Introduction to the Bw-Tree, Wormhole and HydraList

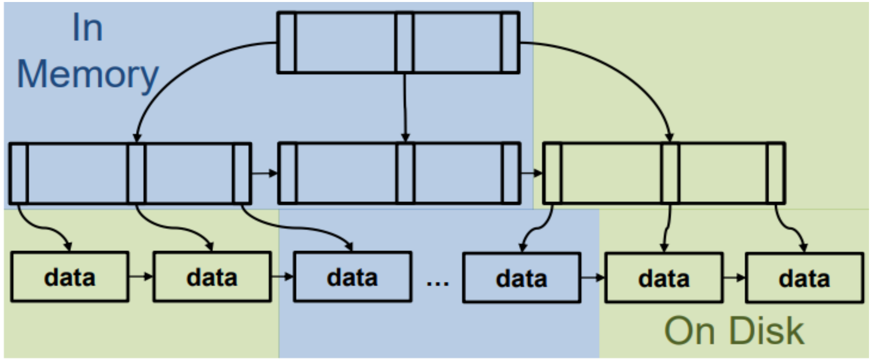
Yizheng Jiao

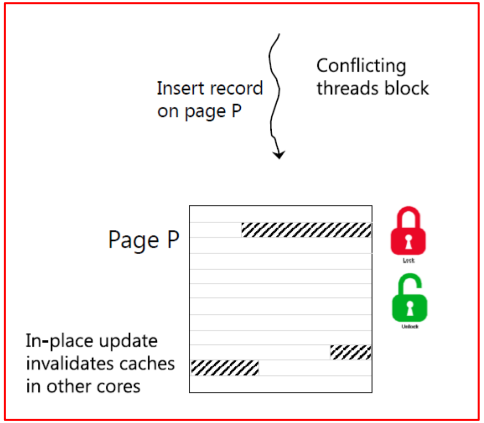
Content

- **Bw-tree**
- Wormhole
- hydraList

Background of Btrees

- Key-ordered access to records
- Pivots in the internal nodes (guide search)
- Data entries in leaf nodes
- Efficient point and range lookups
- Balanced Tree (node split/merge)
- Hand-over-hand locking

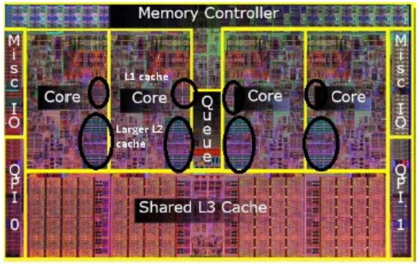




Classical B-tree concurrency control

Why a new B-tree

- Multi-core
  - We live in a high peak performance multi-core world
    - Uni-core speed will be at best increase modestly
  - Multi-core CPUs mandate high concurrency
  - Good multi-core process performance depends on high CPU cache hit ratios
    - Reduce updating memory in place (cache invalidations)
- Modern Storage Devices
  - Flash storage offers higher I/O ops per second than HDDs
  - Random write is still slower than sequential write
    - Need an erase cycle prior to write



Bw-tree Overview

- A new form of B-tree
  - Provide logarithmic access to keyed records
  - Provide linear time access to sub-ranges
- Latch-free: threads almost never block
- Bw-tree installs state changes using the atomic CAS
- Bw-tree only blocks when it needs to fetch a page from stable storage
- Bw-tree performs node updates via “delta updates”
  - Attach the update to an existing page, not via update-in-place
  - Avoid update-in-place reduces CPU cache invalidation and increase CPU cache hit
- Bw-tree targets on flash storage
  - Minimize blocking on reads
    - Large memory buffer + fast random read access
  - Minimize blocking on writes
    - Log structure storage layer enables writing large buffer

Concurrency

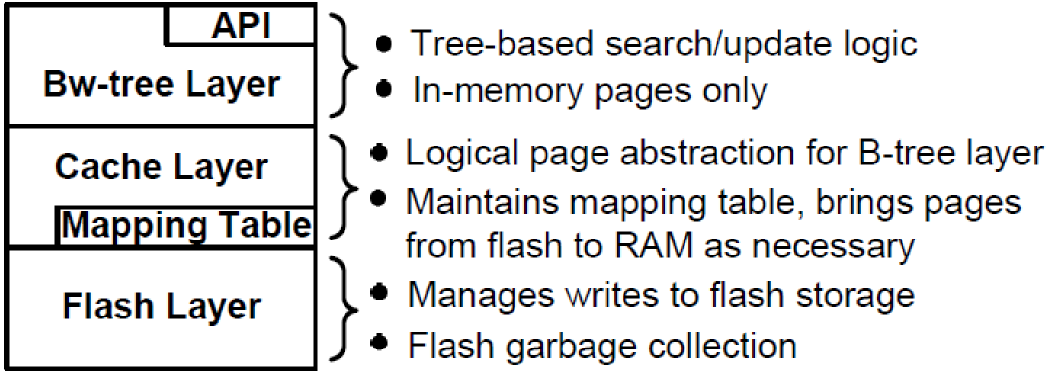
Cache

Flash Friendly

Design Features of Bw-Tree

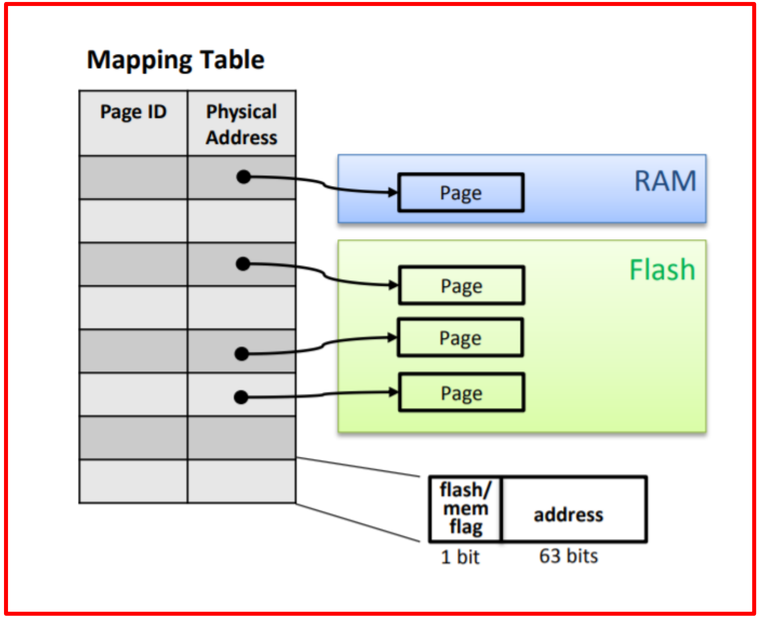
Bw-tree Architecture

- Access method layer (Bw-tree layer)
- Cache layer
- Storage layer



The mapping table

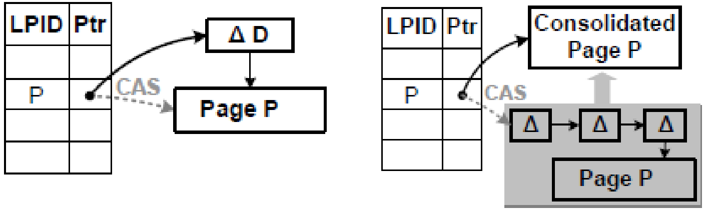
- The cache layer maintains a mapping table
  - Mapping logical pages to physical pages
  - Translate a page identifier (PID) into a flash offset or a memory pointer
- Use PIDs instead of physical pointers in the bw-tree to link nodes of the tree
- Benefits
  - Location change not propagate to the root of the tree
  - Enable delta updating of the node



Bw-tree nodes are logical and do not occupy fixed physical location, either in memory or on flash

Delta Updating

- Page state changes are done by creating a delta record and prepending it to an existing page state
- Bw-tree installs the new memory address of the delta record in the mapping table using CAS
- Consolidate pages
  - Create a new page that applies all delta changes
  - Reduce memory footprint and improve search performance
  - The consolidated page is installed with a CAS, and old page is garbage collected

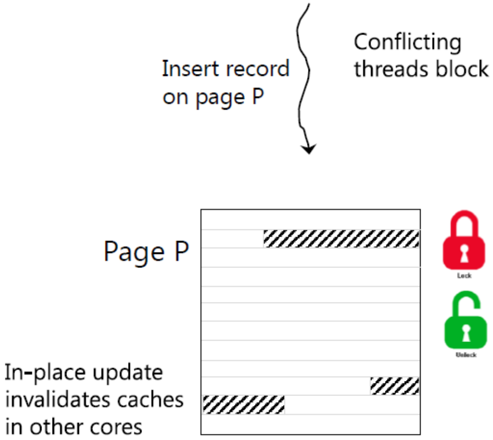


(a) Update using delta record (b) Consolidating a page

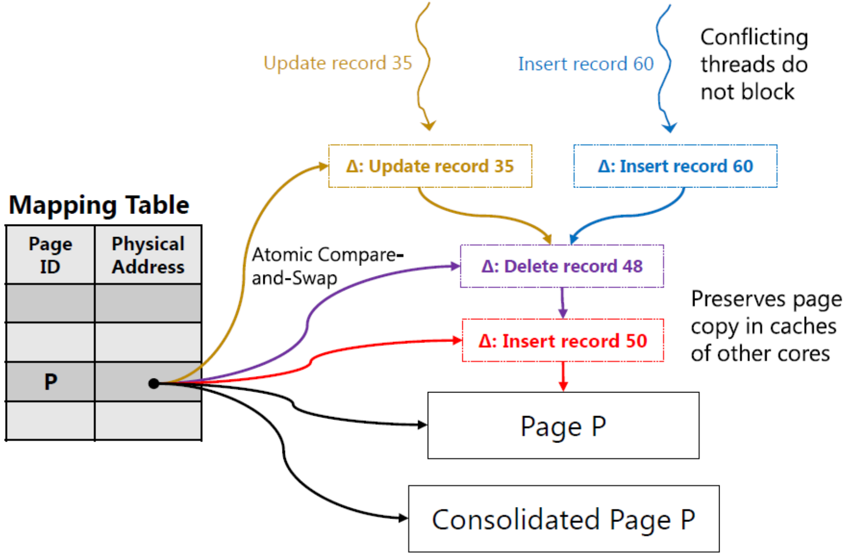
Delta updating enables latch-free access in Bw-tree and preserves processor data caches by avoiding update-in-place

Highly Concurrent Page Updates with Bw-tree

Classical B-Tree

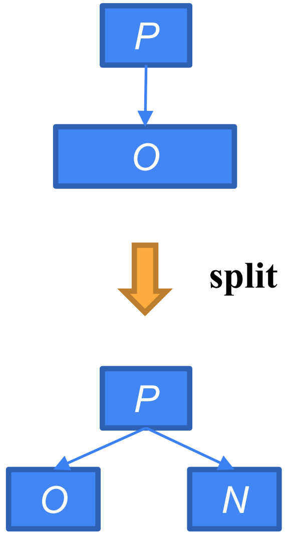


Bw-Tree



Bw-tree Structure Modifications

- How to ensure atomicity for tree structural modifications (SMOs)?
  - CAS can only ensure atomic change to a single page
  - A node split introduces changes to more than one page
  - The situation is similar for merging nodes, but harder
- Bw-tree breaks an SMO into a sequence of atomic actions
  - Each action is installable via a CAS
  - Use a B-link tree design
- A thread can see partial SMO without waiting for it to complete



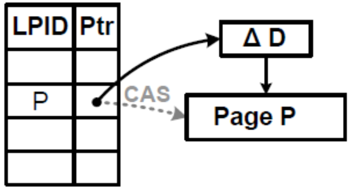
Implementation Details of Bw-Tree

In-memory latch-free pages

- Internal pages contain <pivot, pointer> pairs sorted by pivots
- Leaf pages contain <key, record> pairs
- Pages also contain:
  - A **low key** representing the smallest key value that can be stored on the page (and in the subtree below)
  - A **high key** representing the largest key value that can be stored on the page
  - A **side link** pointer that points to the node's immediate right sibling on the same level
- Bw-tree pages are logical (virtual), no fixed physical location
- Bw-tree pages are elastic , no hard limit on how large a page may grow
- No update-in-place

B-link tree

A page is a base page (a btree node) along with its delta chain



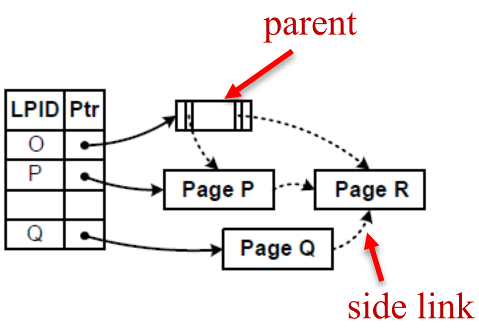
Page search, consolidation and range scans

- Leaf page search involves traversing the delta chain
  - Search stops at the first occurrence of the search key in the chain
- Page Consolidation
  - Search performance eventually degrades if delta chains grow too long
  - Bw-tree occasionally performs page consolidation that creates a new “re-organized” based page
  - Bw-tree triggers consolidation an accessor thread notices the delta chain is too long
- Range scans
  - Specified by a key range <start_key, end_key>
  - Scan the data page and push all records in this range into a vector
    - Records in the vector may be changed by other update threads
    - Rely on external transactional locking or reconstruct the vector when records are updated in the tree
  - Iter->advance is atomic, but entire scan is not atomic
    - no snapshot read, designed as **atomic record stores**

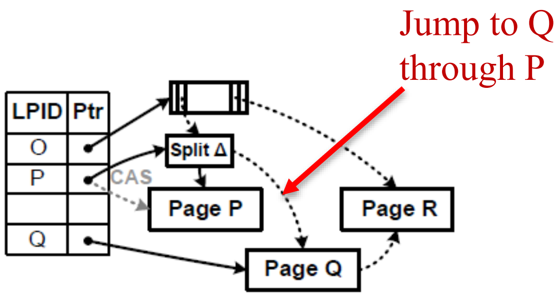
Page Split (1)

- Splits are triggered by an accessor thread that notices a page size has grown beyond a system threshold
- Splits has two phases
- Atomically install the split at the **child level**

- Find a split pivot Kp and create a new page Q
- Copy all records > Kp to Q
- Create a side link in Q to R
- Install the physical address of Q in the mapping table
- Install split delta record to P
- Use Kp to invalidate records in P > Kp
- Add a new side link to Q



(a) Creating sibling page Q



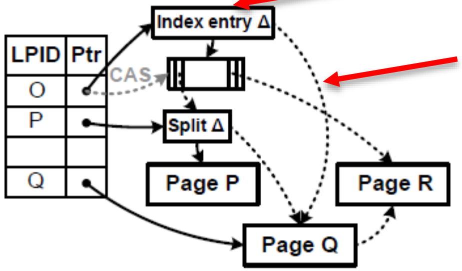
(b) Installing split delta

Page Split (2)

- In order to guide searches directly to Q
  - Prepend a delta to the parent of P and Q
    - Kp, the pivot between P and Q
    - A logical pointer to Q
    - Kq, the pivot between Q and R
- The parent might be merged with other nodes
  - Epoch garbage collection prevents the parent's page from being reclaimed
  - If the parent is deleted, go up the tree to the grandparent node and do a traversal down

Index entry delta to the parent

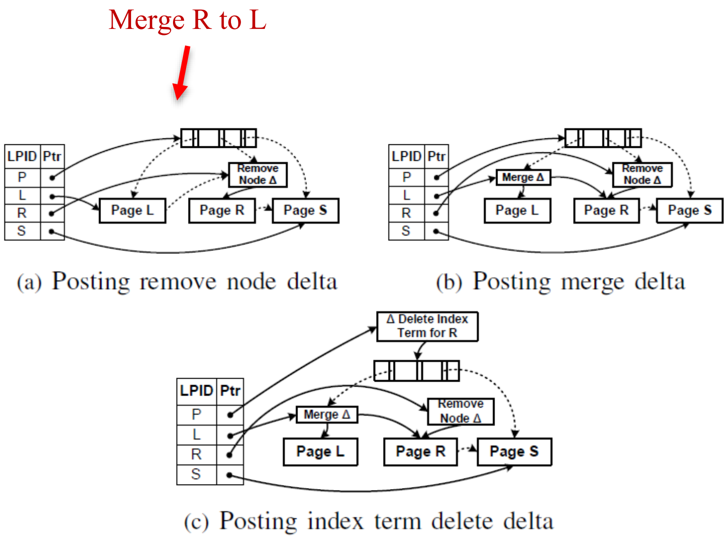
For k >= Kp and k < Kq



(c) Installing index entry delta

Page Merge

- Marking for delete
  - The node R to be merged is updated with a **remove node delta** to stop all further use of R
  - A thread encountering a remove node delta in R needs to go to the left sibling L
- Merging children
  - The left sibling L of R is updated with a node **merge delta** that physically points to the contents of R
- Parent Update
  - The parent node P of R is now updated by deleting its index term associated with R by posting an **index term delete delta**
  - All path to R are blocked and R can be reclaimed



(a) Posting remove node delta

(b) Posting merge delta

(c) Posting index term delete delta

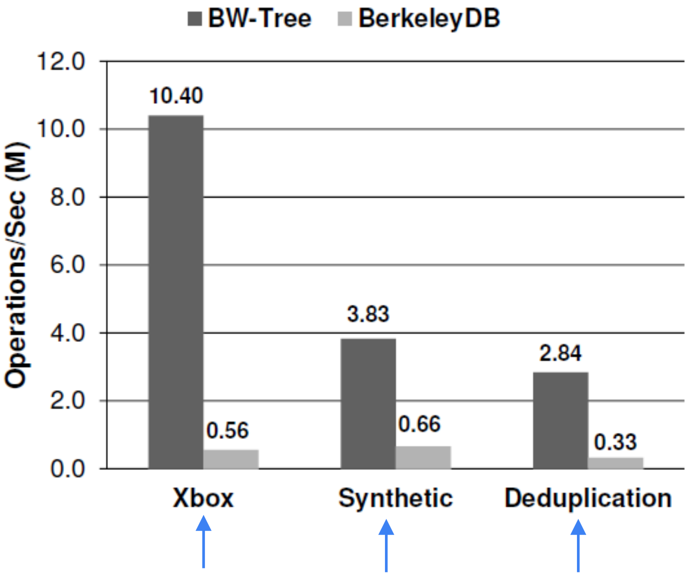
Serializing SMOs and updates

- Bw-tree needs to correctly serialize data updates with SMOs and SMOs with other SMOs
- Treat an SMO as atomic and conceal the fact that there are multiple steps
- If a thread encounters an incomplete SMO, such a thread must complete and commit the SMO before it can post its update or continue with its own SMO
- For split
  - If a thread traverse a side pointer to reach the correct page, it must complete the split SMO by posting the new index term delta to the parent
- Can have a stack of SMOs
  - Need to complete previous SMOs reclusively

Cache Management

- The cache layer is responsible for reading, flushing, and swapping pages between memory and flash
- To keep track of which version of the page is on stable storage and where it is, bw-tree uses a ***flush delta record***
    - Subsequent flush only sends incremental page changes to stable storage
    - When a page flush succeeds, the flush delta contains the new flash offset

Results



rd-to-wr ratio: 7.5:1 2.2:1 5:1

	Bw-tree	Skip List
Synthetic workload	3.83M ops/sec	1.02M ops/sec
Read-only workload	5.71M ops/sec	1.30M ops/sec

TABLE II

BW-TREE AND LATCH-FREE SKIP LIST

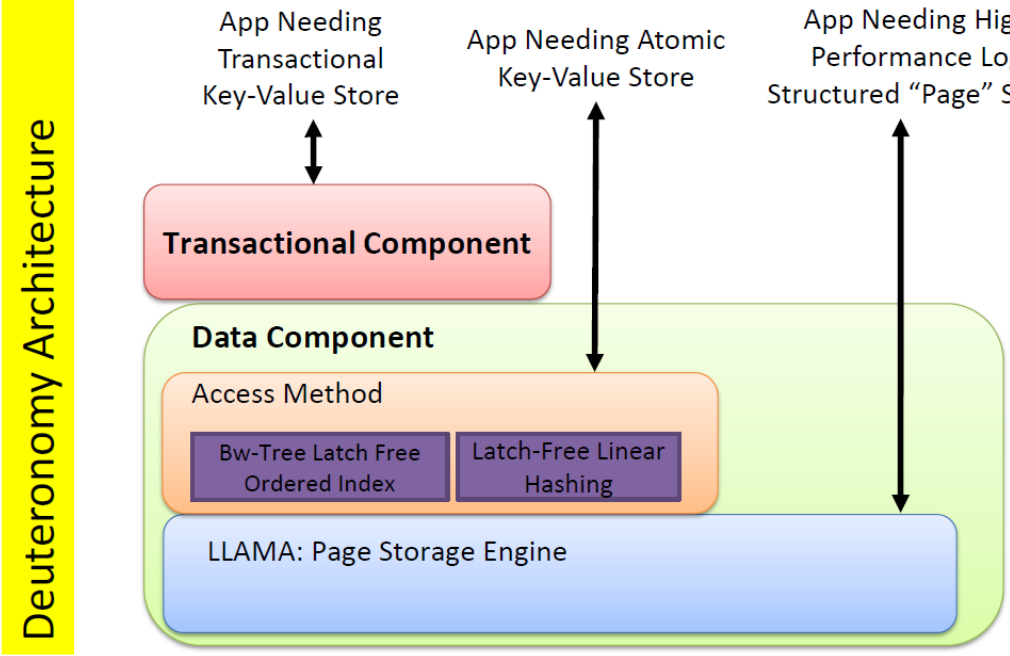
- Latch-freedom: no threads block
- Cache efficiency: delta vs. update-in-place

Know more about Bw-tree --- OpenBw-tree

"Building a Bw-Tree Takes More Than Just Buzz Words" [Sigmod'18]

- Clarify missing points in Microsoft's original design documents
- Present techniques to improve the index's performance
- Show that the Bw-Tree does not perform as well as other concurrent data structures that use locks even with the paper's improvement

Want to use Bw-tree…



Decouple an index engine into three component

Content

- **Bw-tree**
- Wormhole
- hydraList

Wormhole Motivation

- In-memory database host all data and metadata in the main memory
  - Expensive I/O operations are removed (at least from the critical path)
- Index operations become a major source of the system's cost (14-94%) of the query time for in-memory DB
  - The memory becomes increasing large
  - The indexed data can be small
- Lookup cost in a comparison-based ordered index is O(logN) key comparison
  - A B+ tree of one million keys a lookup requires about 20 key comparisons
  - Pointer chasing incurs cache misses
  - Much slow than hash tables

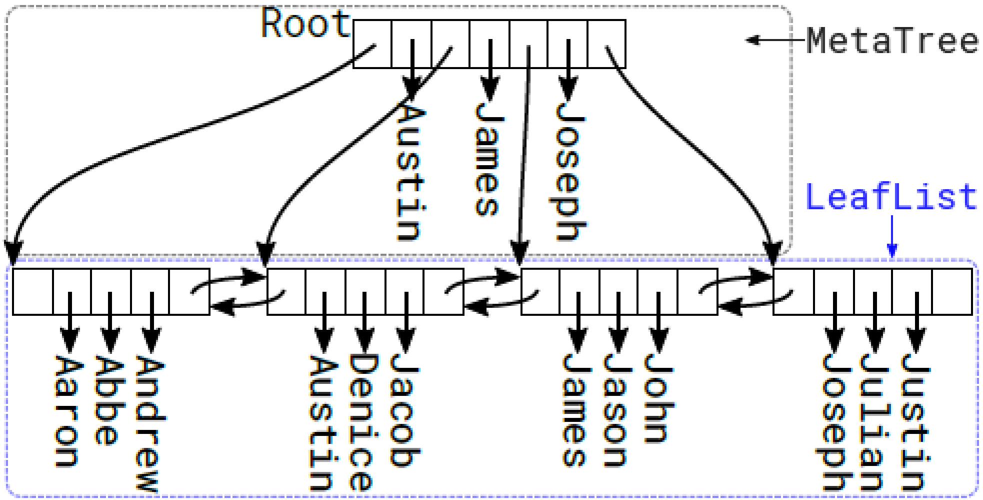
Comparison of Existing Indices

Data Structure	Pros	Cons
B+ Tree	* Space efficient (large leaf nodes) * Support of range operations	High lookup cost with a large *N*
Prefix Tree	Lookup cost not correlated with N	* High lookup cost even with a moderate L * Space inefficiency
Hash Table	O(1) lookup cost	No support of range operations

Wormhole orchestratre B+-tree, prefix tree and hash table in one index structure

- Has O(logL) search cost
- Memory efficient
- Support range search

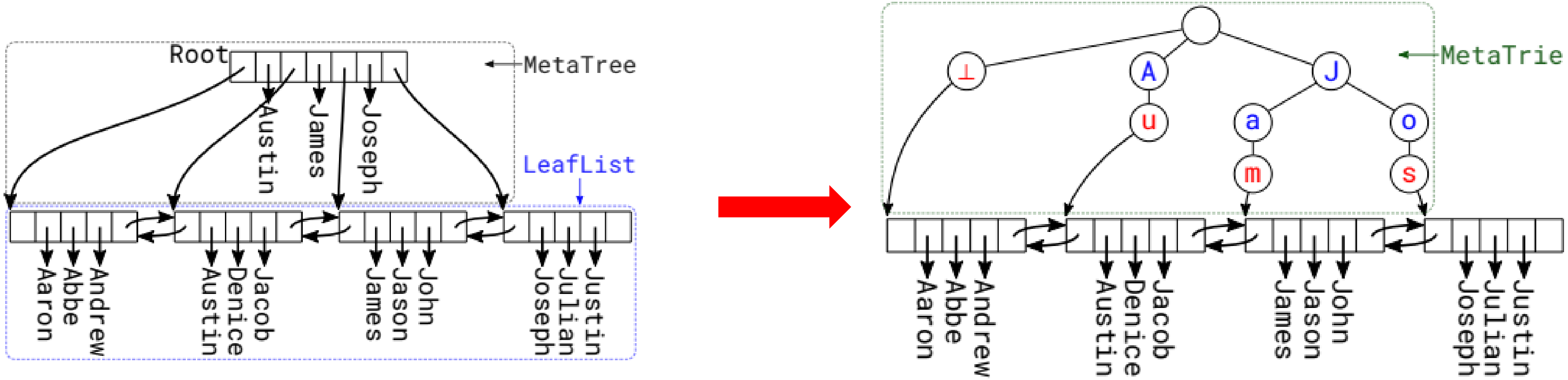
Decompose a B+-tree



- Internal nodes guide the search --- MetaTree
- Data is located in leaf nodes and leaves are linked together --- LeafList

Replace B+-tree's MetaTree with a Trie

"Au", "Jam" and "Jos" are called anchor

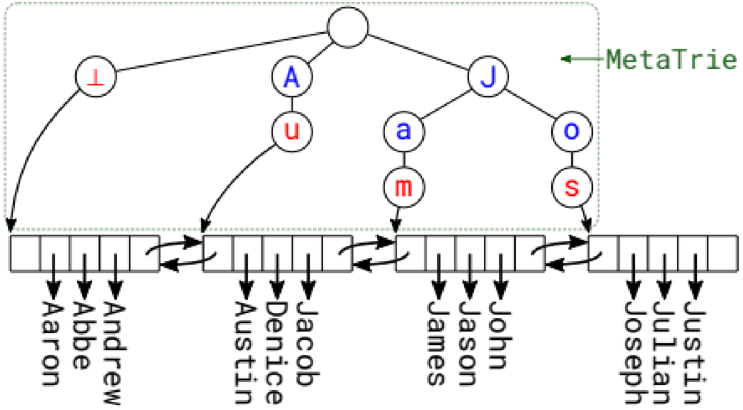


The lookup cost is reduced from O(LogN) to O(L)

How to query

- The search key is fully matched in the MetaTrie
  - Search the leaf node pointed to the search path
- The search key has a token that cannot be matched
  - If the left sibling exists, the search key's target node is the right-most leaf node of the left subtree
  - If the right sibling exists, the left-most leaf node of the right subtree is the target node's immediate next node on the LeafList
- The search key is only partially matched
  - Append a smallest token ⊥ to the search key
  - The same with the previous case

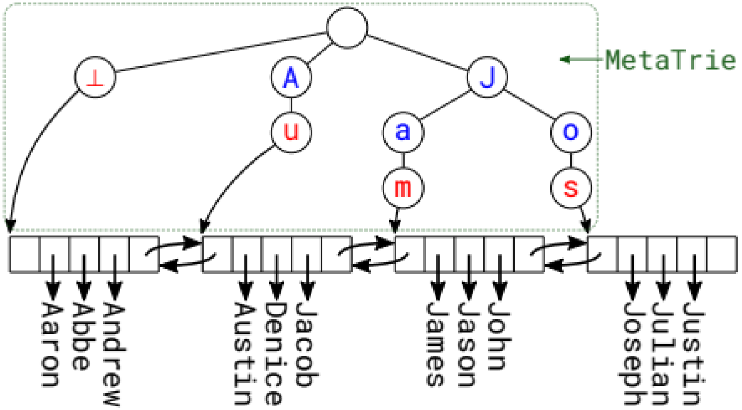
Examples:
"Joseph"
"Denice"
"A"



Accelerate Query with a HashTable

- Walking the MetaTrie has O(L) cost
- It can be further reduced with a HashTable

- Use binary search on prefix lengths to accelerate the match

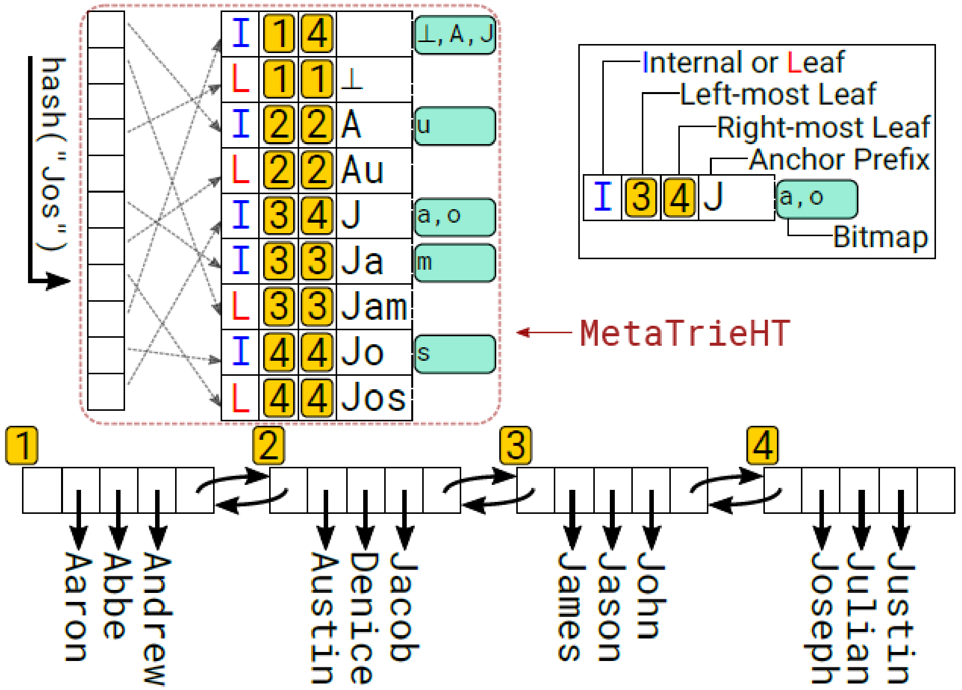


Example:
“Jam” is an anchor, and its prefixes are (“”, “J”, “Ja”, “Jam”) are inserted into the hash table

Woemhole Architecture

Example:
for “James”, only needs to lookup “Ja” and “Jam” in the hashtable

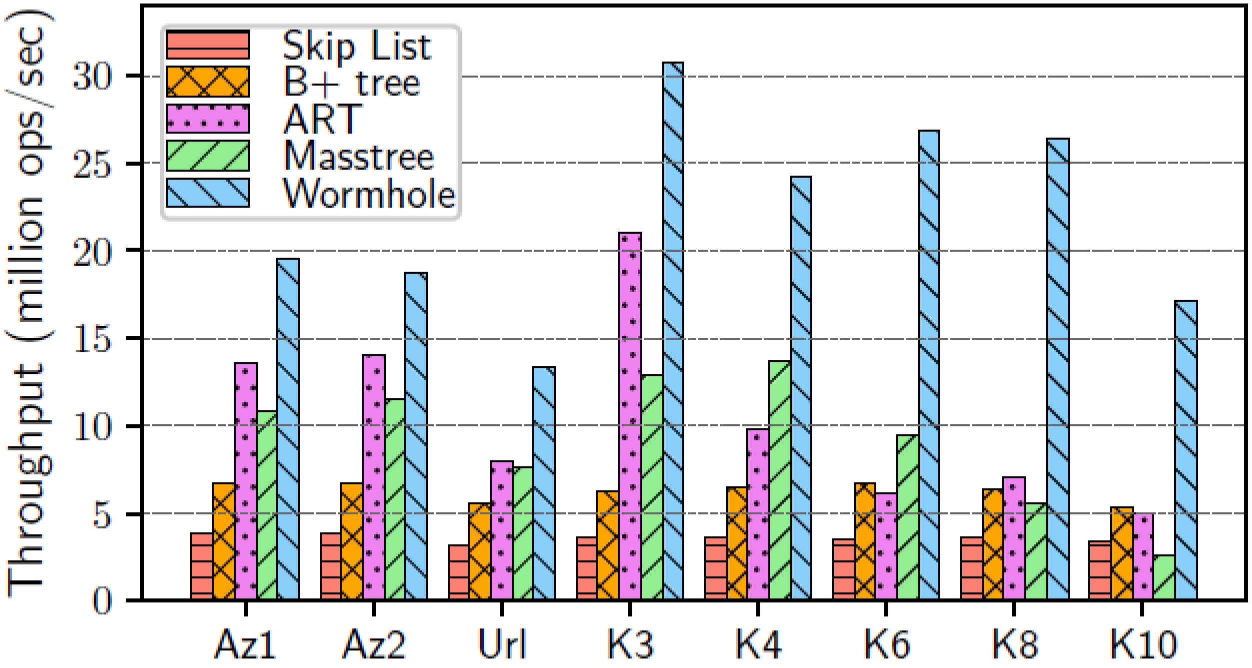
The search cost is O(logL)



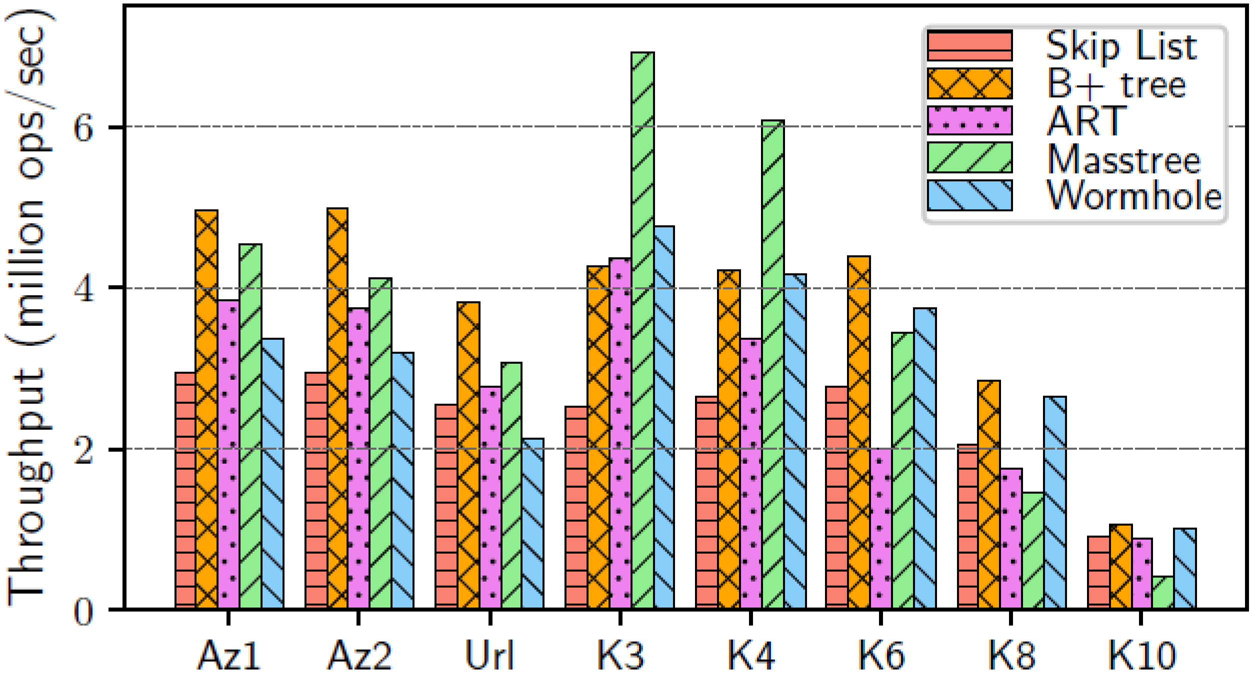
Other Techniques

- Concurrency control
  - RCU
- Improvements to the Hash Table
  - Incremental hashing
- Improvements to the leaf node operation
  - Tag

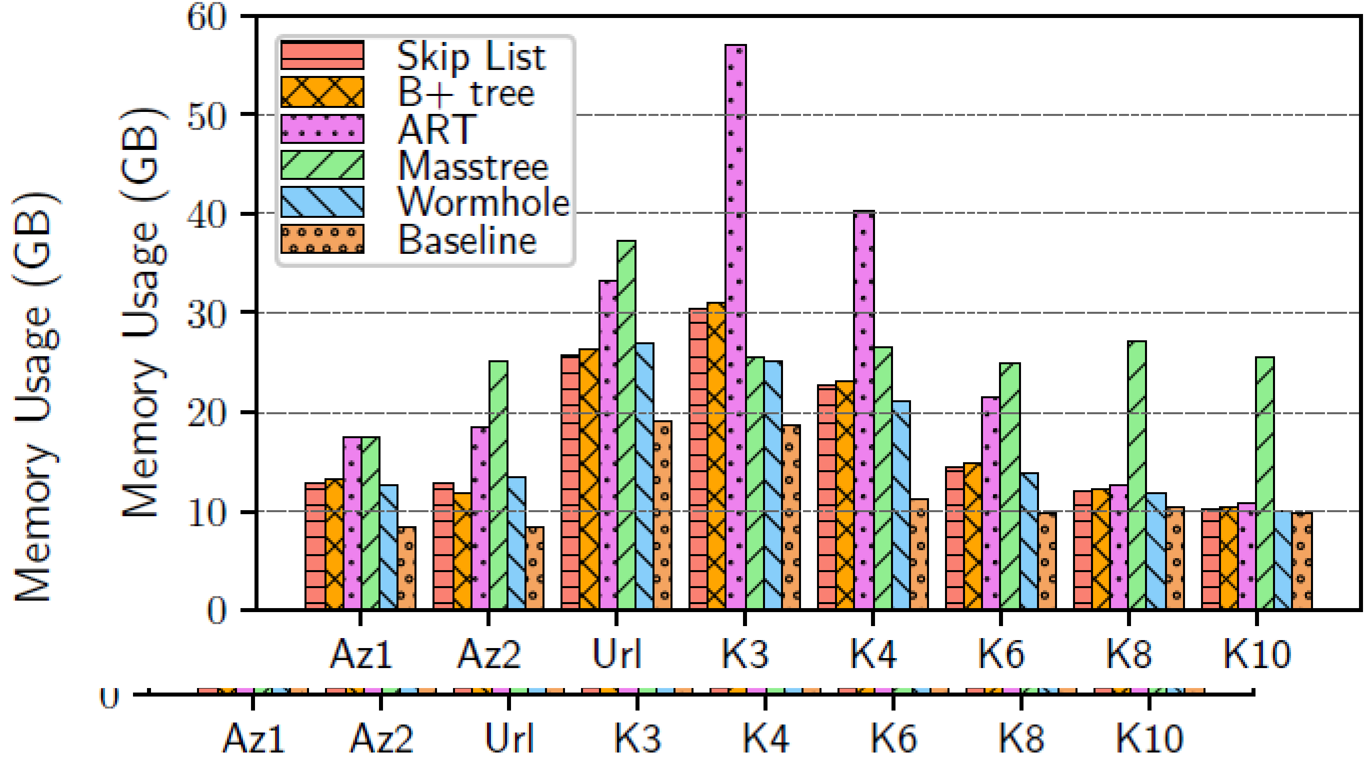
Lookup Througput



Continuous Insertion



Memory Usage



Content

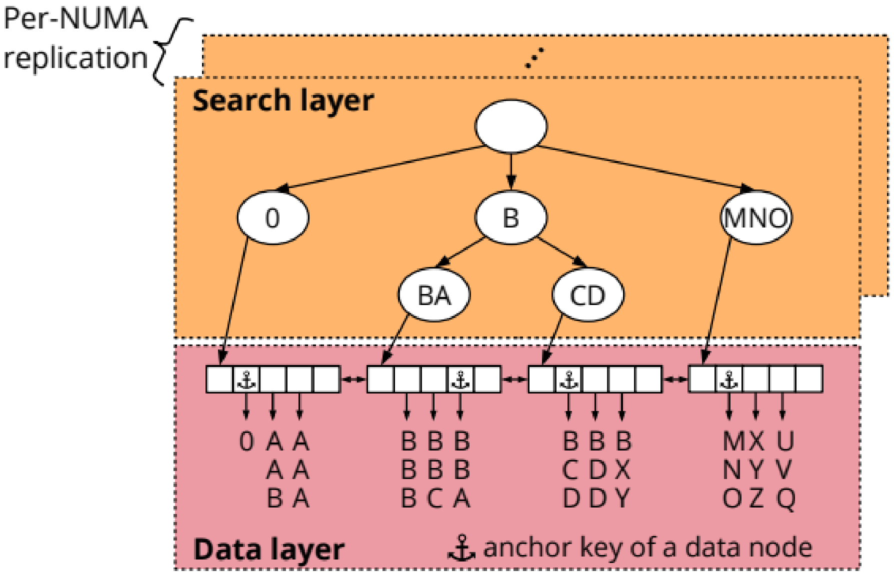
- Bw-tree
- Wormhole
- **HydraList**

Key Ideas

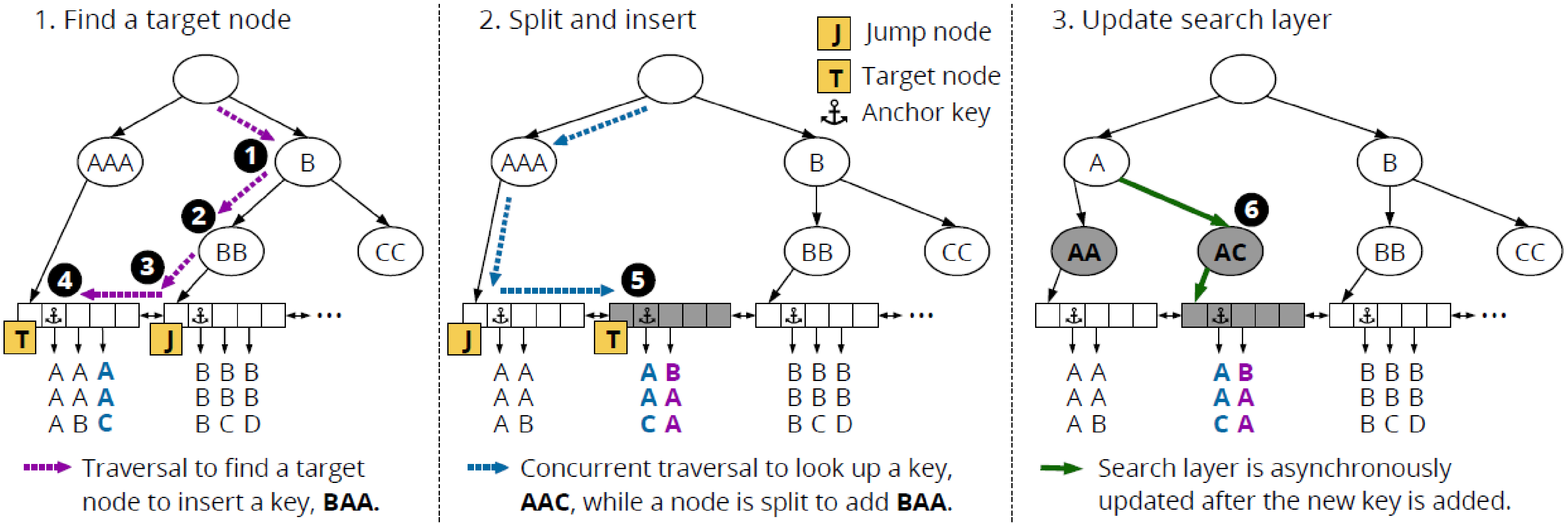
- Separate and individually optimize an index into two components
    - Search layer
    - Data layer
- Decoupled layers
    - Allow asynchronous updates to the search layer
    - Reduce the synchronization overhead with small critical section
- Replicate the search layer across NUMA nodes
    - Reduce memory stalls caused by cross-NUMA accesses

HydraList Architecture

- Search layer is a Adaptive Radix Tree
- Data Layer is a doubly-linked list
- Search layer is replicated

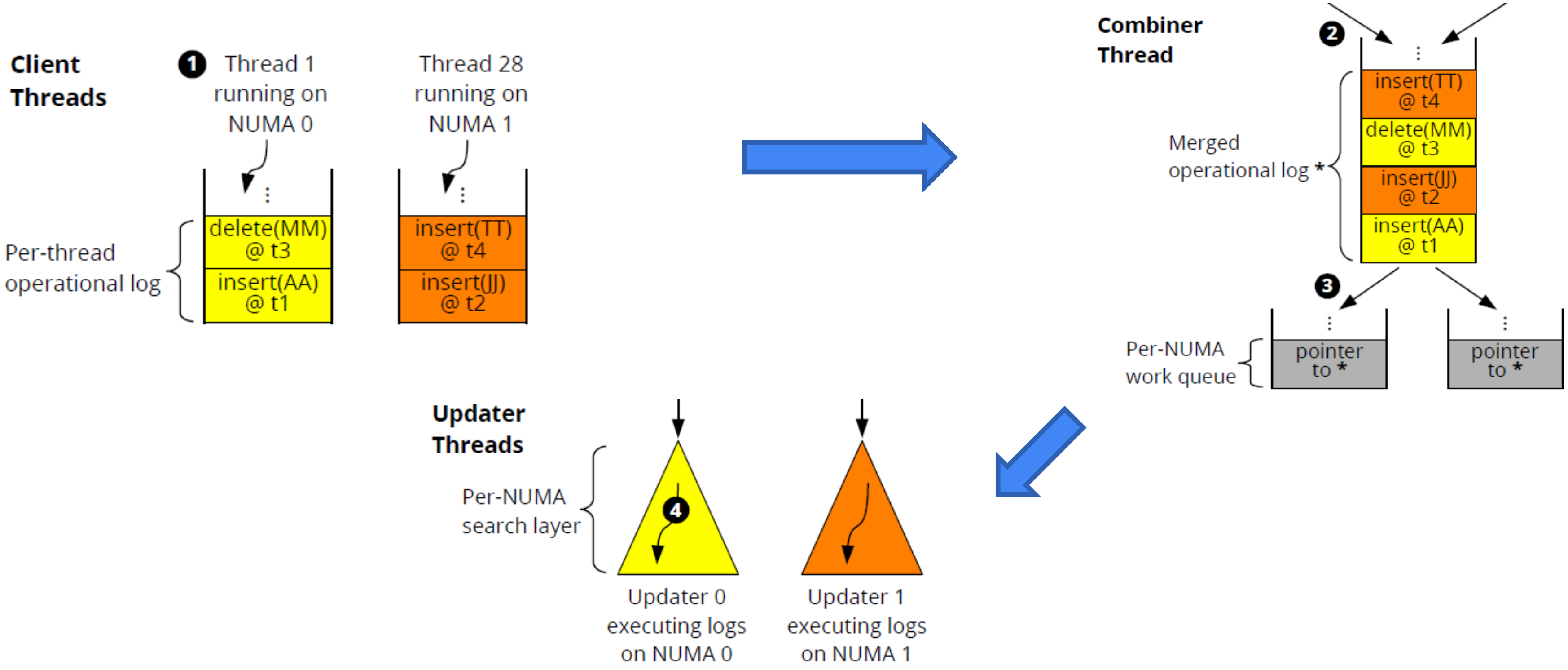


HydraList Updates



Insert "BAA", which causes a leaf to split

Asynchronous Update of the Search Layer



Timestamp

- Timestamp is generated by using ORDO
- Cannot use RDTSC instruction as hardware counters in different sockets have a constant skew between them
- Now special hardware is needed

Concurrency Control

- Decoupled layers allows for different concurrency control for each layer
- Search Layer
  - Updated by a single thread
  - Readers can prioritize the writer
  - Read-Optimized Write Exclusion protocol
- Data Layer
  - Multiple reader and writer
  - Ensure parallelism
  - And reduce cache coherence traffic
  - Use Optimistic Version Locking protocol

Results

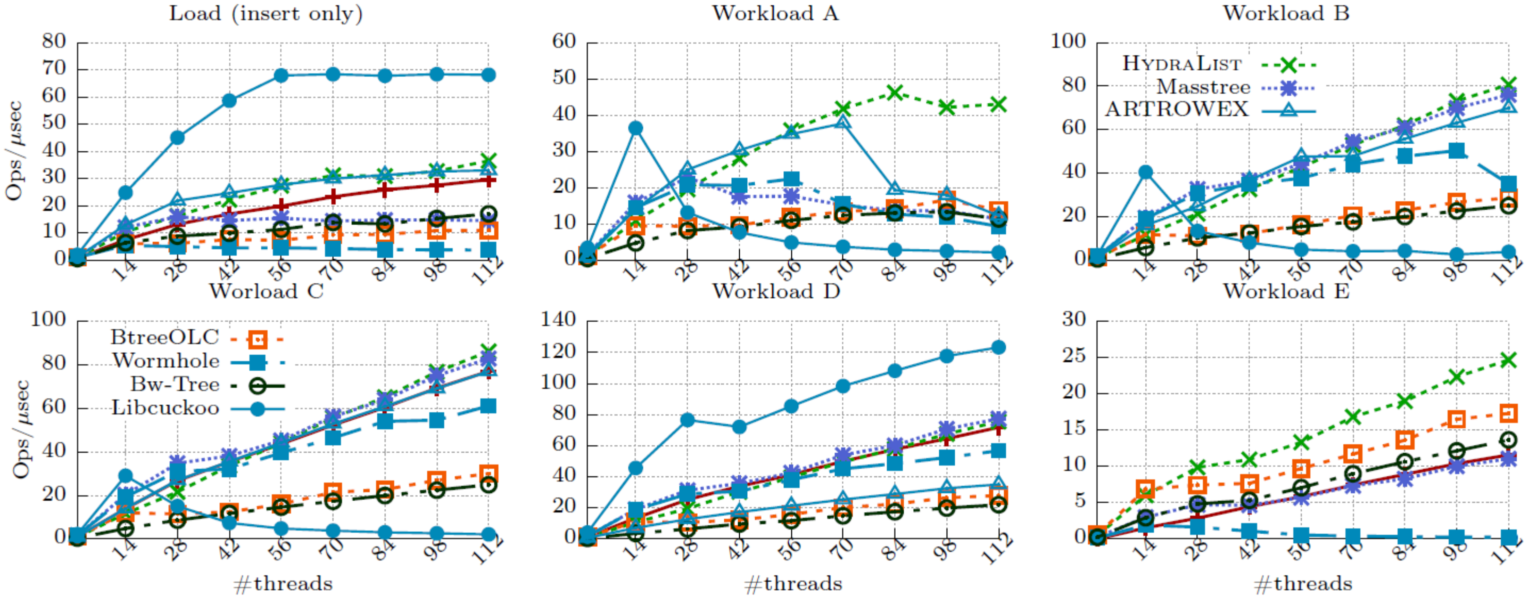


Figure 6: Performance comparison of in-memory indexes for YCSB workload: 50 million operations with 89 million string keys.

HydraList has better performance for A, B, C and E with a large number of threads

Summary

- All three indices are B-tree variant
- Bw-tree
  - Concurrency
  - Cache
  - Flash
- Wormhole
  - Lookup
- HydraList
  - Concurrency
  - Update

Thank you!

Q & A