ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory

Wonbae Kim¹, Chanyeol Park²,⁵, Dongui Kim³,⁵, Hyeongjun Park⁵, Young-ri Choi¹, Alan Sussman⁴, Beomseok Nam⁵

¹UNIST, ²Naver, ³Line, ⁴University of Maryland, College Park, ⁵Sungkyunkwan University
Indexing/Key-Value Store for Persistent Memory

**Persistent Memory (PM)**
- Non-volatile
- Higher capacity than DRAM
- Lower latency than SSD

**PM-Only Index**
- Slower than DRAM index
- mfence and clflush overhead

**DRAM+PM Hybrid Index/KV-Store**
- Use faster DRAM index
- Avoid use of mfence and clflush
- Periodic synchronous checkpointing
  → **High Tail Latency**
Asynchronous Incremental Checkpointing

- **Log-Structured Merge (LSM) tree**
  checkpoints small DRAM index incrementally and asynchronously.
Write Stall Problem

- LSM-trees suffer from *write stall problem*.
Write Stall Problem

- LSM-trees suffer from **write stall problem**.
Challenges to Avoid Write Stalls

1. Write latency gap between DRAM and PM

2. Write amplification

3. PM is more sensitive to NUMA effects than DRAM
Three Novel Designs of ListDB

1. Write latency gap between DRAM and PM
   • → **Index-Unified Logging** (Convert Logs into SkipLists for Faster Flush)

2. Write amplification
   • → **Zipper Compaction** (In-place Merge Sort)

3. PM is more sensitive to NUMA effects than DRAM
   • → **NUMA-aware Braided SkipList**

- ListDB resolves the write stall problem and shows 25x higher write throughput than Intel Pmem-RocksDB
Design of ListDB: High-Level Architecture

- Three-level architecture: MemTable, Log+L0 PMTable, L1 PMTable
Fast MemTable Flush with Index-Unified Logging

The only difference between WAL and L0 SkipList

- Order of Keys

Let’s union WAL and persistent SkipList

- Pre-allocate pointer space in log entries
- Flush only pointers, not key-values
- No need to call persist instructions when flushing
- Pointers are persisted when merging L0 and L1

With Index-Unified Logging, a MemTable can be flushed before the next one becomes full.
Physical PM Layout for MemTable Write

- Ex) Put 8 → Put 2 → Put 5

Logical View

Physical PM Layout

- Element Pointed by MANIFEST
- Persistent Pointers
- Pointers to be Persisted
Physical PM Layout for MemTable Write

Ex) Put 8 → Put 2 → Put 5

Logical View

Physical PM Layout

: Element Pointed by MANIFEST

: Persistent Pointers

: Pointers to be Persisted
Physical PM Layout for MemTable Write

- Ex) Put 8 → Put 2 → Put 5

**Logical View**
- MemTable
- WAL
- L0 PMTables
- L1 PMTable

**Physical PM Layout**
- Element Pointed by MANIFEST
- Persistent Pointers
- Pointers to be Persisted

Ex) Put 8 → Put 2 → Put 5
Physical PM Layout for MemTable Write

- Ex) Put 8 → Put 2 → Put 5

Logical View

Physical PM Layout

- Element Pointed by MANIFEST
- Persistent Pointers
- Pointers to be Persisted
Physical PM Layout for MemTable Write

- Ex) Put 8 → Put 2 → Put 5

**Logical View**

**Physical PM Layout**

- : Element Pointed by MANIFEST
- : Persistent Pointers
- : Pointers to be Persisted
Physical PM Layout for Flush

- Ex) Flush MemTable

Logical View

Physical PM Layout

- : Element Pointed by MANIFEST
- : Persistent Pointers
- : Pointers to be Persisted

- Pointer → PM Offset → Write

Index-Unified Logging

WAL

MemTable

L0 PMTables

L1 PMTable

DRAM

PM

PM

PM

L0

L1

WAL
Physical PM Layout for Flush

- Ex) Flush MemTable

Logical View

Physical PM Layout

- Pointer → PM Offset → Write

: Element Pointed by MANIFEST

: Persistent Pointers

: Pointers to be Persisted
Physical PM Layout for Flush

- Ex) Flush MemTable

Logical View

Physical PM Layout

PM Tables

Still Log Entries
Physical PM Layout for Flush

- **Ex) Flush MemTable**

**Logical View**

**Physical PM Layout**

- **Pointer → PM Offset → Write**

- **Index-Unified Logging**

- **DRAM**

- **PM**

- **MemTable**

- **L0 PMTables**

- **L1 PMTable**

- **Checkpointing**
Physical PM Layout for Flush

- Ex) Flush MemTable

**Logical View**

**Physical PM Layout**

- Pointer → PM Offset → Write
- Index-Unified Logging
- DRAM
- PM
- MemTable
- L0 PMTables
- L1 PMTable

- : Element Pointed by MANIFEST
- : Persistent Pointers
- : Pointers to be Persisted

Checkpointing
Zipper Compaction

In-place merge-sort $L_0 \rightarrow L_1$

- Only pointers are updated
  - Reduce write amplification

Zipper Compaction does not block concurrent reads

- Every step preserves the SkipList invariant for correct search
- SkipList Invariant: Upper layer list is a sorted sub-list of the bottom layer

Two phases to preserve the invariant

(i) Scan: HEAD $\rightarrow$ TAIL
(ii) Merge: TAIL $\rightarrow$ HEAD
Zipper Compaction: Scan Phase

**Scan Phase:**

- Traverse L0 and L1
- Determine where to insert each L0 element in the L1
- Store the pointer updates on a stack
Zipper Compaction: Merge Phase

Merge Phase:

• Pop and apply

• From TAIL to HEAD (bottom first)

→ Preserves the SkipList invariant

→ Correct search result at every step
Zipper Compaction: Merge Phase

For each update item $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Update the $X_n$ to point to $Y$
Zipper Compaction: Merge Phase

For each update item $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Update the $X_n$ to point to $Y$
For each update item $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Update the $X_n$ to point to $Y$
Zipper Compaction: Merge Phase

For each update item $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Update the $X_n$ to point to $Y$

• $[A, D, E]$ is a sub-list of $[A, D, E, F]$ → The invariant is preserved
For each update item $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Update the $X_n$ to point to $Y$

- $[A, D, E]$ is a sub-list of $[A, D, E, F]$

→ The invariant is preserved
Zipper Compaction: Merge Phase

For each update item $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Update the $X_n$ to point to $Y$

- $[A, D, E]$ is a sub-list of $[A, D, E, F]$  
  $\rightarrow$ The invariant is preserved
- $[B, C, F]$ is a sub-list of $[B, C, E, F]$  
  $\rightarrow$ The invariant is preserved
Zipper Compaction: Merge Phase

For each update item $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Update the $X_n$ to point to $Y$

- $[A, D, E]$ is a sub-list of $[A, D, E, F]$
  $\rightarrow$ The invariant is preserved
- $[B, C, F]$ is a sub-list of $[B, C, E, F]$
  $\rightarrow$ The invariant is preserved

- E appears in both L0 and L1
- But correct search results guaranteed for concurrent reads
Zipper Compaction: Merge Phase

For each update $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Set the $X_n$ to point to $Y$

The Invariant is preserved at every step → Correct search result
Zipper Compaction: Merge Phase

For each update $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Set the $X_n$ to point to $Y$

The Invariant is preserved at every step → Correct search result
Zipper Compaction: Merge Phase

For each update $X_n \rightarrow Y$,
1. Update $Y_n$ to the value of $X_n$
2. Set the $X_n$ to point to $Y$

The Invariant is preserved at every step
→ Correct search result
NUMA-Aware Braided SkipList

Correct Searches are guaranteed as long as the invariant is preserved

**NUMA-aware Braided SkipList:**

- Elements in upper layers are NUMA locally linked
- All elements are connected at the bottom layer

→ **The invariant is preserved**
→ **Correct search results**

![Diagram of Braided SkipList](image)

- Elements in NUMA 0
- Elements in NUMA 1

**NUMA-oblivious SkipList**

* Elements are created by clients on random NUMA nodes
NUMA-Aware Braided SkipList

• Ex) a client on NUMA 0 searches key 7

\[\frac{1}{N}\] remote memory accesses compared to conventional SkipLists (\(N\) is the number of NUMA nodes).
Experimental Setup

**Testbed:** 4-socket machine (4 NUMA nodes)
- CPU: 4 x Intel Xeon Gold 5215 (20 vCPUs per socket)
- DRAM: 256 GB
- PM: 2 TB

**Benchmarks:**
- YCSB
- Facebook Benchmark (modeling-based synthetic workload generator [FAST ’20])

**Evaluation Goal:**
- The effectiveness of 3 designs
- Recovery performance of asynchronous checkpointing
- Comparison with Pmem-RocksDB
Load A - 1 client thread, 1 worker thread

Flush is slower than put

Flush is much faster than put

→ Resolve write stall problem
IUL vs. WAL: YCSB

- 100 million queries after Load 100 million records
- # background workers = ½ of clients

(a) Load A (Write 100%)
(b) Workload A (Read 50% : Write 50%)
(c) Workload C (Read 100%)

Without write stalls, IUL outperforms WAL for write-intensive workloads

IUL performs similarly to WAL for read-only workloads
Effectiveness of Each Design (Enabling one by one)

Load A - 500 million records, 80 threads

(a) WAL
Effectiveness of Each Design (Enabling one by one)

- **Load A** - 500 million records, 80 threads

By **reducing NUMA effects**, flush throughput increased → Put throughput increased
Effectiveness of Each Design (Enabling one by one)

- **Load A** - 500 million records, 80 threads

<table>
<thead>
<tr>
<th>Design</th>
<th>Throughput (Mops/sec)</th>
<th>Elapsed Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) WAL</td>
<td></td>
<td>≈290 sec</td>
</tr>
<tr>
<td>(b) WAL+Braided</td>
<td></td>
<td>≈127 sec</td>
</tr>
<tr>
<td>(c) WAL+Braided+Zipper</td>
<td></td>
<td>≈117 sec</td>
</tr>
</tbody>
</table>

Zipper Compaction makes compaction faster
→ Good for search performance
Effectiveness of Each Design (Enabling one by one)

- Load A - 500 million records, 80 threads

(a) WAL ≅ 290 sec

(b) WAL+Braided ≅ 127 sec

(c) WAL+Braided+Zipper ≅ 117 sec

(d) IUL+Braided+Zipper ≅ 63 sec

**Index-Unified Logging** enables faster flush
→ No write stalls
→ Put throughput significantly increased
Recovery Performance

- Checkpointing and Recovery cost for around 100 million records

**Checkpointing overhead for 100M records**

- Writes will be blocked for more than 90 sec at every interval

- ↑ Checkpoint interval → ↑ Log replay time → ↑ Recovery cost

**ListDB uses Asynchronous Checkpointing** → Zero checkpointing overhead

- Instant recovery

(a) Synchronous Checkpointing

(b) ListDB
Comparison with Pmem-RocksDB

- Facebook Benchmark, 80 threads
  - Query Arrival Rate ~ Sine distribution (5 billion queries in total)
  - put : get = 3 : 7

Throughput is saturated due to Write Amplification

Throughput = Query Arrival Rate

25x higher write throughput
More Experiments in Our Paper

- NUMA effects
- Comparison with PM-only indexes
- Comparison with other LSM-based designs
Conclusion

**ListDB avoids write stalls** leveraging byte-addressability and high-performance of PM

- Avoiding data copies by restructuring data in-place
- Reducing write amplification
- NUMA-aware persistent index structure effectively reduces NUMA effects

With its three-level structure and **three novel designs**, **ListDB outperforms** state-of-the-art PM-based key-value stores **in write throughput**.

- Our code is available at [https://github.com/DICL/listdb](https://github.com/DICL/listdb)
- **Looking for a Job!**
- **Email:** wbkim@unist.ac.kr