

# SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory

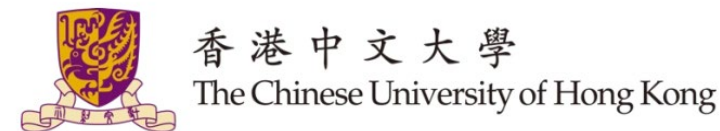
Xuchuan Luo<sup>1</sup>, Pengfei Zuo<sup>2</sup>, Jiacheng Shen<sup>3</sup>, Jiazhen Gu<sup>3</sup>,  
Xin Wang<sup>1,4</sup>, Michael R. Lyu<sup>3</sup>, and Yangfan Zhou<sup>1,4</sup>

<sup>1</sup>*School of Computer Science, Fudan University*

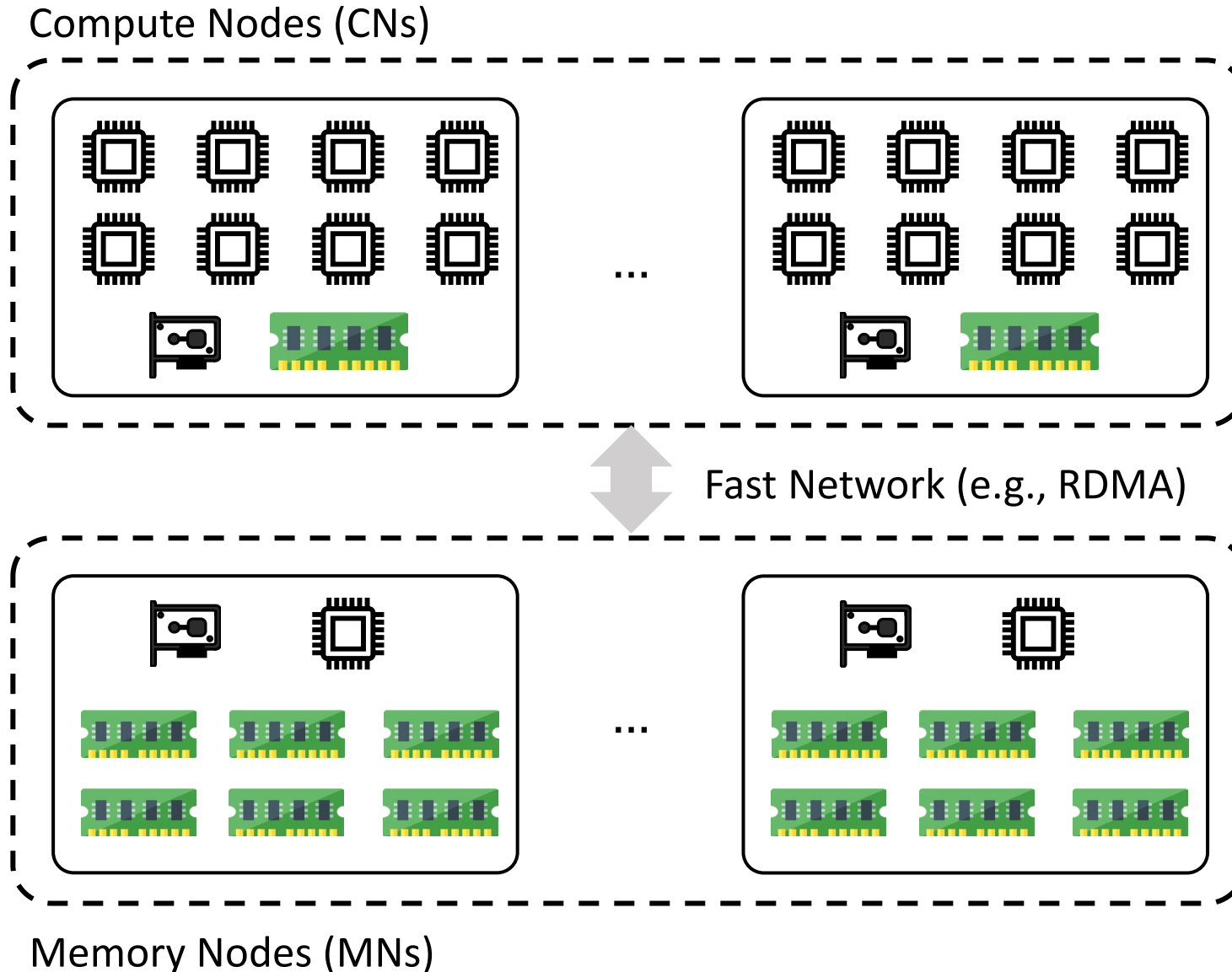
<sup>2</sup>*Huawei Cloud*

<sup>3</sup>*The Chinese University of Hong Kong*

<sup>4</sup>*Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China*



# Disaggregated Memory (DM)



## Benefits:

- ✓ Resource utilization
- ✓ Elasticity

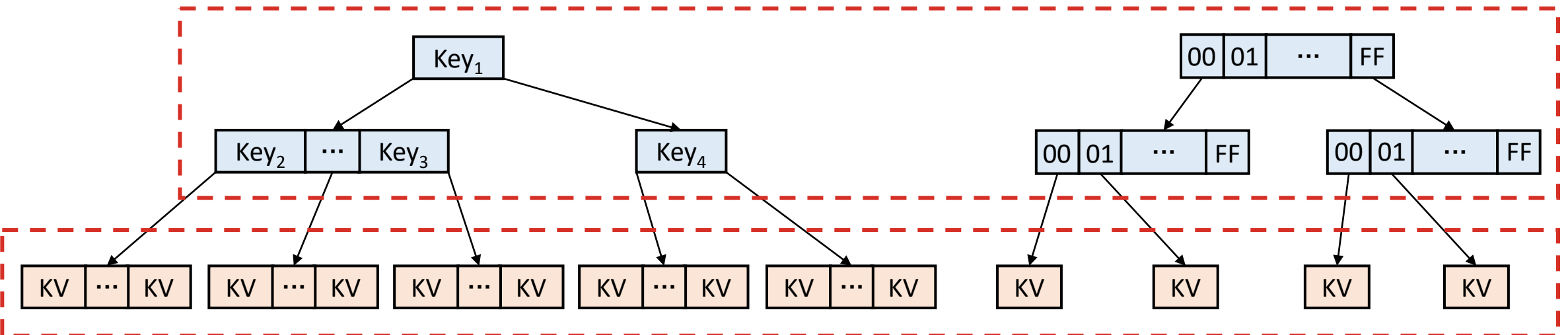
# Tree Indexes

## B+ Tree

- Each internal node stores **entire keys**
- Each leaf node holds **multiple KVs**

## Radix Tree

- Each internal node stores **partial keys**
- Each leaf node holds a **single KV**




# Tree Indexes on Disaggregated Memory

Existing tree indexes on DM are based on the **B+ tree**: FG<sup>[1]</sup>, Sherman<sup>[2]</sup>

Clients 

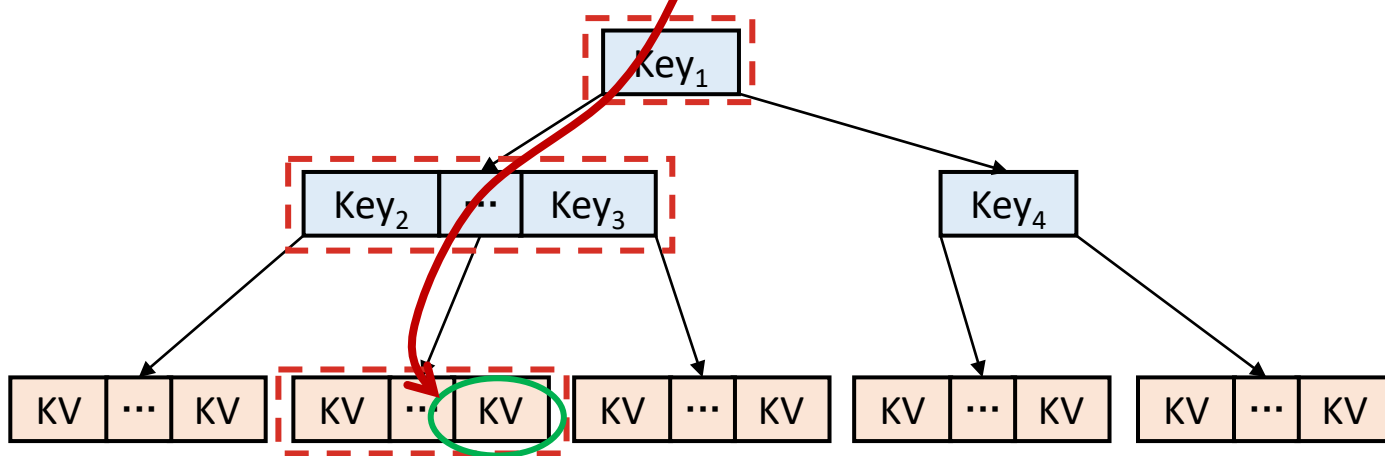
computing-side

memory-side

 Read/write nodes

 **Problem:**

Read/write amplifications of B+ trees  
➤ Exacerbate the network bandwidth bottleneck of DM



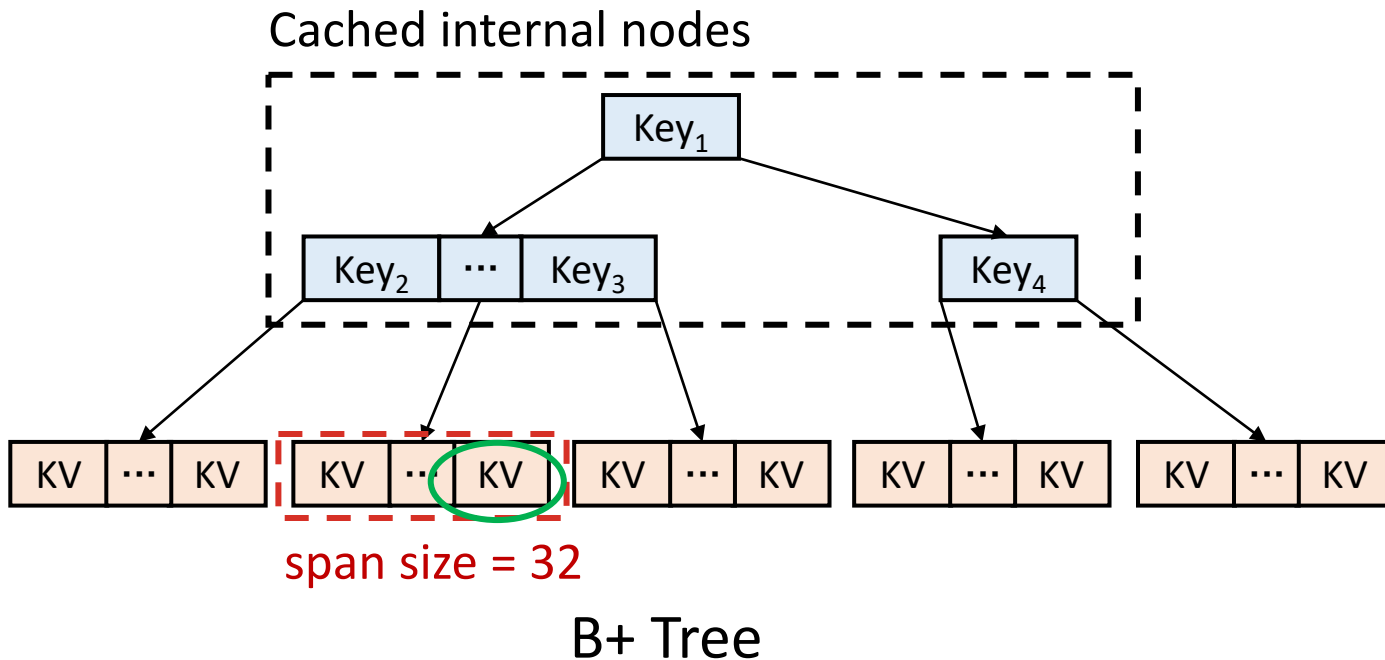
[1] Tobias Ziegler et al. Designing distributed tree-based index structures for fast RDMA-capable networks. SIGMOD 2019.

[2] Qing Wang et al. Sherman: A write-optimized distributed B+ tree index on disaggregated memory. SIGMOD 2022.

# Tree Indexes on Disaggregated Memory

## Read and write amplification factors:

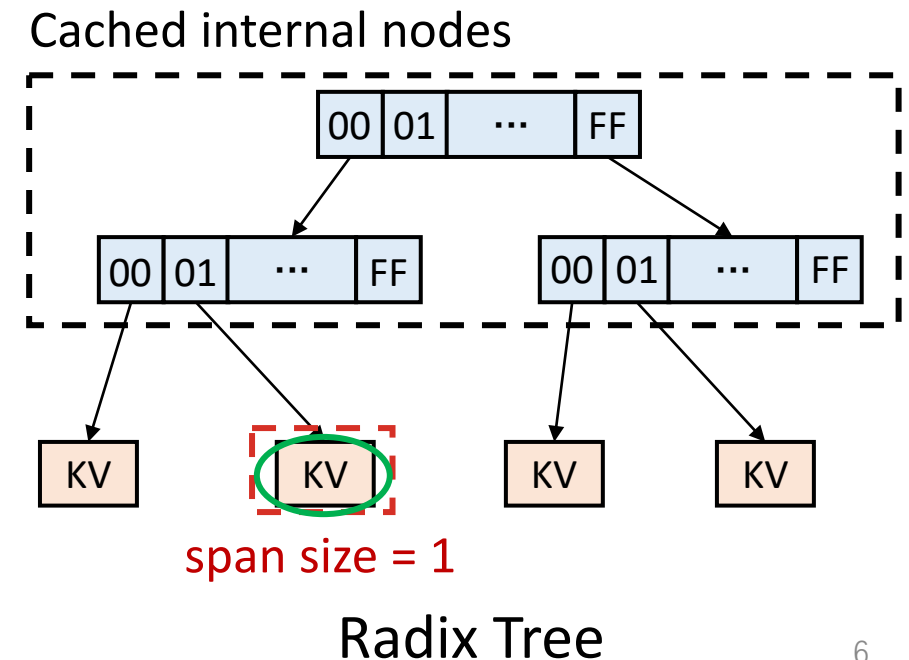
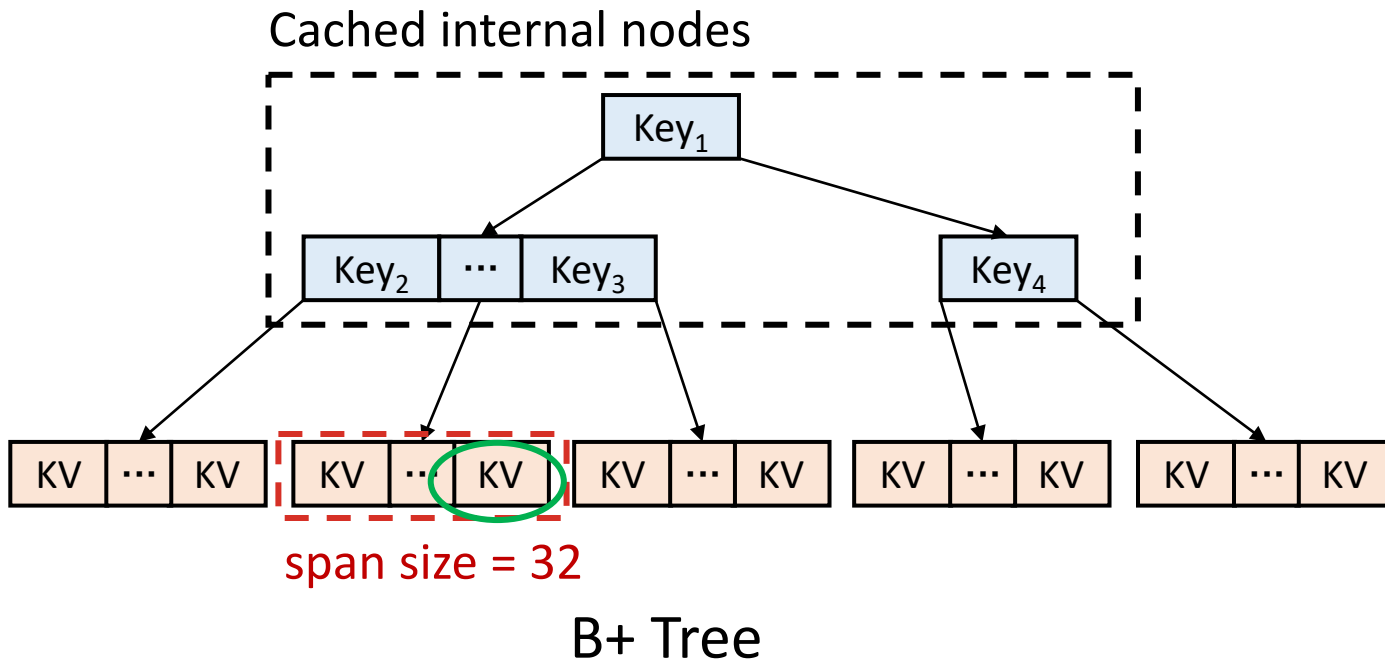
	B+ Tree	Sherman
Read amplification	$\approx 32$	$\approx 33$
Write amplification	$\approx 32$	$\approx 1$



# Tree Indexes on Disaggregated Memory

## Read and write amplification factors:

	B+ Tree	Sherman	Radix Tree
Read amplification	$\approx 32$	$\approx 33$	$\approx 1$
Write amplification	$\approx 32$	$\approx 1$	$\approx 1$



# Tree Indexes on Disaggregated Memory

## Read and write amplification factors:

	B+ Tree	Sherman	Radix Tree
Read amplification	$\approx 32$	$\approx 33$	$\approx 1$
Write amplification	$\approx 32$	$\approx 1$	$\approx 1$

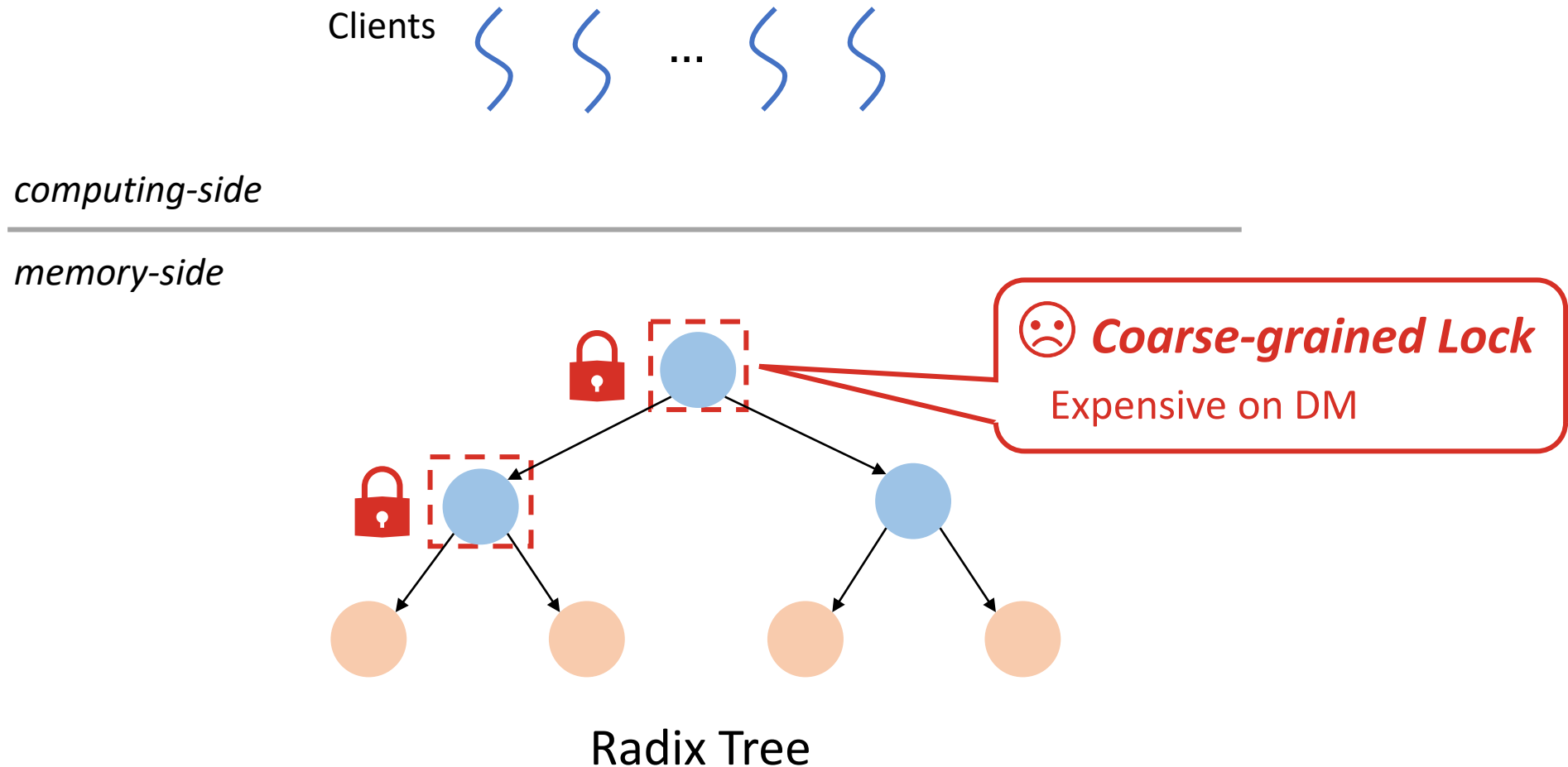
Insight: The radix tree is more suitable for DM than the B+ tree due to smaller read/write amplifications



Our Idea: Using radix tree to build a high-performance tree index on DM

# Challenge 1: Expensive Lock-based Concurrency Control

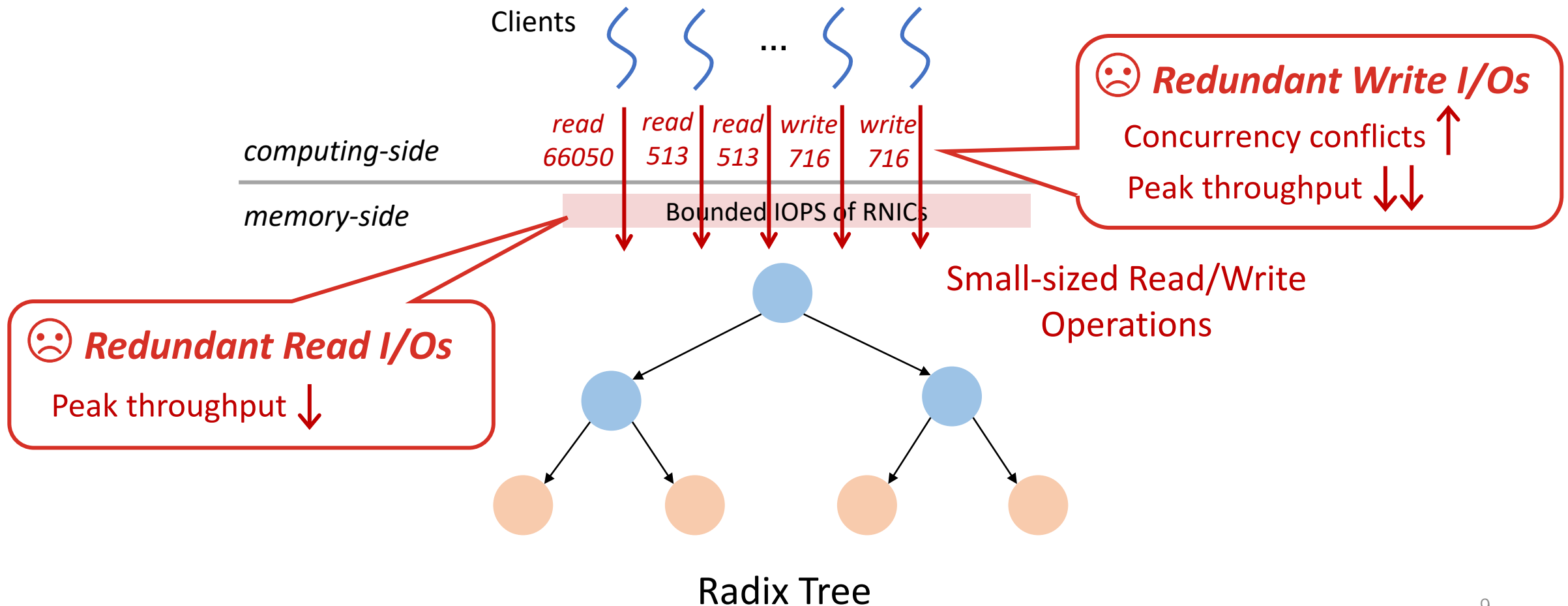
**Lock-based concurrency control of radix trees causes poor write performance**





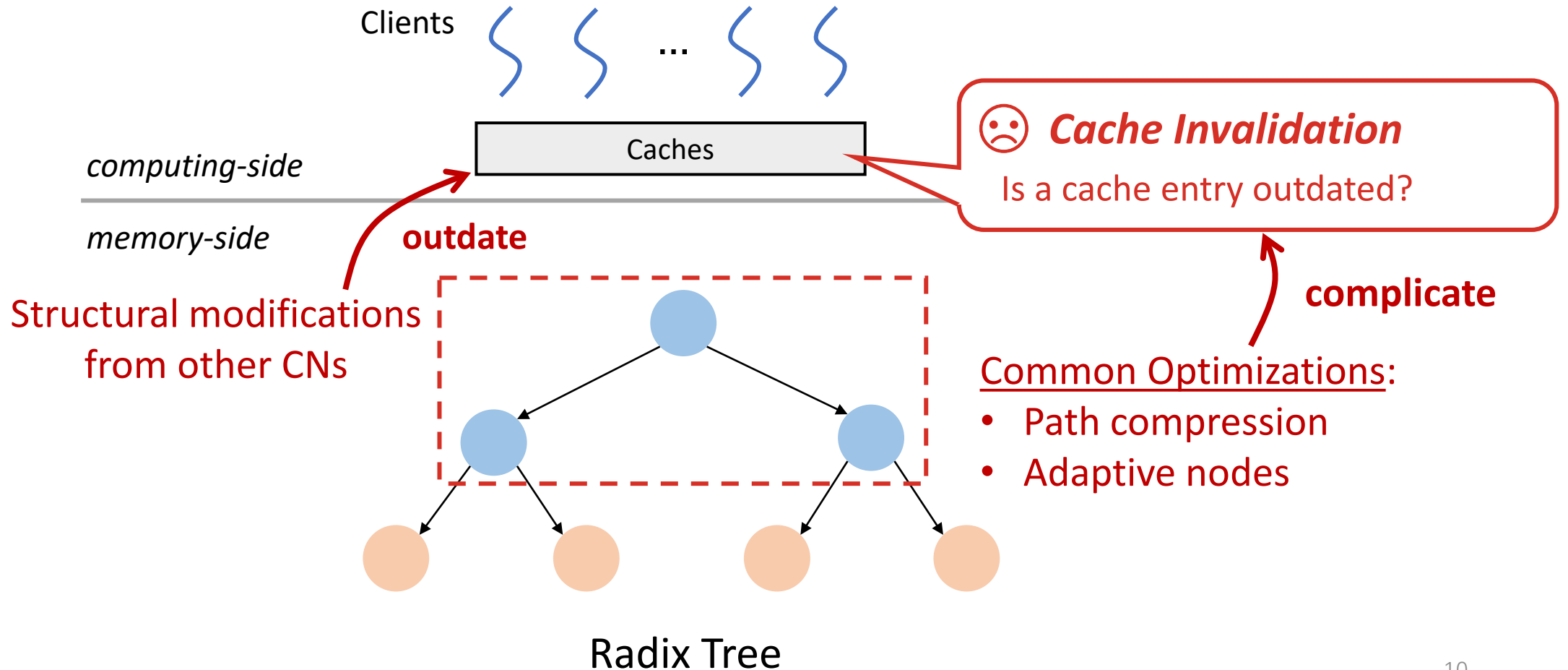
# Challenge 2: Bounded Memory-side IOPS

## Inter-client redundant I/Os on DM waste the limited IOPS of RNICs

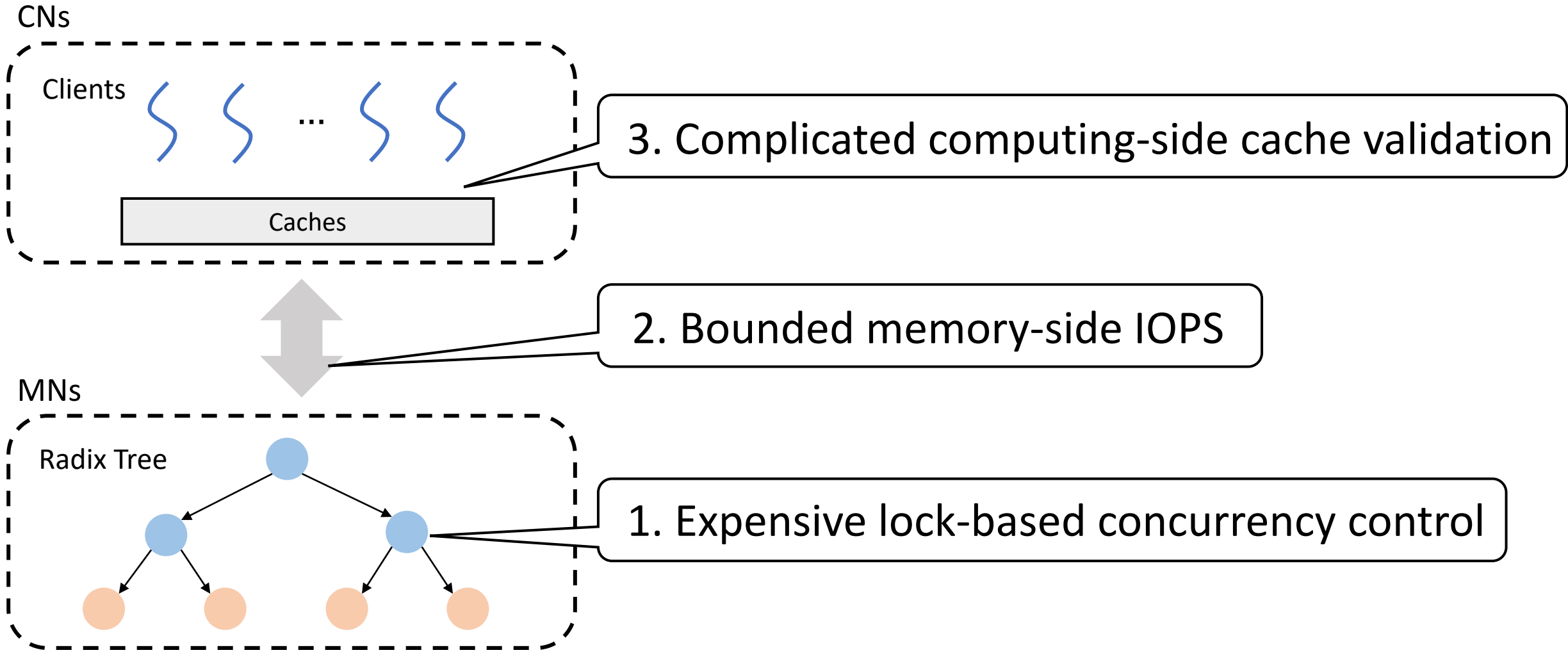


# Challenge 3: Complicated Computing-side Cache Validation

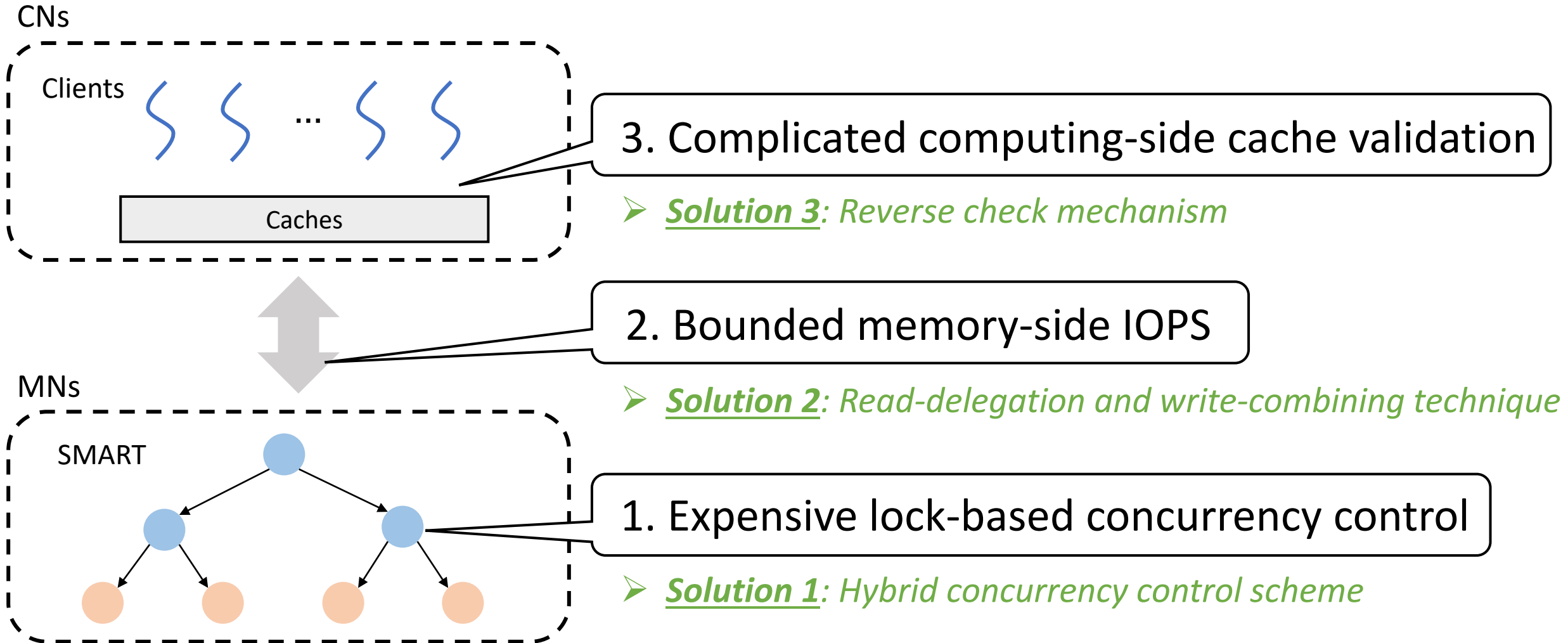
Structural features of radix trees complicate the problem of cache invalidation



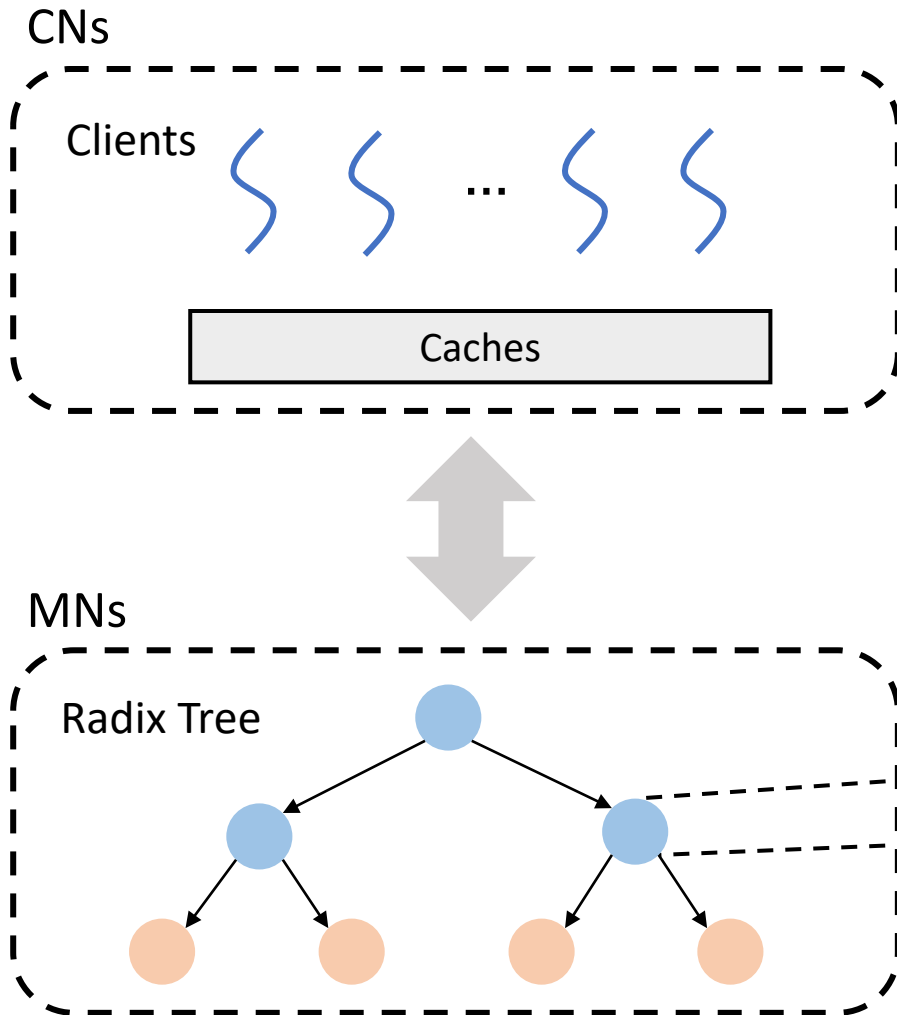
# Challenge Summary



# DiSaggregated-meMory-friendly Adaptive Radix Tree (SMART)



# Hybrid Concurrency Control



**Problem:** Expensive lock-based concurrency control

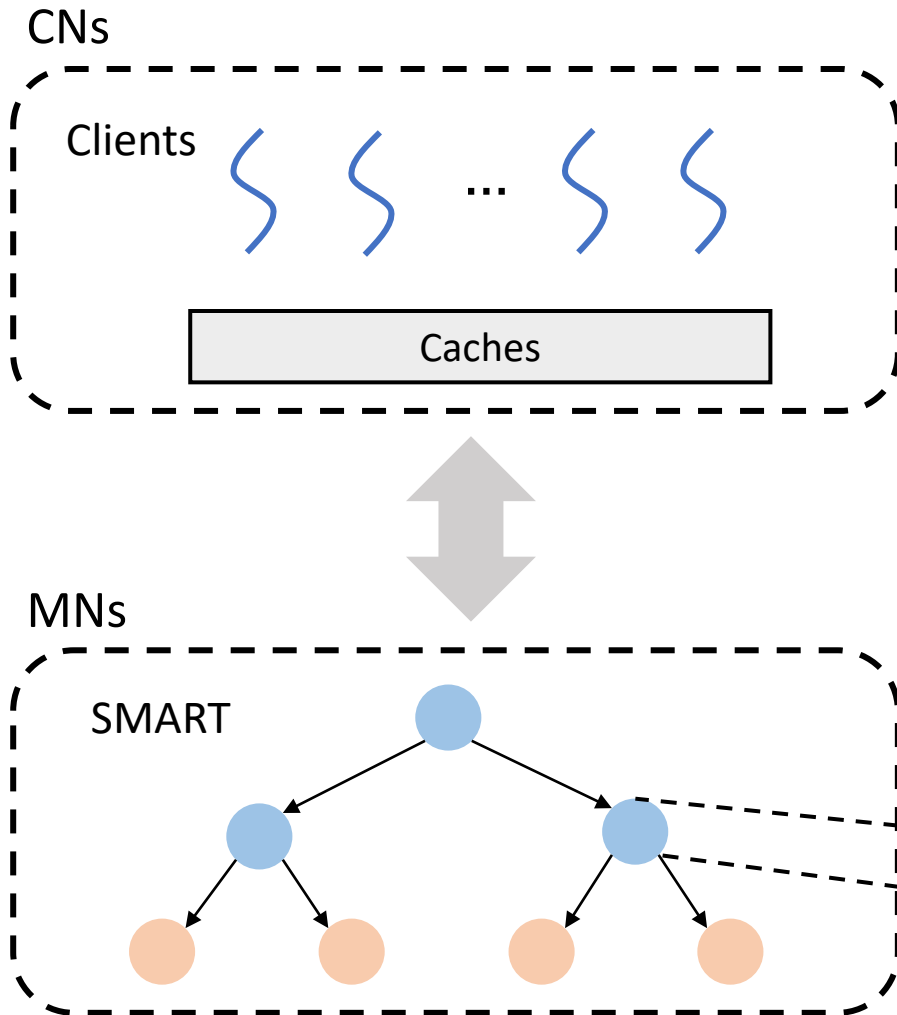
The internal node of the state-of-the-art radix tree <sup>[3]</sup>:



Store partial keys and child pointers separately

[3] Viktor Leis et al. "The adaptive radix tree: ARTful indexing for main-memory databases." ICDE 2013.

# Hybrid Concurrency Control

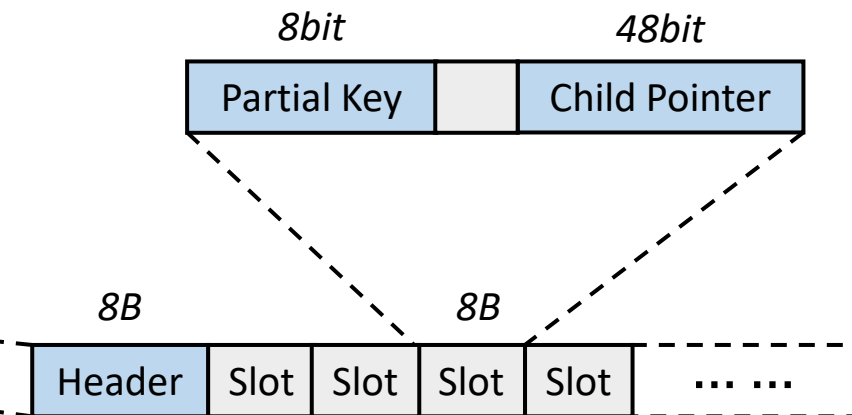


**Problem:** Expensive lock-based concurrency control

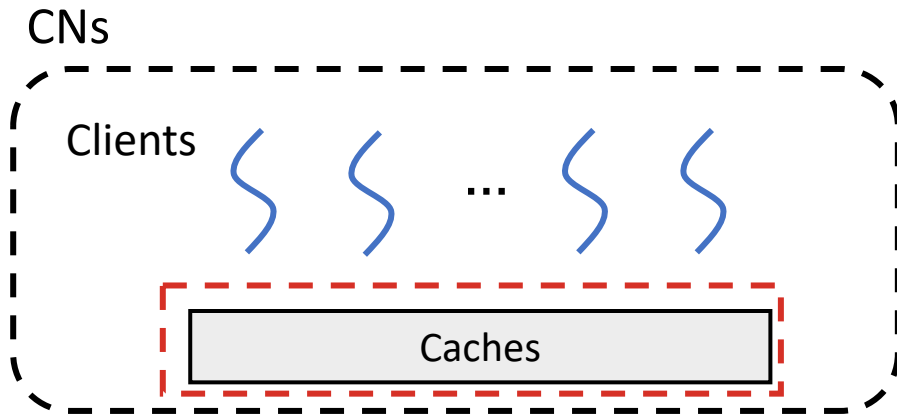
➤ Solution: Lock-free internal nodes

Key Idea:

- Embed each partial key and child pointer into an 8-byte slot
- 8-byte header

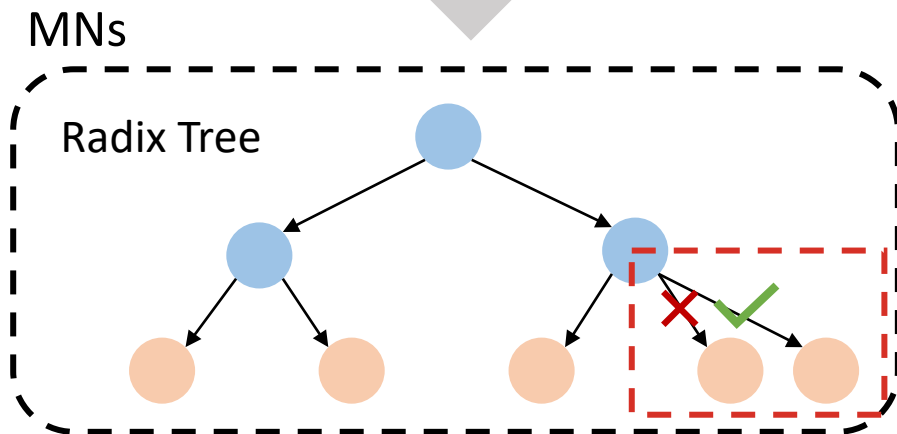


# Hybrid Concurrency Control



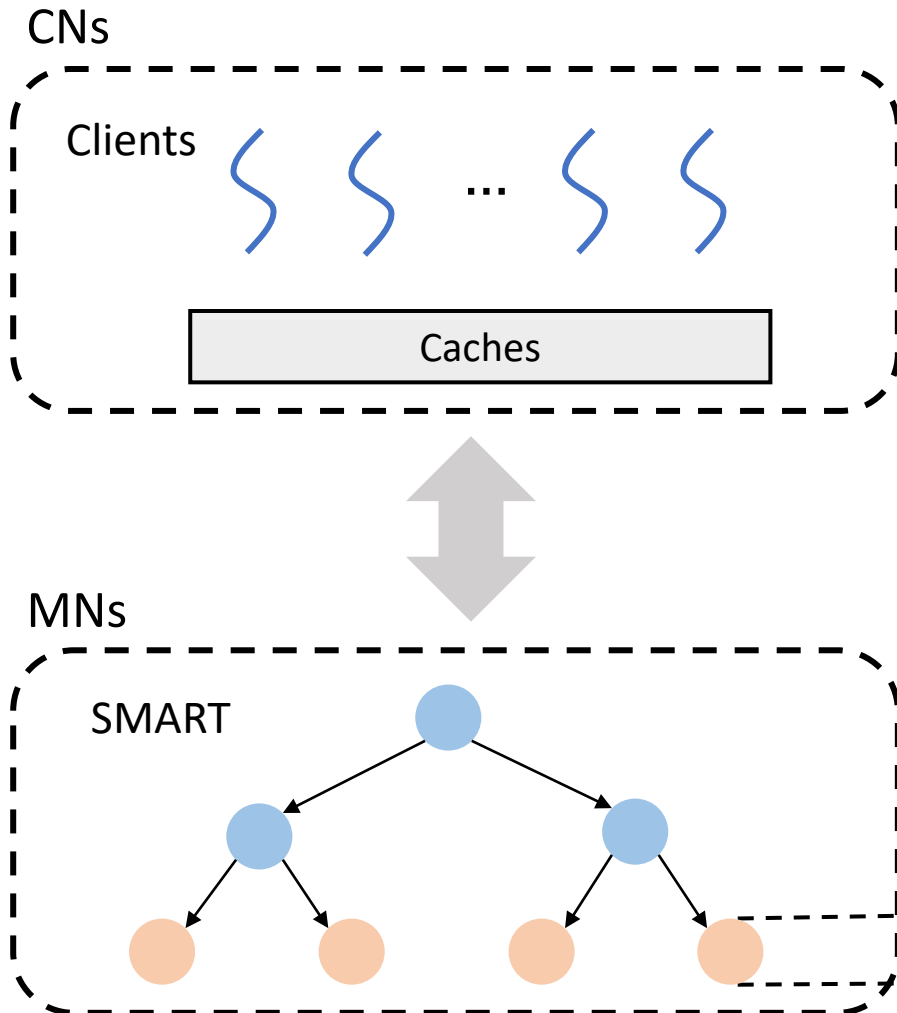
**Problem:** Out-of-place update causes cache thrashing

Cache leaf node addresses



Out-of-place update changes leaf node addresses frequently

# Hybrid Concurrency Control

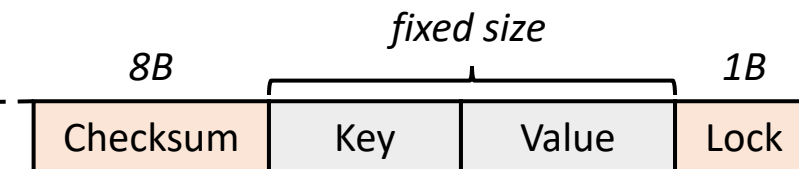


**Problem:** Out-of-place update causes cache thrashing

➤ Solution: Lock-based update-in-place leaf nodes

Key Idea:

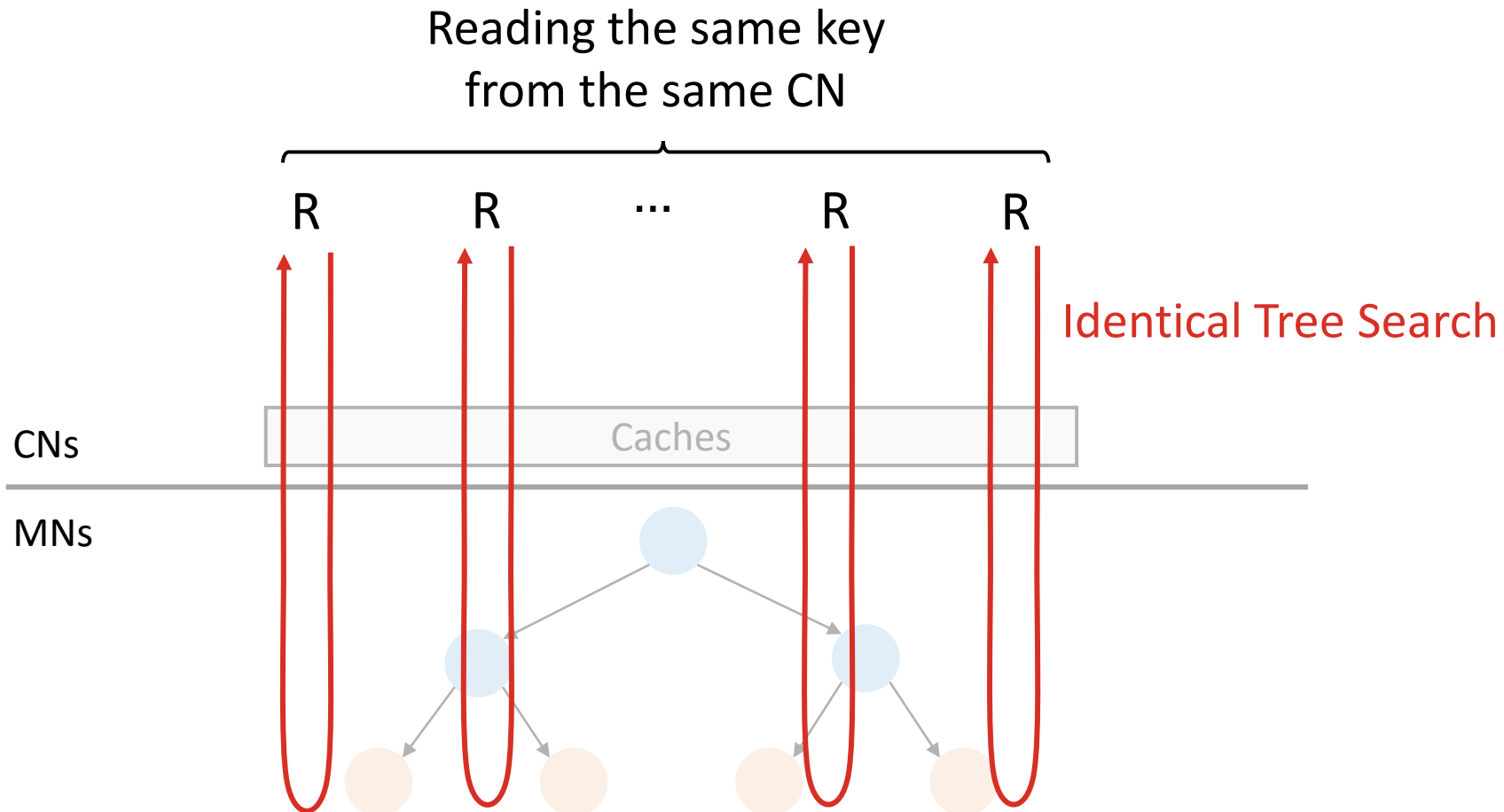
- Write-write conflict ➤ *Rear embedded lock*
  - Combine lock release with writing back
- Read-write conflict ➤ *Checksum-based method*
  - Writer:  $\text{checksum} := \text{CRC}(\text{KV})$
  - Reader:  $\text{checksum} == \text{CRC}(\text{KV})?$





# Read Delegation and Write Combining

**Problem:** Redundant read I/Os



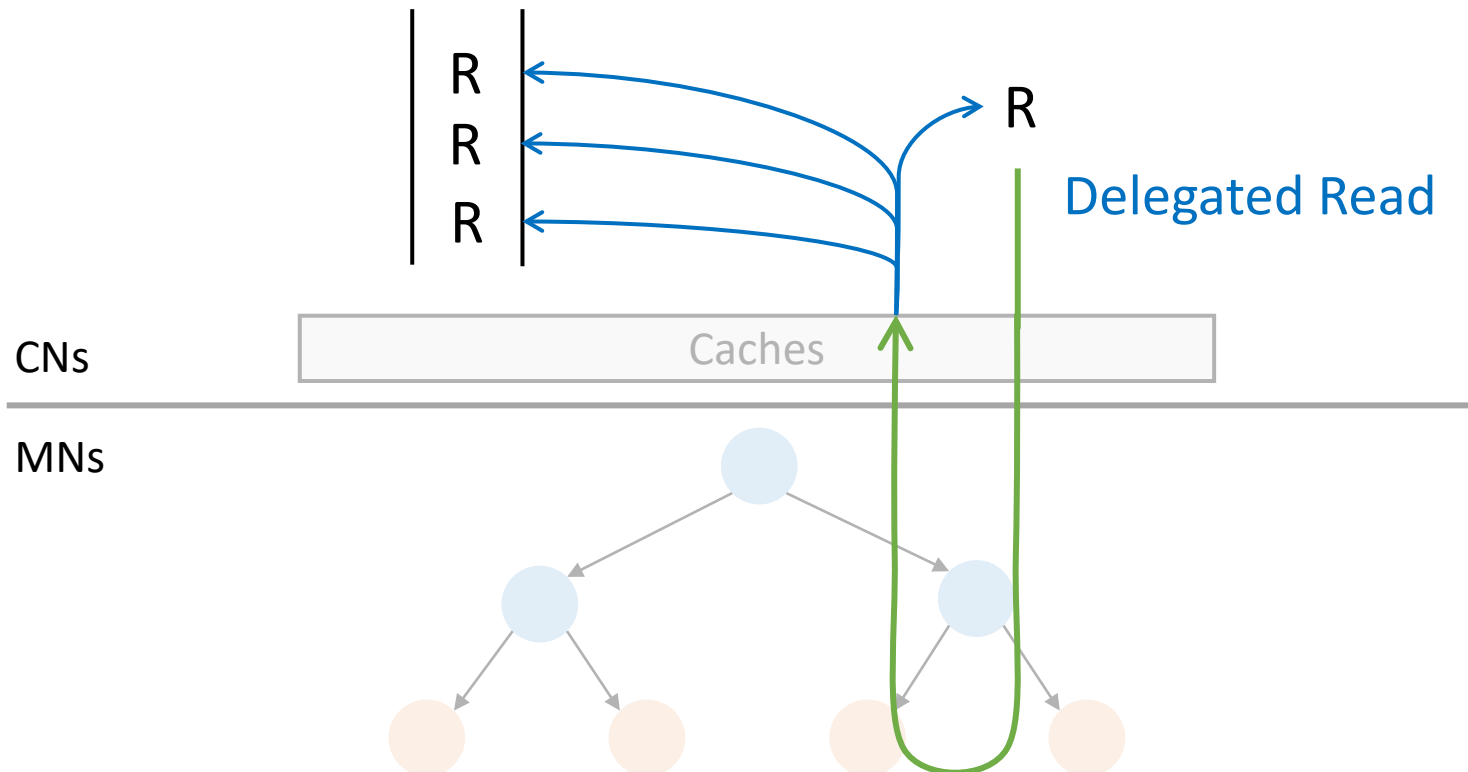
# Read Delegation and Write Combining

**Problem:** Redundant read I/Os

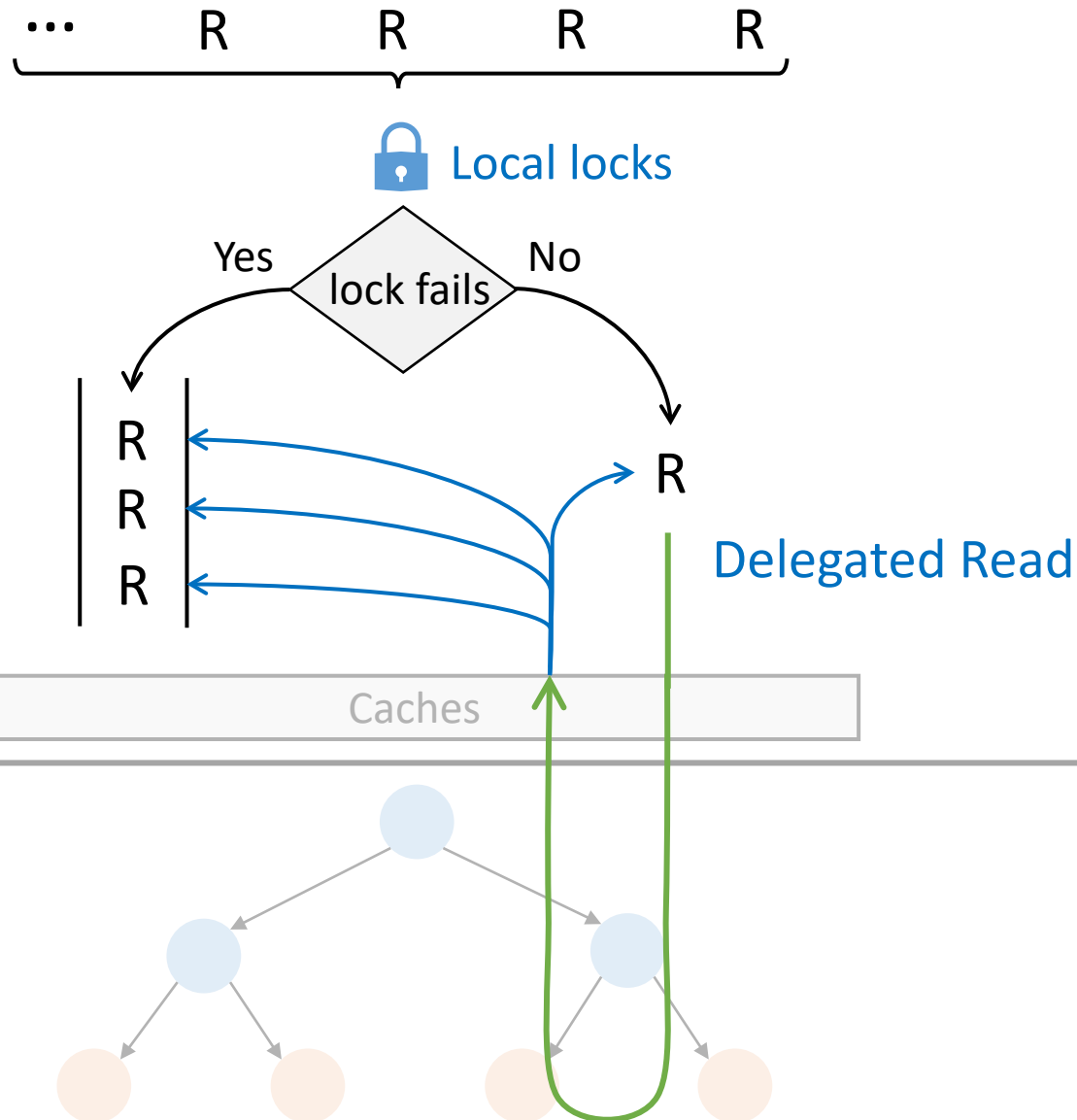
➤ Solution: Read delegation

Key Idea:

- Choose a delegation client on each CN to execute the same read



# Read Delegation and Write Combining



**Problem:** Redundant read I/Os

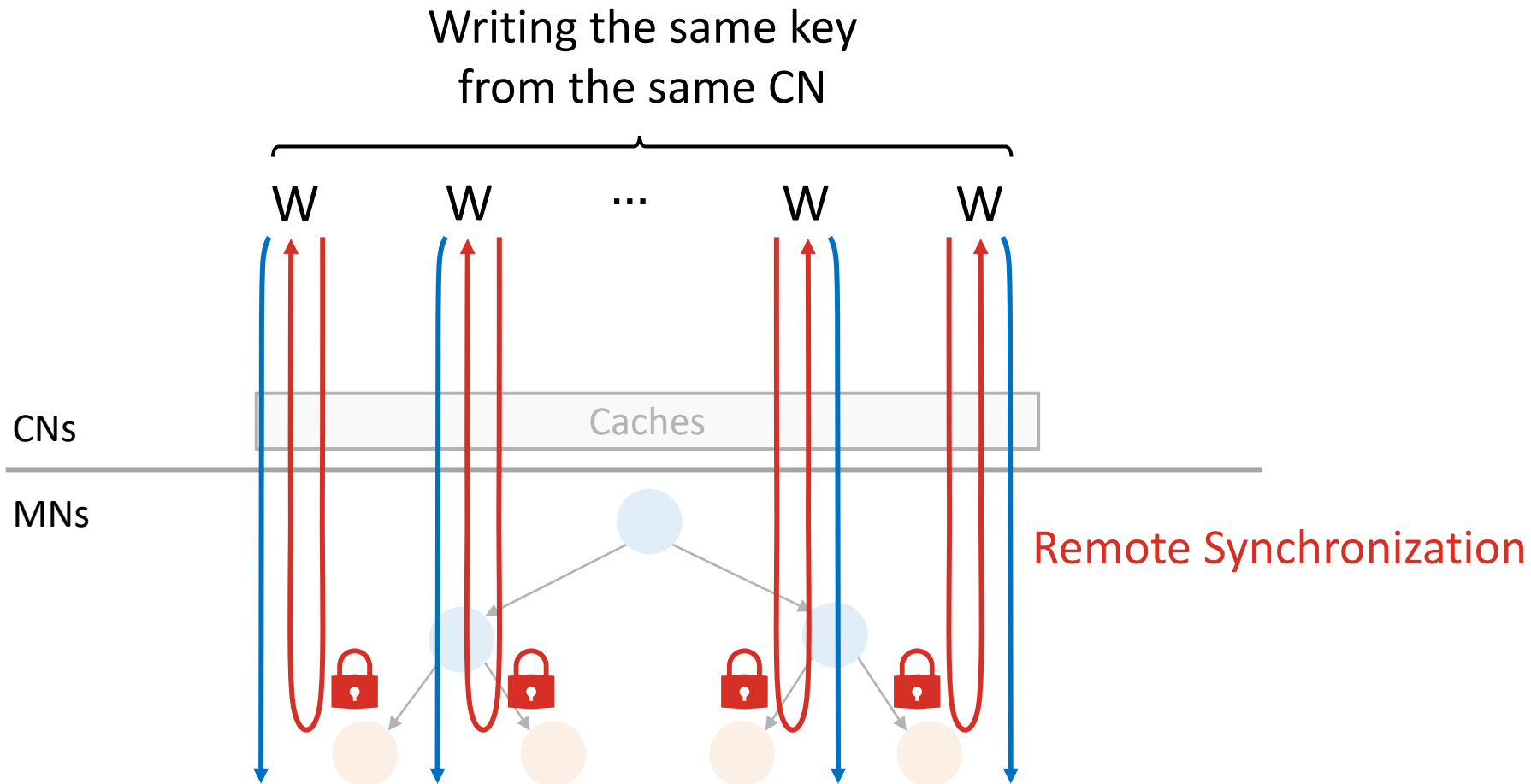
➤ **Solution:** Read delegation

**Key Idea:**

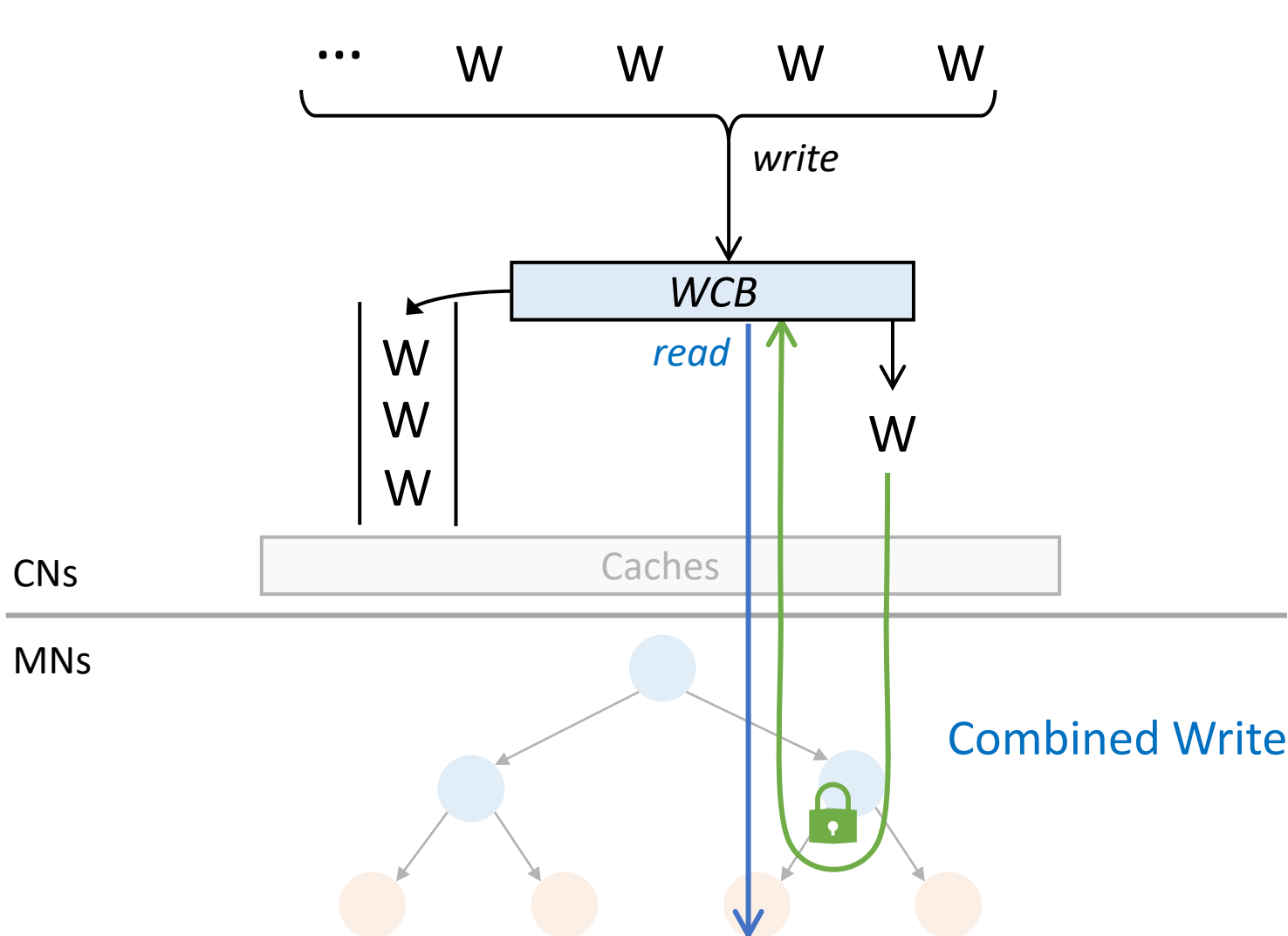
- Choose a delegation client on each CN to execute the same read
- Use local locks to collect concurrent identical reads

# Read Delegation and Write Combining

**Problem:** Redundant write I/Os



# Read Delegation and Write Combining



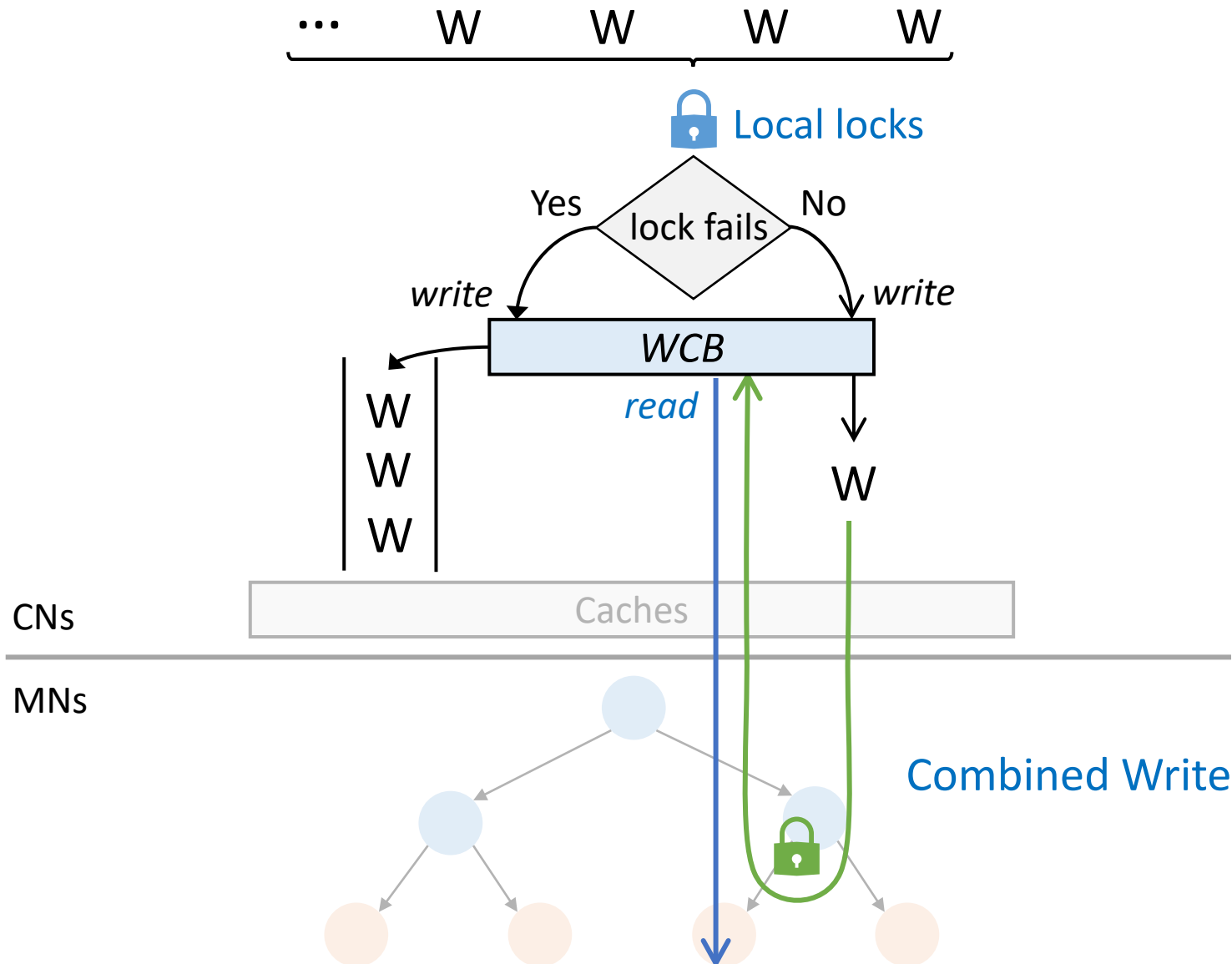
**Problem:** Redundant write I/Os

➤ **Solution:** Write combining

**Key Idea:**

- Combine these writes on a local write combining buffer (WCB)

# Read Delegation and Write Combining



**Problem:** Redundant write I/Os

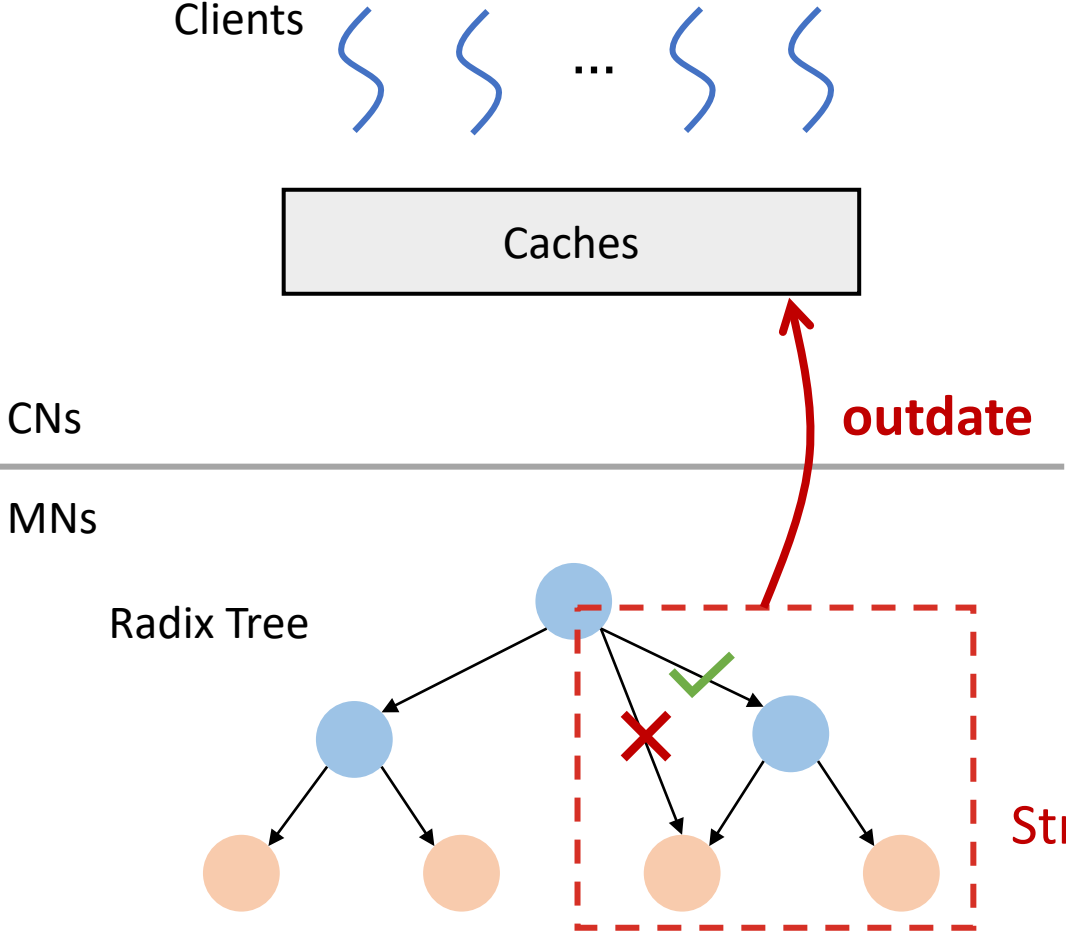
➤ **Solution:** Write combining

**Key Idea:**

- Combine these writes on a local write combining buffer (WCB)
- Use local locks to collect concurrent writes with the same target key

# Reverse Check

**Problem:** Cache invalidation of the radix tree



# Reverse Check

Clients 

Cache entry:



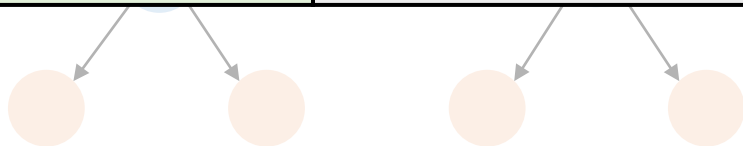
CNs

---

MNs

Read remote node

Reverse check



**Problem:** Cache invalidation of the radix tree

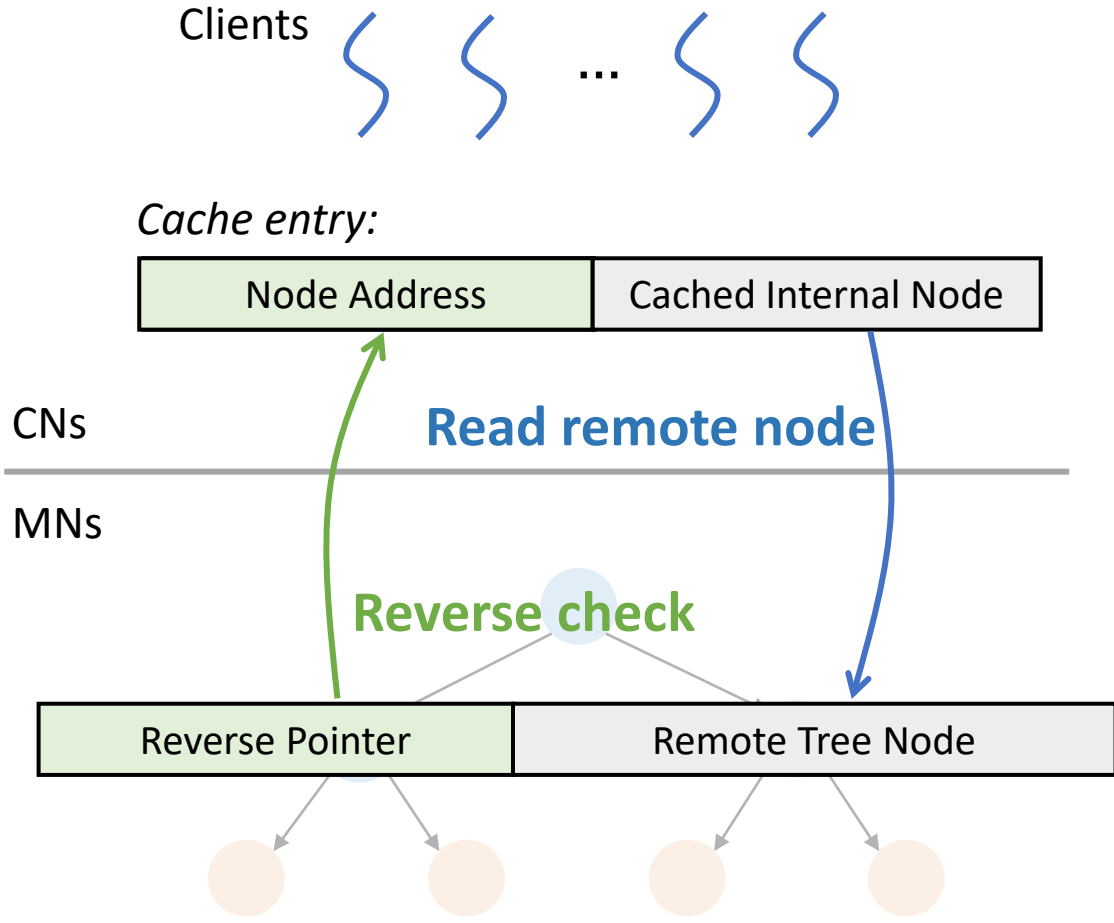
➤ Solution: Reverse check mechanism

Key Idea:

- Store check information in each remote node
- Check: check information == cache content ?



# Reverse Check



**Problem:** Cache invalidation of the radix tree

➤ Solution: Reverse check mechanism

Key Idea:

- Store check information in each remote node
- Check: check information == cache content ?

Example:

- Cache invalidation: *adjustments on the parent-child relationship* of remote nodes
- Store a reverse pointer in the front of each node
- Check: *Reverse Pointer == cached Node Address ?*

# More Details

- Concurrent operations
- Hash-based local locks
- Complete reverse check designs
- Support for variable-sized keys and values
- .....



**SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory**  
Xuchuan Luo<sup>1,\*</sup>, Pengfei Zuo<sup>2</sup>, Jiacheng Shen<sup>3,\*</sup>, Jiazhen Gu<sup>3</sup>,  
Xin Wang<sup>1,4</sup>, Michael R. Lyu<sup>3</sup>, and Yangfan Zhou<sup>1,4</sup>  
<sup>1</sup>School of Computer Science, Fudan University  
<sup>2</sup>Huawei Cloud <sup>3</sup>Tencent  
<sup>4</sup>Shanghai Key Laboratory of Intelligent Information Processing

**Abstract**  
Disaggregated memory (DM) is an increasingly prevalent architecture in academia and industry with high resource utilization. It separates computing and memory resources into two pools and interconnects them with fast networks. Existing range indexes on DM are based on B+ trees, which suffer from large inherent read and write amplifications. The narrow width, resulting in low request throughput and high access latency of B+ trees on DM.  
In this paper, we propose to use the radix tree, which is more suitable for DM than the B+ tree due to smaller read/write amplifications. However, constructing a radix tree on DM is challenging due to the costly lock-based concurrency control, the bounded memory-side IOPS, and the complex computing-side cache validations. To address these challenges, we design SMART, the first radix tree for disaggregated memory with high performance. Specifically, we leverage *hybrid concurrency control* scheme including lock-free internal nodes and fine-grained lock-based leaf nodes to reduce lock overhead, 2) a computing-side *read delegation* and *combining* technique to break through the IOPS upper bound by reducing redundant IOs, and 3) a simple yet effective *reverse check* mechanism for computing-side cache validation. Experimental results show that SMART achieves 6.1x higher throughput under typical write-intensive workloads and 1.5x higher throughput under read-only workloads than the state-of-the-art B+ trees on DM.

**1 Introduction**  
Distributed range indexes are fundamental building blocks of many applications, e.g., databases and key-value stores. To conduct range queries [2, 21, 53, 57, 59]. To improve resource utilization, many new proposals adopt the disaggregated memory (DM) architecture [3, 59]. DM can decouple computing and memory resources into two elastic resource pools.

\*The work was mainly conducted when Xuchuan and Jiacheng were at Huawei.

Figure 7: A step-by-step example of inserting several new keys into SMART with 8-bit partial keys. For clarity, hexadecimal partial keys are shown and reverse pointers are omitted. Each thick dotted box indicates an atomic CAS.

place update scheme overwrites the leaf node at the same address, causing conflicts among readers and writers. To avoid conflicts, we adopt an optimistic lock in each leaf node with a checksum-based consistency check mechanism [40, 53], protected by a checksum. For write-write conflicts, an exclusive lock is used to synchronize the writers. As for read-write conflicts, when a writer modifies the leaf node, the checksum is recalculated based on the new content of the leaf node and after reading the leaf node. The readers verify the checksum after reading the leaf node. If the checksum verification fails, the reader conducts a re-read.

(2) *Rear embedded lock*. To further reduce the overhead of locks, we combine the lock release with the writing back of the updated leaf node by embedding the lock into each leaf node. Therefore, the two operations can be done via one single RDMA\_WRITE. Particularly, to avoid premature lock release, we ensure that the lock release is always triggered after the completion of writing back. We achieve this by placing the lock at the rear of a leaf node, which leverages the in-order delivery property of RNICs [12].

As shown in Figure 6b, a leaf node of SMART consists of an 8-byte reverse pointer, a *Valid bit*, an 8-byte checksum, a 1-byte rear lock and a fixed-sized key-value item. The reverse pointer is used for each validation, which will be illustrated in § 4.3. The *Valid bit* is used to indicate the deleted state.

**4.1.2 Concurrent Operations**  
Based on the above structural modifications, we demonstrate simple, as shown in Figure 7. Except for the in-place leaf operation process will be described in § 4.4.

*Normal insert*. During an insert, the target partial key may not be in the internal node yet. As shown in Figure 7b, after the WRITE of the new leaf node ( $k_4$ ), the client CASes the first empty slot in the node, together with the new partial key (i.e., a new value of the slot written by the return partial key) contains the target partial key. If yes, the client continues to traverse the tree following the return pointer. Otherwise, the client tries the insert again with the next empty slot.

*Leaf split*. If an existing leaf node is found during an insert, a leaf split is needed as shown in Figure 7c. Specifically, the client first calculates the rest of the longest common key prefix of the two leaf nodes ( $k_4$  and  $k_5$ ). Then it allocates sufficient sequentially-connected internal nodes to store the sufficient key prefix in their leaders. The last internal node will contain two child pointers pointing to the old and new leaf nodes. All

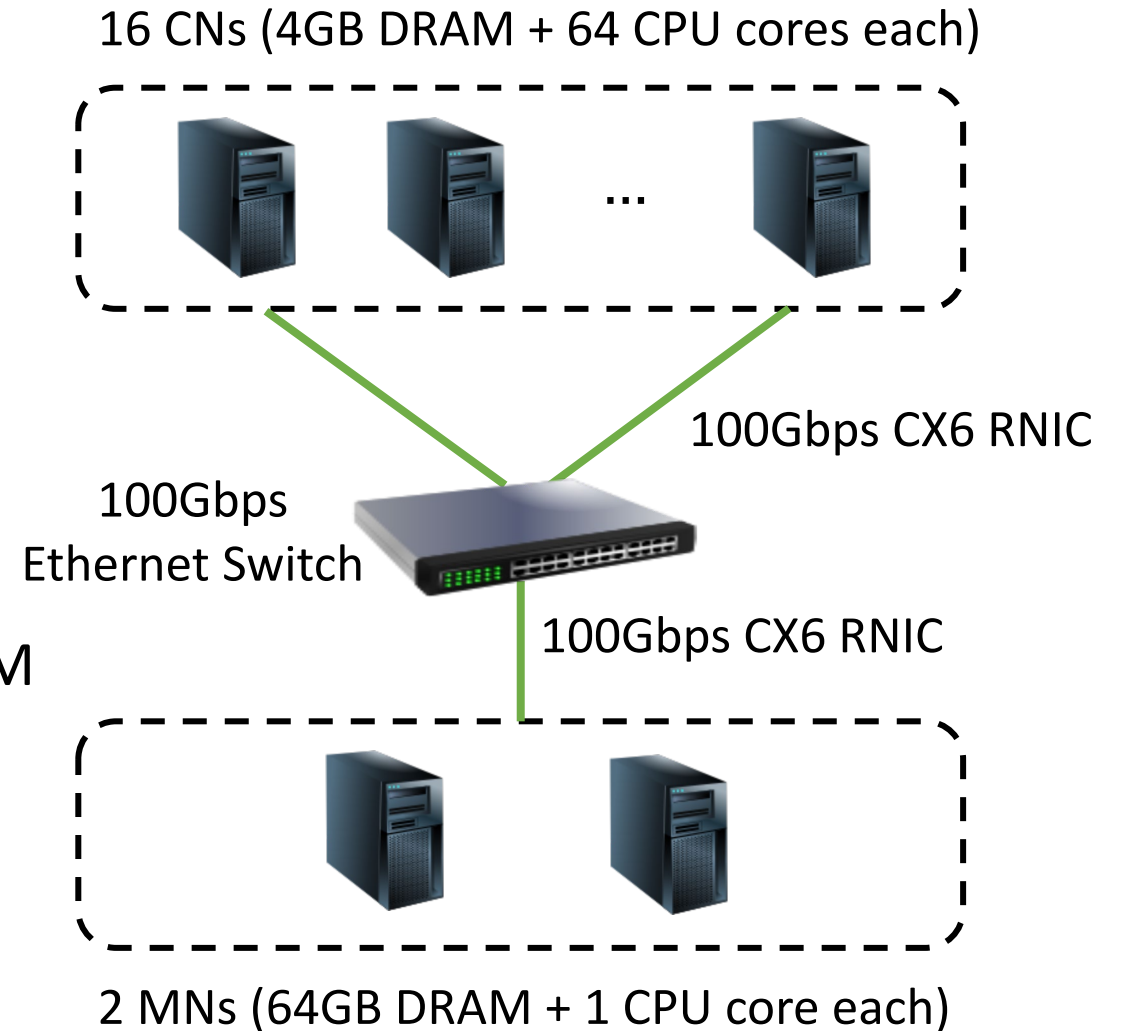
# Evaluation

## Workloads

- YCSB workloads
- 2 key types: integer, string

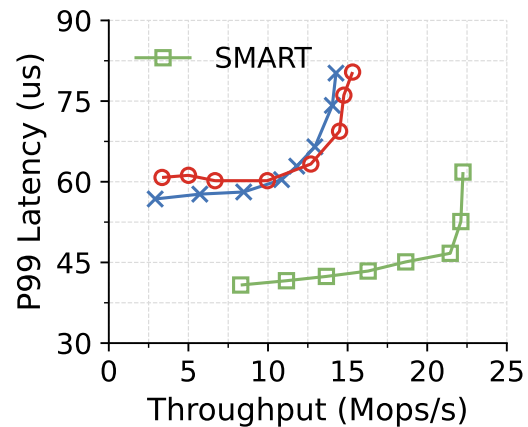
## Comparisons

- Sherman [SIGMOD'22]
  - The state-of-the-art B+ tree design on DM
- ART [ICDE'13]
  - The state-of-the-art radix tree design
  - We port it to DM

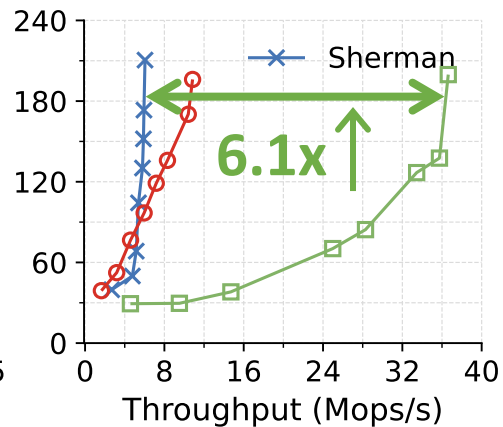


# Performance Comparison

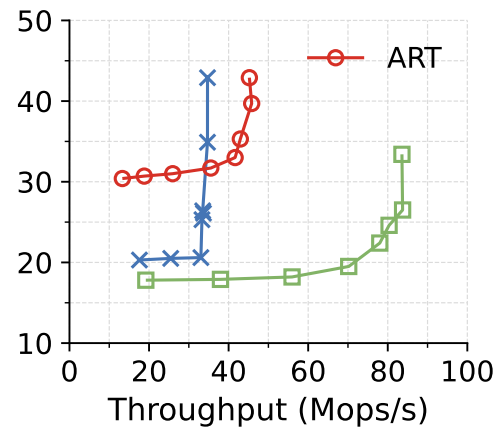
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



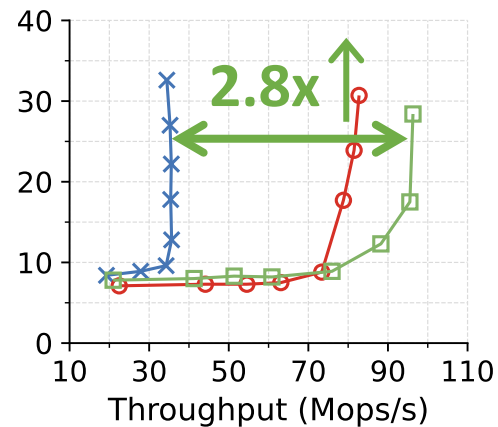
YCSB LOAD



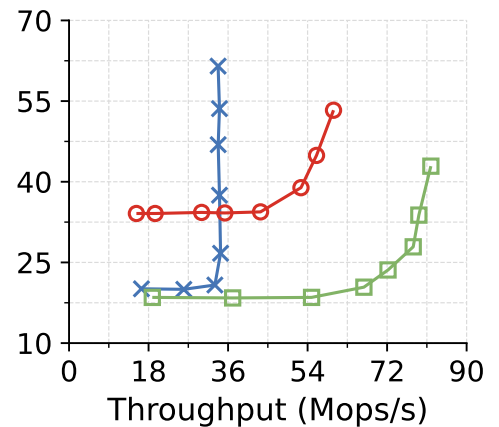
YCSB A



YCSB B



YCSB C

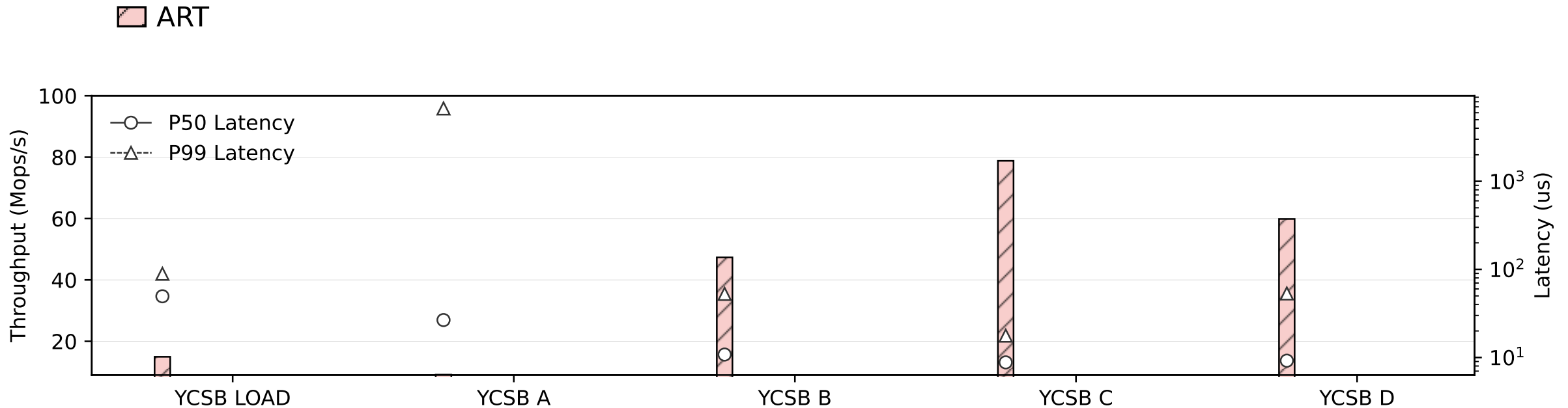


YCSB D

- Compared with Sherman, SMART achieves up to:
  - 6.1x higher throughput and 1.4x lower latency under write-intensive workloads
  - 2.8x higher throughput with similar latency under read-only workloads

# Factor Analysis for SMART design

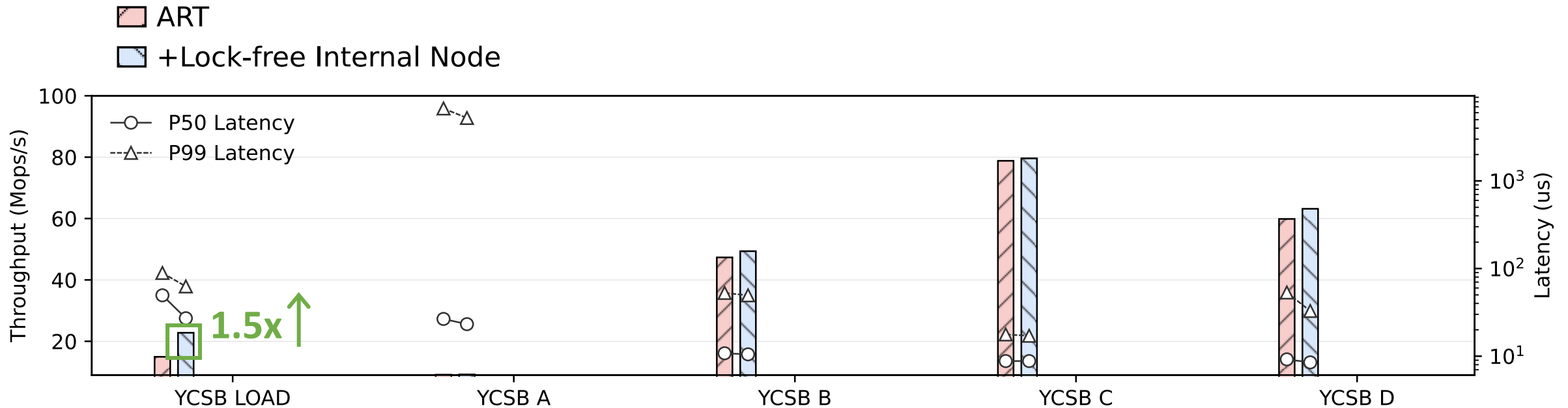
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



- Start with the ART design and apply each proposed technique one by one

# Factor Analysis for SMART design

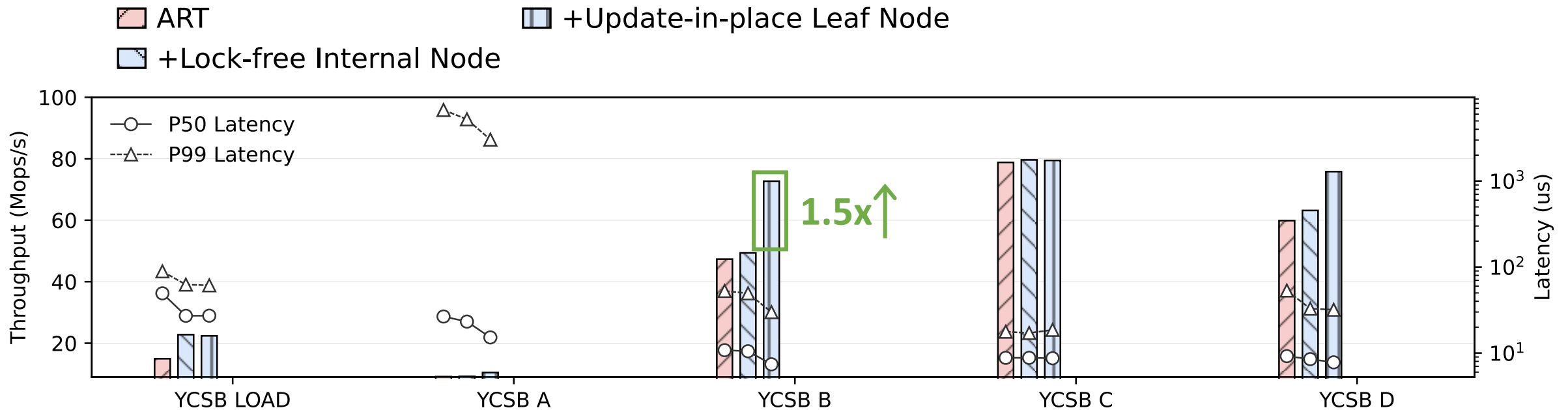
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



- The *lock-free internal node* brings **1.5x improvement** in throughput under the YCSB LOAD workload

# Factor Analysis for SMART design

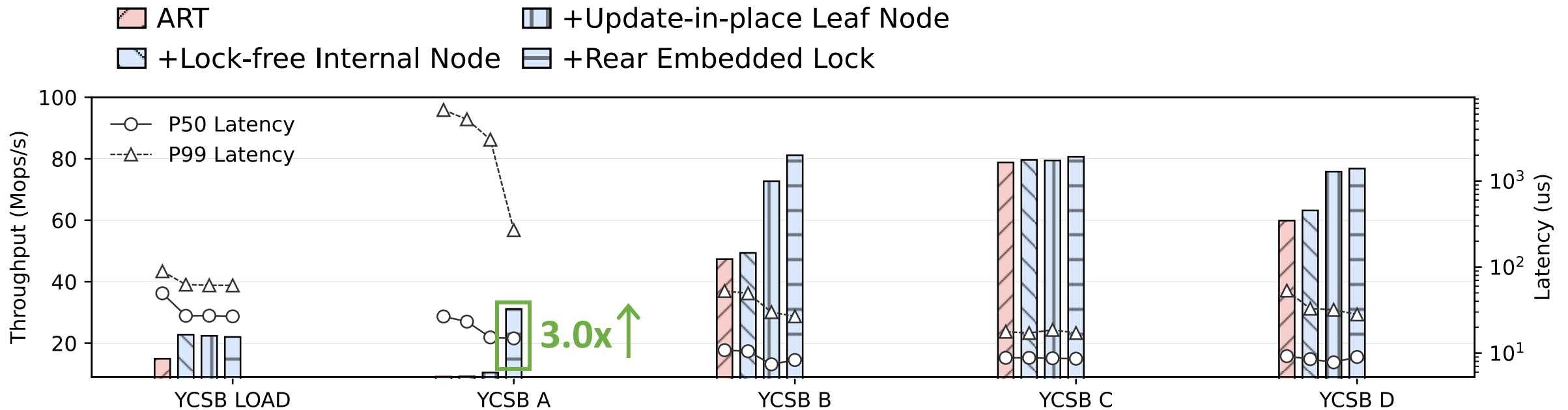
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



- The *update-in-place leaf node* brings **1.5x improvement** in throughput under the YCSB B workload

# Factor Analysis for SMART design

YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert

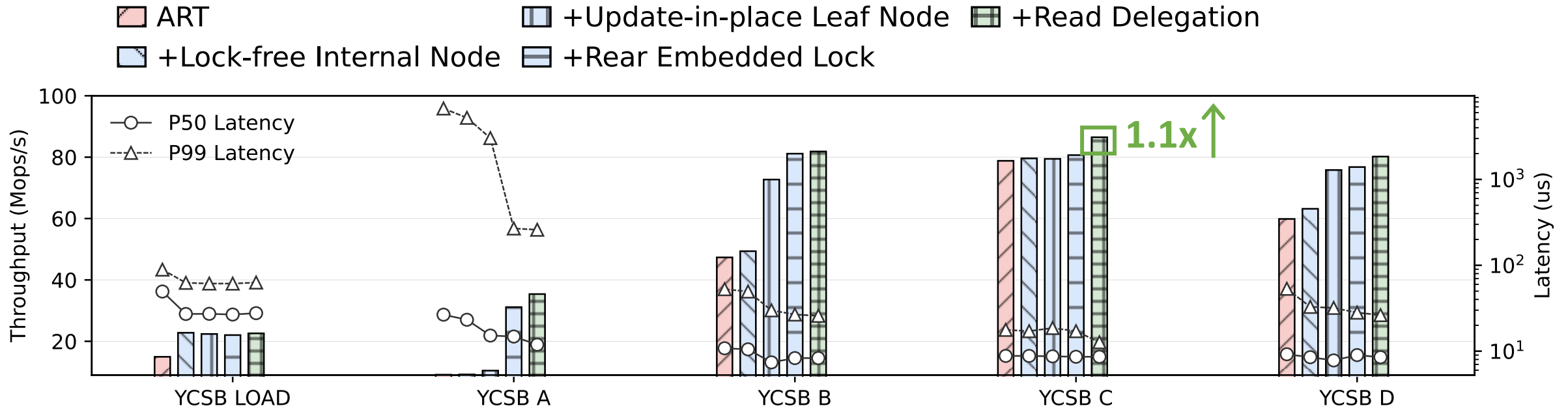


- The *rear embedded lock* brings **3.0x improvement** in throughput under the YCSB A workload



# Factor Analysis for SMART design

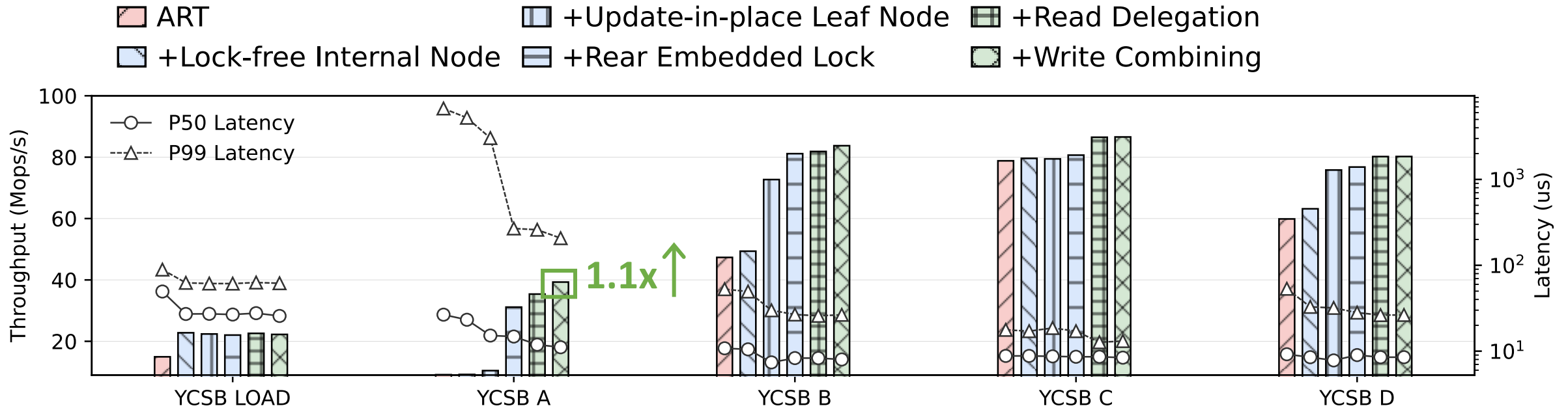
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



➤ The *read delegation* brings **1.1x improvement** in throughput under the YCSB C workload

# Factor Analysis for SMART design

YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



➤ The *write combining* brings **1.1x improvement** in throughput under the YCSB A workload

# Conclusion

- Existing tree indexes on DM are based on B+ trees, which **suffer from large inherent read and write amplifications**
- We propose **SMART**, a high-performance adaptive radix tree for DM
  - Hybrid concurrency control scheme
  - Read-delegation and write-combining technique
  - Reverse check mechanism
- SMART outperforms the state-of-the-art B+ tree on DM by up to **6.1x** under YCSB write-intensive workloads and **2.8x** under YCSB read-only workloads

# Thank you! Q&A



<https://github.com/dmemsys/SMART>



復旦大學  
FUDAN UNIVERSITY



HUAWEI



香港中文大學  
The Chinese University of Hong Kong