USENIX ATC '22

Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory

Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang*, Jiwu Shu

Tsinghua University

**Huawei Technologies Co., Ltd*

Persistent Memory (PM)

PM brings opportunities and challenges to storage systems !

Benefits

- ❖ Persistent storage
- ❖ Byte-addressability

The figure of Optane DIMM is from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Persistent Memory (PM)

PM brings opportunities and challenges to storage systems !

Benefits

- ❖ Persistent storage
- ❖ Byte-addressability

Idiosyncrasies (compare with DRAM)

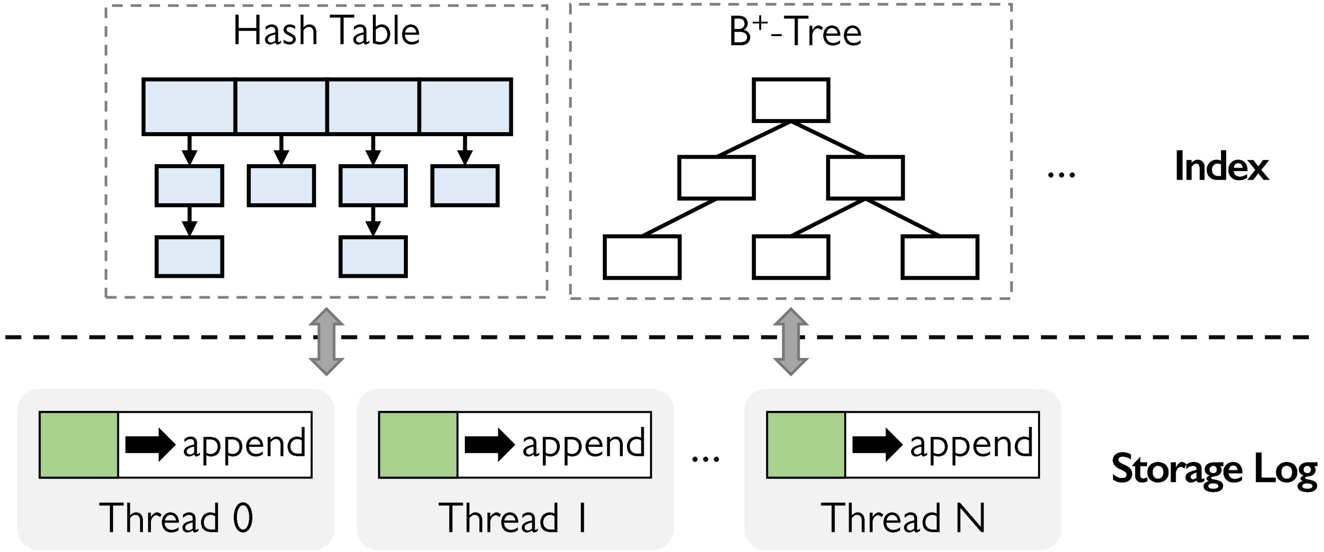
- ❖ High access latency (~300 ns)
- ❖ Limited write bandwidth (2.2 GB/s)
- ❖ Access granularity (256 bytes)

The figure of Optane DIMM is from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Log-structured KV Systems

Advantages

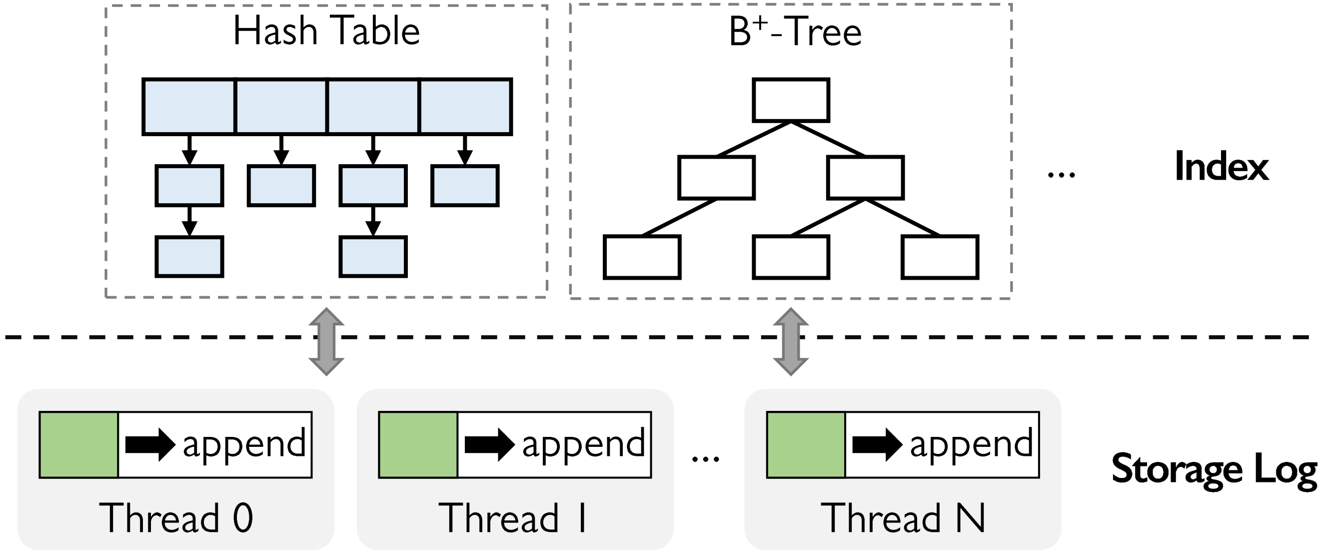
- ❖ Fast allocation
- ❖ High capacity utilization
- ❖ Small write amplification
- ❖ Easy failure recovery

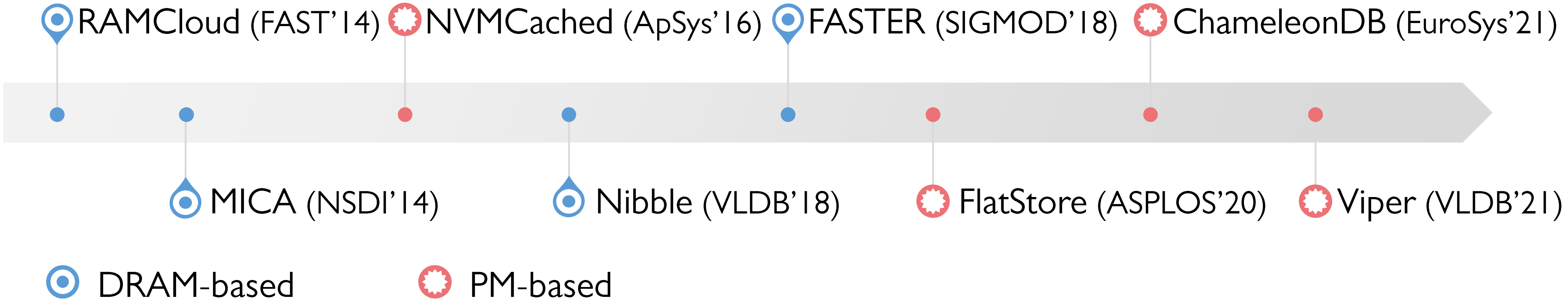


Log-structured KV Systems

Advantages

- ❖ Fast allocation
- ❖ High capacity utilization
- ❖ Small write amplification
- ❖ Easy failure recovery





Garbage Collection Overheads (1)

- **Garbage collection has huge overhead at high capacity utilizations**
  - RAMCloud (FAST'14) drops by 20-50%
  - Nibble (VLDB'18) drops by up to 75%

Garbage Collection Overheads (1)

- **Garbage collection has huge overhead at high capacity utilizations**
  - RAMCloud (FAST'14) drops by 20-50%
  - Nibble (VLDB'18) drops by up to 75%
- **Capacity utilization is important**
  - Google: keep disks full and busy to minimize storage TCO (PDSW-DISCS'17)
  - Facebook: space utilization was far more important than write amplification (FAST'21)

Garbage Collection Overheads (2)

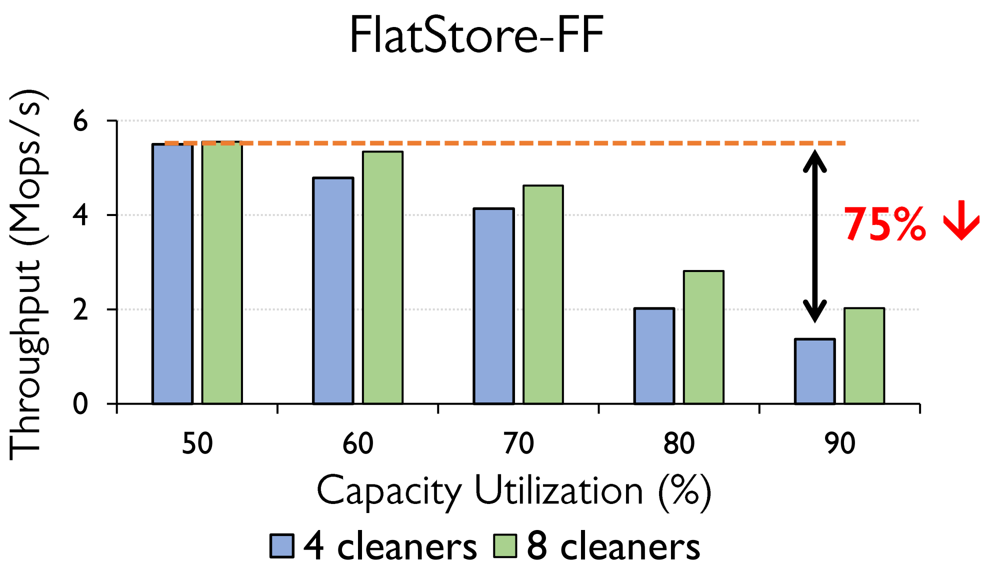
PM’s idiosyncrasies exacerbate the bottleneck of compaction.

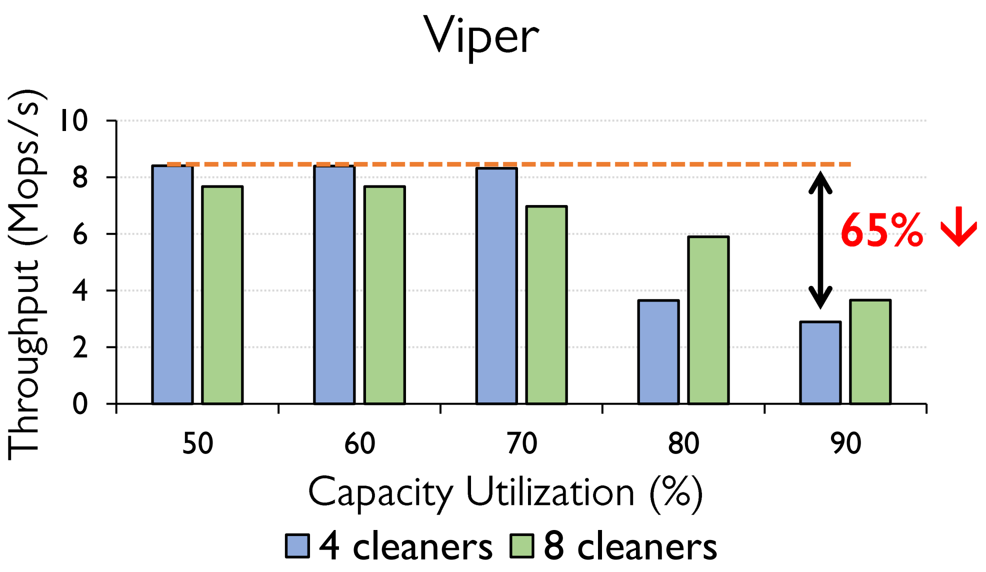
Garbage Collection Overheads (2)

PM's idiosyncrasies exacerbate the bottleneck of compaction.

Experiments on FlatStore (ASPLOS'20) & Viper (VLDB'21)

YCSB-A (50% Get, 50% Put) , 12 service threads, 4/8 cleaners



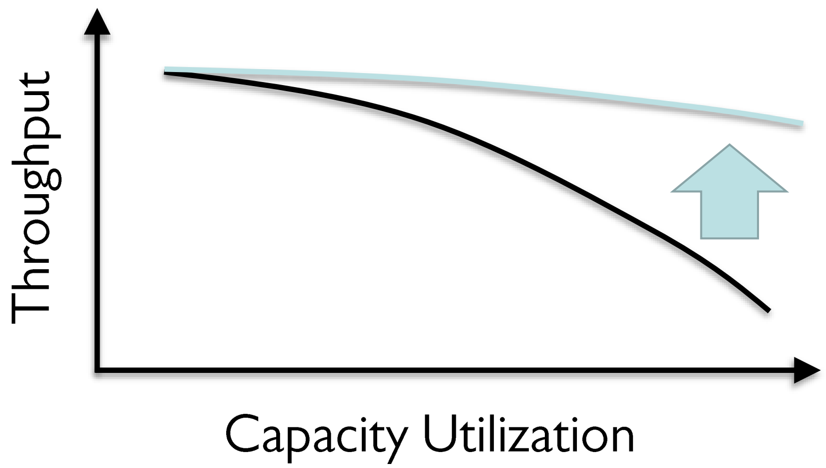


Motivation

Existing compaction approaches are unaware of PM's characteristics.

Motivation

Existing compaction approaches are unaware of PM's characteristics.



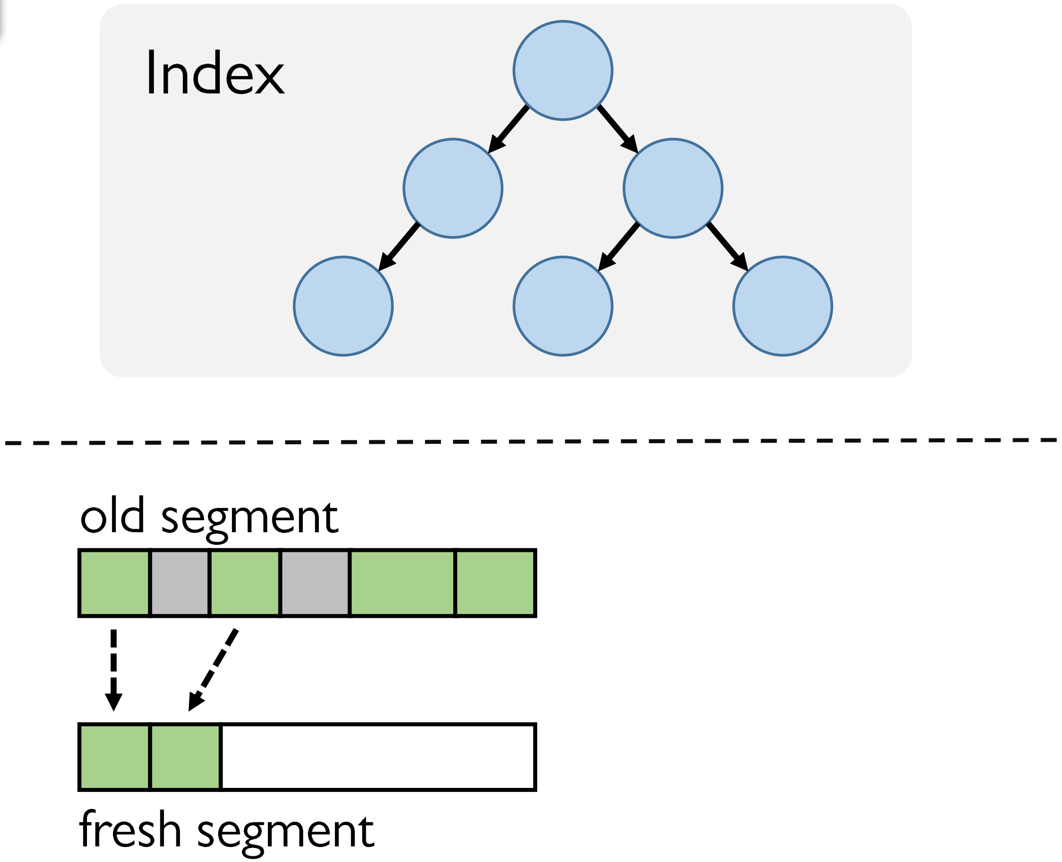
Our goal: Improving the compaction efficiency and log-structured KV store's performance at high capacity utilizations

Outline

- ❖ Background & Motivation
- ❖ Pacman – A PM-aware Compaction Approach for Log-structured KVS
- ❖ Results
- ❖ Conclusion

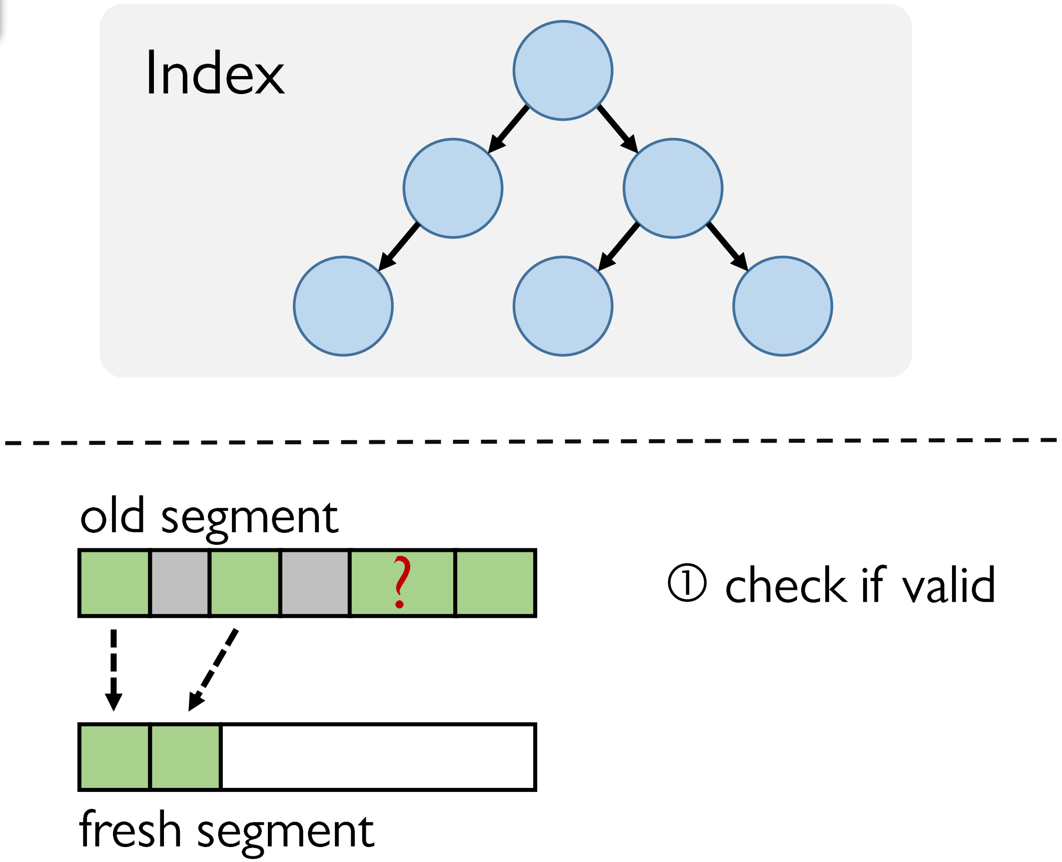
Problem (1) - High-latency Index Traversal

Background Compaction



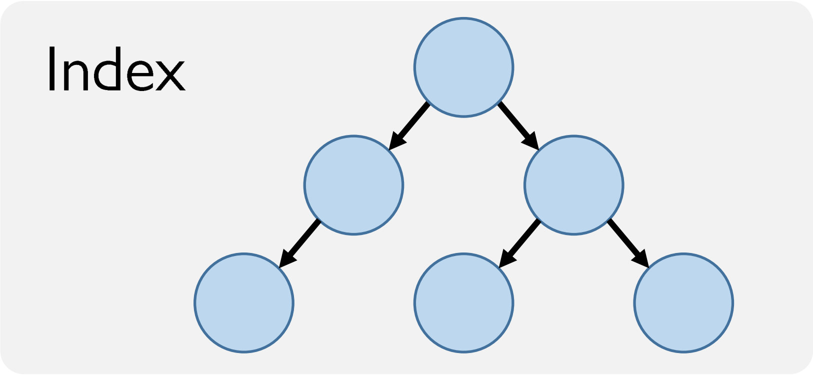
Problem (1) - High-latency Index Traversal

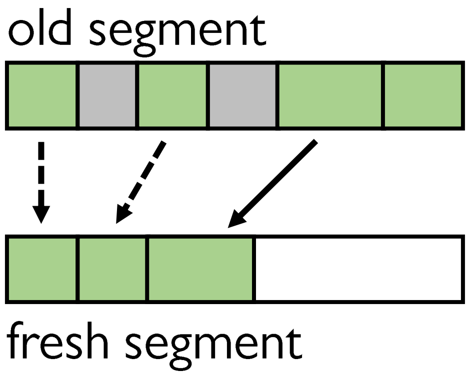
Background Compaction



Problem (1) - High-latency Index Traversal

Background Compaction



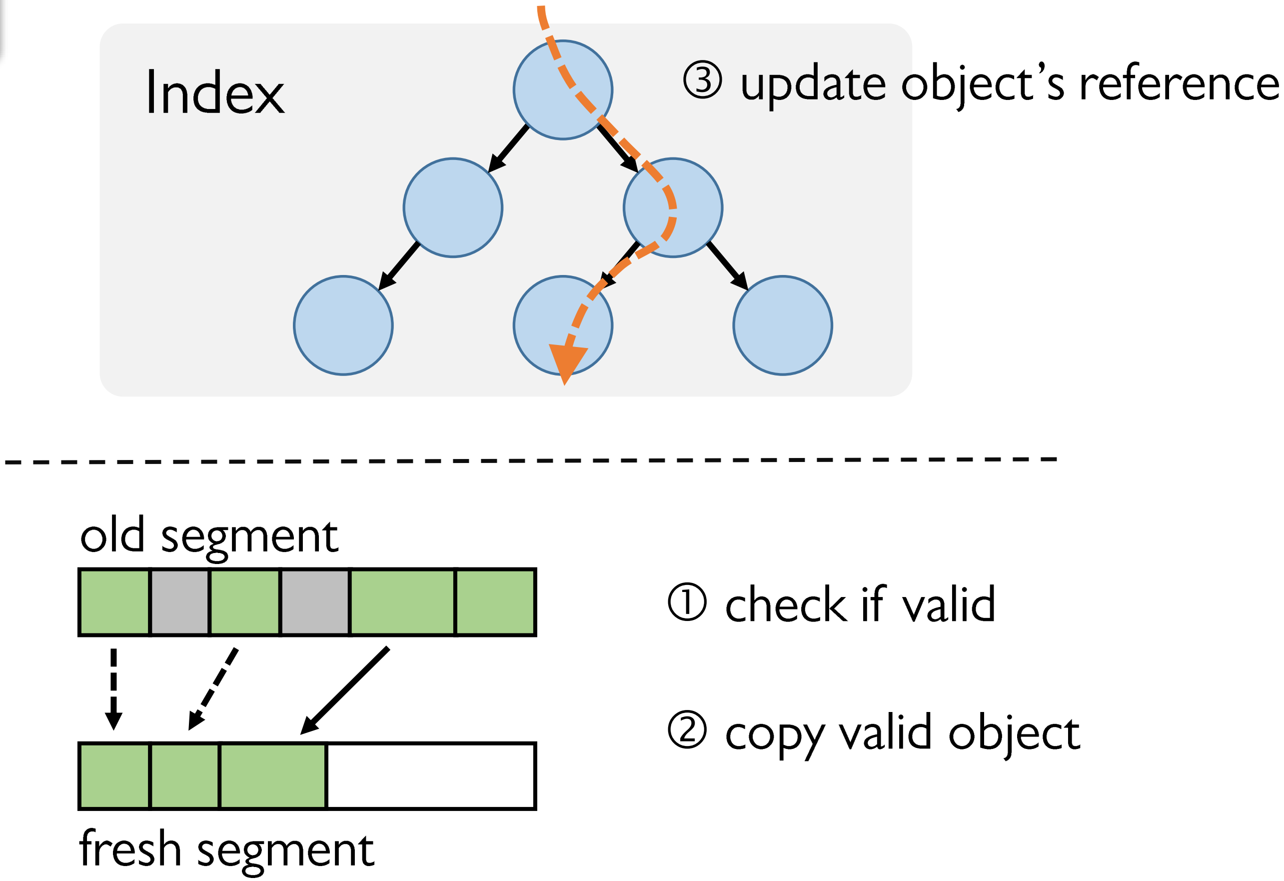


① check if valid

② copy valid object

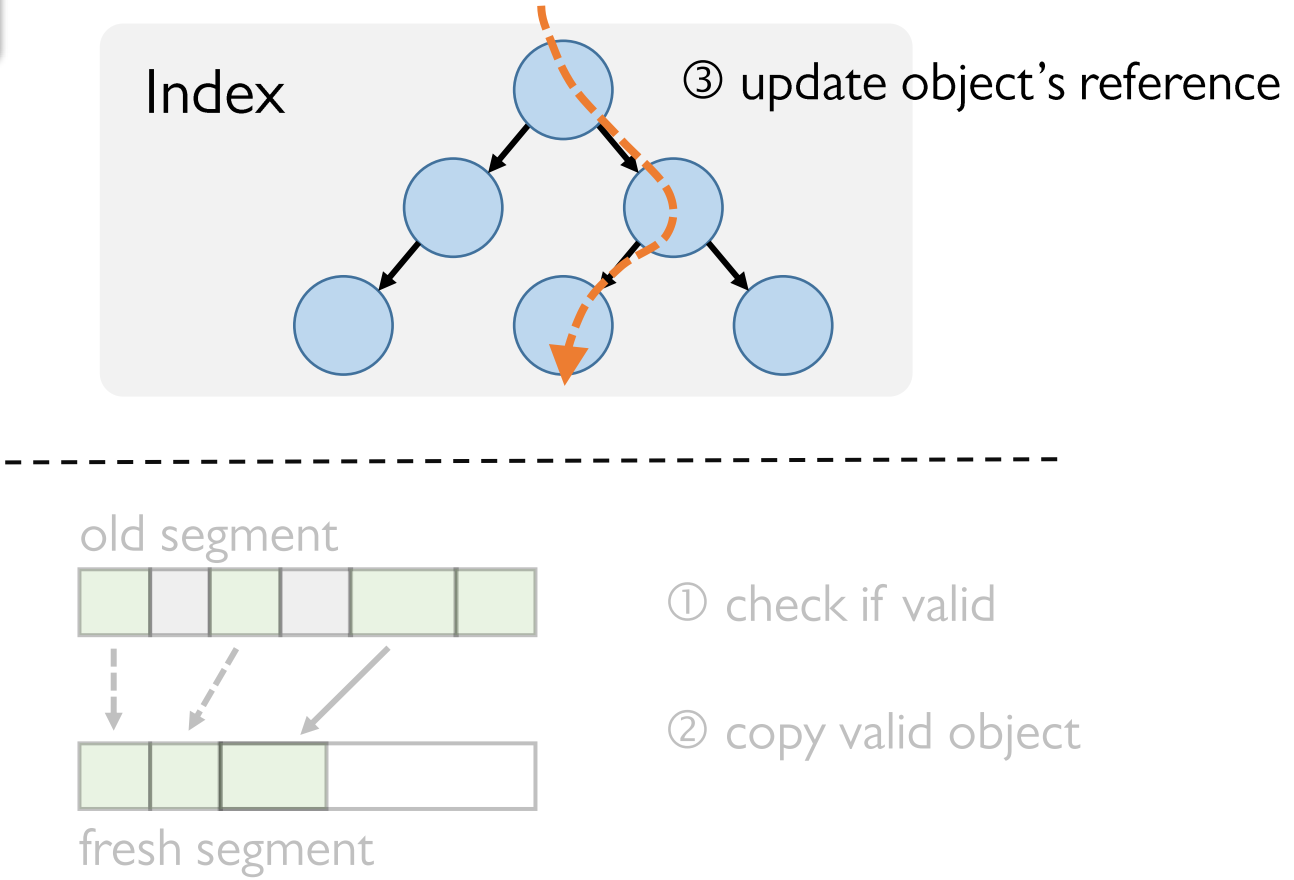
Problem (1) - High-latency Index Traversal

Background Compaction



Problem (1) - High-latency Index Traversal

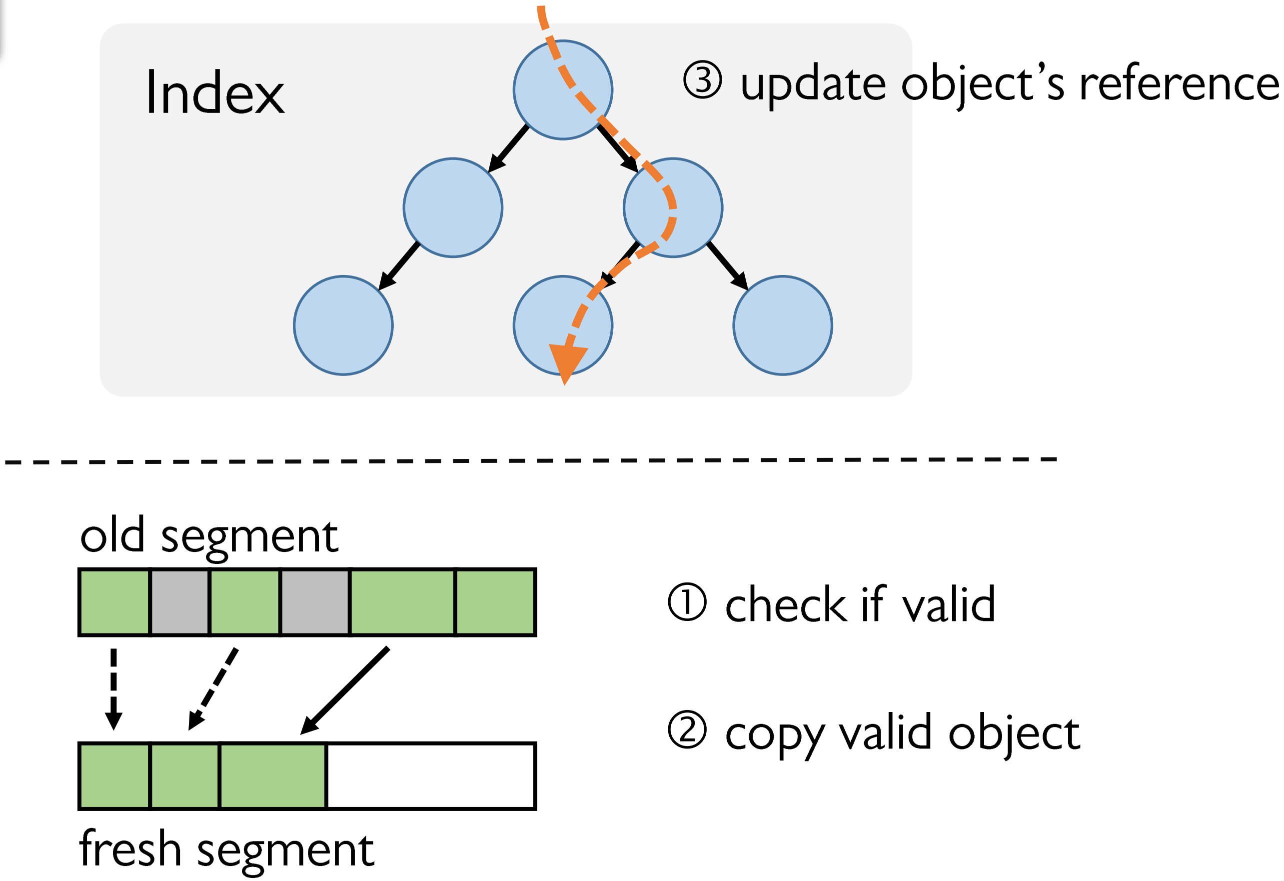
Background Compaction



Problem 1: Traversing the index ↔ high access latency

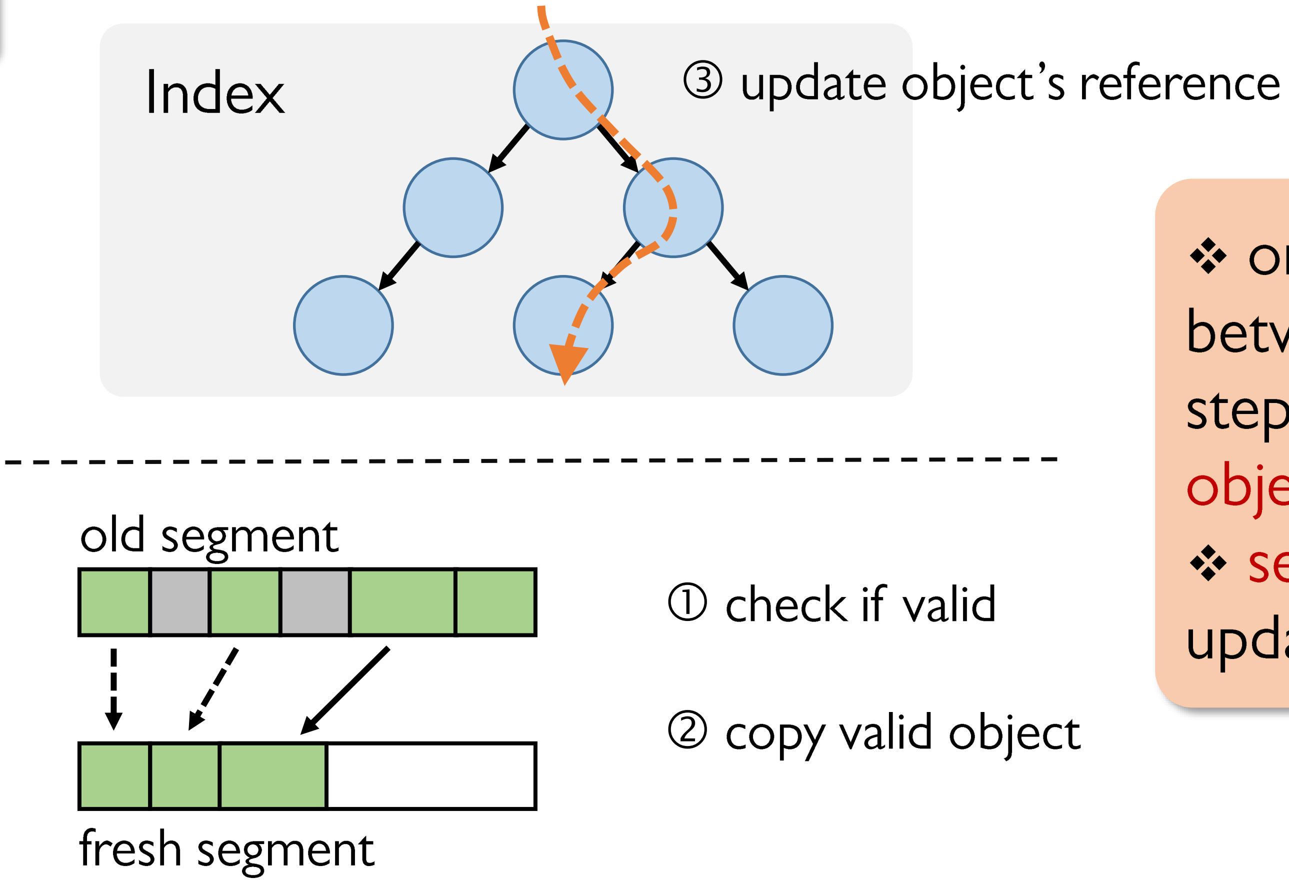
Problem (2) – Extra persistence work

Background Compaction



Problem (2) – Extra persistence work

Background Compaction

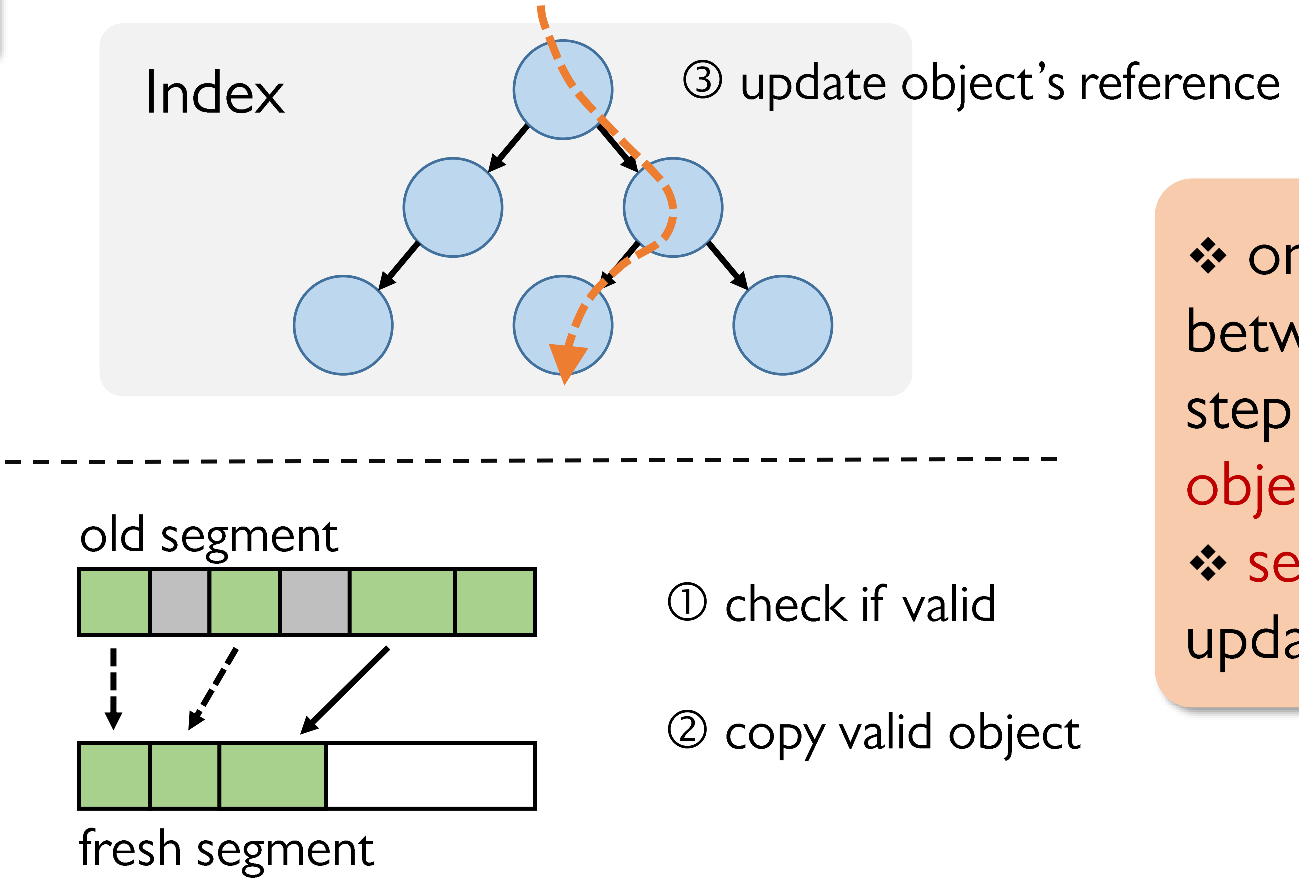


❖ one flush and fence between step ② and step ③ for each valid object

❖ serial flush on each updated reference

Problem (2) – Extra persistence work

Background Compaction

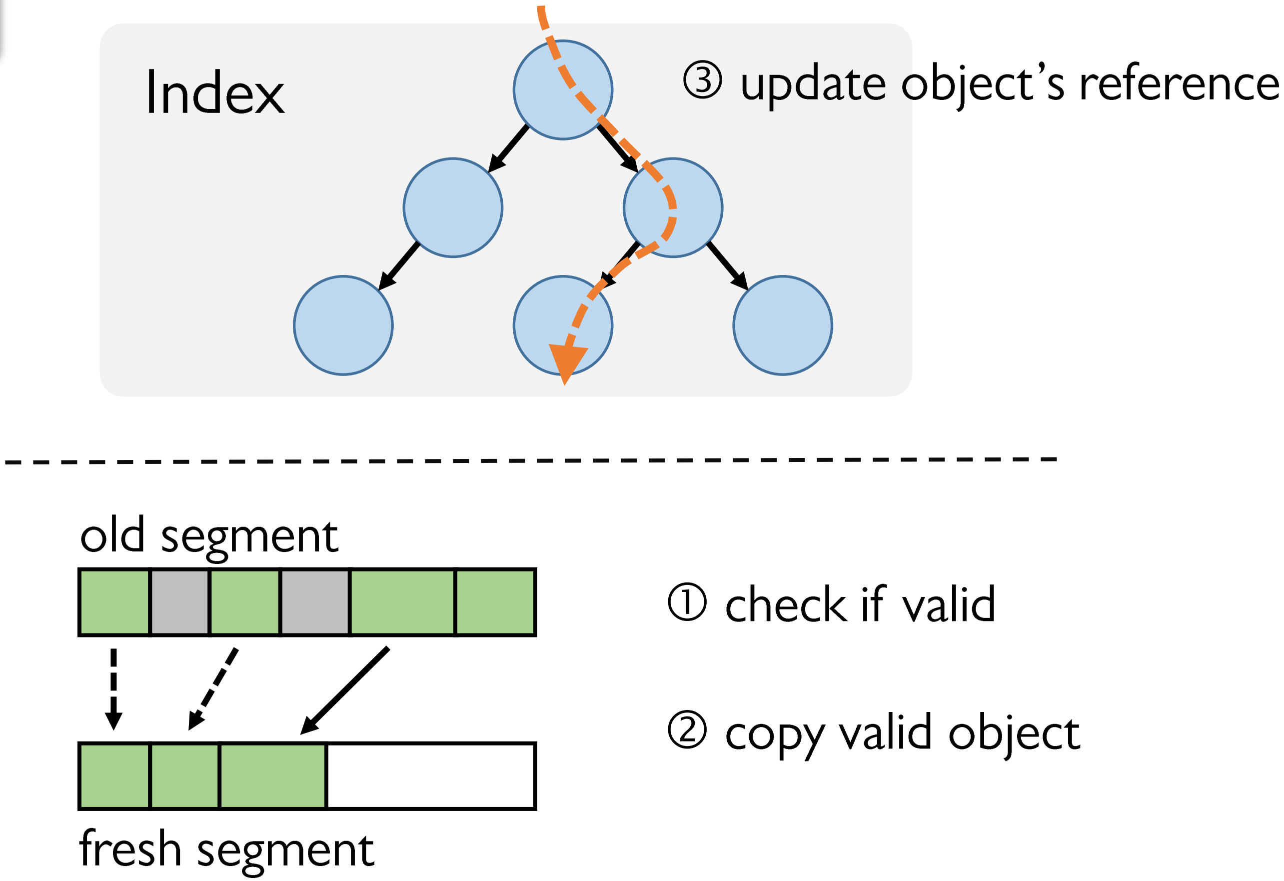


❖ one flush and fence between step ② and step ③ for each valid object
❖ serial flush on each updated reference

Problem 2: Extra persistence work ↔ expensive flush/fence instructions

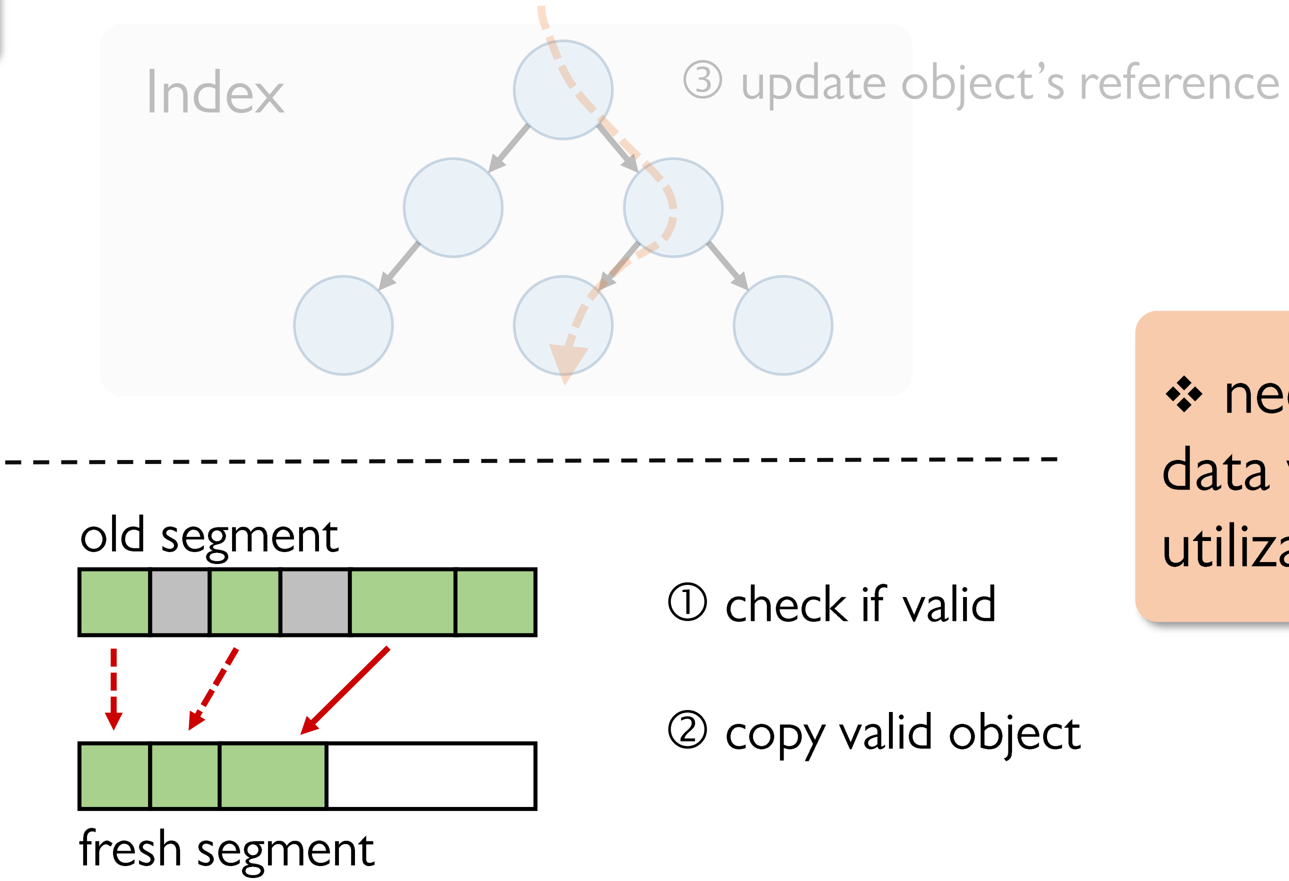
Problem (3) – A large amount of data copying

Background Compaction



Problem (3) – A large amount of data copying

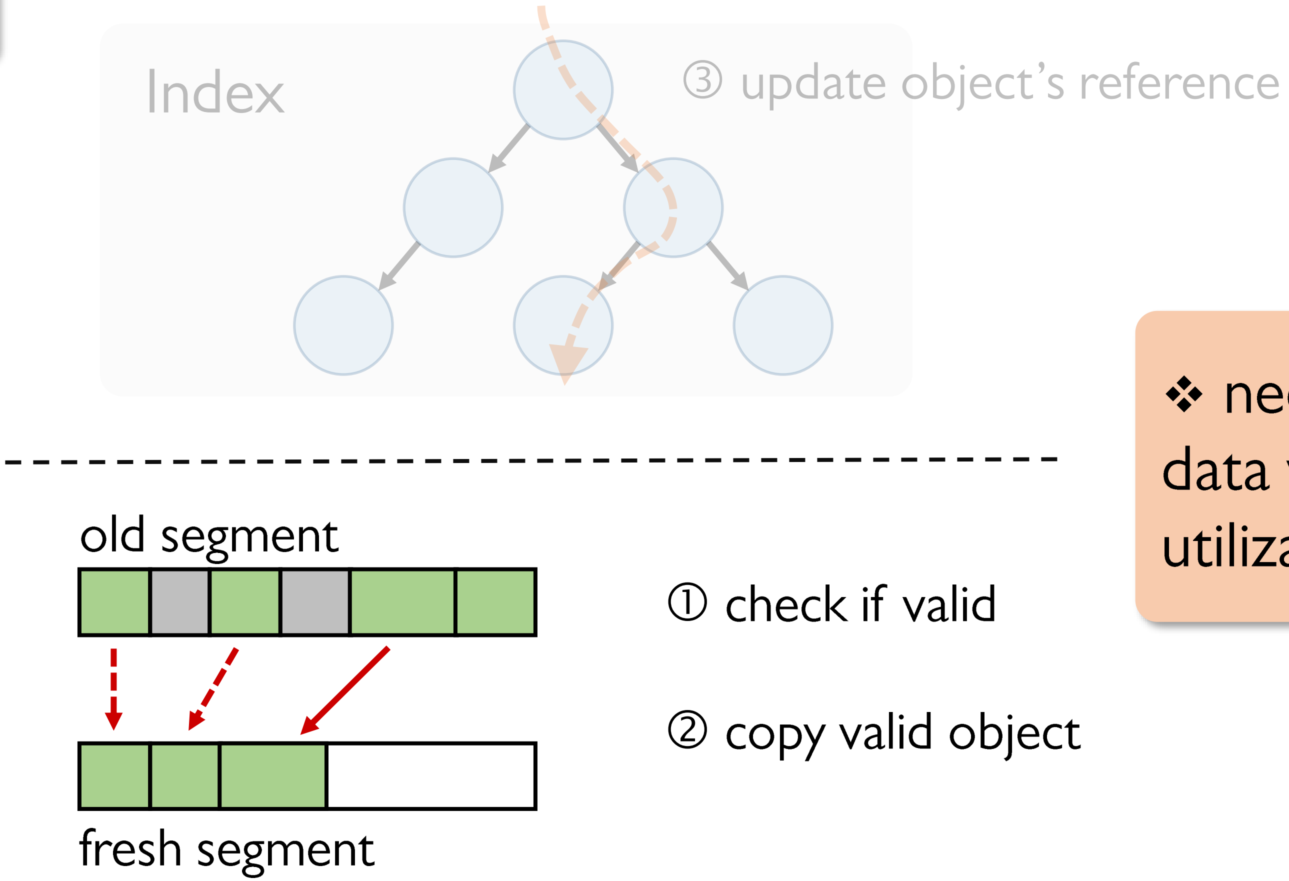
Background Compaction



❖ need to copy more data when the capacity utilization is high

Problem (3) – A large amount of data copying

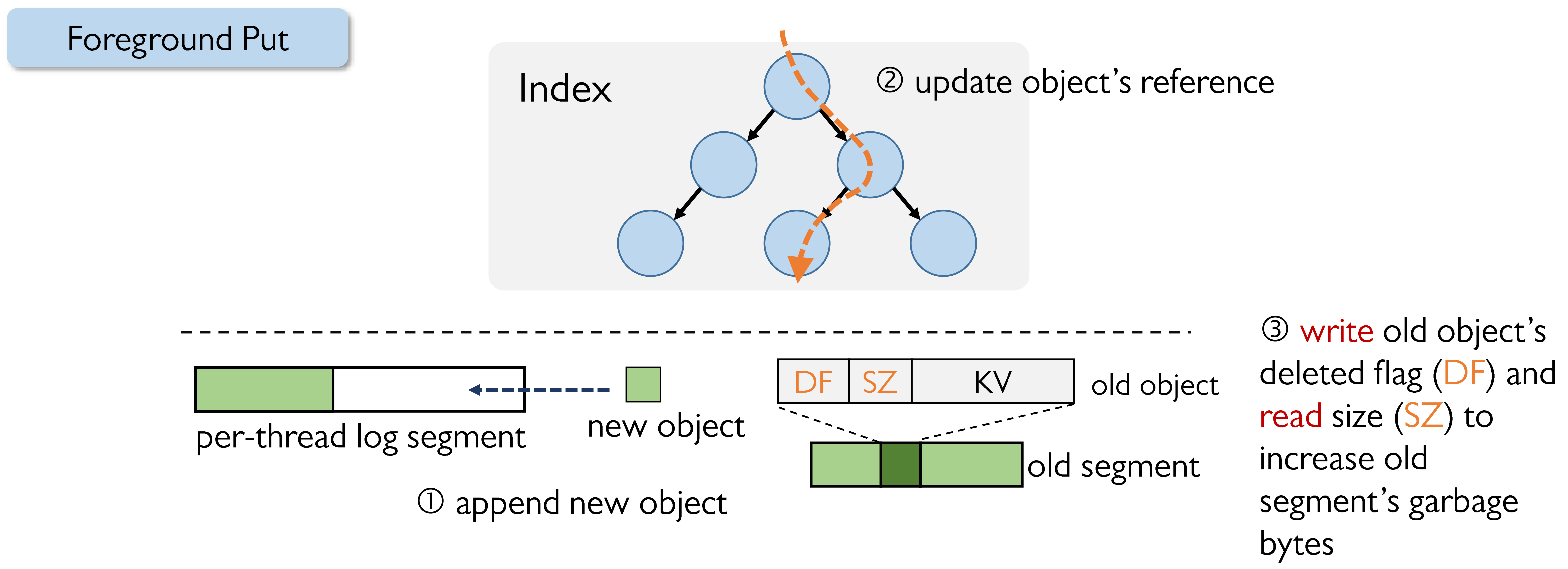
Background Compaction



❖ need to copy more data when the capacity utilization is high

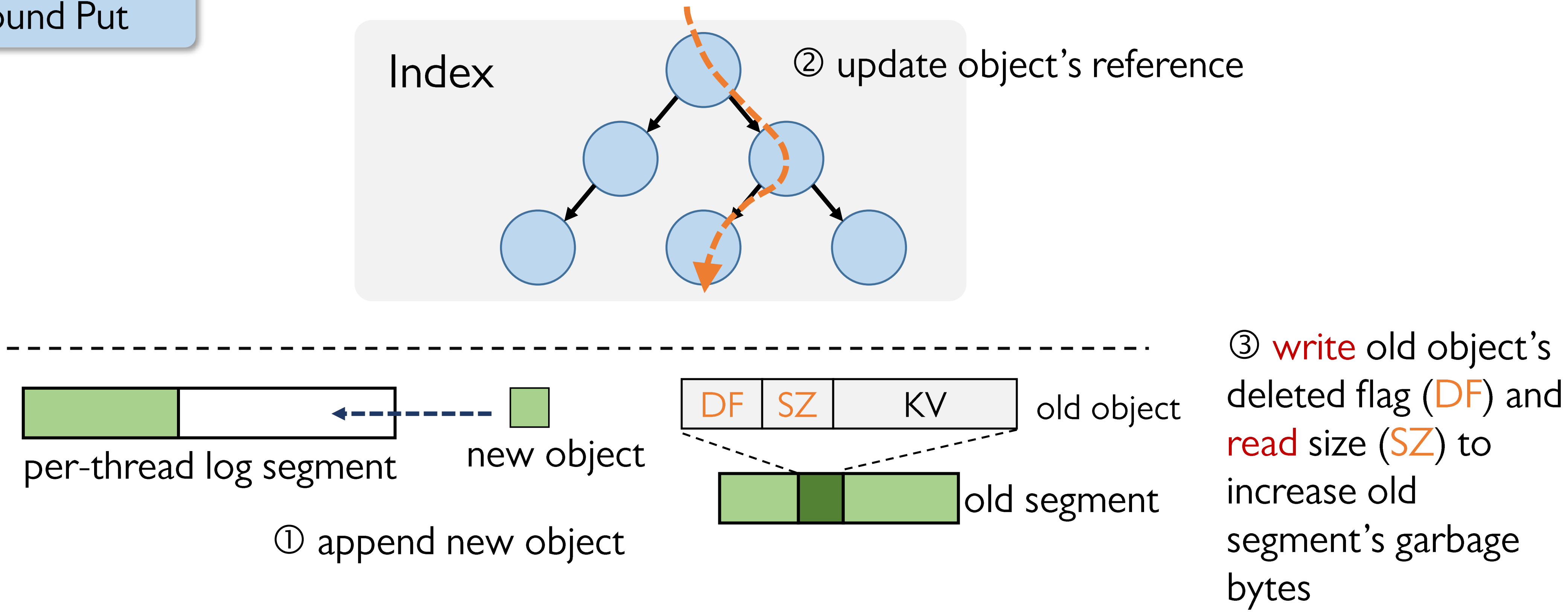
Problem 3: A large amount of data copying on PM ↔ limited write bandwidth

Problem (4) – Excessive small random accesses



Problem (4) – Excessive small random accesses

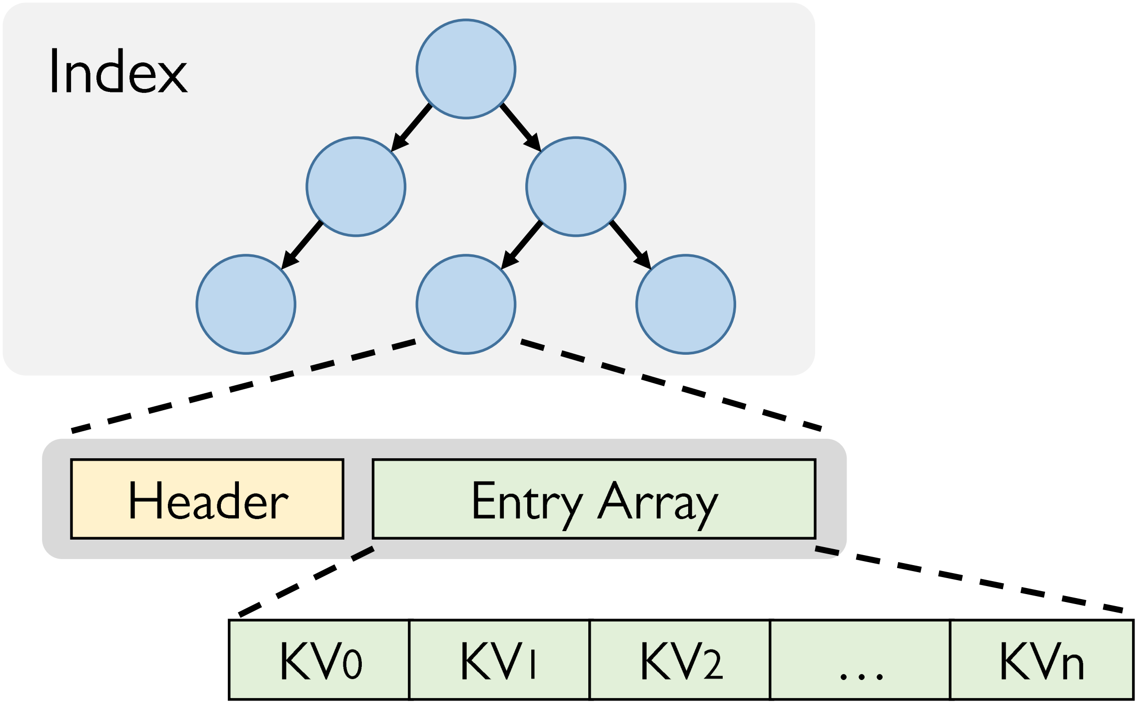
Foreground Put



Problem 4: Excessive small random accesses on PM ↔ high access latency & write amplification

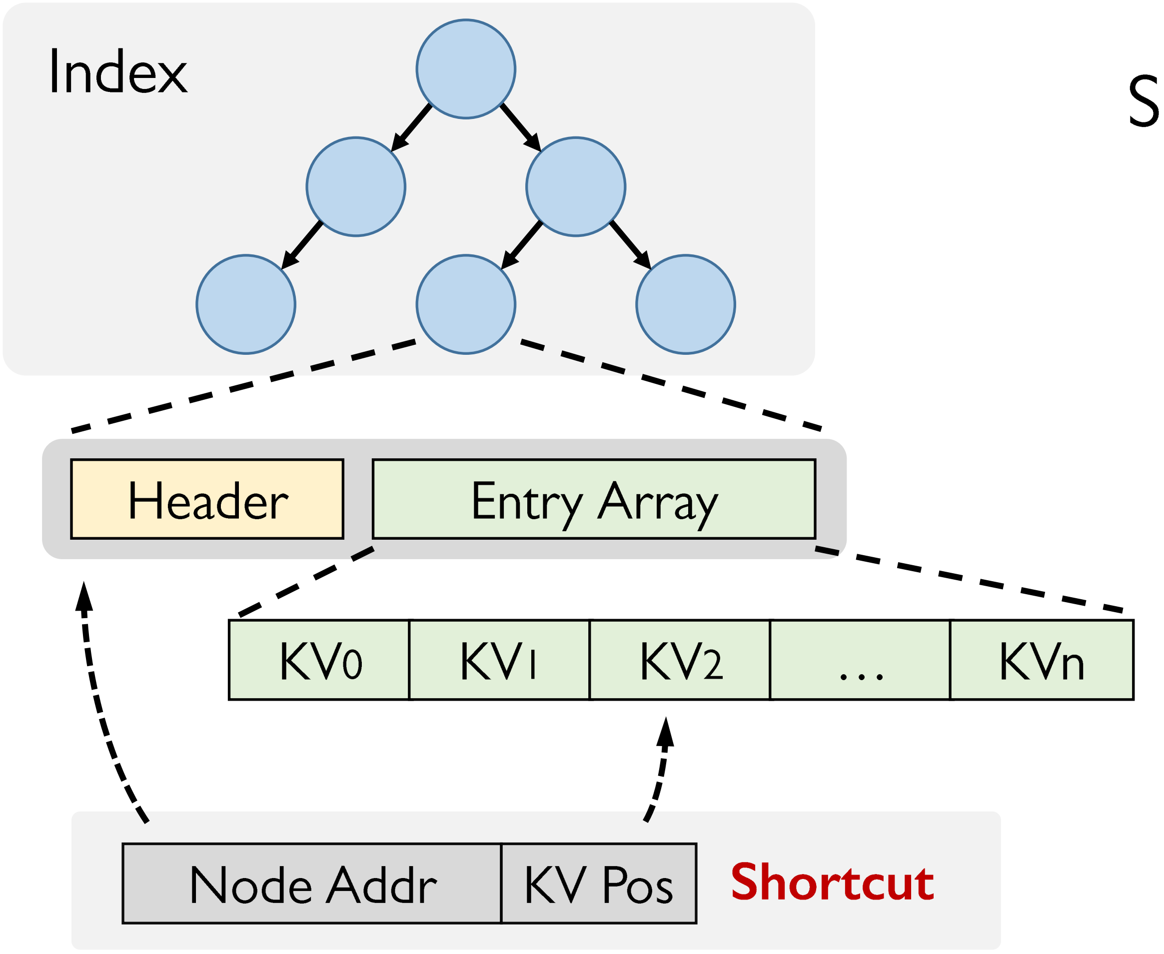
Design (1) - Traverse Index with Shortcut

Pacman uses shortcuts to traverse the index in compaction



Design (1) - Traverse Index with Shortcut

Pacman uses shortcuts to traverse the index in compaction

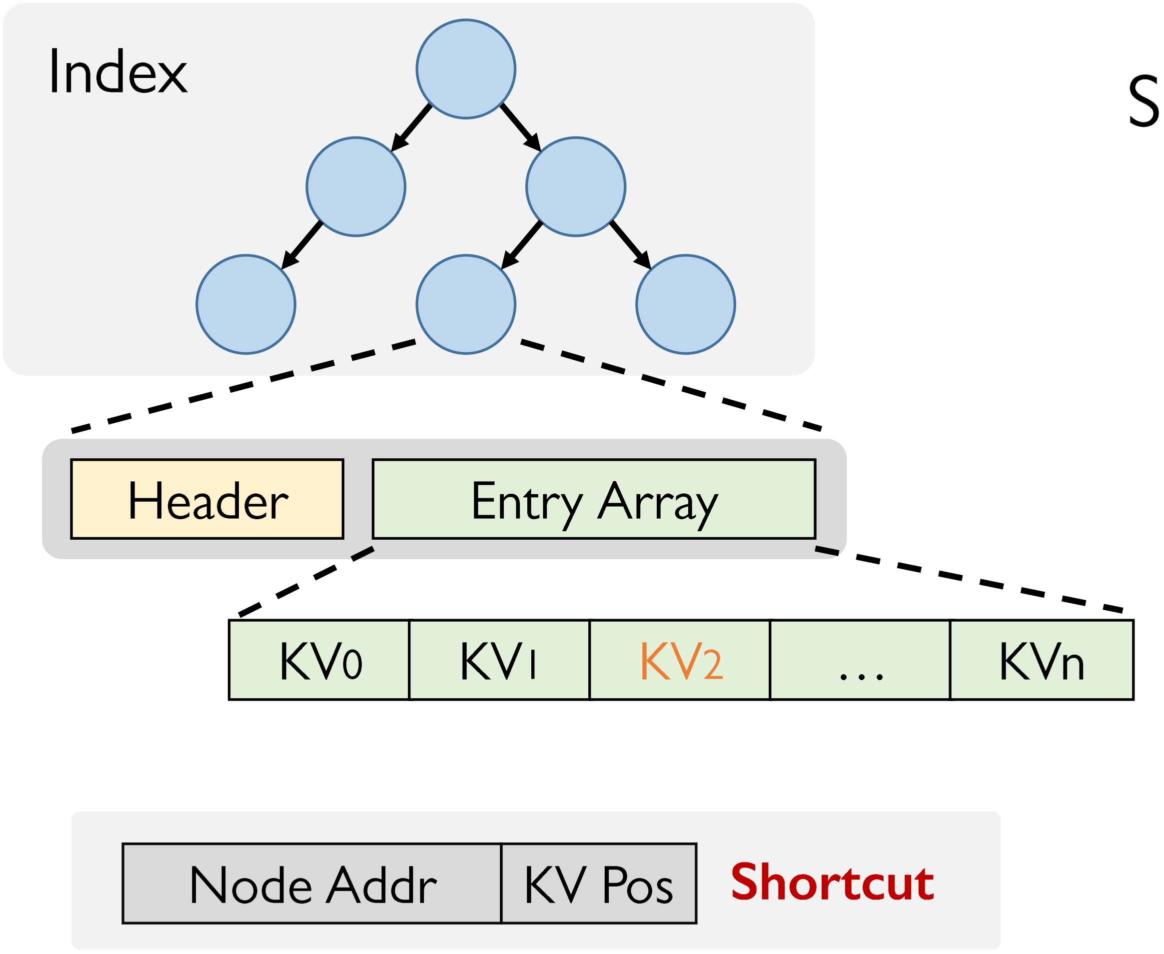


Shortcuts

- ❖ Point at the reference in the index

Design (1) - Traverse Index with Shortcut

Pacman uses shortcuts to traverse the index in compaction

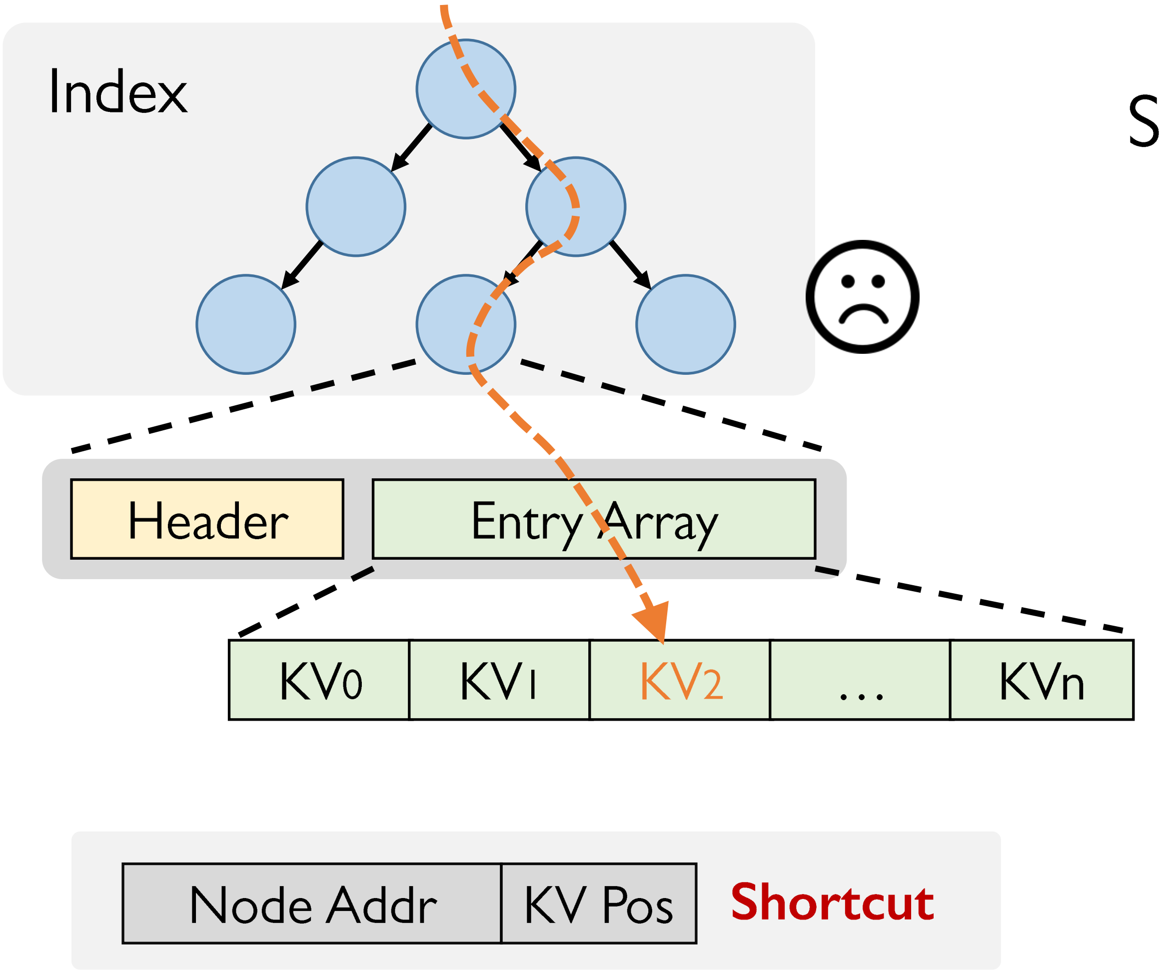


Shortcuts

- Point at the reference in the index

Design (1) - Traverse Index with Shortcut

Pacman uses shortcuts to traverse the index in compaction

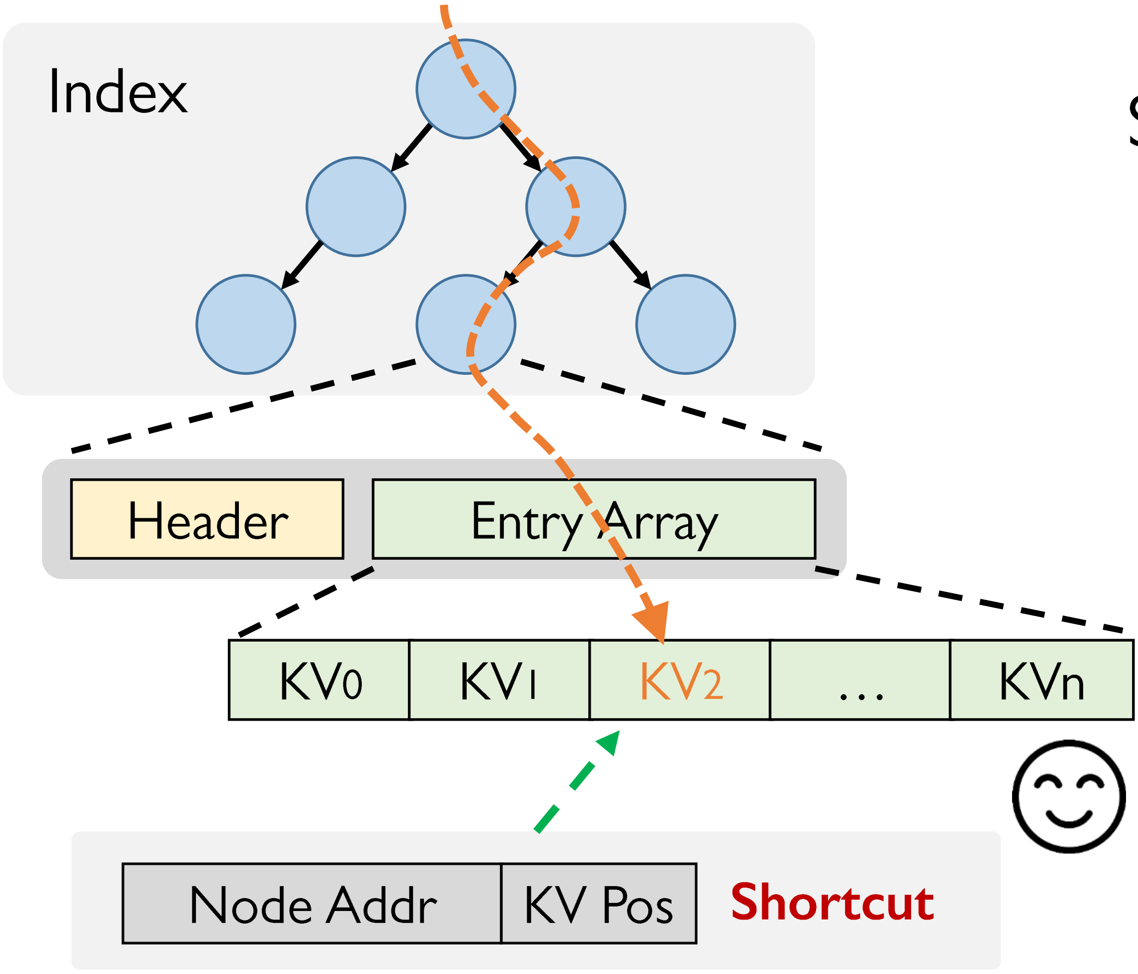


Shortcuts

- Point at the reference in the index

Design (1) - Traverse Index with Shortcut

Pacman uses shortcuts to traverse the index in compaction

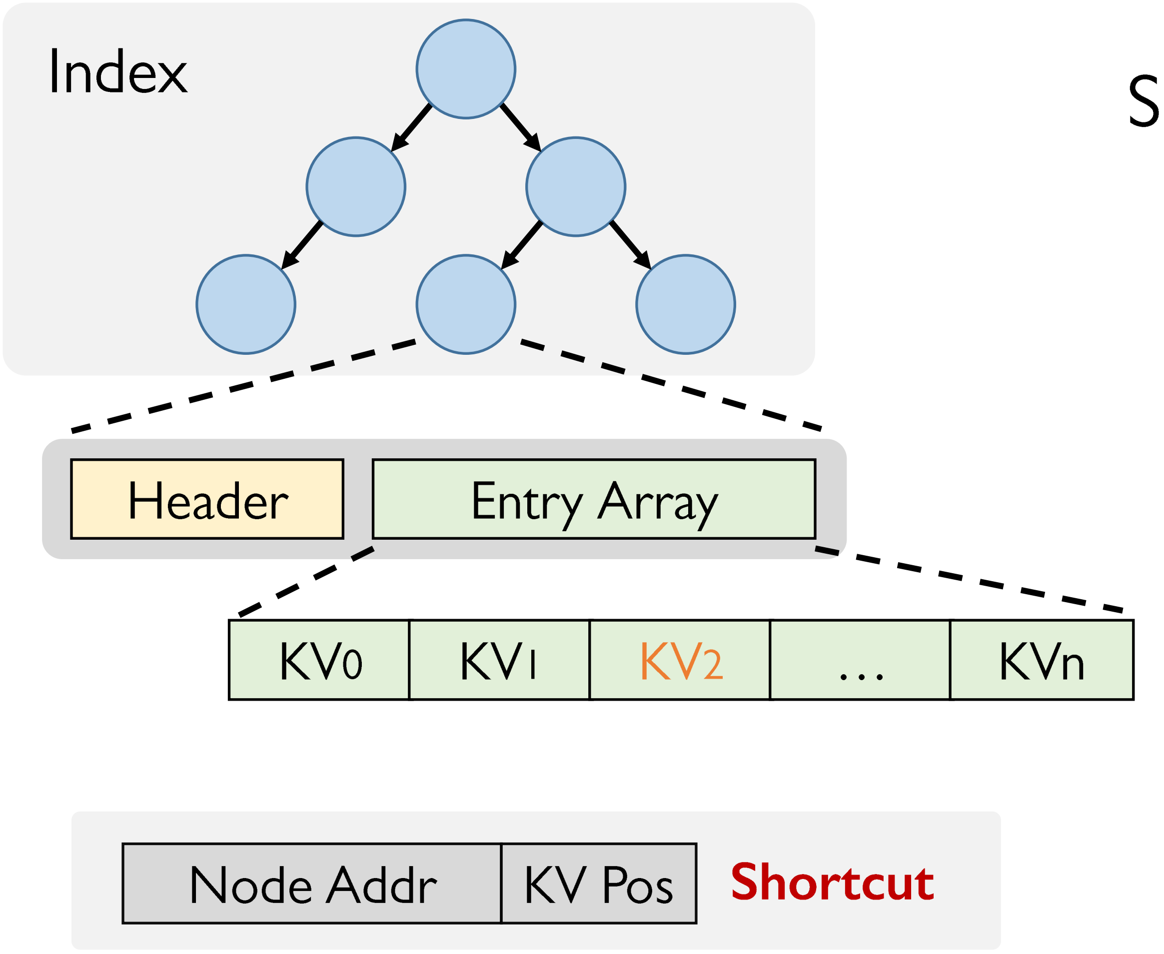


Shortcuts

- Point at the reference in the index
- Skip the high-latency root-to-leaf path

Design (1) - Traverse Index with Shortcut

Pacman uses shortcuts to traverse the index in compaction

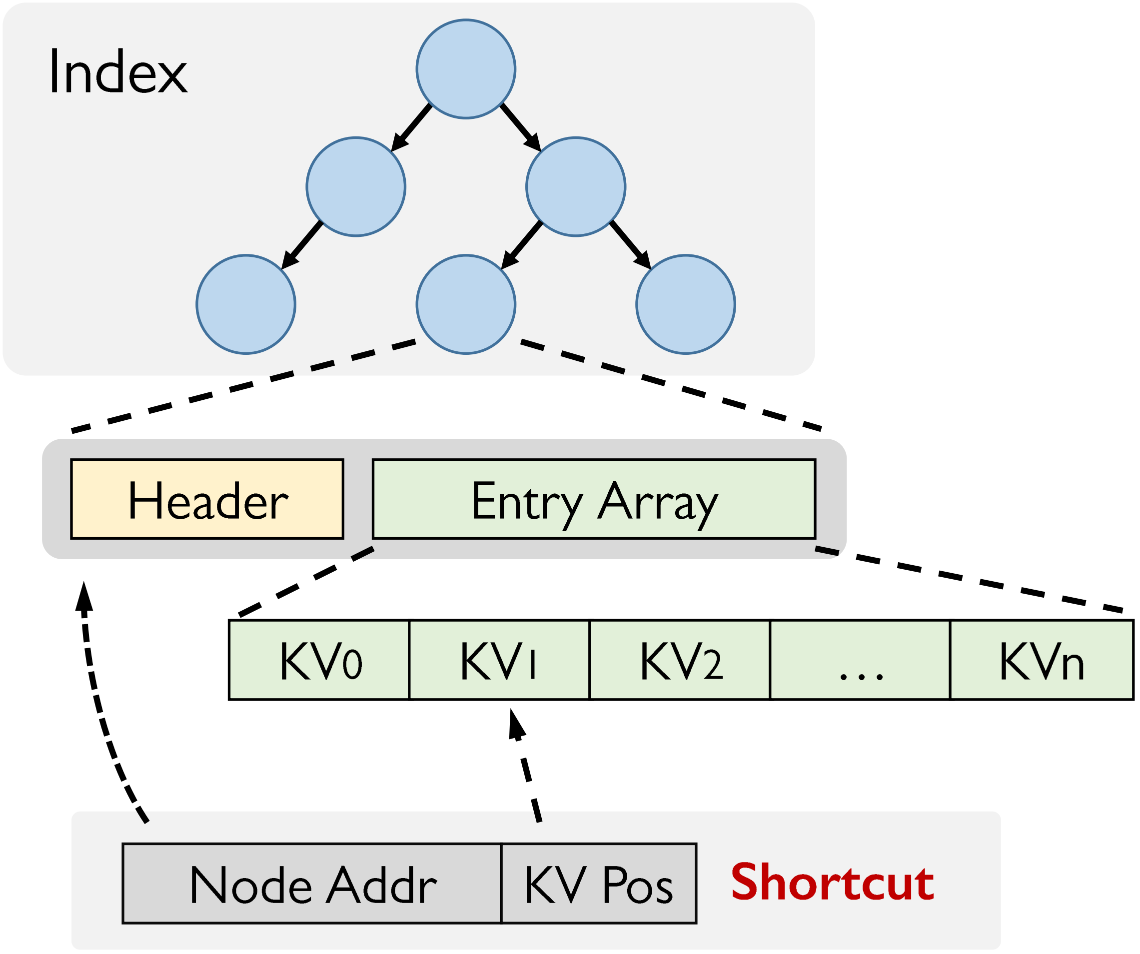


Shortcuts

- Point at the reference in the index
- Skip the high-latency root-to-leaf path
- Recorded by foreground threads in passing when writing new objects
- Stored with corresponding objects in PM log

Design (1) - Traverse Index with Shortcut

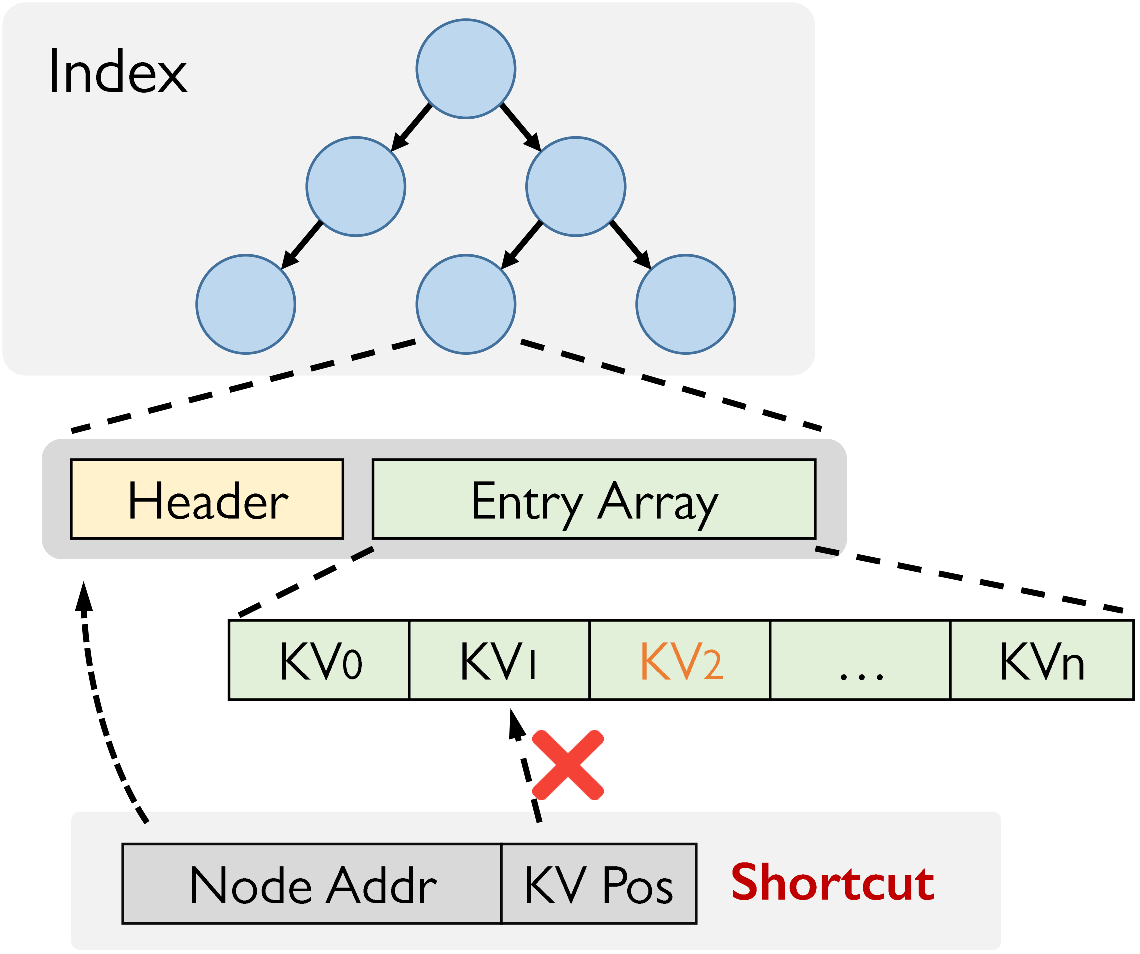
How to handle shortcut invalidation ?



1. *KV Pos* is invalid (e.g., caused by shift)

Design (1) - Traverse Index with Shortcut

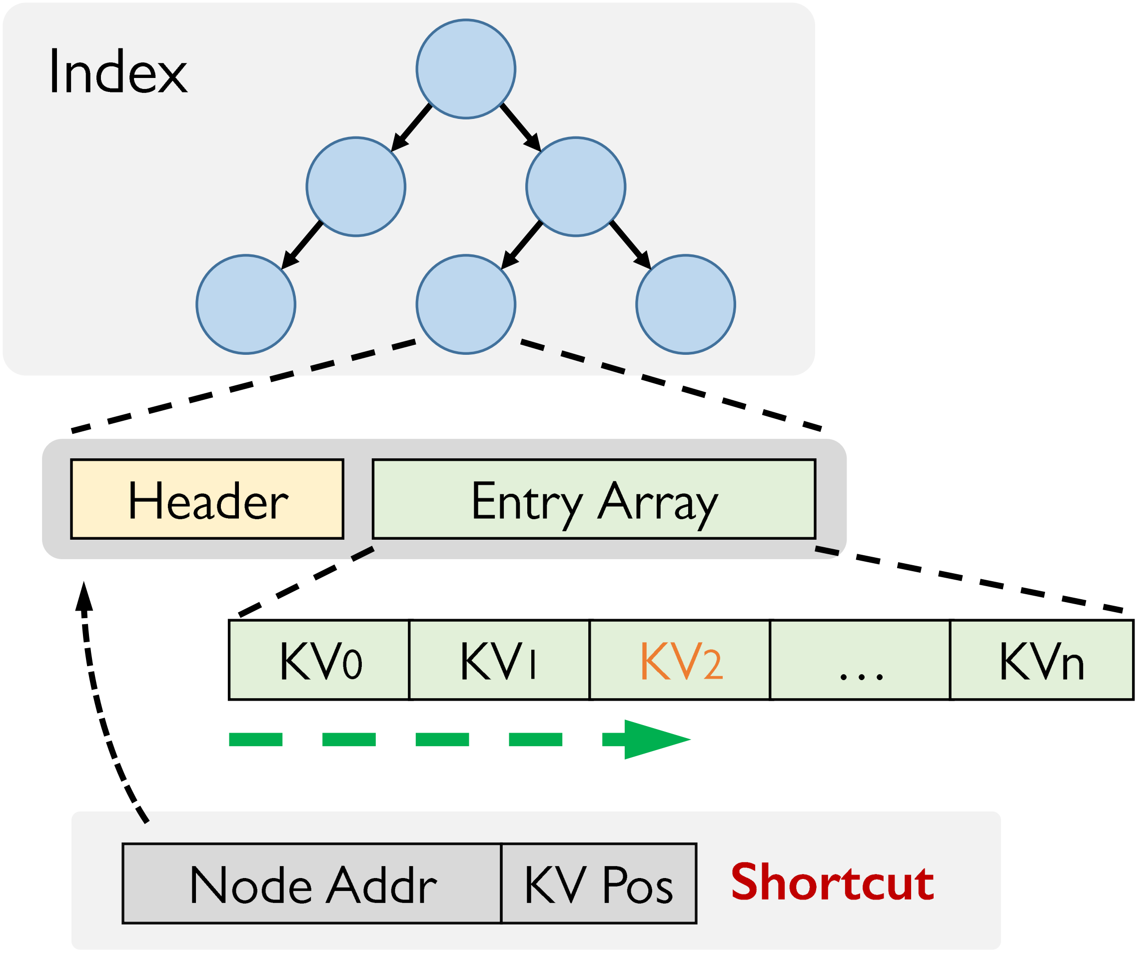
How to handle shortcut invalidation ?



1. *KV Pos* is invalid (e.g., caused by shift)

Design (1) - Traverse Index with Shortcut

How to handle shortcut invalidation ?

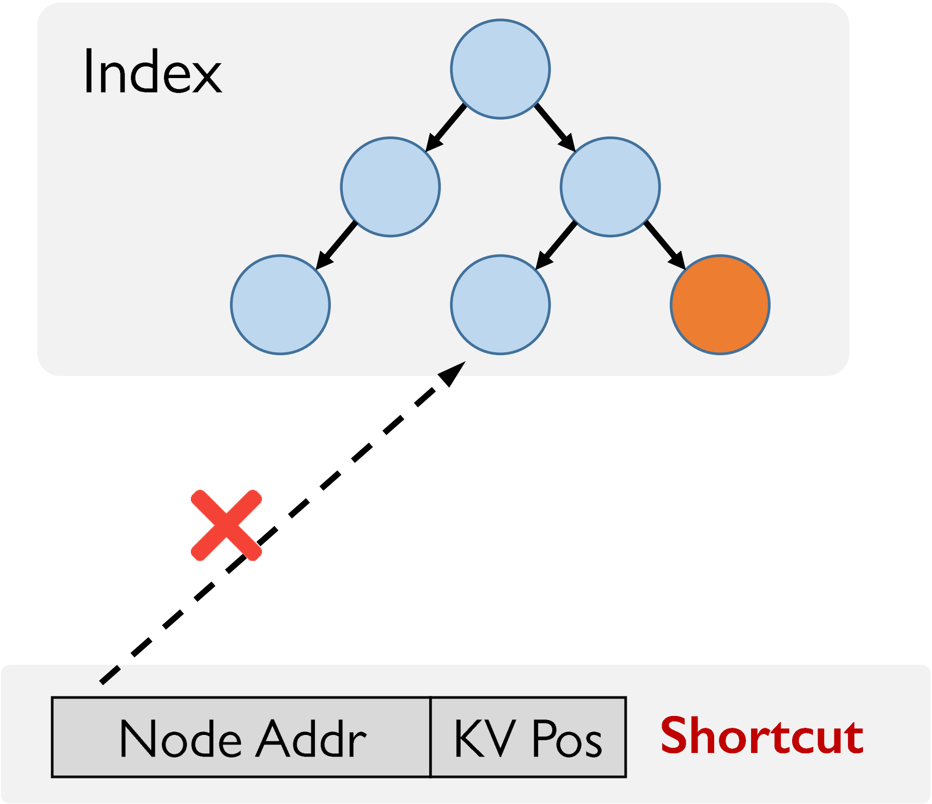


1. *KV Pos* is invalid (e.g., caused by shift)

re-search the leaf node

Design (1) - Traverse Index with Shortcut

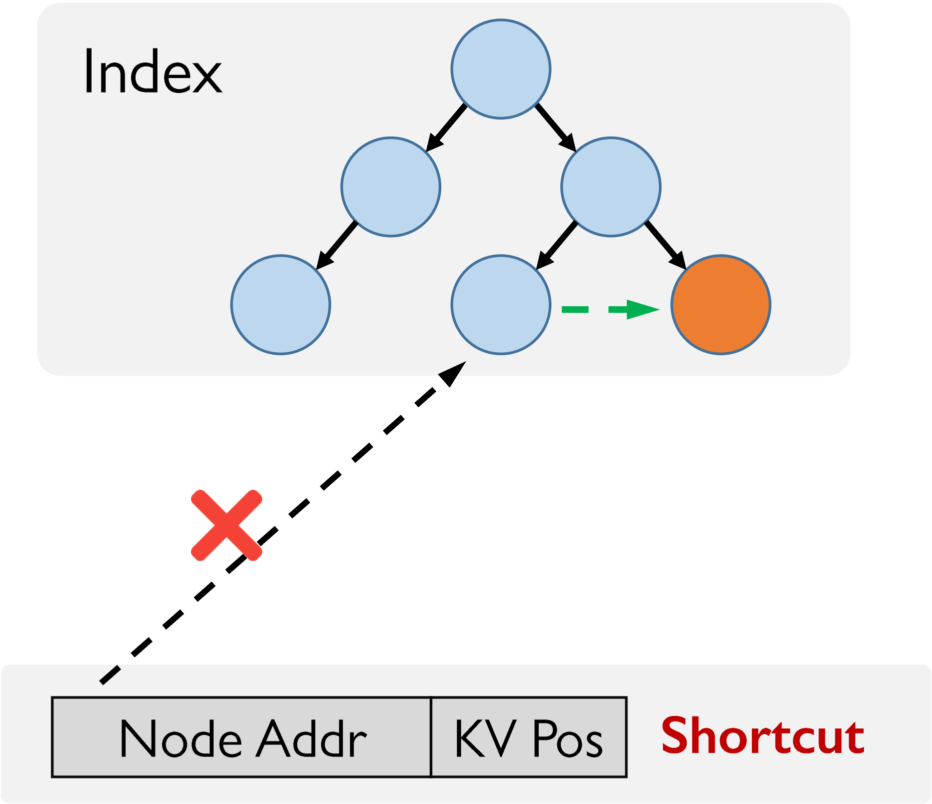
How to handle shortcut invalidation ?



2. the reference has been moved to another node (e.g., caused by shift or rehashing)

Design (1) - Traverse Index with Shortcut

How to handle shortcut invalidation ?

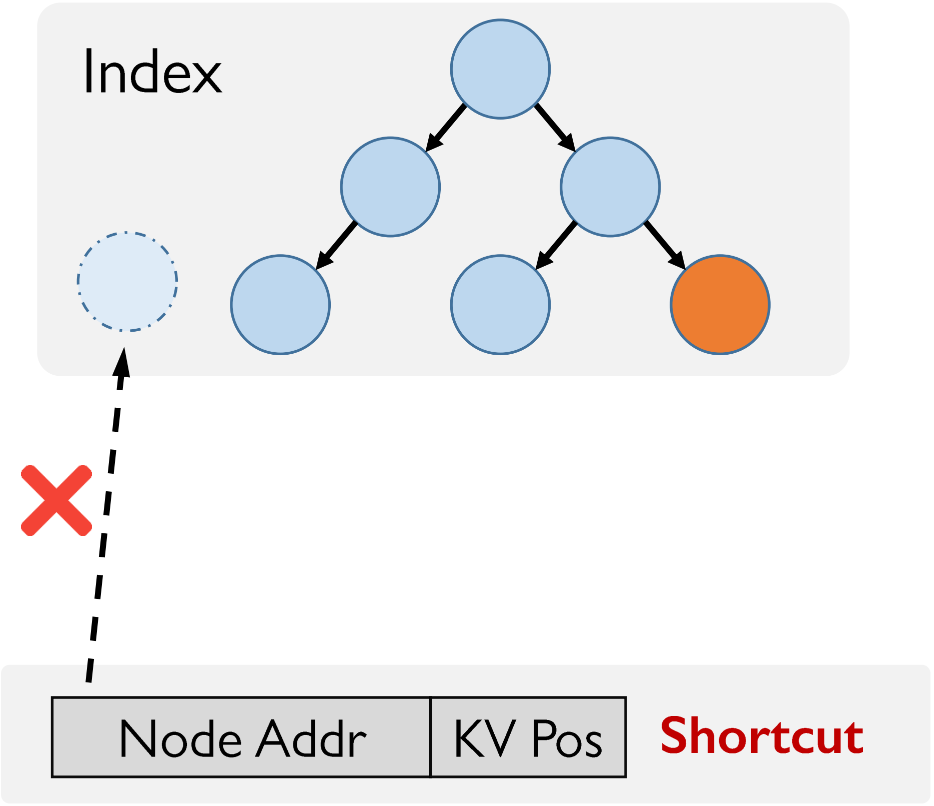


2. the reference has been moved to another node (e.g., caused by shift or rehashing)

try to search the sibling node or fall back to normal search

Design (1) - Traverse Index with Shortcut

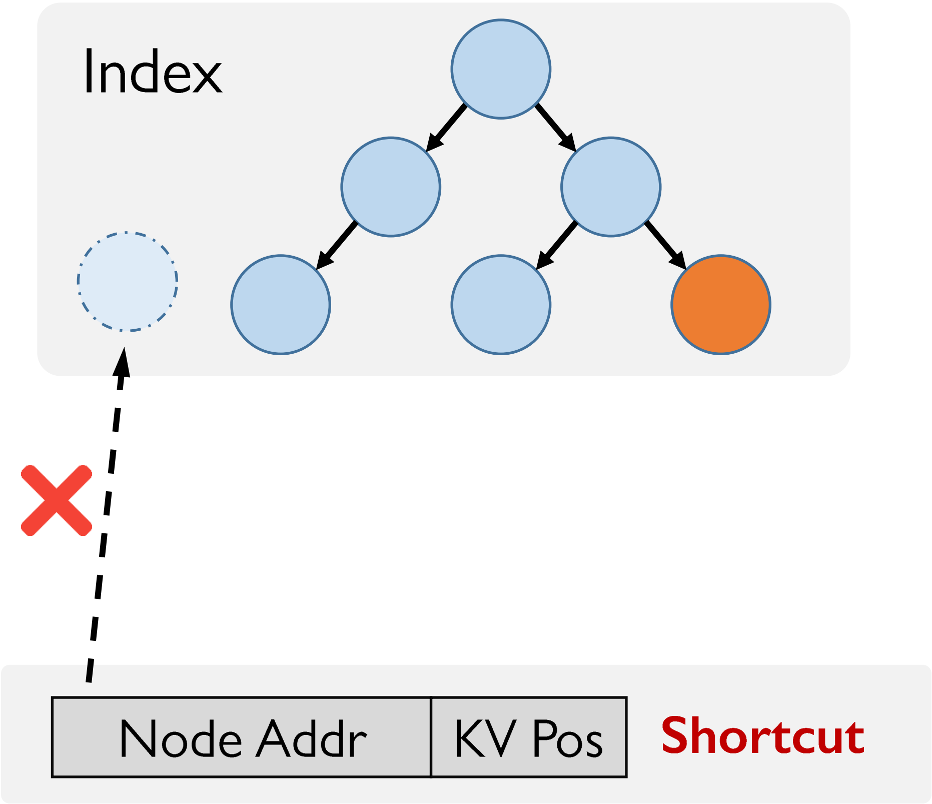
How to handle shortcut invalidation ?



3. the original node pointed by *Node Addr* has been deleted

Design (1) - Traverse Index with Shortcut

How to handle shortcut invalidation ?



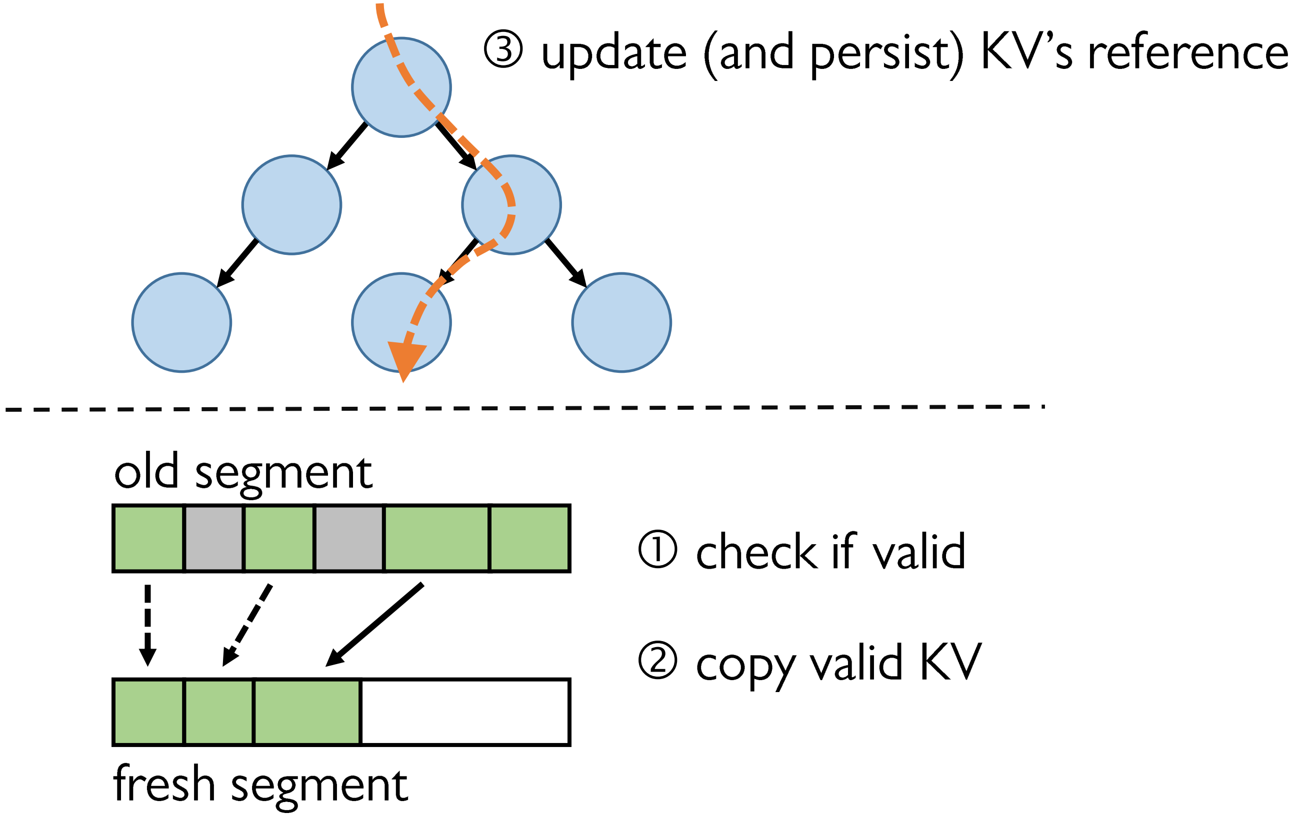
3. the original node pointed by *Node Addr* has been deleted

❖ delete the node logically and reserve its space for future re-allocation

❖ fall back to normal search

Design (2) - Redesign Compaction Pipeline

Background Compaction



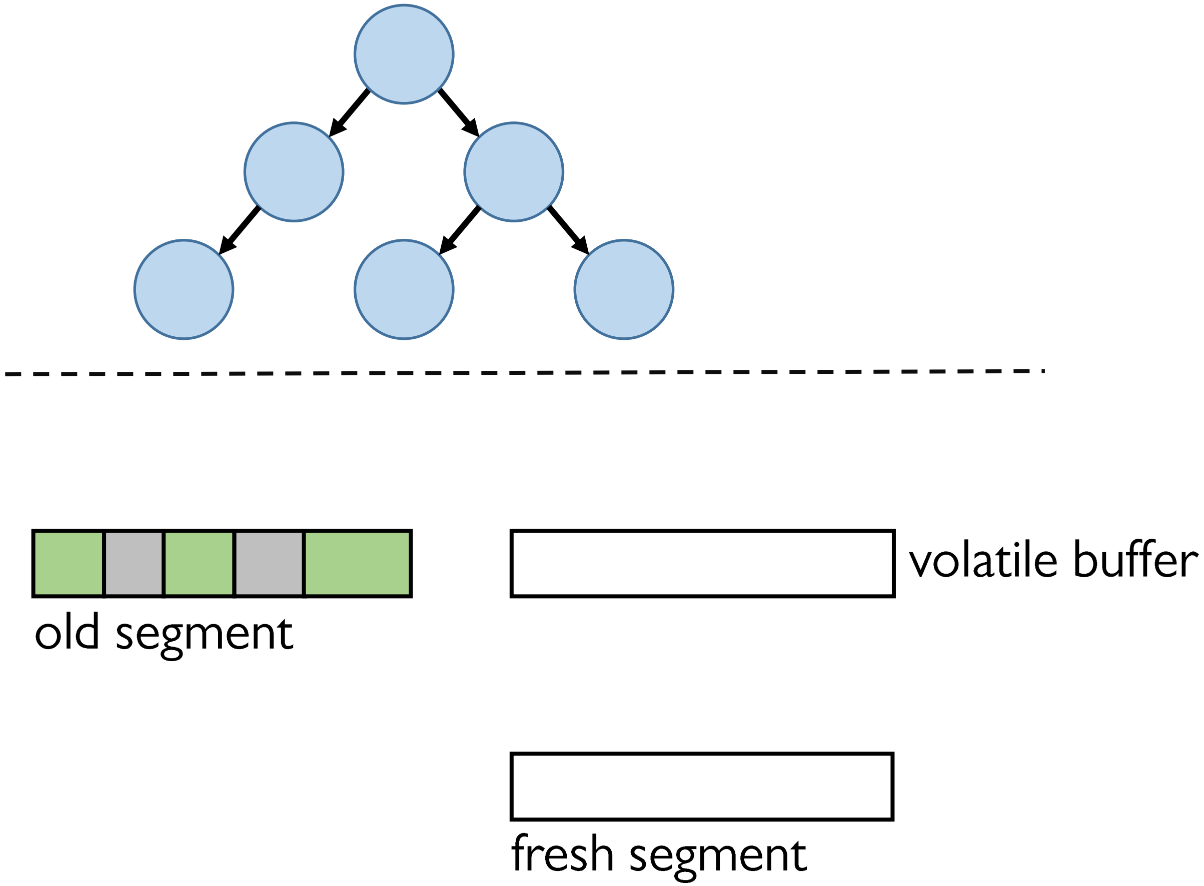
Per-object manner:

❖ one flush and fence between step ② and step ③ for each valid object

❖ serial flush on each updated reference

Design (2) - Redesign Compaction Pipeline

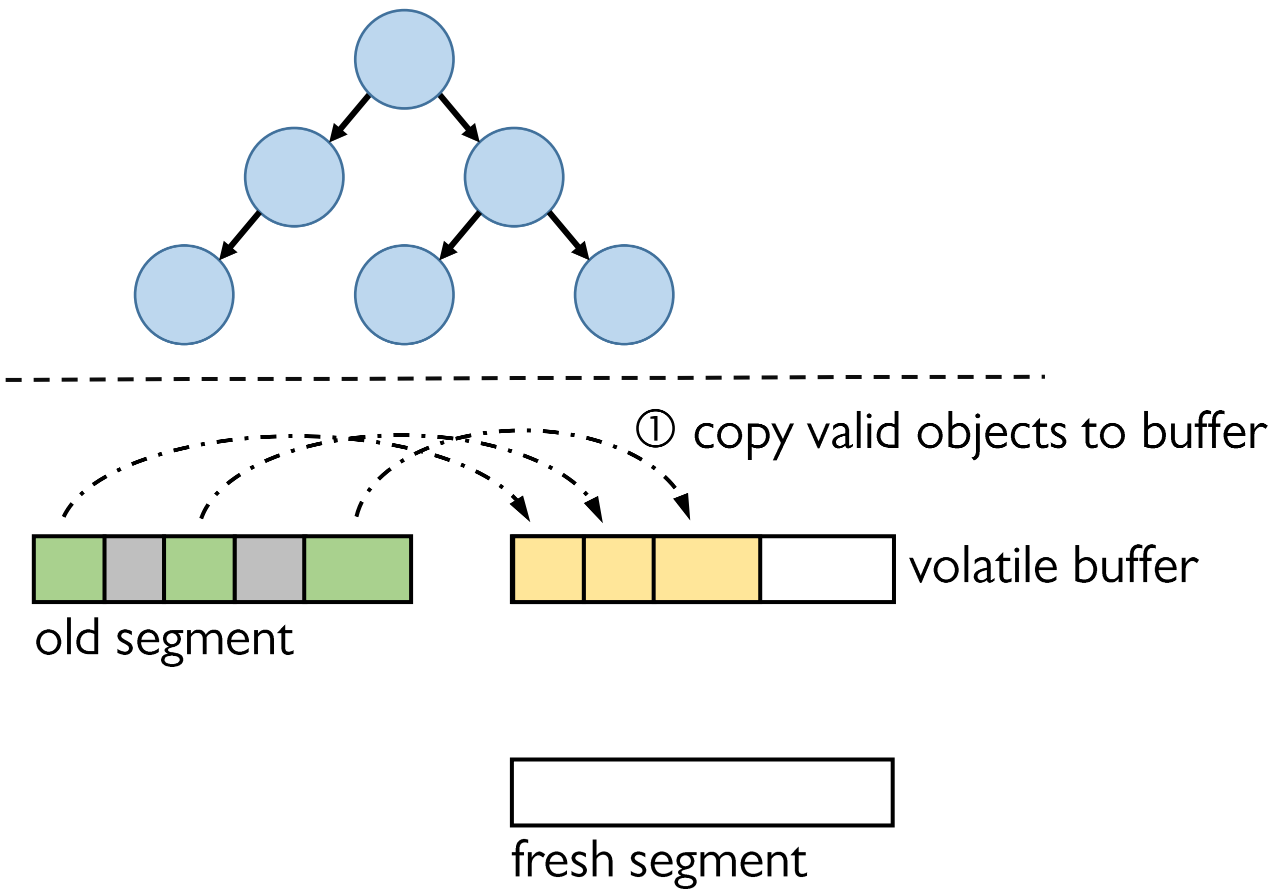
Background Compaction



Batch manner:

Design (2) - Redesign Compaction Pipeline

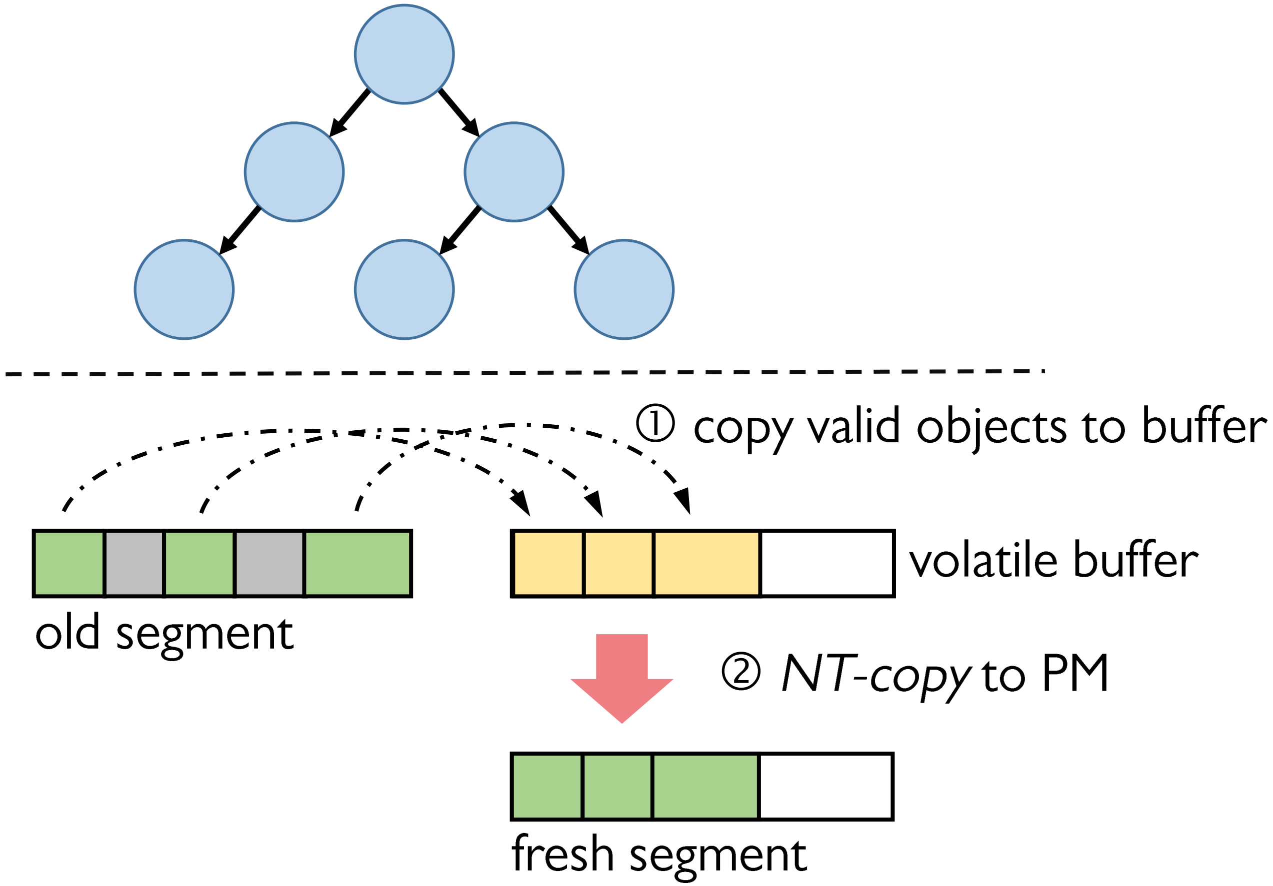
Background Compaction



Batch manner:

Design (2) - Redesign Compaction Pipeline

Background Compaction

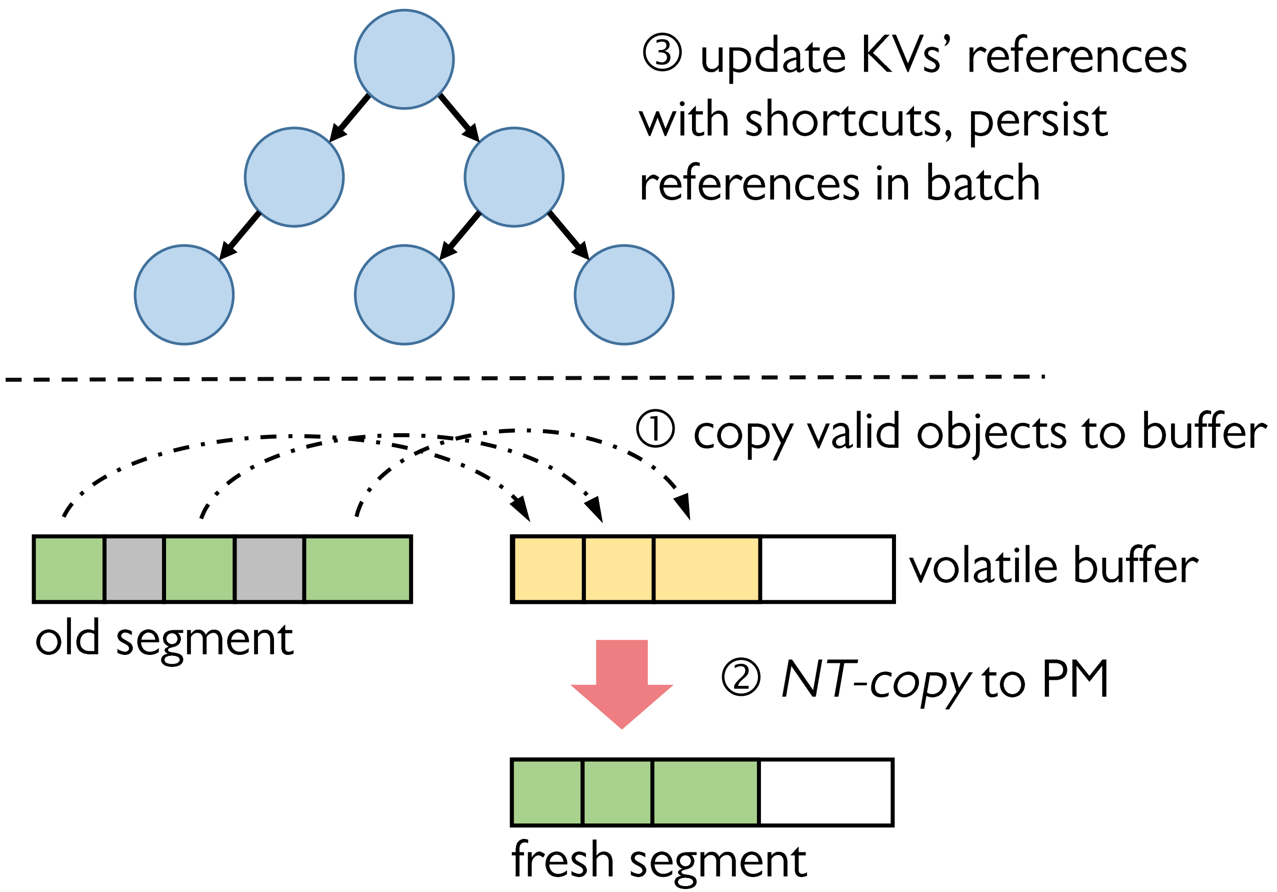


Batch manner:

❖ non-temporal stores (higher bandwidth) to copy bulk of data (step ②)

Design (2) - Redesign Compaction Pipeline

Background Compaction



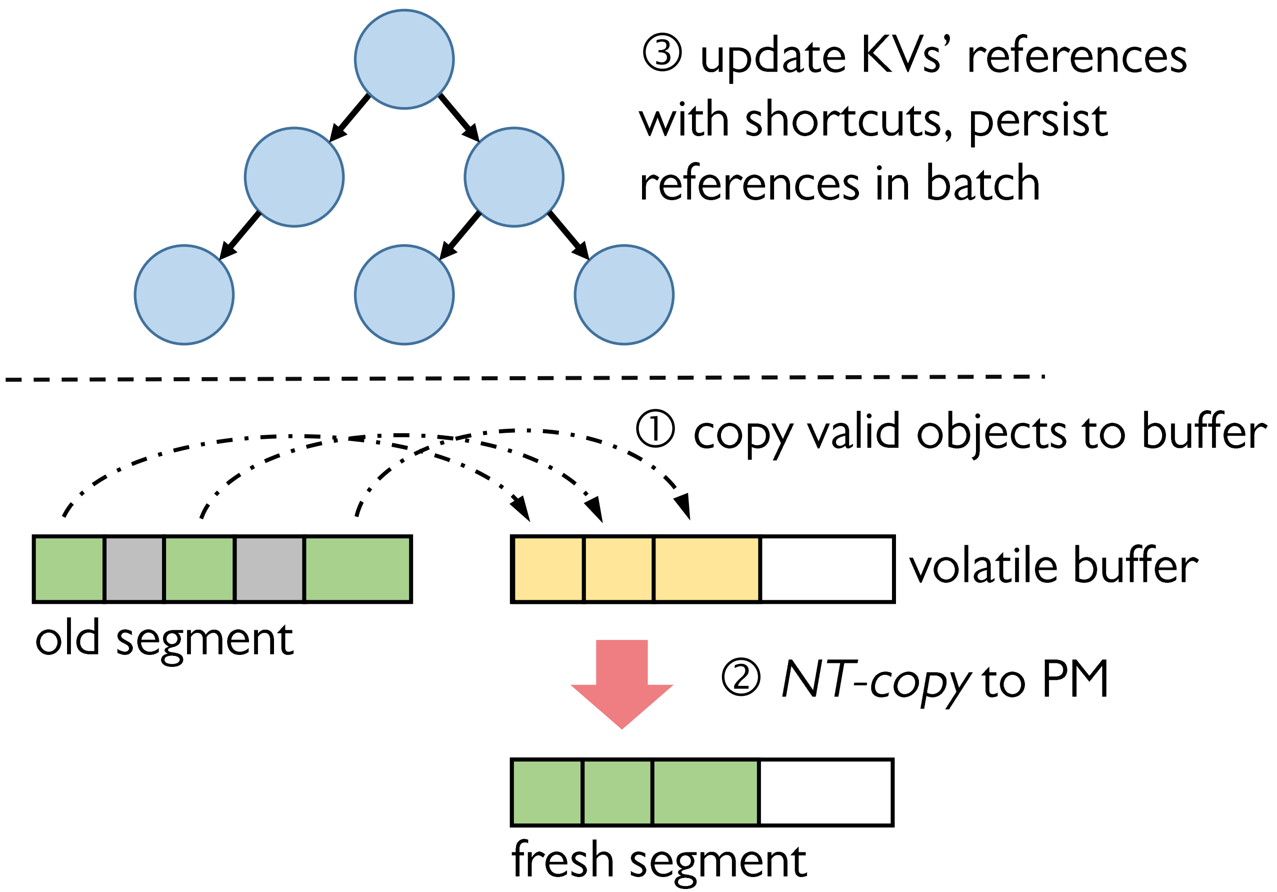
Batch manner:

❖ non-temporal stores (higher bandwidth) to copy bulk of data (step ②)

❖ one fence between step ② and step ③ for the whole segment

Design (2) - Redesign Compaction Pipeline

Background Compaction



Batch manner:

❖ non-temporal stores (higher bandwidth) to copy bulk of data (step ②)

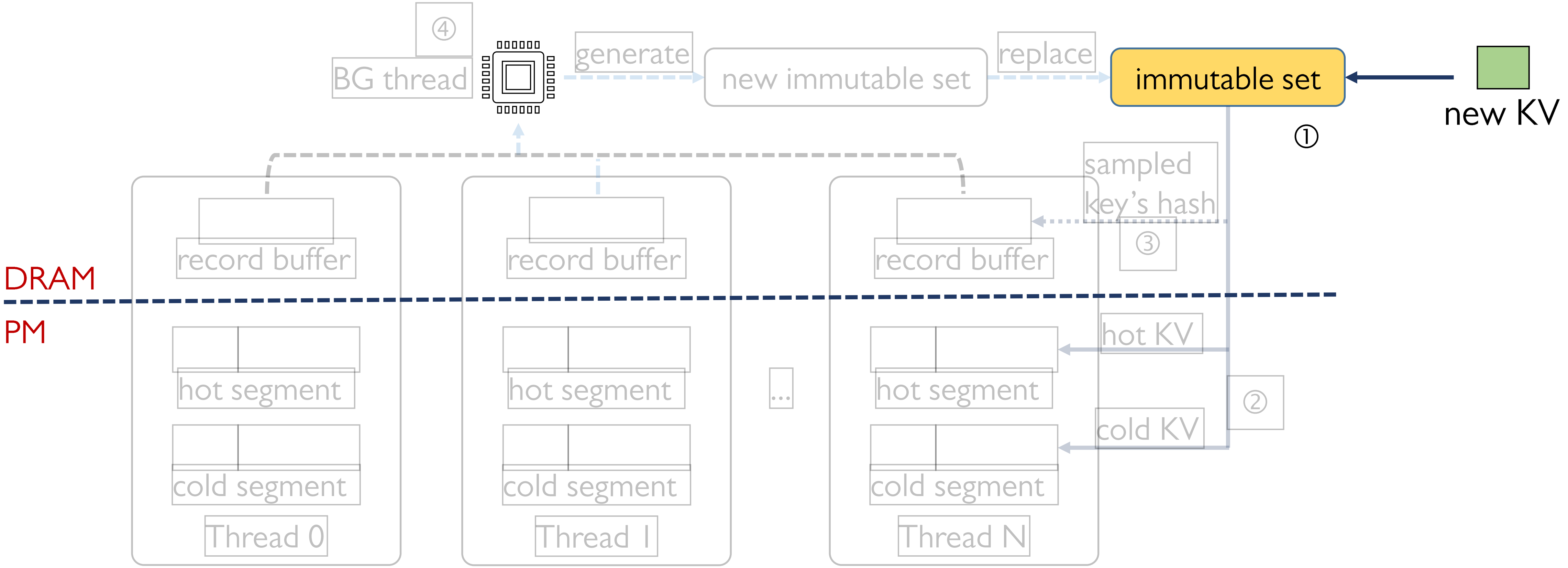
❖ one fence between step ② and step ③ for the whole segment

❖ launch concurrent flushes on batched updated references (step ③)

❖ prefetch references via shortcuts to cover access latency

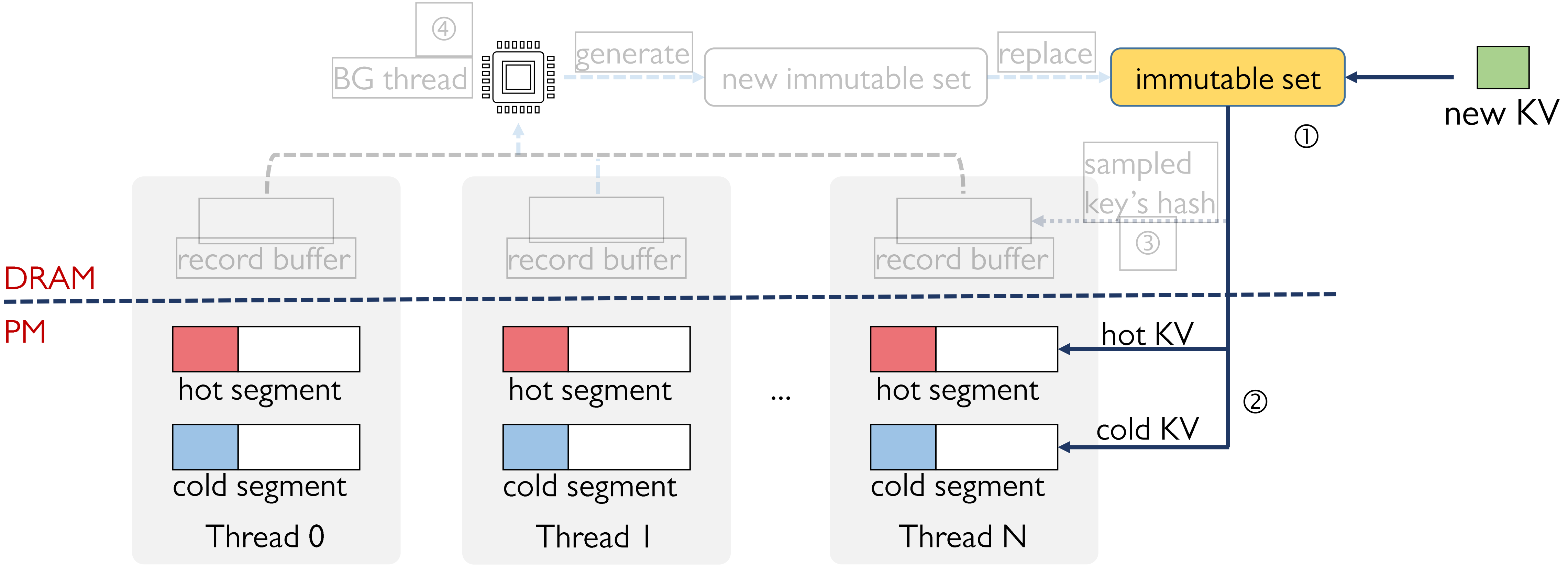
Design (3) - Separate Hot-Cold Data

① a lightweight approach to identify hot and cold KV
② store hot and cold objects in different log segments
③ count hot proportion and record key's hash by sampling
④ a background thread generate new hot set if hotspots shift



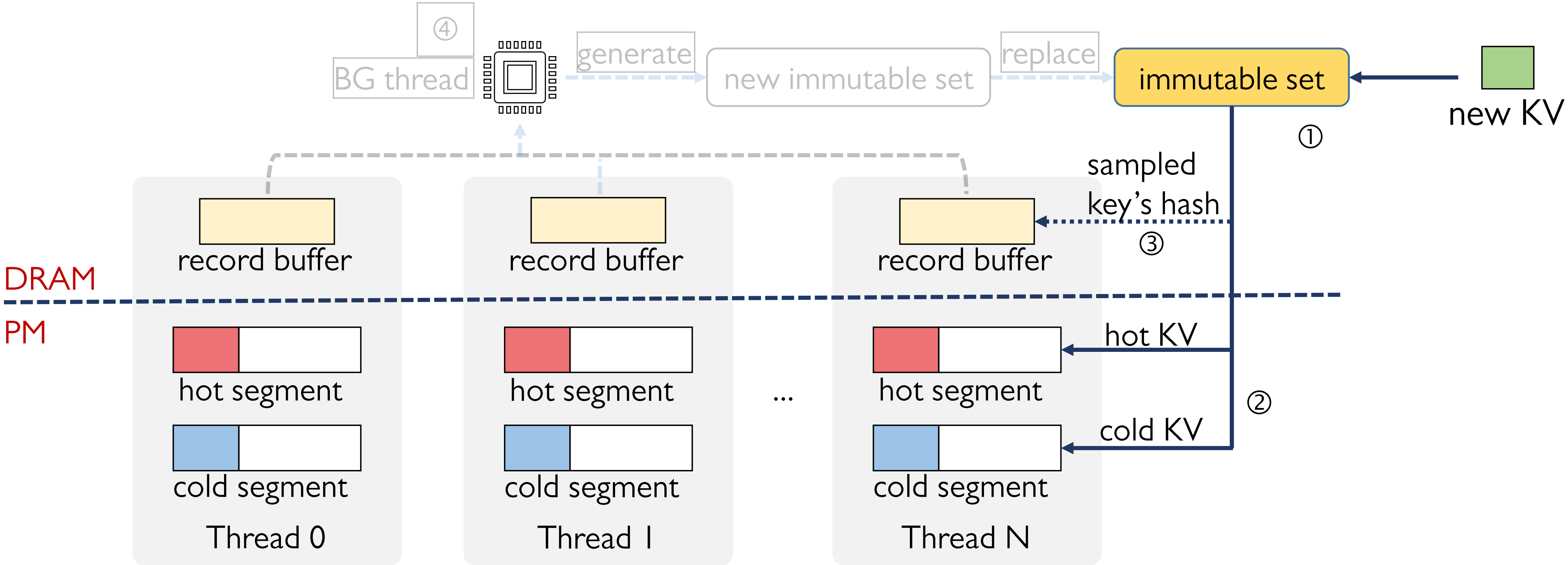
Design (3) - Separate Hot-Cold Data

① a lightweight approach to identify hot and cold KV
② store hot and cold objects in different log segments
③ count hot proportion and record key's hash by sampling
④ a background thread generate new hot set if hotspots shift



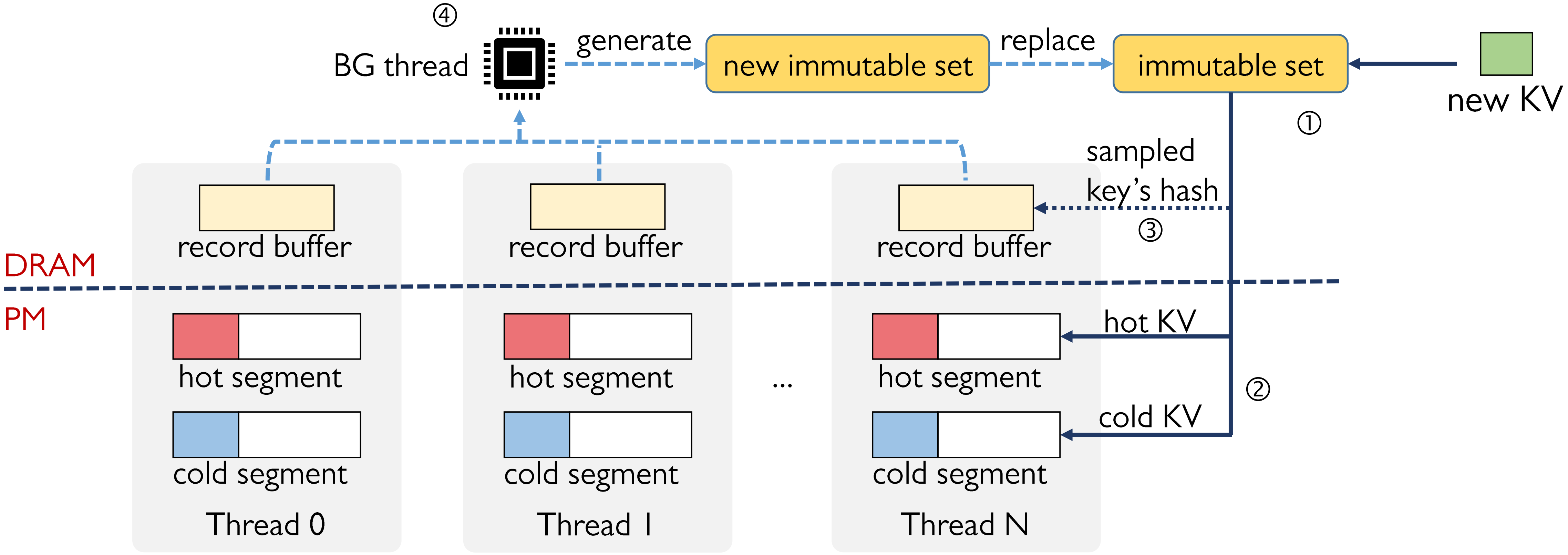
Design (3) - Separate Hot-Cold Data

① a lightweight approach to identify hot and cold KV
② store hot and cold objects in different log segments
③ count hot proportion and record key's hash by sampling
④ a background thread generate new hot set if hotspots shift



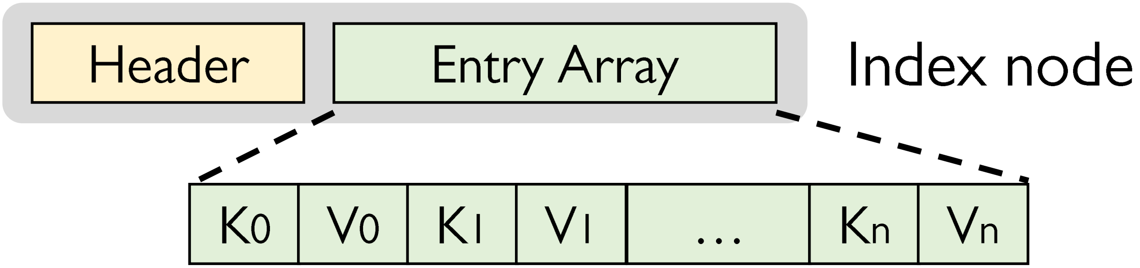
Design (3) - Separate Hot-Cold Data

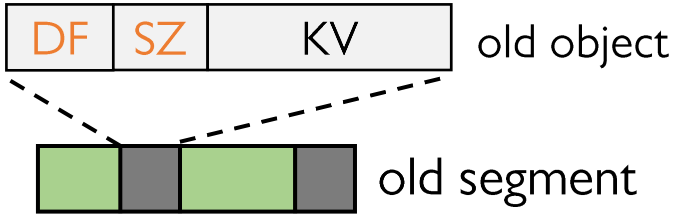
① a lightweight approach to identify hot and cold KV
② store hot and cold objects in different log segments
③ count hot proportion and record key's hash by sampling
④ a background thread generate new hot set if hotspots shift



Design (4) – Reduce excessive PM accesses

Foreground Put

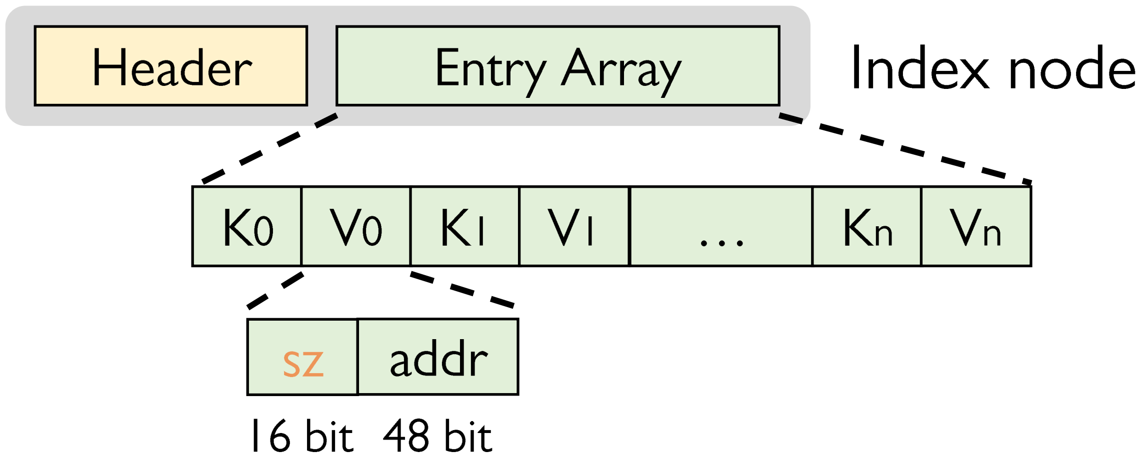


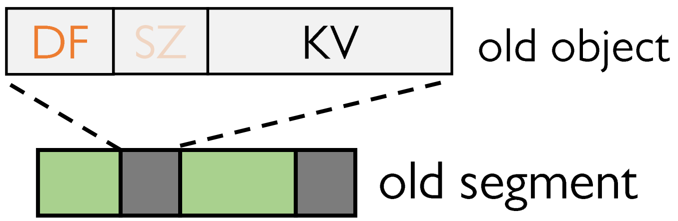


read size (SZ) to increase old segment's garbage bytes and write old object's deleted flag (DF)

Design (4) – Reduce excessive PM accesses

Foreground Put



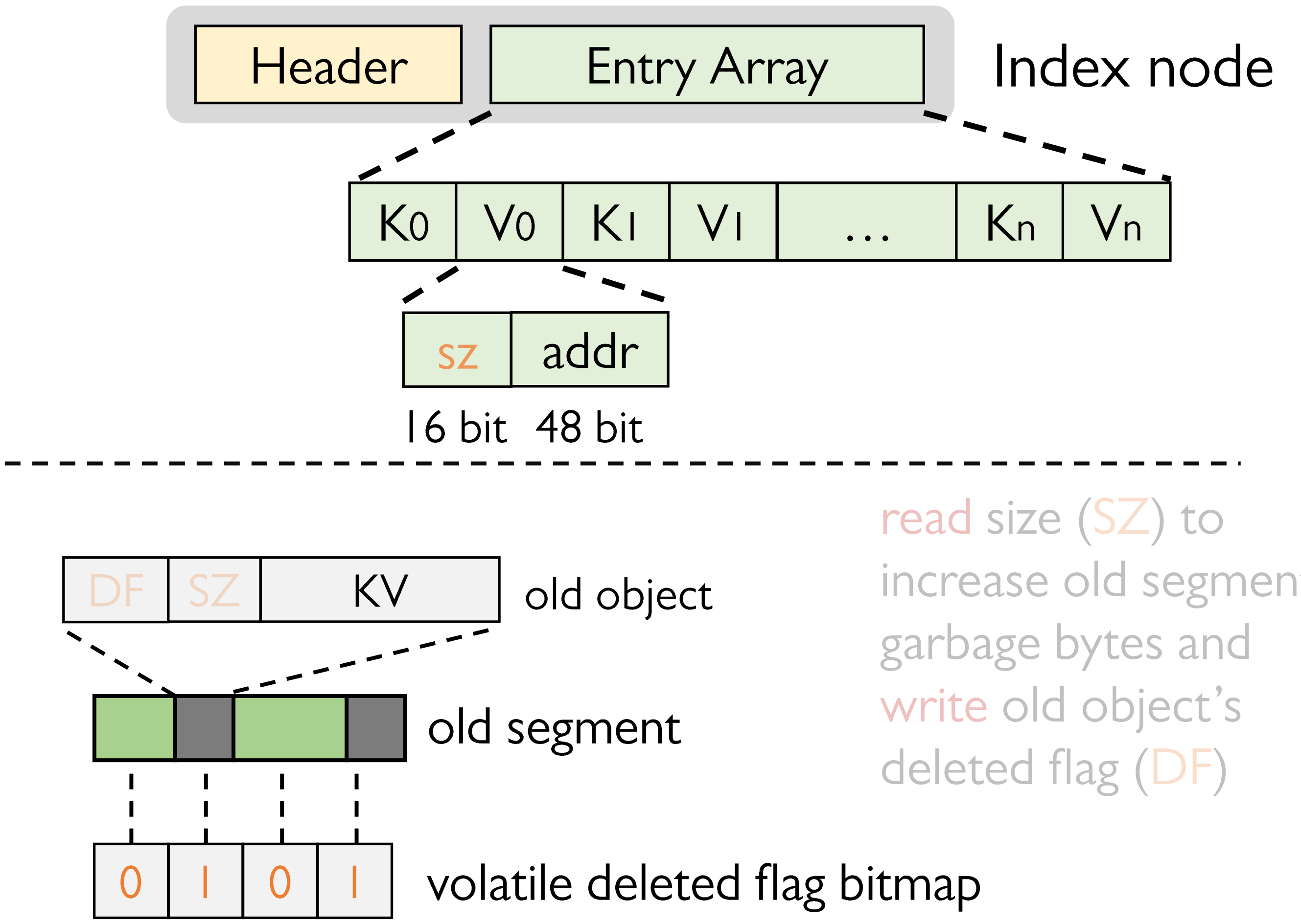


read size (SZ) to increase old segment's garbage bytes and write old object's deleted flag (DF)

❖ Embed size (SZ) in the reference

Design (4) – Reduce excessive PM accesses

Foreground Put

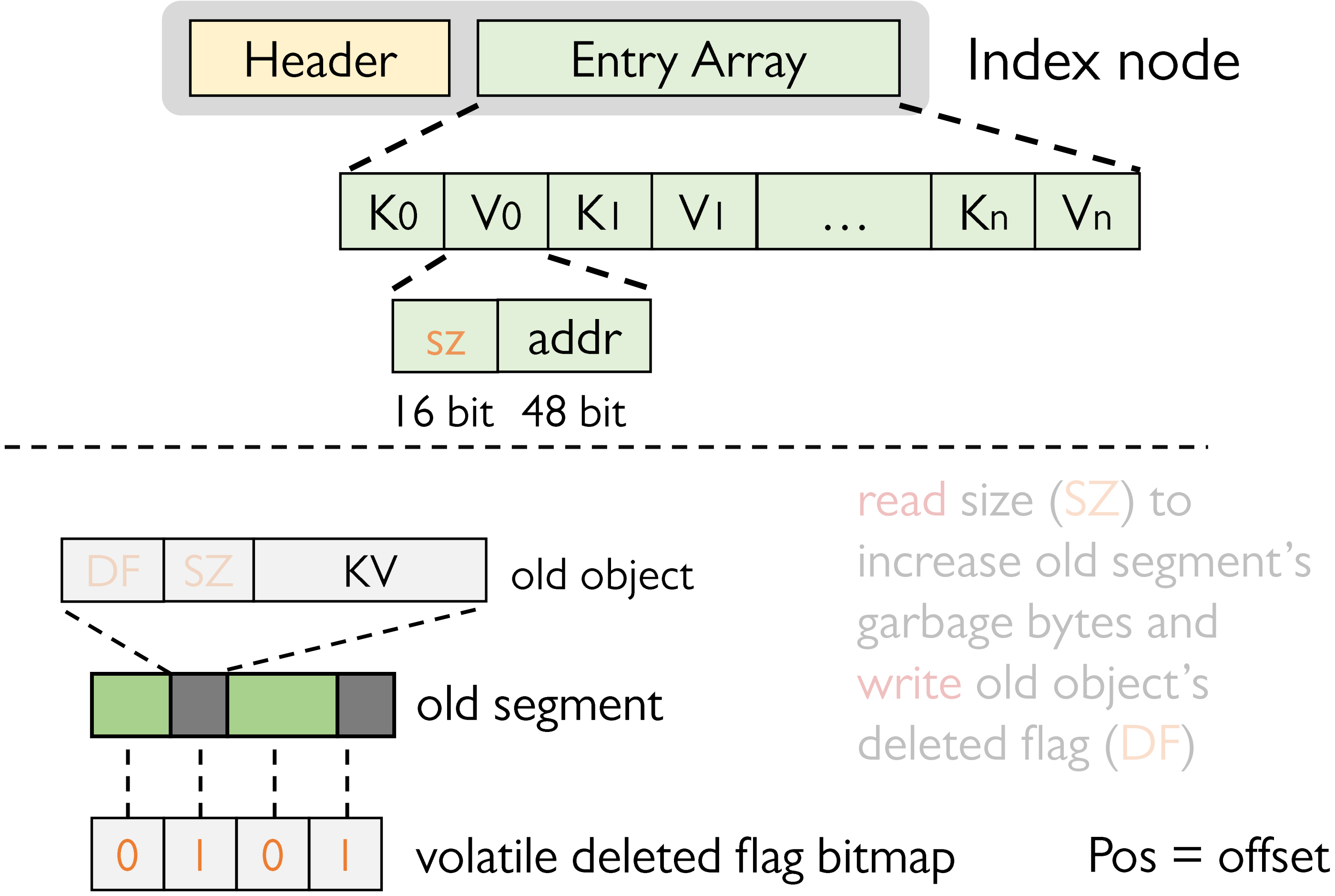


read size (SZ) to increase old segment's garbage bytes and write old object's deleted flag (DF)

- ❖ Embed size (SZ) in the reference
- ❖ In-DRAM deleted flag (DF) bitmap

Design (4) – Reduce excessive PM accesses

Foreground Put



Pos = offset / MIN_SIZE

- Embed size (SZ) in the reference
- In-DRAM deleted flag (DF) bitmap

More design details: Check our paper

Optimize space overhead of shortcut

- compress size of shortcut
- do not store shortcut for hot objects

Recovery of Pacman

- shortcut and DRAM-resident data structures
- crash of batch compaction

Outline

- ❖ Background & Motivation
- ❖ Pacman – A PM-aware Compaction Approach for Log-structured KVS
- ❖ Results
- ❖ Conclusion

Experimental Setup

Hardware Platform (one socket)

CPU	18-core Intel Xeon Gold 6240 CPU
PM	3 * 128 GB Intel Optane DC PMMs
DRAM	96 GB DDR4 DIMMs

Applied KV stores

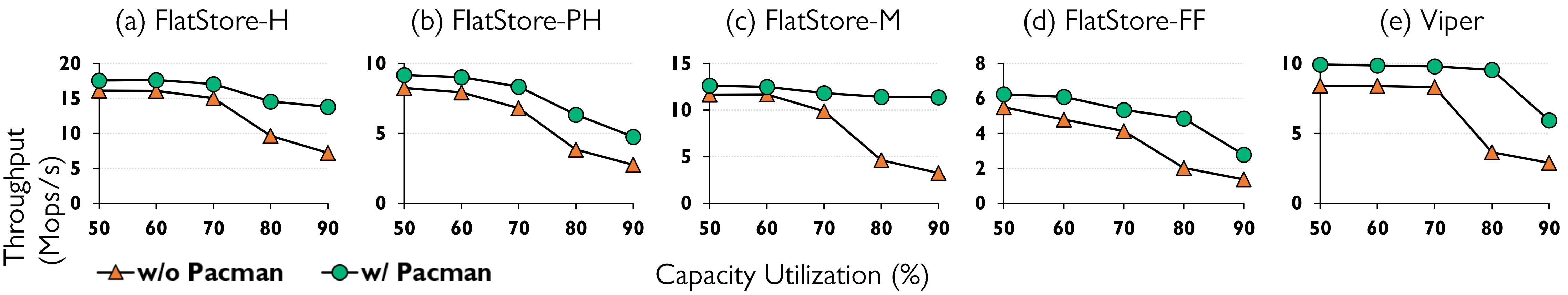
- FlatStore (ASPLOS'20): 4 version: [volatile CCEH, CCEH, Masstree, FastFair] + PM log
- Viper (VLDB'21): volatile CCEH + PM log

Workloads

- YCSB (varying thread count, value size, r/w ratio, etc)
- Facebook ETC Pool: Mixture of small & large KV pairs

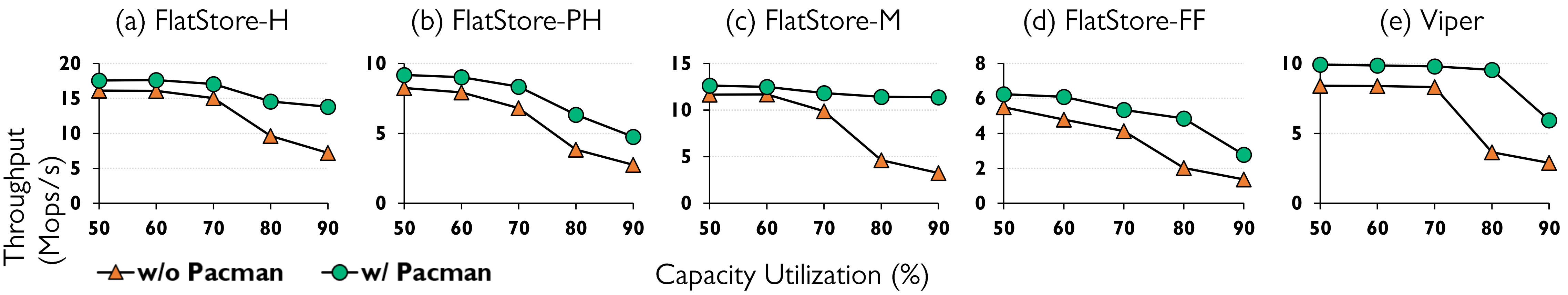
Overall Performance

YCSB-A (50% Get, 50% Put) , 8B key, 48B value, 12 service threads, 4 cleaners



Overall Performance

YCSB-A (50% Get, 50% Put) , 8B key, 48B value, 12 service threads, 4 cleaners



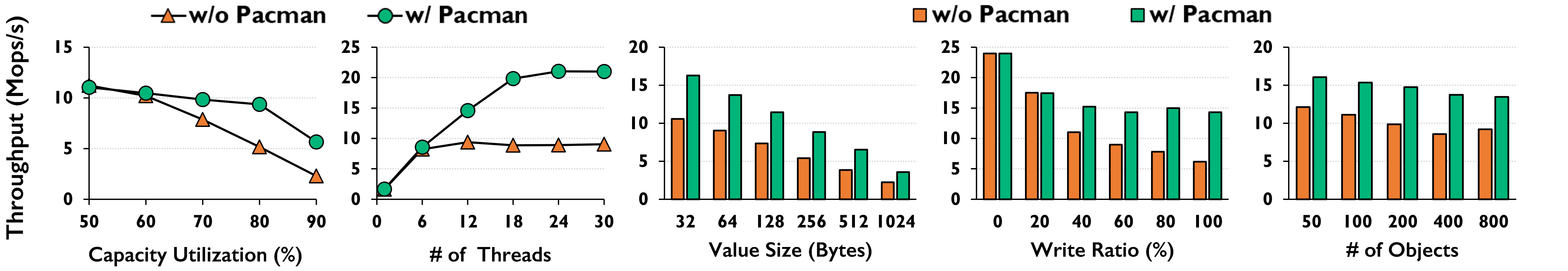
Pacman

- obviously curtails performance decline at high capacity utilizations, 1.5-3.5× speedup
  → efficient PM-aware compaciton
- also enhances performance at low capacity utilizaitons
  → reduces small random accesses and saves PM's limited bandwidth

Sensitivity Analysis

Default settings

distribution	capacity utilization	# of threads	# of cleaners	value size	write ratio	# of objects
Zipfian	80%	12	4	48 bytes	50%	200 M

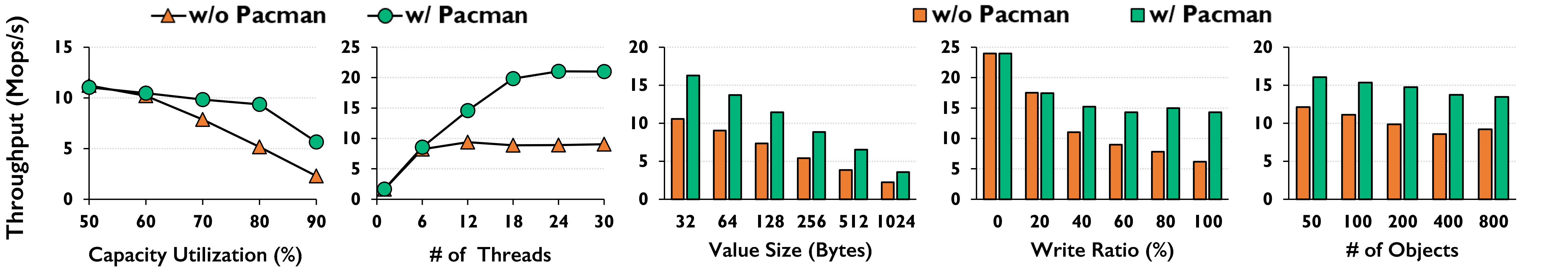


(a) uniform workloads (b) thread scalability (c) varying value sizes (d) varying write ratios (e) varying numbers of objects

Sensitivity Analysis

Default settings

distribution	capacity utilization	# of threads	# of cleaners	value size	write ratio	# of objects
Zipfian	80%	12	4	48 bytes	50%	200 M



(a) uniform workloads (b) thread scalability (c) varying value sizes (d) varying write ratios (e) varying numbers of objects

Pacman is effective in write-intensive workloads with various settings

Facebook ETC workload

- Facebook ETC Pool: mixture of small & large KV pairs
  - Tiny (1-13 bytes, 40%), Zipfian distribution
  - Median (14-300 bytes, 55%), Zipfian distribution
  - Large (> 300 bytes, 5%), uniform distribution

Facebook ETC workload

- Facebook ETC Pool: mixture of small & large KV pairs
  - Tiny (1-13 bytes, 40%), Zipfian distribution
  - Median (14-300 bytes, 55%), Zipfian distribution
  - Large (> 300 bytes, 5%), uniform distribution
- 50% Get, 50% Put

Facebook ETC workload

- Facebook ETC Pool: mixture of small & large KV pairs
  - Tiny (1-13 bytes, 40%), Zipfian distribution
  - Median (14-300 bytes, 55%), Zipfian distribution
  - Large (> 300 bytes, 5%), uniform distribution
- 50% Get, 50% Put
- Compared systems: PMem-RocksDB, pmemkv, ChameleonDB

Facebook ETC workload

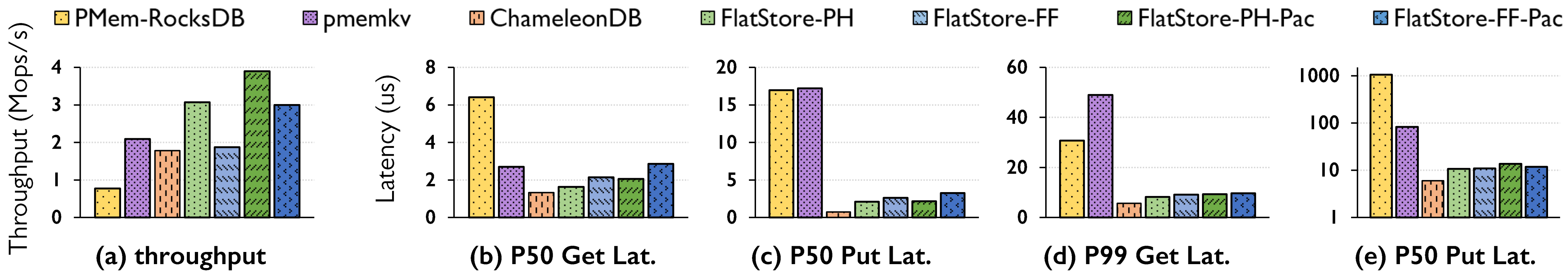
- Facebook ETC Pool: mixture of small & large KV pairs
  - Tiny (1-13 bytes, 40%), Zipfian distribution
  - Median (14-300 bytes, 55%), Zipfian distribution
  - Large (> 300 bytes, 5%), uniform distribution
- 50% Get, 50% Put
- Compared systems: PMem-RocksDB, pmemkv, ChameleonDB
- pmemkv 28 threads; others 24 threads, 4 cleaners

Facebook ETC workload

- Facebook ETC Pool: mixture of small & large KV pairs
  - Tiny (1-13 bytes, 40%), Zipfian distribution
  - Median (14-300 bytes, 55%), Zipfian distribution
  - Large (> 300 bytes, 5%), uniform distribution
- 50% Get, 50% Put
- Compared systems: PMem-RocksDB, pmemkv, ChameleonDB
- pmemkv 28 threads; others 24 threads, 4 cleaners
- 80% capacity utilization for ChameleonDB and FlatStore

Facebook ETC workload

- Facebook ETC Pool: mixture of small & large KV pairs
  - Tiny (1-13 bytes, 40%), Zipfian distribution
  - Median (14-300 bytes, 55%), Zipfian distribution
  - Large (> 300 bytes, 5%), uniform distribution
- 50% Get, 50% Put
- Compared systems: PMem-RocksDB, pmemkv, ChameleonDB
- pmemkv 28 threads; others 24 threads, 4 cleaners
- 80% capacity utilization for ChameleonDB and FlatStore



Outline

- ❖ Background & Motivation
- ❖ Pacman – A PM-aware Compaction Approach for Log-structured KVS
- ❖ Results
- ❖ Conclusion

Conclusion

- Current compaction approaches are unaware of PM's characteristics, which exacerbate the bottleneck of compaction in log-structured KV stores.

Conclusion

- Current compaction approaches are unaware of PM's characteristics, which exacerbate the bottleneck of compaction in log-structured KV stores.

- We propose Pacman, an efficient PM-aware compaction approach for log-structured KV stores on PM.

Conclusion

- Current compaction approaches are unaware of PM's characteristics, which exacerbate the bottleneck of compaction in log-structured KV stores.

- We propose Pacman, an efficient PM-aware compaction approach for log-structured KV stores on PM.

- Enhance state-of-the-art PM-based log-structured KV stores (FlatStore & Viper)

Conclusion

- Current compaction approaches are unaware of PM's characteristics, which exacerbate the bottleneck of compaction in log-structured KV stores.
- We propose Pacman, an efficient PM-aware compaction approach for log-structured KV stores on PM.
- Enhance state-of-the-art PM-based log-structured KV stores (FlatStore & Viper)
- More evaluation results and analysis are in the paper

Thanks & QA

Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory

Open-source code: https://github.com/thustorage/pacman

Contact Information: wangjing19@mails.tsinghua.edu.cn