SOSP 2024

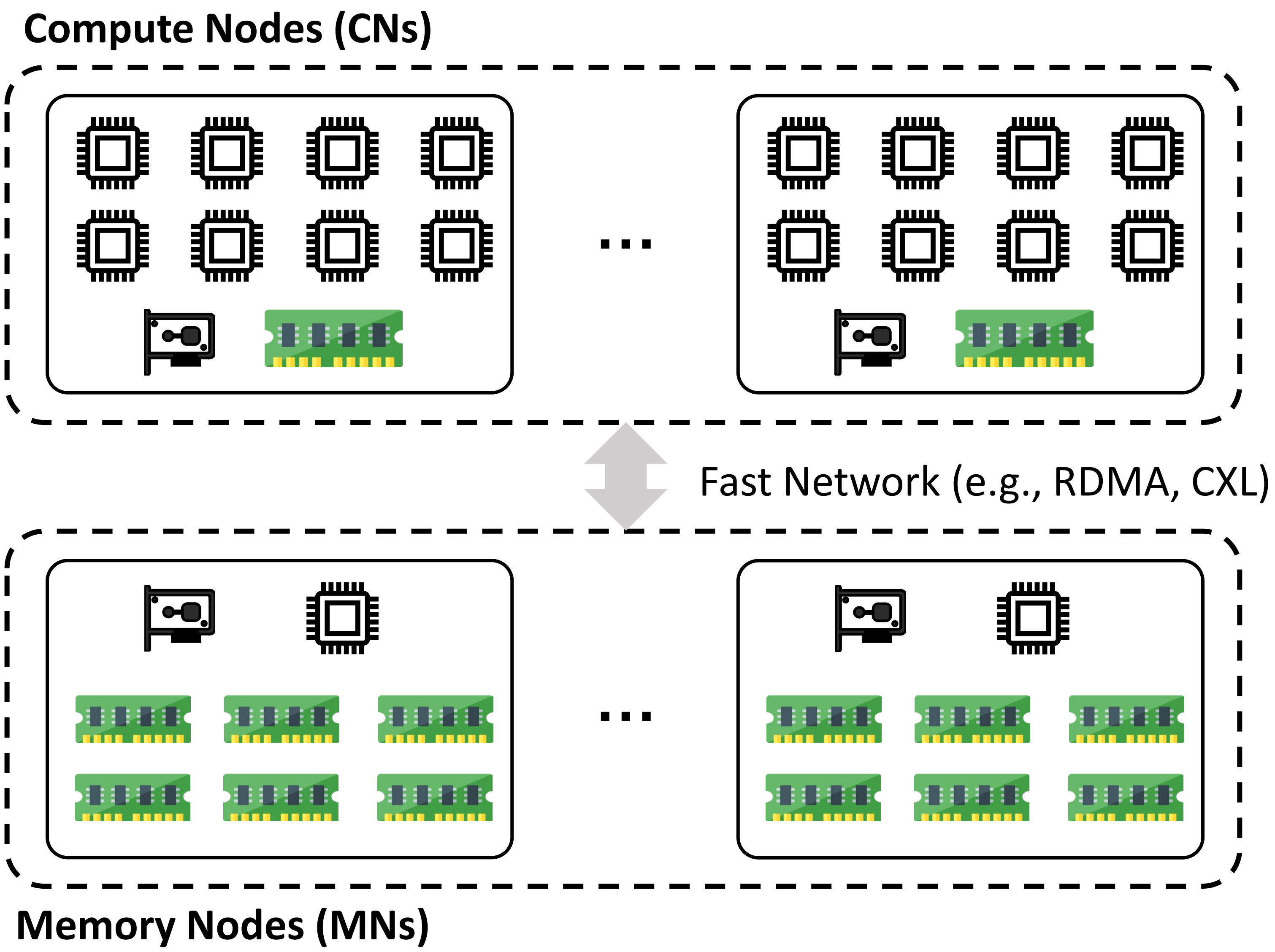
CHIME: A Cache-Efficient and High-Performance Hybrid Index on Disaggregated Memory

Xuchuan Luo[1,2], Jiacheng Shen[3], Pengfei Zuo[4], Xin Wang[1,6], Michael R. Lyu[5], Yangfan Zhou[1,2]

[1]School of Computer Science, Fudan University
[2]National Key Laboratory of Parallel and Distributed Computing, China
[3]Duke Kunshan University *[4]Huawei Cloud* *[5]The Chinese University of Hong Kong*
[6]Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China

复旦大學 FUDAN UNIVERSITY
昆山杜克大学 DUKE KUNSHAN UNIVERSITY
HUAWEI
香港中文大學 The Chinese University of Hong Kong

Disaggregated Memory (DM)



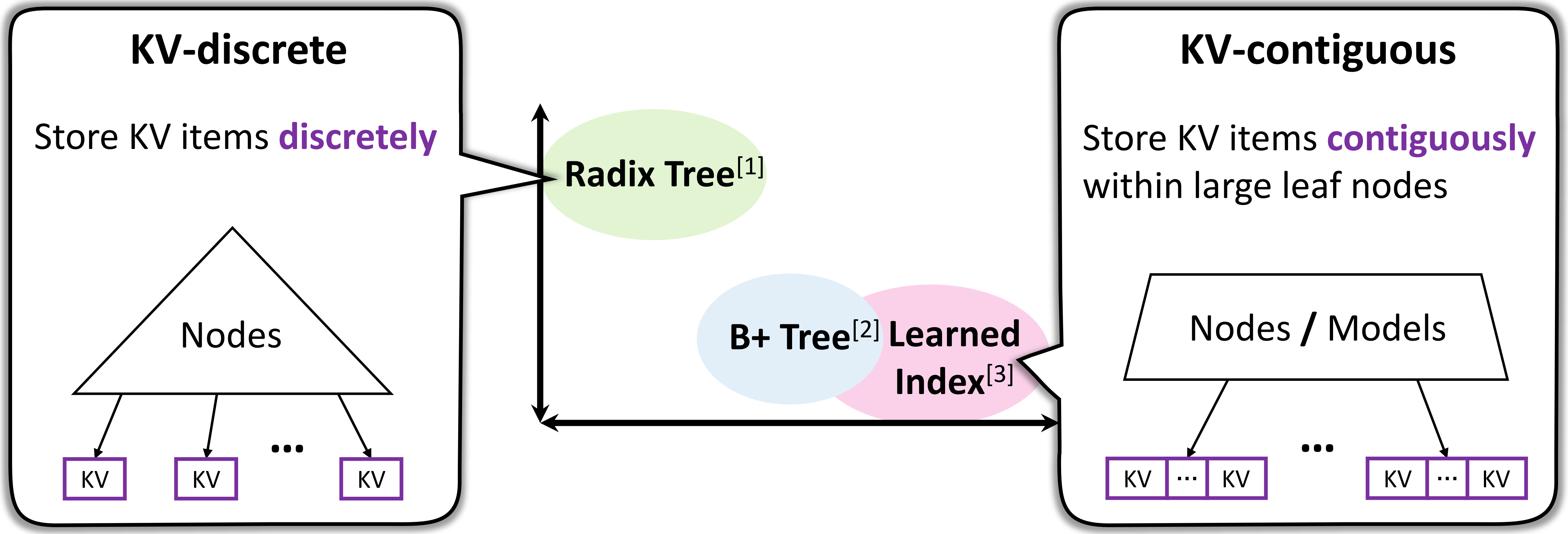
☺ **Benefits:**

✓ Resource utilization

✓ Elasticity

Range Indexes on Disaggregated Memory

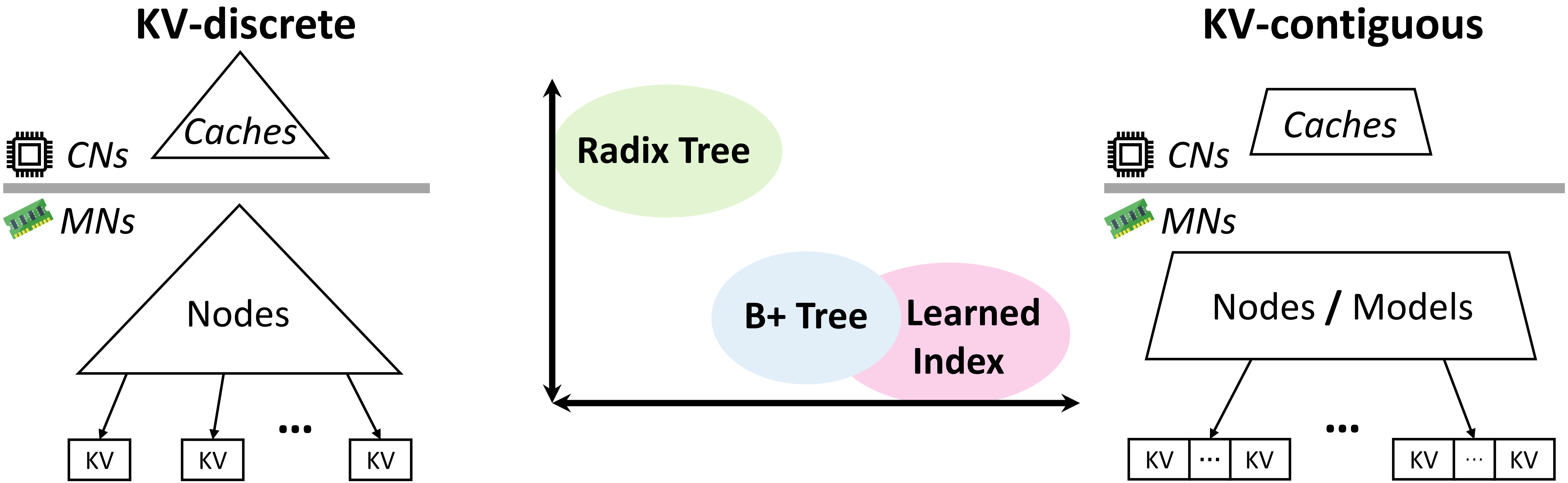
Existing range indexes on DM can be classified into two types:



[1] Xuchuan Luo et al. SMART: A high-performance adaptive radix tree for disaggregated memory. OSDI 2023.
[2] Qing Wang et al. Sherman: A write-optimized distributed B+ tree index on disaggregated memory. SIGMOD 2022.
[3] Pengfei Li et al. ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory. FAST 2023.

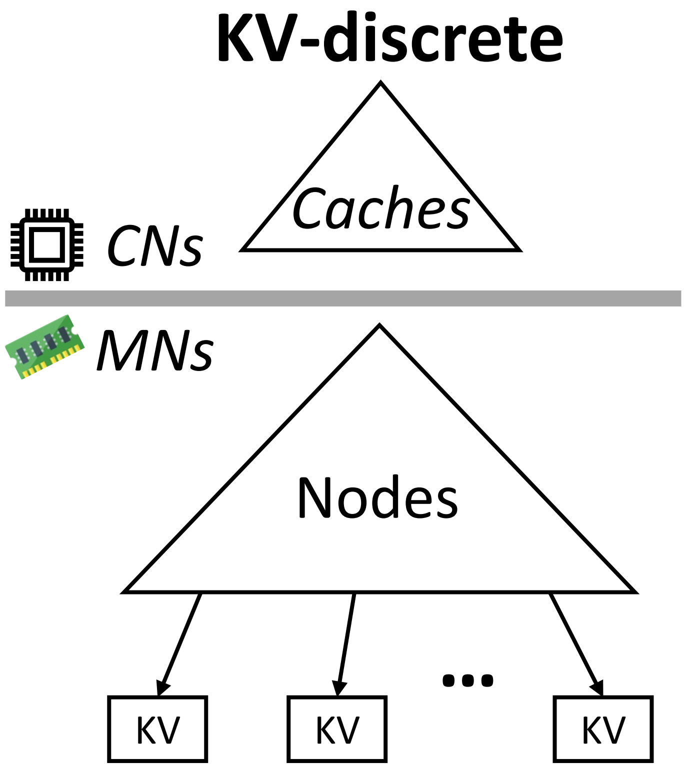
Range Indexes on Disaggregated Memory

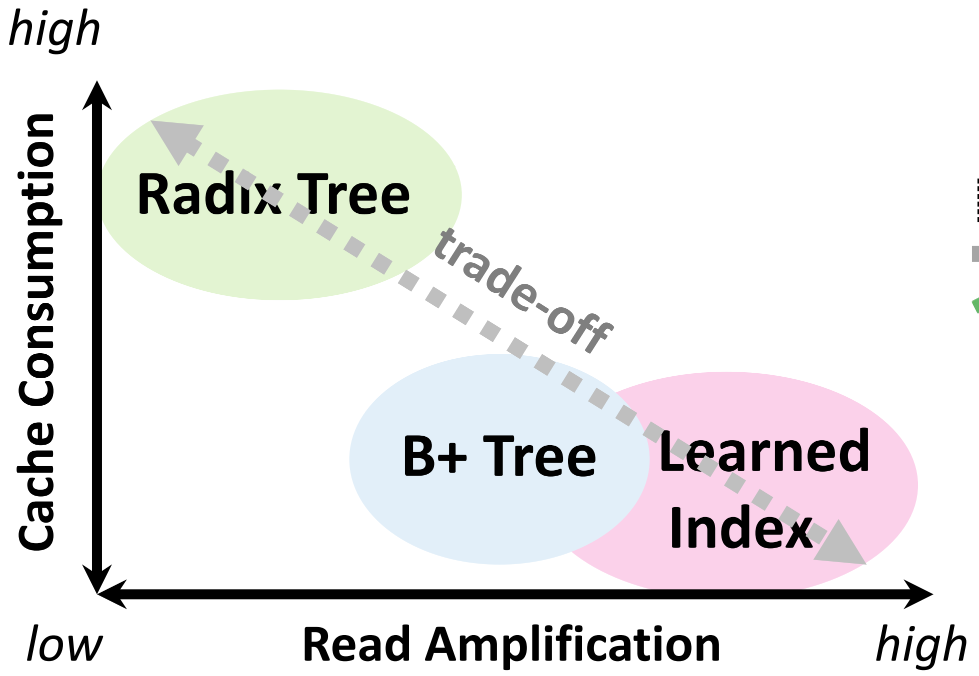
Computing-side caches are adopted to reduce access latency:

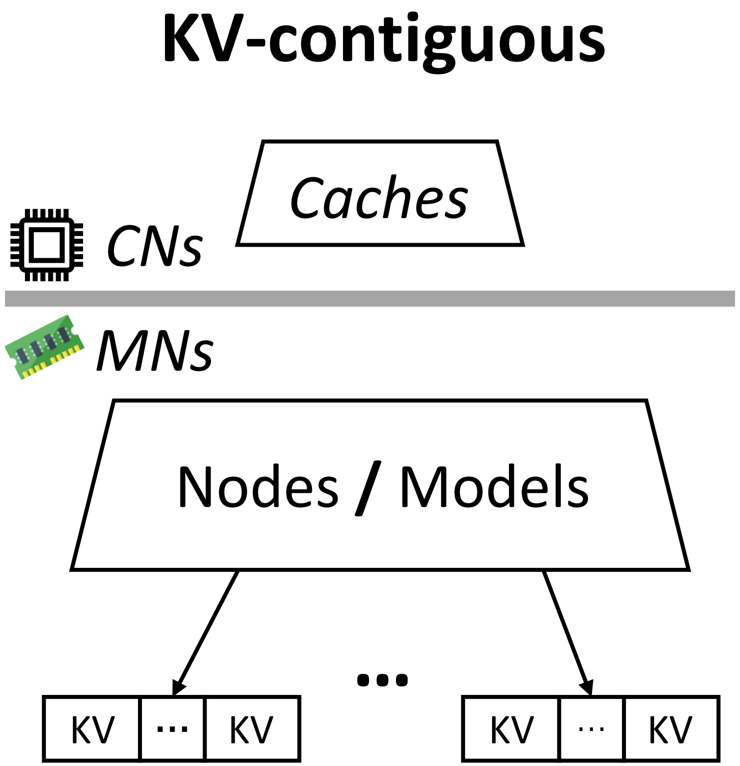


Motivation: A Trade-off for Range indexes on DM

There is a trade-off between read amplifications and cache consumption:

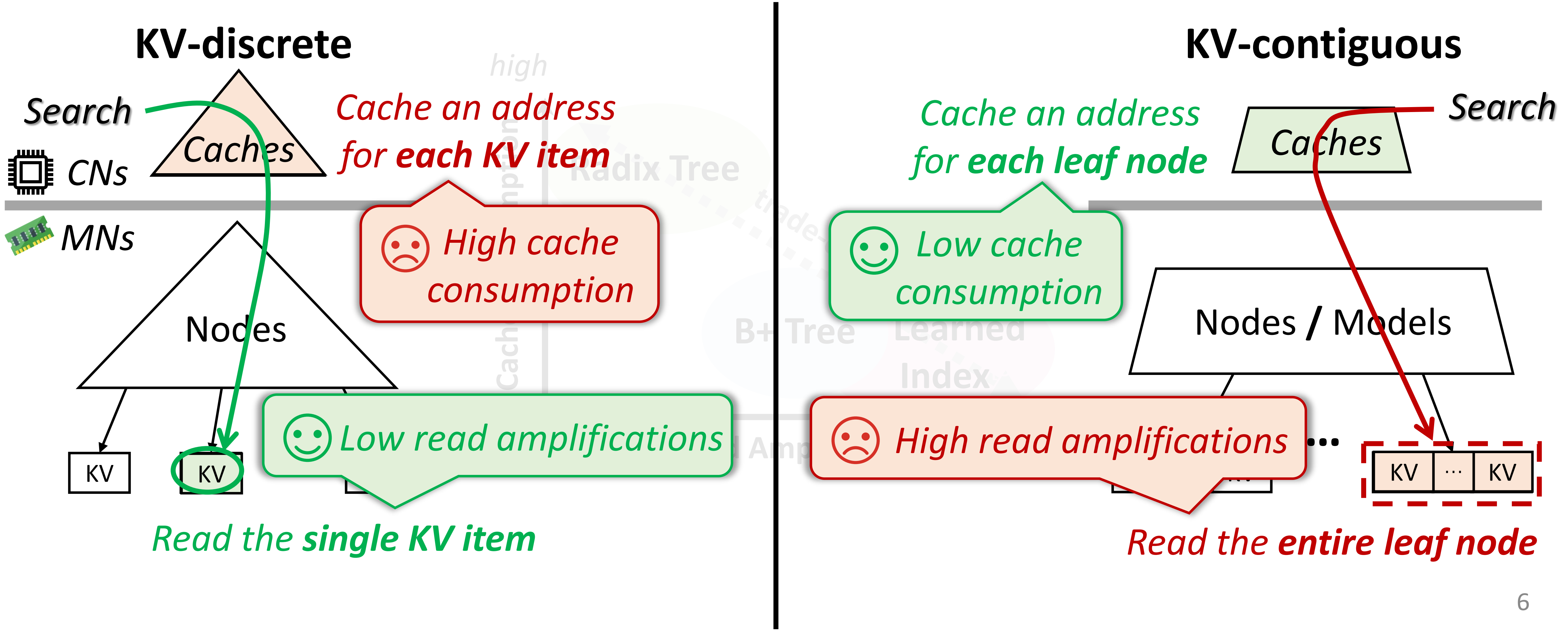






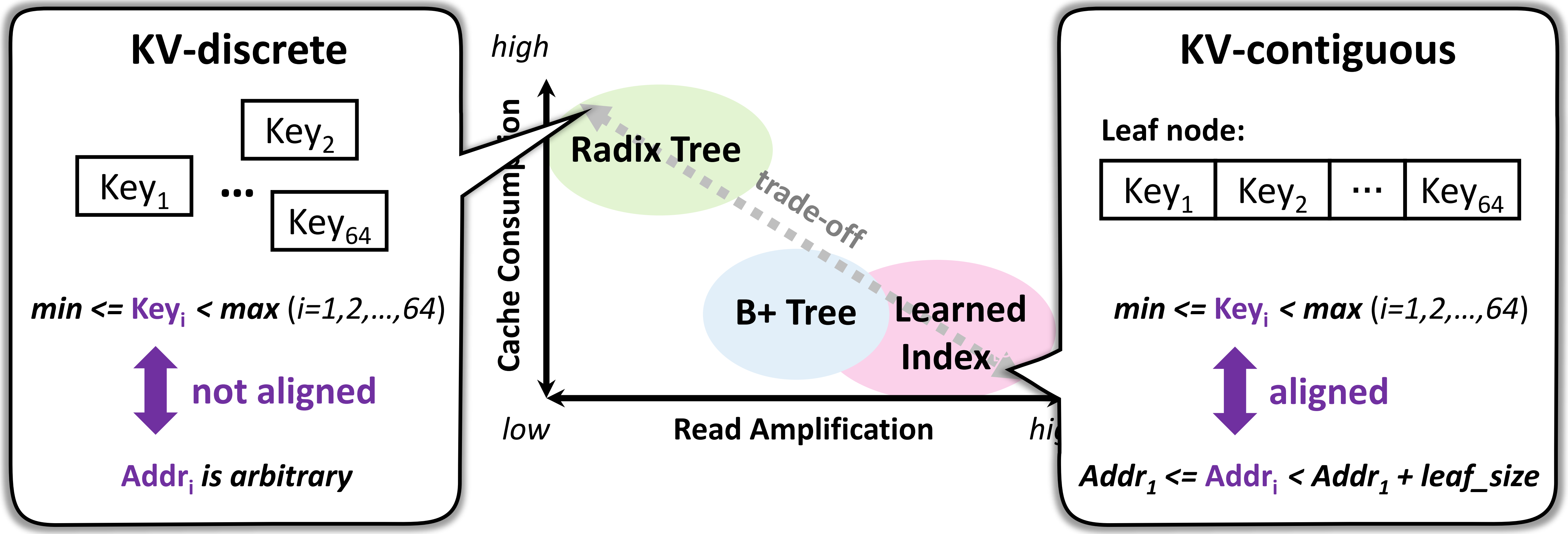
Motivation: A Trade-off for Range indexes on DM

There is a trade-off between read amplifications and cache consumption:



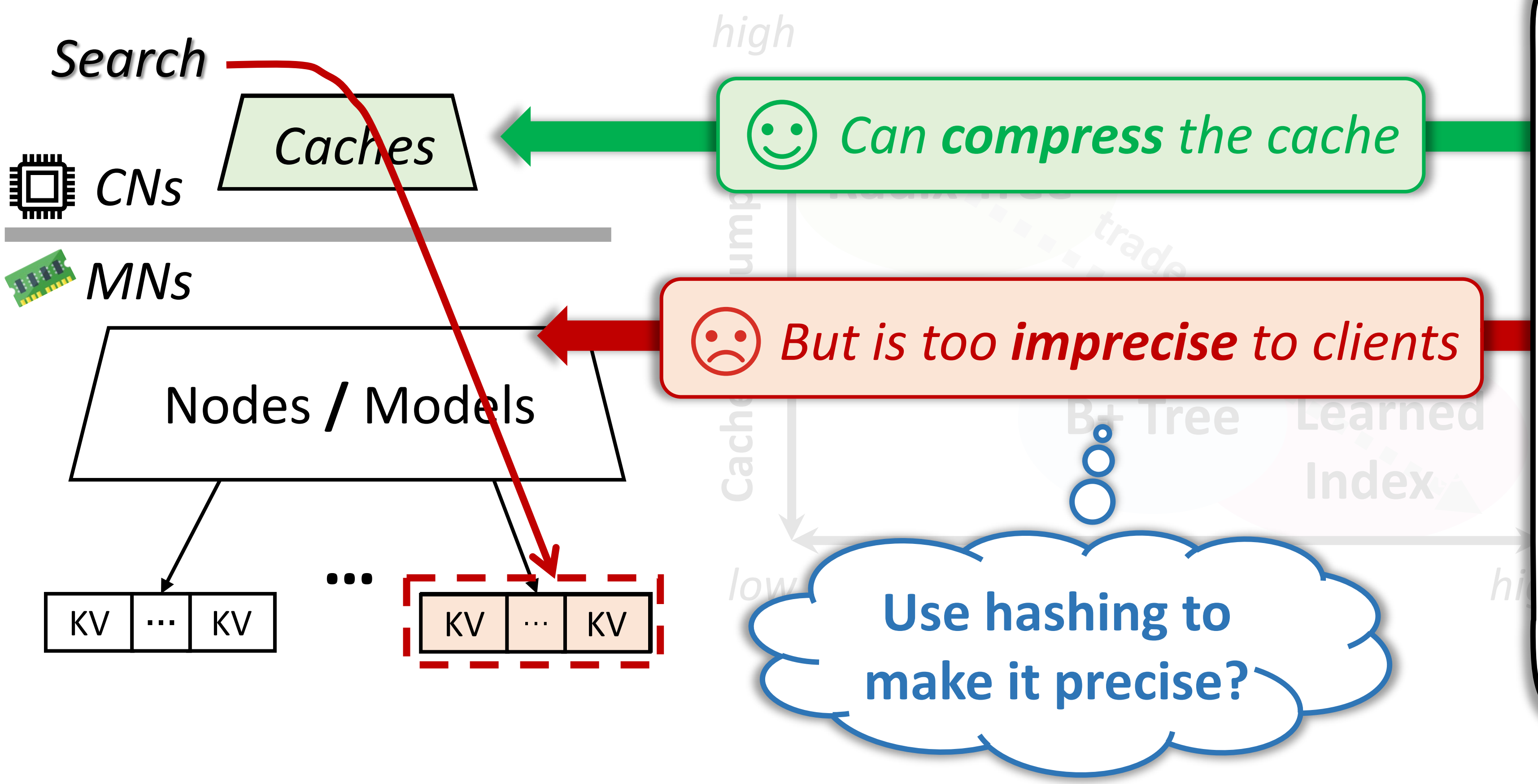
Motivation: A Trade-off for Range indexes on DM

Root Cause: The alignment between keys and memory addresses



Motivation: A Trade-off for Range indexes on DM

Root Cause: The alignment between keys and memory addresses



KV-contiguous

Leaf node:

Key_1	Key_2	···	Key_{64}

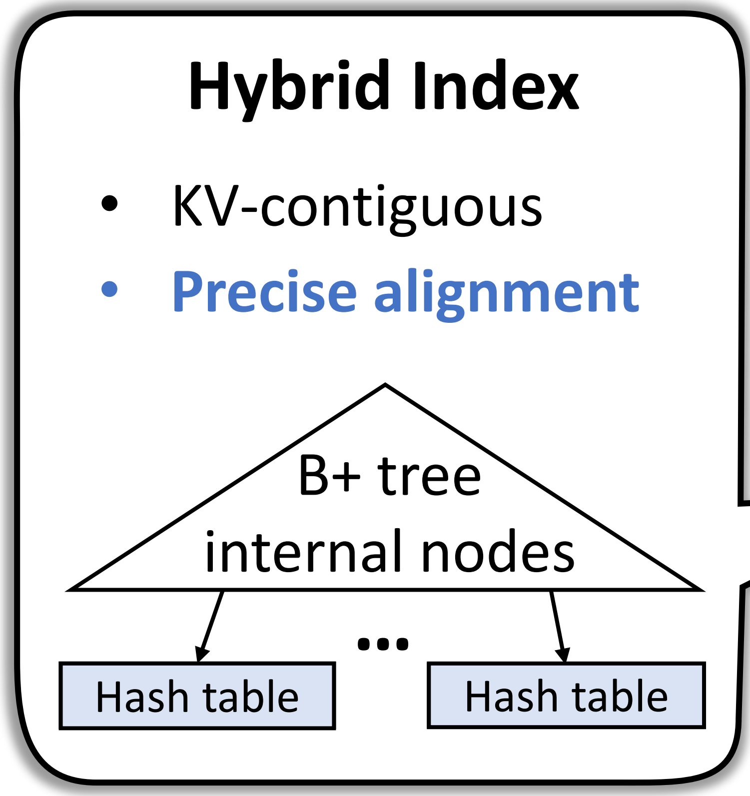
$min <= Key_i < max$ ($i=1,2,\ldots,64$)

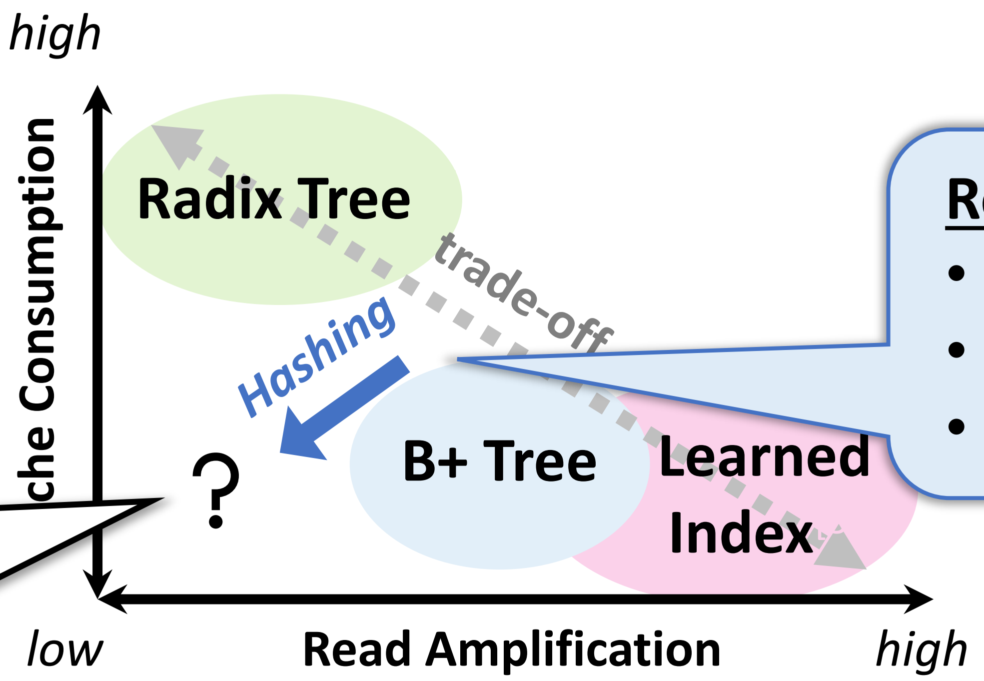
aligned

$Addr_1 <= Addr_i < Addr_1 + leaf\_size$

Straightforward Idea

Use a KV-contiguous index (*e.g., B+ tree*) with hash-table-based leaf nodes





Requirements:

- Few remote accesses
- Low read amplifications
- High space efficiency

Widely Choose a Suitable Hashing Scheme

Remote accesses

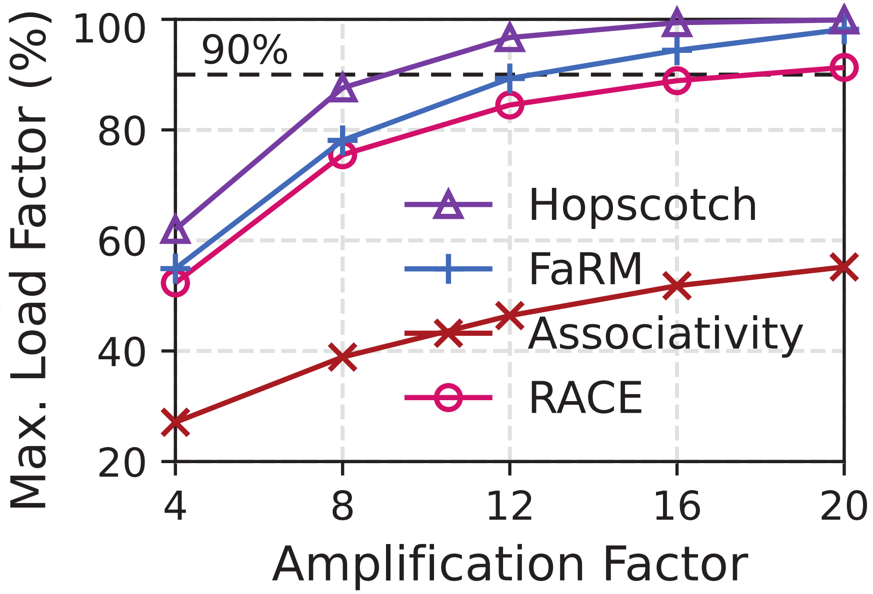
<u>*Many accesses:*</u> ✗

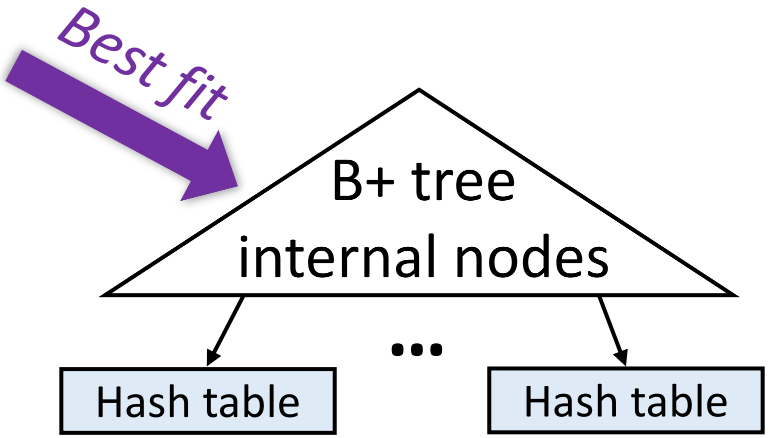
- Cuckoo hashing
- Clustering hashing
- …

<u>*Few accesses:*</u> ✓

- Simple associativity
- Hopscotch hashing[1]
- FaRM[2]
- RACE[3]

Read amplifications & Space efficiency





☺ *Hopscotch hashing best fits the requirements*

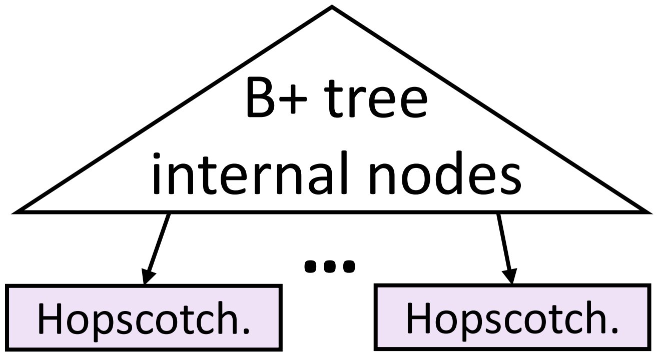
[1] Maurice Herlihy et al. Hopscotch hashing. DISC 2008.
[2] Aleksandar Dragojevic et al. FaRM: Fast Remote Memory. NSDI 2014.
[3] Pengfei Zuo et al. One-sided RDMA-conscious extendible hashing for disaggregated memory. ATC 2021.

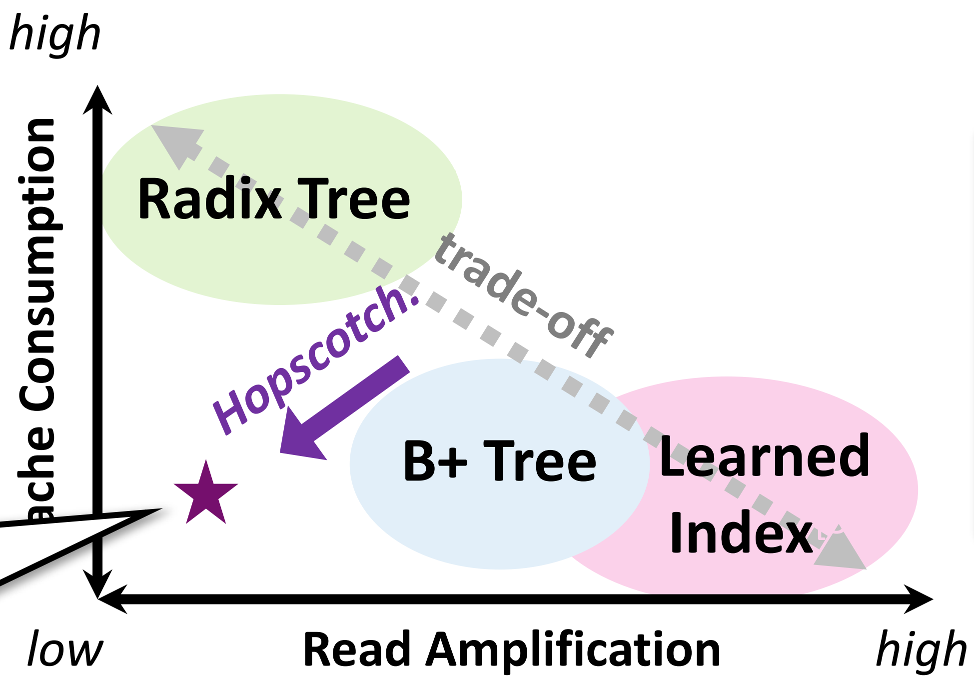
Viable Idea

Use a hybrid index combining a B+ tree with hopscotch hashing

Hybrid Index

- KV-contiguous
- **Precise alignment**
- **High load factor**



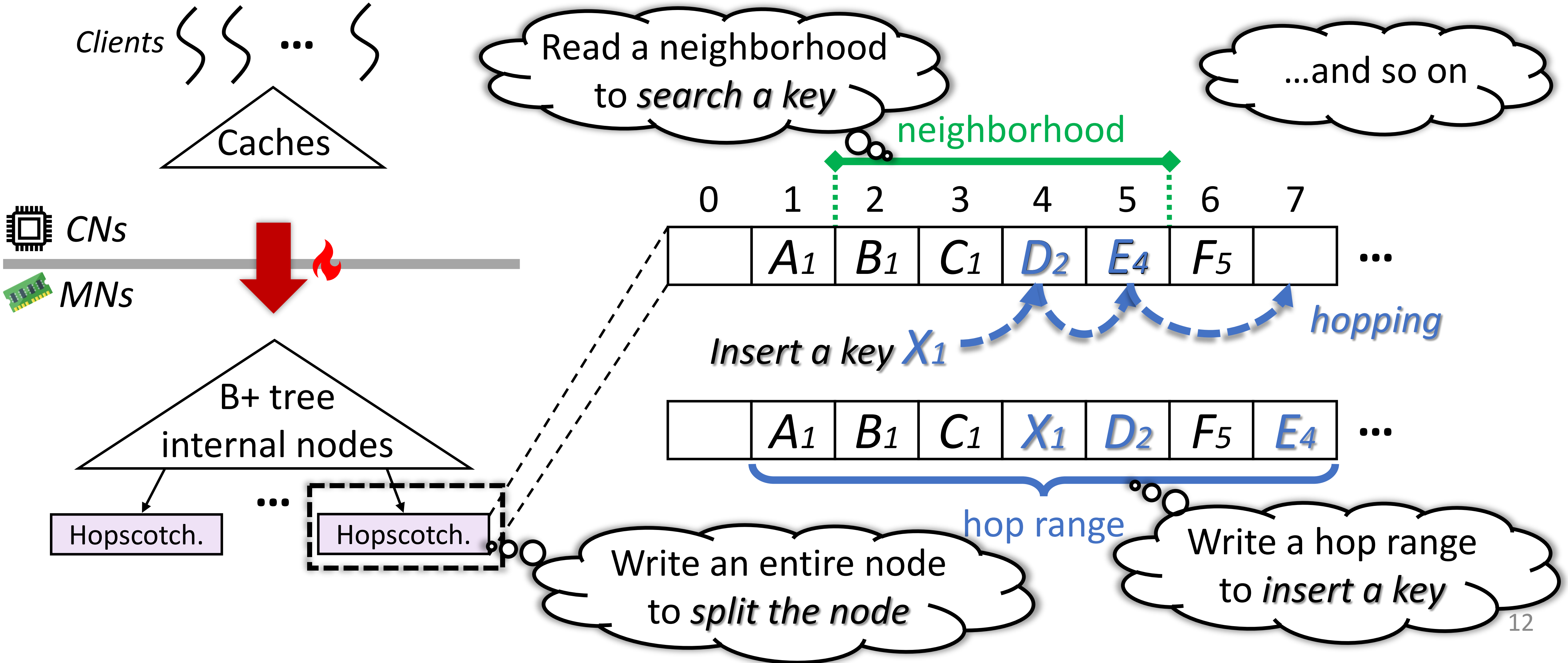


☺ **Benefit:**

✓ Cache-efficient
✓ Bandwidth-efficient
✓ Space-efficient

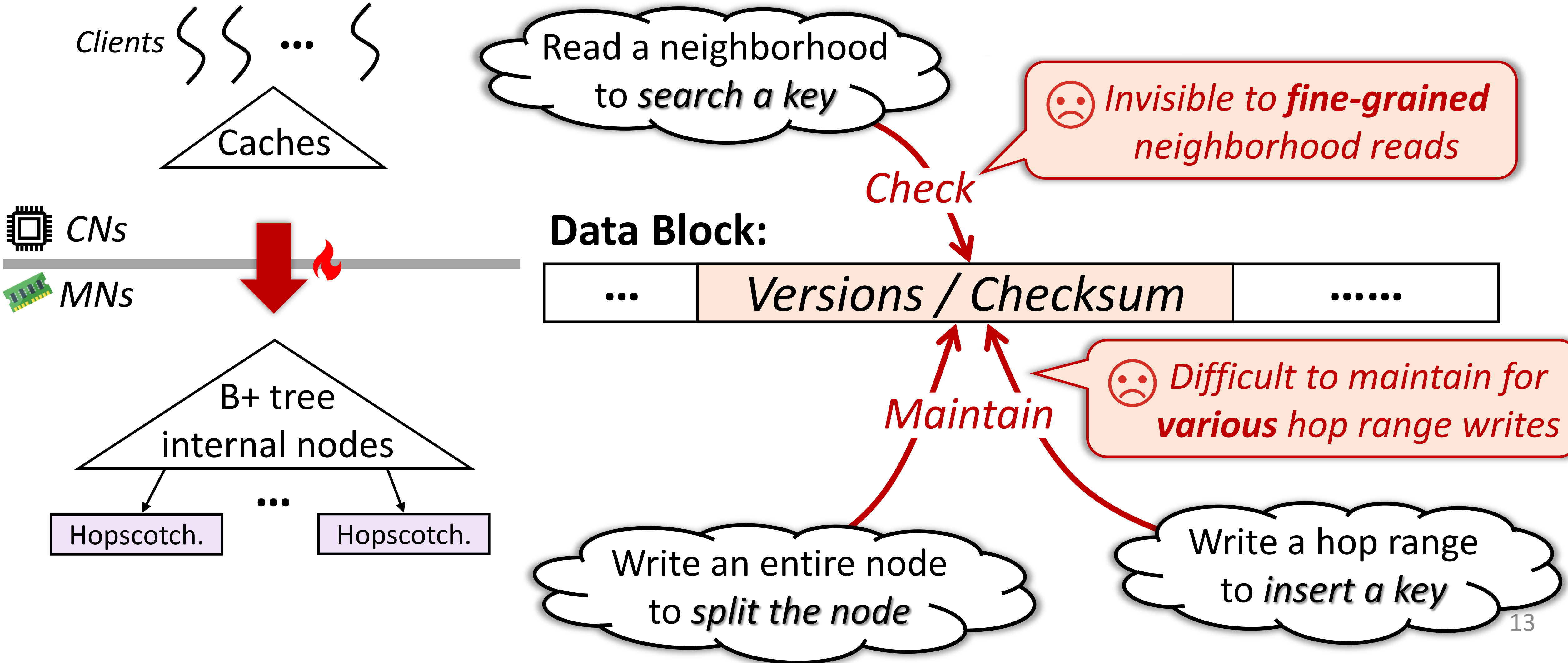
Challenge 1: Complicated Optimistic Synchronization

Various granularities in reads/writes complicate the optimistic synchronization



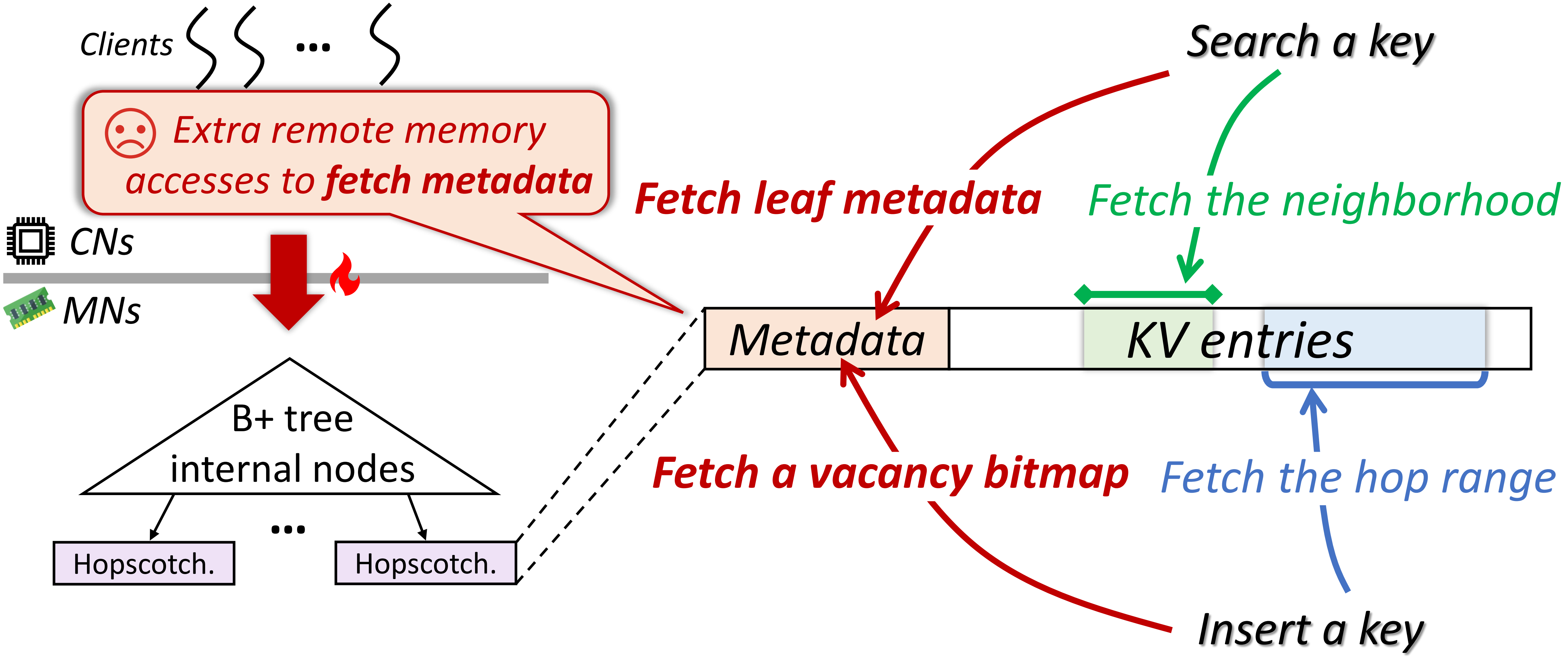
Challenge 1: Complicated Optimistic Synchronization

Various granularities in reads/writes complicate the optimistic synchronization



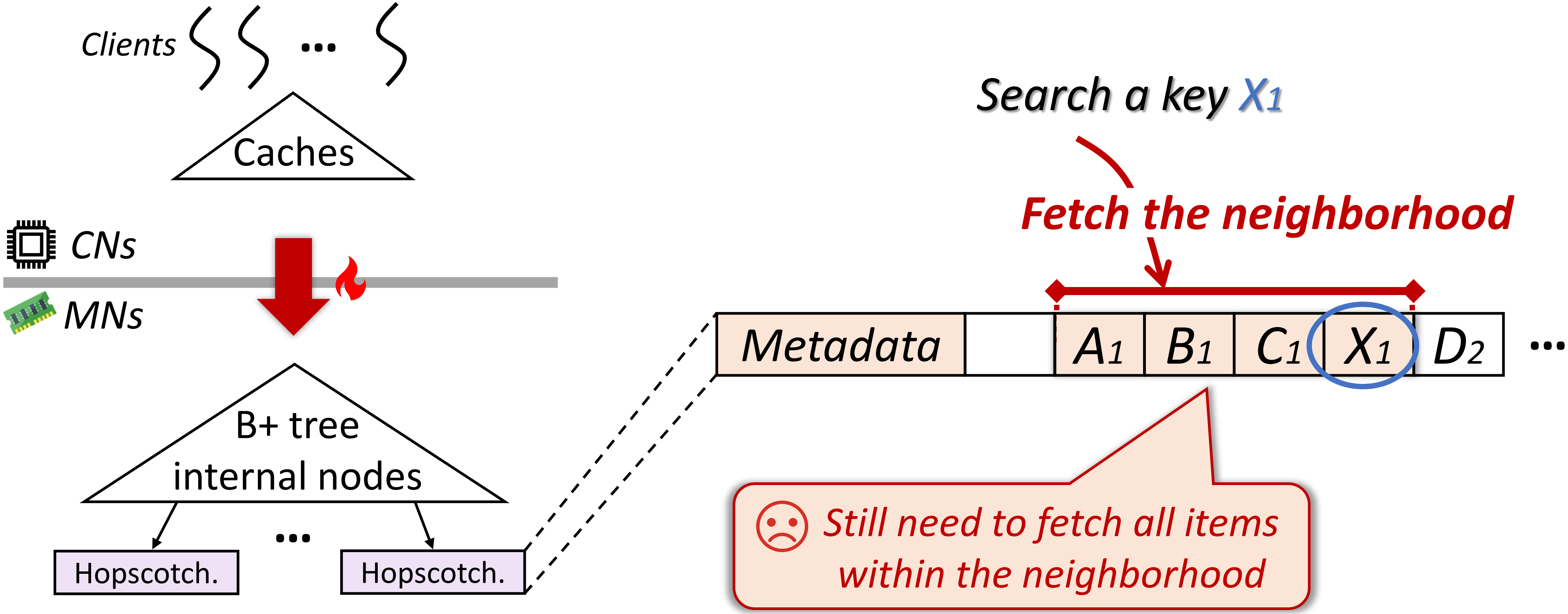
Challenge 2: Extra Metadata Accesses

Metadata for B+ trees and hopscotch hashing induces extra remote accesses

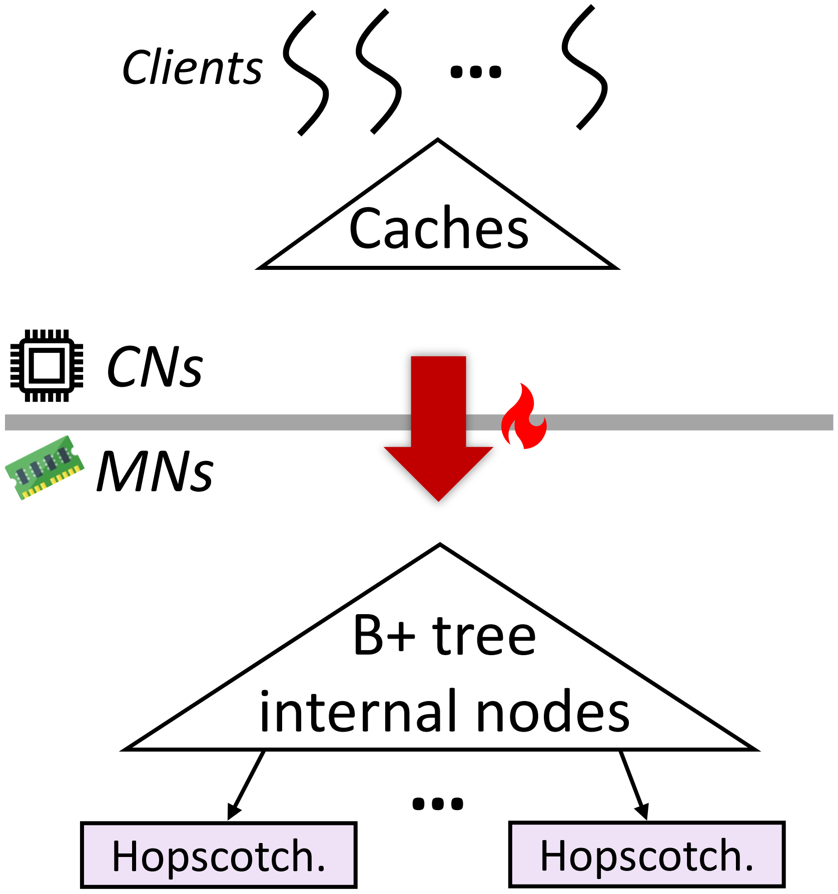


Challenge 3: Read Amplifications of Hopscotch Hashing

Hopscotch hashing still incurs read amplifications compared with reading a KV



Challenge Summary

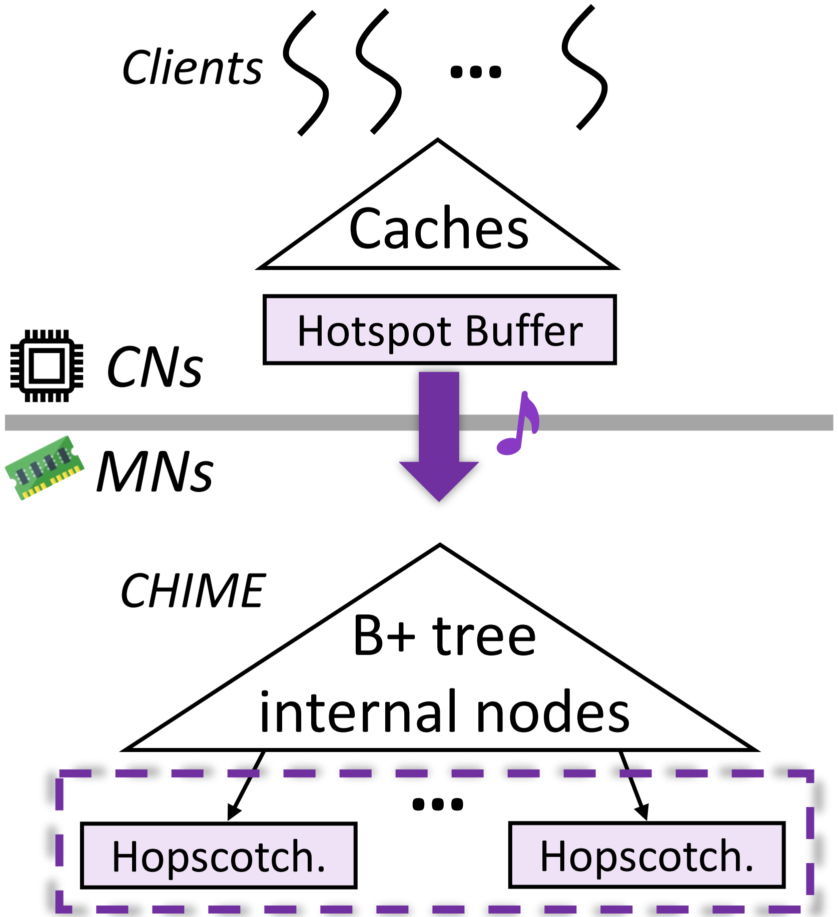


1. Complicated Optimistic Synchronization

2. Extra Metadata Accesses

3. Read Amplifications of Hopscotch Hashing

The CHIME Design



1. Complicated Optimistic Synchronization

Solution 1: *Three-Level Optimistic Synchronization*

2. Extra Metadata Accesses

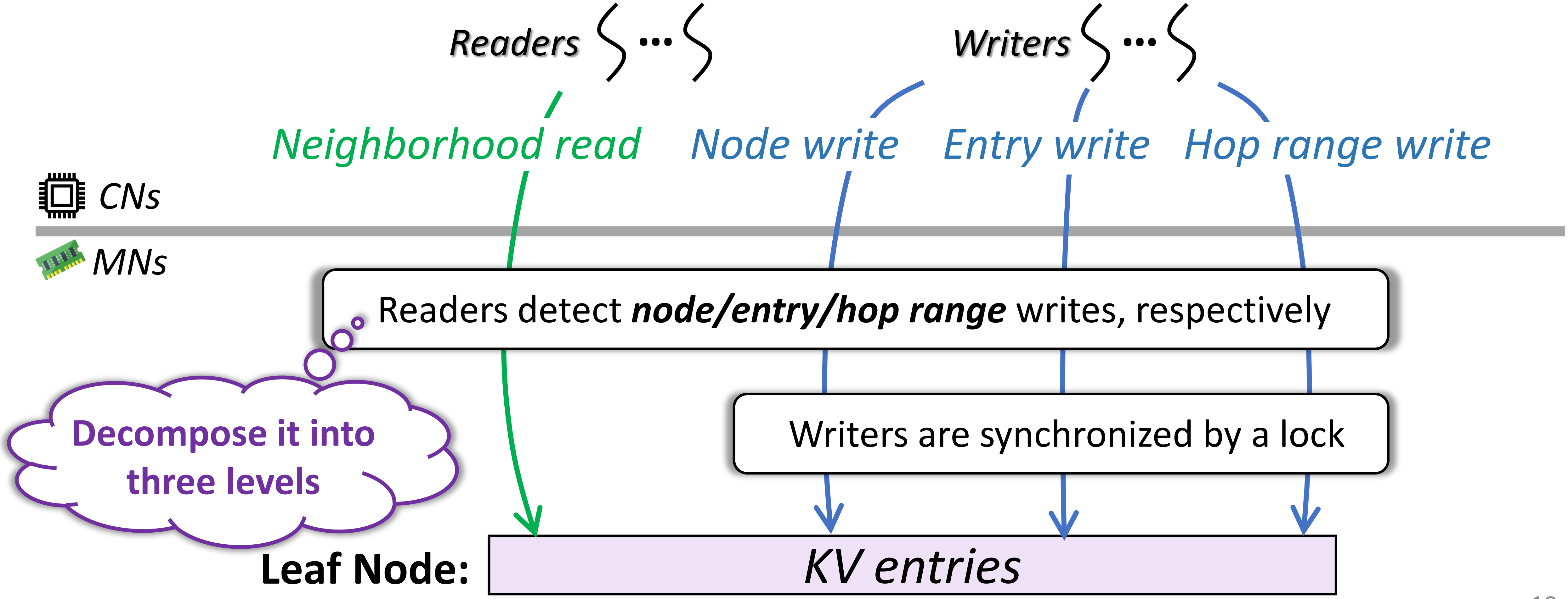
Solution 2: *Access-Aggregated Metadata Management*

3. Read Amplifications of Hopscotch Hashing

Solution 3: *Hotness-Aware Speculative Read*

Three-level Optimistic Synchronization

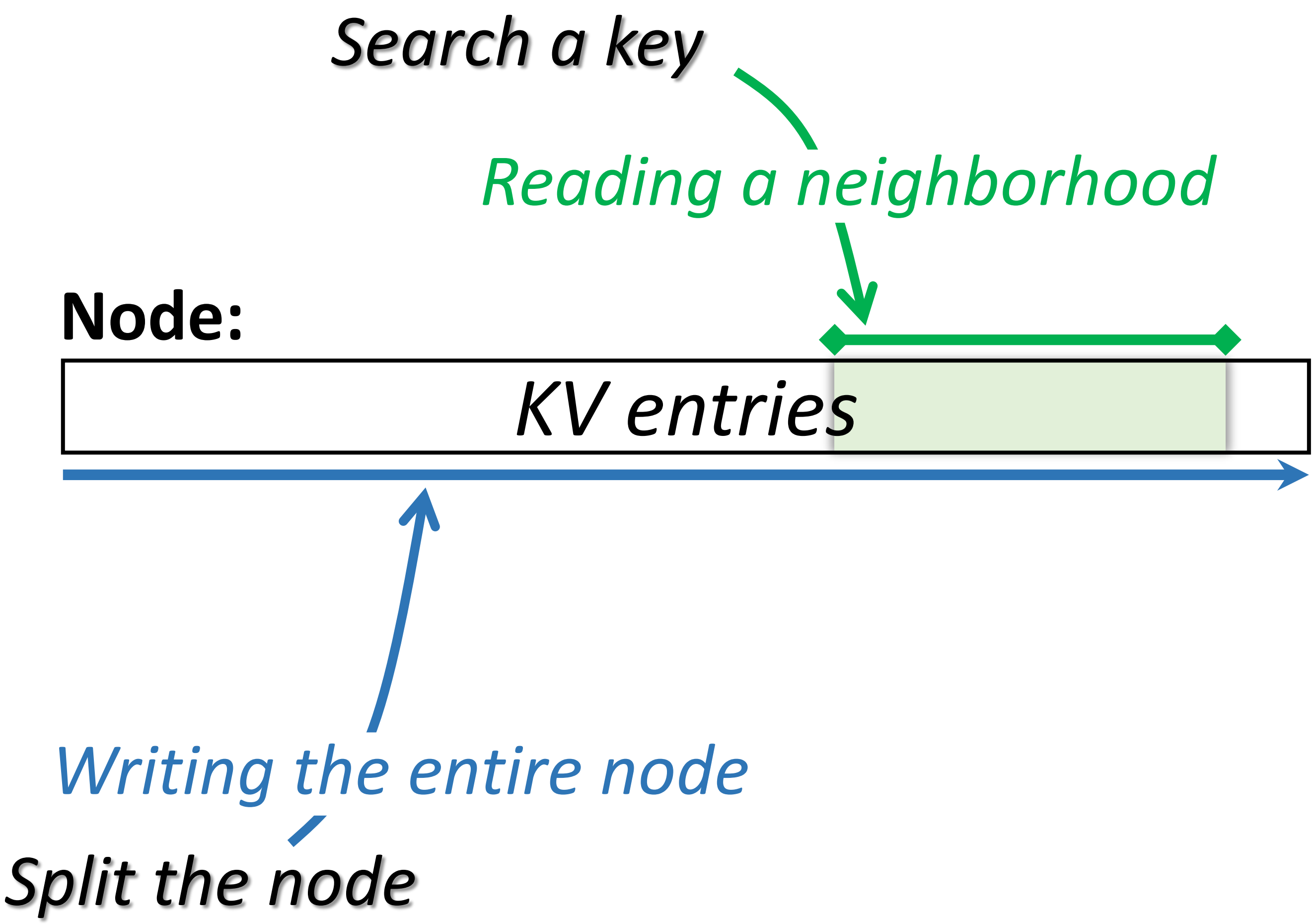
Synchronization Overview



Three-level Optimistic Synchronization

Level 1: Detect the *node write*

<u>**Problem**</u>: Readers cannot perceive the node write

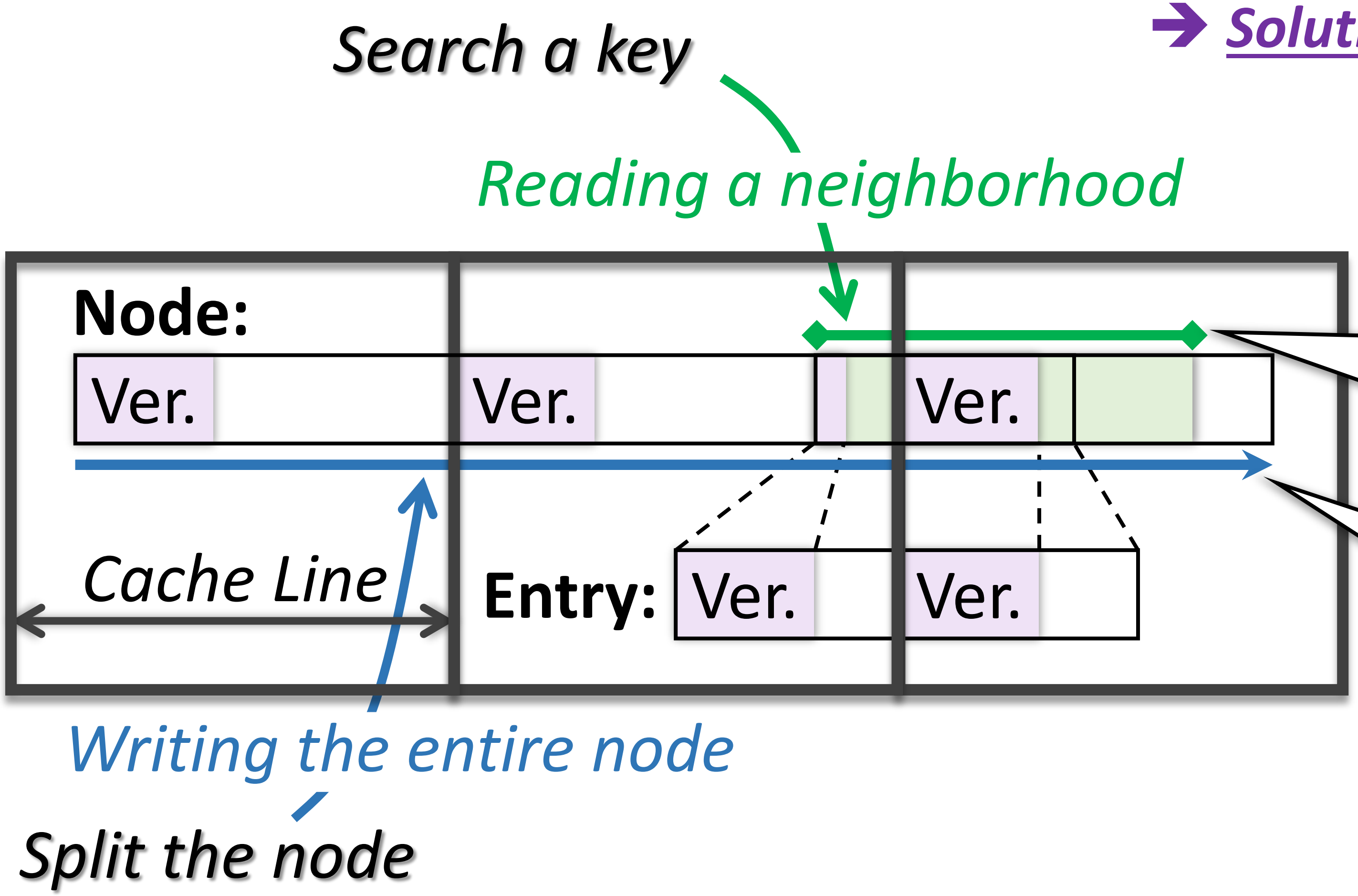


Three-level Optimistic Synchronization

Level 1: Detect the *node write*

Problem: Readers cannot perceive the node write

➔ ***Solution**: Use cache line versioning*[1]



Readers:

- Check only **fetched** versions
- Retry if they cannot match

Writers:

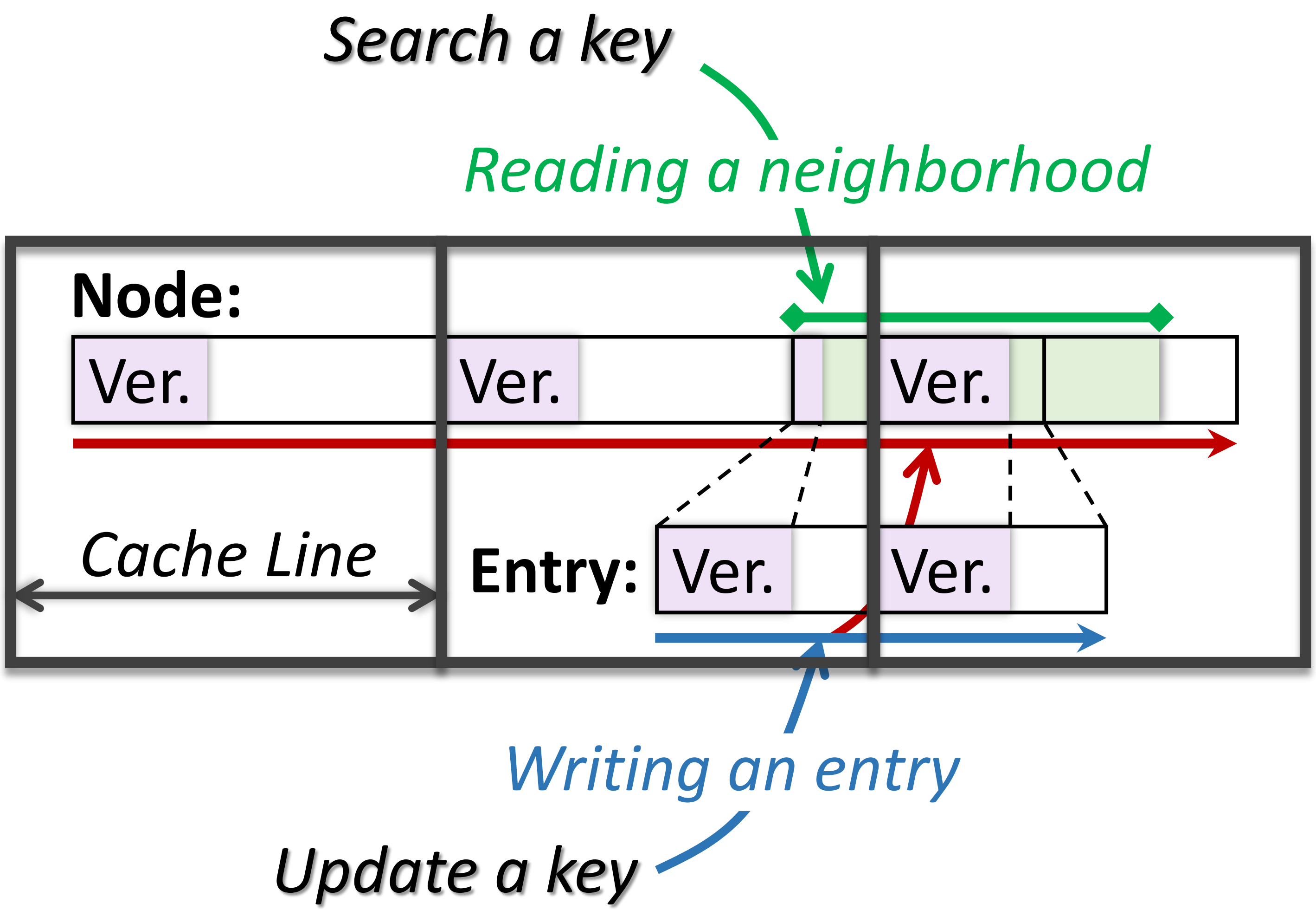
- Increment **all** versions in the node

[1] Aleksandar Dragojevic et al. FaRM: Fast Remote Memory. NSDI 2014.

Three-level Optimistic Synchronization

Level 2: Detect the *entry write*

Problem: Writers have to update all versions in the node

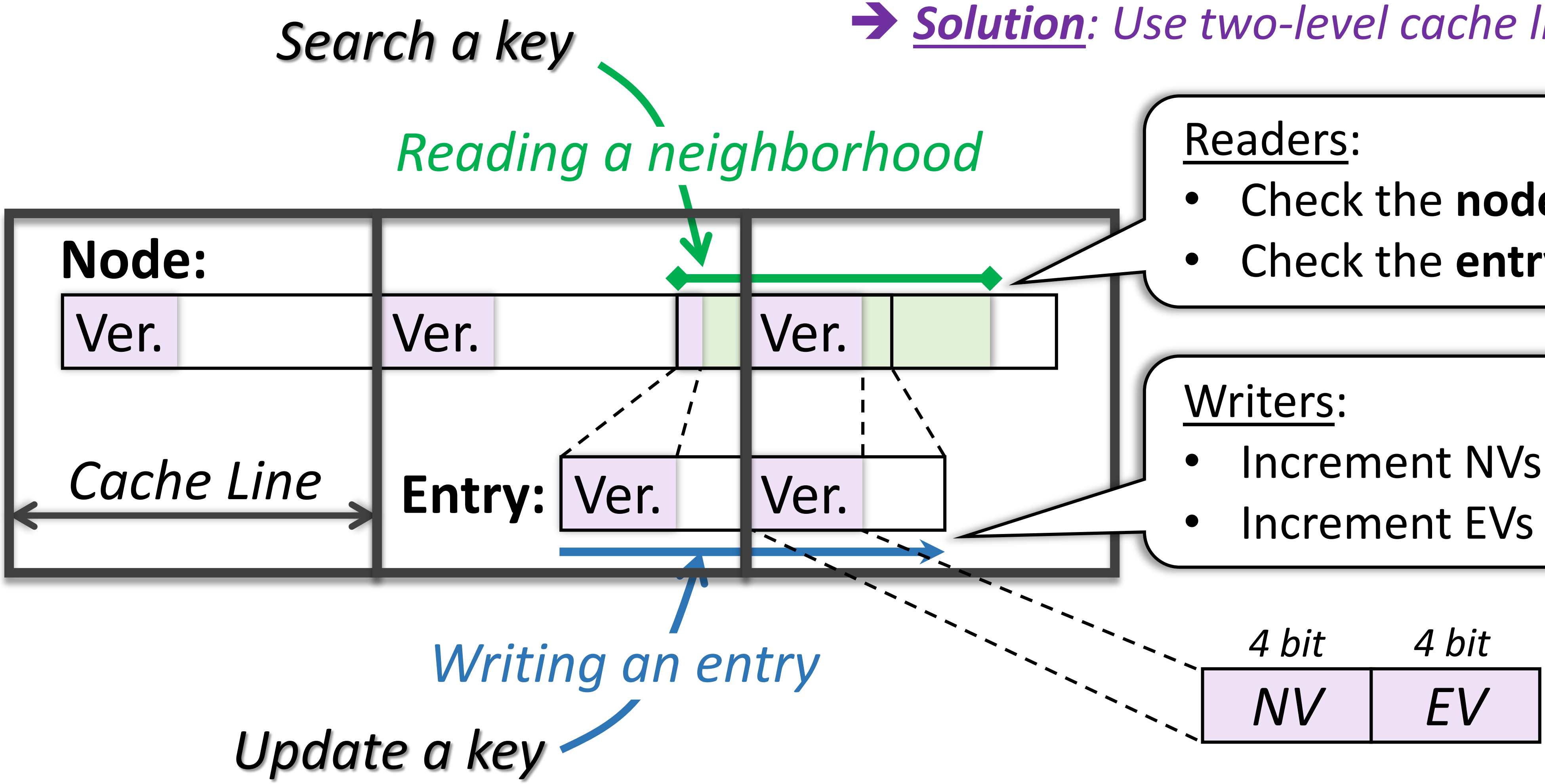


Three-level Optimistic Synchronization

Level 2: Detect the *entry write*

Problem: Writers have to update all versions in the node

➔ ***Solution**: Use two-level cache line versioning*



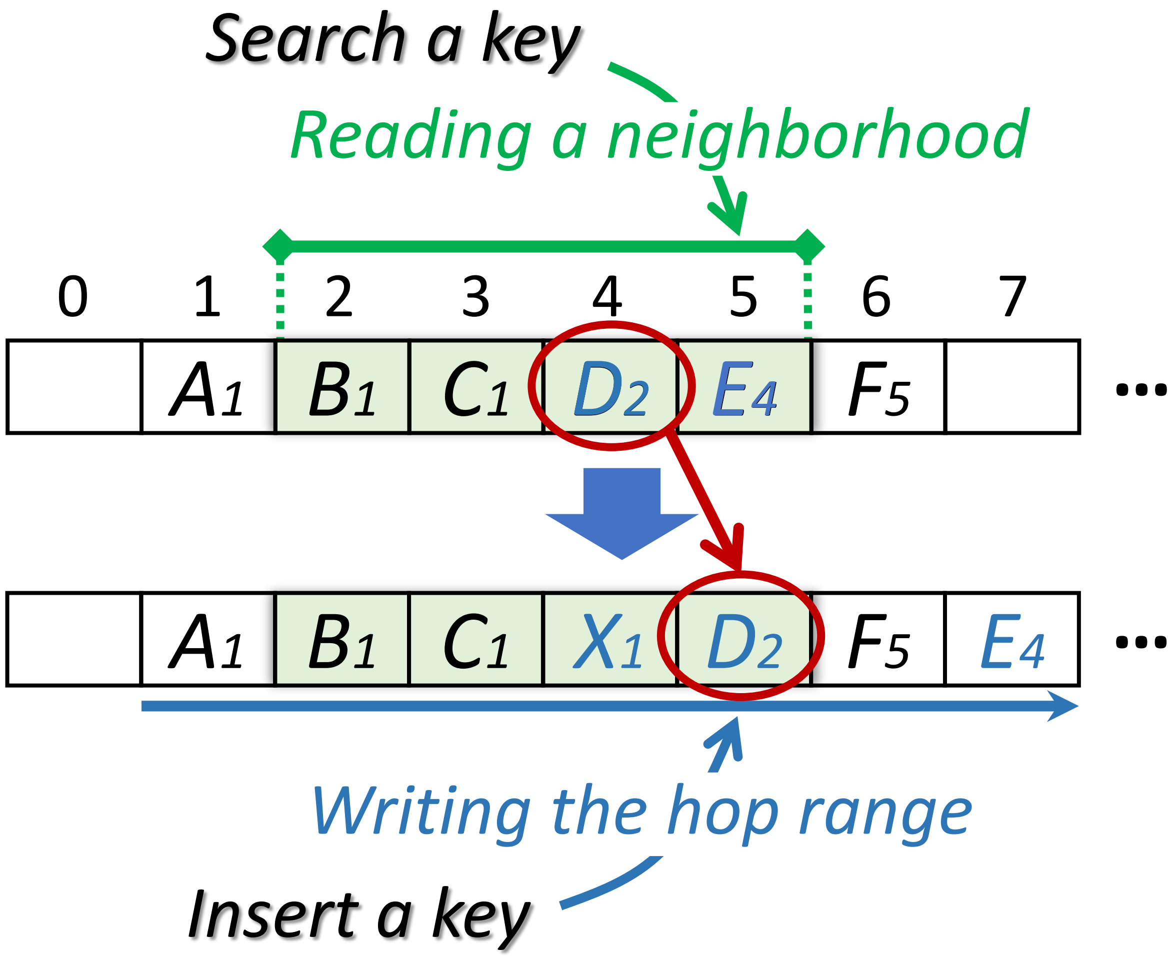
Readers:
- Check the **node write** via NVs
- Check the **entry write** via EVs

Writers:
- Increment NVs during a **node write**
- Increment EVs during an **entry write**

Three-level Optimistic Synchronization

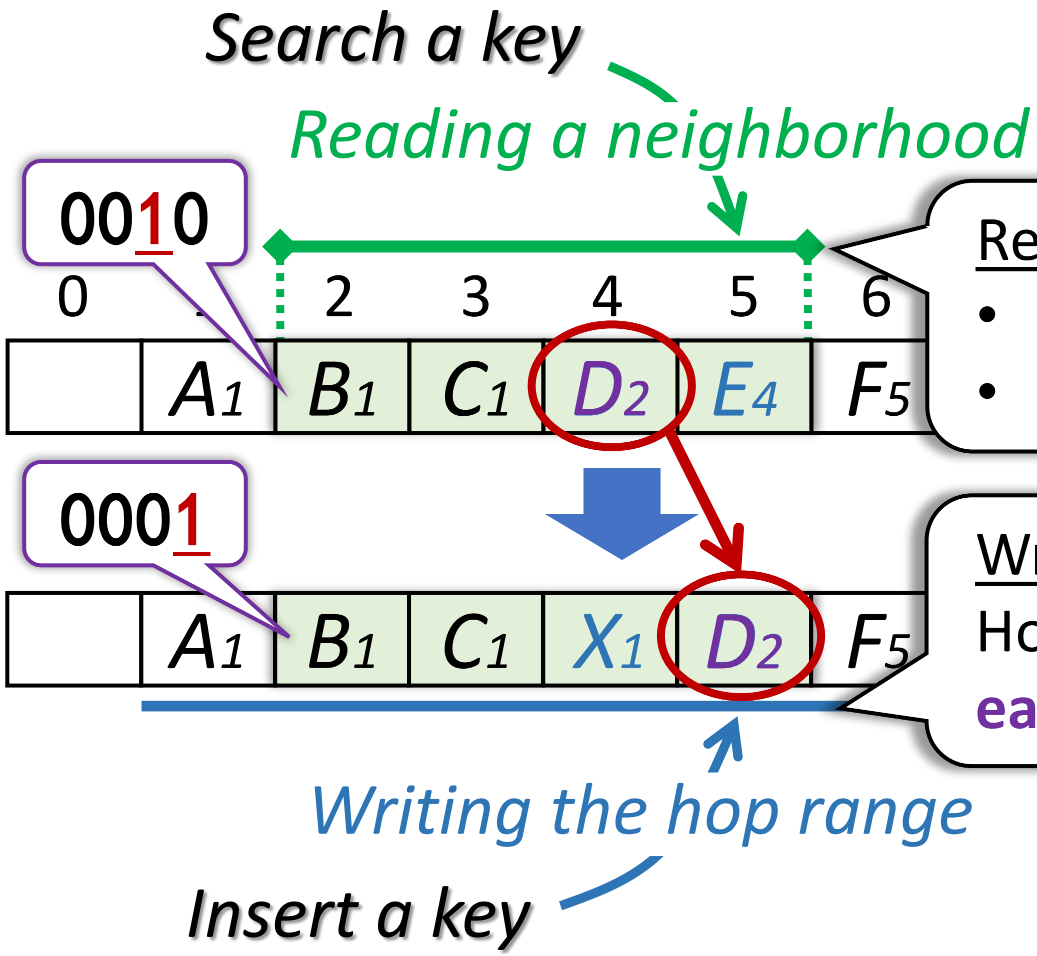
Level 3: Detect the *hop range write*

Problem: Location changes of hopped items



Three-level Optimistic Synchronization

Level 3: Detect the *hop range write*



Problem: Location changes of hopped items

➔ ***Solution**: Reuse the hopscotch bitmaps*

Readers:

- **Re-construct** the bitmap according to fetched keys
- Retry if the two bitmaps cannot match

Writers:

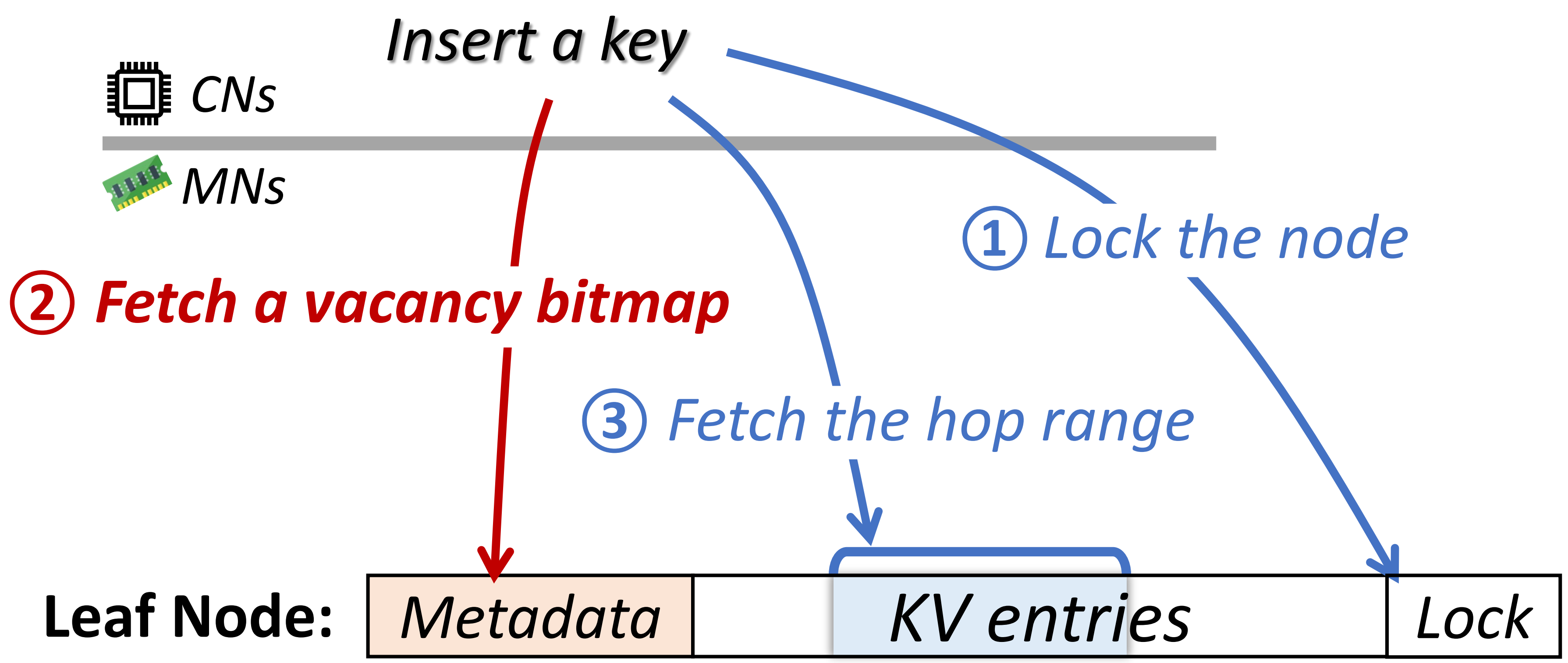
Hopscotch hashing **has maintained a bitmap inside each neighborhood** to track the occupancy status[1]

[1] Maurice Herlihy et al. Hopscotch hashing. DISC 2008.

Access-Aggregated Metadata Management

Metadata for hopscotch hashing

Problem: Vacancy bitmaps induce extra accesses



Access-Aggregated Metadata Management

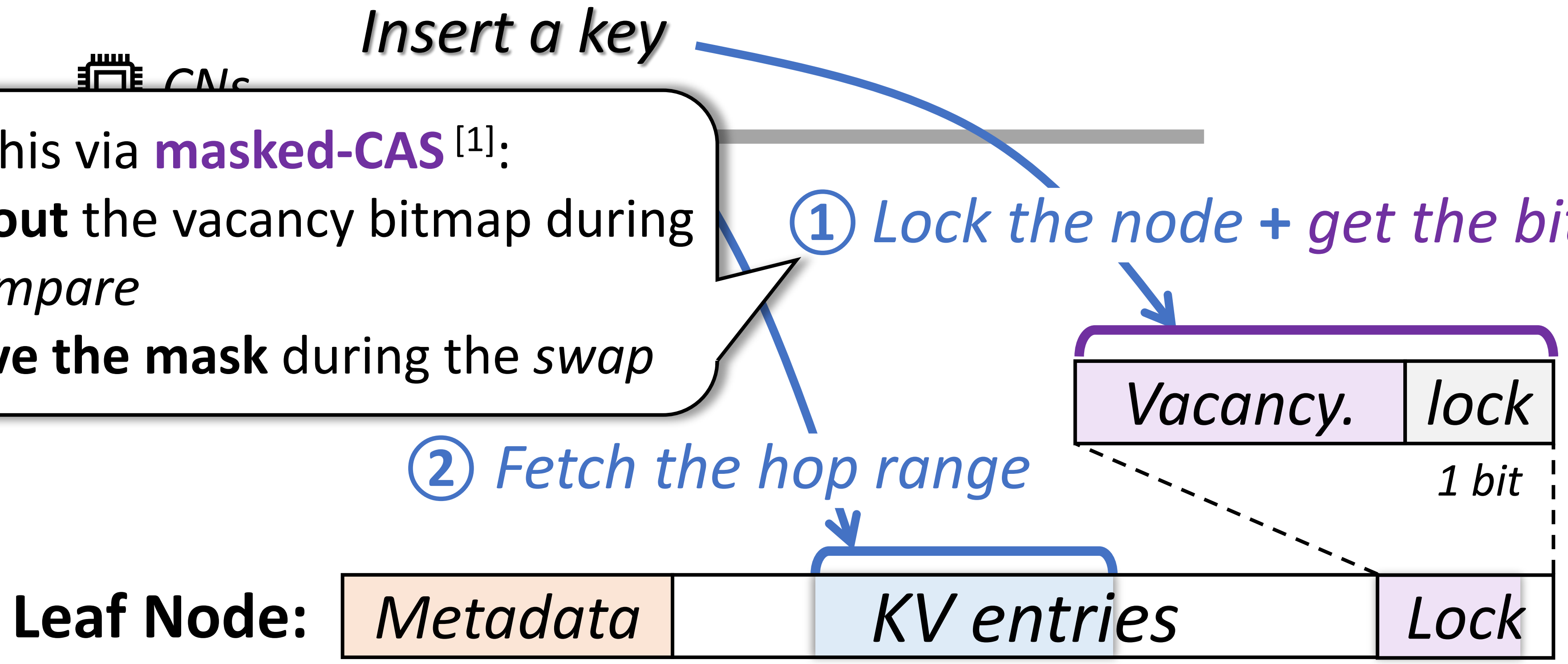
Metadata for hopscotch hashing

Problem: Vacancy bitmaps induce extra accesses

➔ ***Solution**: Piggyback the vacancy bitmap*

Achieve this via **masked-CAS** [1]:

- **Mask out** the vacancy bitmap during the *compare*
- **Remove the mask** during the *swap*

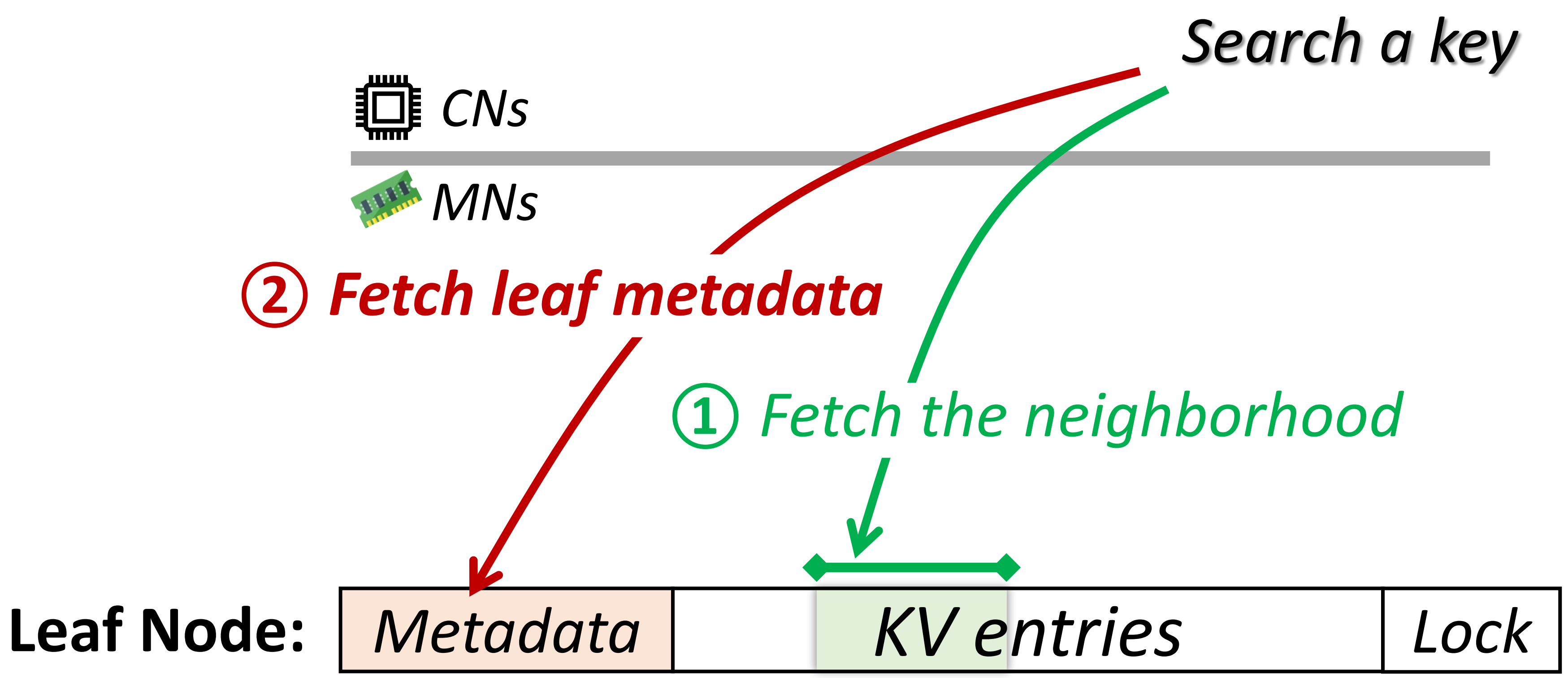


[1] NVIDIA Corporation. RDMA Aware Networks Programming User Manual v1.7.

Access-Aggregated Metadata Management

Metadata for the B+ tree

Problem: Leaf metadata induce extra accesses

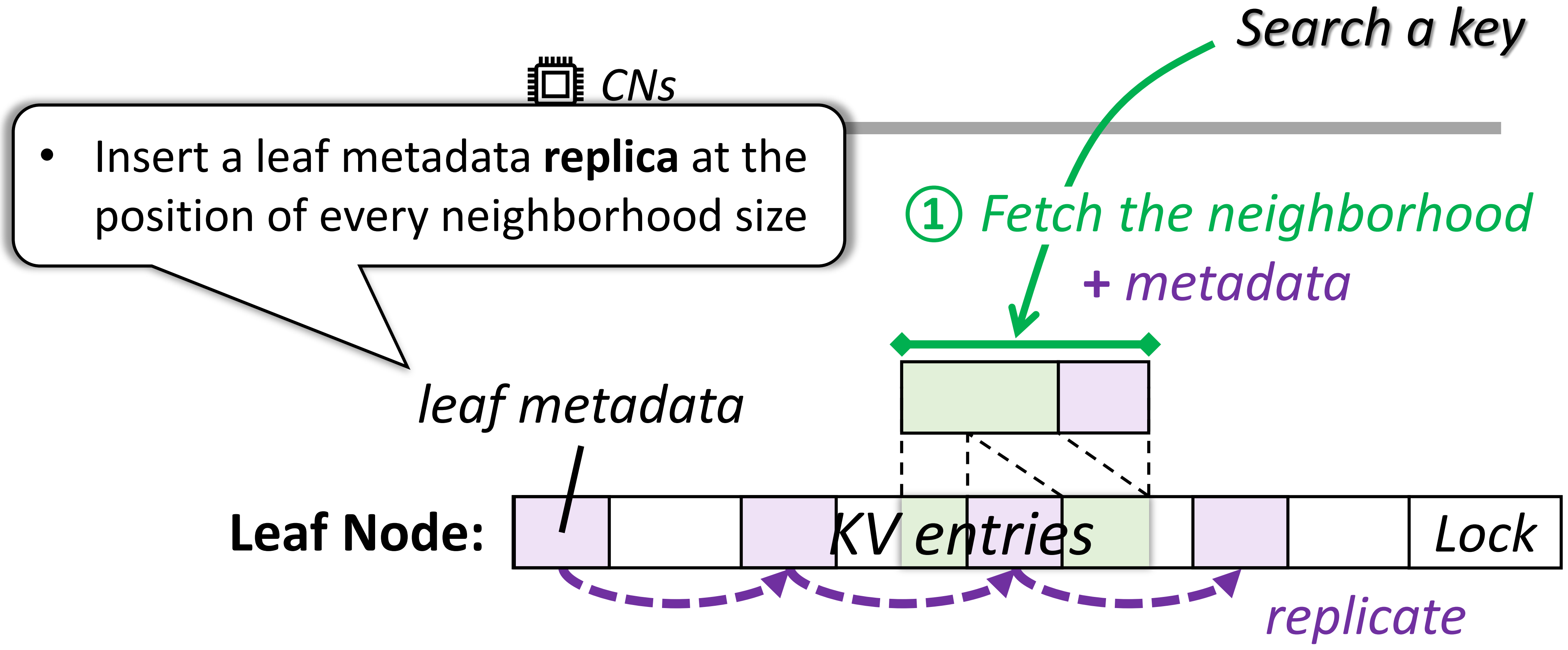


Access-Aggregated Metadata Management

Metadata for the B+ tree

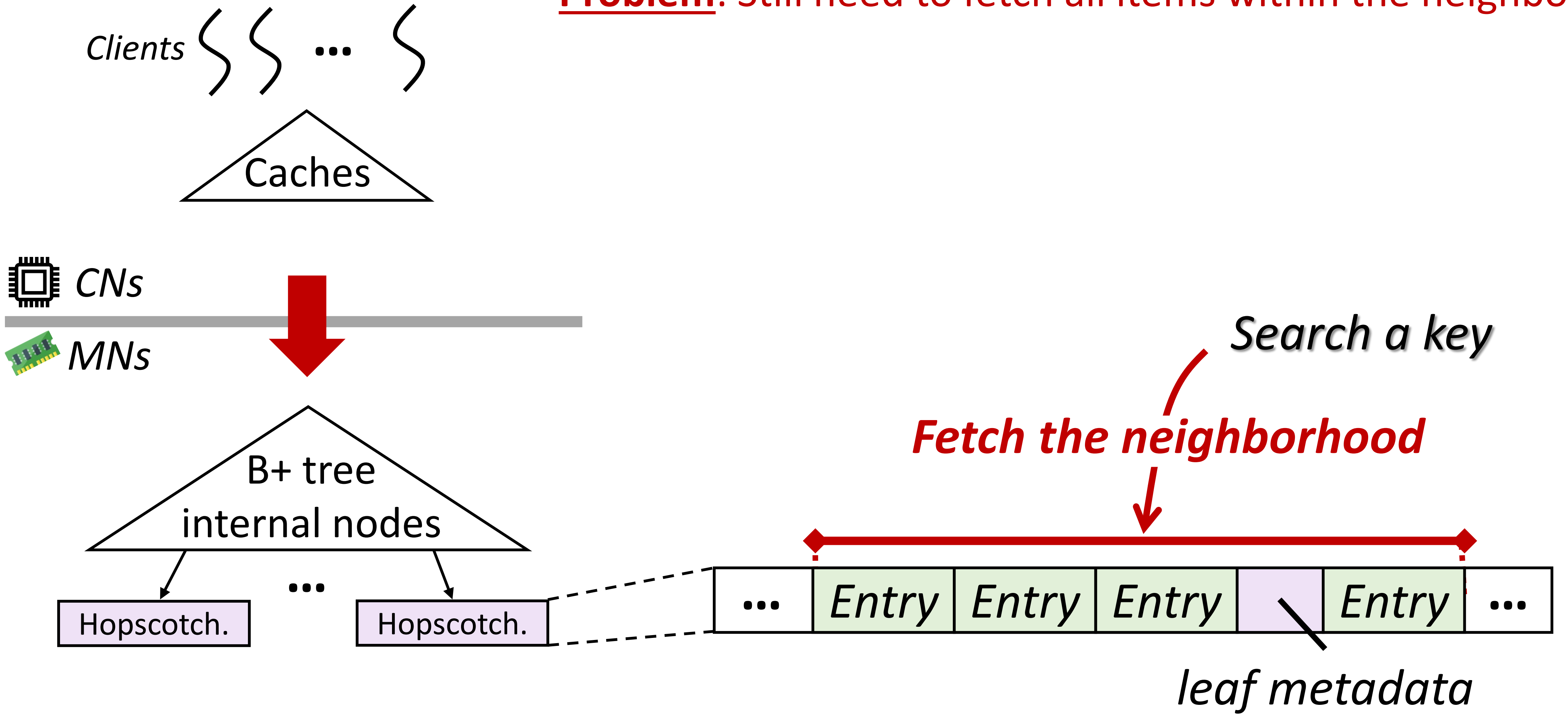
Problem: Leaf metadata induce extra accesses

➔ ***Solution***: *Replicate the leaf metadata*



Hotness-Aware Speculative Read

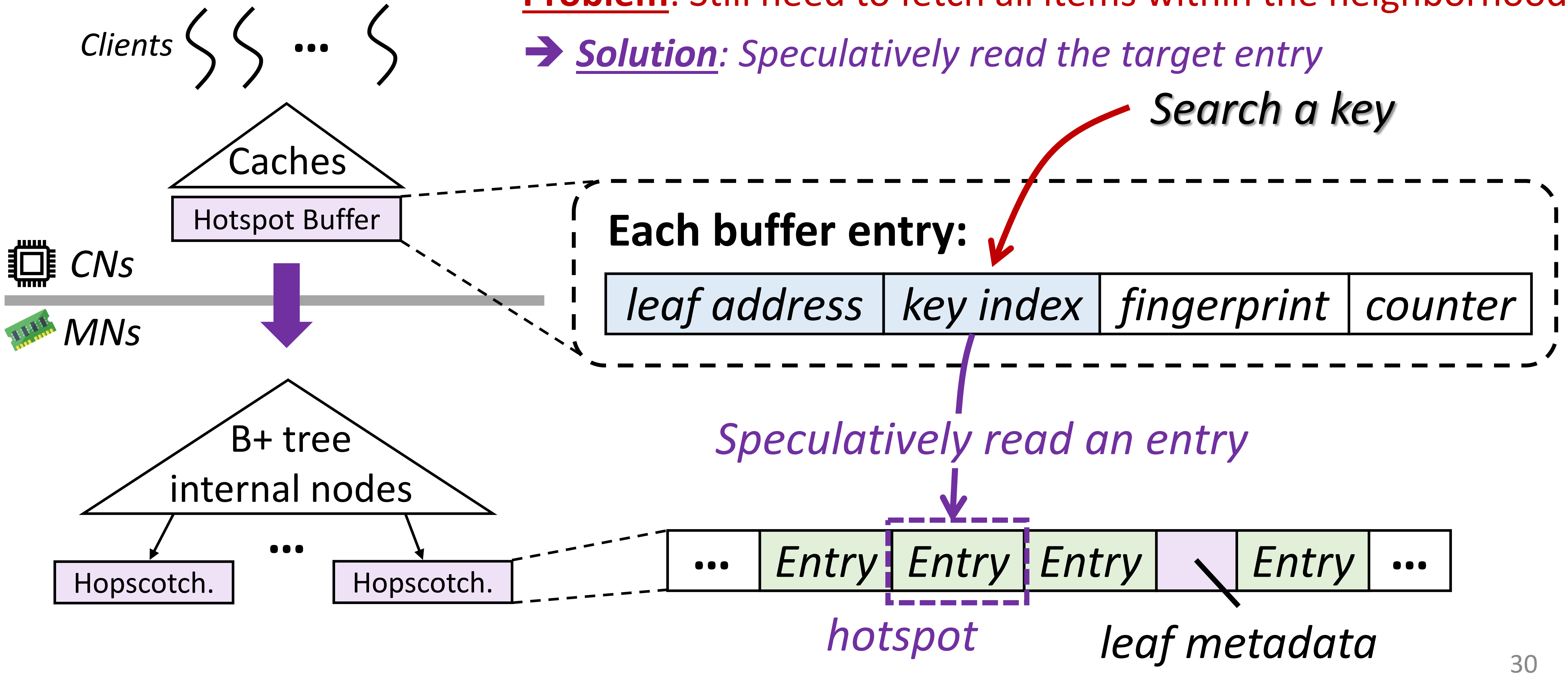
Problem: Still need to fetch all items within the neighborhood



Hotness-Aware Speculative Read

Problem: Still need to fetch all items within the neighborhood

➔ ***Solution**: Speculatively read the target entry*



More Details

- ✓ Sibling-based validation
- ✓ Support for variable-sized keys and values
- ✓ Applicability to the learned Index
- ✓ Detailed operations
- ✓ ……

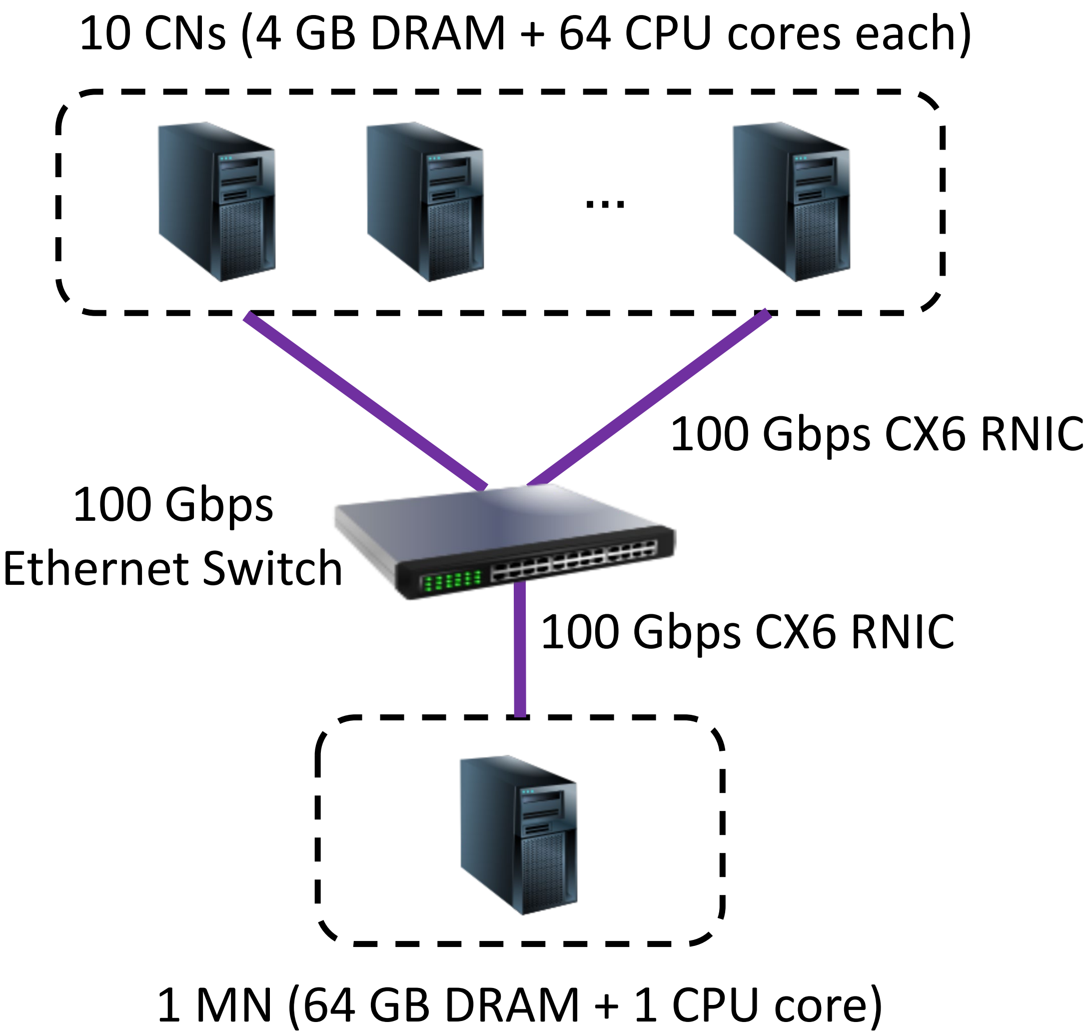
Evaluation

Workloads and Parameters

- YCSB workloads
- 8-byte keys and 8-byte values
- Limit the cache size to 100 MB per CN

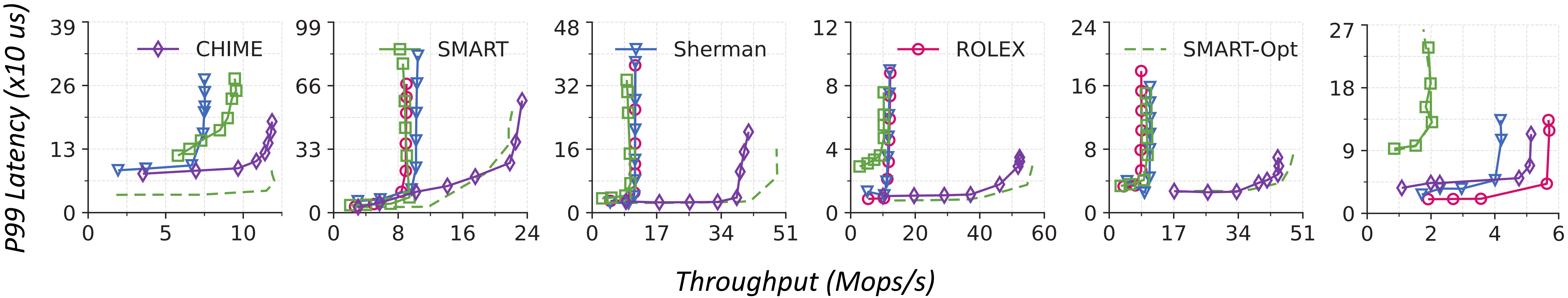
Comparisons

- SMART *[OSDI’23]*
  - The latest radix tree design on DM
- Sherman *[SIGMOD’22]*
  - The classic B+ tree design on DM
- ROLEX *[FAST’23]*
  - The latest learned index on DM
- SMART-Opt *(Optimal case)*
  - SMART with sufficient caches



Performance Comparison

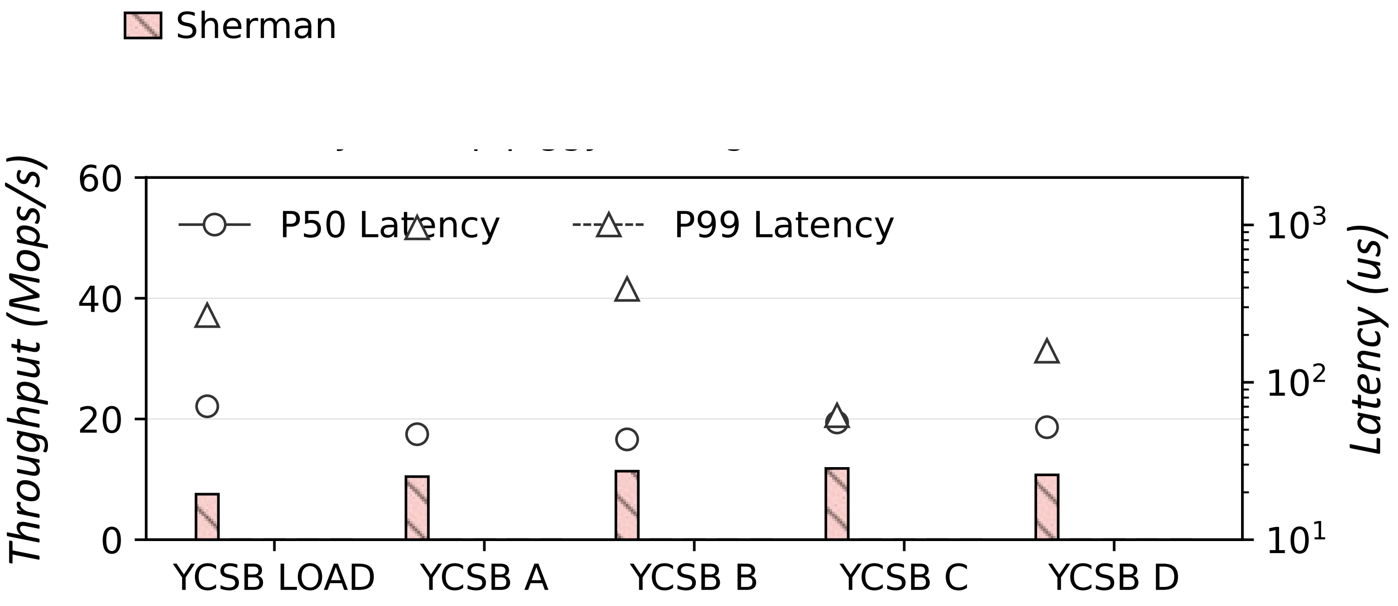
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D	YCSB E
100% insert	50% read 50% update	95% read 5% update	100% read	95% read 5% insert	95% scan 5% insert



- CHIME achieves:
  - Up to 4.3x higher throughput than Sherman and ROLEX
  - Up to 5.1x higher throughput than SMART
  - A close performance to the optimal case, with up to 8.7x lower cache consumption ***(57.6 MB vs. 503.6 MB)***

Factor Analysis

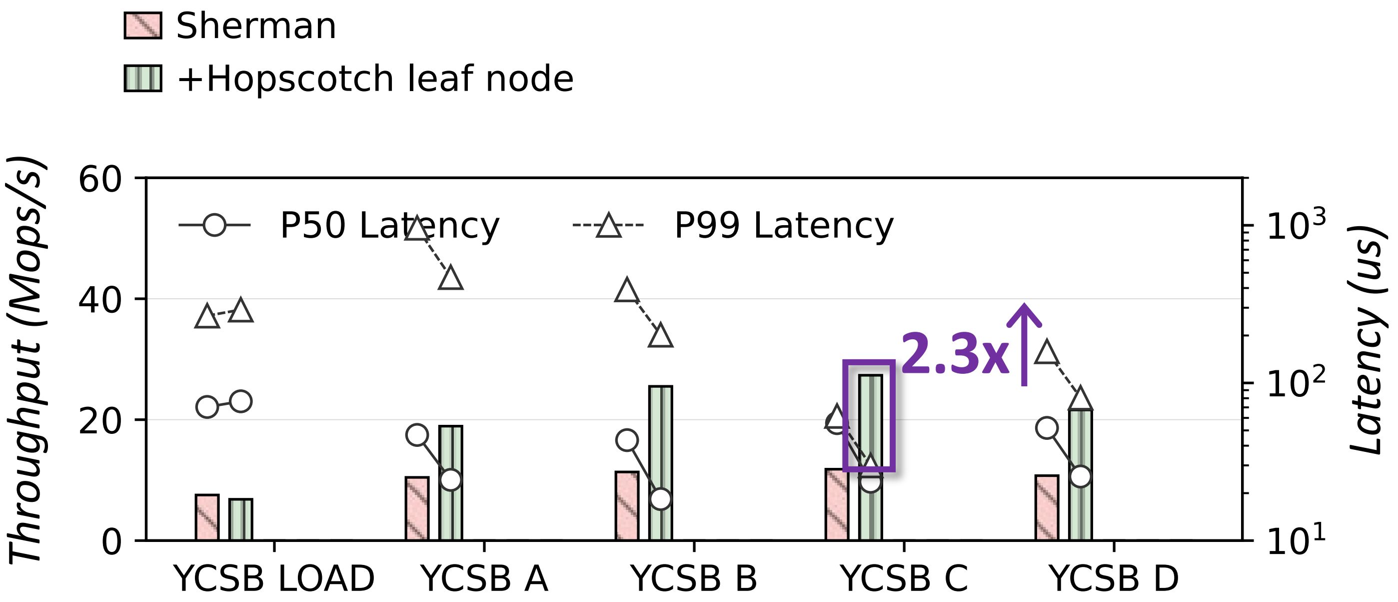
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read 50% update	95% read 5% update	100% read	95% read 5% insert



- Start with Sherman and apply each proposed technique one by one

Factor Analysis

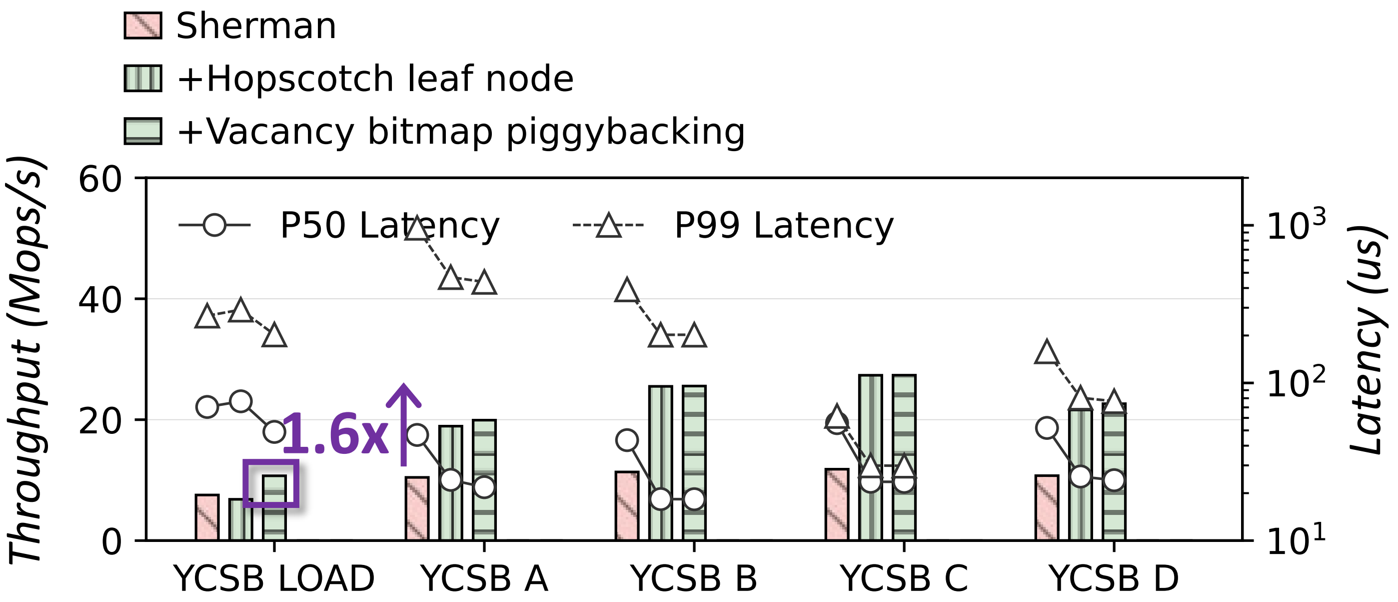
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read 50% update	95% read 5% update	100% read	95% read 5% insert



- The *hopscotch leaf node* enables fetching the neighborhood rather than the entire leaf node

Factor Analysis

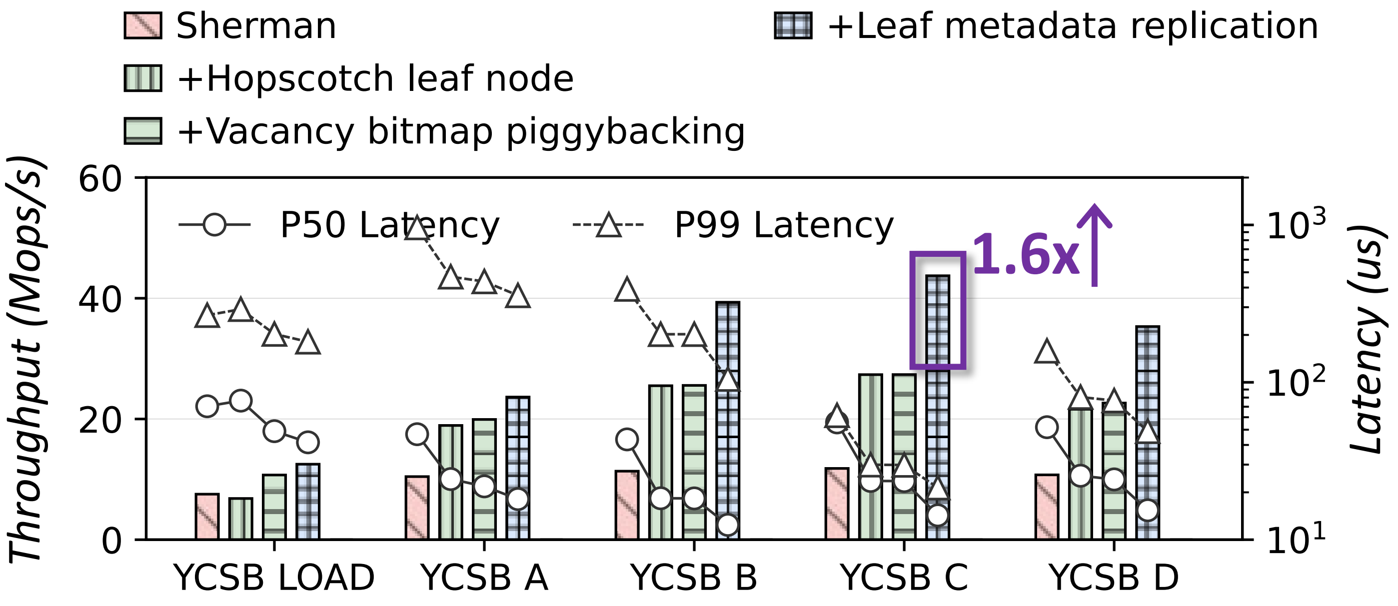
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read 50% update	95% read 5% update	100% read	95% read 5% insert



- The *vacancy bitmap piggybacking* enables fetching the hop range rather than the entire leaf node, without inducing extra remote accesses

Factor Analysis

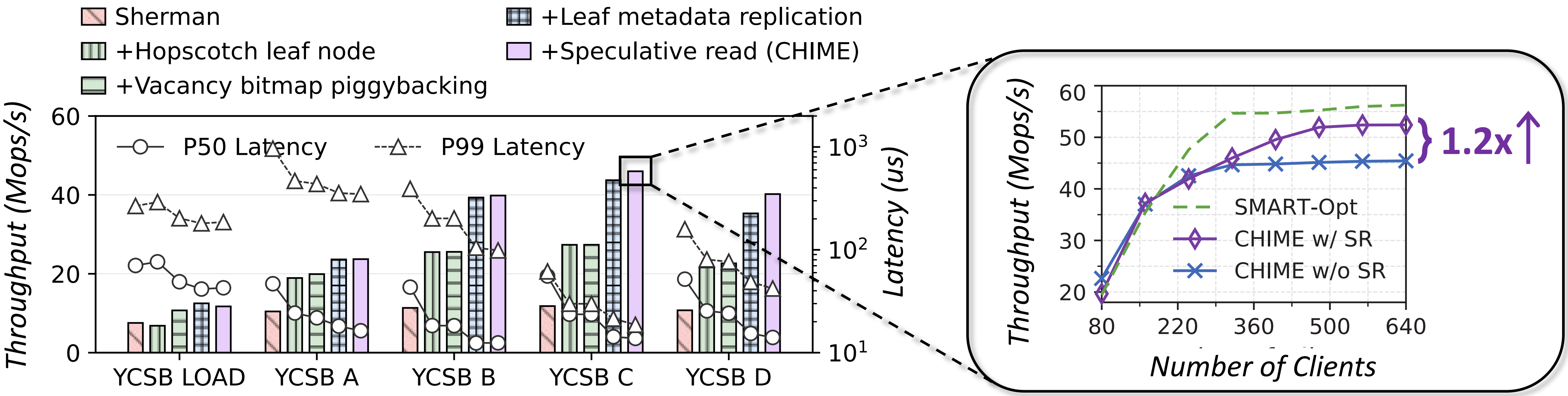
YCSB LOAD	YCSB A	YCSB B	YCSB C	YCSB D
100% insert	50% read 50% update	95% read 5% update	100% read	95% read 5% insert



- The *leaf metadata replication* avoids the extra remote accesses of fetching in-header leaf metadata

Factor Analysis

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 *speculative read* enables greedily fetching the target entry rather than the entire neighborhood

Conclusion

- This paper identifies the **trade-off** between read amplifications and cache consumption for range indexes on DM
- We propose **CHIME**, a hybrid index combining the B+ tree with hopscotch hashing to break the trade-off:
  - Three-level optimistic synchronization
  - Access-aggregated metadata management
  - Hotness-aware speculative read
- CHIME outperforms the state-of-the-art range indexes on DM by up to **5.1x in throughput** with the same cache size and achieves similar performance with up to **8.7x lower cache consumption**

Thank you! Q&A

https://github.com/dmemsys/CHIME

Artifacts Available acm V1.1
Artifacts Evaluated acm Functional V1.1
Results Reproduced acm V1.1

复旦大学 FUDAN UNIVERSITY
昆山杜克大学 DUKE KUNSHAN UNIVERSITY
HUAWEI
香港中文大學 The Chinese University of Hong Kong