WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems

Se Kwon Lee

K. Hyun Lim[1], Hyunsub Song, Beomseok Nam, Sam H. Noh

UNIST

[1]Hongik University

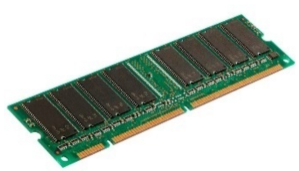
Persistent Memory (PM)

- **Persistent memory is expected to replace both DRAM & NAND**

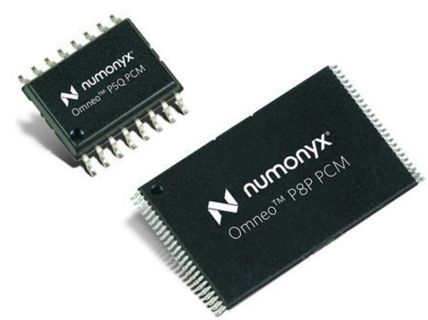
	NAND	STT-MRAM	PCM	DRAM
Non-volatility	o	o	o	x
Read (ns)	2.5×10^4	5 - 30	20 – 70	10
Write (ns)	2×10^5	10 - 100	150 - 220	10
Byte-addressable	x	o	o	o
Density	185.8 $Gbit/cm^2$	0.36 $Gbit/cm^2$	13.5 $Gbit/cm^2$	9.1 $Gbit/cm^2$

K. Suzuki and S. Swanson. “A Survey of Trends in Non-Volatile Memory Technologies: 2000-2014”, IMW 2015

Non-volatile

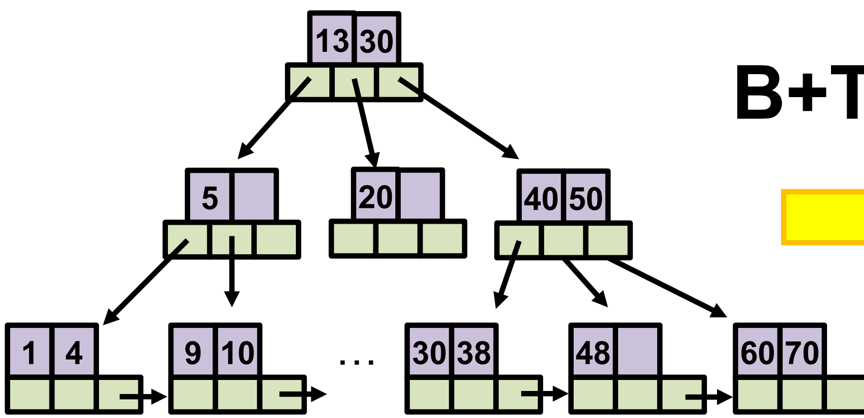


High performance



Persistent Memory

Indexing Structure for PM Storage Systems



B+Tree

Consistency Issue of B+tree in PM

- **B+tree is a block-based index**
  - Key sorting → Block granularity write
  - Rebalancing → Multi-blocks granularity write
- **Persistent memory**
  - Byte-addressable → Byte granularity write
  - Write reordering

Can result in consistency problem

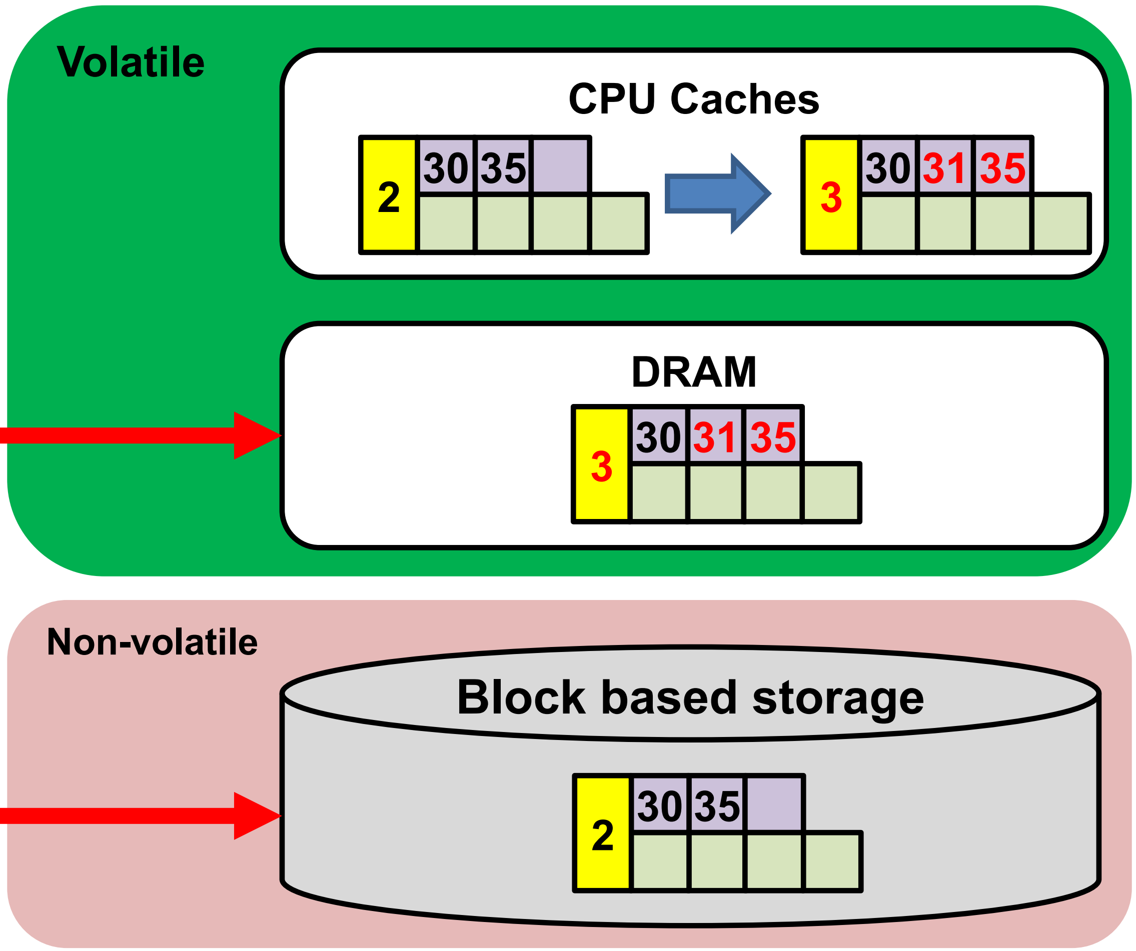
Consistency Issue of B+tree in PM

- **Traditional case**

Write reordering

Not persistent data

Block granularity update



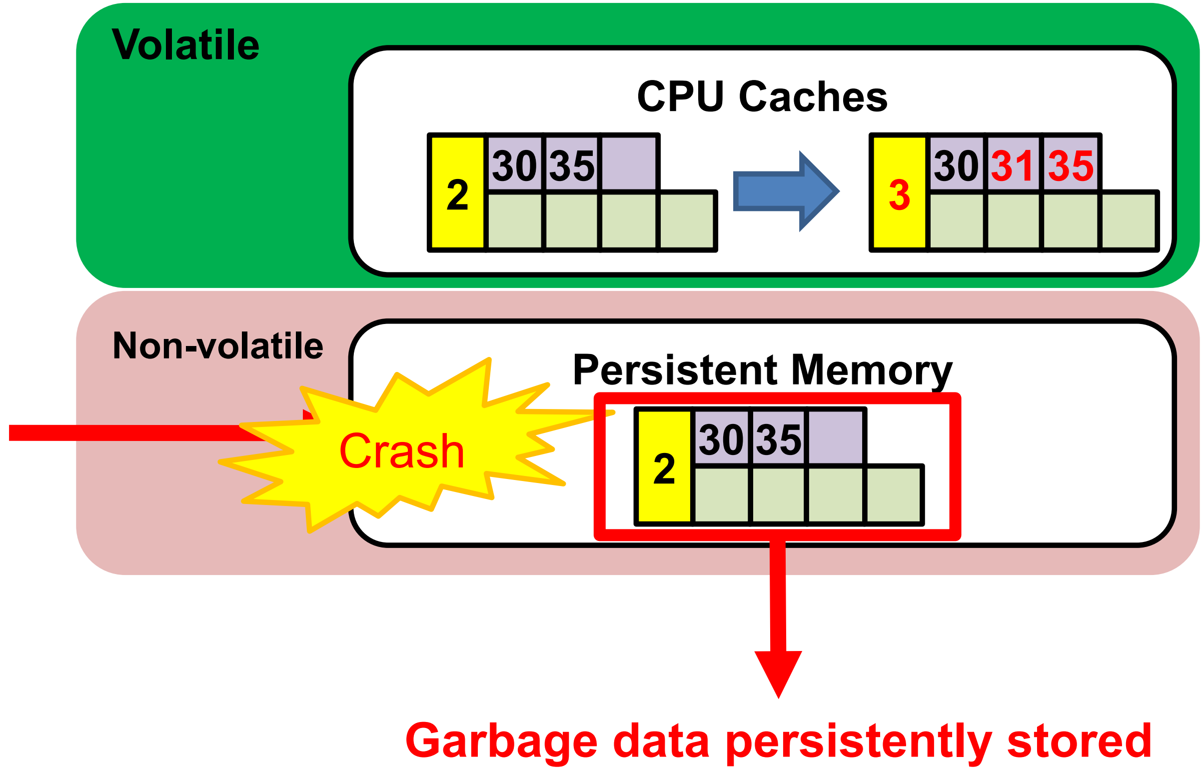
Consistency Issue of B+tree in PM

- **PM case**

Byte granularity update

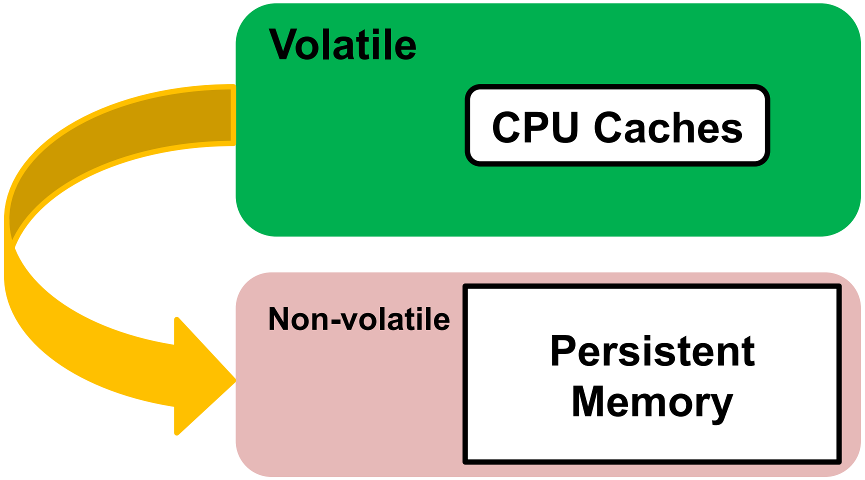
Write reordering

Persistent data



Primitives for Data Consistency in PM

- **Durability**
  - ***CLFLUSH*** (Flush cache line)
    - Can be reordered
- **Ordering**
  - ***MFENCE*** (Load and Store fence)
    - Order CPU cache line flush instructions



Primitives for Data Consistency in PM

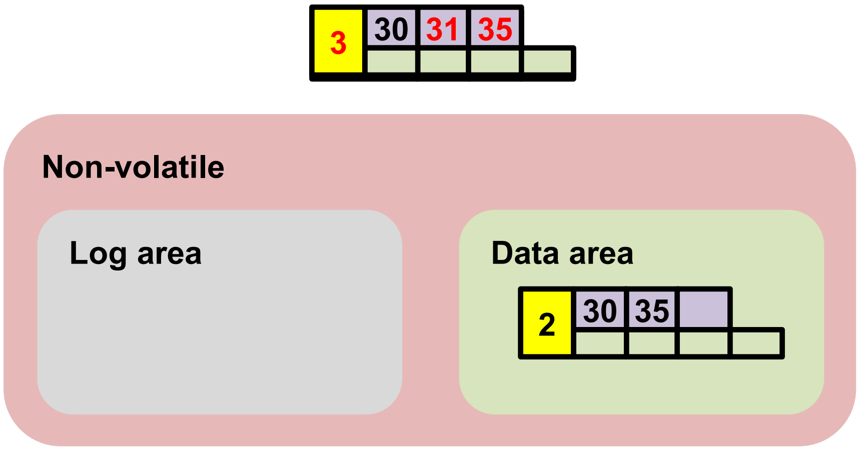
- D
  -
- C
  -

Serialization of *CLFLUSH* and *MFENCE* is known to cause **large overhead**

instructions

Primitives for Data Consistency in PM

- **Atomicity**
  - 8-byte failure atomicity
    - Need only CLFLUSH
  - Logging or CoW based atomicity (more than 8 bytes)
    - Requires duplicate copies



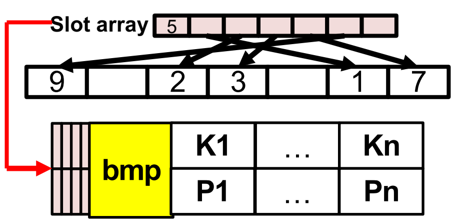
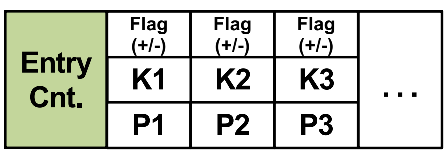
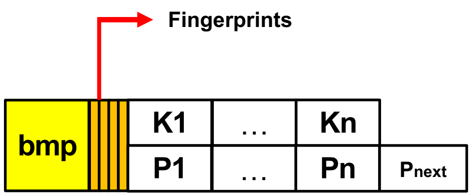
Primitives for Data Consistency in PM

Logging increases cache line flush overhead

B+tree Variants for Persistent Memory

How can we ensure consistency using failure-atomic writes without logging?

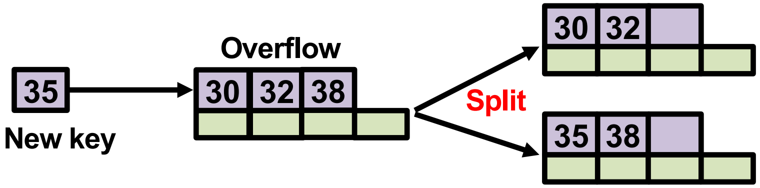
Unsorted keys → Append-only with metadata
Failure-atomic update of metadata

wB+Tree (VLDB' 15)	NVTree (FAST' 15)	FPTree (SIGMOD' 16)
		

Unsorted key → Decreases search performance

B+tree Variants for Persistent Memory

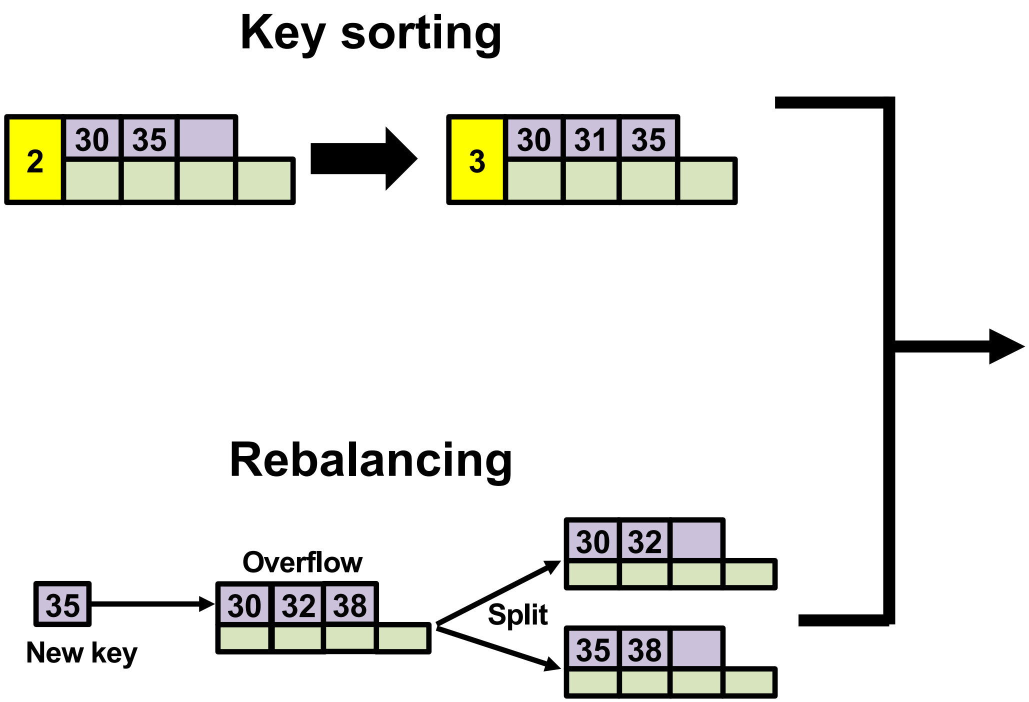
- **Logging still necessary**
  - Multi-block granularity updates due to node splits and merges
    - Cannot update atomically



  - Logging-based solution
    - wB+Tree, FPTree
  - Tree reconstruction based solution
    - NVTree

→ **large overhead**

B+tree Variants for Persistent Memory



Fundamental characteristics of B+tree cause problems

B+tree Variants for Persistent Memory

Why use B+ trees in the first place?

Perhaps there is a better tree data structure more suited for PM?

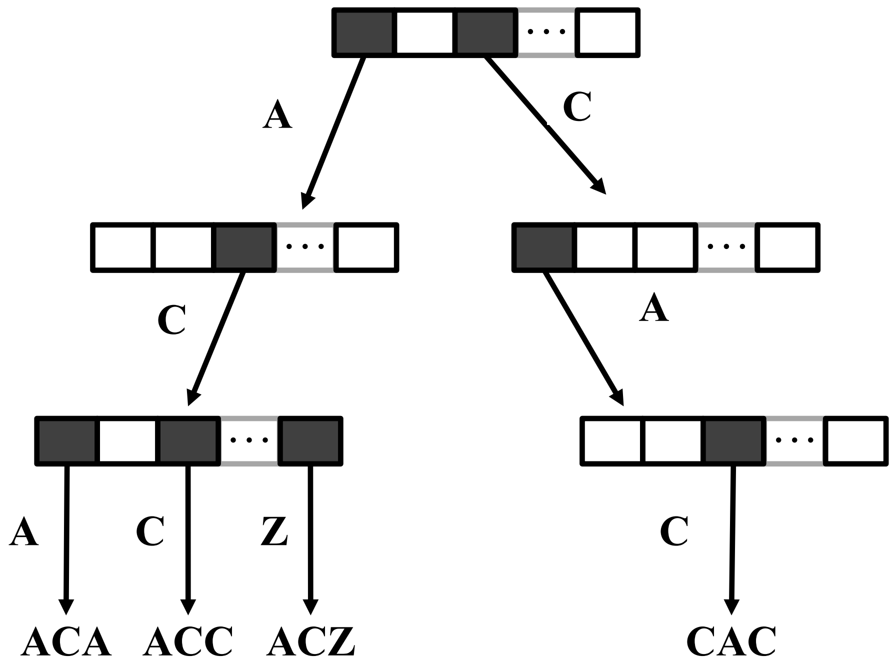
Our Contributions

- **Show Radix Tree is a suitable data structure for PM**
- **Propose optimal radix tree variants WORT and WOART**
  - WORT: Write Optimal Radix Tree
  - WOART: Write Optimal redesigned Adaptive Radix Tree (ART)
    - Optimal: maintain consistency only with single failure-atomic write without any duplicate copies

Radix Tree

Background

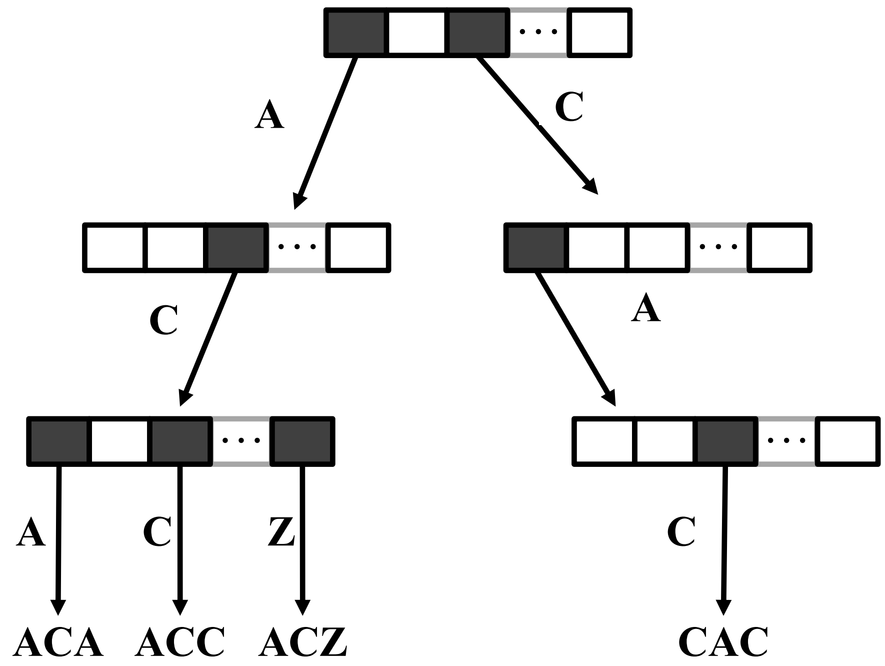
- **Deterministic structure**



Radix Tree

Background

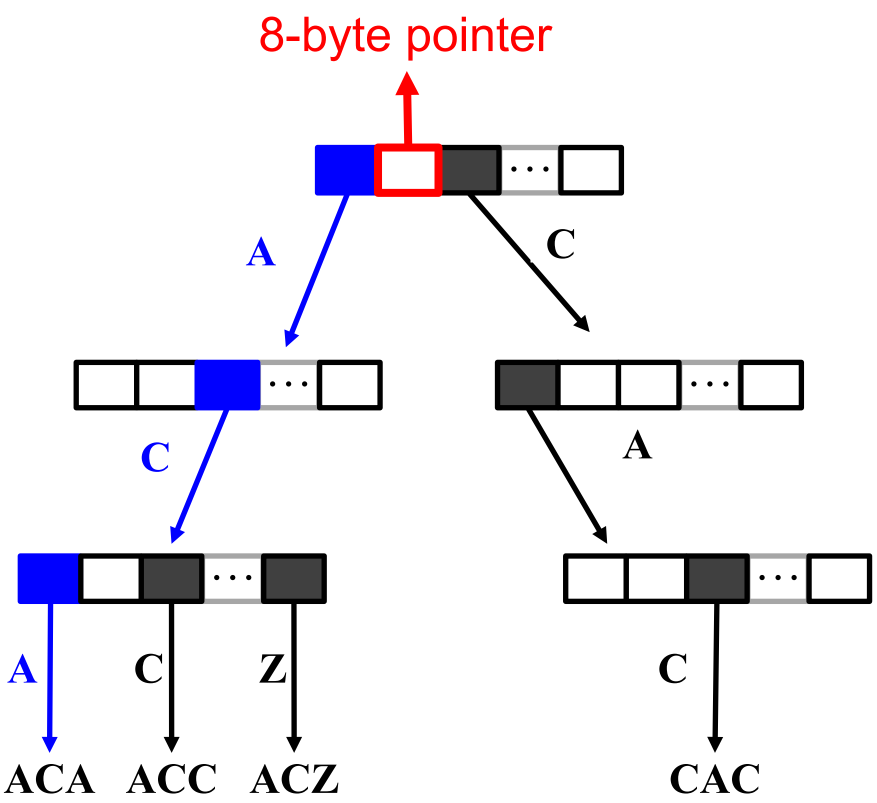
- Deterministic structure
  - No key comparison



Next-generation Embedded / Computer System Software Technology

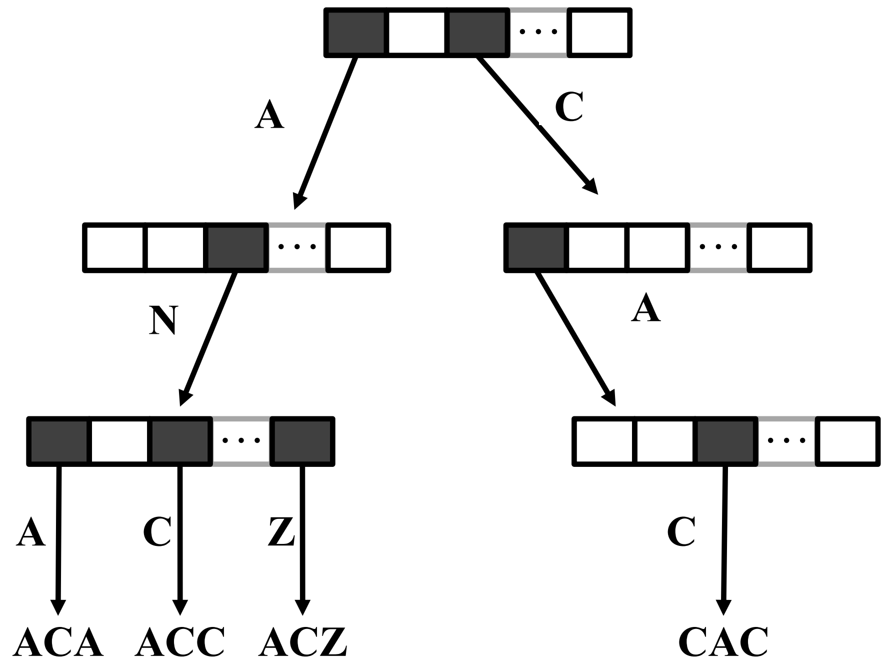
Radix Tree

- **Deterministic structure**
  - No key comparison
    - Only 8-byte pointer entries
    - Implicitly stored keys



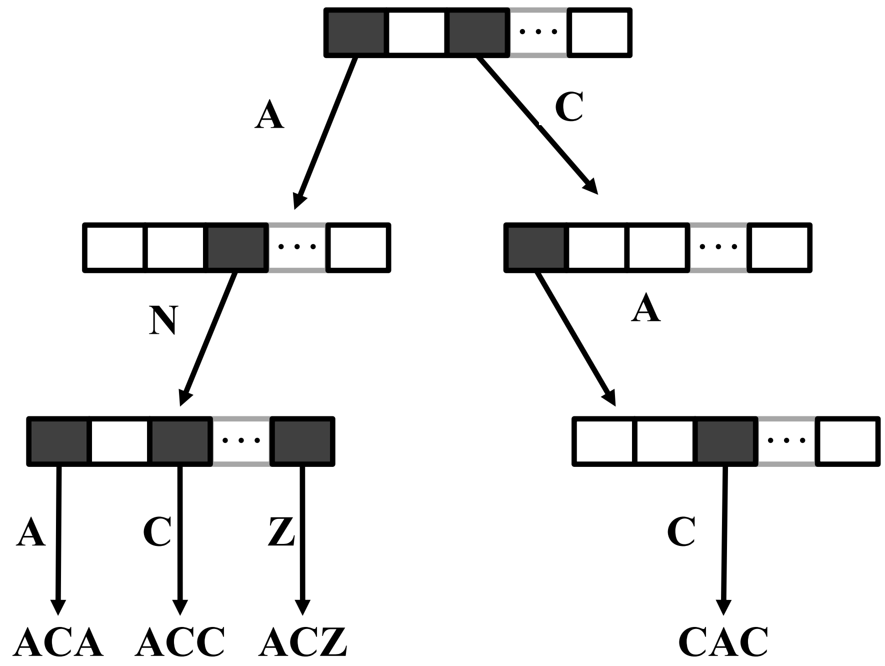
Radix Tree

- **Deterministic structure**
  - No key comparison
    - Only 8-byte pointer entries
    - Implicitly stored keys
    - No problem caused by key sorting



Radix Tree

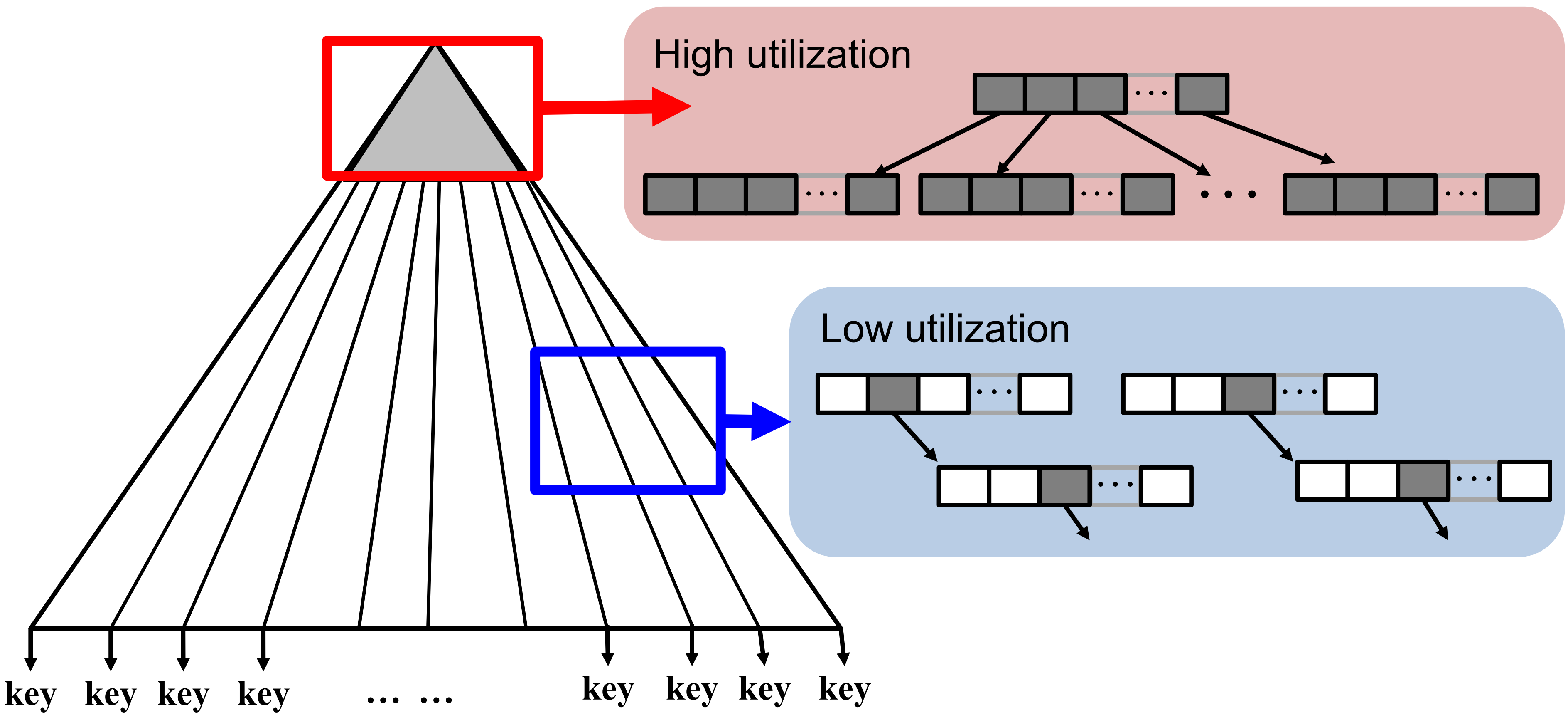
- **Deterministic structure**
  - No key comparison
    - Only 8-byte pointer entries
    - Implicitly stored keys
    - No problem caused by key sorting
  - No modification of other keys
    - Single 8-byte pointer write per node
    - Easy to use failure-atomic write



Problem of Deterministic Structure

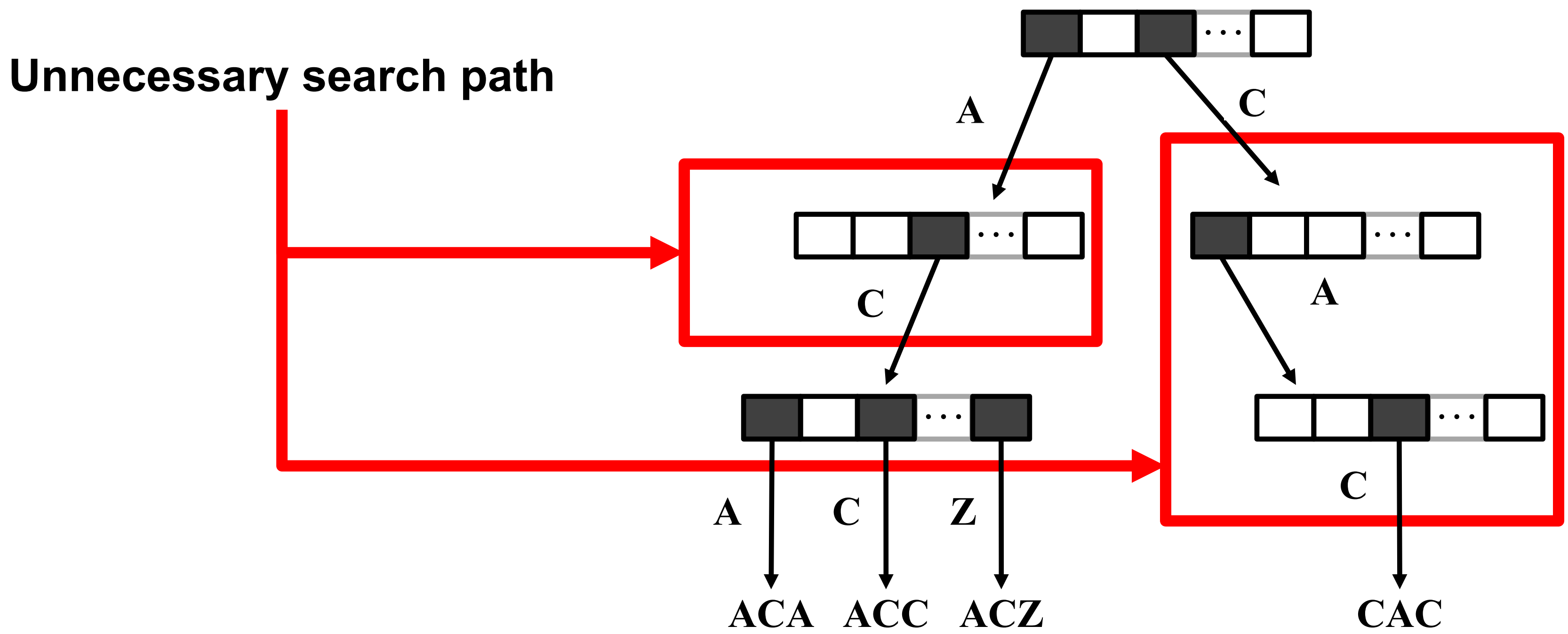
Background

- **For sparse key distribution**
  - Waste excessive memory space → Optimized through path compression



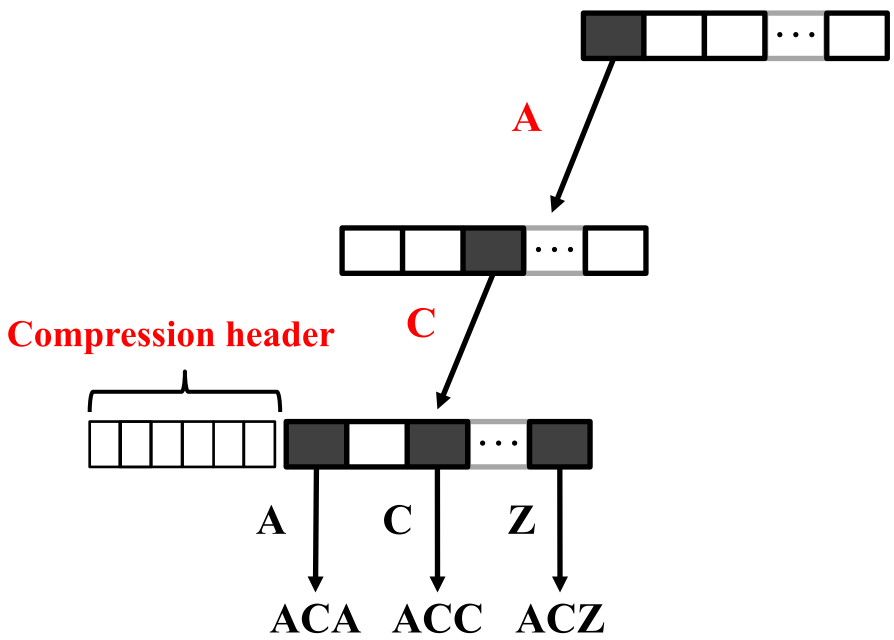
Path Compression in Radix Tree

- **Path compression**
  - Search paths that do not need to be distinguished can be removed



Path Compression in Radix Tree

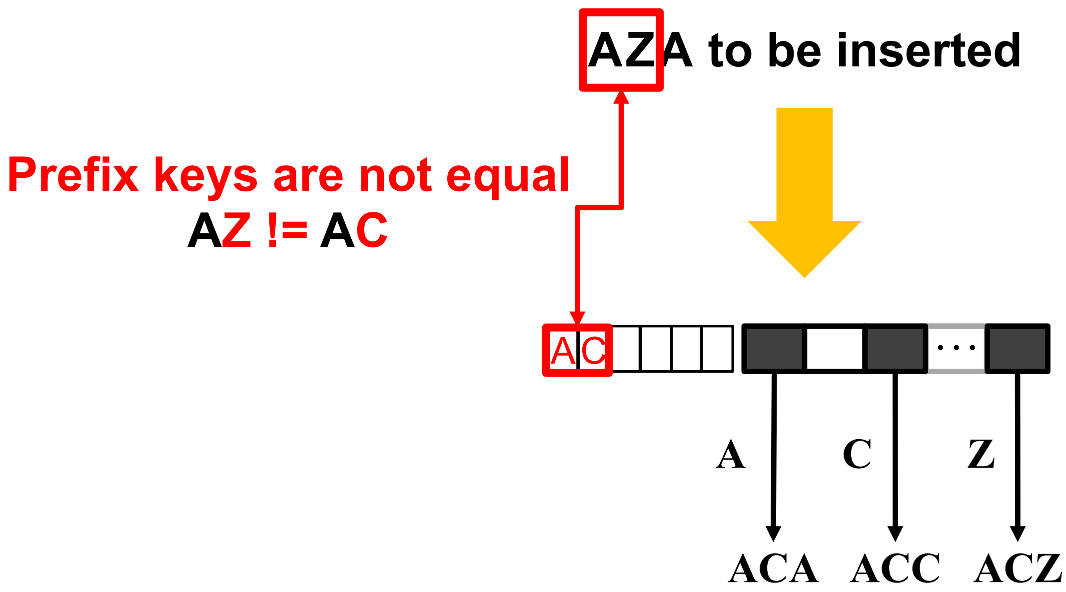
- **Path compression**
  - Common search path is compressed in header
  - Improve memory utilization & indexing performance



Node Split with Path Compression

Background

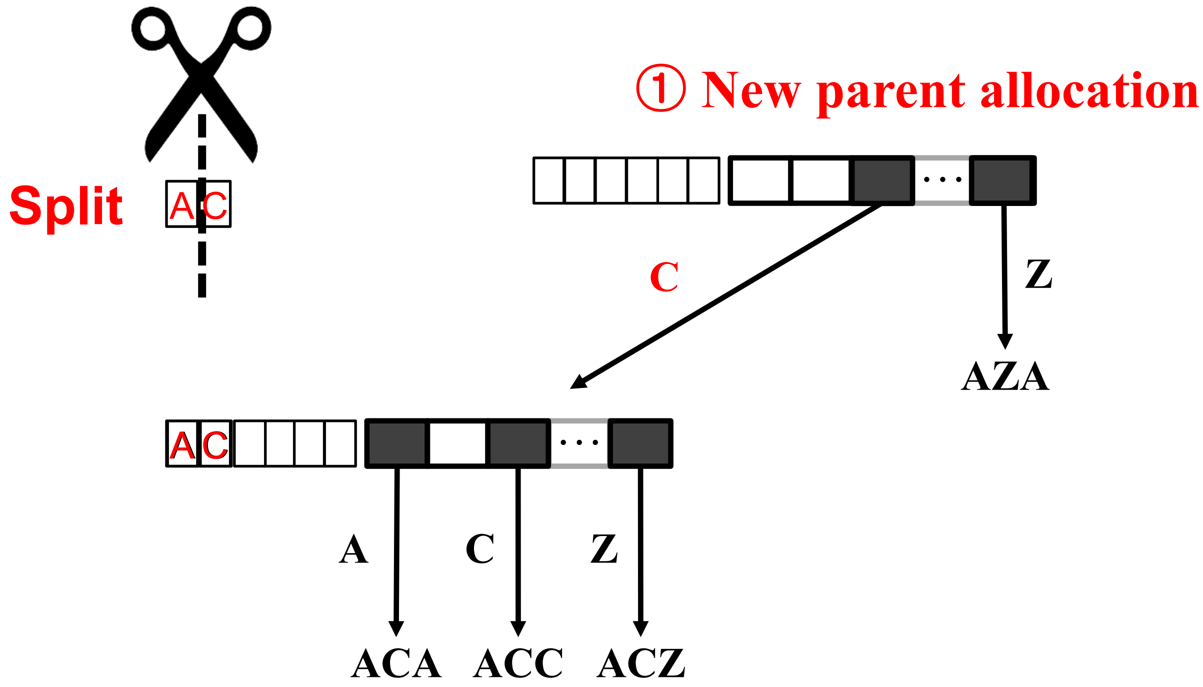
- **Path compression split**



Node Split with Path Compression

Background

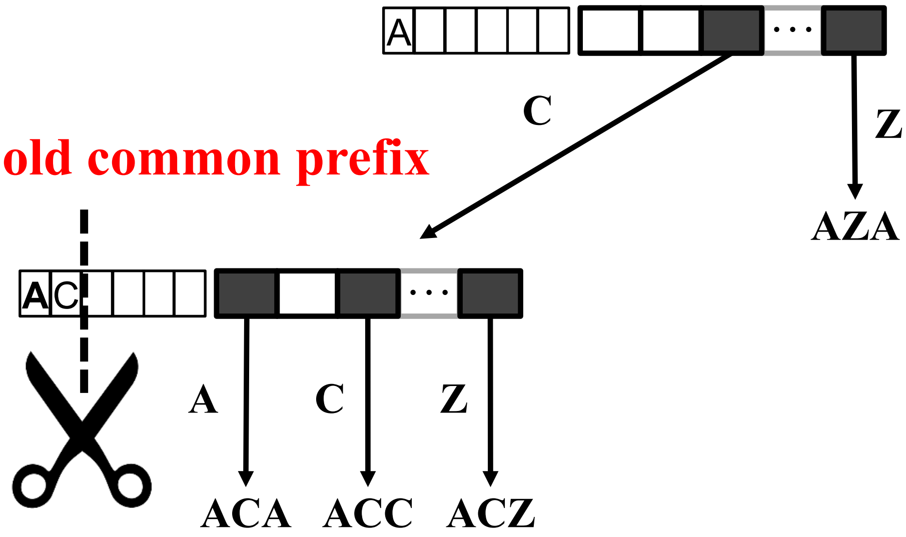
- **Path compression split**



Node Split with Path Compression

- **Path compression split**

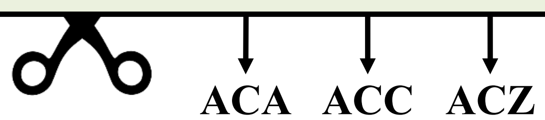
② Decompression of old common prefix



Node Split with Path Compression

- **Path compression split**

However, this split process causes consistency problem in PM.



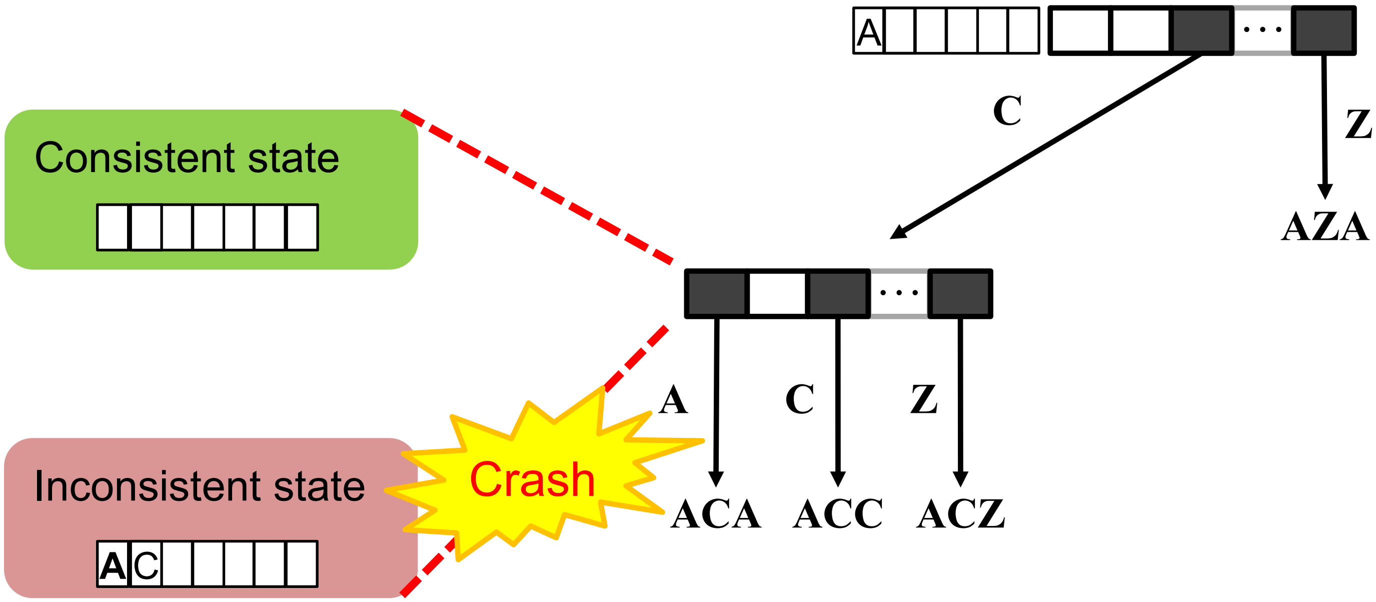
Path compression

Problem

in PM

Consistency Issue of Path Compression

- **Path compression split**
  - cause updates of multiple nodes
  - have to employ expensive logging methods



Path compression

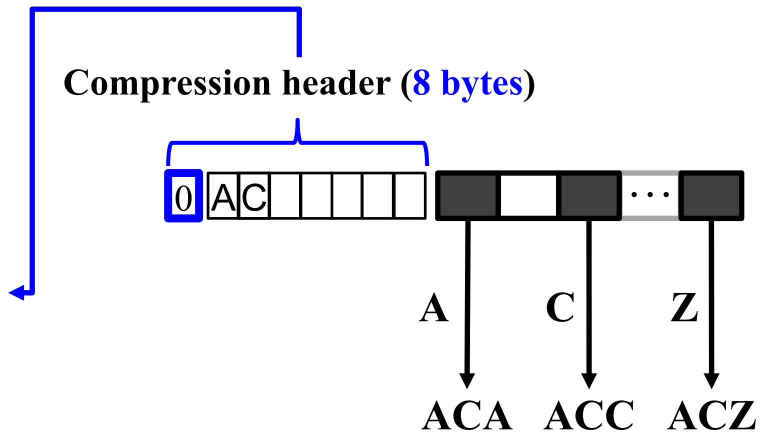
Solution

WORT (Write-Optimal Radix Tree) for PM

Our solution

- ***Failure-atomic path compression***
  - Add node depth field to compression header

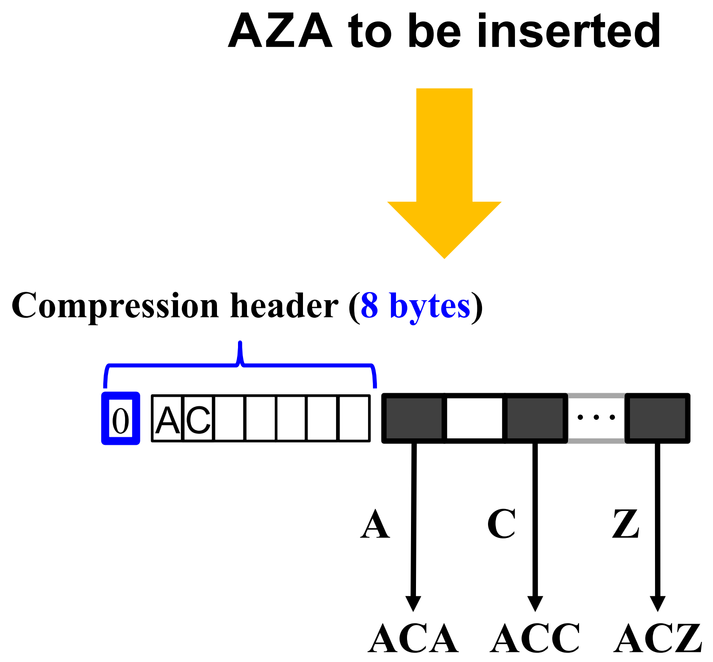
```
struct Header {
    unsigned char depth;
    unsigned char PrefixArr[7];
}
```



WORT (Write-Optimal Radix Tree) for PM

Our solution

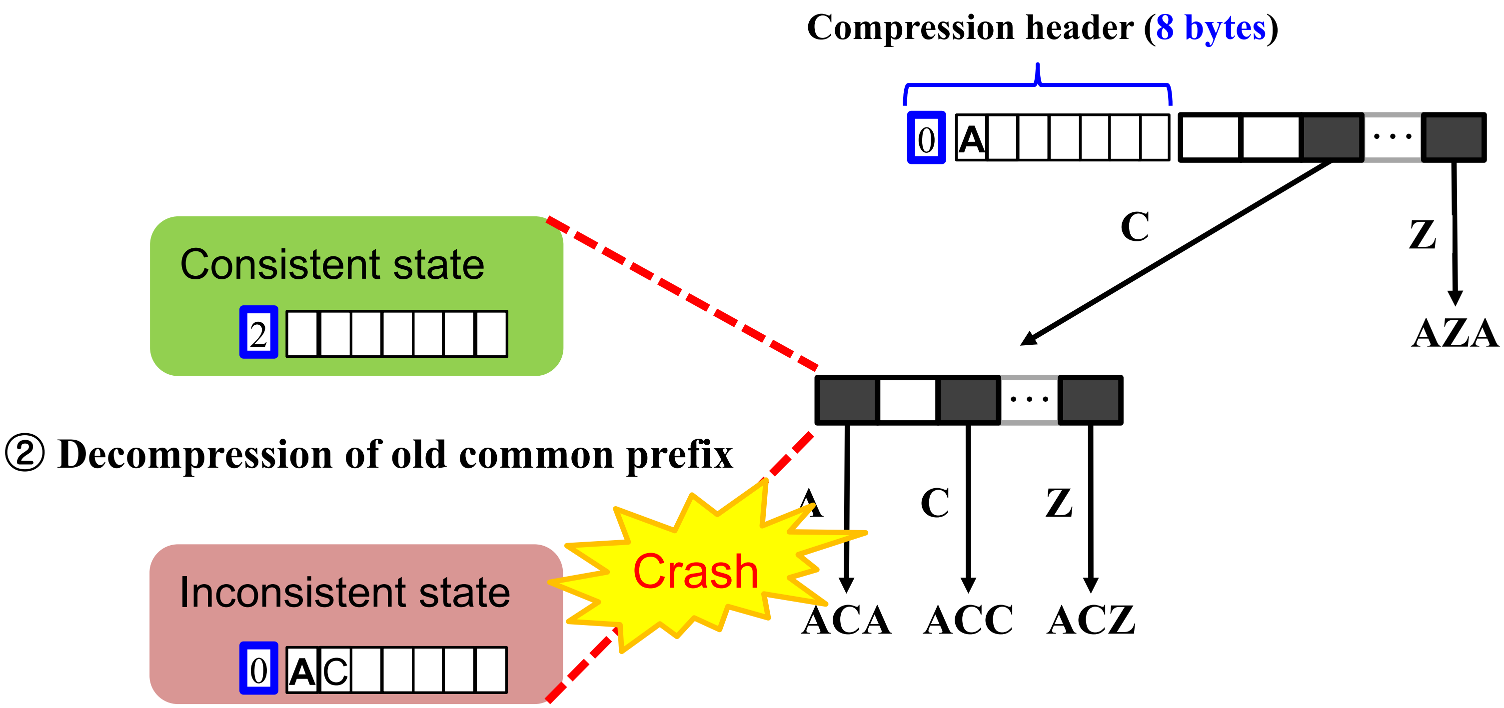
- ***Failure-atomic path compression***
  - **Add node depth field to compression header**



WORT (Write-Optimal Radix Tree) for PM

Our solution

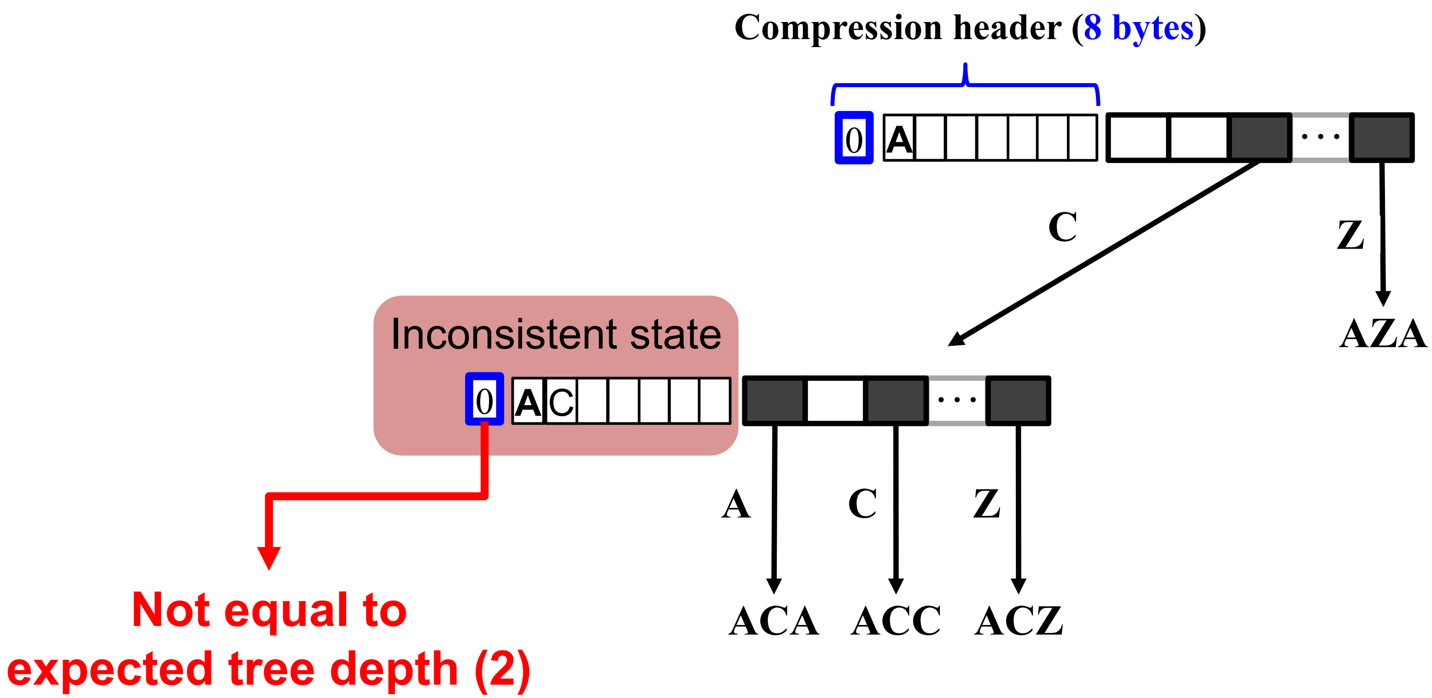
- ***Failure-atomic path compression***
  - **Add node depth field to compression header**



WORT (Write-Optimal Radix Tree) for PM

Our solution

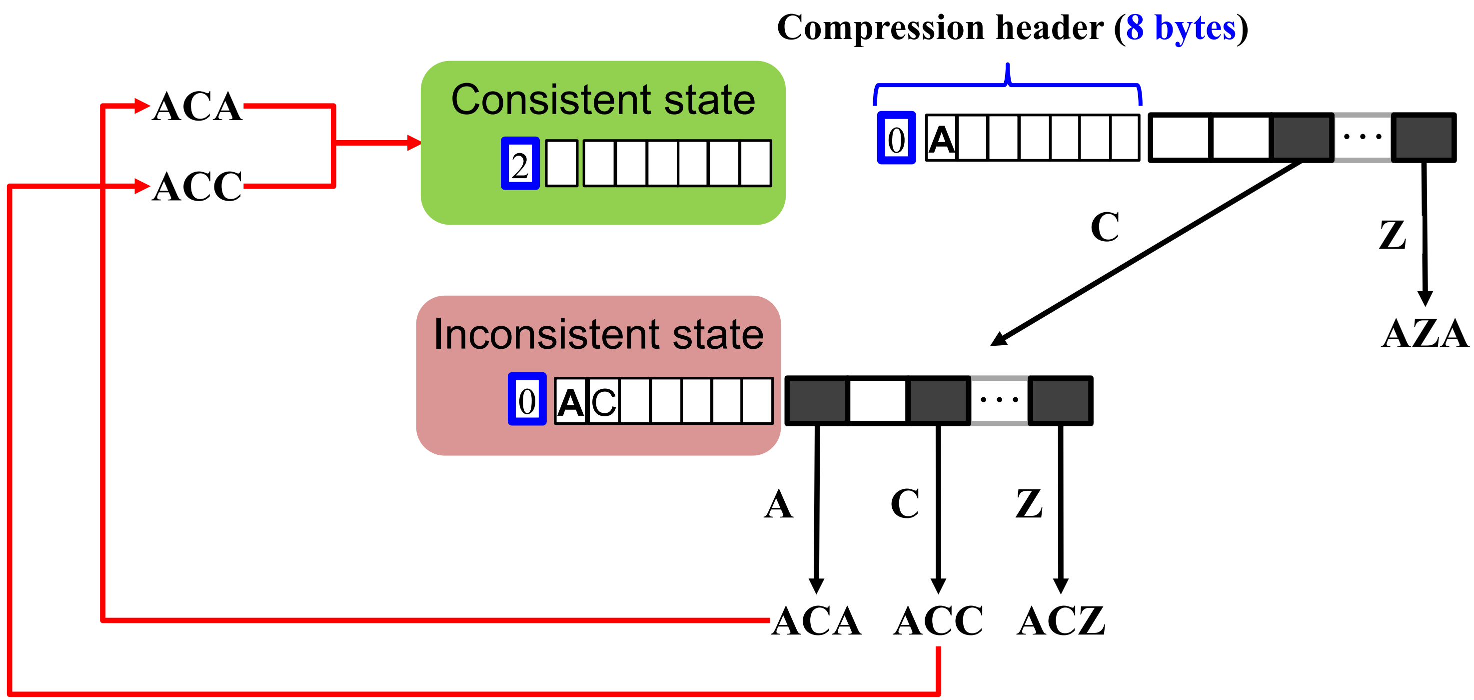
- ***Failure-atomic path compression***
  - *Failure detection in WORT*
    - Depth in a header ≠ Counted depth → Crashed header



WORT (Write-Optimal Radix Tree) for PM

Our solution

- ***Failure-atomic path compression***
  - *Failure recovery in WORT*
    - Compression header can be reconstructed → Atomically overwrite



UNIST ECE SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING CISSR NECSST Next-generation Embedded / Computer System Software Technology

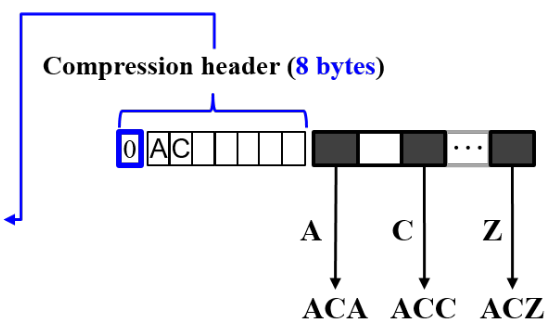
Write Optimal Data Structure for PM

- **Our proposed radix tree variant is optimal for PM**
  - Consistency is always guaranteed with a single 8-byte failure-atomic write without any additional copies for logging or CoW

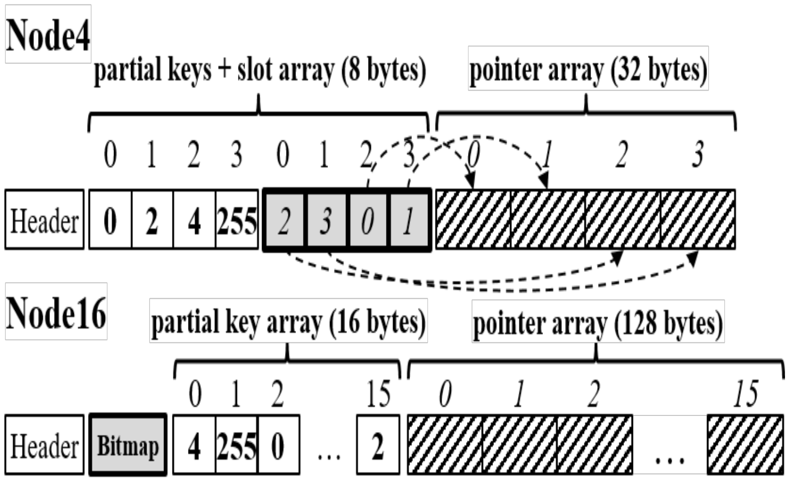
WORT (Write Optimal Radix Tree)	
WOART (Write Optimal Adaptive Radix Tree)	
1. Failure-atomic path compression	2. Redesigned adaptive node

1. Failure-atomic path compression

```
struct Header {
    unsigned char depth;
    unsigned char PrefixArr[7];
}
```



2. Redesigned adaptive node



- **Experimental environment**

System configuration

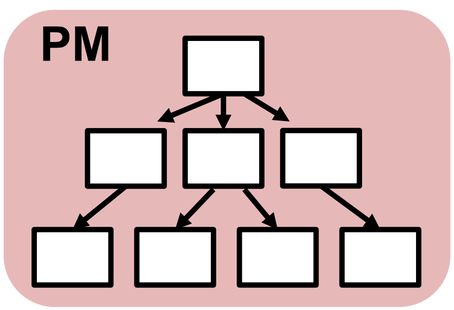
	Description
CPU	Intel Xeon E5-2620V3 X 2
OS	Linux CentOS 6.6 (64bit) kernel v4.7.0
PM	Emulated with 256GB DRAM Write latency: Injecting additional stall cycles

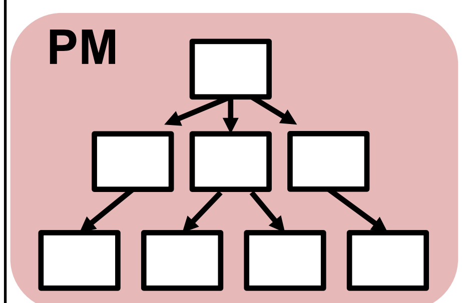
Evaluation

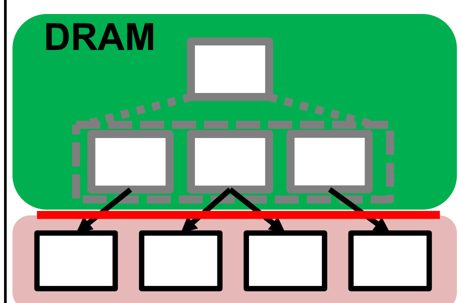
- **Experimental environment**

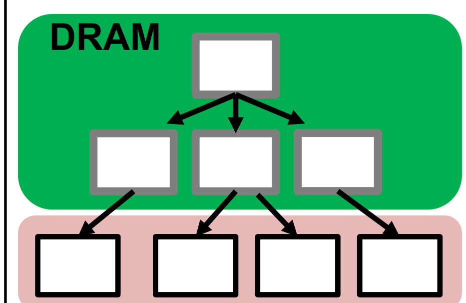
Comparison group

Radix tree variants	B+tree variants		
WORT	wB+Tree (VLDB' 15)	NVTree (FAST' 15)	FPTree (SIGMOD' 16)







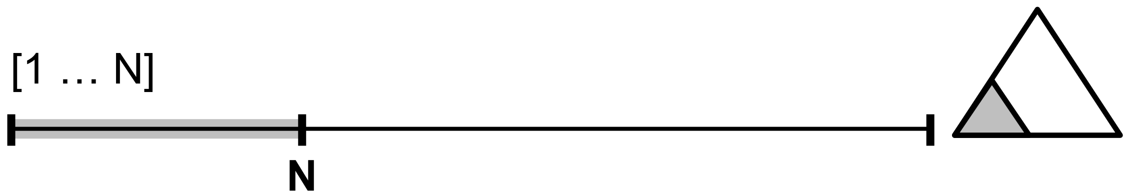


Evaluation

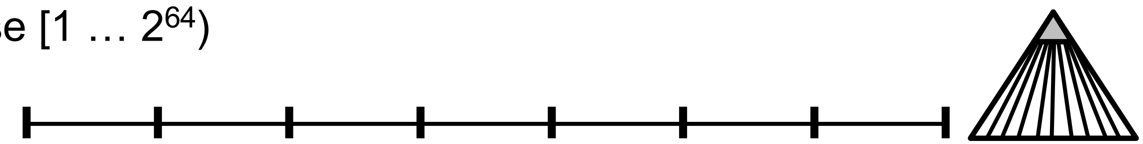
- **Experimental environment**

Synthetic Workload Characteristics

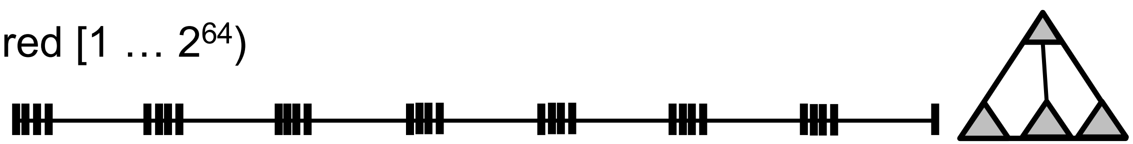
- Dense [1 … N]



- Sparse [1 … 2^{64})

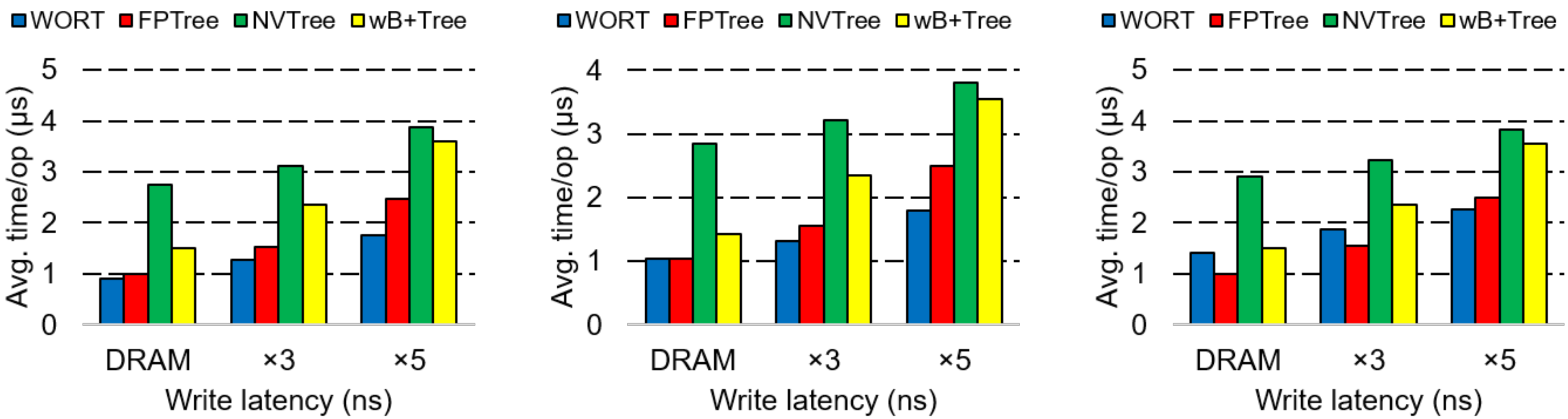


- Clustered [1 … 2^{64})



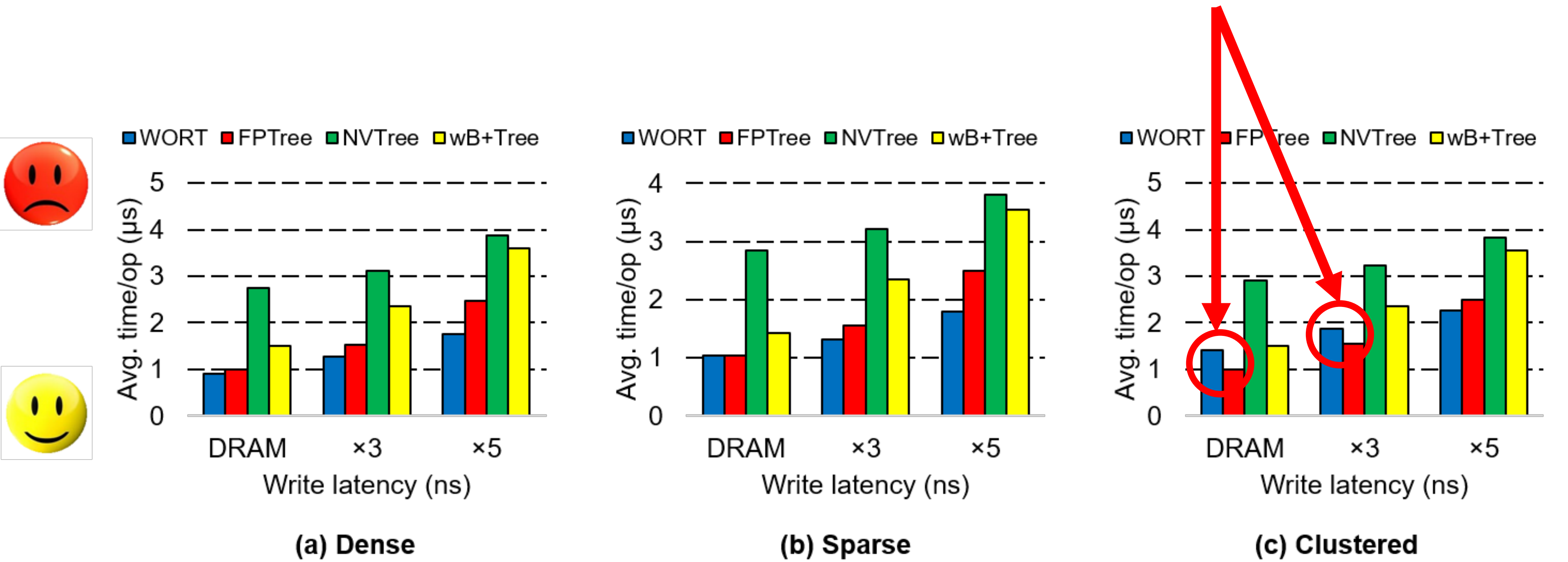
Evaluation

- **Insertion performance**
  - WORT outperform the B+tree variants in general



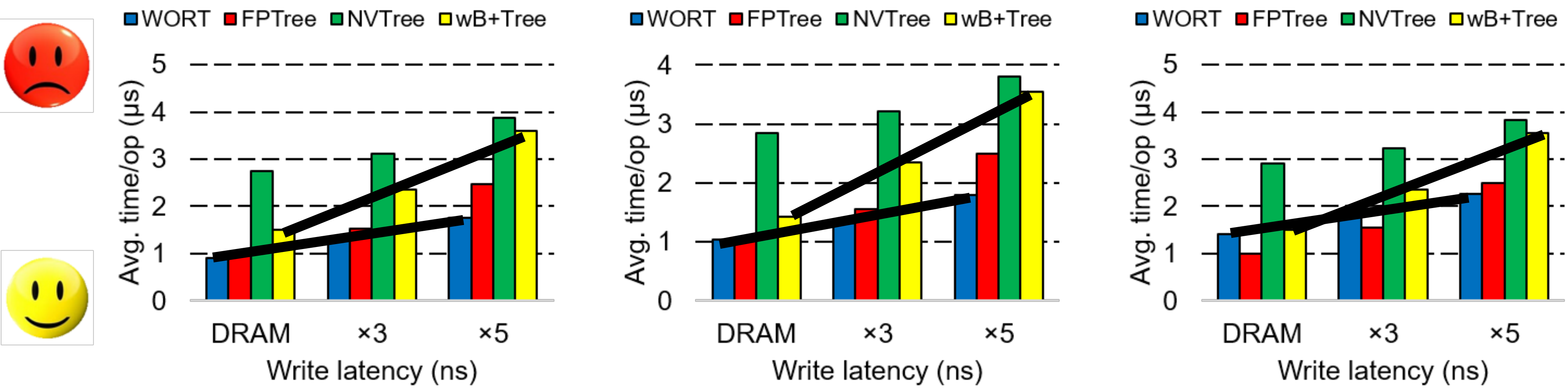
(a) Dense **(b) Sparse** **(c) Clustered**

- **Insertion performance**
  - WORT outperform the B+tree variants in general
    - DRAM-based internal node → more favorable performance for FPTree



Evaluation

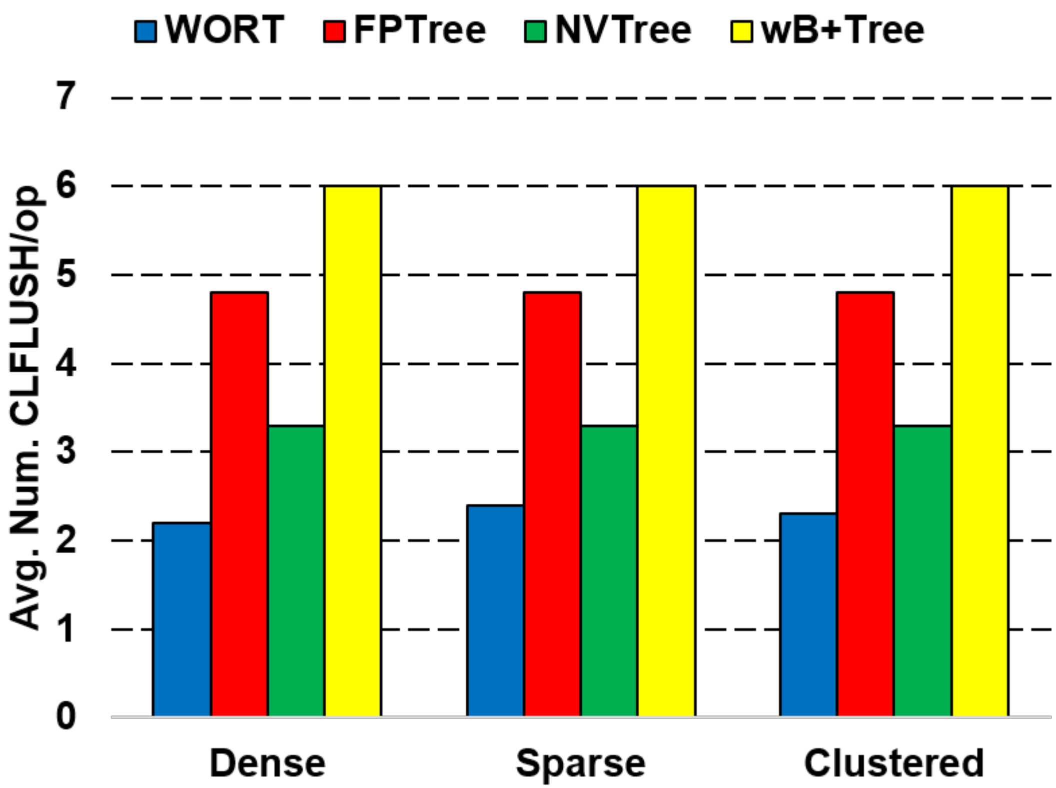
- **Insertion performance**
  - WORT vs wB+Tree
    - Performance differences increase in proportion to write latency



(a) Dense **(b) Sparse** **(c) Clustered**

Evaluation

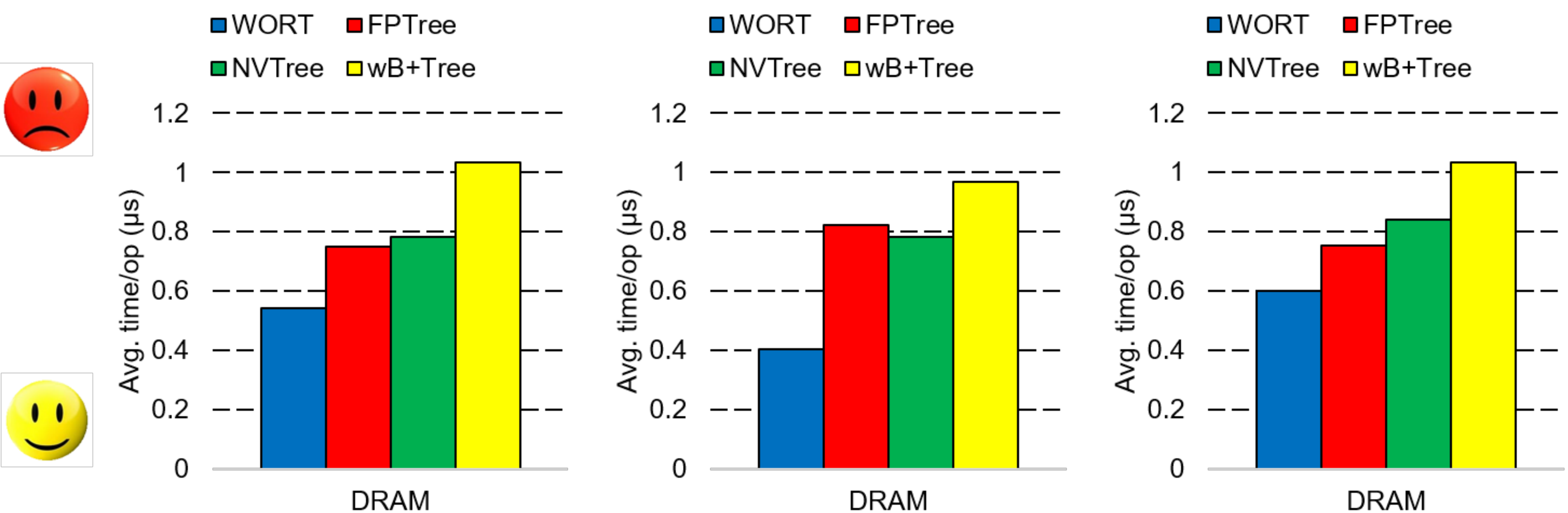
- **CLFLUSH count per operation**
  - B-tree variants incur more cache flush instructions



Evaluation

Search performance

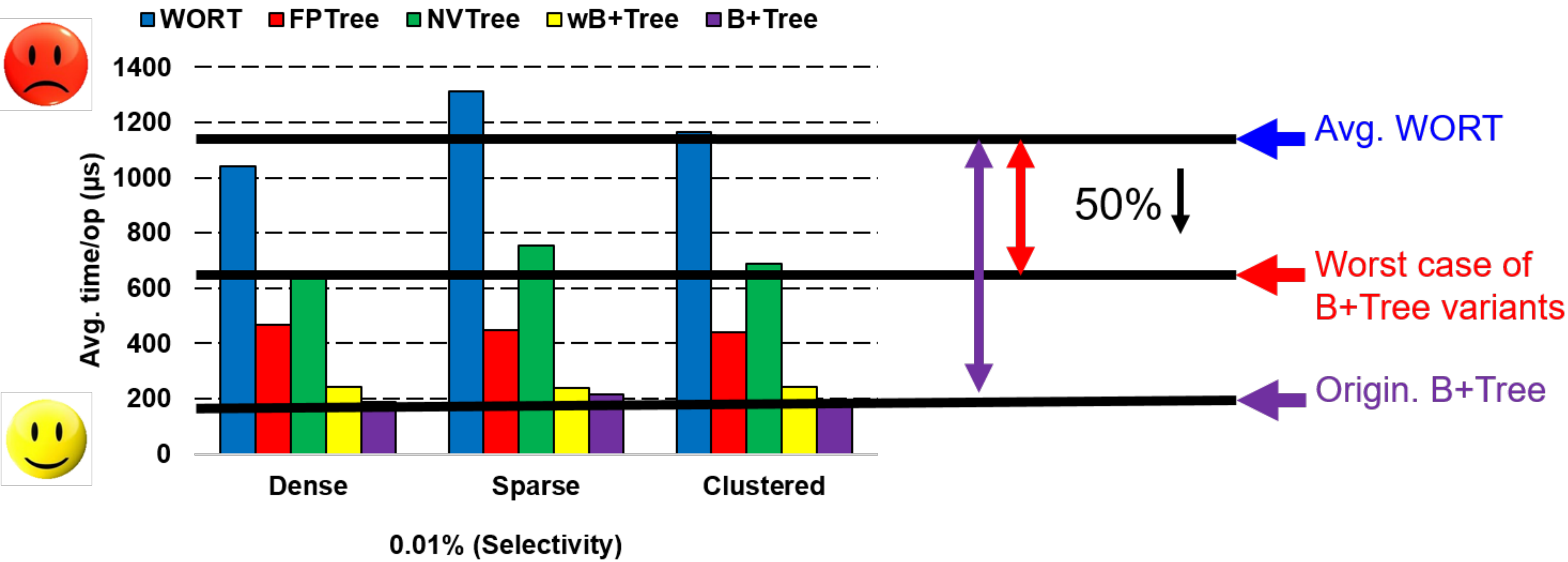
- WORT always perform better than B+Tree variants



(a) Dense (b) Sparse (c) Clustered

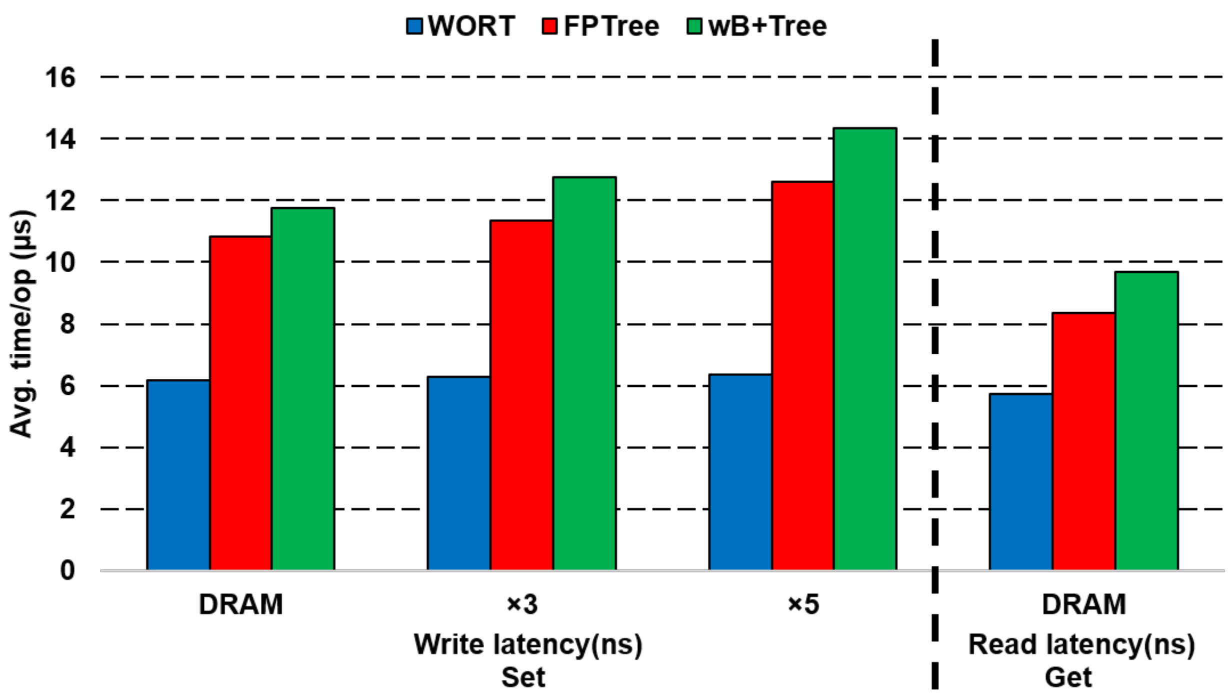
Evaluation

- **Range query performance**
  - Performance gap for range query decreases for PM indexes compared with it between WORT and original B+Tree
    - B+Tree variants do not keep the keys sorted → Rearrangement overhead



Evaluation

- **MC-benchmark performance on Memcached**
  - WORT outperform B+Tree variants in both SET and GET
    - Additional indirection & flush overhead in B-tree variants



Conclusion

- **Showed suitability of radix tree as PM indexing structure**
- **Proposed optimal radix tree variants WORT and WOART**
  - Optimal: maintain consistency only with single failure-atomic write without any duplicate copies


Q&A