Lecture #06

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

OLTP Indexes (Whole-Key Data Structures)

@Andy_Pavlo // 15-721 // Spring 2020

UPCOMING DATABASE EVENTS

Snowflake Optimizer Talk

→ Monday Feb 3rd @ 4:30pm

→ GHC 9115

WHOLE-KEY DATA STRUCTURE

A "whole-key" order preserving data structure stores all the digits of a key together in nodes.

→ A worker thread has to compare the entire search key with keys in the data structure during traversal.

We will discuss "partial-key" data structures (i.e., tries) next class.

TODAY'S AGENDA

In-Memory T-Tree

Latch-Free Bw-Tree

B+Tree Optimistic Latching

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?

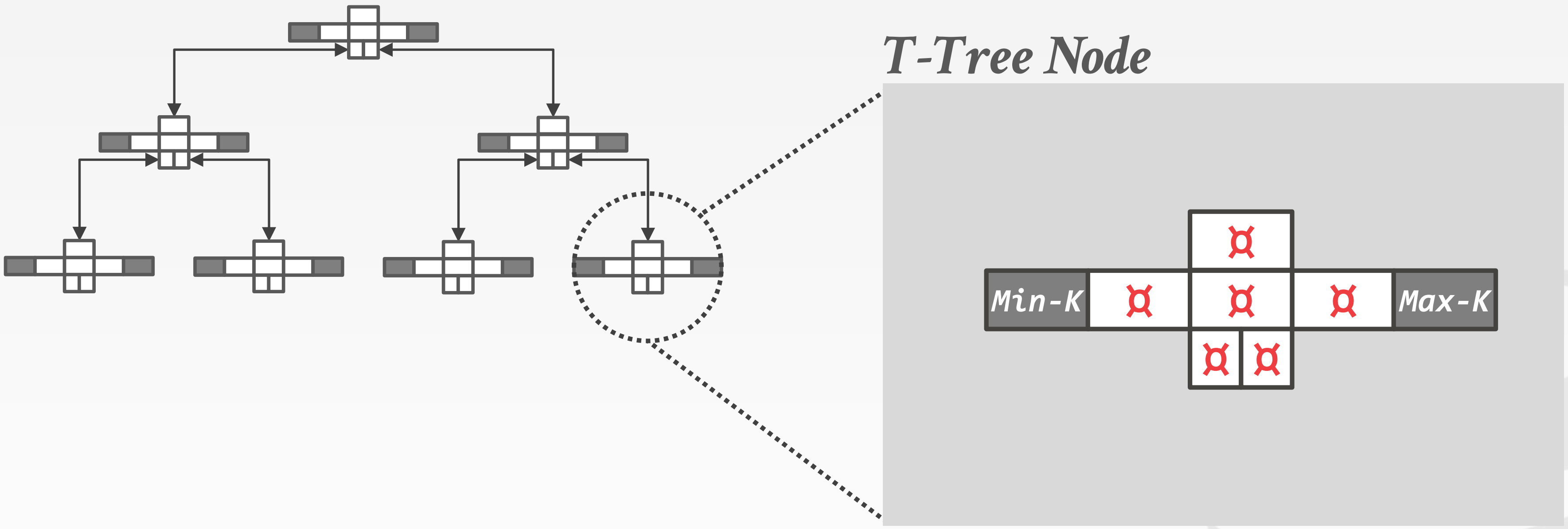
T-TREES

Based on AVL Trees. Instead of storing keys in nodes, store pointers to their original values.

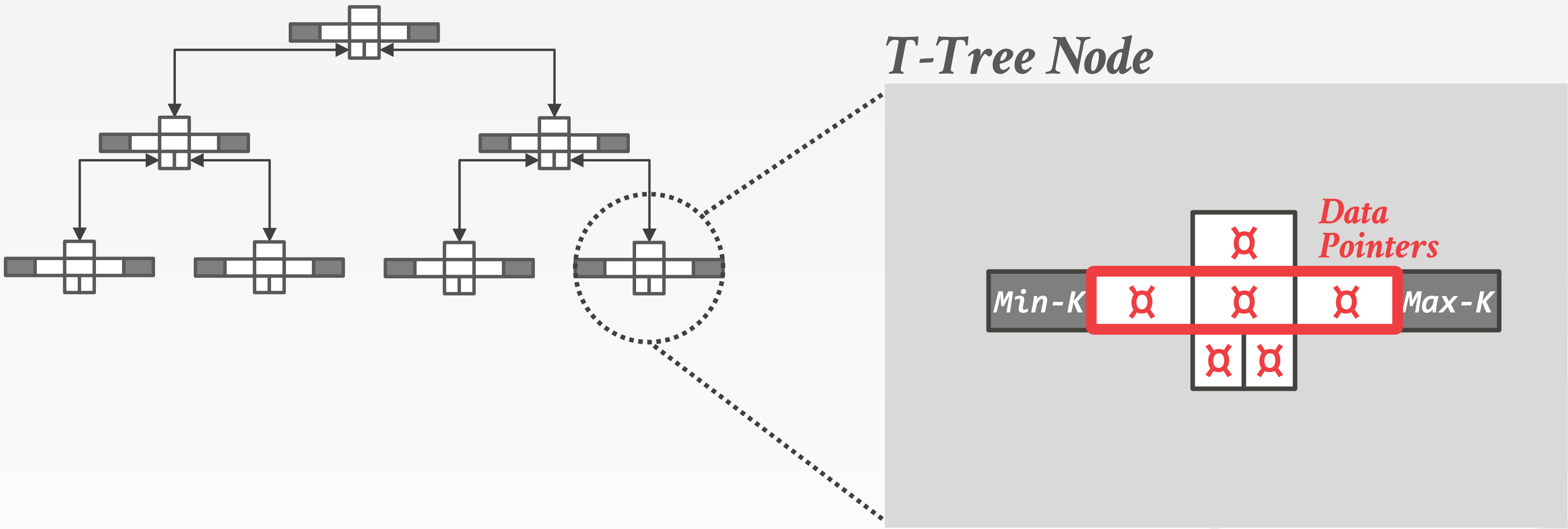
Proposed in 1986 from Univ. of Wisconsin
Used in TimesTen and other early in-memory DBMSs during the 1990s.

A STUDY OF INDEX STRUCTURES FOR MAIN MEMORY DATABASE MANAGEMENT SYSTEMS
VLDB 1986

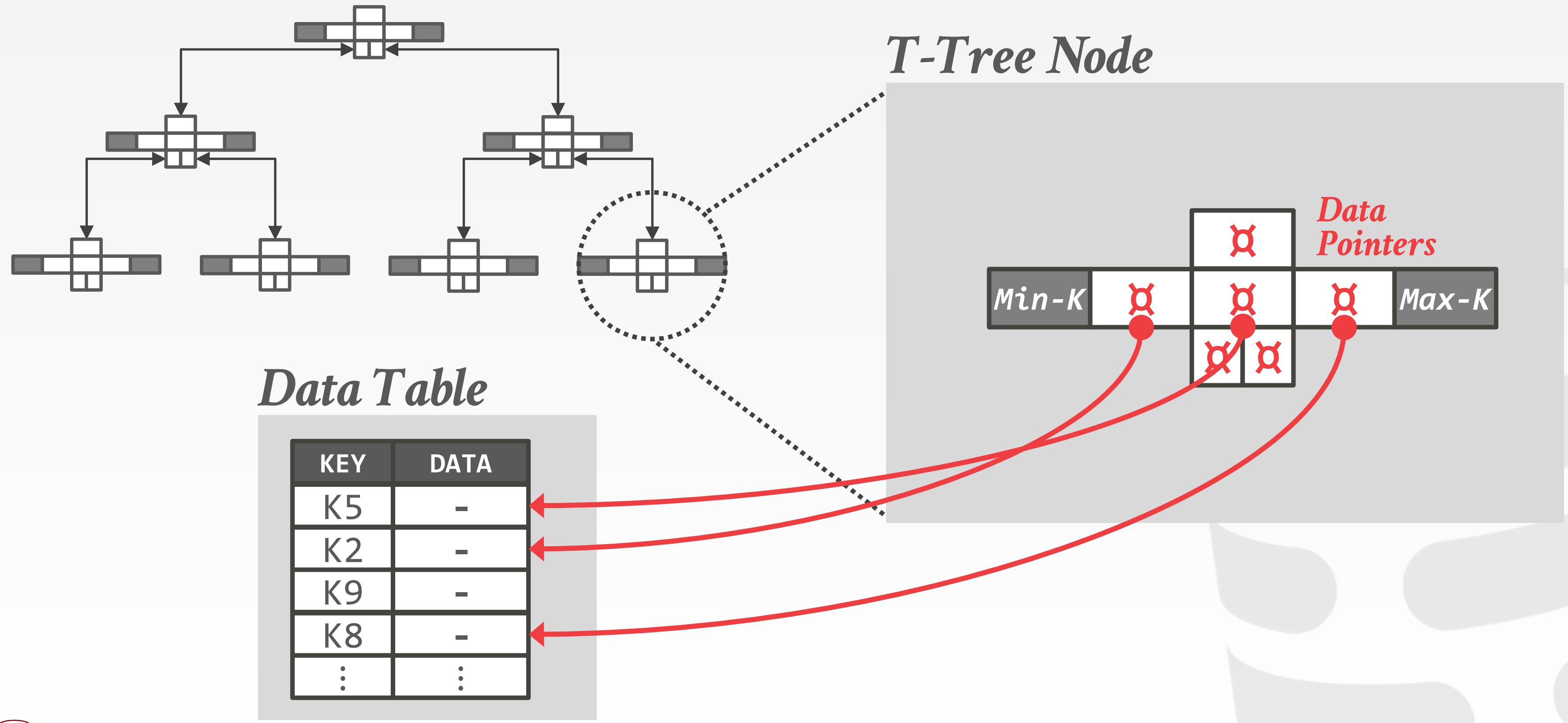
T-TREES



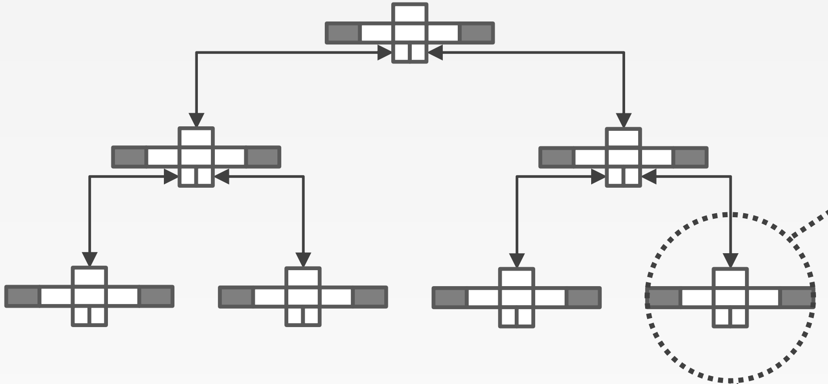
T-TREES



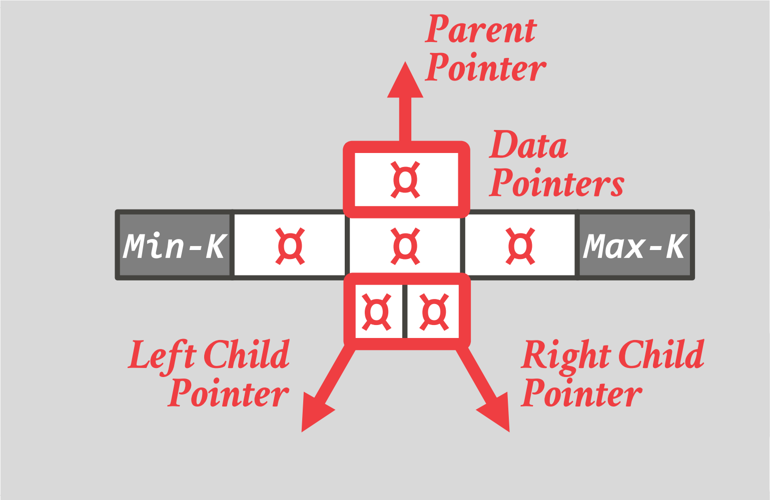
T-TREES


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

T-TREES



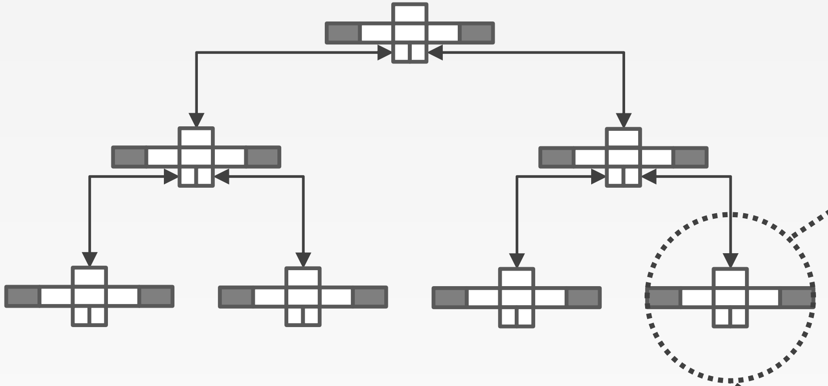
T-Tree Node



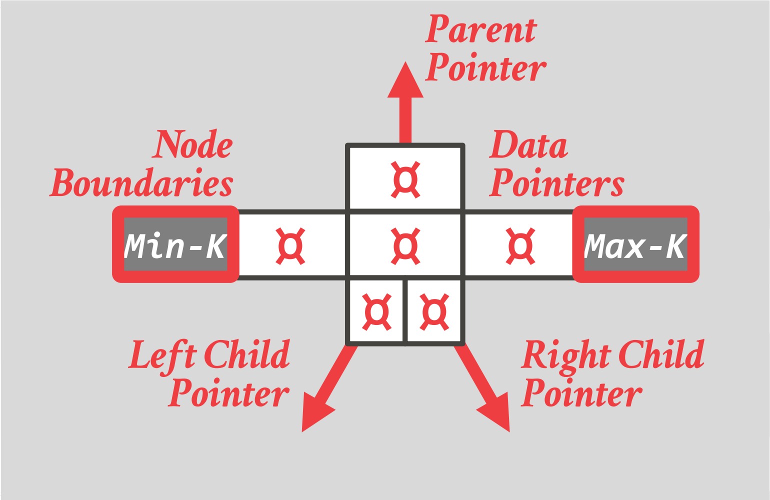
Data Table

KEY	DATA
K5	-
K2	-
K9	-
K8	-
⋮	⋮

T-TREES



T-Tree Node

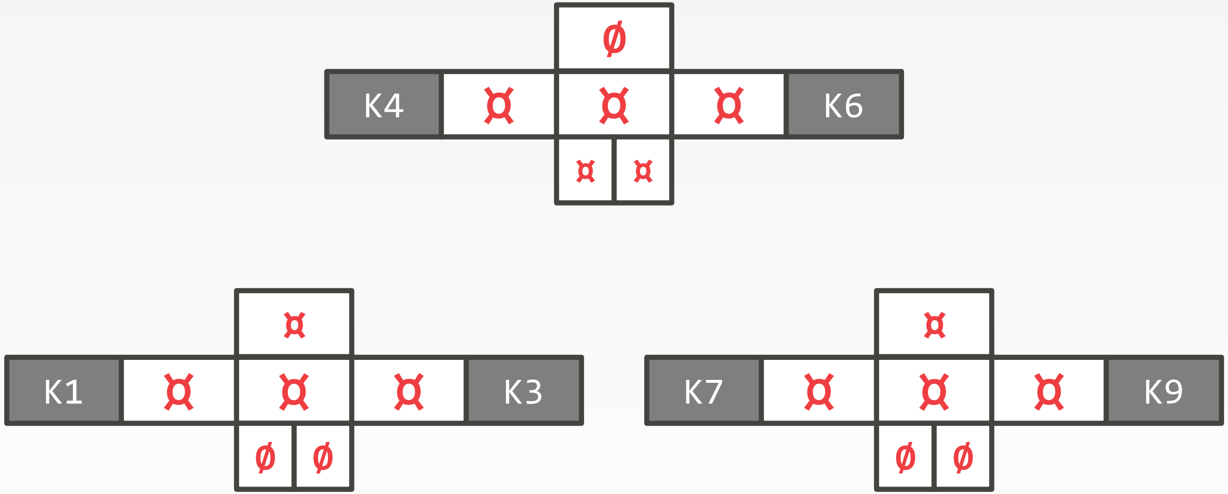


Data Table

KEY	DATA
K5	-
K2	-
K9	-
K8	-
⋮	⋮

T-TREES: SEARCH

Find K2

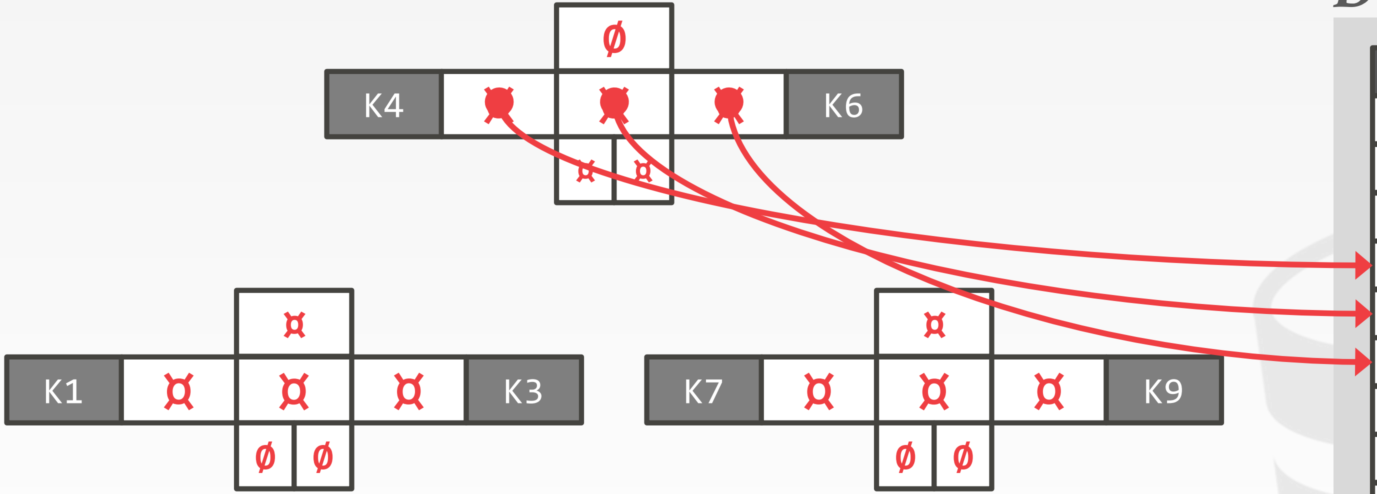


Data Table

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

T-TREES: SEARCH

Find K2

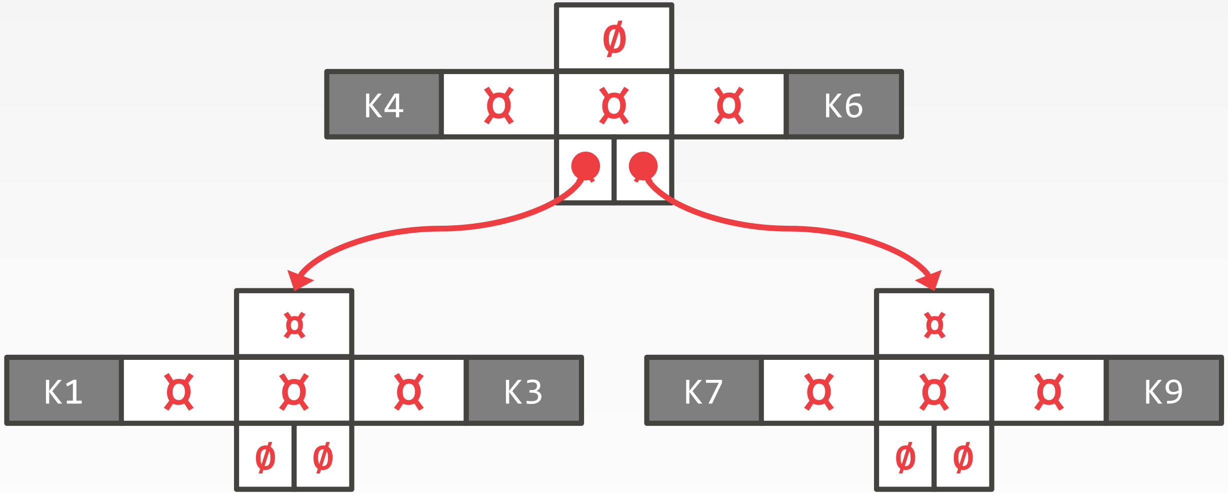


Data Table

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

T-TREES: SEARCH

Find K2

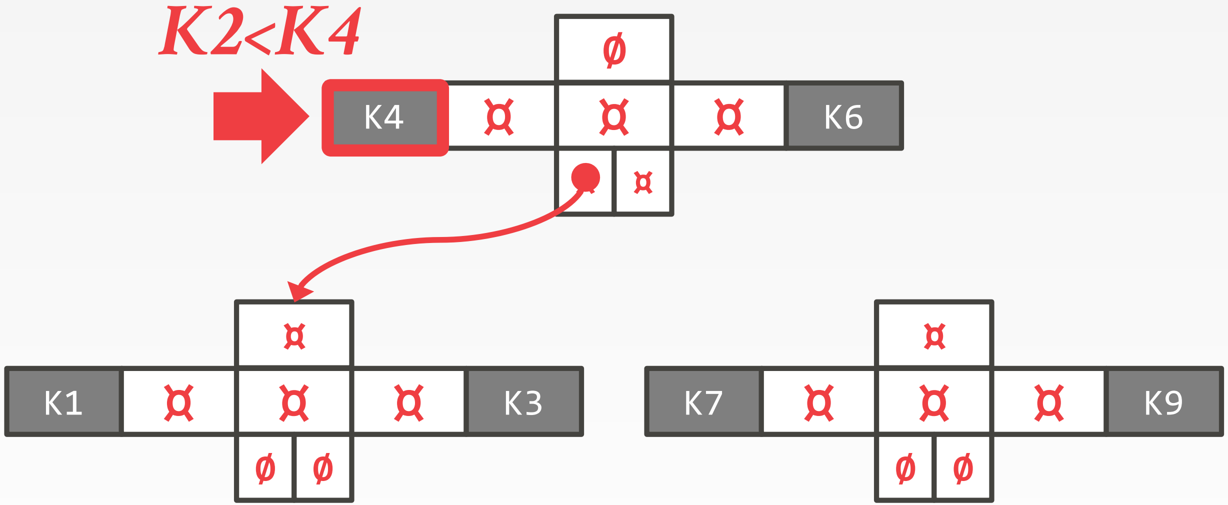


Data Table

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

T-TREES: SEARCH

Find K2

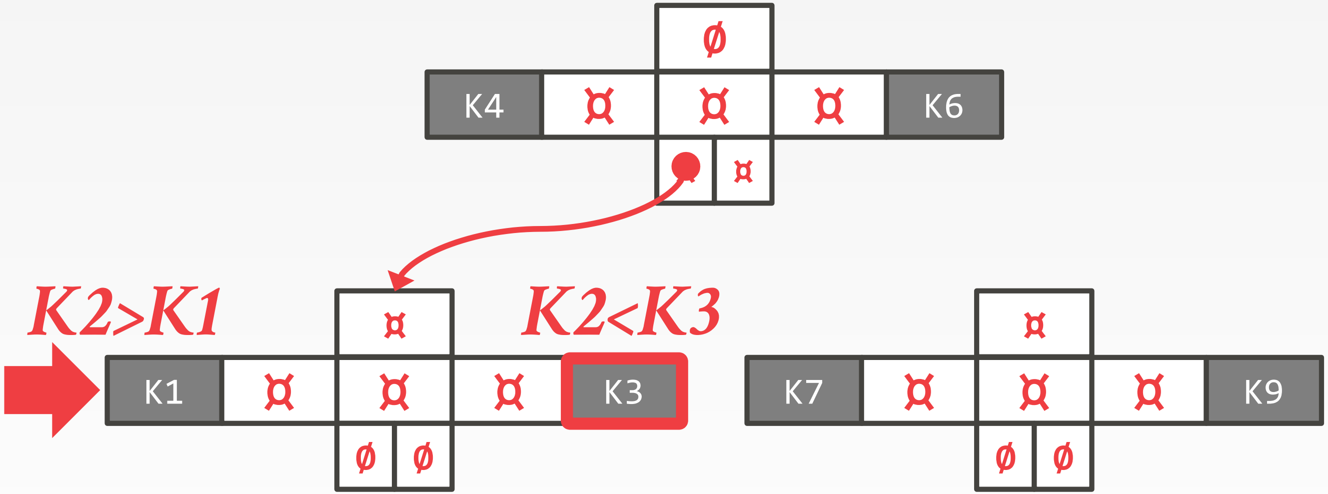


Data Table

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

T-TREES: SEARCH

Find K2

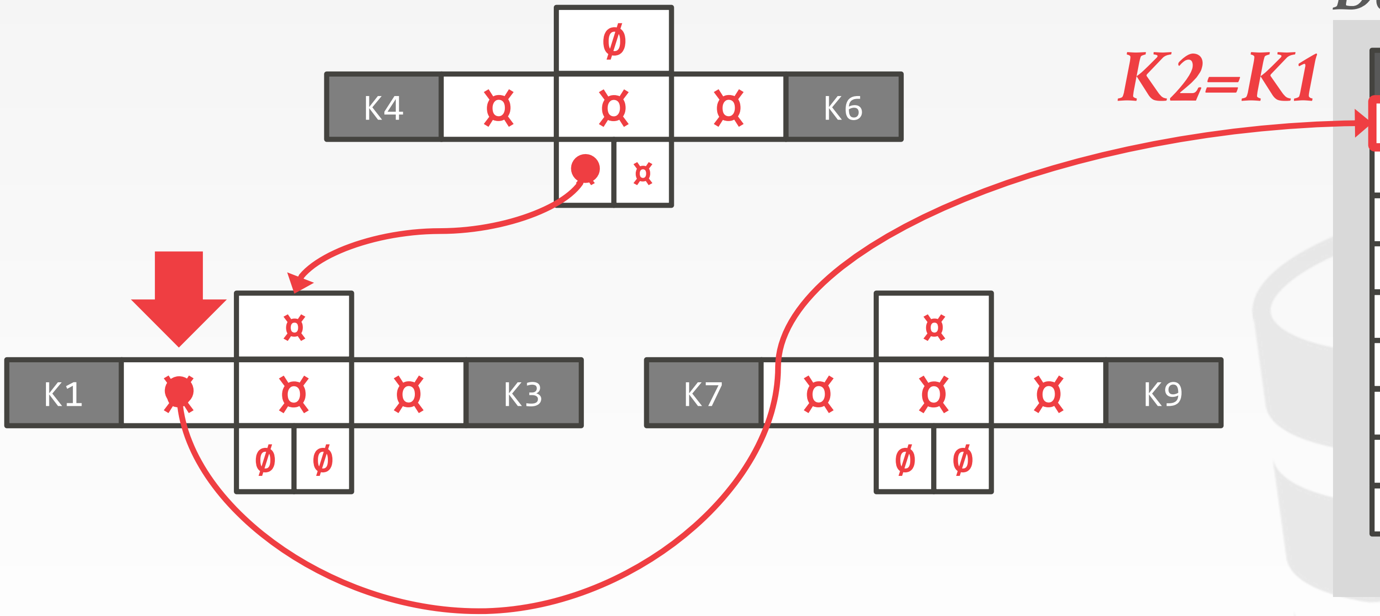


Data Table

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

T-TREES: SEARCH

Find K2

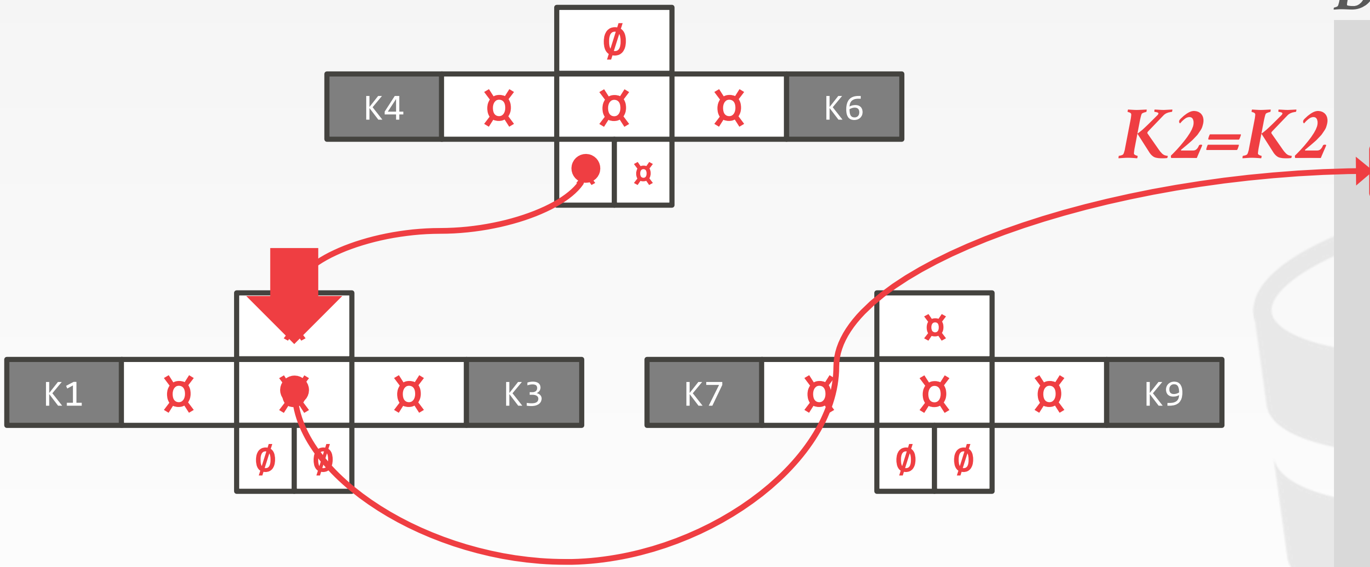


Data Table

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

T-TREES: SEARCH

Find K2



Data Table

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

T-TREE: ADVANTAGES

Uses less memory because it does not store 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-TREES: DISADVANTAGES

Difficult to rebalance.

Difficult to implement safe concurrent access.

Must chase pointers when scanning range or performing binary search inside of a node.
→ This greatly hurts cache locality.

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 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: Deltas
→ No updates in place
→ Reduces cache invalidation.

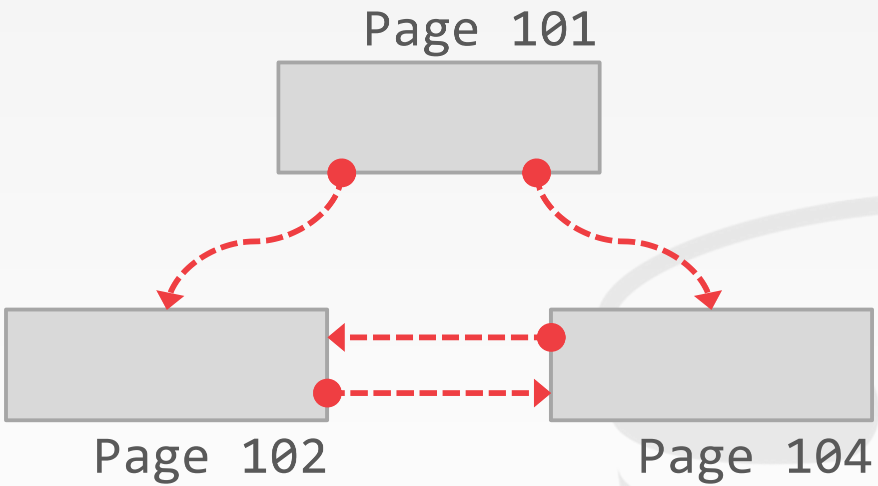
Key Idea #2: Mapping Table
→ Allows for CaS of physical locations of pages.

THE BW-TREE: A B-TREE FOR NEW HARDWARE
ICDE 2013

BW-TREE: MAPPING TABLE

Mapping Table

PID	Addr
101	
102	
103	
104	



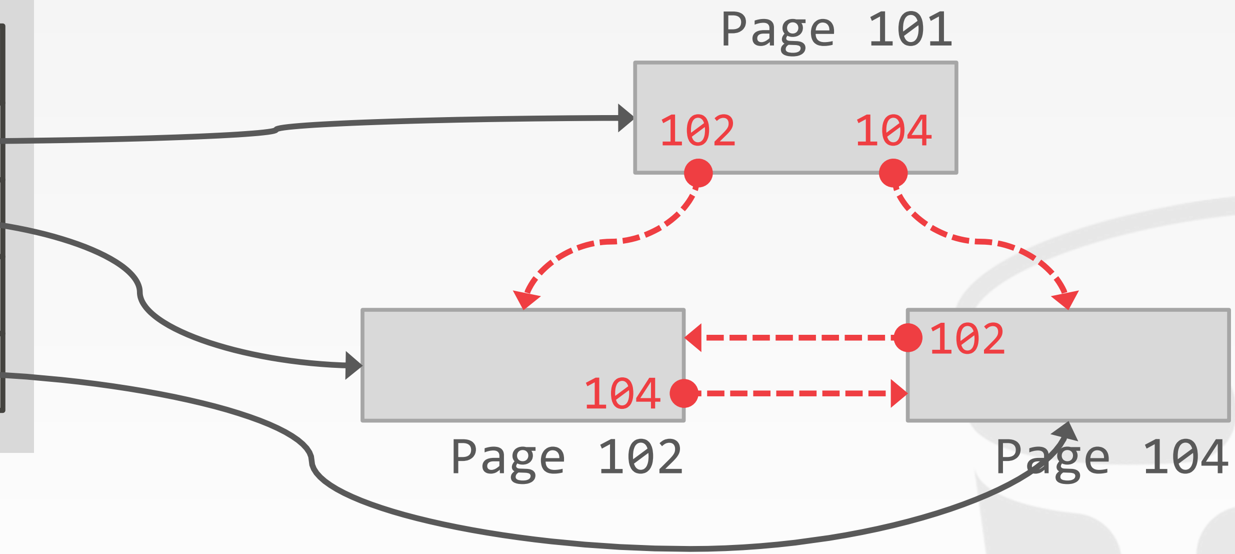
Logical Pointer

Physical Pointer

BW-TREE: MAPPING TABLE

Mapping Table

PID	*Addr*
101	
102	
103	
104	

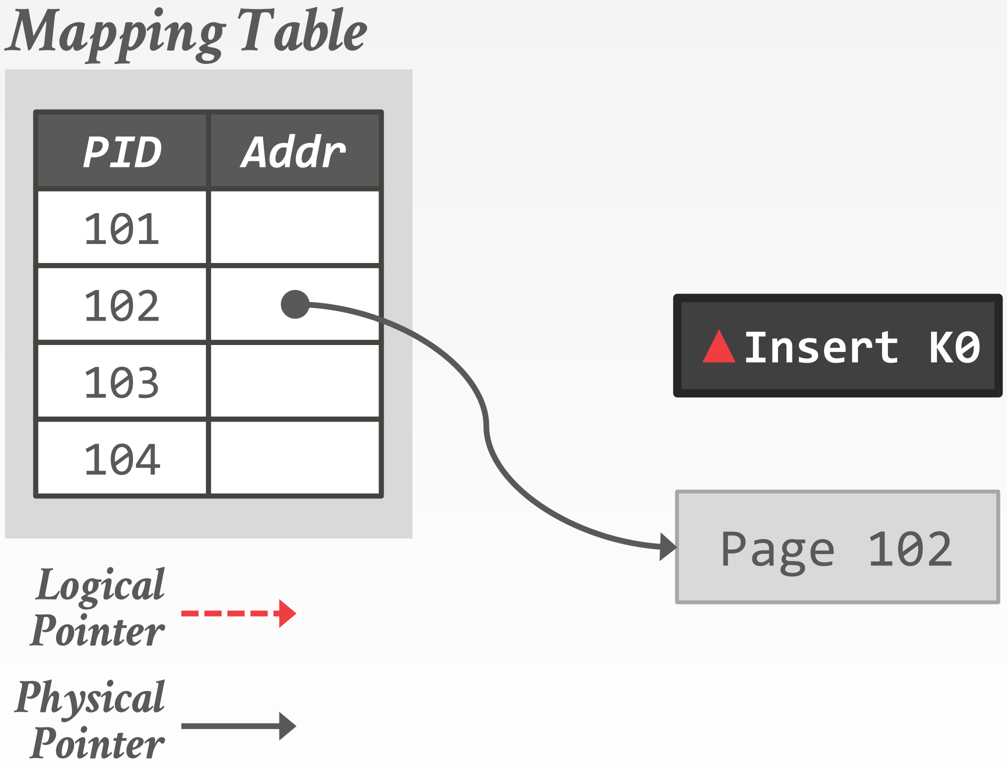


Logical Pointer

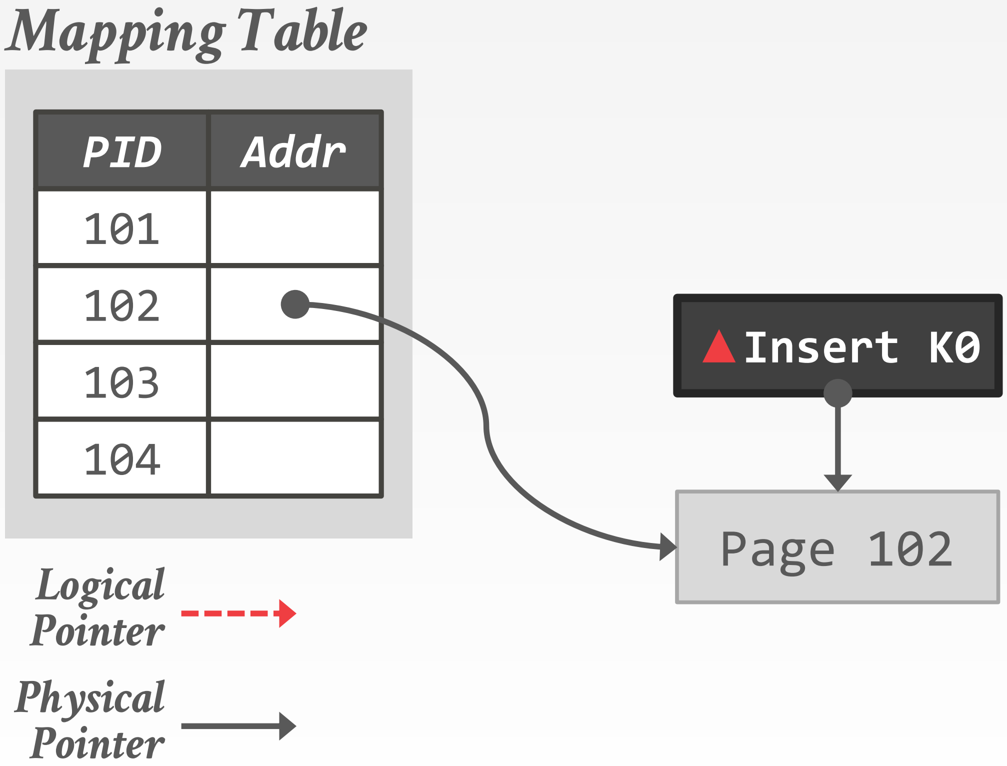
Physical Pointer

BW-TREE: DELTA UPDATES

Each update to a page produces a new delta.



BW-TREE: DELTA UPDATES

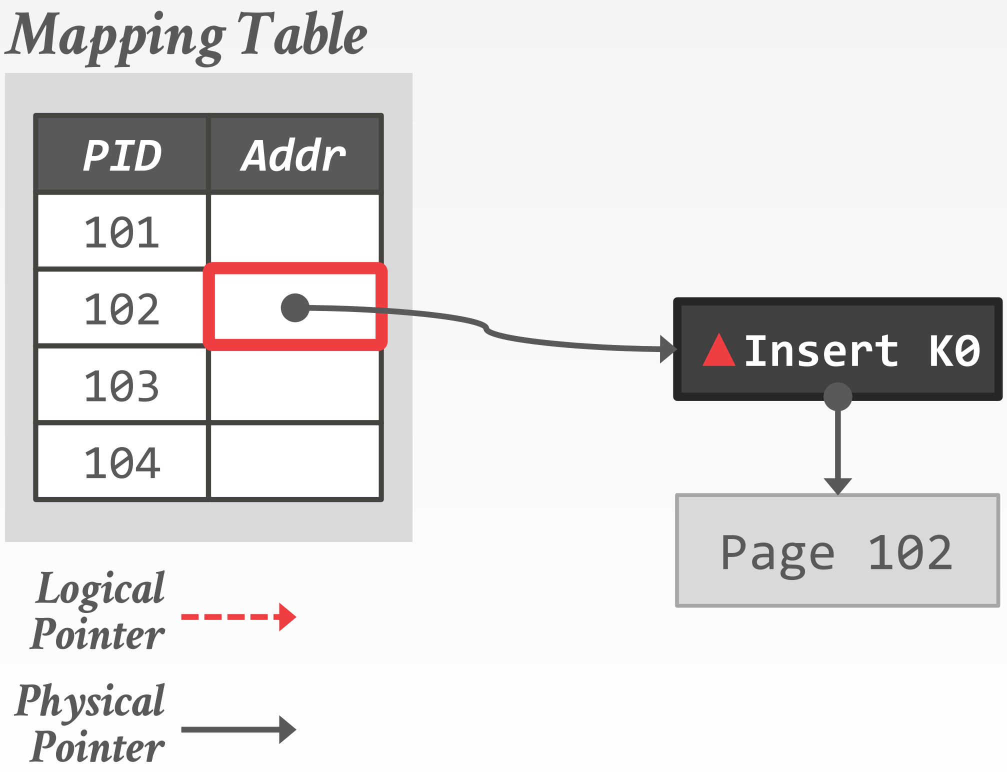


Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CaS.

BW-TREE: DELTA UPDATES

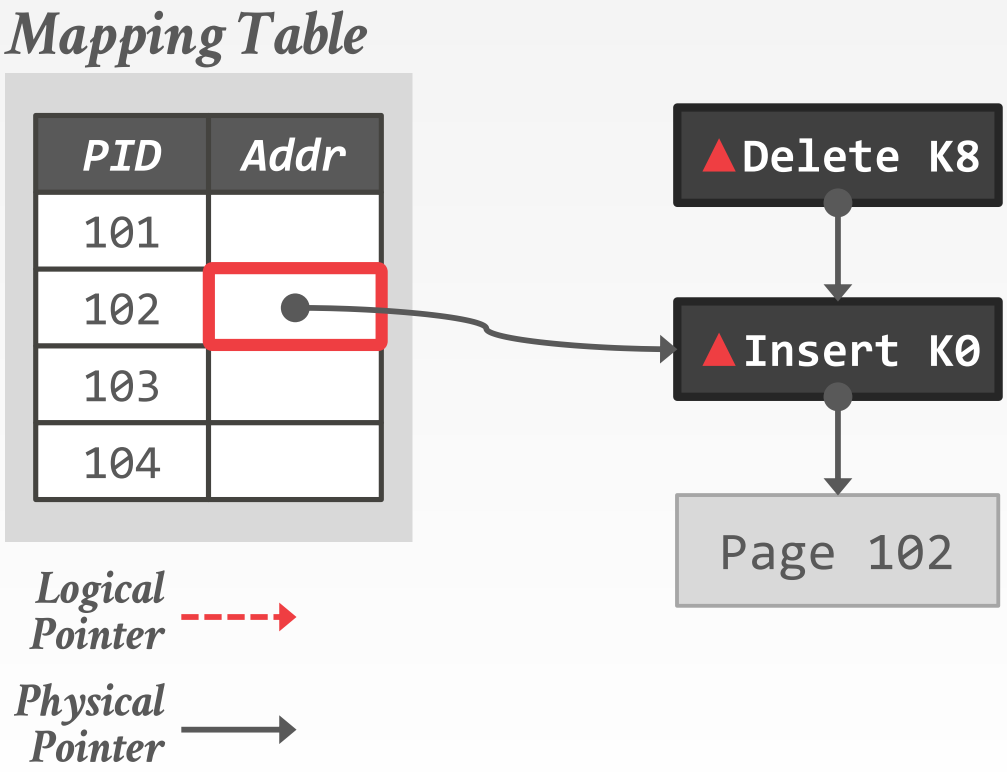


Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CaS.

BW-TREE: DELTA UPDATES

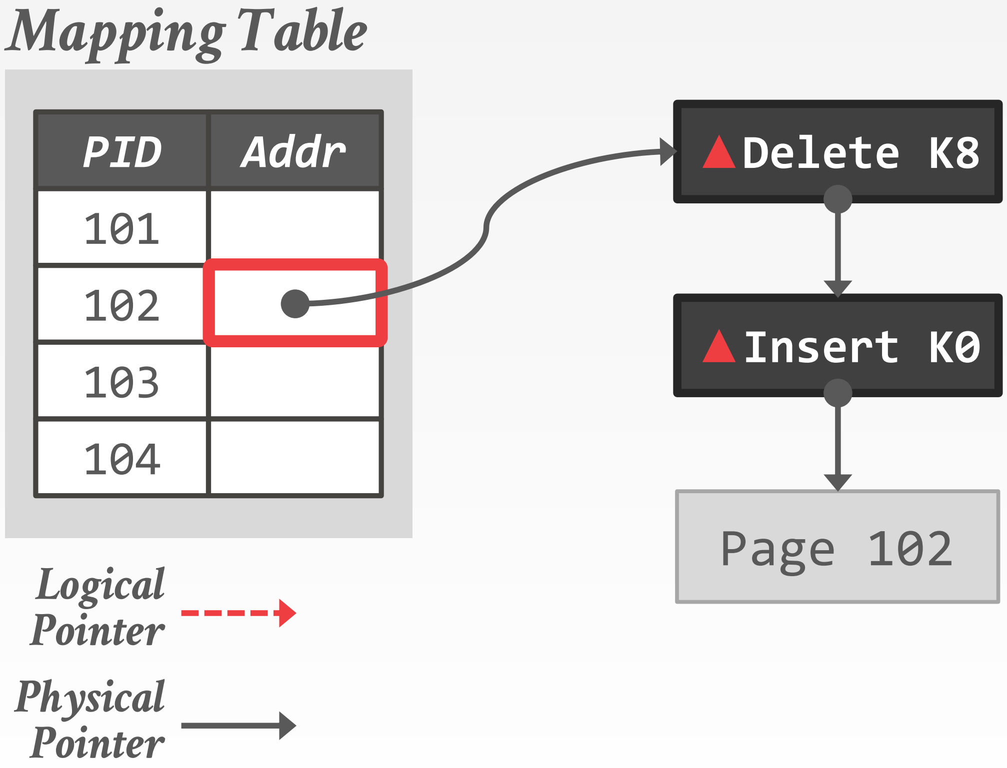


Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CaS.

BW-TREE: DELTA UPDATES



Each update to a page produces a new delta.

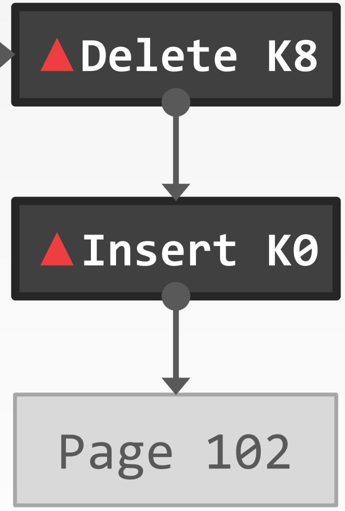
Delta physically points to base page.

Install delta address in physical address slot of mapping table using CaS.

BW-TREE: SEARCH

Mapping Table

PID	Addr
101	
102	
103	
104	



Logical Pointer

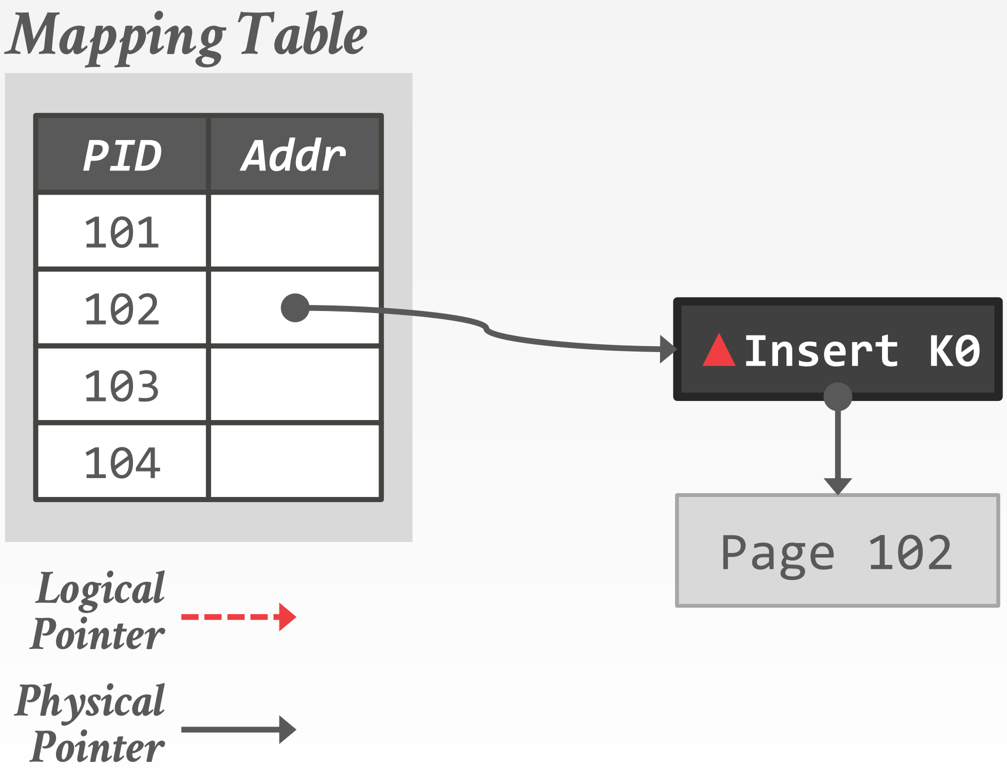
Physical Pointer

Traverse tree like a regular B+tree.

If mapping table points to delta chain, stop at first occurrence of search key.

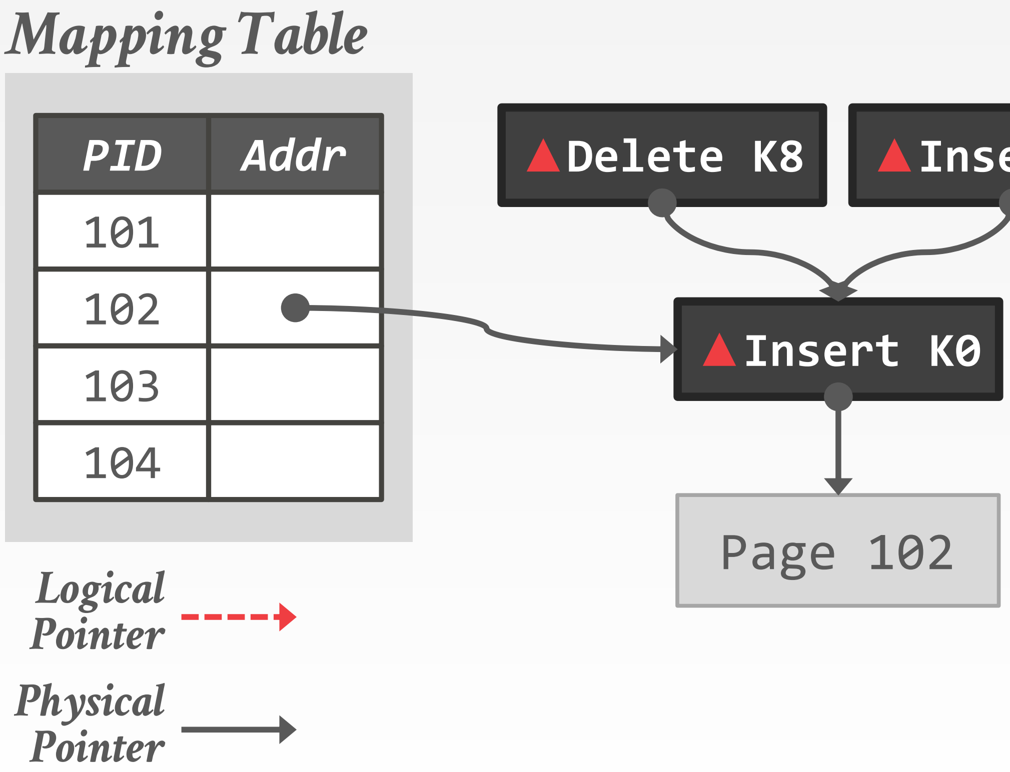
Otherwise, perform binary search on base page.

BW-TREE: CONTENTION UPDATES



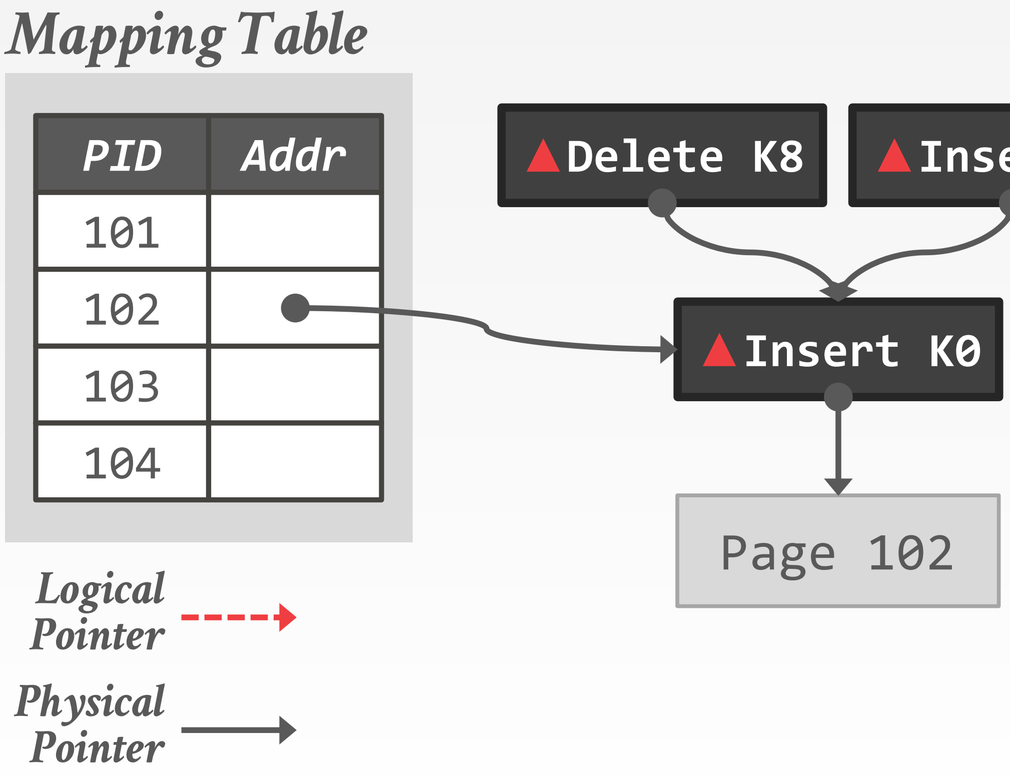
Threads may try to install updates to same page.

BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

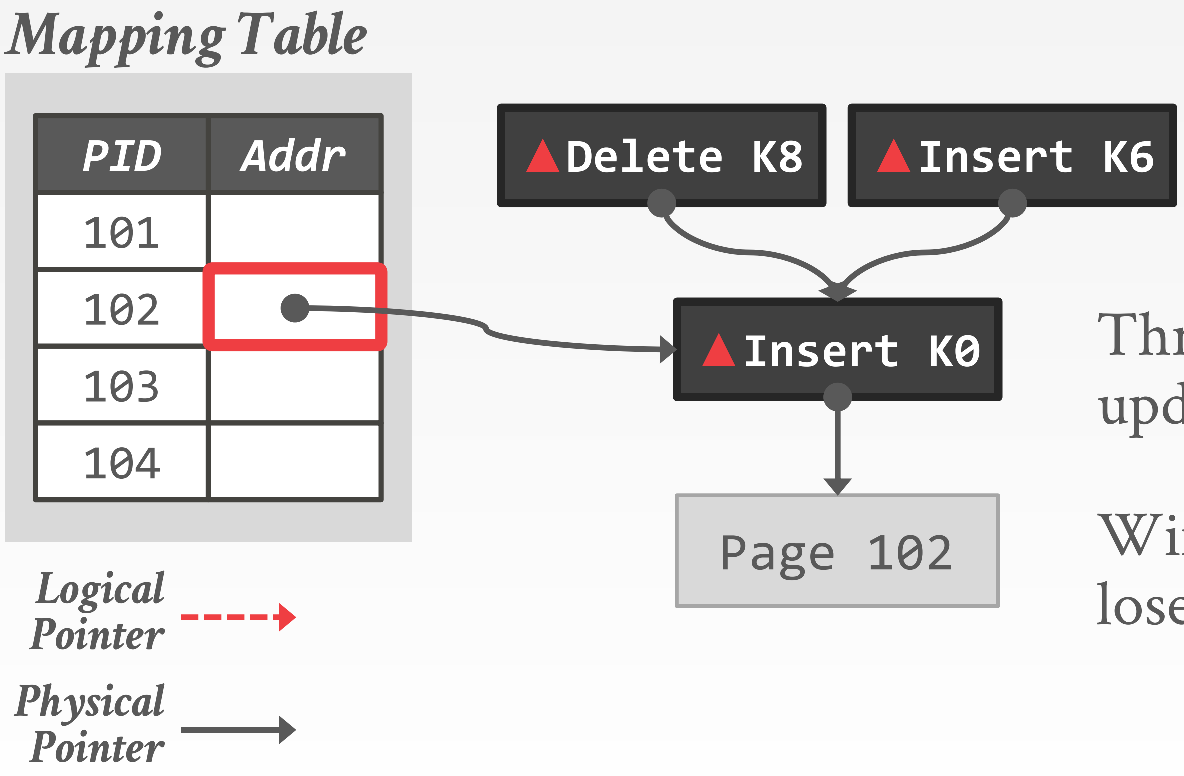
BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

Winner succeeds, any losers must retry or abort

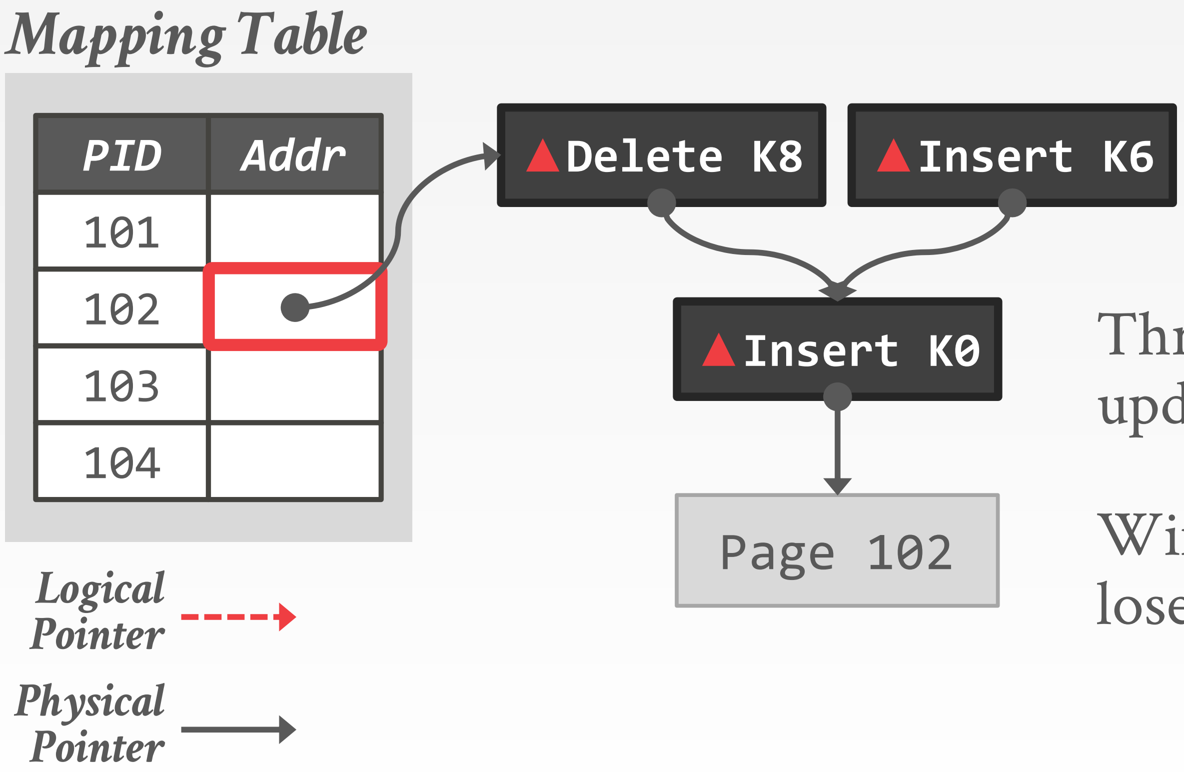
BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

Winner succeeds, any losers must retry or abort

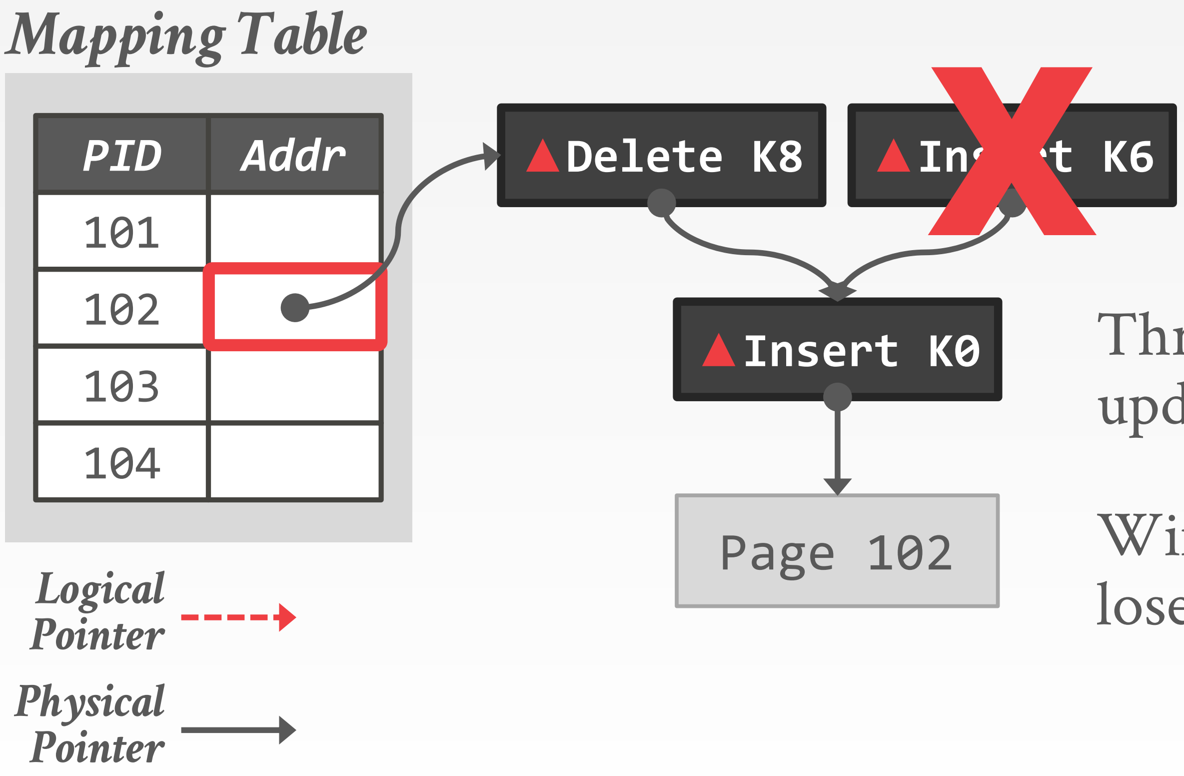
BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

Winner succeeds, any losers must retry or abort

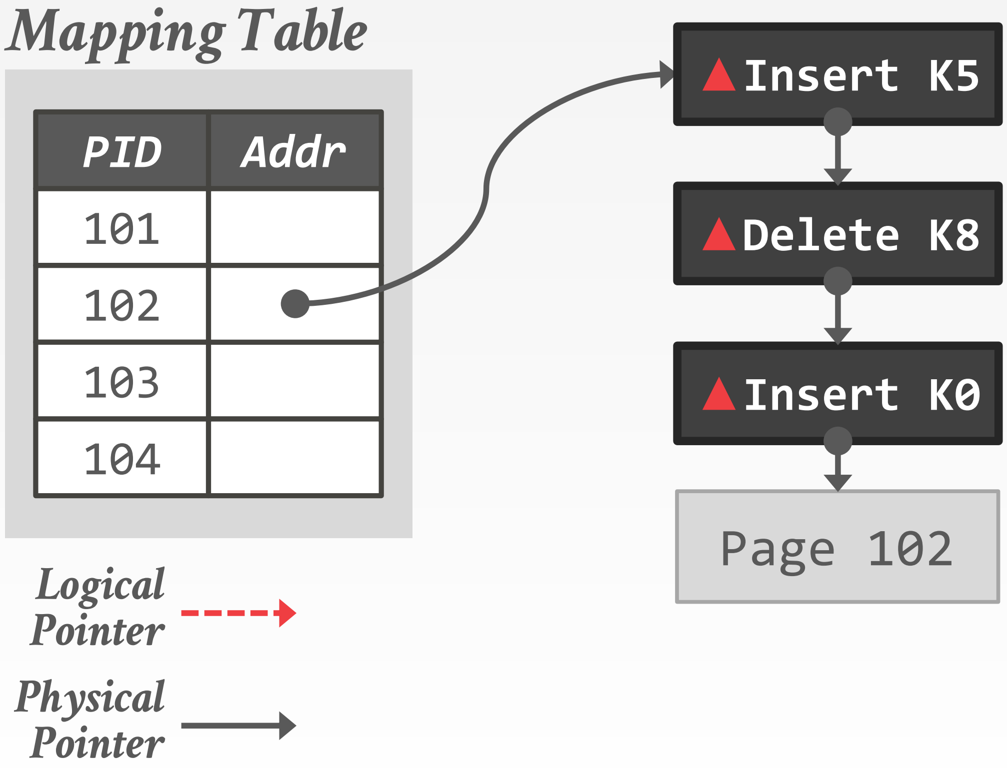
BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

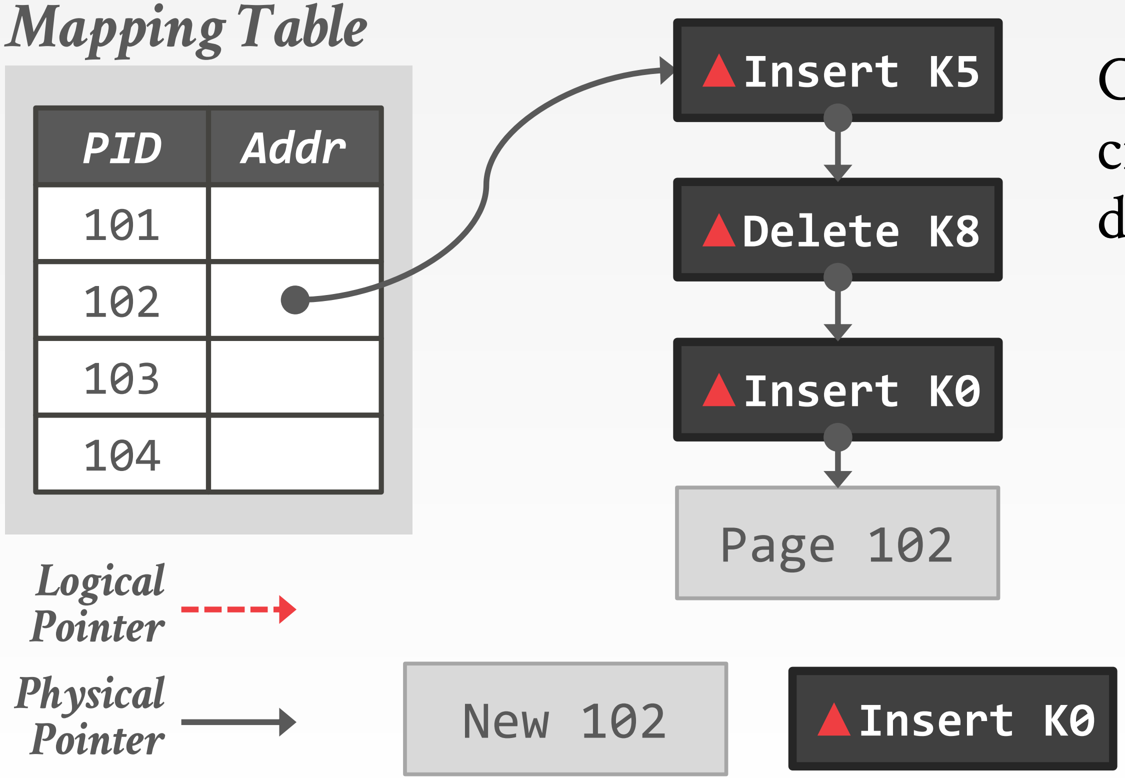
Winner succeeds, any losers must retry or abort

BW-TREE: CONSOLIDATION



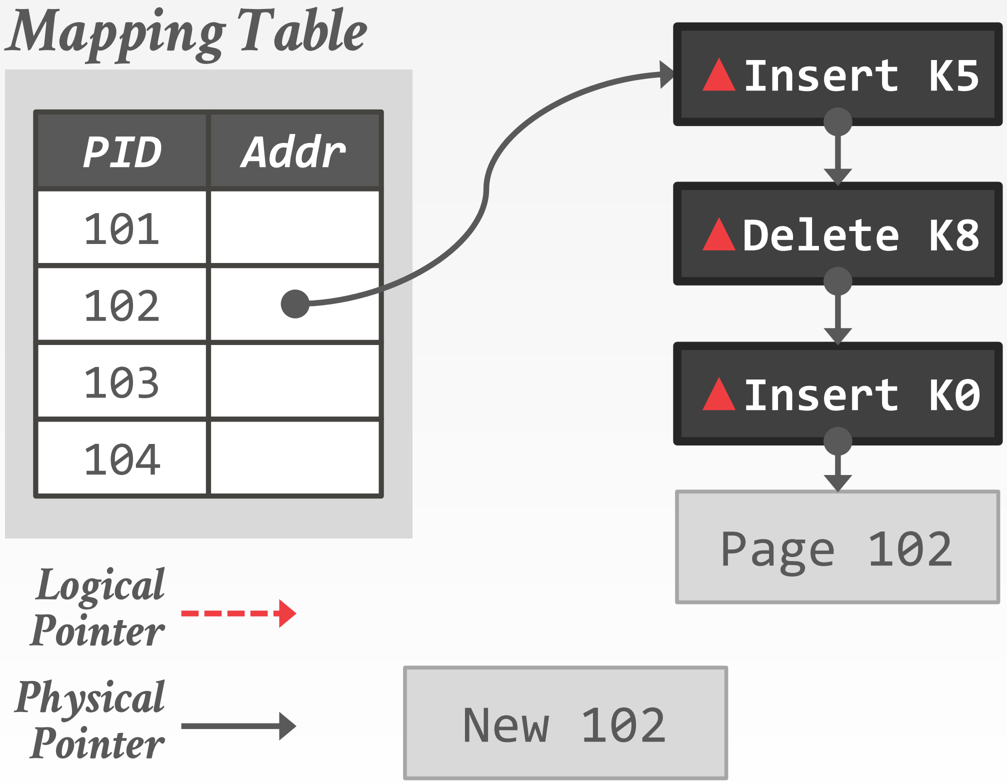
Consolidate updates by creating new page with deltas applied.

BW-TREE: CONSOLIDATION



Consolidate updates by creating new page with deltas applied.

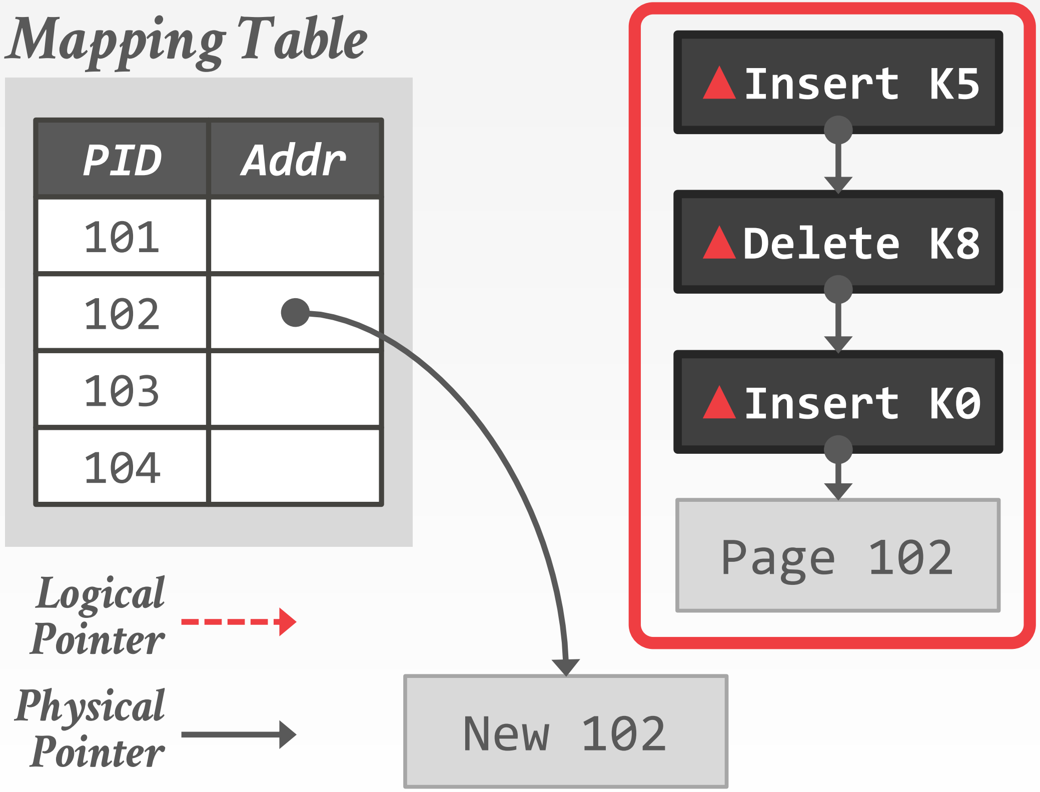
BW-TREE: CONSOLIDATION



Consolidate updates by creating new page with deltas applied.

CaS-ing the mapping table address ensures no deltas are missed.

BW-TREE: CONSOLIDATION



Consolidate updates by creating new page with deltas applied.

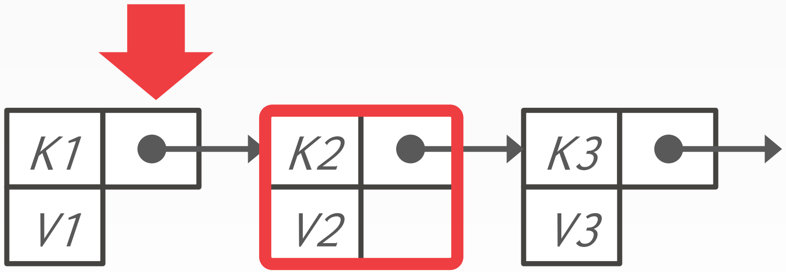
CaS-ing the mapping table address ensures no deltas are missed.

Old page + deltas are marked as garbage.

GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

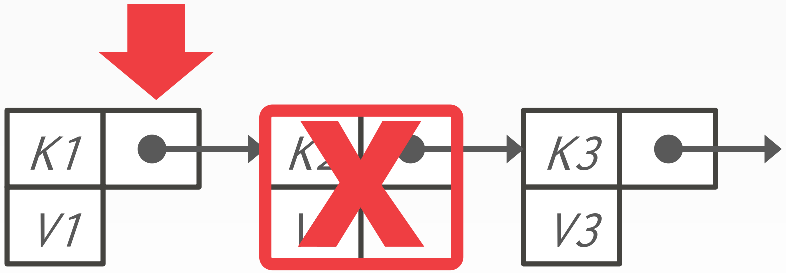
→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

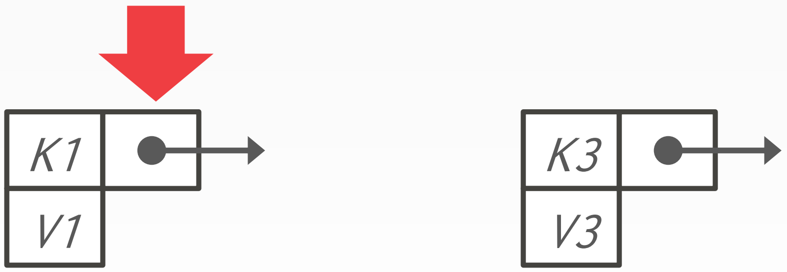
→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

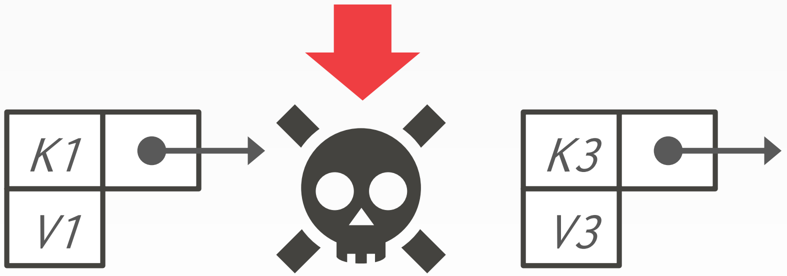
→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



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.

Source: Stephen Tu

EPOCH 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).

Also known as ***Read-Copy-Update*** (RCU) in Linux.

BW-TREE: GARBAGE COLLECTION

Operations are tagged with an **epoch**

→ Each epoch tracks the threads that are part of it and the objects that can be reclaimed.

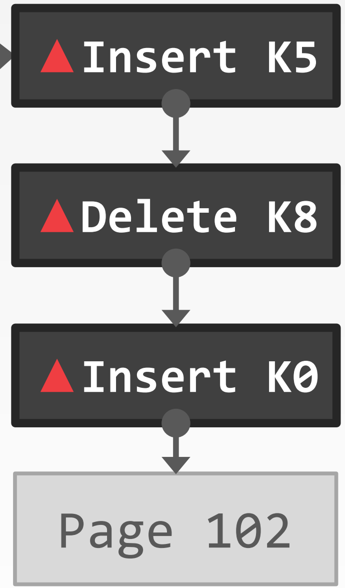
→ Thread joins an epoch prior to each operation and post objects that can be reclaimed for the current epoch (not necessarily the one it joined)

Garbage for an epoch reclaimed only when all threads have exited the epoch.

BW-TREE: GARBAGE COLLECTION

Mapping Table

PID	Addr
101	
102	
103	
104	

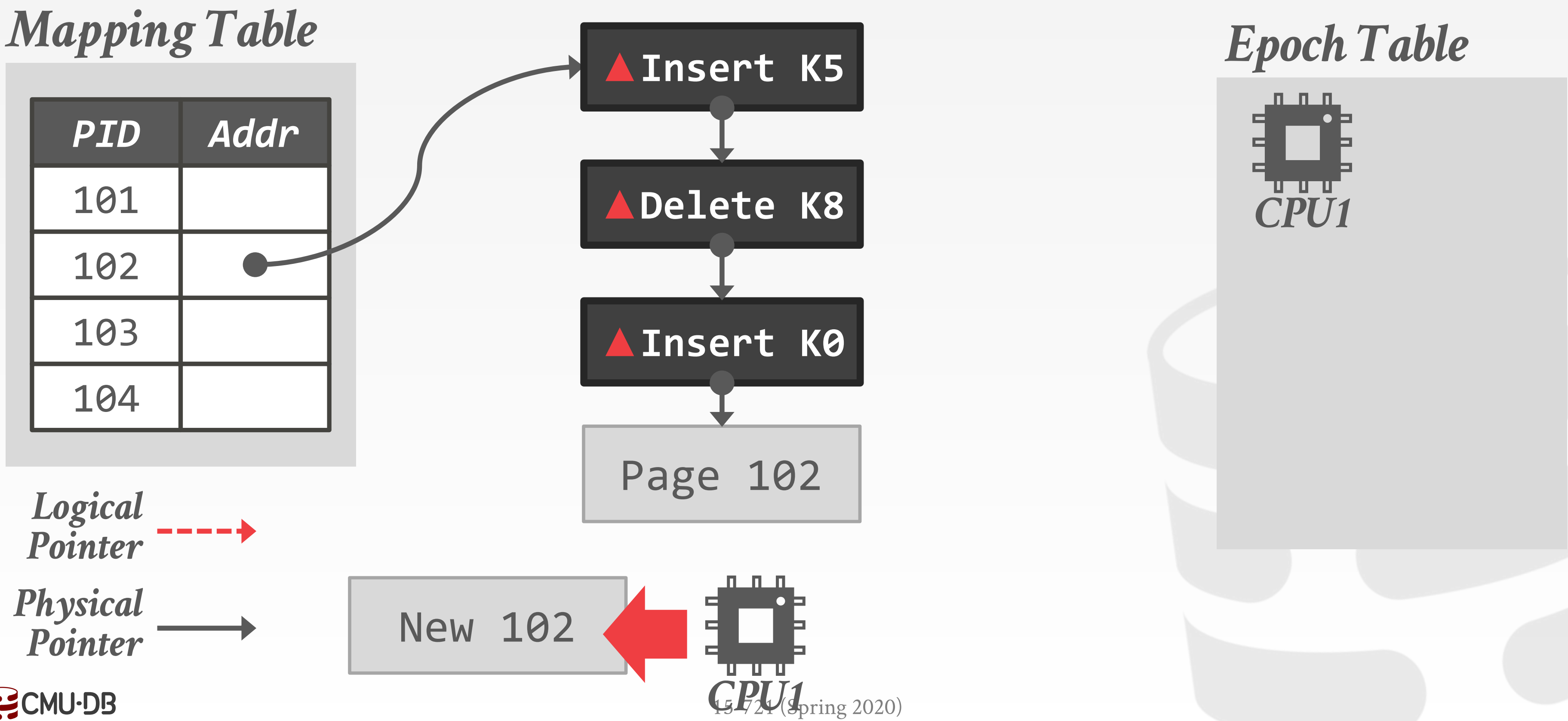


Epoch Table

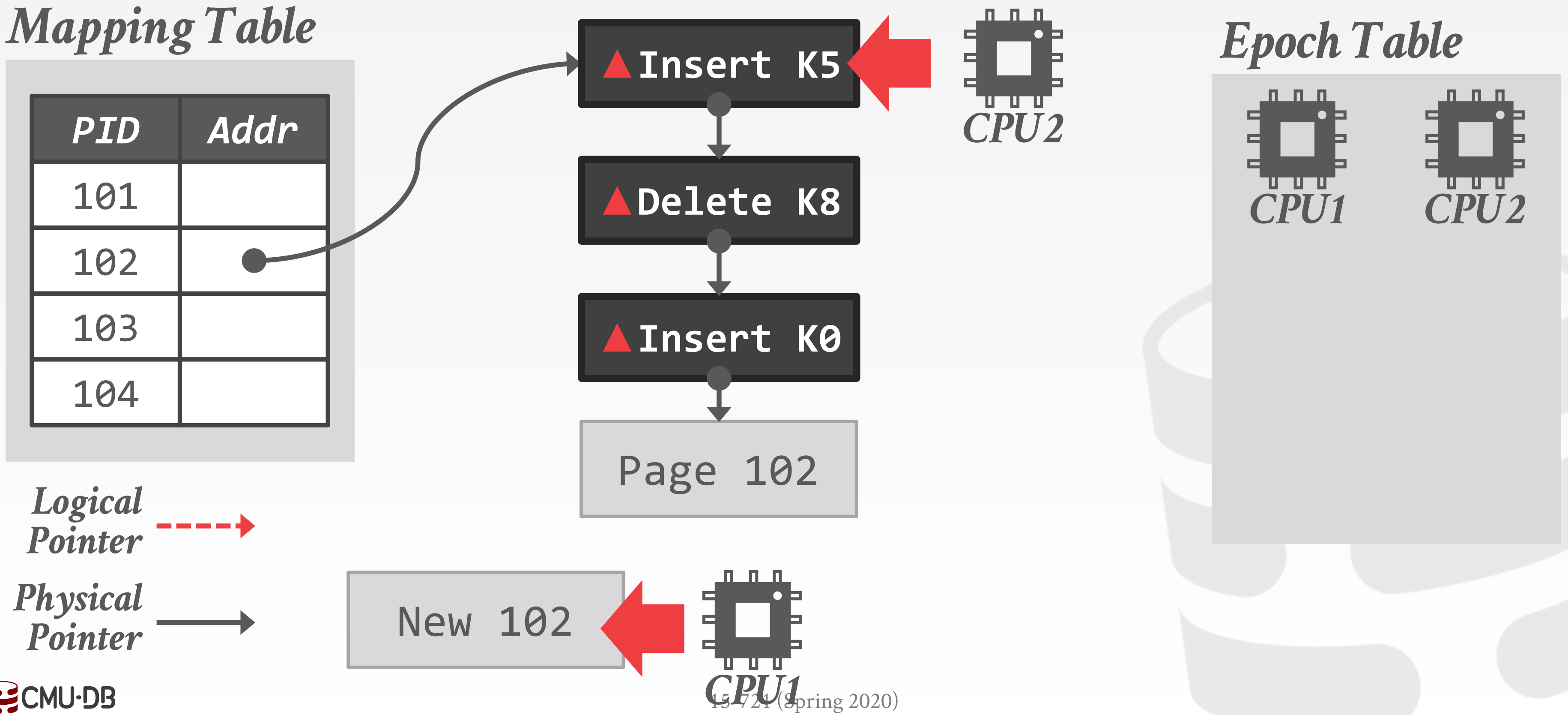
Logical Pointer

Physical Pointer

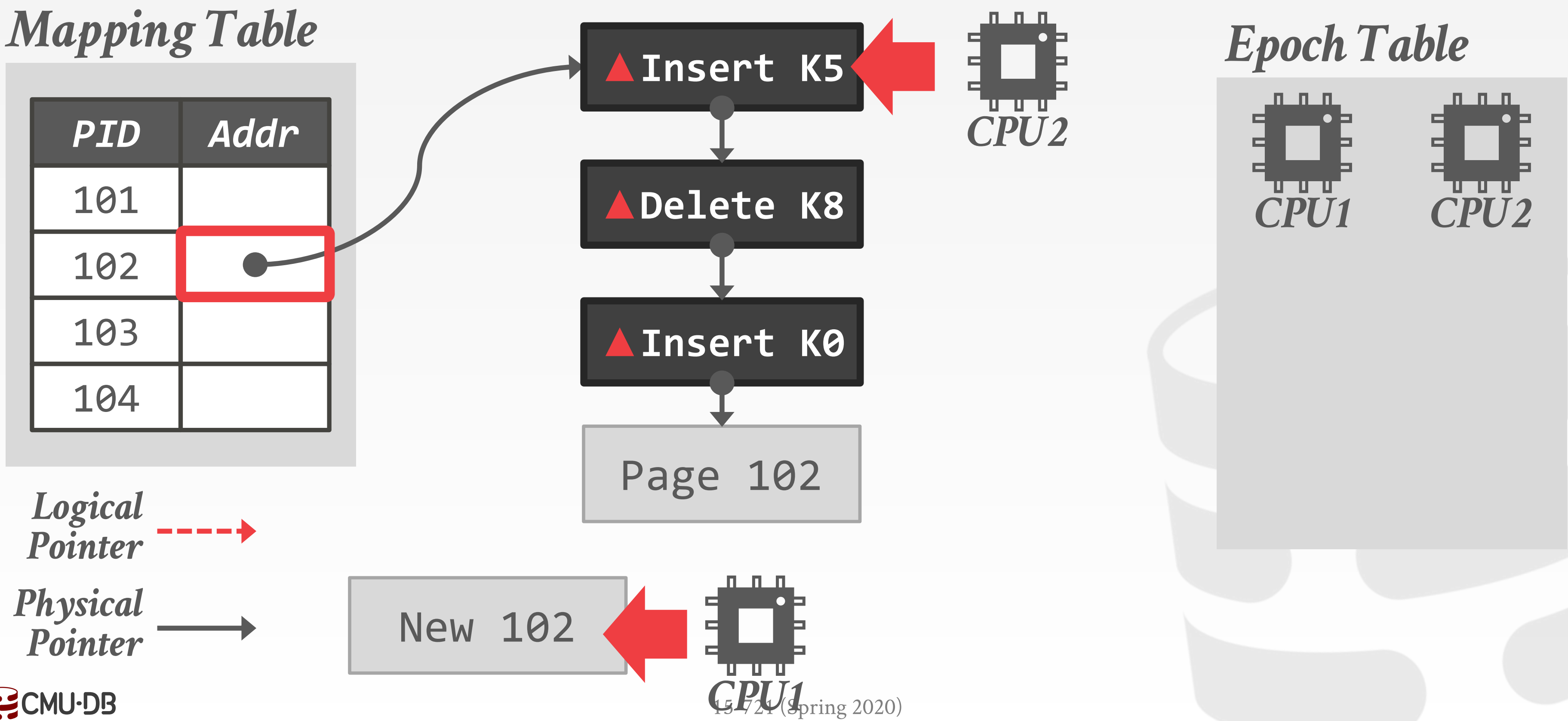
BW-TREE: GARBAGE COLLECTION



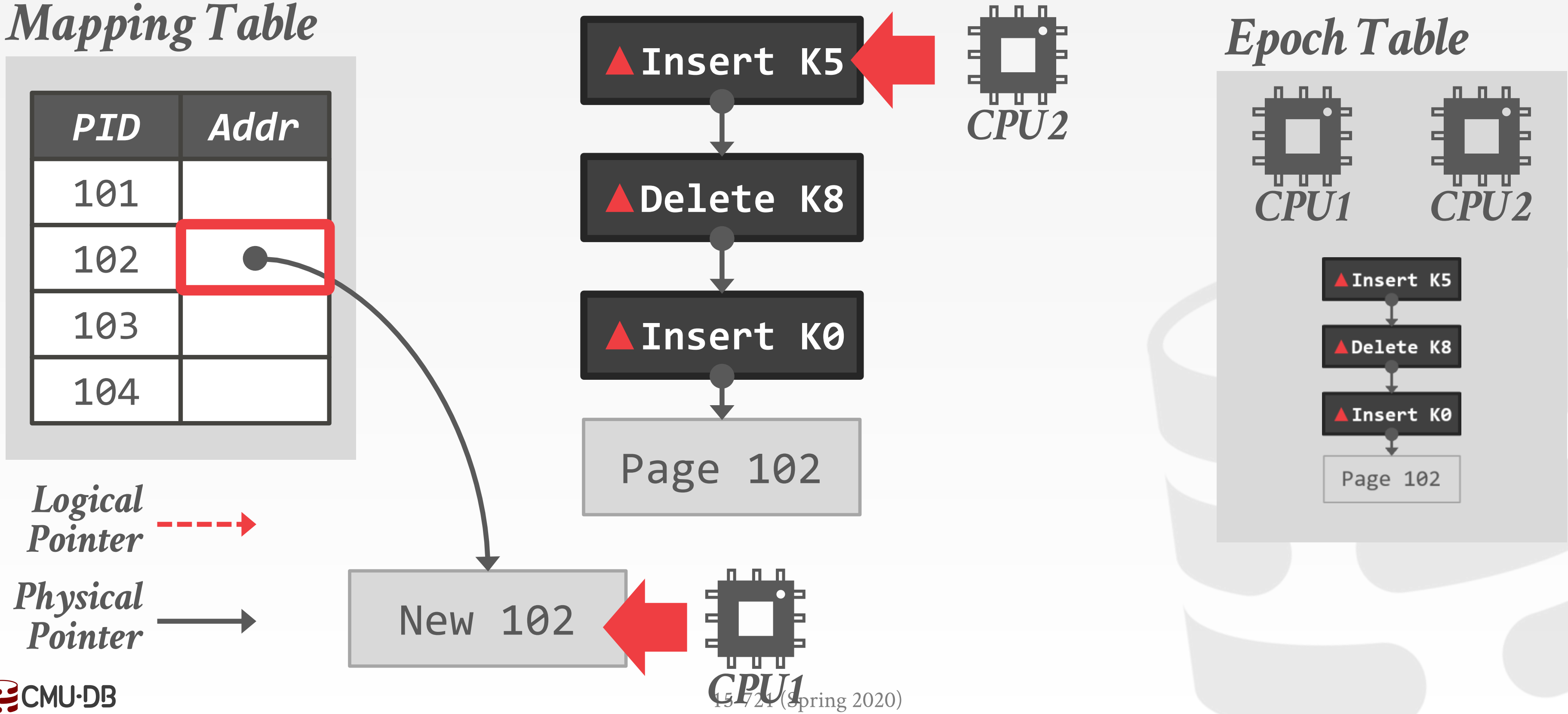
BW-TREE: GARBAGE COLLECTION



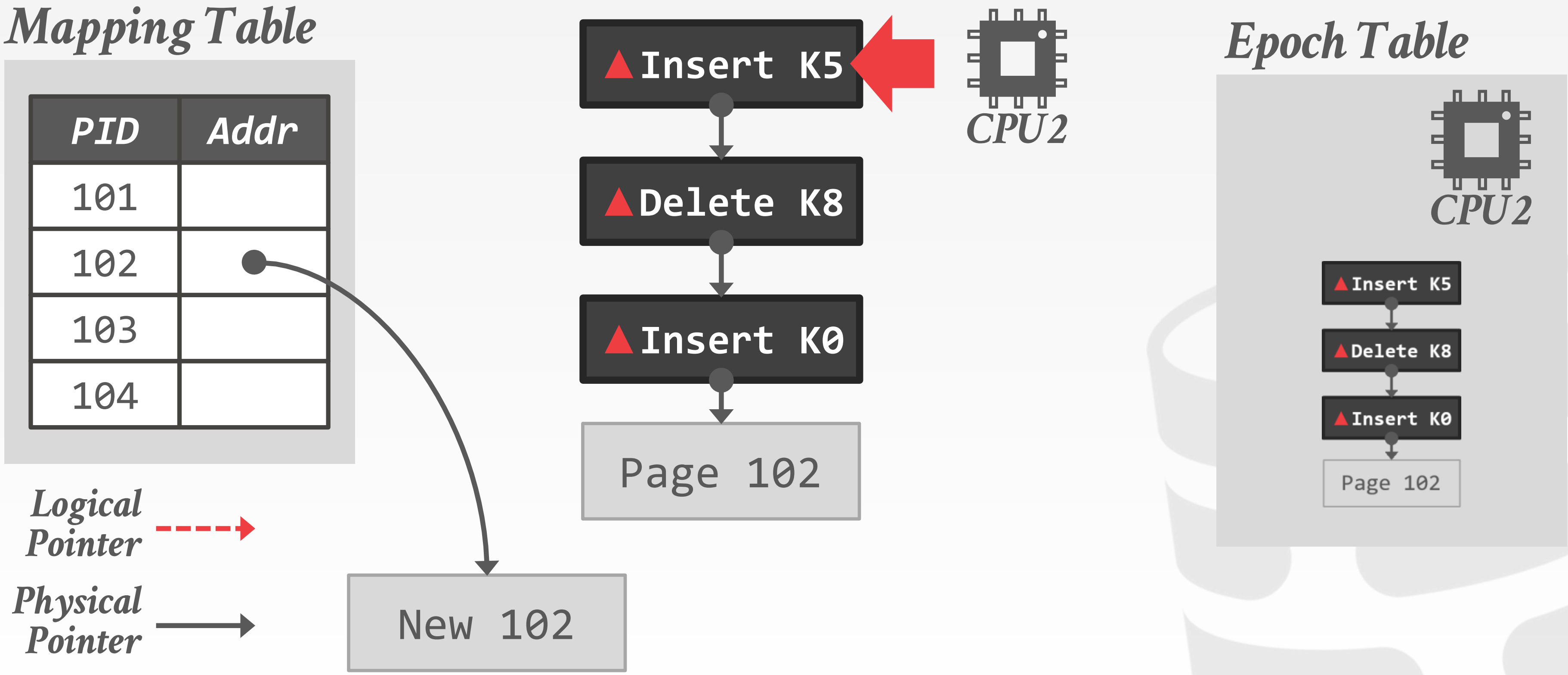
BW-TREE: GARBAGE COLLECTION



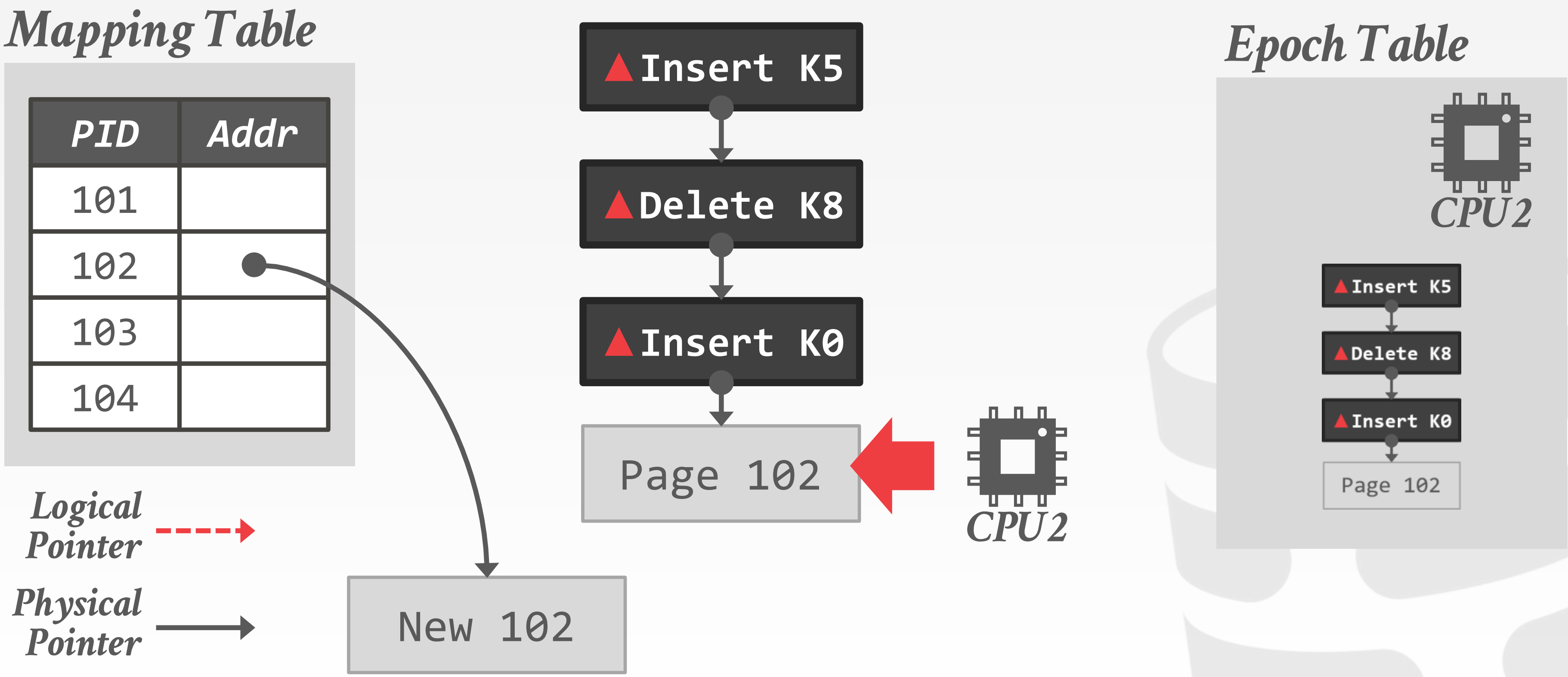
BW-TREE: GARBAGE COLLECTION



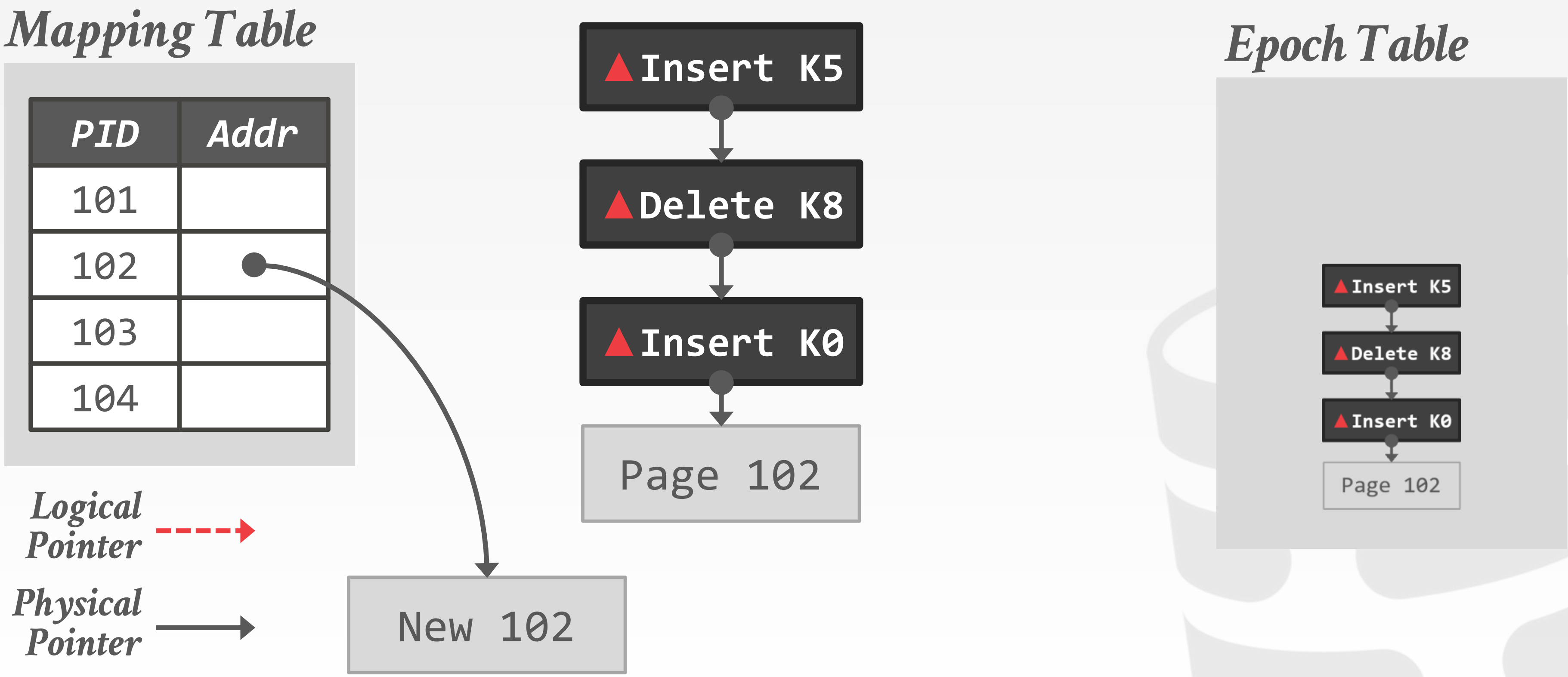
BW-TREE: GARBAGE COLLECTION



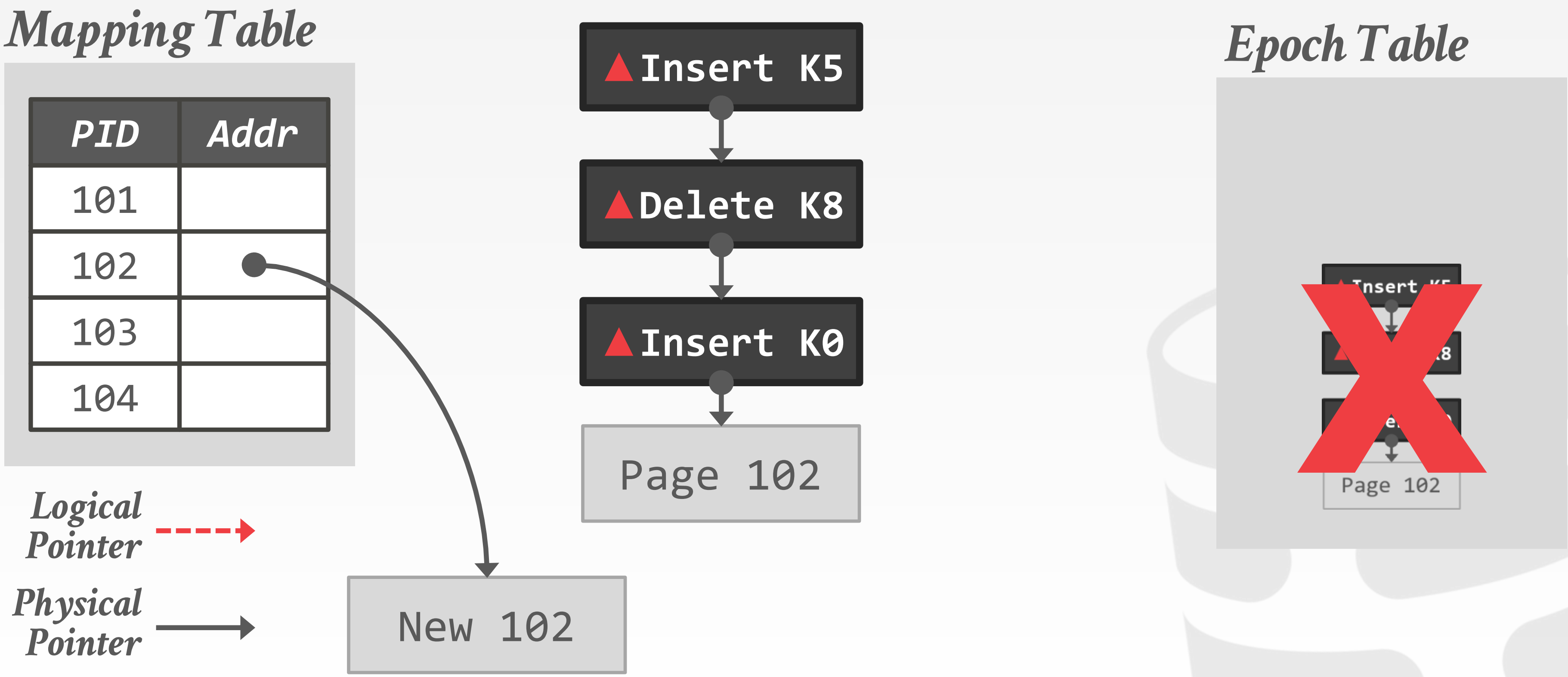
BW-TREE: GARBAGE COLLECTION



BW-TREE: GARBAGE COLLECTION



BW-TREE: GARBAGE COLLECTION



BW-TREE: STRUCTURE MODIFICATIONS

Split Delta Record

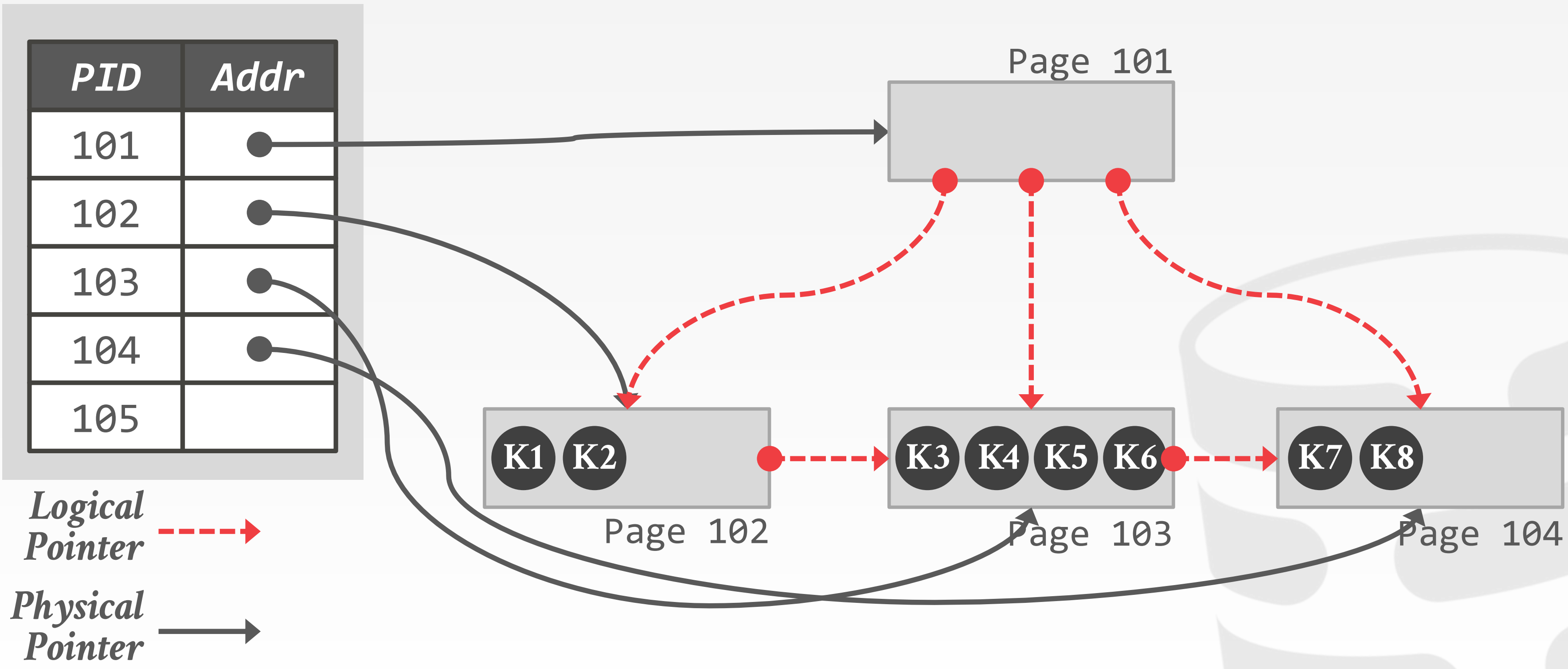
→ Mark that a subset of the base page's key range is now located at another page.
→ Use a logical pointer to the new page.

Separator Delta Record

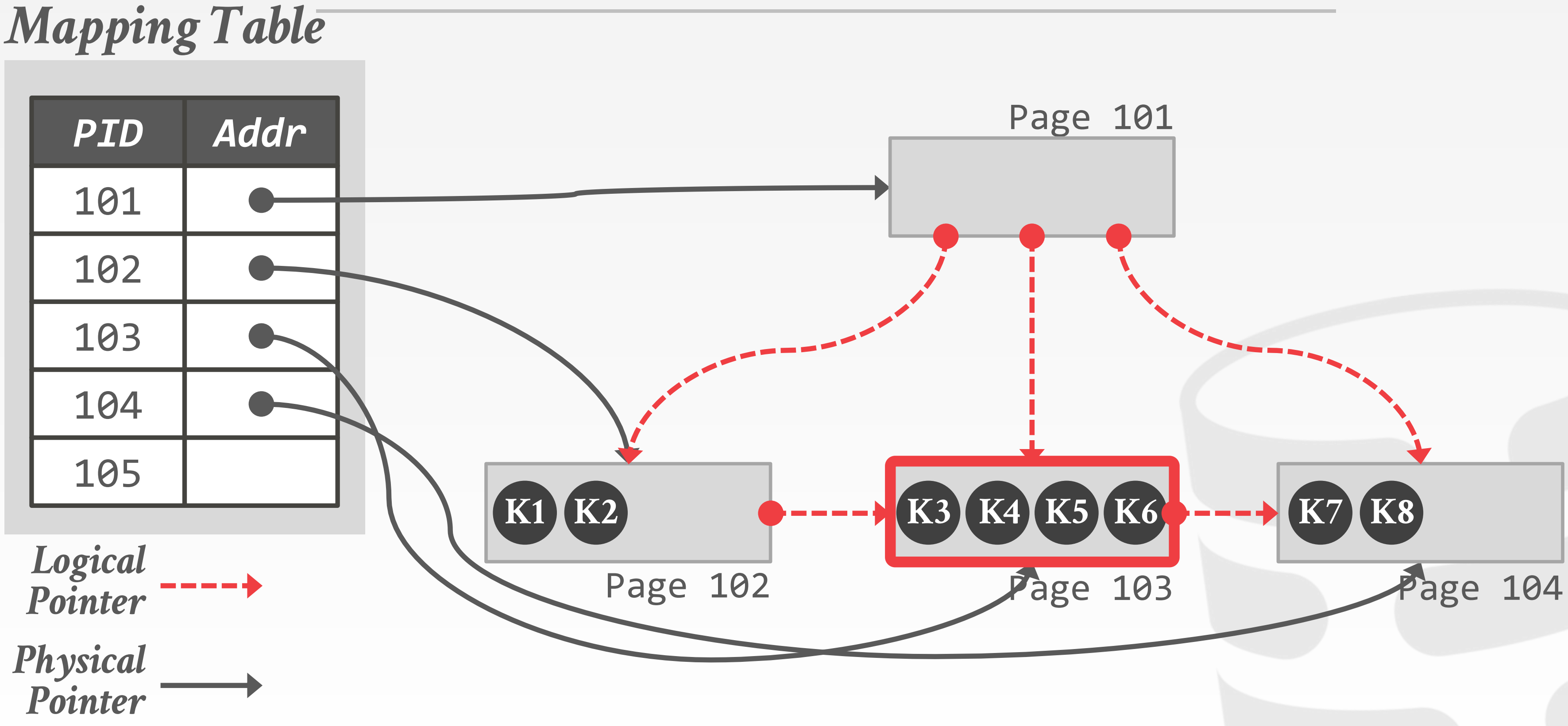
→ Provide a shortcut in the modified page's parent on what ranges to find the new page.

BW-TREE: STRUCTURE MODIFICATIONS

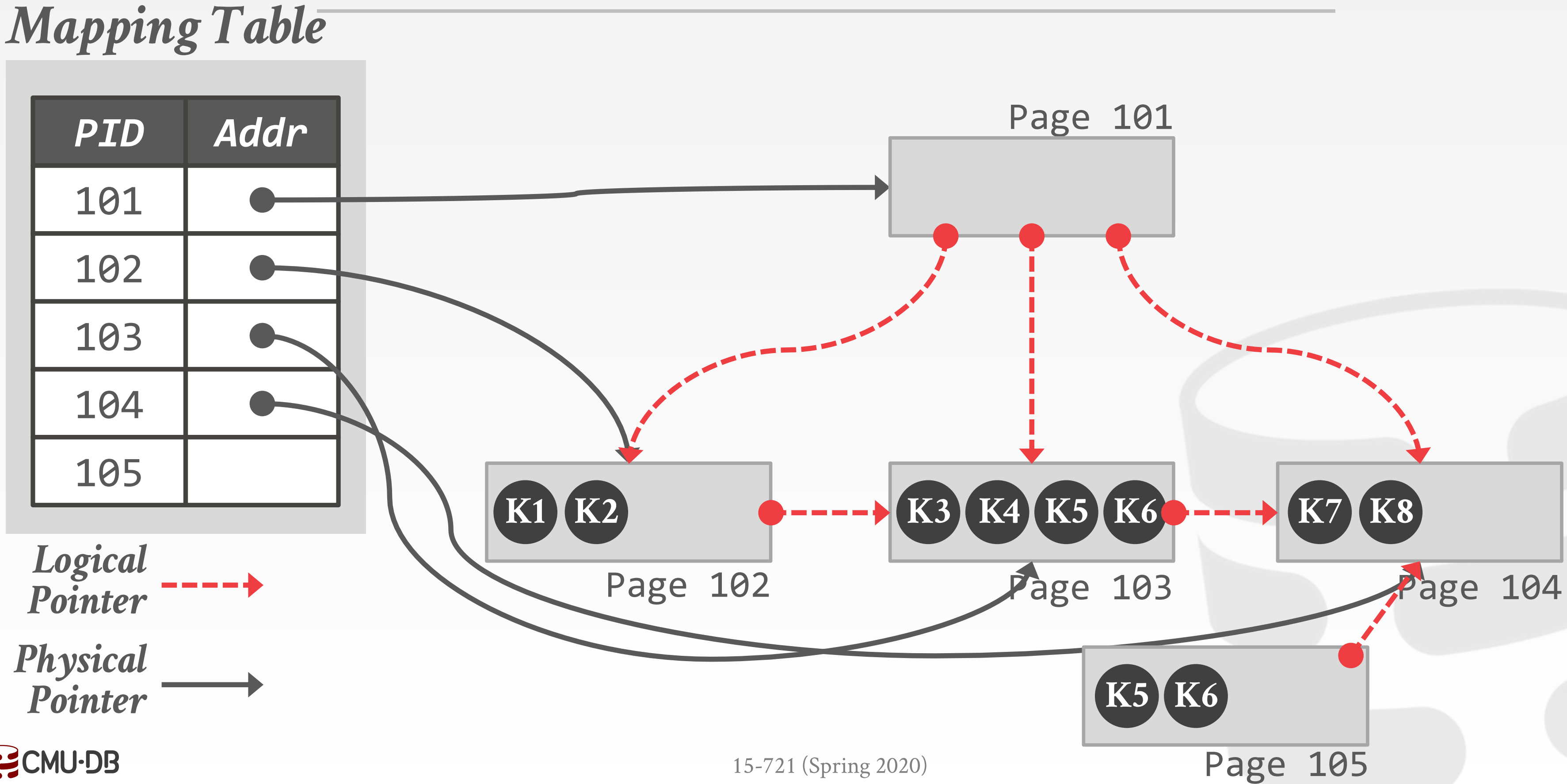
Mapping Table



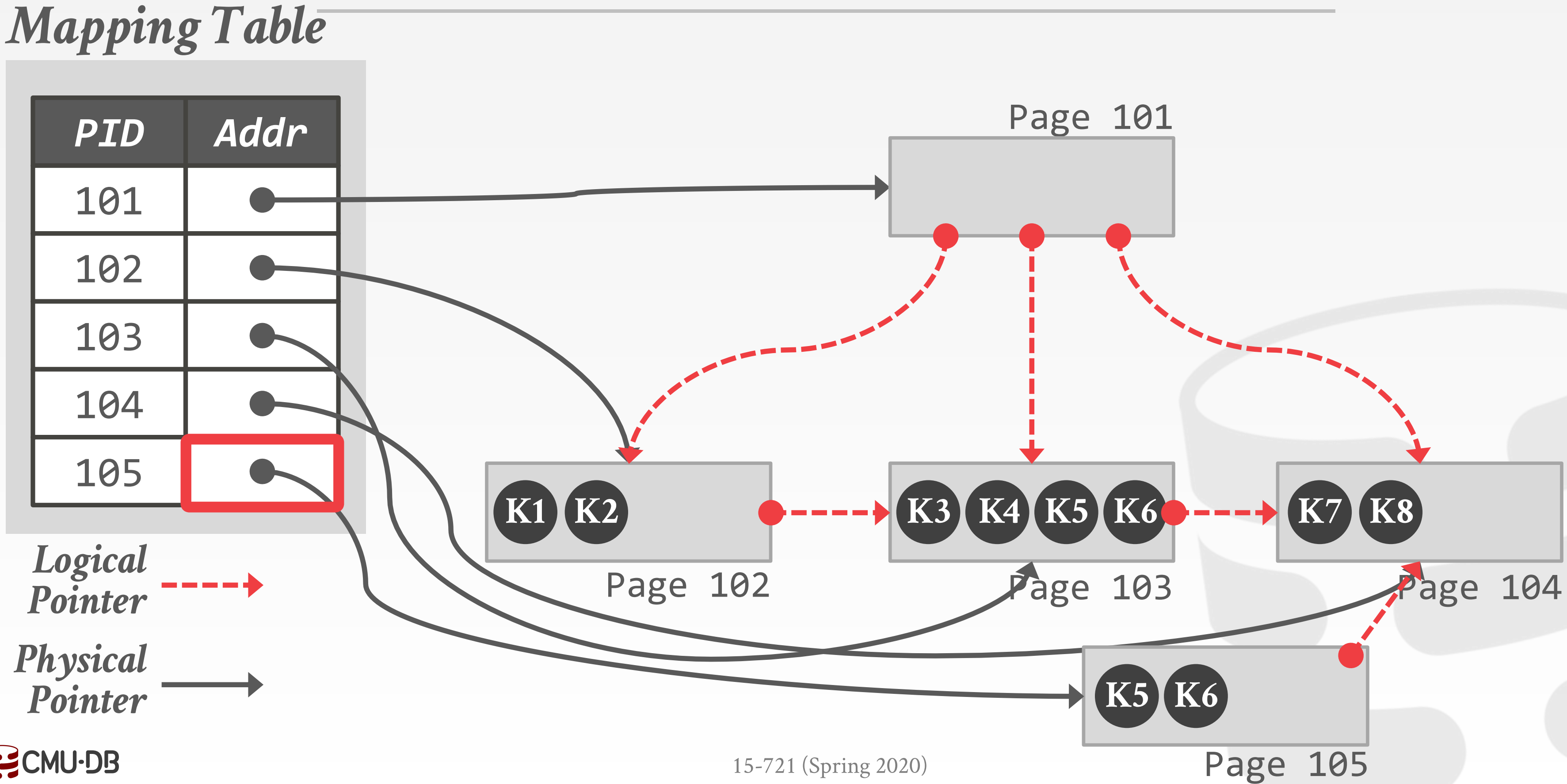
BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS



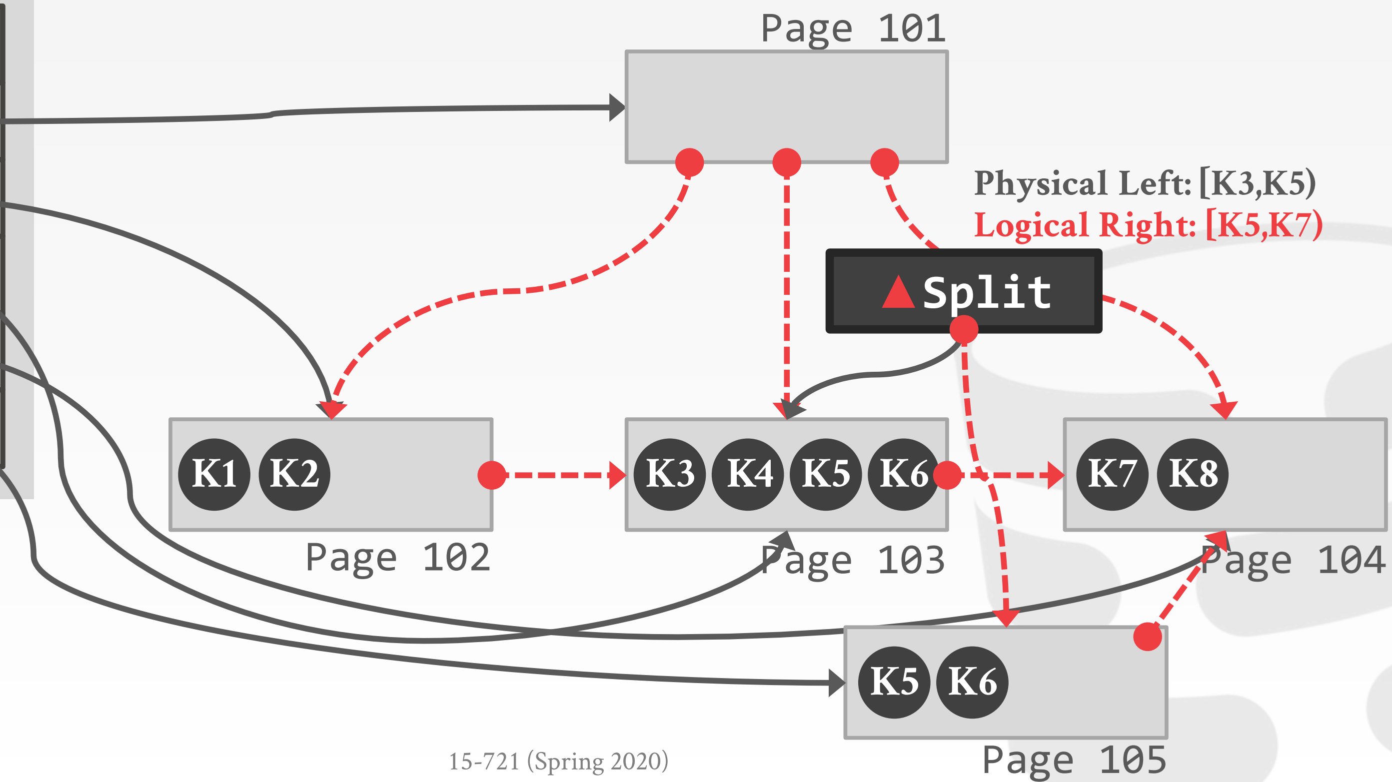
BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

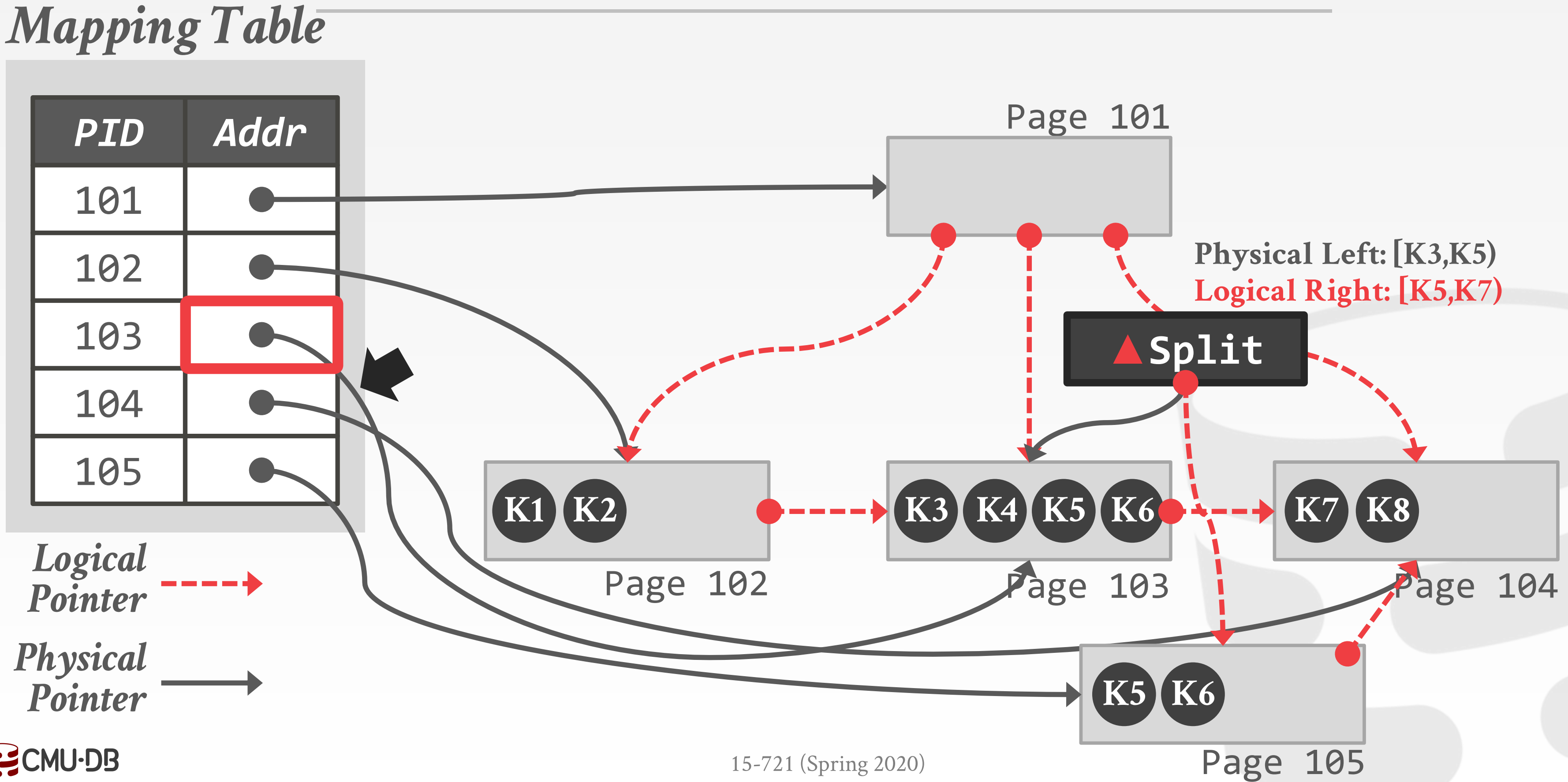
PID	Addr
101	•
102	•
103	•
104	•
105	•



Logical Pointer

Physical Pointer

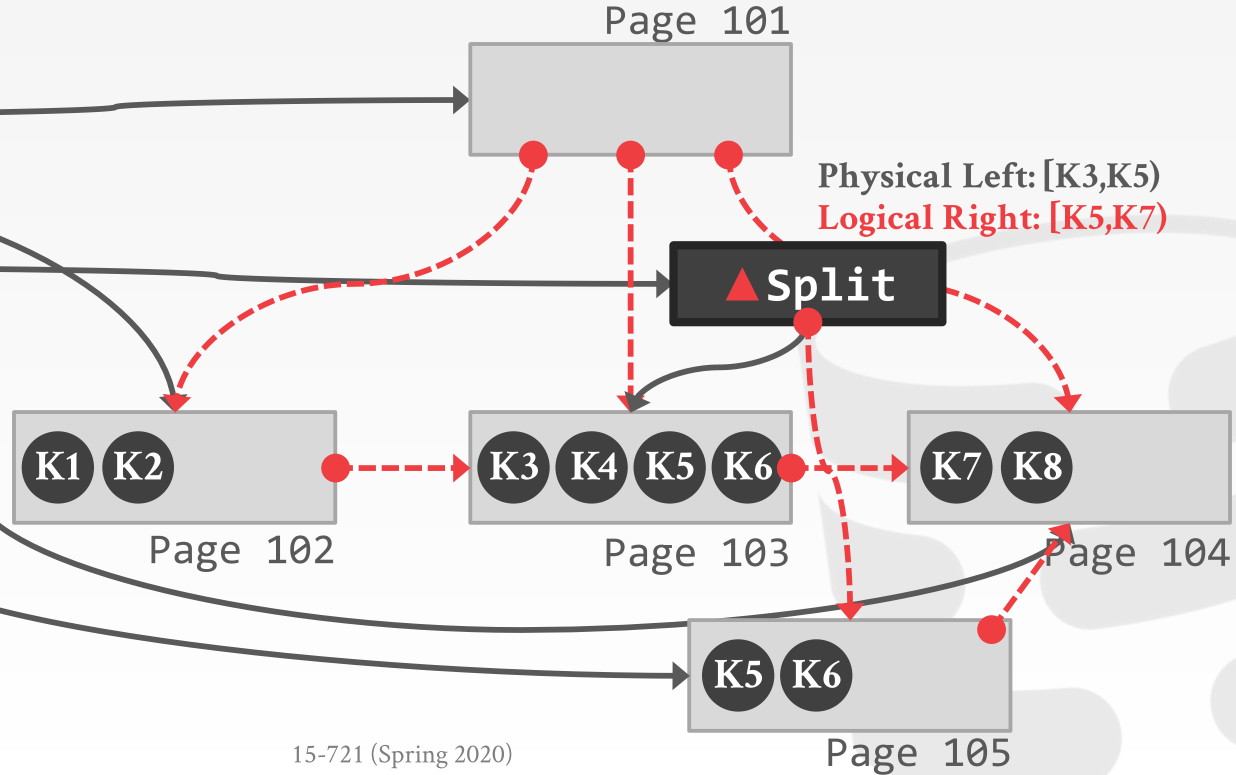
BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

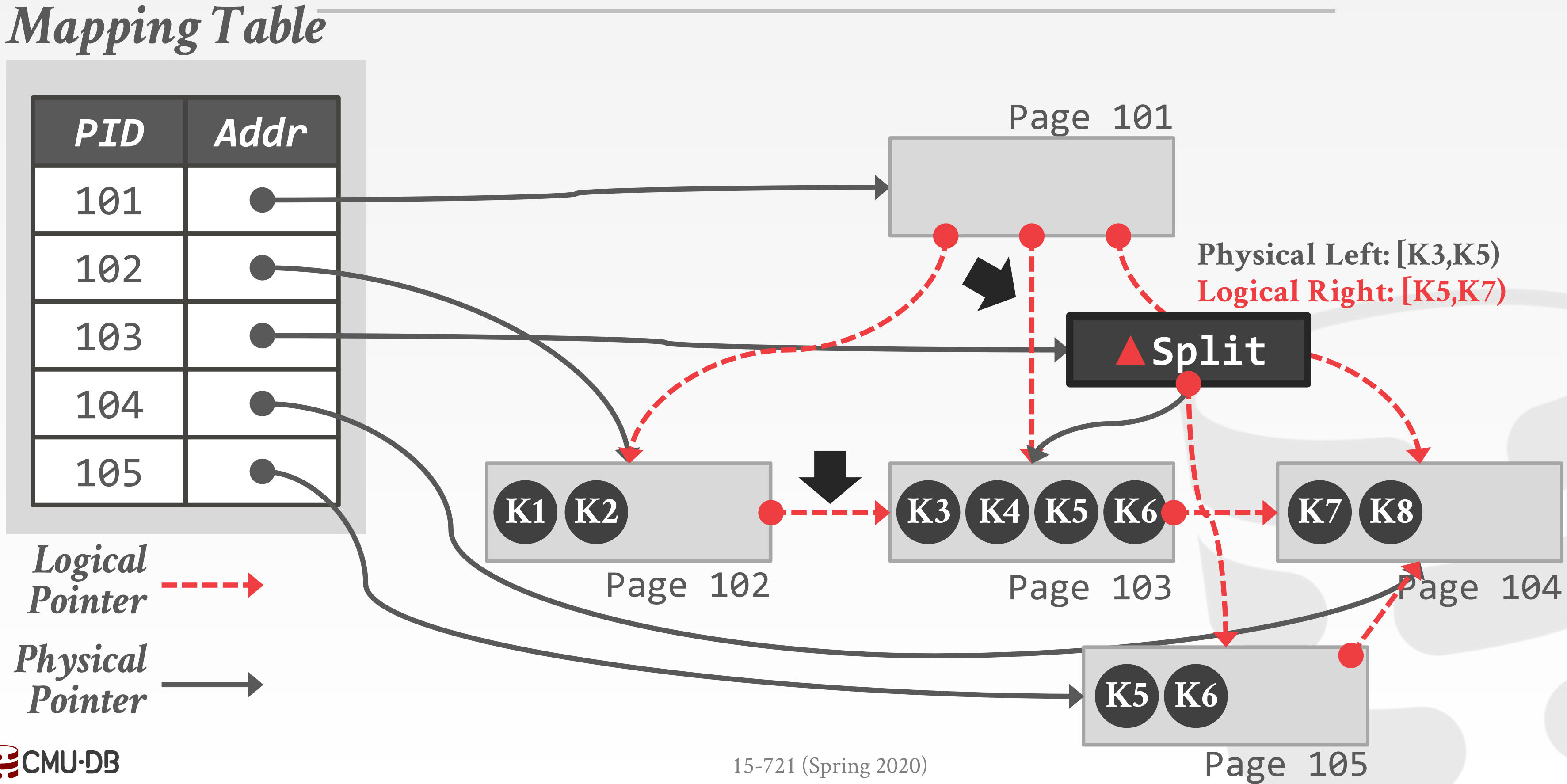
PID	*Addr*
101	●
102	●
103	●
104	●
105	●



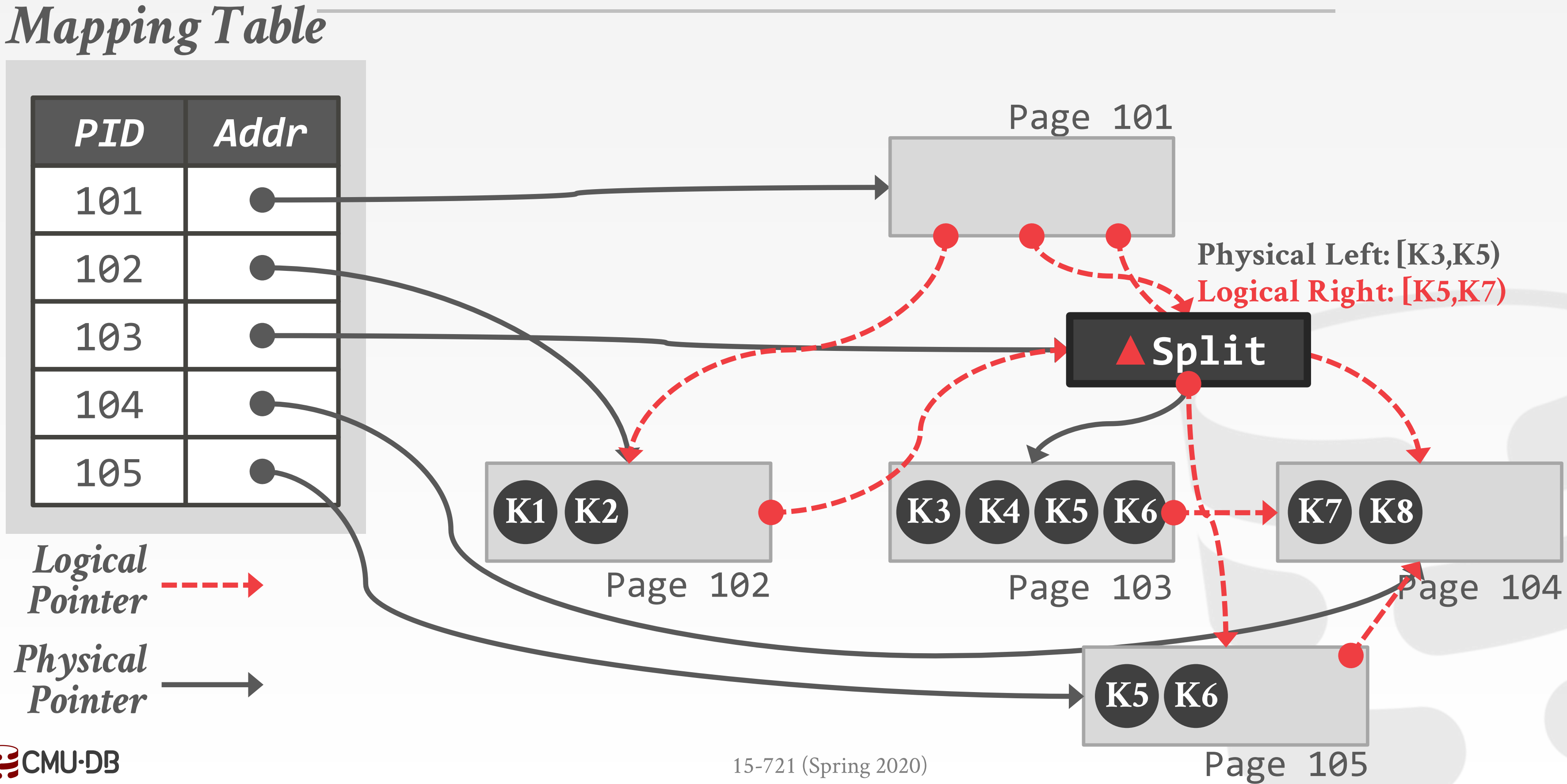
Logical Pointer

Physical Pointer

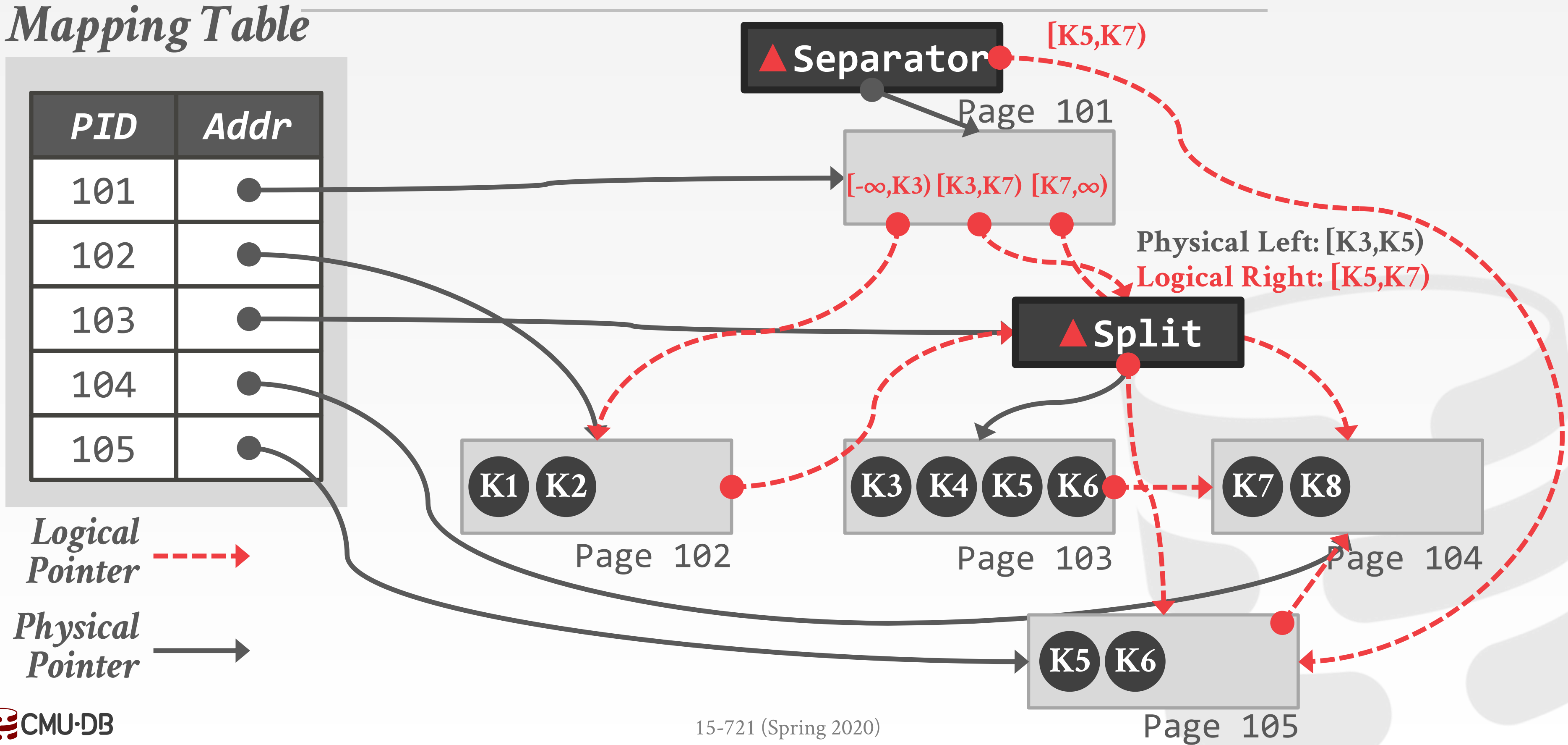
BW-TREE: STRUCTURE MODIFICATIONS



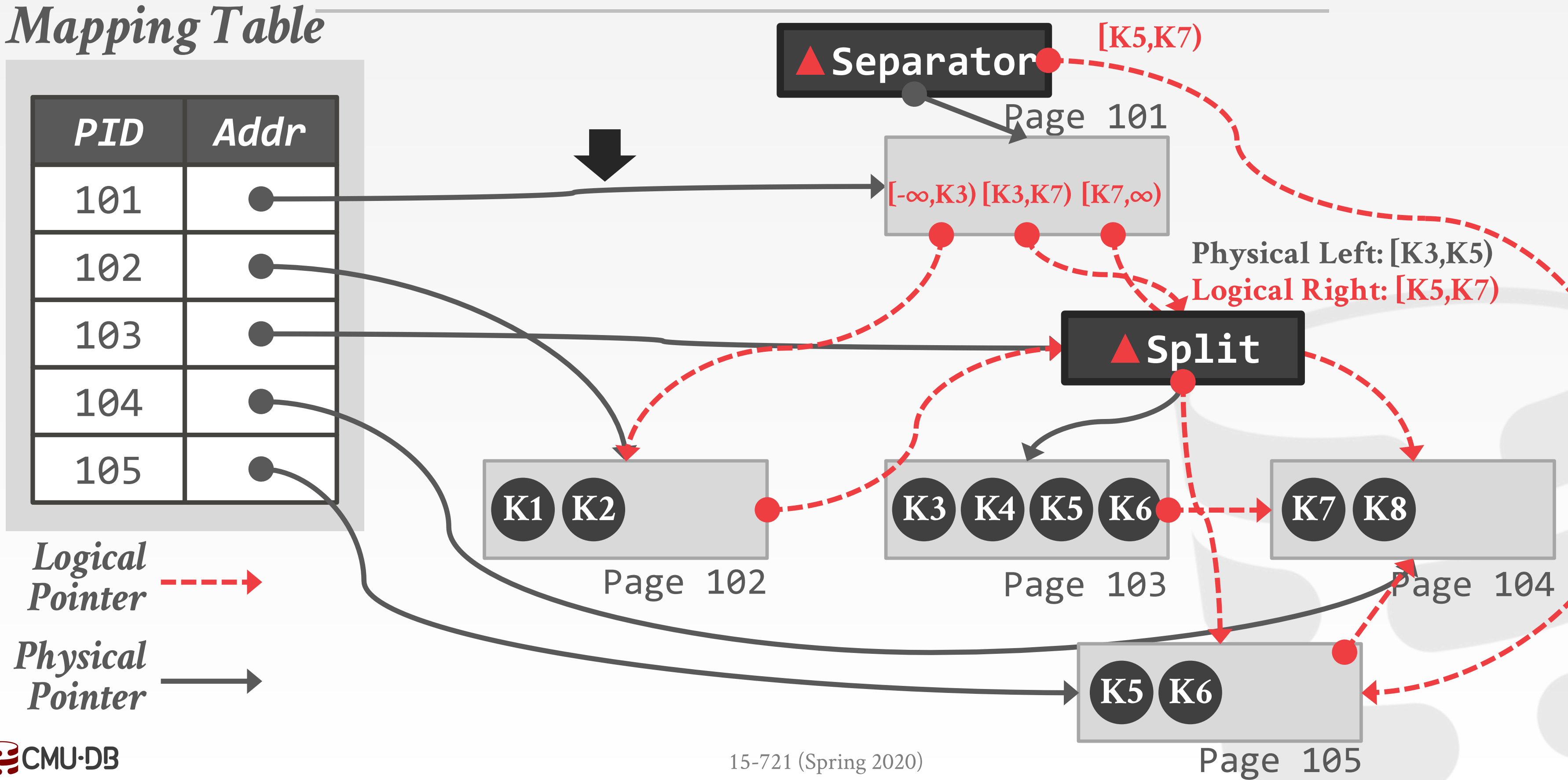
BW-TREE: STRUCTURE MODIFICATIONS



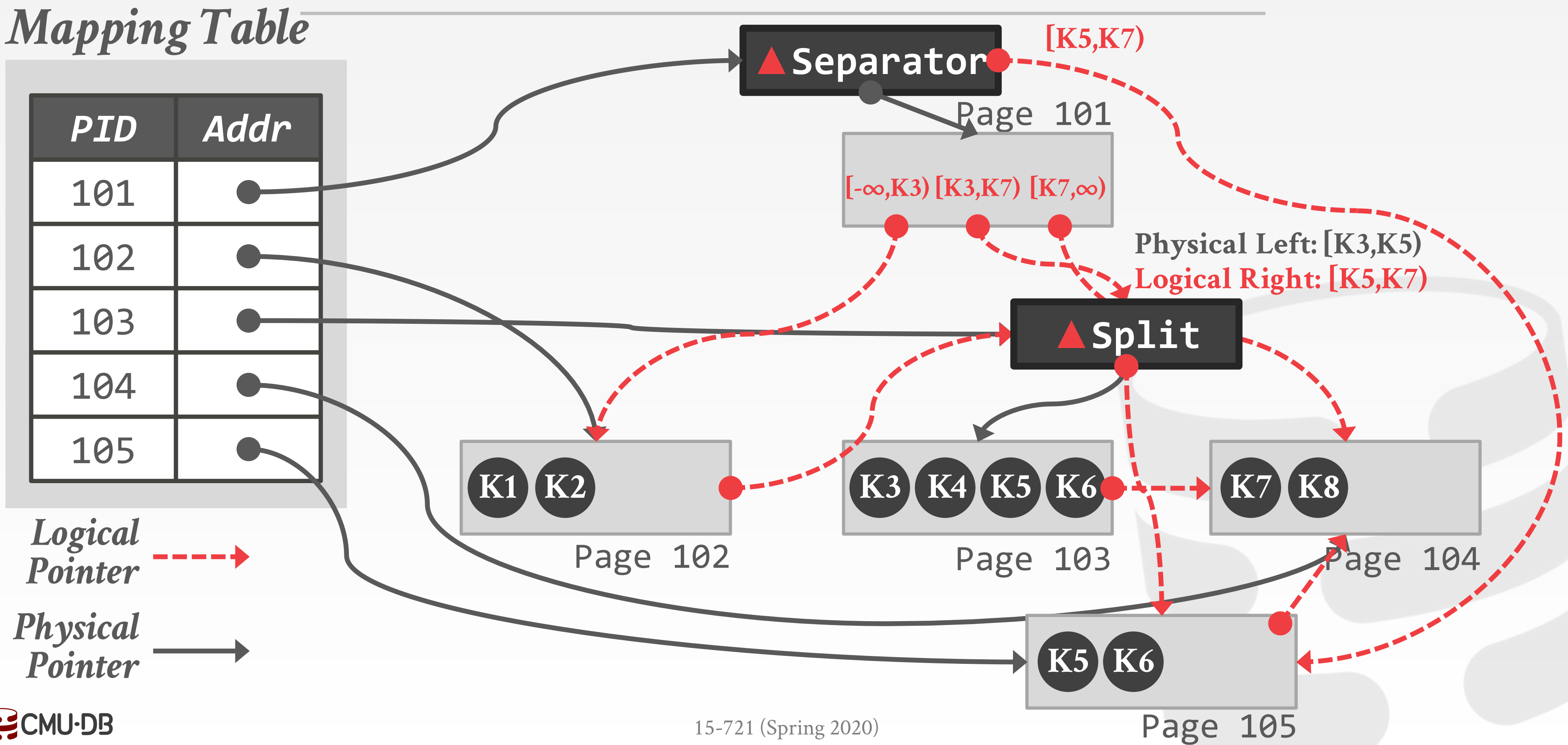
BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS



CMU OPEN BW-TREE

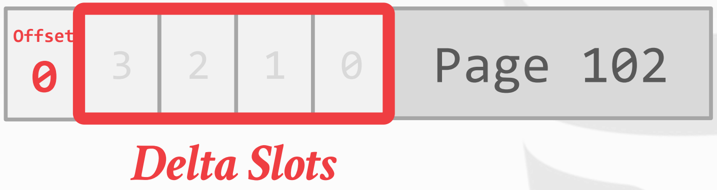
Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.

→ Avoids small object allocation, list traversal.

Mapping Table

PID	Addr
102	



BUILDING THE BW-TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

CMU OPEN BW-TREE

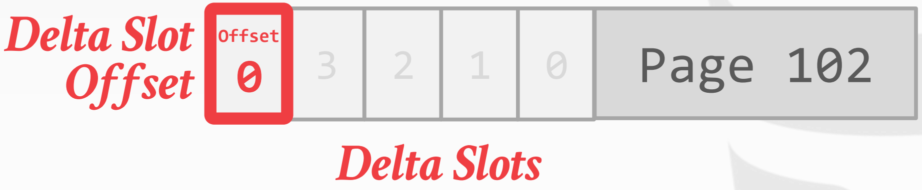
Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.

→ Avoids small object allocation, list traversal.

Mapping Table

PID	Addr
102	

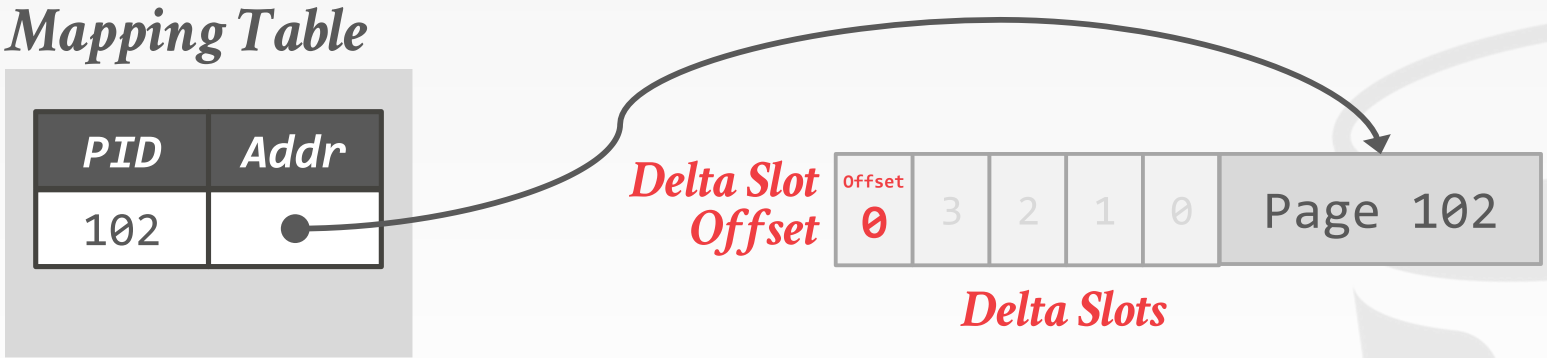


BUILDING THE BW-TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

CMU OPEN BW-TREE

Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.
→ Avoids small object allocation, list traversal.



BUILDING THE BW-TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

CMU OPEN BW-TREE

Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.
→ Avoids small object allocation, list traversal.

BUILDING THE BW-TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

CMU OPEN BW-TREE

Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.
→ Avoids small object allocation, list traversal.

BUILDING THE BW-TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

CMU OPEN BW-TREE

Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.
→ Avoids small object allocation, list traversal.

BUILDING THE BW-TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

CMU OPEN BW-TREE

Optimization #2: Mapping Table Expansion

→ Fastest associative data structure is a plain array.

→ Allocating the full array for each index is wasteful

→ Old Peloton: 1m nodes per index = 8MB

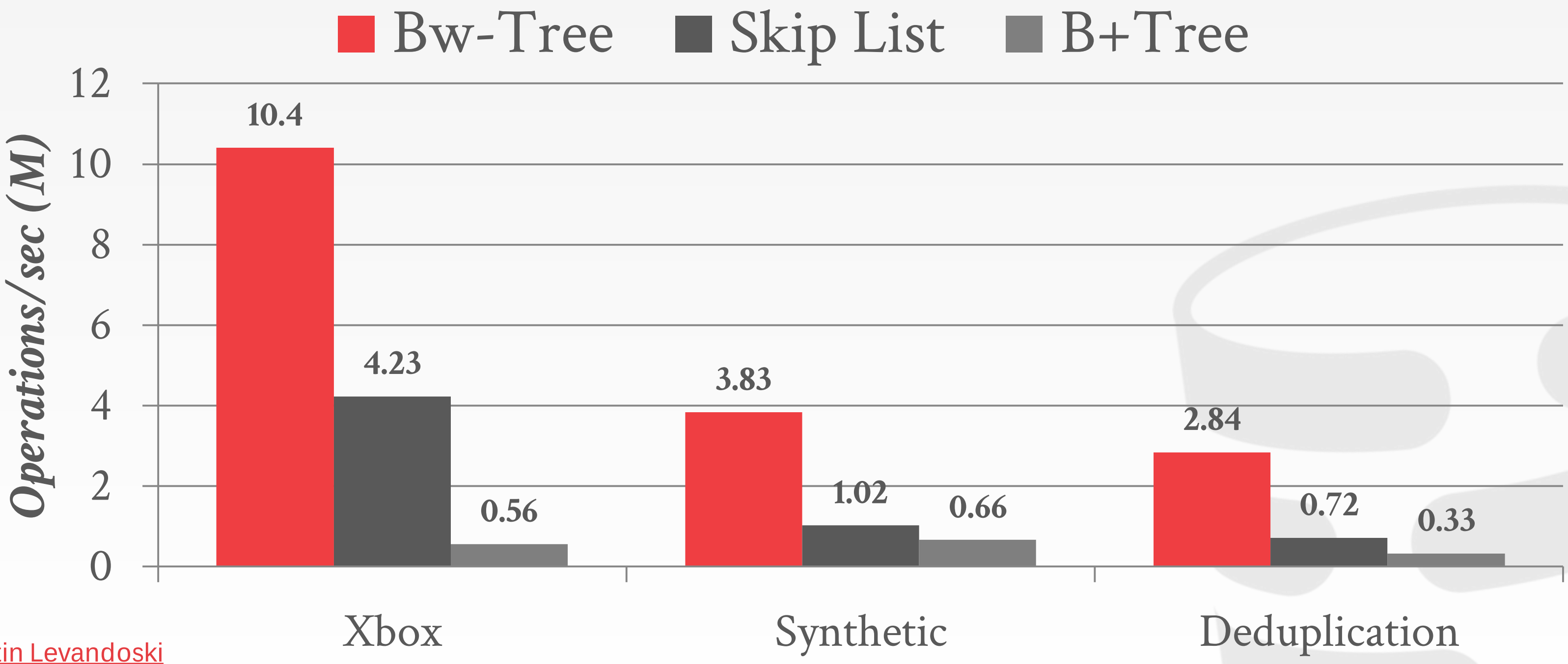
Use virtual memory to allocate the entire array without backing it with physical memory.

→ Only need to allocate physical memory when threads access higher offsets in the array.

BUILDING THE BW-TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

BW-TREE: PERFORMANCE

Processor: 1 socket, 4 cores w/ 2×HT



Source: Justin Levandoski

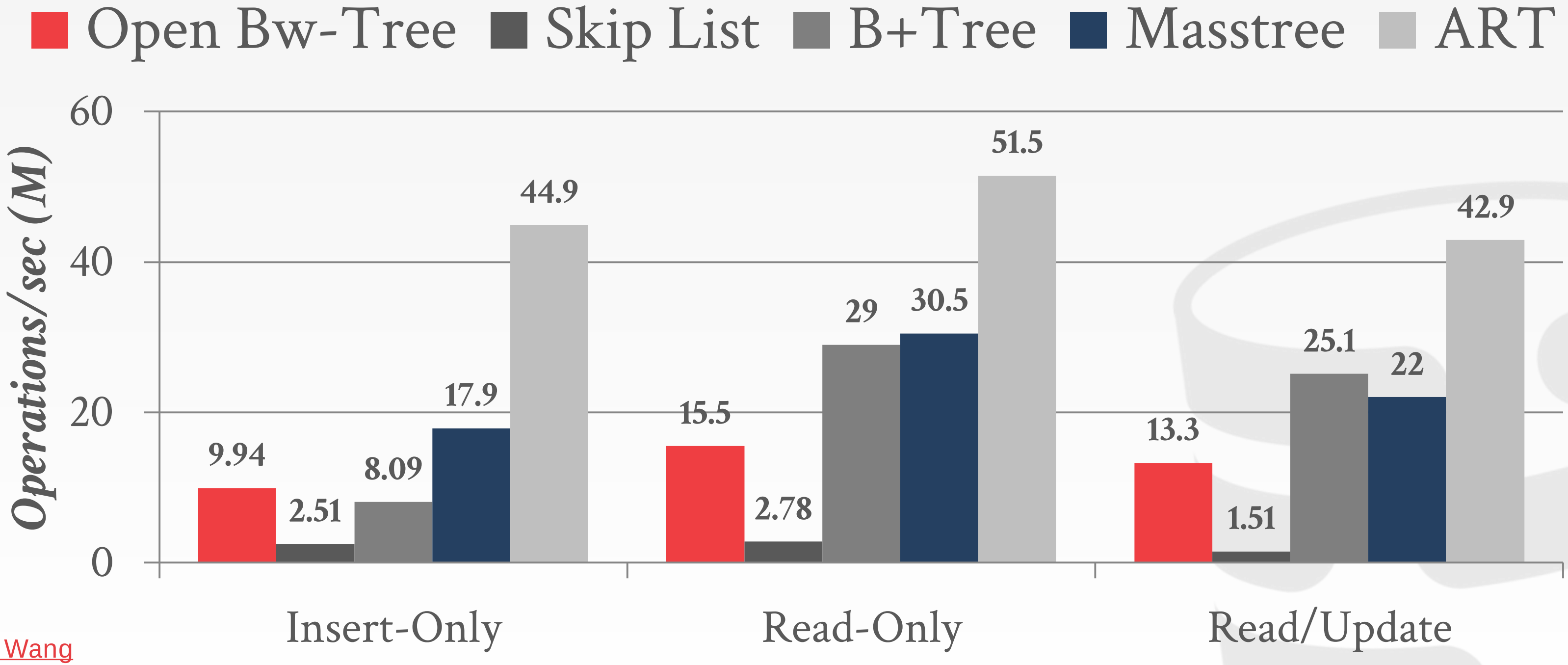
BW-TREE: PERFORMANCE

Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)

Source: Ziqi Wang

BW-TREE: PERFORMANCE

Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)



Source: Ziqi Wang

PARTING THOUGHTS

Managing a concurrent index looks a lot like managing a database.

A Bw-Tree is hard to implement.

Versioned latch coupling provides some the benefits of optimistic methods with wasting too much work.

NEXT CLASS

Latch Implementations

Trie Data Structures