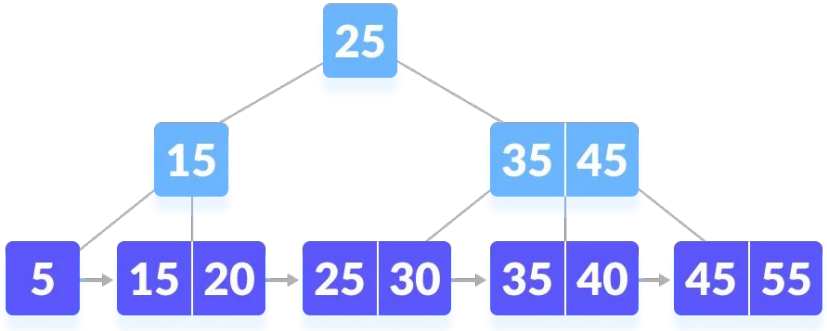
# Persistent B+-Tree in Non-Volatile Main Memory

Richard Andreas, Hantian Liu,
Jingyu Su, Xingkun Yin

# Problem proposed

Non Volatile Memory

Data Structure

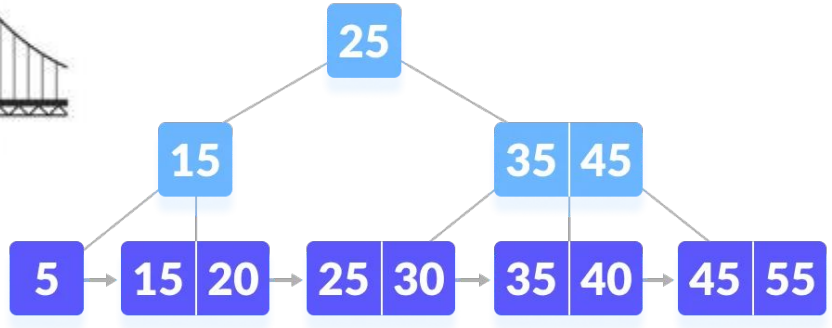


# Problem proposed

Non Volatile Memory

Bridge

Data Structure



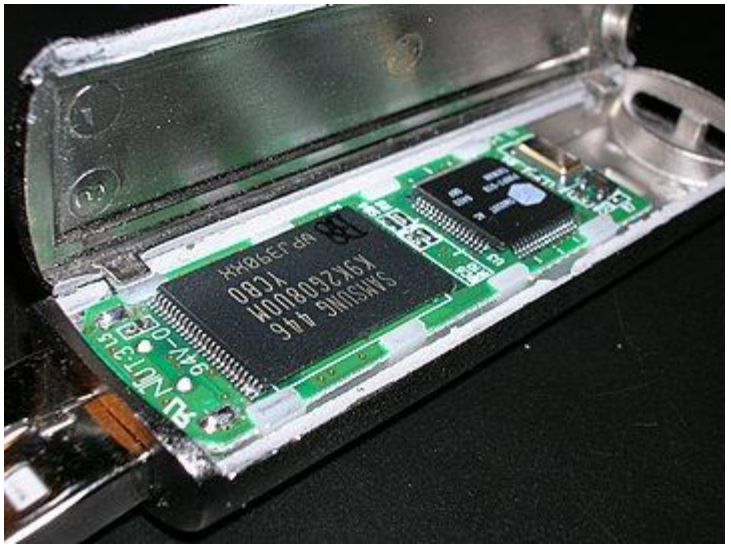
# Background

Non Volatile Memory

- do not need power to retain data
- enable smaller feature size

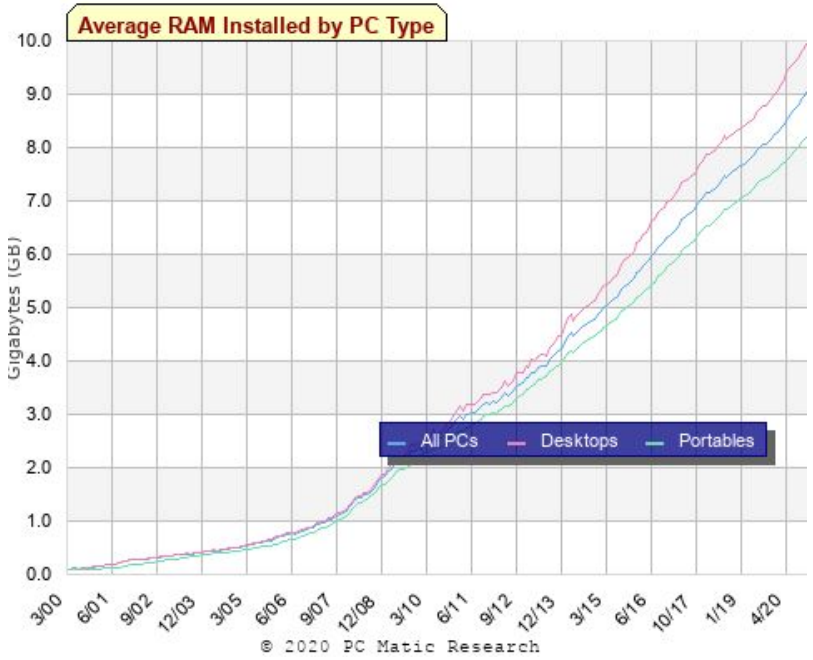
Type

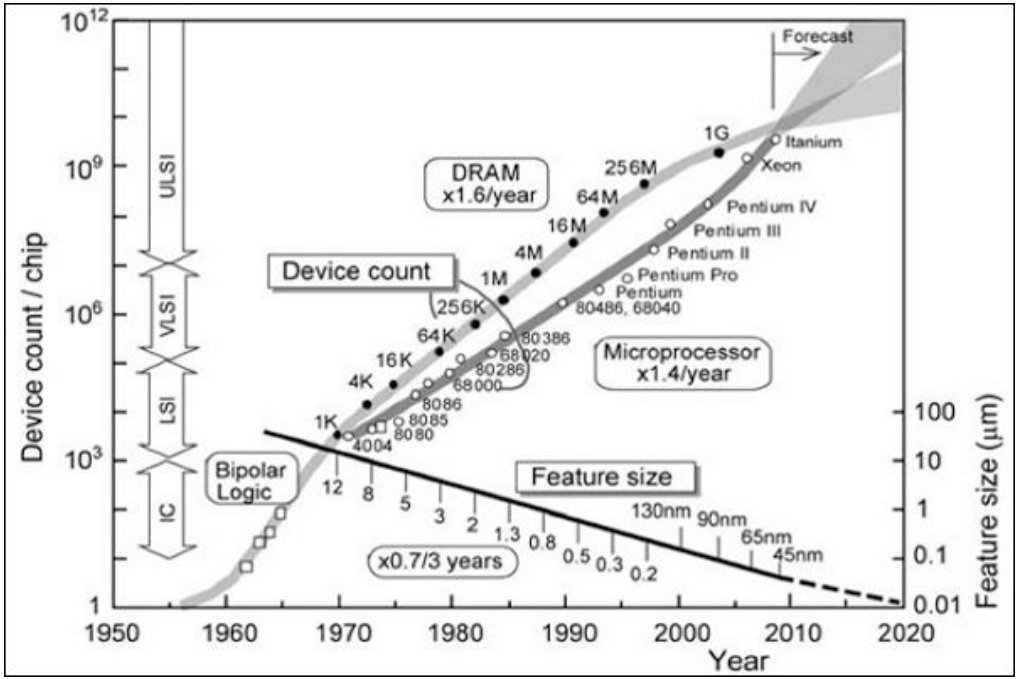
- Phase Change Memory (PCM)
- Ferroelectric RAM (F-RAM)
- Magnetoresistive RAM (MRAM)



# Background

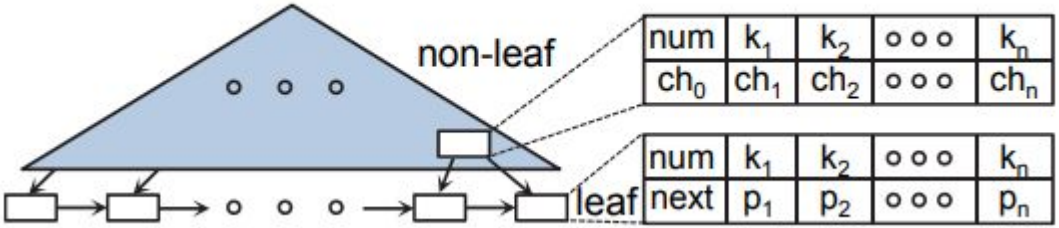
Growth in main memory



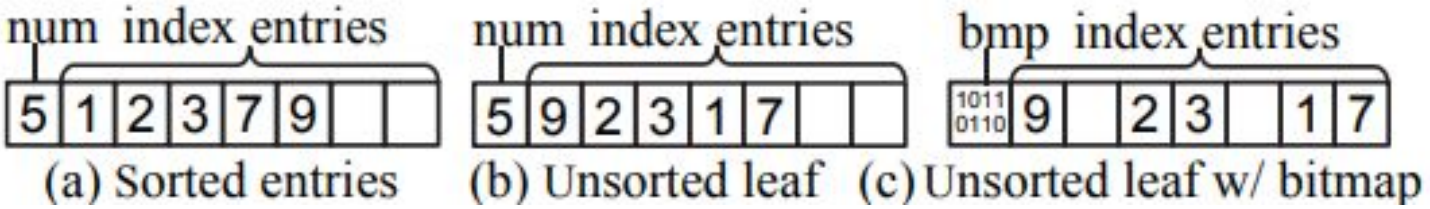



# Background

B+ Tree



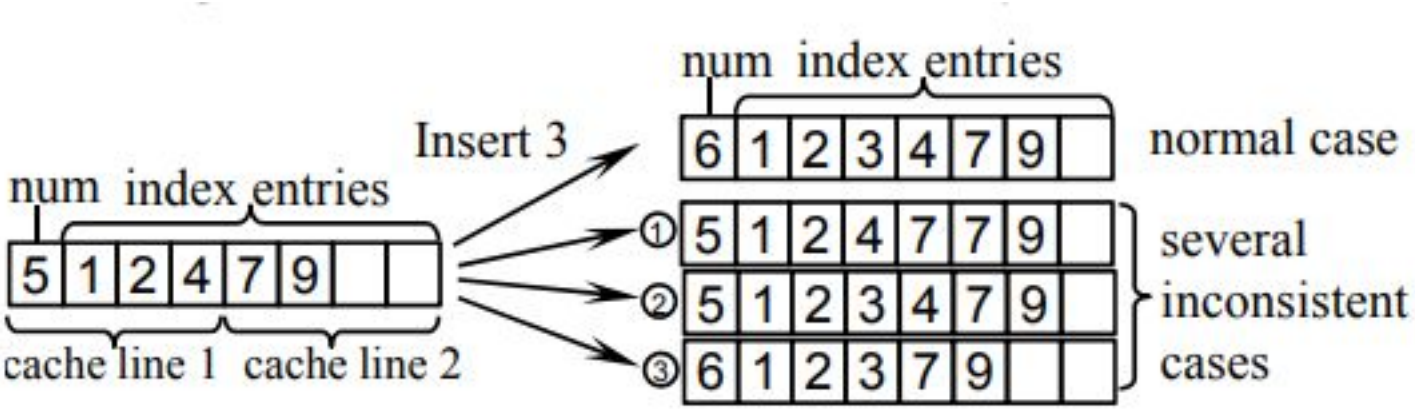
# Background




(a) Sorted entries (b) Unsorted leaf (c) Unsorted leaf w/ bitmap

# Challenge

Inconsistent



# Tools

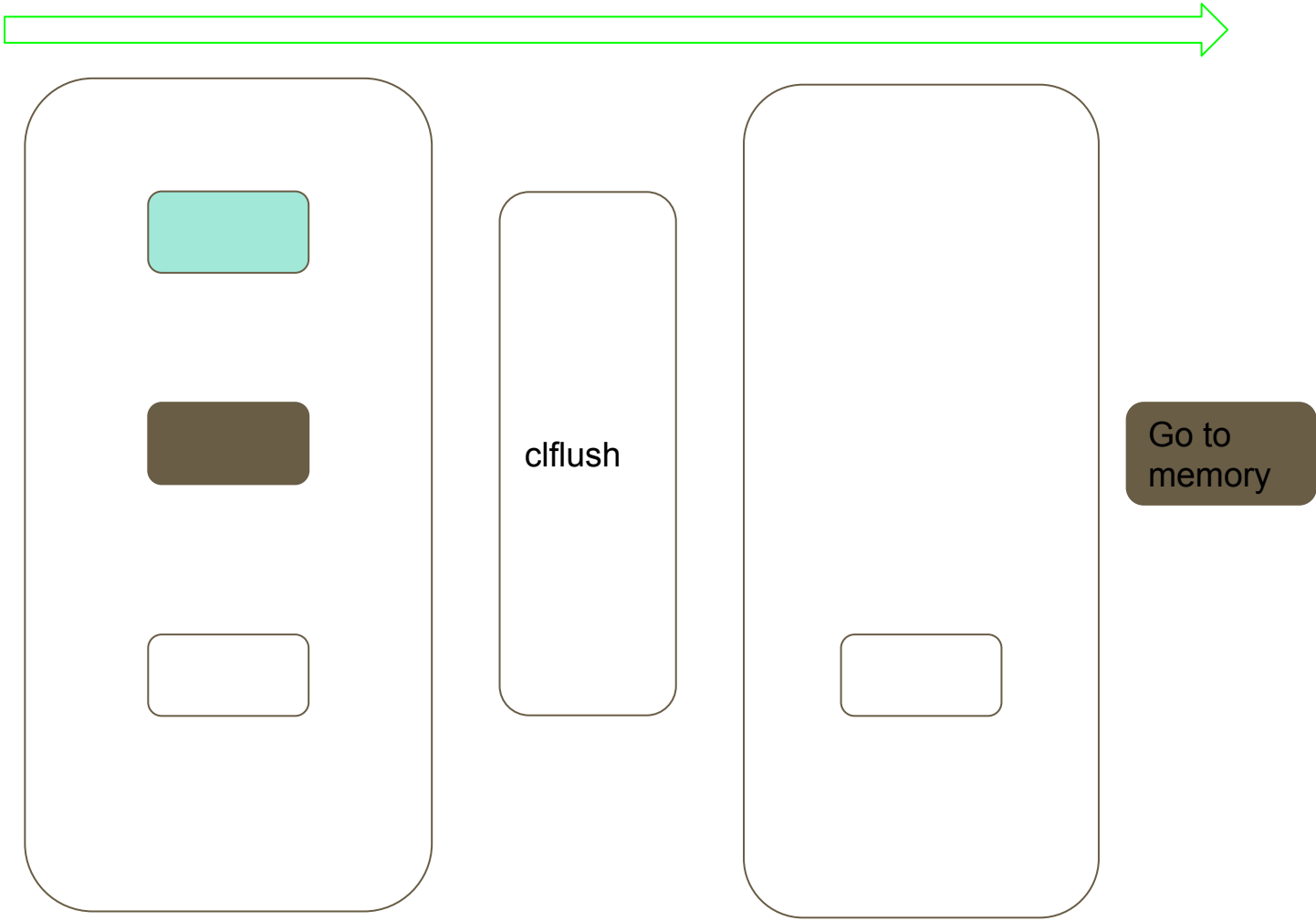
x86 instructions

- clflush
- mfence

# Tools

x86 instructions

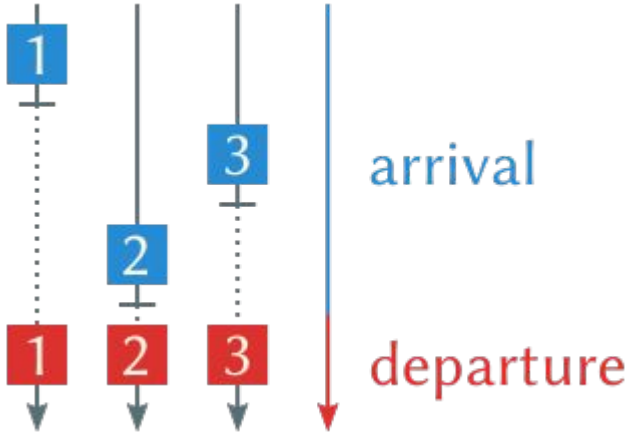
- **clflush**
- mfence



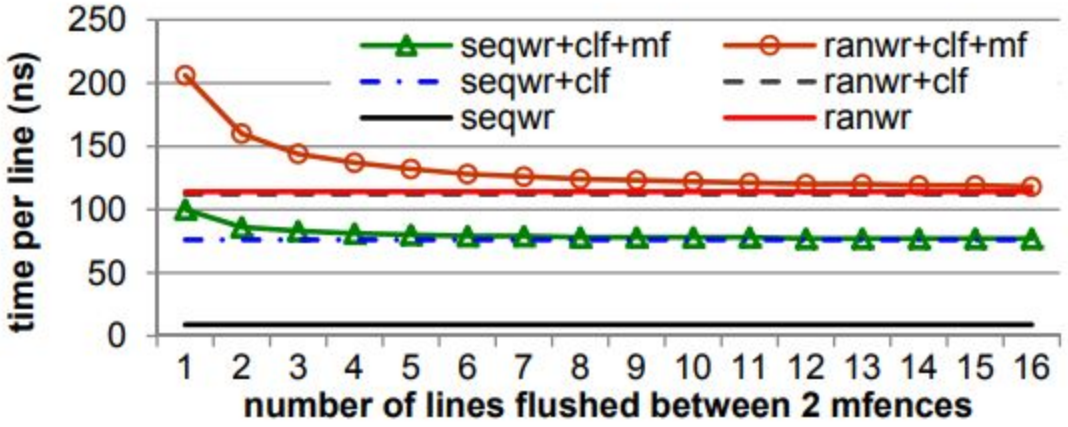
# Tools

x86 instructions

- clflush
- **mfence**



# Challenge

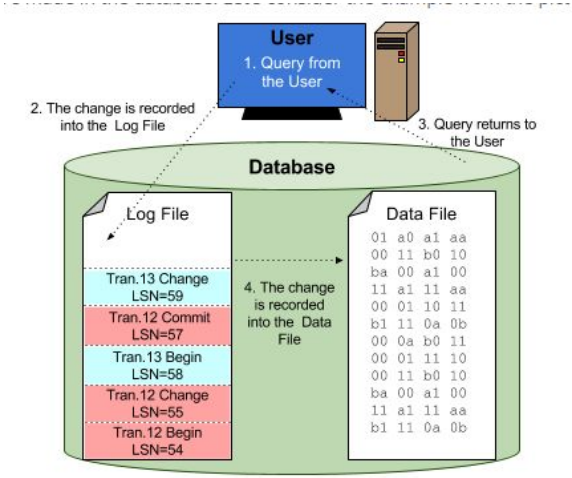

250
200
150
100
50
0
time per line (ns)
seqwr+clf+mf
seqwr+clf
seqwr
ranwr+clf+mf
ranwr+clf
ranwr
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
number of lines flushed between 2 mfences

# Existing Solutions

- Follows 2 traditional Principles:
  - Logging
  - Shadowing

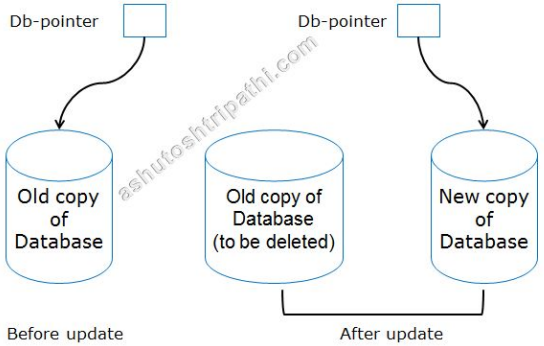
# Logging

- Records REDO and UNDO information, for every update, in a log
- Log: an ordered list of REDO/UNDO actions
  - Contains information of each update
- Ensures the database fulfills the **atomicity** principle



# Shadowing

- The process of creating multiple copies of data.
- Ensures that the original data is not being updated till the very end to ensure database is **durable**




Shadow-copy technique for atomicity and durability

# Undo-Redo Logging

Divide memory into 2 parts:

- **Persistent**: Holds persistent tree nodes
- **Volatile**: stores the buffer pool

To protect in-place NVM writes requires undo-redo logging

- Requires **clflush** and **mfence** to ensure log content is stable before performing actual write

```
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
```

**Figure 5: NVMM write protected by undo-redo logging.**

# Undo-Redo Logging

Divide memory into 2 parts:

- **Persistent**: Holds persistent tree nodes
- **Volatile**: stores the buffer pool

To protect in-place NVM writes requires undo-redo logging

- Requires **clflush** and **mfence** to ensure log content is stable before performing actual write

```
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
```

**Figure 5: NVMM write protected by undo-redo logging.**

# Undo-Redo Logging

Divide memory into 2 parts:

- **Persistent**: Holds persistent tree nodes
- **Volatile**: stores the buffer pool

To protect in-place NVM writes requires undo-redo logging

- Requires **clflush** and **mfence** to ensure log content is stable before performing actual write

```
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
```

**Figure 5: NVMM write protected by undo-redo logging.**

# Logging Continued

- Only applicable if a **newly written** value is **not** to be **accessed** again **before commit**

- Re-reading the **newly written** value **before commit** will cause an **error**.

```
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.**

# Logging Continued

- Only applicable if a **newly written** value is **not** to be **accessed** again **before commit**

- Re-reading the **newly written** value **before commit** will cause an **error**.

```
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.**

# Shadowing

- **Short-Circuit Shadowing**
  - Taking advantage of the 8 byte atomic write feature in NVM.
  - Automatically modify the leaf node pointer in the updated node's parent.
  - A single 8 byte pointer points to newly created node copy

```
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.**

# Shadowing

- **Short-Circuit Shadowing**
  - Taking advantage of the 8 byte atomic write feature in NVM.
  - Automatically modify the leaf node pointer in the updated node's parent.
  - A single 8 byte pointer points to newly created node copy

```
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.**

# Write Atomic B+-Trees

**Design Goals:**

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

# wB+ - Tree Structures

- Introduced a small indirection array to a bitmap-only unsorted node.
- Slot+Bitmap nodes contain both a bitmap and indirection slot array.
- Combine slot-only nodes, slot+bitmap nodes, bitmap-only leaf nodes to form 3 wB+ - Tree structures

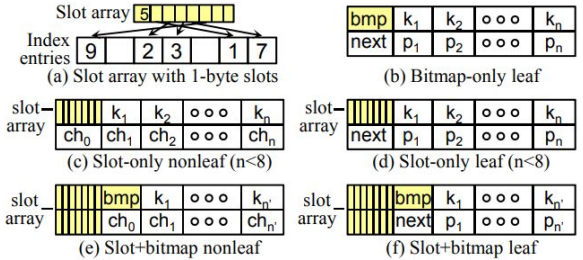
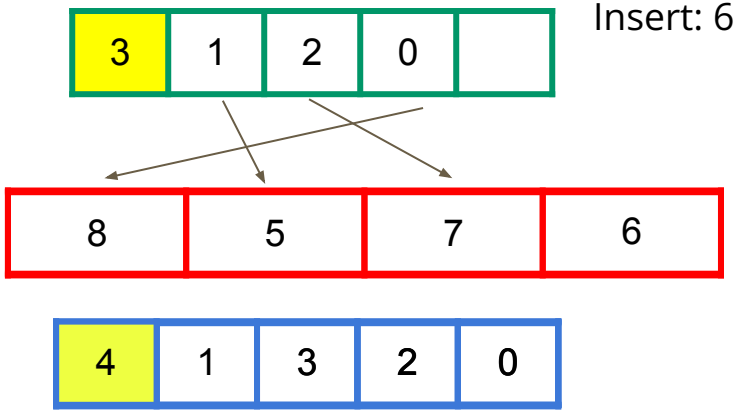



**Figure 8: wB$^+$-Tree node structures. (For a slot+bitmap node, the lowest bit in the bitmap indicates whether the slot array is valid. Slot 0 records the number of valid entries in a node.)**

**Table 2: wB$^+$-Tree structures considered in this paper.**

| Structure | Leaf Node | Non-leaf Node |
|---|---|---|
| wB$^+$-Tree | slot+bitmap leaf | slot+bitmap non-leaf |
| wB$^+$-Tree w/ bitmap-only leaf | bitmap-only leaf | slot+bitmap non-leaf |
| wB$^+$-Tree w/ slot-only nodes | slot-only leaf | slot-only non-leaf |

# Insertion (Slot only)

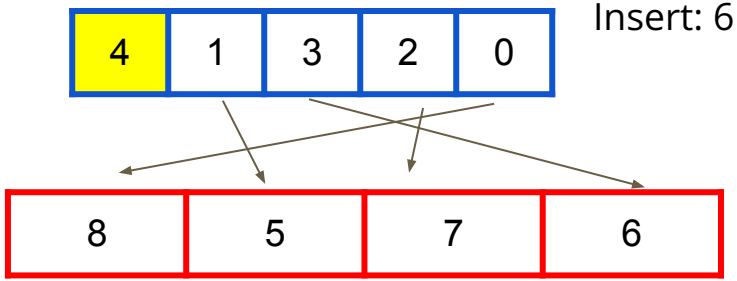



Insert position in slot array: 1

Unused entry offset: 3

```
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≥pos; j - -) do
10:         tempslot[j+1]= leaf.slot[j];
11:     end for
12:     tempslot[pos]=u;
13:     for (j=pos-1; j≥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
```

# Insertion (Slot only)



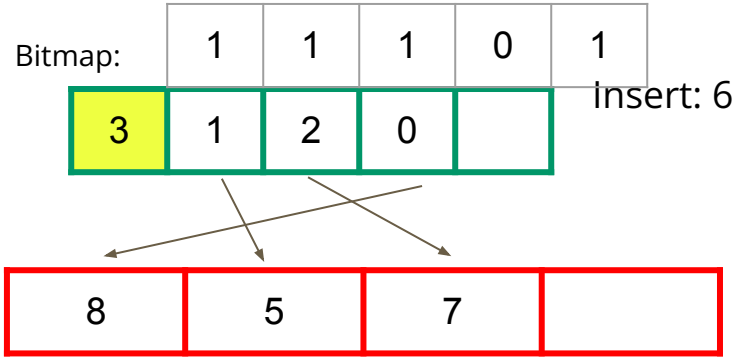

Insert: 6

Insert position in slot array: 1

Unused entry offset: 3

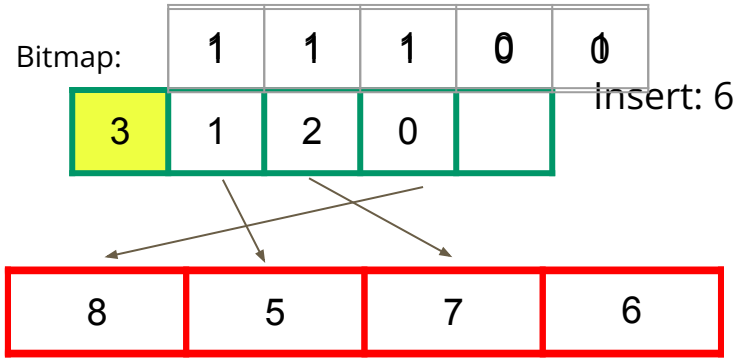
```
1:  procedure INSERT2SLOTONLY_ATOMIC(leaf, newEntry)
2:      /* Slot array is valid */
3:      pos= leaf.GetInsertPosWithBinarySearch(newEntry);
4:      /* Write and flush newEntry */
5:      u= leaf.GetUnusedEntryWithSlotArray();
6:      leaf.entry[u]= newEntry;
7:      clflush(&leaf.entry[u]); mfence();
8:      /* Generate an up-to-date slot array on the stack */
9:      for (j=leaf.slot[0]; j≥pos; j - -) do
10:         tempslot[j+1]= leaf.slot[j];
11:     end for
12:     tempslot[pos]=u;
13:     for (j=pos-1; j≥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
```

# Insertion (Slot + bitmap)




```
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,
            building the slot array, and setting the slot valid bit;
4:      end if
5:      pos= leaf.GetInsertPosWithBinarySearch(newEntry);
6:      /* Disable the slot array */
7:      leaf.bitmap = leaf.bitmap - 1;
8:      clflush(&leaf.bitmap); mfence();
9:      /* Write and flush newEntry */
10:     u= leaf.GetUnusedEntryWithBitmap();
11:     leaf.entry[u]= newEntry;
12:     clflush(&leaf.entry[u]);
13:     /* Modify and flush the slot array */
14:     for (j=leaf.slot[0]; j≥pos; j - -) do
15:         leaf.slot[j+1]= leaf.slot[j];
16:     end for
17:     leaf.slot[pos]=u;
18:     for (j=pos-1; j≥1; j - -) do
19:         leaf.slot[j]= leaf.slot[j];
20:     end for
21:     leaf.slot[0]=leaf.slot[0]+1;
22:     for (j=0; j≤leaf.slot[0]; j += 8) do
23:         clflush(&leaf.slot[j]);
24:     end for
25:     mfence(); /* Ensure new entry and slot array are stable */
26:     /* Enable slot array, new entry and flush bitmap */
27:     leaf.bitmap = leaf.bitmap + 1 + (1<<u);
28:     clflush(&leaf.bitmap); mfence();
29: end procedure
```

# Insertion (Slot + bitmap)

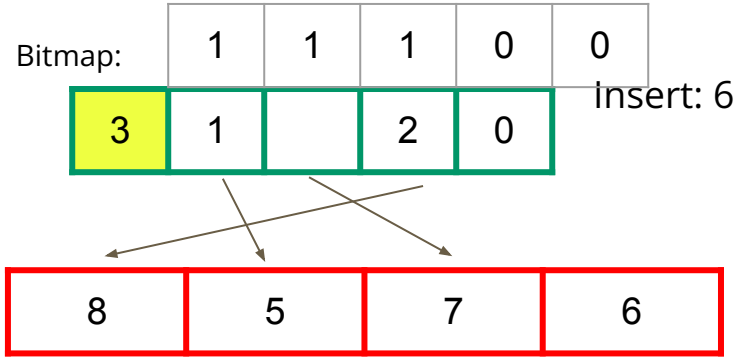



Insert position in slot array: 1

Unused entry offset: 3

```
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,
            building the slot array, and setting the slot valid bit;
4:      end if
5:      pos= leaf.GetInsertPosWithBinarySearch(newEntry);
6:      /* Disable the slot array */
7:      leaf.bitmap = leaf.bitmap - 1;
8:      clflush(&leaf.bitmap); mfence();
9:      /* Write and flush newEntry */
10:     u= leaf.GetUnusedEntryWithBitmap();
11:     leaf.entry[u]= newEntry;
12:     clflush(&leaf.entry[u]);
13:     /* Modify and flush the slot array */
14:     for (j=leaf.slot[0]; j≥pos; j - -) do
15:         leaf.slot[j+1]= leaf.slot[j];
16:     end for
17:     leaf.slot[pos]=u;
18:     for (j=pos-1; j≥1; j - -) do
19:         leaf.slot[j]= leaf.slot[j];
20:     end for
21:     leaf.slot[0]=leaf.slot[0]+1;
22:     for (j=0; j≤leaf.slot[0]; j += 8) do
23:         clflush(&leaf.slot[j]);
24:     end for
25:     mfence(); /* Ensure new entry and slot array are stable */
26:     /* Enable slot array, new entry and flush bitmap */
27:     leaf.bitmap = leaf.bitmap + 1 + (1<<u);
28:     clflush(&leaf.bitmap); mfence();
29: end procedure
```

# Insertion (Slot + bitmap)

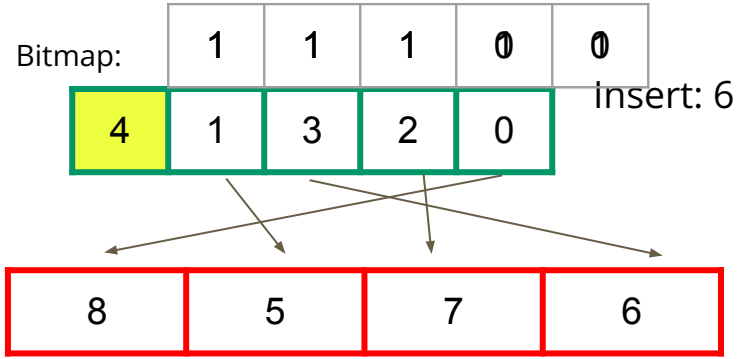



Insert position in slot array: 1

Unused entry offset: 3

```
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,
            building the slot array, and setting the slot valid bit;
4:      end if
5:      pos= leaf.GetInsertPosWithBinarySearch(newEntry);
6:      /* Disable the slot array */
7:      leaf.bitmap = leaf.bitmap - 1;
8:      clflush(&leaf.bitmap); mfence();
9:      /* Write and flush newEntry */
10:     u= leaf.GetUnusedEntryWithBitmap();
11:     leaf.entry[u]= newEntry;
12:     clflush(&leaf.entry[u]);
13:     /* Modify and flush the slot array */
14:     for (j=leaf.slot[0]; j≥pos; j - -) do
15:         leaf.slot[j+1]= leaf.slot[j];
16:     end for
17:     leaf.slot[pos]=u;
18:     for (j=pos-1; j≥1; j - -) do
19:         leaf.slot[j]= leaf.slot[j];
20:     end for
21:     leaf.slot[0]=leaf.slot[0]+1;
22:     for (j=0; j≤leaf.slot[0]; j += 8) do
23:         clflush(&leaf.slot[j]);
24:     end for
25:     mfence(); /* Ensure new entry and slot array are stable */
26:     /* Enable slot array, new entry and flush bitmap */
27:     leaf.bitmap = leaf.bitmap + 1 + (1<<u);
28:     clflush(&leaf.bitmap); mfence();
29: end procedure
```

# Insertion (Slot + bitmap)

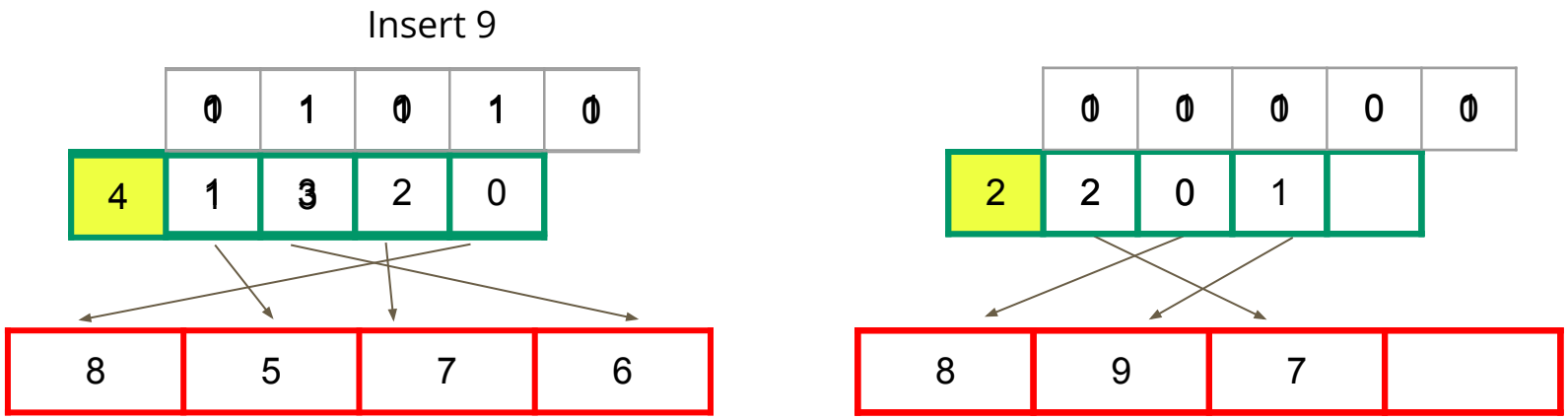



Insert position in slot array: 1

Unused entry offset: 3

```
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,
            building the slot array, and setting the slot valid bit;
4:      end if
5:      pos= leaf.GetInsertPosWithBinarySearch(newEntry);
6:      /* Disable the slot array */
7:      leaf.bitmap = leaf.bitmap - 1;
8:      clflush(&leaf.bitmap); mfence();
9:      /* Write and flush newEntry */
10:     u= leaf.GetUnusedEntryWithBitmap();
11:     leaf.entry[u]= newEntry;
12:     clflush(&leaf.entry[u]);
13:     /* Modify and flush the slot array */
14:     for (j=leaf.slot[0]; j≥pos; j - -) do
15:         leaf.slot[j+1]= leaf.slot[j];
16:     end for
17:     leaf.slot[pos]=u;
18:     for (j=pos-1; j≥1; j - -) do
19:         leaf.slot[j]= leaf.slot[j];
20:     end for
21:     leaf.slot[0]=leaf.slot[0]+1;
22:     for (j=0; j≤leaf.slot[0]; j += 8) do
23:         clflush(&leaf.slot[j]);
24:     end for
25:     mfence(); /* Ensure new entry and slot array are stable */
26:     /* Enable slot array, new entry and flush bitmap */
27:     leaf.bitmap = leaf.bitmap + 1 + (1<<u);
28:     clflush(&leaf.bitmap); mfence();
29: end procedure
```

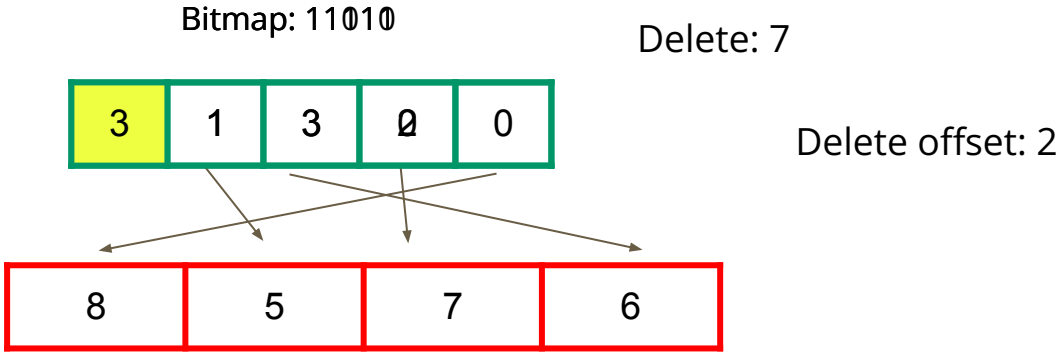
# Insertion - Node Split



# Deletion

**Deletion:**

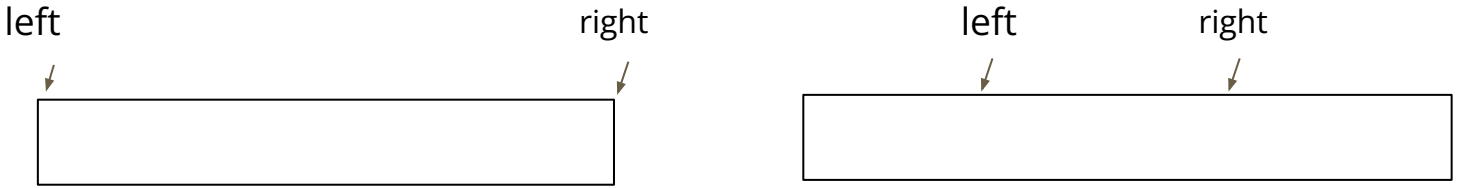
- Does not move data around
- Modify slot array and/or bitmap to reflect deletion



# Search

**Search:**

- Apply binary search on slot array
- Stop binary search when range < 8 slots for lower overhead



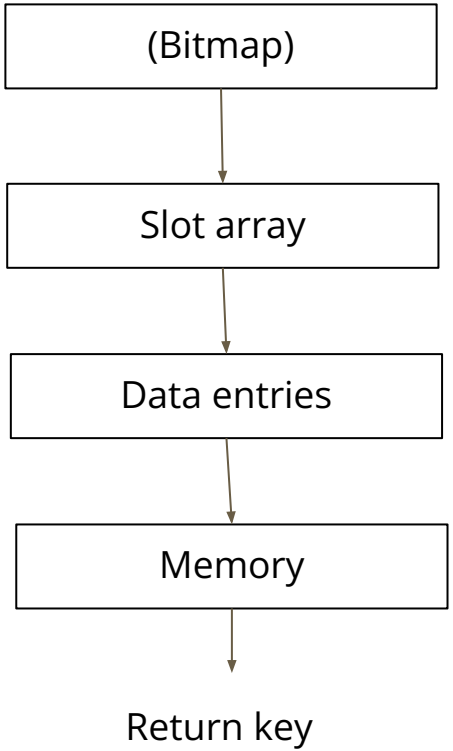
# Variable Sized Key

Solution:

- Store 8-byte pointers to key

Disadvantage:

- Key pointer dereference overhead
- Larger keys takes longer to retrieve & process



# Comparison With Previous Solutions

| 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$^+$-Tree node |
| $n'$ | Total number of entries in a wB$^+$-Tree node |
| $m$ | Number of valid entries in a tree node |
| $l$ | Number of levels of nodes that are split in an insertion |

| | |
|---|---|
| B$^+$-Trees undo-redo logging | $N_w = 4m + 12$, $N_{clf} = N_{mf} = m + 3$ |
| B$^+$-Trees shadowing | $N_w = 2m + 11$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2.5$ |
| wB$^+$-Tree | $N_w = 0.125m + 4.25$, $N_{clf} = \frac{1}{64}m + 3\frac{1}{32}$, $N_{mf} = 3$ |

# Evaluation

## Experiment Setup

1. B+ trees implementations:

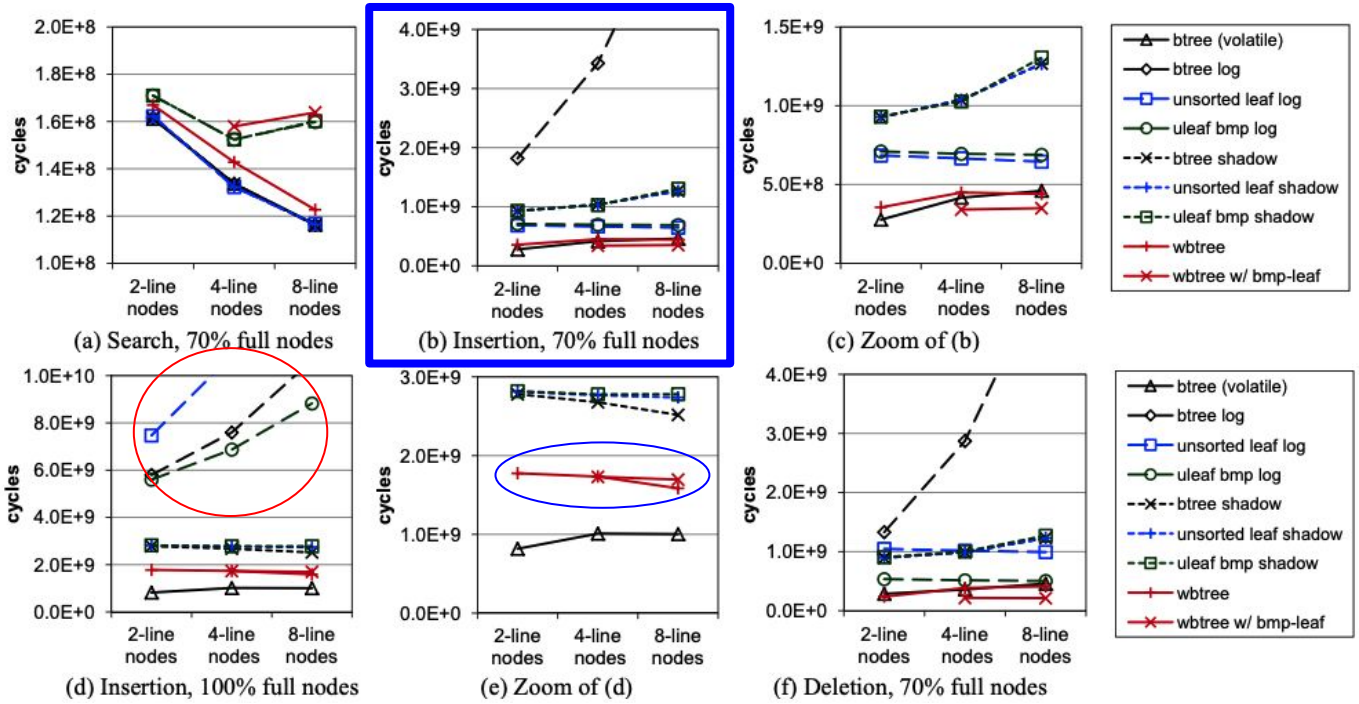
   9 different trees
2. Memcached Implementation
3. B+ tree workload

**Table 4: Experimental Setup.**

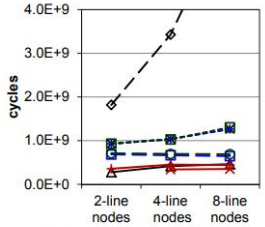
| Real Machine Description | |
|---|---|
| Processor | 2 Intel Xeon E5-2620, 6 cores/12 threads, 2.00GHz |
| CPU cache | 32KB L1I/core, 32KB L1D/core, 256KB L2/core<br>15MB shared L3, all caches with 64B lines |
| OS<br>Compiler | Ubuntu 12.04, Linux 3.5.0-37-generic kernel<br>gcc 4.6.3, compiled with -O3 |
| **Simulator Description** | |
| Processor | Out-of-order X86-64 core, 3GHz |
| CPU cache | Private L1D (32KB, 8-way, 4-cycle latency),<br>private L2 (256KB, 8-way, 11-cycle latency),<br>shared L3 (8MB, 16-way, 39-cycle latency),<br>all caches with 64B lines,<br>64-entry DTLB, 32-entry write back queue |
| PCM | 4 ranks, read latency for a cache line: 230 cycles,<br>write latency per 8B modified word: 450 cycles,<br>$E_{rb}$ = 2 pJ, $E_{wb}$ = 16 pJ |

Computer for experiment

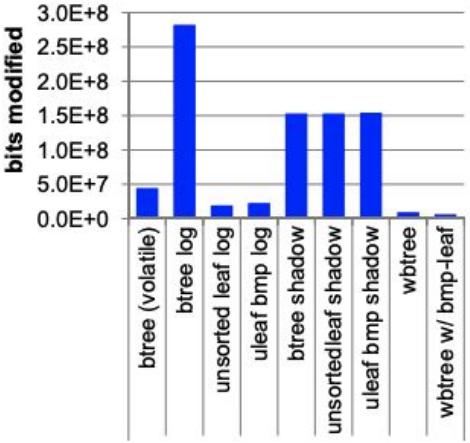
# Simulation Modeling PCM

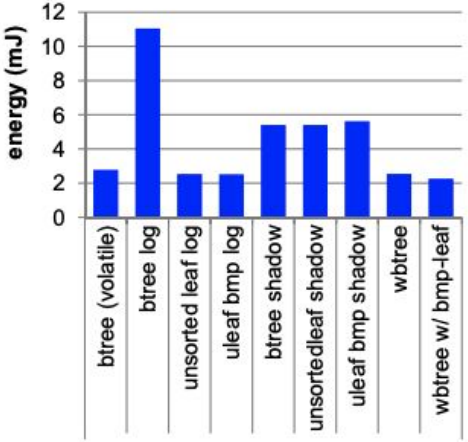


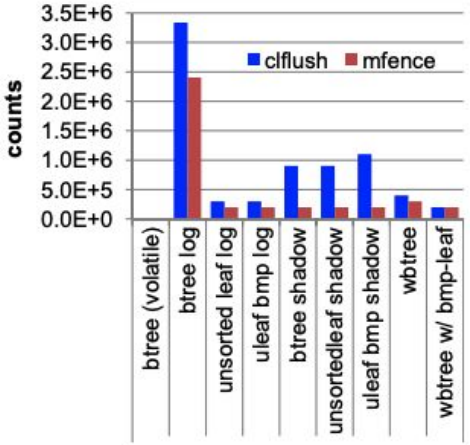
# Simulation Modeling PCM



(b) Insertion, 70% full nodes



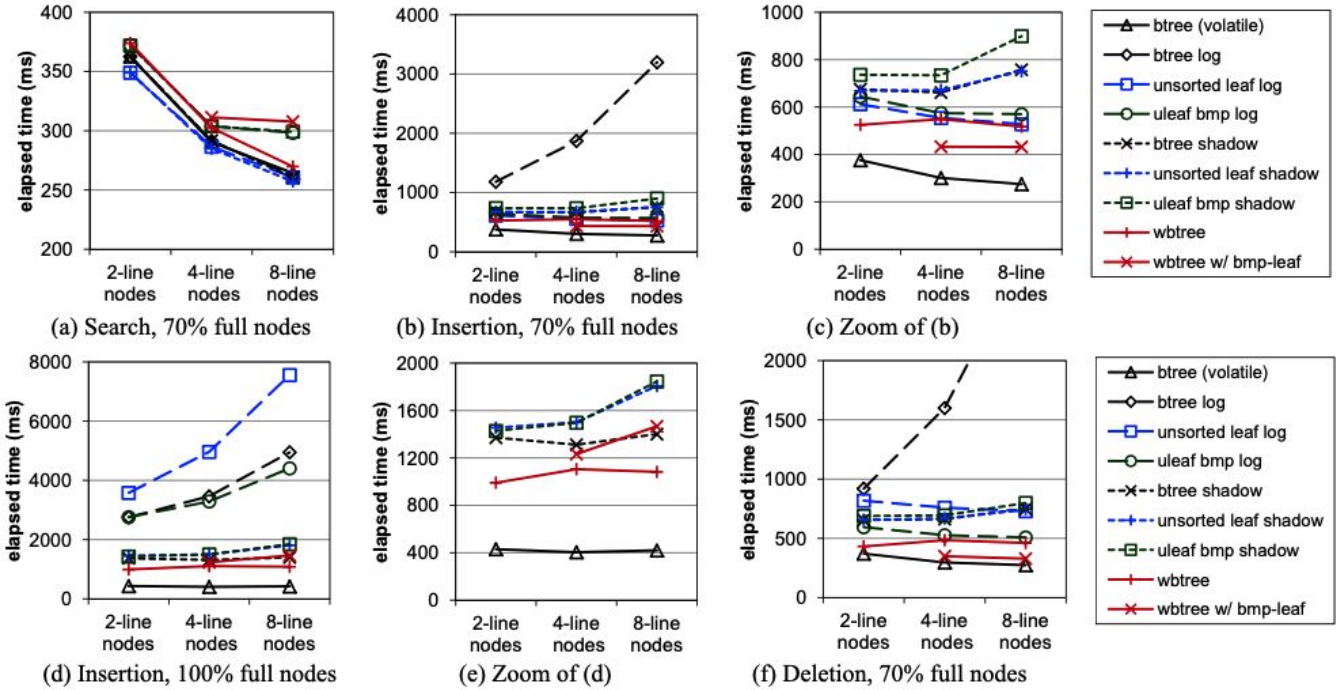




| | |
|---|---|
| B$^+$-Trees undo-redo logging | $N_w = 4m + 12$, $N_{clf} = N_{mf} = m + 3$ |

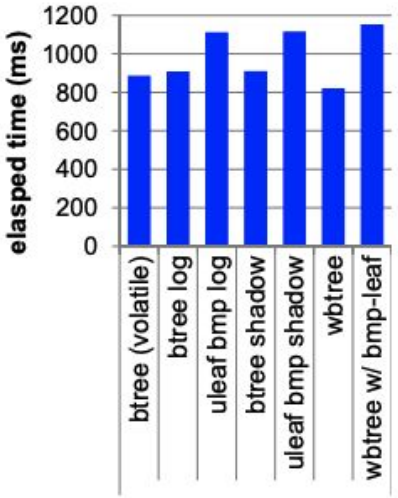
| | |
|---|---|
| B$^+$-Trees shadowing | $N_w = 2m + 11$, $N_{mf} = 2$, $N_{clf} = 0.25m + 2.5$ |

| | |
|---|---|
| wB$^+$-Tree | $N_w = 0.125m + 4.25$, $N_{clf} = \frac{1}{64}m + 3\frac{1}{32}$, $N_{mf} = 3$ |

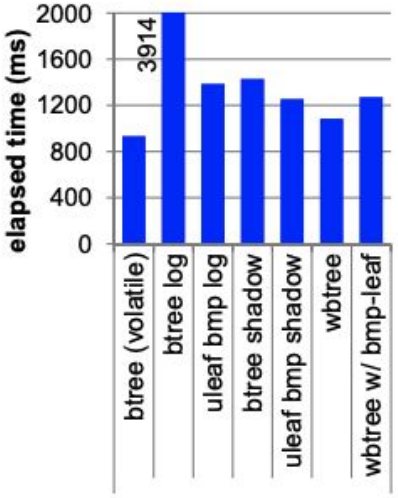
# Real Machine Experiments Modeling Fast DRAM-Like NVM



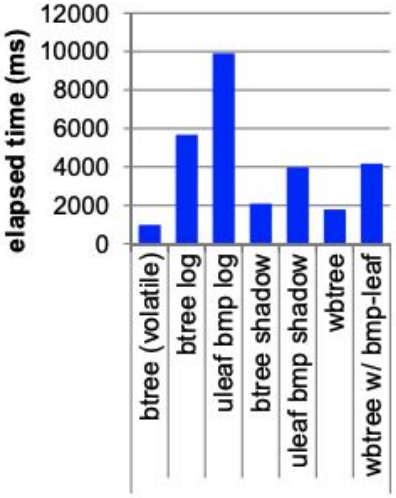
# Real Machine Experiments for Trees with String Keys



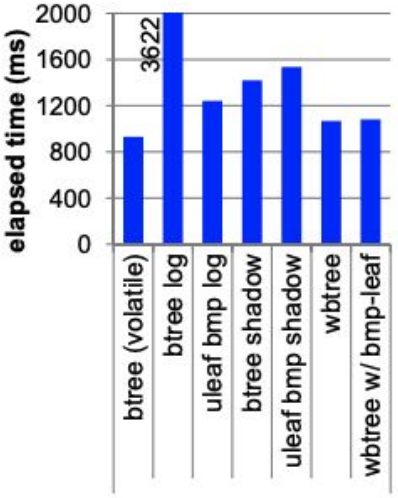

(a) Search, 70% full nodes, 20-byte string keys




(b) Insertion, 70% full nodes, 20-byte string keys

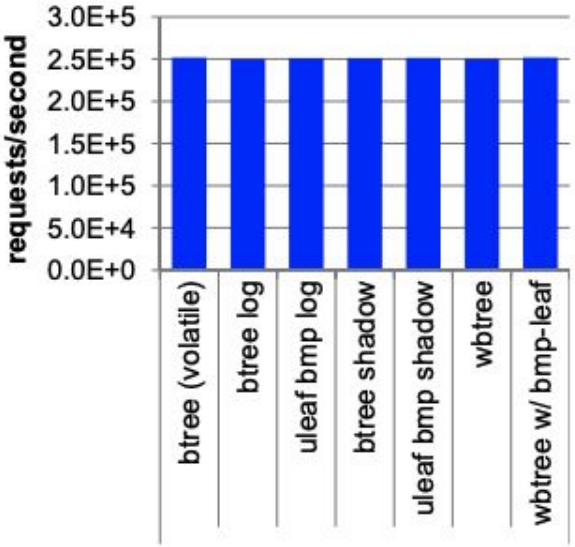



(c) Insertion, 100% full nodes, 20-byte string keys

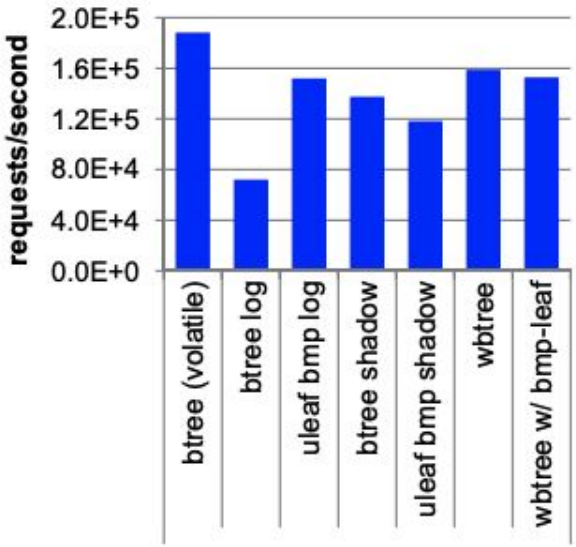



(d) Deletion, 70% full nodes, 20-byte string keys

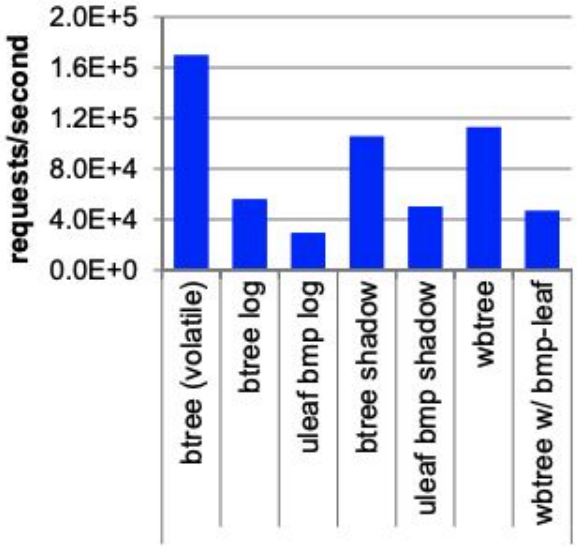
# Real-Machine Memcached Performance




(a) Search, 70% full nodes




(b) Insertion, 70% full nodes




(c) Insertion, 100% full nodes

# Conclusion

**Logging** and **shadowing** incurs high overhead due to **NVM writes** & **CL flushes**.

The factors affecting performance have **different weights** for **different NVM technologies**.

wB+-Trees **significantly improve** the **insertion** and **deletion** performance while achieving good **search** performance.

# Thank you!