Georgia Tech

Lecture 17: Modern OLTP Indexes (Part 1)

CREATING THE NEXT®

Administrivia

- Assignment 4 has been released

Today's Agenda

- Recap
- T-Tree
- Versioned Latch Coupling
- Latch-Free Bw-Tree
- Conclusion

Recap

Concurrency Control

- We need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores and hide disk I/O stalls.
- A **concurrency control protocol** is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.
- **Physical Correctness:** Is the internal representation of the data structure valid?

Today's Agenda

- T-Tree
- Versioned Latch Coupling
- Latch-Free Bw-Tree

T-Tree

Observation

- The original B+Tree was designed for efficient access of data stored on slow disks.
- Is there an alternative data structure that is specifically designed for **in-memory databases**?
- We assume that both the index and the actual data are fully kept in memory

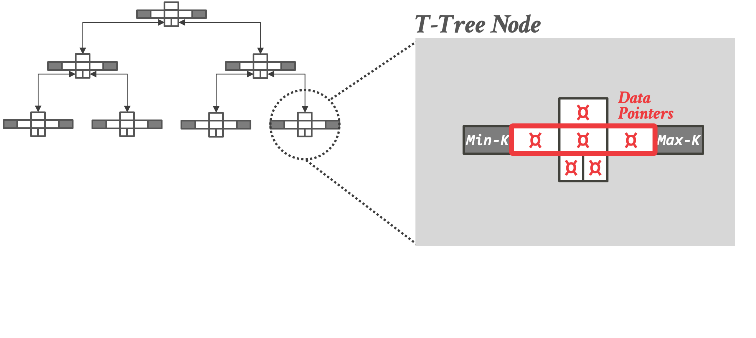
T-Tree

- Based on AVL Tree.
- Proposed in 1986 from Univ. of Wisconsin
- Used in early in-memory DBMSs during the 1990s (*e.g.*, TimesTen, DataBlitz).
- Reference

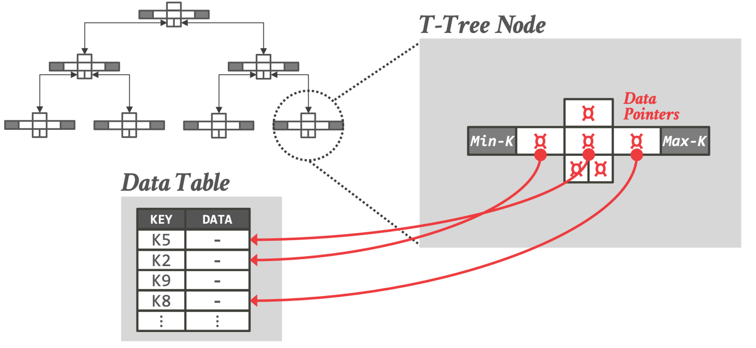
T-Tree

- Instead of storing keys in nodes, store **pointers** to the tuples (*a.k.a.*, data pointers).
- The nodes are still sorted order based on the keys.
- In order to find out the actual value of the key, you have to follow the tuple pointer.

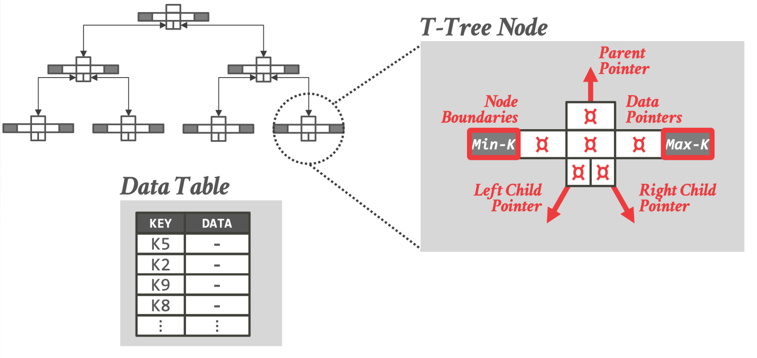
T-Tree



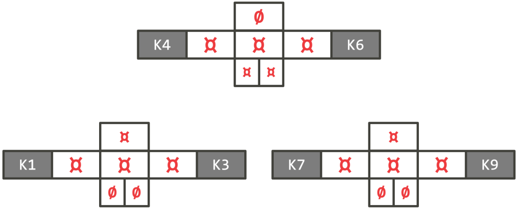
T-Tree


T-Tree Node
Data Pointers
Min-K
Max-K
Data Table
KEY
DATA
K5
-
K2
-
K9
-
K8
-
⋮
⋮

T-Tree


T-Tree Node
Parent Pointer
Node Boundaries
Data Pointers
Min-K
Max-K
Left Child Pointer
Right Child Pointer
Data Table
KEY
DATA
K5
K2
K9
K8

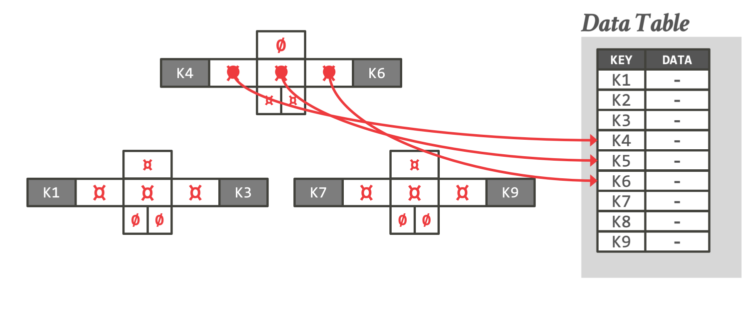
T-Tree: Find K2



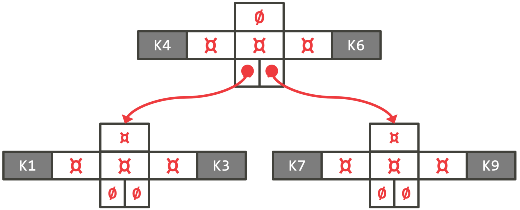
Data Table

KEY	DATA
K1	-
K2	-
K3	-
K4	-
K5	-
K6	-
K7	-
K8	-
K9	-

T-Tree: Find K2



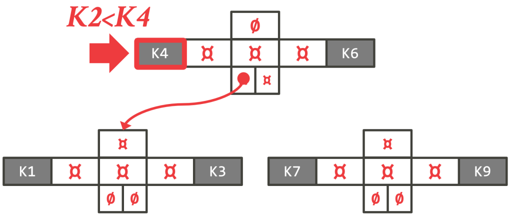
T-Tree: Find K2



Data Table

KEY	DATA
K1	-
K2	-
K3	-
K4	-
K5	-
K6	-
K7	-
K8	-
K9	-

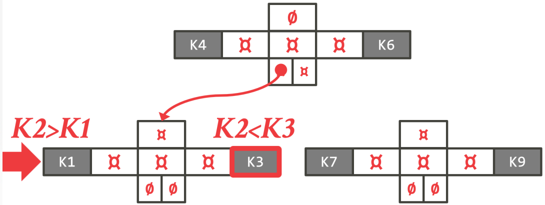
T-Tree: Find K2



Data Table

KEY	DATA
K1	-
K2	-
K3	-
K4	-
K5	-
K6	-
K7	-
K8	-
K9	-

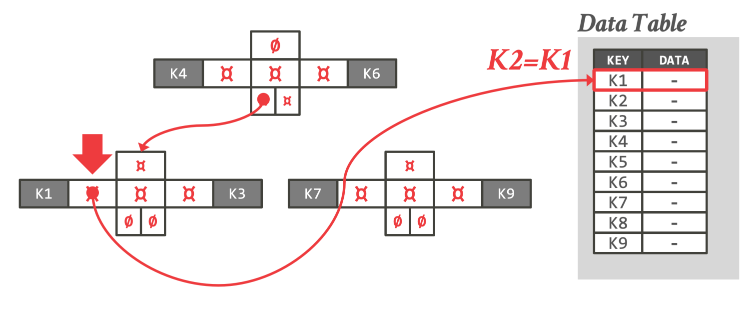
T-Tree: Find K2



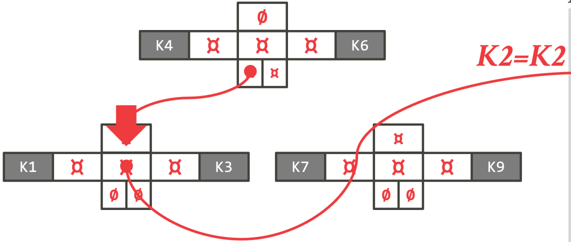
Data Table

KEY	DATA
K1	-
K2	-
K3	-
K4	-
K5	-
K6	-
K7	-
K8	-
K9	-

T-Tree: Find K2



T-Tree: Find K2



KEY	DATA
K1	-
K2	-
K3	-
K4	-
K5	-
K6	-
K7	-
K8	-
K9	-

T-Tree: Advantages

- Uses less memory because it does **not** store raw keys inside of each node.
- The DBMS evaluates all predicates on a table at the same time when accessing a tuple (*i.e.*, not just the predicates on indexed attributes).

T-Tree: Disadvantages

- Difficult to rebalance.
- Difficult to support safe concurrent access.
- Must chase pointers when scanning range or performing binary search inside of a node.
  - This greatly hurts **cache locality**.

Versioned Latch Coupling

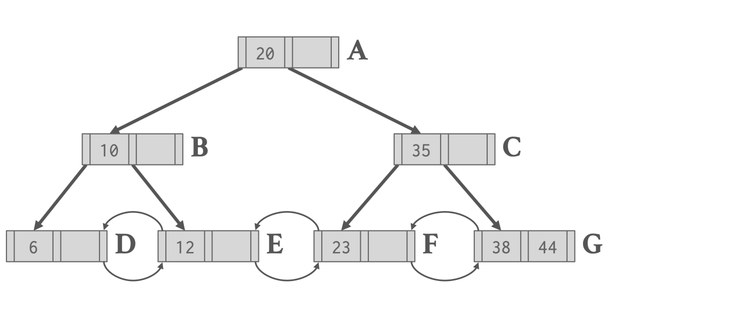
Latch Coupling

- Protocol to allow multiple threads to access/modify **B+Tree** at the same time.
- Basic Idea:
  - Get latch for parent.
  - Get latch for child
  - Release latch for parent if "safe".
- A **safe node** is one that will **not split or merge** when updated.
  - Not full (on insertion)
  - More than half-full (on deletion)

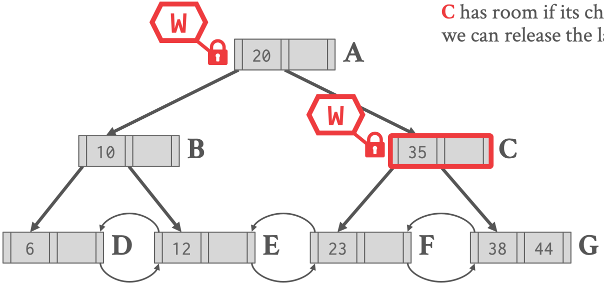
Latch Coupling

- **Find:** Start at root and go down; repeatedly,
  - Acquire read (**R**) latch on child
  - Then unlock the parent node.
- **Insert/Delete:** Start at root and go down, obtaining write (**W**) latches as needed. Once child is locked, check if it is safe:
  - If child is **safe**, release all locks on ancestors.

Latch Coupling: Insert 40

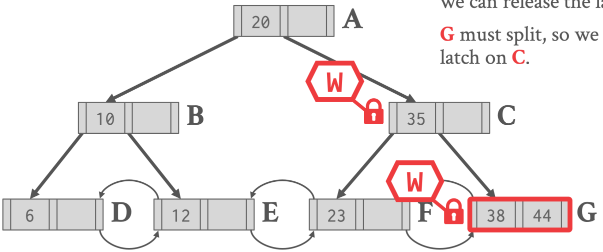


Latch Coupling: Insert 40



C has room if its child has to split, so we can release the latch on A.

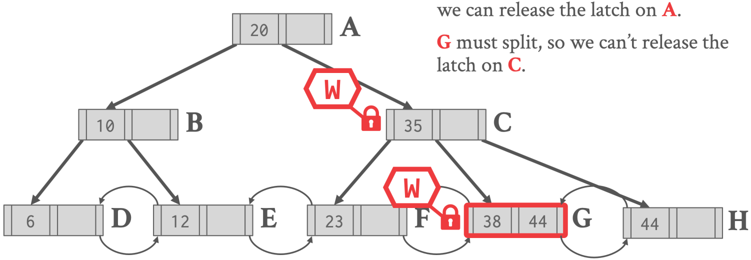
Latch Coupling: Insert 40



C has room if its child has to split, so we can release the latch on **A**.

G must split, so we can't release the latch on **C**.

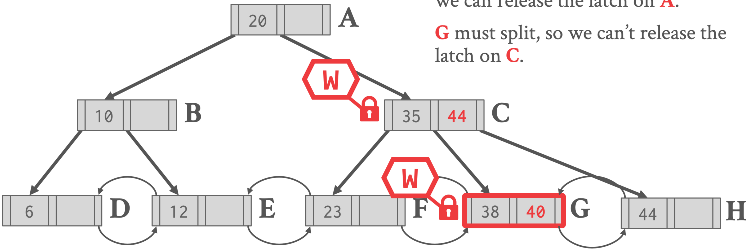
Latch Coupling: Insert 40



C has room if its child has to split, so we can release the latch on A.

G must split, so we can't release the latch on C.

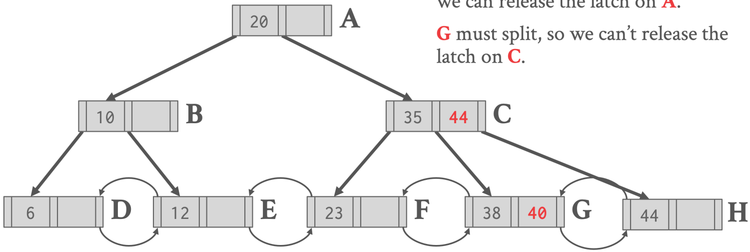
Latch Coupling: Insert 40



C has room if its child has to split, so we can release the latch on A.

G must split, so we can't release the latch on C.

Latch Coupling: Insert 40



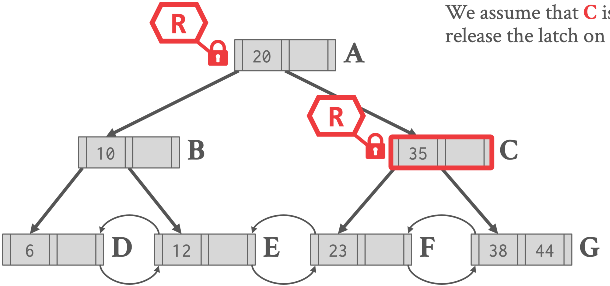
C has room if its child has to split, so we can release the latch on **A**.

G must split, so we can't release the latch on **C**.

Better Latch Coupling

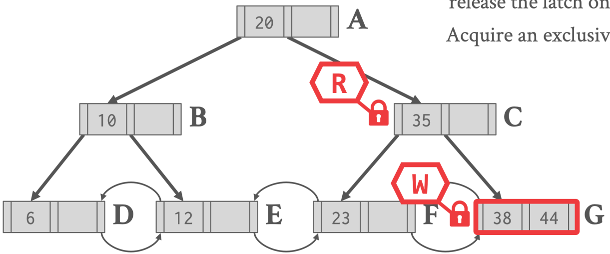
- The basic latch crabbing algorithm always takes a write latch on the root for any update.
  - This makes the index essentially single threaded.
- A better approach is to **optimistically** assume that the target leaf node is safe.
  - Take R latches as you traverse the tree to reach it and verify.
  - If leaf is not safe, then do previous algorithm.
- Reference

Better Latch Coupling: Delete 44



We assume that **C** is safe, so we can release the latch on **A**.

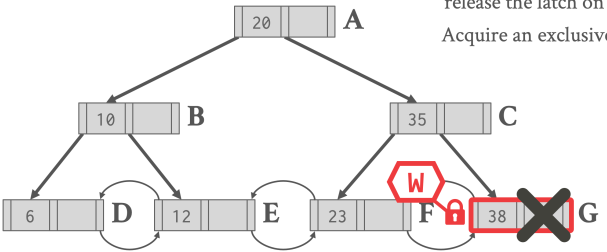
Better Latch Coupling: Delete 44



We assume that **C** is safe, so we can release the latch on **A**.

Acquire an exclusive latch on **G**.

Better Latch Coupling: Delete 44



We assume that **C** is safe, so we can release the latch on **A**.

Acquire an exclusive latch on **G**.

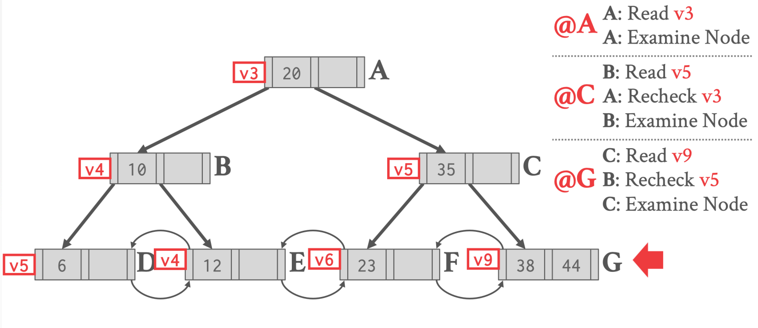
Versioned Latch Coupling

- Optimistic coupling scheme where writers are **not** blocked on readers.
- Provides the benefits of optimistic coupling without wasting too much work.
- Every latch has a **version counter**.
- Writers traverse down the tree like a reader
  - Acquire latch in target node to block other writers.
  - Increment version counter before releasing latch.
  - Writer thread increments version counter and acquires latch in a single **compare-and-swap** instruction.
- Reference

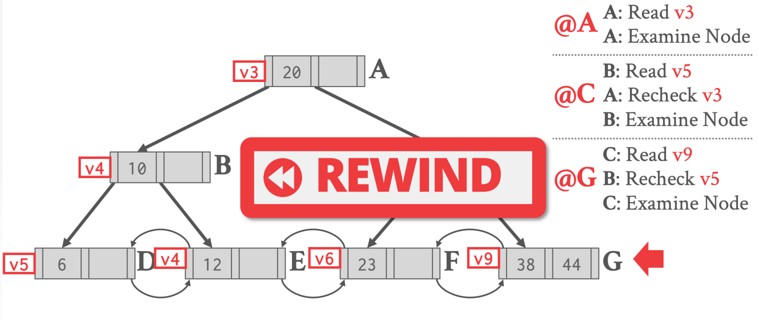
Versioned Latch Coupling

- Readers do not acquire latches.
- Readers traverse down the tree optimistically.
- Detect concurrent modifications by checking version counter.
- If version does not match, need to restart operation.
- May lead to **unnecessary aborts** if the node modification does not actually affect the reader thread.
- Rely on epoch-based **garbage collector** of **old nodes** to ensure node pointers are valid.

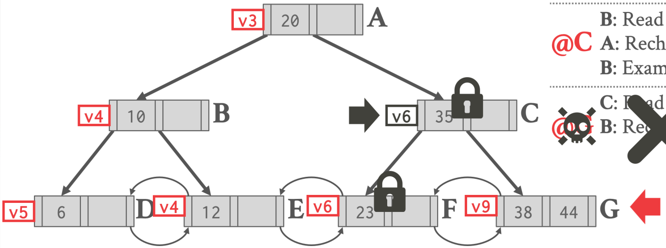
Versioned Latch Coupling: Find 44



Versioned Latch Coupling: Find 44



Versioned Latch Coupling: Find 44



@A
A: Read v3
A: Examine Node

@C
B: Read v5
A: Recheck v3
B: Examine Node

C: R ad v
B: Re ck v5

Test-and-Set (TAS)

- Takes one parameter: an **address**
- Sets the contents of the address to one, and returns the **old value**
- Used for implementing a spin latch
- Very efficient (single instruction to latch/unlatch)
- Example: std::atomic<T>

```
std::atomic_flag latch; // atomic of boolean type (lock-free)

while (latch.test_and_set(...)) {
^^I// Retry? Yield? Abort?
}
```

Compare-and-Swap (CAS)

- More **flexible** and **slower** than test-and-set instruction.
- Takes three parameters: an **address**, an **expected value** for that address, and a **new value** for the address
- Atomically compare the contents of the address to an **expected value** and swap in the **new value** if and only if the comparison is true.

Georgia Tech

Compare-and-Swap (CAS)

- Atomically compare the contents of the location to an **expected value** and swap in the **new value** if and only if the comparison is true.

```
std::atomic<int> ai;

int  tst_val= 4;
int  new_val= 5;
bool exchanged= false;

ai= 3;

// tst_val != ai   ==>  tst_val is modified
exchanged= ai.compare_exchange_strong( tst_val, new_val );

// tst_val == ai   ==>  ai is modified
exchanged= ai.compare_exchange_strong( tst_val, new_val );
```

Latch-Free Bw-Tree

Observation

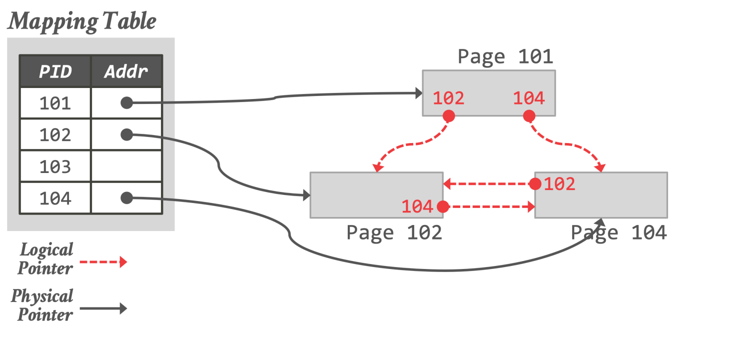
- Because CaS only updates a **single address at a time**, this limits the design of our data structures
- We cannot build a latch-free B+Tree because we need to update **multiple pointers** on node split/merge operations.
- What if we had an **indirection layer** that allowed us to update multiple addresses atomically?

Bw-Tree

- Latch-free B+Tree index built for the Microsoft Hekaton project.
- **Key Idea 1: Delta Updates**
  - No in-place updates.
  - Reduces cache invalidation.
- **Key Idea 2: Mapping Table**
  - Allows for CaS of physical locations of pages.
- Reference

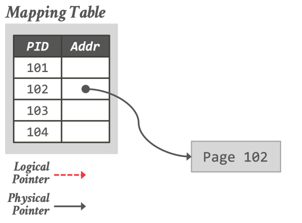
Georgia Tech

Bw-Tree: Mapping Table

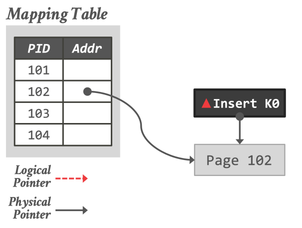


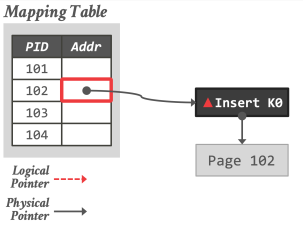
Bw-Tree: Delta Updates

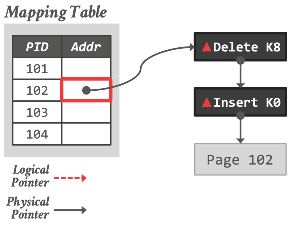
- Each update to a page produces a new **delta record**.
- Delta record physically points to base page.
- Install delta record's address in physical address slot of mapping table using CaS.



Bw-Tree: Delta Updates

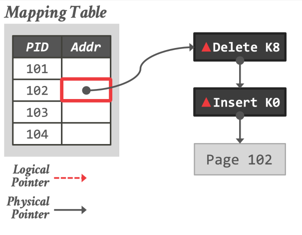






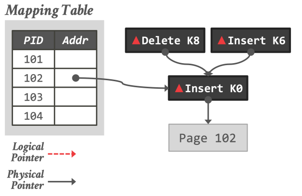
Bw-Tree: Find

- Traverse tree like a regular B+tree.
- If mapping table points to delta chain, stop at first occurrence of search key.
- Otherwise, perform binary search on base page.



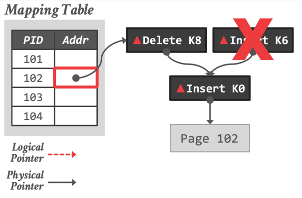
Bw-Tree: Conflicting Updates

- Threads may try to install updates to same page.
- Winner succeeds, any losers must retry or abort



Bw-Tree: Conflicting Updates

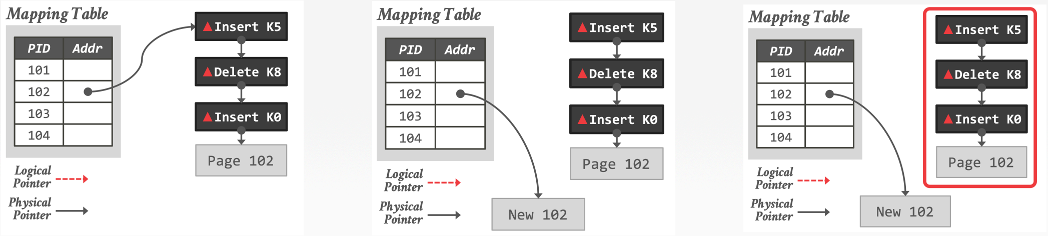
- Threads may try to install updates to same page.
- Winner succeeds, any losers must retry or abort



Bw-Tree: Node Consolidation

- Consolidate updates by creating new page with deltas applied.
- CaS-ing the mapping table address ensures no deltas are missed.
- Old page + deltas are marked as garbage.

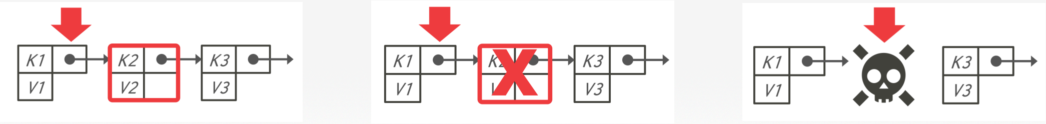
Bw-Tree: Node Consolidation



Garbage Collection

- We need to know when it is **safe** to reclaim memory for deleted nodes in a latch-free index.
- Approaches for thread-safe garbage collection:
  - Reference Counting
  - Epoch-based Reclamation
  - Hazard Pointers

Garbage Collection



Reference Counting

- Maintain a counter for each node to keep track of the number of threads that are accessing it.
  - Increment the counter before accessing.
  - Decrement it when finished.
  - A node is only safe to delete when the count is zero.
- This has bad performance for multi-core CPUs
  - Incrementing/decrementing counters causes a lot of **cache coherence traffic**.

Observation

- We don't care about the actual value of the reference counter. We only need to know when it reaches zero.
- We don't have to perform garbage collection immediately when the counter reaches zero.

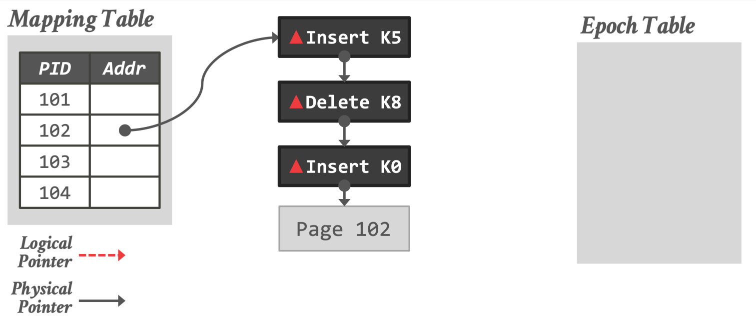
Epoch-based Garbage Collection

- Maintain a **global epoch counter** that is periodically updated (*e.g.*, every 10 ms).
  - Keep track of what threads enter the index during an epoch and when they leave.
- Mark the current epoch of a node when it is marked for deletion.
  - The node can be reclaimed once all threads have left that epoch (and all preceding epochs).
- *a.k.a.*, **Read-Copy-Update (RCU)** in Linux.

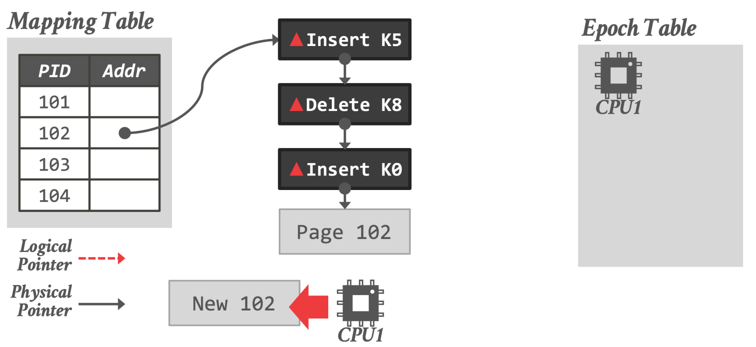
Bw-Tree: Epoch-based Garbage Collection

- Operations are tagged with an **epoch number**
- Each epoch tracks the threads that are part of it and the **objects** that can be reclaimed.
- Thread joins an epoch prior to each operation
- Garbage for an epoch reclaimed only when **all threads** have exited the epoch.

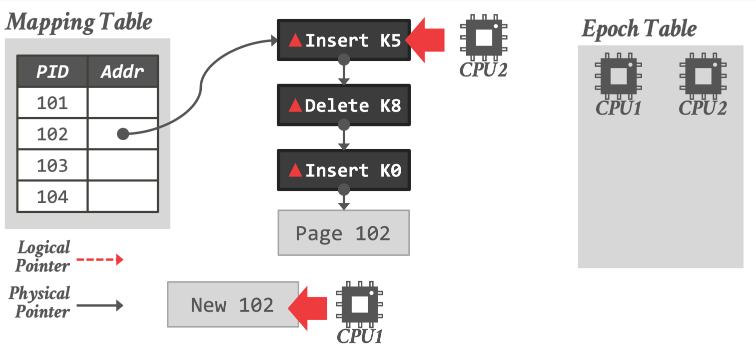
Bw-Tree: Epoch-based Garbage Collection



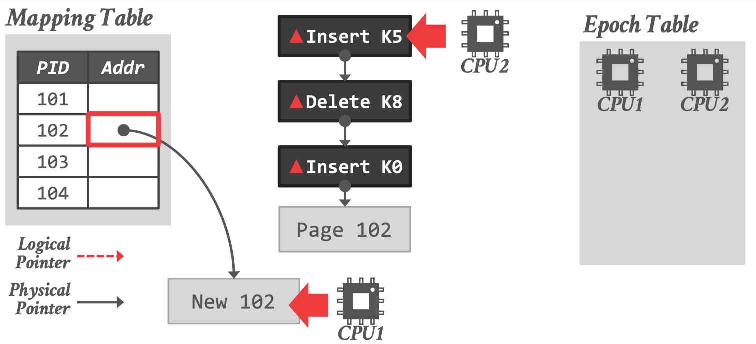
Bw-Tree: Epoch-based Garbage Collection



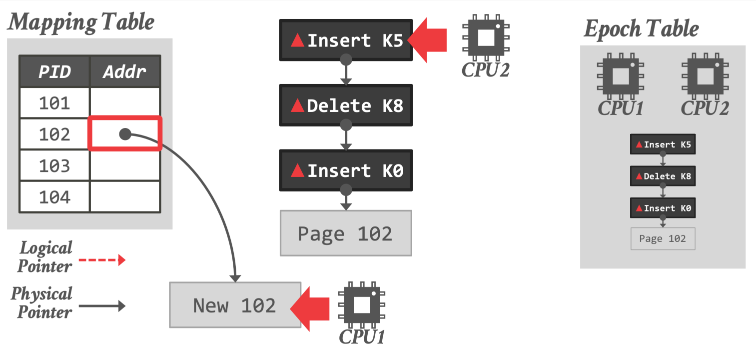
Bw-Tree: Epoch-based Garbage Collection



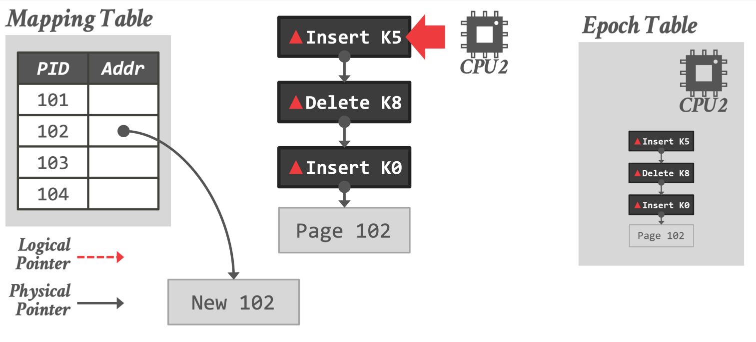
Bw-Tree: Epoch-based Garbage Collection



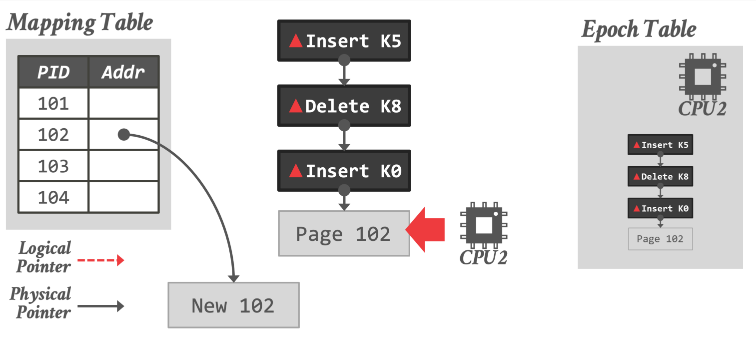
Bw-Tree: Epoch-based Garbage Collection



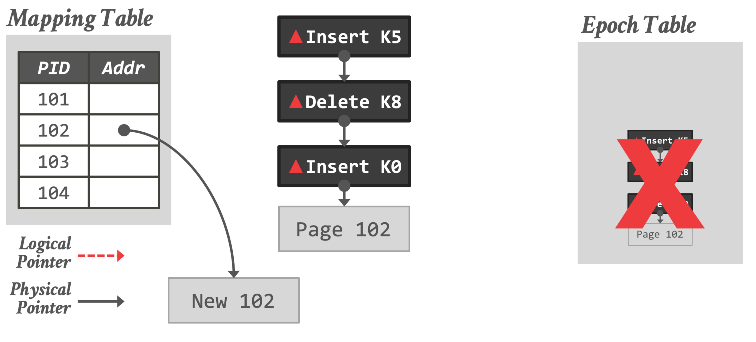
Bw-Tree: Epoch-based Garbage Collection



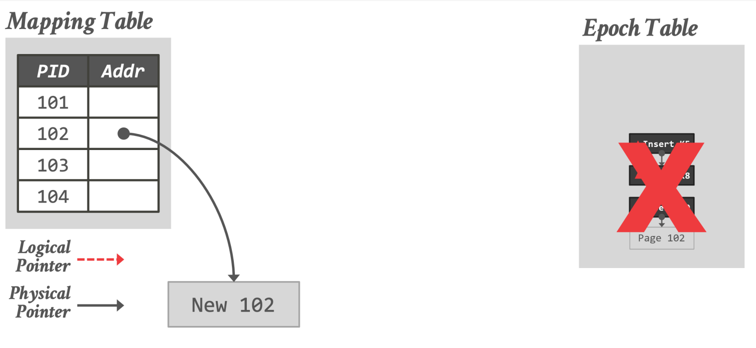
Bw-Tree: Epoch-based Garbage Collection



Bw-Tree: Epoch-based Garbage Collection



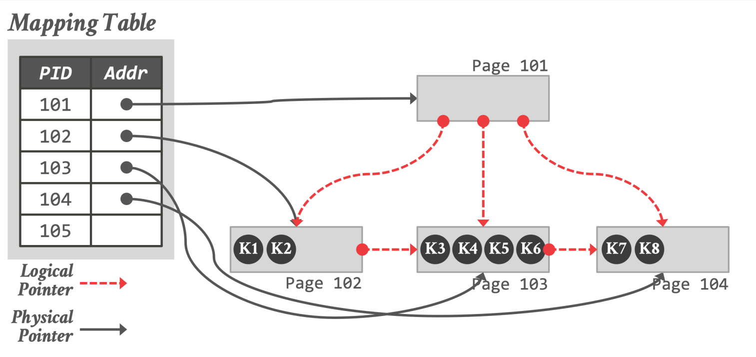
Bw-Tree: Epoch-based Garbage Collection



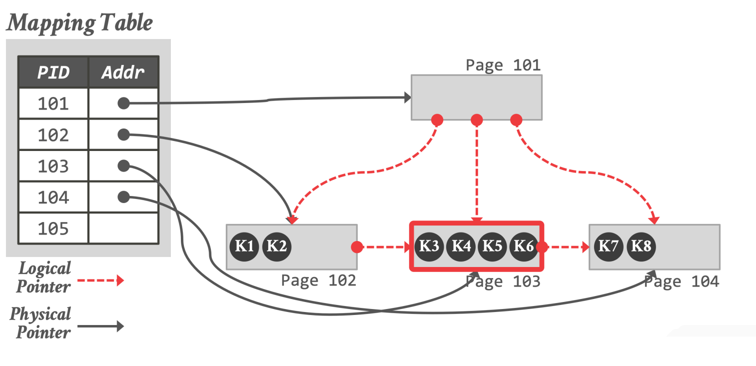
Bw-Tree: Structure Modification Operations

- **Split Delta Record**
  - Mark that a subset of the base page's key range is now located at another page.
  - Use a logical pointer to the new page.
- **Separator Delta Record**
  - Provide a shortcut in the modified page's parent on what ranges to find the new page.

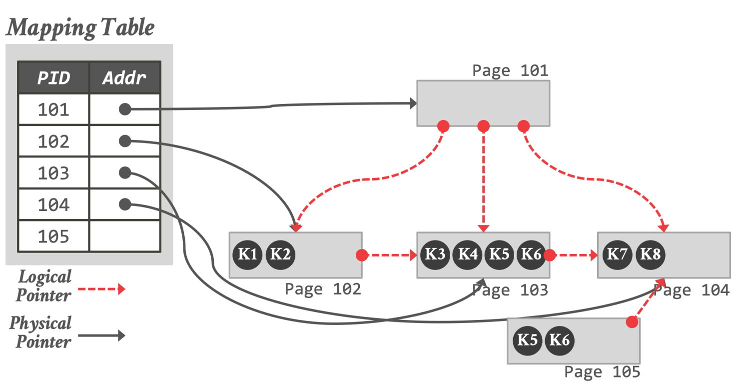
Bw-Tree: Structure Modification Operations



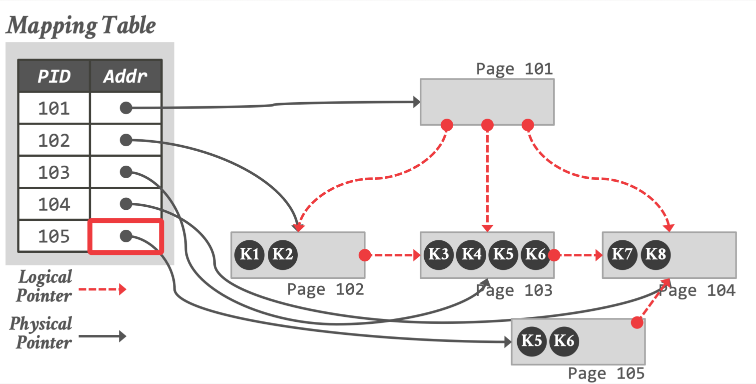
Bw-Tree: Structure Modification Operations



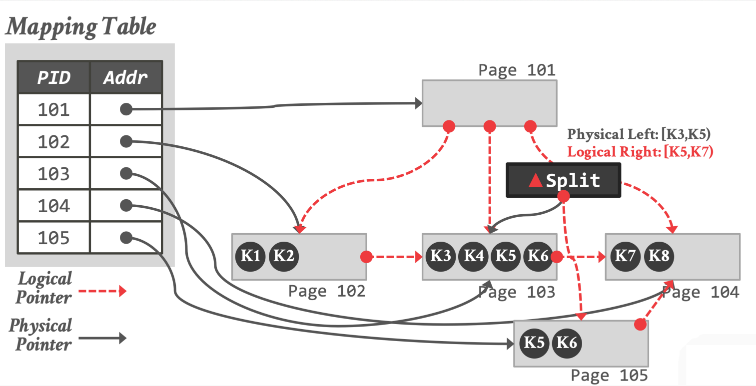
Bw-Tree: Structure Modification Operations



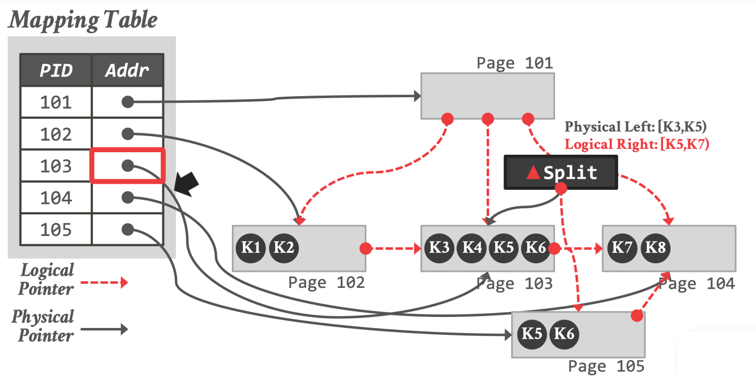
Bw-Tree: Structure Modification Operations



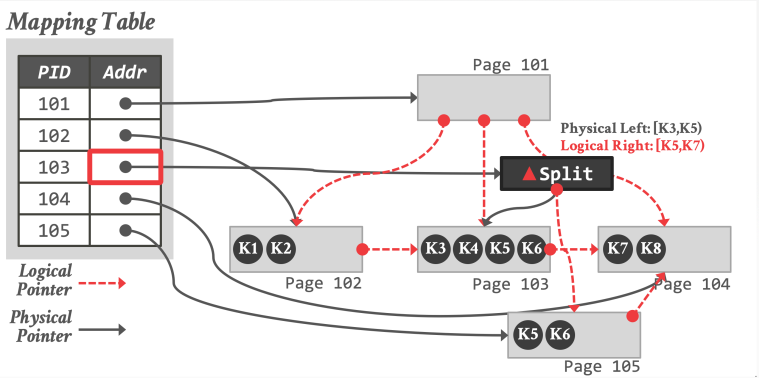
Bw-Tree: Structure Modification Operations



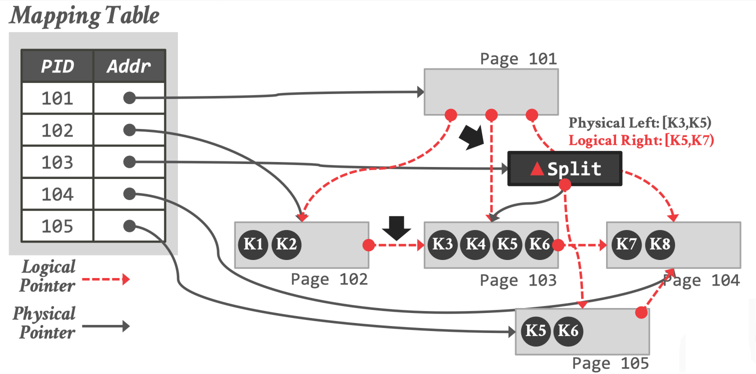
Bw-Tree: Structure Modification Operations



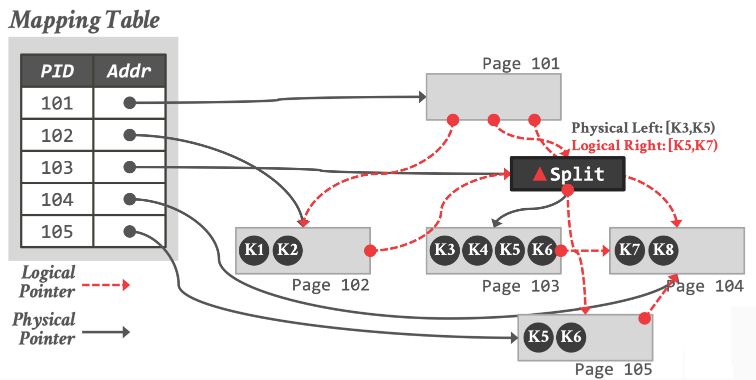
Bw-Tree: Structure Modification Operations



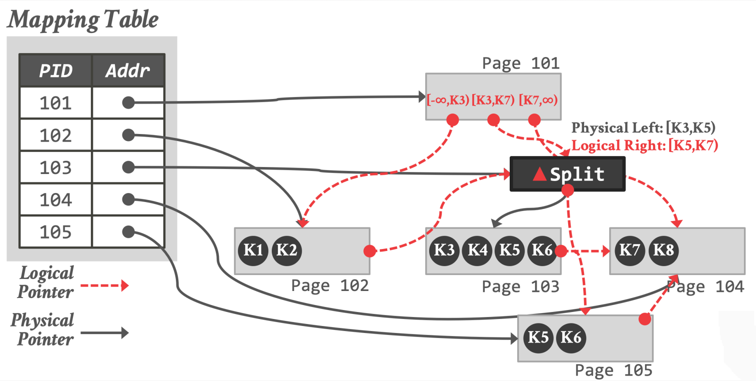
Bw-Tree: Structure Modification Operations



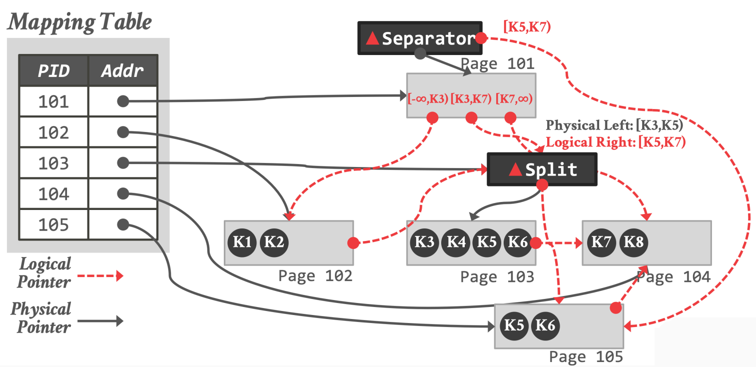
Bw-Tree: Structure Modification Operations



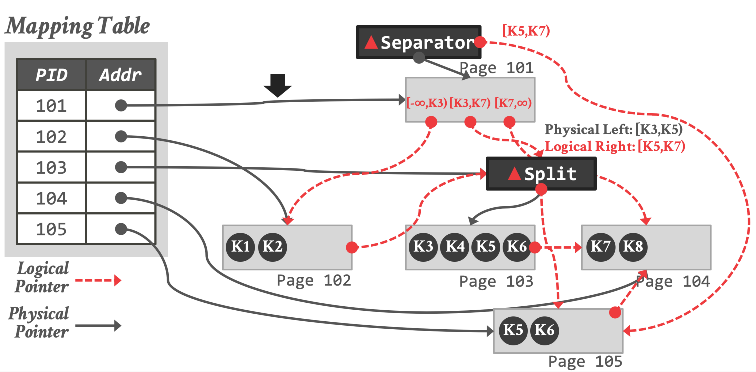
Bw-Tree: Structure Modification Operations



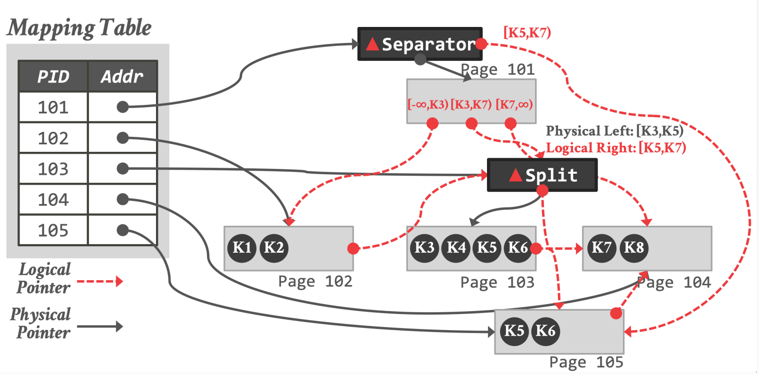
Bw-Tree: Structure Modification Operations



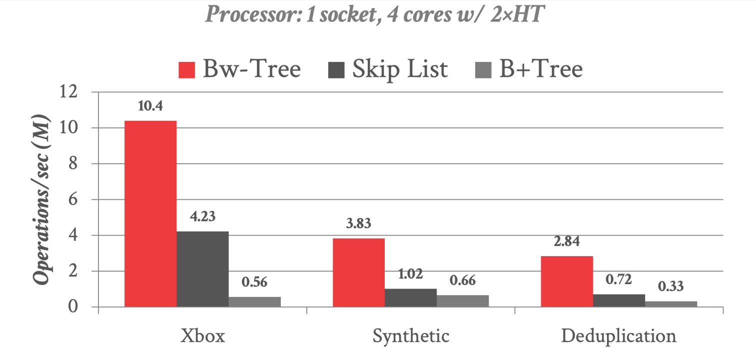
Bw-Tree: Structure Modification Operations



Bw-Tree: Structure Modification Operations

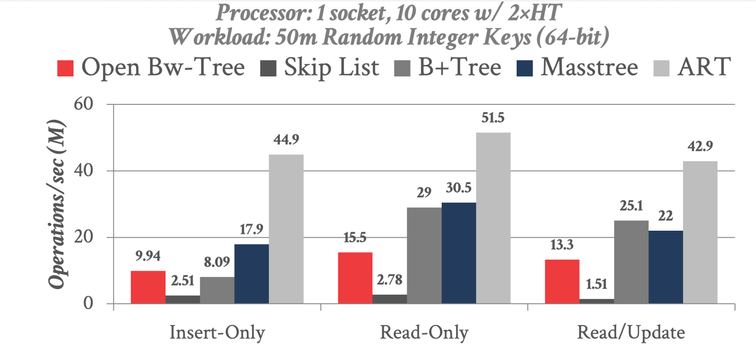


Bw-Tree: Performance



Source

Bw-Tree: Performance



Source

Conclusion

Conclusion

- Managing a concurrent index looks a lot like managing a database.
- Versioning and garbage collection are widely used mechanisms for increasing concurrency.
- BwTree illustrates how to design complex, latch-free data structures with only CaS instruction.