SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory

Xuchuan Luo¹, Pengfei Zuo², Jiacheng Shen³, Jiazhen Gu³, Xin Wang¹,⁴, Michael R. Lyu³, and Yangfan Zhou¹,⁴

¹School of Computer Science, Fudan University
²Huawei Cloud
³The Chinese University of Hong Kong
⁴Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China
Disaggregated Memory (DM)

**Benefits:**
- Resource utilization
- Elasticity

**Compute Nodes (CNs)**

**Memory Nodes (MNs)**

Fast Network (e.g., RDMA)
Tree Indexes

B+ Tree

- Each internal node stores **entire keys**
- Each leaf node holds **multiple KVs**

Radix Tree

- Each internal node stores **partial keys**
- Each leaf node holds a **single KV**
Tree Indexes on Disaggregated Memory

Existing tree indexes on DM are based on the B+ tree: FG [1], Sherman [2]

Problem:
Read/write amplifications of B+ trees
Exacerbate the network bandwidth bottleneck of DM

Tree Indexes on Disaggregated Memory

Read and write amplification factors:

<table>
<thead>
<tr>
<th></th>
<th>B+ Tree</th>
<th>Sherman</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read amplification</td>
<td>≈ 32</td>
<td>≈ 33</td>
</tr>
<tr>
<td>Write amplification</td>
<td>≈ 32</td>
<td>≈ 1</td>
</tr>
</tbody>
</table>

Cached internal nodes

span size = 32

B+ Tree
Tree Indexes on Disaggregated Memory

Read and write amplification factors:

<table>
<thead>
<tr>
<th></th>
<th>B+ Tree</th>
<th>Sherman</th>
<th>Radix Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read amplification</td>
<td>≈ 32</td>
<td>≈ 33</td>
<td>≈ 1</td>
</tr>
<tr>
<td>Write amplification</td>
<td>≈ 32</td>
<td>≈ 1</td>
<td>≈ 1</td>
</tr>
</tbody>
</table>

Cached internal nodes

B+ Tree

radix size = 32

Radix Tree

span size = 1
Tree Indexes on Disaggregated Memory

Read and write amplification factors:

<table>
<thead>
<tr>
<th></th>
<th>B+ Tree</th>
<th>Sherman</th>
<th>Radix Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read amplification</td>
<td>( \approx 32 )</td>
<td>( \approx 33 )</td>
<td>( \approx 1 )</td>
</tr>
<tr>
<td>Write amplification</td>
<td>( \approx 32 )</td>
<td>( \approx 1 )</td>
<td>( \approx 1 )</td>
</tr>
</tbody>
</table>

**Insight:** The radix tree is more suitable for DM than the B+ tree due to smaller read/write amplifications.

**Our Idea:** Using radix tree to build a high-performance tree index on DM.
Challenge 1: Expensive Lock-based Concurrency Control

Lock-based concurrency control of radix trees causes poor write performance.

Clients

\[ \vdash \vdash \vdash \ldots \vdash \vdash \]

\textit{computing-side}

\underline{memory-side}

\textit{Coarse-grained Lock}
Expensive on DM

Radix Tree
Challenge 2: Bounded Memory-side IOPS

Inter-client redundant I/Os on DM waste the limited IOPS of RNICs

- Redundant Read I/Os: Peak throughput ↓
- Redundant Write I/Os: Concurrency conflicts ↑, Peak throughput ↓

Small-sized Read/Write Operations

Radix Tree
Challenge 3: Complicated Computing-side Cache Validation

Structural features of radix trees complicate the problem of cache invalidation.

- **Common Optimizations:**
  - Path compression
  - Adaptive nodes

Is a cache entry outdated?

Radix Tree
Challenge Summary

1. Expensive lock-based concurrency control
2. Bounded memory-side IOPS
3. Complicated computing-side cache validation
DiSaggregated-meMory-friendly Adaptive Radix Tree (SMART)

1. Expensive lock-based concurrency control
   - **Solution 1**: Hybrid concurrency control scheme

2. Bounded memory-side IOPS
   - **Solution 2**: Read-delegation and write-combining technique

3. Complicated computing-side cache validation
   - **Solution 3**: Reverse check mechanism
Hybrid Concurrency Control

Problem: Expensive lock-based concurrency control

The internal node of the state-of-the-art radix tree [3]:

Store partial keys and child pointers separately

Hybrid Concurrency Control

**Problem**: Expensive lock-based concurrency control

> **Solution**: Lock-free internal nodes

**Key Idea**:
- Embed each partial key and child pointer into an 8-byte slot
- 8-byte header
Hybrid Concurrency Control

**Problem**: Out-of-place update causes cache thrashing

- **CNs**: Clients
- **MNs**: Radix Tree
- **Cache leaf node addresses**: Out-of-place update changes leaf node addresses frequently
Hybrid Concurrency Control

**Problem:** Out-of-place update causes cache thrashing

- **Solution:** Lock-based update-in-place leaf nodes

**Key Idea:**
- Write-write conflict ➢ *Rear embedded lock*
  - Combine lock release with writing back
- Read-write conflict ➢ *Checksum-based method*
  - Writer: checksum := CRC(KV)
  - Reader: checksum == CRC(KV)?
Read Delegation and Write Combining

**Problem**: Redundant read I/Os

- Reading the same key from the same CN

![Diagram showing identical tree search and redundant read I/Os]
Read Delegation and Write Combining

**Problem:** Redundant read I/Os

- **Solution:** Read delegation

**Key Idea:**
- Choose a delegation client on each CN to execute the same read.
Read Delegation and Write Combining

**Problem:** Redundant read I/Os

- **Solution:** Read delegation

**Key Idea:**
- Choose a delegation client on each CN to execute the same read
- Use local locks to collect concurrent identical reads
Read Delegation and Write Combining

**Problem:** Redundant write I/Os

Writing the same key from the same CN

- W W W W

Remote Synchronization
Problem: Redundant write I/Os

Solution: Write combining

Key Idea:
• Combine these writes on a local write combining buffer (WCB)
Problem: Redundant write I/Os

Solution: Write combining

Key Idea:
- Combine these writes on a local write combining buffer (WCB)
- Use local locks to collect concurrent writes with the same target key
Reverse Check

Problem: Cache invalidation of the radix tree

Radix Tree

Structural modifications from other CNs
Reverse Check

**Problem:** Cache invalidation of the radix tree

- **Solution:** Reverse check mechanism

**Key Idea:**
- Store check information in each remote node
- Check: check information == cache content?
Reverse Check

**Problem**: Cache invalidation of the radix tree

- **Solution**: Reverse check mechanism

**Key Idea**:
- Store check information in each remote node
- Check: check information == cache content ?

**Example**:
- Cache invalidation: *adjustments on the parent-child relationship* of remote nodes
- Store a reverse pointer in the front of each node
- Check: *Reverse Pointer == cached Node Address*?
More Details

- Concurrent operations
- Hash-based local locks
- Complete reverse check designs
- Support for variable-sized keys and values
- ......
Evaluation

Workloads

• YCSB workloads
• 2 key types: integer, string

Comparisons

• Sherman [SIGMOD’22]
  • The state-of-the-art B+ tree design on DM
• ART [ICDE’13]
  • The state-of-the-art radix tree design
  • We port it to DM
Performance Comparison

- Compared with Sherman, SMART achieves up to:
  - 6.1x higher throughput and 1.4x lower latency under write-intensive workloads
  - 2.8x higher throughput with similar latency under read-only workloads
Factor Analysis for SMART design

<table>
<thead>
<tr>
<th>YCSB LOAD</th>
<th>YCSB A</th>
<th>YCSB B</th>
<th>YCSB C</th>
<th>YCSB D</th>
</tr>
</thead>
<tbody>
<tr>
<td>100% insert</td>
<td>50% read, 50% update</td>
<td>95% read, 5% update</td>
<td>100% read</td>
<td>95% read, 5% insert</td>
</tr>
</tbody>
</table>

- Start with the ART design and apply each proposed technique one by one
The lock-free internal node brings **1.5x improvement** in throughput under the YCSB LOAD workload.
The update-in-place leaf node brings **1.5x improvement** in throughput under the YCSB B workload.
## Factor Analysis for SMART design

<table>
<thead>
<tr>
<th>YCSB LOAD</th>
<th>YCSB A</th>
<th>YCSB B</th>
<th>YCSB C</th>
<th>YCSB D</th>
</tr>
</thead>
<tbody>
<tr>
<td>100% insert</td>
<td>50% read, 50% update</td>
<td>95% read, 5% update</td>
<td>100% read</td>
<td>95% read, 5% insert</td>
</tr>
</tbody>
</table>

- **The rear embedded lock brings 3.0x improvement in throughput under the YCSB A workload**

![Graph showing performance improvements](graph.png)
The read delegation brings 1.1x improvement in throughput under the YCSB C workload.
The write combining brings **1.1x improvement** in throughput under the YCSB A workload.
Conclusion

- Existing tree indexes on DM are based on B+ trees, which suffer from large inherent read and write amplifications.

- We propose **SMART**, a high-performance adaptive radix tree for DM:
  - Hybrid concurrency control scheme
  - Read-delegation and write-combining technique
  - Reverse check mechanism

- SMART outperforms the state-of-the-art B+ tree on DM by up to **6.1x** under YCSB write-intensive workloads and **2.8x** under YCSB read-only workloads.
Thank you! Q&A

https://github.com/dmemeSYS/SMART