

# Persistent B+-Trees in Non-Volatile Main Memory

Shimin Chen

State Key Laboratory of Computer Architecture  
Institute of Computing Technology  
Chinese Academy of Sciences

[chensm@ict.ac.cn](mailto:chensm@ict.ac.cn)

Qin Jin\*

Computer Science Department  
School of Information  
Renmin University of China

[qjin@ruc.edu.cn](mailto:qjin@ruc.edu.cn)

Presentation By

Tarikul Islam Papon

# Outline

---

Background

Existing Solutions

Write Atomic B<sup>+</sup> Trees

Experimental Evaluations

Conclusion

# Background

# Background

---

Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory

# Background

---

Persistent B<sup>+</sup> Trees in **Non-Volatile Main Memory**

# Background

---

**Persistent** B<sup>+</sup> Trees in Non-Volatile Main Memory

# Background

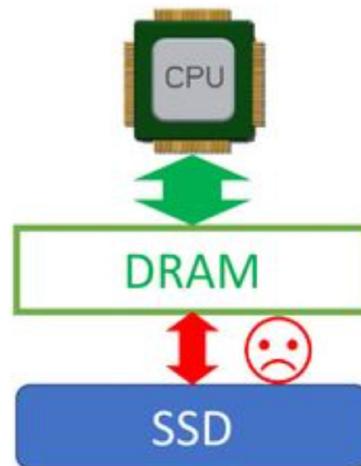
---

Persistent **B<sup>+</sup> Trees** in Non-Volatile Main Memory

# Persistent B<sup>+</sup> Trees in **Non-Volatile Main Memory**

---

Main Memory	Secondary Storage
Fast	Slow
Volatile	Non-Volatile



# Persistent B<sup>+</sup> Trees in **Non-Volatile Main Memory**

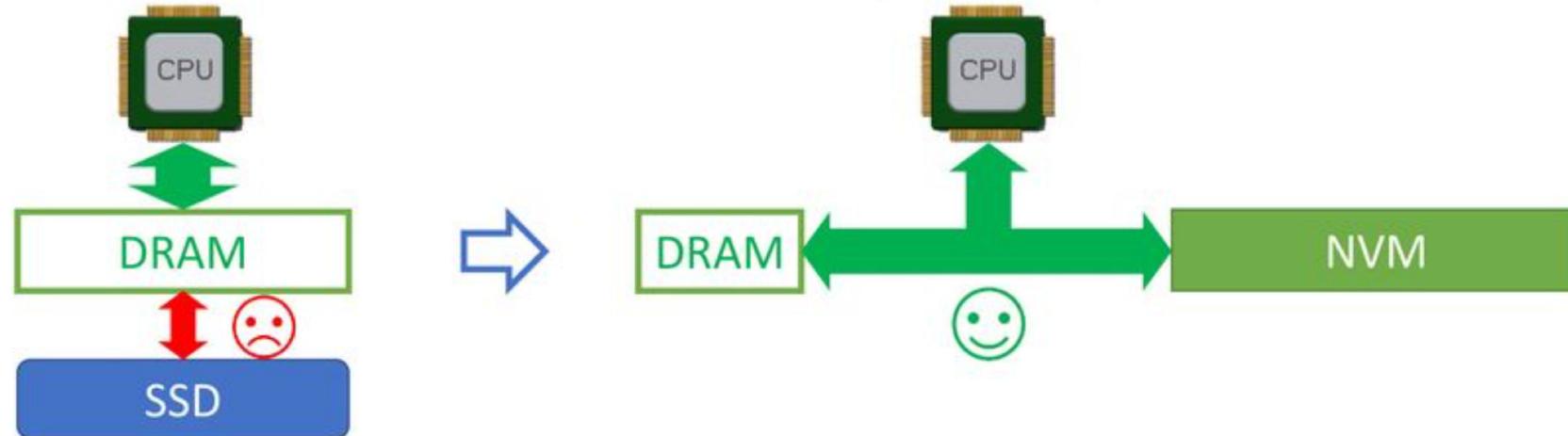
---

Main Memory	Secondary Storage
Fast	Slow
Volatile	Non-Volatile



# Persistent B<sup>+</sup> Trees in **Non-Volatile Main Memory**

Main Memory	Secondary Storage
Fast	Slow
Volatile	Non-Volatile



# Persistent B<sup>+</sup> Trees in **Non-Volatile Main Memory**

---

## Basic Properties

- Byte-addressable reads and writes
- Performance very close to DRAM
- Requires lower power than DRAM
- Non-volatile
- Writes are generally slow
- Low endurance

# Persistent B<sup>+</sup> Trees in **Non-Volatile Main Memory**

---

## Different Types of NVMM

- Phase Change Memory (PCM)
- STT-MRAM
- Memristor



# Persistent B<sup>+</sup> Trees in **Non-Volatile Main Memory**

---

## Different Types of NVMM

- PCM has much slower writes (e.g., 200ns – 1μs) than reads (e.g., 50ns)
- STT-MRAM and Memristors show faster read and write performance, but are not mature enough yet

# Persistent B<sup>+</sup> Trees in **Non-Volatile Main Memory**

---

## Key Assumption

- NVM chip can guarantee atomic writes to aligned **8-byte** words

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory

---

Essential for instantaneous failure recovery (especially in NVMM)

In case of power failure or system crash data structure can be in

- Inconsistent state
- Non-recoverable state

# Motivation of **Persistent** Data Structure

---

Two General Trends

- NVMM
- Main memory database systems



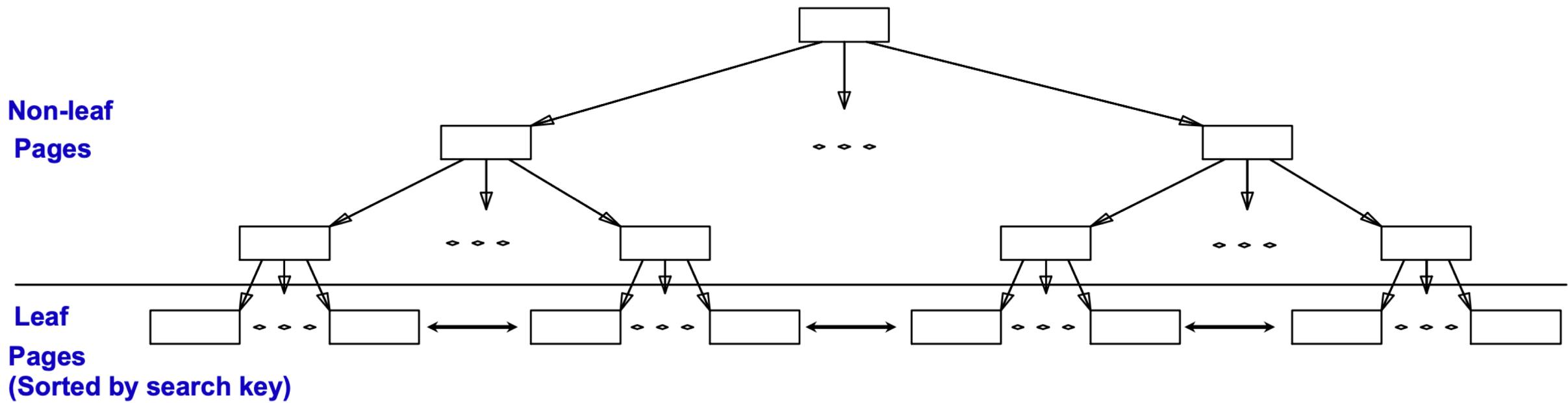
# Persistent **B<sup>+</sup> Trees** in Non-Volatile Main Memory

---

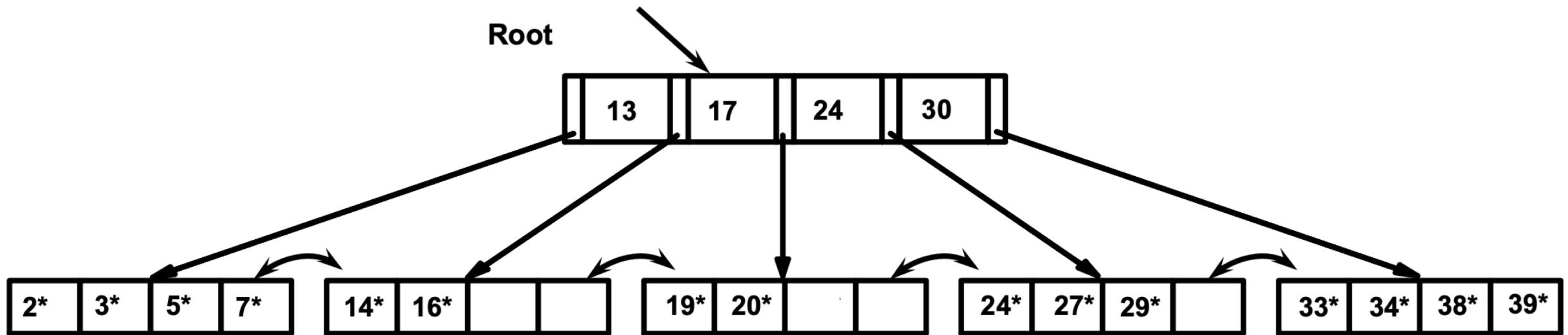
Widely used in Database Systems

- Supports equality and range-searches efficiently.
- Insert/Delete at  $\log_F N$  cost ( $F = \text{fanout}$ ,  $N = \#\text{leaf pages}$ )
- Minimum 50% occupancy (except for root)
- Each node contains  $d \leq m \leq 2d$  entries

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory

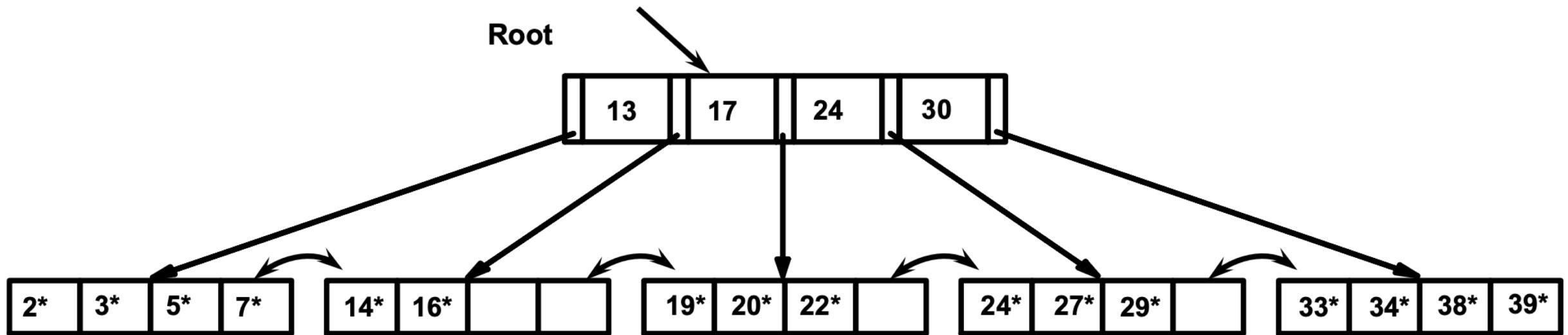


# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory



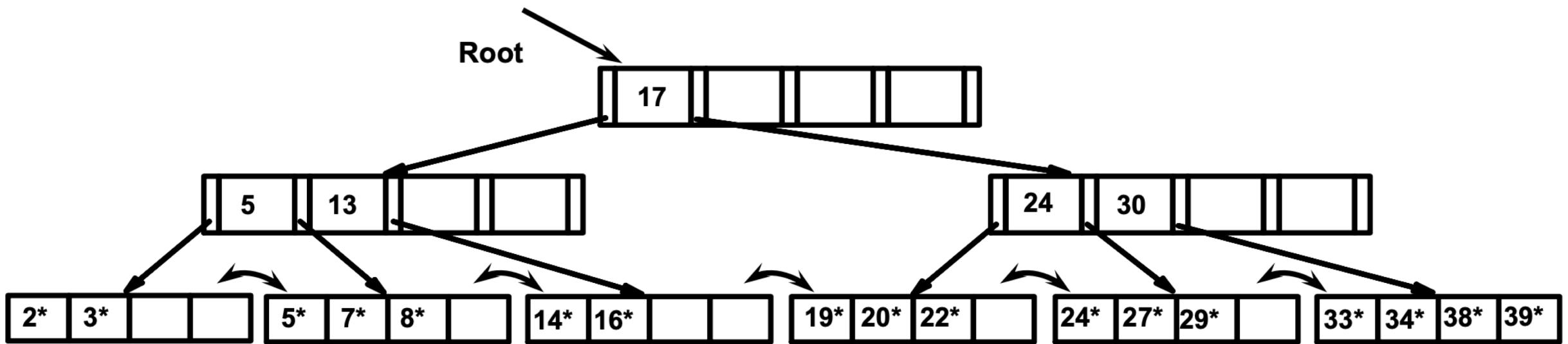
Insert 22

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory



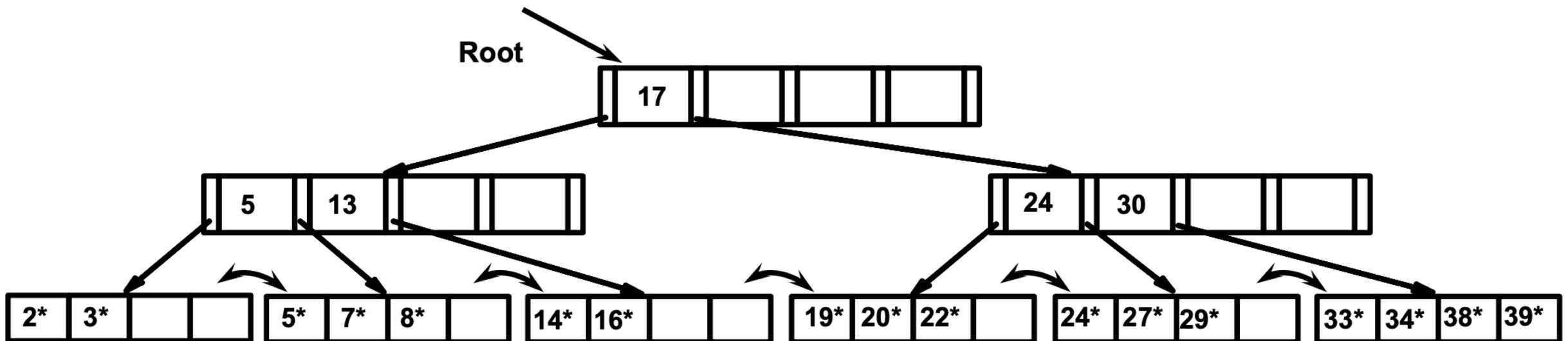
Insert 8

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory



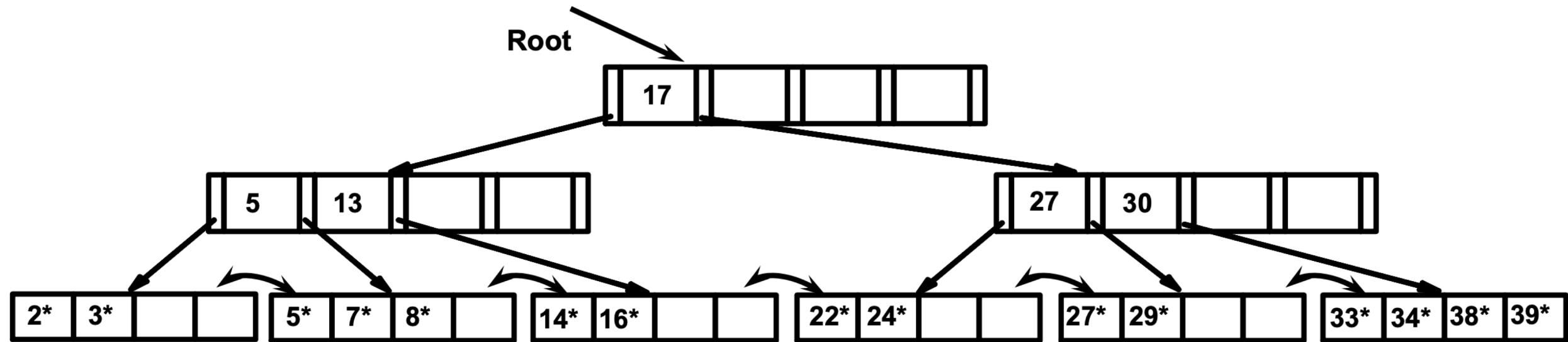
After inserting 8 (Root was *split*)

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory



Delete 19, 20

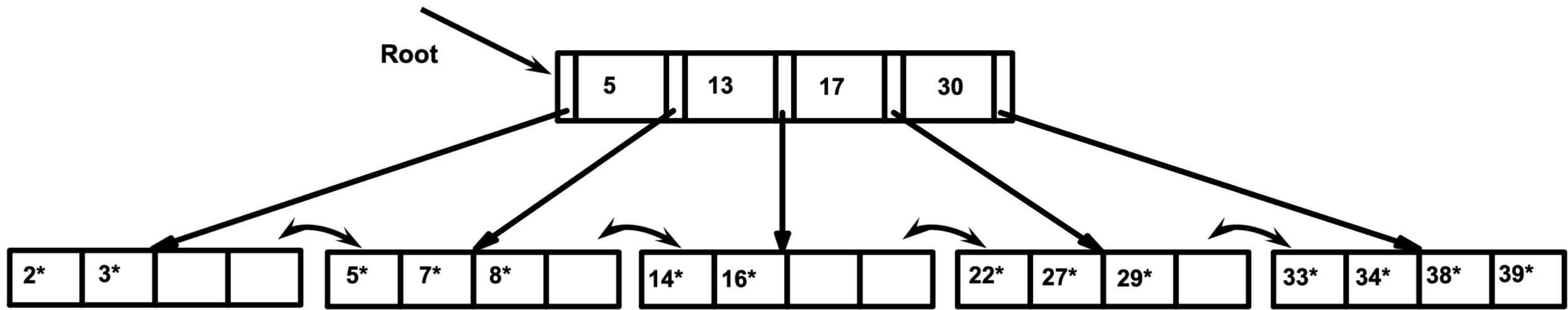
# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory



After deleting 19, 20 (*Redistribution was performed*)

Now Delete 24

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory



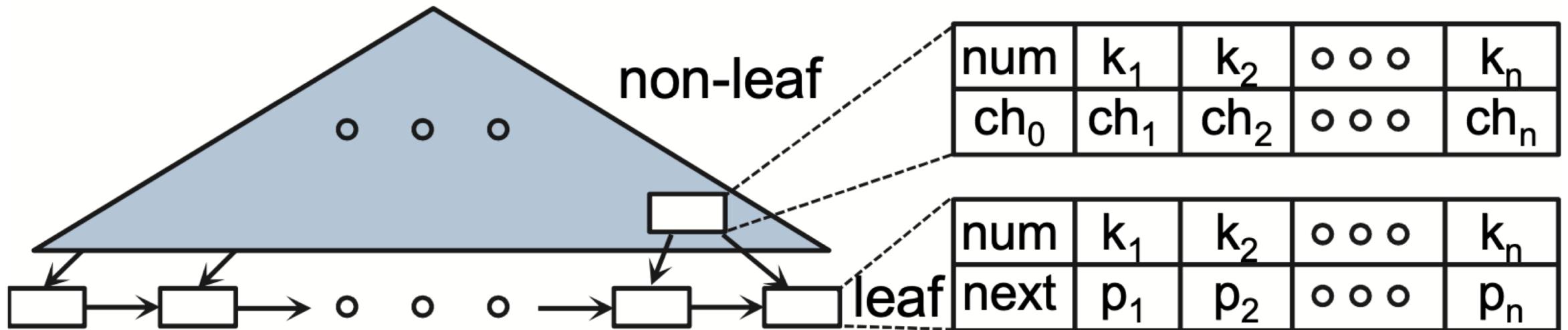
After deleting 24 (*Merging* was performed)

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory

---

- Leaf nodes are connected by sibling pointers
- For disk-based B<sup>+</sup> Trees, the node size is a few disk pages (e.g., 4KB–256KB)
- The node of main-memory B<sup>+</sup> Trees is typically a few cache lines large (e.g., 2–8 64-byte cache lines)

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory



**Figure 1: The main-memory B<sup>+</sup>-Tree structure.**

A non-leaf node contains  $n$  keys and  $n + 1$  child pointers.

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory

---

**A non-leaf node contains  $n$  keys and  $n + 1$  child pointers.**

Suppose each tree node is eight 64-byte cache lines large. If the keys are 8-byte integers in a 64-bit system, then what is the value of  $n$ ?

$$(2n+1)8 = 64 \cdot 8$$

$$\Rightarrow n = 31$$

- A non-leaf node can hold 31 8-byte keys, 32 8-byte child pointers, and a number field.
- A leaf node has space for 31 8-byte keys, 31 8-byte record pointers, a number field, and an 8-byte sibling pointer.

# Persistent B<sup>+</sup> Trees in Non-Volatile Main Memory

---

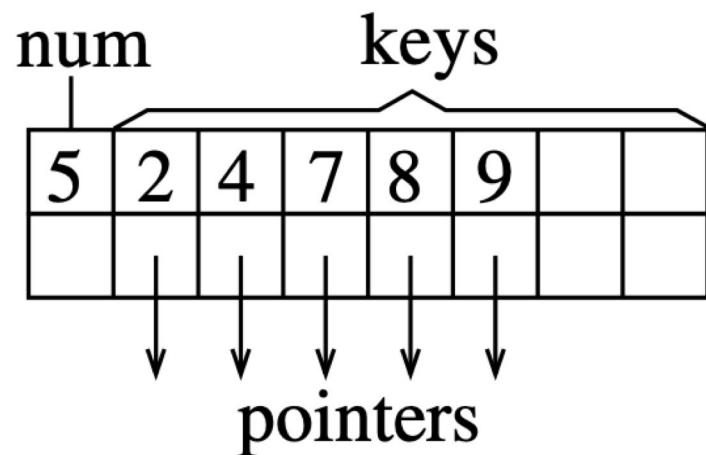
## Key Assumptions

- NVM chip can guarantee atomic writes to aligned 8-byte words
- Both key and value/record pointer are 8 byte
- If not otherwise mentioned, each node is eight 64-byte cache lines large

# PCM-friendly B<sup>+</sup> tree

---

Chen et al. proposed PCM-friendly B<sup>+</sup>-Tree

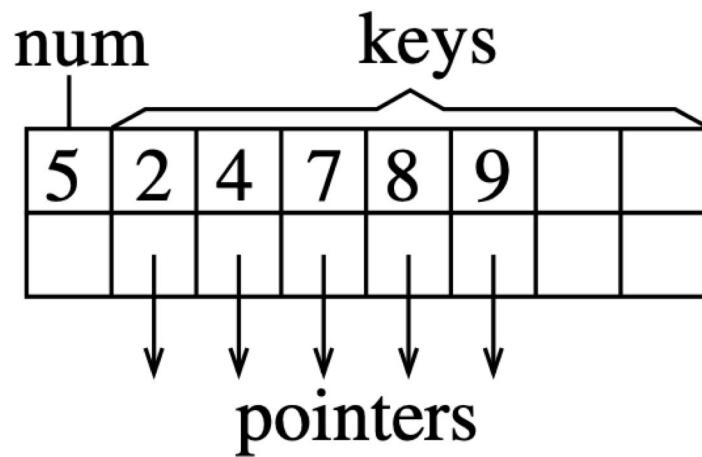


(a) Sorted

# PCM-friendly B<sup>+</sup> tree

---

Chen et al. proposed PCM-friendly B<sup>+</sup>-Tree

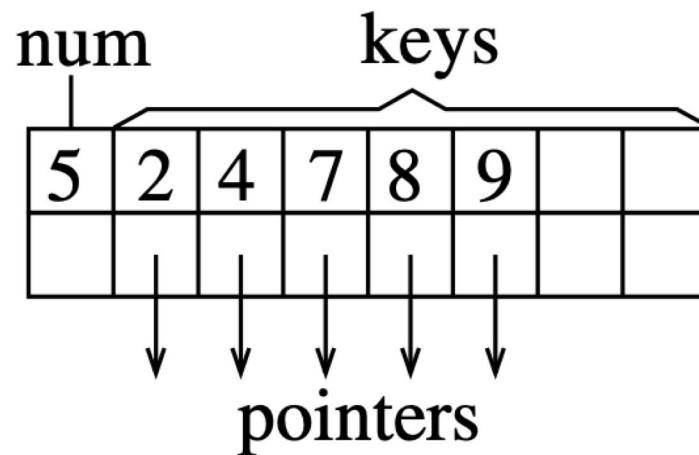


(a) Sorted

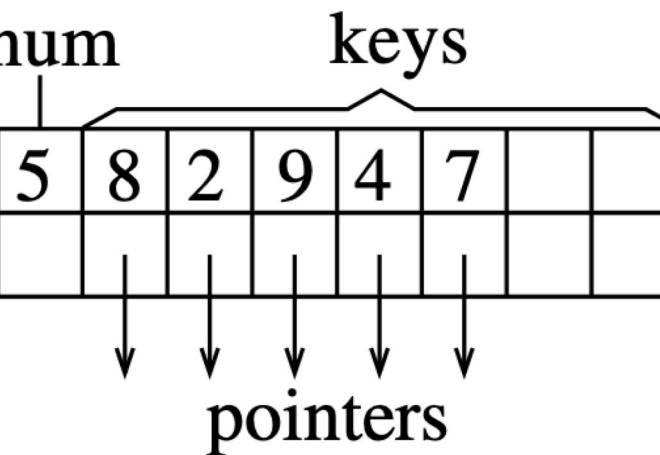
(b) Unsorted

# PCM-friendly B<sup>+</sup> tree

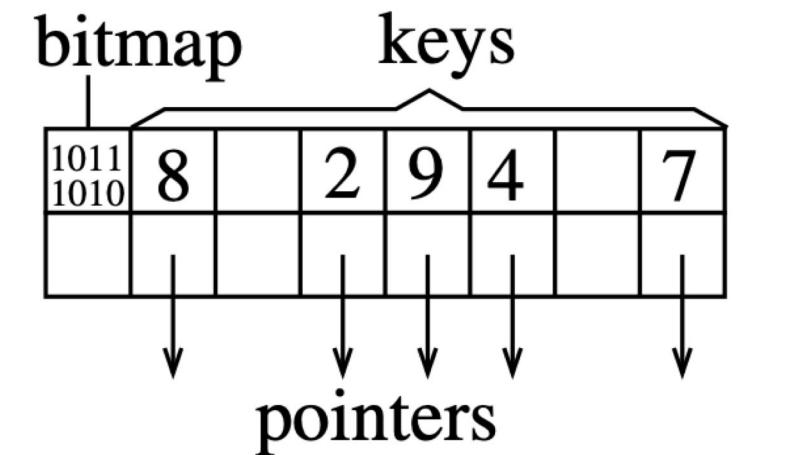
Chen et al. proposed PCM-friendly B<sup>+</sup>-Tree



(a) Sorted



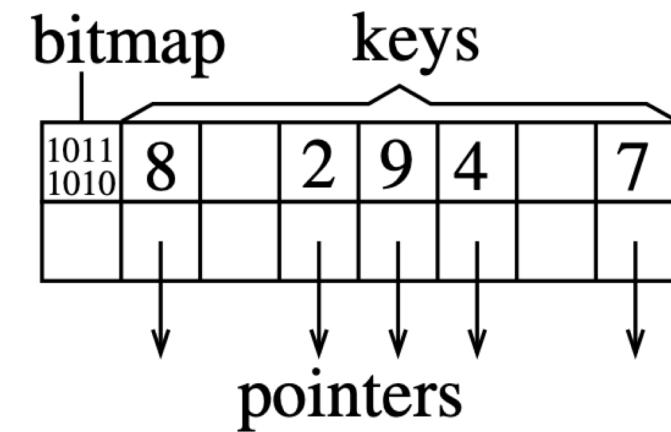
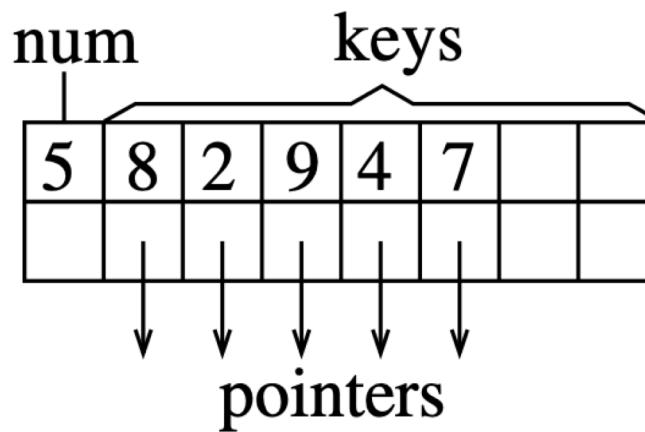
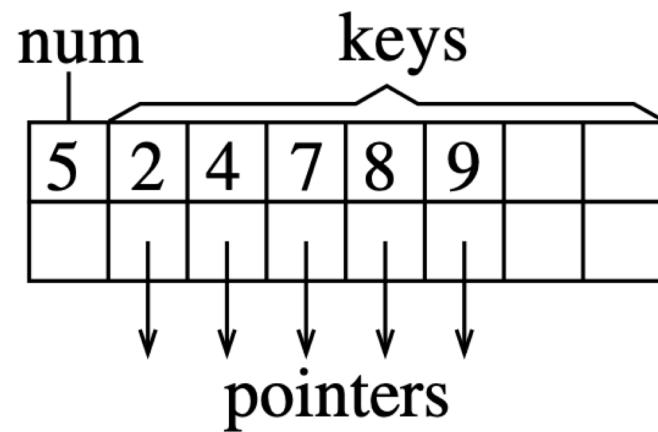
(b) Unsorted



(c) Unsorted w/ bitmap

# PCM-friendly B<sup>+</sup> tree

- **Sorted non-leaf nodes and unsorted leaf nodes with bitmap**
- Requires linear search, but reduces number of NVM writes.



(a) Sorted

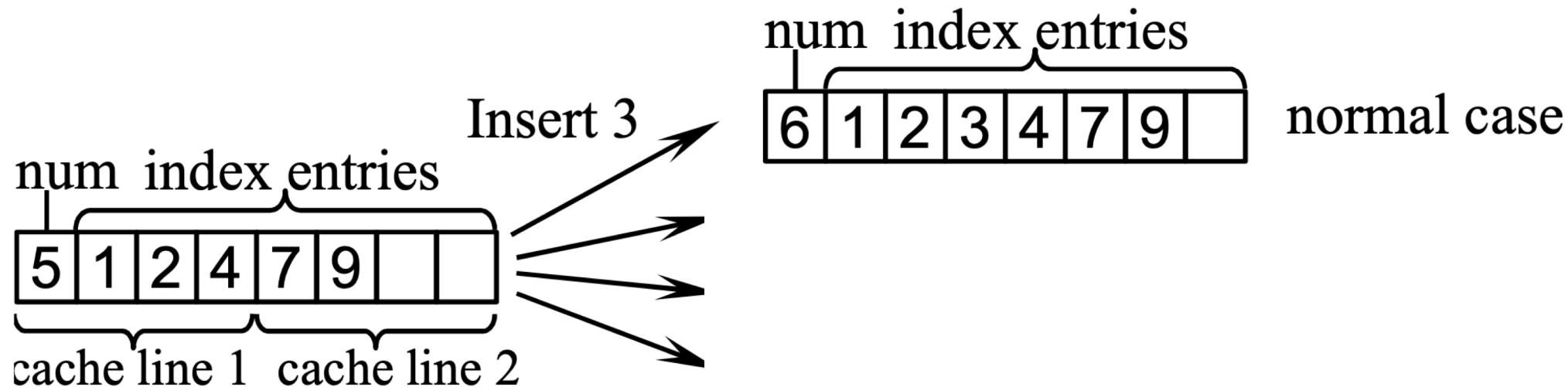
(b) Unsorted

(c) Unsorted w/ bitmap

# Data Structure Inconsistency Problem

Normal sequence of actions:

- Move 9, 7 and 4 one slot to the right
- Insert 3
- Increment the number field

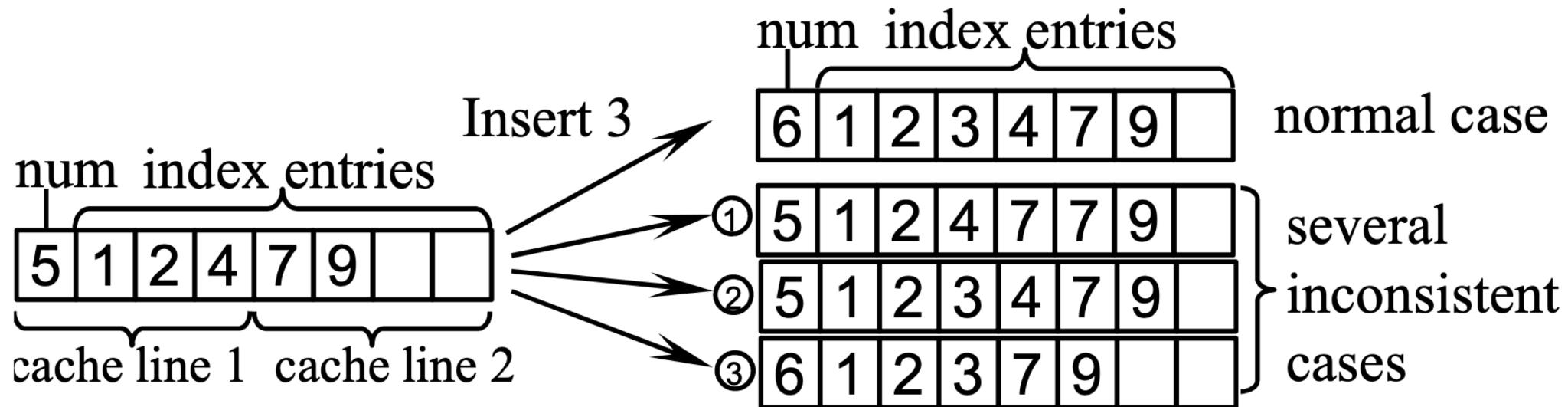


**Figure 3: Potential inconsistencies upon failure.**

# Data Structure Inconsistency Problem

Normal sequence of actions:

- Move 9, 7 and 4 one slot to the right
- Insert 3
- Increment the number field



**Figure 3: Potential inconsistencies upon failure.**

# Challenge: CPU Cache Hierarchy

---

- Limited control over CPU cache
- Can **NOT** guarantee when and in which order dirty cache line is written

## clflush and mfence Instructions

---

clflush: invalidates the cache line on all levels of cache and broadcasts invalidation to all CPUs.

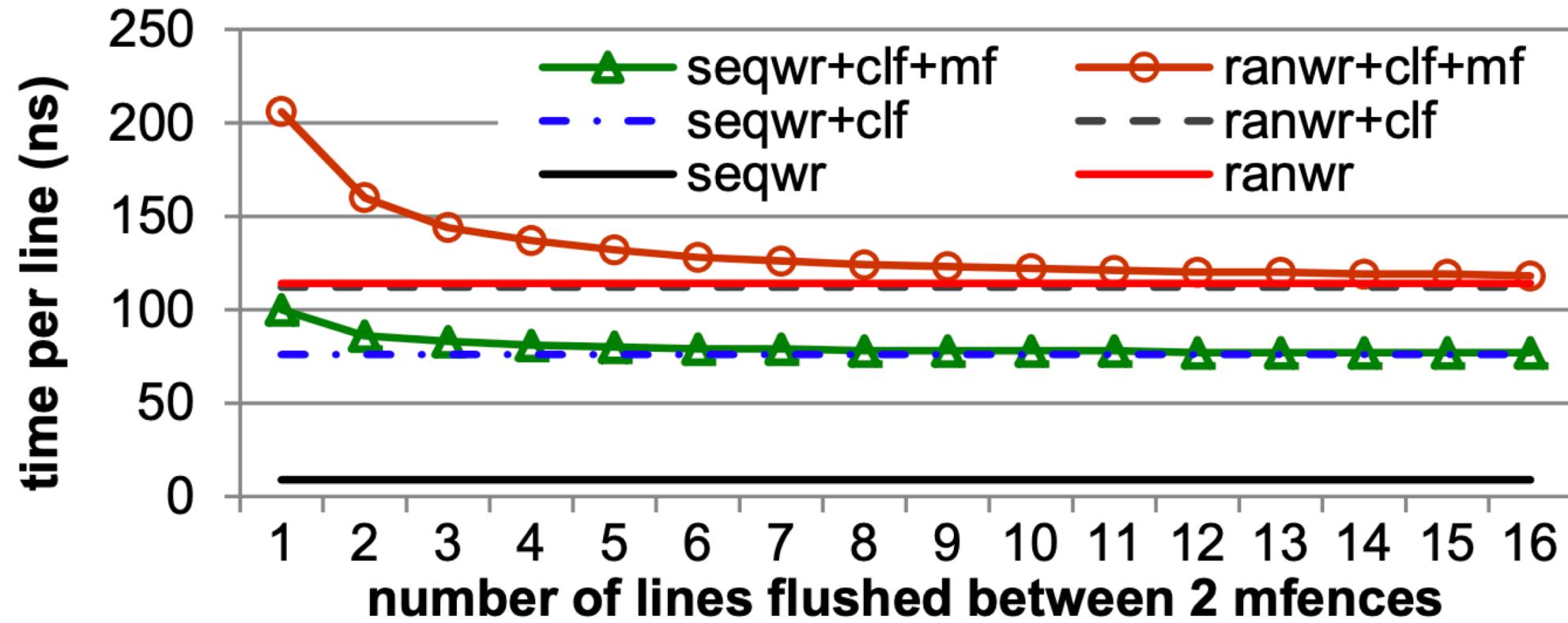
## clflush and mfence Instructions

---

mfence: guarantees that all writes and reads that happened before mfence are globally visible before any of writes or reads that happen after mfence.

# clflush and mfence Instructions

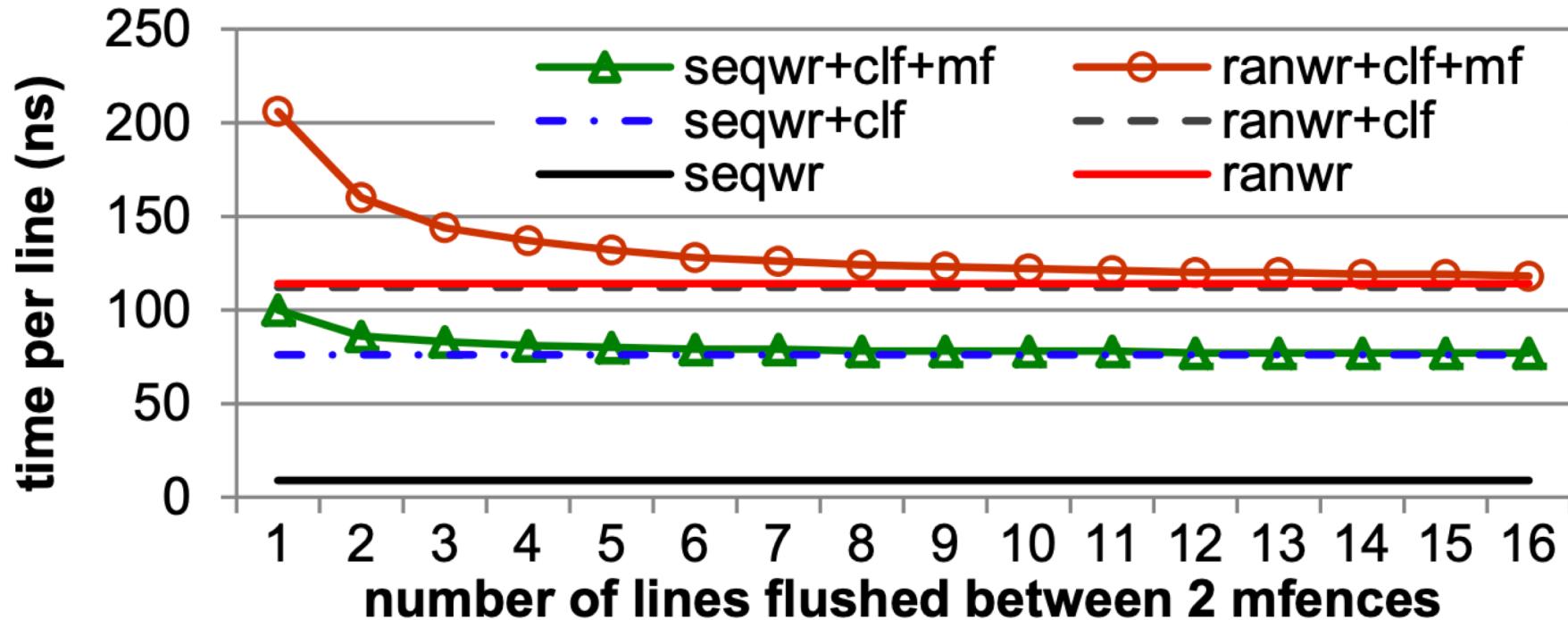
---



**Figure 4: Performance impact of clflush and mfence instructions on sequential and random writes on a real machine.**

# clflush and mfence Instructions

---



- cflush significantly slows down sequential writes
- cflush has negligible impact on random writes
- Reducing the relative frequency of mfence is desirable

# Metrics for Persistent Data Structures

---

$N_w$  – number of writes

$N_{clf}$  – number of cflush

$N_{mf}$  – number of mfence

# Challenges

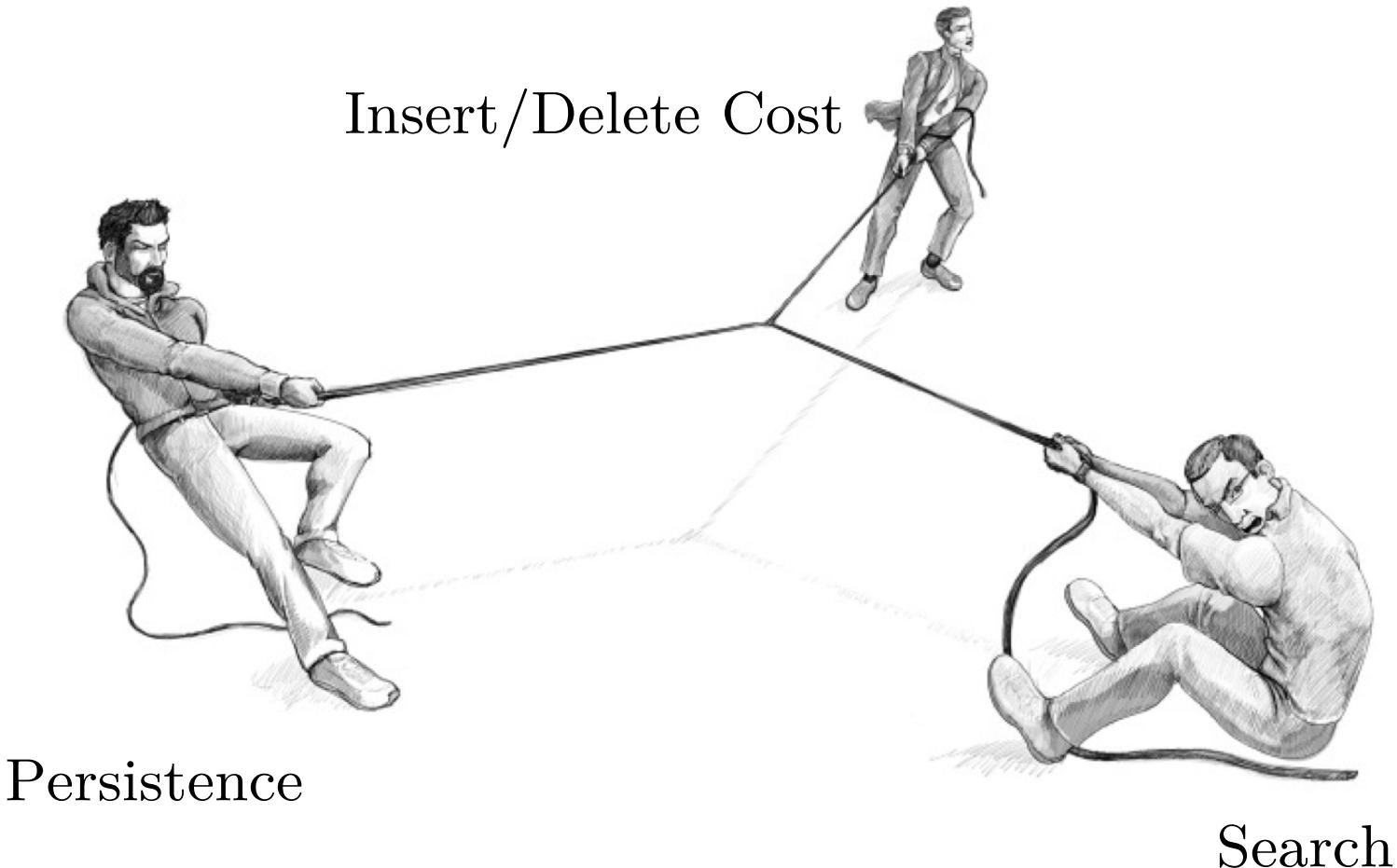
---

- Limited control of CPU cache
  - Can **NOT** guarantee when and in which order dirty cache line is written
  - Needs special CPU instructions (clflush and mfence) which have **non-trivial overhead**
- Different NVM technologies have different characteristics

*Undo-Redo Logging and Shadowing* can both incur drastic overhead because of extensive additional NVM writes and cache line flush instructions.

# Challenges

---



# Existing Solutions

# Existing Solution

---

- Logging
- Shadowing

# Undo-Redo Logging

---

```
1: procedure WRITEUNDOREDO(addr,newValue)
2:   log.write (addr, *addr, newValue);
3:   log.clflush_mfence ();
4:   *addr= newValue;
5: end procedure
```

- One clflush and a mfence per NVM write
- Multiple NVM writes to the log (three extra 8-byte writes for each 8-byte update)

# NewRedo and CommitNewRedo

---

```
1: procedure WRITEUNDOREDO(addr,newValue)
2:   log.write (addr, *addr, newValue);
3:   log.clflush_mfence ();
4:   *addr= newValue;
5: end procedure
6: procedure NEWREDO(addr,newValue)
7:   log.write (addr, newValue);
8:   *addr= newValue;
9: end procedure
10: procedure COMMITNEWREDO
11:   log.clflush_mfence ();
12: end procedure
```

- Writes in unused location, thus reducing some overhead
- Fewer clflush and mfence call
- Failure recovery easier because of using unused location.

# Redo-Only Logging

---

**It is applicable only if a newly written value is not to be accessed again before commit.**

---

```
1: procedure WRITEREDOONLY(addr,newValue)
2:   log.write (addr, newValue);
3: end procedure
4: procedure COMMITREDOWRITES
5:   log.clflush_mfence ();
6:   for all (addr,newValue) in log do
7:     *addr= newValue;
8:   end for
9: end procedure
```

---

**Figure 6: Redo-only logging.**

# Terminology

---

**Table 1: Terms used in analyzing persistent data structures.**

Term	Description
$N_w$	Number of words written to NVMM
$N_{clf}$	Number of cache line flush operations
$N_{mf}$	Number of memory fence operations
$n$	Total number of entries in a B <sup>+</sup> -Tree node
$n'$	Total number of entries in a wB <sup>+</sup> -Tree node
$m$	Number of valid entries in a tree node
$l$	Number of levels of nodes that are split in an insertion

# Insertion Cost Analysis of Logging (Basic B<sup>+</sup> Tree)

---

```
1: procedure WRITEUNDOREDO(addr,newValue)
2:   log.write (addr, *addr, newValue);
3:   log.clflush_mfence ();
4:   *addr= newValue;
5: end procedure
```

## Sorted leaf B<sup>+</sup> tree ([without node splits](#)) using Undo-Redo Logging

- Moves on avg  $m/2$  entries, inserts new entry, and increments the number.
- This requires writing  $m + 3$  words (Each entry = 2 words)
- For each word write, Undo-Redo incurs 3 extra writes, a clflush and mfence.

$$N_w = 4m + 12$$

$$N_{clf} = N_{mf} = m + 3$$

# Insertion Cost Analysis of Logging (PCM-Friendly B<sup>+</sup> Tree)

---

```
1: procedure WRITEUNDOREDO(addr,newValue)
2:   log.write (addr, *addr, newValue);
3:   log.clflush_mfence ();
4:   *addr= newValue;
5: end procedure
6: procedure NEWREDO(addr,newValue)
7:   log.write (addr, newValue);
8:   *addr= newValue;
9: end procedure
10: procedure COMMITNEWREDO
11:   log.clflush_mfence ();
12: end procedure
```

For both packed unsorted and with bitmap,

- Writes the new index entry to an unused location using NewRedo
- Updates the number/bitmap using WriteUndoRedo

$$N_w = 2*3 + 1*4 = 10$$

$$N_{clf} = N_{mf} = 2$$

# Deletion Cost Analysis of Logging (PCM-Friendly B<sup>+</sup> Tree)

---

```
1: procedure WRITEUNDOREDO(addr,newValue)
2:   log.write (addr, *addr, newValue);
3:   log.clflush_mfence ();
4:   *addr= newValue;
5: end procedure
6: procedure NEWREDO(addr,newValue)
7:   log.write (addr, newValue);
8:   *addr= newValue;
9: end procedure
10: procedure COMMITNEWREDO
11:   log.clflush_mfence ();
12: end procedure
```

- For a packed unsorted leaf node, a deletion needs to move the last entry to fill the hole which must use WriteUndoRedo. Hence,  $N_w = 3*4 = 12$
- For an unsorted leaf node with bitmap, only the bitmap needs to be overwritten. Hence,  $N_w = 4$ .

# Shadowing

---

- Create copy of the node, update copy, flush copy, and commit. Update node's parent pointer as well.
- Propagate the same procedure to root.
- **Short-circuit Shadowing:** Use atomic in-place write instead of copying
- Problem: What about leaf sibling pointer?



Use `clflush` and `mfence` to solve this problem

# Shadowing

---

```
1: procedure INSERTTOLEAF(leaf,newEntry,parent,ppos,sibling)
2:   copyLeaf= AllocNode();
3:   NodeCopy(copyLeaf, leaf);
4:   Insert(copyLeaf, newEntry);
5:   for i=0; i < copyLeaf.UsedSize(); i+=64 do
6:     clflush(&copyleaf + i);
7:   end for
8:   WriteRedoOnly(&parent.ch[ppos], copyLeaf);
9:   WriteRedoOnly(&sibling.next, copyLeaf);
10:  CommitRedoWrites();
11:  FreeNode(leaf);
12: end procedure
```

**Figure 7: Shadowing for insertion when there is no node splits.**

---

```
1: procedure WRITEREDOONLY(addr,newValue)
2:   log.write (addr, newValue);
3: end procedure
4: procedure COMMITREDOWRITES
5:   log.clflush_mfence ();
6:   for all (addr,newValue) in log do
7:     *addr= newValue;
8:   end for
9: end procedure
```

**Figure 6: Redo-only logging.**

# Insertion Cost Analysis of Shadowing (Basic B<sup>+</sup> Tree)

---

- $2m + 4$  writes for copying the entries, the number field, and the sibling pointer field, and inserting the new entry.
- The two WriteRedoOnlys require 4 word writes, and the actual pointer updates require 2 writes.
- AllocNode will require an additional log write, clflush, and mfence to ensure persistence of the allocation operation.

$$N_w = 2m + 11$$

$$N_{clf} = (2m + 4)\frac{8}{64} + 1 + 1 = 0.25m + 2.5$$

$$N_{mf} = 2$$

## Cost Analysis of Shadowing (PCM Friendly B<sup>+</sup> Tree)

---

- Since shadowing requires copying the whole node, unsorted leaves do not provide advantage.
- Deletion cost is similar

# Write-Atomic B<sup>+</sup> Trees (wB<sup>+</sup> trees)

# Design Goals

---

- Atomic write to commit all changes
- Minimize the movement of index entries
- Good search performance

# wB<sup>+</sup> Trees

---

## Previous Proposal

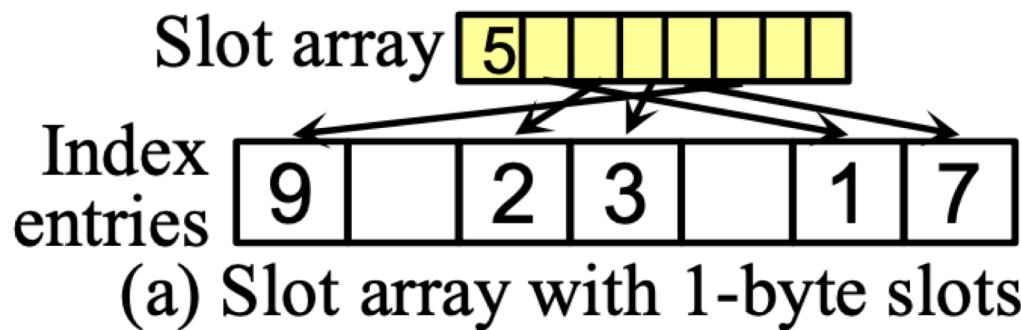
bmp	$k_1$	$k_2$	○○○	$k_n$
next	$p_1$	$p_2$	○○○	$p_n$

(b) Bitmap-only leaf

- Write atomicity possible if bitmap size is less than 8-byte word
- Binary search impossible because of unsortedness

Can we achieve both write atomicity and good search performance?

# Slot Array



slot array

bmp	$k_1$	$\circ \circ \circ$	$k_{n'}$
$ch_0$	$ch_1$	$\circ \circ \circ$	$ch_{n'}$

(e) Slot+bitmap nonleaf

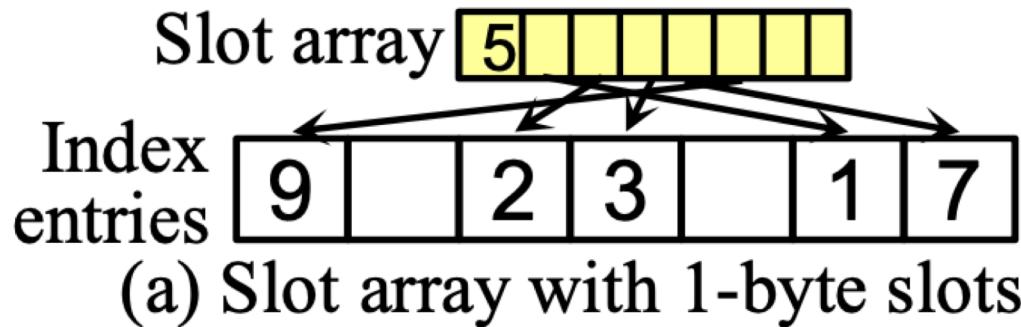
  

bmp	$k_1$	$\circ \circ \circ$	$k_{n'}$
next	$p_1$	$\circ \circ \circ$	$p_{n'}$

(f) Slot+bitmap leaf

- A small indirection array to a bitmap-only unsorted node
- The indirection slot array remembers the sorted order
- Slot 0 records the number of valid entries in the node.
- Lowest bit of the bitmap to indicate whether the slot array is valid.

# Slot Array: Optimization



slot-array	[ 5   5   5   5   5 ]	$k_1$	$k_2$	$\circ \circ \circ$	$k_n$
	$ch_0$	$ch_1$	$ch_2$	$\circ \circ \circ$	$ch_n$

(c) Slot-only nonleaf ( $n < 8$ )

slot-array	[ 5   5   5   5   5 ]	$k_1$	$k_2$	$\circ \circ \circ$	$k_n$
	next	$p_1$	$p_2$	$\circ \circ \circ$	$p_n$

(d) Slot-only leaf ( $n < 8$ )

- If the tree node size is small (number of index entry  $< 8$ ), no need for bitmap. Entire slot array can fit into 8-byte word.
- Write atomicity can be achieved

# wB<sup>+</sup> Trees in This Paper

---

**Table 2: wB<sup>+</sup>-Tree structures considered in this paper.**

Structure	Leaf Node	Non-leaf Node
wB <sup>+</sup> -Tree	slot+bitmap leaf	slot+bitmap non-leaf
wB <sup>+</sup> -Tree w/ bitmap-only leaf	bitmap-only leaf	slot+bitmap non-leaf
wB <sup>+</sup> -Tree w/ slot-only nodes	slot-only leaf	slot-only non-leaf

} When the node size is large  
When the node size is small

# Space Overhead Analysis

---

- An 8-byte bitmap can support up to 63 index entries
- 1-byte sized slots can support up to 255 index entries

If an index entry is 16-byte large (with 8-byte keys and 8-byte pointers), then a slot + bitmap node can be as large as 1KB (16 cache lines)

# Insertion

---

- Find insert position
- Find unused position
- Write
- cflush and mfence
- Generate up-to-date slot array
- Atomic write to update slot array

$$N_w = 3$$

$$N_{clf} = N_{mf} = 2$$

Similar algorithm for bitmap-only nodes

```
1: procedure INSERT2SLOTONLY_ATOMIc(leaf, newEntry)
2:   /* Slot array is valid */
3:   pos= leaf.GetInsertPosWithBinarySearch(newEntry);
4:   /* Write and flush newEntry */
5:   u= leaf.GetUnusedEntryWithSlotArray();
6:   leaf.entry[u]= newEntry;
7:   clflush(&leaf.entry[u]); mfence();
8:   /* Generate an up-to-date slot array on the stack */
9:   for (j=leaf.slot[0]; j $\geq$ pos; j - -) do
10:    tempslot[j+1]= leaf.slot[j];
11:   end for
12:   tempslot[pos]=u;
13:   for (j=pos-1; j $\geq$ 1; j - -) do
14:    tempslot[j]= leaf.slot[j];
15:   end for
16:   tempslot[0]=leaf.slot[0]+1;
17:   /* Atomic write to update the slot array */
18:   *((UInt64 *)leaf.slot)= *((UInt64 *)tempslot);
19:   clflush(leaf.slot); mfence();
20: end procedure
```

Figure 9: Insertion to a slot-only node with atomic writes.<sup>63</sup>

# Insertion

---

- Recover slot array if it is invalid
- Mark slot array as invalid
- Write and flush new entry
- Modify and flush slot array
- mfence to make new entry and slot array stable
- Update bitmap atomically and mfence

```
1: procedure INSERT2SLOTBMP_ATOMIC(leaf, newEntry)
2:   if (leaf.bitmap & 1 == 0) /* Slot array is invalid? */ then
3:     Recover by using the bitmap to find the valid entries,
4:     building the slot array, and setting the slot valid bit;
5:   end if
6:   pos= leaf.GetInsertPosWithBinarySearch(newEntry);
7:   /* Disable the slot array */
8:   leaf.bitmap = leaf.bitmap - 1;
9:   clflush(&leaf.bitmap); mfence();
10:  /* Write and flush newEntry */
11:  u= leaf.GetUnusedEntryWithBitmap();
12:  leaf.entry[u]= newEntry;
13:  clflush(&leaf.entry[u]);
14:  /* Modify and flush the slot array */
15:  for (j=leaf.slot[0]; j $\geq$ pos; j--) do
16:    leaf.slot[j+1]= leaf.slot[j];
17:  end for
18:  leaf.slot[pos]=u;
19:  for (j=pos-1; j $\geq$ 1; j--) do
20:    leaf.slot[j]= leaf.slot[j];
21:  end for
22:  leaf.slot[0]=leaf.slot[0]+1;
23:  for (j=0; j $\leq$ leaf.slot[0]; j += 8) do
24:    clflush(&leaf.slot[j]);
25:  end for
26:  mfence(); /* Ensure new entry and slot array are stable */
27:  /* Enable slot array, new entry and flush bitmap */
28:  leaf.bitmap = leaf.bitmap + 1 + (1<<u);
29:  clflush(&leaf.bitmap); mfence();
29: end procedure
```

---

Figure 10: Insertion to a slot+bitmap node with atomic writes.

# Search

---

- Usage of slot array allows the logarithmic complexity of the search
- Slot array is especially useful for non-leaf nodes
- Slot array dereference overhead is nontrivial!!!
  - Optimize it by stopping binary search when range is narrow (<8 slot)
  - Retrieve everything in 8-byte integer
  - Use shift and logic operation

# Deletion

---

- Similar to insertion
- No need to move any entries
- Simply update slot array and/or the bitmap
- Either atomic writes or redo-only logging can be employed

# Comparison

---

**Table 3: Comparison of persistent B<sup>+</sup>-Tree solutions.**

Solution	Insertion without node splits	Insertion with $l$ node splits	Deletion without node merges
B <sup>+</sup> -Trees undo-redo logging	$N_w = 4m + 12$ , $N_{clf} = N_{mf} = m + 3$	$N_w = l(4n + 15) + 4m + 19$ , $N_{clf} = l(0.375n + 3.25) + m + 4.125$ , $N_{mf} = l(0.25n + 2) + m + 5$	$N_w = 4m$ , $N_{clf} = N_{mf} = m$
Unsorted leaf undo-redo logging	$N_w = 10$ , $N_{clf} = 2$ , $N_{mf} = 2$	$N_w = l(4n + 15) + n + 4m + 19$ , $N_{clf} = l(0.375n + 3.25) + 0.25n + m + 4.125$ , $N_{mf} = l(0.25n + 2) + 0.25n + m + 5$	$N_w = 12$ , $N_{clf} = 3$ , $N_{mf} = 3$
Unsorted leaf w/ bitmap undo-redo logging	$N_w = 10$ , $N_{clf} = 2$ , $N_{mf} = 2$	$N_w = l(4n + 15) - n + 4m + 19$ , $N_{clf} = l(0.375n + 3.25) - 0.25n + m + 4.125$ , $N_{mf} = l(0.25n + 2) - 0.25n + m + 5$	$N_w = 4$ , $N_{clf} = 1$ , $N_{mf} = 1$
B <sup>+</sup> -Trees shadowing	$N_w = 2m + 11$ , $N_{mf} = 2$ , $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$ , $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$ , $N_{mf} = 2$	$N_w = 2m + 7$ , $N_{mf} = 2$ , $N_{clf} = 0.25m + 2$
Unsorted leaf shadowing	$N_w = 2m + 11$ , $N_{mf} = 2$ , $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$ , $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$ , $N_{mf} = 2$	$N_w = 2m + 7$ , $N_{mf} = 2$ , $N_{clf} = 0.25m + 2$
Unsorted leaf w/ bitmap shadowing	$N_w = 2m + 11$ , $N_{mf} = 2$ , $N_{clf} = 0.25m + 2.5$	$N_w = l(2n + 5) + 2m + 12$ , $N_{clf} = l(0.25n + 1.5) + 0.25m + 2.625$ , $N_{mf} = 2$	$N_w = 2m + 7$ , $N_{mf} = 2$ , $N_{clf} = 0.25m + 2$
wB <sup>+</sup> -Tree	$N_w = 0.125m + 4.25$ , $N_{clf} = \frac{1}{64}m + 3\frac{1}{32}$ , $N_{mf} = 3$	$N_w = l(1.25n' + 9.75) + 0.125m + 8.25$ , $N_{clf} = l(\frac{19}{128}n' + 1\frac{105}{128}) + \frac{1}{64}m + 3\frac{13}{32}$ , $N_{mf} = 3$	$N_w = 0.125m + 2$ , $N_{clf} = \frac{1}{64}m + 2$ , $N_{mf} = 3$
wB <sup>+</sup> -Tree w/ bitmap-only leaf	$N_w = 3$ , $N_{clf} = 2$ , $N_{mf} = 2$	$N_w = l(1.25n' + 9.75) - 0.25n' + 0.125m + 7.5$ , $N_{clf} = l(\frac{19}{128}n' + 1\frac{105}{128}) - \frac{3}{128}n' + \frac{1}{64}m + 3\frac{43}{128}$ , $N_{mf} = 3$	$N_w = 1$ , $N_{clf} = 1$ , $N_{mf} = 1$
wB <sup>+</sup> -Tree w/ slot-only nodes	$N_w = 3$ , $N_{clf} = 2$ , $N_{mf} = 2$	$N_w = l(n + 9) + 7$ , $N_{clf} = l(0.125n + 1.75) + 2.375$ , $N_{mf} = 2$	$N_w = 1$ , $N_{clf} = 1$ , $N_{mf} = 1$

# Comparison

---

	$N_w$	$N_{clf}$	$N_{mf}$	$N_w$	$N_{clf}$	$N_{mf}$
B <sup>+</sup> Tree (log)	$4m + 12$	$m + 3$	$m + 3$	$4m$	$m$	$m$
PCM B <sup>+</sup> w/o bitmap (logging)	10	2	2	12	3	3
PCM B <sup>+</sup> with bitmap (logging)	10	2	2	4	1	1
B <sup>+</sup> Tree (Shadow)	$2m + 11$	$0.25m + 2.5$	2	$2m + 7$	$0.25m+2$	2
wB <sup>+</sup> Tree	$\frac{m}{8} + 4.25$	$\frac{m}{64} + 3\frac{1}{32}$	3	$\frac{m}{8} + 2$	$\frac{m}{64} + 2$	3
wB <sup>+</sup> Tree (bitmap-only)	3	2	2	1	1	1
wB <sup>+</sup> Tree (slot-only)	3	2	2	1	1	1

# wB+ Trees for Variable Sized Keys

---

- Contains 8-byte keys (*key pointers*), which are pointers to the actual variable sized keys.
- A slot+bitmap node has two indirection layers
  - First indirection layer is the key pointers
  - Second indirection layer is the slot array.

# Experimental Evaluations

# Experiment Setup

---

- Real machine modeling DRAM-like fast NVMM
  - Achieve up to 8.8x speedup
- Simulation modeling PCM-based NVMM
  - Achieve up to 27.1x speedup
- Full system experiment: replaced Memcached's internal hash on real machine
  - Achieve up to 3.8x improvements

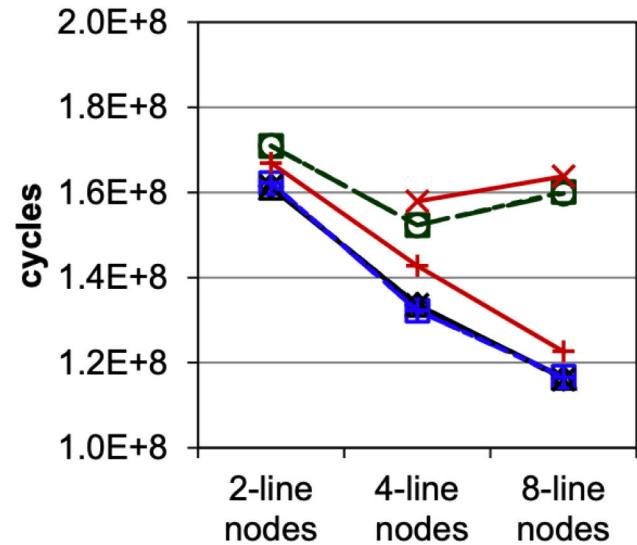
# B<sup>+</sup> Tree Implementations

---

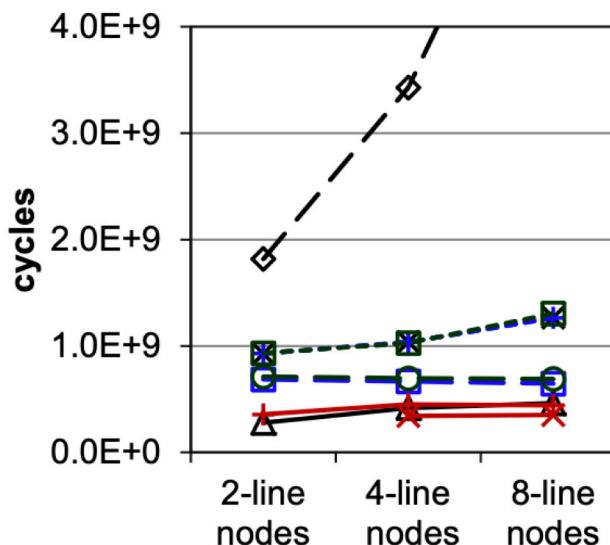
Implemented 9 B<sup>+</sup> tree solutions for fixed-size keys

- btree (volatile)
- btree log: employ undo-redo logging for btree
- unsorted leaf log: employ undo-redo logging for B<sup>+</sup>-Tree with unsorted leaf nodes
- uleaf bmp log: employ undo-redo logging for B<sup>+</sup>-Tree with bitmap-only unsorted leaf nodes
- btree shadow: employ shadowing for btree
- unsorted leaf shadow: employ shadowing for B<sup>+</sup>-Tree with unsorted leaf nodes
- uleafbmp shadow: employ shadowing for B<sup>+</sup>-Tree with bitmap-only unsorted leaf nodes
- wbtree: wb<sup>+</sup> tree
- wbtree w/bmp-leaf : wB -Tree with bitmap-only leaf nodes.

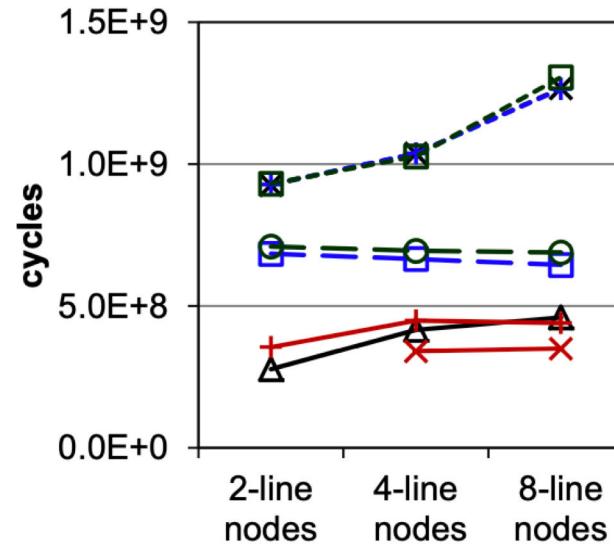
\*\* If the node size  $\leq 2$  cache lines, they used wB<sup>+</sup>-Tree with slot-only nodes to replace both (8) and (9), and report results as wbtree.



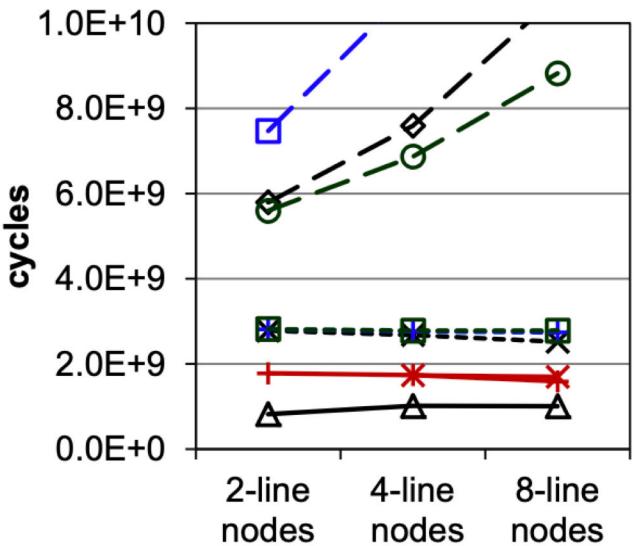
(a) Search, 70% full nodes



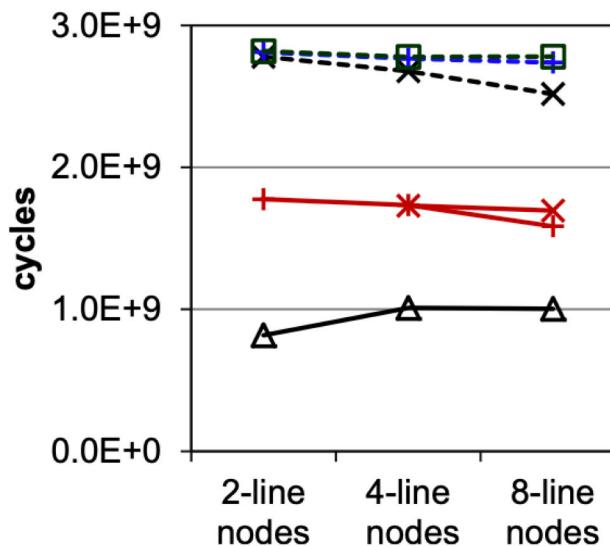
(b) Insertion, 70% full nodes



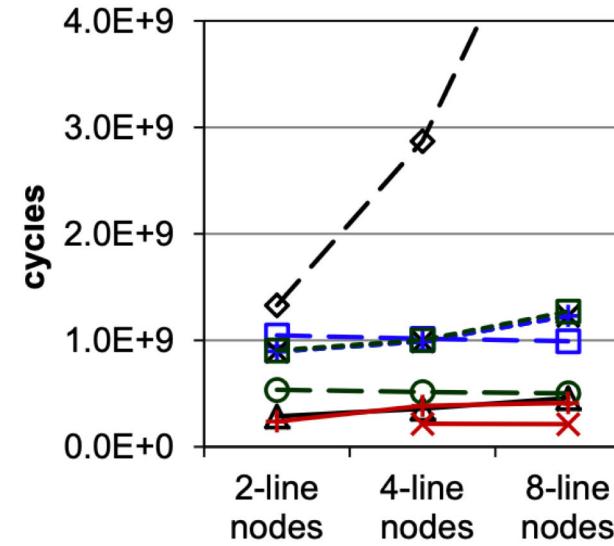
(c) Zoom of (b)



(d) Insertion, 100% full nodes



(e) Zoom of (d)



(f) Deletion, 70% full nodes

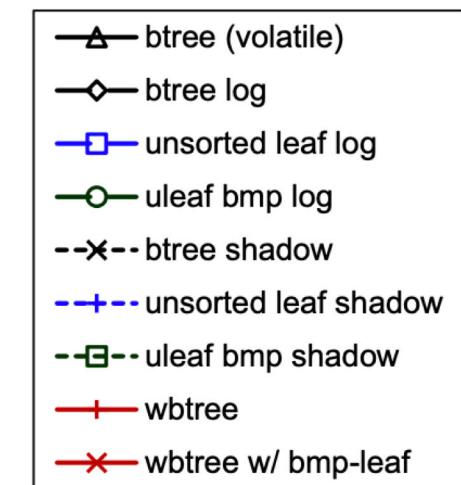
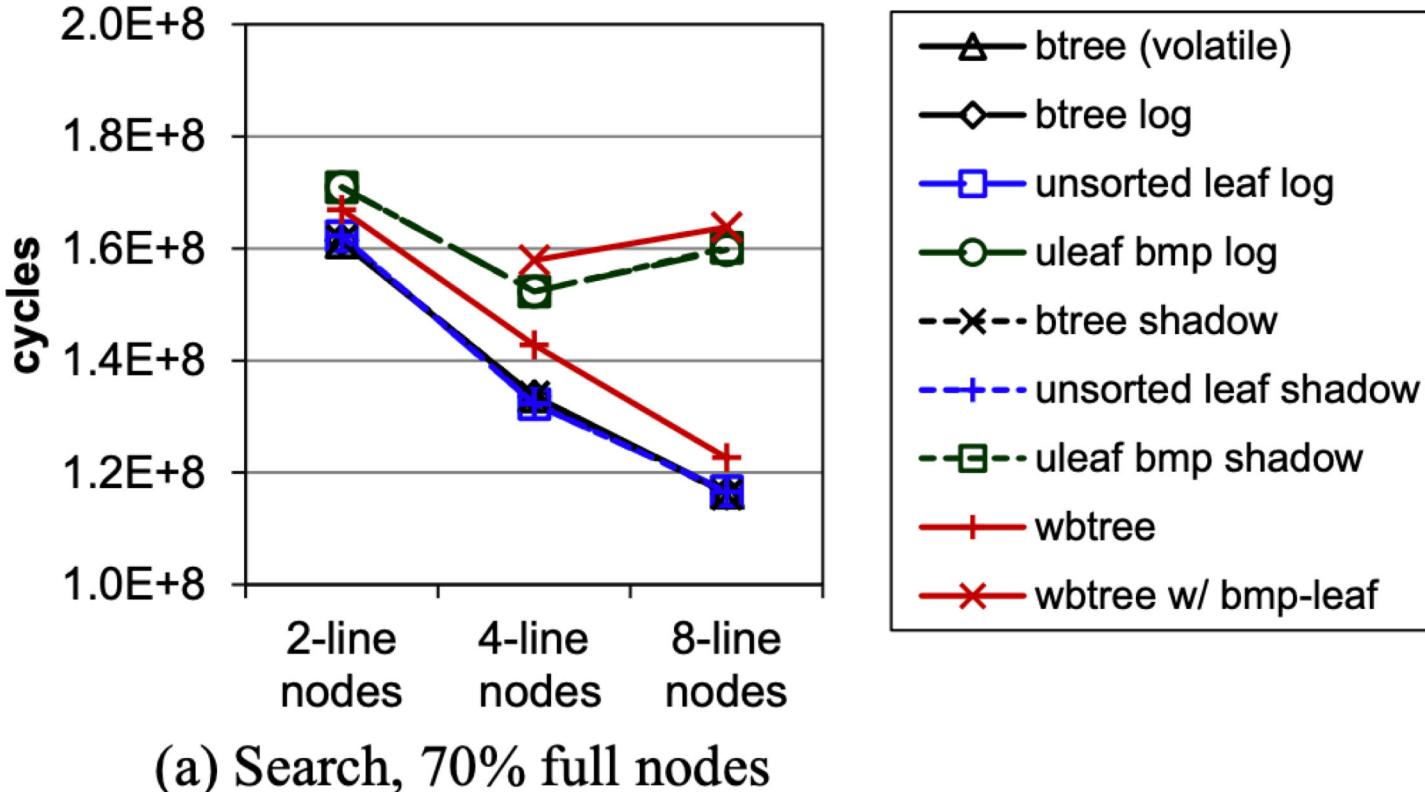


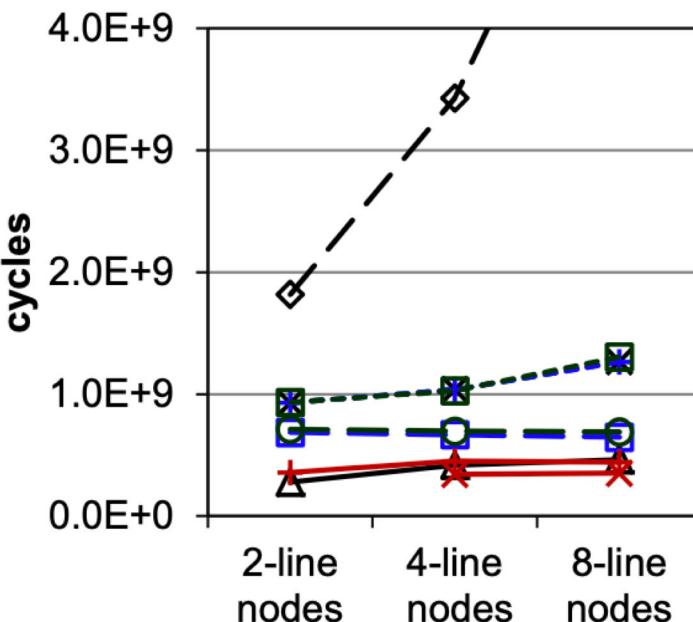
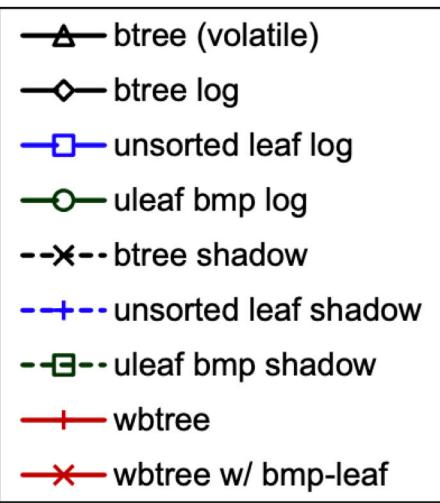
Figure 11: Index performance on a cycle-accurate simulator modeling PCM-based NVMM. (We bulkload a tree with 20M entries, then perform 100K random back-to-back lookups, insertions, or deletions. Keys are 8-byte integers.)

# Experimental Results

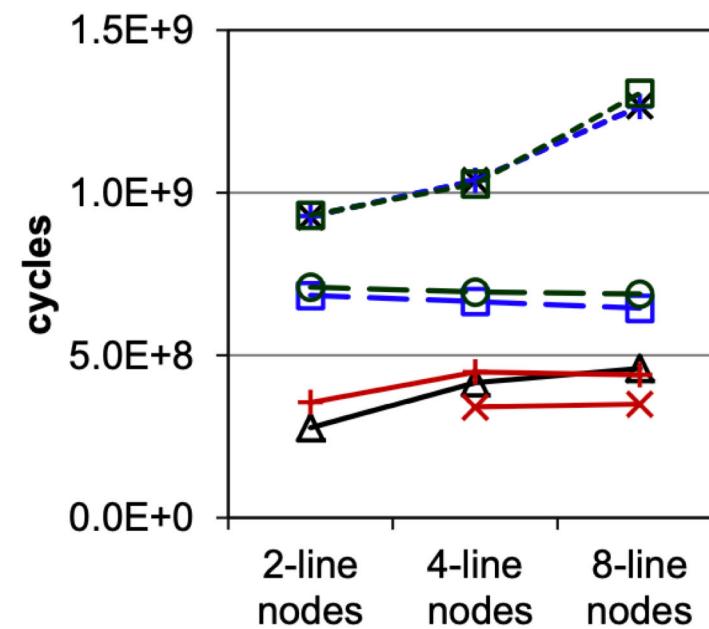


- wB<sup>+</sup>-Tree achieves similar search performance as baseline
- Bitmap-only leaf nodes see up to 16% slowdowns because of the sequential search overhead in leaf nodes

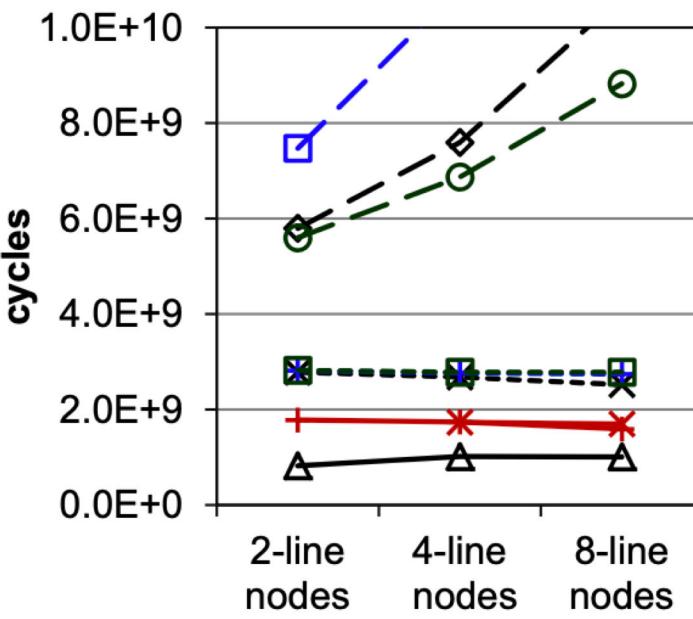
- Undo-redo logging is extremely slow
- Shadowing is also slow
- wbtree w/bmp-leaf achieves slightly better insertion and deletion performance than wbtree, but sees worse search performance.



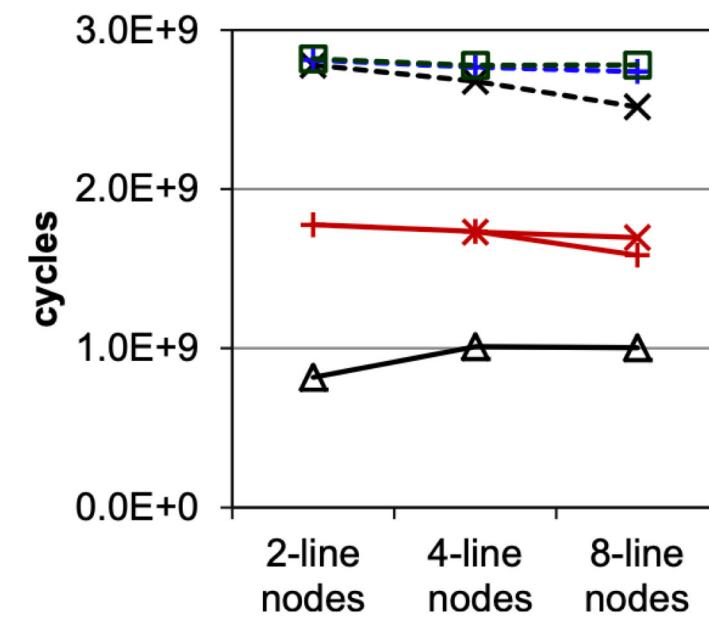
(b) Insertion, 70% full nodes



(c) Zoom of (b)

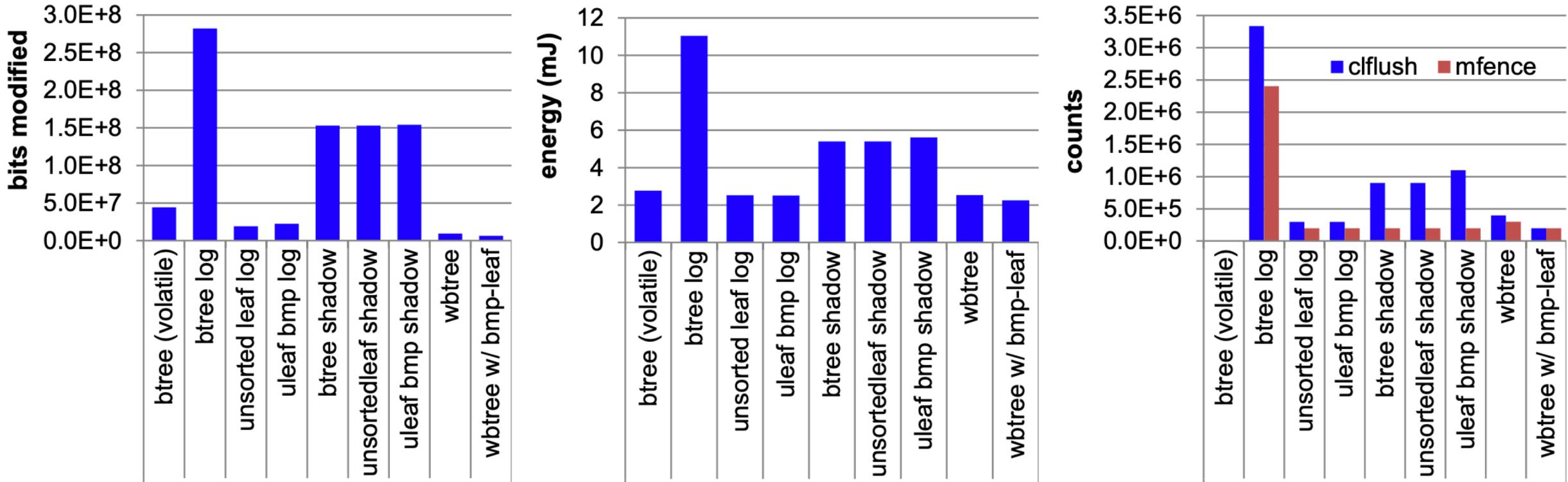


(d) Insertion, 100% full nodes



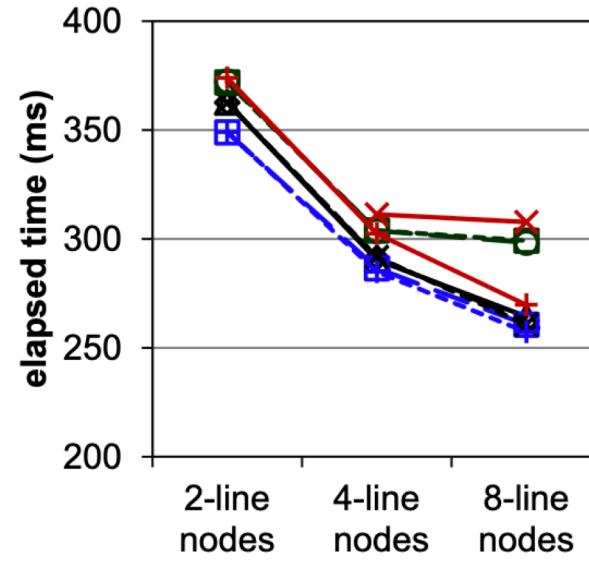
(e) Zoom of (d)

# Experimental Results

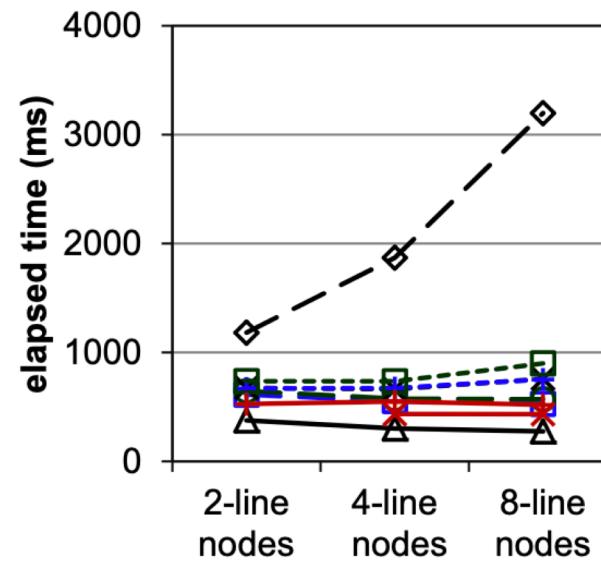


**Figure 12: Wear, energy, and clflush/mfence counts of index operations for Figure 11(b) with 8-line nodes.**

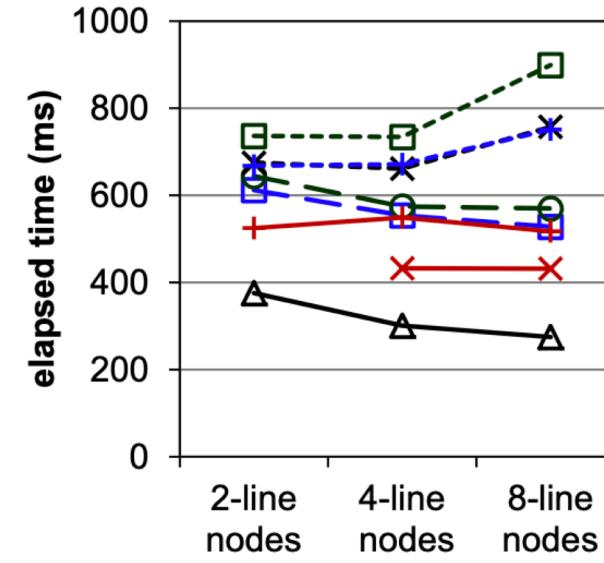
- Bits written for wbtrees are much less
- Slightly higher clflush than PCM-friendly B+ trees.



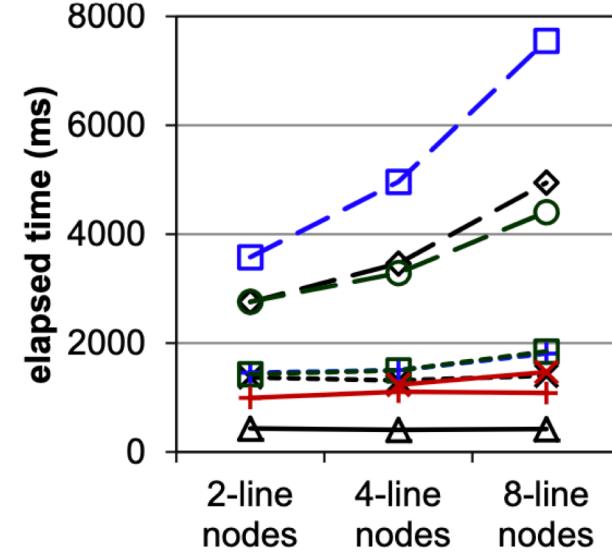
(a) Search, 70% full nodes



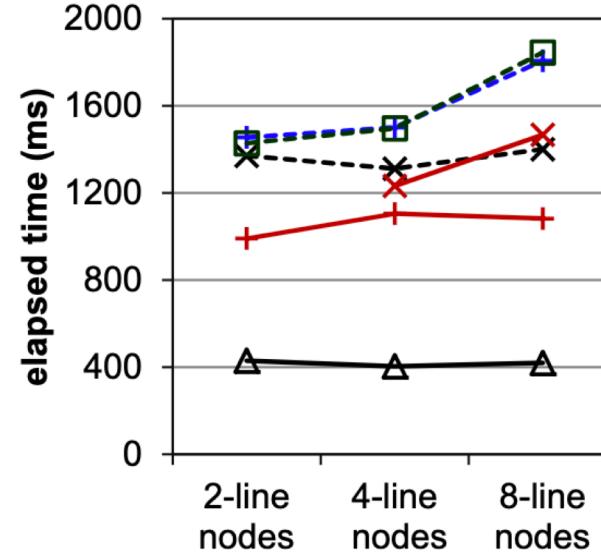
(b) Insertion, 70% full nodes



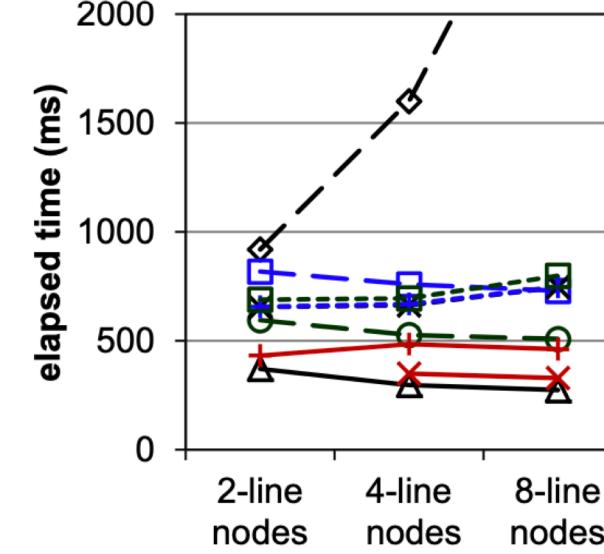
(c) Zoom of (b)



(d) Insertion, 100% full nodes



(e) Zoom of (d)



(f) Deletion, 70% full nodes

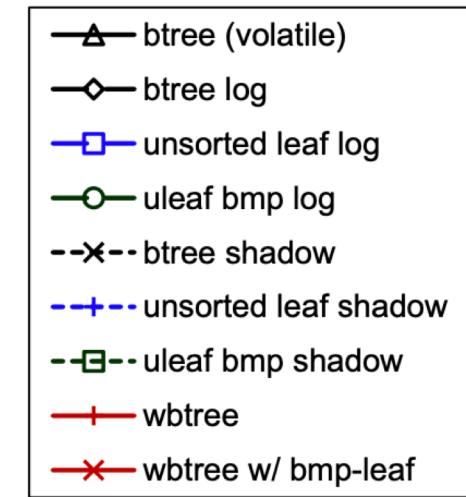


Figure 13: Index performance on a real machine modeling DRAM-like fast NVMM. (We bulkload a tree with 50M entries, then perform 500K random back-to-back lookups, insertions, or deletions. Keys are 8-byte integers.)

# Difference between Real Machine and Simulation

---

- Bar charts of the simulation have similar shape to the **bits modified** charts
- Bar charts of the real machine results have similar shape to the **clflush** charts

PCM writes play a major role in determining the elapsed times on PCM based NVMM  
Cache line flushes are the major factor in elapsed time on fast DRAM-like NVMM

# Experimental Results

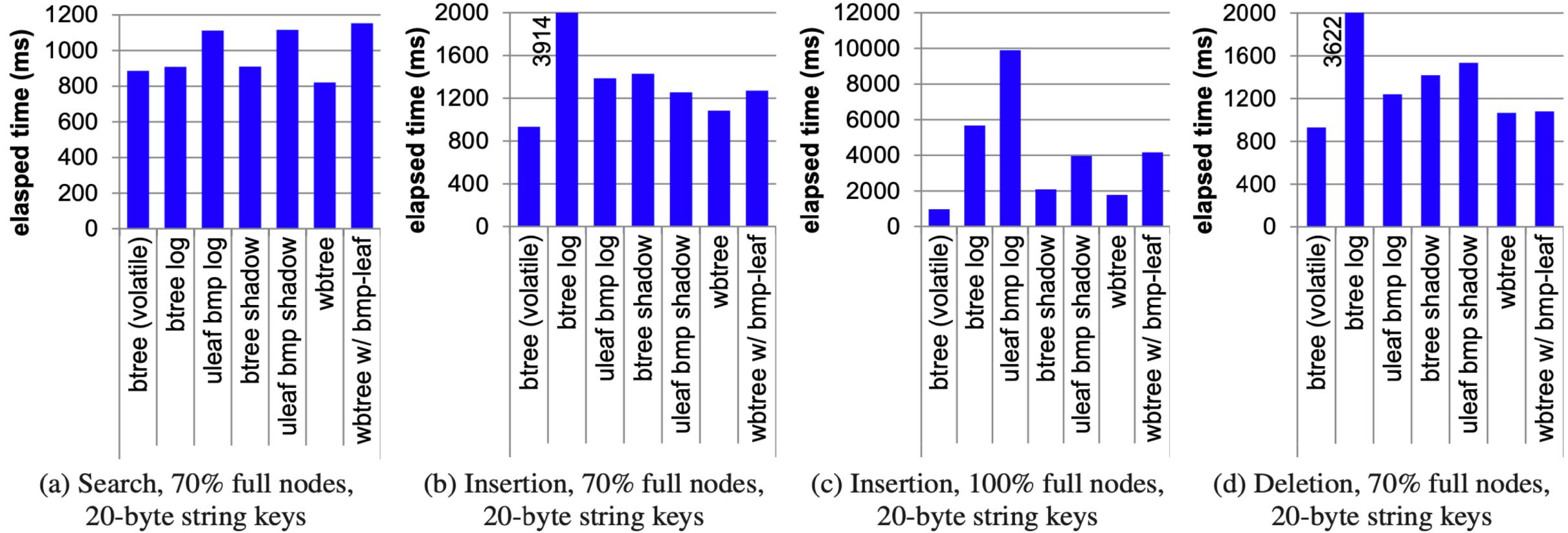
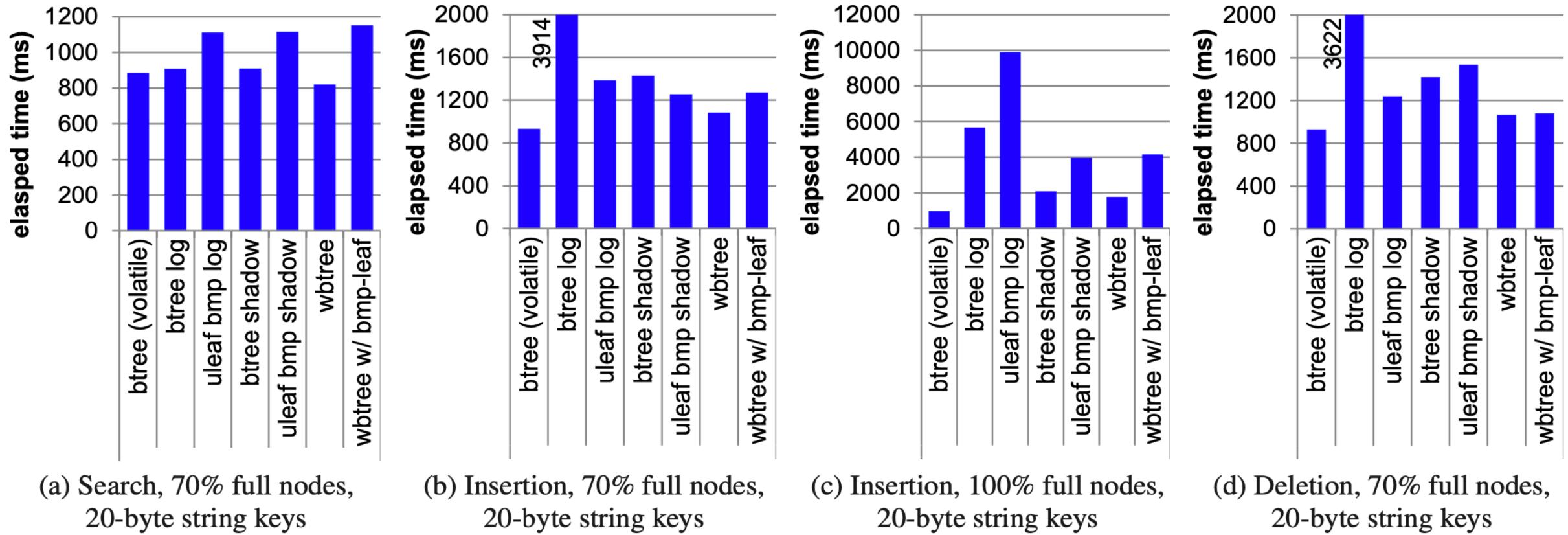


Figure 14: Index performance with string keys on a real machine. (We bulkload a tree with 50M entries, then perform 500K random back-to-back lookups, insertions, or deletions).

- Searches are costly than fixed-size

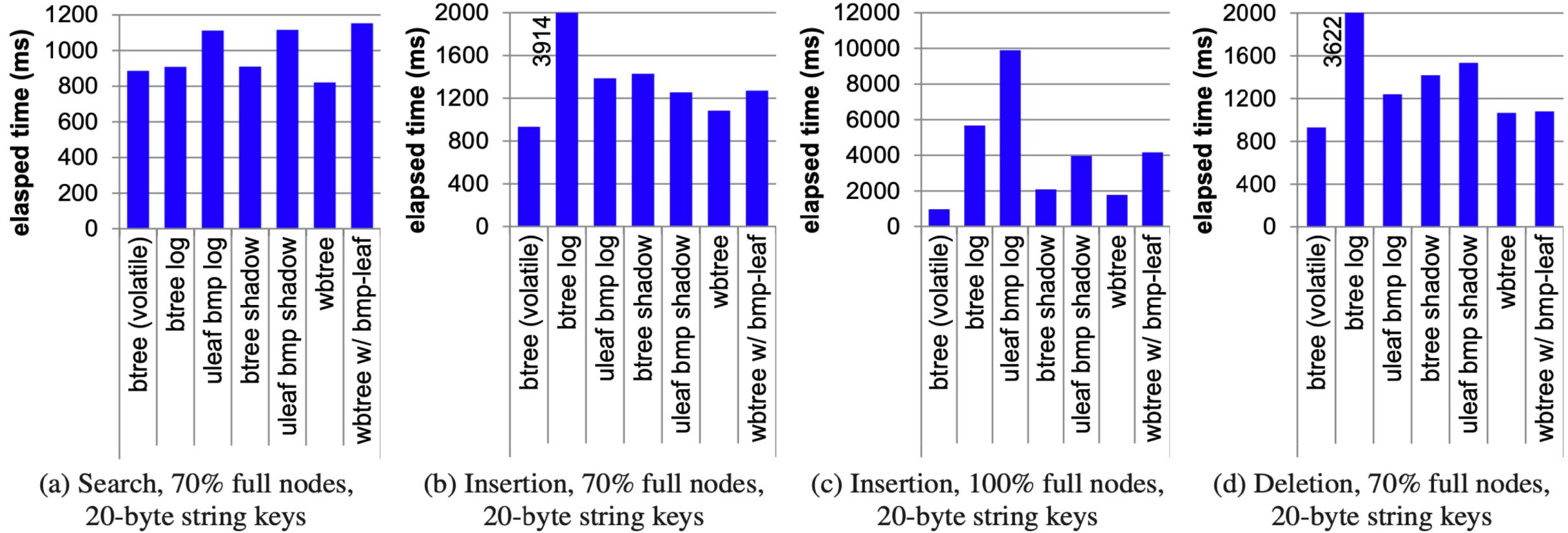
# Experimental Results



**Figure 14: Index performance with string keys on a real machine. (We bulkload a tree with 50M entries, then perform 500K random back-to-back lookups, insertions, or deletions).**

- wbtree is the best persistent tree solution.

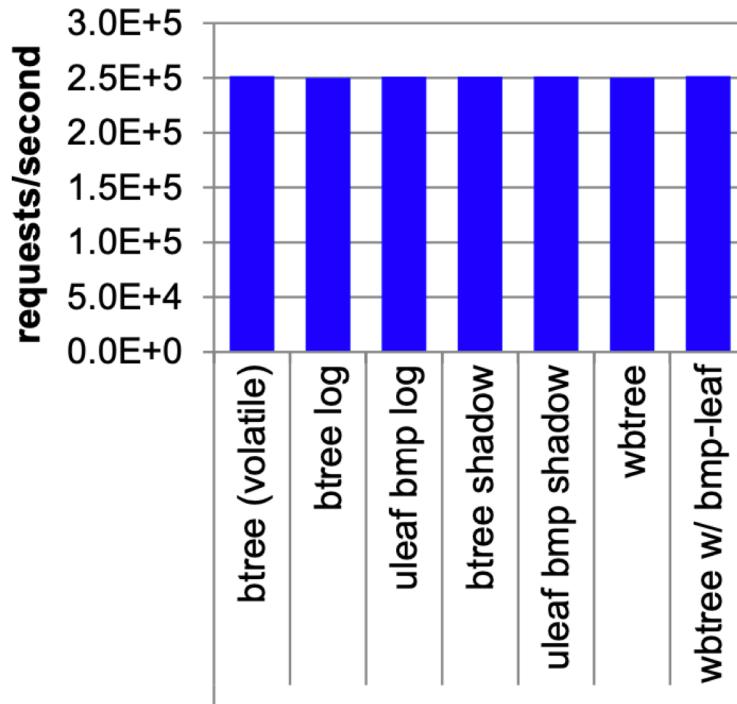
# Experimental Results



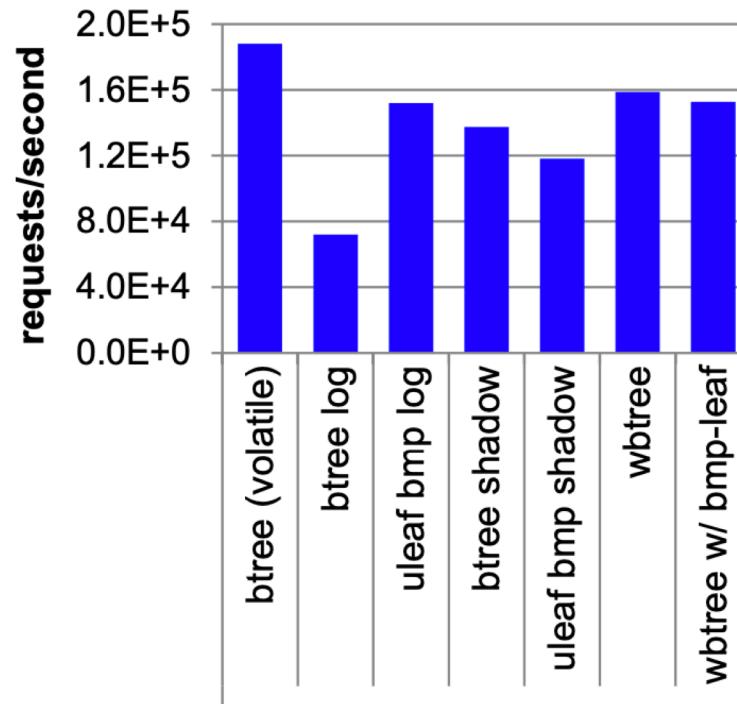
**Figure 14: Index performance with string keys on a real machine. (We bulkload a tree with 50M entries, then perform 500K random back-to-back lookups, insertions, or deletions).**

- wbtree w/bmp-leaf has significantly poorer performance

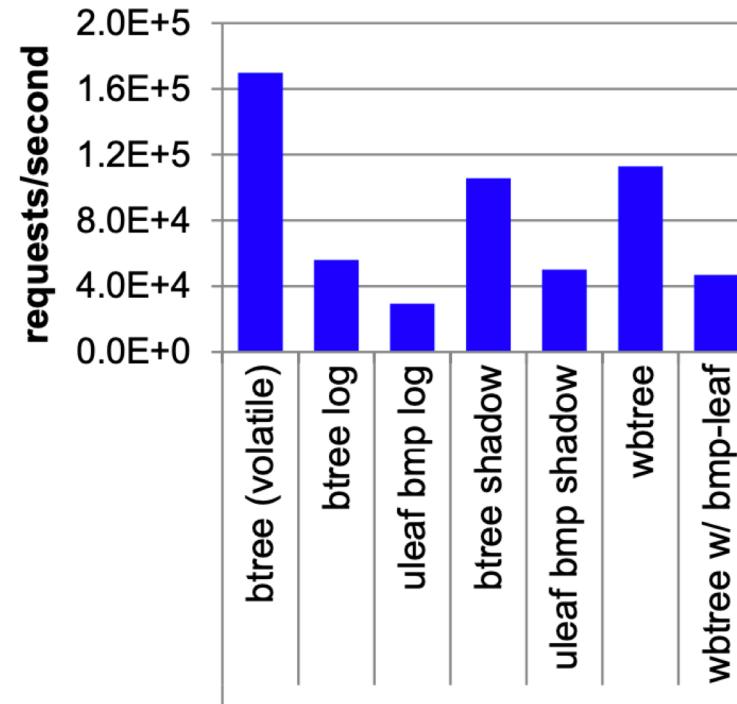
# Experimental Results



(a) Search, 70% full nodes



(b) Insertion, 70% full nodes

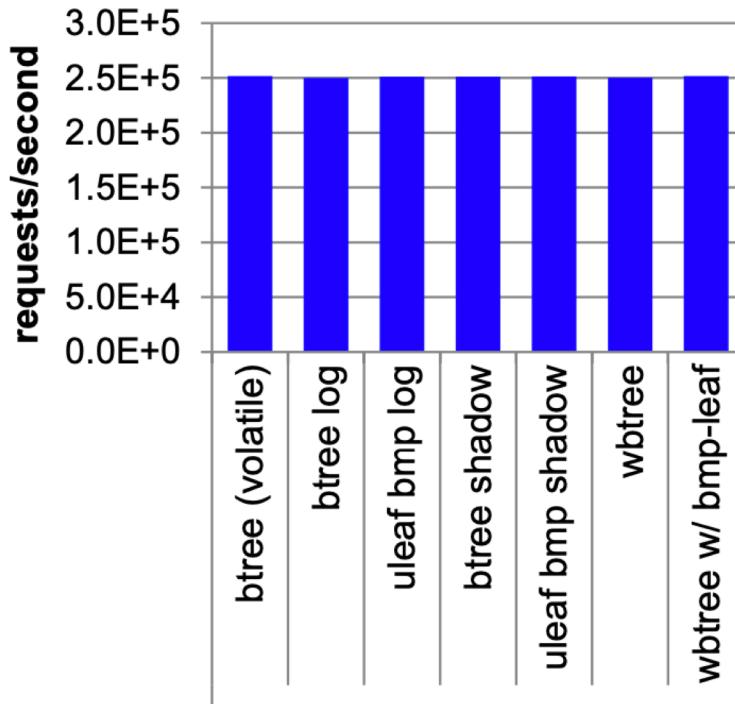


(c) Insertion, 100% full nodes

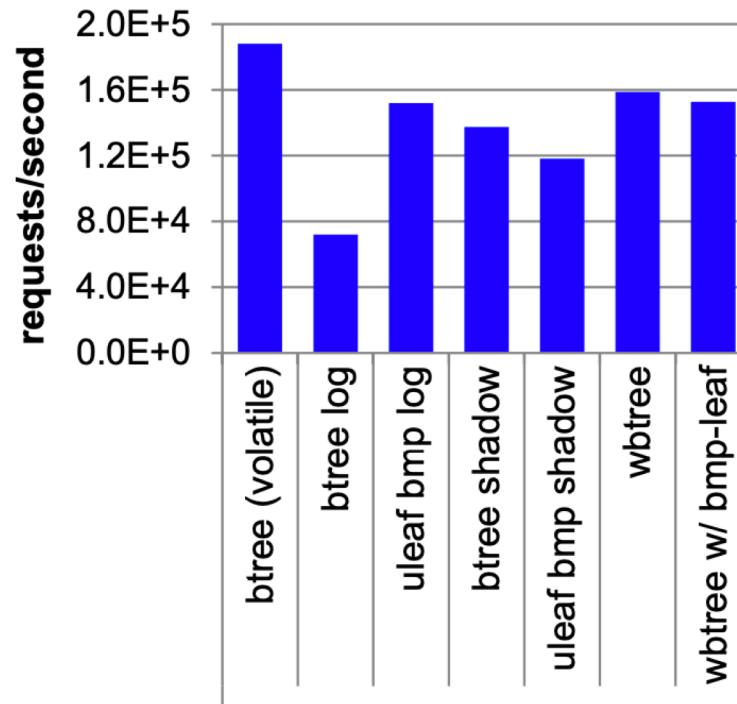
**Figure 15: Memcached throughput on a real machine. (We replace the hash index in Memcached with various types of trees. We bulkload a tree with 50M entries, and use mc-benchmark to insert and search 500K random keys. Keys are 20-byte random strings.)**

- Performance difference across solutions is smaller because of the communication overhead

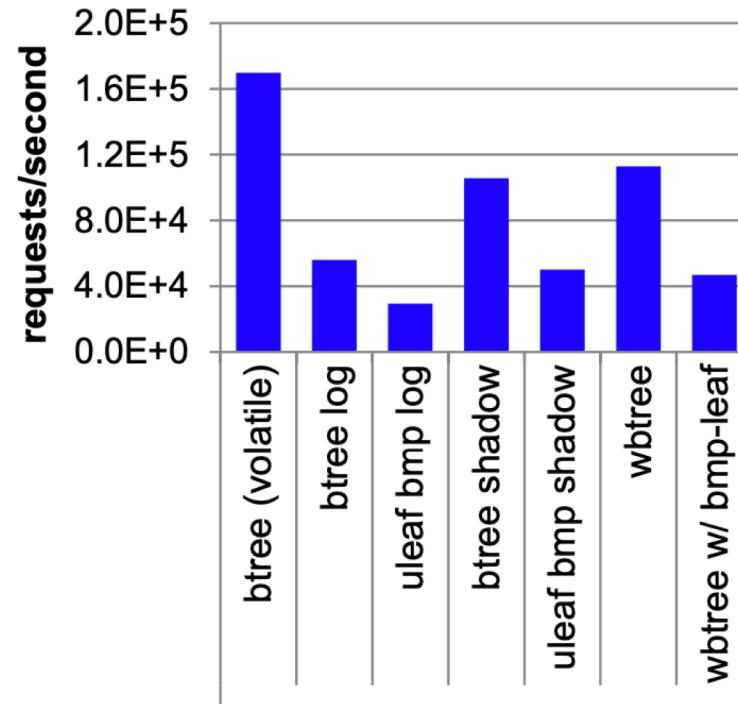
# Experimental Results



(a) Search, 70% full nodes



(b) Insertion, 70% full nodes

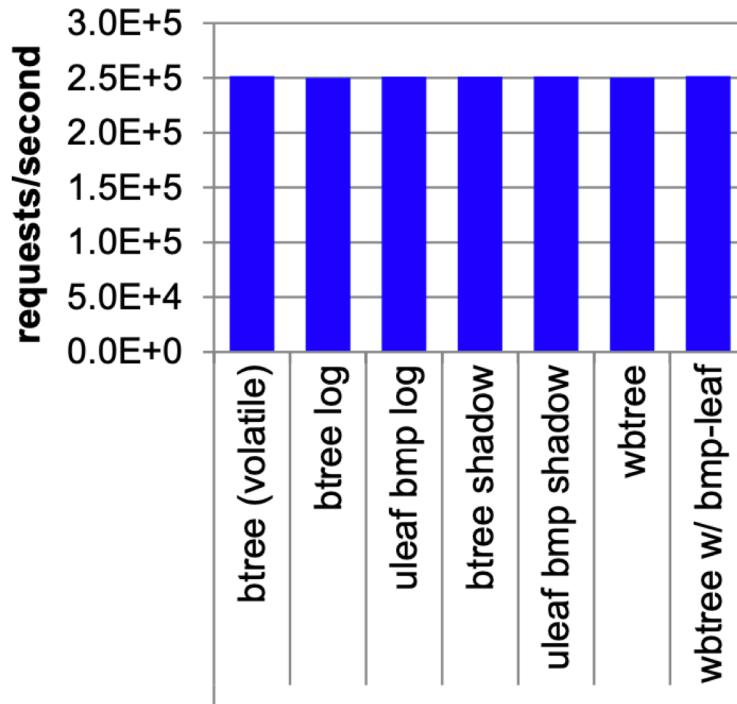


(c) Insertion, 100% full nodes

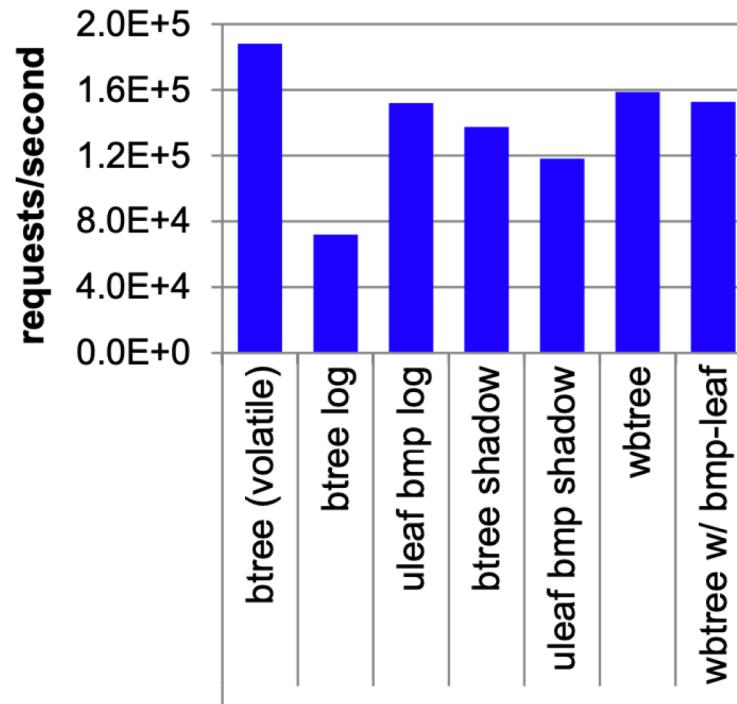
**Figure 15: Memcached throughput on a real machine. (We replace the hash index in Memcached with various types of trees. We bulkload a tree with 50M entries, and use mc-benchmark to insert and search 500K random keys. Keys are 20-byte random strings.)**

- The shorter search time is outweighed more by the communication overhead

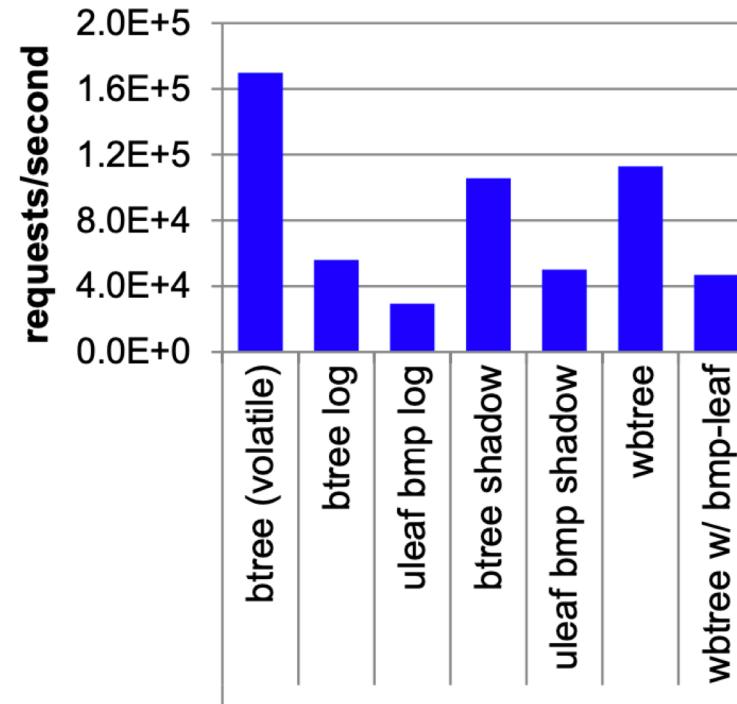
# Experimental Results



(a) Search, 70% full nodes



(b) Insertion, 70% full nodes



(c) Insertion, 100% full nodes

**Figure 15: Memcached throughput on a real machine. (We replace the hash index in Memcached with various types of trees. We bulkload a tree with 50M entries, and use mc-benchmark to insert and search 500K random keys. Keys are 20-byte random strings.)**

- wbtree achieves the highest throughput for insertions among persistent tree structures

# Conclusion

# Conclusion

---

- Persistence is crucial for NVMM data structures
- Undo-redo logging and shadowing perform extensive NVM writes and cacheline flushes
- Leaving leaves unsorted reduces writes, but makes search less effective
- The factors affecting performance have different weights for different NVM technologies
- Proposed wB<sup>+</sup>-Trees improve the insertion and deletion performance, while achieving good search performance

# Questions?

---



# Insertion with Node Splitting

---

- Allocate new node and balance entries between old and new node
- No need to move entries in old node (because unsorted)
- Write bitmap/slot fields and sibling pointer of the new node
- Update bitmap/slot field and sibling pointer of old node. (redo-logging)
- Insert new lead node to parent node using insertion algorithm and commit redo writes.

# Workload

---

- 8-byte integer keys for fixed-sized keys and 20-byte strings for variable sized keys
- $B$  is large enough so that the tree size is much larger than LLC
- For simulation,  $B = 20$  million, for real machine experiments,  $B = 50$  million
- Total size of valid leaf entries is 320MB in simulation and 800MB on the real machine
- For variable sized keys, they perform only real-machine experiments,  $B = 50$  million
- There will be an additional 1GB memory space for storing the actual strings on the real machine