Persistent $B^+$-Trees in Non-Volatile Main Memory

Evgenii Zheltonozhskii
January 12, 2020

Technion
Table of contents

1. Background
2. Existing Solutions
3. Write-Atomic B⁺-Trees
4. Experimental Results
5. Conclusions
Background
Persistent B\(^+\)-Trees in **Non-Volatile Main Memory**

- Non-volatile memory (NVM) is a type of memory which does not require energy to preserve its content.
• Non-volatile memory (NVM) is a type of memory which does not require energy to preserve its content.

• Non-volatile main memory (NVMM) is NVM used as main memory (RAM replacement)
• Non-volatile memory (NVM) is a type of memory which does not require energy to preserve its content.
• Non-volatile main memory (NVMM) is NVM used as main memory (RAM replacement)
• We assume NVM chip can guarantee atomic writes to aligned 8-byte words
Different types of NVM have different properties

- PCM has much slower writes (e.g., 200ns – 1\(\mu\)s) than reads (e.g., 50\(ns\))
Persistent B⁺-Trees in Non-Volatile Main Memory

Different types of NVM have different properties

- 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$^+$-Trees in Non-Volatile Main Memory

- **Persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.
• **Persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.

• **Different levels of persistence**
• **Persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.

• Different levels of persistence
  • Partial – only latest version can be modified.
Persistent B⁺-Trees in Non-Volatile Main Memory

- **Persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.

- Different levels of persistence
  - Partial – only latest version can be modified.
  - Full – any version can be modified.
Persistent B$^+$-Trees in Non-Volatile Main Memory

- **Persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.

- Different levels of persistence
  - Partial – only latest version can be modified.
  - Full – any version can be modified.
  - Confluent – any two versions can be merged.
• **Persistent data structure** is a data structure that always preserves the previous version of itself when it is modified.

• Different levels of persistence
  • Partial – only latest version can be modified.
  • Full – any version can be modified.
  • Confluent – any two versions can be merged.

• We mostly care about ability to restore state after failure
Persistent $\mathbf{B}^+$-Trees in Non-Volatile Main Memory

- $\mathbf{B}^+$-trees is $m$-ary tree where each node except root has between $m/2$ and $m$ children.

Figure 1: An example of $\mathbf{B}^+$-tree. Picture from https://www.qwertee.io/blog/postgresql-b-tree-index-explained-part-1/
Persistent $B^+$-Trees in Non-Volatile Main Memory

- $B^+$-trees is $m$-ary tree where each node except root has between $m/2$ and $m$ children.
- As opposed to B-trees, keys are stored only in leafs.

Figure 1: An example of $B^+$-tree. Picture from https://www.qwertyee.io/blog/postgresql-b-tree-index-explained-part-1/
Persistent B\(^{+}\)-Trees in Non-Volatile Main Memory

- **B\(^{+}\)-trees** is \(m\)-ary tree where each node except root has between \(m/2\) and \(m\) children.
- As opposed to B-trees, keys are stored only in leafs.
- It can be understood as a search structure over sorted list or array.

**Figure 2:** Search in B\(^{+}\)-tree. Picture from https://www.qwertee.io/blog/postgresql-b-tree-index-explained-part-1/
• During insertion we split nodes if needed, during deletion we merge them (or redistribute children).
• During insertion we split nodes if needed, during deletion we merge them (or redistribute children).

• Using binary search we achieve $O(\log_2 m \cdot \log_m n) = O(\log_2 n)$ for all operation (insertion, deletion, search).
• During insertion we split nodes if needed, during deletion we merge them (or redistribute children).
• Using binary search we achieve $O(\log_2 m \cdot \log_m n) = O(\log_2 n)$ for all operation (insertion, deletion, search).
• However, amount of disc accesses is $O(\log_m n)$. 

Persistent $B^+$-Trees in Non-Volatile Main Memory
• We assume both key and value are 8 byte
• We assume both key and value are 8 byte
• Leaf nodes are connected by sibling pointers
• We assume both key and value are 8 byte
• Leaf nodes are connected by sibling pointers
• For disk-based B\(^+\)-Trees, the node size is a few disk pages (e.g., 4KB–256KB).
• We assume both key and value are 8 byte
• Leaf nodes are connected by sibling pointers
• For disk-based B⁺-Trees, the node size is a few disk pages (e.g., 4KB–256KB).
• The node of main-memory B+-Trees is typically a few cache lines large (e.g., 2–8 64-byte cache lines)
• We assume both key and value are 8 byte
• Leaf nodes are connected by sibling pointers
• For disk-based B\textsuperscript+-Trees, the node size is a few disk pages (e.g., 4KB–256KB).
• The node of main-memory B+-Trees is typically a few cache lines large (e.g., 2–8 64-byte cache lines)
• We suppose each tree node is eight 64-byte cache lines large.
Persistent B$^+$-Trees in Non-Volatile Main Memory

- Chen et al. [2011] proposed PCM-friendly B$^+$-Tree, which leaves leaf node unsorted, and uses bitmask for used items.

![B$^+$-Tree Diagram](image)

**Figure 3:** B$^+$-tree with sorted leaves. Figure by Chen et al. [2011].
Chen et al. [2011] proposed PCM-friendly B$^+$-Tree, which leaves leaf node unsorted, and uses bitmask for used items.

- Requires linear search, but reduces number of reads/writes.

**Figure 4:** Unsorted node with and without bitmap. Figure by Chen et al. [2011].
Normal sequence of actions, leading to the correct end state shown at the top in fig. 5:

- Move 9, 7 and 4 one slot to the right

Figure 5: Possible inconsistencies upon failure. Figure by Chen and Jin [2015]
Normal sequence of actions, leading to the correct end state shown at the top in fig. 5:

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

**Figure 5:** Possible inconsistencies upon failure. Figure by Chen and Jin [2015]
Normal sequence of actions, leading to the correct end state shown at the top in fig. 5:

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

**Figure 5:** Possible inconsistencies upon failure. Figure by Chen and Jin [2015]
· **clflush**: invalidates the cache line on all levels of cache and broadcasts invalidation to all CPUs.
clflush and mfence Instructions

- **clflush**: invalidates the cache line on all levels of cache and broadcasts invalidation to all CPUs.
- **mfence**: guarantees that all writes and reads that happened before **mfence** are globally visible before any of writes or reads that happen after **mfence**.
Figure 6: Influence of clflush and mfence on performance on an Intel Xeon x86-64 machine with 64-byte cache lines. Figure by Chen and Jin [2015].
Metrics for Persistent Data Structures

Which metrics are important for persistent data structures on NVMM?
Which metrics are important for persistent data structures on NVMM?

• $N_w$ – number of writes.
Which metrics are important for persistent data structures on NVMM?

- $N_w$ – number of writes.
- $N_{clf}$ – number of clflush.
Metrics for Persistent Data Structures

Which metrics are important for persistent data structures on NVMM?

- $N_w$ – number of writes.
- $N_{clf}$ – number of clflush.
- $N_{mf}$ – number of mfence.
Existing Solutions
Undo-Redo Logging

- For any memory change store record address, old value, and new value.
Undo-Redo Logging

- For any memory change store record address, old value, and new value.
- Undo logging does not affect individual operation time complexity.
Undo-Redo Logging

Figure 7: NVMM write protected by undo-redo logging

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, *addr, newValue);  
8:   *addr=newValue;  
9: end procedure  
10: procedure COMMITNEWREDO  
11:   log.clflush_mfence();  
12: end procedure
Figure 8: Optimized redo-only logging

1: procedure WRITE_REDO_ONLY(addr, newValue)
2:    log.write(addr, *addr, newValue);
3: end procedure
4: procedure COMMIT_REDO_WRITES(addr, newValue)
5:    log.clflush_mfence();
6: for addr, newValue in log do
7:    *addr=newValue;
8: end for
9: end procedure
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- The insertion moves on average $m/2$ entries, inserts a new entry, and increments the number field.
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- The insertion moves on average $m/2$ entries, inserts a new entry, and increments the number field.
- This requires writing $m + 3$ words.
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- The insertion moves on average $m/2$ entries, inserts a new entry, and increments the number field.
- This requires writing $m + 3$ words.
- For each word write, the undo-redo logging incurs 3 extra writes, a `clflush` and `mfence`. 
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- The insertion moves on average $m/2$ entries, inserts a new entry, and increments the number field.
- This requires writing $m + 3$ words.
- For each word write, the undo-redo logging incurs 3 extra writes, a `clflush` and `mfence`.
- $N_w = 4m + 12$, $N_{clf} = N_{mf} = m + 3$
What are \( N_w \), \( N_{clf} \), and \( N_{mf} \) for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- A PCM-friendly \( B^+ \)-Tree writes the new index entry to an unused location and updates the number/bitmap field.

\[ N_w = 2 \cdot 3 + 1 \cdot 4 = 10 \]
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- A PCM-friendly $B^+$-Tree writes the new index entry to an unused location and updates the number/bitmap field.
- We use `NewRedo` for the former and `WriteUndoRedo` for the latter.
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- A PCM-friendly $B^+$-Tree writes the new index entry to an unused location and updates the number(bitmap) field.
- We use `NewRedo` for the former and `WriteUndoRedo` for the latter.
- For each word write, the undo-redo logging incurs 3 extra writes, a `clflush` and `mfence`.
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- A PCM-friendly $B^+$-Tree writes the new index entry to an unused location and updates the number(bitmap) field.
- We use `NewRedo` for the former and `WriteUndoRedo` for the latter.
- For each word write, the undo-redo logging incurs 3 extra writes, a `clflush` and `mfence`.
- $N_w = 2 \cdot 3 + 1 \cdot 4 = 10$ and $N_{clf} = N_{mf} = 2$
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- If node splitting occurs, the only difference introduced by unsorted leaves is for the leaf node.
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- If node splitting occurs, the only difference introduced by unsorted leaves is for the leaf node.
- For deletion, using bitmap provides advantage:
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- If node splitting occurs, the only difference introduced by unsorted leaves is for the leaf node.
- For deletion, using bitmap provides advantage:
  - For regular $B^+\text{-Tree}$ $N_w = 4m$, $N_{clf} = N_{mf} = m$
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- If node splitting occurs, the only difference introduced by unsorted leaves is for the leaf node.
- For deletion, using bitmap provides advantage:
  - For regular $B^+$-Tree $N_w = 4m$, $N_{clf} = N_{mf} = m$
  - In unsorted case, $N_w = 12$, $N_{clf} = N_{mf} = 3$
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- If node splitting occurs, the only difference introduced by unsorted leaves is for the leaf node.
- For deletion, using bitmap provides advantage:
  - For regular $B^+$-Tree $N_w = 4m$, $N_{clf} = N_{mf} = m$
  - In unsorted case, $N_w = 12$, $N_{clf} = N_{mf} = 3$
  - With bitmaps, $N_w = 4$, $N_{clf} = N_{mf} = 1$
• Create a copy of the node we are going to change and then swap it.
Shadowing

• Create a copy of the node we are going to change and then swap it.

• Naively, propagate to the parent.
Shadowing

- Create a copy of the node we are going to change and then swap it.
- Naively, propagate to the parent.
- Short-circuit: use atomic writes to change parent.
Shadowing

• Create a copy of the node we are going to change and then swap it.
• Naively, propagate to the parent.
• Short-circuit: use atomic writes to change parent.
• Leaf sibling pointers make everything harder.
Figure 9: Shadowing for insertion.

1: procedure
    \texttt{INSERTTOLEAF(leaf,newEntry,parent,ppos,sibling)}
2:     copyLeaf = \texttt{AllocNode();}
3:     \texttt{NodeCopy(copyLeaf, leaf);}
4:     \texttt{Insert(copyLeaf, newEntry);}
5:     \texttt{for i=0; i < copyLeaf.UsedSize(); i+=64 do}
6:         \texttt{clflush(&copyleaf + i);}
7:     \texttt{end for}
8:     \texttt{WriteRedoOnly(&parent.ch[ppos], copyLeaf);}
9:     \texttt{WriteRedoOnly(&sibling.next, copyLeaf);}
10:    \texttt{CommitRedoWrites();}
11:    \texttt{FreeNode(leaf);}
12: \texttt{end procedure}
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- Copying the node and inserting requires $2m + 4$ writes for copying and inserting. WriteRedoOnlys, the actual pointer updates, and AllocNode require 7 more.

$N_{clf} = \frac{2m + 4}{64} + 1 + 1 = 0.25m + 2$.
Shadowing

What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- Copying the node and inserting requires $2m + 4$ writes for copying and inserting. WriteRedoOnlys, the actual pointer updates, and AllocNode require 7 more.

- $N_{clf} = (2m + 4) \cdot \frac{8}{64} + 1 + 1 = 0.25m + 2.5$
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- Copying the node and inserting requires $2m + 4$ writes for copying and inserting. WriteRedoOnlys, the actual pointer updates, and AllocNode require 7 more.
- $N_{clf} = (2m + 4) \cdot \frac{8}{64} + 1 + 1 = 0.25m + 2.5$
- CommitRedoWrites and AllocNode require mfence, thus thus $N_{mf} = 2$. 
Shadowing

What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- Copying the node and inserting requires $2m + 4$ writes for copying and inserting. WriteRedoOnlys, the actual pointer updates, and AllocNode require 7 more.
- $N_{clf} = (2m + 4) \cdot \frac{8}{64} + 1 + 1 = 0.25m + 2.5$
- CommitRedoWrites and AllocNode require mfence, thus thus $N_{mf} = 2$.
- Since shadowing requires copying the whole node, unsorted leaves do not provide advantage.
What are $N_w$, $N_{clf}$, and $N_{mf}$ for sorted, unsorted leaf, and unsorted leaf with bitmap cases?

- Copying the node and inserting requires $2m + 4$ writes for copying and inserting. `WriteRedoOnly`s, the actual pointer updates, and `AllocNode` require 7 more.

  - $N_{clf} = (2m + 4) \cdot \frac{8}{64} + 1 + 1 = 0.25m + 2.5$
  
  - `CommitRedoWrites` and `AllocNode` require `mfence`, thus thus $N_{mf} = 2$.

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

- Deletion is similar
Write-Atomic $B^+$-Trees
• Atomic write to commit all changes
Design Goals

- Atomic write to commit all changes
- Minimize the movement of index entries
Design Goals

• Atomic write to commit all changes
• Minimize the movement of index entries
• Good search performance
Slot array

- We use slot array, a sorted index of entries in the node

Figure 10: Different options for slot-bitmap combinations. Figure by Chen and Jin [2015]
Slot array

• We use slot array, a sorted index of entries in the node
• Slot array requires less memory

Figure 10: Different options for slot-bitmap combinations. Figure by Chen and Jin [2015]
• We use slot array, a sorted index of entries in the node
• Slot array requires less memory
• If the node is small, we can get rid of bitmap

Figure 10: Different options for slot-bitmap combinations. Figure by Chen and Jin [2015]
Slot array

- We use slot array, a sorted index of entries in the node
- Slot array requires less memory
- If the node is small, we can get rid of bitmap
- Else, the last bit of bitmap determines validity of the slot array

Figure 10: Different options for slot-bitmap combinations. Figure by Chen and Jin [2015]
1. Recover slot array if it is invalid
2. Mark slot array as invalid
3. Write and flush new entry
4. Modify and flush slot array
5. Emit `mfence` to ensure new entry and slot array are stable
6. Flush bitmap and emit one more `mfence`
Analysis of wB$^+$-Trees

• Usage of slot array restore the logarithmic complexity of the search
Analysis of wB$^+$-Trees

• Usage of slot array restore the logarithmic complexity of the search

• Variable length-keys can be supported by replacing the key with pointer
Analysis of wB\(^+\)-Trees

- Usage of slot array restore the logarithmic complexity of the search
- Variable length-keys can be supported by replacing the key with pointer
- Insertion without splitting requires
  \[ N_w = 0.125m + 4.25, \quad N_{clf} = \frac{1}{64}m + 3\frac{1}{32}, \quad \text{and} \quad N_{mf} = 3 \]
Analysis of wB\textsuperscript{+}-Trees

- Usage of slot array restore the logarithmic complexity of the search
- Variable length-keys can be supported by replacing the key with pointer
- Insertion without splitting requires
  \[ N_w = 0.125m + 4.25, \quad N_{clf} = \frac{1}{64}m + \frac{31}{32}, \quad \text{and} \quad N_{mf} = 3 \]
- Alternatively, keeping leaves bitmap-only reduces to
  \[ N_w = 3 \quad \text{and} \quad N_{clf} = N_{mf} = 2 \]
Analysis of wB+ -Trees

• Usage of slot array restore the logarithmic complexity of the search
• Variable length-keys can be supported by replacing the key with pointer
• Insertion without splitting requires
  \[ N_w = 0.125m + 4.25, \quad N_{clf} = \frac{1}{64}m + 3\frac{1}{32}, \quad \text{and} \quad N_{mf} = 3 \]
• Alternatively, keeping leaves bitmap-only reduces to
  \[ N_w = 3 \quad \text{and} \quad N_{clf} = N_{mf} = 2 \]
• Deletion is similar up to additive constant.
## Comparison of different methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Insertion</th>
<th>Deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N_w$</td>
<td>$N_{clf}$</td>
</tr>
<tr>
<td>$B^+$ tree (log)</td>
<td>$4m + 12$</td>
<td>$m + 3$</td>
</tr>
<tr>
<td>ULB (log)</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>$B^+$ tree (shadow)</td>
<td>$2m + 11$</td>
<td>$0.25m + 2.5$</td>
</tr>
<tr>
<td>wB$^+$ tree</td>
<td>$0.125m + 4.25$</td>
<td>$m/64 + 31/32$</td>
</tr>
<tr>
<td>wB$^+$ tree (bitmap)</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Experimental Results
Experimental Results

• Real machine modeling DRAM-like fast NVMM
Experimental Results

- Real machine modeling DRAM-like fast NVMM
  - The best wB⁺-Tree result is 1.2–1.6× better performance
Experimental Results

- Real machine modeling DRAM-like fast NVMM
  - The best $\text{wB}^+$-Tree result is 1.2–1.6× better performance
- Simulation modeling PCM-based NVMM
  - 1.2–5.6× speedups
  - Getting rid of bitmap is important
Experimental Results

- Real machine modeling DRAM-like fast NVMM
  - The best wB⁺-Tree result is 1.2–1.6× better performance

- Simulation modeling PCM-based NVMM
  - The best wB⁺-Tree result is 1.5–2.4× better performance
Experimental Results

- Real machine modeling DRAM-like fast NVMM
  - The best $wB^+$-Tree result is 1.2–1.6× better performance
- Simulation modeling PCM-based NVMM
  - The best $wB^+$-Tree result is 1.5–2.4× better performance
- Real machine experiments for trees with string keys
Experimental Results

- Real machine modeling DRAM-like fast NVMM
  - The best $wB^+$-Tree result is $1.2-1.6 \times$ better performance

- Simulation modeling PCM-based NVMM
  - The best $wB^+$-Tree result is $1.5-2.4 \times$ better performance

- Real machine experiments for trees with string keys
  - $1.2-5.6 \times$ speedups
Experimental Results

- Real machine modeling DRAM-like fast NVMM
  - The best wB⁺-Tree result is 1.2–1.6× better performance
- Simulation modeling PCM-based NVMM
  - The best wB⁺-Tree result is 1.5–2.4× better performance
- Real machine experiments for trees with string keys
  - 1.2–5.6× speedups
  - Getting rid of bitmap is important
**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.) Figure by Chen and Jin [2015]
Figure 12: Wear, energy, and clflush/mfence counts of index operations for fig. 11(b) with 8-line nodes. Figure by Chen and Jin [2015]
Experimental Results

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.) Figure by Chen and Jin [2015]
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). Figure by Chen and Jin [2015]
Experimental Results

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.) Figure by Chen and Jin [2015]
Conclusions
Conclusions

- Persistence is important for NVMM data structures
Conclusions

• Persistence is important for NVMM data structures
• Undo-redo logging and shadowing perform extensive NVM writes and cacheline flushes
Conclusions

• Persistence is important 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
Conclusions

• Persistence is important 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
Conclusions

• Persistence is important 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⁺-Trees improve the insertion and deletion performance, while achieving good search performance
Questions?