



# Modern OLTP Indexes (Part 1)

(1) Feedback — Lab 2

(2) Plagiarism ✓ Sheet 2

Recap

(3) Material ✓ Sheet 3

— Lecture 16

Index Structure

## 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? string

# Today's Agenda

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



# T-Tree

## Observation

PM DB MS

□ PM

↳ DS

- 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

DRAM



# 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

DRAM

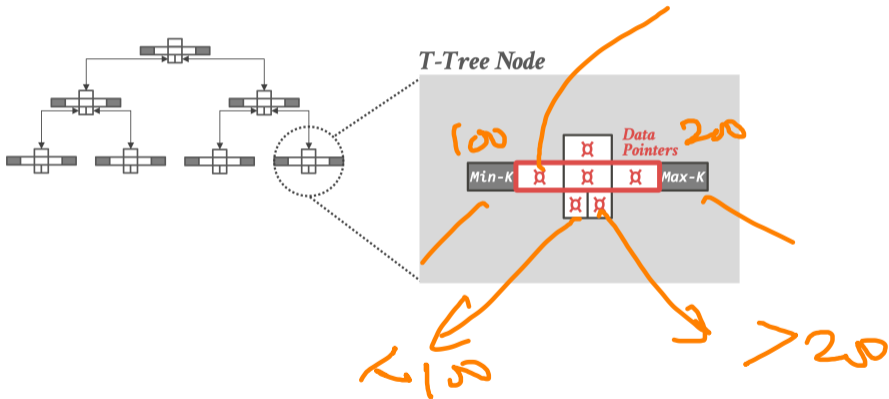
DRAM

- Instead of storing keys in nodes, store pointers to the tuples (*i.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.

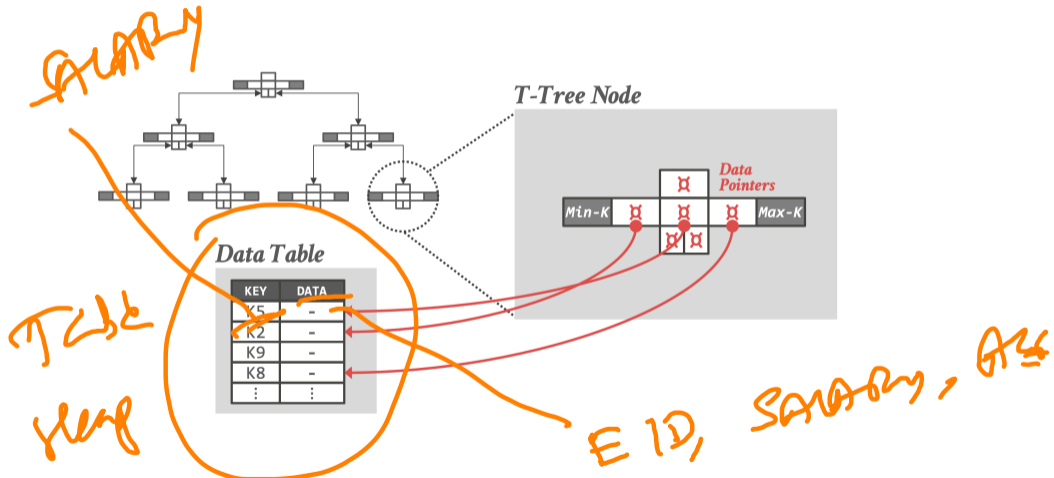
pointers chaining

## T-Tree

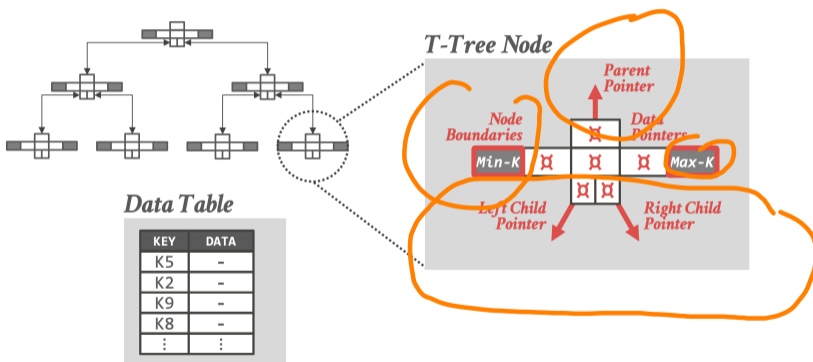
Index: SALARY



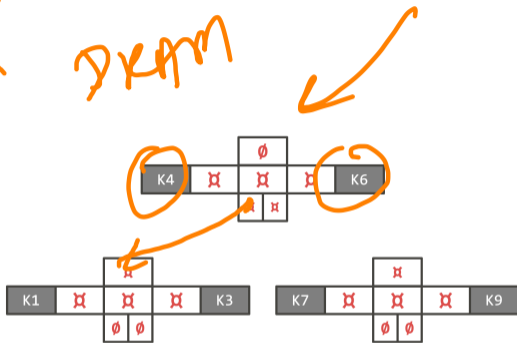
# T-Tree



# T-Tree



## T-Tree: Find K2



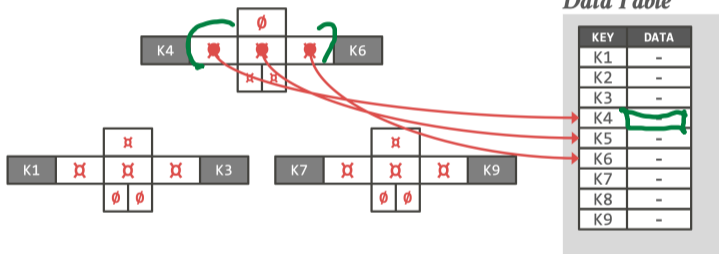
Data Table

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

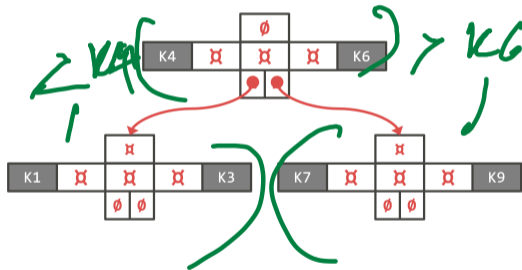


# T-Tree: Find K2

\* triple - ptr



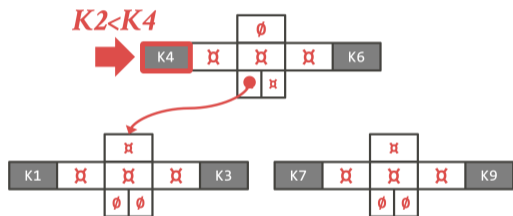
## T-Tree: Find K2



Data Table

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

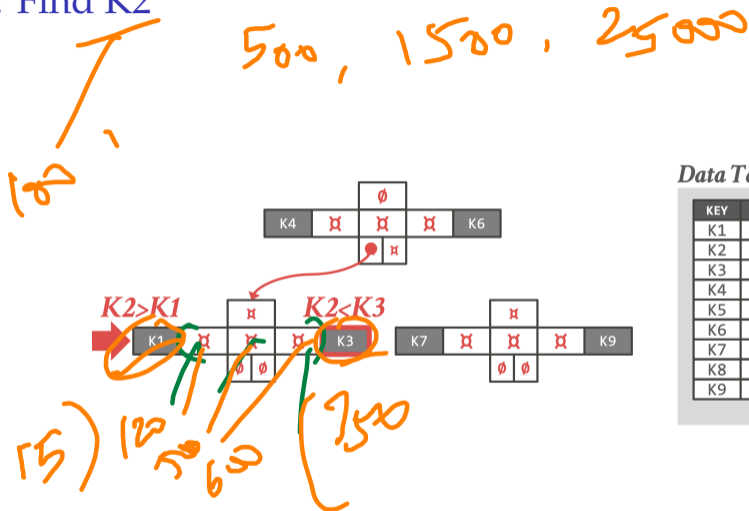
## T-Tree: Find K2



Data Table

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

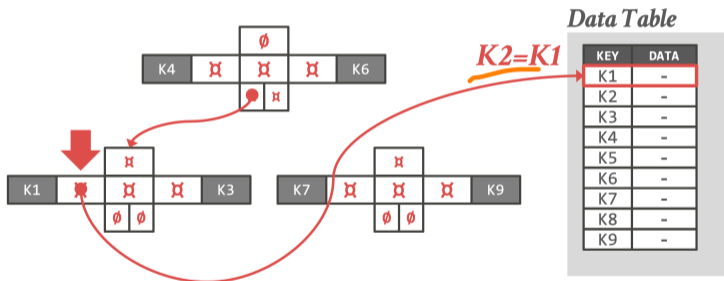
# T-Tree: Find K2



Data Table

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

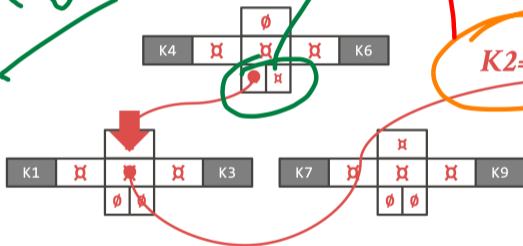
## T-Tree: Find K2



# T-Tree: Find K2

(i) block  
(ii) keys

K Capacity  
256 pointers



Data Table

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

K2=K2

same  
A2E

## 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).

SALARY > 100 & AGE > 75

## 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.

*CPU cache misses*



# Versioned Latch Coupling

B+ Trees

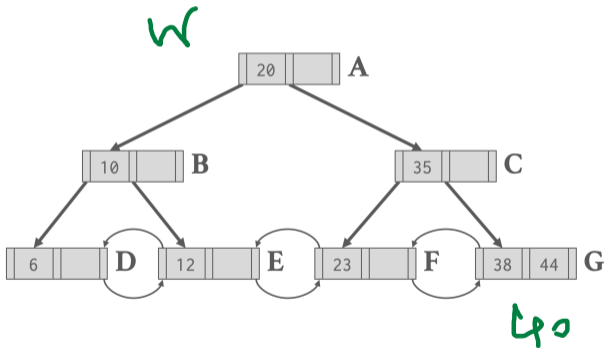
# Latch Coupling

- 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)

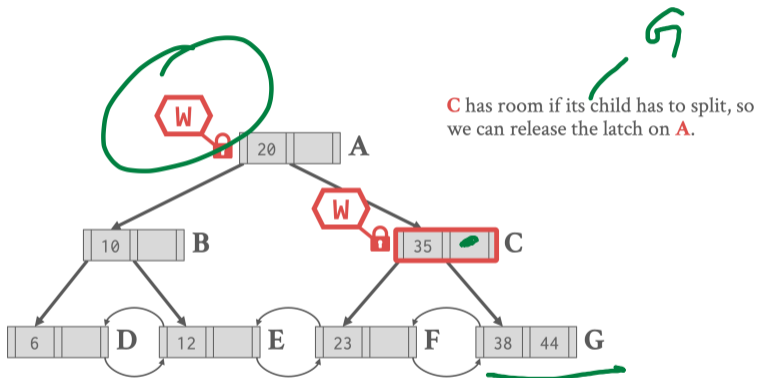
# Latch Coupling

- **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.

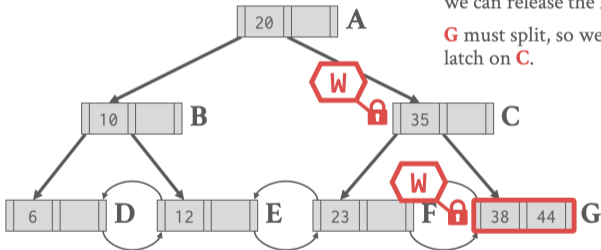
# Latch Coupling: Insert 40



# Latch Coupling: Insert 40



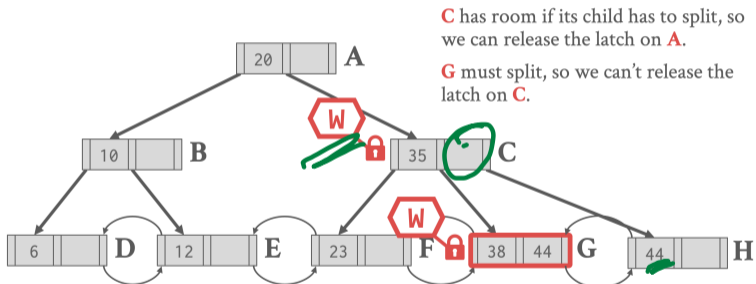
# Latch Coupling: Insert 40



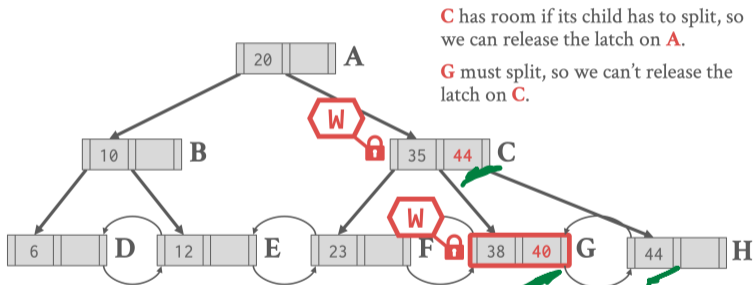
**C** has room if its child has to split, so we can release the latch on **A**.

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

# Latch Coupling: Insert 40

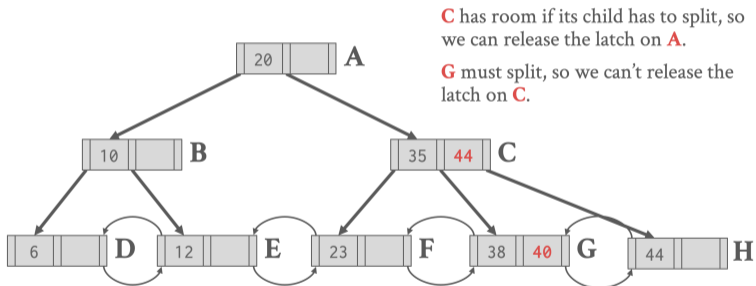


# Latch Coupling: Insert 40





# Latch Coupling: Insert 40



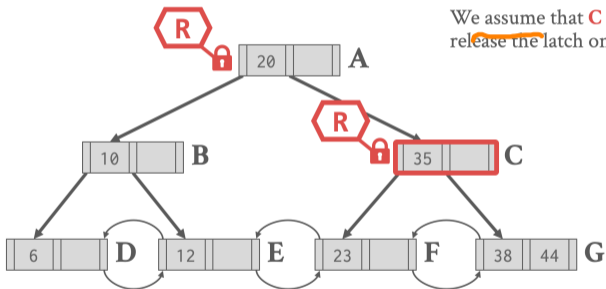
## Better Latch Coupling

- 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

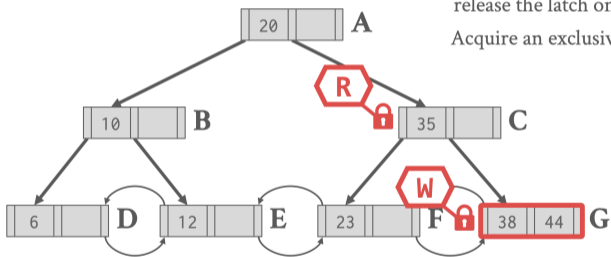
latches  
physical

Logical  
Locks  
Index

# Better Latch Coupling: Delete 44



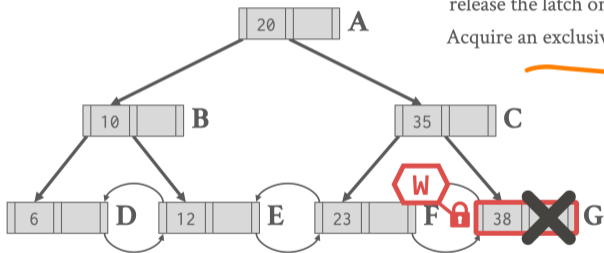
# Better Latch Coupling: Delete 44



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

Versioning

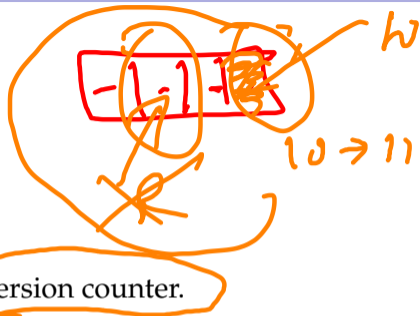
- Optimistic coupling scheme where writers are not blocked on readers.
- Provides the benefits of optimistic coupling without wasting too much work.
- Every latch has a version counter.
- Writers traverse down the tree like a reader
  - ▶ Acquire latch in target node to block other writers.
  - ▶ Increment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

• Reference

Modern Practical Synchronization

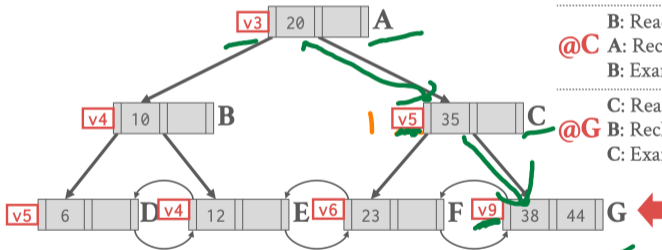
## Versioned Latch Coupling

- 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.



node

# Versioned Latch Coupling: Find 44



- @A A: Read v3  
A: Examine Node

---

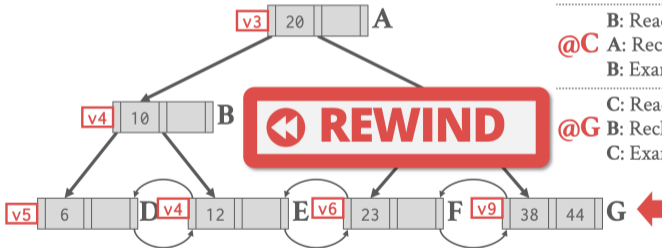
- B: Read v5
- @C A: Recheck v3  
B: Examine Node

---

- C: Read v9
- @G B: Recheck v5  
C: Examine Node



# Versioned Latch Coupling: Find 44



- @A A: Read v3  
A: Examine Node

---

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

---

- C: Read v9
- @G B: Recheck v5  
C: Examine Node



## Test-and-Set (TAS)

Spin latch

- 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(...)) {  
    // Retry? Yield? Abort?  
}
```

1 bit

## Compare-and-Swap (CAS)

instruction

32 bit

- 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.

## 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

3

## Observation

sync. primitive

- 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?

3 locations  
 $2^3$  states

# 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

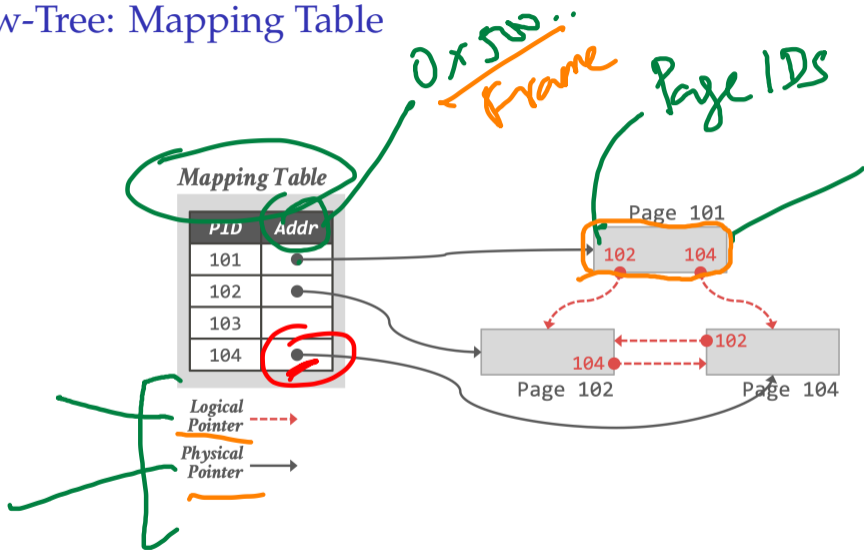
Sikarin

CaS behavior

CAS



## Bw-Tree: Mapping Table



## Bw-Tree: Delta Updates

*immutable*

- 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.

Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer ----->

Physical  
Pointer ----->

Page 102

# Bw-Tree: Delta Updates

Bw-Tree

CAS (Compare-And-Swap) (K<sub>MT</sub>[3])

Page 102's address, Insert Delta Record

Mapping Table

PID	Addr
101	
102	
103	
104	

Logical Pointer →  
Physical Pointer →

▲Insert K0

Page 102

immutable  
↓ use page

Mapping Table

PID	Addr
101	
102	
103	
104	

Logical Pointer →  
Physical Pointer →

▲Insert K0

Page 102

Mapping Table

PID	Addr
101	
102	
103	
104	

Logical Pointer →  
Physical Pointer →

▲Delete K8

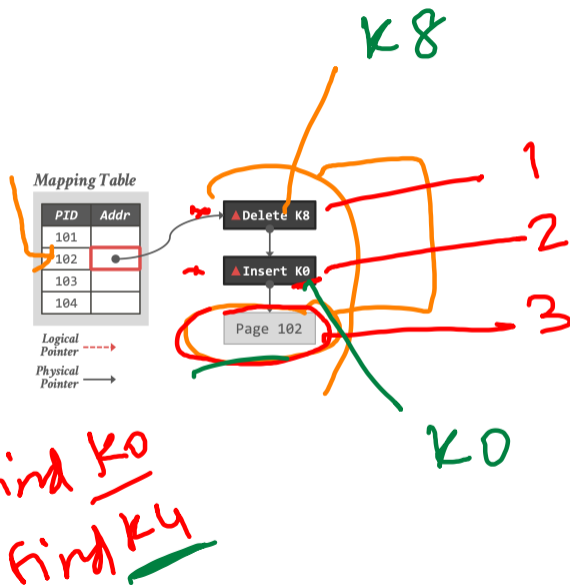
▲Insert K0

Page 102

Delta Records

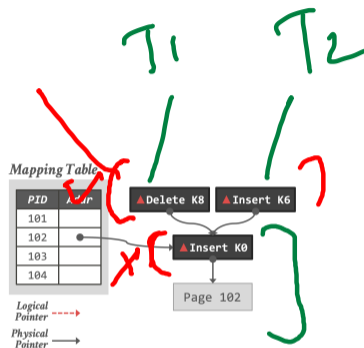
## Bw-Tree: Find

- 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



CAS

expected value

## Bw-Tree: Conflicting Updates

CAS: sync. primitive

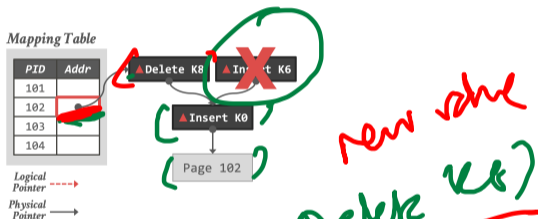
- Threads may try to install updates to same page.
- Winner succeeds, any losers must retry or abort

Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer →

Physical  
Pointer →



✓ T1: CAS (old val, Insert K0, Delete K8)

✗ T2: CAS (old val, Insert K0, Insert K6)

## 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

*pointer changing  
caching locality*

Mapping Table

PID	Addr
101	
102	
103	
104	

Logical Pointer →  
Physical Pointer →



Mapping Table

PID	Addr
101	
102	
103	
104	

Logical Pointer →  
Physical Pointer →



Mapping Table

PID	Addr
101	
102	
103	
104	

Logical Pointer →  
Physical Pointer →



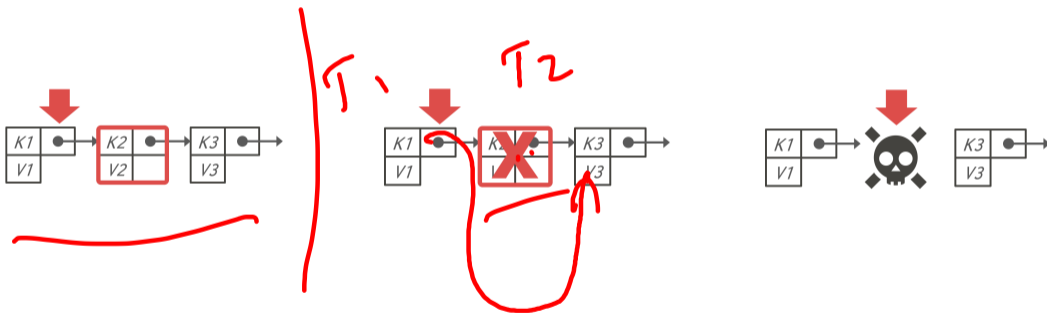


# Garbage Collection

Verbisig

- 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

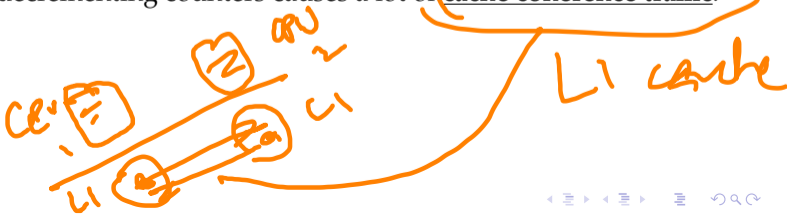
# Garbage Collection



# Reference Counting

Ref. Counter  
Buffer Pool Latches

- 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

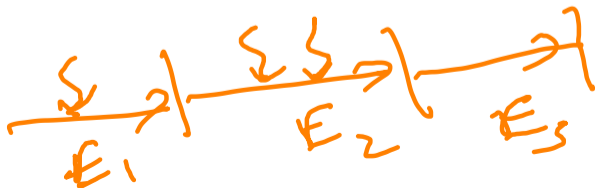
0 / not 0

- 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.

lazy

# 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

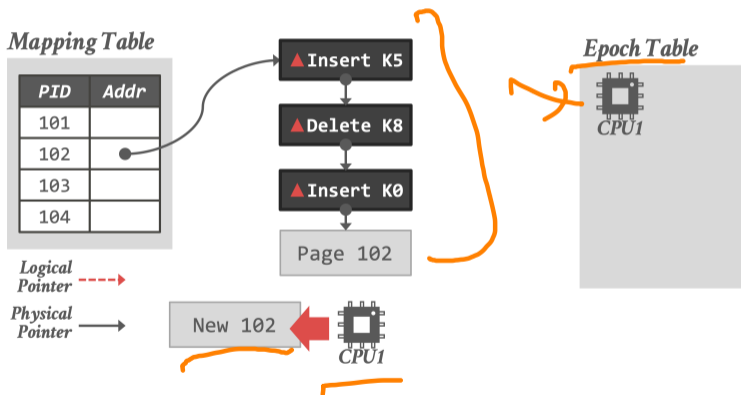
- Operations are tagged with an epoch number
- Each epoch tracks the threads that are part of it and the objects that can be reclaimed.
- Thread joins an epoch prior to each operation
- Garbage for an epoch reclaimed only when all threads have exited the epoch.

Main Idea

# Bw-Tree: Epoch-based Garbage Collection



# Bw-Tree: Epoch-based Garbage Collection





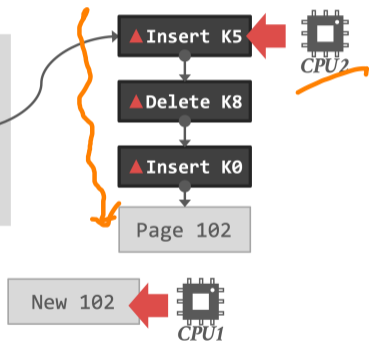
# Bw-Tree: Epoch-based Garbage Collection

*Mapping Table*

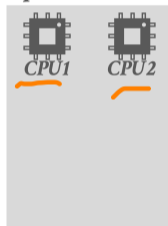
PID	Addr
101	
102	●
103	
104	

*Logical Pointer* →

*Physical Pointer* →



*Epoch Table*



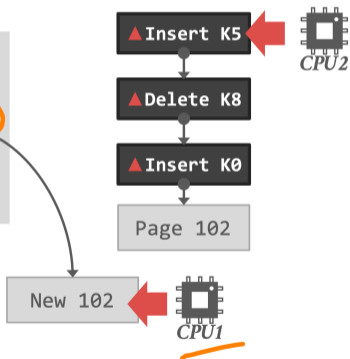
# Bw-Tree: Epoch-based Garbage Collection

*Mapping Table*

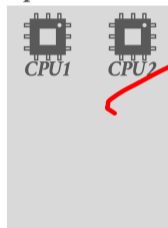
PID	Addr
101	
102	
103	
104	

*Logical Pointer* →

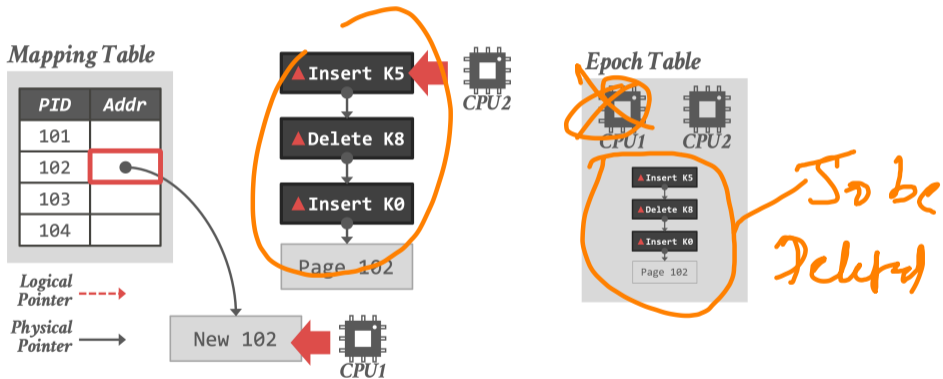
*Physical Pointer* →



*Epoch Table*



# Bw-Tree: Epoch-based Garbage Collection



# Bw-Tree: Epoch-based Garbage Collection

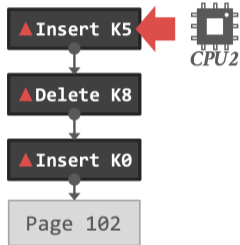
*Mapping Table*

PID	Addr
101	
102	●
103	
104	

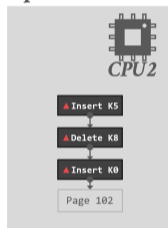
*Logical Pointer* →

*Physical Pointer* →

New 102



*Epoch Table*



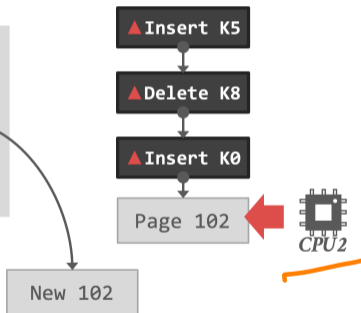
# Bw-Tree: Epoch-based Garbage Collection

*Mapping Table*

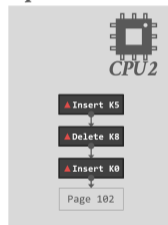
PID	Addr
101	
102	●
103	
104	

*Logical Pointer* →

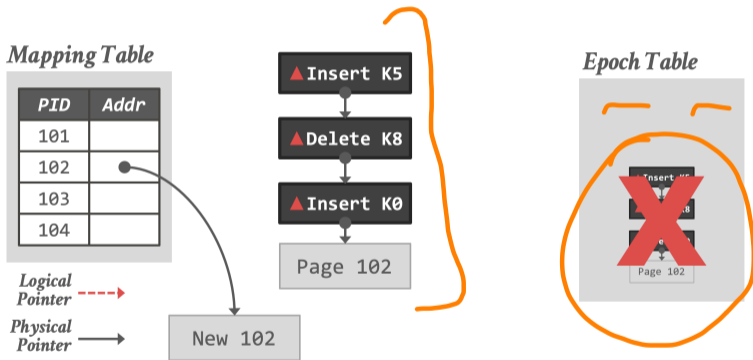
*Physical Pointer* →



*Epoch Table*

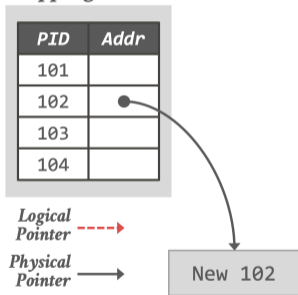


# Bw-Tree: Epoch-based Garbage Collection



# Bw-Tree: Epoch-based Garbage Collection

*Mapping Table*



*Epoch Table*



## Bw-Tree: Structure Modification Operations

Node Split SWCAS

- 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

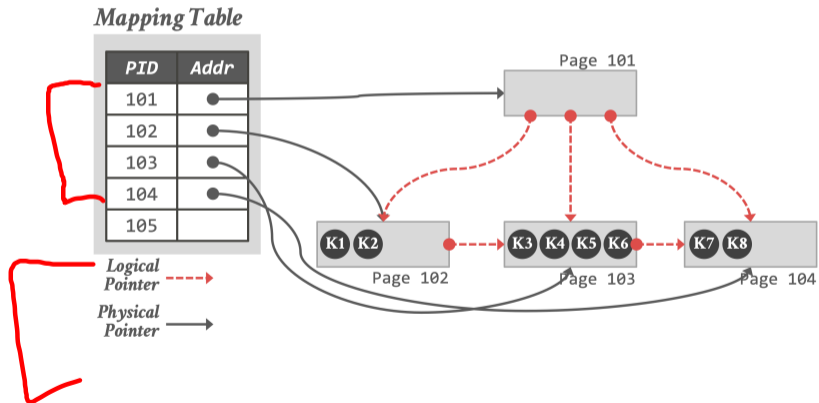
- ▶ Provide a shortcut in the modified page's parent on what ranges to find the new page.

Separator key



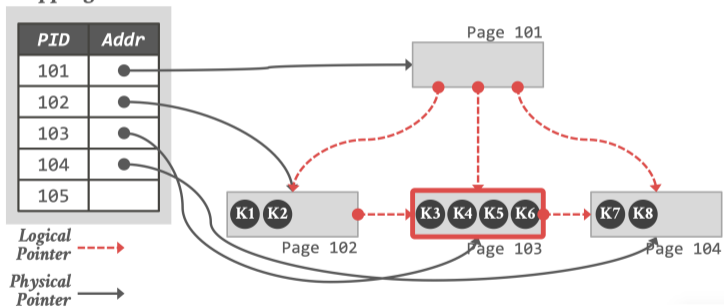


# Bw-Tree: Structure Modification Operations



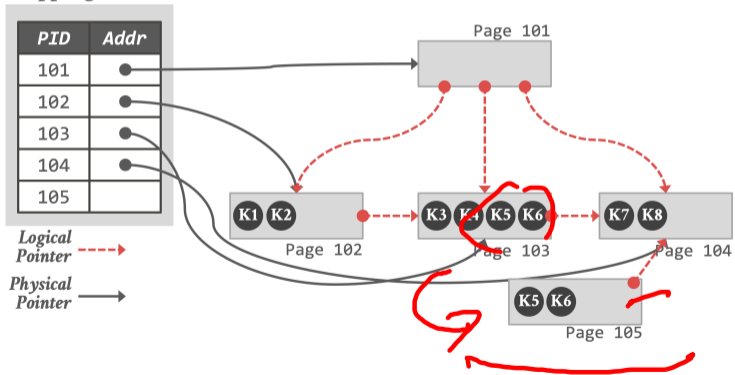
# Bw-Tree: Structure Modification Operations

*Mapping Table*

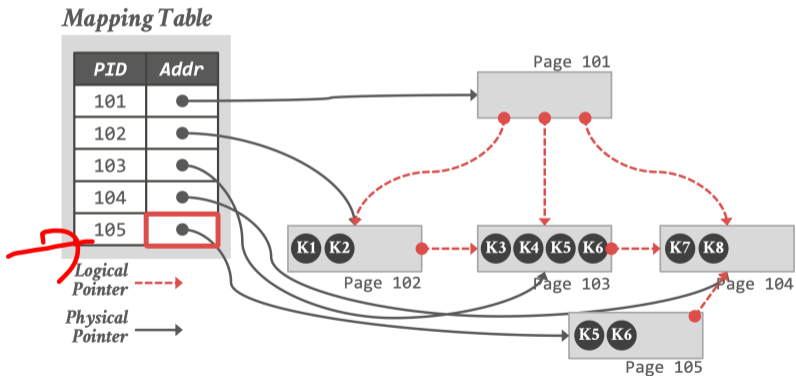


# Bw-Tree: Structure Modification Operations

*Mapping Table*

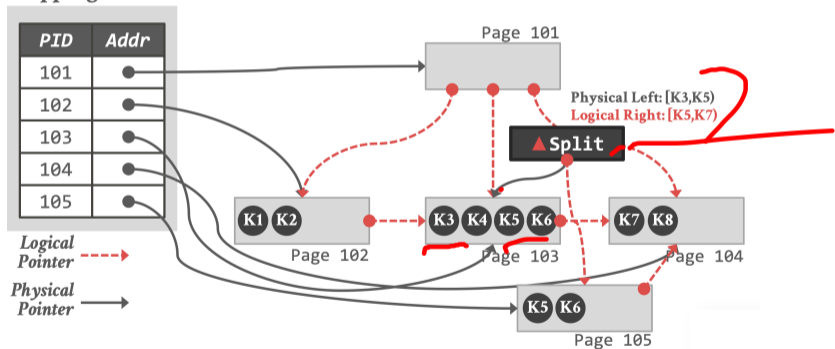


# Bw-Tree: Structure Modification Operations



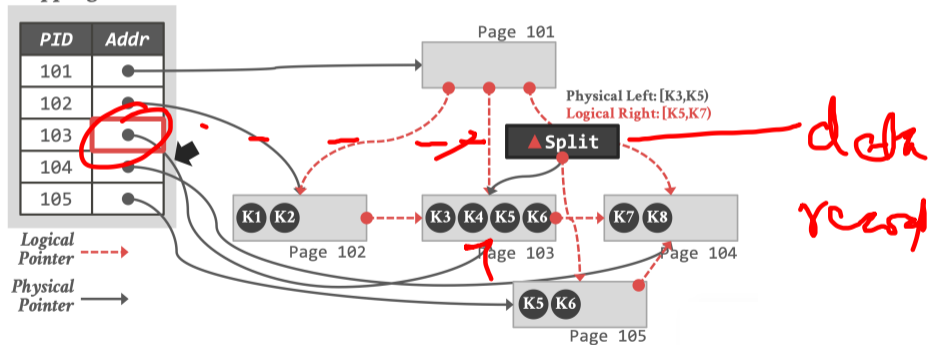
# Bw-Tree: Structure Modification Operations

*Mapping Table*



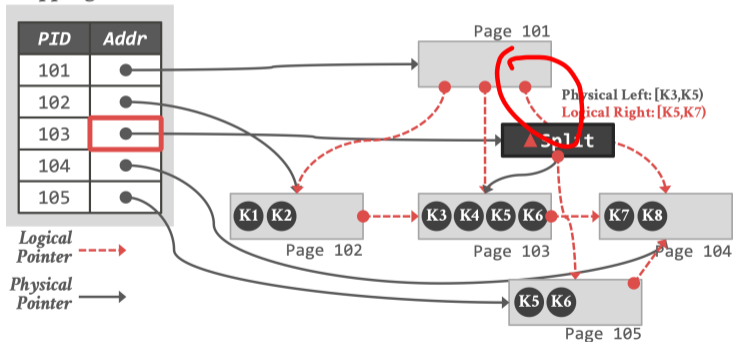
# Bw-Tree: Structure Modification Operations

*Mapping Table*



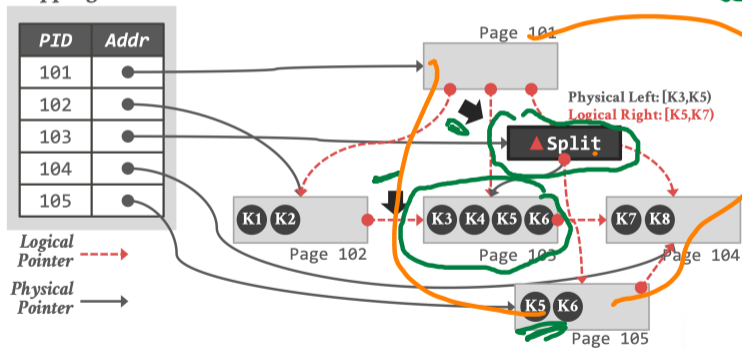
# Bw-Tree: Structure Modification Operations

*Mapping Table*



# Bw-Tree: Structure Modification Operations

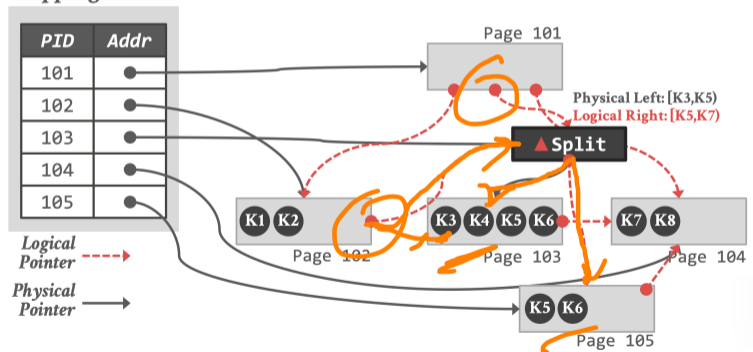
*Mapping Table*





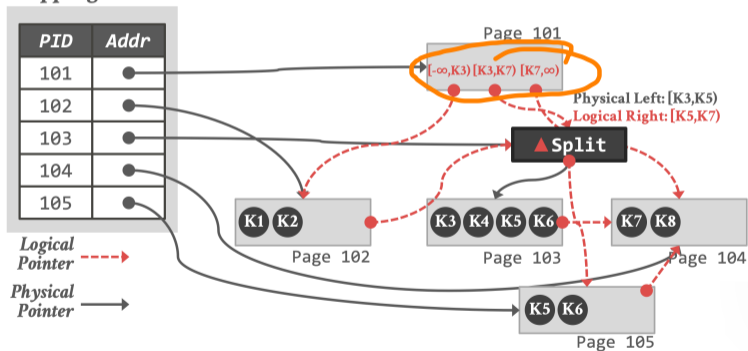
# Bw-Tree: Structure Modification Operations

*Mapping Table*

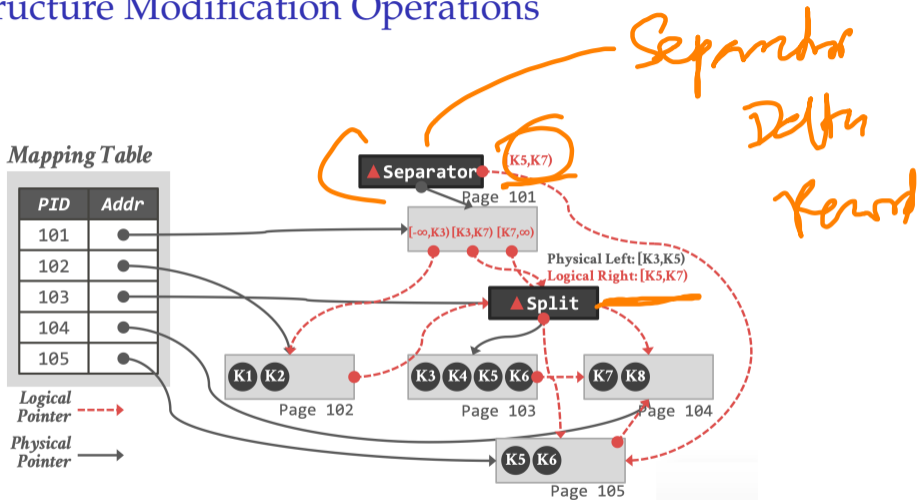


# Bw-Tree: Structure Modification Operations

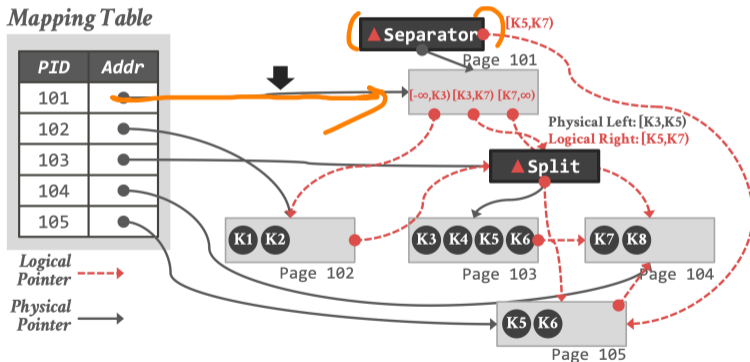
*Mapping Table*



## Bw-Tree: Structure Modification Operations



# Bw-Tree: Structure Modification Operations



# Bw-Tree: Structure Modification Operations

Hexagon

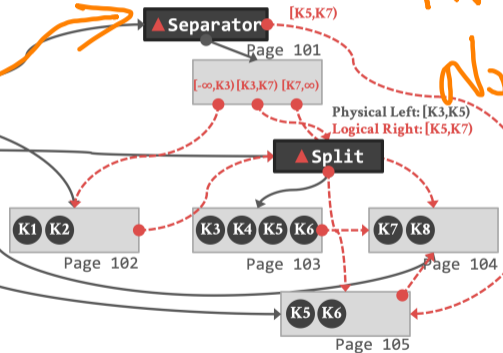
SMO

~~Whole Consistent~~  
Node Split  
Node merge

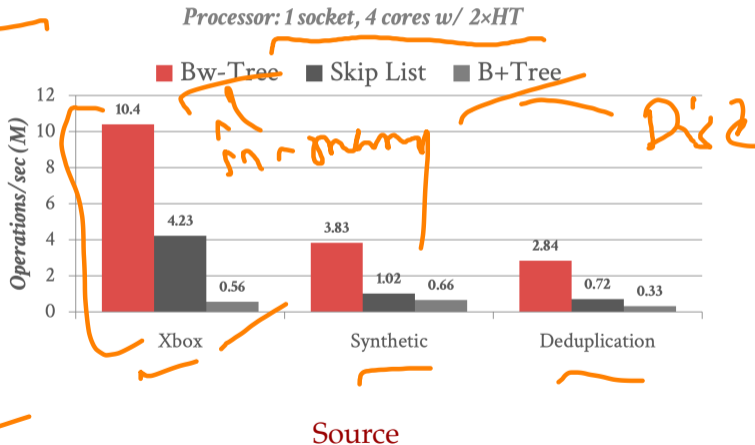
Mapping Table

PID	Addr
101	•
102	•
103	•
104	•
105	•

Logical Pointer  $\dashrightarrow$   
Physical Pointer  $\rightarrow$



## Bw-Tree: Performance



## 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.
- Next Class
  - ▶ We will learn about modern trie data structures.

single location  
at a time