ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

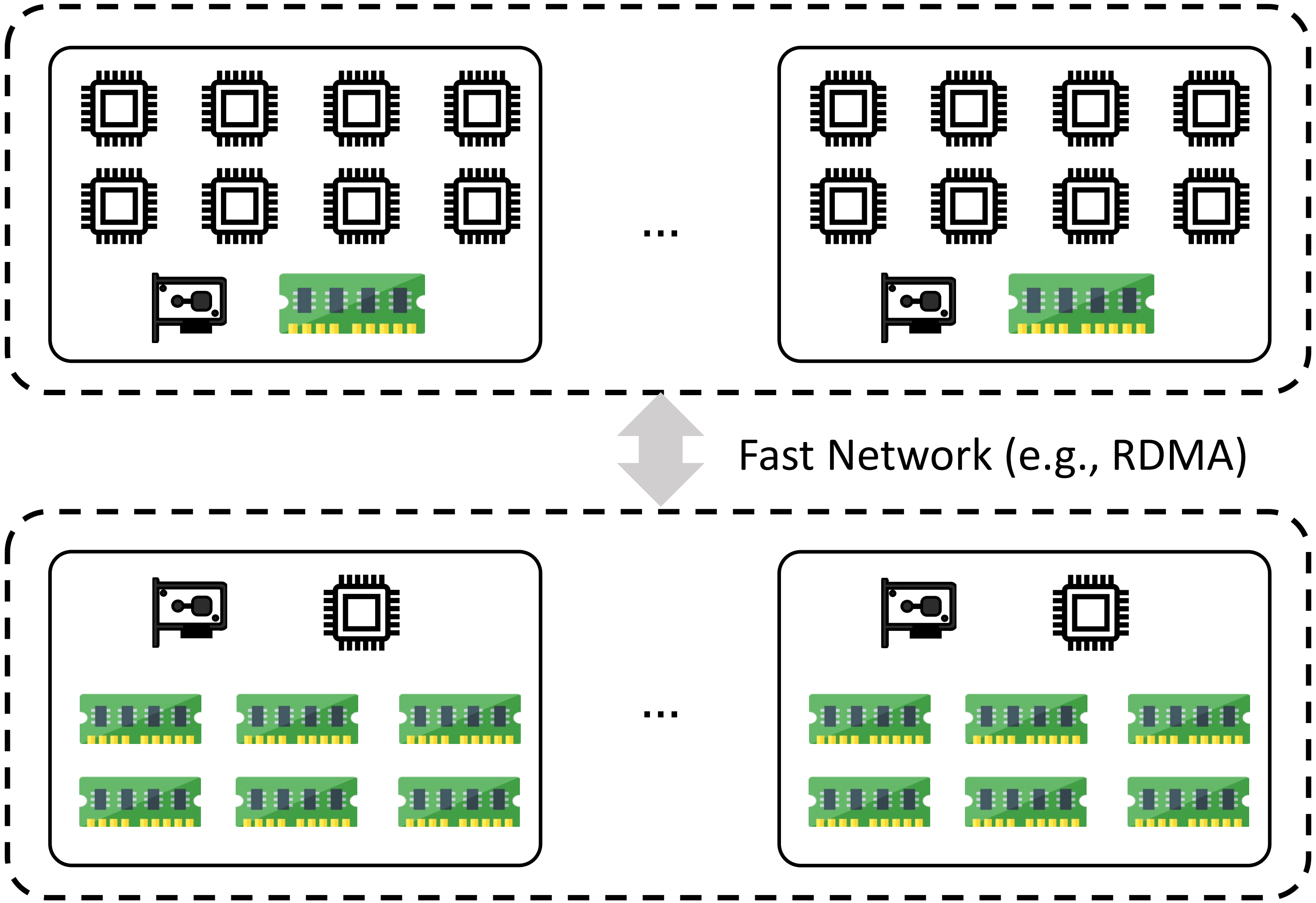
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

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

Xuchuan Luo[1], Pengfei Zuo[2], Jiacheng Shen[3], Jiazhen Gu[3],
Xin Wang[1,4], Michael R. Lyu[3], and Yangfan Zhou[1,4]

[1]*School of Computer Science, Fudan University*
[2]*Huawei Cloud* [3]*The Chinese University of Hong Kong*
[4]*Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China*

Disaggregated Memory (DM)



Compute Nodes (CNs)

…

Fast Network (e.g., RDMA)

…

Memory Nodes (MNs)

☺ **Benefits:**

✓ Resource utilization

✓ Elasticity

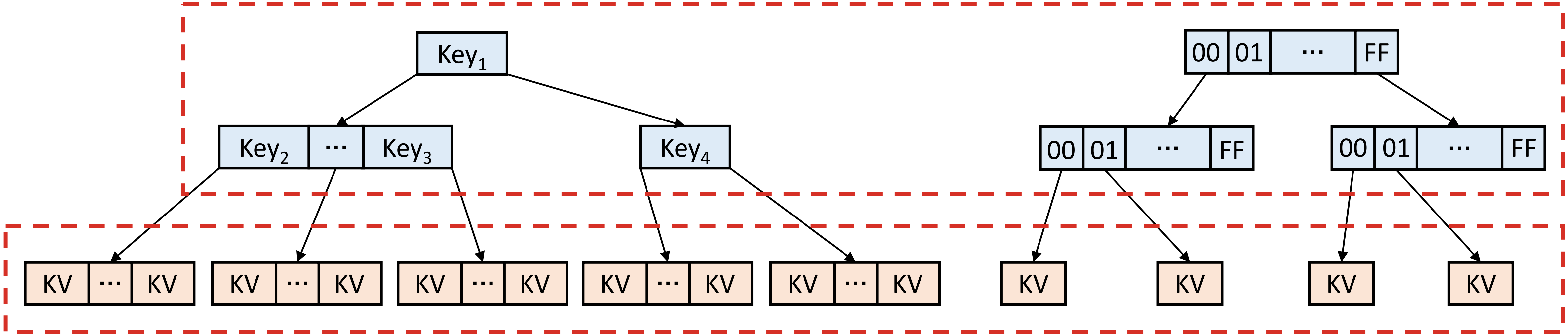
Tree Indexes

B+ Tree

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

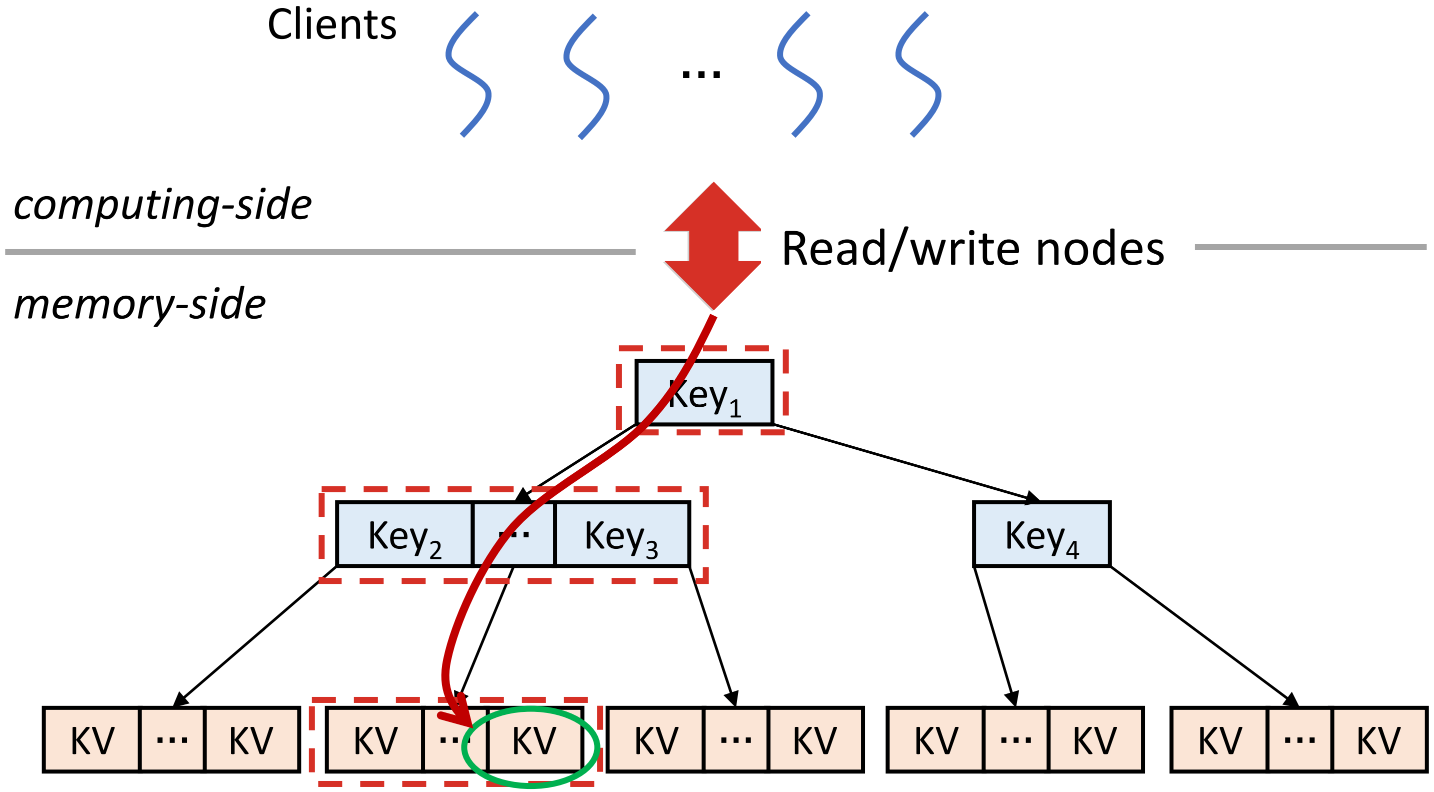
Radix Tree

- 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 [1], Sherman [2]



☹ **Problem:**

Read/write amplifications of B+ trees

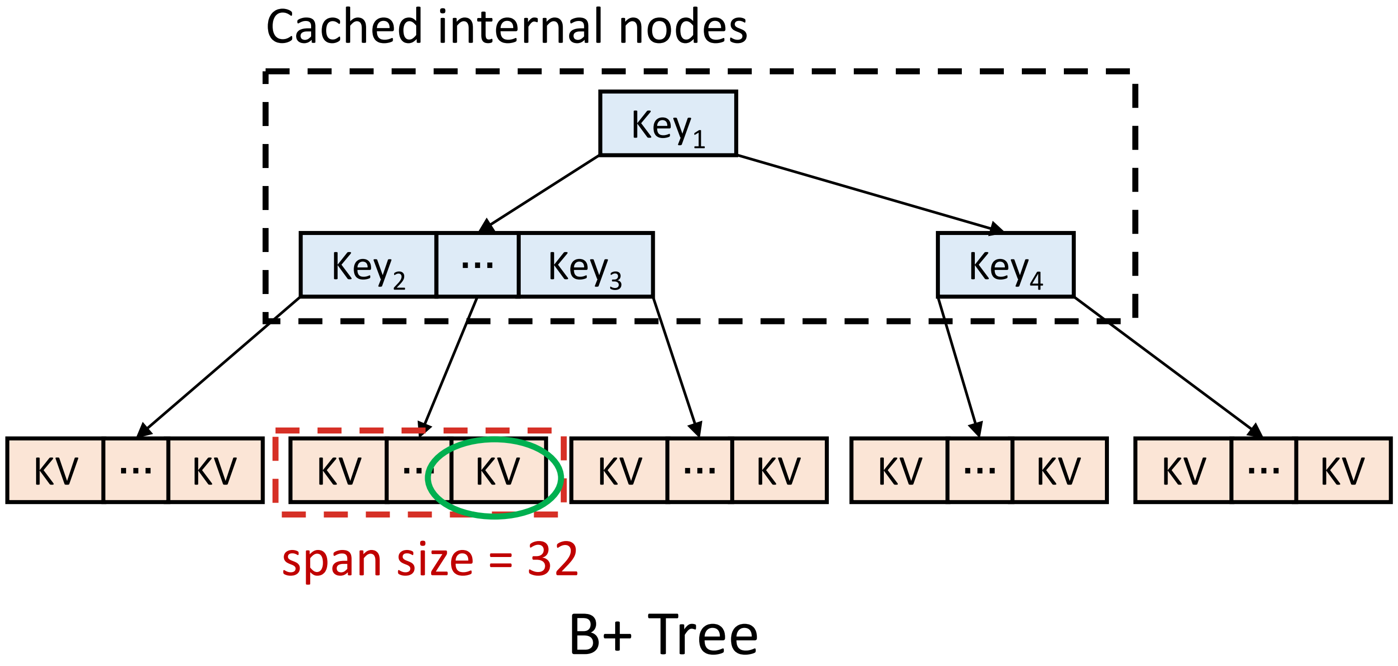
- ➢ Exacerbate the network bandwidth bottleneck of DM

[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	≈ 32	≈ 33
Write amplification	≈ 32	≈ 1

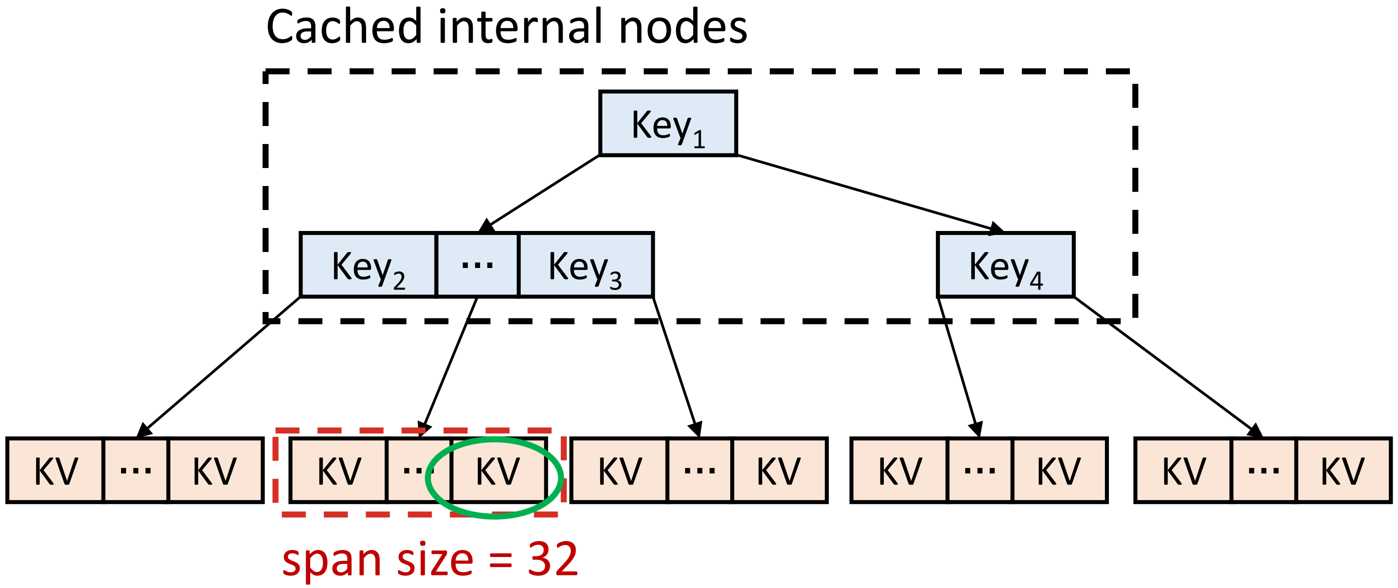


B+ Tree

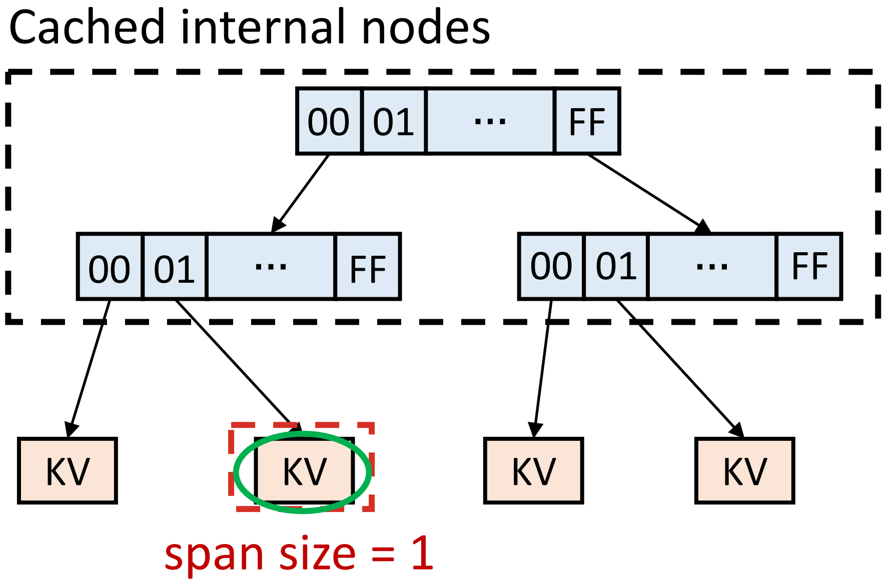
Tree Indexes on Disaggregated Memory

Read and write amplification factors:

	B+ Tree	Sherman	Radix Tree
Read amplification	≈ 32	≈ 33	≈ 1
Write amplification	≈ 32	≈ 1	≈ 1



B+ Tree



Radix Tree

Tree Indexes on Disaggregated Memory

Read and write amplification factors:

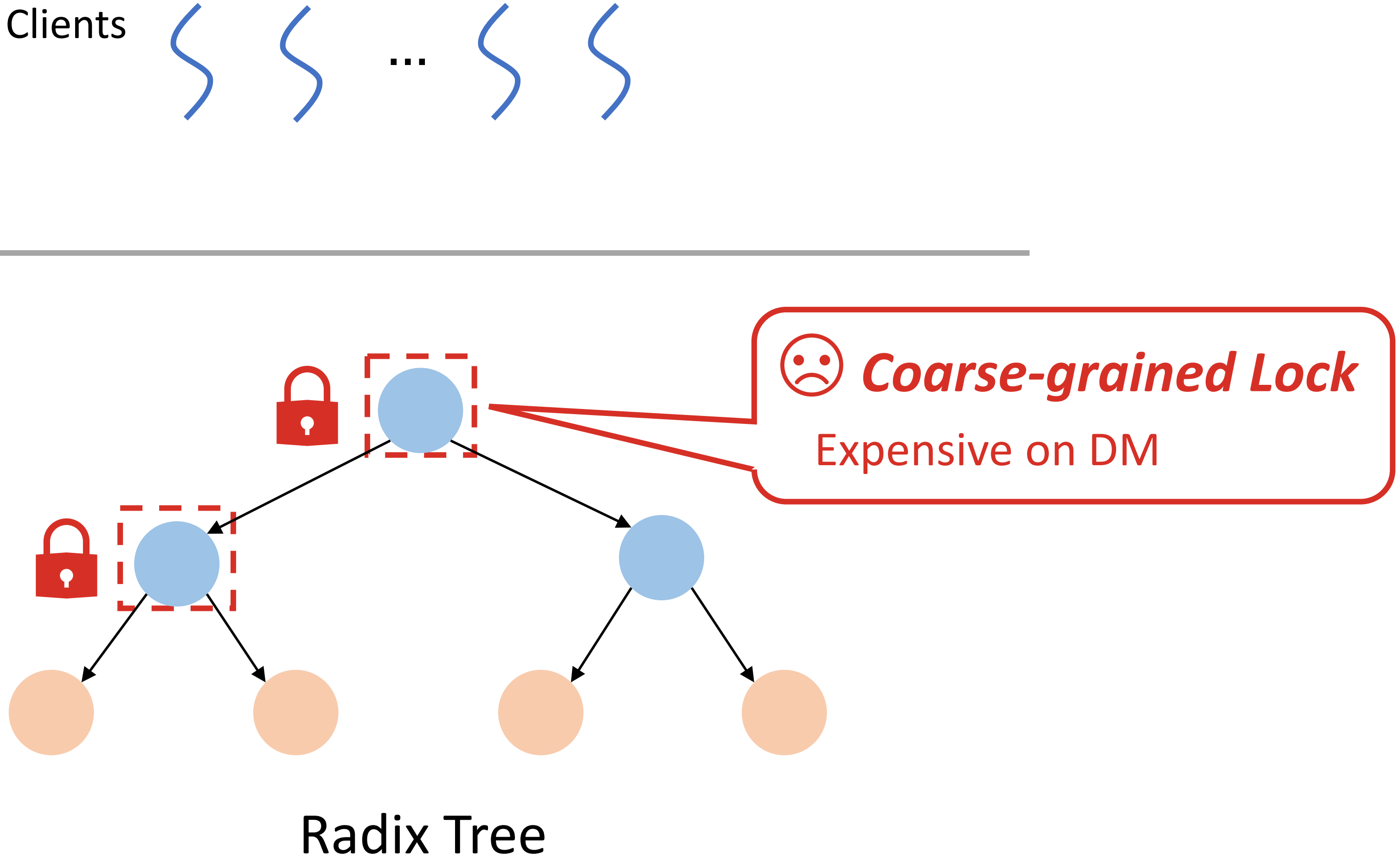
	B+ Tree	Sherman	Radix Tree
Read amplification	≈ 32	≈ 33	≈ 1
Write amplification	≈ 32	≈ 1	≈ 1

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

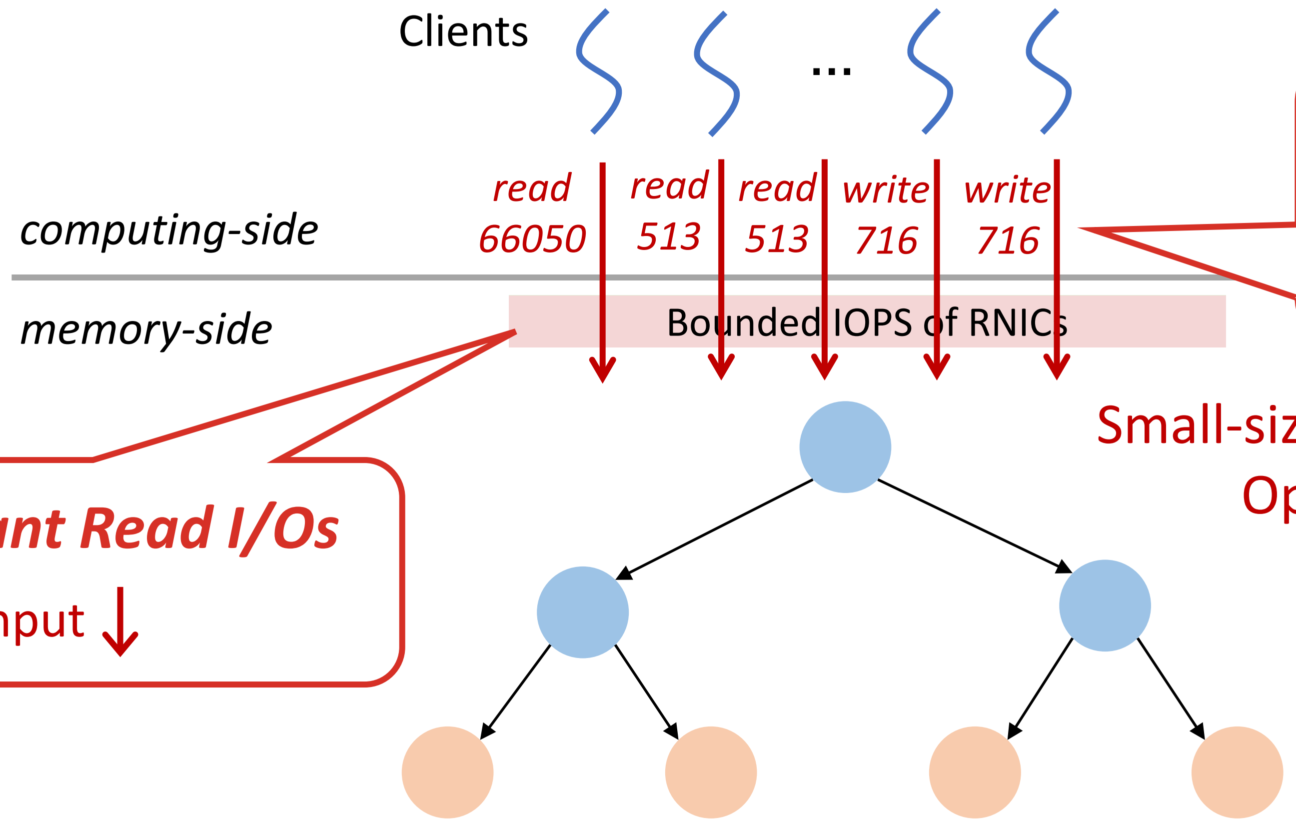


☹ ***Coarse-grained Lock***

Expensive on DM

Challenge 2: Bounded Memory-side IOPS

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



☹ ***Redundant Write I/Os***

Concurrency conflicts ↑

Peak throughput ↓↓

Small-sized Read/Write Operations

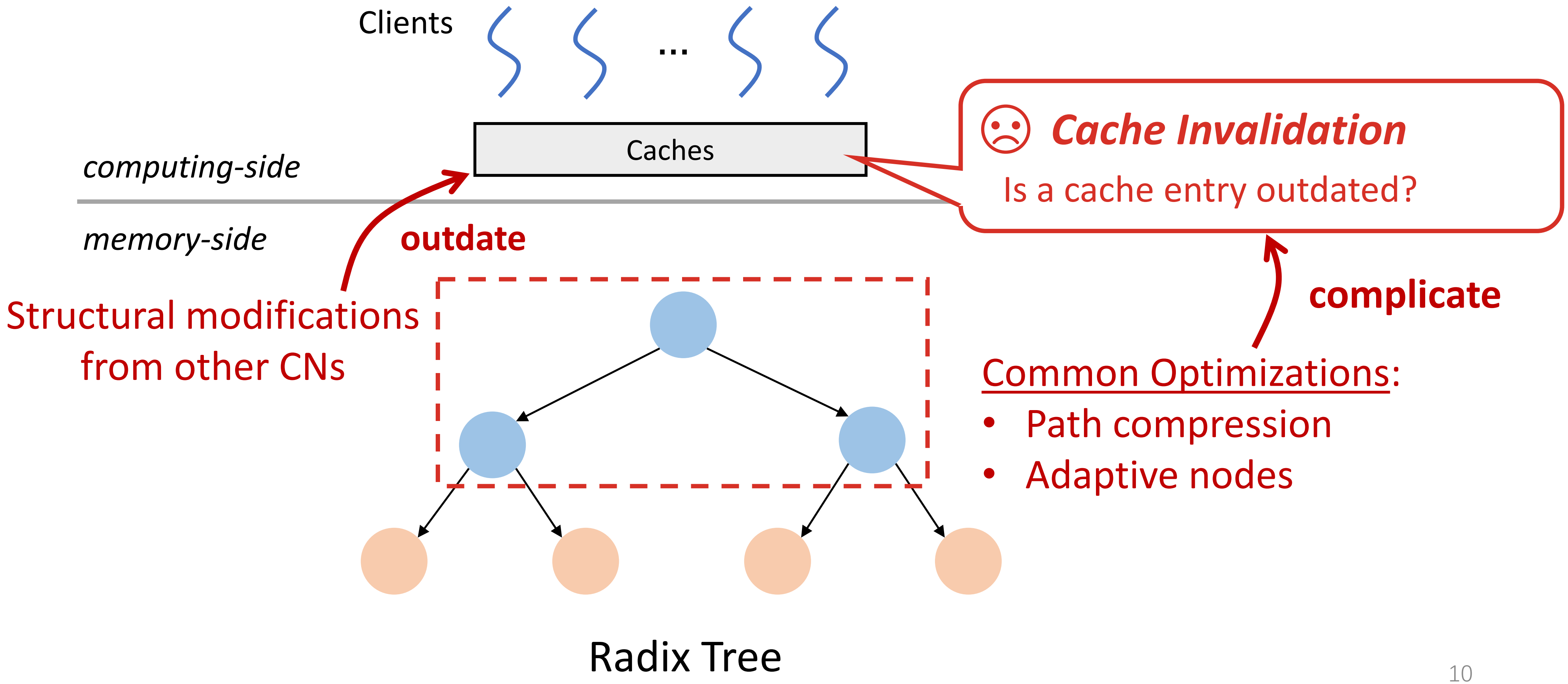
☹ ***Redundant Read I/Os***

Peak throughput ↓

Radix Tree

Challenge 3: Complicated Computing-side Cache Validation

Structural features of radix trees complicate the problem of cache invalidation



☹ ***Cache Invalidation***

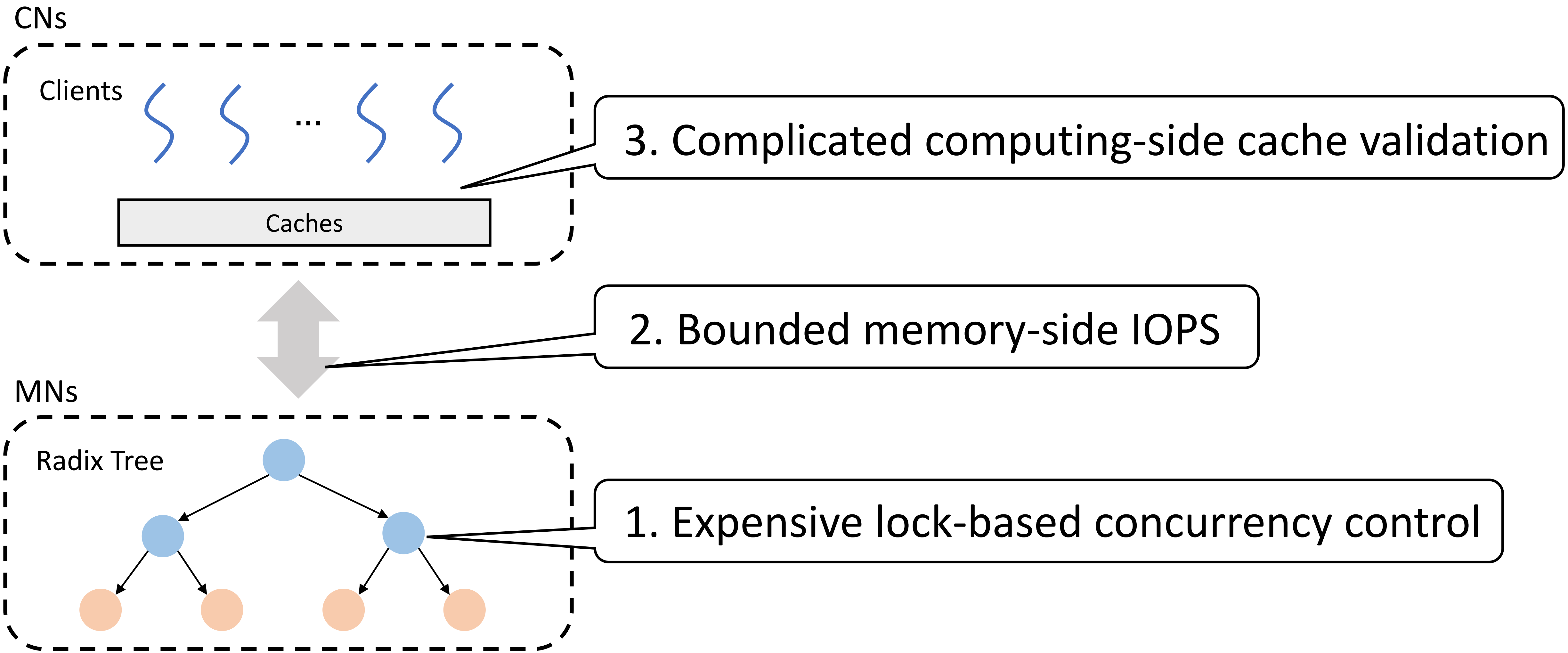
Is a cache entry outdated?

complicate

Common Optimizations:

- Path compression
- Adaptive nodes

Challenge Summary

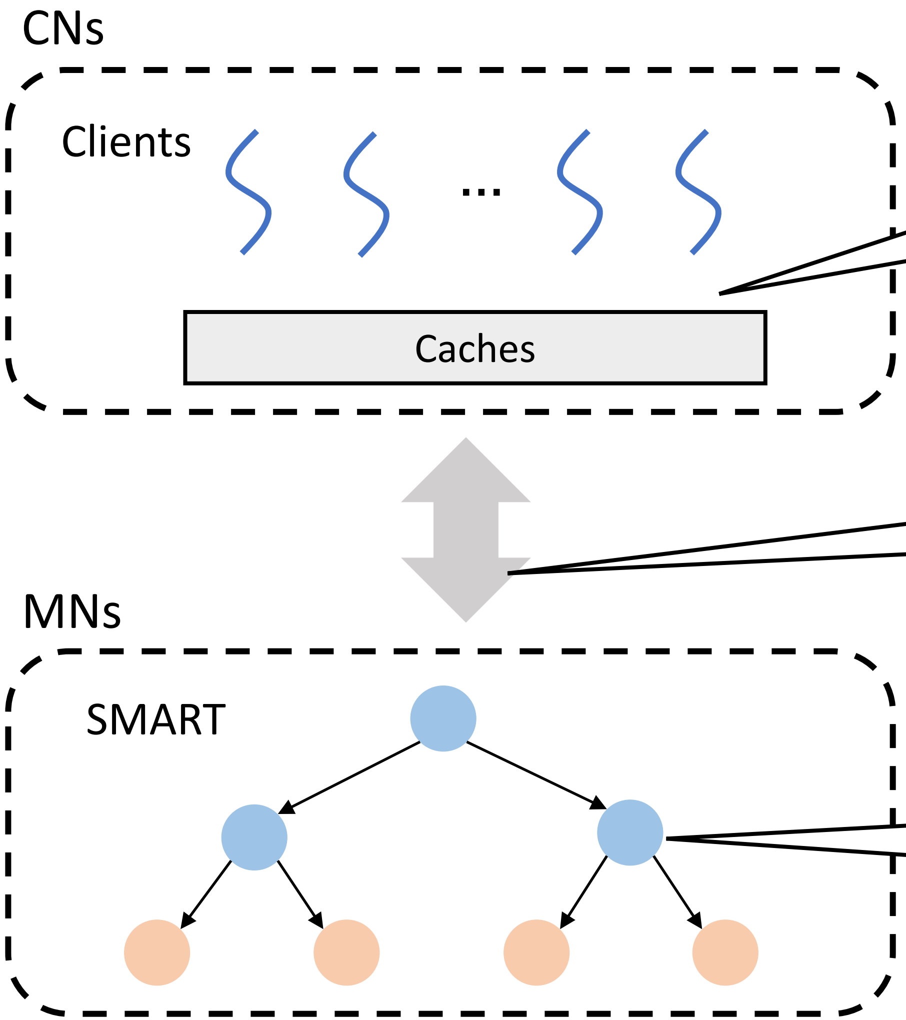


3. Complicated computing-side cache validation

2. Bounded memory-side IOPS

1. Expensive lock-based concurrency control

DiSaggregated-meMory-friendly Adaptive Radix Tree (SMART)



3. Complicated computing-side cache validation

- ➢ ***Solution 3***: *Reverse check mechanism*

2. Bounded memory-side IOPS

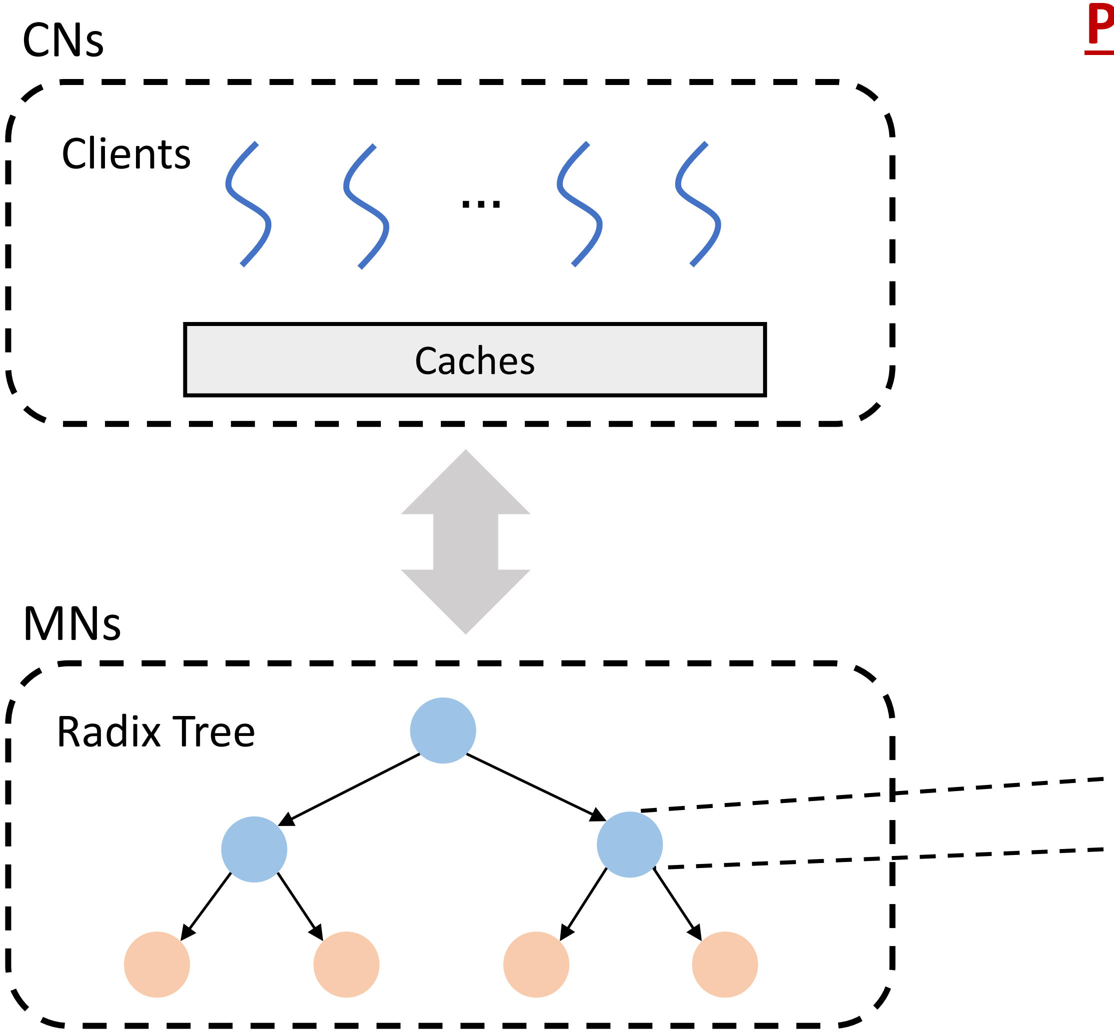
- ➢ ***Solution 2***: *Read-delegation and write-combining technique*

1. Expensive lock-based concurrency control

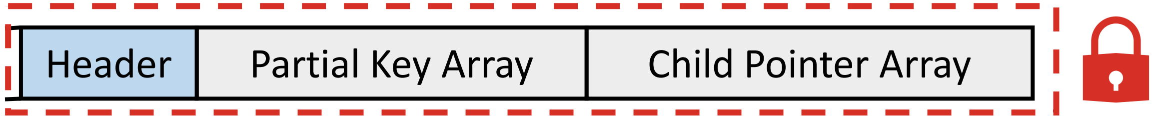
- ➢ ***Solution 1***: *Hybrid concurrency control scheme*

Hybrid Concurrency Control

Problem: Expensive lock-based concurrency control



The internal node of the state-of-the-art radix tree [3]:

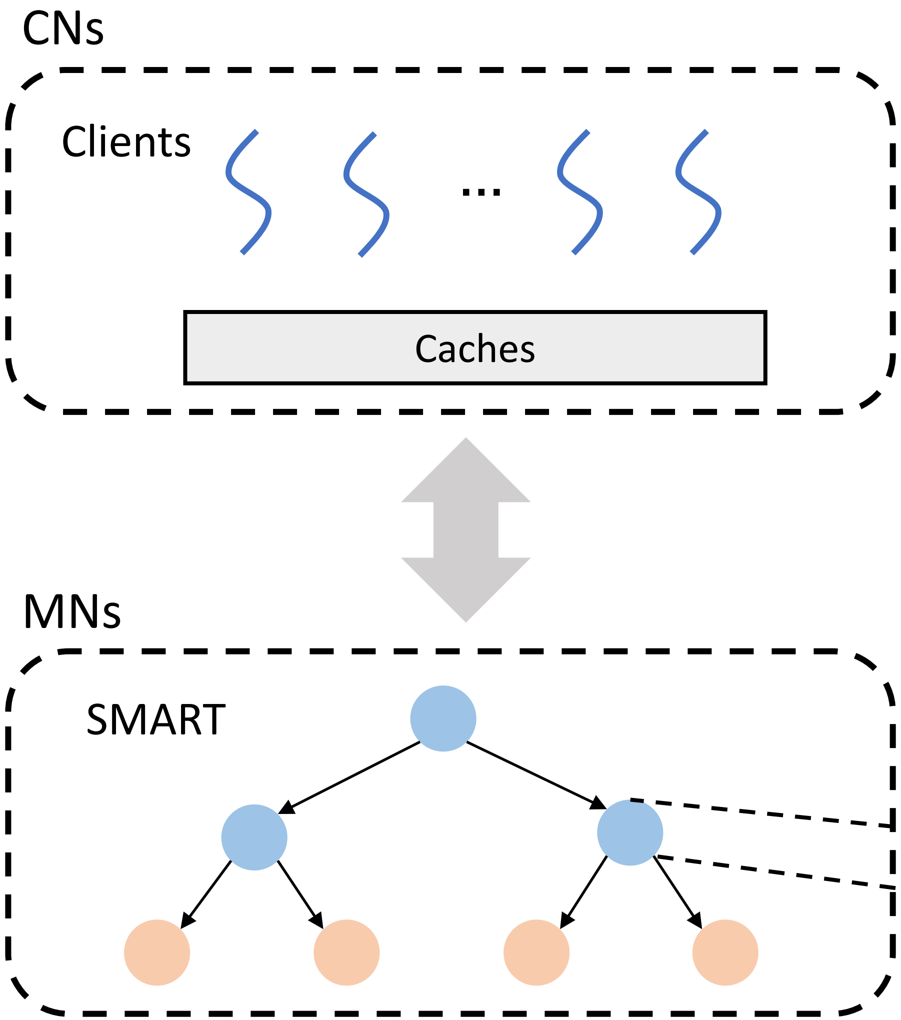


Store partial keys and child pointers separately

---

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

Hybrid Concurrency Control

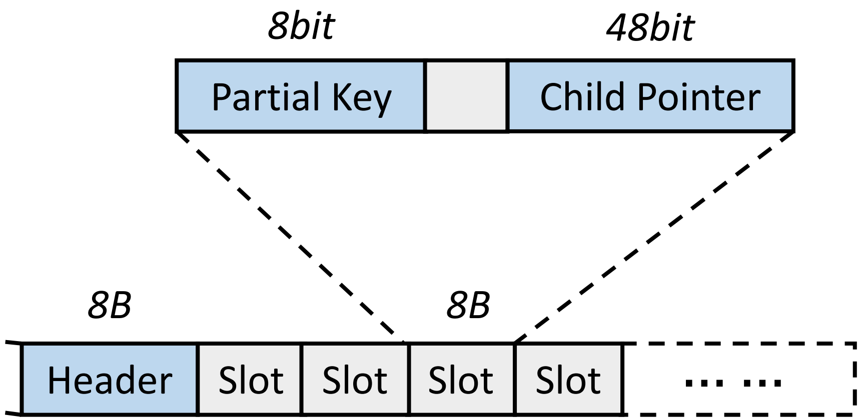


Problem: Expensive lock-based concurrency control

➢ *Solution: Lock-free internal nodes*

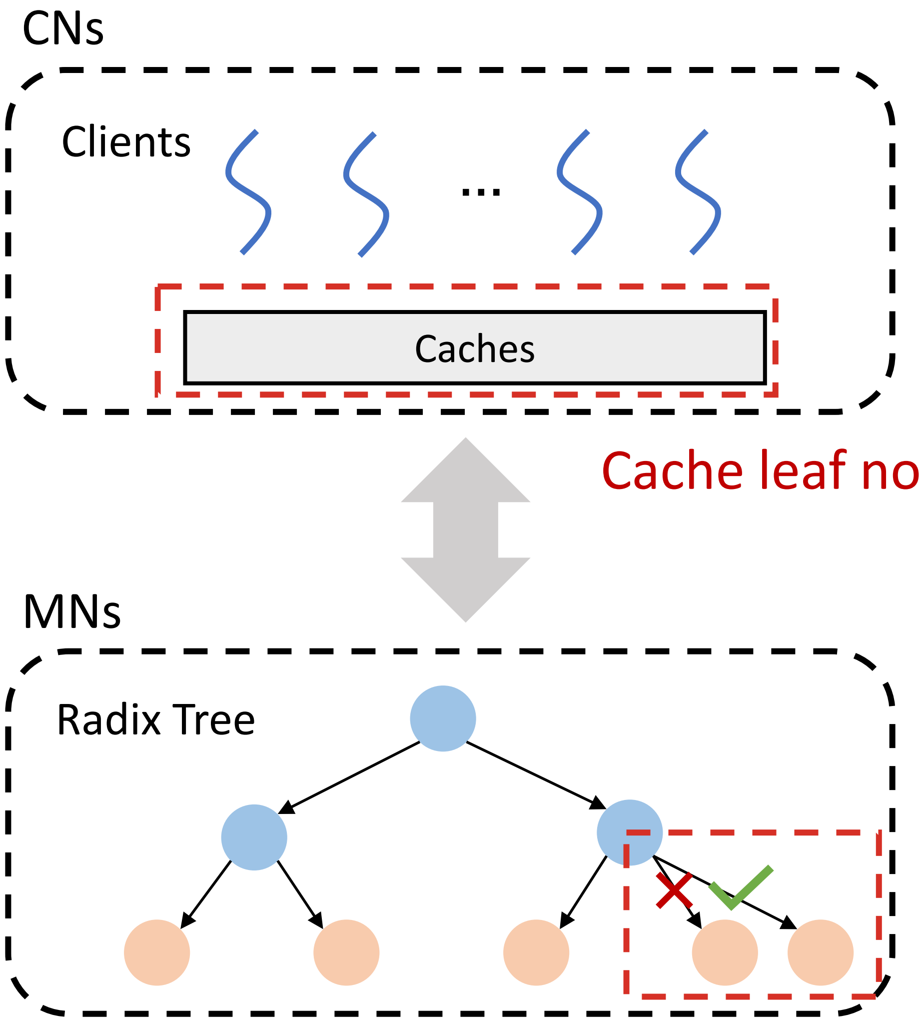
Key Idea:

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



Hybrid Concurrency Control

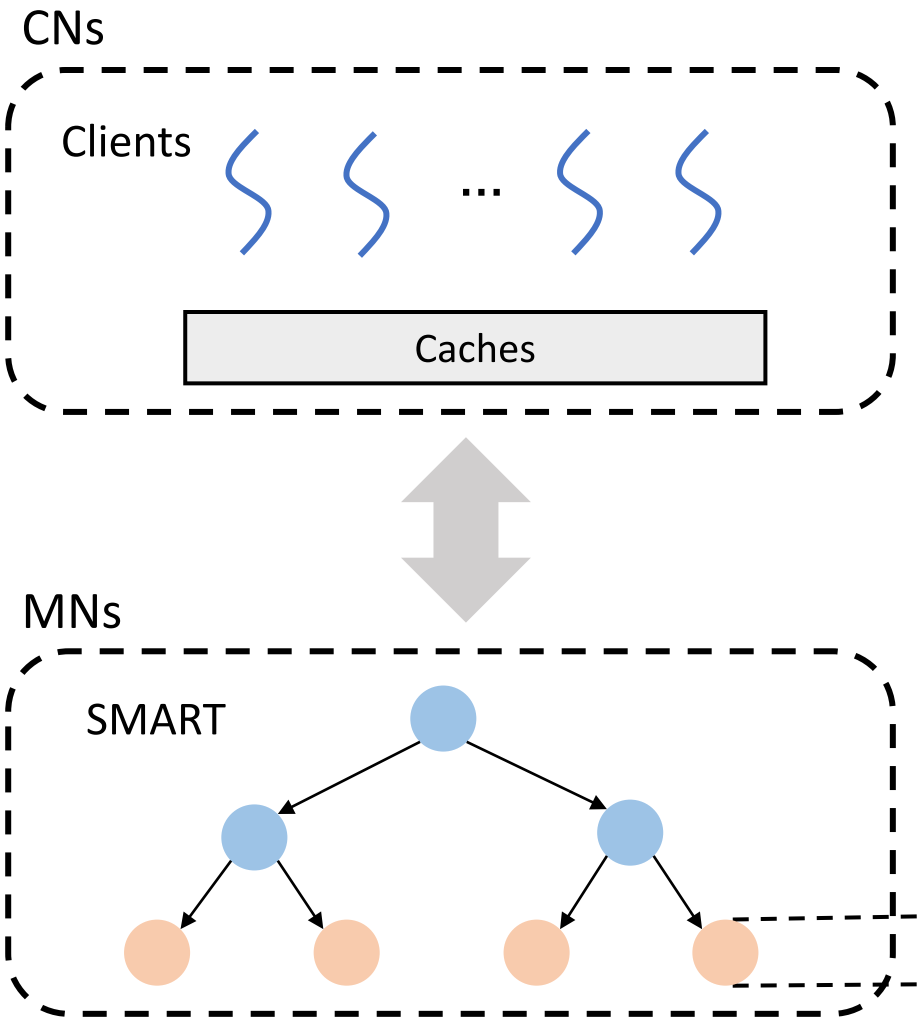
Problem: Out-of-place update causes cache thrashing



Cache leaf node addresses

Out-of-place update changes leaf node addresses frequently

Hybrid Concurrency Control

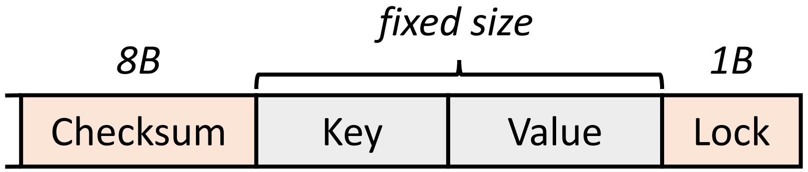


Problem: Out-of-place update causes cache thrashing

➢ *Solution: Lock-based update-in-place leaf nodes*

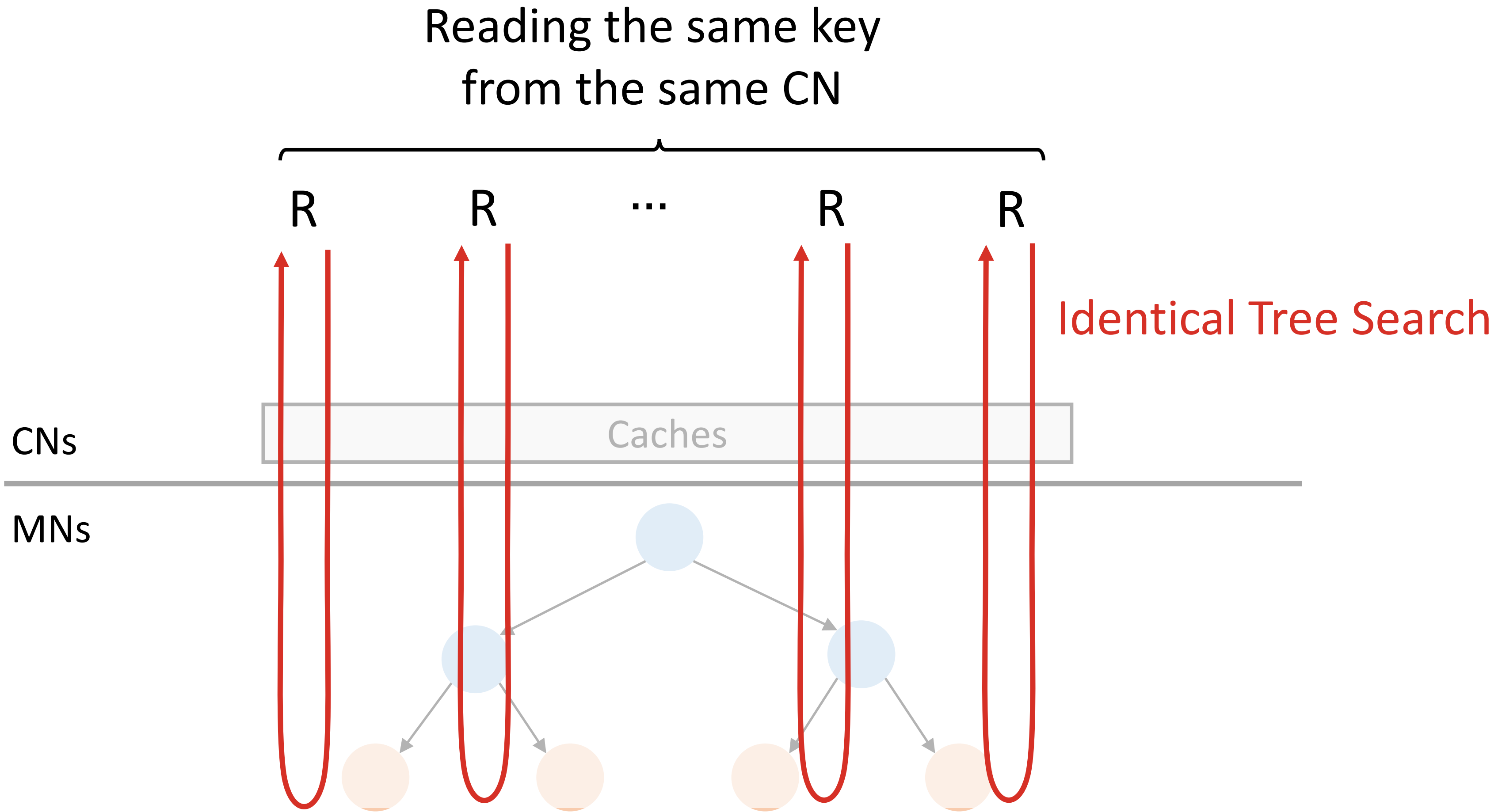
Key Idea:

- Write-write conflict ➢ *Rear embedded lock*
  - Combine lock release with writing back
- Read-write conflict ➢ *Checksum-based method*
  - Writer: checksum := CRC(KV)
  - Reader: checksum == CRC(KV)?

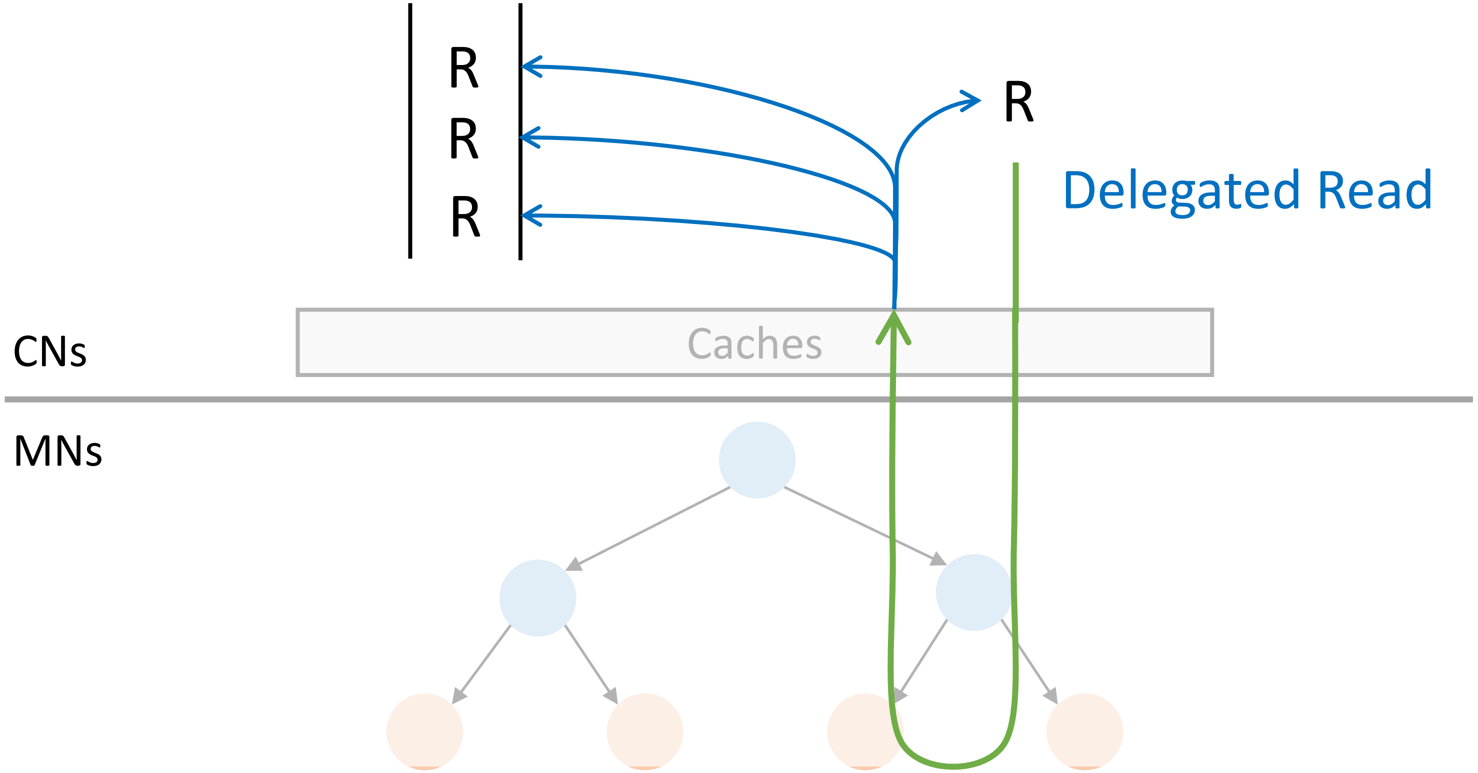


Read Delegation and Write Combining

Problem: Redundant read I/Os



Read Delegation and Write Combining



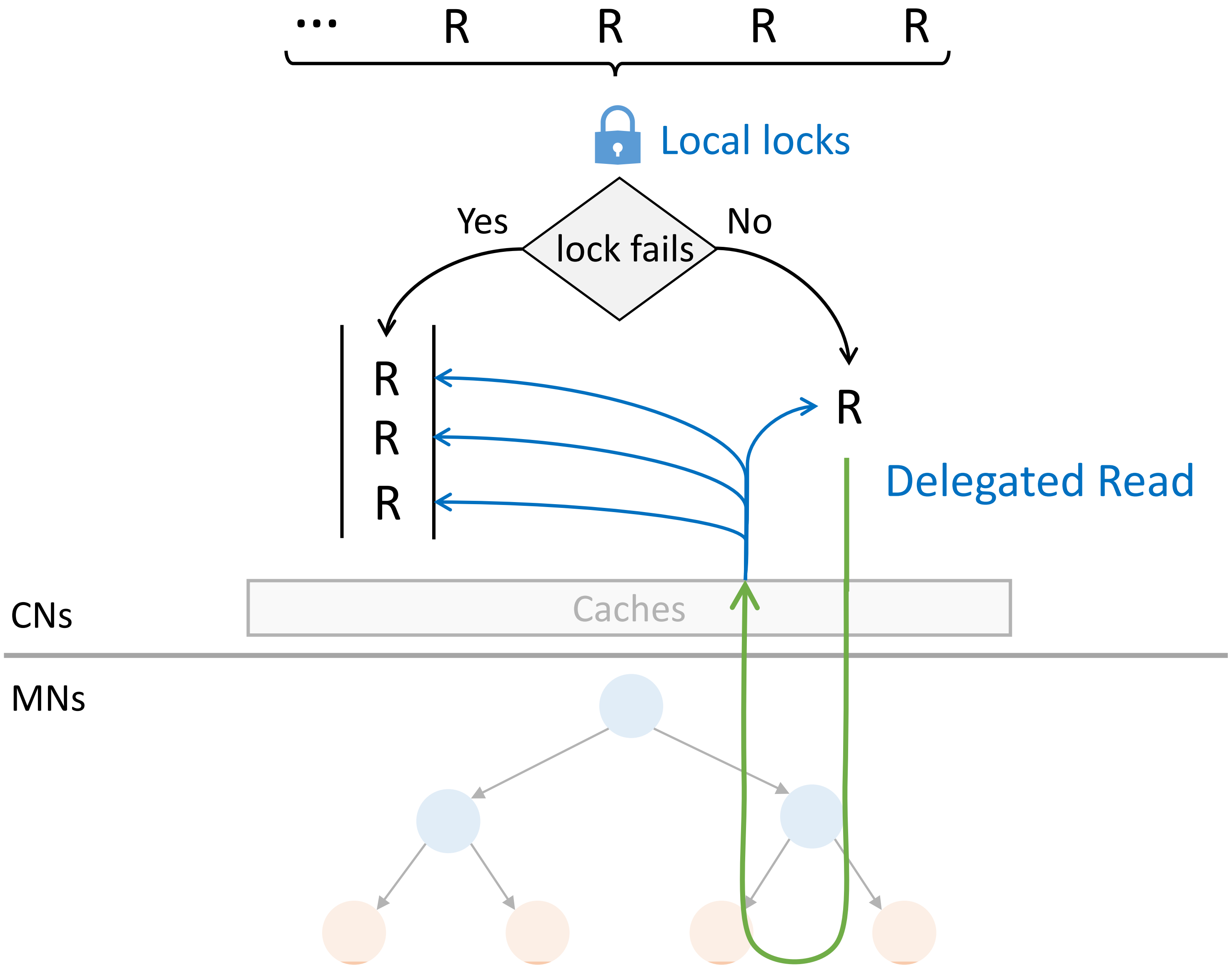
Problem: Redundant read I/Os

➢ *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

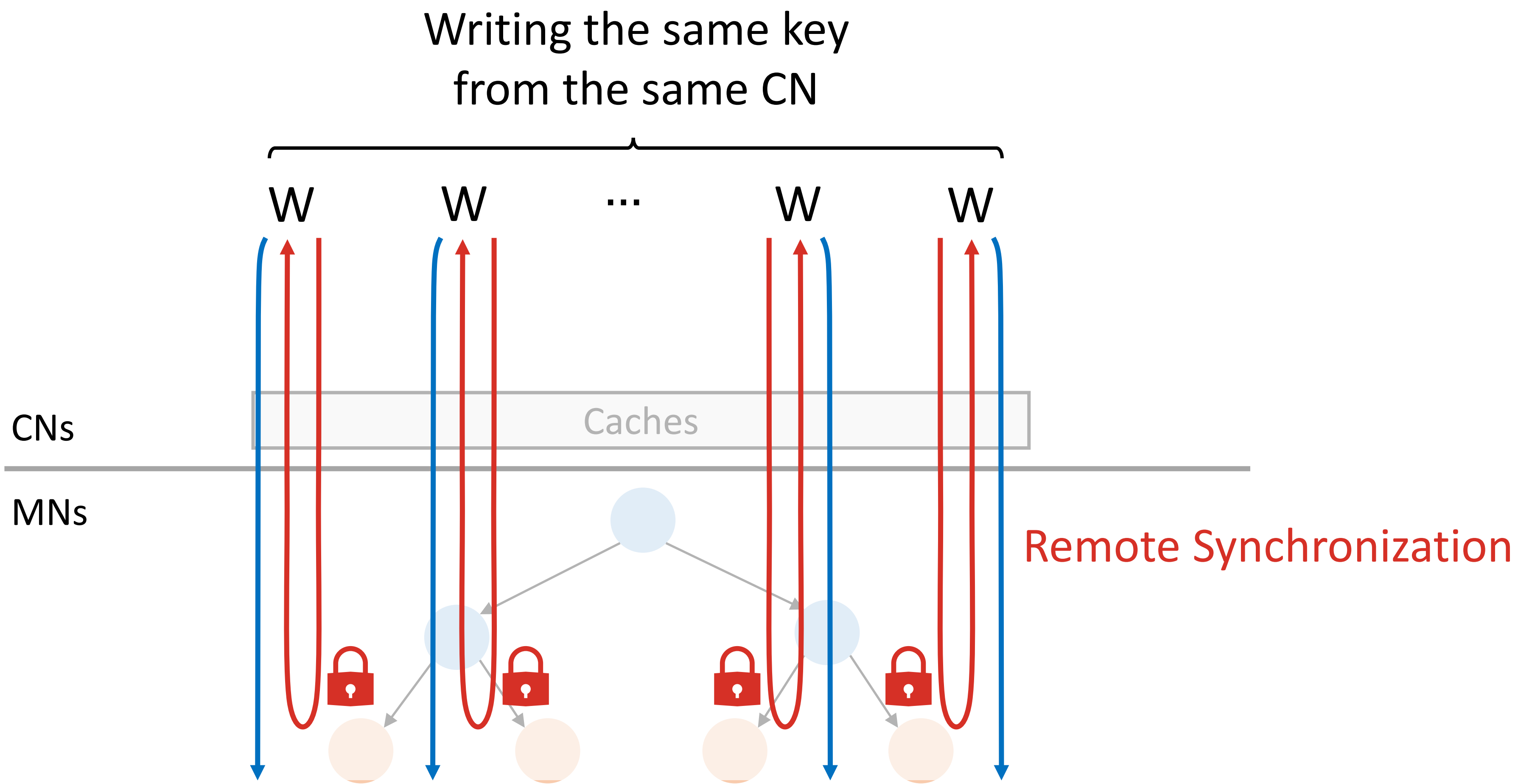
- ➢ *Solution: Read delegation*

Key Idea:

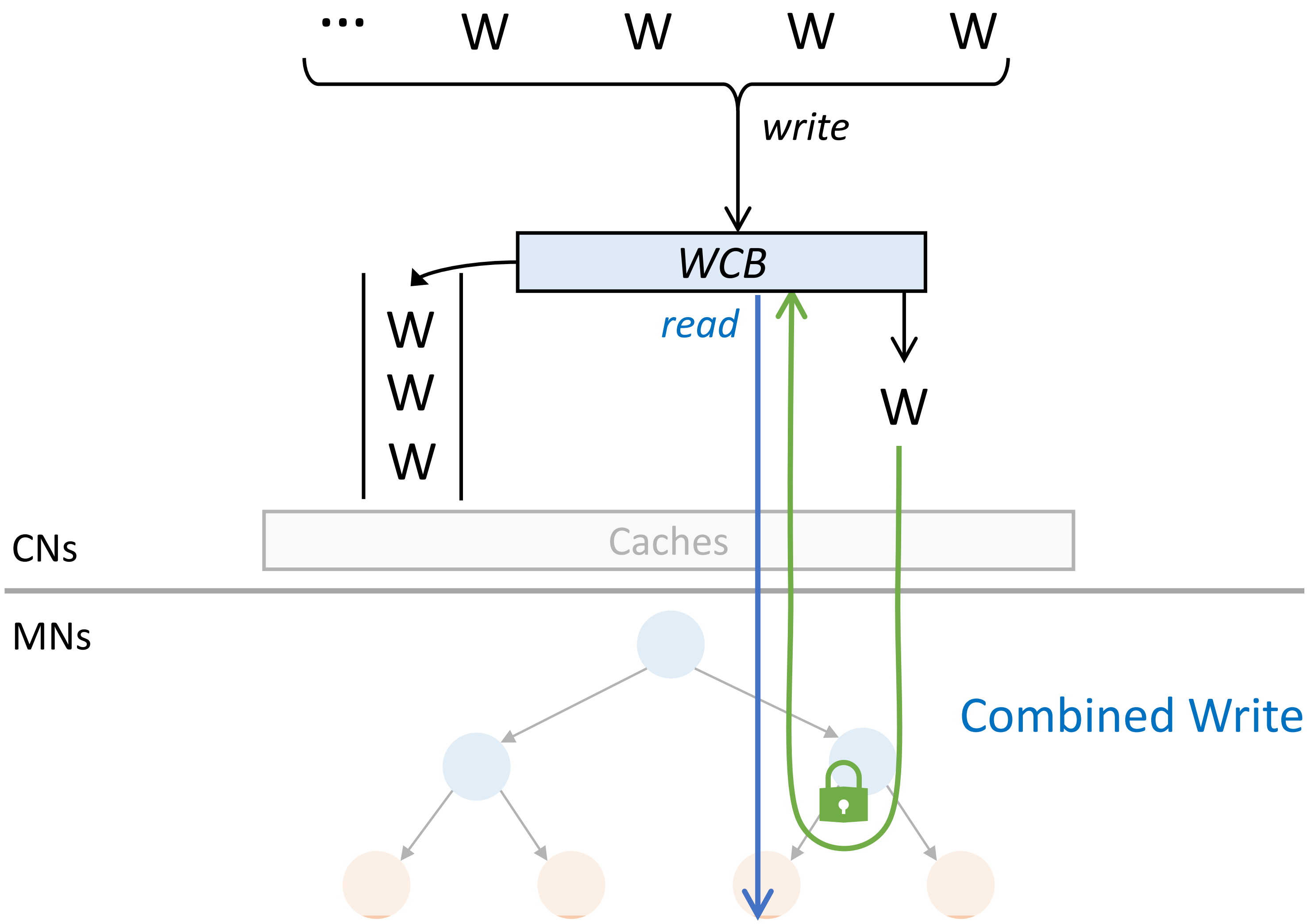
- Choose a delegation client on each CN to execute the same read
- Use local locks to collect concurrent identical reads

Read Delegation and Write Combining

Problem: Redundant write I/Os



Read Delegation and Write Combining



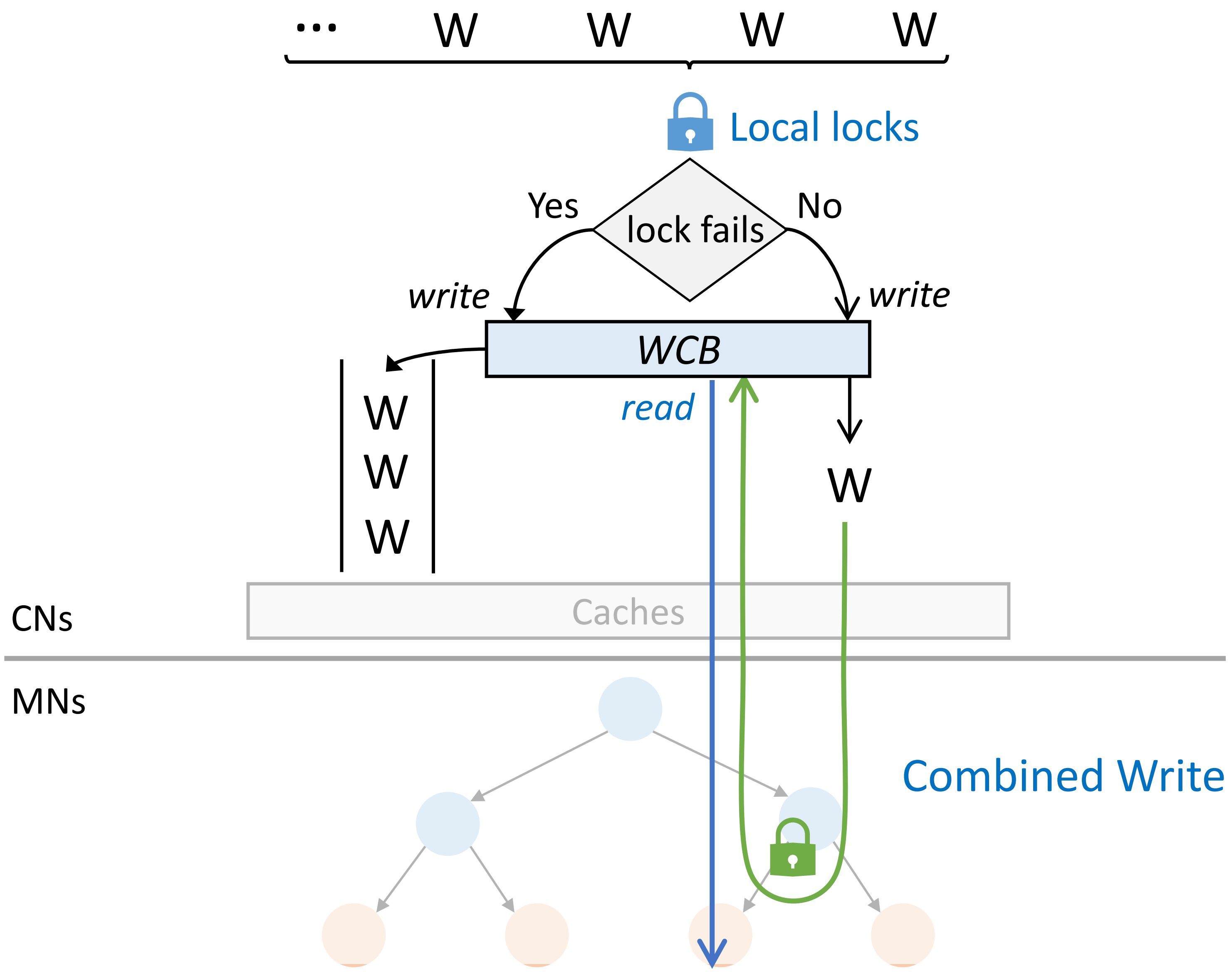
Problem: Redundant write I/Os

➢ *Solution: Write combining*

Key Idea:

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

Read Delegation and Write Combining



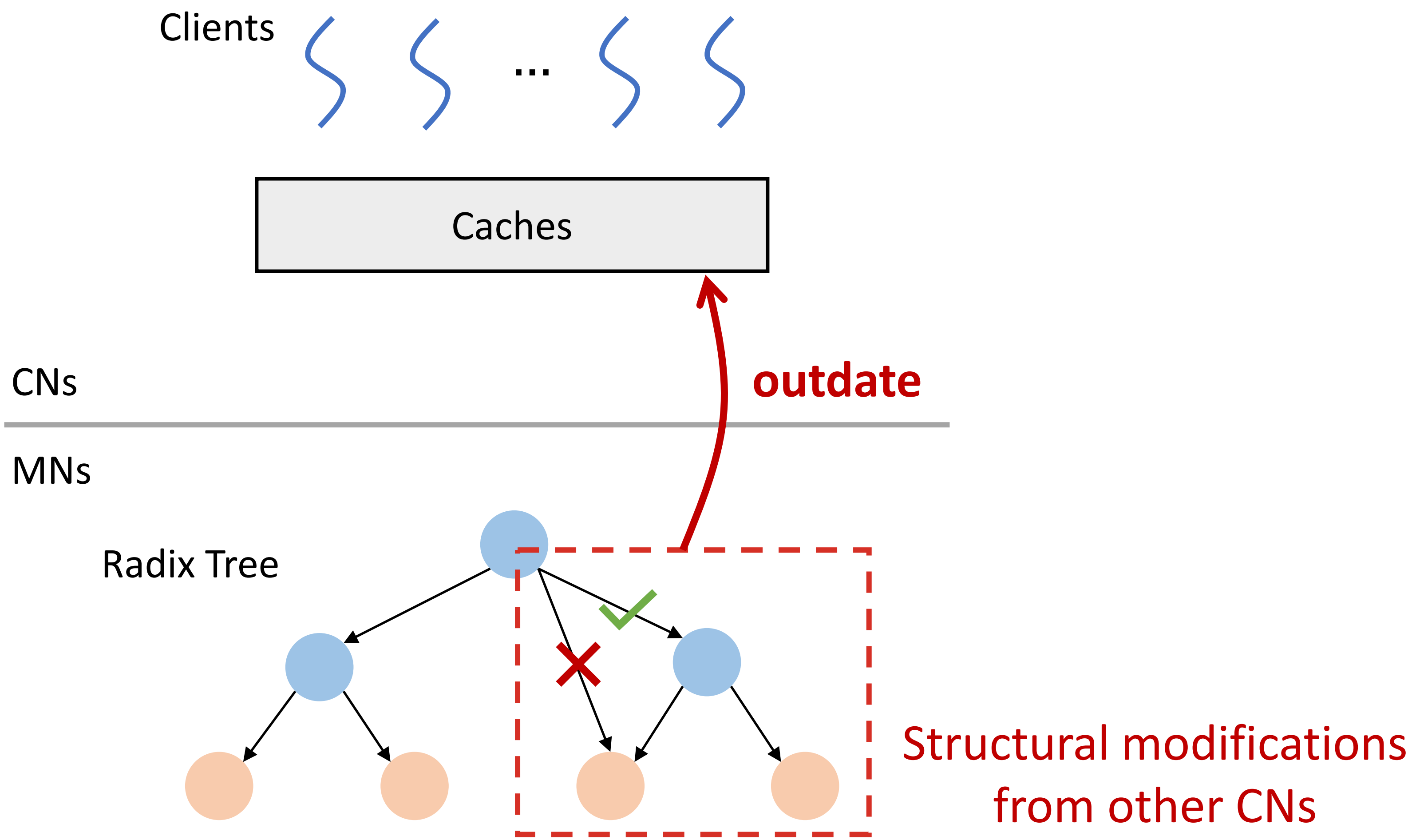
Problem: Redundant write I/Os

- *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

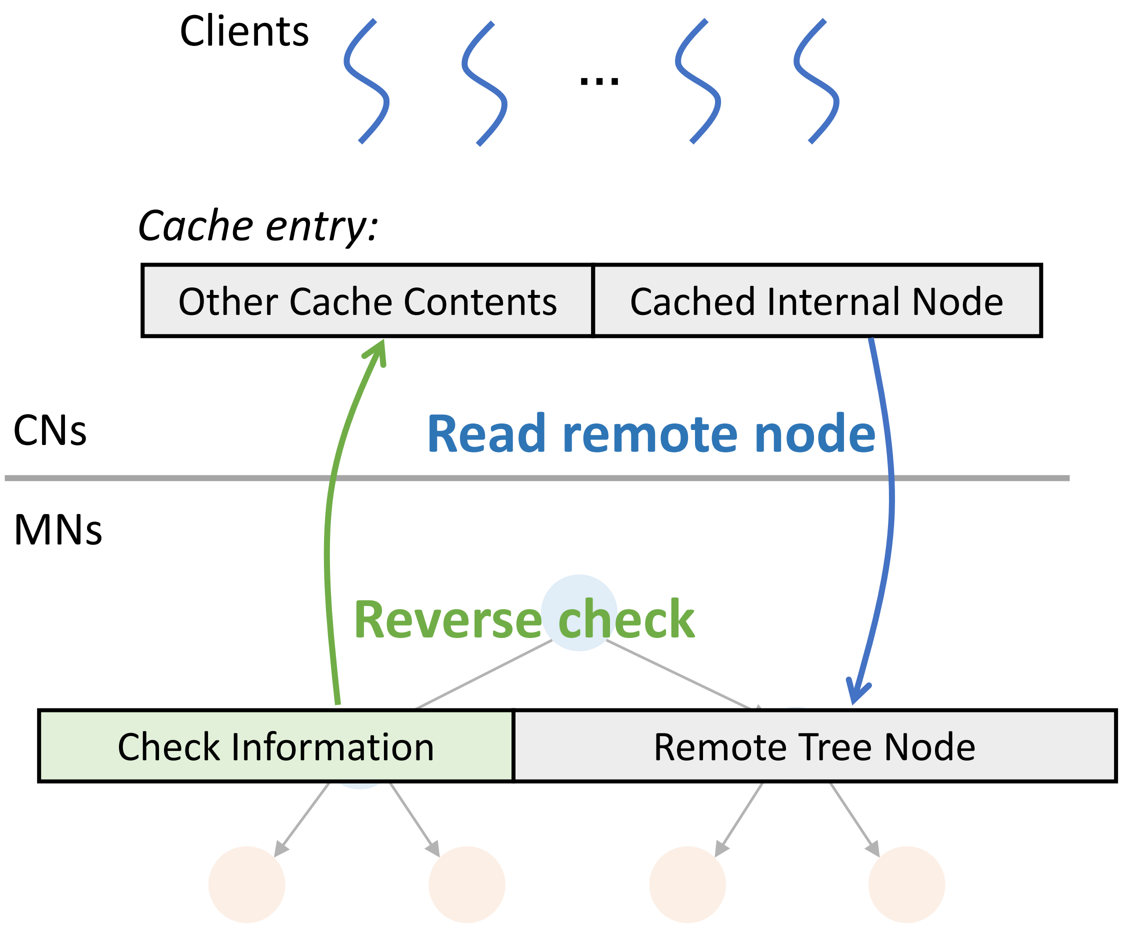


outdate

Structural modifications from other CNs

Problem: Cache invalidation of the radix tree

Reverse Check



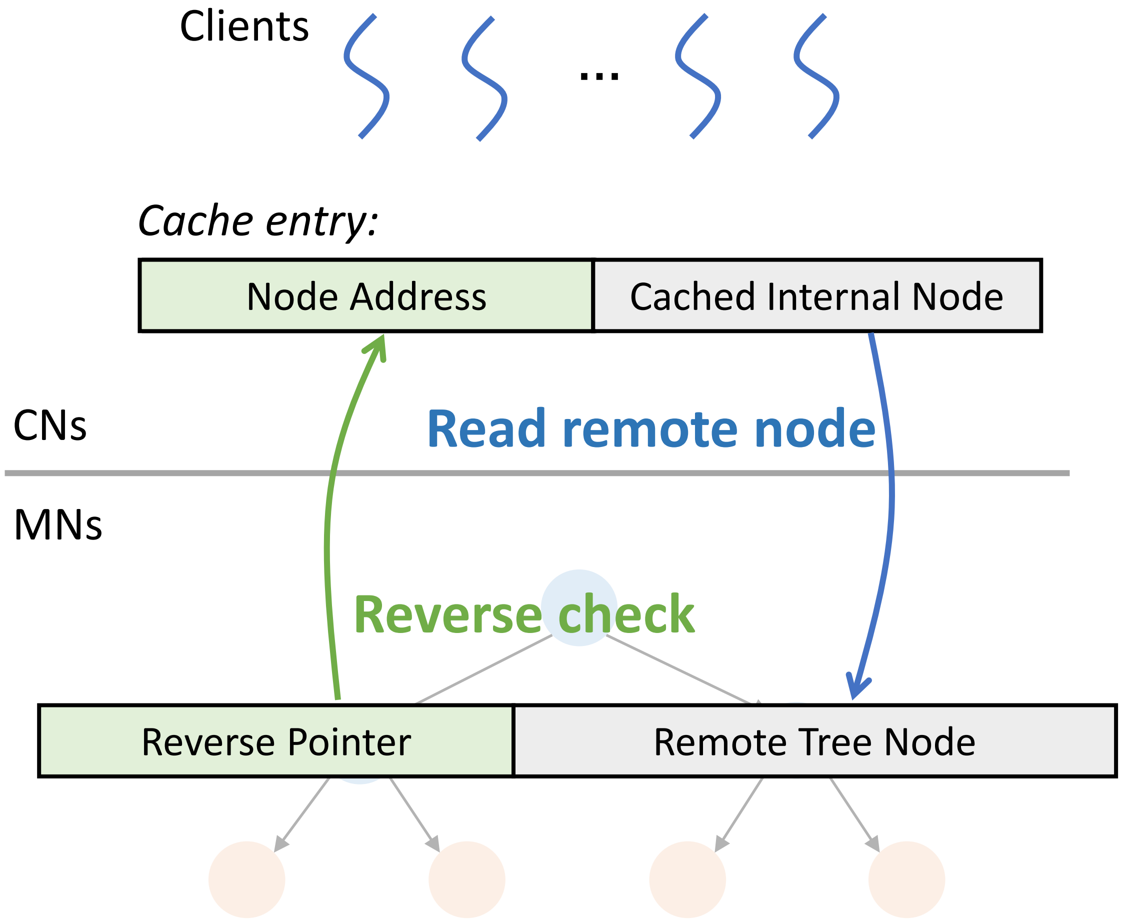
<u>Problem</u>: Cache invalidation of the radix tree

- *<u>Solution</u>: Reverse check mechanism*

<u>Key Idea</u>:

- 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:

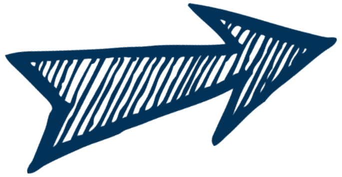
- Store check information in each remote node
- Check: check information == cache content ?

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[1,*], Pengfei Zuo[2], Jiacheng Shen[3,*], Jiazhen Gu[3], Xin Wang[1,4], Michael R. Lyu[3], and Yangfan Zhou[1,4]

Abstract

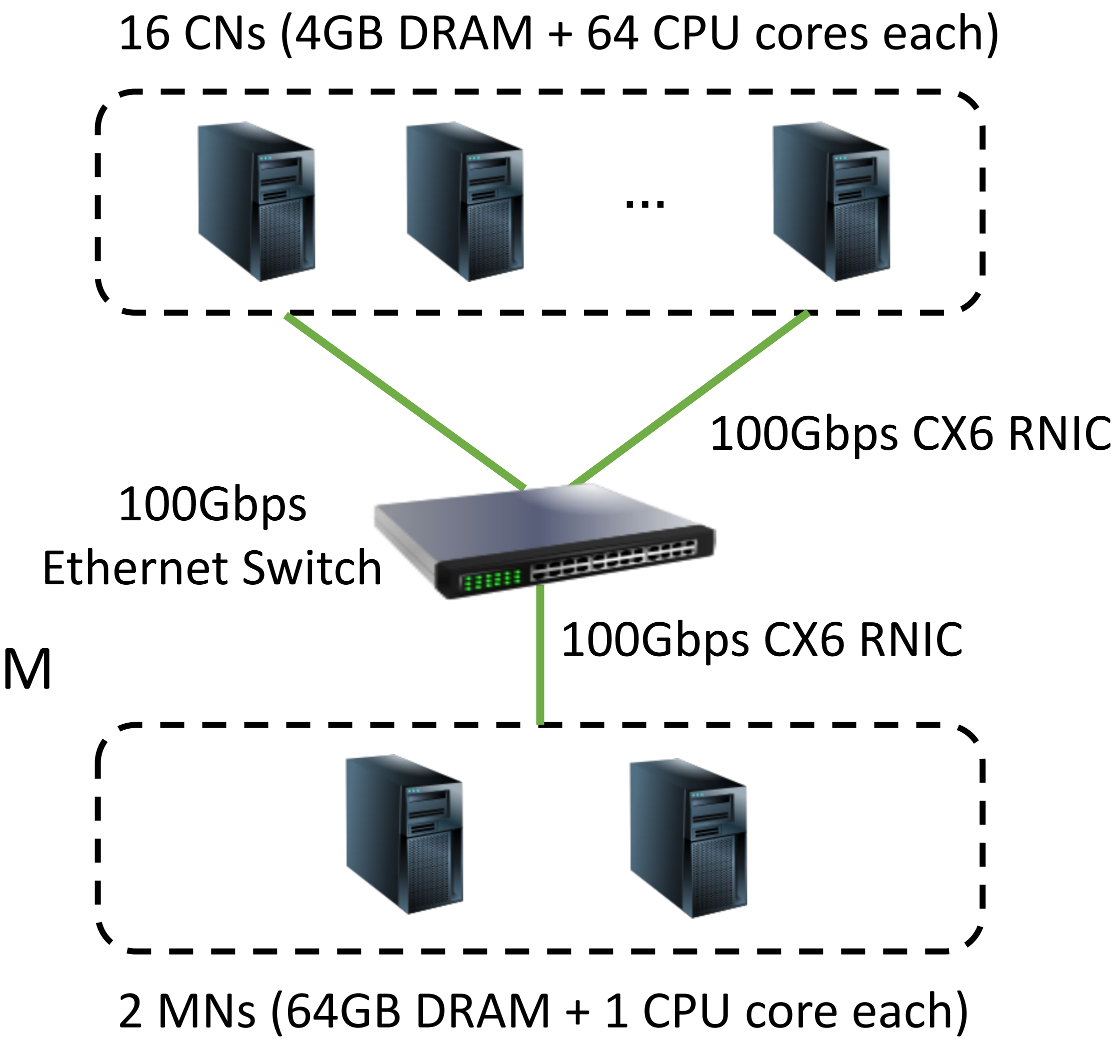
Evaluation

Workloads

- YCSB workloads
- 2 key types: integer, string

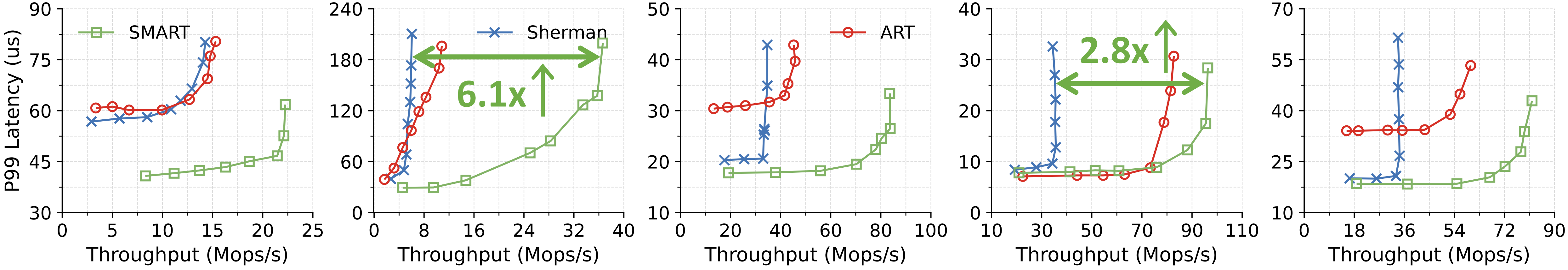
Comparisons

- 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

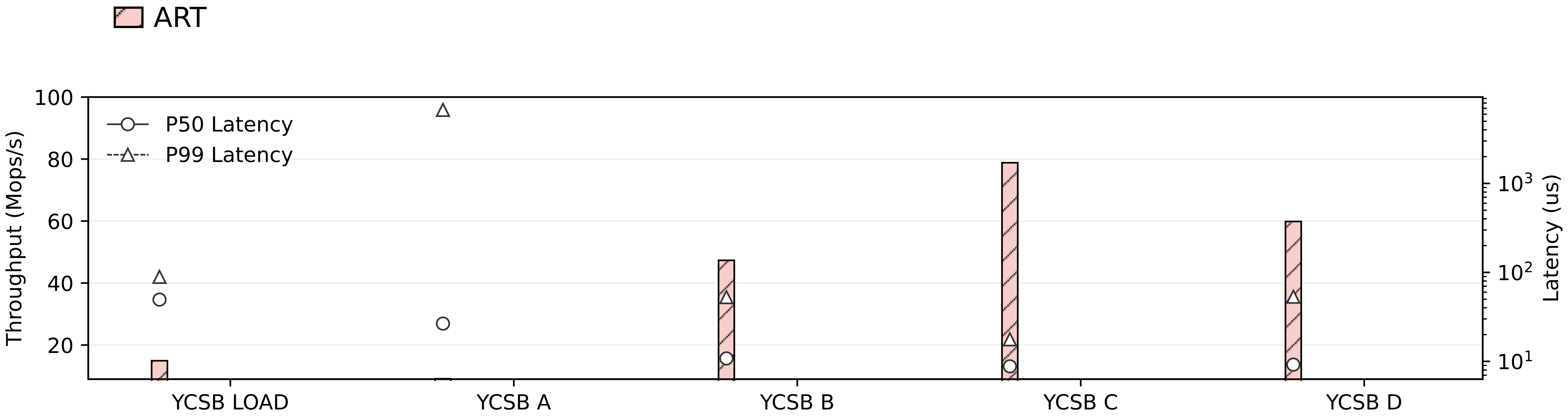
YCSB C

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

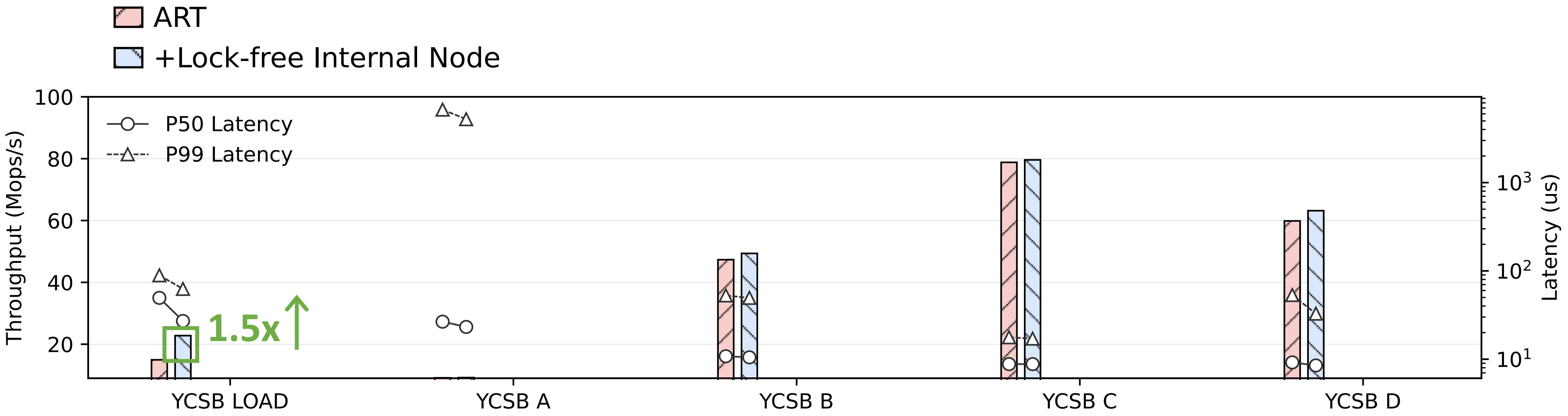
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



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

Factor Analysis for SMART design

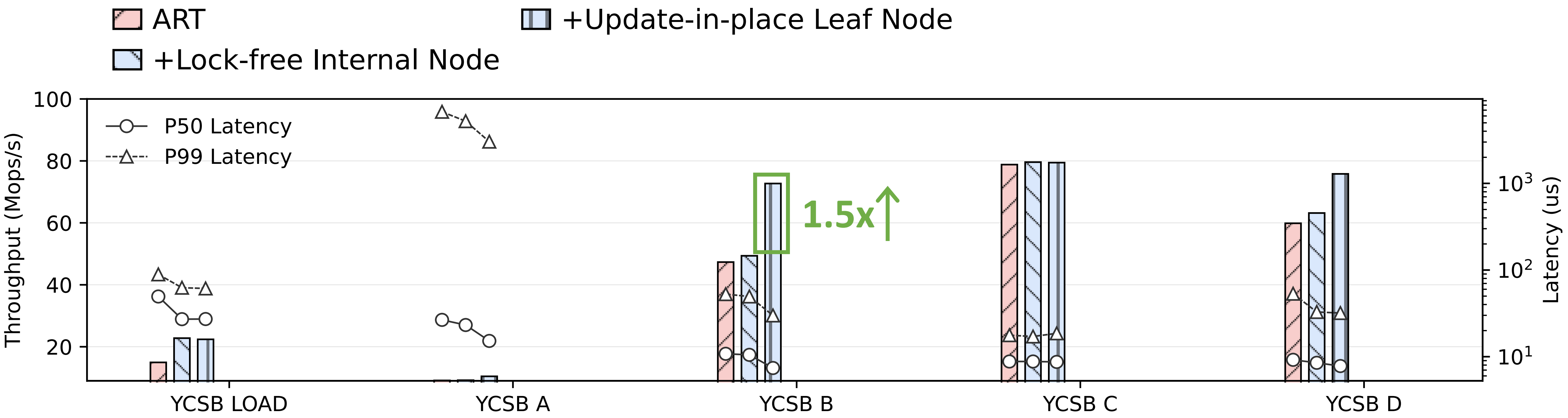
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



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

Factor Analysis for SMART design

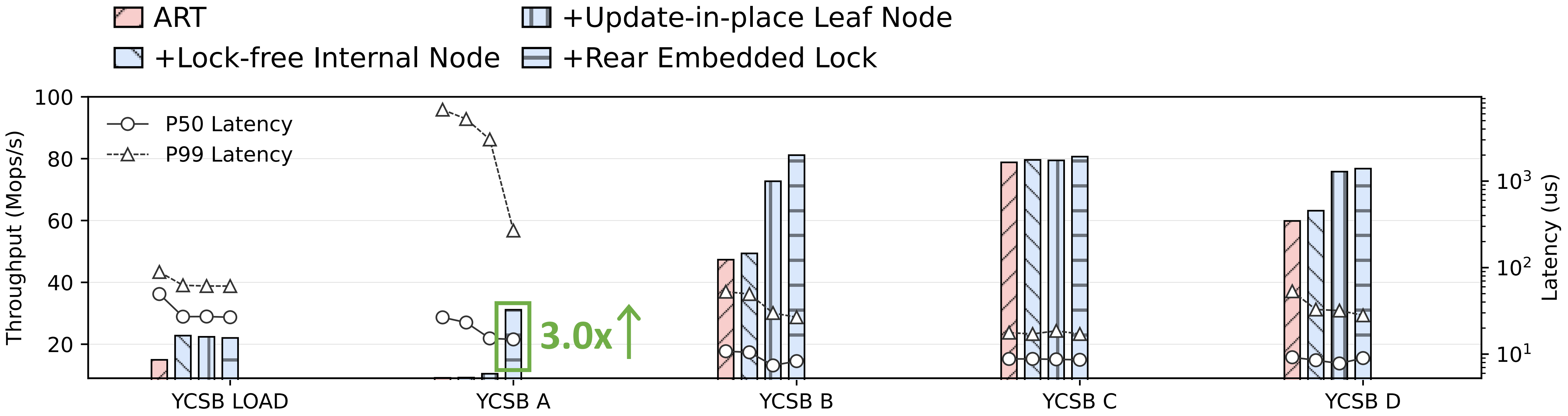
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



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

Factor Analysis for SMART design

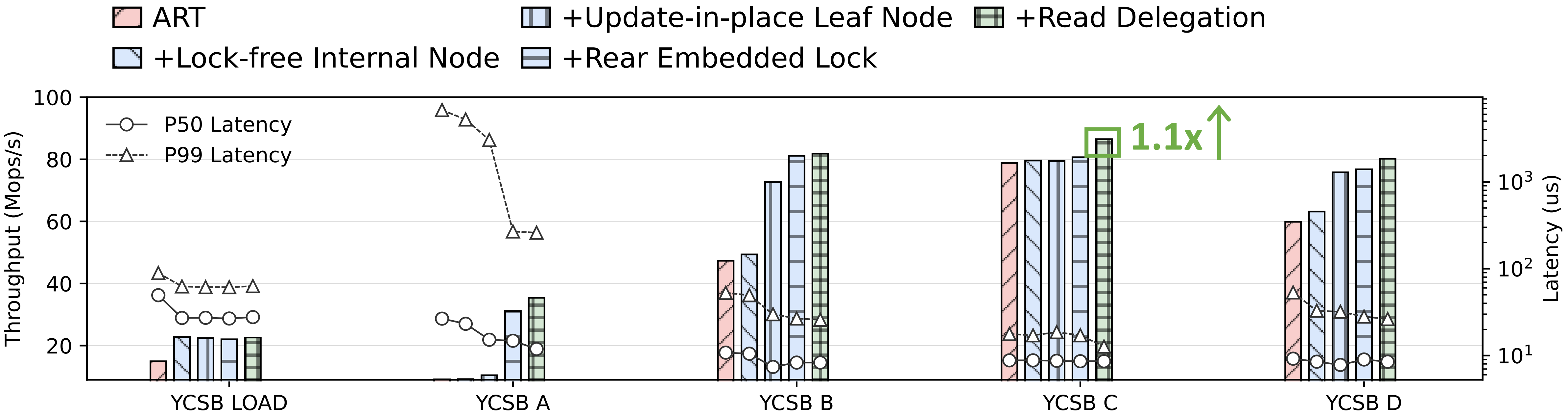
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



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

Factor Analysis for SMART design

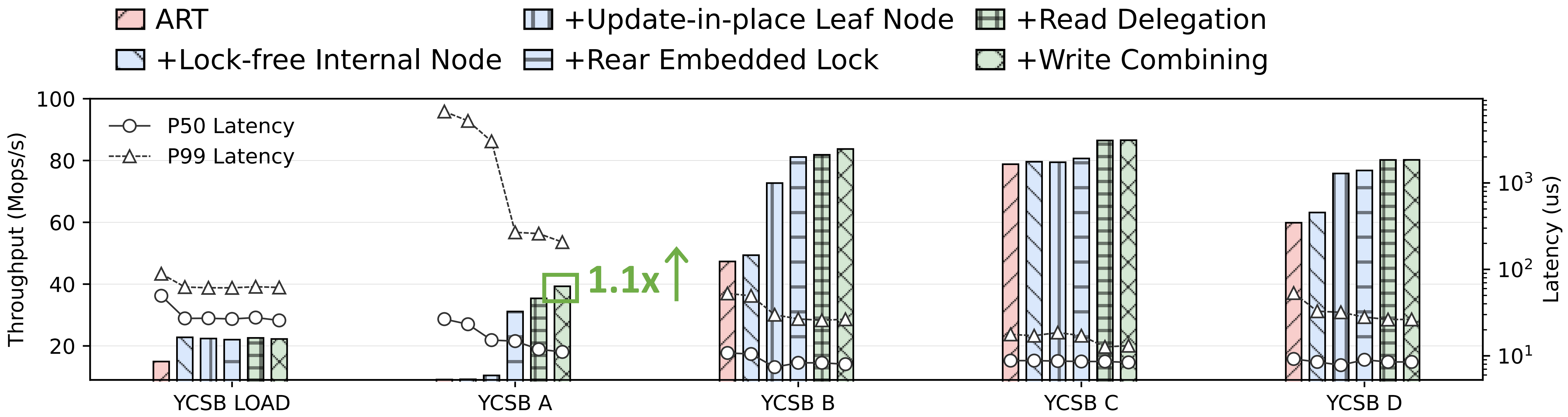
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read, 50% update	95% read, 5% update	100% read	95% read, 5% insert



- ➢ 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