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)
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)
PM’s idiosyncrasies exacerbate the bottleneck of compaction.
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

FlatStore-FF

Viper
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

Index

old segment

fresh segment
Problem (1) - High-latency Index Traversal

Background Compaction

Index

old segment

fresh segment

① check if valid
Problem (1) - High-latency Index Traversal

Background Compaction

Index

old segment

fresh segment

① check if valid

② copy valid object
Problem (1) - High-latency Index Traversal

Background Compaction

Index

1. check if valid
2. copy valid object
3. update object’s reference

old segment

fresh segment
Problem (1) - High-latency Index Traversal

Background Compaction

Index

③ update object’s reference

---

old segment

① check if valid

fresh segment

② copy valid object

Problem 1: Traversing the index ↔ high access latency
Problem (2) – Extra persistence work

1. check if valid
2. copy valid object
3. update object’s reference

Background Compaction

old segment

fresh segment
Problem (2) – Extra persistence work

Background Compaction

Index

③ update object’s reference

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

old segment

① check if valid

fresh segment

② copy valid object
Problem (2) – Extra persistence work

- Background Compaction

- Index

- ① check if valid
- ② copy valid object
- ③ update object’s reference

- old segment

- fresh segment

- 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

Index

③ update object’s reference

old segment

① check if valid

② copy valid object

fresh segment
Problem (3) – A large amount of data copying

Background Compaction

1. check if valid
2. copy valid object
3. update object’s reference

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

Index

old segment

fresh segment
Problem (3) – A large amount of data copying

Background Compaction

old segment

fresh segment

Index

① check if valid
② copy valid object

③ update object’s reference

❖ 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

1. Append new object
2. Update object’s reference
3. Write old object’s deleted flag (DF) and read size (SZ) to increase old segment’s garbage bytes
Problem (4) – Excessive small random accesses

**Problem 4:** Excessive small random accesses on PM $\leftrightarrow$ 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

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

Shortcuts
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?

1. 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

③ update (and persist) KV’s reference

Per-object manner:
- one flush and fence between step ② and step ③ for each valid object
- serial flush on each updated reference

old segment
① check if valid
② copy valid KV

fresh segment
Design (2) - Redesign Compaction Pipeline

Background Compaction

Batch manner:

old segment

volatile buffer

fresh segment
Design (2) - Redesign Compaction Pipeline

Background Compaction

Batch manner:

1. copy valid objects to buffer

<table>
<thead>
<tr>
<th>Old segment</th>
<th>Volatile buffer</th>
<th>Fresh segment</th>
</tr>
</thead>
</table>
Design (2) - Redesign Compaction Pipeline

Background Compaction

Batch manner:
- non-temporal stores (higher bandwidth) to copy bulk of data (step ②)

Diagram:
- ① copy valid objects to buffer
- ② NT-copy to PM

(old segment) → volatile buffer → fresh segment
**Design (2) - Redesign Compaction Pipeline**

**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

1. copy valid objects to buffer
2. NT-copy to PM
3. update KVs’ references with shortcuts, persist references in batch

**Background Compaction**
Design (2) - Redesign Compaction Pipeline

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

![Diagram showing DRAM and PM with record buffers, hot segments, and cold segments. BG thread generates a new immutable set, which replaces the old KV set.]

new KV
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

Header  |  Entry Array  |  Index node

K₀  V₀  K₁  V₁  …  Kₙ  Vₙ

old object

DF  SZ  KV

old segment

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
  - Header
  - Entry Array
  - Index node
    - Key: $K_0, V_0, K_1, V_1, \ldots, K_n, V_n$
    - Size (sz) and Address (addr)
      - sz: 16 bit
      - addr: 48 bit

- Embed size (SZ) in the reference
  - Read size (SZ) to increase old segment’s garbage bytes and write old object’s deleted flag (DF)
Design (4) – Reduce excessive PM accesses

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

Foreground Put

Header | Entry Array

Index node

K0 | V0 | K1 | V1 | ... | Kn | Vn

sz | addr

16 bit | 48 bit

DF | SZ | KV

old object

old segment

0 1 0 1

volatile deleted flag bitmap

read size (SZ) to increase old segment’s garbage bytes and write old object’s deleted flag (DF)
Design (4) – Reduce excessive PM accesses

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

**Foreground Put**

- Header
- Entry Array

Index node:

- $K_0$, $V_0$, $K_1$, $V_1$, ..., $K_n$, $V_n$
- $sz$, $addr$

- 16 bit 48 bit

- Old object:
  - DF
  - SZ
  - KV

- Old segment

- Volatile deleted flag bitmap

- Pos = offset / MIN_SIZE
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)

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>18-core Intel Xeon Gold 6240 CPU</td>
</tr>
<tr>
<td>PM</td>
<td>3 * 128 GB Intel Optane DC PMMs</td>
</tr>
<tr>
<td>DRAM</td>
<td>96 GB DDR4 DIMMs</td>
</tr>
</tbody>
</table>

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

(a) FlatStore-H  (b) FlatStore-PH  (c) FlatStore-M  (d) FlatStore-FF  (e) Viper

Throughput (Mops/s)  Capacity Utilization (%)
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 compaction
- also enhances performance at low capacity utilizations
  → reduces small random accesses and saves PM’s limited bandwidth
Sensitivity Analysis

Default settings

<table>
<thead>
<tr>
<th>distribution</th>
<th>capacity utilization</th>
<th># of threads</th>
<th># of cleaners</th>
<th>value size</th>
<th>write ratio</th>
<th># of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zipfian</td>
<td>80%</td>
<td>12</td>
<td>4</td>
<td>48 bytes</td>
<td>50%</td>
<td>200 M</td>
</tr>
</tbody>
</table>

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

Default settings

<table>
<thead>
<tr>
<th>distribution</th>
<th>capacity utilization</th>
<th># of threads</th>
<th># of cleaners</th>
<th>value size</th>
<th>write ratio</th>
<th># of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zipfian</td>
<td>80%</td>
<td>12</td>
<td>4</td>
<td>48 bytes</td>
<td>50%</td>
<td>200 M</td>
</tr>
</tbody>
</table>

(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