Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory

Baoyue Yan¹, Xuntao Cheng², Bo Jiang¹,* , Shibin Chen², Canfang Shang², Jianying Wang², Gui Huang², Xinjun Yang², Wei Cao², Feifei Li²

¹Beihang University and ²AZFT
Background and Challenge

- **LSM-tree based OLTP database**
  - Fast writes, high compression
  - MyRocks, X-engine in RDS

- **Existing problems**
  - Write overheads in sync WAL
  - Data piling in level 0
  - Recovery overheads by volatile data structures

- **New opportunities by PM**
  - Larger capacity (tens of TB)
  - Persistent writes
  - Byte-addressable
  - Lower cost (30% of DRAM)
Background and Challenge

- How to design efficient persistent memtable?
  - High overheads for data persistence
  - Better sequential access for PM
  - Atomicity guarantee

- Can the WAL be removed with PM?
  - Only support 8-byte atomic write
  - With no hardware transaction memory support

- Can the tiered level 0 be replaced?
  - Larger capacity of PM enables a larger memory buffer
  - Redesign level 0 data structure with PM

- How to manage the persistent memory allocation?
  - Efficient memory allocations
  - Memory leaks
  - Memory fragmentations
Design Overview

- **Semi-persistent memtable**
  - Optimized persistent index for memtable
  - High performance and fast recovery

- **Reorder Ring**
  - Log-as-data to avoid WAL
  - Concurrent lock-free transaction commit in PM
  - Batching to reduce random writes

- **Global Index**
  - Persistent index working as level 0
  - Globally sorted to reduce read APM
  - No-blocking writes with in-memory compaction

- **Halloc**
  - PM allocator specifically designed for LSM-tree
  - Pool based object memory pool
  - log-free object allocation
  - hybrid memory management
Design for Semi-persistent Memtable

- Three typical range index designs for PM?
  - Fully persistent - F&F, BzTree
  - Semi-persistent - NVTree, FPTree
  - Non-persistent - for DRAM

- Memtable features
  - MVCC
  - Append-only writes
  - Batch memory reclaim
  - Small memory footprint

- Solutions
  - Keep index nodes volatile
  - Stores only pointers to record in PM
  - batch write to reduce write AMP
  - ART + OLC + EBR

<table>
<thead>
<tr>
<th></th>
<th>Full</th>
<th>Semi</th>
<th>Non</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>low</td>
<td>middle</td>
<td>high</td>
</tr>
<tr>
<td>Recovery</td>
<td>fast</td>
<td>middle</td>
<td>no</td>
</tr>
</tbody>
</table>

Optimized for MVCC
Batch writes
Batch reclaim by Halloc
Append-only
runtime key parsing
Design for Reorder Ring

- **Requirements**
  - Atomic writes for multiple KVs to one memtable
  - Atomic persistence for OLTP transactions
  - Performance scales for multi-core platform

- **Design choices**
  - Volatile indexes in memtables
  - Append-only writes in memtables

- **Existing solutions**
  - Log-as-data
  - Build indexes over logs

- **Limitation**
  - Life cycle differences
  - Sequential writes
Design for Reorder Ring

- **Distinguish life cycle differences**
  - Independent memory regions for LSM-trees
  - May not guarantee atomicity

- **Solutions**
  - 8-byte gidx pointer
  - Store current pos and previous pos
  - Store LSN and number of heads to update

One transaction may cover multiple heads, which cannot guarantee atomicity

- Solutions
  - 8-byte gidx pointer
  - Store current pos and previous pos
  - Store LSN and number of heads to update

ChainLog item

ChainLog item
Design for Reorder Ring

- Enable concurrent persistence
  - ChainLog requires sequential persistence
  - PM has better performance of sequential writes

- Solutions
  - Batching to merge small transaction buffers
  - Concurrent ring to enable parallel persistence
  - Core principle: write ahead

- Why “Reorder”? 
  - Get ChainLog descriptors -- serial
  - Parallel buffer initialization -- parallel
  - Grant memory space -- serial
  - Parallel persistence -- parallel
  - No-blocking commit -- serial

One leader with multiple followers
Design for Global Index

● Requirements
  • A globally sorted level 0 in PM
  • No need of transactional writes for multiple records
  • Non-blocking while compacting to disk

● Solutions
  • Introduce Global Index (GI) as new level 0
  • Do not directly store records
  • In-memory compactions to merge memtable into GI
  • Snapshot to enable foreground merge while compaction

● Implementation
  • SoTA persistent range indexes without extra transactional support
  • Volatile range indexes to give a better performance
Design for Halloc

- Specifically designed for LSM-tree
  - Append-only memory allocation
  - Batch memory reclaim
  - Pre-formatted for LSM-tree

- Persistent object pool
  - Log-free persistent freelist
  - Automatic memory leak detection and reclaim
  - Space reservation to reduce fragmentation

- Hybrid memory management
  - Fix-size zone object as minimal unit to allocate
  - Append-only writes for zone object in memtable
  - Delegated to jemalloc for volatile purpose
Evaluation

- **Hardware configuration**
  - Xeon(R) Platinum 8269CY CPUs 104 cores
  - 187GB DRAM
  - 1TB PM
  - 2TB ESSD, 2GB R/W bandwidth, 200K random 4KB
  - Linux kernel 4.19.81

- **Overall benchmarks**
  - YCSB
  - TPCC
  - Run for 30 minutes
Evaluation for Overall Performance

- **YCSB configuration**
  - 8-byte key, 500-byte value
  - 800 million KV items, 16 tables, 500GB dataset
  - 32 client threads

- **TPCC configuration**
  - Storage plugin for MYSQL server
  - 80GB initialized dataset
  - 64 client threads

Graphs showing throughput and latency comparisons for different configurations.
Conclusion

- Semi-persistent memtable significantly outperforms baselines.
- Reorder Ring enables high-performance atomic log-free transactions in PM.
- Global Index largely reduces the read AMP compared with the in-disk one.
- Halloc provides efficient PM allocation specifically for LSM-tree.
- The overall performance improves the baselines by 3.8x in YCSB and 2.0x in TPCC.
Evaluation for Semi-persistent Memtable

- How does the semi-persistent memtable perform compared with SoTA indexes?
- Memtable configuration
  - 8-byte key, 32-byte value, 50 million items
  - 2GB for single memtable
- Comparisons
  - SLM: Skiplist in DRAM
  - FFM: FAST&FAIR in PM (full)
  - FPM: FPTree in PM (semi)
  - SPM-D: Semi- with indexes in DRAM
  - SPM-P: Semi- with indexes in volatile PM
- Results
  - Random insert by up to 8.3x
  - Sequential insert by up to 6.0x
  - Random point lookup by 16x
  - Random scan slower than Skiplist
  - Fast recovery about 3s for 32GB memtable

Significantly improved
Evaluation for Reorder Ring

- Is the ROR algorithm efficient enough? Yes

- **Configuration**
  - 24-byte KV items
  - 1 million items per thread

- **Impact of batch size (thrd=32)**
  - Higher throughput with longer latency
  - Hardware saturated with batch=90

- **Impact of thread number (batch=50)**
  - More threads, higher throughput, longer latency
  - Hardware saturated with thrd=16
  - P99 latency less than 200us
Evaluation for Global Index

● Does the global Index significantly outperform the in-disk one? Yes

● Configuration
  • Disable compactions for L1,L2,…
  • All data written into memtable and level 0
  • Random read/write with 1:1

● Results
  • Significantly reduce the impact of data piling for original level 0
  • Performance drops less than 35% even for 343GB level 0
Evaluation for Halloc

- How does the Halloc perform compared with SoTA general PM allocators?
  - Significantly improved

**Configuration**
- Average latency for 1 million allocations
- Persist each object into a persistent list
- halloc-pool: allocate object from pool
- halloc-zone: grant memory space for memtable

**Results**
- Less than 1us for all cases with Halloc
- General PM allocators are expensive for large allocation