



ROART: Range-query Optimized Persistent ART

Shaonan Ma[1], Kang Chen[1], Shimin Chen[2, 3], Mengxing Liu[1], Jianglang Zhu[1], Hongbo Kang[1], Yongwei Wu[1]

[1] Tsinghua University, [2] SKL of Computer Architecture, ICT, CAS,
[3] University of Chinese Academy of Sciences

Persistent Indexes

- Building blocks for key-value stores and databases
- ~10 years of simulation-based research
  - NVDIMM or DRAM with latencies



Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

Deukyeon Hwang *UNIST* · Wook-Hee Kim *UNIST* · Youjip Won *Hanyang University* · Beomseok Nam *UNIST*

Abstract

With the emergence of byte-addressable persistent memory (PM), a cache line, instead of a page, is expected to be the unit of data transfer between volatile and non-volatile devices, but the failure-atomicity of write operations is guaranteed in the granularity of 8 bytes rather than cache lines. This granularity mismatch problem has generated interest in redesigning block-based data structures such as B+-trees. However, various methods of modifying B+-trees for PM degrade the efficiency of B+-trees, and attempts have been made to use in-memory data structures for PM.

In this study, we develop Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalance (FAIR) algorithms to resolve the granularity mismatch problem. Every 8-byte store instruction used in the FAST and FAIR algorithms transforms a B+-tree into another consistent state or a *transient inconsistent* state that read operations can tolerate. By making read operations tolerate transient inconsistency, we can avoid expensive copy-on-write, logging, and even the necessity of read latches so that read transactions can be non-blocking. Our experimental results show that legacy B+-trees with FAST and FAIR schemes outperform the state-of-the-art persistent indexing structures by a large margin.

1 Introduction

Recent advances in byte-addressable persistent memories (PM), such as phase change memory[48], spin-transfer torque MRAM[17], and 3D Xpoint[2] have opened up new opportunities for the applications to warrant durability or persistency without relying on legacy heavy-duty interfaces offered by the filesystem and/or by the block device [21, 24].

In legacy block devices, B+-tree has been one of the most popular data structures. The primary advantage of a B+-tree is its efficient data access performance due to its high degree of node fan-out, a balanced tree height, and dynamic resizing. Besides, the large CPU cache in modern processors[11] enables B+-tree to exhibit a good cache line locality. As a result, B+-tree shows good performance as an in-memory data structure as well. Thus, the cache-conscious variants of B+-tree including CSS-tree [36], CSB-tree [37], and FAST [19] are shown to perform better than legacy binary counterparts such as T-tree [26].

Byte-addressable PM raises new challenges in using B+-tree because legacy B+-tree operations are implemented upon the assumption that block I/O is failure atomic. In modern computer architectures, the unit of atomicity guarantee and the unit of transfer do not coincide. The unit of atomicity in memory operations corresponds to a word, e.g. 64 bits whereas the unit of transfer between the CPU and memory corresponds to a cache line, e.g. 64 Bytes. This granularity mismatch is of no concern in current memory since it is volatile. When the memory is non-volatile, unexpected system failures may cause the result of incomplete cache flush operations to be externally visible after the system recovers. To make the granularity mismatch problem even worse, modern processors often make the most use of memory bandwidth by changing the order of memory operations. Besides, recently proposed memory persistency models such as *epoch persistency* [9] even allow cache lines to be written back out of order. To prevent the reordering of memory write operations and to ensure that the written data are flushed to PM without compromising the consistency of the tree structure, B+-trees for persistent memory use explicit memory fence operations and cache flush operations to control the order of memory writes [5, 42].

The recently proposed B+-tree variants such as NV-tree [49], FP-tree [34], and wB+-tree [5] have pointed out and addressed two problems of byte-addressable persistent B+-trees. The first problem is that a large number of fencing and cache flush operations are needed to maintain the sorted order of keys. And, the other problem is that the logging demanded by tree rebalancing operations

WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems

Se Kwon Lee, K. Hyun Lim†, Hyunsub Song, Beomseok Nam, Sam H. Noh
UNIST (Ulsan National Institute of Science and Technology), †Hongik University

Abstract

Recent interest in persistent memory (PM) has stirred development of index structures that are efficient in PM. Recent such developments have all focused on variations of the B-tree. In this paper, we show that the radix tree, which is another less popular indexing structure, can be more appropriate as an efficient PM indexing structure. This is because the radix tree structure is determined by the prefix of the inserted keys and also does not require tree rebalancing operations and node granularity updates. However, the radix tree as-is cannot be used in PM. As another contribution, we present three radix tree variants, namely, WORT (Write Optimal Radix Tree), WOART (Write Optimal Adaptive Radix Tree), and ART+CoW. Of these, the first two are optimal for PM in the sense that they only use one 8-byte failure-atomic write per update to guarantee the consistency of the structure and do not require any duplicate copies for logging or CoW. Extensive performance studies show that our proposed radix tree variants perform considerable better than recently proposed B-tree variants for PM such NVTree, wB+Tree, and FPTree for synthetic workloads as well as in implementations within Memcached.

1 Introduction

Previous studies on indexing structures for persistent memory (PM) have concentrated on B-tree variants. In this paper, we advocate that the radix tree can be better suited for PM indexing than B-tree variants. We present radix tree variant indexing structures that are optimal for PM in that consistency is always guaranteed by a single 8-byte failure-atomic write without any additional copies for logging or CoW.

Emerging persistent memory technologies such as phase-change memory, spin-transfer torque MRAM, and 3D Xpoint are expected to radically change the landscape of various memory and storage systems [4, 5, 7, 9, 10, 14]. In the traditional block-based storage device, the failure atomicity unit, which is the update unit where consistent state is guaranteed upon any system failure, has been the disk block size. However, as persistent memory, which is byte-addressable and non-volatile, will be accessible though the memory bus rather than via the PCI interface, the failure atomicity unit for persistent memory is generally expected to be 8 bytes or no larger than a cache line [5, 6, 12, 13, 15, 19].

The smaller failure atomicity unit, however, appears to be a double-edged sword in the sense that though this allows for reduction of data written to persistent store, it can lead to high overhead to enforce consistency. This is because in modern processors, memory write operations are often arbitrarily reordered in cache line granularity and to enforce the ordering of memory write operations, we need to employ memory fence and cache line flush instructions [21]. These instructions have been pointed out as a major cause of performance degradation [3, 9, 15, 20]. Furthermore, if data to be written is larger than the failure-atomic write unit, then expensive mechanisms such as logging or copy-on-write (CoW) must be employed to maintain consistency.

Recently, several persistent B-tree based indexing structures such as NVTree [20], wB+Tree [3], and FP-Tree [15] have been proposed. These structures focus on reducing the number of calls to the expensive memory fence and cache line flush instructions by employing an append-only update strategy. Such a strategy has been shown to significantly reduce duplicate copies needed for schemes such as logging resulting in improved performance. However, this strategy does not allow these structures to retain one of the key features of B-trees, that is, having the keys sorted in the nodes. Moreover, this strategy is insufficient in handling node overflows as node splits involve multiple node changes, making logging necessary.

While B-tree based structures have been popular in-memory index structures, there is another such structure,

Write-Optimized Dynamic Hashing for Persistent Memory

Moohyeon Nam†, Hokeun Cha‡, Young-ri Choi†, Sam H. Noh†, Beomseok Nam‡
UNIST (Ulsan National Institute of Science and Technology)†
Sungkyunkwan University‡

Abstract

Low latency storage media such as byte-addressable persistent memory (PM) requires rethinking of various data structures in terms of optimization. One of the main challenges in implementing hash-based indexing structures on PM is how to achieve efficiency by making effective use of cachelines while guaranteeing failure-atomicity for *dynamic hash expansion and shrinkage*. In this paper, we present *Cacheline-Conscious Extendible Hashing* (CCEH) that reduces the overhead of dynamic memory block management while guaranteeing constant hash table lookup time. CCEH guarantees failure-atomicity without making use of explicit logging. Our experiments show that CCEH effectively adapts its size as the demand increases under the fine-grained failure-atomicity constraint and its maximum query latency is an order of magnitude lower compared to the state-of-the-art hashing techniques.

1 Introduction

In the past few years, there have been numerous efforts to leverage the byte-addressability, durability, and high performance of persistent memory (PM) [7, 13, 18, 32, 34, 39, 40, 45, 47]. In particular, latency critical transactions on storage systems can benefit from storing a small number of bytes to persistent memory. The fine-grained unit of data I/O in persistent memory has generated interest in redesigning block-based data structures such as B+-trees [2, 11, 20, 28, 46]. Although a large number of previous studies have improved tree-based indexing structures for byte-addressable persistent memory, only a few have attempted to adapt hash-based indexing structures to persistent memory [48, 49]. One of the main challenges in hash-based indexing for PM is in achieving efficient *dynamic rehashing* under the fine-grained failure-atomicity constraint. In this paper, we present Cacheline-Conscious Extendible Hashing (CCEH), which is a variant of extendible hashing [6] optimized for PM to minimize cacheline accesses and satisfy failure-atomicity without explicit logging.

Due to the static flat structure of hash-based indexes, they can achieve constant lookup time. However, static hashing does not come without limitations. Such traditional hashing schemes must typically estimate the size of hash tables and allocate sufficient buckets in advance. For certain applications, this is a feasible task. For example, in-memory hash tables in key-value stores play a role of fixed-sized buffer cache, i.e., recent key-value records replace old records. Hence, we can set the hash table size a priori based on the available memory space.

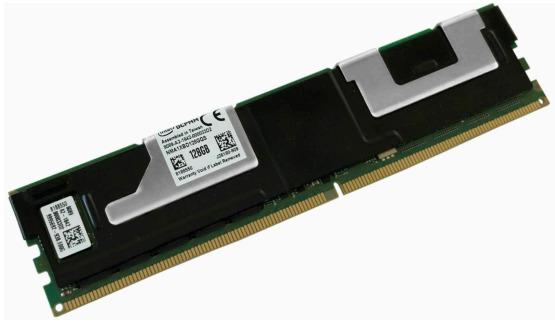
However, not all applications can estimate the hash table size in advance, with database systems and file systems being typical examples. If data elements are dynamically inserted and deleted, static fixed-sized hashing schemes suffer from hash collisions, overflows, or under-utilization. To resolve these problems, dynamic resizing must be employed to adjust the hash table size proportional to the number of records. In a typical situation where the load factor (bucket utilization) becomes high, a larger hash table must be created, and a *rehash* that moves existing records to new bucket locations must be performed.

Unfortunately, rehashing is not desirable as it degrades system throughput as the index is prevented from being accessed during rehashing, which significantly increases the tail latency of queries. To mitigate the rehashing overhead, various optimization techniques, such as *linear probing*, *separate chaining*, and *cuckoo hashing*, have been developed to handle hash collisions [4, 14, 25, 29, 31]. However, these optimizations do not address the root cause of hash collisions but defer the rehashing problem. As such, static hashing schemes have no choice but to perform expensive full-table (or 1/3-table [49]) rehash operations later if the allocated hash table size is not sufficient.

In light of PM, rehashing requires a large number of writes to persistent memory. As writes are expected to induce higher latency and higher energy consumption in PM, this further aggravates performance. Furthermore, with lifetime of PM expected to be shorter than DRAM, such extra writes can be detrimental to systems employing PM.

Motivations

- Commercial persistent memory was available in 2019.04
  - Intel Optane DC Persistent Memory
- Previous works almost only focused on performance



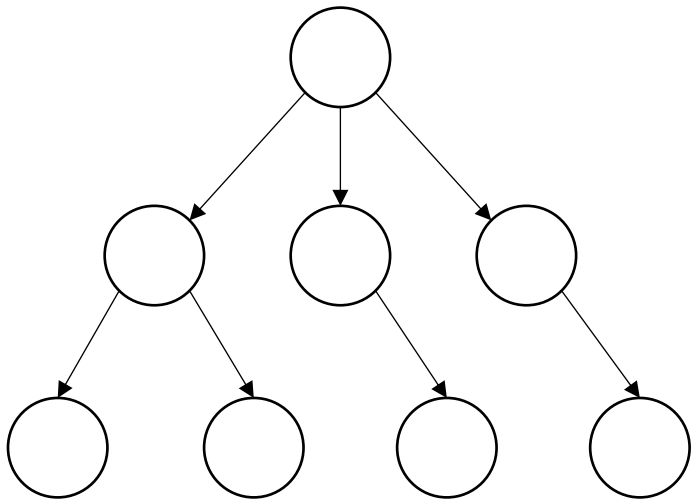
Performance is enough?

functionality **performance** **correctness**

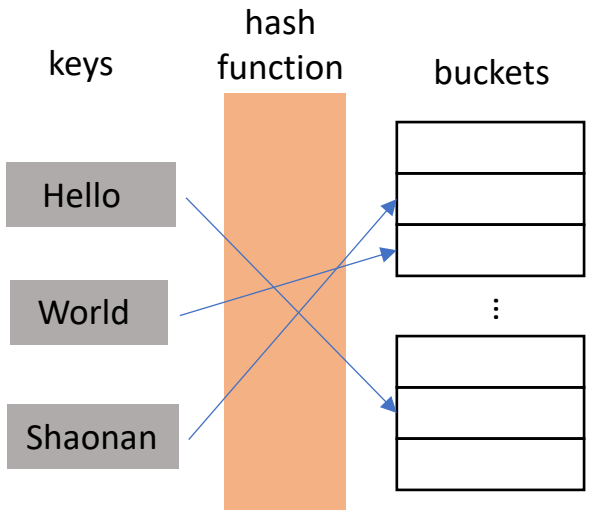
Functionality

- Range queries support

This paper focuses on



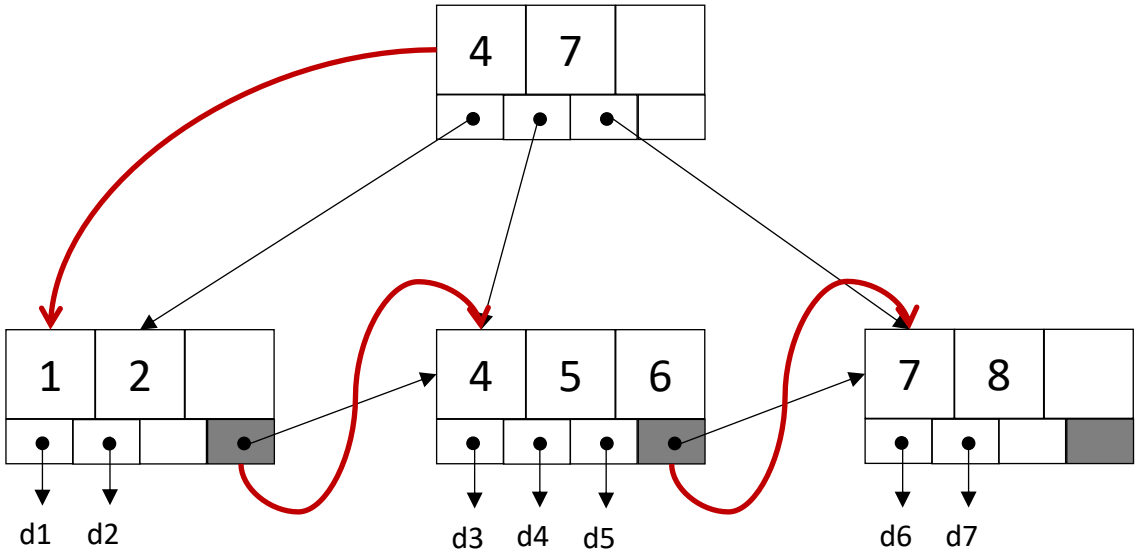
tree-based index
naturally support range query



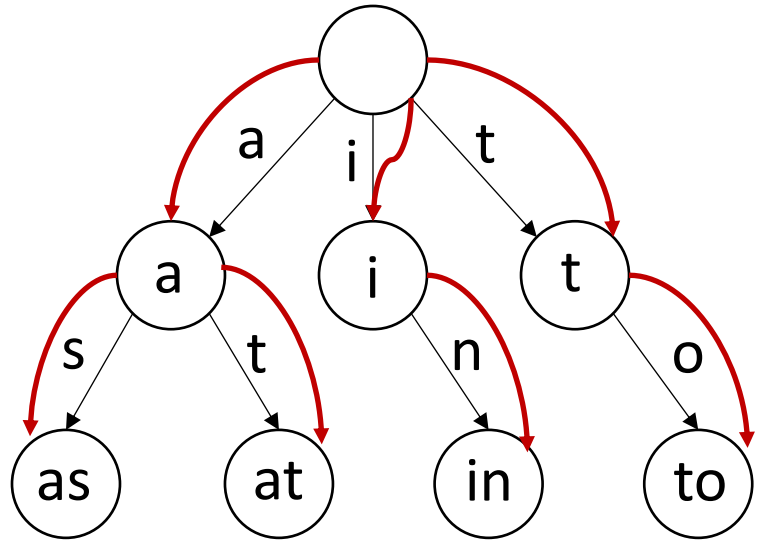
hash-based index

- Variable-sized keys support
  - Requirement of most KV stores (Redis, Memcached) and databases (MySQL, PostgreSQL)

Functionality——Range Queries Support



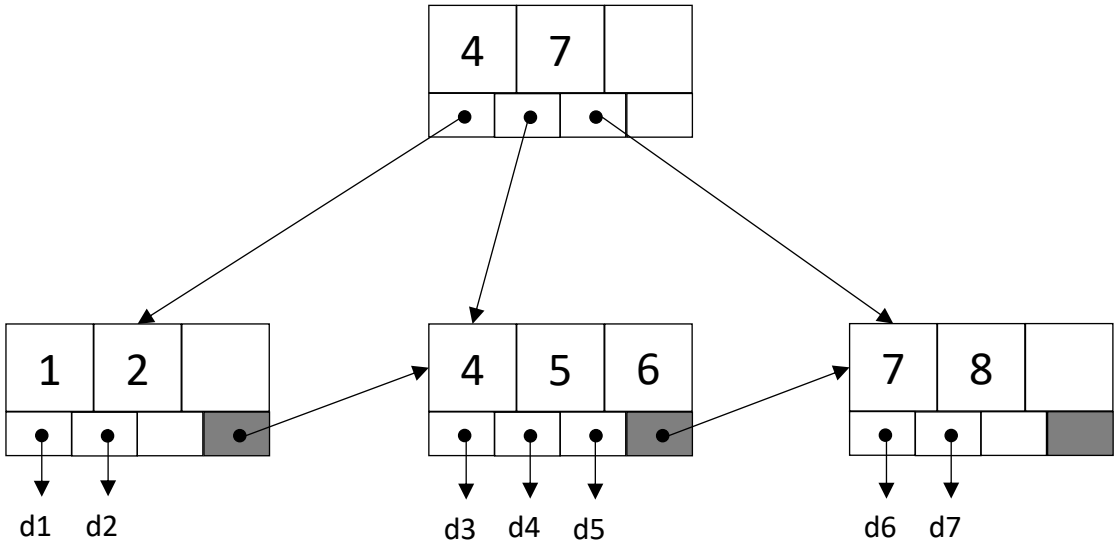
B+Tree



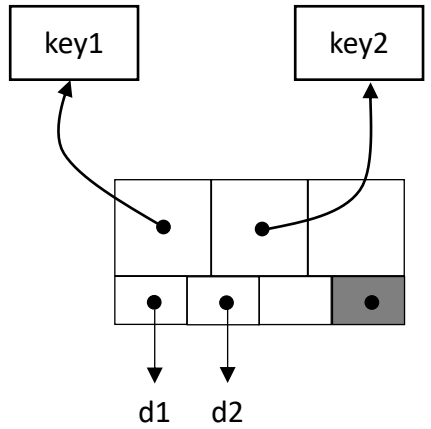
Trie

Trie suffers more pointer chasing in range queries

Functionality——Variable-sized Keys Support



B+Tree (fixed-sized keys)

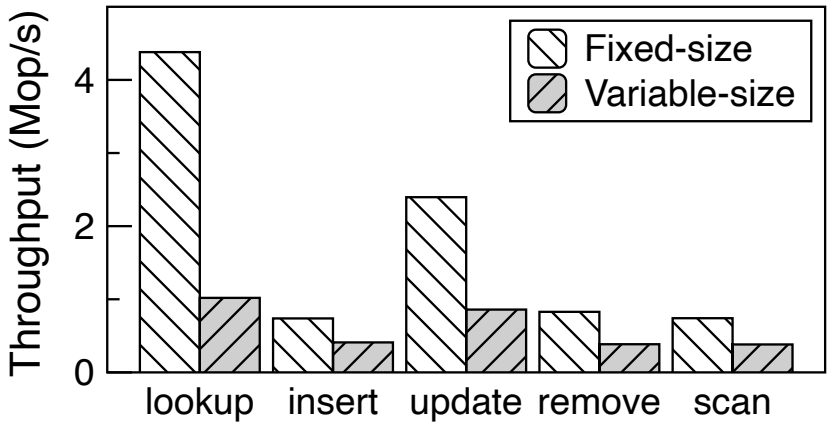


B+Tree Node (Variable-sized keys)

Extra string comparison
Extra pointer chasing
Extra persistence

- NVTree [FAST'15]
- wB+Tree [VLDB'15]
- FPTree [SIGMOD'16]
- LB+Tree [VLDB'20]
- FAST&FAIR [FAST'18]

Functionality——Evaluation



FAST&FAIR, 8-byte keys

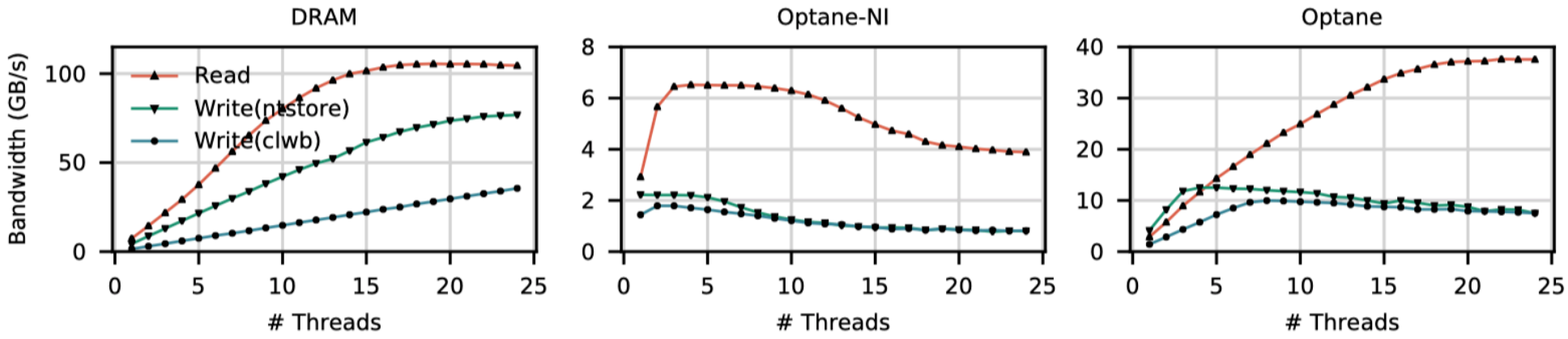
Performance degradation between two methods

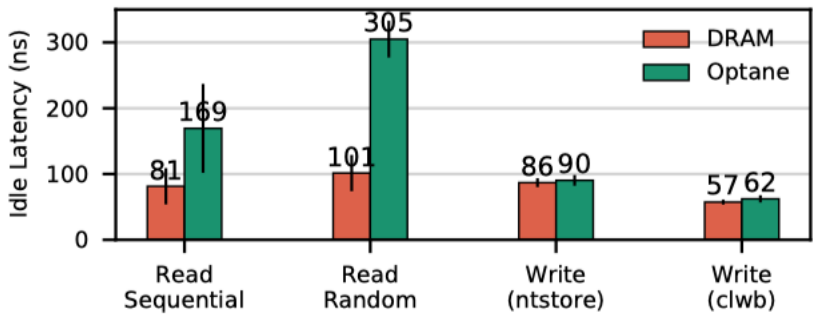
1.8~3.9x performance degradation

Shortcomings in B+Trees, a new structure is necessary

Our choice: Range-query Optimized Adaptive Radix Tree

Performance





PM Performance Characteristics [1]

- PM is 2-3x slower than DRAM
- persistent write has poor scalability of bandwidth

Key: number of store/flush/fence instructions

Our choice: ART-oriented optimization

[1] FAST'20, An Empirical Guide to the Behavior and Use of Scalable Persistent Memory.

Correctness

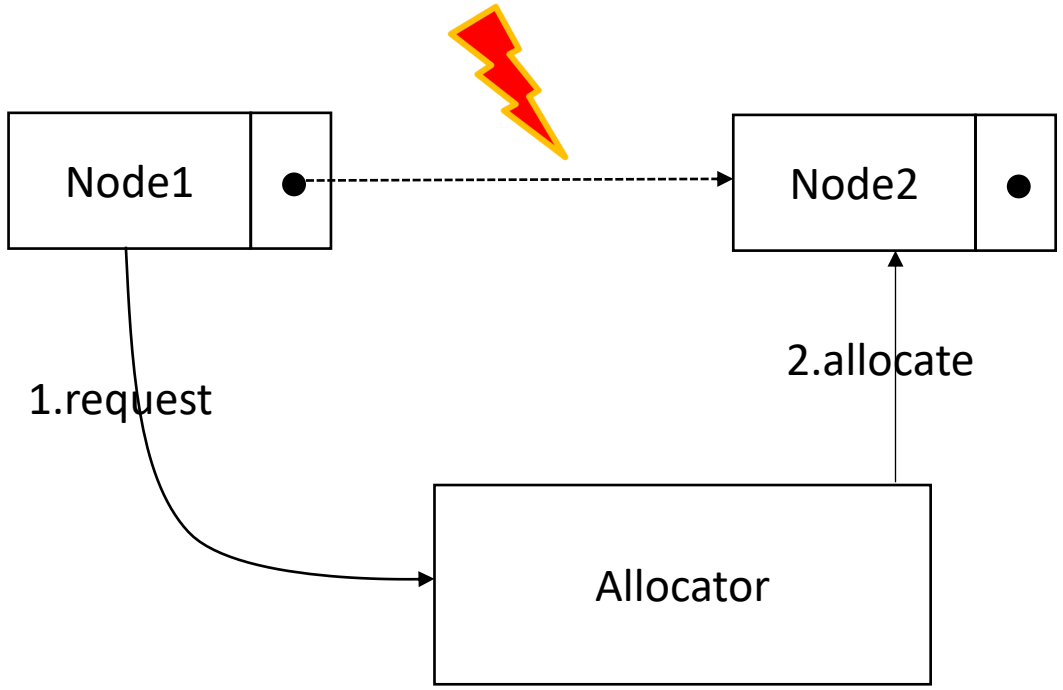
- Programming on PM is error-prone [1,2,3].

- Two aspects
  - Memory Safety
    - Persistent memory leaks during allocation and GC
  - Anomaly Resolution in persistent data structures (e.g. dirty read and lost update, more in paper)

[1] HotStorage'17, Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory.
[2] APSys'17, Programming for Non-Volatile Main Memory Is Hard.
[3] Programming Persistent Memory: A Comprehensive Guide for Developers

Correctness——Memory Safety

Volatile Allocator

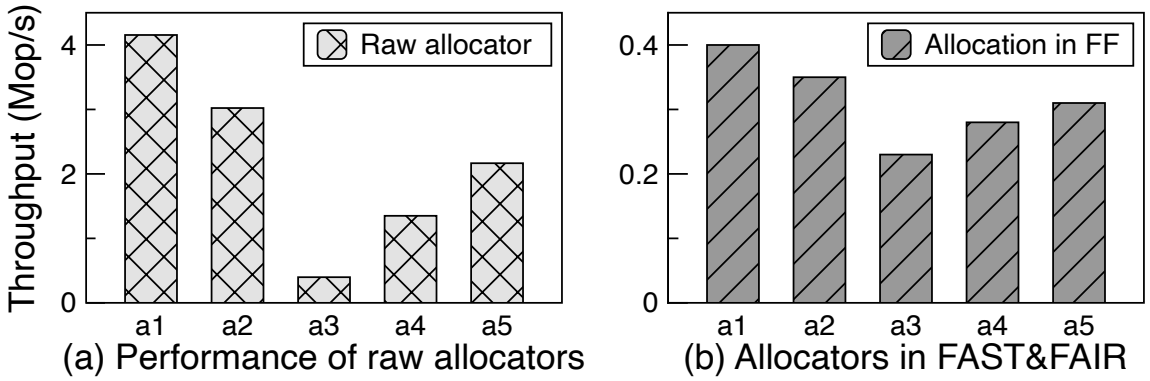


Persistent Allocators

- Logging-based
  - Constant time recovery
  - Slow allocation/deallocation
- Post-crach GC
  - Fast allocation/deallocation
  - Slow recovery

Correctness——Performance of Allocators

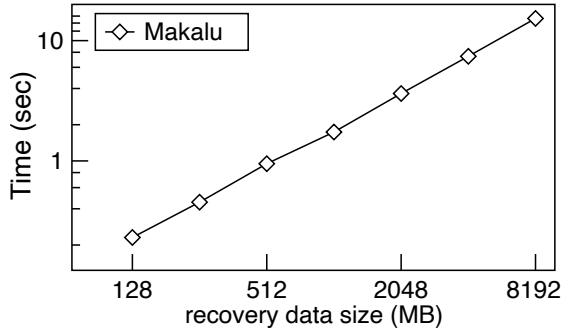
a1: malloc
a2: libvmmalloc
a3: PMDK
a4: nvm_malloc
a5: Makalu



Comparison of different allocators

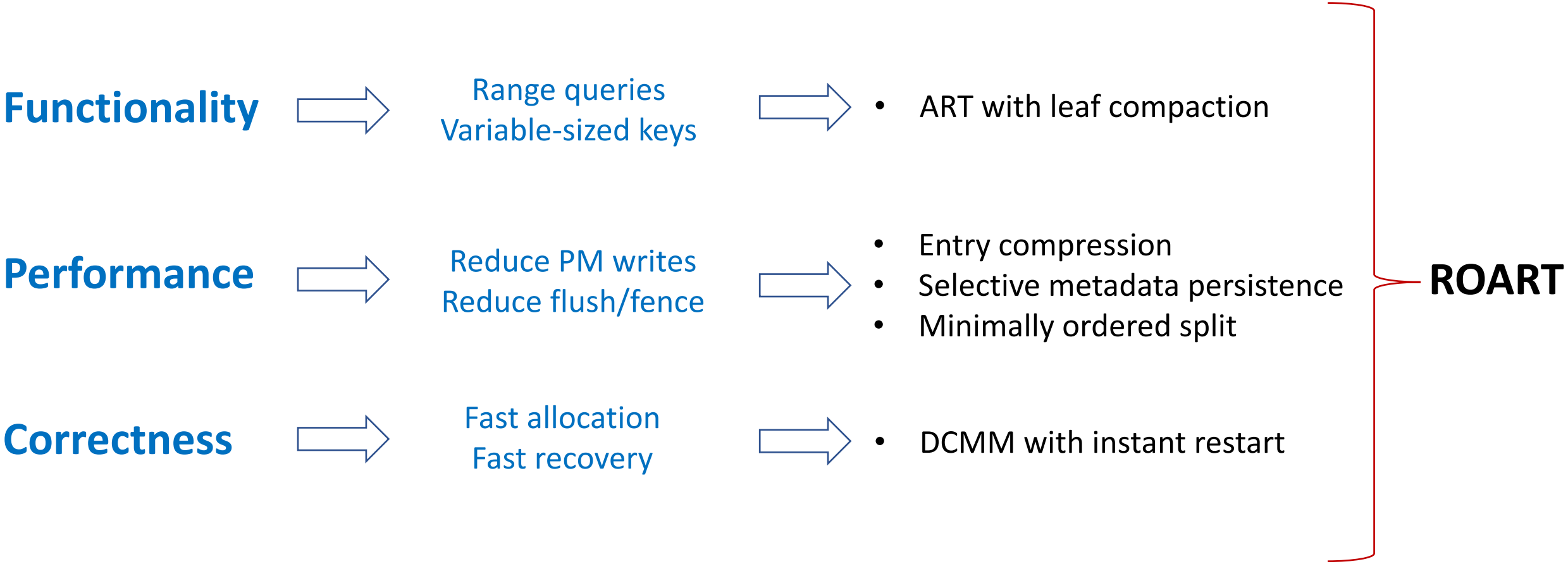
Existing persistent allocators have their own drawbacks

Our choice: a better persistent allocator

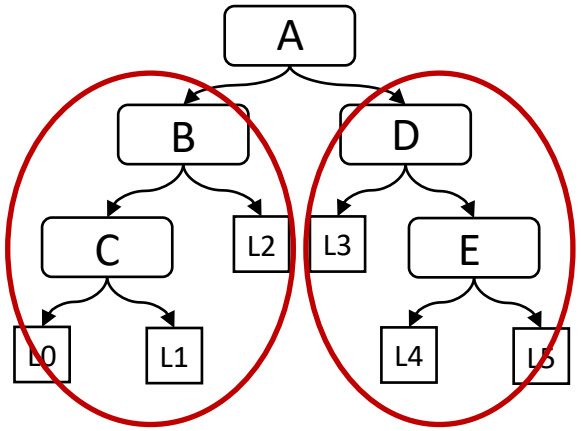


Recovery time of post-crash GC

Design Choices

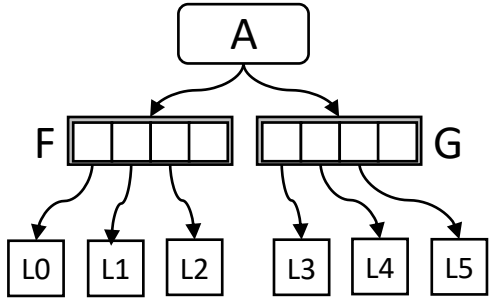


Leaf Compaction



ART [1]
(Adaptive Radix Tree)





ROART

Leaf compaction (LC) compacts the *pointers* of leaf nodes into a leaf array in the radix tree.

If a *subtree* of the radix tree has less than or equal to *m* leaf nodes, the subtree is compacted into a leaf array.

Disadvantage: leaf compaction causes key comparison in leaf arrays (e.g. Node F, G)

Solution: 16-bit fingerprints with pointers to reduce comparison

[1] ICDE'13, The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases.

More on Performance

Leaf Compaction

- Reduce height of tree -> accelerate traversal
- Reduce pointer chasing in range queries -> accelerate range queries
- Reduce leaf node split -> accelerate insertion

Entry Compression

- Compress 1-byte entry and pointer into an 8-byte field -> reduce persistent instructions (e.g. *clwb/sfence*)

Selective Metadata Persistence

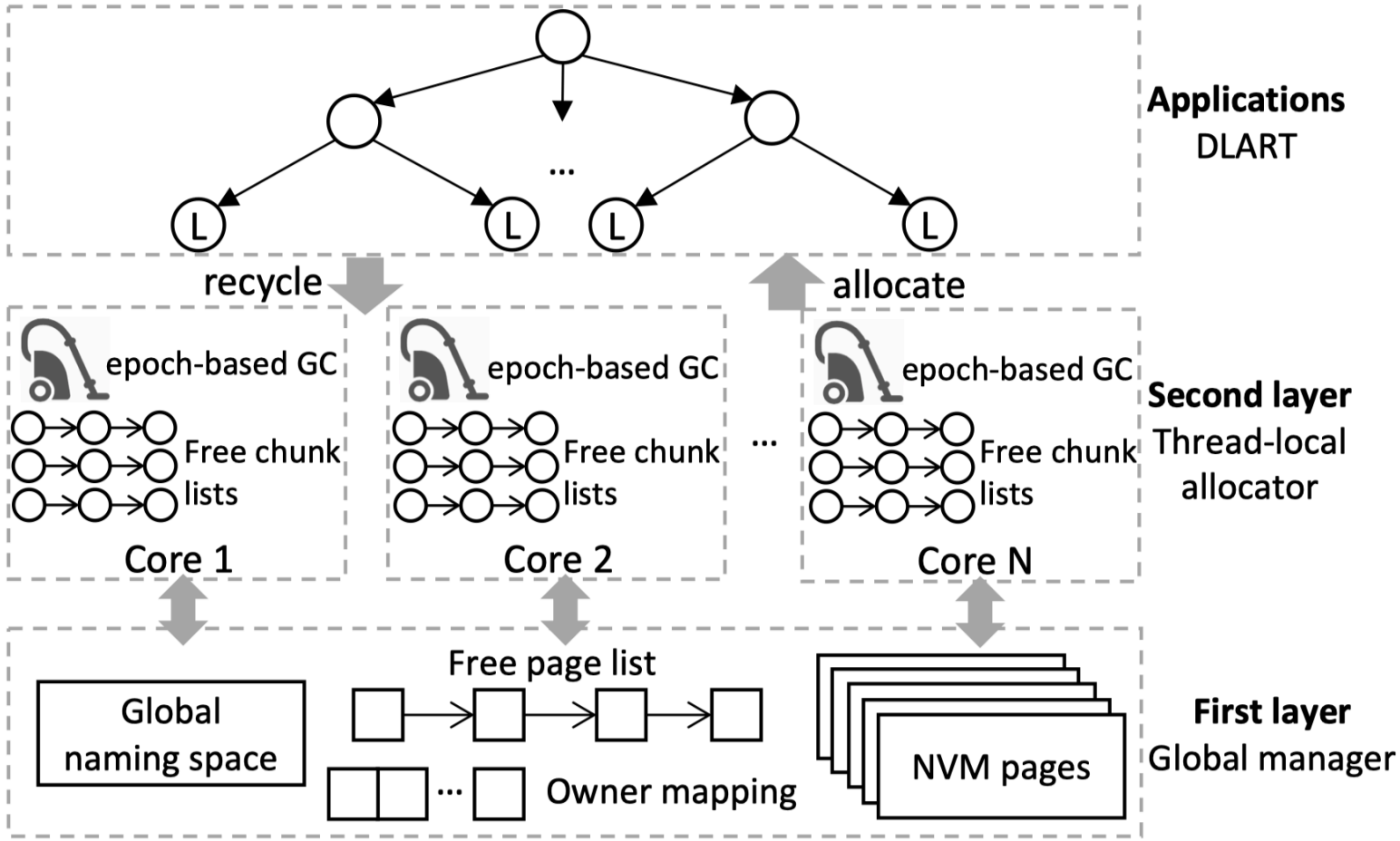
- Make metadata volatile with constant recovery time -> further reduce persistence overhead

Minimally Ordered Split

- Reorder steps of internal node split -> reduce *sfence* instructions

More details in the paper

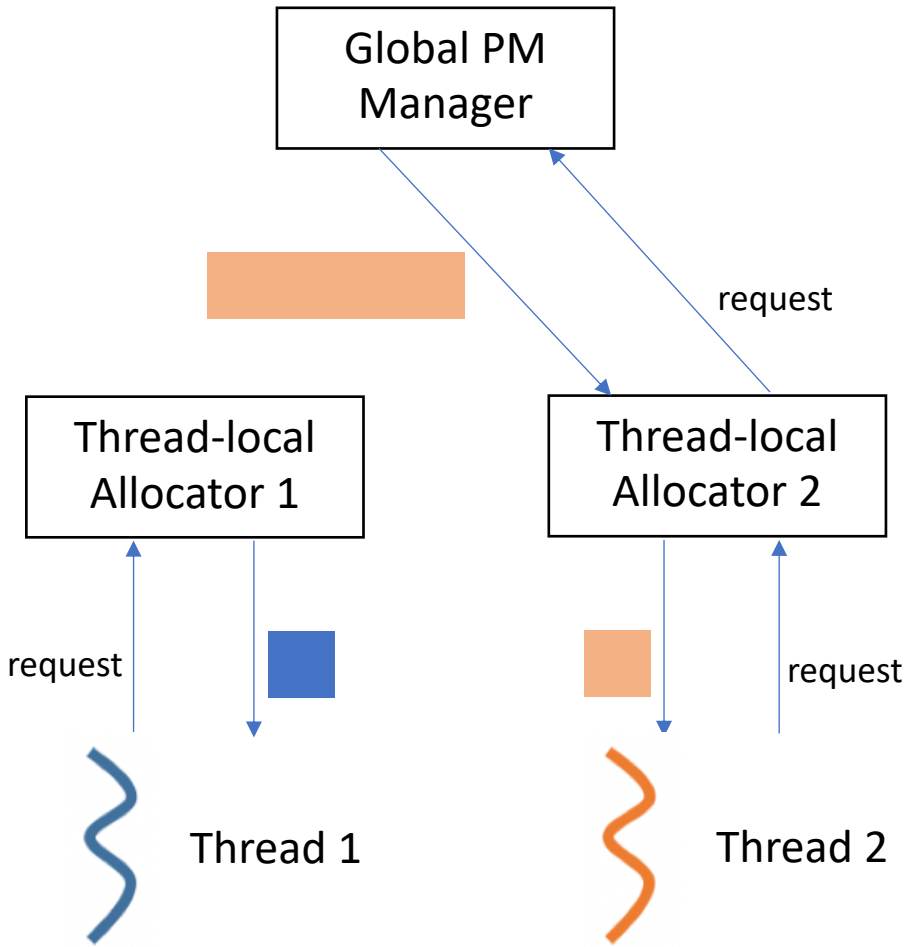
Making ROART Correct



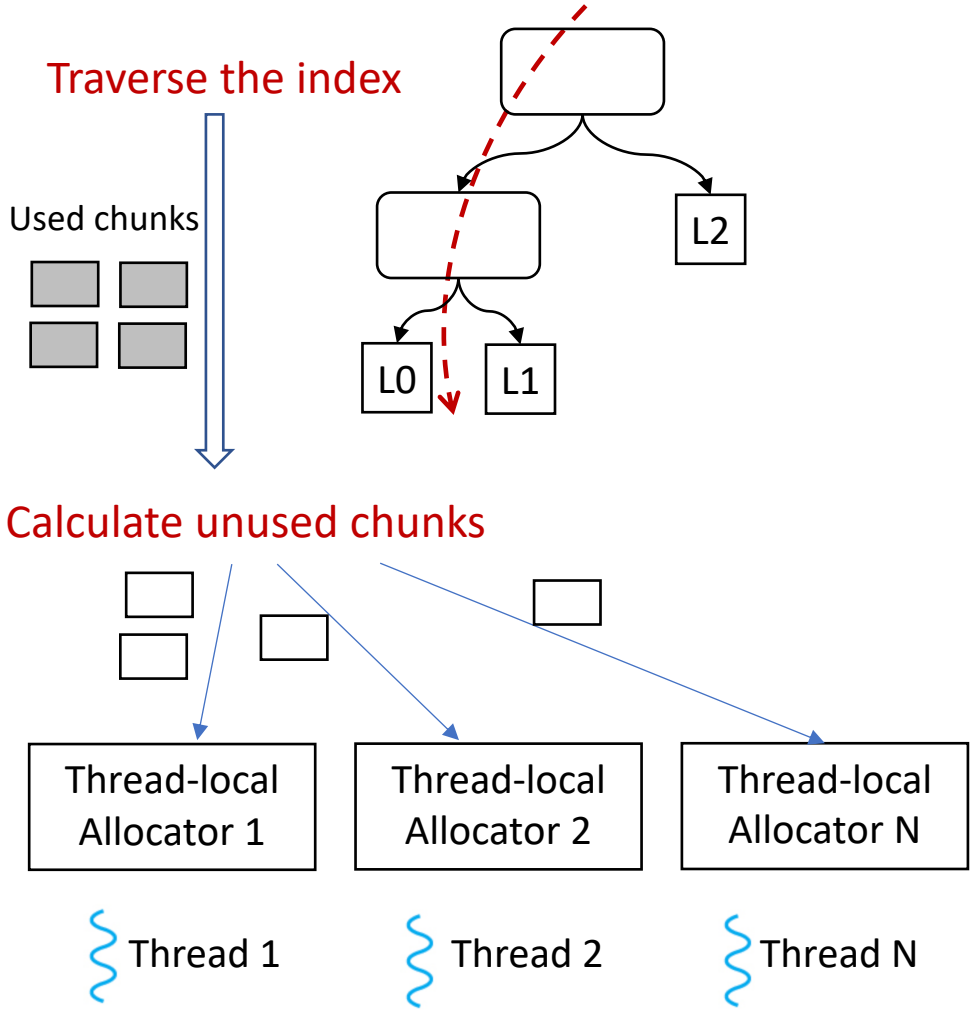
Delayed Check Memory Management (DCMM)
An Allocator based on post-crash GC

DCMM

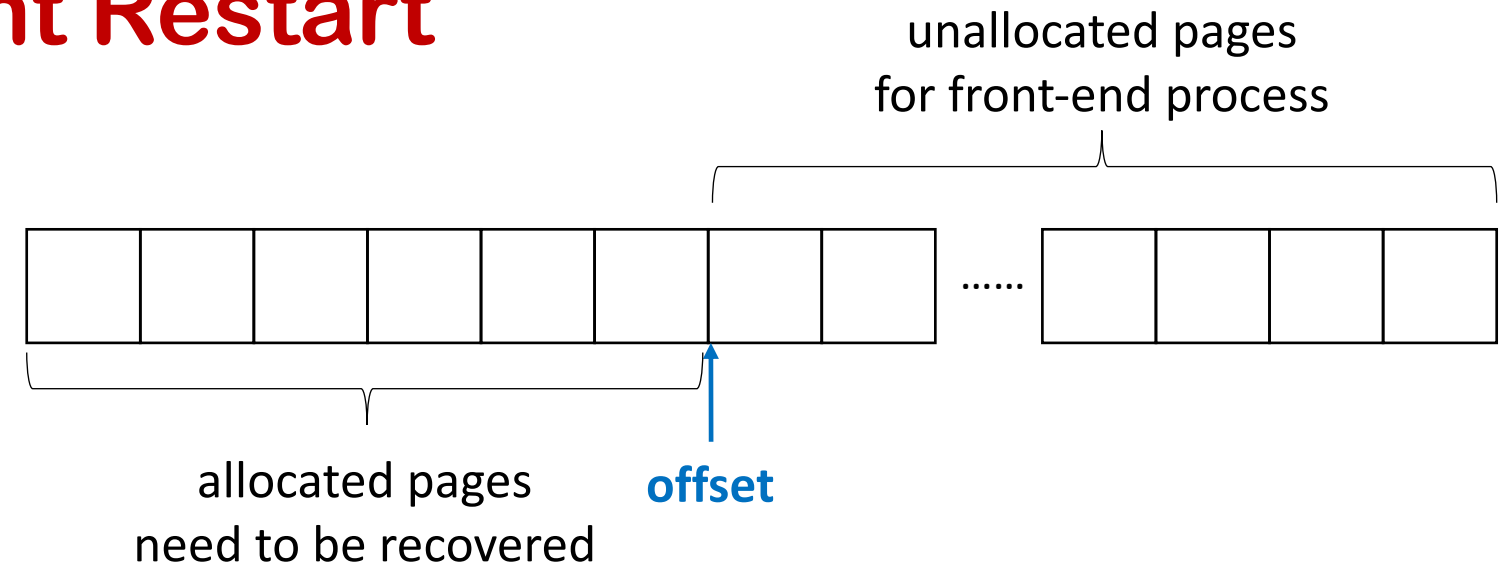
Allocation



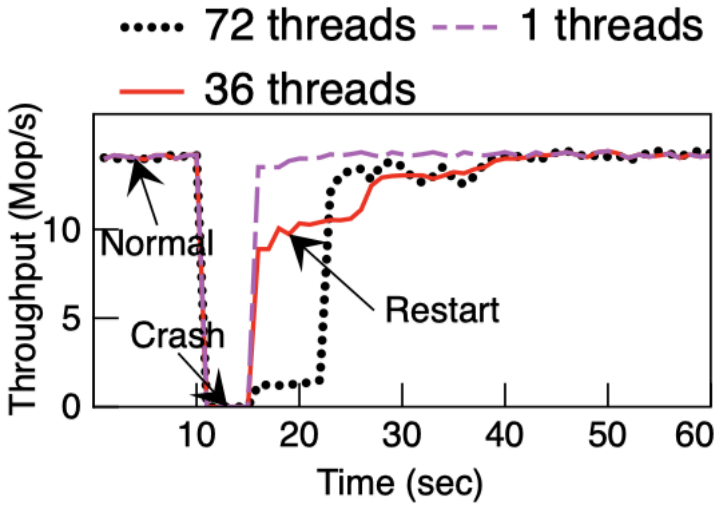
Recovery



Instant Restart



DCMM recovery is parallel with front-end process operations, with constant recovery time



Evaluation Setup

- Dell PowerEdge R740 server
- Four Intel(R) Xeon(R) Gold 5220 processors supporting clwb
- 6×128GB Optane DC PMM per socket
- 32KB L1-cache, 1MB L2-cache, and 25MB L3-cache
- Persistent memory is managed by a DAX file system

Indexes to Compare

- P-ART [SOSP'19]
- PMwCAS-ART (based on PMwCAS [ICDE'18])
- FAST&FAIR [FAST'18]
- Lock-free SkipList [ATC'18]
- BzTree [VLDB'18]

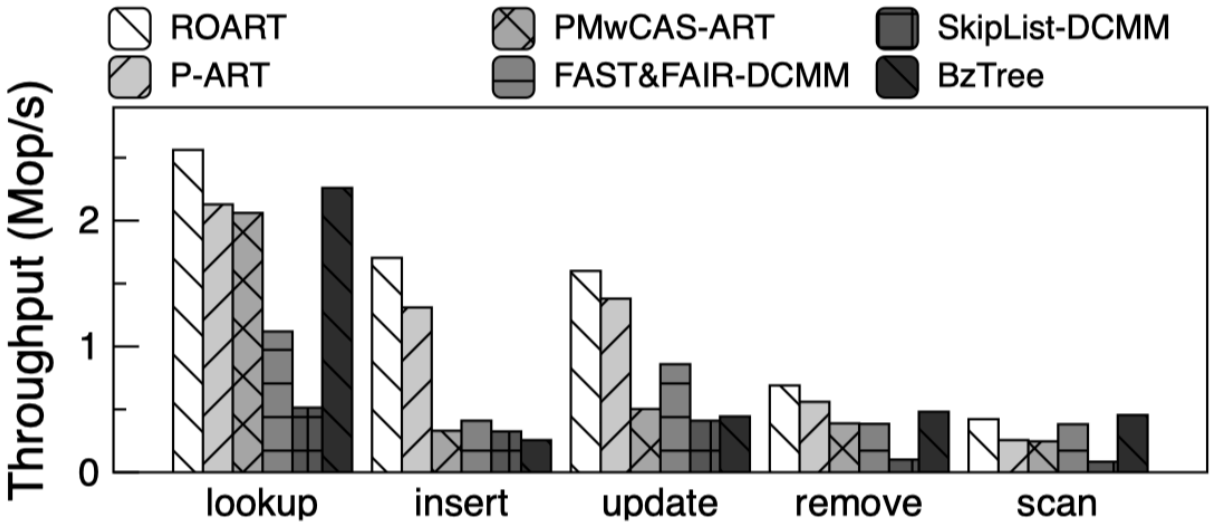
Some modifications

- We modify P-ART, FAST&FAIR and SkipList with DCMM allocator for a fair comparison.
- PMwCAS and BzTree use their own persistent allocators based on PMDK.
- We also implement some missing operations in these indexes.

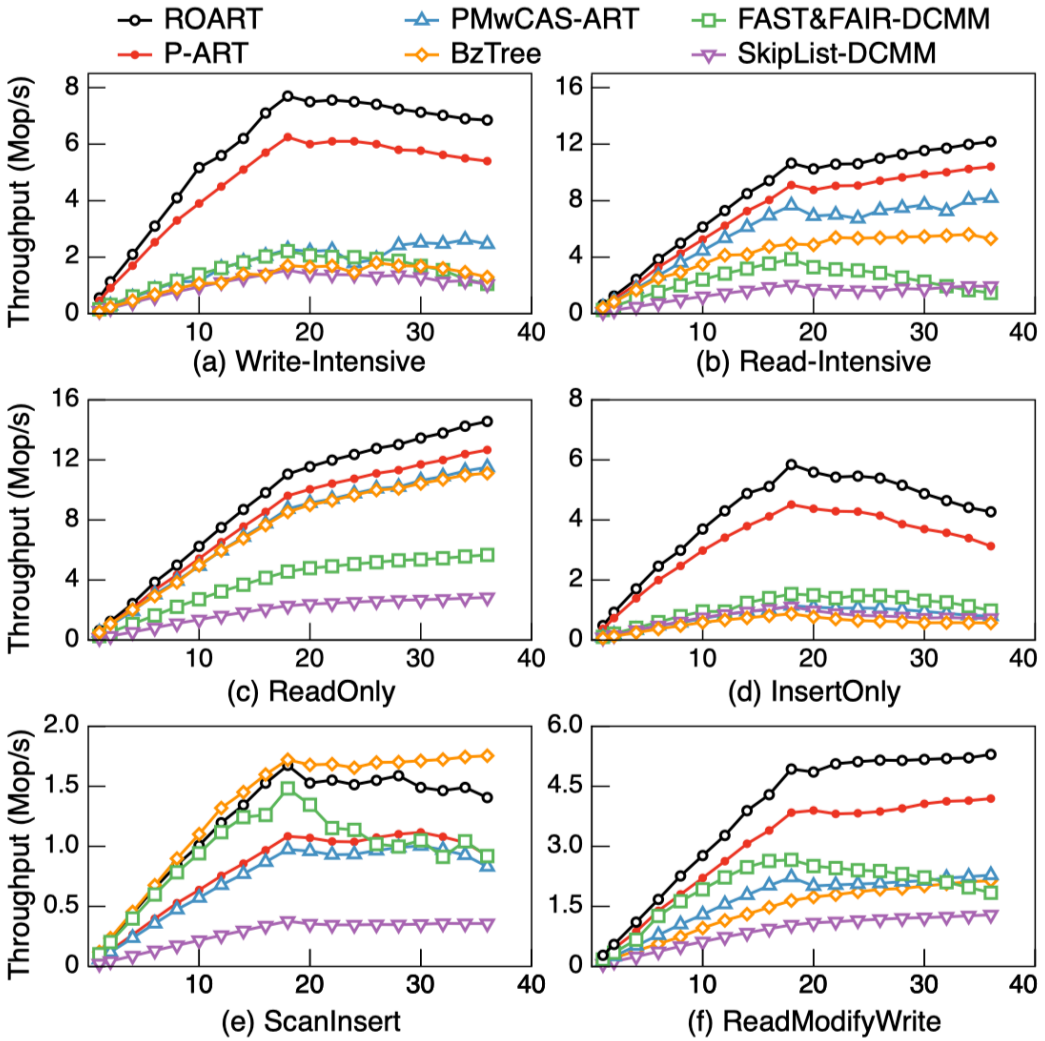
Overall Performance

Workloads

- Keys are randomly generated with sizes between 4 to 128 bytes.
- Values are fixed as 8 bytes.
- Each test warms up using 30 million KVs.
- Each test runs 20 seconds for different workloads and reports the average throughput.

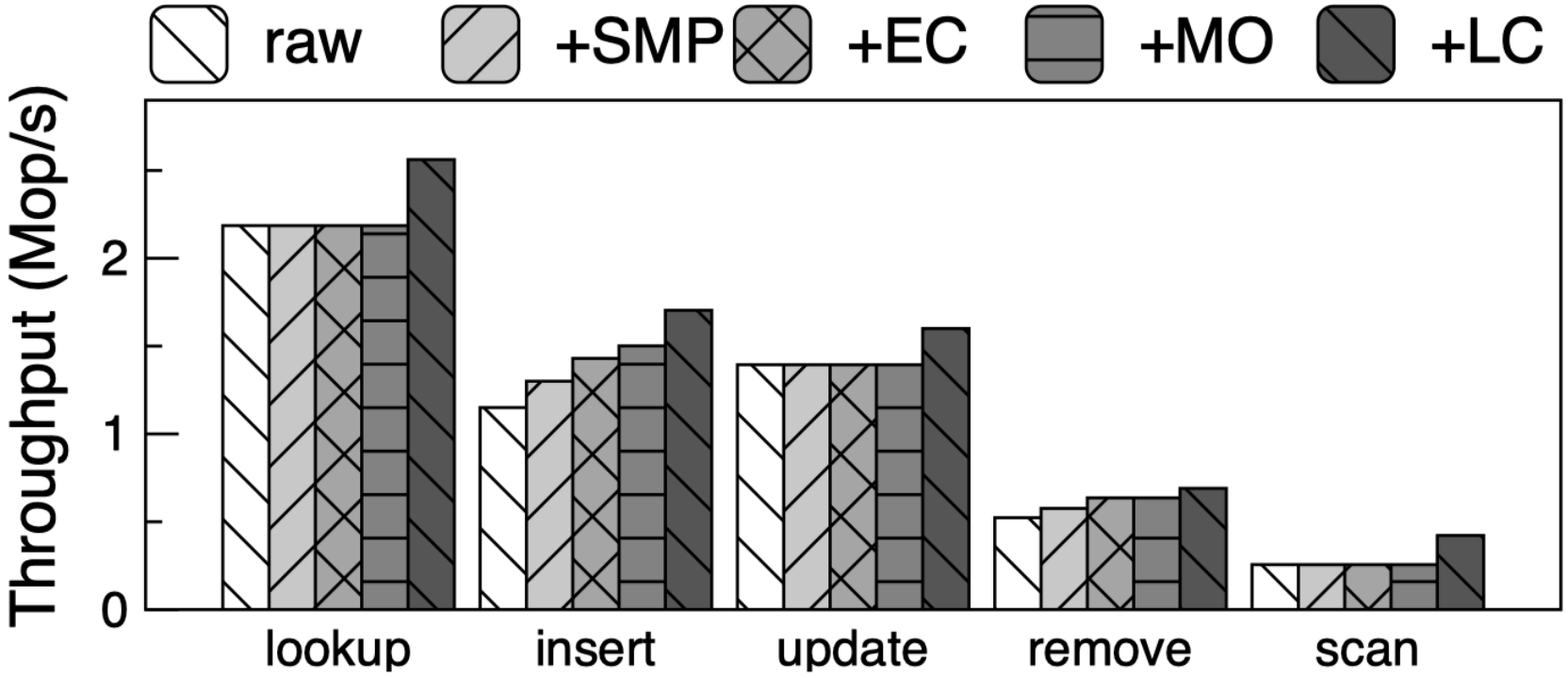


Microbench with 4 threads



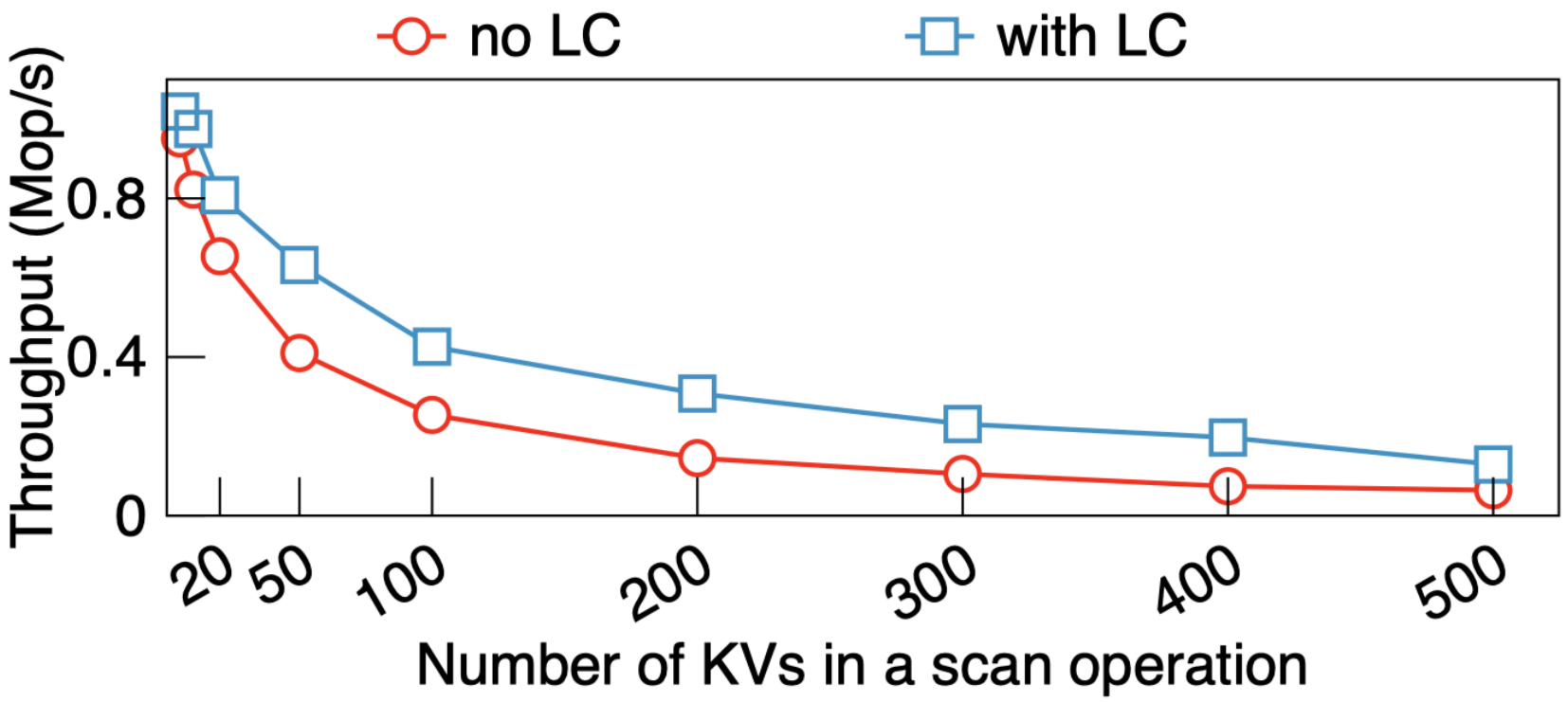
YCSB

Performance Breakdown



Performance improvement brought by each optimization

Range Query Performance



Performance improvement of range queries from Leaf Compaction

Conclusion

- Analyze three practical aspects
  - functionality
  - Performance
  - correctness
- A new persistent index, ROART, taking all these practical aspects into account.
  - Leaf compaction
  - Entry compression
  - Selective metadata persistence
  - Minimally ordered split
  - DCMM with instant restart
- Better performance than the state-of-the-art persistent indexes

Thanks
Q&A

Anomaly Resolution

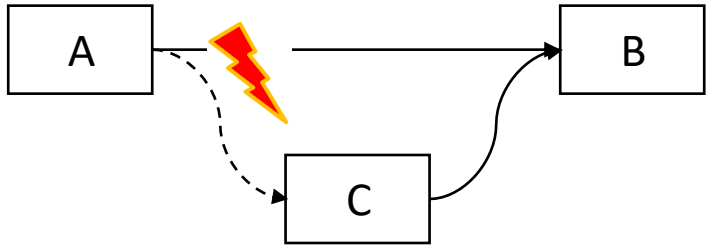
Steps of insertion in a lock-free linked list

- Create a new node, set its next pointer, and persist the new node
- Update the next pointer of the predecessor to the new node
- Persist the next pointer of the predecessor

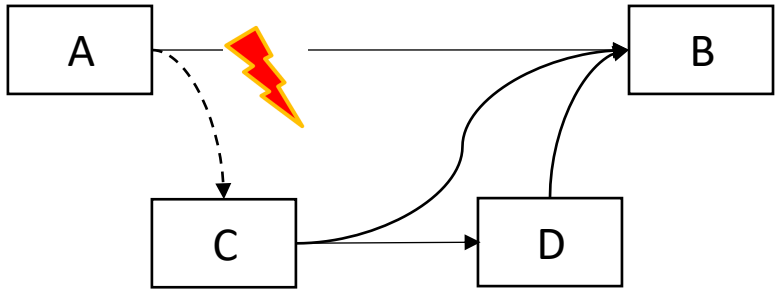
States of the predecessor's next pointer during three steps

next pointer	Step (i)	Step (ii)	Step (iii)
volatile state	old	new	new
persistent state	old	old	new

Dirty Read



Lost Update

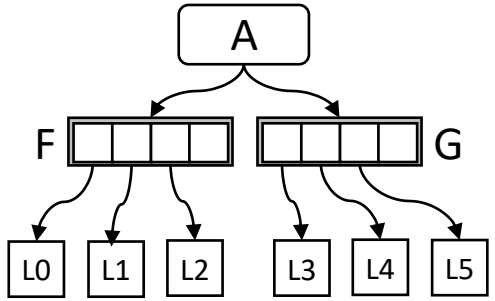


Some existing ways can be used to fix anomalies

- Link-and-persist
- PMwCAS
- non-temporal instructions

Disadvantages of Leaf Compaction

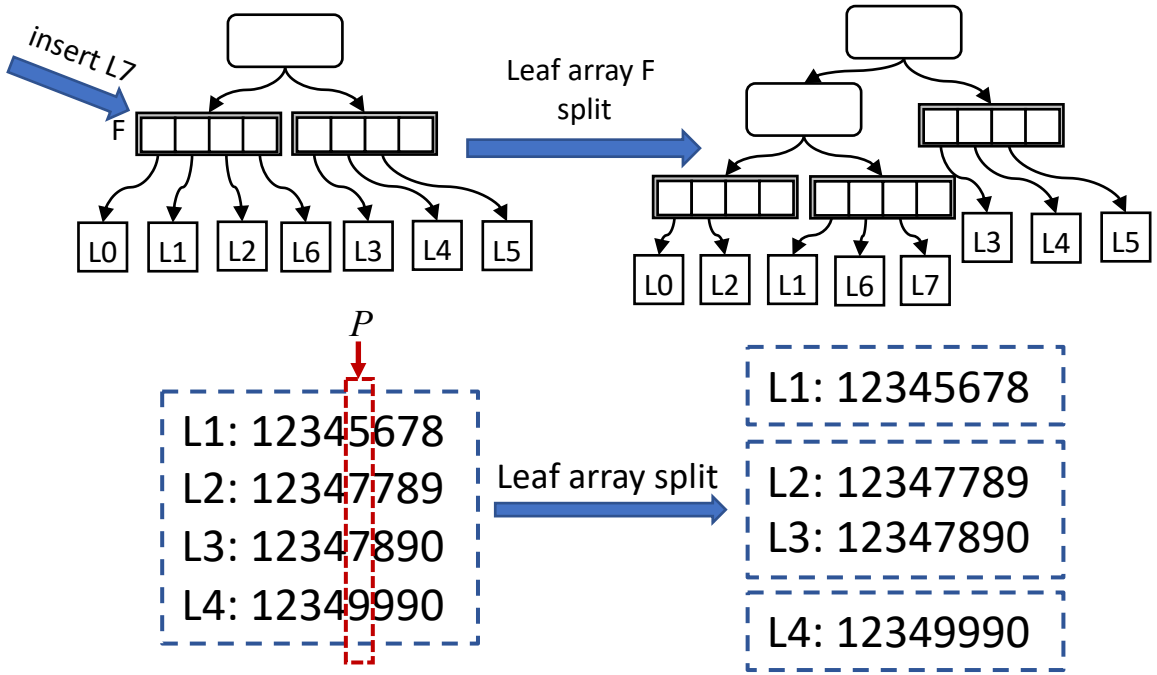
string comparison in leaf array



In ART, search will reach the corresponding leaf.
But in ROART, search only reaches a leaf array,
and string comparison is necessary

16-bit fingerprints with pointers to reduce comparison

heavy leaf array split



Leaf array split is rare

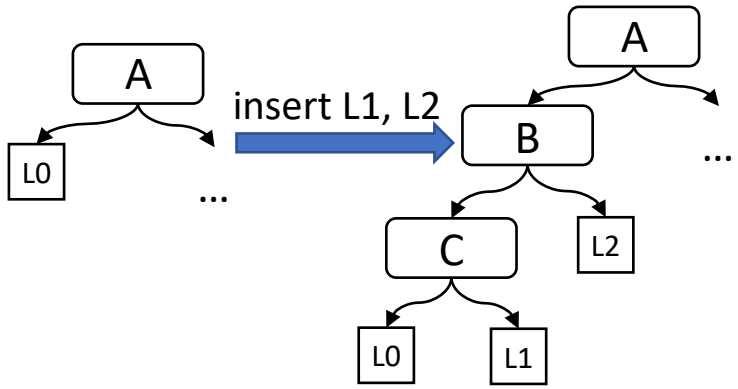
Advantages and Disadvantages

Advantages

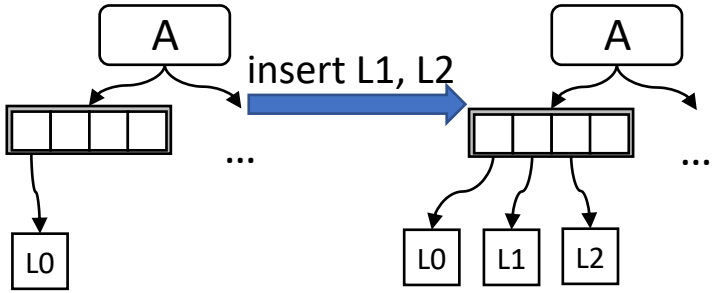
Reduce height of tree → accelerate traversal

Reduce pointer chasing in range queries → accelerate range queries

Reduce leaf node split → accelerate insertion



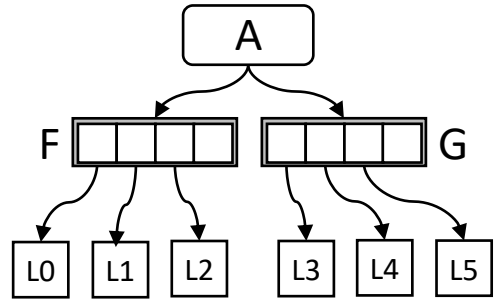
Insertion in ART



Insertion in ROART

Disadvantages

string comparison in leaf array

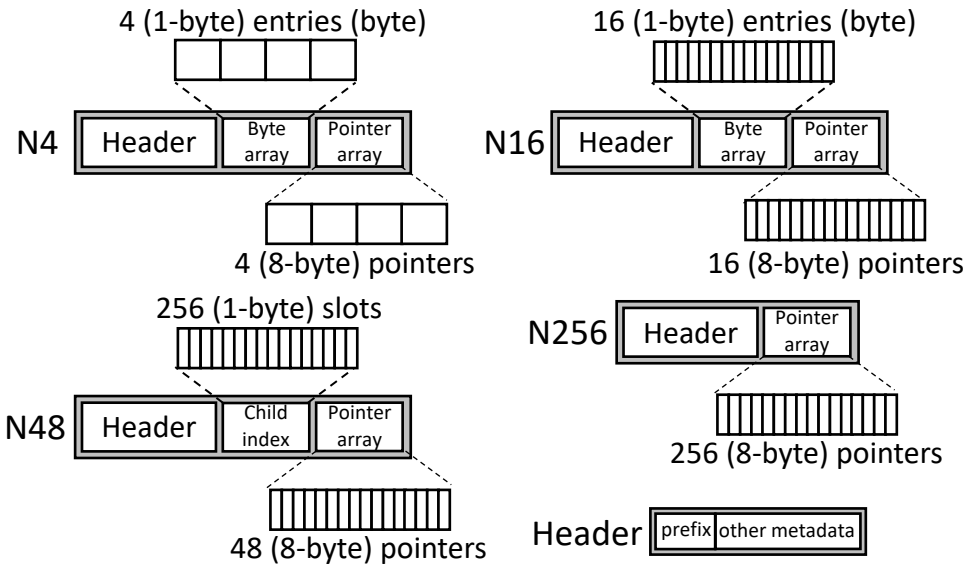


In ART, search will reach the corresponding leaf.
But in ROART, search only reaches a leaf array, and string comparison is necessary

Solution: 16-bit fingerprints with pointers to reduce comparison

Persistency Optimization

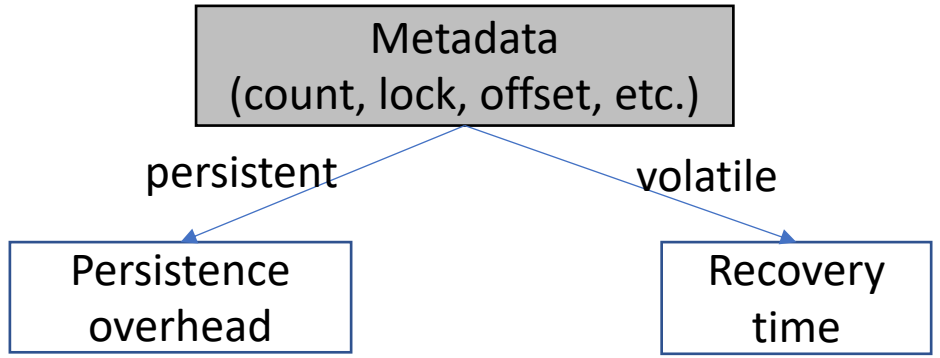
Entry Compression



ART Node Type

N4, N16, N48 all have Byte array (1-byte per entry) and Pointer array (8-byte per entry)
Pointers in current architecture only use 6-byte, so compression can be adopted.

Selective Metadata Persistence



Persistence overhead and *recovery time* is a trade-off.

Selective Metadata Persistence (SMP) minimizes the two overhead at the same time.

Global generation number is used to check whether metadata of the node needs to be restored when accessing a node. Such a method has no effect on performance.