FASTER

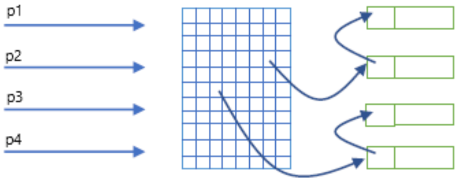
A Concurrent Key-Value Store with In-Place Updates

Requirements

Large State

High Update Intensity

Strong Temporal Locality

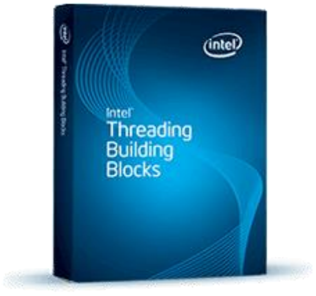


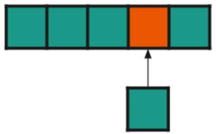
Optimized for Point Operations

Analytics Readiness

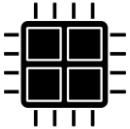
Distributed In-memory Storage Systems

Problems?

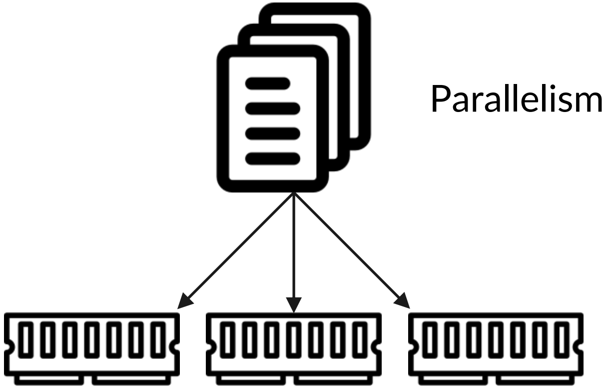




In-place updates

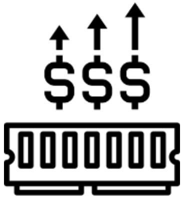


Concurrency



Parallelism

No Persistence

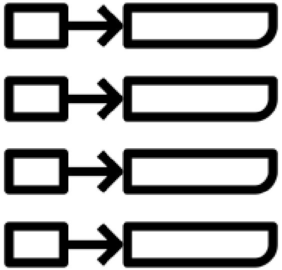


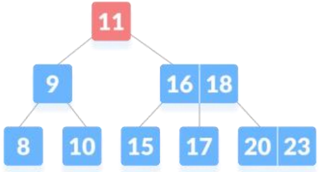
Cost

Inefficient Utilization

Existing Key Value Stores


Problems?






Blind Updates
Range Deletes
Range Scans


Point Queries
Read-Modify-Write


Persistence


Few million updates per second

Caching Systems

Problems?

redis

Memcached

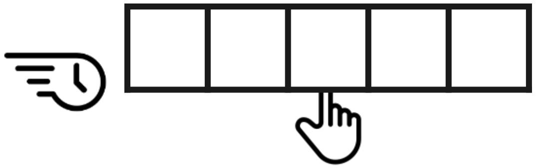
In memory

External system for persistence

Point Queries

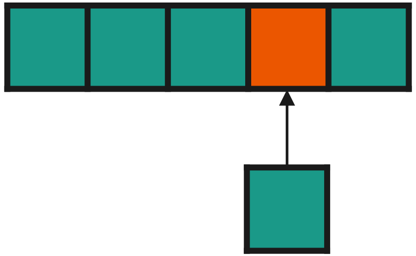
Increased Latency and Overhead

Design Philosophy



Faster point queries using
concurrent hash index

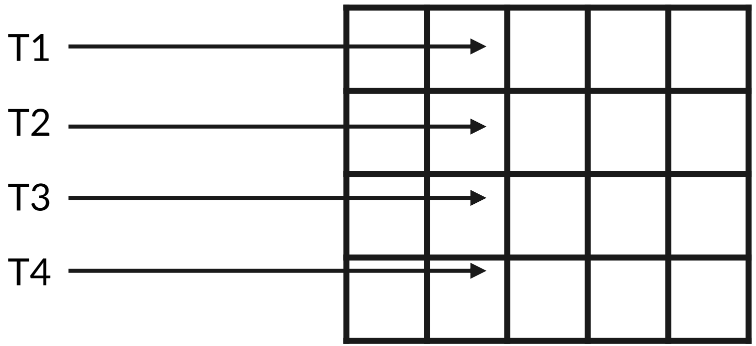
Choosing how and when to perform
expensive activities



In-place updates

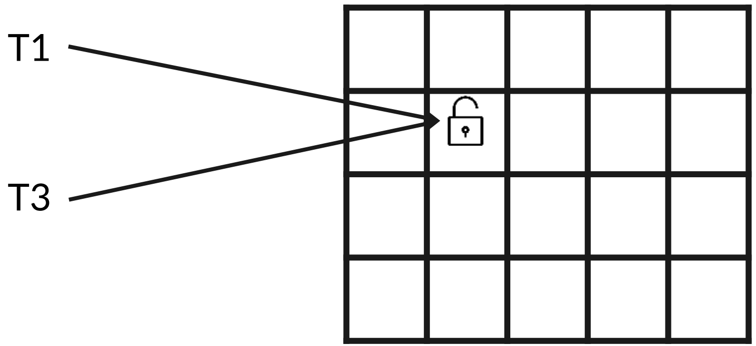
What is FASTER?

Faster is a concurrent latch-free key-value store that is designed for high performance and scalability across threads.



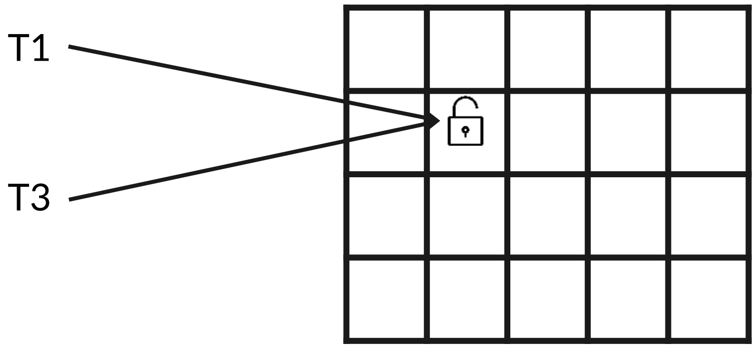
What is FASTER?

Faster is a concurrent latch-free key-value store that is designed for **high performance and scalability** across threads.

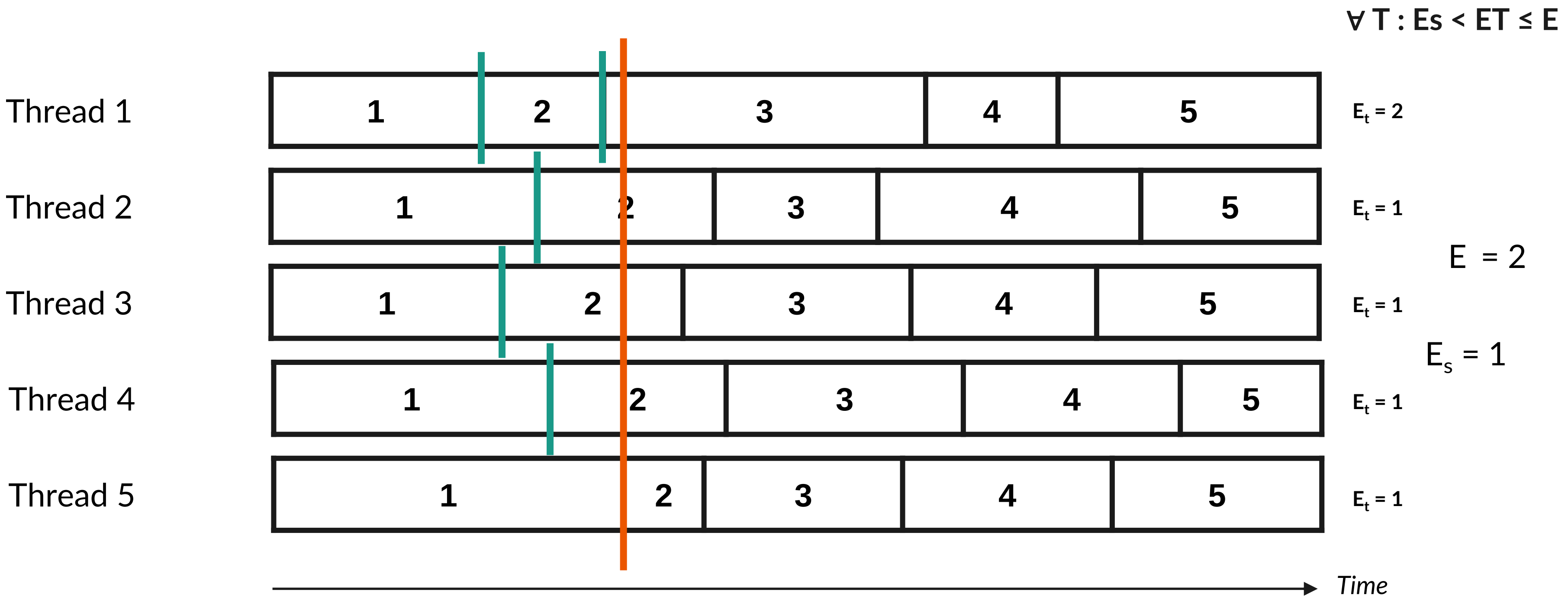


What is FASTER?

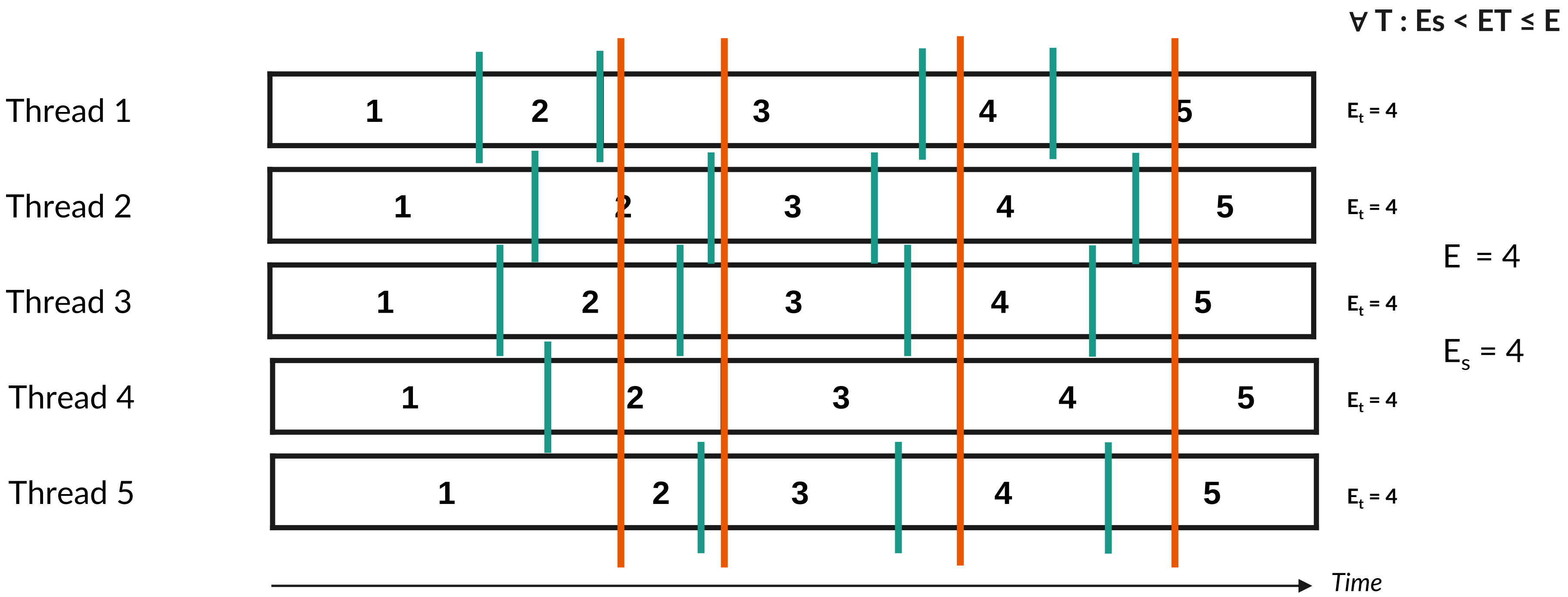
Faster is a concurrent latch-free key-value store that is designed for **high performance and scalability** across threads.



Lazy Synchronization: Epoch Protection

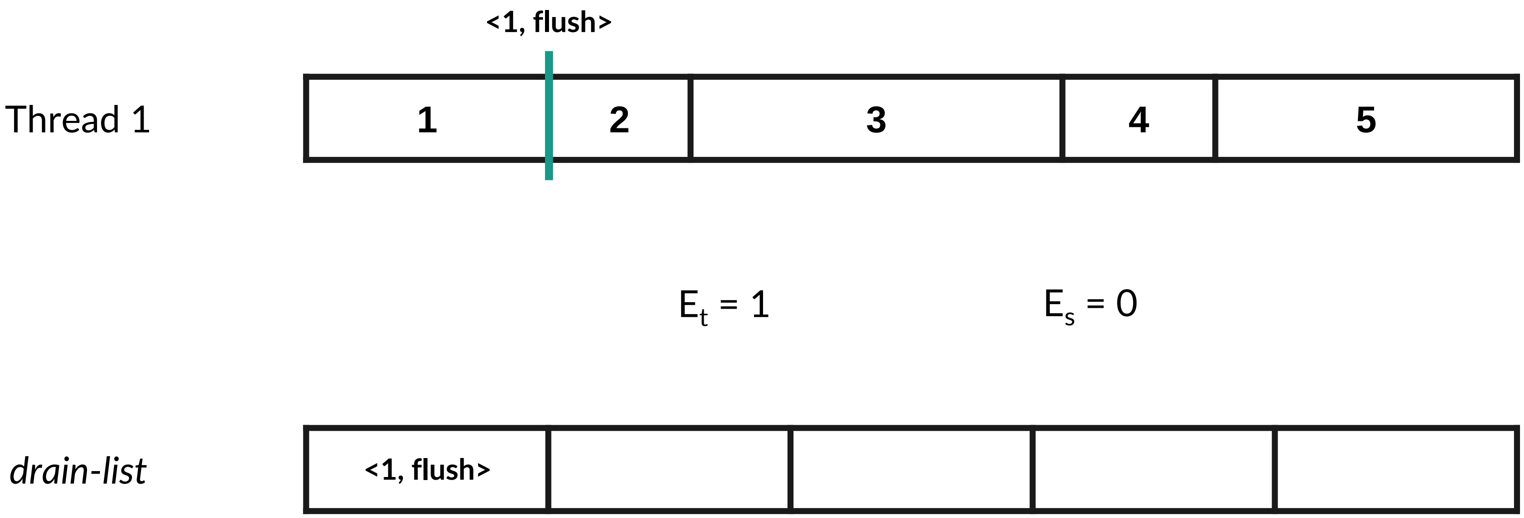


Lazy Synchronization: Epoch Protection



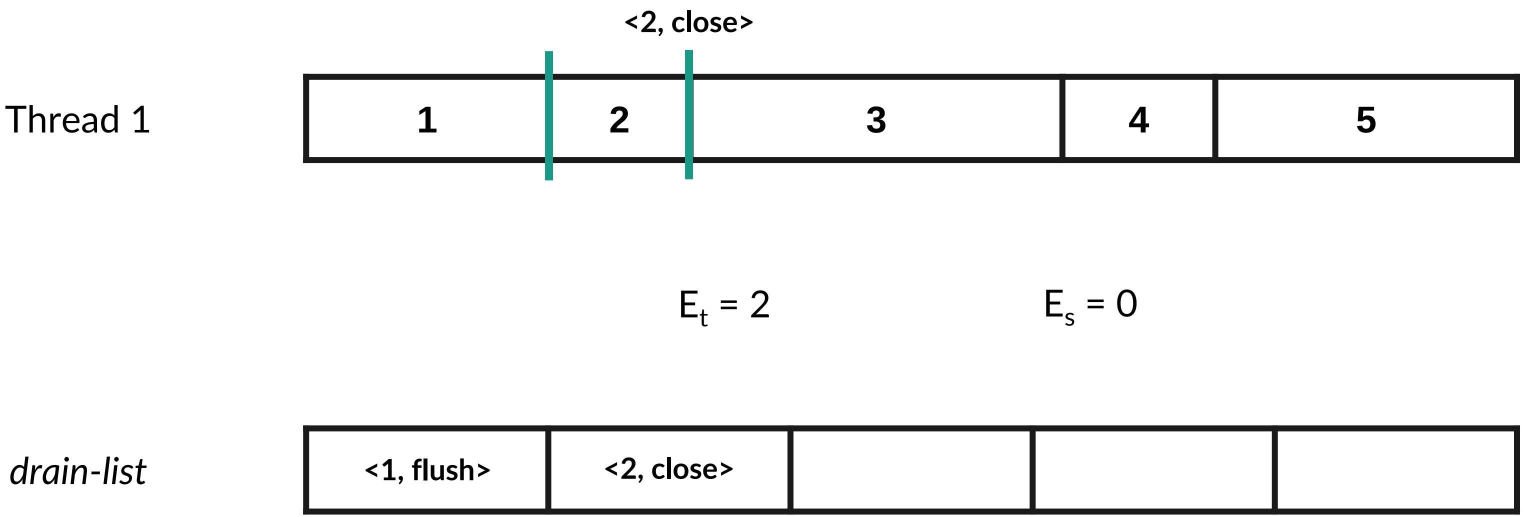
Lazy Synchronization: Trigger Actions

<epoch, action>

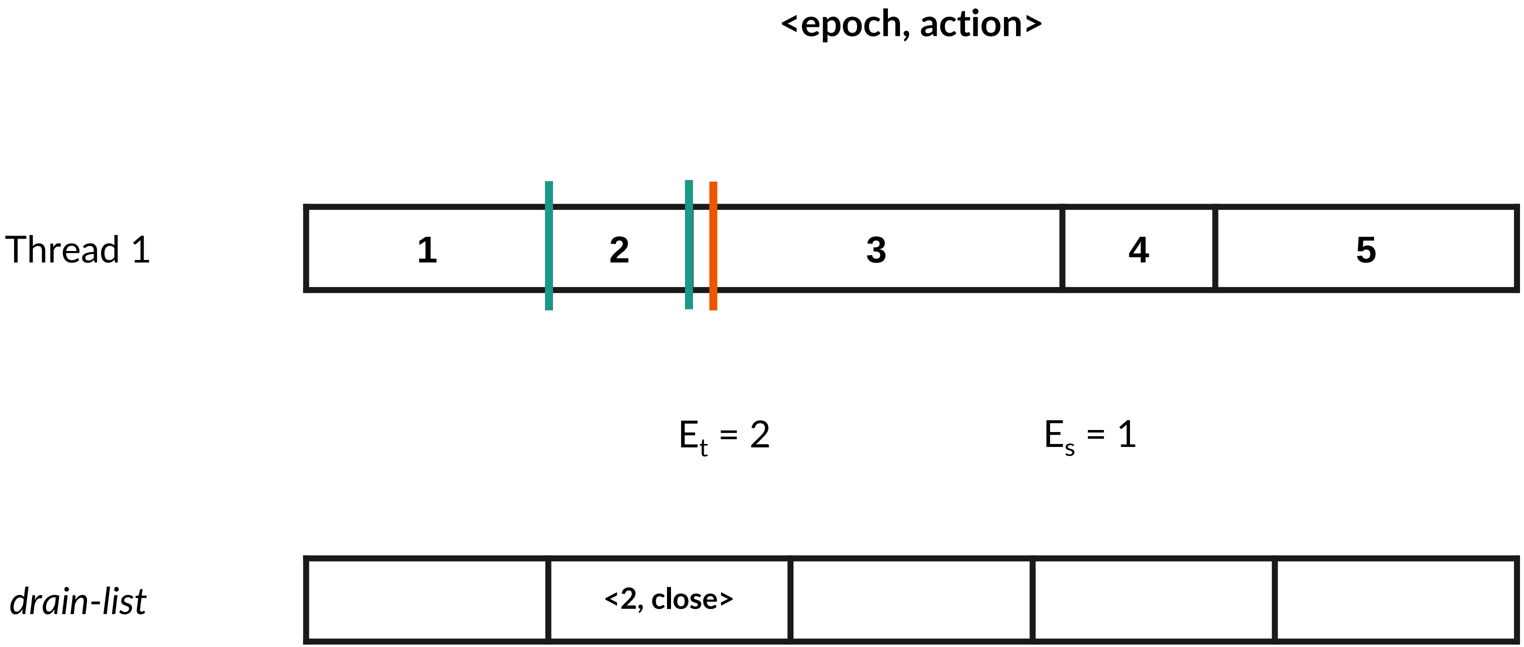


Lazy Synchronization: Trigger Actions

<epoch, action>

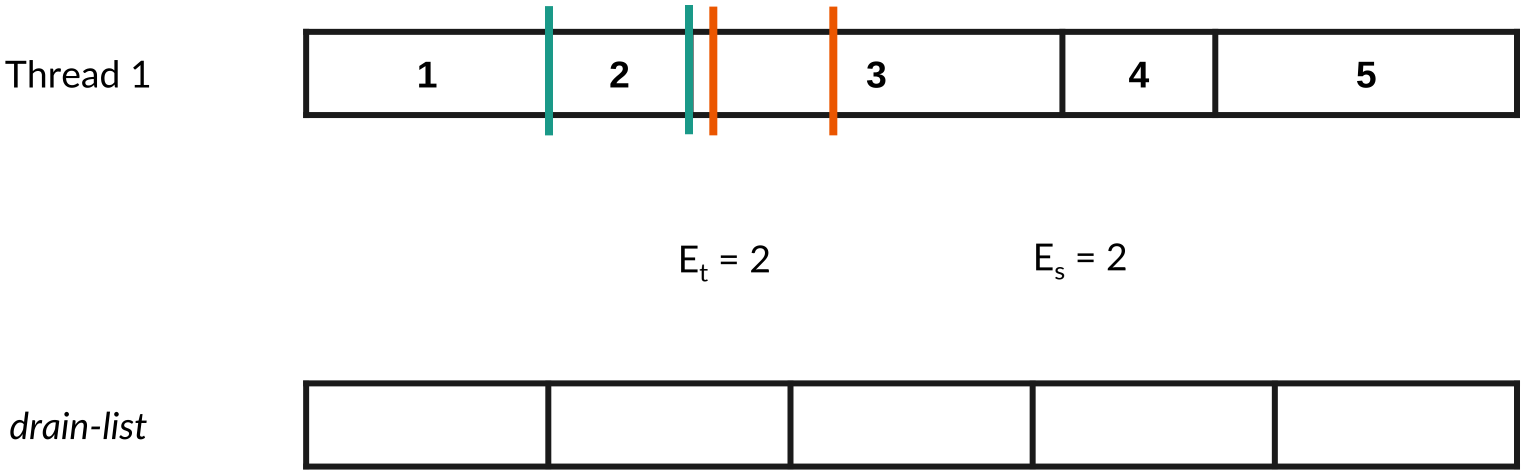


Lazy Synchronization: Trigger Actions

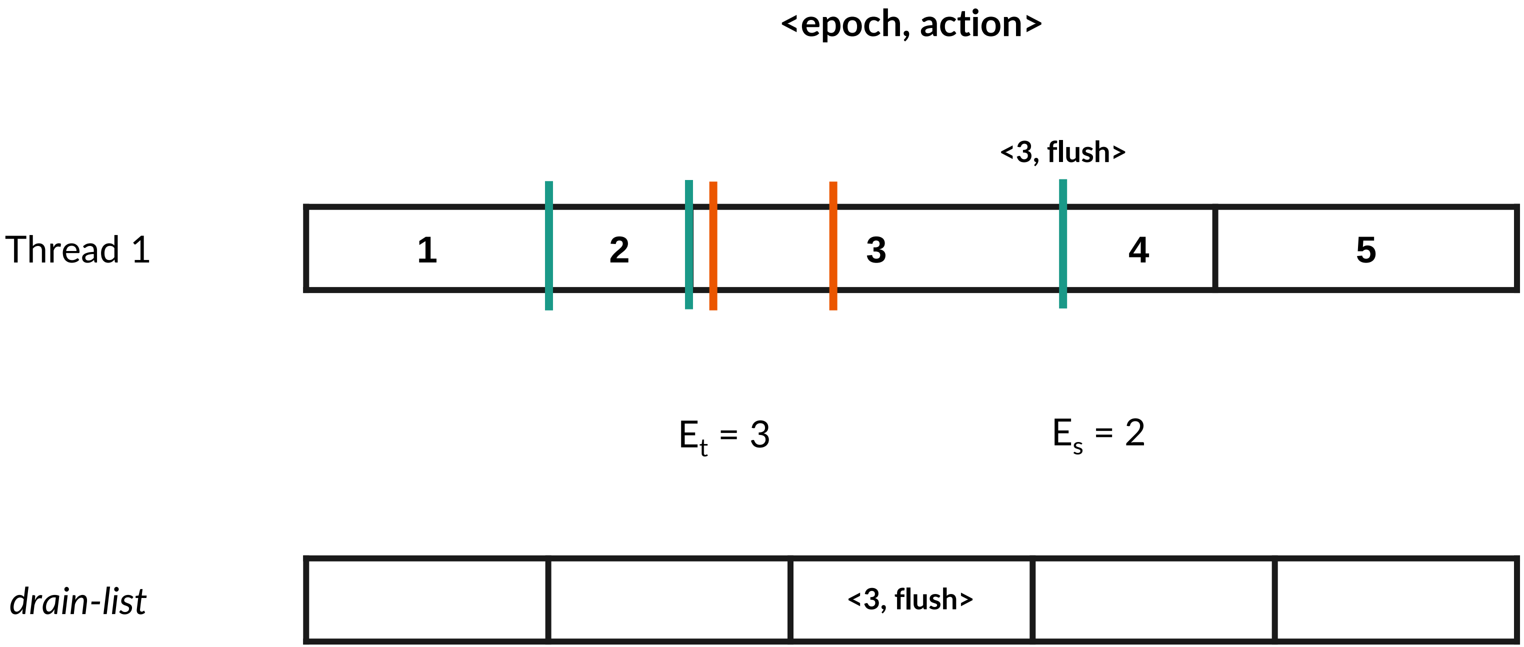


Lazy Synchronization: Trigger Actions

<epoch, action>

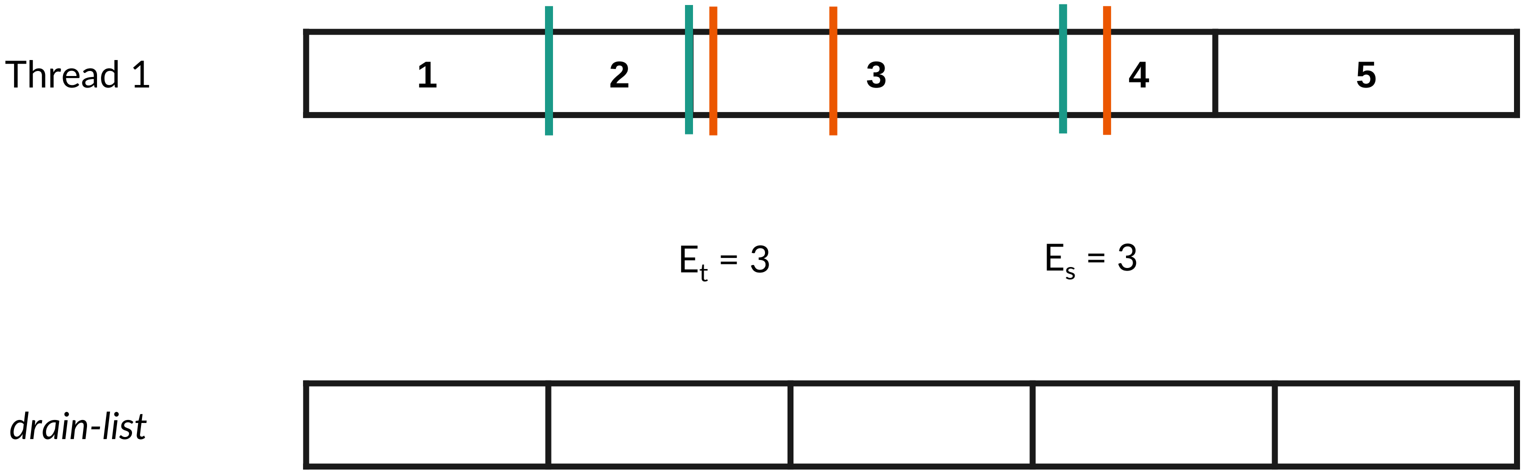


Lazy Synchronization: Trigger Actions



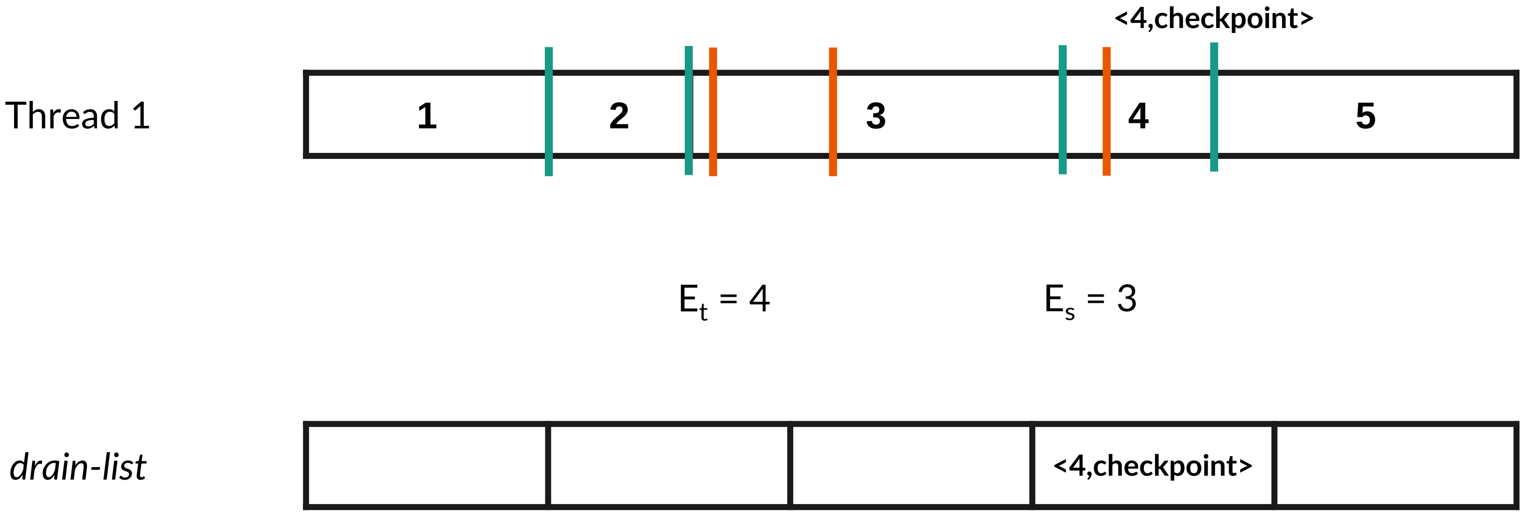
Lazy Synchronization: Trigger Actions

<epoch, action>



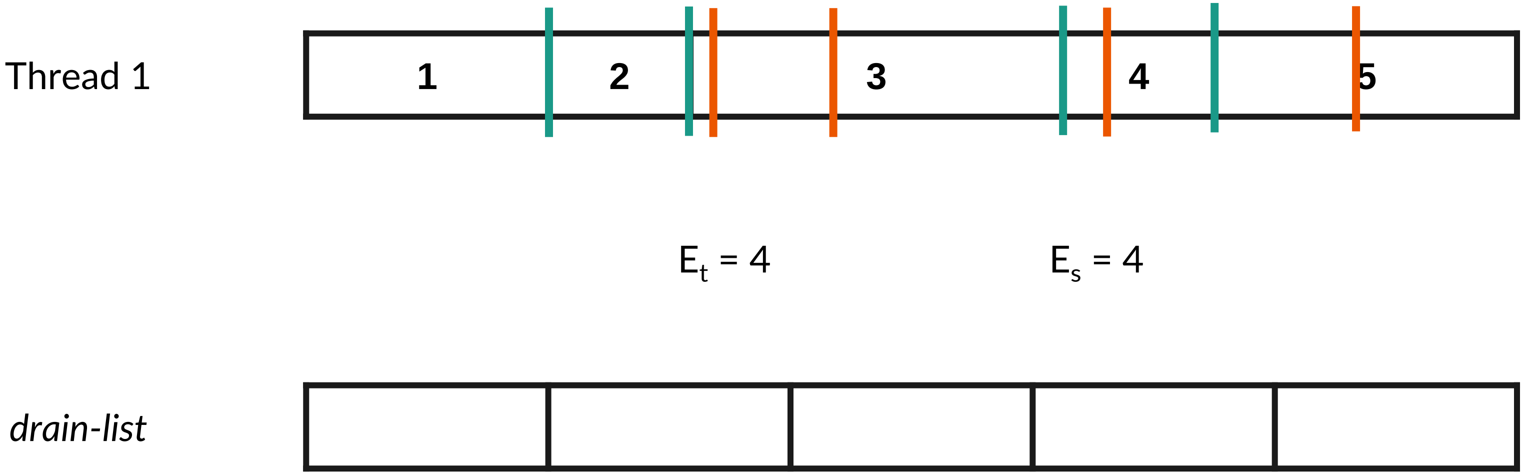
Lazy Synchronization: Trigger Actions

<epoch, action>

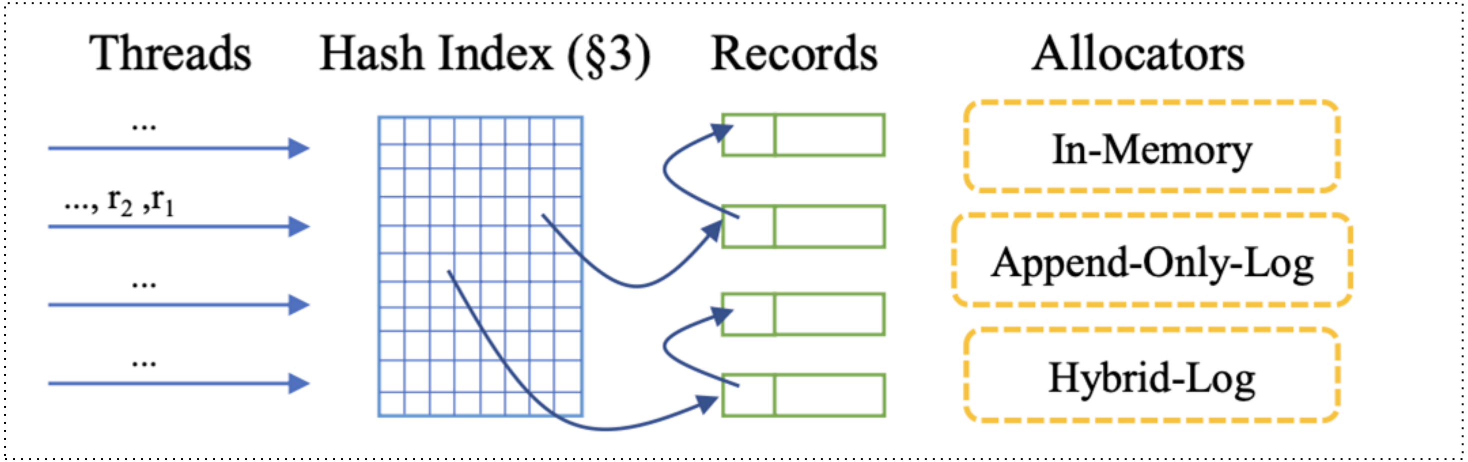


Lazy Synchronization: Trigger Actions

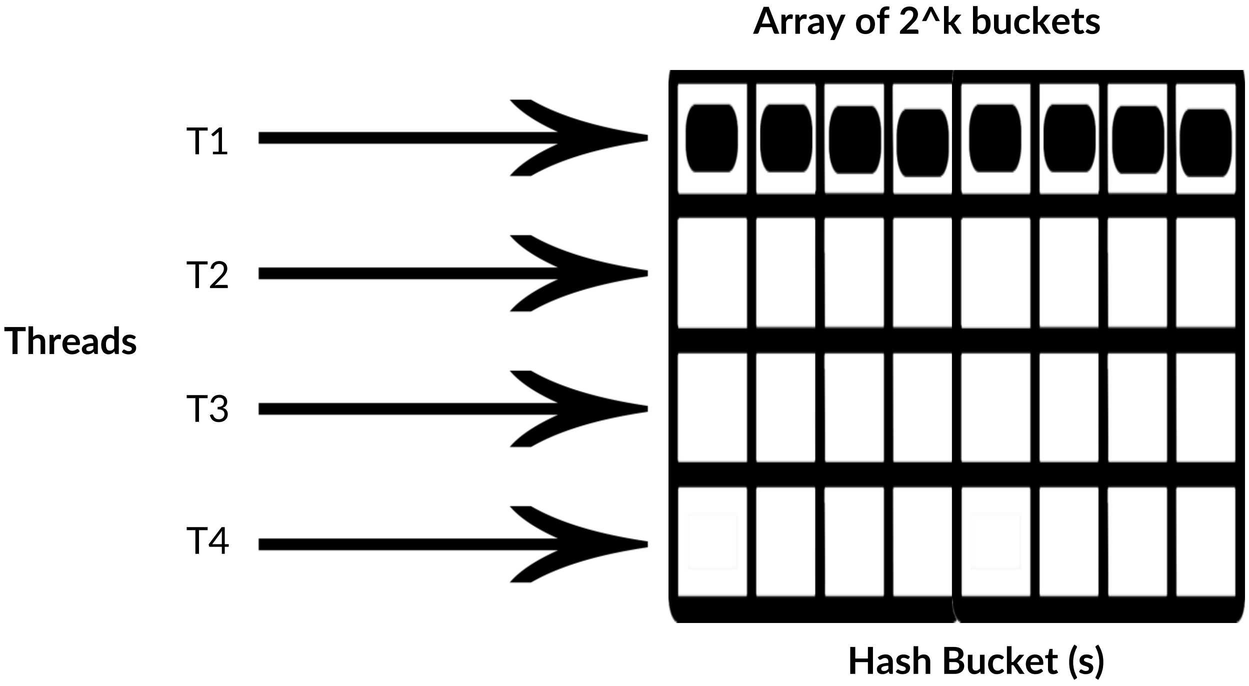
<epoch, action>



FASTER Architecture



Hash Index



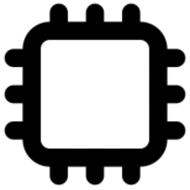
Latch Free

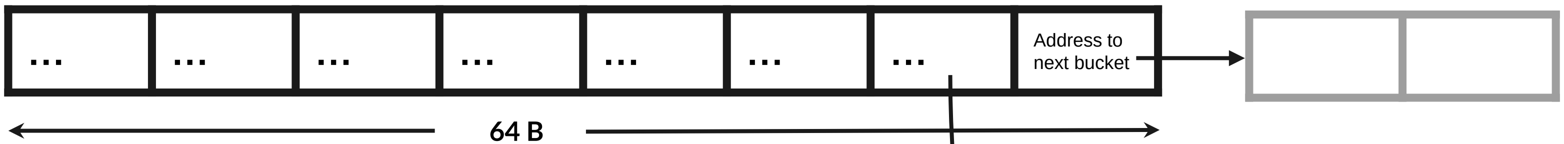
Concurrent

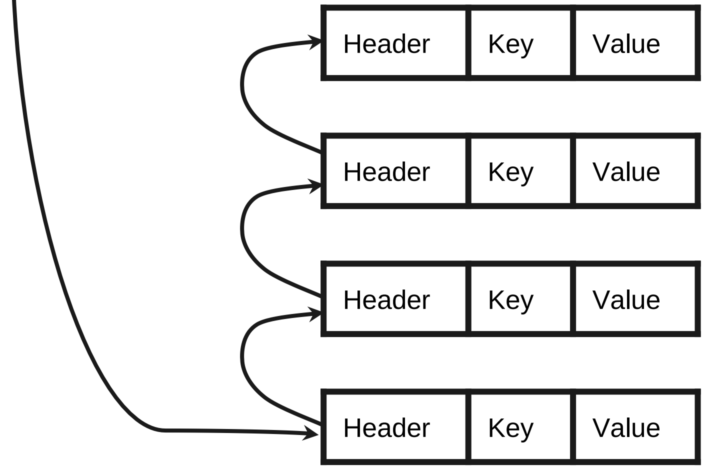
Scalable

Resizable

Hash Bucket





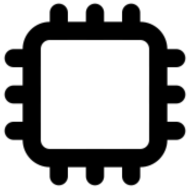


Each bucket consist of seven 8 B hash buckets *entries*

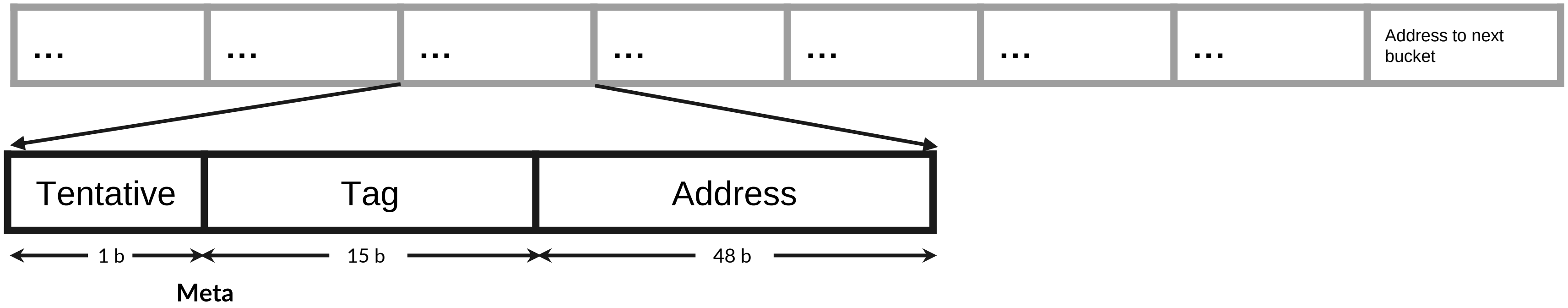
One 8 B entry to serve overflow bucket pointer

Why 8 byte entry

Hash Bucket Entry



48 bit



Entry value 0 (zero)→Empty slot

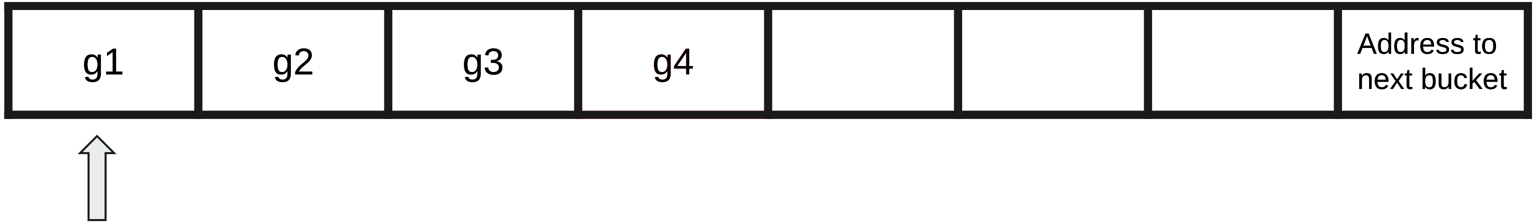
Tentative →Used by Latch Free Two-phase Algorithm

Tag→May increase hashing resolution

Address→Physical/Logical record address

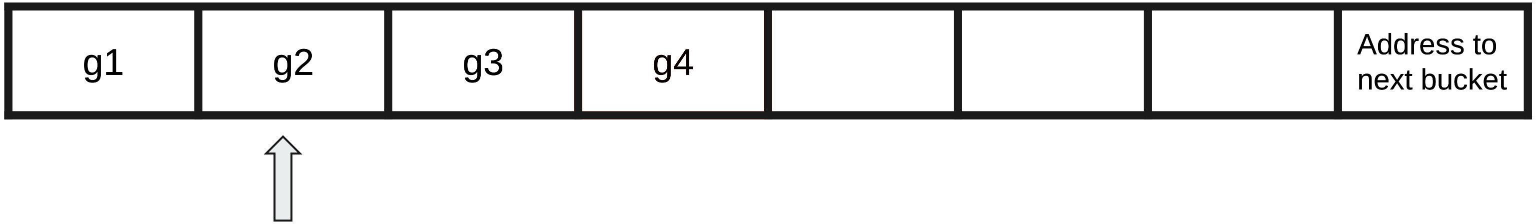
Why Keys are not part of Hash Index

Hash Index Operations (Find)



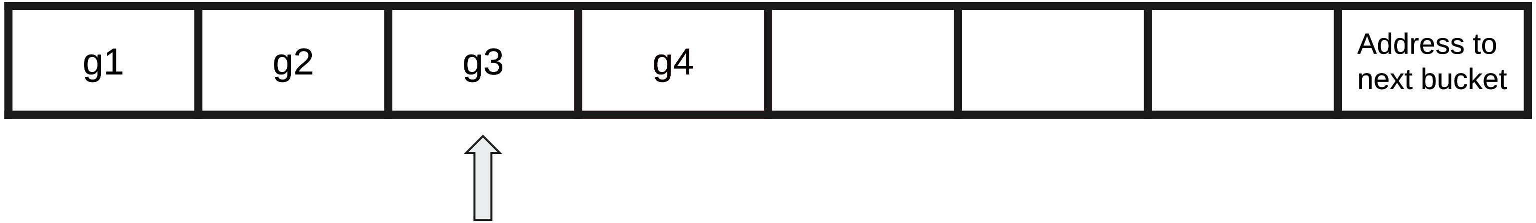
T1 find G4

Hash Index Operations (Find)



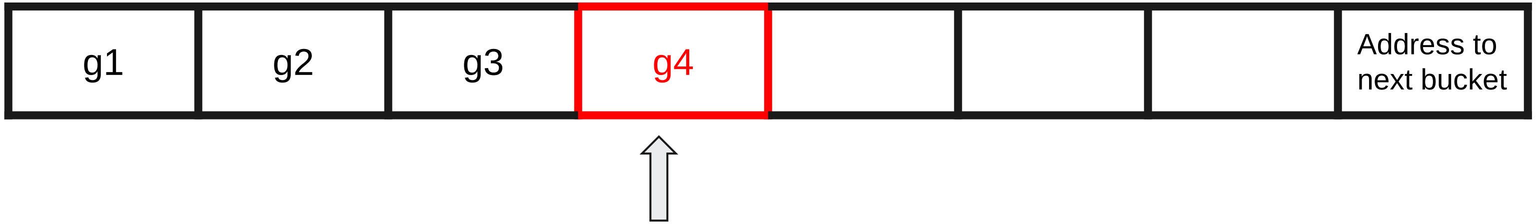
T1 find G4

Hash Index Operations (Find)



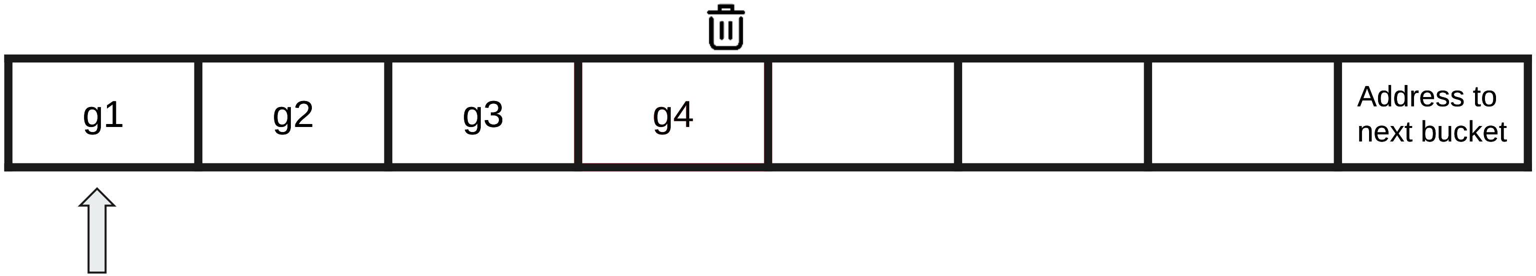
T1 find G4

Hash Index Operations (Find)



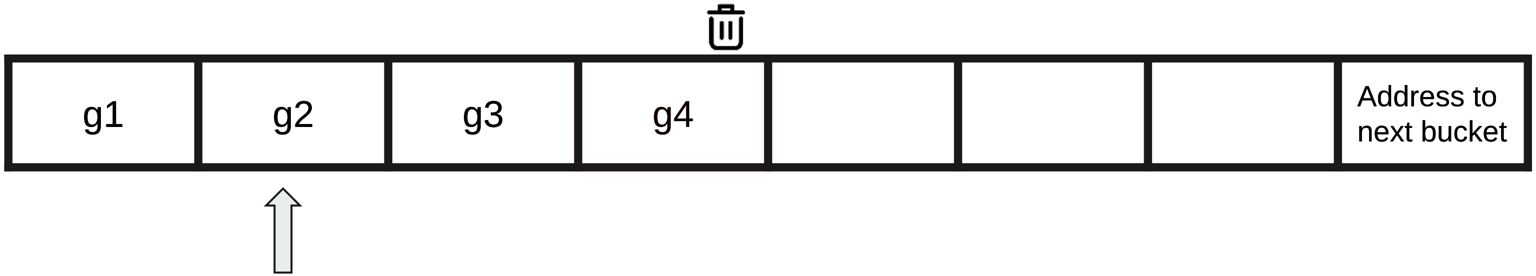
T1 find G4

Hash Index Operations (Delete)



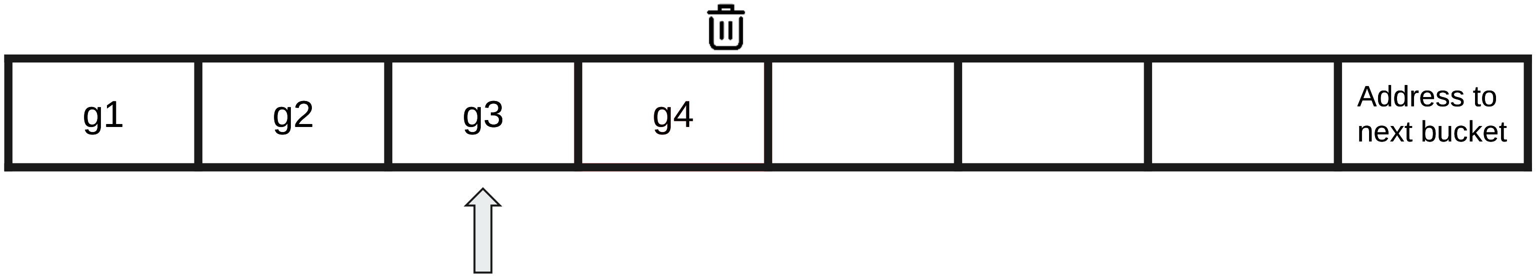
T2 delete G4

Hash Index Operations (Delete)



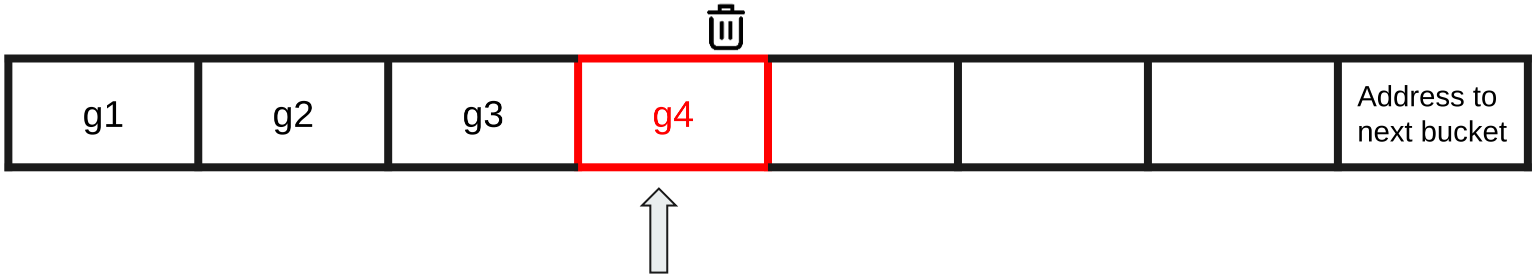
T2 delete G4

Hash Index Operations (Delete)



T2 delete G4

Hash Index Operations (Delete)



T2 delete G4

Hash Index Operations (Inserts)

Any Issue ?

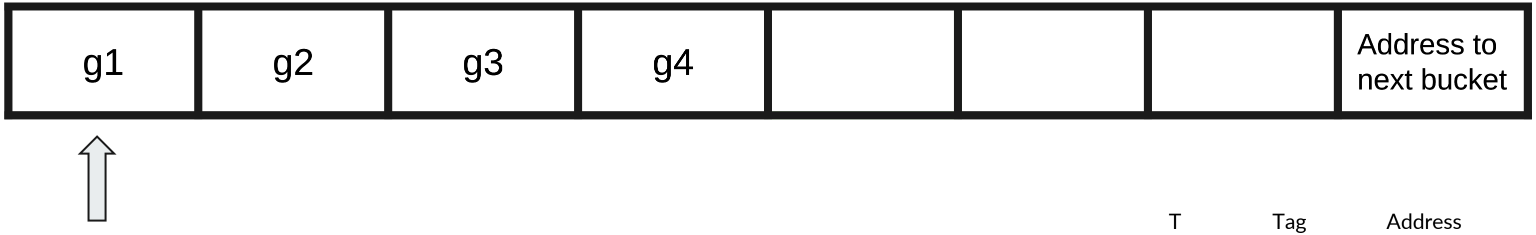
g1	g2	g3	g4				Address to next bucket

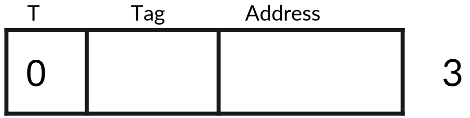
T	Tag	Address	
0			3

T1 insert G5
T2 delete G3 & insert G5

Hash Index Operations (Inserts)

Any Issue ?

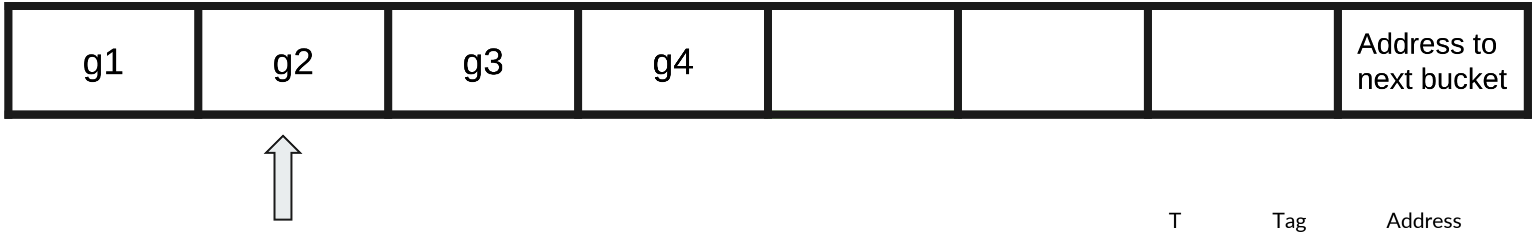




T1 insert G5
T2 delete G3 & insert G5

Hash Index Operations (Inserts)

Any Issue ?

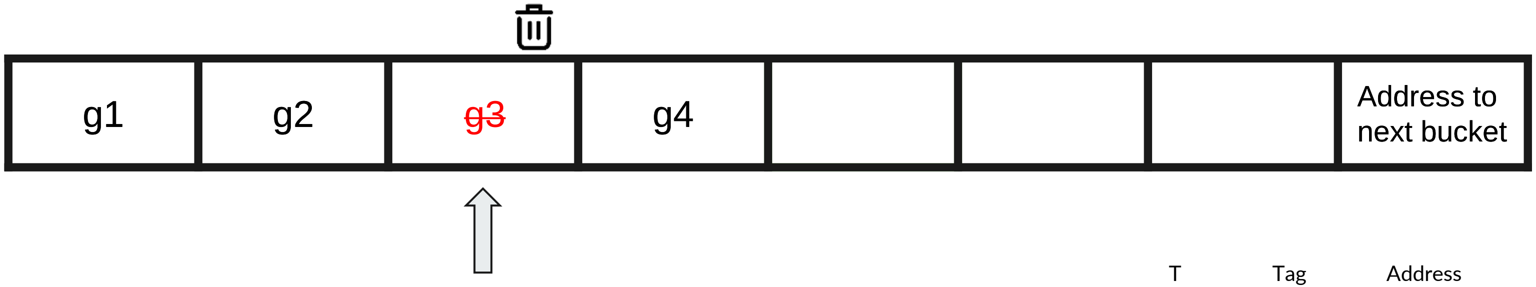


T	Tag	Address	
0			3

T1 insert G5
T2 delete G3 & insert G5

Hash Index Operations (Inserts)

Any Issue ?

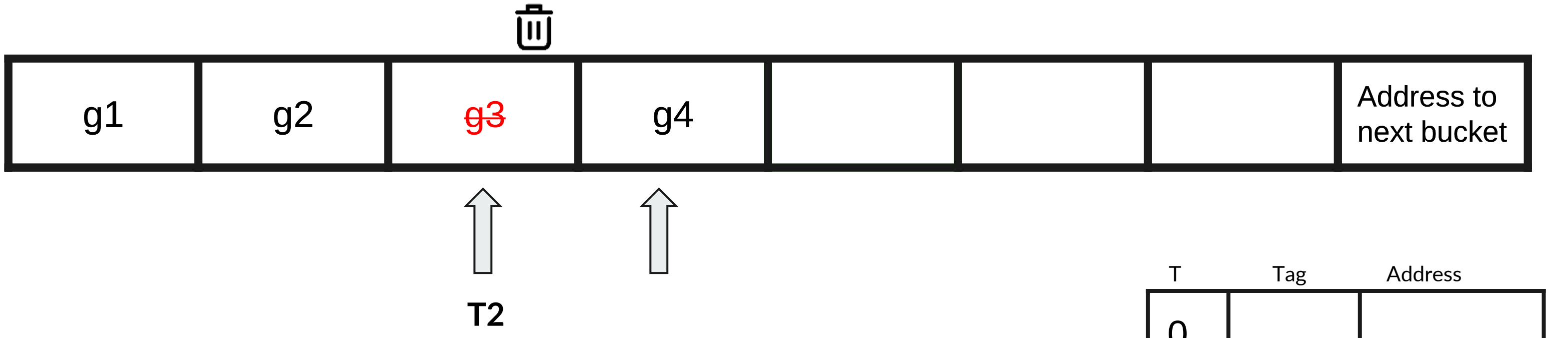


T2

T1 insert G5
T2 delete G3 & insert G5

Hash Index Operations (Inserts)

Any Issue ?



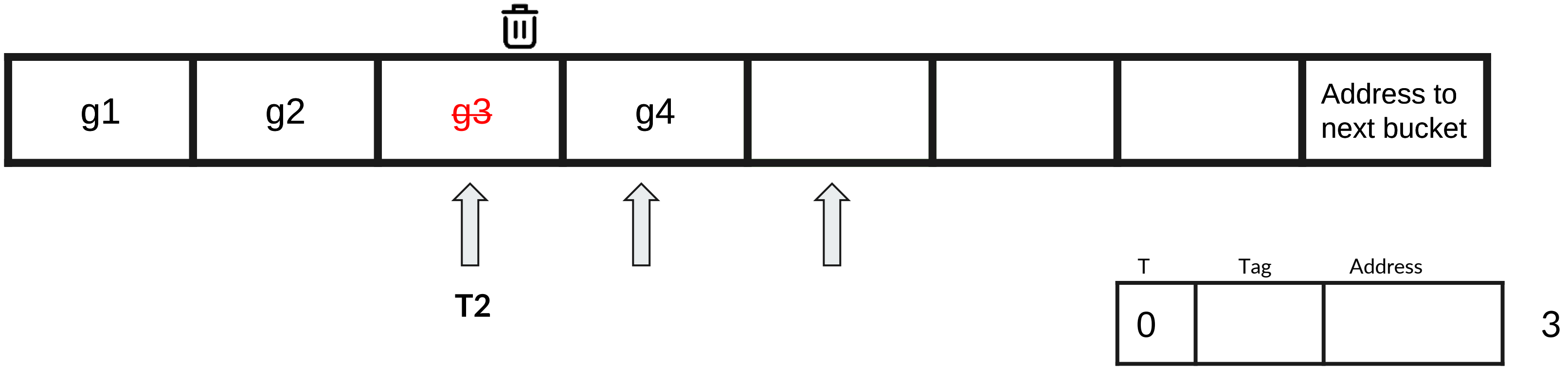
T	Tag	Address
0		

3

T1 insert G5
T2 delete G3 & insert G5

Hash Index Operations (Inserts)

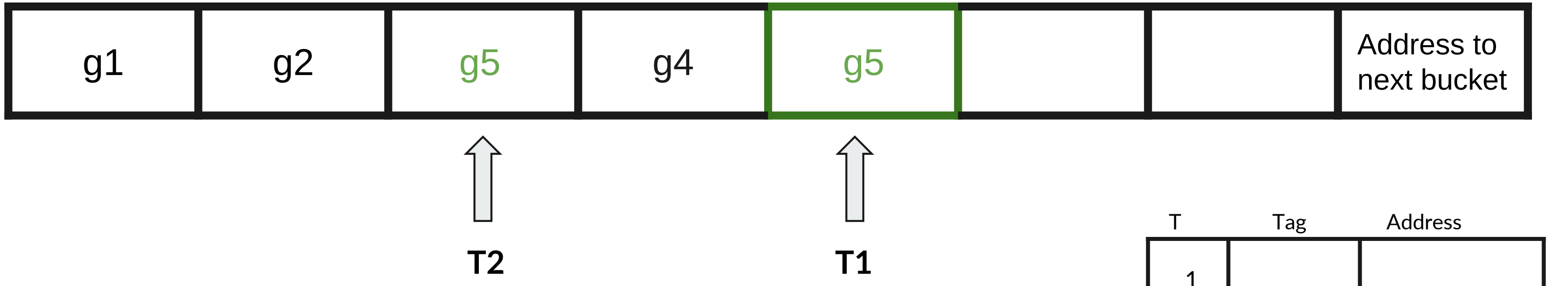
Any Issue ?



T1 insert G5
T2 delete G3 & insert G5

Hash Index Operations (Inserts)

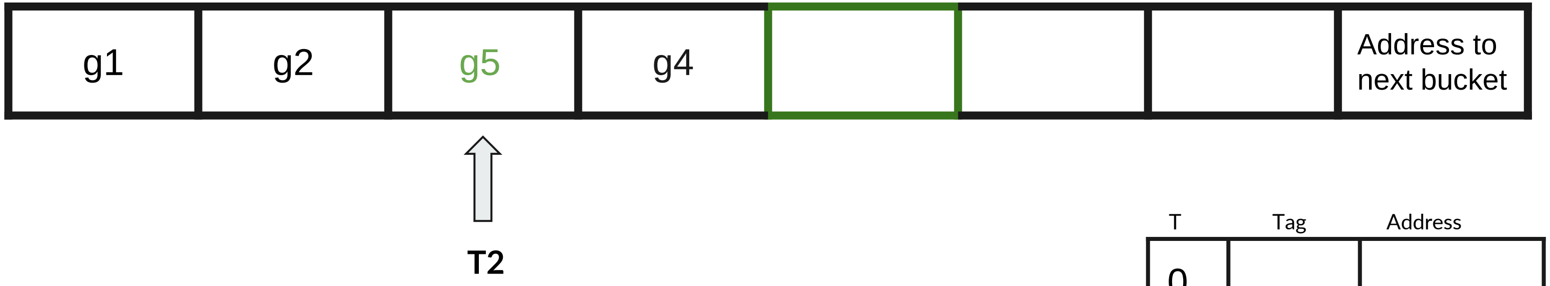
Any Issue ?



T1 insert G5
T2 delete G3 & insert G5

Hash Index Operations (Inserts)

Any Issue ?

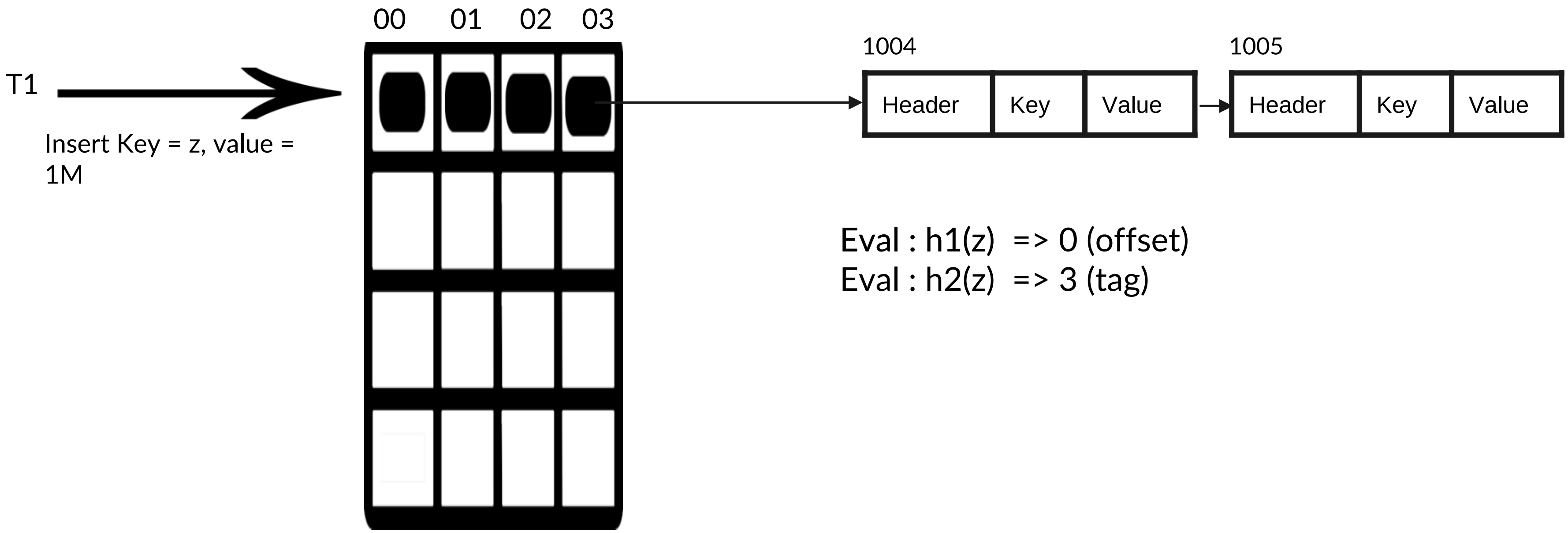


T	Tag	Address	
0			3
0			5

T1 insert G5
T2 delete G3 & insert G5

In-Memory Key-Value Store

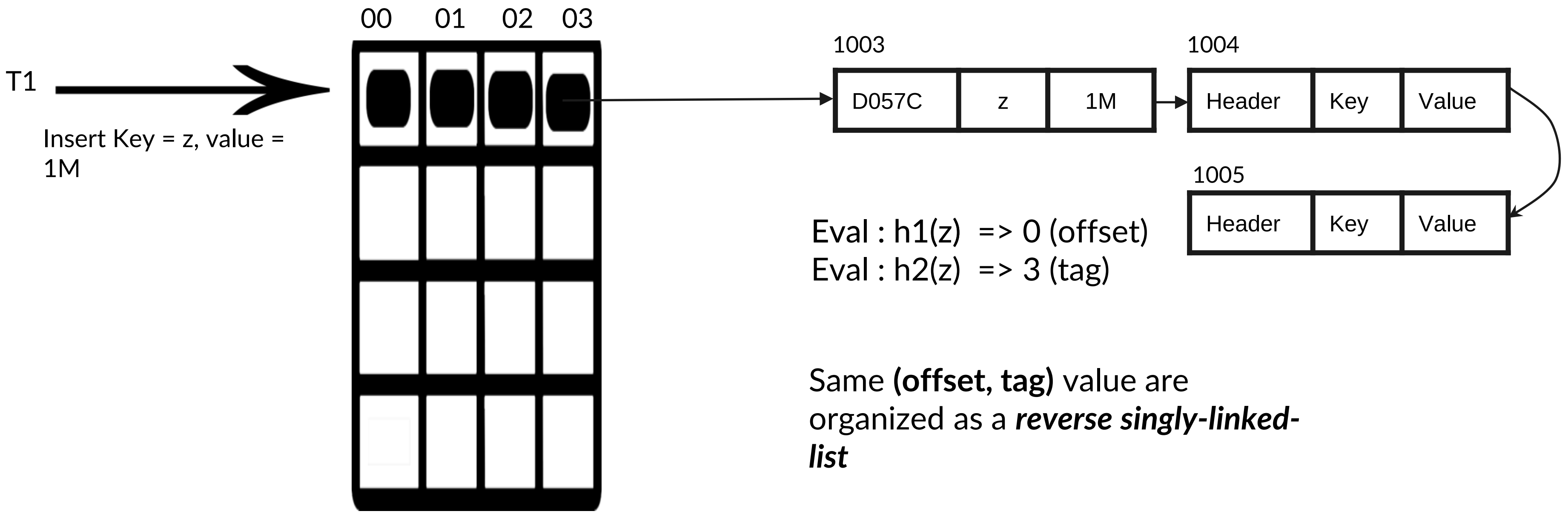
In-Memory



Eval : h1(z) => 0 (offset)
Eval : h2(z) => 3 (tag)

In-Memory Key-Value Store

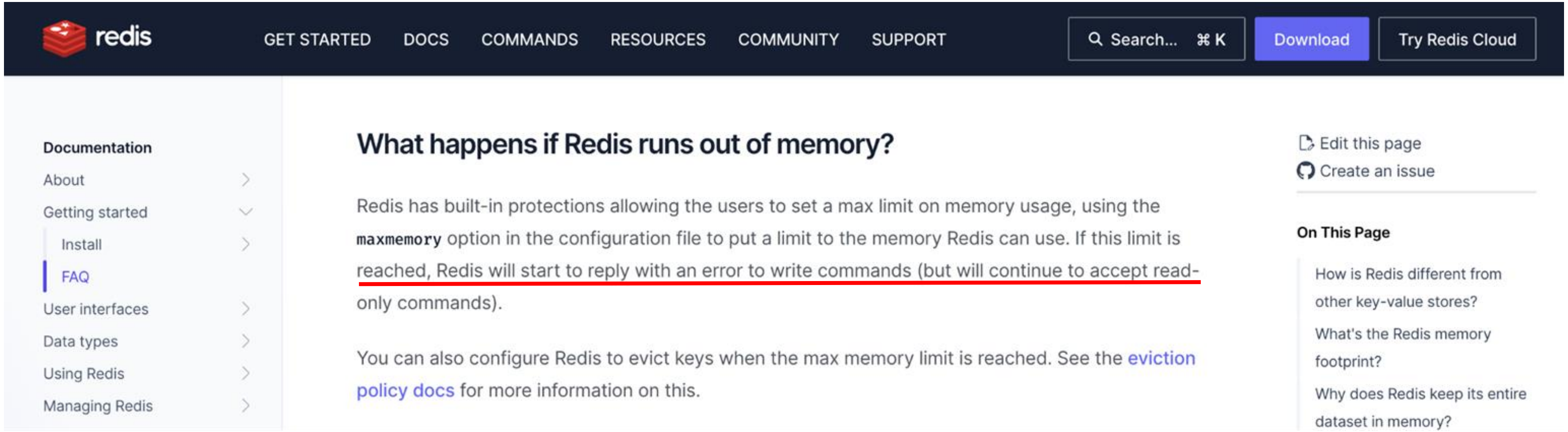
In-Memory



Same **(offset, tag)** value are organized as a ***reverse singly-linked-list***

What If Data Larger than Memory?

Append-Only-Log


redis GET STARTED DOCS COMMANDS RESOURCES COMMUNITY SUPPORT Search... ⌘ K Download Try Redis Cloud

Documentation
About
Getting started
Install
FAQ
User interfaces
Data types
Using Redis
Managing Redis

What happens if Redis runs out of memory?

Redis has built-in protections allowing the users to set a max limit on memory usage, using the `maxmemory` option in the configuration file to put a limit to the memory Redis can use. If this limit is reached, Redis will start to reply with an error to write commands (but will continue to accept read-only commands).

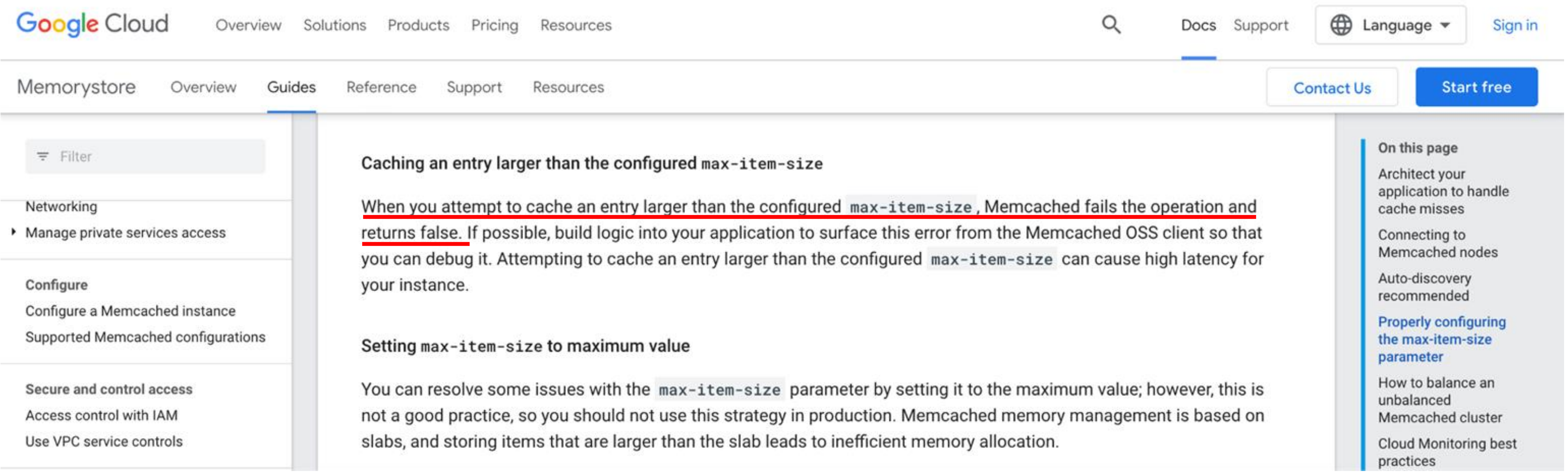
You can also configure Redis to evict keys when the max memory limit is reached. See the eviction policy docs for more information on this.

Edit this page
Create an issue

On This Page
How is Redis different from other key-value stores?
What's the Redis memory footprint?
Why does Redis keep its entire dataset in memory?

What If Data Larger than Memory?

Append-Only-Log



Google Cloud Overview Solutions Products Pricing Resources Docs Support Language Sign in

Memorystore Overview Guides Reference Support Resources Contact Us Start free

Filter

Networking
Manage private services access

Configure
Configure a Memcached instance
Supported Memcached configurations

Secure and control access
Access control with IAM
Use VPC service controls

Caching an entry larger than the configured `max-item-size`

When you attempt to cache an entry larger than the configured `max-item-size`, Memcached fails the operation and returns false. If possible, build logic into your application to surface this error from the Memcached OSS client so that you can debug it. Attempting to cache an entry larger than the configured `max-item-size` can cause high latency for your instance.

Setting `max-item-size` to maximum value

You can resolve some issues with the `max-item-size` parameter by setting it to the maximum value; however, this is not a good practice, so you should not use this strategy in production. Memcached memory management is based on slabs, and storing items that are larger than the slab leads to inefficient memory allocation.

On this page
Architect your application to handle cache misses
Connecting to Memcached nodes
Auto-discovery recommended
Properly configuring the max-item-size parameter
How to balance an unbalanced Memcached cluster
Cloud Monitoring best practices

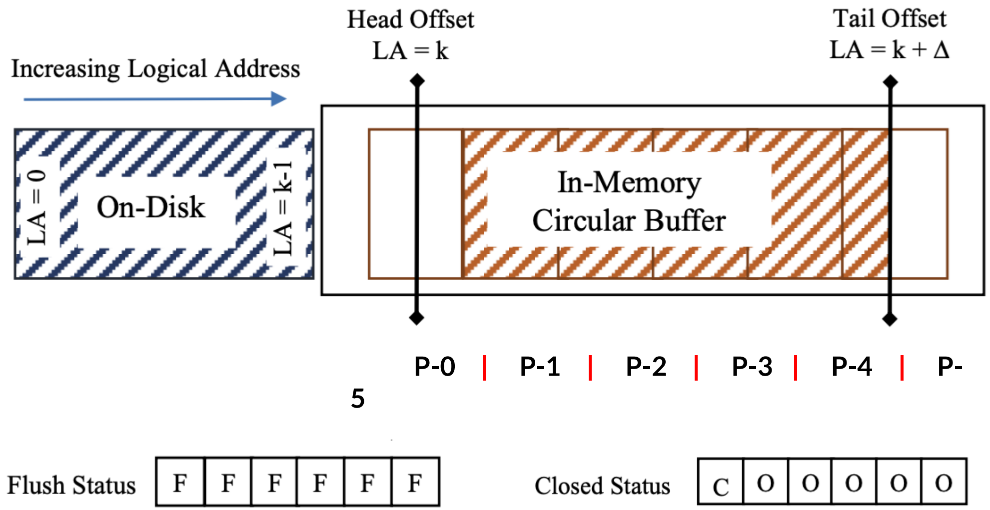
Handling Larger Data In FASTER

Append-Only-Log

- **Circular Buffer** (*Circular Queue*)

When can we flush the *Page* on the disk

Do you see anything that could go wrong



Handling Larger Data In FASTER

Append-Only-Log

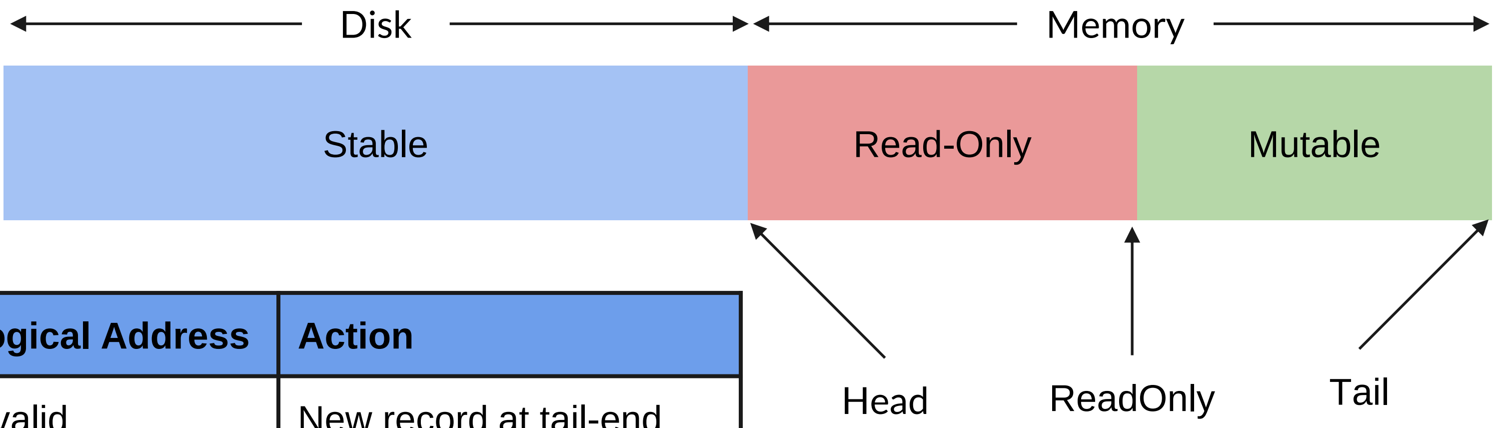
The log-structured allocator is a step in the correct direction but it comes at the cost of

- updating an atomic increment of the tail offset to create a new record
- copying data from the previous location
- atomic replacement of the logical address in the hash index

An append-only log grows fast with update-intensive workloads making disk I/O a bottleneck.

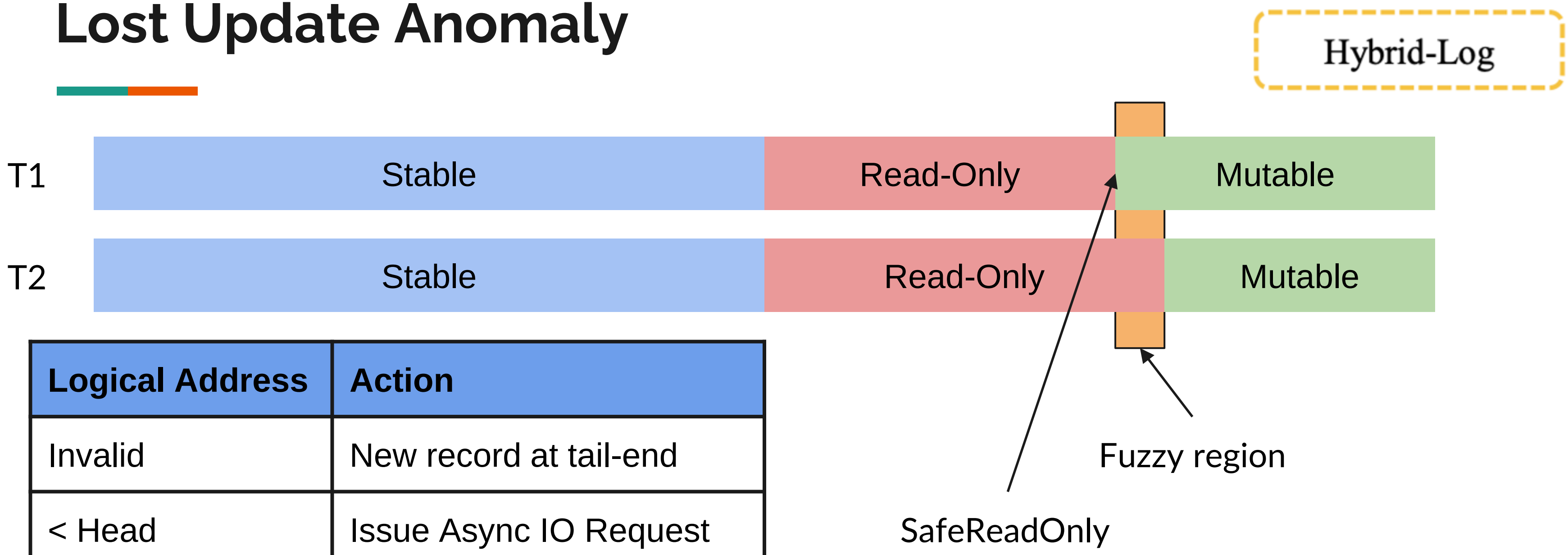
Handling Larger Data In FASTER

Hybrid-Log



Logical Address	Action
Invalid	New record at tail-end
< HeadOffset	Issue Async IO Request
< ReadOnlyOffset	Mutable copy at tail-end
< ∞	Update in-place

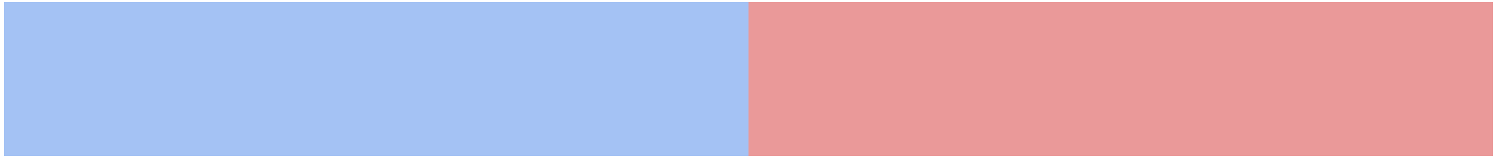
Lost Update Anomaly



Logical Address	Action
Invalid	New record at tail-end
< Head	Issue Async IO Request
< SafeReadOnly	Add to pending list
< ReadOnly	Mutable copy at tail-end
< ∞	Update in-place

Handling Larger Data In FASTER

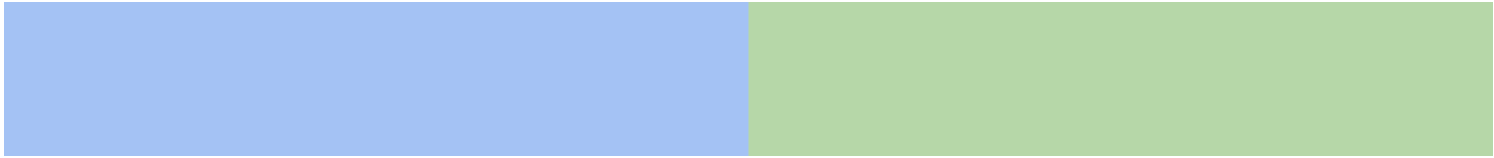
Hybrid-Log



Lag = 0 is an append only log

Handling Larger Data In FASTER

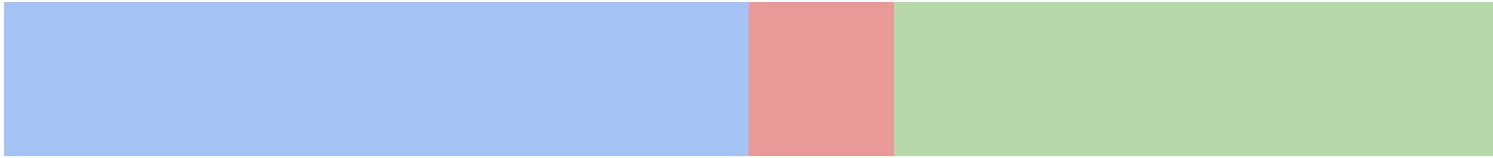
Hybrid-Log



Lag = buffer size is an in-memory store

Handling Larger Data In FASTER

Hybrid-Log



90:10 division for good performance

EVALUATIONS

Throughput

Memory budget

Faster and HybridLog

Setup and Workload

Machine – Dell PowerEdge R730 server

- 2 socket, 14 cores per socket, 2 hyper-threads per core (56 Hyper Threads)
- 256GB RAM, 3.2TB FusionIO NVMe SSD

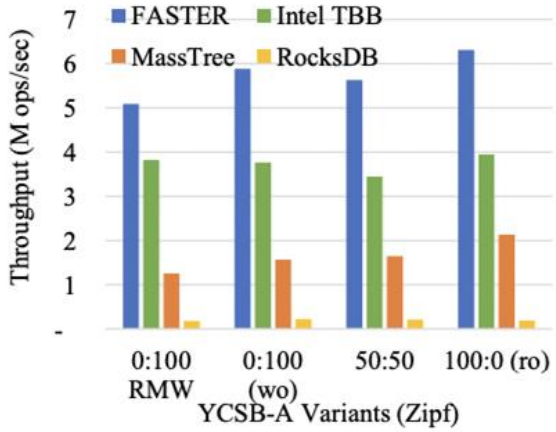
Baseline Systems

- In-memory structures: Intel TBB hashmap, Masstree
- Key-value store: RocksDB

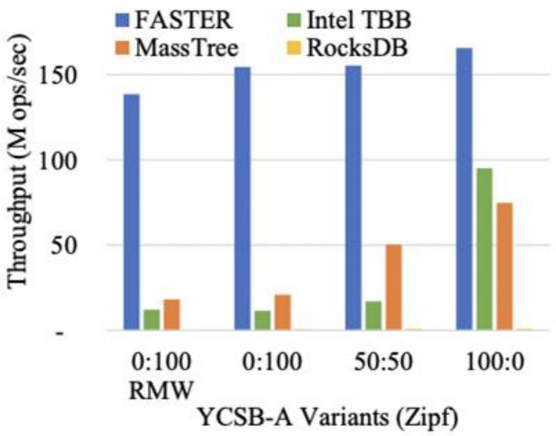
Modified YCSB-A workload

- 250 million distinct 8-byte keys, values of 8 and 100 bytes
- Varying fraction of reads, blind updates, read-modify-writes

Throughput - Single and MultiThreaded

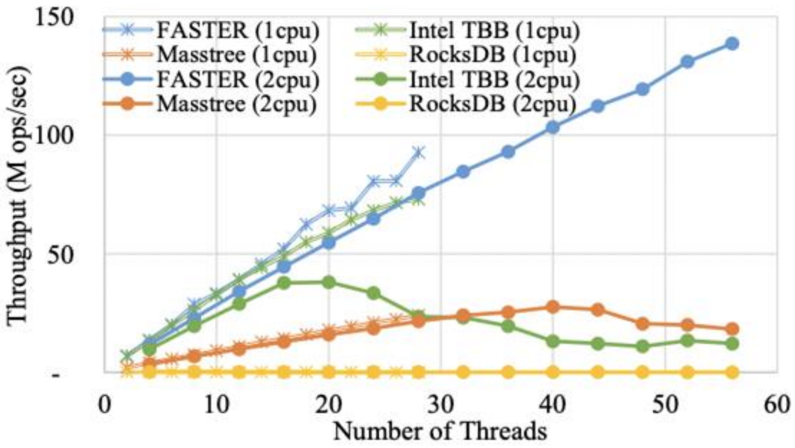


Single Thread

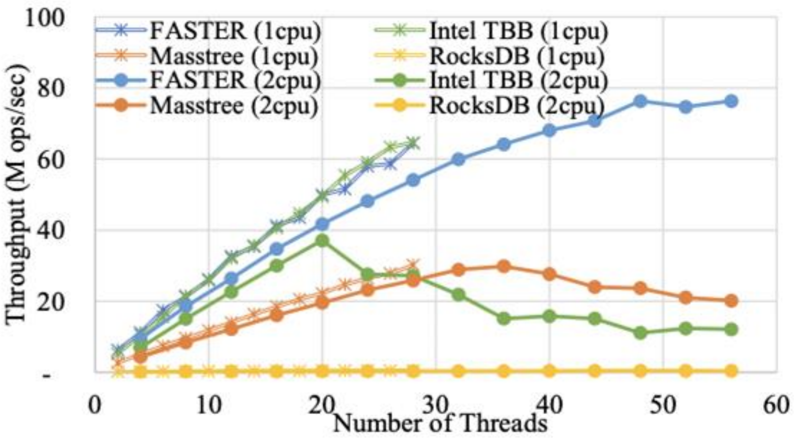


Multi thread

Scalability with number of Threads



(a) RMW updates; 8-byte payloads.



(b) Blind updates; 100-byte payloads.

Throughput - Increasing Memory Budget

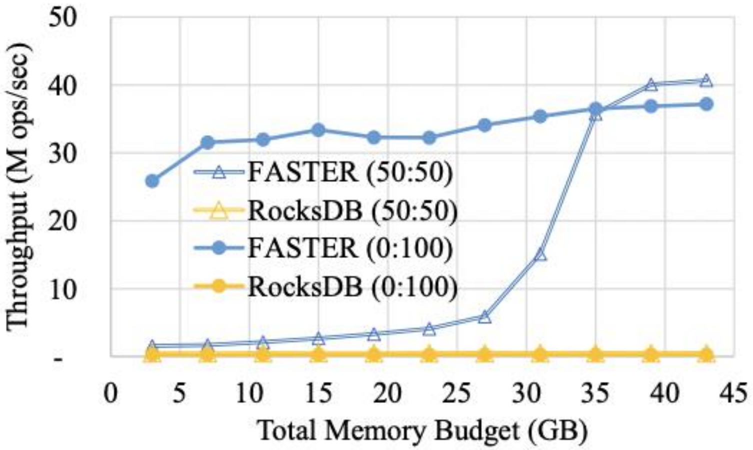
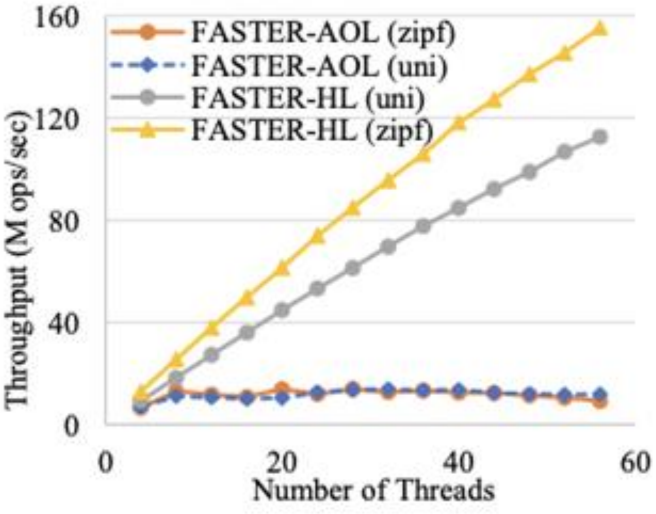


Figure 10: Throughput with increasing memory budget, for 27GB dataset.

Throughput - Append-only vs. Hybrid logs



?

Why hybrid log is better

Conclusions

FASTER can “**HAVE IT ALL**”

Larger-Than-Memory

Temporal locality

Degrade gracefully

High Performance
(up to 160 million ops/sec)