Lecture #07

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

OLTP Indexes (Latch-Free Data Structures)

@Andy_Pavlo // 15-721 // Spring 2019

TODAY'S AGENDA

T-Tree
Skip List
Bw-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?

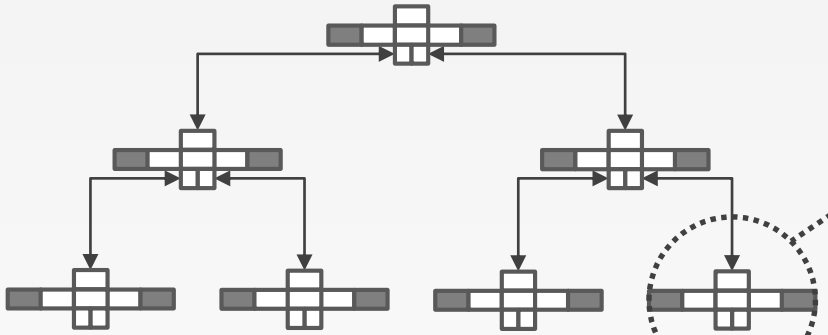
T-TREES

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

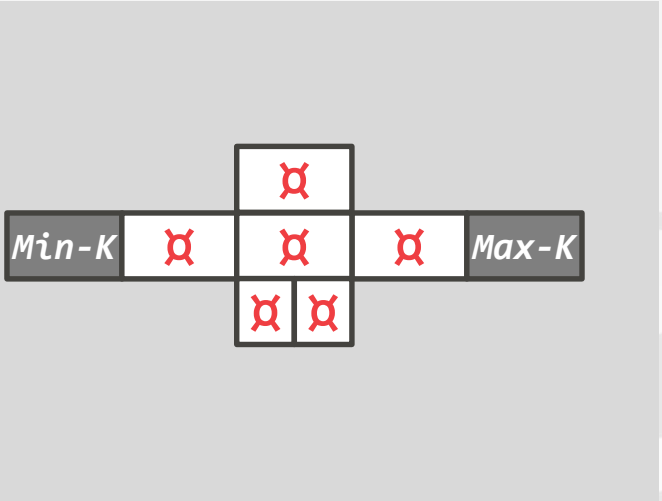
Proposed in 1986 from Univ. of Wisconsin
Used in TimesTen and other early in-memory DBMSs during the 1990s.

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

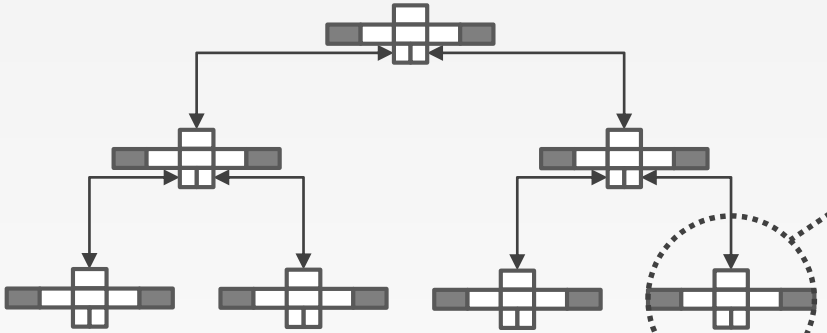
T-TREES



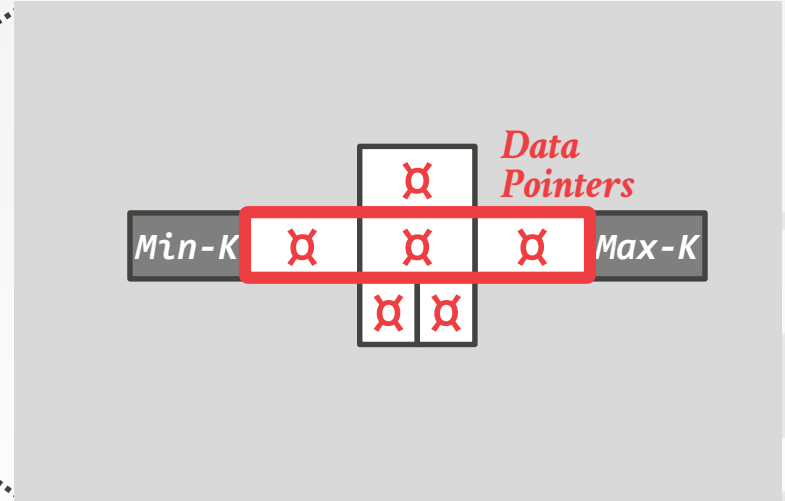
T-Tree Node



T-TREES



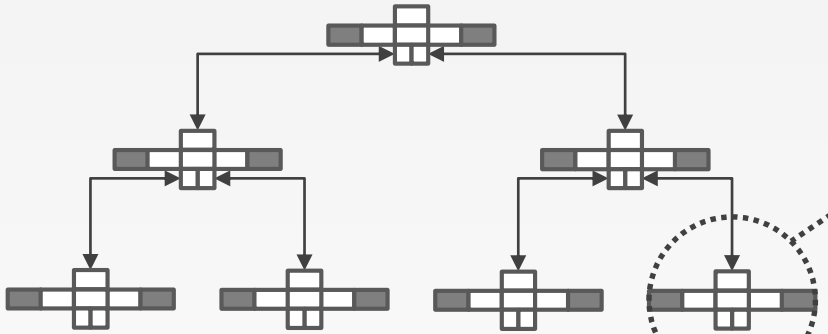
T-Tree Node



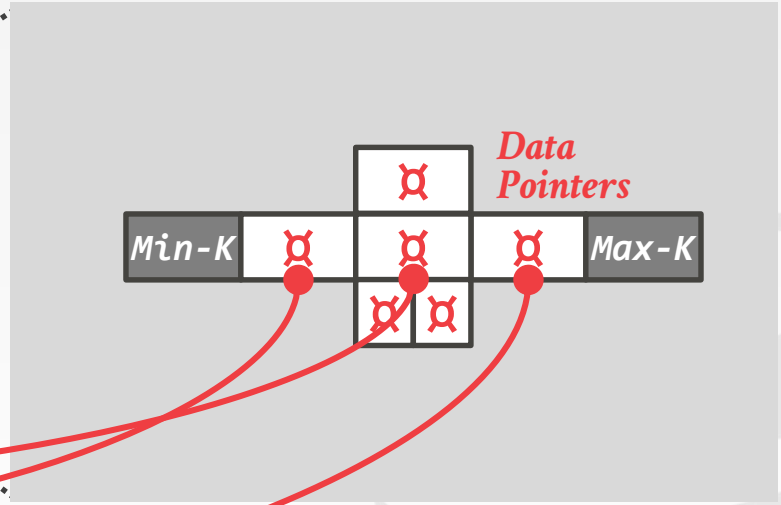
Data Table

KEY	DATA
K5	-
K2	-
K9	-
K8	-
⋮	⋮

T-TREES



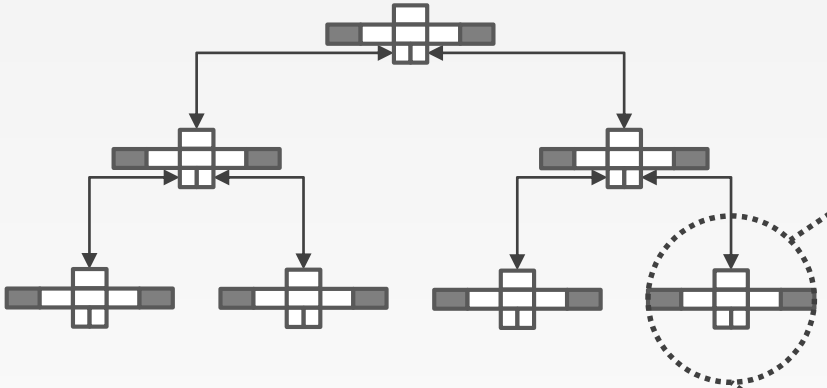
T-Tree Node



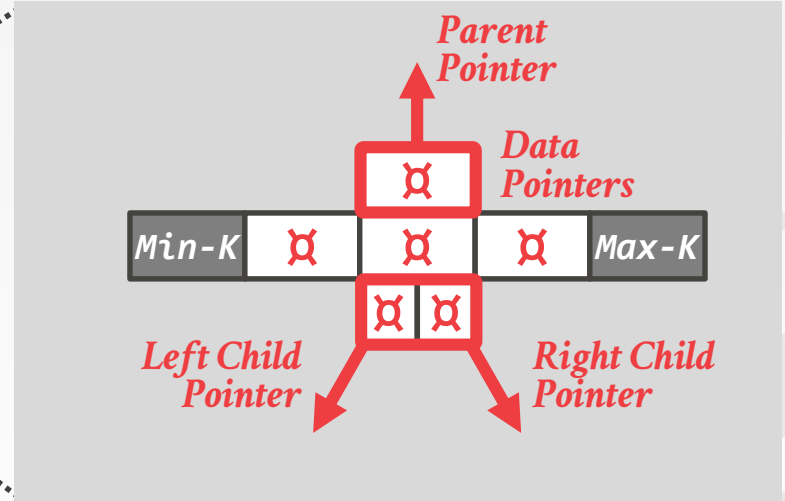
Data Table

KEY	DATA
K5	-
K2	-
K9	-
K8	-
⋮	⋮

T-TREES



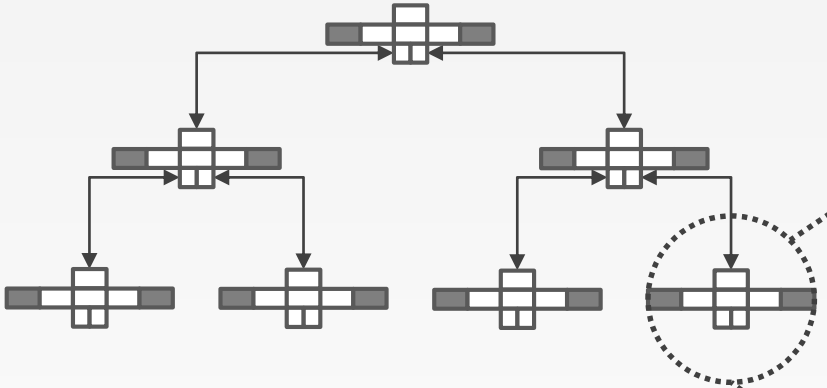
T-Tree Node



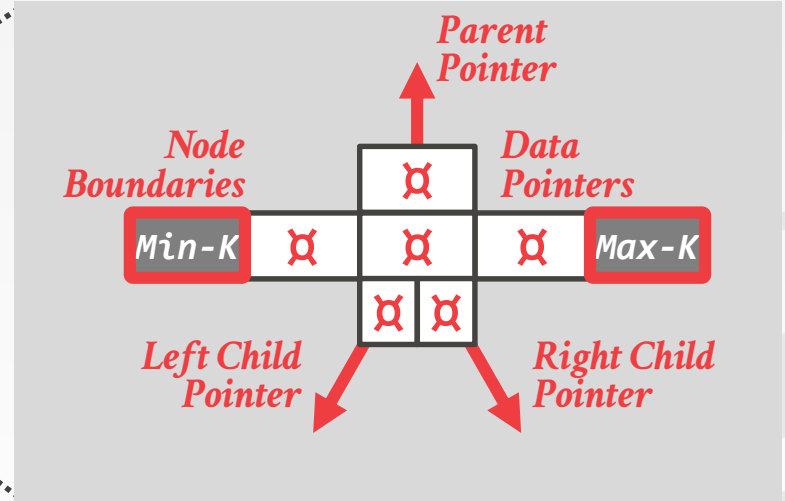
Data Table

KEY	DATA
K5	-
K2	-
K9	-
K8	-
⋮	⋮

T-TREES



T-Tree Node

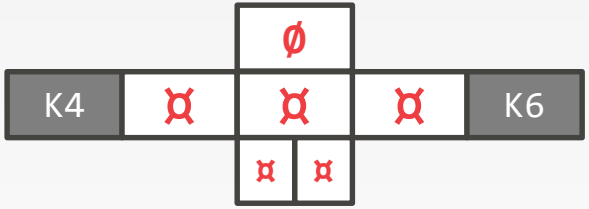


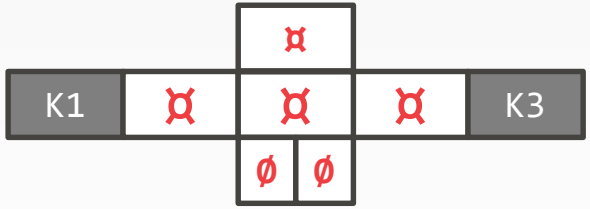
Data Table

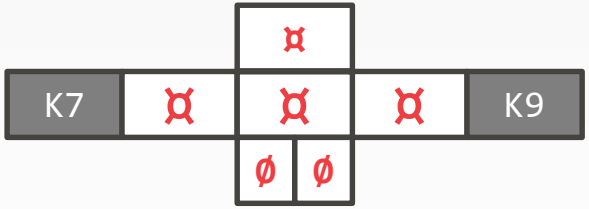
KEY	DATA
K5	-
K2	-
K9	-
K8	-
⋮	⋮

T-TREES: SEARCH

Find K2





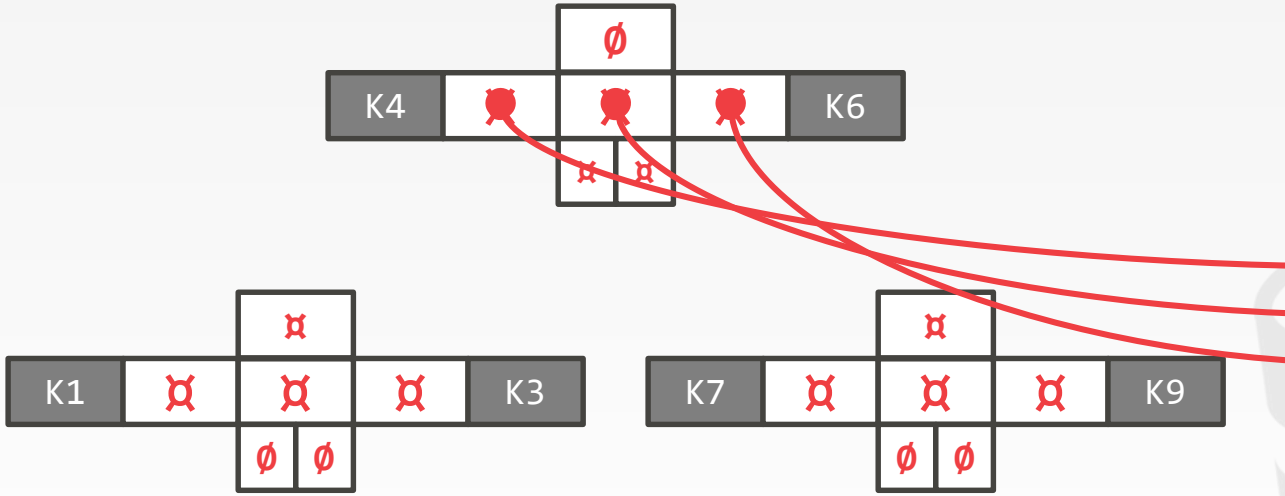


Data Table

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

T-TREES: SEARCH

Find K2

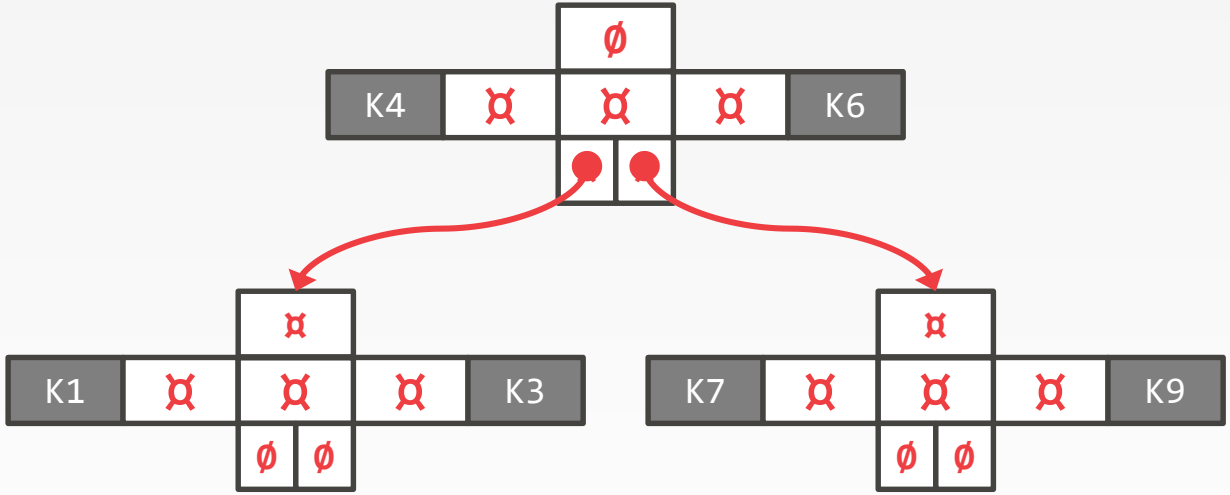


Data Table

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

T-TREES: SEARCH

Find K2

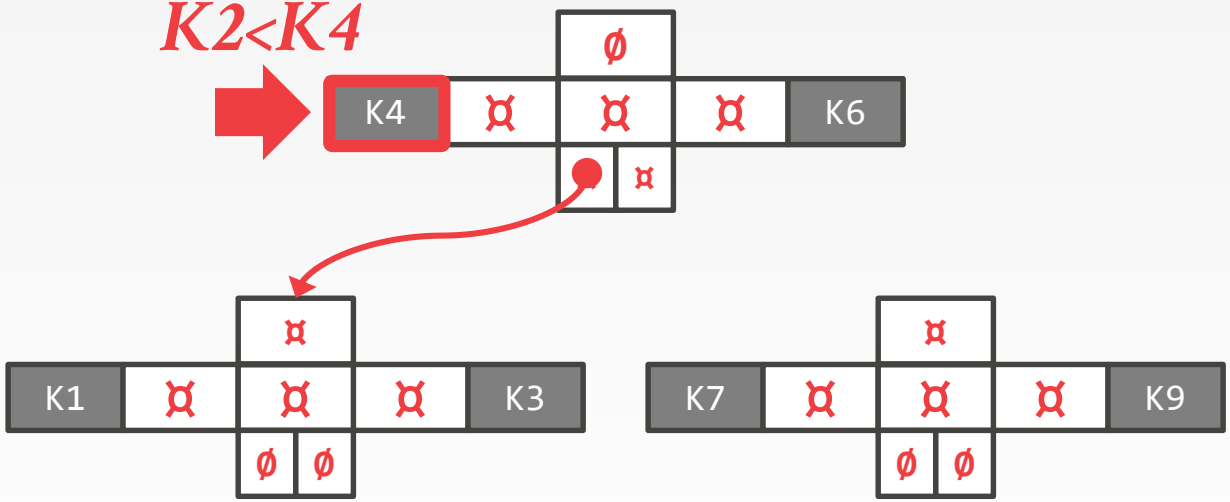


Data Table

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

T-TREES: SEARCH

Find K2

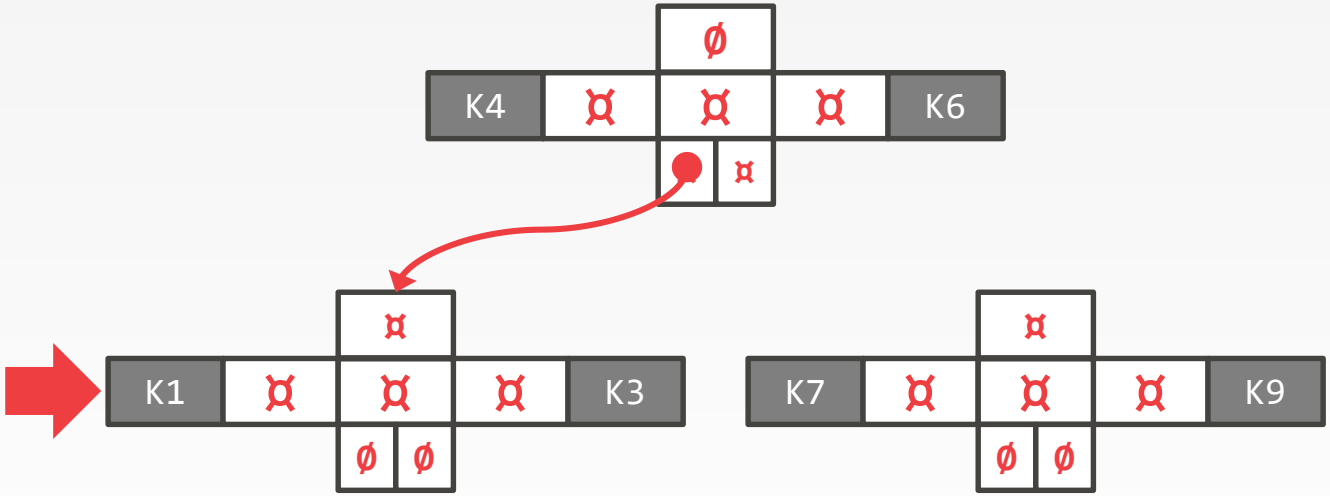


Data Table

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

T-TREES: SEARCH

Find K2

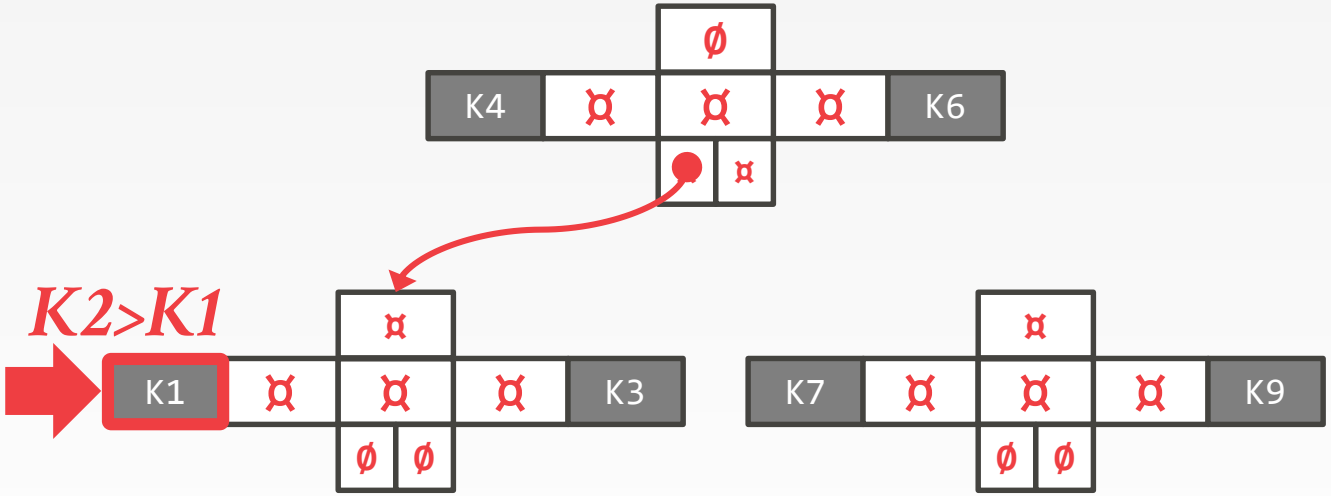


Data Table

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

T-TREES: SEARCH

Find K2

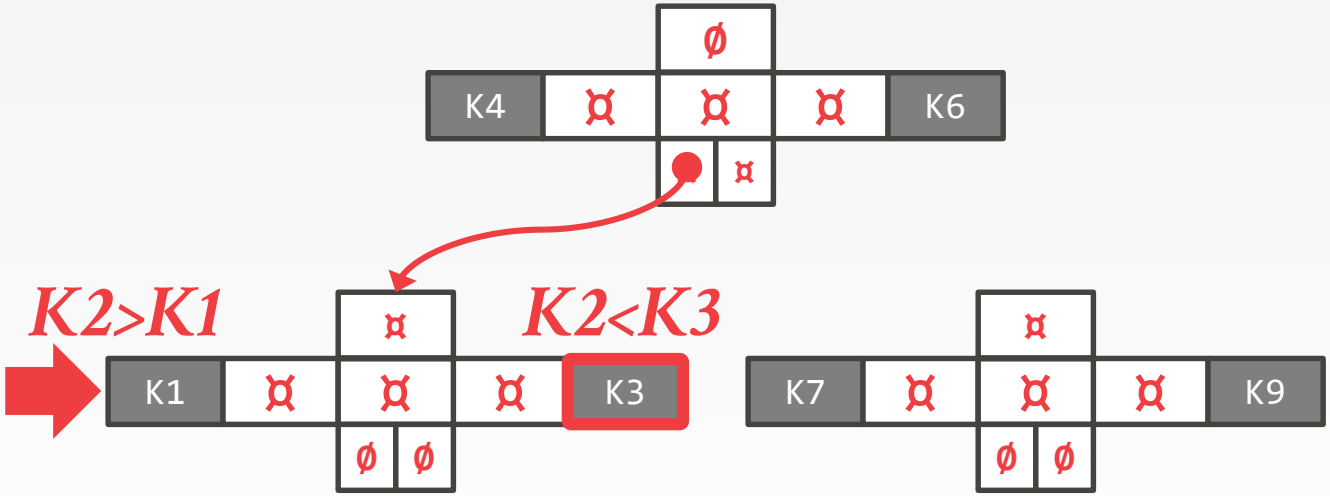


Data Table

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

T-TREES: SEARCH

Find K2

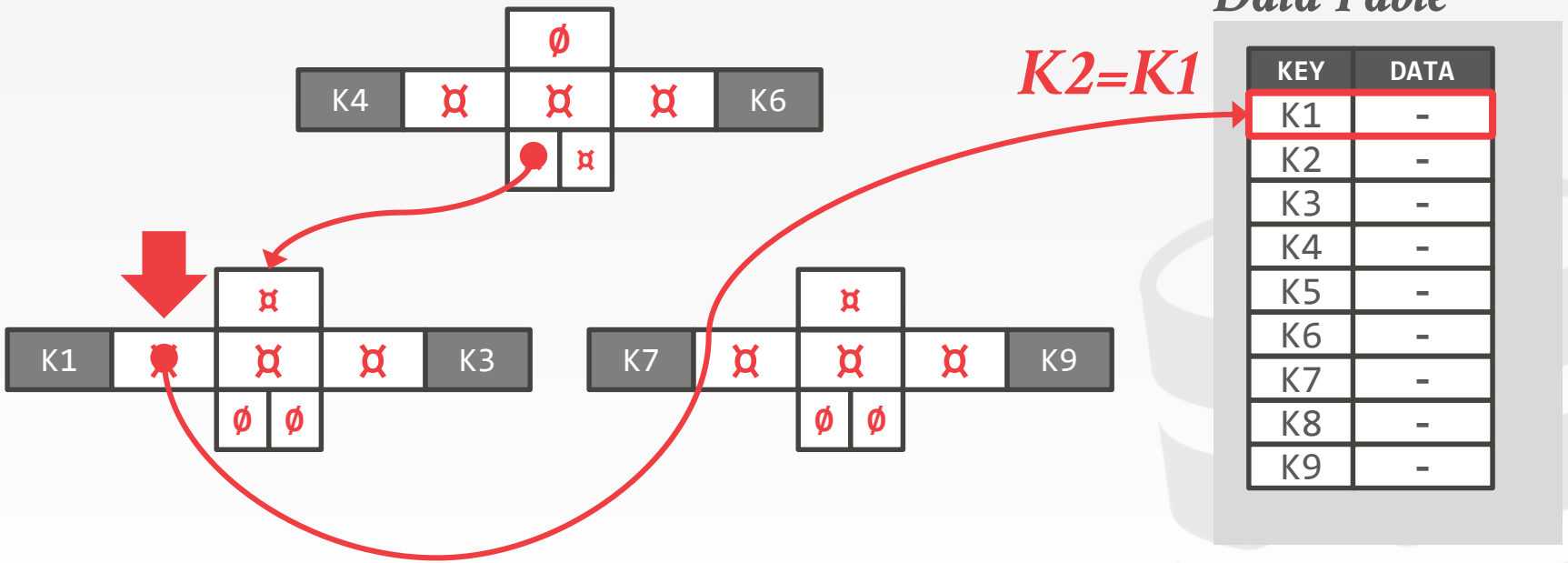


Data Table

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

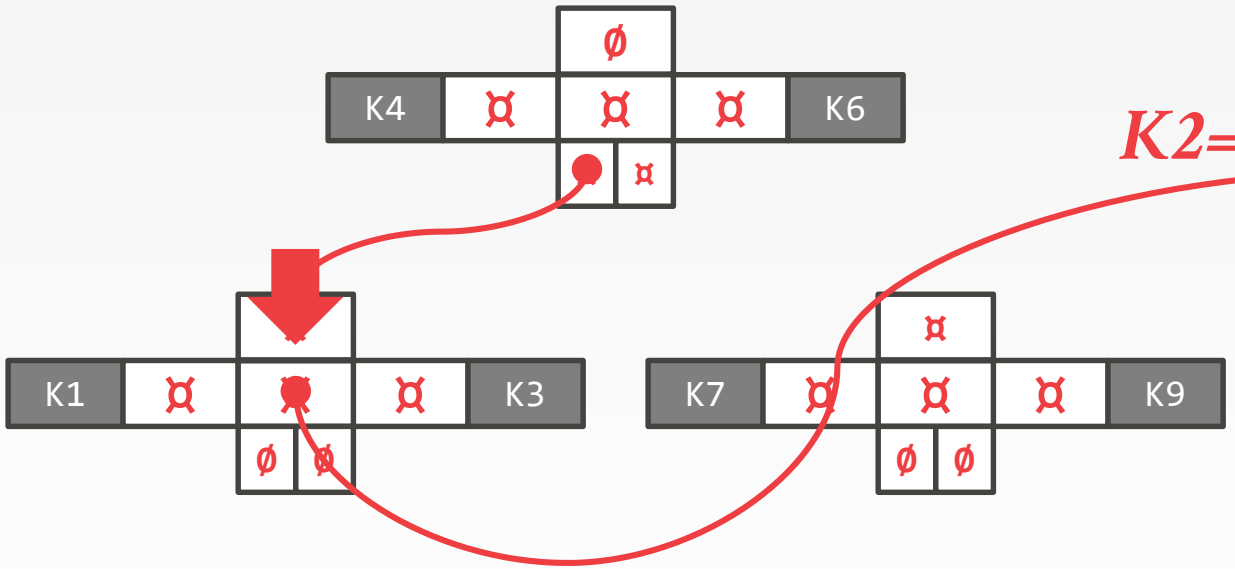
T-TREES: SEARCH

Find K2



T-TREES: SEARCH

Find K2



Data Table

K2=K2

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

T-TREES

Advantages

→ Uses less memory because it does not store keys inside of each node.
→ Inner nodes contain key/value pairs (like B-Tree).

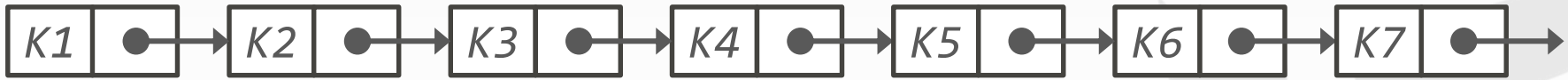
Disadvantages

→ Difficult to rebalance.
→ Difficult to implement safe concurrent access.
→ Have to chase pointers when scanning range or performing binary search inside of a node.

OBSERVATION

The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list.

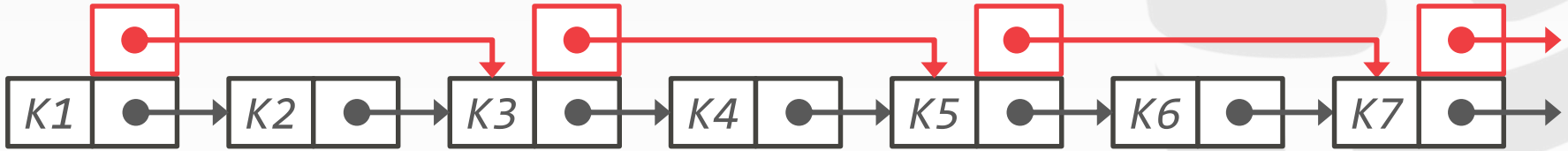
All operations have to linear search.
→ Average Cost: O(N)



OBSERVATION

The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list.

All operations have to linear search.
→ Average Cost: O(N)

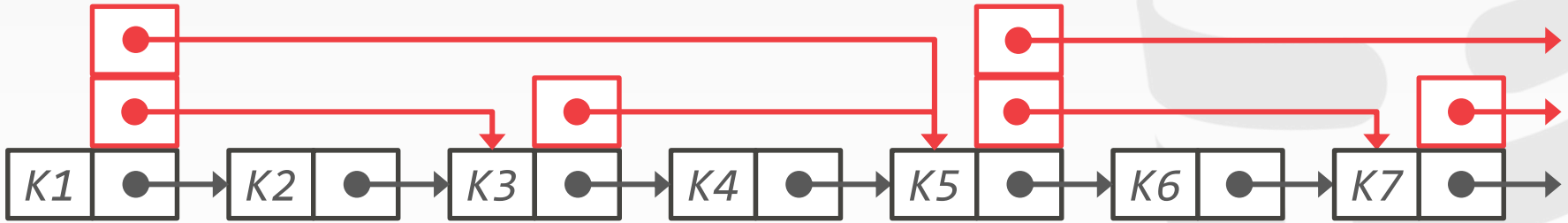


OBSERVATION

The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list.

All operations have to linear search.

→ Average Cost: O(N)



SKIP LISTS

Multiple levels of linked lists with extra pointers that skip over intermediate nodes.

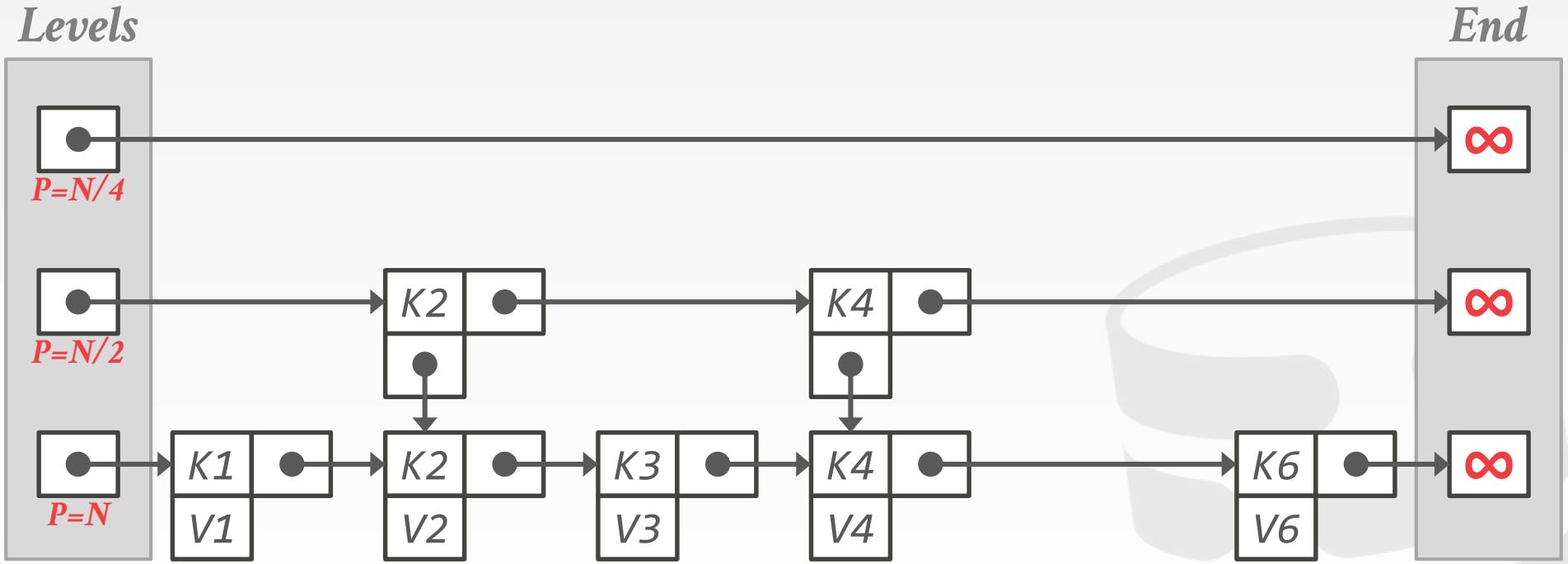
→ Lowest level is a sorted, singly linked list of all keys
→ 2nd level links every other key
→ 3rd level links every fourth key
→ In general, a level has half the keys of one below it

To insert a new key, flip a coin to decide how many levels to add the new key into.

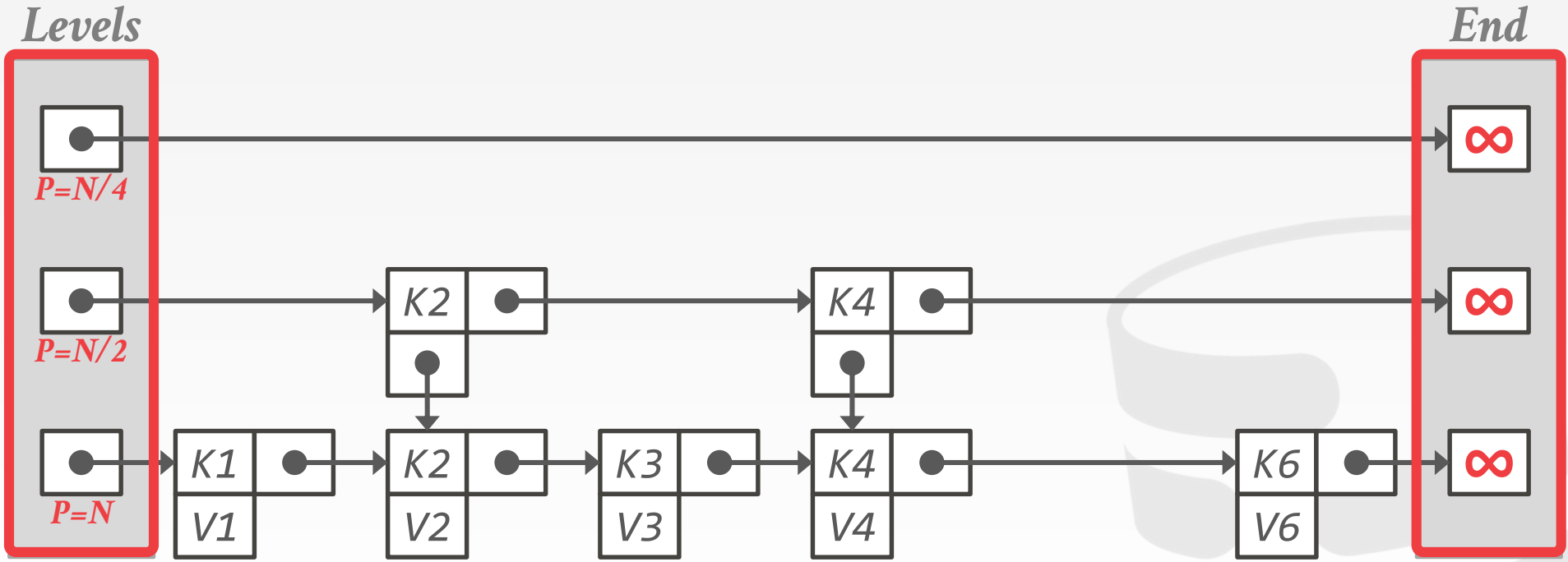
→ Provides approximate O(log n) search times.

SKIP LISTS: A PROBABILISTIC ALTERNATIVE TO BALANCED TREES
CACM VOLUME 33 ISSUE 6 1990

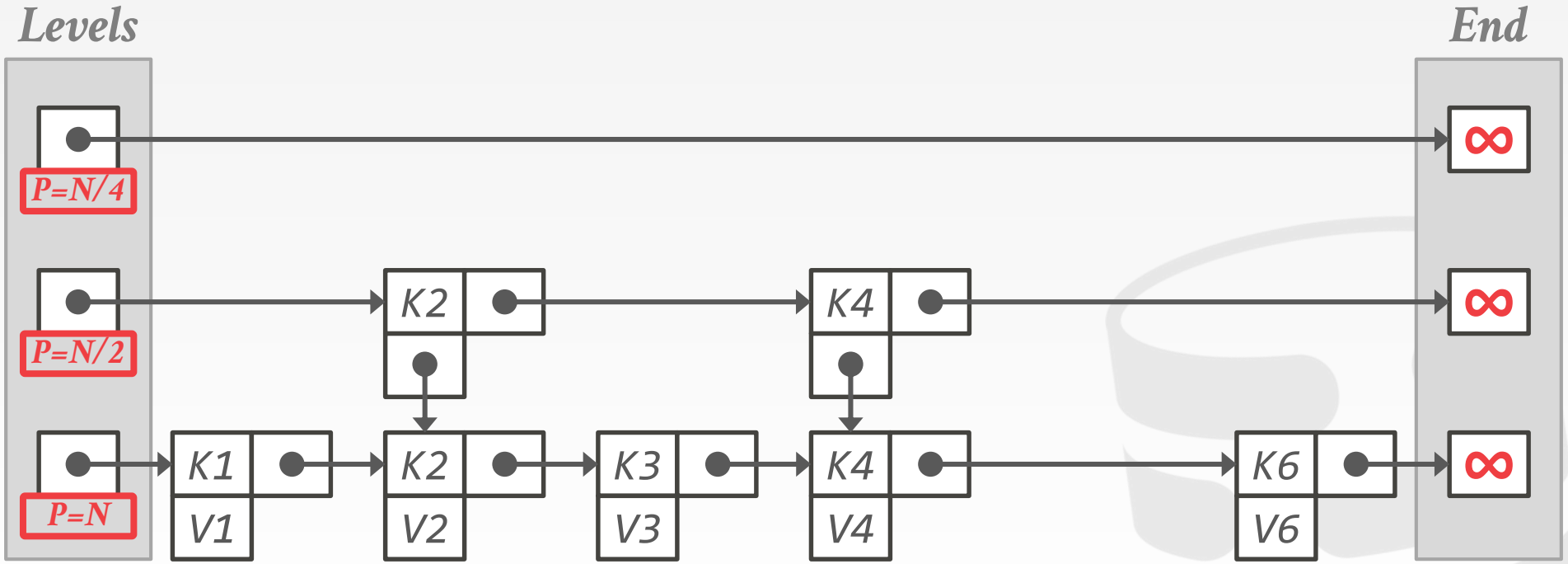
SKIP LISTS: EXAMPLE



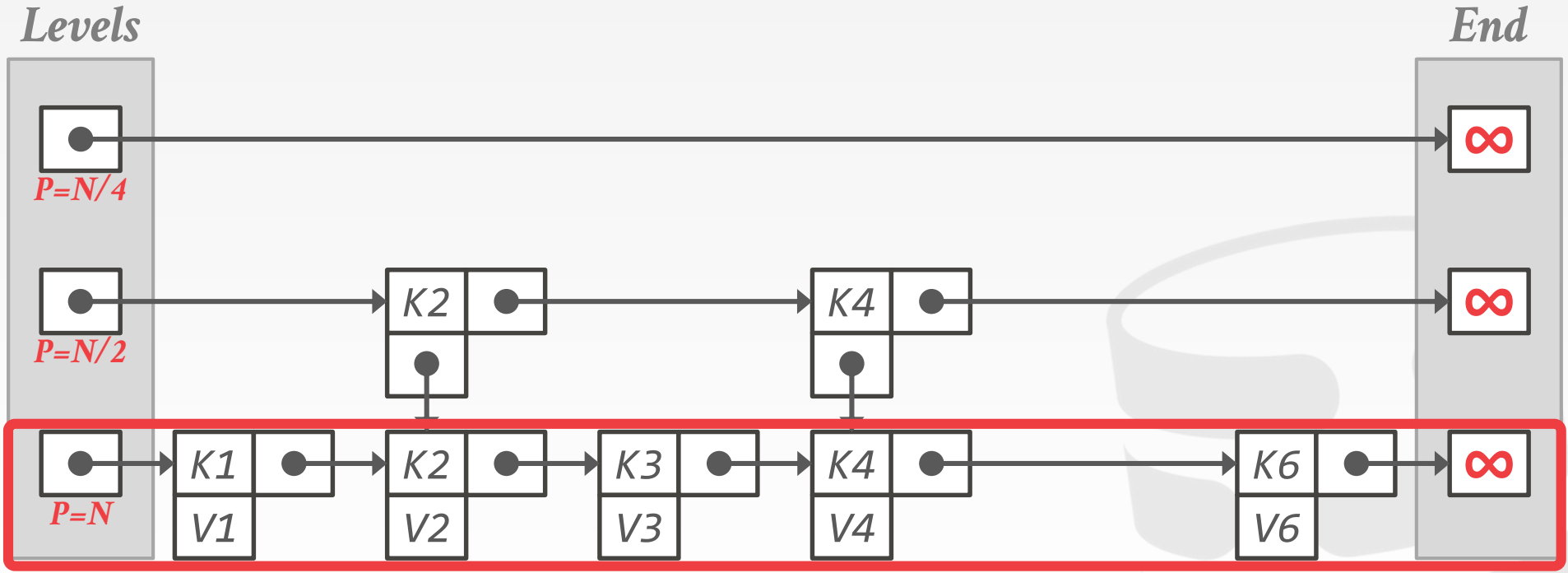
SKIP LISTS: EXAMPLE



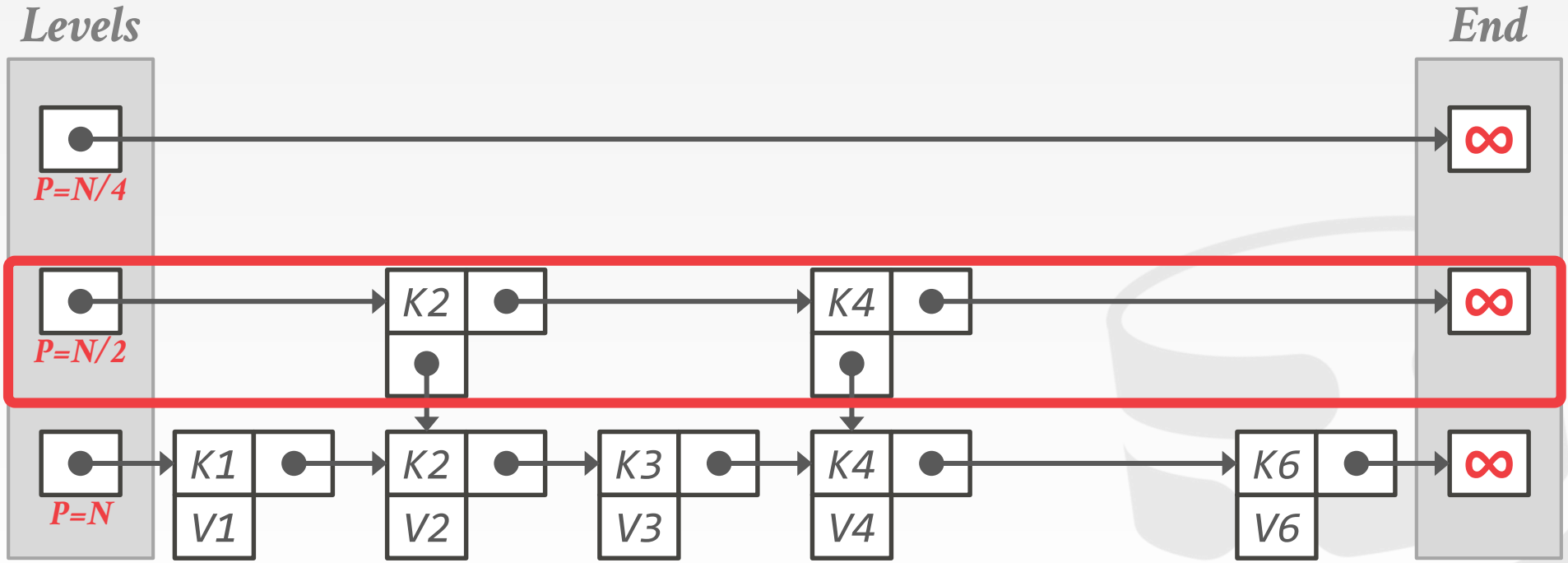
SKIP LISTS: EXAMPLE



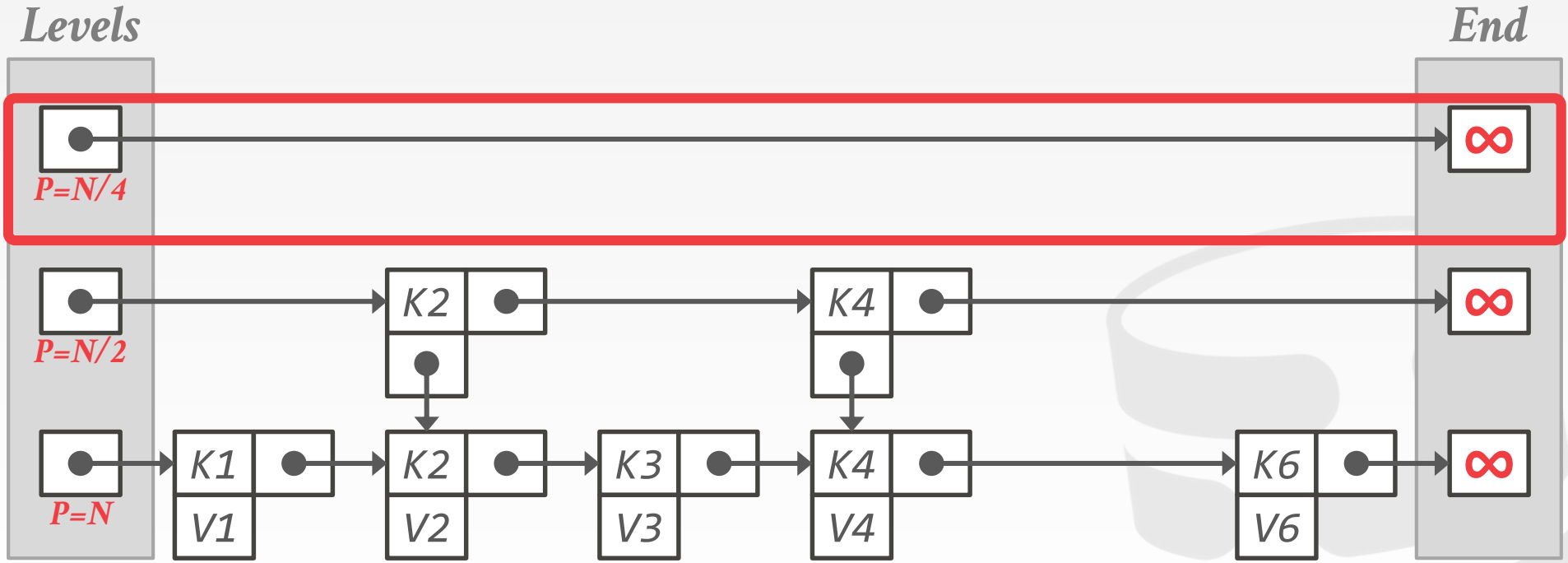
SKIP LISTS: EXAMPLE



SKIP LISTS: EXAMPLE

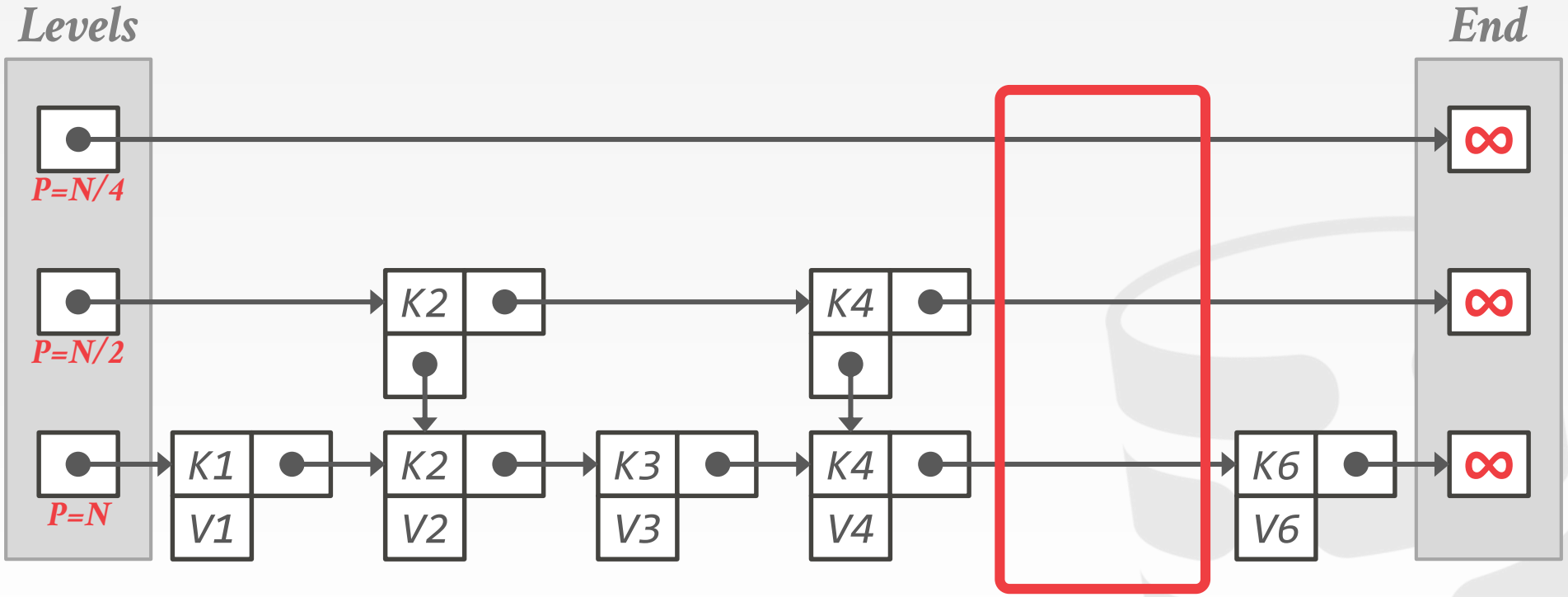


SKIP LISTS: EXAMPLE



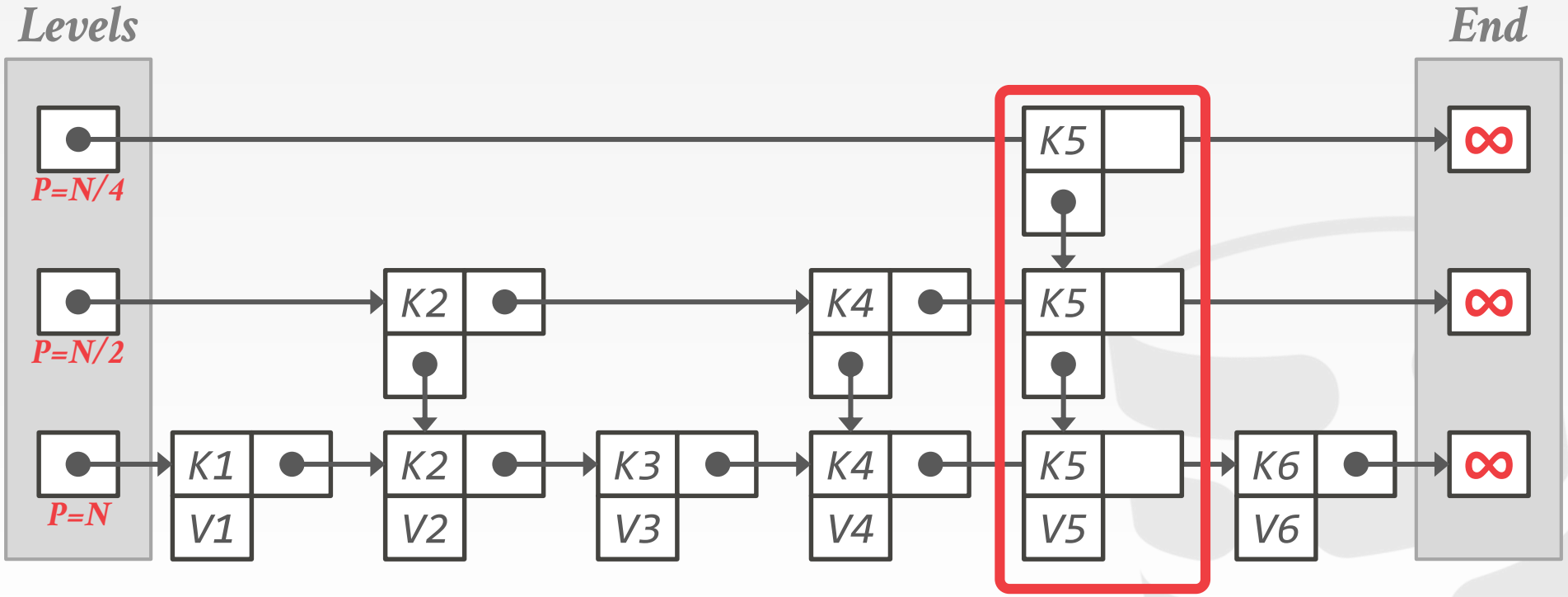
SKIP LISTS: INSERT

Insert K5



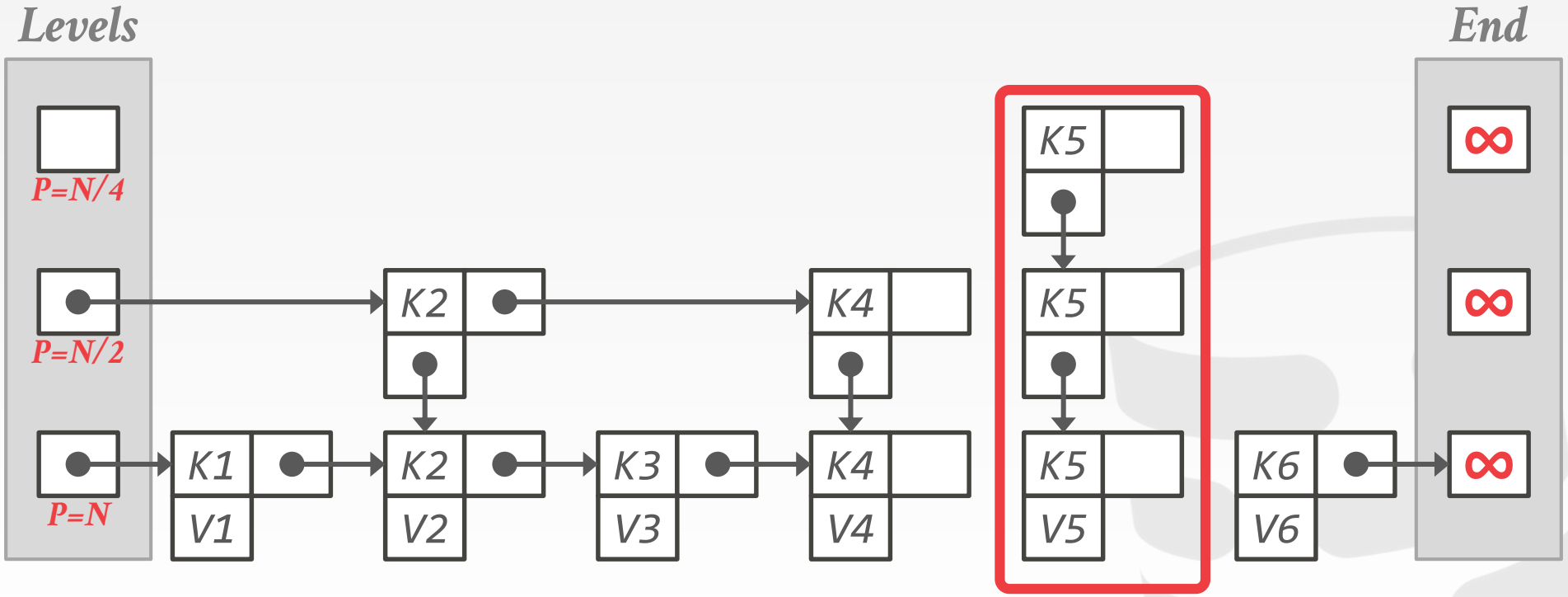
SKIP LISTS: INSERT

Insert K5



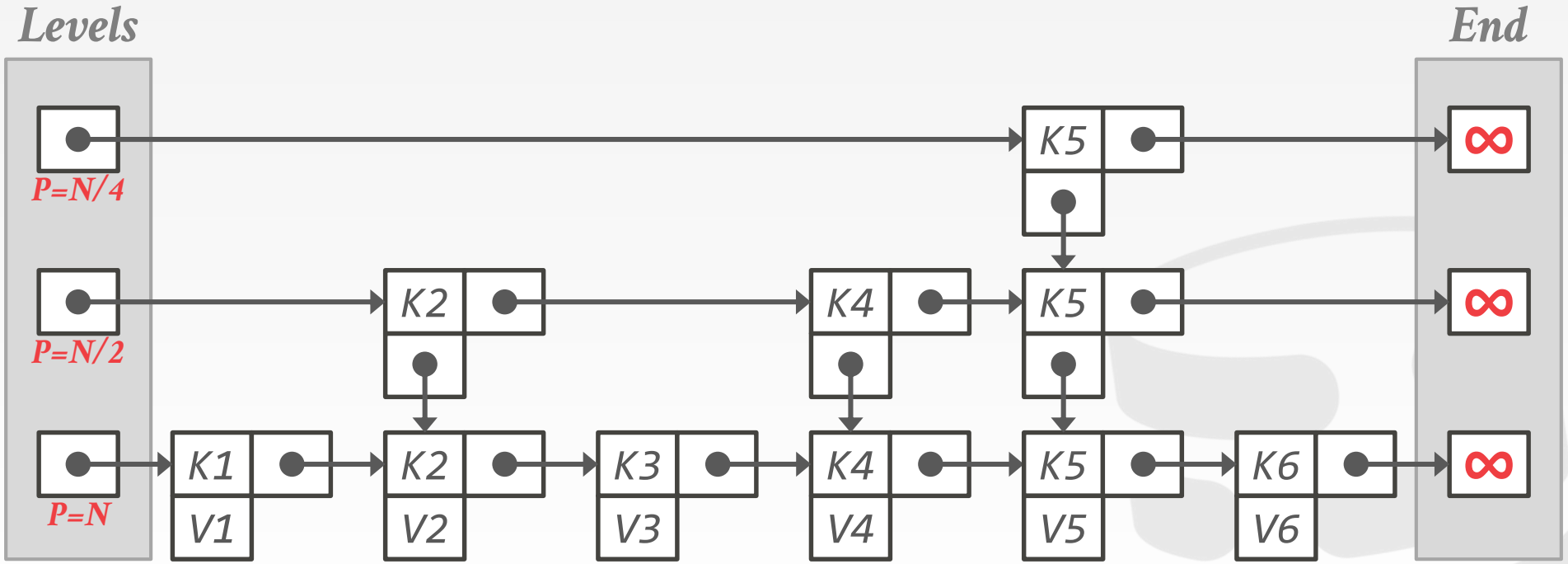
SKIP LISTS: INSERT

Insert K5



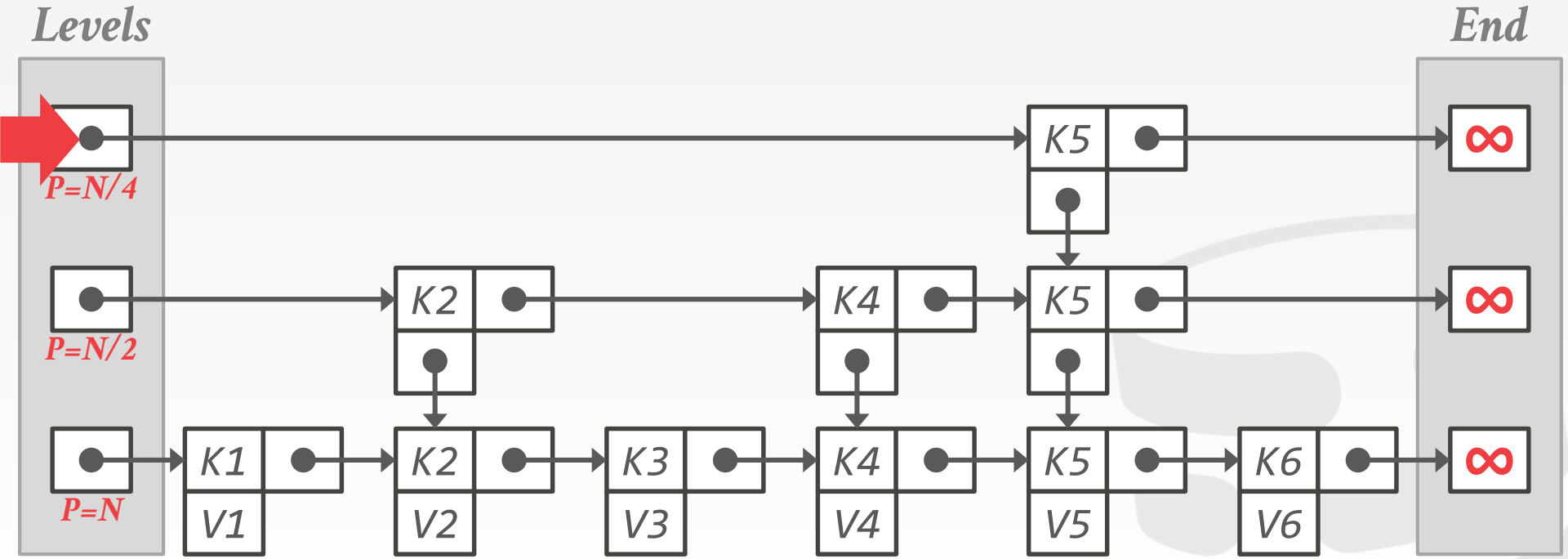
SKIP LISTS: INSERT

Insert K5

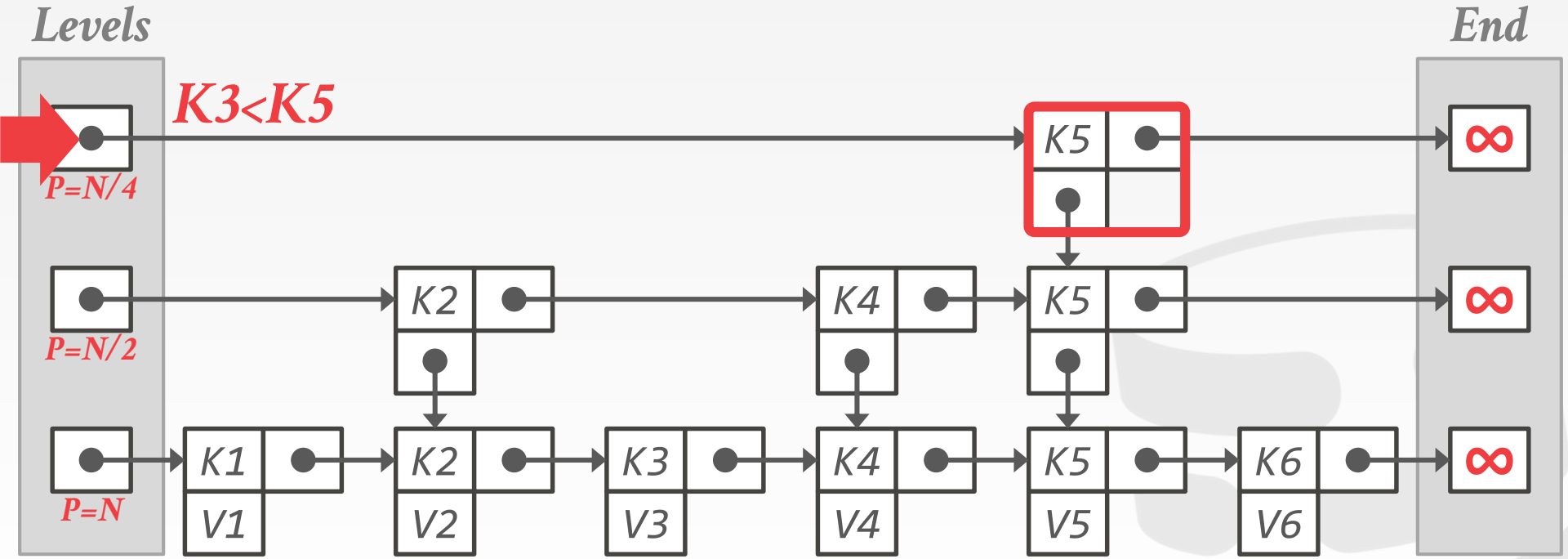


SKIP LISTS: SEARCH

Find K3

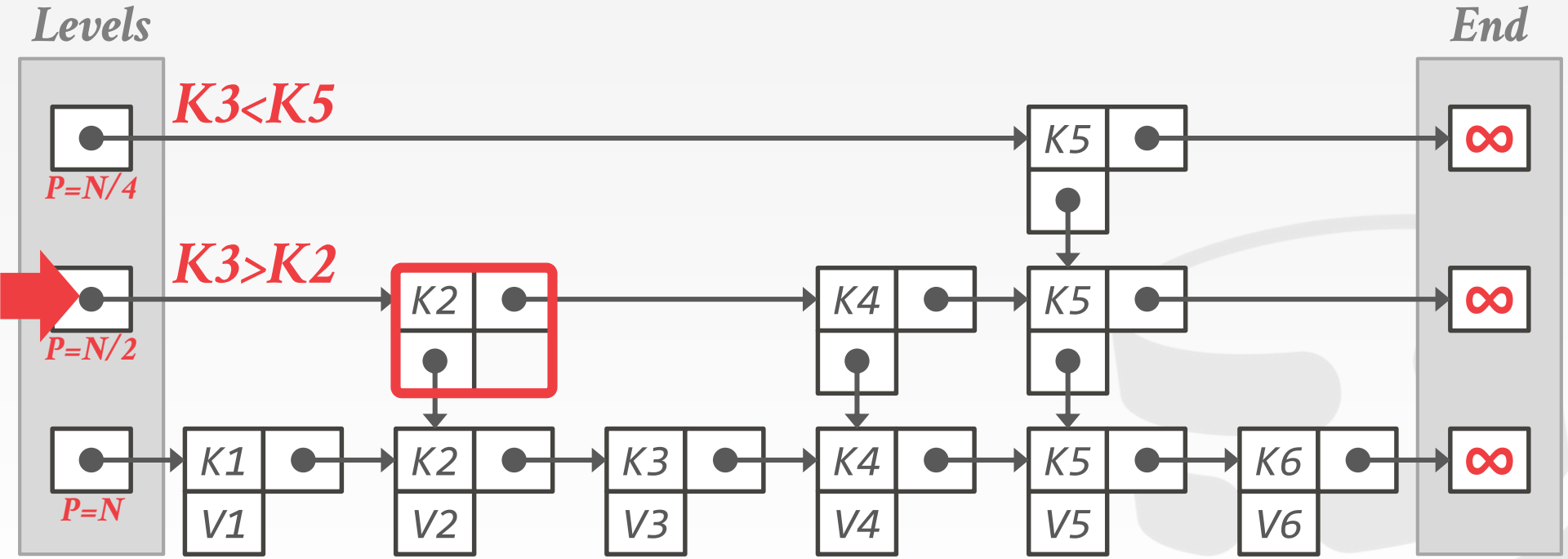


SKIP LISTS: SEARCH



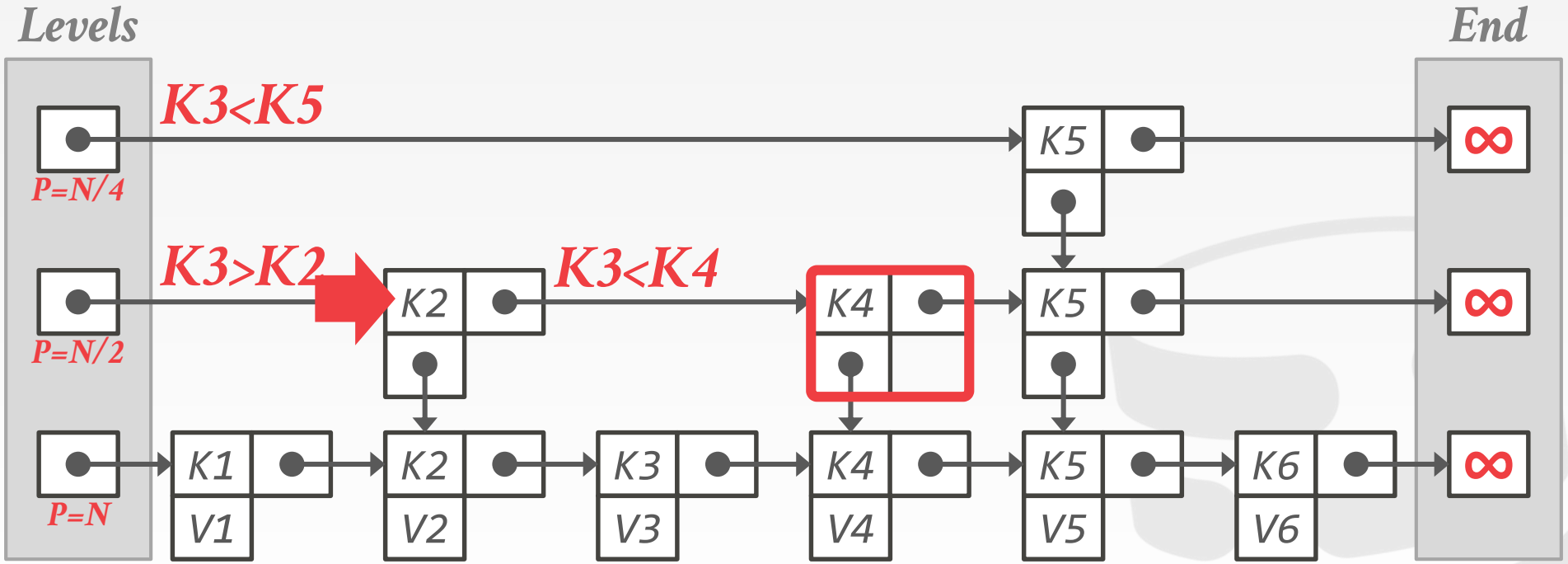
SKIP LISTS: SEARCH

Find K3



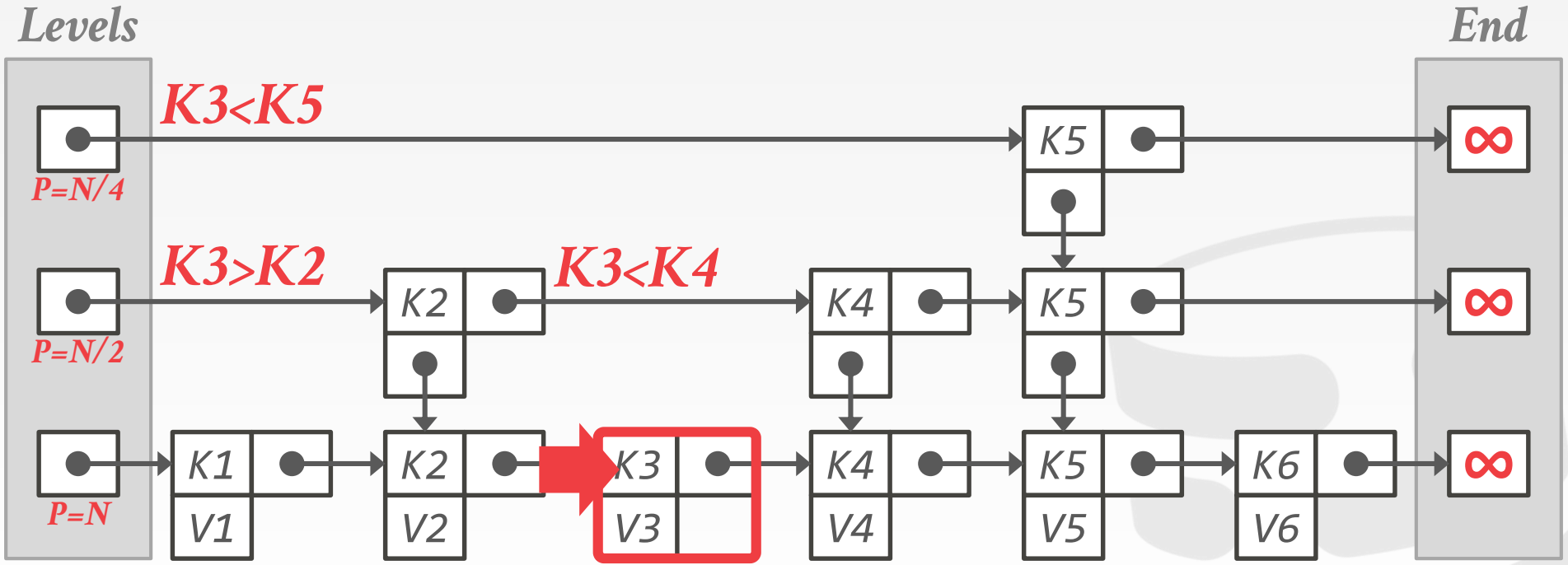
SKIP LISTS: SEARCH

Find K3



SKIP LISTS: SEARCH

Find K3



SKIP LISTS: ADVANTAGES

Uses less memory than a typical B+tree (only if you don't include reverse pointers).

Insertions and deletions do not require rebalancing.

We can implement a concurrent skip list using only CaS instructions.

CONCURRENT SKIP LIST

Can implement insert and delete without locks using CaS operations.

→ Only support links in one direction because CaS can only swap one pointer atomically.

If the DBMS invokes operation on the index, it can never "fail"

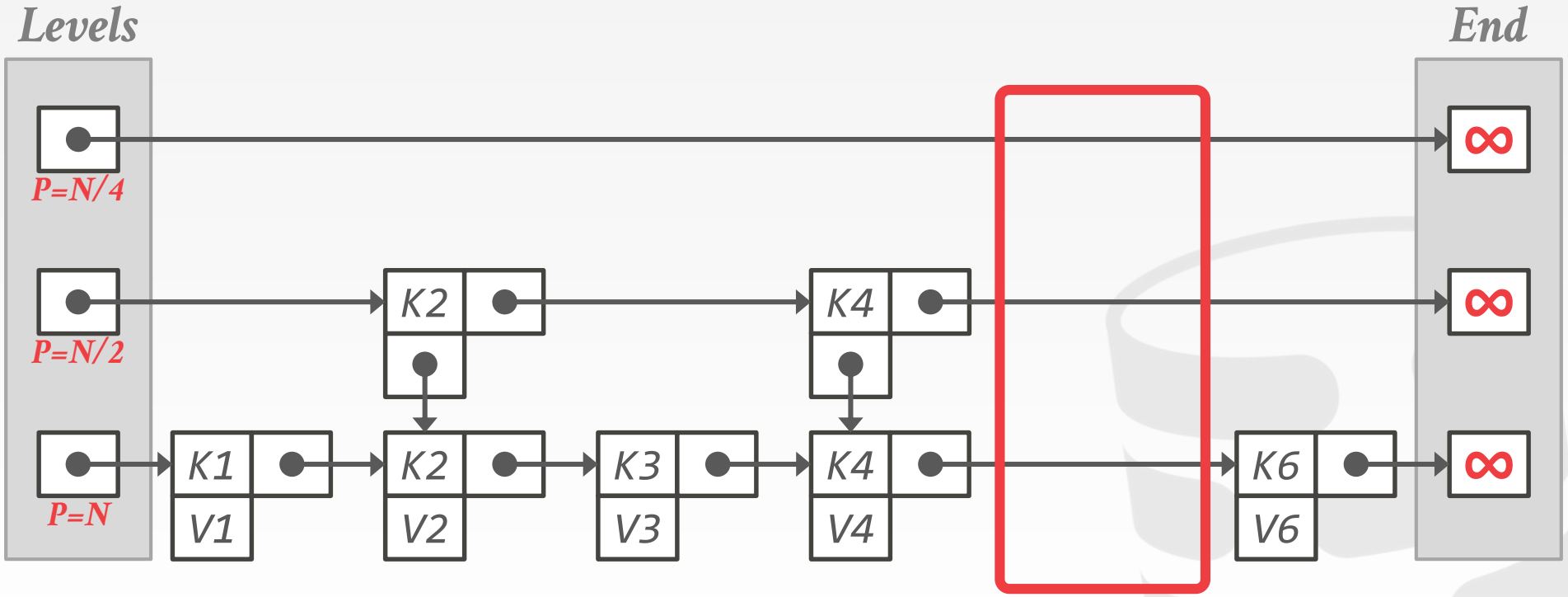
→ A txn can only abort due to higher-level conflicts.

→ If a CaS fails, then the index will retry until it succeeds.

CONCURRENT MAINTENANCE OF SKIP LISTS
UNIV. OF MARYLAND TECH REPORT 1990

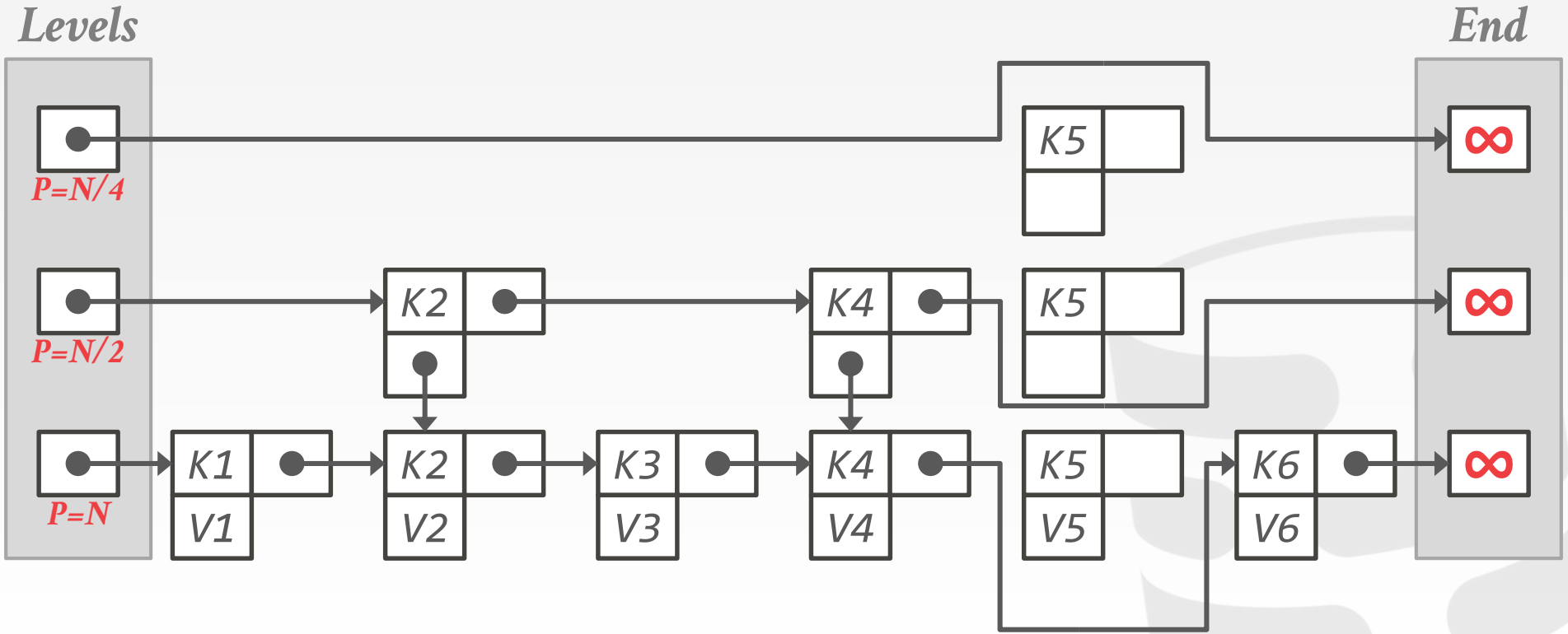
SKIP LISTS: INSERT

Insert K5



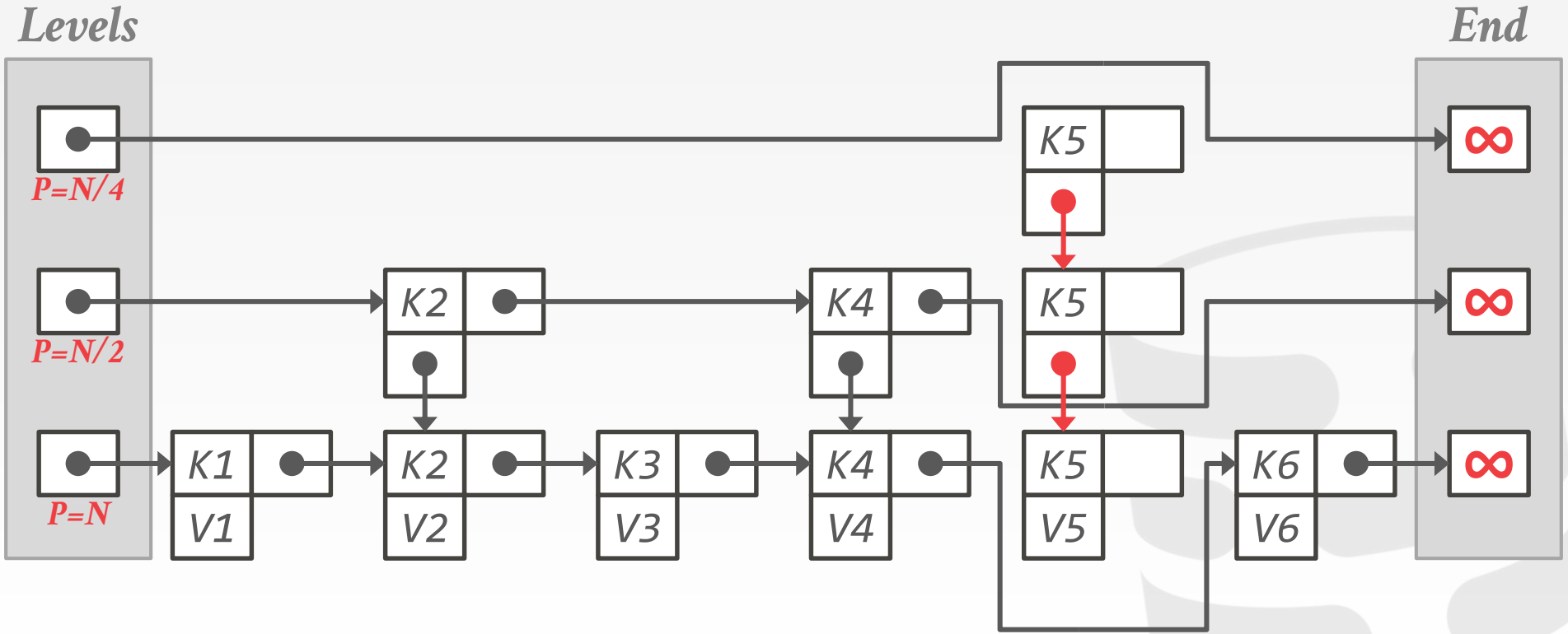
SKIP LISTS: INSERT

Insert K5



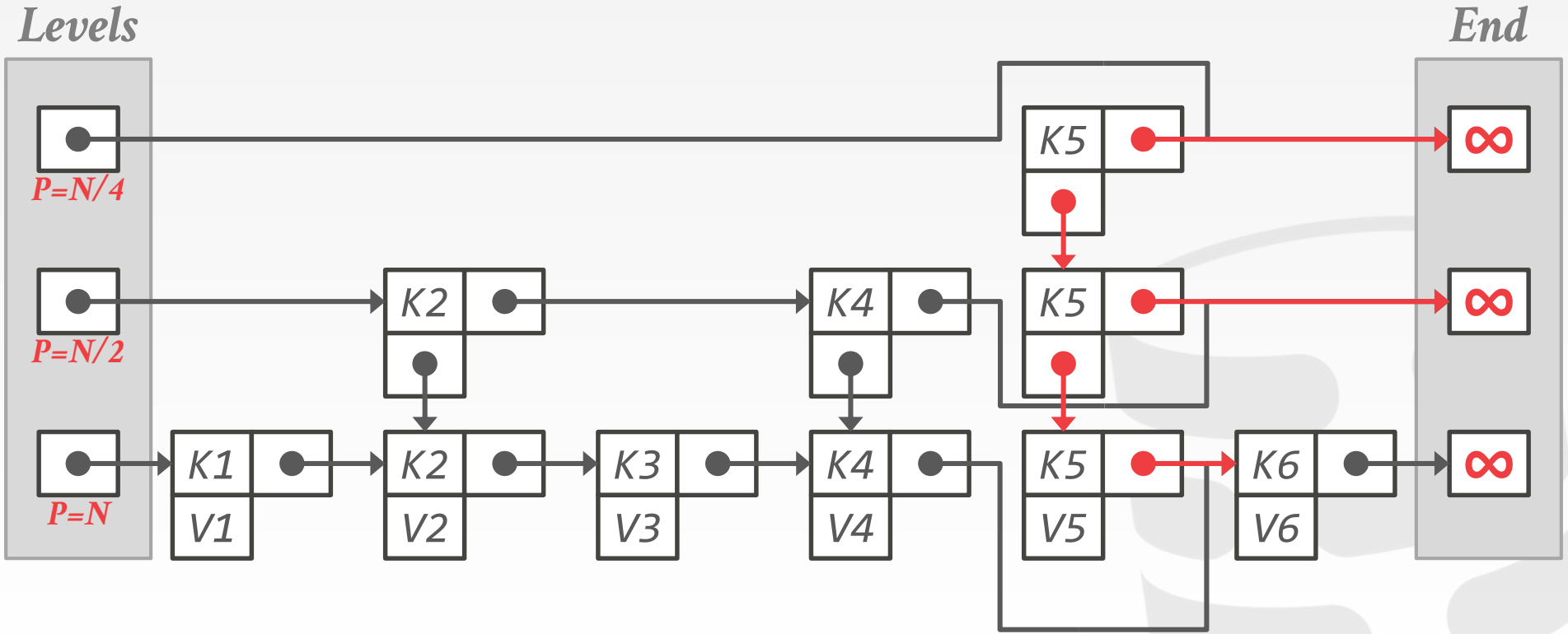
SKIP LISTS: INSERT

Insert K5



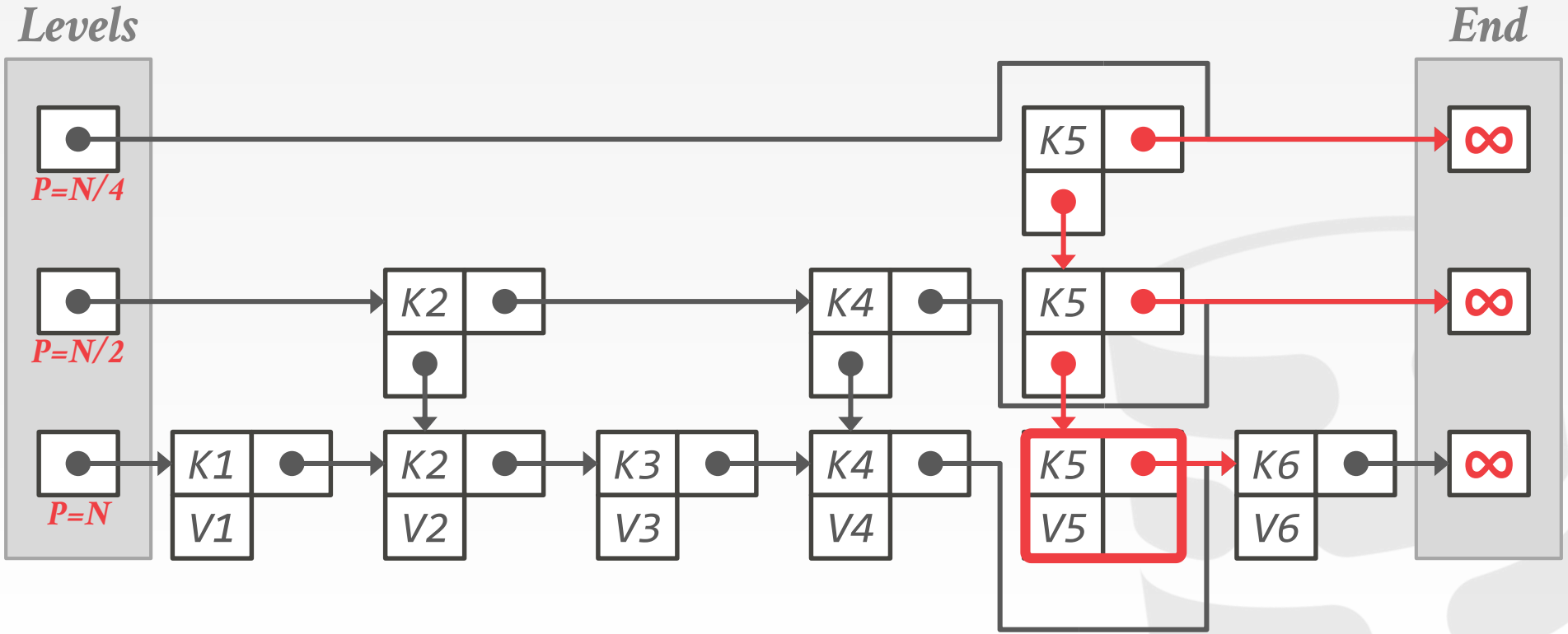
SKIP LISTS: INSERT

Insert K5



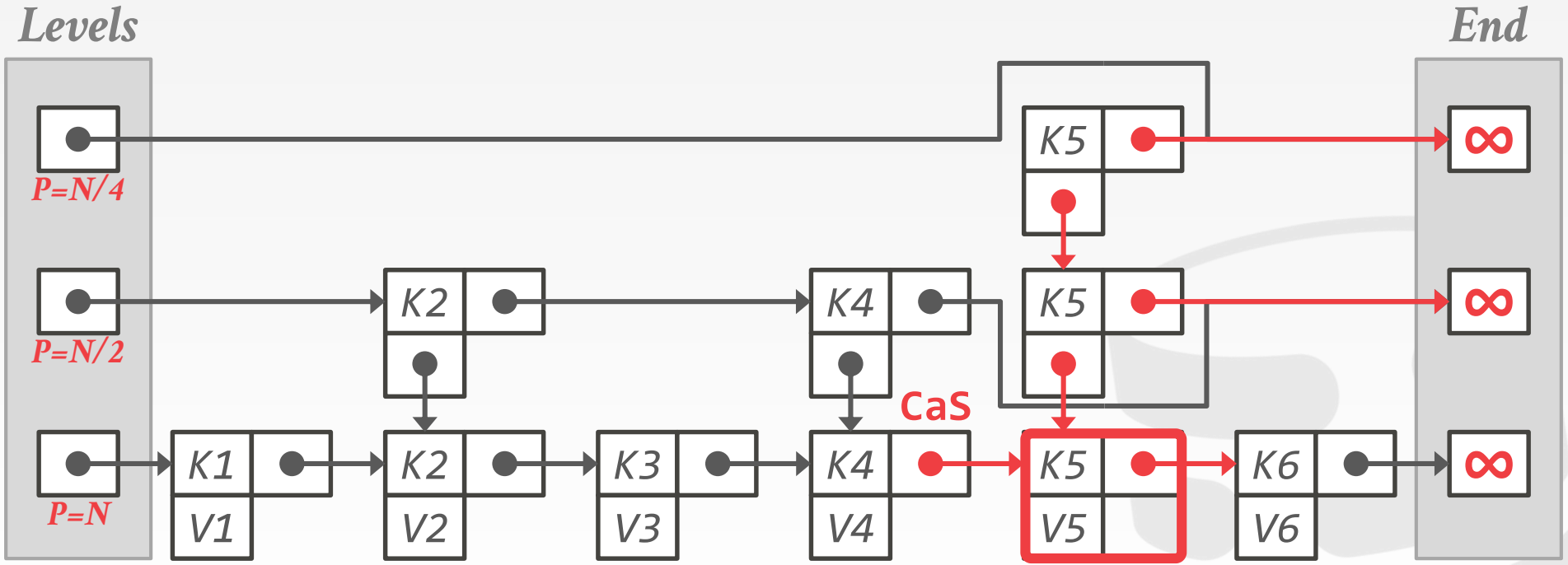
SKIP LISTS: INSERT

Insert K5

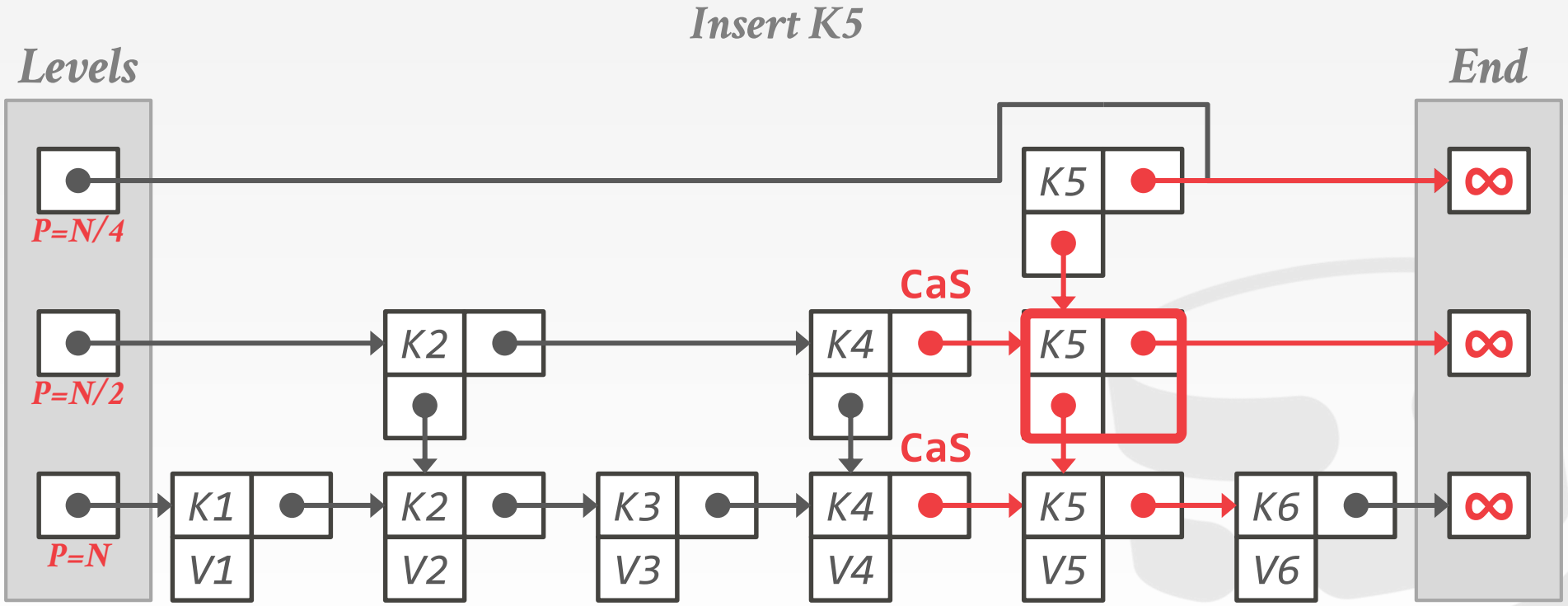


SKIP LISTS: INSERT

Insert K5

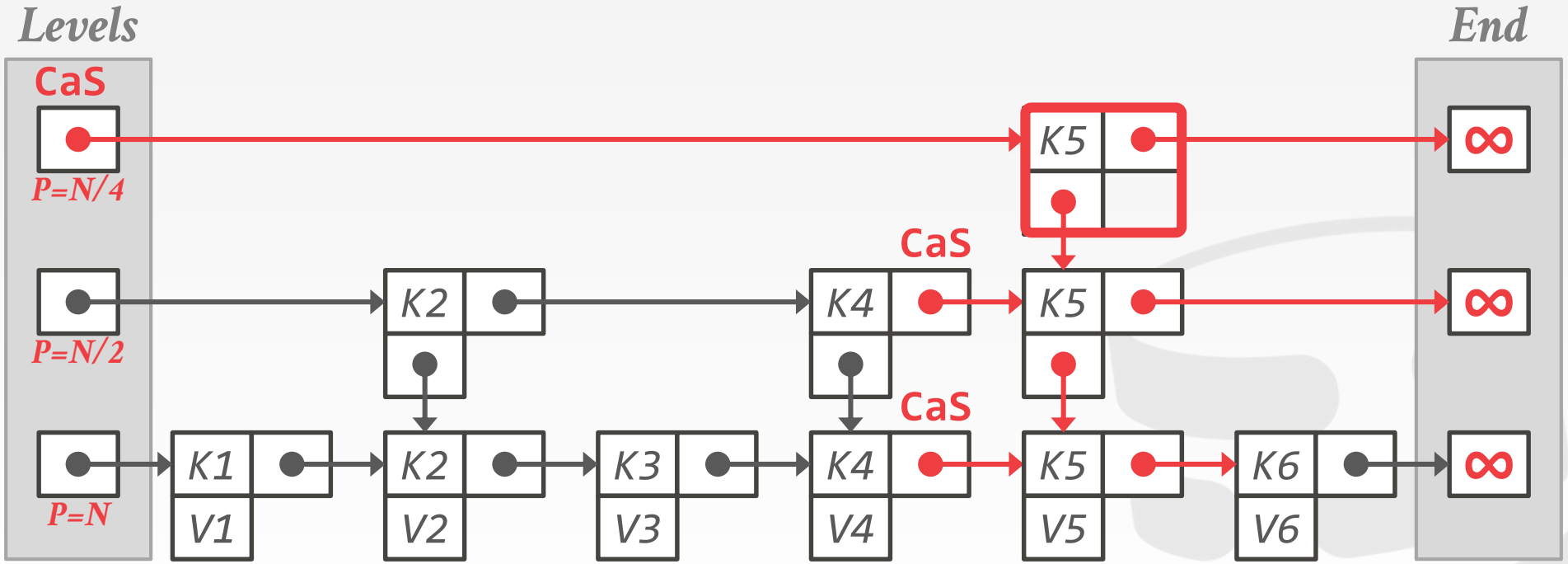


SKIP LISTS: INSERT



SKIP LISTS: INSERT

Insert K5



SKIP LISTS: DELETE

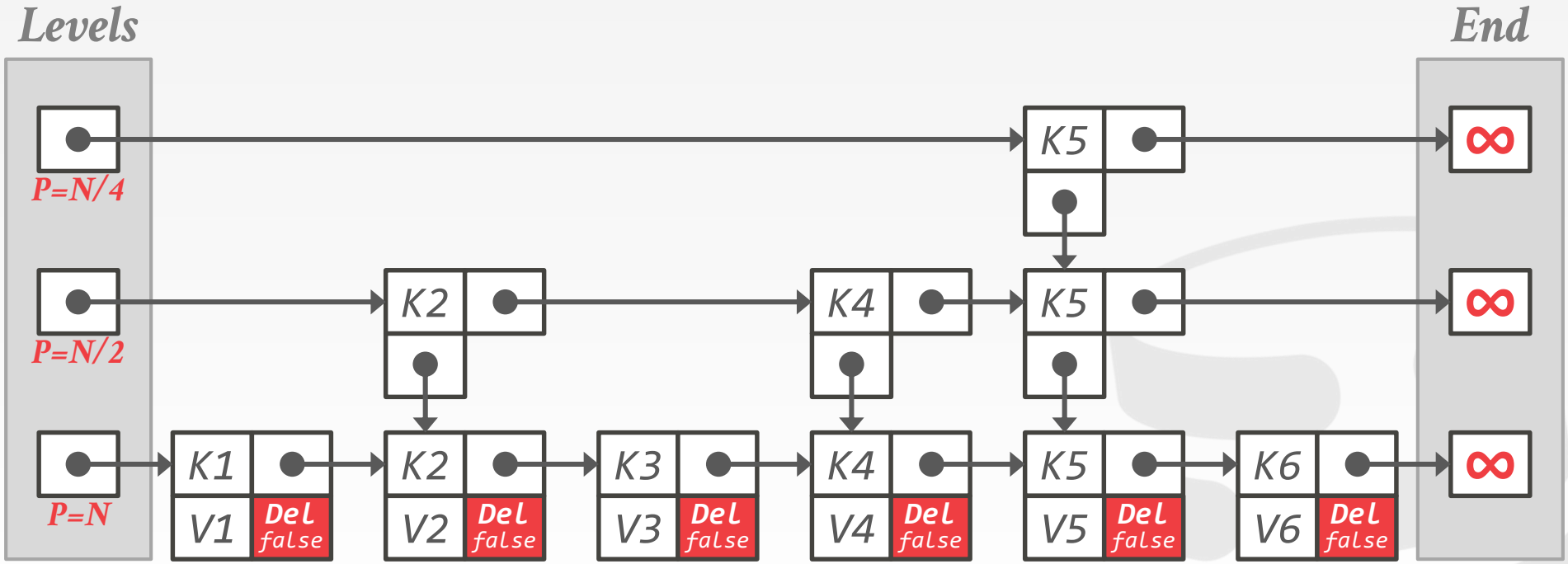
First **logically** remove a key from the index by setting a flag to tell threads to ignore.

Then **physically** remove the key once we know that no other thread is holding the reference.
→ Perform CaS to update the predecessor's pointer.

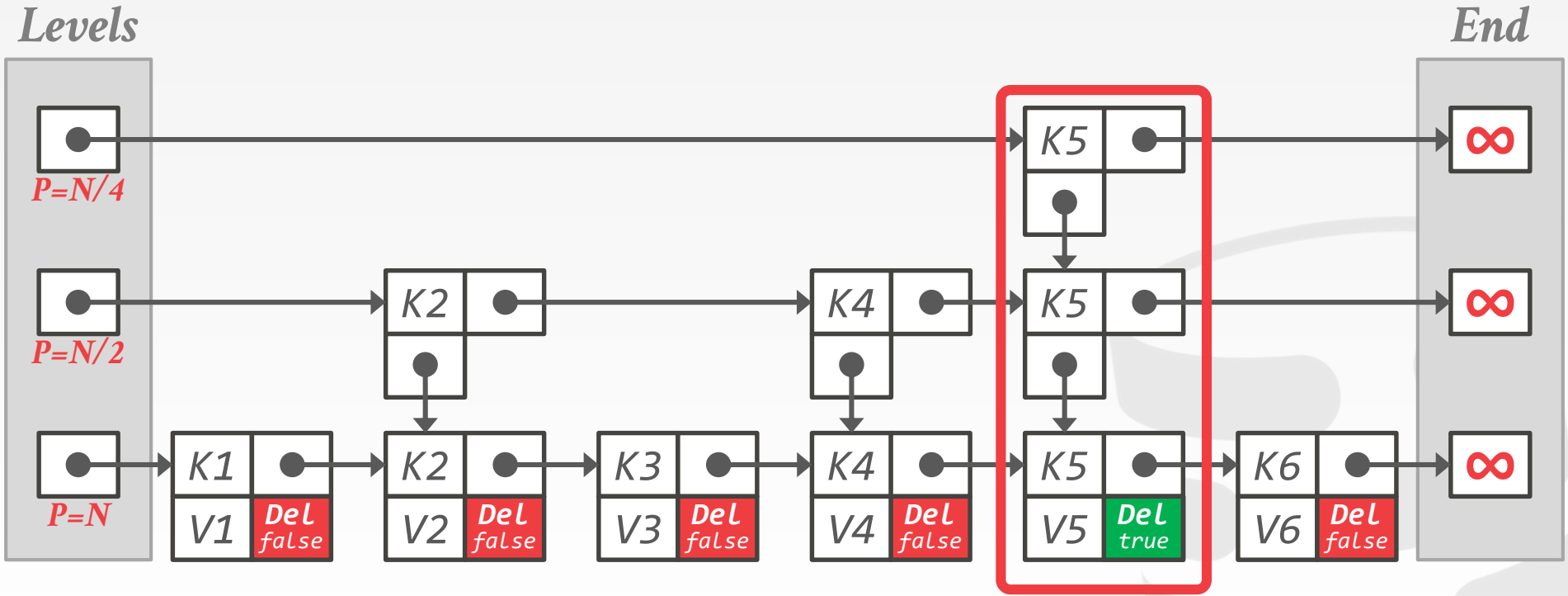
Source: Stephen Tu

SKIP LISTS: DELETE

Delete K5

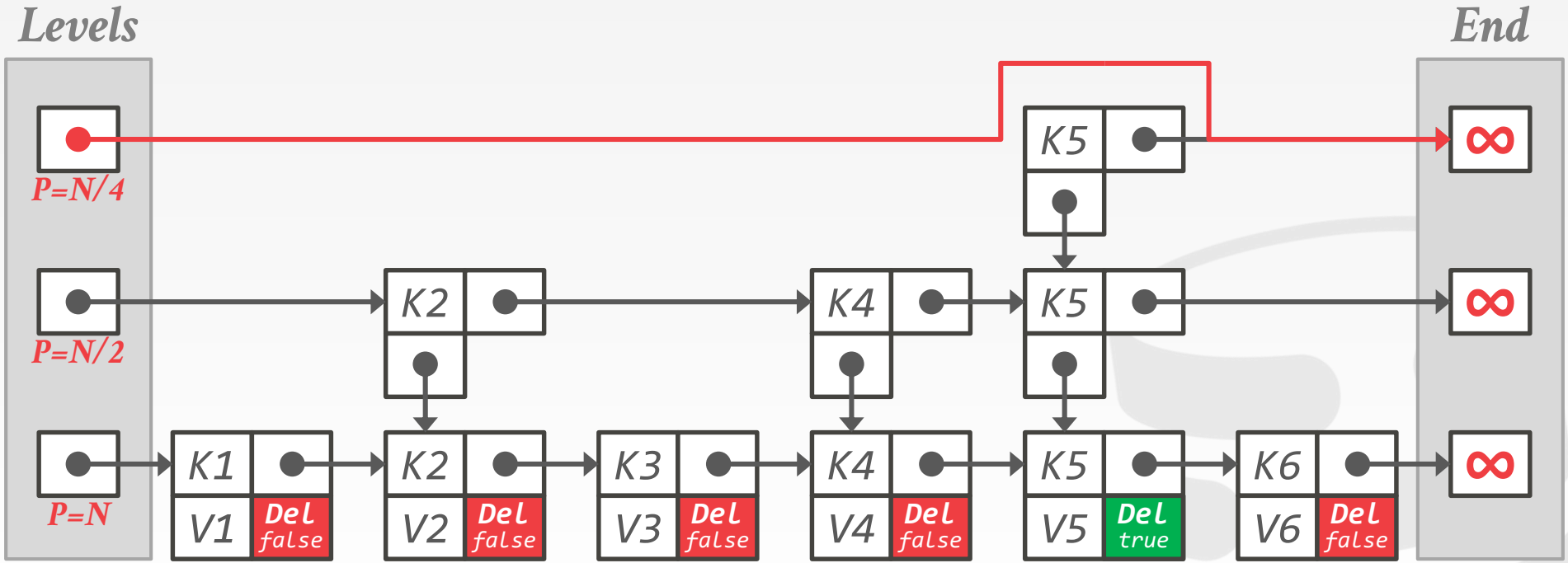


SKIP LISTS: DELETE



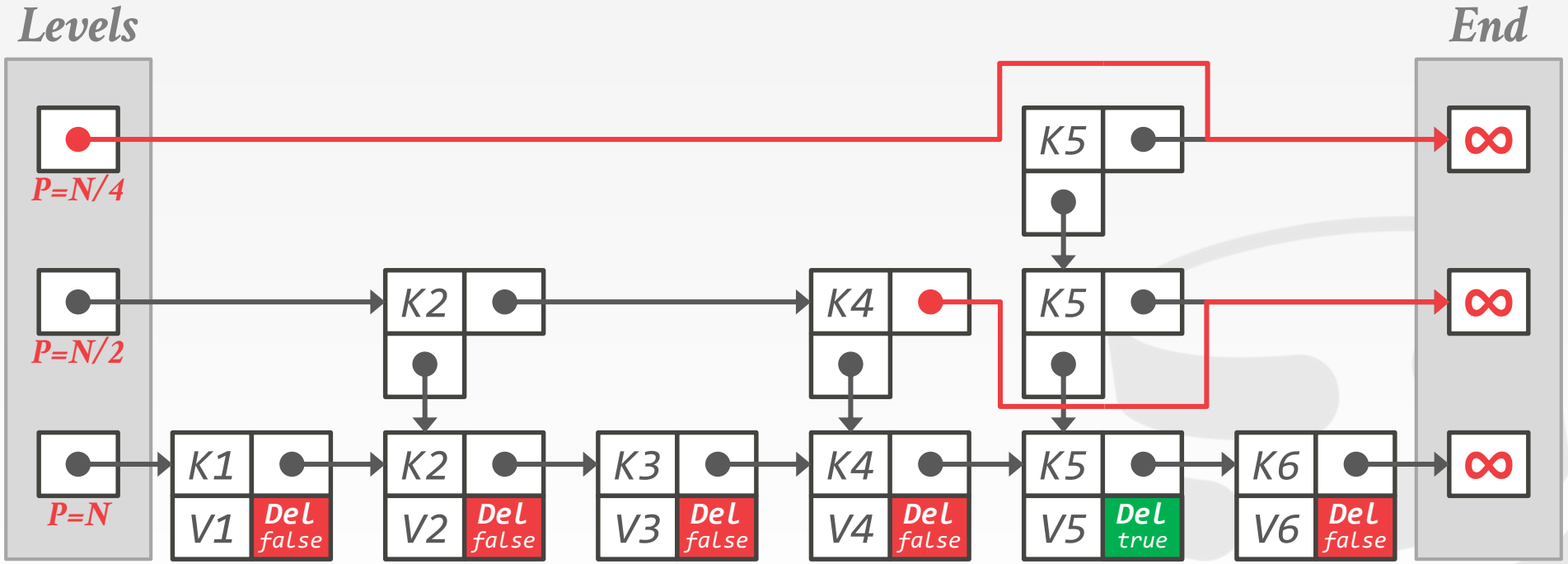
SKIP LISTS: DELETE

Delete K5



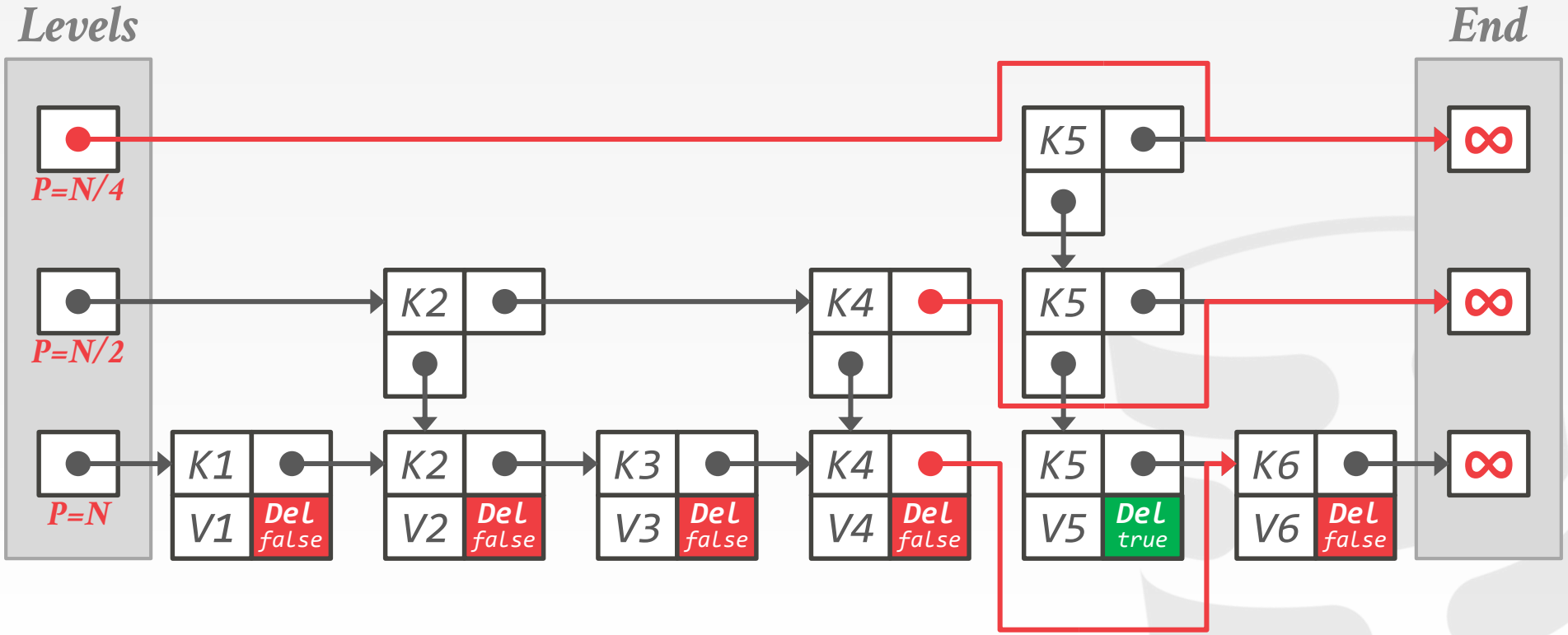
SKIP LISTS: DELETE

Delete K5



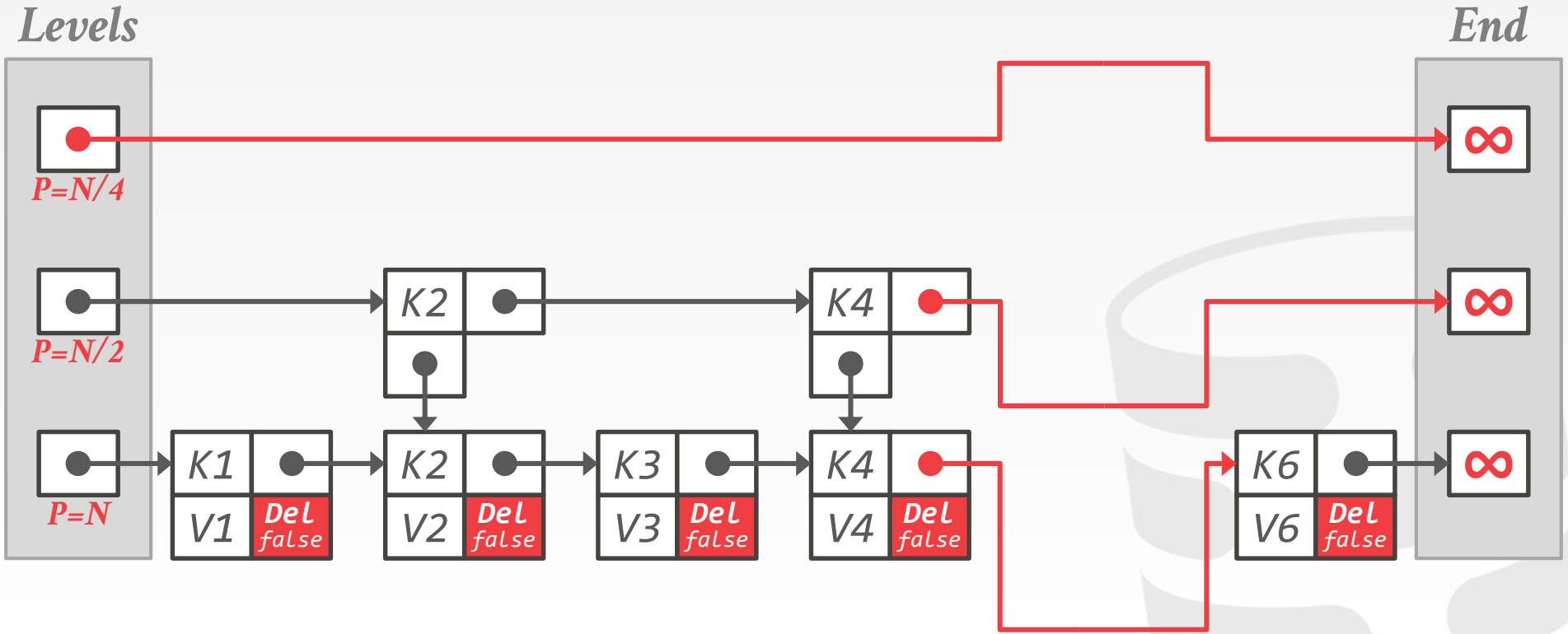
SKIP LISTS: DELETE

Delete K5



SKIP LISTS: DELETE

Delete K5



OBSERVATION

Because CaS only updates a single address at a time, this limits the design of our data structures

→ We cannot have reverse pointers in a latch-free concurrent Skip List.

→ We cannot build a latch-free B+Tree.

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

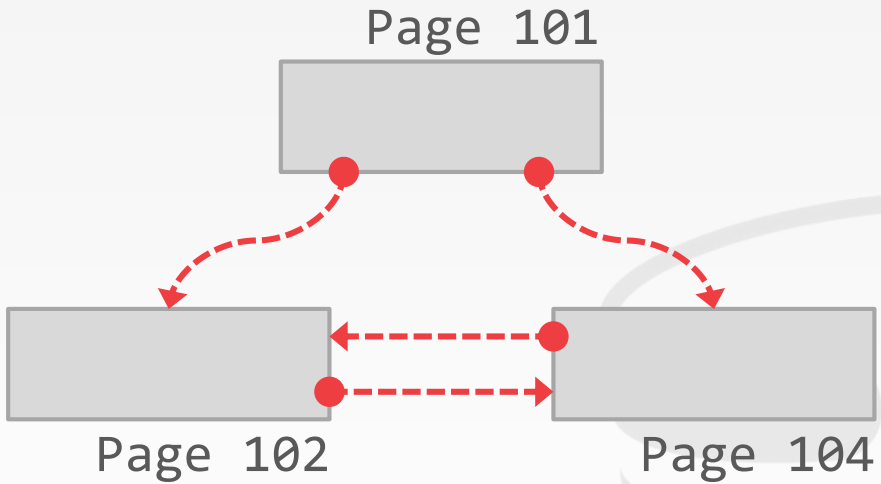
→ Allows for CaS of physical locations of pages.

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

BW-TREE: MAPPING TABLE

Mapping Table

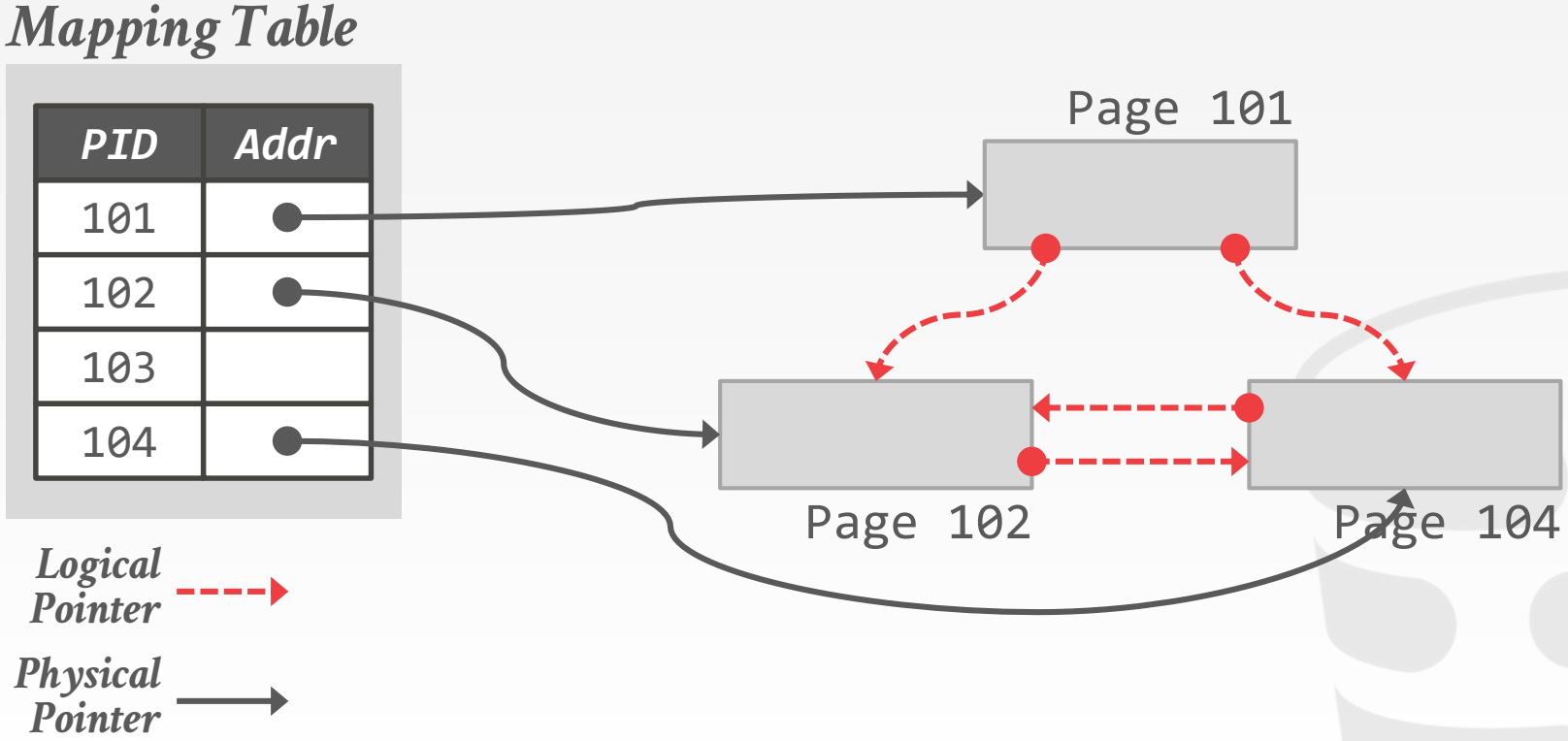
PID	Addr
101	
102	
103	
104	



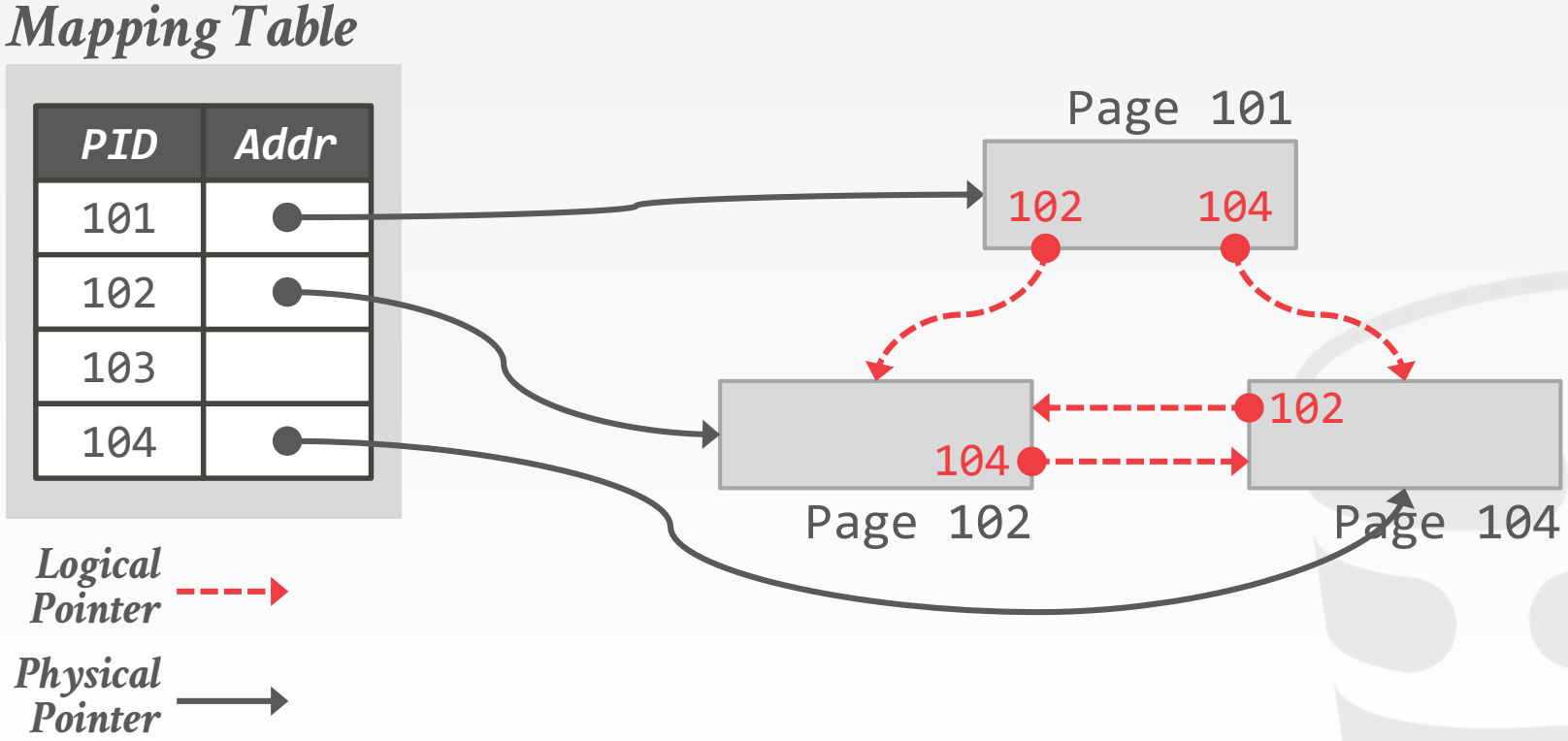
Logical Pointer ----->

Physical Pointer ——>

BW-TREE: MAPPING TABLE



BW-TREE: MAPPING TABLE



BW-TREE: DELTA UPDATES

Mapping Table

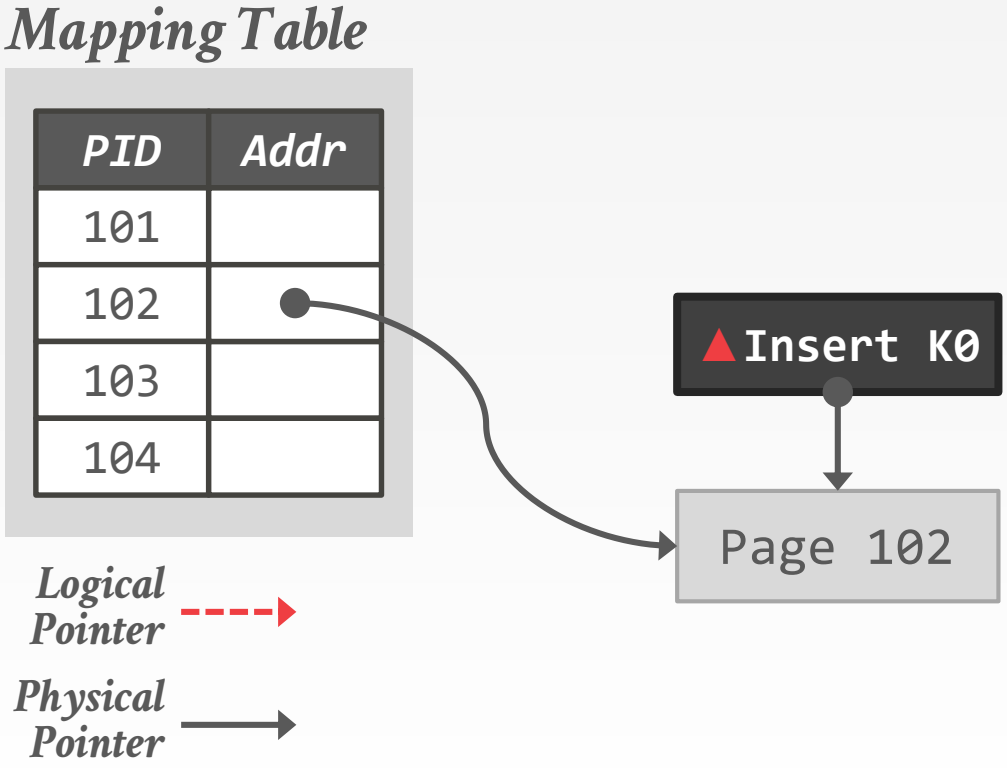
PID	Addr
101	
102	
103	
104	

Logical Pointer ----→

Physical Pointer ——→

Each update to a page produces a new delta.

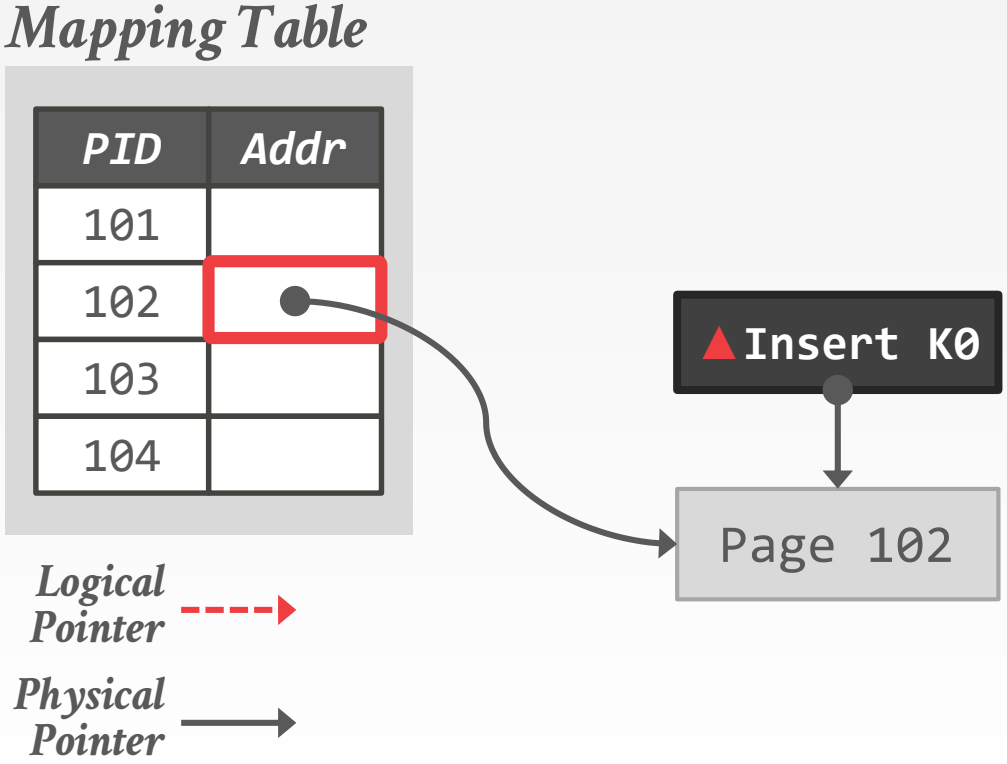
BW-TREE: DELTA UPDATES



Each update to a page produces a new delta.

Delta physically points to base page.

BW-TREE: DELTA UPDATES

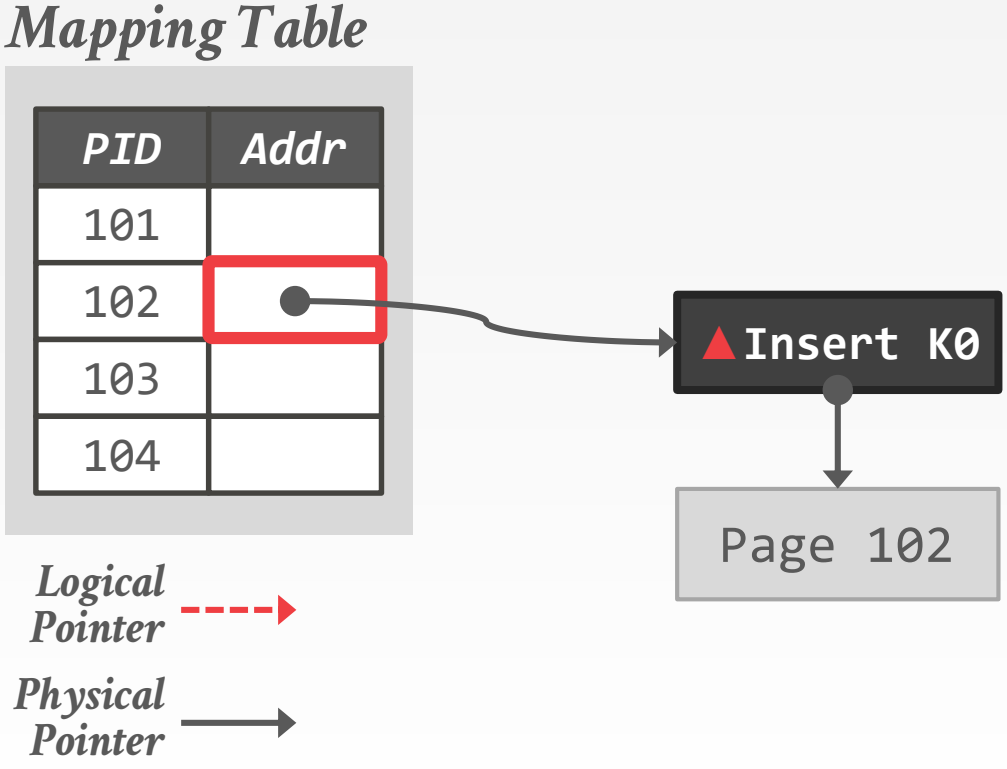


Each update to a page produces a new delta.

Delta physically points to base page.

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

BW-TREE: DELTA UPDATES

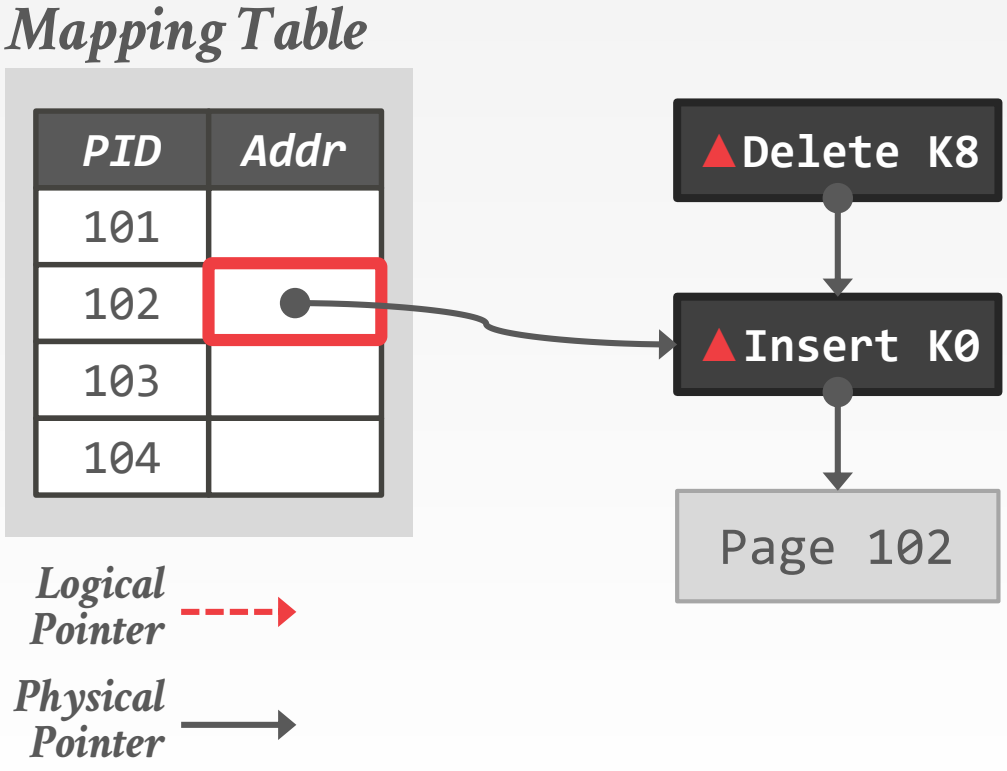


Each update to a page produces a new delta.

Delta physically points to base page.

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

BW-TREE: DELTA UPDATES

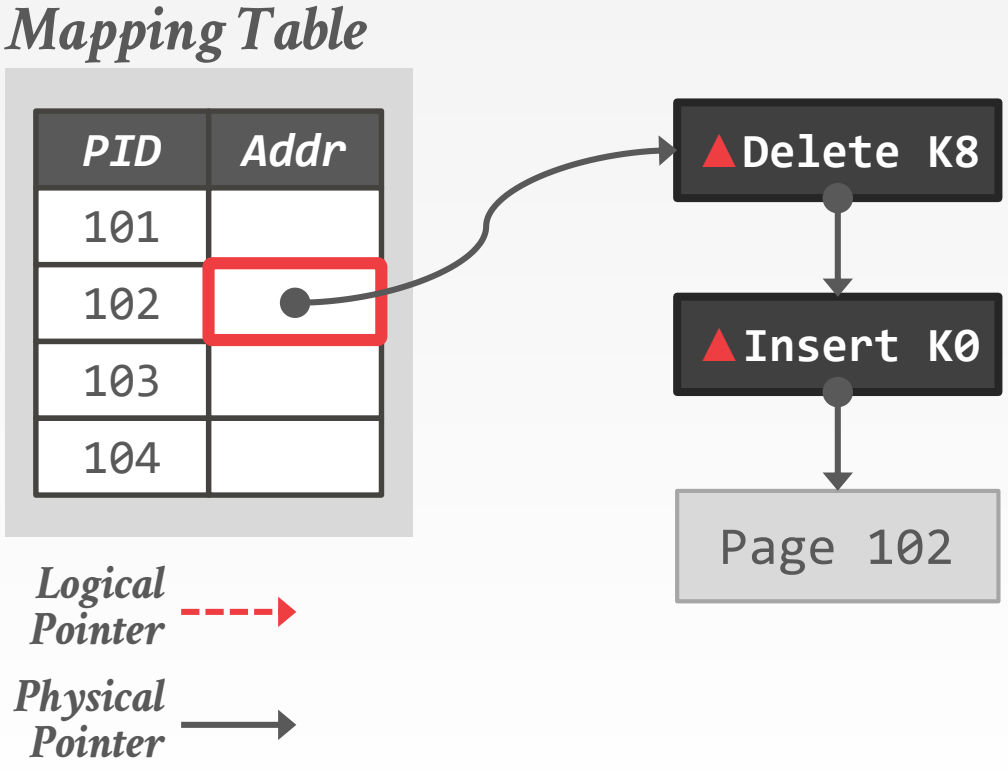


Each update to a page produces a new delta.

Delta physically points to base page.

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

BW-TREE: DELTA UPDATES

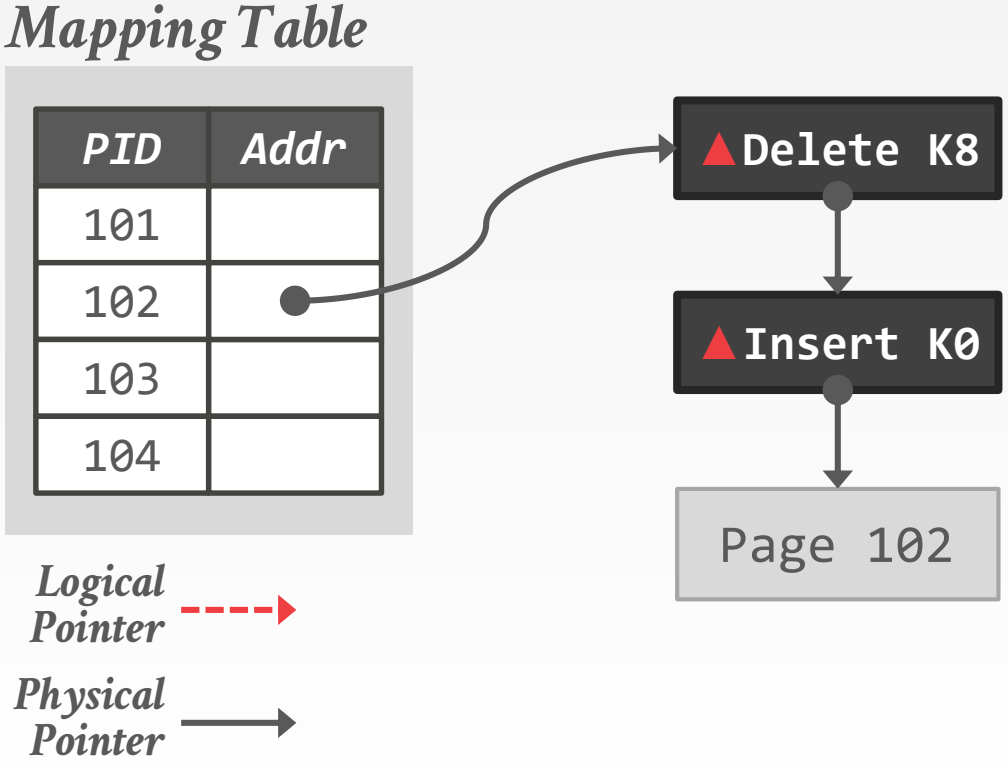


Each update to a page produces a new delta.

Delta physically points to base page.

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

BW-TREE: SEARCH

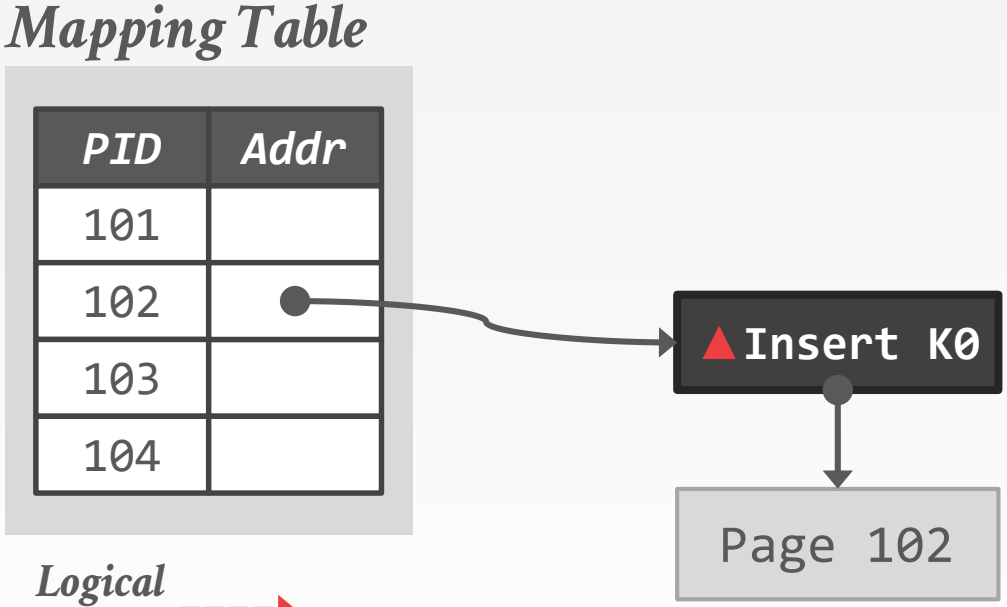


Traverse tree like a regular B+tree.

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

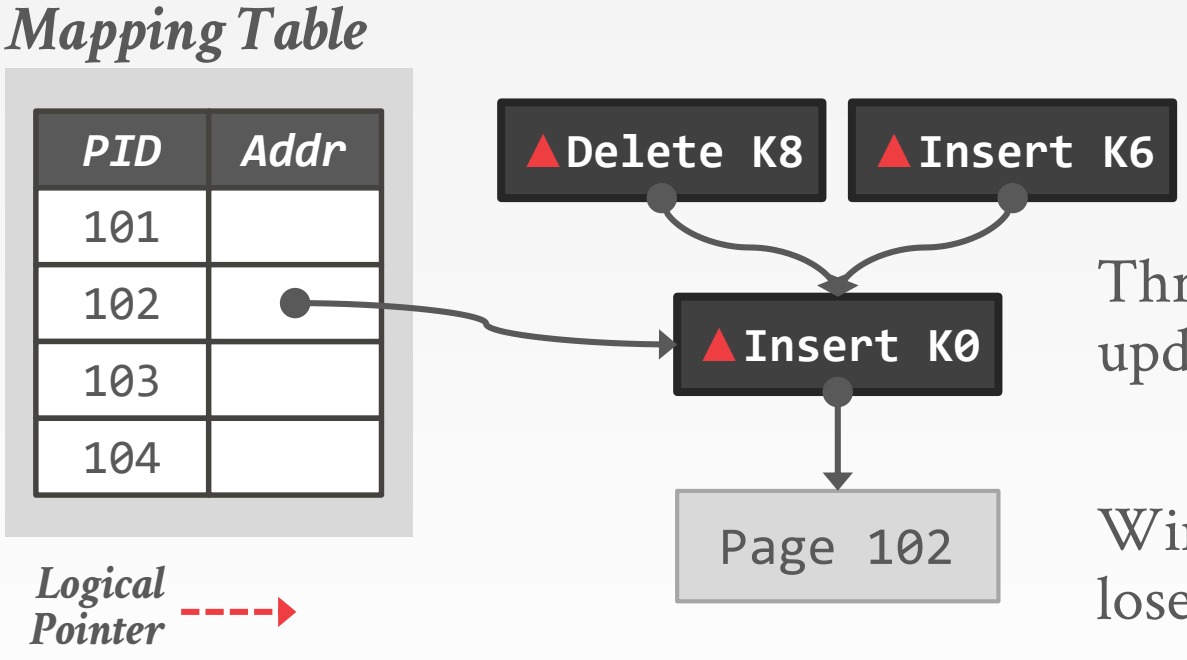
Otherwise, perform binary search on base page.

BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

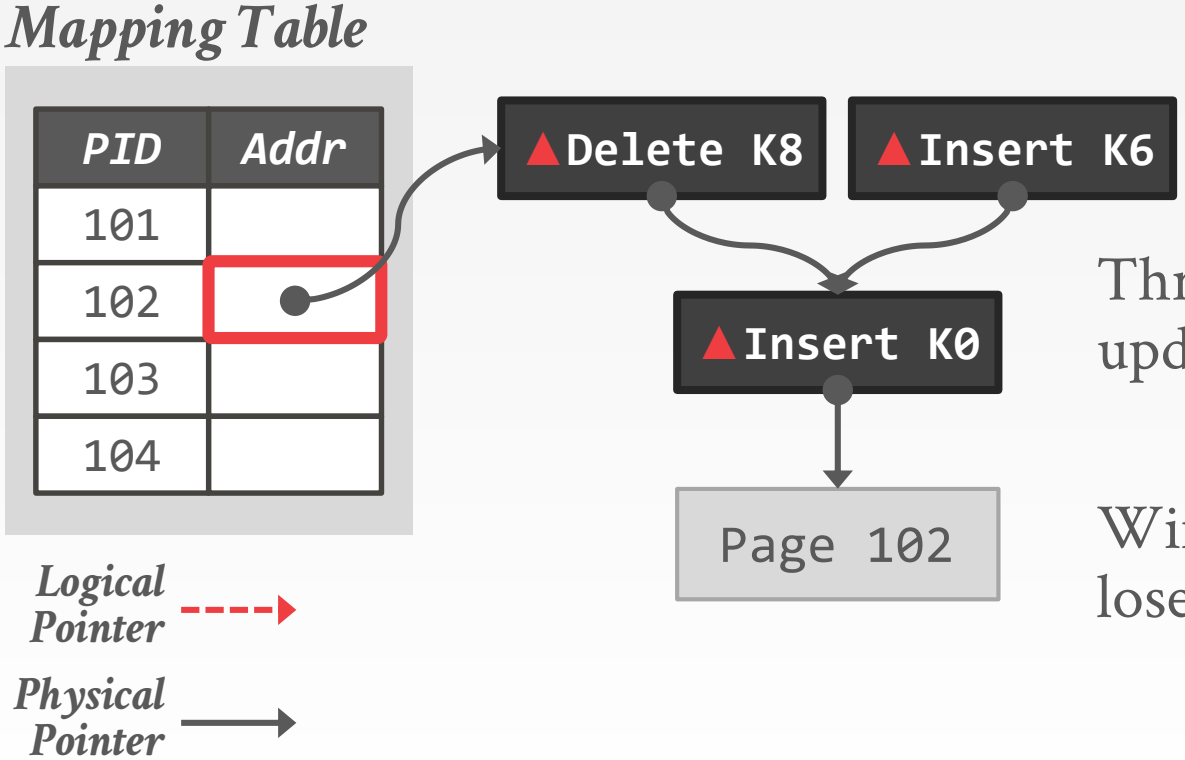
BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

Winner succeeds, any losers must retry or abort

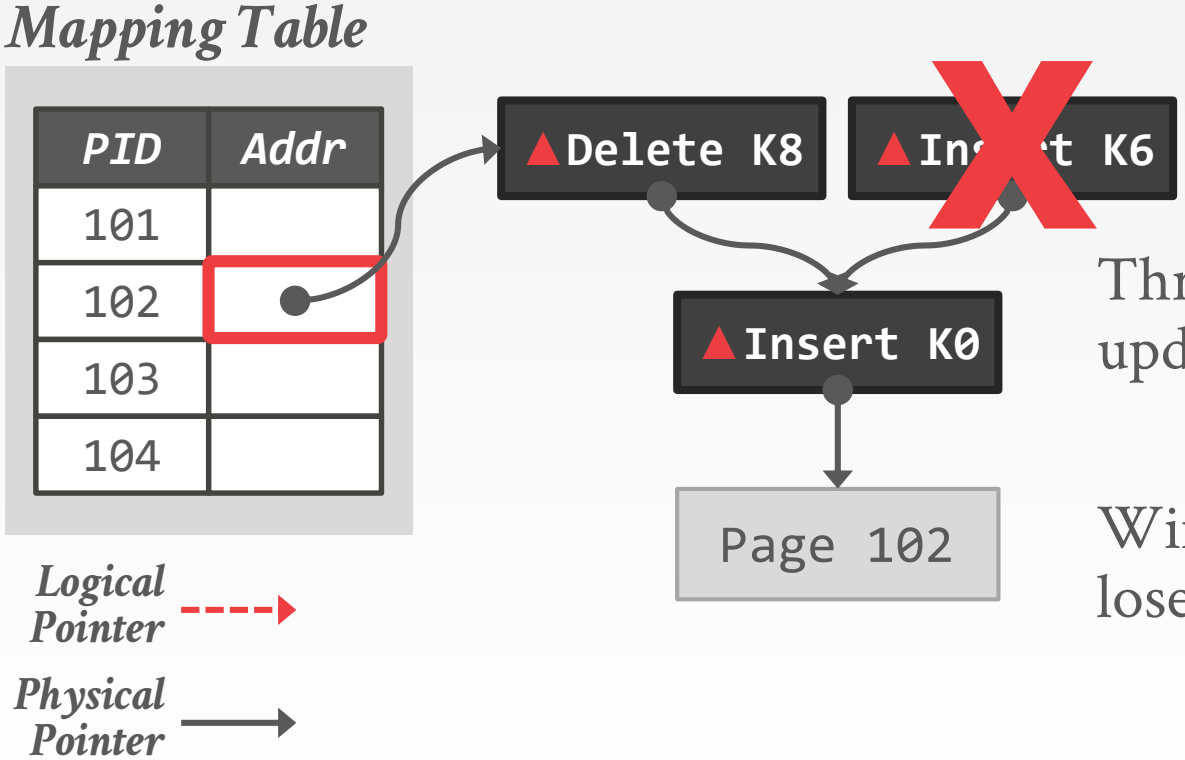
BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

Winner succeeds, any losers must retry or abort

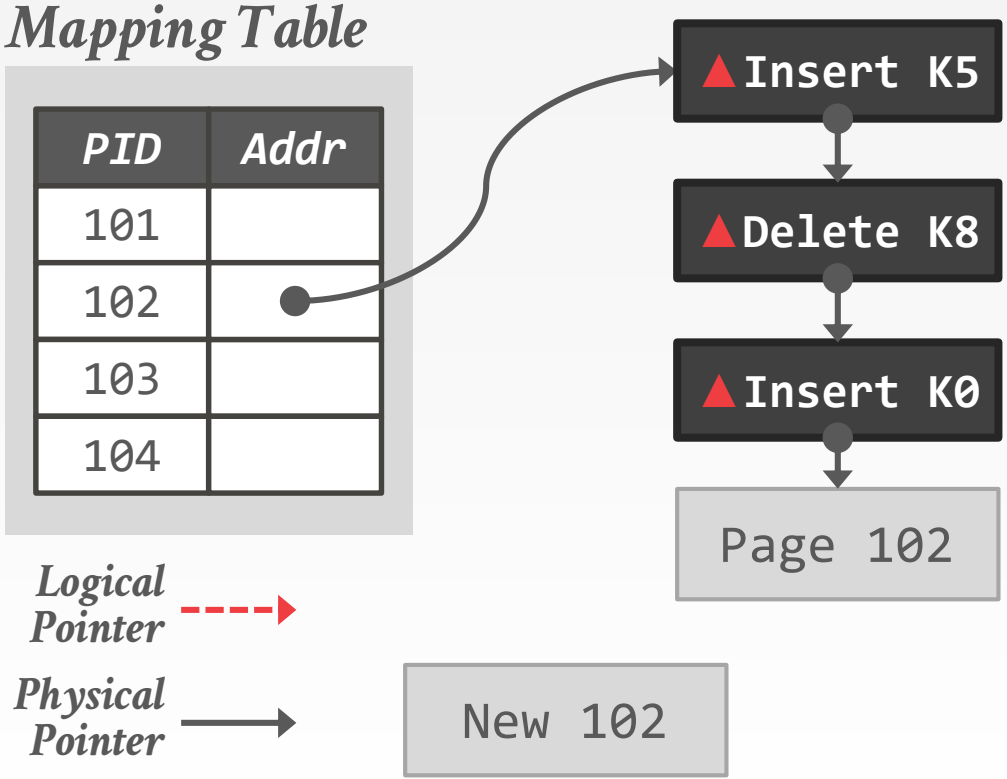
BW-TREE: CONTENTION UPDATES



Threads may try to install updates to same page.

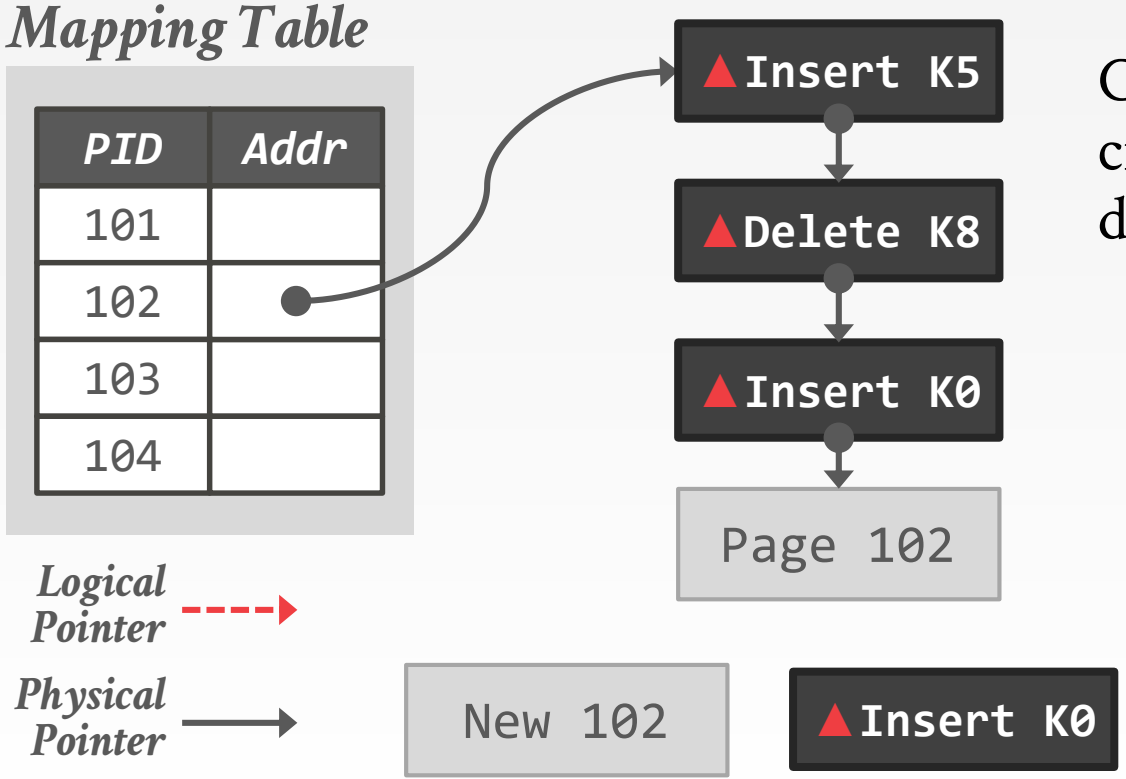
Winner succeeds, any losers must retry or abort

BW-TREE: CONSOLIDATION



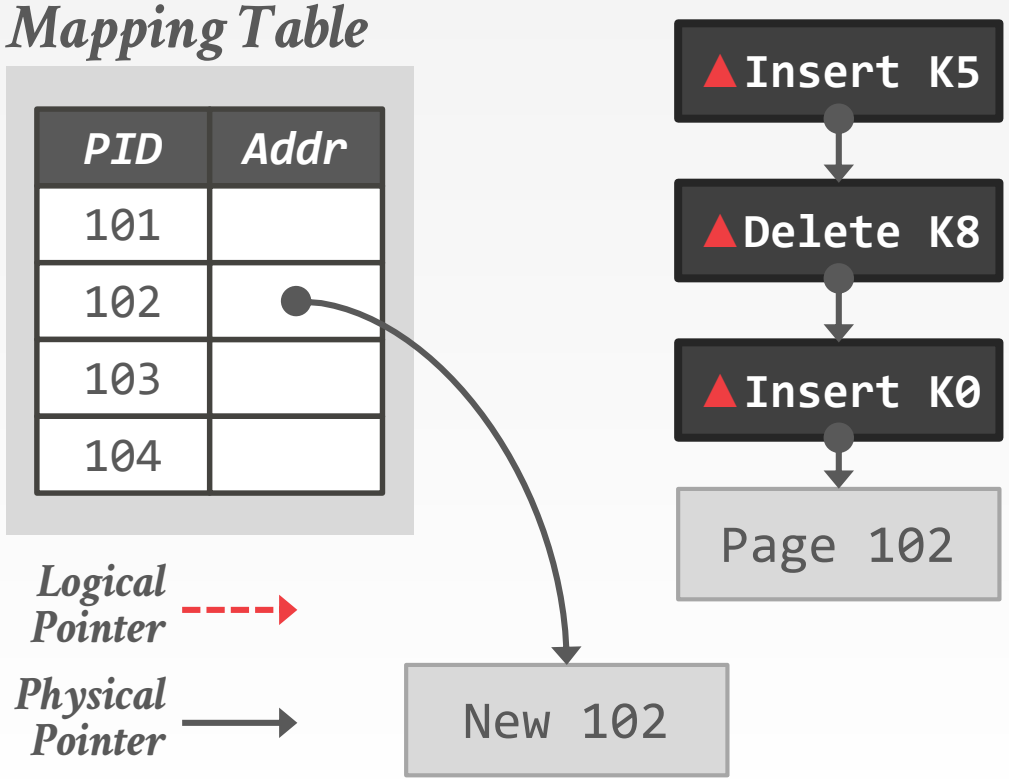
Consolidate updates by creating new page with deltas applied.

BW-TREE: CONSOLIDATION



Consolidate updates by creating new page with deltas applied.

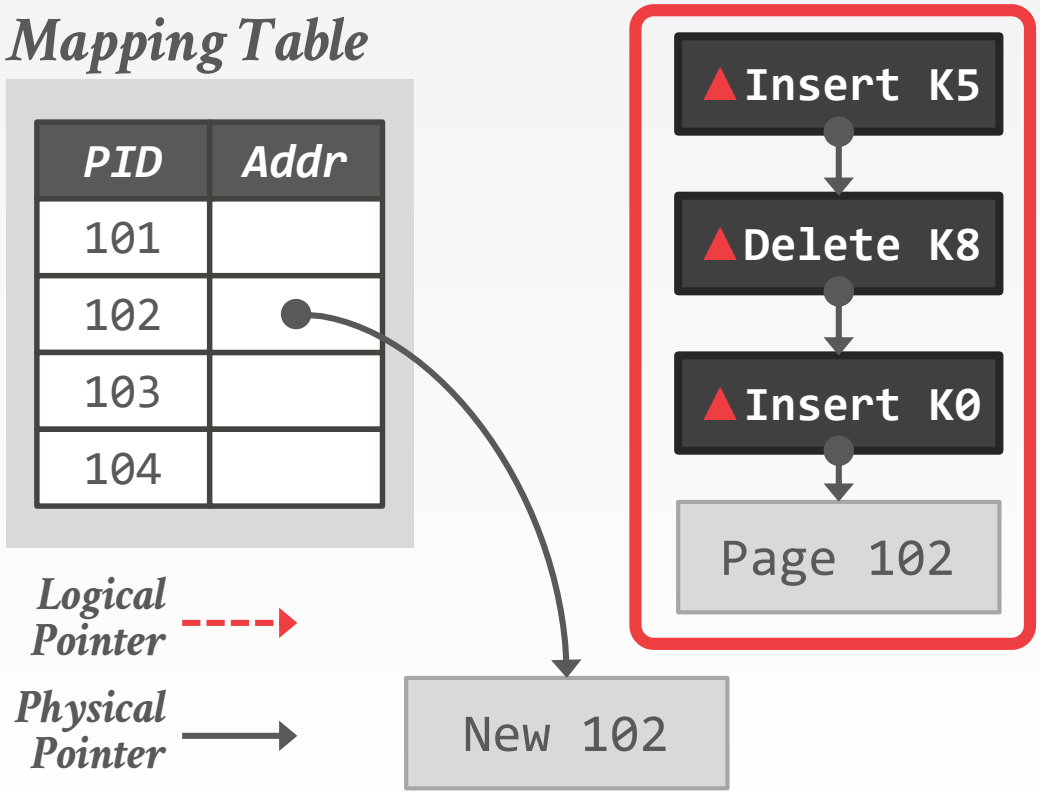
BW-TREE: CONSOLIDATION



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.

BW-TREE: GARBAGE COLLECTION

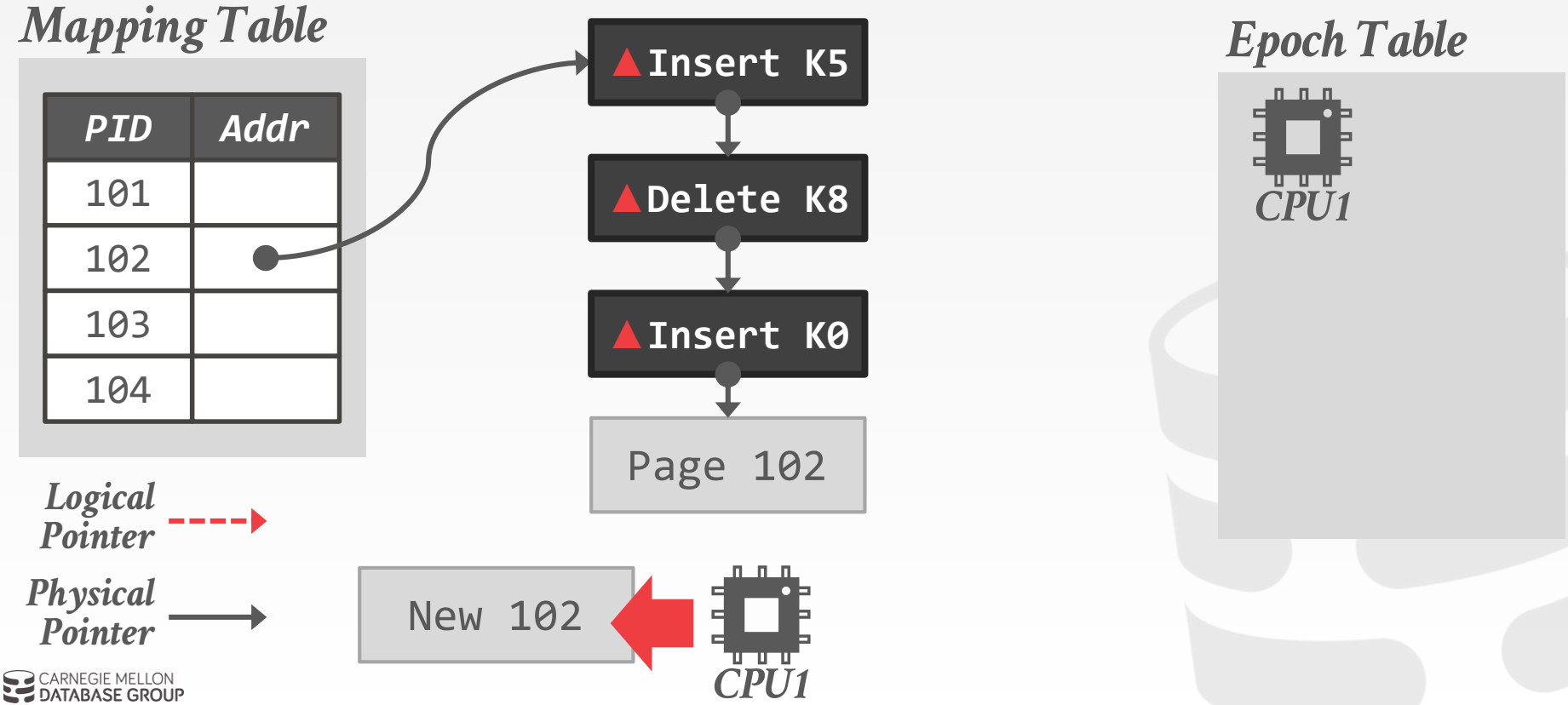
Operations are tagged with an **epoch**

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

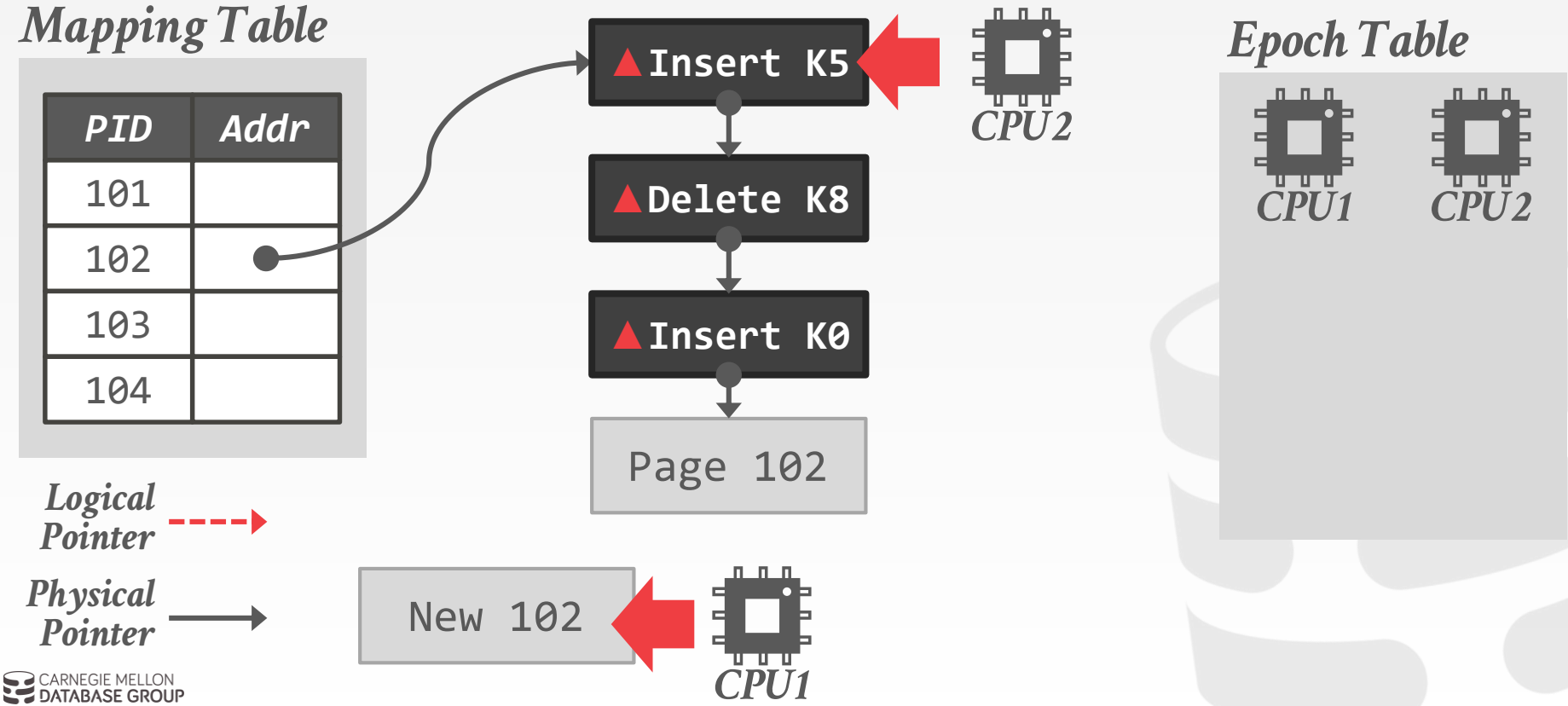
→ Thread joins an epoch prior to each operation and post objects that can be reclaimed for the current epoch (not necessarily the one it joined)

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

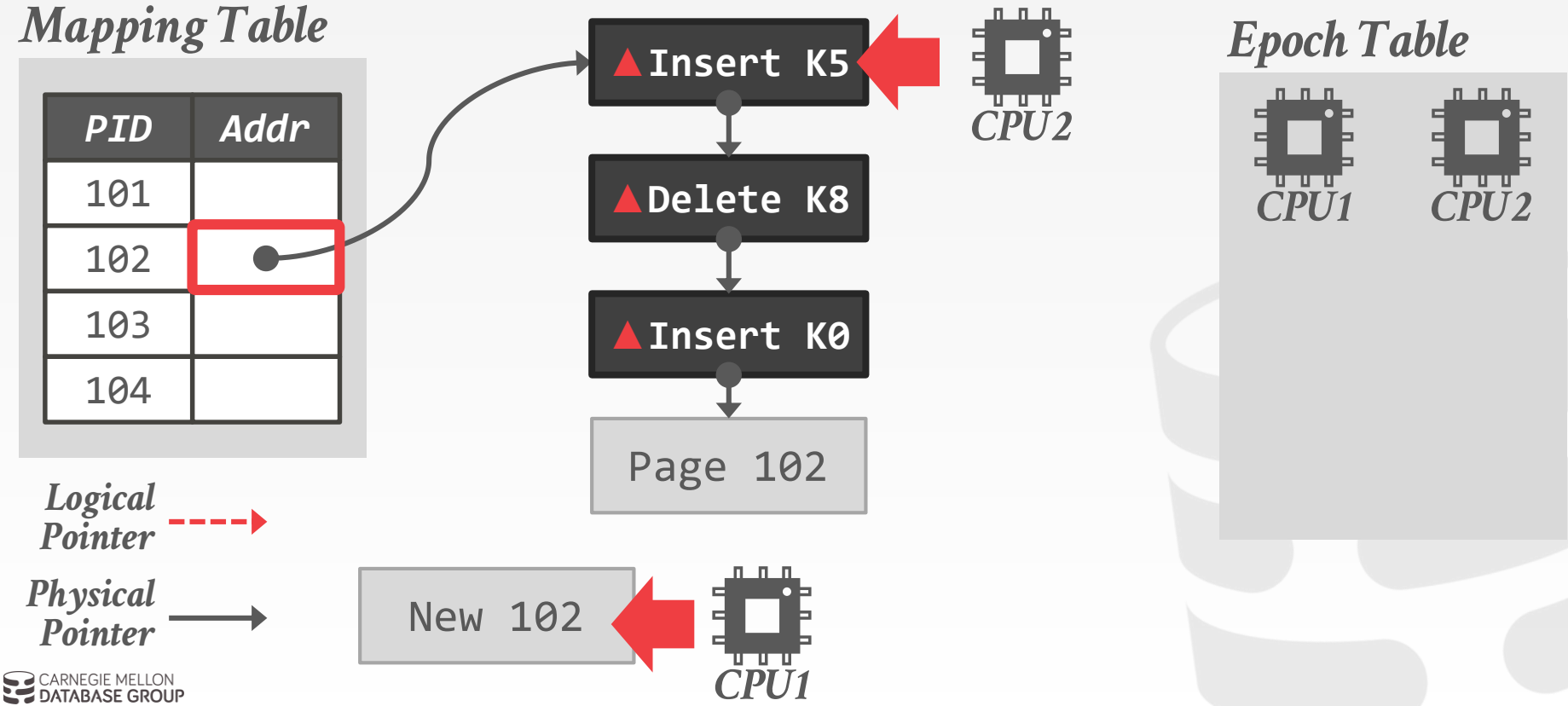
BW-TREE: GARBAGE COLLECTION



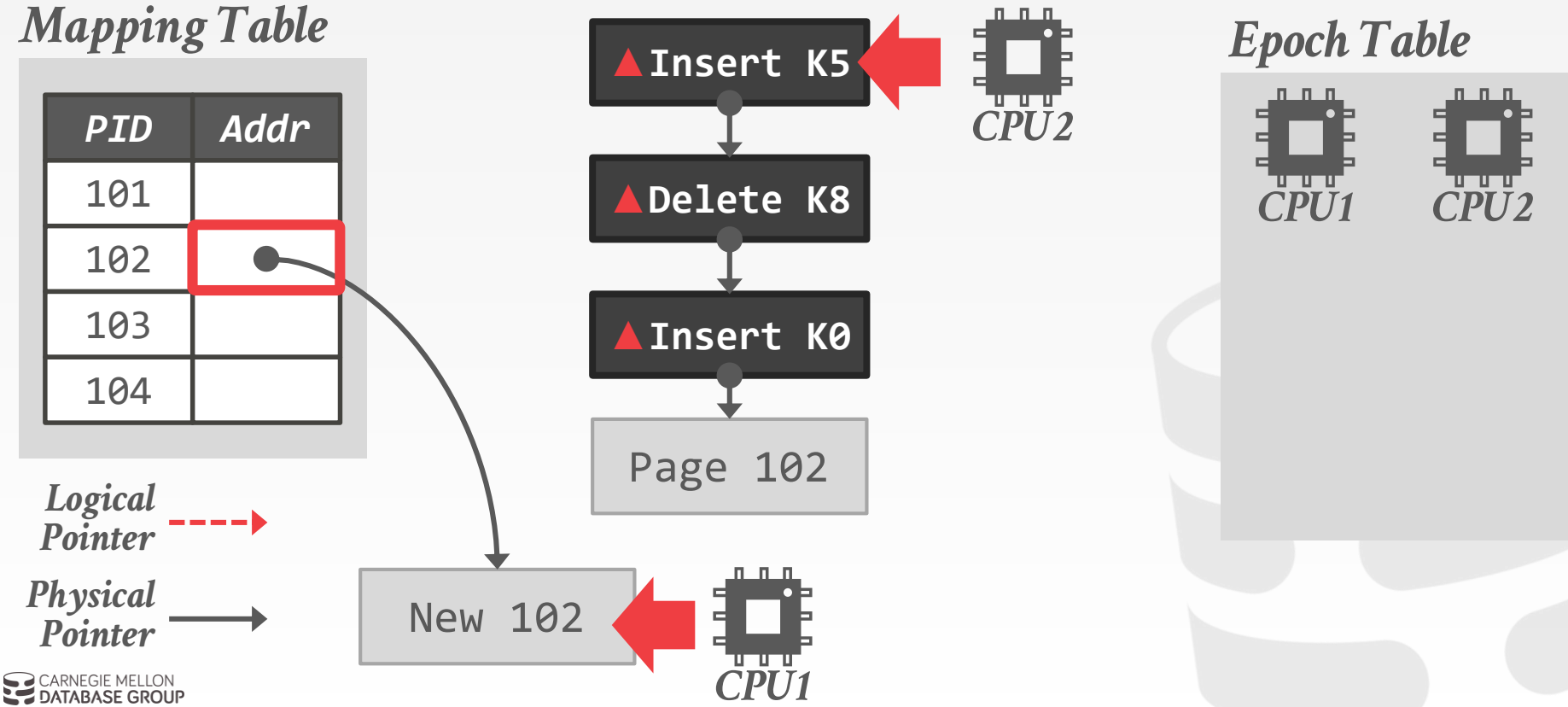
BW-TREE: GARBAGE COLLECTION



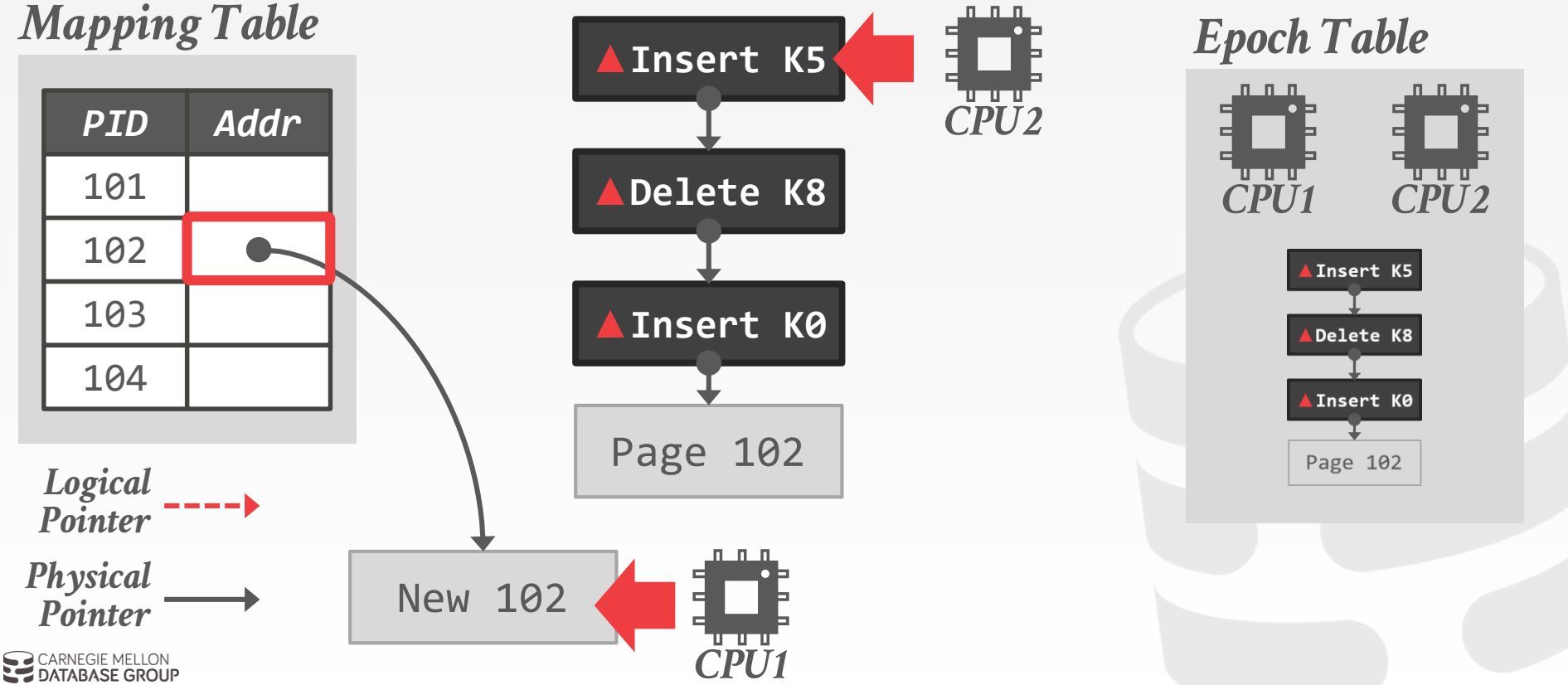
BW-TREE: GARBAGE COLLECTION



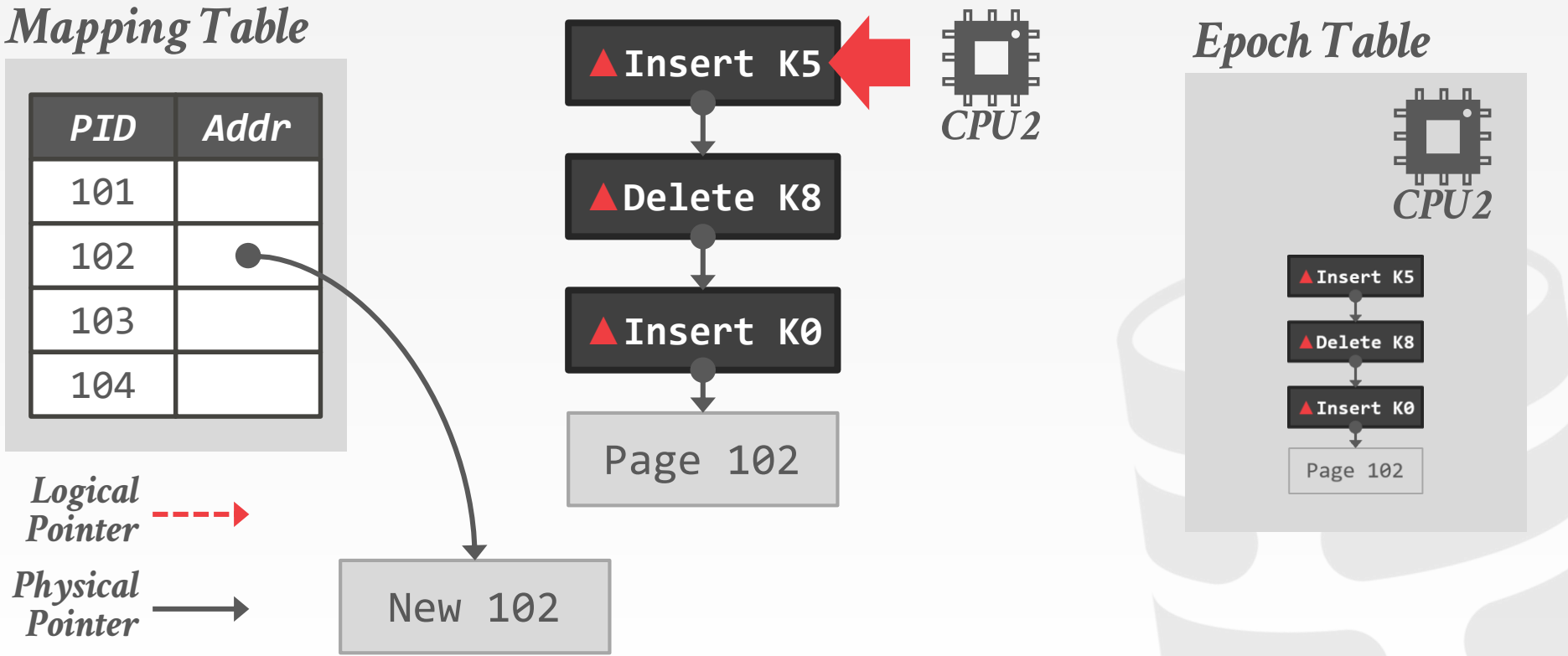
BW-TREE: GARBAGE COLLECTION



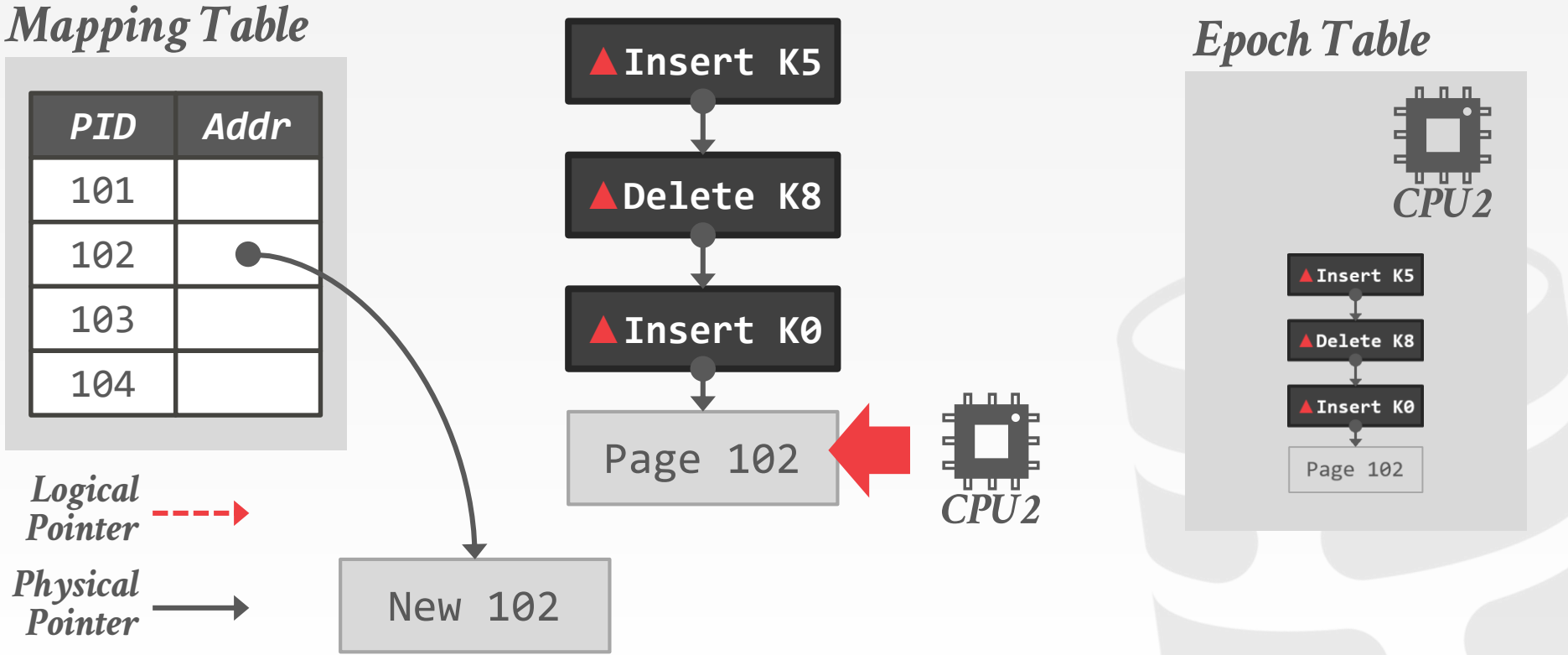
BW-TREE: GARBAGE COLLECTION



BW-TREE: GARBAGE COLLECTION



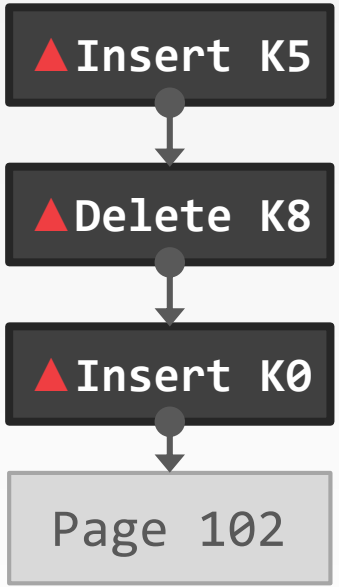
BW-TREE: GARBAGE COLLECTION



BW-TREE: GARBAGE COLLECTION

Mapping Table

PID	Addr
101	
102	
103	
104	



Epoch Table

Logical Pointer ----→

Physical Pointer ——→

New 102

BW-TREE: GARBAGE COLLECTION

Mapping Table

PID	Addr
101	
102	
103	
104	

Logical Pointer ----->

Physical Pointer ——>

New 102

Epoch Table

BW-TREE: STRUCTURE MODIFICATIONS

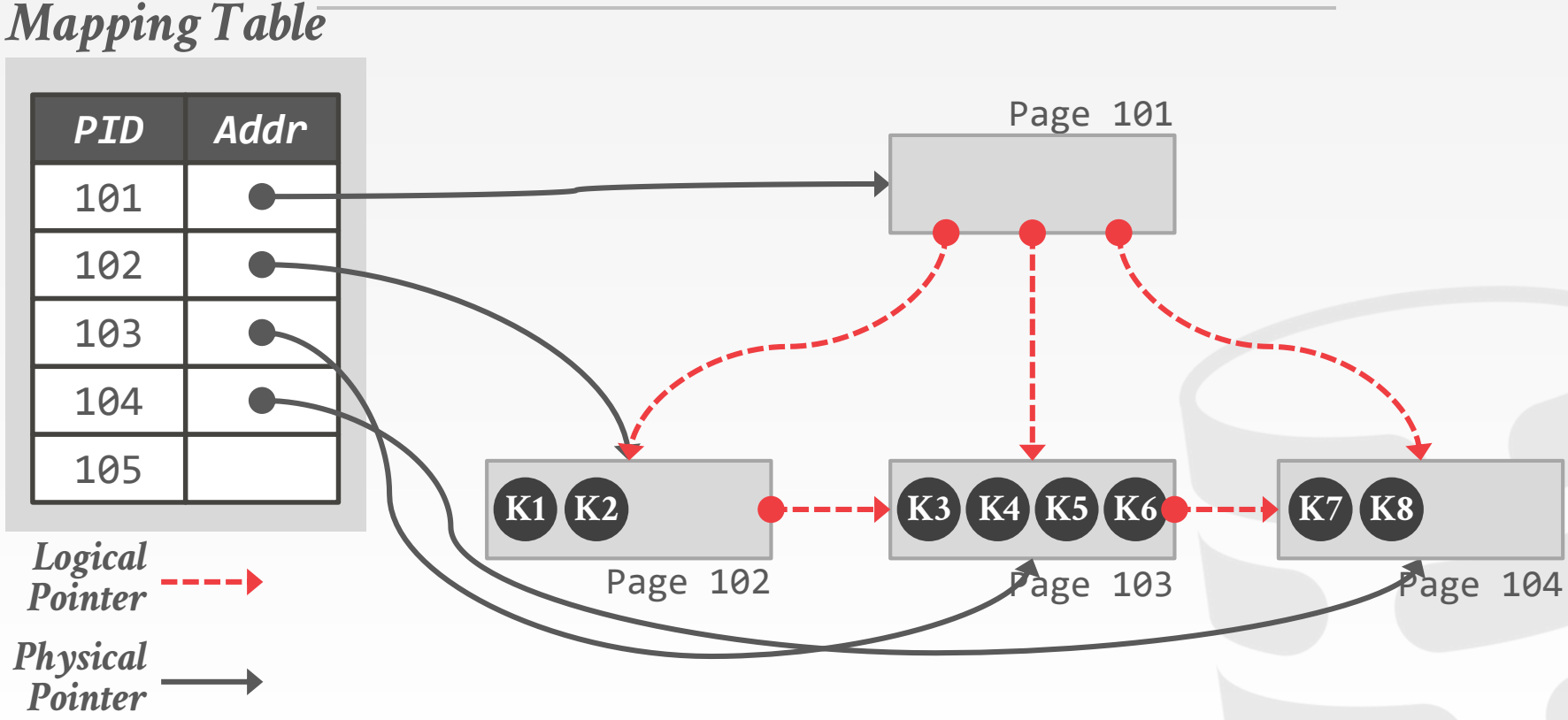
Split Delta Record

→ 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.

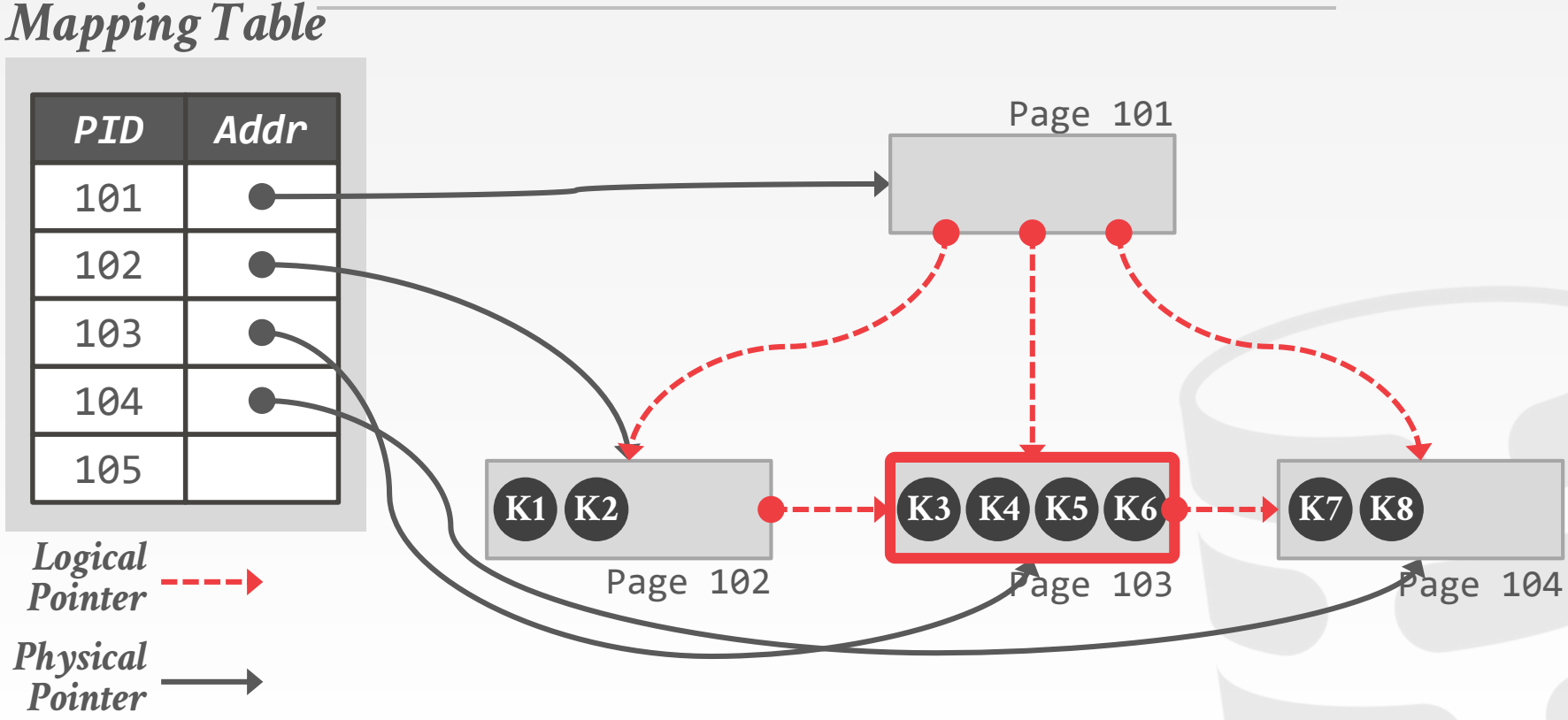
Separator Delta Record

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

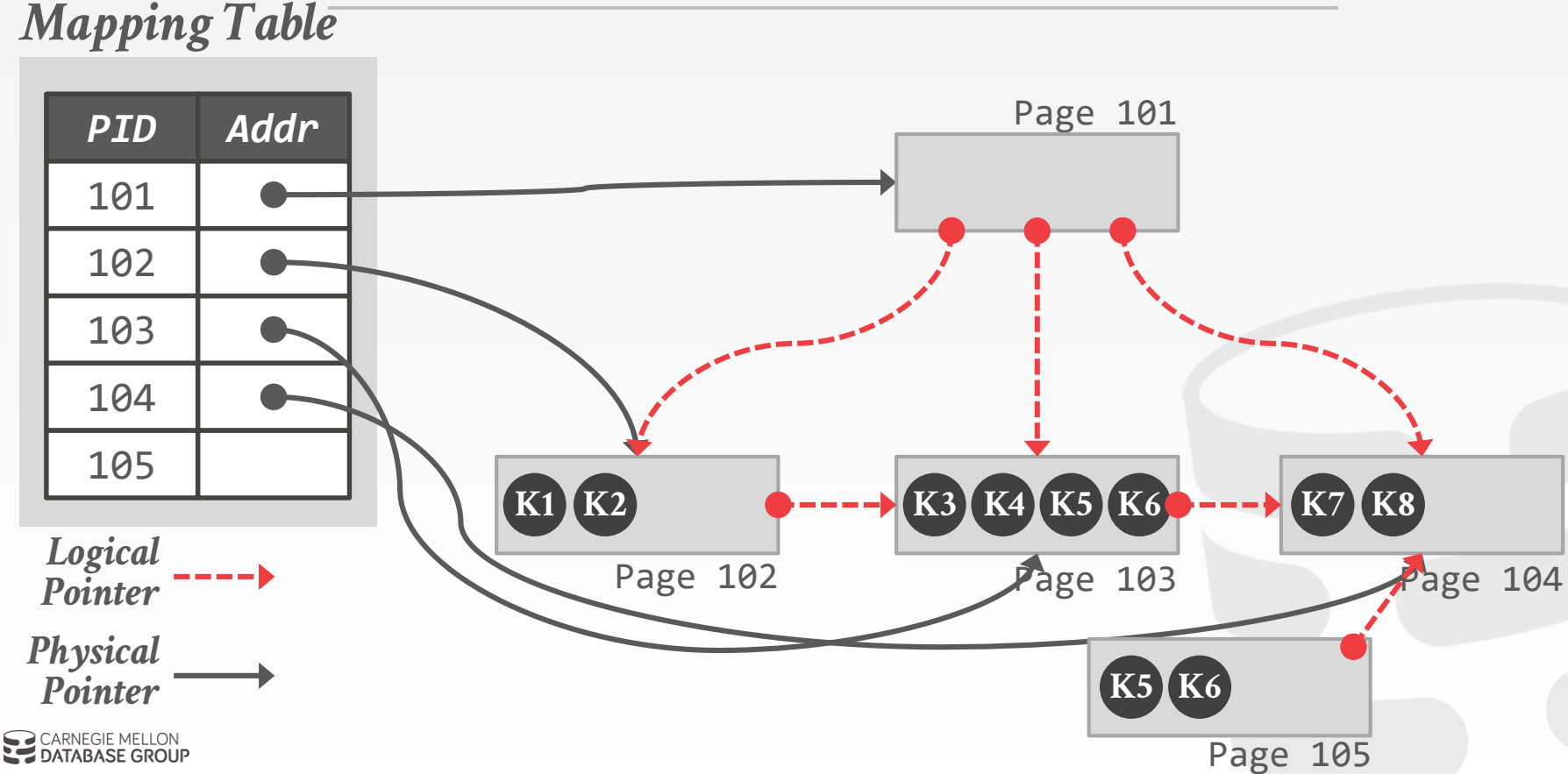
BW-TREE: STRUCTURE MODIFICATIONS



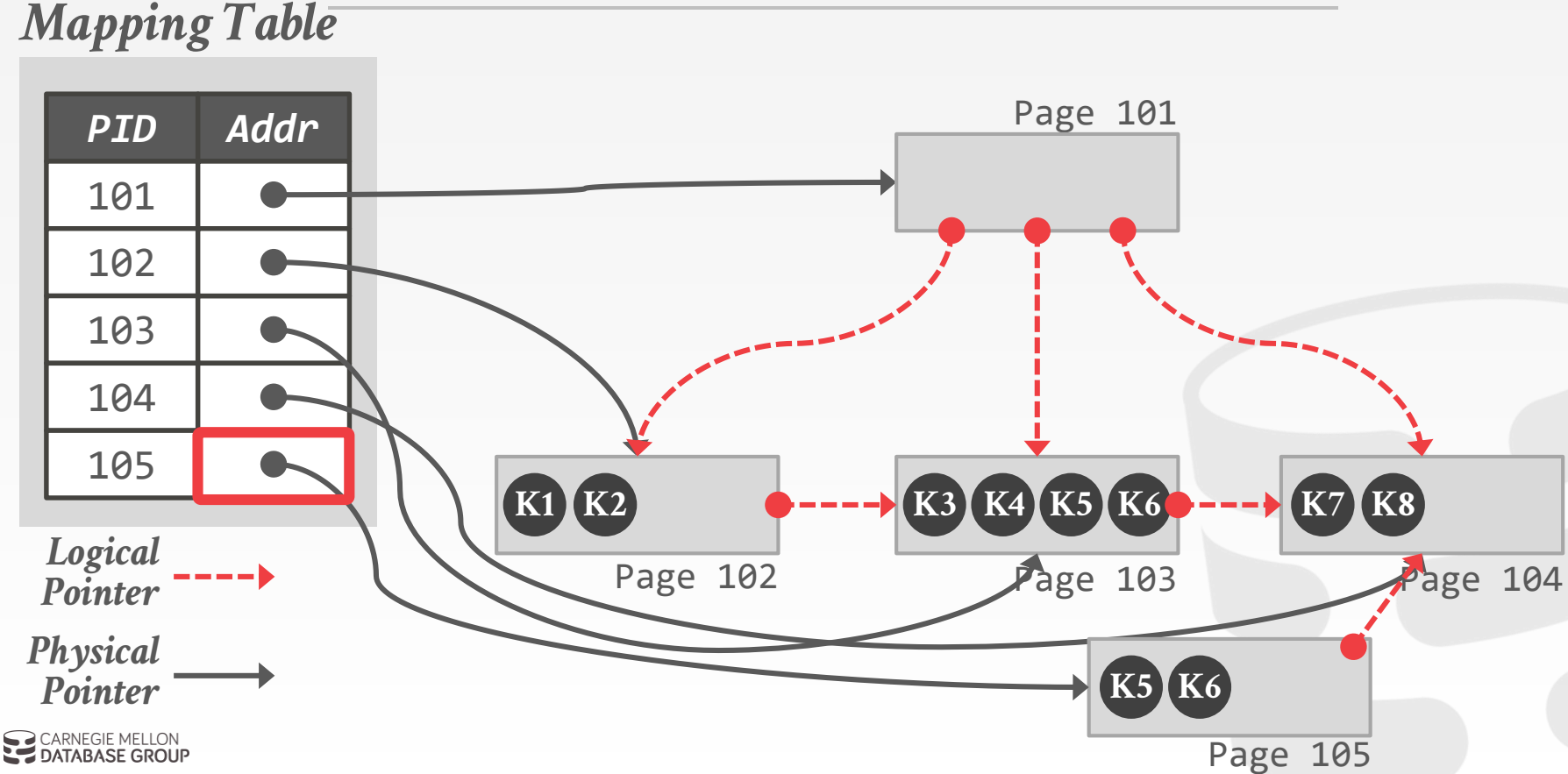
BW-TREE: STRUCTURE MODIFICATIONS



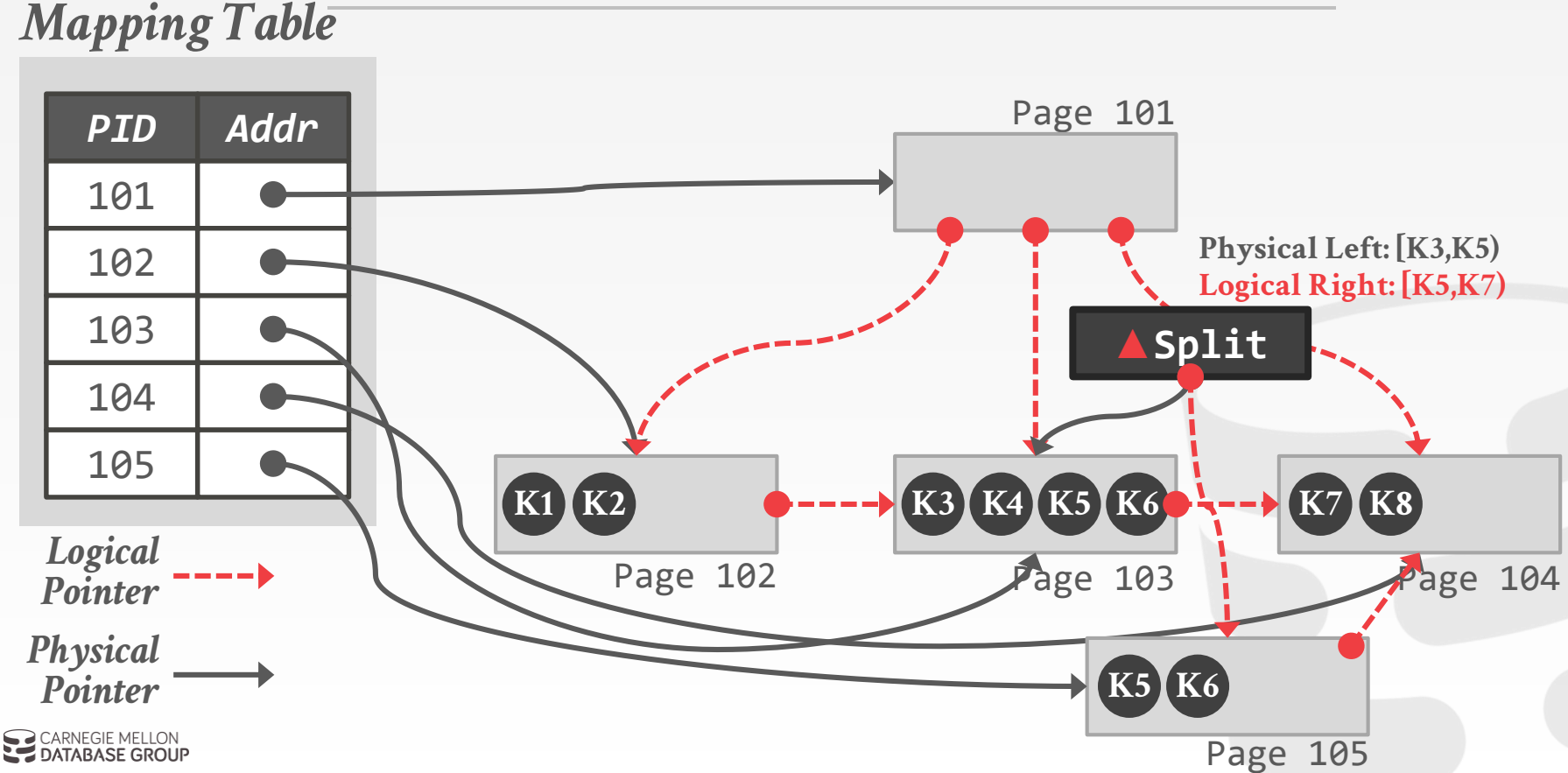
BW-TREE: STRUCTURE MODIFICATIONS



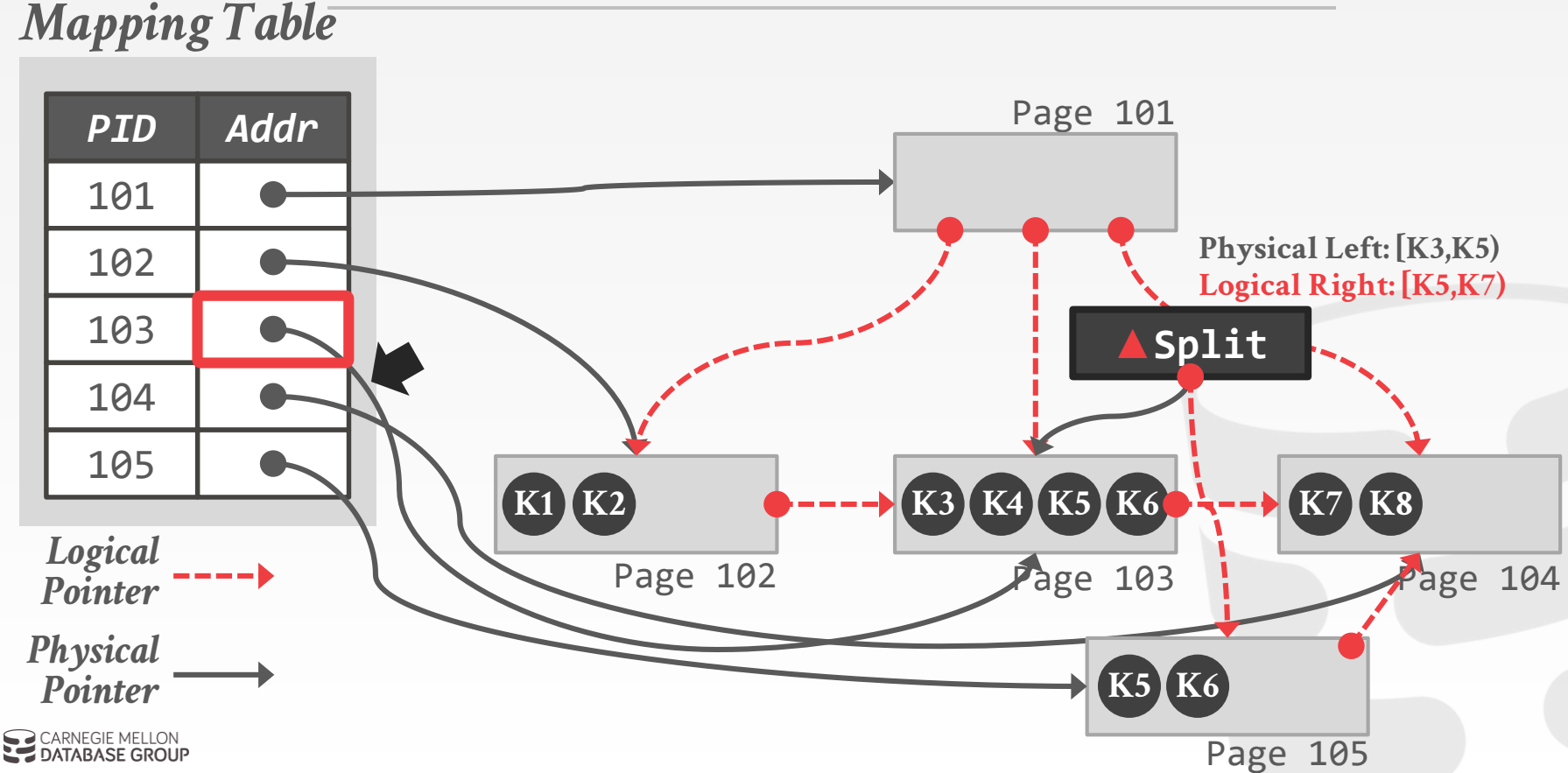
BW-TREE: STRUCTURE MODIFICATIONS



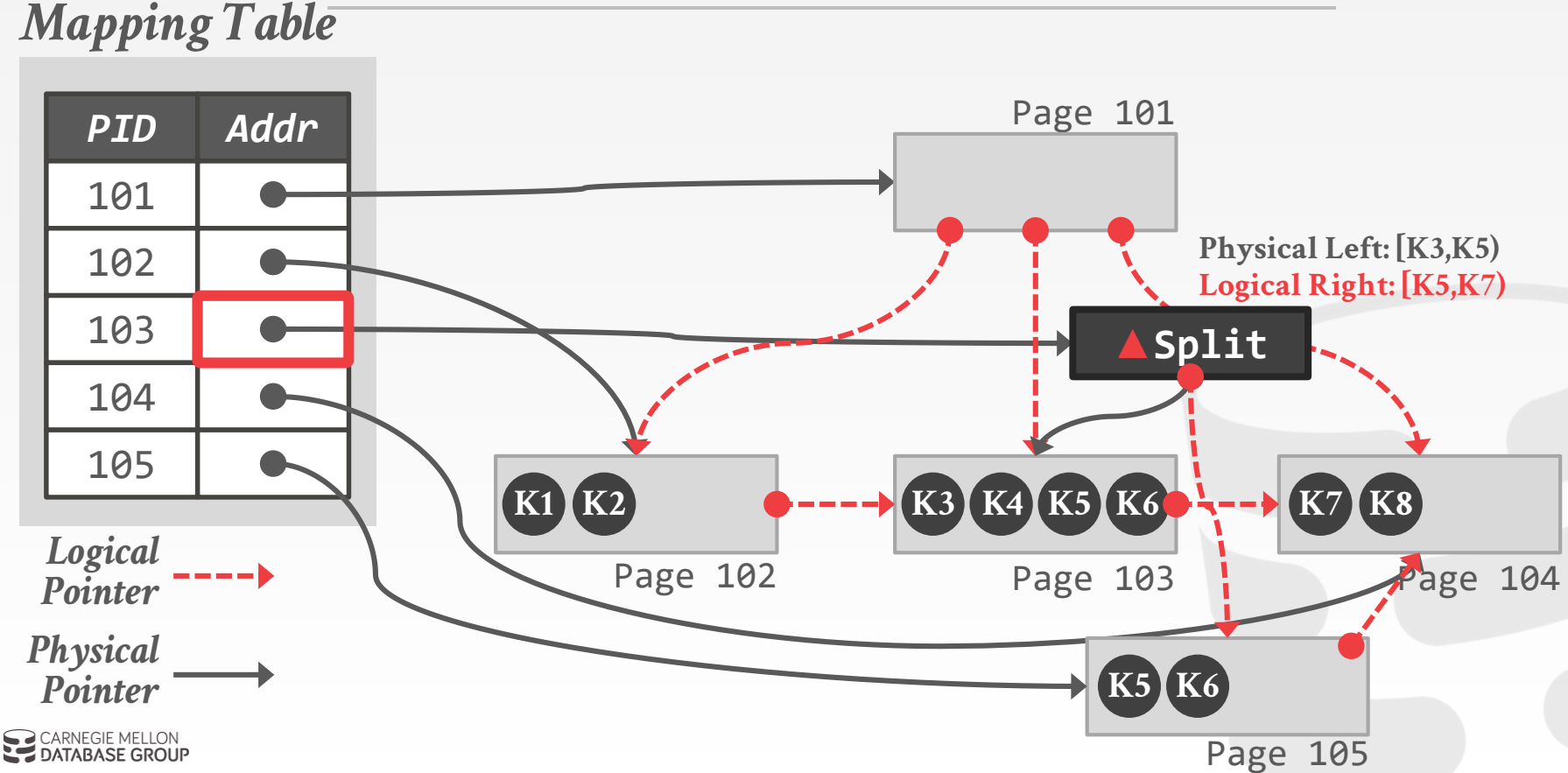
BW-TREE: STRUCTURE MODIFICATIONS



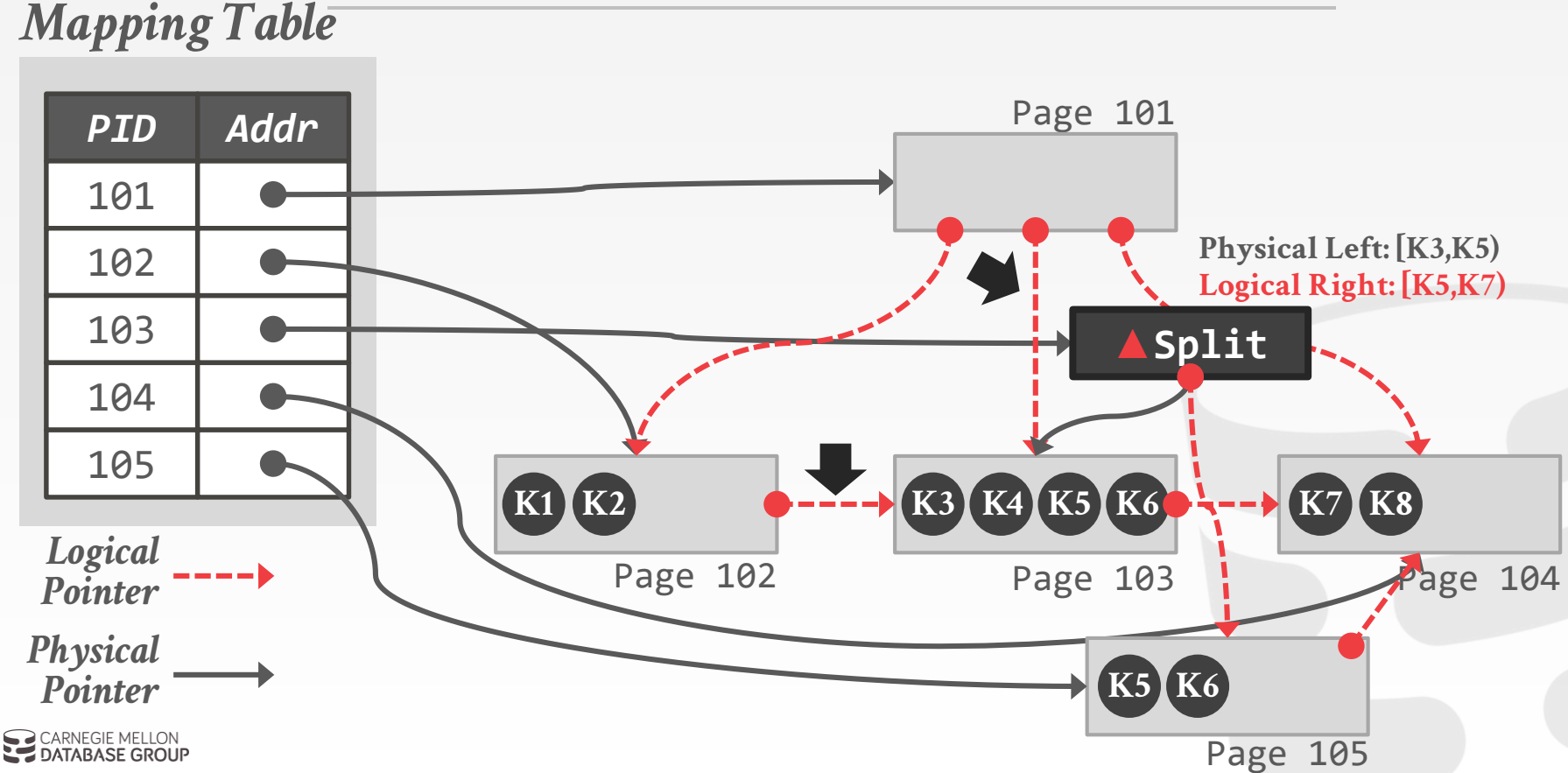
BW-TREE: STRUCTURE MODIFICATIONS



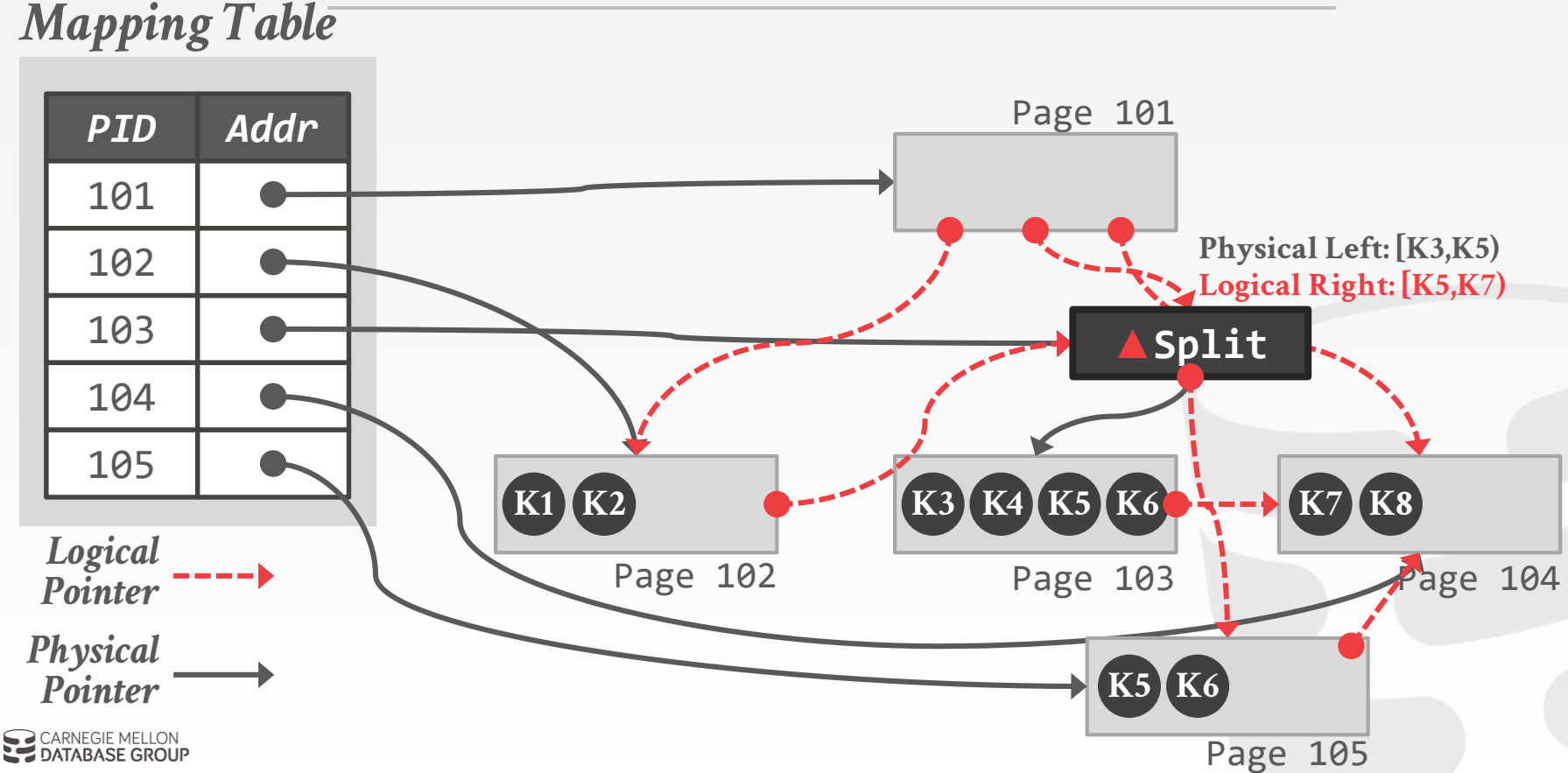
BW-TREE: STRUCTURE MODIFICATIONS



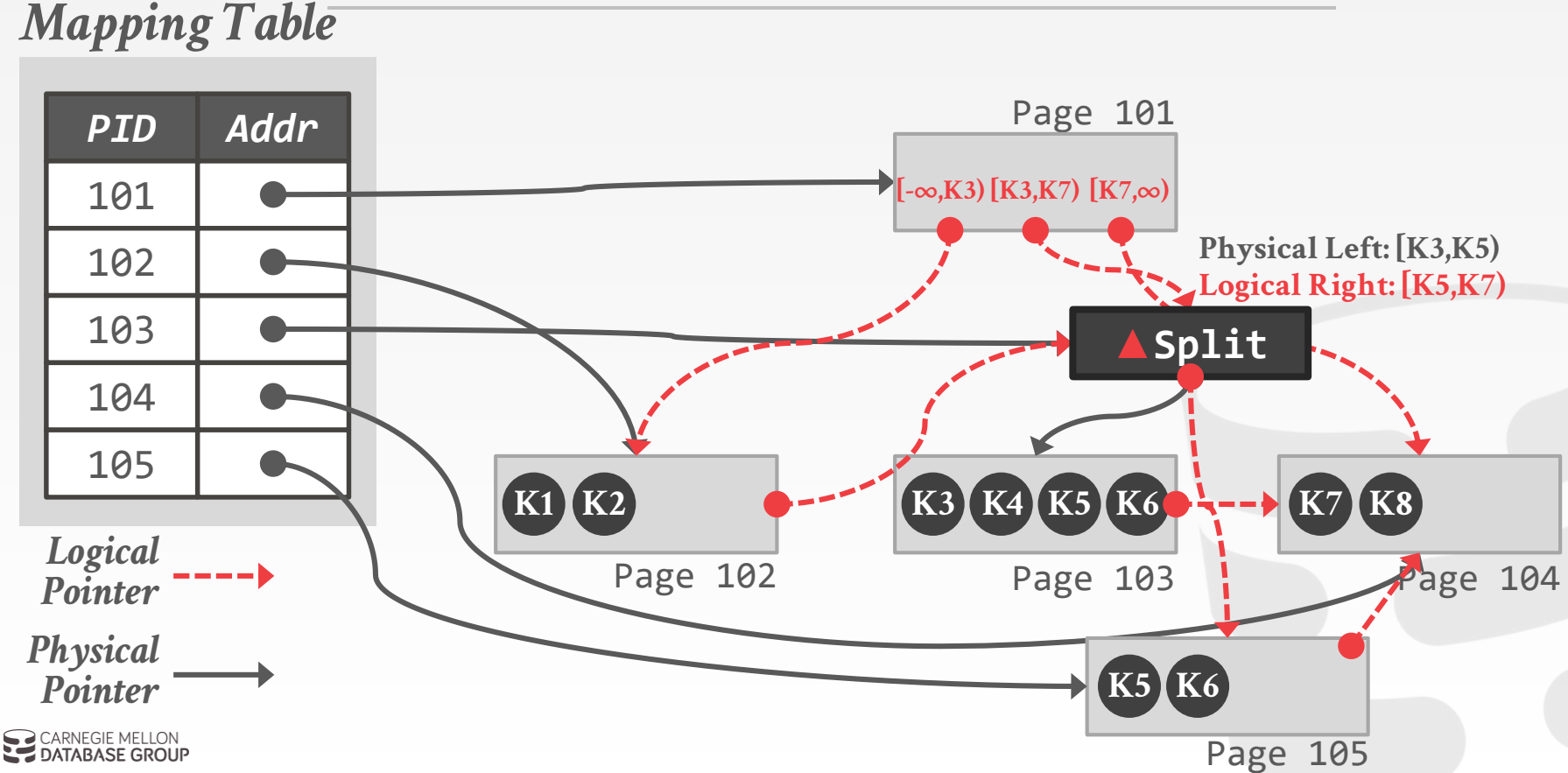
BW-TREE: STRUCTURE MODIFICATIONS



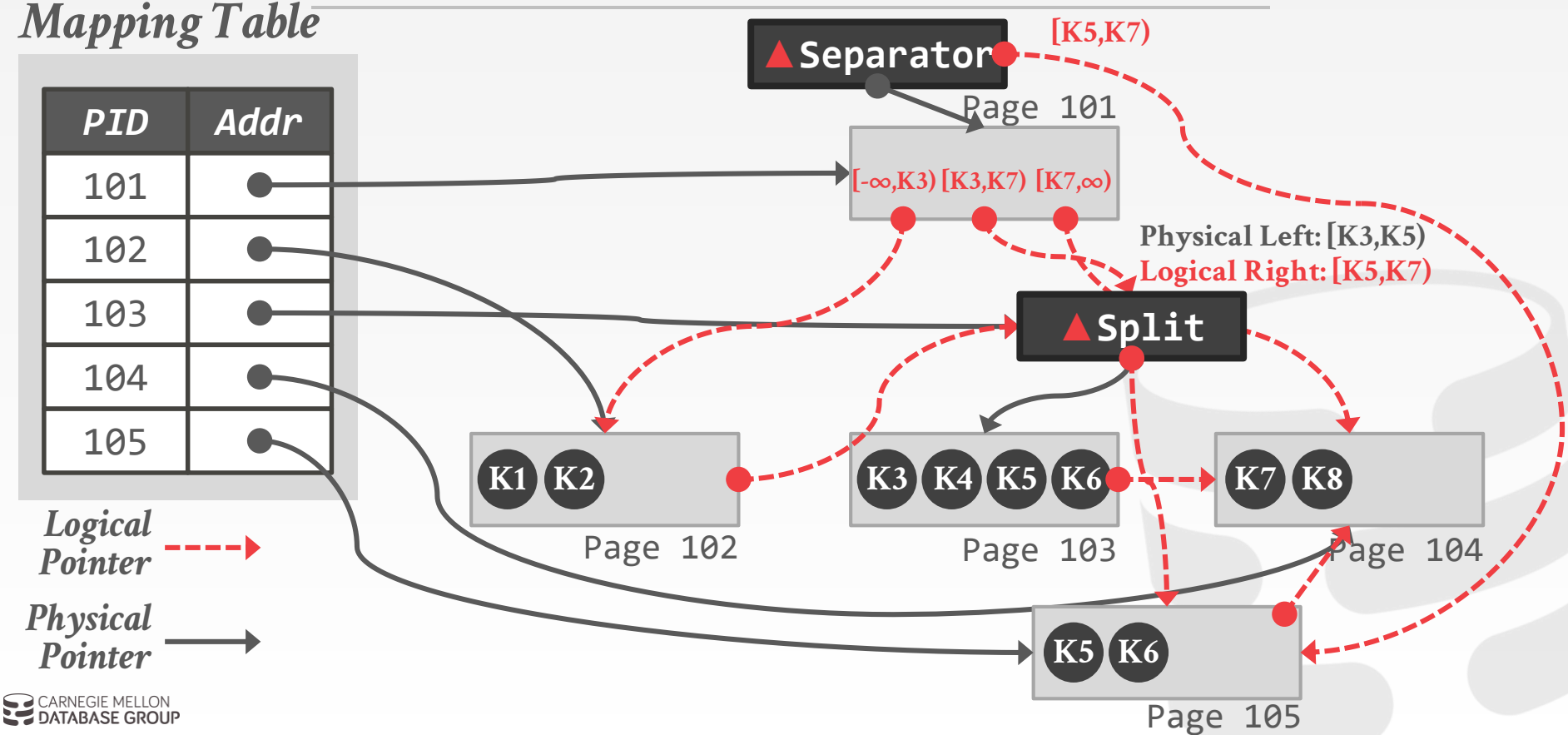
BW-TREE: STRUCTURE MODIFICATIONS



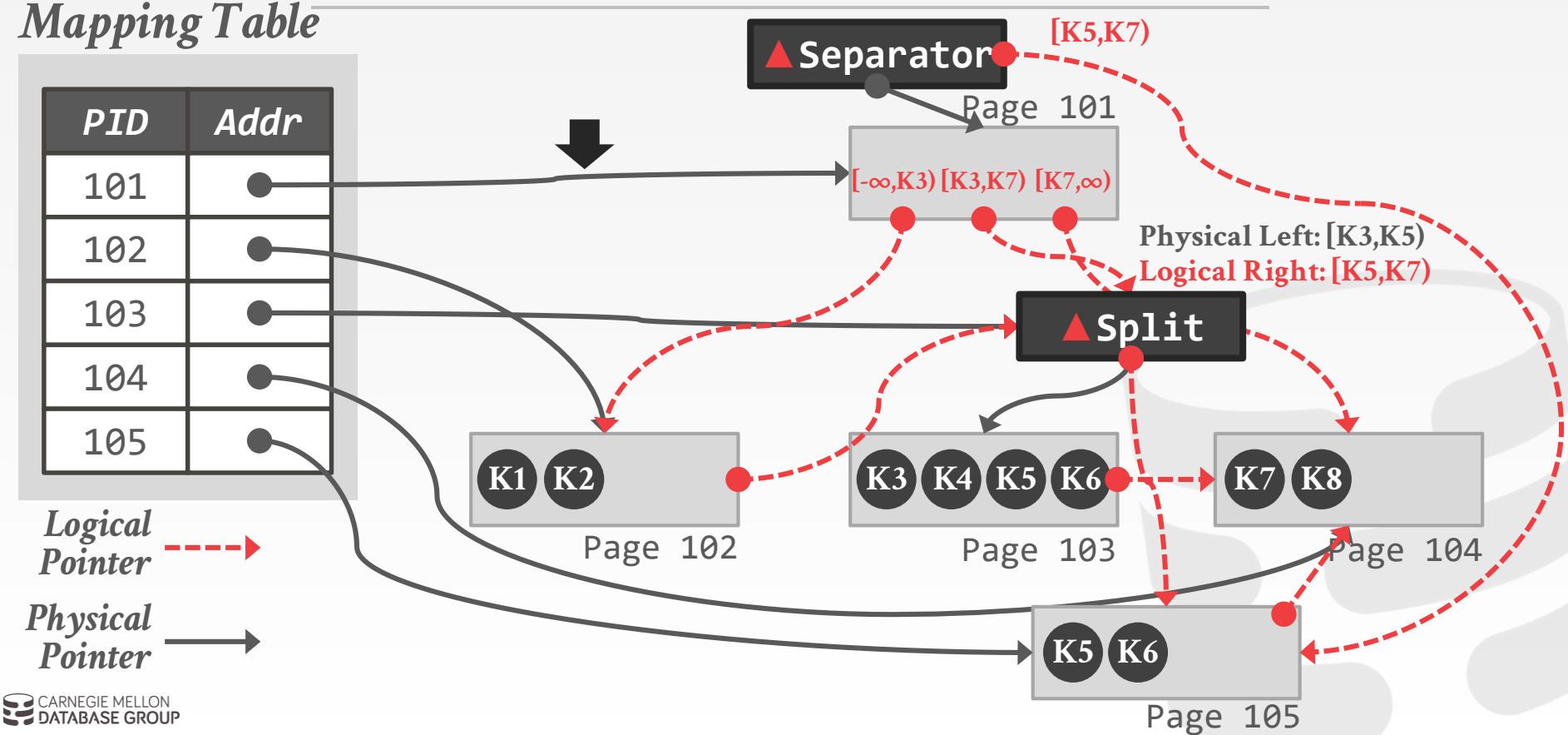
BW-TREE: STRUCTURE MODIFICATIONS



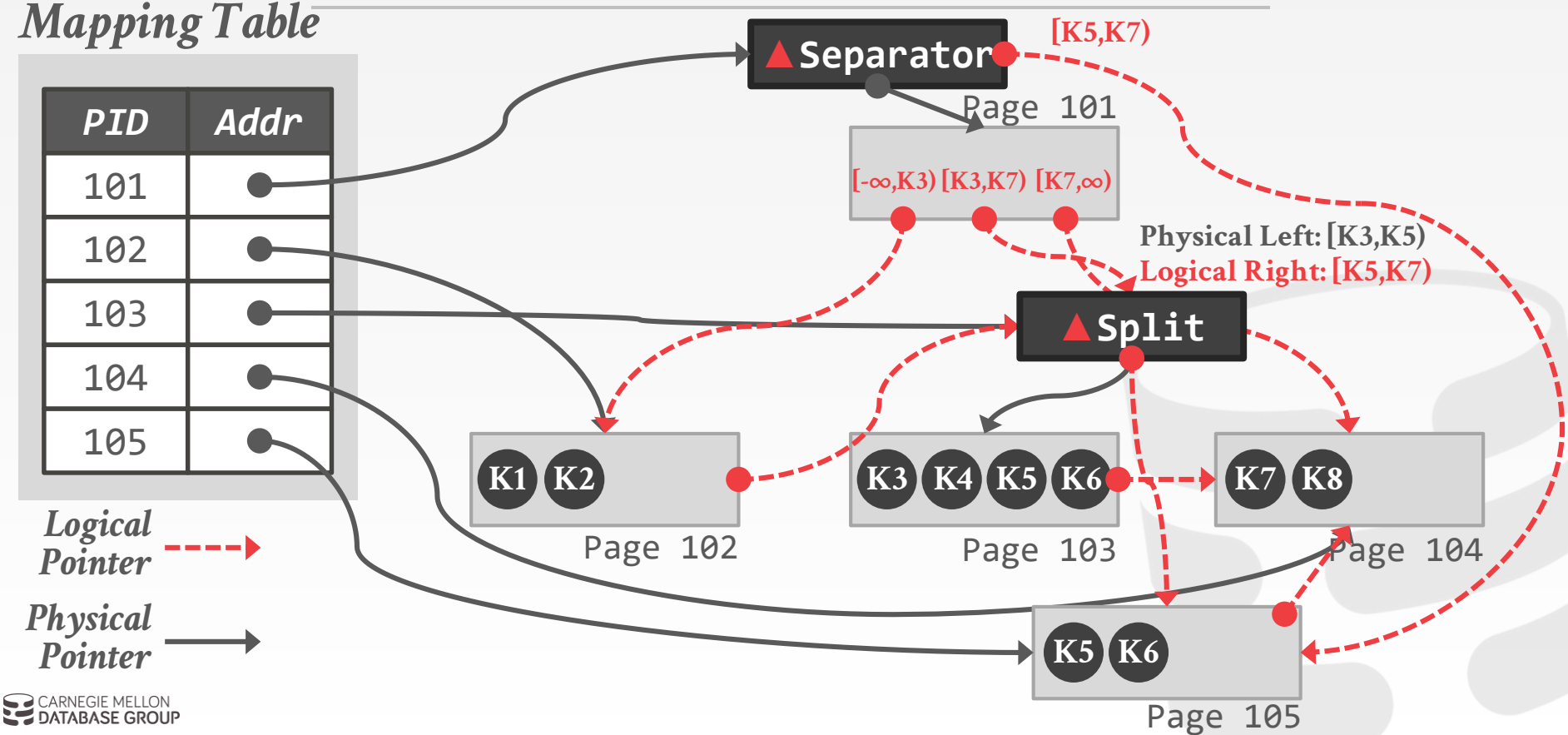
BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS



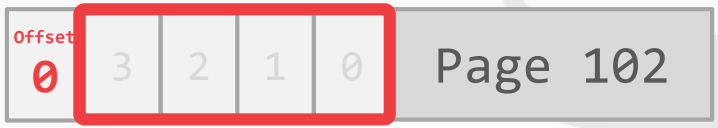
CMU OPEN BW-TREE

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	



Delta Slots

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

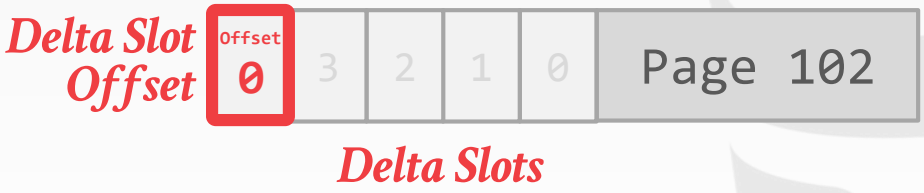
CMU OPEN BW-TREE

Optimization #1: Pre-Allocated Delta Records

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

Mapping Table

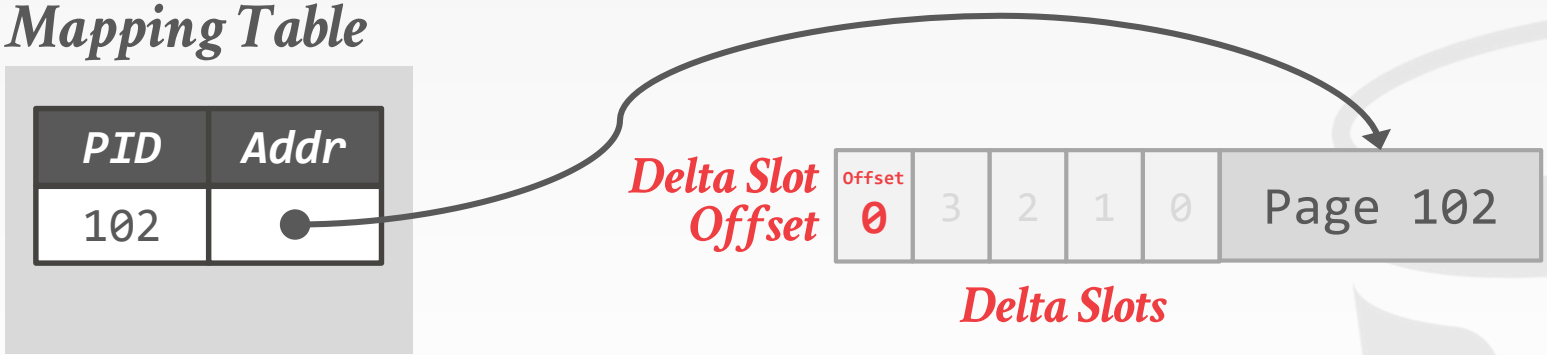
PID	Addr
102	



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.



CMU OPEN BW-TREE

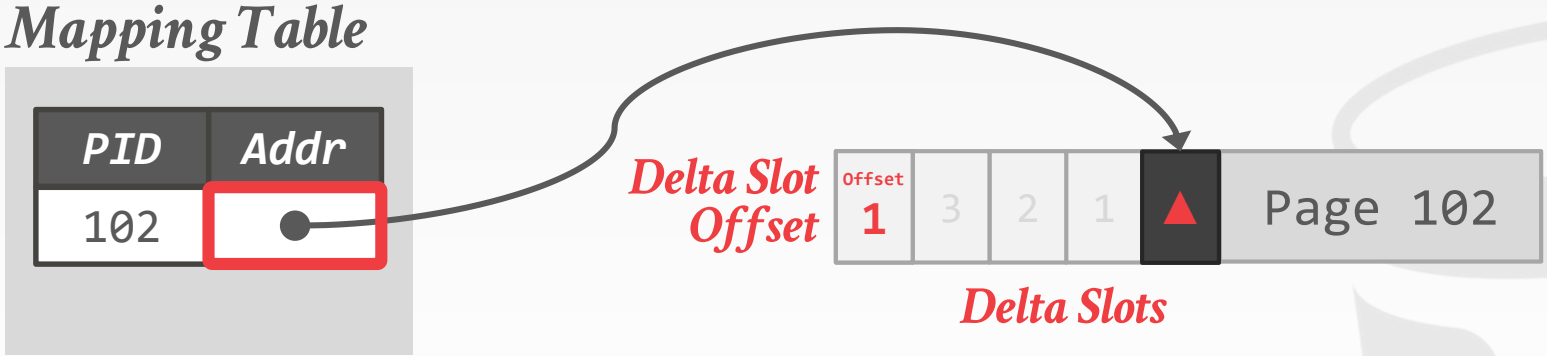
Optimization #1: Pre-Allocated Delta Records

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

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.



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

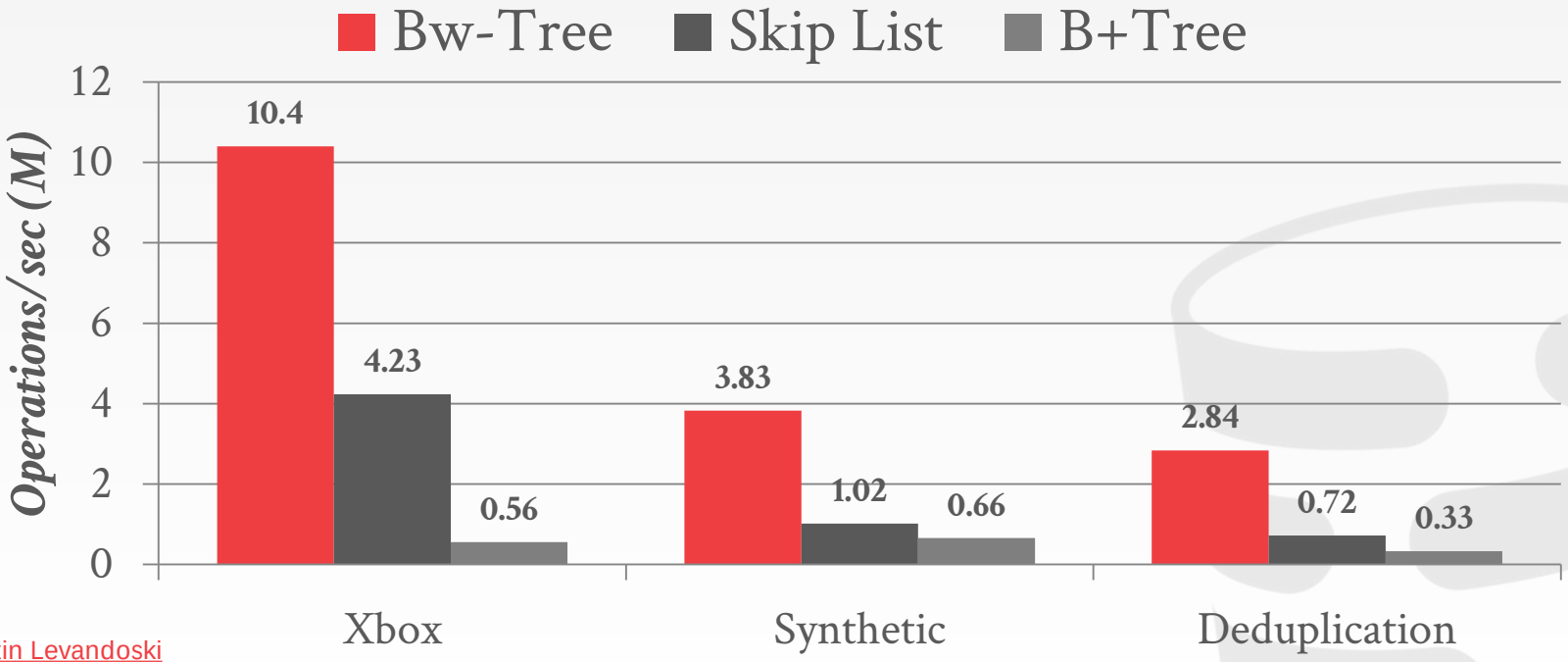
Use virtual memory to allocate the entire array without backing it with physical memory.

→ 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

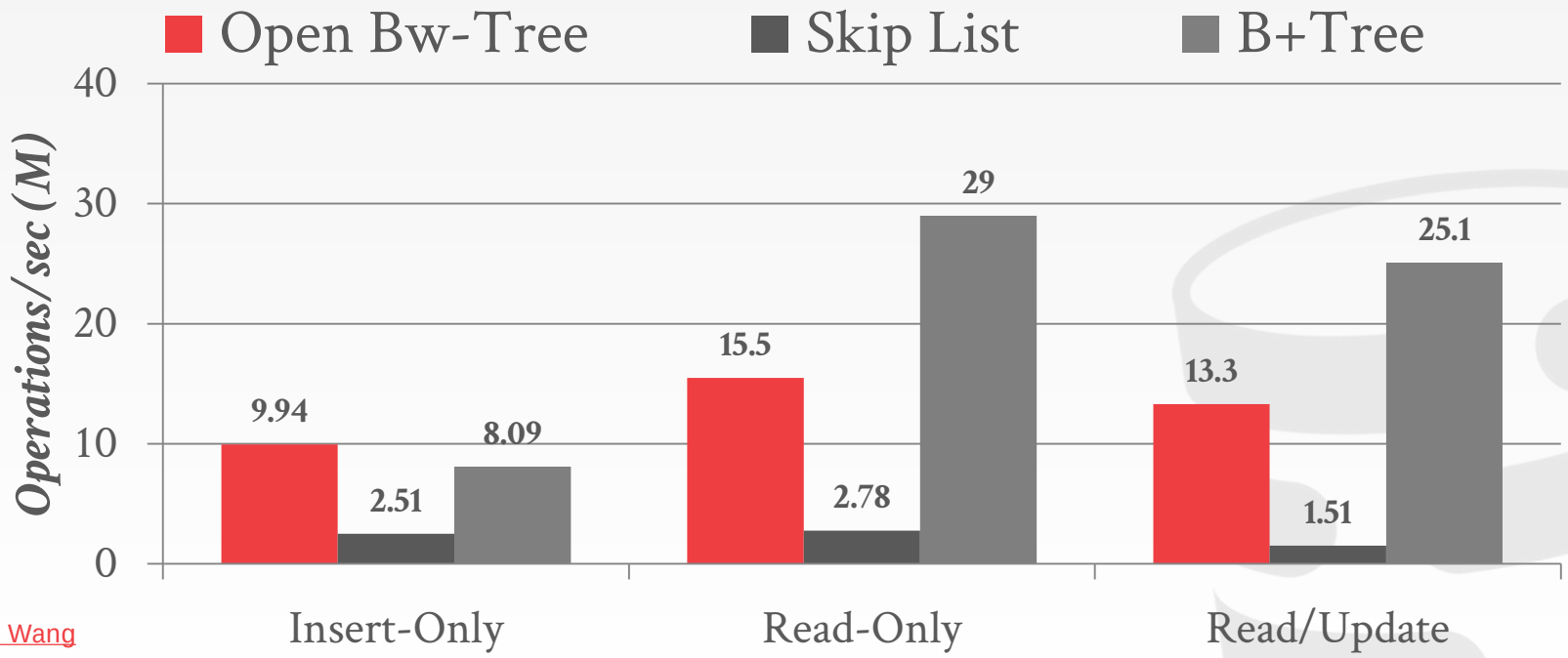
Processor: 1 socket, 4 cores w/ 2×HT



Source: Justin Levandoski

BW-TREE: PERFORMANCE

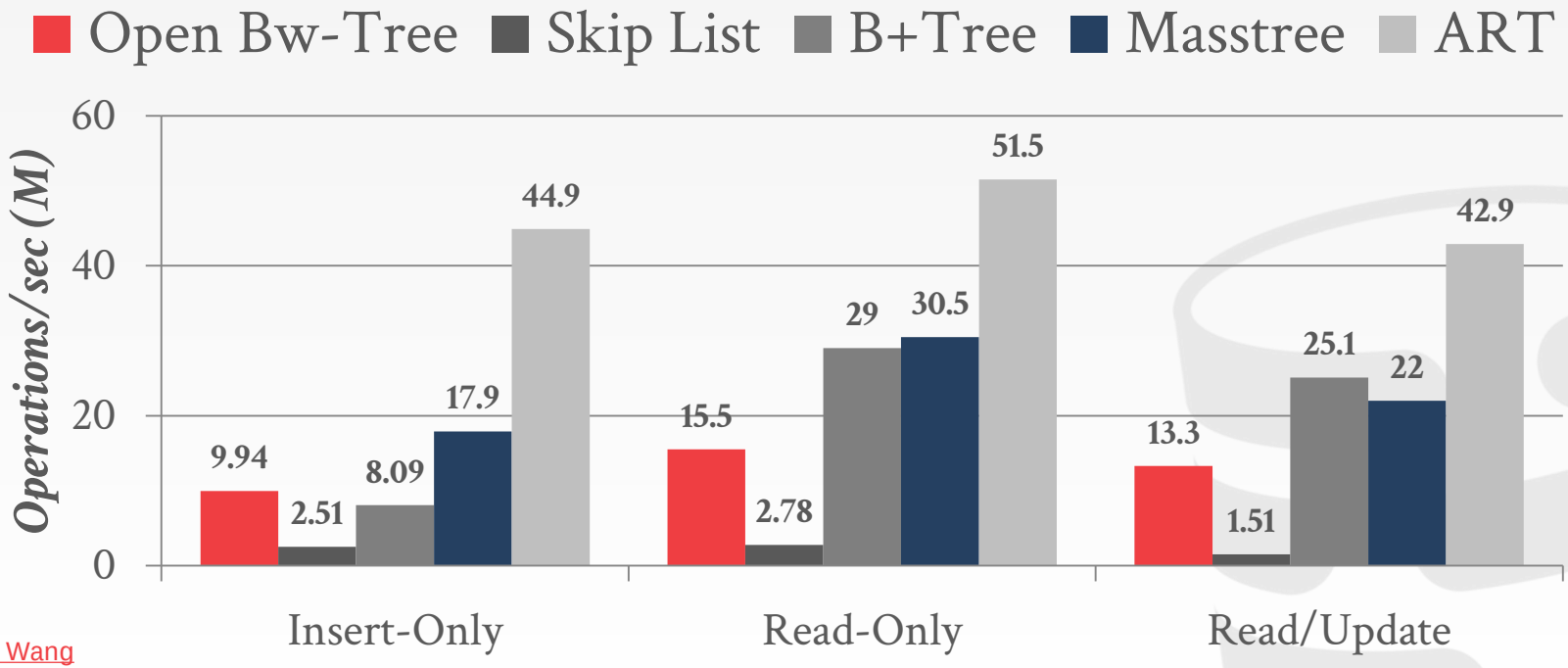
Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)



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.

Non-concurrent Skip List is easy to implement.

A Bw-Tree is hard to implement.

NEXT CLASS

Index Key Representation

Memory Allocation & Garbage Collection

Trie Data Structures