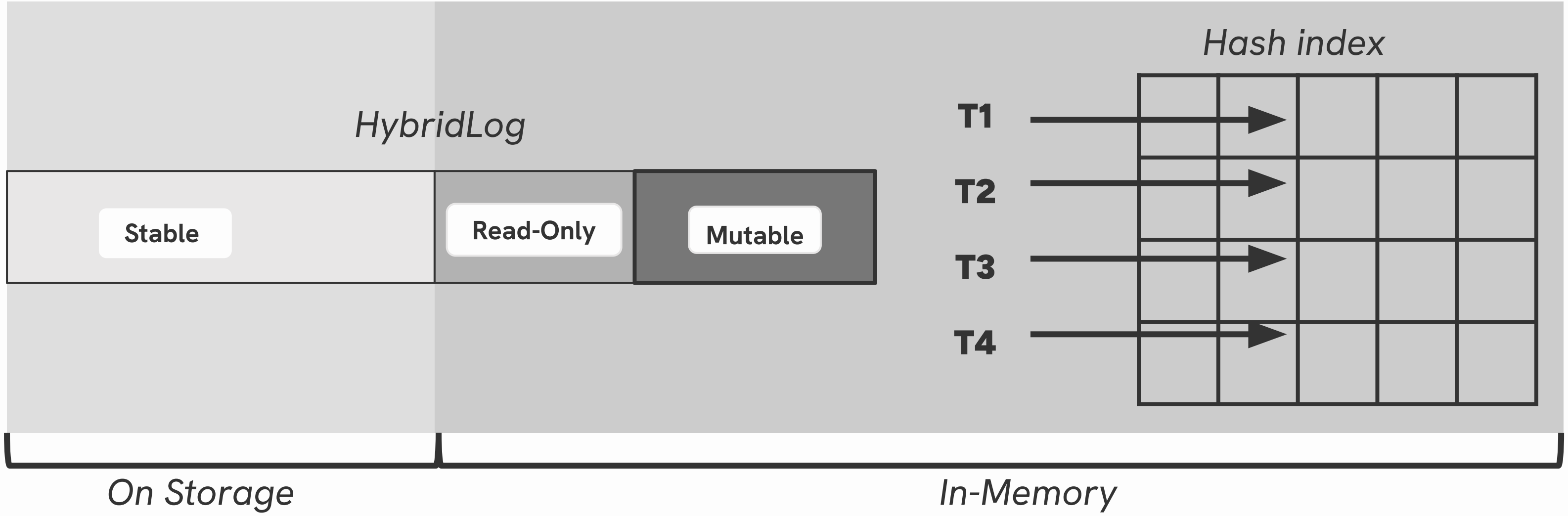


Allocators (HybridLog)



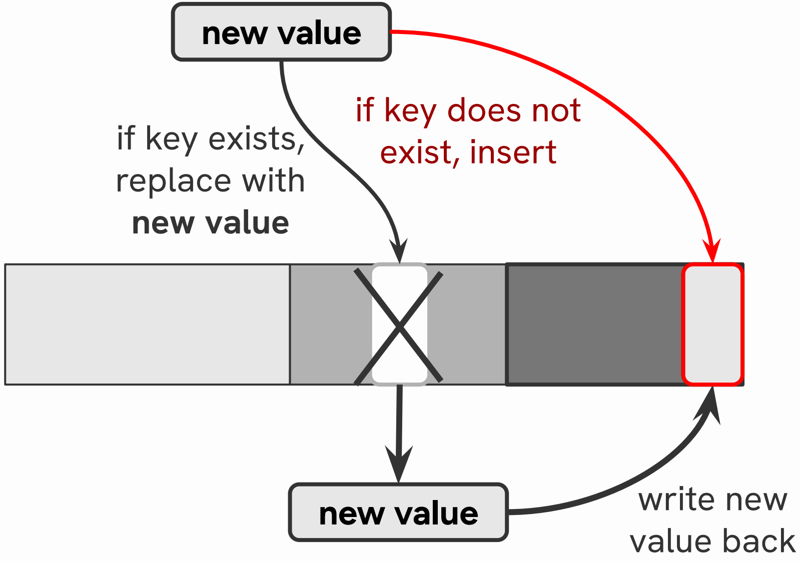
- ☑ **Latch-free**
- ☑ **In-place updates**
- ☑ **Handles larger-than-memory data**

In-Memory vs. On Storage



Operation Definitions

Upserts (Blind Updates)



new value

if key exists, replace with **new value**

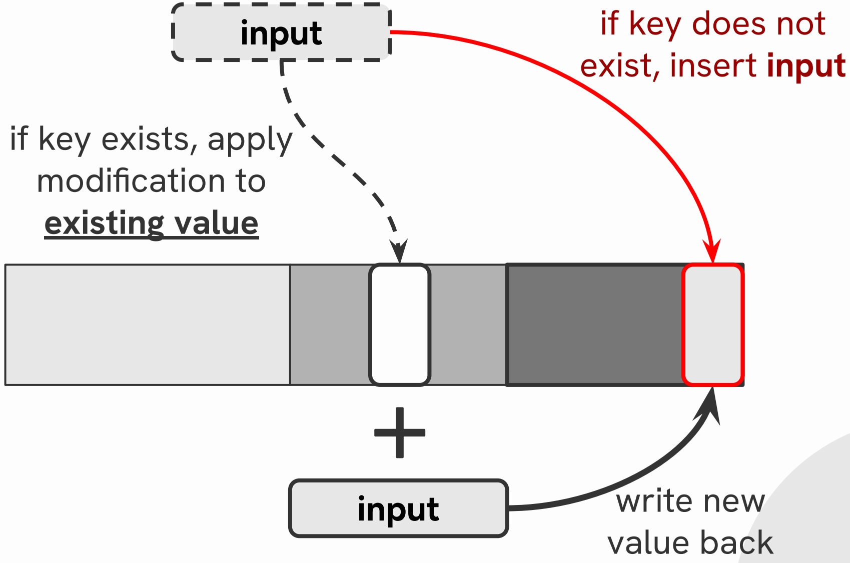
if key does not exist, insert

new value

write new value back

Read-Modify-Write (RMW)

Example: summation-based update



input

if key does not exist, insert **input**

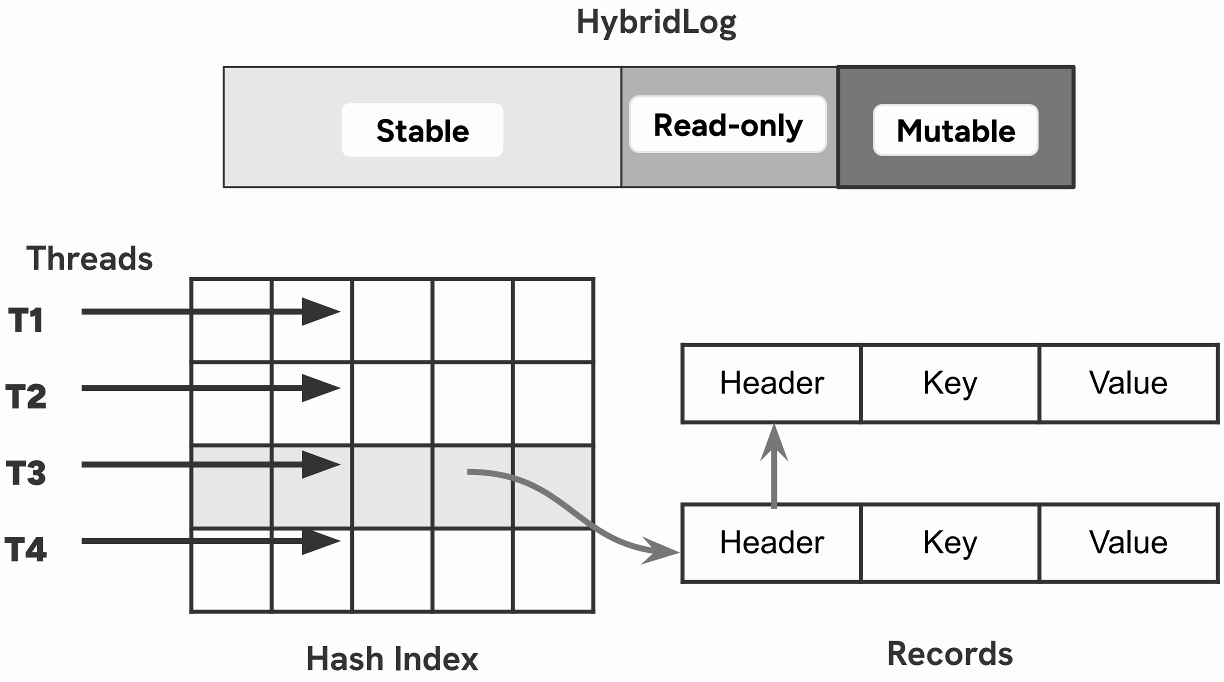
if key exists, apply modification to **existing value**

+

input

write new value back

Overall FASTER Architecture



HybridLog

Stable

Read-only

Mutable

Threads

T1

T2

T3

T4

Hash Index

Header	Key	Value

Header	Key	Value

Records

FASTER 's Hash Index

Features:

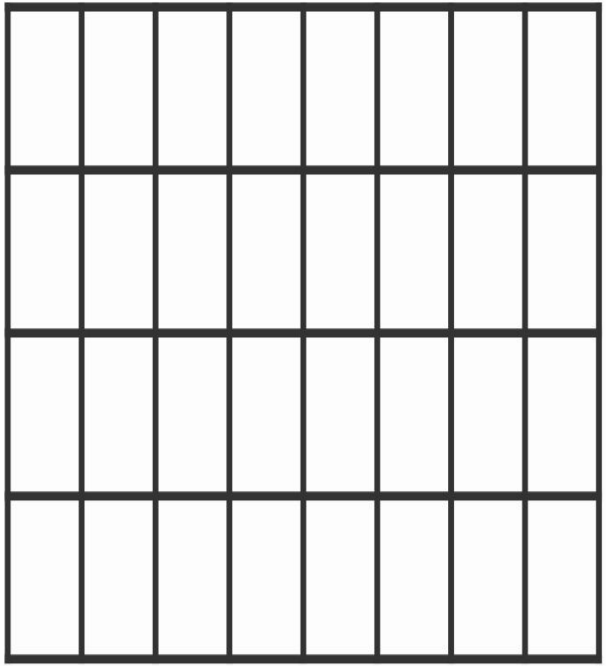
Concurrent

Latch-Free

Scalable

Resizable

FASTER Hash Index

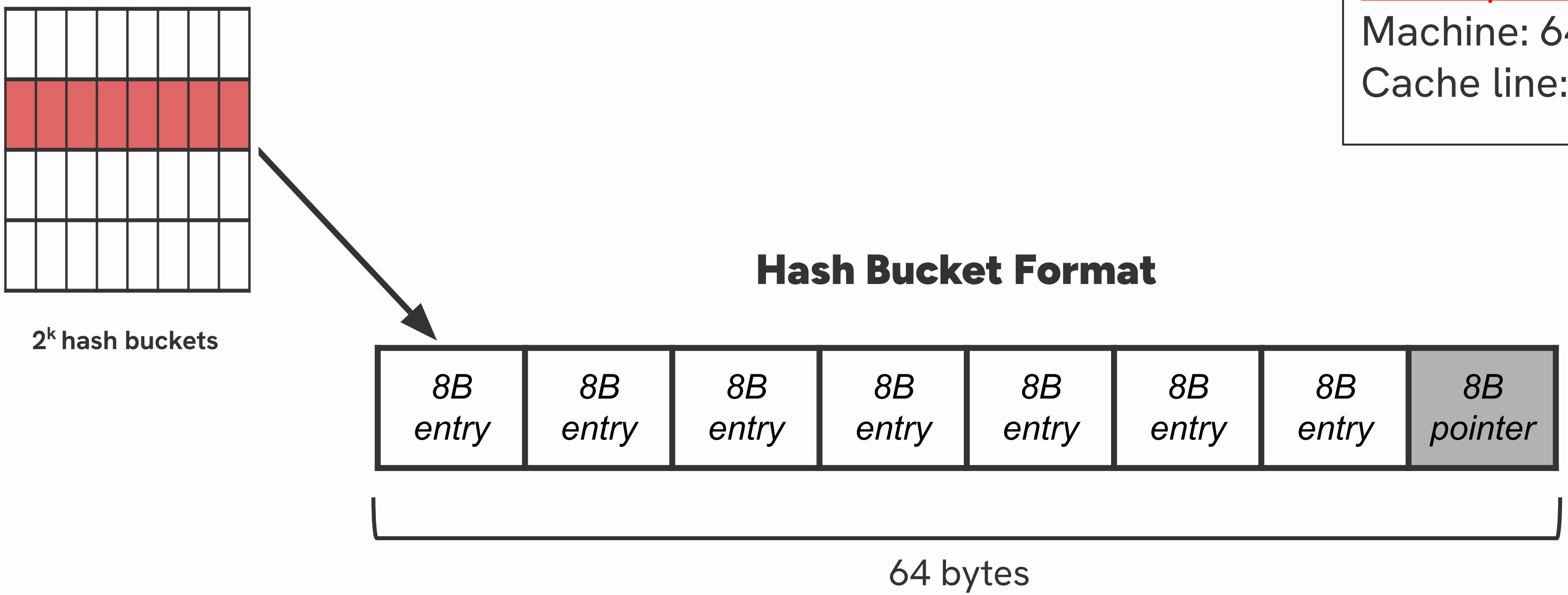


2^k hash buckets

Assumptions:
Machine: 64 bits
Cache line: 64 bytes

"FASTER Index is a ***cache-aligned*** array"

FASTER Hash Index



2^k hash buckets

Hash Bucket Format

8B entry	8B entry	8B entry	8B entry	8B entry	8B entry	8B entry	8B pointer

64 bytes

Assumptions:

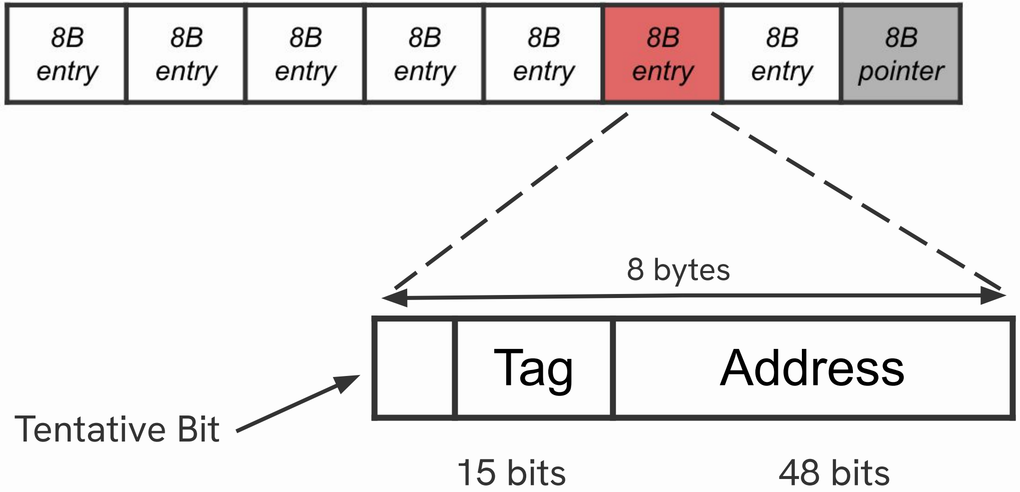
Machine: 64 bits

Cache line: 64 bytes

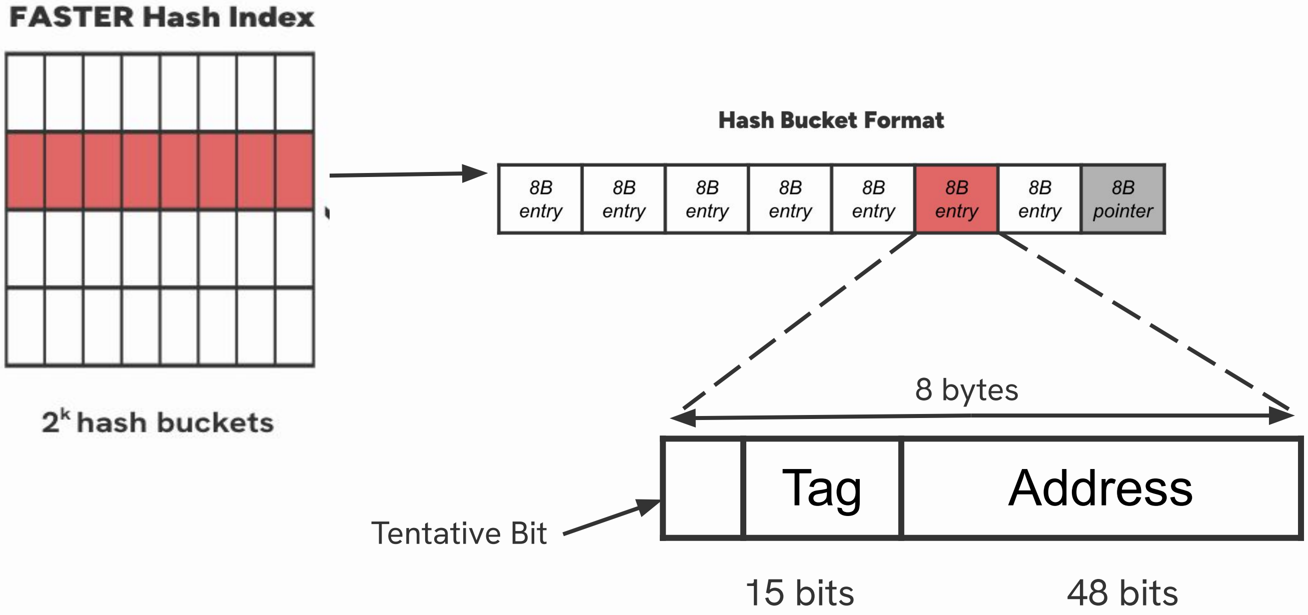
7 entries per bucket (8 bytes each)

1 overflow bucket pointer (8 bytes)

Hash Bucket Format



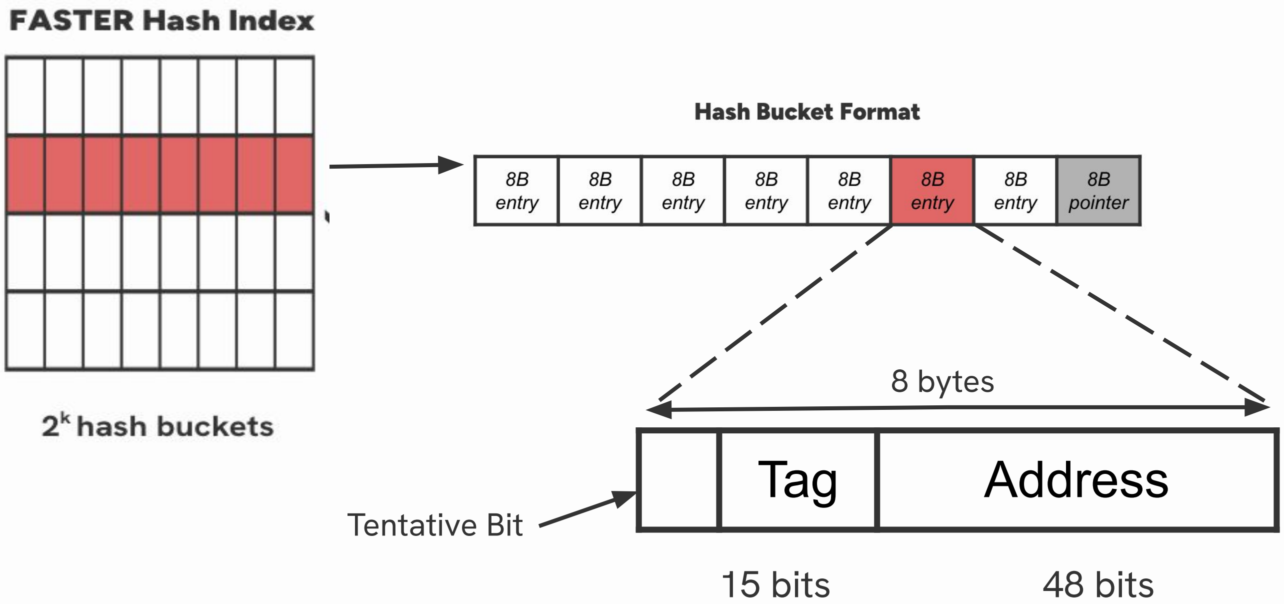
Address: Physical or logical place in memory
Tag: Increase hashing resolution
Tentative Bit: Used to keep latch-free concurrency



*Hash value: **h***

First *k* bits = **offset**
(which bucket)

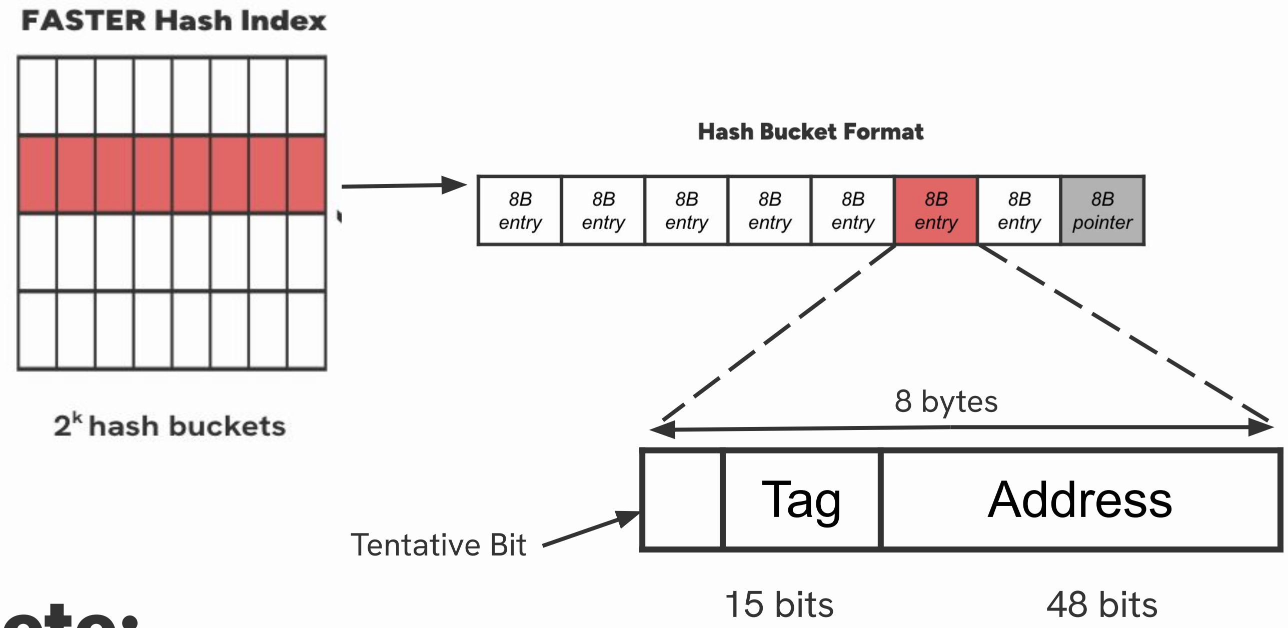
Next 15 bits = **tag**
(entry within bucket)



Search:

h → *(offset,tag)*

1. Find offset of bucket (first *k* of ***h***)
2. Scan through bucket to find tag



Delete:

h → (*offset,tag*)

1. Find offset of bucket (first *k* of ***h***)
2. Scan through bucket to find tag
3. **Replace matching entry with *zero* (compare-and-swap)**

Reminder:

(offset,tag) must be unique
AND
FASTER Hash Index is **Concurrent**

FOR THE CLASS:

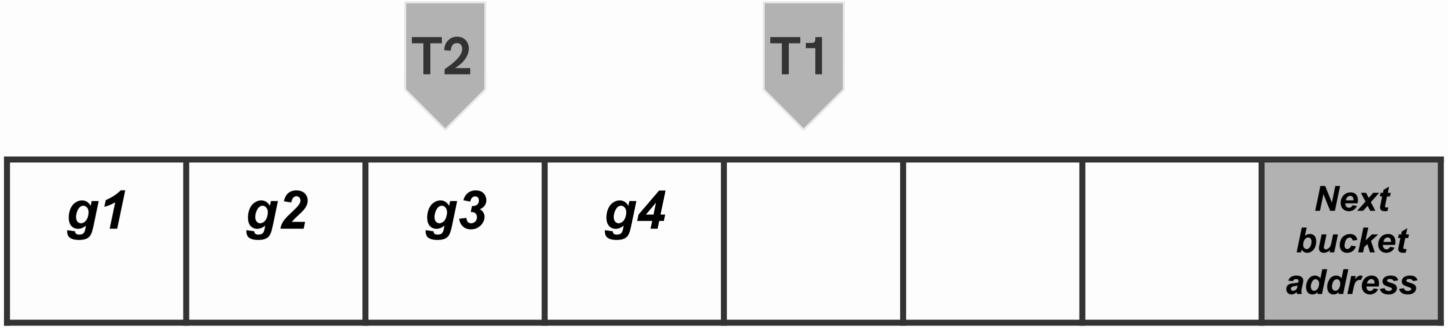
What problems do you see arising with inserts?

g1	***g2***	***g3***	***g4***				***Next bucket address***

Scenario:

T1: Insert g5

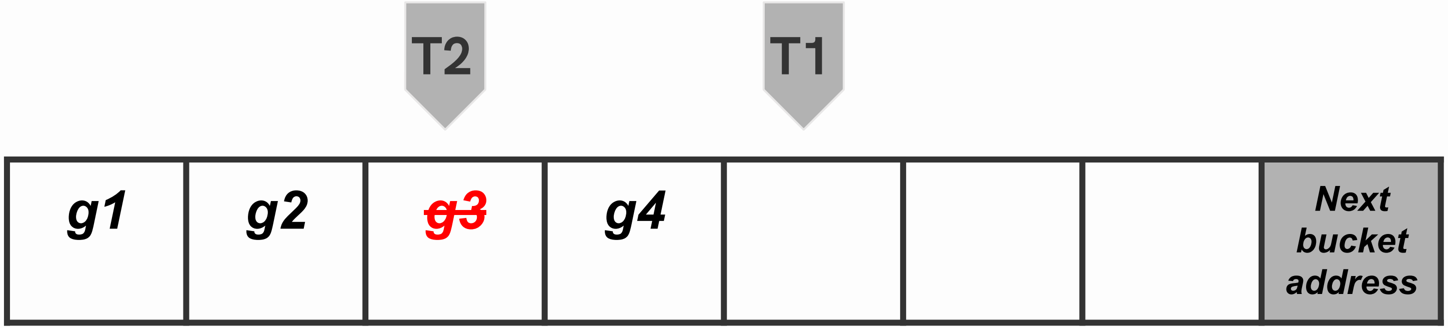
T2: Delete g3 AND insert g5



Scenario:

T1: Insert g5

T2: Delete g3 AND insert g5



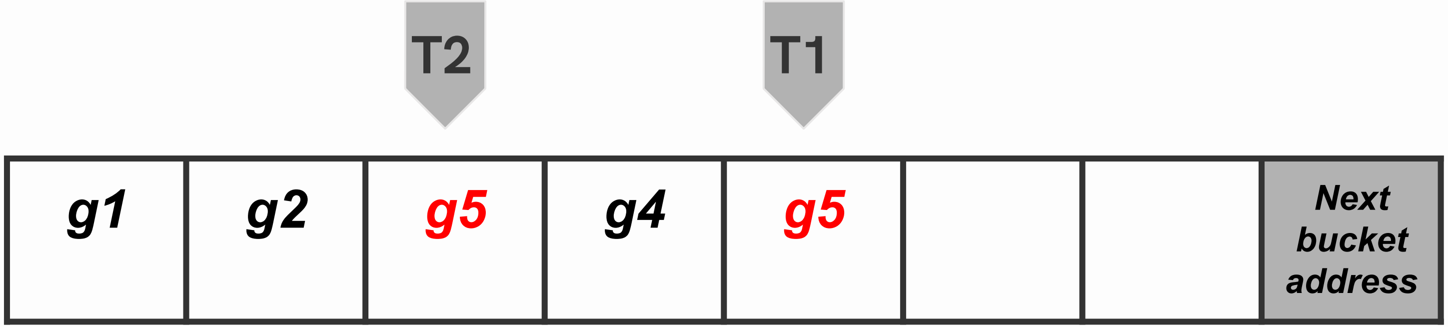
FOR THE CLASS:

What is the problem here?

Scenario:

T1: Insert g5

T2: Delete g3 AND insert g5

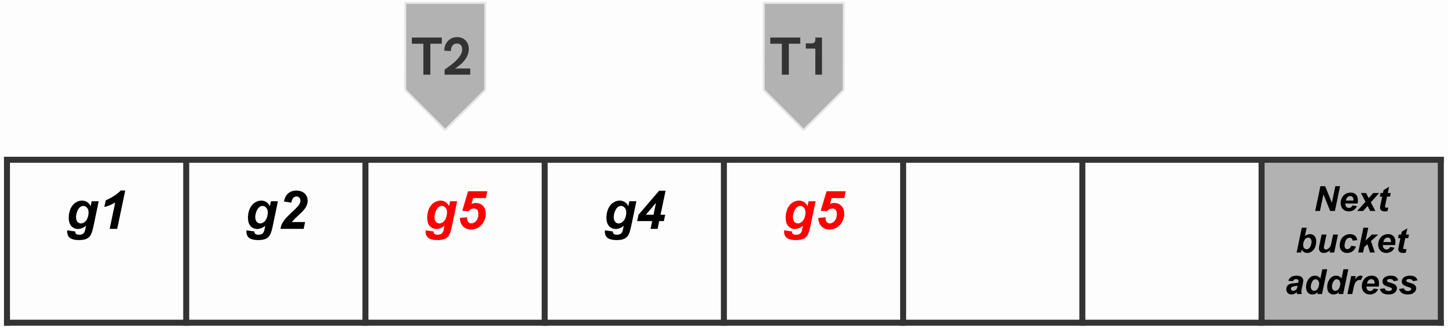


Same offset,tag inserted!

Scenario:

T1: Insert g5

T2: Delete g3 AND insert g5



g1	*g2*	*g5*	*g4*	*g5*			*Next bucket address*

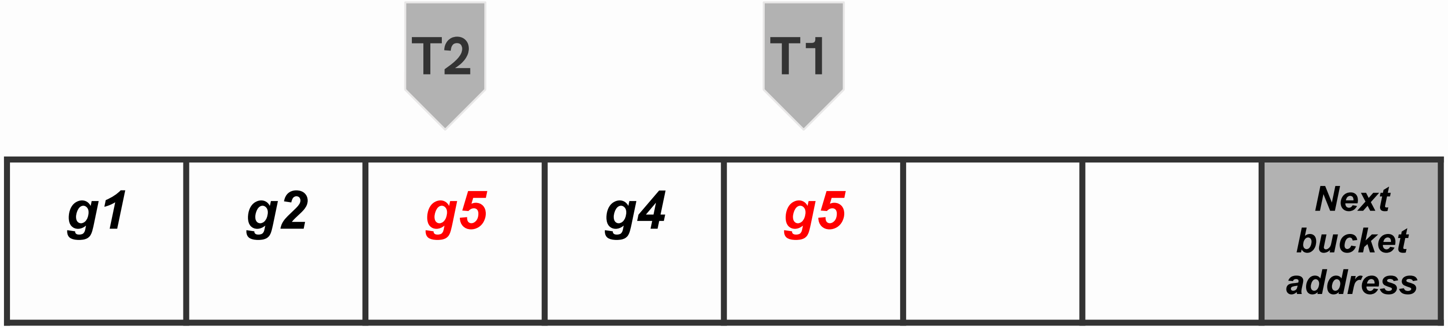
FOR THE CLASS:

Why do we not just lock the bucket?

Scenario:

T1: Insert g5

T2: Delete g3 AND insert g5

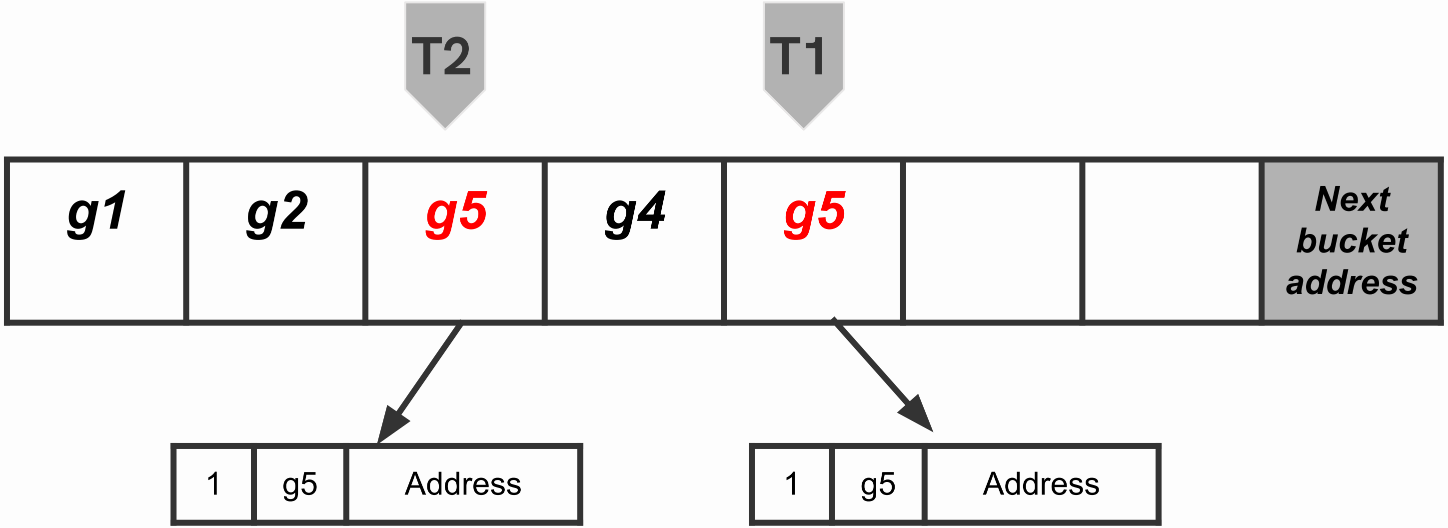


Concurrent **Latch-Free** **Scalable** **Resizable**

How do we maintain concurrency that is ***latch-free?***

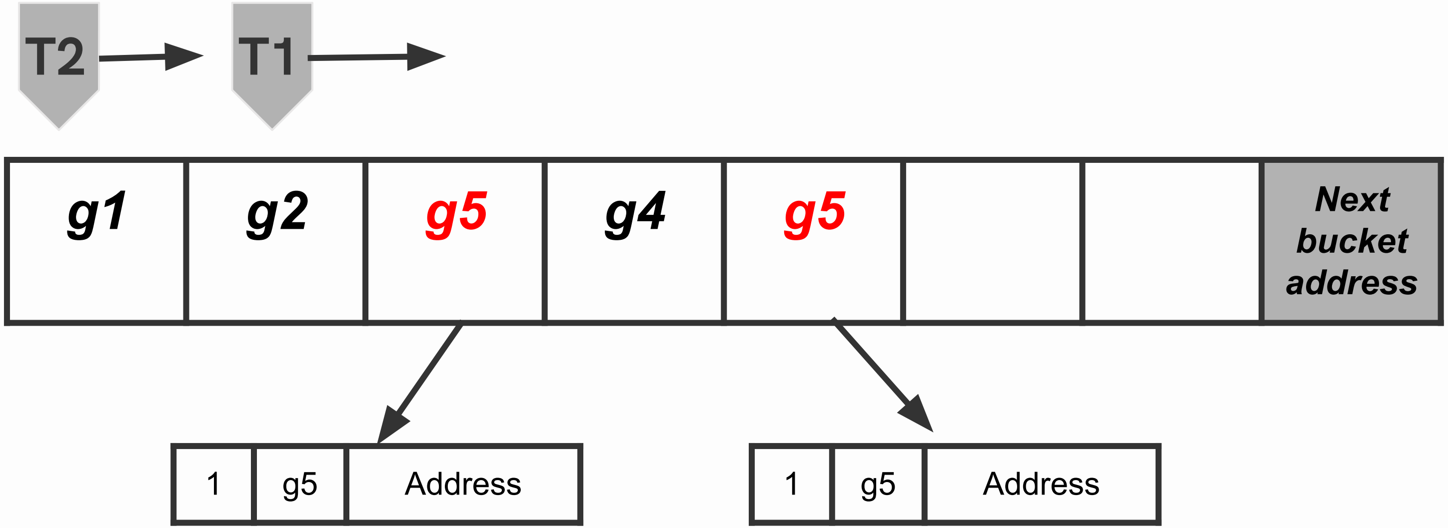
Latch-free two-phase Insert Algorithm

1. Insert record with tentative bit **set**



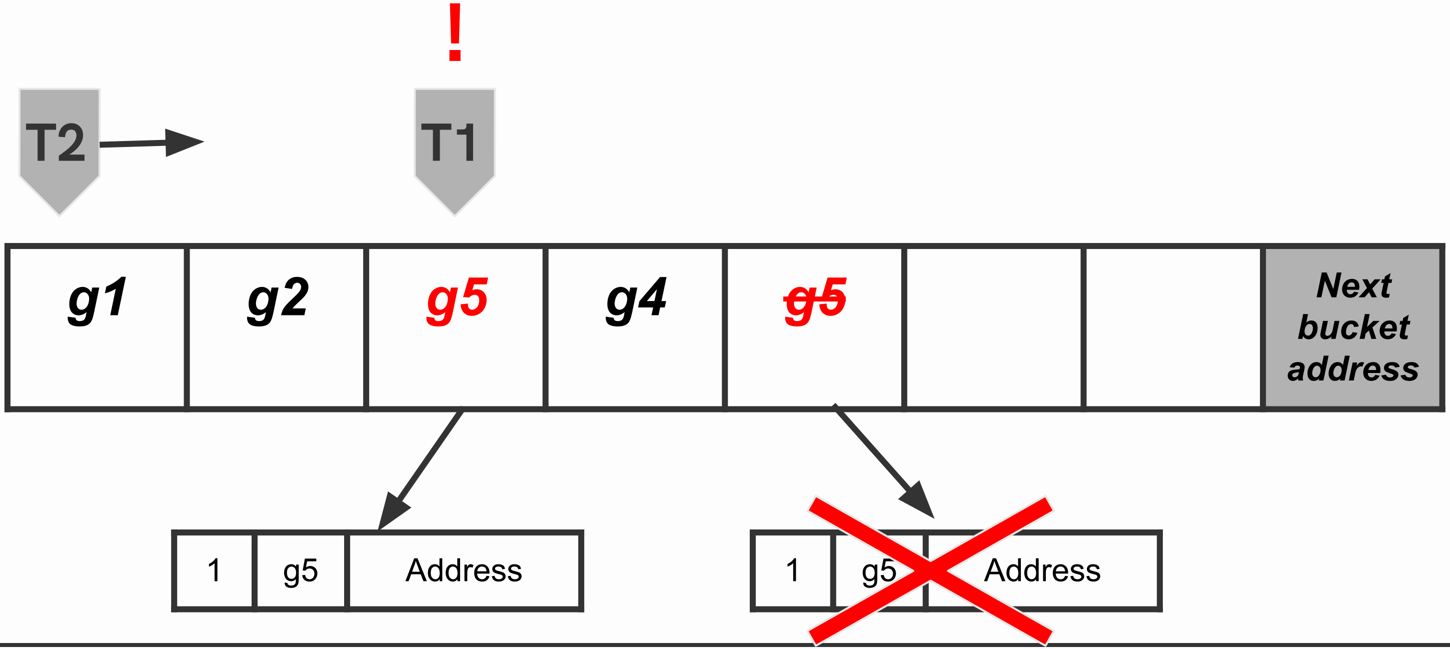
Latch-free two-phase Insert Algorithm

2. Rescan bucket for duplicate tag



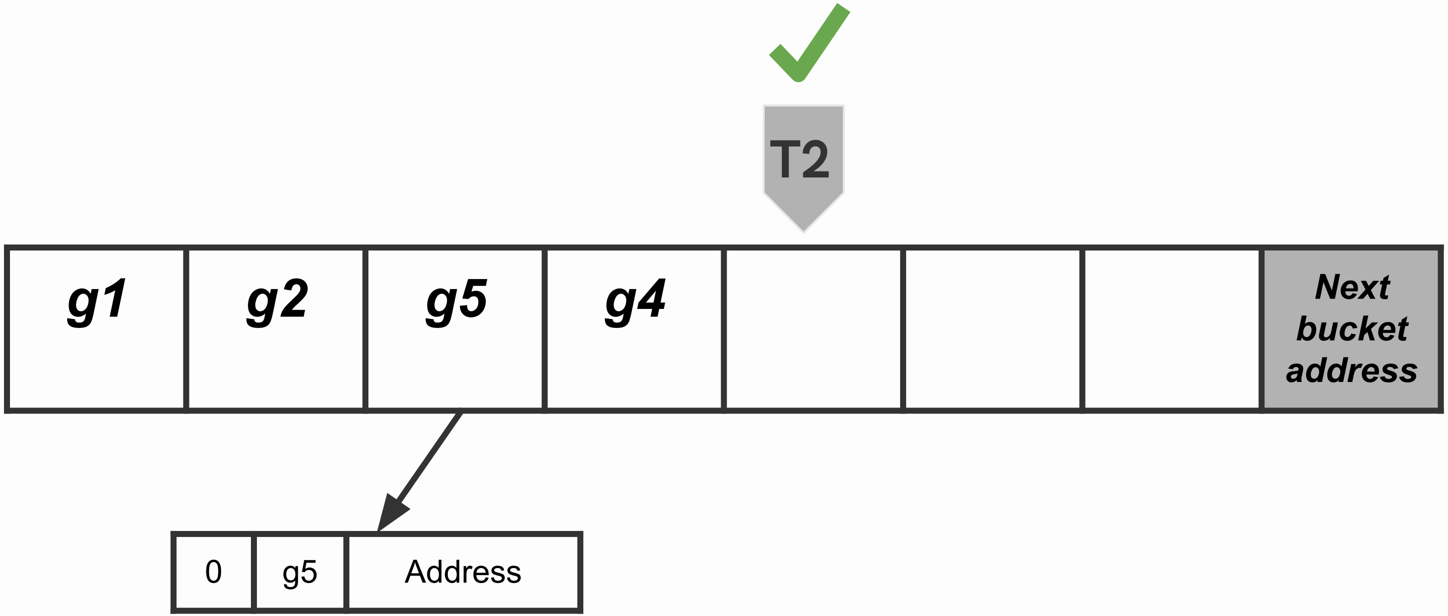
Latch-free two-phase Insert Algorithm

2. If a match is found: back off and retry



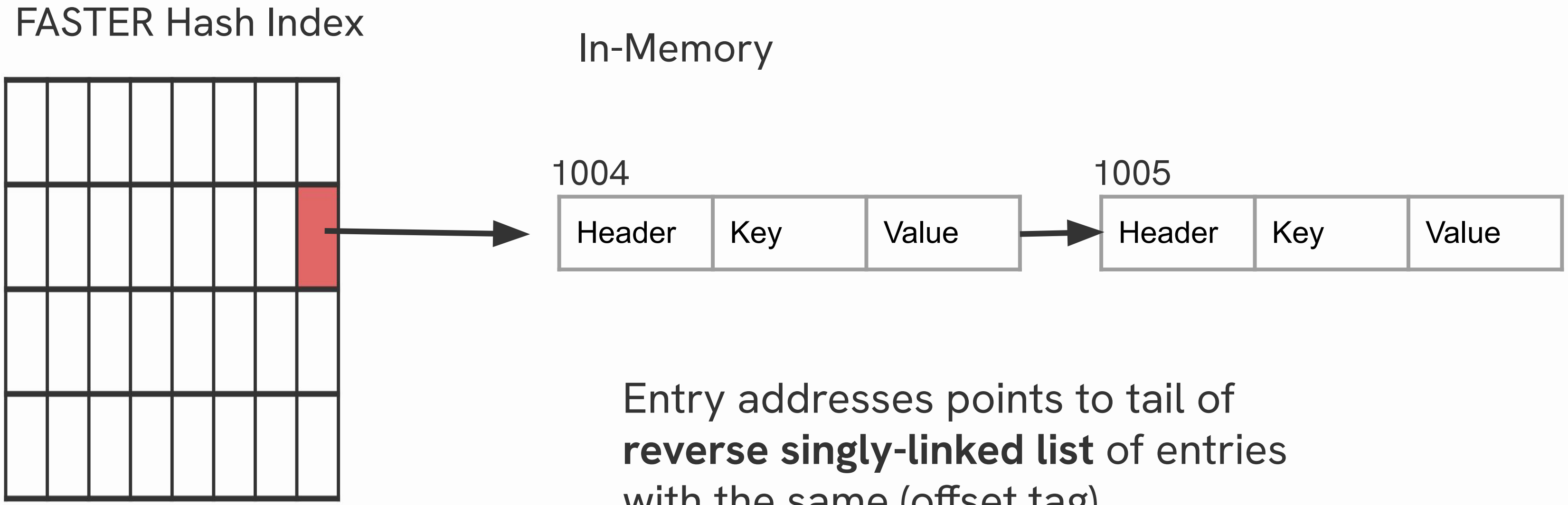
Latch-free two-phase Insert Algorithm

2. Otherwise: reset tentative bit to finalize



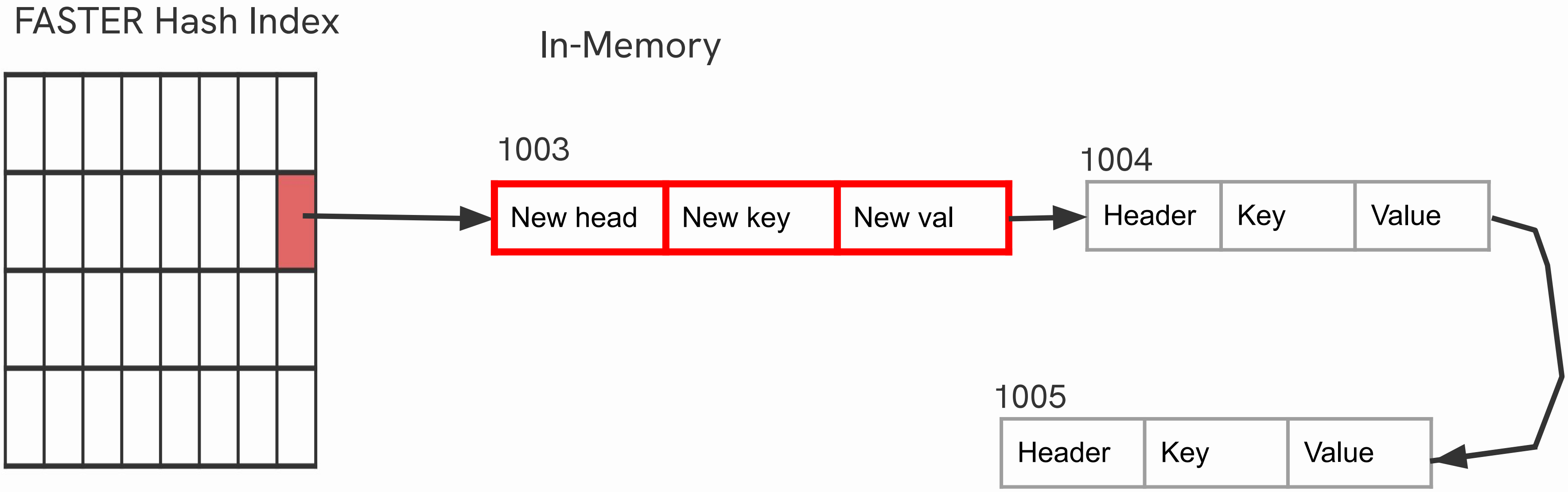
In-Memory Key Value Store

Structure Of In-Memory Store



Entry addresses points to tail of **reverse singly-linked list** of entries with the same (offset,tag)

Structure Of In-Memory Store



Data Larger Than Memory

FASTER is designed to support frequent in-place updates AND large data…

Data Larger Than Memory

FASTER is designed to support frequent in-place updates AND large data…

How do we proceed if the data does not fit in memory?

The “Strawman” Solution

An **append-only log** using a **circular buffer**

Manage flushing and eviction safety using **epochs with triggers**

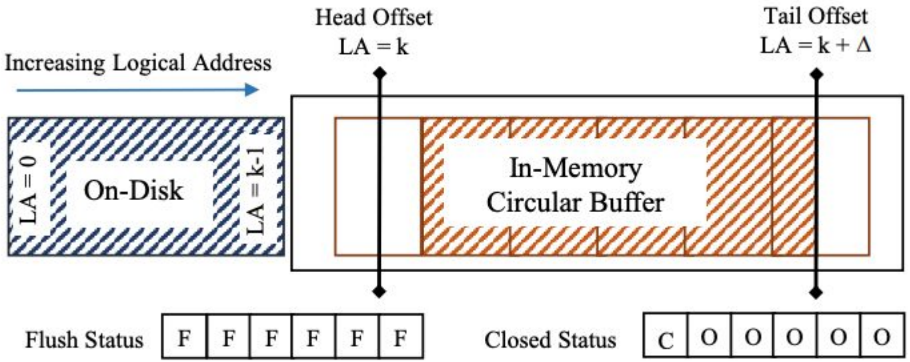


Figure 4: Tail Portion of the Log-Structured Allocator

Append-Only Log

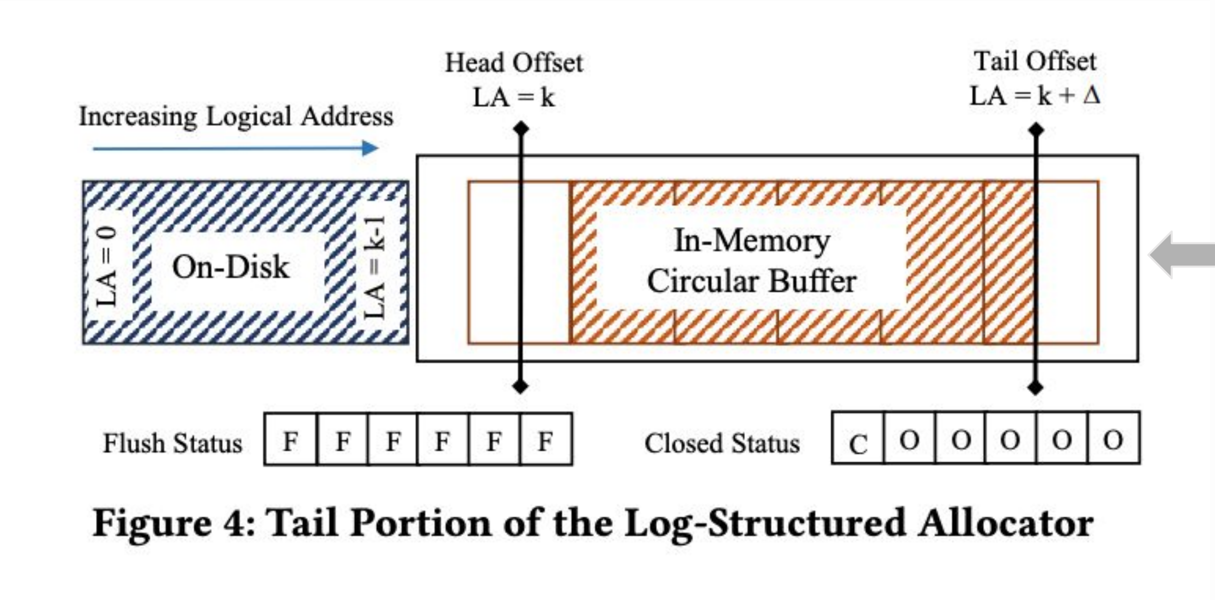


Figure 4: Tail Portion of the Log-Structured Allocator

Blind Updates: Appended to tail, update hash

Append-Only Log

Read +
async I/O

Read

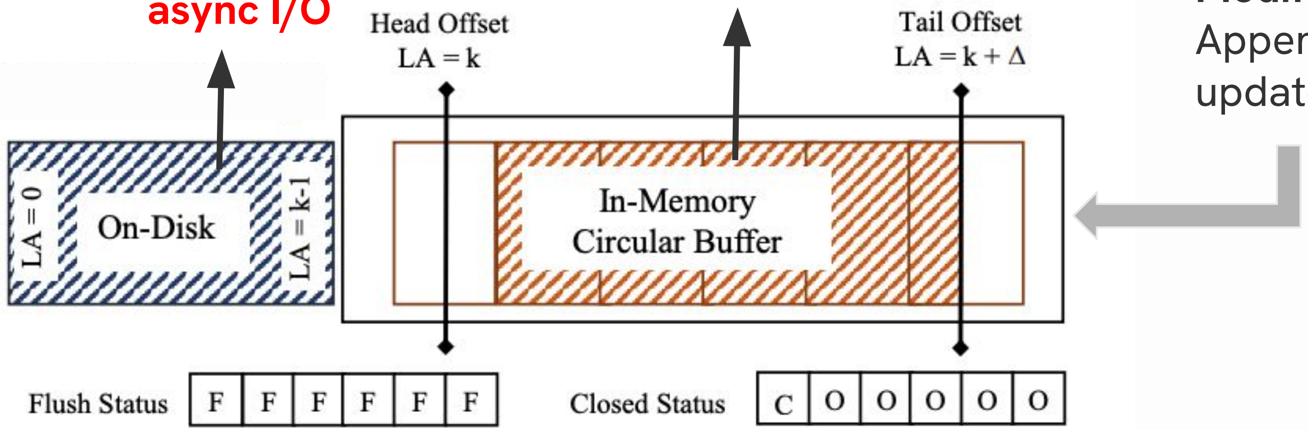


Figure 4: Tail Portion of the Log-Structured Allocator

Modify+Write:
Appended to tail,
update hash

Append-Only Log

FOR THE CLASS:

What drawbacks jump out with the append-only log, especially for our desired workload?

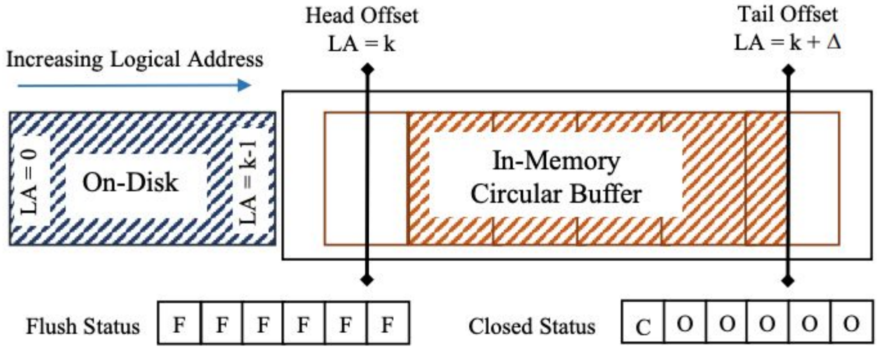


Figure 4: Tail Portion of the Log-Structured Allocator

Our Workload is Update-Intensive!

Every update requires:

- Atomic increment of tail offset
- Copying data from a previous location
- Atomic update of logical address in the hash index

Fast growing append log becomes a **bottleneck**

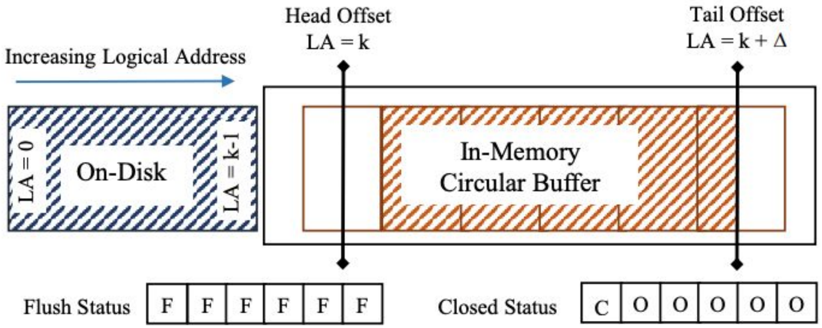
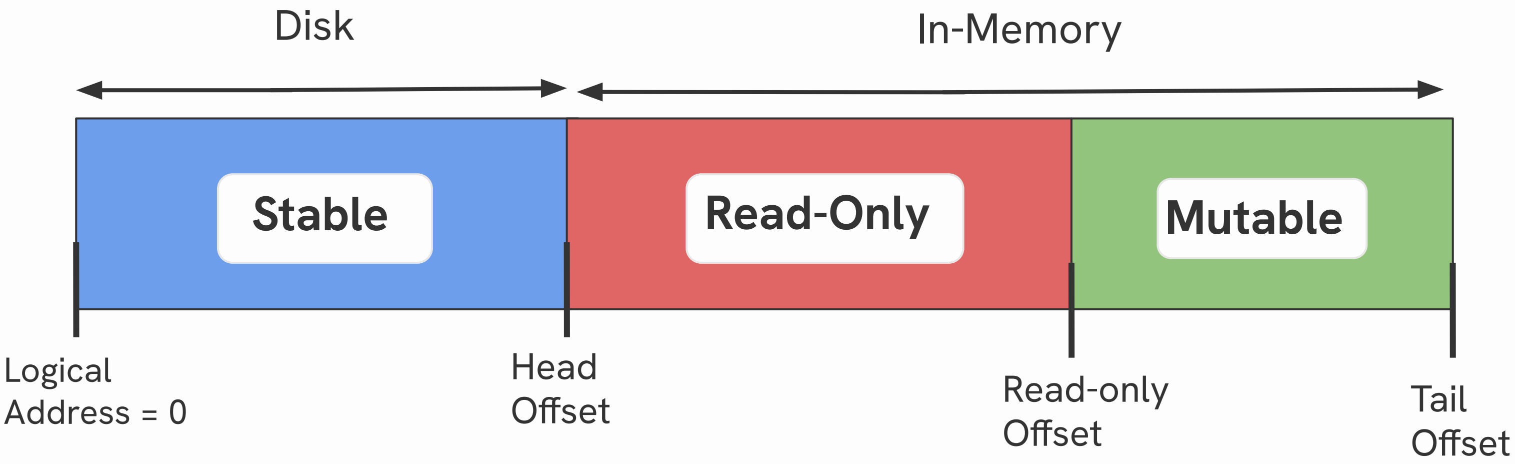


Figure 4: Tail Portion of the Log-Structured Allocator

HybridLog

Solution: The HybridLog

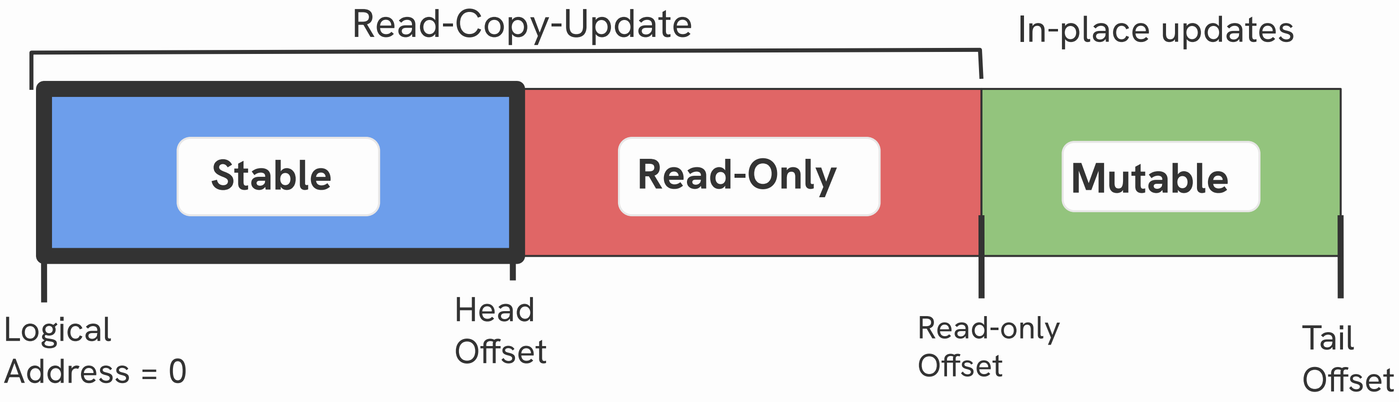


The HybridLog

Logical Address	**Update Action**
Invalid	Make new record on tail end

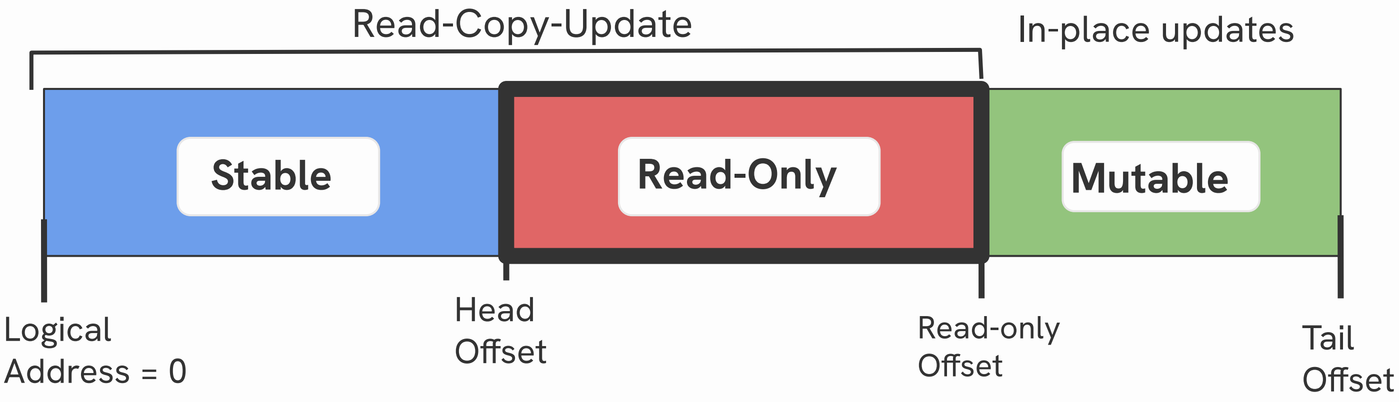
The HybridLog

Logical Address	Update Action
Invalid	Make new record on tail end
< Head Offset	Make async IO request on disk



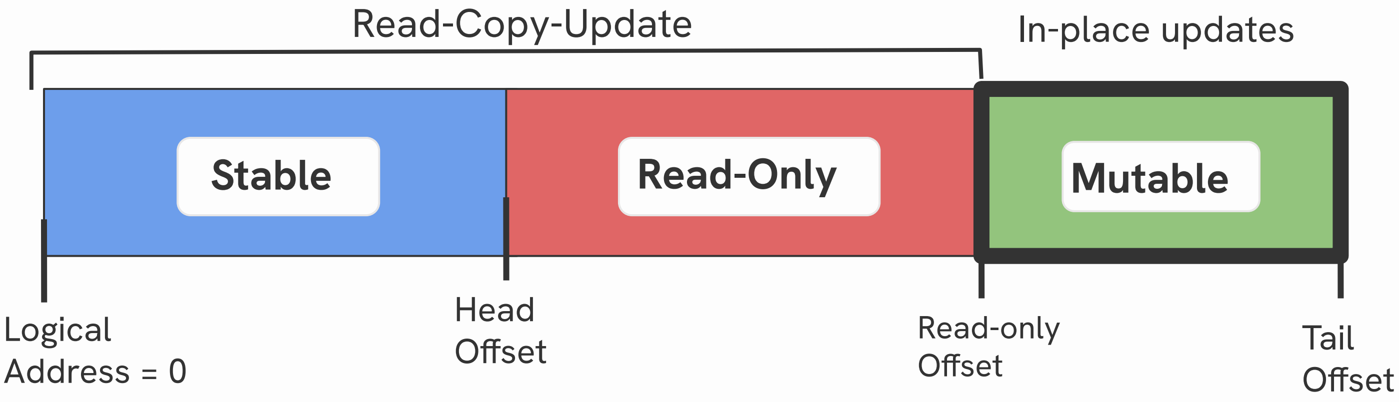
The HybridLog

Logical Address	Update Action
Invalid	Make new record on tail end
< Head Offset	Make async IO request on disk
< Read-only Offset	Make a mutable copy on tail end



The HybridLog

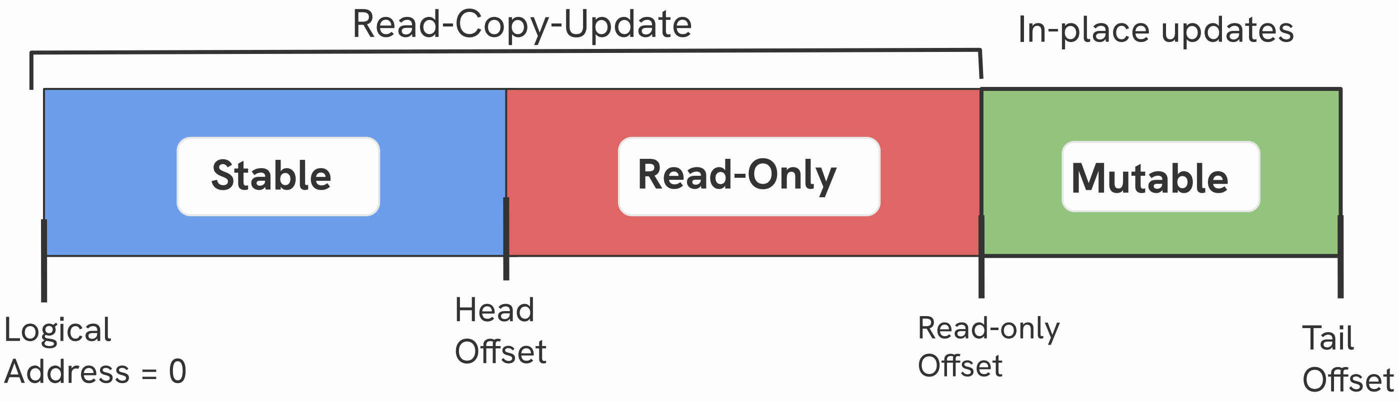
Logical Address	**Update Action**
Invalid	Make new record on tail end
< Head Offset	Make async IO request on disk
< Read-only Offset	Make a mutable copy on tail end
< ∞	Update-in place



The HybridLog

It is safe to flush **read-only without pinning** records in the bufferpool!

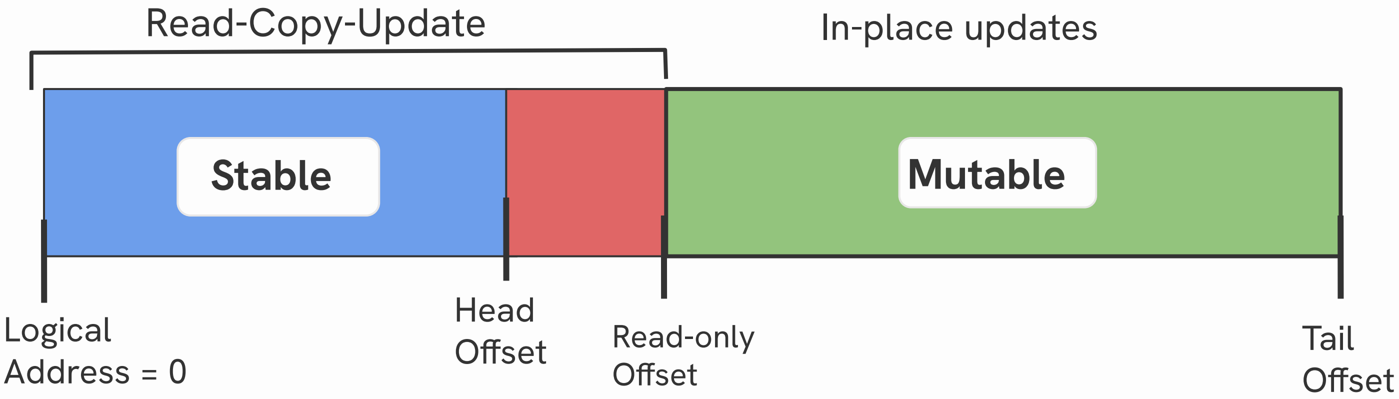
Logical Address	**Update Action**
Invalid	Make new record on tail end
< Head Offset	Make async IO request on disk
< Read-only Offset	Make a mutable copy on tail end
< ∞	Update-in place



The HybridLog

FOR THE CLASS:

How does this scheme mitigate the problems of the append-only log?



HybridLog Benefits

01

Cache for Hot Records

Frequently accessed records remain in-memory and are updated in place

02

Minimizes Disk I/O

Hot records in-memory do not require I/Os to disk

03

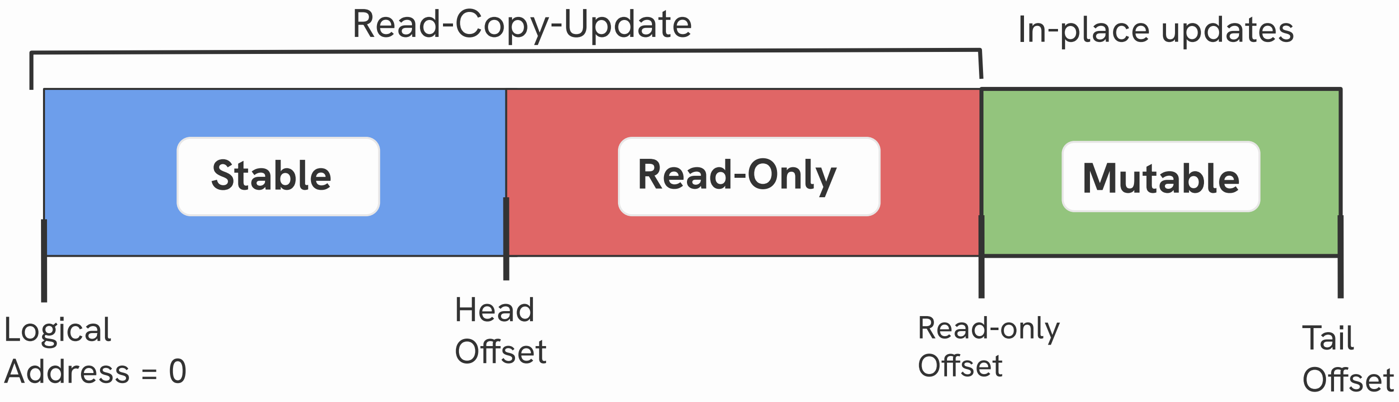
Fewer Hash Updates

The hottest records in mutable will not need to update the hash index

The HybridLog

FOR THE CLASS:

What other aspect of the target workload does this problem solve?

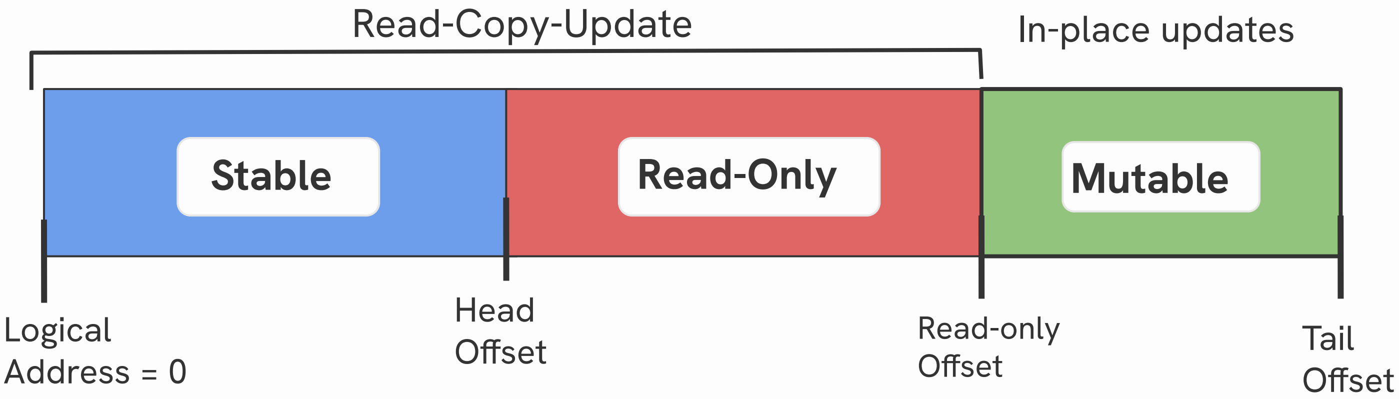


The HybridLog

Adaptable to changing hot/cold sets!

FOR THE CLASS:

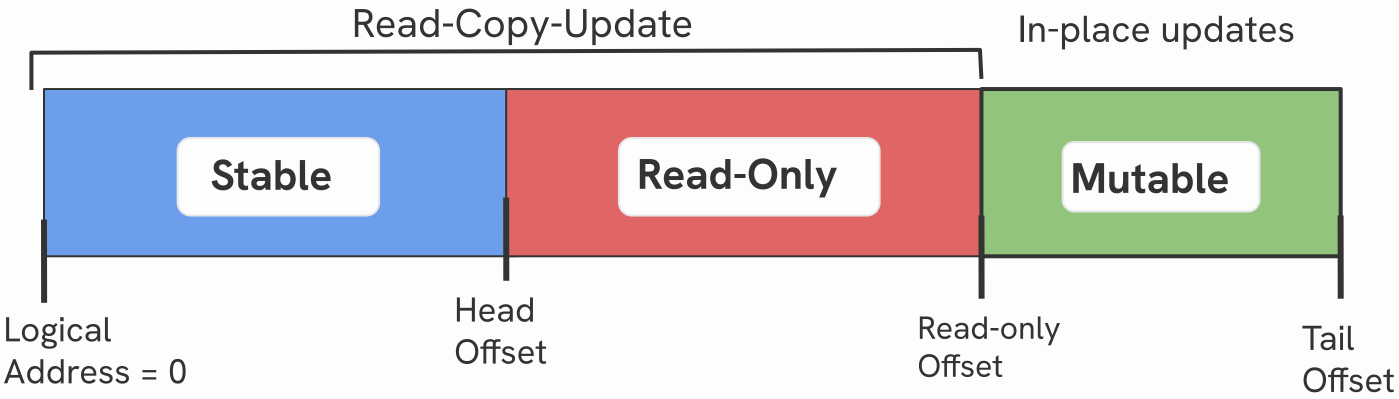
What other aspect of the target workload does this problem solve?



The HybridLog

What about **range-queries**?

Logical Address	**Update Action**
Invalid	Make new record on tail end
< Head Offset	Make async IO request on disk
< Read-only Offset	Make a mutable copy on tail end
< ∞	Update-in place

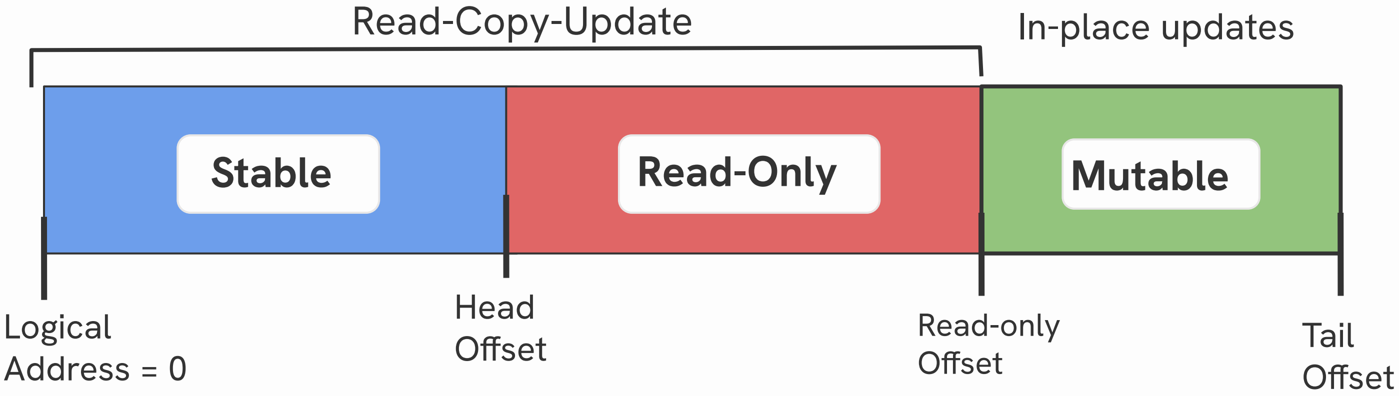


The HybridLog

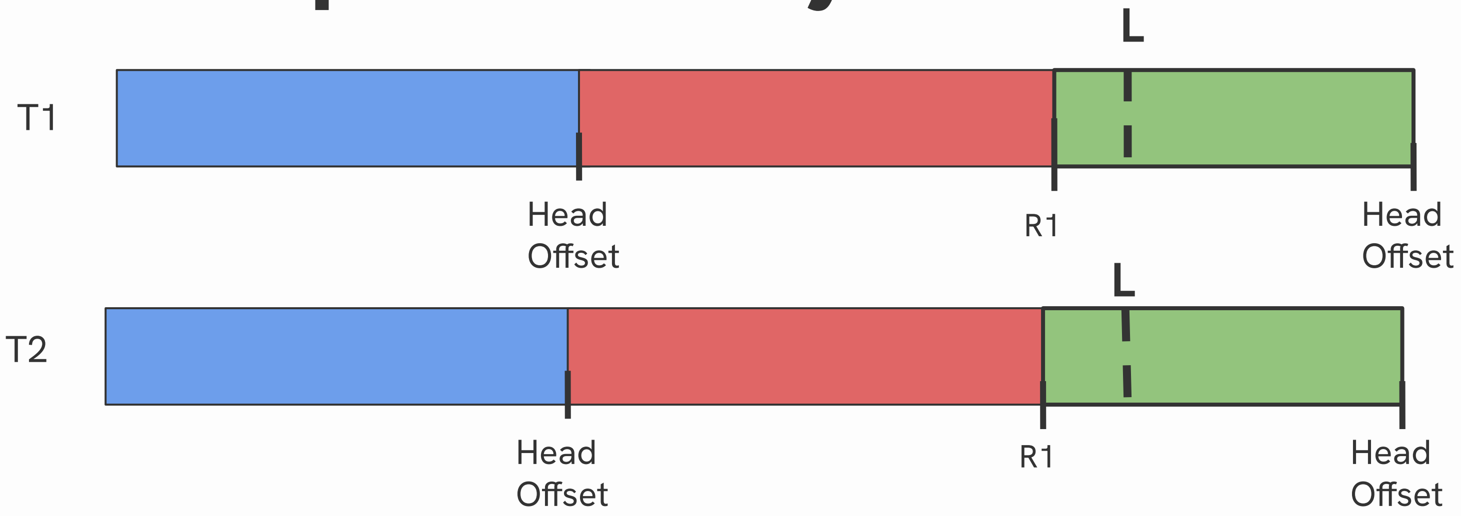
The read-only offset shifts with the tail offset...

FOR THE CLASS:

What problems can this cause with multiple threads?

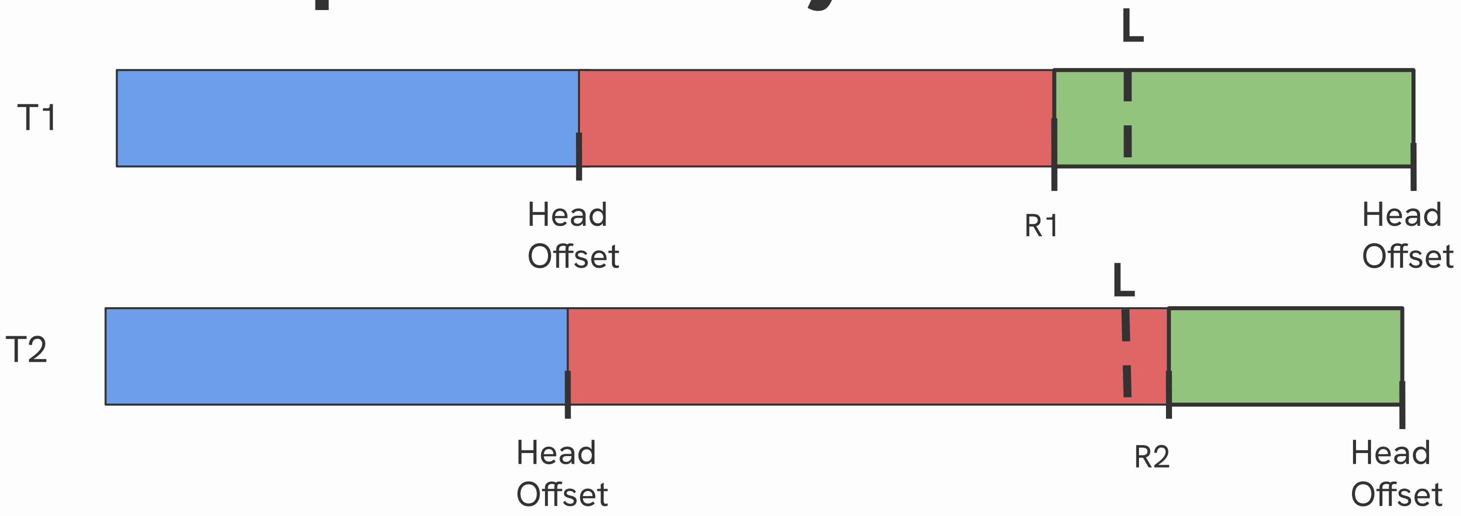


Lost-Update Anomaly



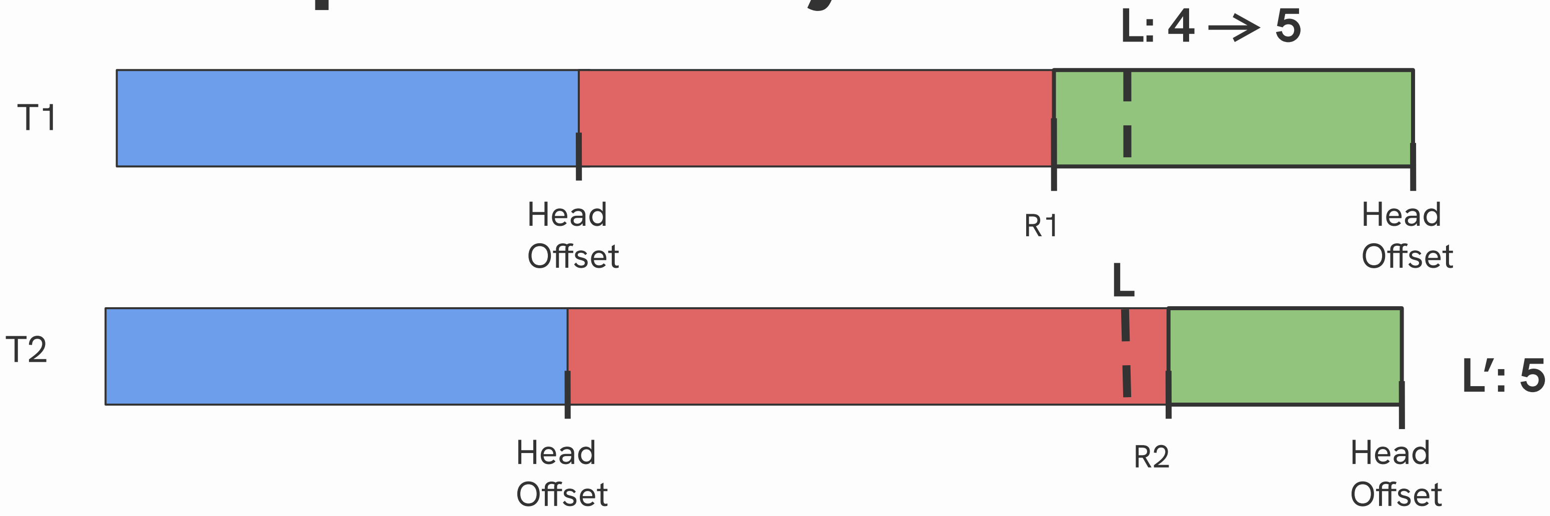
Both threads obtain address **L** | T1 determines **L > R1**

Lost-Update Anomaly



A **new thread** updates R1 →R2 | T2 determines **L < R2**

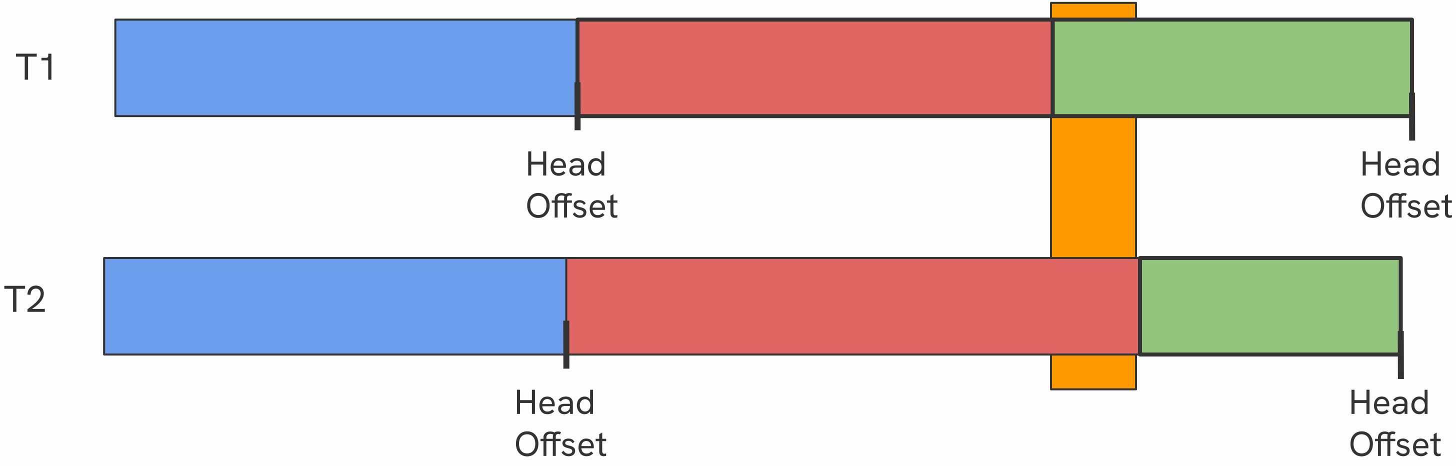
Lost-Update Anomaly



T1 **updates in place** | T2 **read-copy-writes**

Safe Read-Only: Using Epoch Protections

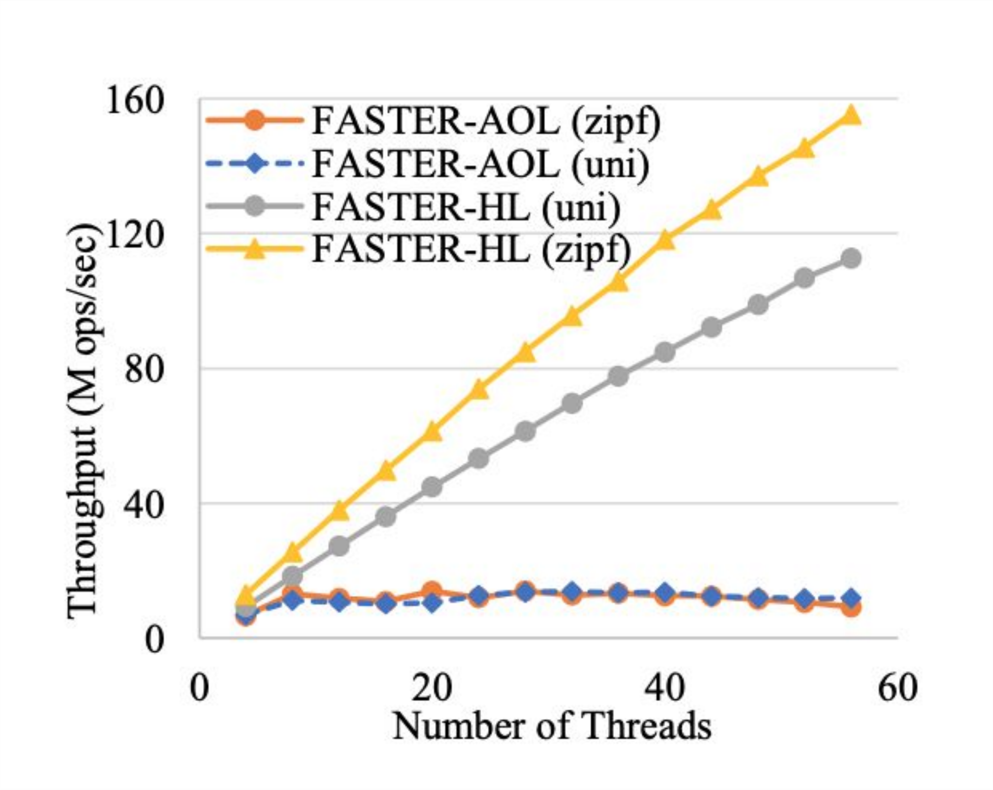
The Fuzzy Region



The space between the **safe read-only offset** and a threads **read-only offset**

Evaluation and Results

Proof of Hybrid vs Append-Only



Setup

Setup

Dell PowerEdge R730 machines, 2.60GHz Intel Xeon CPU 5E-2690 v4 CPUs

- 2 sockets, 14 cores per socket, 2 hyperthreads per core (56 total)
- 256GB RAM, 3.2TB FusionIO NVMe SSD

Workloads

Extended YCSB-A workload from Yahoo Cloud Serving Benchmark:

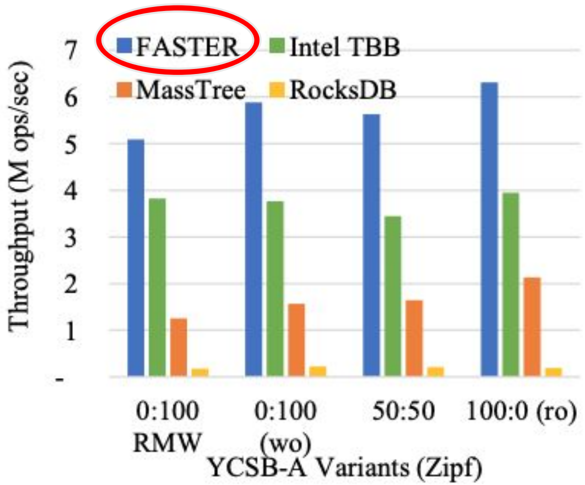
- 250 million distinct 8-byte keys, value sizes of 8 and 100 bytes
- R:BU and add RMW at 100%

Benchmarks

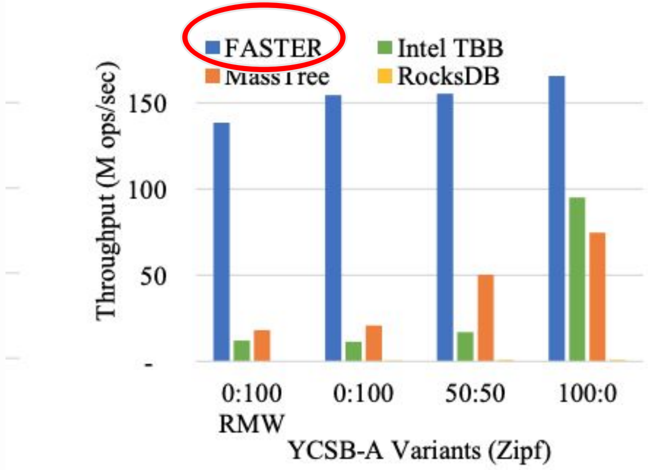
In-memory: Masstree, Intel TBB concurrent hashmap

Larger than memory: RocksDB

In-Memory: Single & Multi-Thread



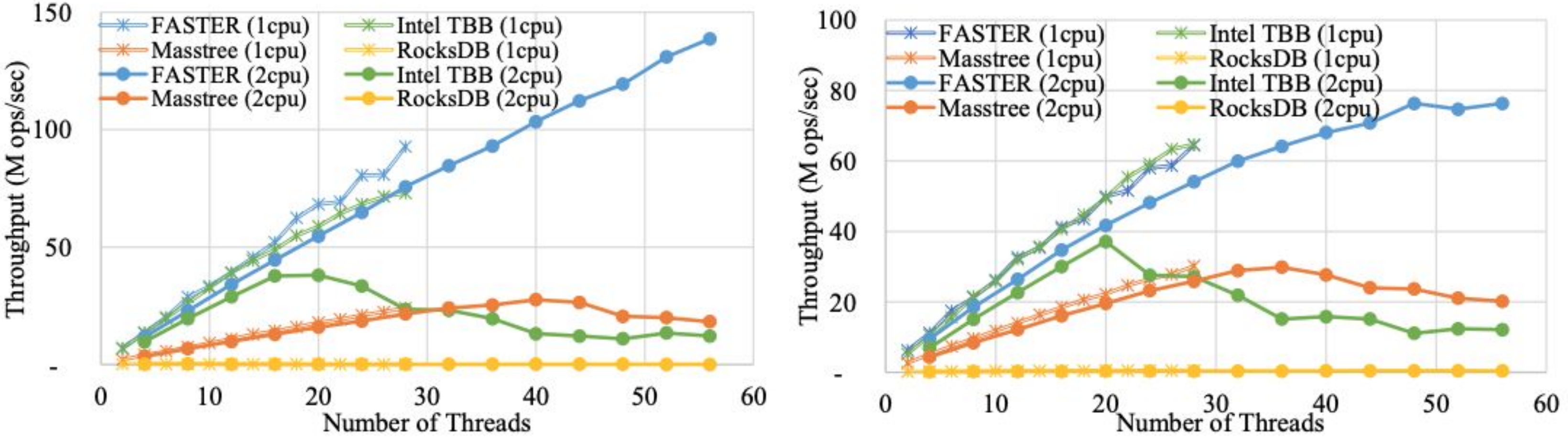
(b) Single thread; Zipf distr.



(d) All threads; Zipf distr.

Workload: 8-byte YCSB payload

In-Memory: Scalability

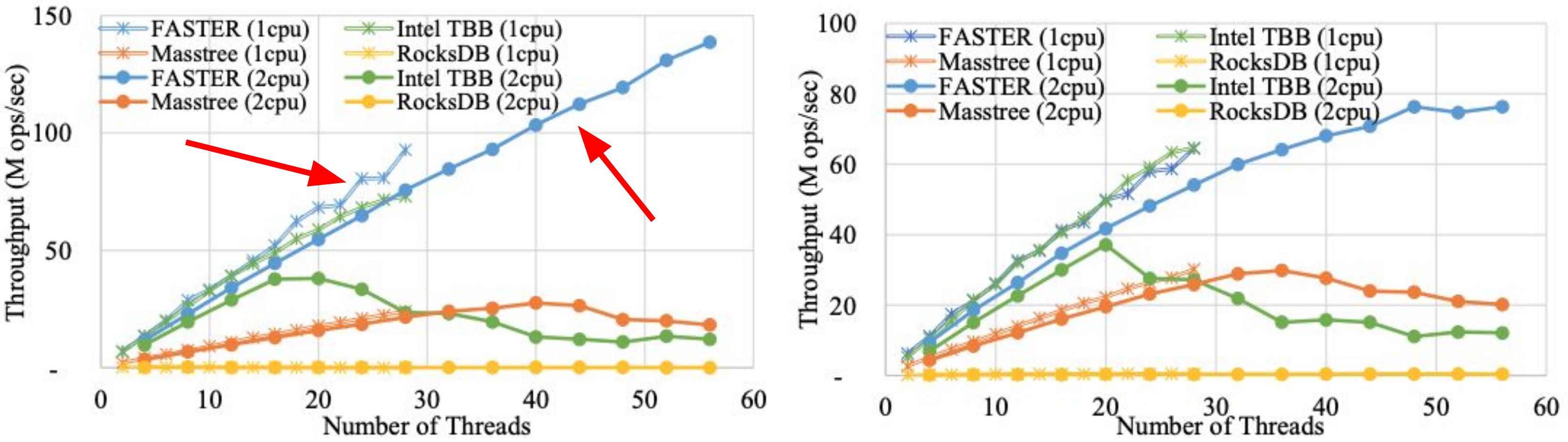


(a) RMW updates; 8-byte payloads. **(b) Blind updates; 100-byte payloads.**

Workload: 8-byte payload, 100% RMW

Workload: 100-byte payload, 0:100 blind upsert

In-Memory: Scalability



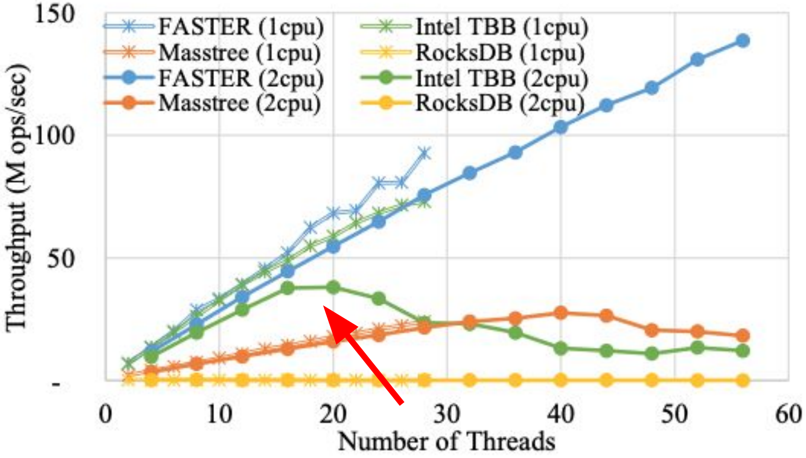
(a) RMW updates; 8-byte payloads.

(b) Blind updates; 100-byte payloads.

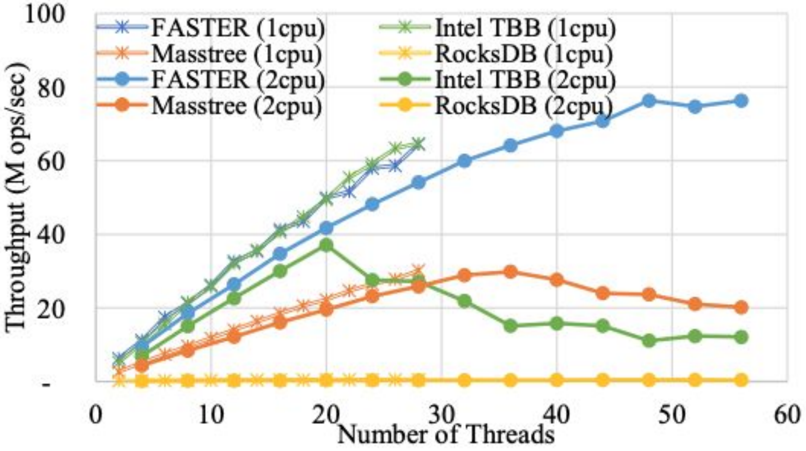
Workload: 8-byte payload, 100% RMW

Workload: 100-byte payload, 0:100 blind upsert

In-Memory: Scalability



(a) RMW updates; 8-byte payloads.

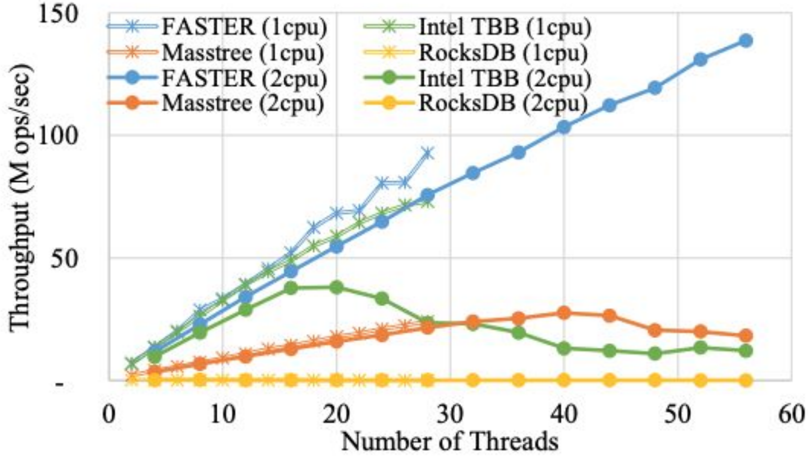


(b) Blind updates; 100-byte payloads.

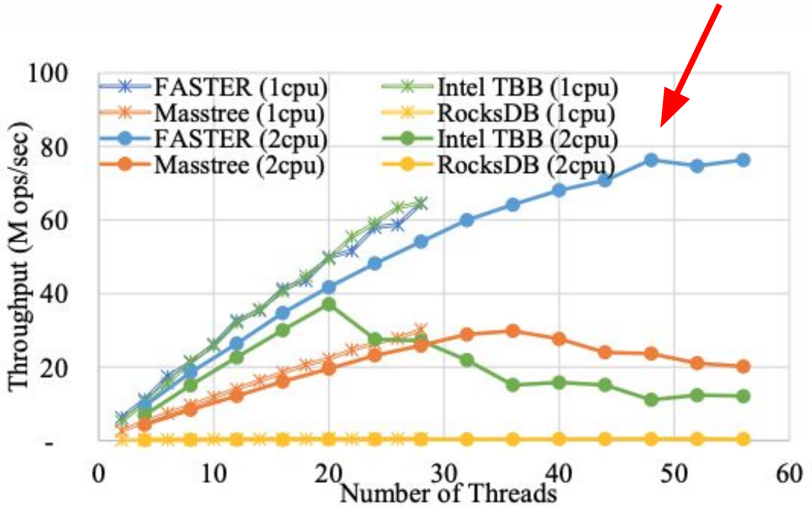
Workload: 8-byte payload, 100% RMW

Workload: 100-byte payload, 0:100 blind upsert

In-Memory: Scalability



(a) RMW updates; 8-byte payloads.

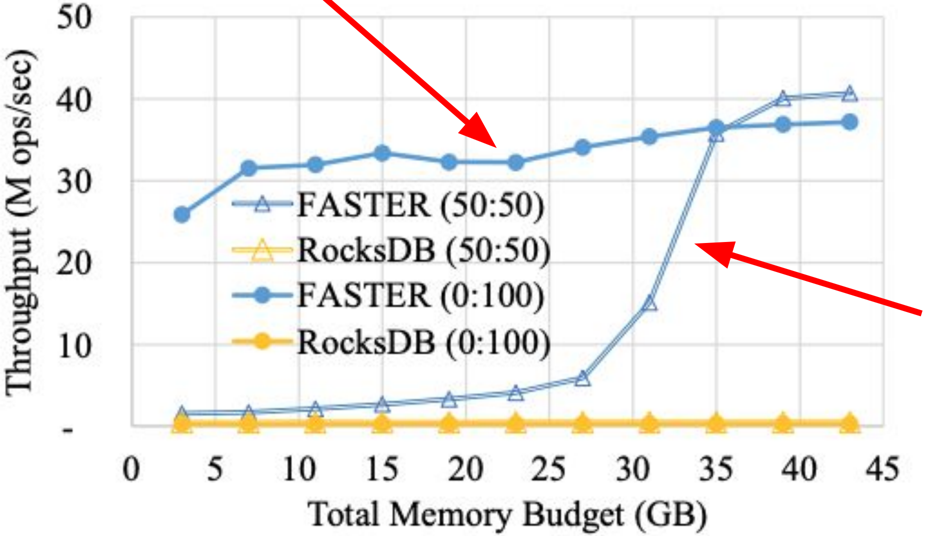


(b) Blind updates; 100-byte payloads.

Workload: 8-byte payload, 100% RMW

Workload: 100-byte payload, 0:100 blind upsert

Larger-Than Memory

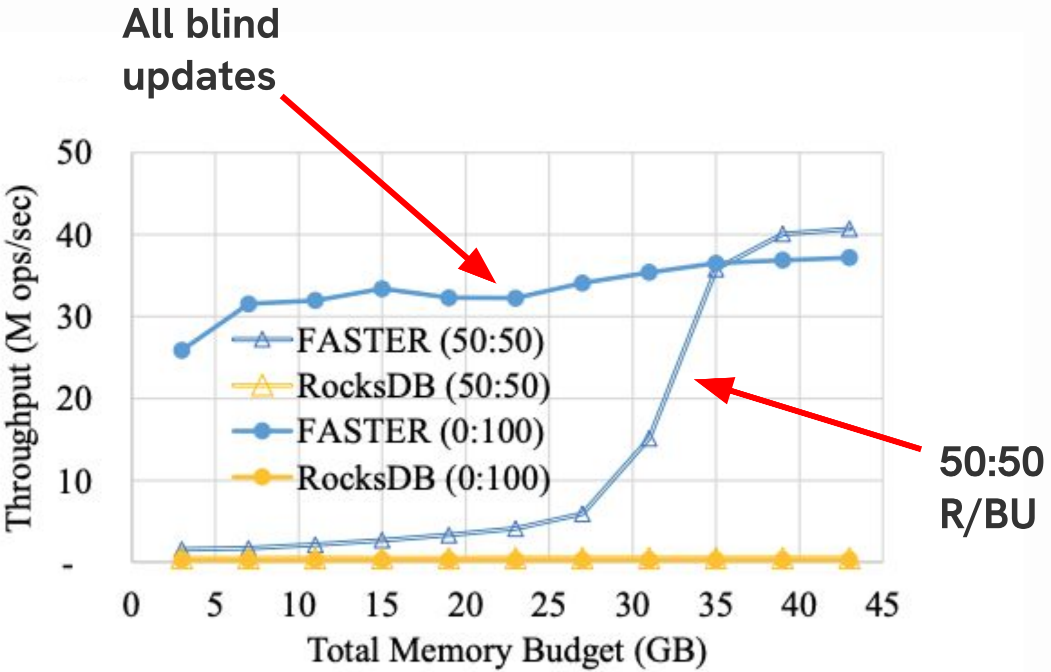


All blind updates

50:50 R/BU

Larger-Than Memory

Why does the 50:50 R:BU stall at lower memory budgets?



Conclusion

FASTER Supports:

In-Place Updates

Update-Intensive Workloads + Changing Hot Sets

Larger-Than-Memory

Latch-Free Concurrency

Future Work and Next Steps

Optimize I/O Path

Mitigate steep dropoff

Improve efficiency of random access

Apply to Other Systems

Extend to scan-based log analytic systems

Making more versatile

Optimize Recovery After Failure

Currently: eliminates need for WAL

Enhance monotonicity for consistent results

Our Thoughts

Alex

- FASTER seems to be great for write-intensive/update-intensive data, and I am interested by its capabilities for handling data with a lot of edge device traffic.
- However, because it is optimized for update-intensive workloads, I do think there is room for improvement for handling more diverse workloads (like more reads)..

Abbie

- I think FASTER does very well for what it is designed to do and is optimized for. I would like to see the future work on how to optimize the I/O path and for how to better handle reads.

Thank you!