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

Baoyue Yan[1], Xuntao Cheng[2], Bo Jiang[1,*], Shibin Chen[2], Canfang Shang[2], Jianying Wang[2], Gui Huang[2], Xinjun Yang[2], Wei Cao[2], Feifei Li[2]
[1]Beihang University and [2]AZFT
{beyuer.yan,chengxuntao,gongbell,chenshibin,shangcanfang,jianyingse}@gmail.com
{kenryhuang,yangjimmy,cyg.cao,ricosfeifei}@gmail.com

ABSTRACT

The recent byte-addressable and large-capacity commercialized persistent memory (PM) is promising to drive database as a service (DBaaS) into unchartered territories. This paper investigates how to leverage PMs to revisit the conventional LSM-tree based OLTP storage engines designed for DRAM-SSD hierarchy for DBaaS instances. Specifically, we (1) propose a light-weight PM allocator named Halloc customized for LSM-tree, (2) build a high-performance Semi-persistent Memtable utilizing the persistent in-memory writes of PM, (3) design a concurrent commit algorithm named Reorder Ring to aschieve log-free transaction processing for OLTP workloads and (4) present a Global Index as the new globally sorted persistent level with non-blocking in-memory compaction. The design of Reorder Ring and Semi-persistent Memtable achieves fast writes without synchronized logging overheads and achieves near instant recovery time. Moreover, the design of Semi-persistent Memtable and Global Index with in-memory compaction enables the byte-addressable persistent levels in PM, which significantly reduces the read and write amplification as well as the background compaction overheads. The overall evaluation shows that the performance of our proposal over PM-SSD hierarchy outperforms the baseline by up to 3.8x in YCSB benchmark and by 2x in TPC-C benchmark.

PVLDB Reference Format:
Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, Feifei Li. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. PVLDB, 14(10): 1872-1885, 2021.
doi:10.14778/3467861.3467875

1 INTRODUCTION

Recent commercialized persistent memory (PM) products have significant potentials in driving database as a service (DBaaS) into uncharted territories. Compared with the mainstream volatile DRAMs (usually 8GB to 32GB) used in DBaaS instances, PMs can be much larger (hundreds of GBs or larger) and persistent at an economically viable cost with the same byte-addressability[2]. Currently, many DBaaS systems for OLTP workloads rely on synchronized logging for durable transactions. The precious DRAM capacity forces the database to flush dirty pages frequently at the cost of a stable performance, or give up processing analytical queries with large memory footprints. With the PMs, we are now able to revisit how the DBaaS, especially the underlying storage engines, see and utilize the main memory. For example, Intel 3D XPoint memory [1] is a byte-addressable, large-capacity, and persistent main memory compared with DRAMs. In this paper, we exploit it to achieve persistent in-memory writes with a competitive level of overall performance to pave the way for more possibilities in the future.

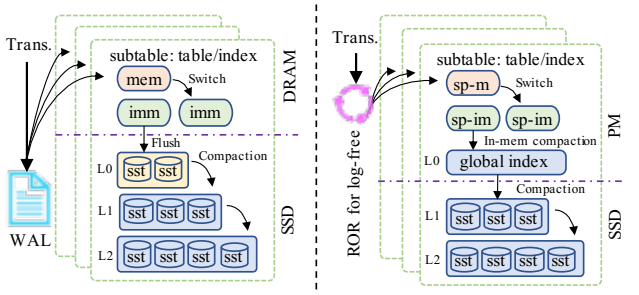

Figure 1: The state-of-the-art LSM-tree based OLTP storage engine (the left one) v.s. our proposal (the right one).

As a starting point, we base our work on popular LSM-tree [42] based key-value storage engines. Such engines have been widely used in DBaaS for various workloads [26, 39]. For now, these engines are deployed on the conventional DRAM-SSD storage hierarchy. Although the LSM-tree data structure itself achieves fast writes through append-only inserts, the synchronized logging, which is necessary for ACID-compliant transactions, drags slow disk I/Os into the write pipeline and constraints the final write throughput. Similarly, periodically merging the in-memory deltas with the base on SSDs in LSM-tree is necessary and expensive in terms of CPUs and I/Os consumed. These issues have been increasingly troublesome in the public cloud, where databases are usually deployed on multi-tenant virtualized machines and customers pay for transaction/query processing, not for expensive background operations.

Specifically, LSM-tree based OLTP engines on DRAM-SSDs have the following issues. Firstly, appending and accumulating changes in memtables (indexed in-memory buffers, shown as "mem" or "imm" in Figure 1) increases the workload for crash recoveries and disk space usages for logs, although accessing memtables for transactions/queries has low latencies[25]. In extreme cases (e.g., replaying huge WALs in a small instance), it may take hours for a database instance to recover in the cloud, which affects the overall system availability. Secondly, the tiering compactions [6, 38] for the first level (L_0) delivers a fast flush [26] while trading off query latencies, because there can be data blocks (flushed from memtables)

*Bo Jiang is the corresponding author.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 10 ISSN 2150-8097.
doi:10.14778/3467861.3467875

with overlapping key ranges in the L_0. Compaction operations have to be frequently called to merge such blocks, which may even stall transaction/query processing in the worst case[18, 19, 54]. Thirdly, memtables compete with other in-memory data structures such as caches and indexes for memory spaces. It requires careful tuning of these structures for performance guarantees, which in turns prevents cloud customers from using the database with ease.

With PMs, we are now able to address these issues in this work. We firstly make the memtable persistent in PM (denoted as "spm" in Figure 1), thereby extra logs and flushes into the SSD for memtables are no longer needed. Then we move and redesign the originally disk-resident L_0 of LSM-tree into the byte-addressable PM (denoted as "global index" in Figure 1), and replace the original I/O-expensive compactions for L_0 with in-memory ones. Compared with the state of art solutions, this design reduces the CPUs and I/Os consumed by background compactions. Thirdly, although the large capacity of the PM can be utilized to reduce memory contentions among memtables, caches and indexes, we find that existing PMs support only 8-byte atomicity write and require expensive flushes and fences to guarantee persistence [35, 53], causing no-trivial challenges on the memory allocation, the design of persistent indexes and the guarantee of atomic multi-words writes in ACID-compliant transactions for OLTP workloads. To address these challenge, we make the following contributions:

- We design a **light-weight and efficient memory allocator of PM, Halloc,** for LSM-tree based OLTP storage engine. Halloc employs the log-free pool-based memory allocation, and supports both persistent and volatile memory allocations.
- We design a **Semi-persistent Memtable structure**, separating index nodes from the data nodes and storing them in volatile and persistent memory spaces, respectively. The persistent data nodes employ fast sequential writes. And the recovery of the small-size index nodes are very fast.
- We propose a **log-free transaction commit algorithm, Reorder Ring (ROR)**, to achieve log-free persistent transaction processing in PMs. ROR guarantees log-free multi-words atomic persistence in PM and enables a pipelined commit protocol to improve the scalability in multi-core platforms.
- We introduce a **globally sorted level in PM, Global Index**, as the new L_0 for LSM-tree to maintain a large persistent level with non-blocking in-memory compaction. The memtables are merged into the Global Index with in-memory compaction by the pointer operations, thus no data copies occur for records in PM.

The overall evaluation shows that our design over PM-SSD storage achieves up to 3.8x performance improvement compared with the baseline over DRAM-SSD storage for write-intensive workload on YCSB[17] benchmark and 2x performance improvement in TPC-C benchmark. More importantly, our proposal achieves competitive performance with slower PM compared with DRAM while keeping near instant recovery time. To the best of our knowledge, our solution is the first attempt to shift the design from DRAM to PM for LSM-tree based OLTP storage engines with a specifically designed PM allocator, and is also the first one to entirely eliminate logs off the system critical writing path both for PM and disk.

The rest of the paper is organized as follows. Section 2 discusses background and challenges in details. Section 3 describes the design of Halloc, ROR, Semi-persistent Memtable and Global Index. Section 4 presents the evaluation of our work compared with existing design proposals as well as performance analysis. Section 5 summarizes the related work and section 6 concludes the paper.

2 BACKGROUND AND CHALLENGES

In the section, we discuss the common design of the LSM-tree based KV stores, the features of persistent memory and the challenges of the design shift considering persistent memory in OLTP workloads.

2.1 LSM-tree Based OLTP Engines

As shown in the left part of Figure 1, the typical LSM-tree based OLTP storage engine employs multiple LSM-tree instances in a database, each of which is used to store a table, or a partition of a table, or an index (all referred as a subtable in this paper), along with the WAL to achieve the ACID-compliant transactions. Each LSM-tree instance buffers updates or inserts in fast DRAM by append-only method and persists data sequentially as a sorted run to disk with multi-level merge tree. The core principle for LSM-tree is to apply updates out-of-place and use sequential disk accesses to avoid random I/Os in disk. Since the inserted KV pairs are firstly buffered in volatile memtable, the WAL (Write-Ahead Log[40]) shall be written to persistent storage for durability guarantee. Current LSM-tree implementations such as RocksDB[39] and X-Engine[26] also employ the WAL as transaction logs to guarantee durability. When system crashes, the WAL is replayed to bring the system into a consistent state. Larger WAL size leads to longer recovery time. As a result, the memtable must be frequently flushed to disk to purge the WAL.

Data in disk normally consists of exponentially increased levels $(L_0, L_1, \ldots, L_n)$ with a fixed capacity ratio $T (T \geq 2)$. Data levels in disk are usually organized by partitioned SSTables (Sorted String Table) and slowly trickled down the levels by background compactions. Since levels normally contain disjoint sets of keys, a read operation shall visit these levels in top-down manner. Compactions are performed to purge stale values and apply the deletions in disk levels, which are much heavier than flushes and may block the flushes or cause data block piling in the low level. As a result, modern LSM-tree based OLTP engines designed for DRAM-SSDs normally employ tiering compactions for L_0 to deliver fast flushes as well as reduce compaction overheads. However, the design results in unordered data blocks in L_0 especially in case of the write-heavy workloads, which brings higher read amplification[6, 24, 26, 54].

2.2 Persistent Memory

The byte-addressable persistent memory[1, 20, 41, 53] enables CPUs to access data directly with load and store instructions while providing persistence. The new hardware provides lower power consumption and cost as well as larger capacity than DRAM[7, 21, 29, 46, 48]. Unfortunately, it is non-trivial to make use of such devices because of the large differences in programing model between PM and DRAM. As shown for intel DCPMM (a kind of commercialized PM) in Figure 2, the majority of the data writing path from CPU registers to DCPMM controller are volatile. Intel enables ADR

(Asynchronous DRAM Refresh) feature to guarantee persistence for CPU stores reaching the ADR domain. Furthermore, modern CPUs employ complex out-of-order execution, so persistent instructions shall be explicitly and expensively ordered to guarantee consistency[15, 28, 43]. Current intel ISA provides *flush* instructions (*clflush*, *clflushopt*, *clwb*) to flush data in cache to persistent memory, along with the *ntstore* to bypass cache, and a *fence* instruction to ensure that previous stores are persisted with correct order.

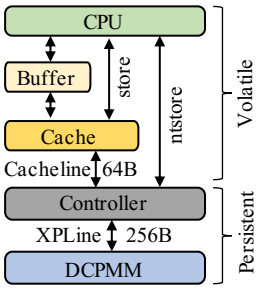

Figure 2: Data path.

For Intel DCPMM (Figure 2), the atomic persistent store is in 8 bytes. Therefore, instructions that store more than 8 bytes may be torn in power outage. The communication between the CPU and the DCPMM controller is in cache line granularity(64-bytes), but DCPMM media access granularity is 256 bytes. As a result, a small access request will be translated into the large 256-bytes access request, leading to read/write amplification. Our experiment also confirms that random writes with size less than 256 bytes have similar performance in PM. Fortunately, memory controller for DCPMM employ a small combining buffer to alleviate the problem by merging adjacent writes, which significantly benefits sequential memory writes[53]. DCPMM supports two operation modes: Memory mode for DRAM extensions without persistence and App Direct mode functioned as DAX persistent devices. In this paper we focus on the App Direct mode to enable persistence.

2.3 Challenges

We now discuss the challenges in detail about the allocation of PM memory, the design of persistent indexes and the guarantee of atomic multi-words writes in ACID-compliant transactions for LSM-tree based engines with PMs.

Expensive General Purpose PM Allocators. Current general purpose PM allocators consider both random allocations and deallocations, which results in very fragmented memory allocation scheme and expensive cacheline flushes and fences. Our experiment with the same configuration in Section 4 shows that the performance of popular PM allocators (PMDK[5] and Ralloc[11]) fail to scale especially for large object allocation. Even for small object allocation, e.g., less than 100 bytes, their performance are still orders of magnitude slower than jemalloc[22] in DRAM. Moreover, our new requirement to reduce DRAM footprint relies on both volatile and persistent memory management. However, current volatile PM allocators[12] are normally designed as an extra volatile PM pool that requires exclusive memory space, which leads to large memory consumption.

Expensive Persistent Range Indexes. Introducing a persistent index in memtable becomes a common approach to reduce WAL overhead[30]. However, the updates for index nodes in PM normally covers multiple words and results in multiple small random writings, which is expensive in PM[14, 37, 53]. And updates for multiple words introduce logging or other costly approaches[13, 16, 37, 49]

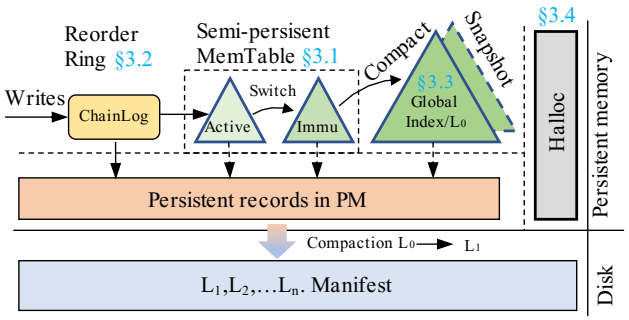

Figure 3: System architecture for our solution, where all persistent memory is managed by Halloc.

to guarantee the atomicity[5, 47]. We have also tested the state-of-the-art B+tree FAST&FAIR[27] that is designed specifically for PM. We observe that its performance is only 10% of the raw throughput of Optane DCPMM hardware. These observations tell us that maintaining a persistent range index in memtable is expensive.

Indispensable Transaction Logs. The transaction log is normally encapsulated in WAL for LSM-tree based OLTP storage engines[40], which invalidates the assumption that introducing a persistent memtable can eliminate the WAL overhead[30, 31, 36]. The transaction mechanism implies the atomic durable batch operations over a group of KVs. And it leads to extra cost of logging and expensive cacheline flushes and fences for maintaining WAL in LSM-tree based OLTP storage engines. Wang et al.[48, 50] points out that transaction logs incur up to 35% performance overheads in PMDK[5]. As a result, these approaches still suffer from the large overhead incurred by WAL for OLTP workloads.

3 DESIGN

We proposes four key techniques to utilize PM to address the challenges while achieving high performance (shown in Figure 3). This includes the Semi-persistent Memtable(§3.1), the Reorder Ring with ChainLog(§3.2) to enable log-free transactions, the Global Index(§3.3) to maintain a large globally sorted persistent level in PM as the L_0 in LSM-tree, and the specifically designed PM allocator named Halloc(§3.4),. The other levels ($L_1, \ldots, L_n$) are kept unchanged in SSD. Note that we still use DRAM in application runtime for block/row cache and volatile indexes.

As shown in Figure 3, the updated entries are firstly batched and queued by reorder ring, and then inserted into the active semi-persistent memtable. When the active memtable is full, it is switched to immutable state and compacted into the proposed L_0 by the light-weight in-memory compaction. Since the proposed L_0 guarantees the persistence in PM, no flushes to SSD are needed. When the L_0 is full, its immutable snapshot is created and compacted into L_1 in SSD without blocking the foreground writes. A lookup finds the most recent version of the given key by probing the semi-persistent memtable, the proposed L_0 and other levels in SSD, and terminates when the target key matches.

The design brings three significant benefits for LSM-tree based storage engines: (1) we avoid the overheads of both WAL and additional logging by PM programming libraries and achieve fast recovery by reorder ring and semi-persistent memtable; (2) the

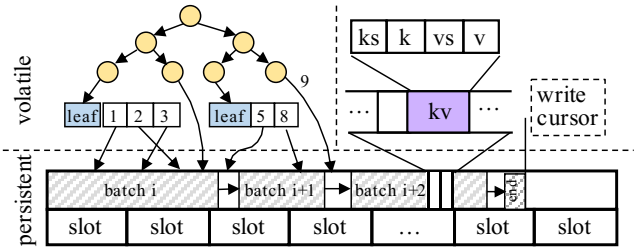

Figure 4: The structure of semi-persistent memtable, *ks*: key size, *k*: key, *vs*: value size and *v*: value.

semi-persistent memtable and proposed L_0 guarantee the persistence in PM thereby no flushes to SSD are needed. Therefore, we can significantly reduce the amount of data written to SSD as well as background flush overheads; (3) the proposed L_0 is globally sorted, so the read amplification for L_0 is reduced and no heavy background compaction for L_0 is needed when compared with the conventional LSM-tree based engines for DRAM-SSD storage architecture (e.g., RocksDB, LevelDB and X-Engine).

3.1 Semi-persistent Memtable

Updates for persistent indexes normally covers multiple words and results in multiple small random writes, which leads to write amplification and overheads to guarantee the consistency of index nodes in PM. We propose the semi-persistent memtable, which employs two optimizations to address the problem.

Keeping Index Nodes Volatile. We keep the index nodes inside memtables volatile. The design is inspired by the observation that industry LSM-tree based OLTP engines do not keep a very large single memtable (normally 64-256MB) because of two reasons. Firstly, we find that cloud users often purchase small database instances with only 8-32GB main memories each, which satisfy their performance requirements at an acceptable cost. Secondly, keeping a small memtable reduces the data size per flush that helps amortizing the I/O consumptions. Given a 256MB memtable, our experiment shows that rebuilding volatile index nodes by scanning it in PM takes less than 10ms. This performance during recovery is fast enough, so that we can tolerate volatile index nodes. And, expensive persistent range indexes are not required and the range index can be kept volatile. In this paper, we adopt ART[33] with optimistic lock coupling (OLC)[34] and Epoch Based Reclamation (EBR)[23] as the volatile range index because of ART's good cache locality and fast prefix matching[51]. Specifically, the KV pair with one version value(e.g., version 9 in Figure 4) is directly stored with 8-byte pointer in index node. Multiple multi-versioned values (e.g., leaf nodes with version 1,2,3 and 5,8 in Figure 4) for a key are stored into a sorted array, the pointers of which are attached to a leaf node in ART. The record in PM is organized by encapsulating the key and value together with built-in metadata. The keys are not stored into index nodes, because the index is volatile and is rebuilt by scanning the keys in PM.

Batching to Reduce Write Amplification. Random writes are natively processed as seperate 256-byte accesses in the PM, causing write amplifications. To avoid this, we batch small writes into a large *WriteBatch*, and flush it to the PM sequentially as a whole. Each batch is persisted after issuing one fence. And, these batches in a memtable are logically linked together to enable correct recovery

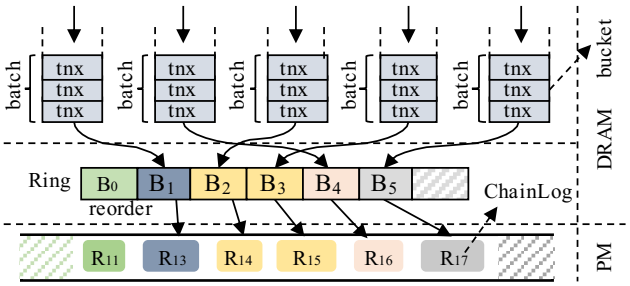

Figure 5: The architecture for Reorder Ring.

(the *batch* shown in Figure 4, where the slot is used to store 8-byte zone id from Halloc and the zone is a large fixed-size PM memory chunk). We export the batch size here as a tunning knob that affects the trade-off between write latency and throughput.

3.2 Reorder Ring

The Reorder Ring(ROR) is a concurrent ring based transaction log-free commit algorithm for persistent transaction processing in PMs, compared with conventional approaches that write extra transaction logs to guarantee atomicity for transactions in OLTP workloads. To achieve the goal, ROR employs three key techniques: ChainLog, batching and concurrent ring. As shown in Figure 5, the ChainLog guarantees log-free atomic multi-words writes to PM. The batching is used to merge small transaction buffers into large ones to avoid small random writes to the PM. And the array-based concurrent ring enables concurrent persistence by reordering ChainLog items to improve multi-core scalability.

3.2.1 ChainLog. The ChainLog is a linked persistent data structure to achieve atomic multi-word writes for transactions in OLTP workloads without extra logging overheads in PM, as shown in the example of Figure 7. That is the reason why we name it as ChainLog, i.e., chained log. Specifically, a group of transaction buffers (in DRAM, Figure 5) are firstly batched as $B_i = \{b_1, b_2, \cdots, b_k\}$ where the b_k are the records to write into memtable m_k. The B_i performs **grant** operation from the memtable set $M = \{m_1, m_2, \cdots, m_n\}$ to get a ChainLog $R_i = \{r_{i1}, r_{i2}, \cdots, r_{ik}\}$ where $k \leq n$, before it is persisted. Each $r_{ik} \in R_i$ is associated to one unique $m_j \in M$. All writes to r_{ik} from m_j are append-only. The R_i should satisfy the following two conditions:

(1) **Atomicity.** The R_i is atomically persisted iff $\forall r_{ik} \in R_i$ are atomically persisted.
(2) **Monotonicity.** If R_i is granted before R_j, then R_i shall be persisted before R_j.

The satisfaction of the first condition guarantees that a batch is atomically persisted. And the satisfaction of the second one ensures that all successfully persistent batches are visible after a system crash to simplify recovery. If batches are persisted out of order, we can not distinguish which one is inconsistent after a crash except to check all batches. Note that we do not need to consider the atomicity for indexes in memtable m_j since they are volatile. Whenever the R_j is atomically persisted, the indexes of it is constructed to m_j.

Data Structure of ChainLog. The data structures of R_i and other necessary structures are defined below.

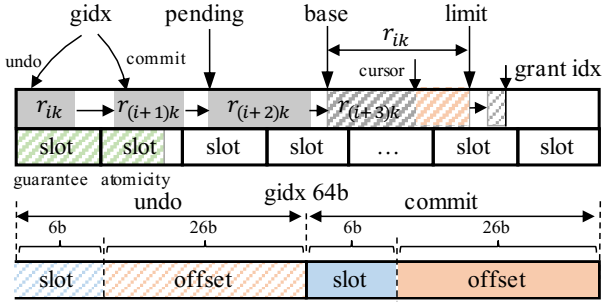

Figure 6: The structure of memory space allocated for the memtable. The *gidx* is used to record the current commit point and the previous one, each of which takes up 4 bytes and manages 4GB memory space at most.

- $R_i.seq$ - The sequence number of R_i. All $r_{ik} \in R_i$ have the same sequence number which persisted in PM. The sequence number is monotonically increased.
- $r_{ik}.base, cursor, limit$ - A write to r_{ik} shall begin from the *base* position, where $cursor \in [base, limt)$, shown in Figure 6. All of them are volatile.
- $r_{ik}.seq$ - Equal to $R_i.seq$. The *seq* is persisted and used for consistency checking in recovery time.
- $m_j.gidx, pending$ - The *gidx* is a persistent and monotonically increased commit position for m_j, where memory space $[0, m_j.gidx]$ is guaranteed to be atomically persisted, shown in Figure 6. The *pending* records the next r_k to be persisted.
- *commit_id* - The last commit sequence of ChainLog. It is used to perform recovery, where all ChainLog items with sequences larger than the value are dropped in recovery.

Atomicity Guaranteeing. A commit for R_i entails all commits for $r_{ik} \in R_i$. Considering a commit operation of R_i, each $r_{ik} \in R_i$ is associated with $m_j \in M$ and the $m_j.gidx$ is atomically updated (8-byte atomicity supported by hardware) before the *commit_id* is updated. If a system crash happens before the updates of *commit_id*, then all committed *gidx* will be rolled back by simply right-shifting to extract the previous r_{ik} from *undo* field. Figure 7 shows an example with R_2 and R_3, where the R_2 has been successfully committed and the R_3 is the next item to commit. In this example, we assume a torn write where a crash hits before $m_3.gidx$ is updated but the $m_1.gidx$ and $m_2.gidx$ have been successfully updated. Since the last commit id $commit\_id = 2$ is not updated, the $m_1.gidx$ is rolled back by right-shifting in recovery.

Recovery. In memtable m_j, each r_{ik} has a *next* field to link the next $r_{(i+1)k}$ (Figure 6) All of them are persisted in PM. For ChainLog, all $m_j \in M$ can be recovered parallelly by scanning the r_{ik} list for each m_j to rebuild index nodes since there are no order constraints between m_j in rebuilding indexes. The last r_{ik} indicated by $m_i.gidx$ is fetched for each m_i; the $r_{ik}.seq$ is checked whether it is larger than *commit_id*, where the last r_{ik} with torn write is rolled back by simply right-shifting the $m_i.gidx$; an index rebuilding task is constructed for each m_j and these tasks can be processed parallelly. The example in Figure 7 shows that the $m_1.gdx$ is rolled back and both m_2 and m_3 keep unchanged.

3.2.2 Batching. We use batching to merge small ChainLog items thereby avoiding the small random writes to PM. A transaction

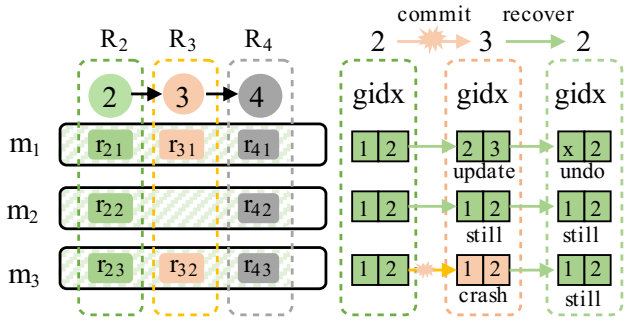

Figure 7: An example where a crash hits in the commit of R_3.

is firstly encapsulated as a *WriteBatch* then several *WriteBatch*es are batched as a ChainLog item to be persisted in PM. We follow the original design for transaction isolation like RocksDB, where the 2PL and the MVCC are adopted. As illustrated in Figure 5, the system uses fixed-size current buckets, each of which has one leader thread to perform writing to PM. When a client thread commits a *WriteBatch*, it firstly finds a bucket and checks whether it is a leader or follower. A follower delegates its *WriteBatch* to the leader. And the leader waits a tunable time then assembles multiple *WriteBatch*es as an enlarged batch. The batch size is designed as a tunable parameter in our system.

3.2.3 Concurrent Ring. In ChainLog, the R_i can not be persisted unless the all previous ones has been persisted (**Monotonicity**). However, the serial persistence exposes scalability bottlenecks on multi-core platforms. To overcome the problem, ROR enables pipelined writes for R_i based on the ring[45]. The basic lock-free ring has a head to indicate the write position and a tail to indicate the read position. We add a send index (sid) and another head index (hid) to enable lock-free ChainLog initialization and lock-free commit. The concurrency is implemented by buffering multiple writes of R_i directly in PM and performing pipelined persistence.

Data Structures of Concurrent Ring. The ROR is based on the concurrent ring. The data structures are listed below.

- *ROR.bid* - The buffer index. A **grant** call allocates and then initializes a ChainLog buffer from ROR ring.
- *ROR.sid* - The sending index. After the initializing the buffer id, the ROR updates the *sid* and grant memory space for R_i.
- *ROR.hid* - The head index - The *hid* is introduced to guarantee that the R_i is always persisted serially.
- *ROR.pid* - The processing index. The processing index indicates the first log item to persist.
- *ROR.cap* - The max capacity of the ring in ROR. The data field limits the maximum number of concurrent R_i to persist.
- *ROR.grants* - The fixed size array of ChainLog descriptors.

The *bid* and *sid* are used to allocate a R_i handle concurrently from the ring of ROR while the *hid* and *pid* guarantees wait-free serial order for the persistence of R_i.

Pipeline Stages. To write a ChainLog item, the thread shall perform the following steps serially: ① gets a handle for ChainLog item R_i from ring; ② initializes the ChainLog descriptor for the handle; ③ grants memory space for R_i; ④ writes R_i to PM; ⑤ finishes the write of R_i. Getting handles from ring between threads shall be

Algorithm 1: Perform concurrent **grant**

```
1  Function grant(B_i, M):
       // Stage 1
2      bid ← fetch_inc(ROR.bid)
3      cap ← ROR.cap
4      busy_wait(bid − ROR.pid ≥ cap − 1)
5      R_i ← ROR.grants[bid%cap]
6      init_fill(R_i, bid)
       // Stage 2
7      busy_wait(ROR.sid ≠ bid)
8      R_i.seq ← grant_id + 1
9      grant_id ← grant_id + B_i.count
10     foreach b_k ∈ B_i do
11         R_i.r_ik ← grant_memory_space(M, b_k)
12     end
13     ROR.grants[(bid + 1)%cap].flag ← false
14     FETCH_INC(ROR.sid)
15     return R_i
16 end
```

Algorithm 2: Perform concurrent **finish**

```
1  Function grant_finish(R_i, cb, ctx):
       // Stage 3
2      bid ← R_i.bid
3      go_next ← true
4      R_i.(cb, ctx) ← (cb, ctx)
5      persist(r_ik) foreach r_ik ∈ R_i
6      if CAS(&ROR.hid, bid, bid + 1) then
7          while go_next do
8              do_commit(&ROR.grants[bid%ROR.cap], bid)
9              bid ← bid + 1
10             go_next ←
                 LOAD(ROR.grants[bid%ROR.cap].flag) &&
                 CAS(&ROR.hid, bid, bid + 1)
11         end
12     else
13         STORE(ROR.grants[bid%cap].flag, true)
14         if CAS(&ROR.hid, bid, bid + 1) then
15             while go_next do
16                 do_commit(&ROR.grants[bid%ROR.cap], bid)
17                 bid ← bid + 1
18                 go_next ←
                     LOAD(ROR.grants[bid%ROR.cap].flag)
                     && CAS(&ROR.hid, bid, bid + 1)
19             end
20         end
21     end
22 end
```

serial because of the constraint of concurrent ring. We denote the serial step as stage 1 (Alg. 1, stage 1). Granting memory space from memtables for multiple threads shall be serial since the memtable forces the append-only writes. We denote the serial step as stage 2 (Alg. 1, stage 2). ChainLog items shall follow the **Monotonicity** condition, therefore threads shall serially complete the writes of R_i. We denote the serial step as stage 3 (Alg. 2, stage 3). When threads complete writes in the same time in Alg. 2, we have to resolve the order between threads to force a serial persistence. We add an extra serial stage (Alg. 2, stage 4) to address the problem. Both Step ② and step ④ between threads can be parallelly performed since there are no order constrains for the two steps. However, when threads come to the serial stages, we have to reorder them to guarantee that they are processed in correct order. For example, if thread *a* gets the ChainLog handle before thread *b*, thread *a* should perform step ③ before thread *b* and also perform ⑤ before thread *b*. That is the reason why we call the method Reorder Ring.

Concurrent Grant. The **grant** operation is performed by concurrent ring. As shown in Alg. 1, a buffer descriptor of log item is firstly allocated from the ring (Lines 2-4). The allocation is blocked by busy waiting if the ring is overloaded (Line 4). Then, a successfully allocated R_i is initialized (Line 6) with id *bid*. Then it is pushed into the ring in the allocated order (Lines 7-12), where the memory space is granted for each r_{ik} (Lines 10-12). Finally, the terminator is inserted to the next buffer to give a termination hint for concurrent committing (Line 13).

Concurrent Finish. When the writing of a log item R_i is finished, the thread explicitly calls **grant_finish** (Alg. 2) to tell ROR that the item R_i can be safely persisted. The thread firstly persist each $r_{ik} \in R_i$ to into persistent memory (Line 5). The ROR performs batched flushing and requires only one sfence to persist each batch. Thus it achieves higher throughput than the approaches employing the state-of-the-art range indexes as their memtables. After R_i is safely persisted, the thread checks whether all previous log items have been persisted by an atomic compare-and-swap(CAS) instruction (Line 6); If the R_i satisfies the condition of commit, the thread will perform the actual commit (Line 8) and try to commit the next R_{i+1} (Lines 9-10) by atomic instructions; If R_i can not be persisted due to the fact that previous log items have not been persisted, the thread will mark R_i that it can be committed and check again (Lines 13-14); Otherwise it will immediately leave since the R_i will be committed by other thread. Note that if the R_i fails to commit, then all other log items granted before R_i will be entirely purged as the persistence shall satisfy the **Monotonicity** constraint.

Perform Commit. In case that two adjacent log items are checked (Line 14 of Alg. 2) simultaneously by two or more threads, the thread shall check the consistency between *ROR.pid* and R_i.*pid* (Line 2 of Alg. 3). After that, the thread updates the *gidx* for each $r_{ik} \in R_i$ and waits for persistence (Lines 3-9). Then, the *commit_id* is updated to claim that the log item R_i is persisted (Lines 10-11). A user callback is called finally if it exists (Lines 12-14).

3.3 Global Index

The Global Index (GI) is an indexed data structure to maintain a globally sorted L_0 in PM for LSM-tree. Modern LSM-tree based OLTP engines designed for DRAM-SSD storage normally employ

Algorithm 3: Do serial commit

```
1  Function do_commit(R_i, pid):
       // Stage 4
2      busy_wait(ROR.pid ≠ pid)
3      foreach r_ik ∈ R_i do
4          new_gidx ← r_ik.gidx << 32
5          new_gidx.(off, slot) ← r_ik.pending.(off, slot)
6          pmem_flush(r_ik.gidx ← new_gidx, 8)
7          r_ik.pending ← r_ik.limit
8      end
9      pmem_drain()
10     commit_id ← commit_id + R_i.counts
11     pmem_persist(&commit_id,8)
12     if R_i.cb exists then
13         R_i.cb(R_i.ctx)
14     end
15     FETCH_INC(ROR.pid)
16 end
```

unordered L_0 to reduce compaction overheads and alleviate write stall problem[54] at the expense of increasing read amplification. Thanks to the byte-addressable persistent memory, we tackle the problem by redesigning the L_0 data structure in PM.

In our implementation, the GI employs the same volatile index as semi-persistent memtable, where KV pairs are persisted in PM. The records written to memtables are firstly merged into GI by key-granularity in-memory compaction. All records merged from memtables involve only pointer operations therefore no data copy occurs for in-memory compaction. When the memory size of GI exceeds the space limit, its snapshot is created and merged into SSD with L_1. Then it is reclaimed and all KV pairs along with GI are also deleted. Note that the GI can employ any range volatile indexes because the index nodes are rebuilt upon startup by scanning the data records.

In-memory Compaction. The merging operation from memtables to GI is performed by in-memory compaction. Similar to semi-persistent memtable, the key in GI is stored in the leaf node and all multi-versioned values are stored in a sorted array attached to the leaf node. When performing in-memory compaction, a key is firstly inserted into GI if it does not exist. Then stale values belonging to the key are purged from GI while the new values are inserted to the sorted array of the leaf. Since all records are managed by Halloc, key-granularity memory deallocation for KV pairs in PM is not allowed. The memory is freed only when the compaction for the snapshot of GI is completed.

Snapshot. We design the snapshot for GI to guarantee that the GI is still writable when performing compactions from GI to SSD, so that the merging from memtables to GI will not be blocked. In GI, the snapshot is implemented by freezing the current GI and creating a new one. All records in frozen GI are managed by Halloc and reclaimed only when the snapshot is entirely compacted, during which the creation of the new GI is not allowed. The design brings improvements on writes while incurring more overheads on reading as reading may cross two indexes. Moreover, Halloc improves the performance of the memory allocator by employing batch reclaiming strategy while trading off memory management efficiency. One can achieve key-granularity snapshot by specifying new PM allocator that supports fine-granularity memory allocation. However, the design incurs more management costs for persistent memory. We defer the proposal of an efficient persistent global index with key-granularity snapshot to future work.

PM→ SSD Compaction. Since the GI is globally sorted and the snapshot of GI is immutable when performing compaction, the compaction does not block the other writes for GI. Moreover, the range of GI can be conveniently split to perform paralleled compactions even for L_0.

Consistency. The compaction from PM to SSD involves the change of database state, which should avoid inconsistency caused by system crashes. We address the problem by maintaining the manifest log in SSD to record database state since it is not in the critical path of writes. The snapshot of GI manages records coming from several memtables. The metadata of the memtables is kept in Halloc until it is completely compacted into SSD. When a system crashes during compaction from PM to SSD, the data records in PM are removed by replaying manifest logs. And the index nodes for records in PM are rebuilt upon startup.

3.4 Halloc

Halloc addresses the drawbacks of general purpose PM allocators with three key designs: pool based objects preservation, application-aware memory management and unified memory allocation. Many general purpose persistent PM allocators consider both random allocations and deallocations, which results in very fragmented memory allocation and expensive cacheline flushes and fences. Moreover, they suffer from the memory space consumption when working with current transient PM allocators (e.g., memkind[12]), which are normally designed with extra volatile PM pool. Halloc does not enforce the fixed position of memory mapping. All objects for an object pool are addressed by internal 8 bytes integer and their actual memory addresses are remapped from a DAX mapped file upon restart.

3.4.1 Pool Based Objects Preservation. To reduce memory fragmentation, Halloc employs persistent objects pool to preserve non-overlapping memory space between pools. And each pool stores fixed size objects and the allocation or deallocation of memory is performed in object granularity. An object pool consists of a metadata region to store the description of pool (e.g., pool size, object size and object count), an objects array, and a log-free freelist (persistent linked list) to track freed objects. The objects allocation/deallocation of a persistent pool involves multi-words updates of freelist, which often exceeds the hardware limit for atomicity guarantee. As a result, a power outage may lead to persistent memory leaks for objects managed by a persistent freelist. Existing approaches, e.g., PMDK[5], employs transactional semantics addressing the problem, but they incur extra overheads of logging. We avoid logging by employing the approach described below.

Specifically, we reserve one bit for each object index of freelist and design four interfaces to guarantee the atomic object allocation/deallocation: **Get** to get a free object from freelist, **Commit** to

persist object, **Check** to check whether the given object is committed thereby applications can determine whether an object should be dropped, and **Release** to release an object. As shown in Figure 9, the first three operations are used to guarantee atomic allocation. To allocate an object, (1) the object *a* pointed by *hd* is allocated by **Get** (❶); (2) the object *a* is initialized and persisted by applications (❷); (3) applications call **Commit** (❷) to toggle the bit of object index to mark *a* as persisted. If ❷ fails, i.e., because of a power outage, the object *a* will be leaked without **Commit**. If ❸ fails, the object *a* will be overwritten without **Check**. **Release** is implemented by resetting the bit of object index and putting it to the *freelist*. In case of a power outage, the object is still reclaimed upon startup.

To recover an objects pool during startup, the freelist is firstly traversed at startup and all objects that can be reached from freelist are marked in a temporary bitmap. An object that is not committed and not reachable in freelist is marked as a leaked object and reclaimed. The design incurs extra overheads during system recovery. However, we observe that scanning one million objects costs only milliseconds. The recovery cost is negligible in our system.

3.4.2 Application-aware Memory Management. Halloc provides two memory management pools for the LSM-tree: customized objects pool and zone pool. This is based on the memory use pattern of target applications. Specifically, the memory allocation of memtable for LSM-tree is in append-only and batch reclaiming mode.

Customized Objects Pool. As shown in Figure 8, Halloc maintains two kinds of customized persistent object pools to provide metadata persistence for LSM-tree: the Subtable pool to persist the column family object[6, 26] and the memtable pool to manage PM memories for memtable. A Subtable object in Subtable pool contains a list of memtable objects linked by *memlist* similar to the log-free freelist, where the first memtable is mutable and the others are immutable, similar to the specification of RocksDB. And the memtable pool is an object pool, where each memtable object consists of fixed number slots and each slot manages one memory region to enable application-specific PM management. Since the memtable of LSM-tree follows the append-only memory allocation, the *gidx*(a newly designed memory pointer seen in §3.3) of memtable object is designed to reflect the allocation position.

Zone Pool. The zone pool is a built-in pool in Halloc to allow applications to manage their own runtime memory. We design the zone pool because the customized objects pool can not perform memory allocation for objects with variable sizes and of unknown number in runtime. In LSM-tree, KV pairs normally have different sizes. Therefore, we can not preserve object pool for all KV pairs. Specifically, the zone pool is utilized both by memtable for persistent memory management and by volatile manager for transient memory management. The zone objects allocated for memtable follows the append-only and batch reclaiming memory allocation scheme, which is similar to ZNS[9] for SSD and MSLAB[4] for DRAM. The zone objects allocated for volatile manager are further split into smaller units for random allocation/deallocation.

3.4.3 Unified Memory Allocation. Halloc builds volatile memory manager directly on the zone pool so that both volatile allocation and persistent allocation can be enabled from one PM pool. The design enables a unified PM allocation scheme both for volatile

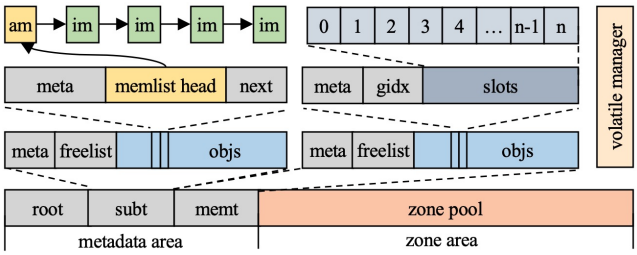

Figure 8: The structure of Halloc, *subt*: Subtable pool, *memt*: memtable pool, *am*: active memtable and *im*: immutable memtable.

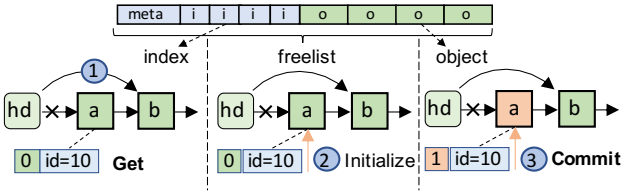

Figure 9: Steps of allocating an object from a pool where the *hd* is a dummy head stored in *meta* field.

allocation and persistent allocation, which improves the space efficiency of memory management. Similar to memkind, Halloc implements the volatile memory allocation by wrapping the jemalloc as the volatile manager, where the configurable memory arenas (memory regions in jemalloc) from PM are registered to jemalloc to manage transient allocations. Zone objects allocated from zone pool for volatile purposes do not perform commit, hence all the zone objects allocated for volatile purposes are reclaimed after the system restart.

One limitation of the design for volatile manager based on zone pool is that the size of a volatile object can not exceed the size of a zone object, because pool based design for zones guarantees only one continuous memory region in one zone object. However, we can divide large volatile object into multiple small objects and perform small allocation for each object. Moreover, if the memory footprint for large volatile objects can be predicted, we can preserve such objects specifically in our objects pool for volatile purposes.

4 EVALUATION

In this section, we first use the YCSB and TPCC benchmarks to evaluate the overall performance of our proposal. Then we evluate each individual component designed in Section 3 with microbenchmarks.

Implementation. We base our implementation on an existing storage engine (denoted as XS) over DRAM-SSD hierarchy which is in turn based on the forked versions of LevelDB and RocksDB. We denote our new proposal over PM-SSD architecture in this work as XP. Specifically, XP replaces the original concurrent skiplist based memtable in XS with our semi-persistent memtable and use ROR algorithm to avoid WAL. Moreover, XS originally employs unsorted L_0 data blocks on SSD. In XP, we replace the L_0 by the Global Index in PM with in-memory compaction. Specially, XS enables intra-L_0 compaction feature to merge several unordered data blocks in L_0 into a large one to reduce the read amplification in L_0 in write heavy workload.

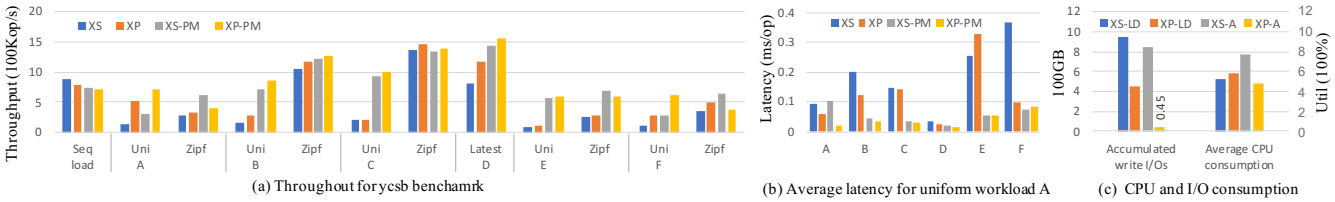

Figure 10: YCSB evaluation, A: reads/updates(50/50), B: reads/updates(95/5), C: reads(100), D: read-latest/updates(95/5), E: scans/updates(95/5) and F: reads/read-modify-writes(50/50).

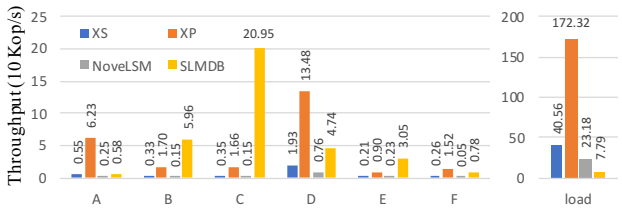

Figure 11: Evaluation for single LSM-tree with 40GB dataset.

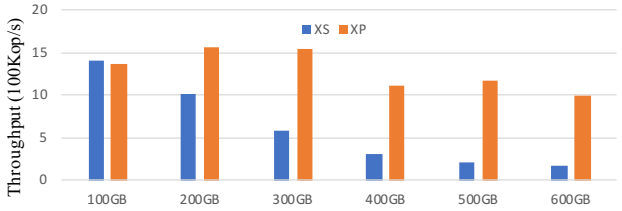

Figure 12: Database size sensitivity for YCSB workload D under 32 client threads.

Experimental Setup. Unless otherwise stated, we configure a single memtable, the global index (formerly the L_0) and L_1 of a single LSM-tree (i.e., a subtable or a partition of a table) as 256MB, 8GB, and 8GB, respectively. In the baseline, we use 256MB L_0 instead. Our proposal enables a much larger L_0 than that in the baseline. Therefore, a large portion of L_0-related I/O amplification is removed by replacing L_0 with a PM-resident global index. We use synchronous WAL for all baselines to guarantee durability and employ the direct I/O to bypass the OS page cache for all test cases.

We run the evaluations in an ECS instance on Alibaba Cloud (ecs.ebmre6p.26xlarge). The instance is equipped with two Intel(R) Xeon(R) Platinum 8269CY CPUs, with a 32KB L1 cache and a 1MB L2 cache per core. All cores in a socket share a 36MB L3 cache. The ECS instance has 187GB DRAM and 1TB Optane DCPMM persistent memory across two NUMA nodes, each of which has 4 interleaved DCPMMs (512GB in total) in AppDirect mode. We attach a 2TB ESSD of the highest performance level PL3 (Enhanced SSD [3]) to the instance as the durable storage. For all experiments, we use *mmap* to map a file in ext4 file system by DAX mode and employ *libpmem* of PMDK for the basic flush and persistence operations. The instance runs Linux with kernel version 4.19.81. And all DCPMM modules are exported as two special storage devices, where each NUMA node is attached with one storage device.

4.1 Overall Performance

YCSB Benchmark. We pre-load 800 million records with 8-byte key and 500-byte value spanning 16 subtables (i.e., LSM-trees), which results in around 500GB of data in total. We consider four configurations: (1) XS with WAL flushed to the ESSD cloud storage (XS), (2) our proposal with 200GB space managed by Halloc in the PM (XP), (3) the XS with the WAL and all levels of the LSM-tree stored in the PM (XS-PM), and (4) our proposal with all data stored in the PM (XP-PM), similarly. Configuration (1) and (2) resemble how we deploy LSM-tree in practice now, and how we plan to use the PM with our proposal, respectively. In configuration (3) and (4), we aim to examine how they perform if the durable storage is replaced with the PM. For all cases, we use 32 client threads with both uniform and zipfian request distributions for 30 minutes.

Write-intensive Workloads(A,F). As shown in Figure 10, with an uniform distribution in the workload, XP/XP-PM outperforms XS/XS-PM by 3.8x/2.3x for workload A and by 2.7x/2.2x for workload F, respectively. And, XP has lower access latencies (36% smaller) than XS (Figure 10(b)). With the skewed distribution (zipfian factor = 1), the gap between XP and XS reduces, while XP-PM performs worse than XS-PM. These results show in general that the efforts made by our proposed XP to reduce disk I/Os pay off, compared with the baseline where significant accesses fall in the storage. The XS-PM performs well, especially when the workload is skewed, because all the data fits in the DRAM and the PM, which is considered to be an expensive architecture in this work.

Read-intensive Workloads(B,C,D,E). With the uniform distribution, XP/XP-PM outperform XS/XS-PM by 1.7x/1.2x on workload B and by 1.4x/1.1x on D and have lower access latencies (reduced by 39%/26% on B and D respectively) because both workloads benefit from faster writes of XP. With the zipfian distribution, XP/XP-PM has a little gain (1.1x/1.05x) for workload B because accesses are skewed towards the DRAM part of XS/XS-PM. Both XP and XP-PM have no performance improvement over XS and XS-PM on workload C and workload E, because most data has been compacted into the storage. We leave a PM-resident cache as our future work.

CPU and I/O Consumption. The CPU consumption and accumulated write I/O consumption in YCSB load and workload A are shown in Figure 10(c). The result shows that XP has better CPU utilization and reduces the accumulated write I/O by 94% compared with XS in workload A. This is because the XP removes WALs and flushes and enables the enlarged persistent L_0 in PM to absorb more updates by in-memory compactions. While in YCSB load, XP has higher CPU consumption than XS because we load data sequentially with 4 client threads and the ROR employs the busy waiting in write pipeline.

Database Size Sensitivity. We vary the database size from 100GB to 600GB and run the YCSB D as a typical OLTP workload in this evaluation. Figure 12 shows that the performance of XS degrades by 88% with the growth of databse size while the XP degrades by 27%. This is because XP always stores the latest hot

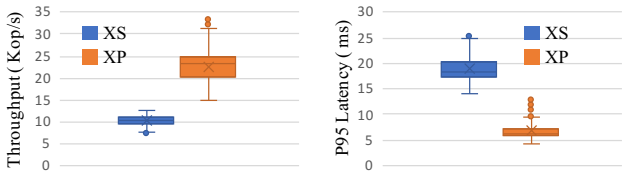

Figure 13: Evaluation for TPC-C benchmark.

data into the larger and faster PM. However, the XS has to read latest updates from the slow disk each time we start the test.

Single LSM-tree Evaluation. We now compare our proposal with the state-of-the-art PM-based LSM-tree alternatives SLMDB and NoveLSM, all configured as a single LSM-tree. We run YCSB workloads with 4 client threads and load 40GB of data in total. Figure 11 shows that XP has the highest write throughput (22x faster than SLMDB and 7x faster than NoveLSM) while loading data. We find that SLMDB and NoveLSM do not support concurrent writes. SLMDB incurs large overheads in maintaining the consistency between disk records and PM records in flushes and compactions, which exacerbates the write performance. The SLMDB achieves the highest read performance as shown in workload C, because it employs a single level architecture with PM and each read costs at most one disk I/O.

TPC-C Benchmark. We integrate our proposal into MySQL as a storage engine plugin and run TPC-C benchmark with 1000 warehouse (80GB initial database size in total) for 30 minutes. Figure 13 shows that the performance of XP outperforms XS by more than 2x and decreases the P95 latency by 62% for transactions. However, XP suffers from larger performance fluctuations because it employs all-to-all compactions from L_0 in PM to L_1 in disk, which causes more aggressive cache evictions in the LSM-tree. We leave the optimizations of compactions and caches as future work.

4.2 Evaluation of Semi-persistent Memtable

In this section, we disable all flushes and compactions to evaluate the performance of the semi-persistent memtable. We bypass ROR by setting the batch size of ROR to 50 as the batch size has saturated the DCPMM hardware. We compare the DRAM-based skiplist (SLM) with our semi-persistent memtable (shown in Figure 14). We also implement the FAST&FAIR[27] (a persistent B+tree) based memtable (denoted as FFM) and the variant of FPTree[44] (a persistent B+tree with internal nodes in DRAM) based one with optimistic lock coupling (denoted as FPM). Since both FAST&FAIR and FPTree do not originally support variably-sized keys, we add a runtime key parser and store only key pointers for them. We evaluate the semi-persistent memtable with the volatile index nodes managed by the jemalloc in the DRAM (SPM-D) and the proposed Halloc in the PM (SPM-P), separately. We insert 30 million records with 8-byte key and 32-byte value (1.5GB dataset in total), which is translated into the 50-byte record with 17-byte internal key and 33-byte value in the memtable.

Insertion Performance. Figure 14 (a) and (b) show the write throughput while increasing the number of threads from 1 to 32. Firstly, SPM-P perform similarly with SPM-D in most cases, although its volatile index nodes stay in the slower PM. Secondly, compared with LSM/FFM/FPM, SPM-D is up to 5.9x/5.8x/8.3x faster for sequential writes, and 2.9x/5.7x/6.0x for random writes. Both SPM-D and SPM-P significantly outperform SLM in terms of insertion performance although the SLM is entirely in DRAM. This is because the SPM-D is a tries-based index without binary search and puts the internal nodes into faster DRAM. Although the FPM also puts the internal nodes into DRAM, it depends on the runtime key parser in our implementation to load keys from the slow PM.

Lookup Performance. Shown in Figure 14(c), the point lookup performance of SPM-D consistently outperforms SLM/FFM/FPM by up to 2.7x/14x/16x, respectively. For lookups, SPM adopts prefix matching while the baselines adopt binary searching. FPM employs the fingerprint to speedup point lookup. However, in the memtable, a point lookup is transformed into a short range scan to get the latest sequence for a key, reducing the benefits of the fingerprint. And, the design of out-of-order leaf nodes in FPM exacerbates the scan performance. The SPM-P is slightly slower than SPM-D as the SPM-P puts the volatile internal nodes into slower PM. For scan performance (Figure 14 (d)), both SPM-D and SPM-P are significantly slower than SLM. However, the slowdown is not caused by the index but the slower PM I/Os. Although the trie-based index suffers from poor scan performance in DRAM-based solutions, it is not the bottleneck in our solution. In fact, our profiling shows that 70% of the time for SPM is consumed by reading records from PM for random scan while SLM reads records from the faster DRAM.

4.3 Evaluation of Reorder Ring

In this experiment, we disable all flushes and compactions to evaluate the performance of ROR. Each thread inserts one million records with 24 bytes. We evaluate the impacts of both the thread number and the batch size.

Impact of Batch Size. Figure 16(a) reports the correlation of throughput and latency (**avg** and **P99** for the 50% and 99% of the longest time for inserting each record, respectively) when we fix the thread number to 32 and adjust the batch size from 1 to 100. The result shows that the throughput gains largely by 49x when the batch size increases from 1 to 90. However, the latency of avg and P99 increases by 1.3x and 1.7x, respectively. This is because each batch is processed parallelly after ROR has reordered them. The throughput outperforms the raw performance (24Mops/s of random writes) of DCPMM hardware by 1.3x when batch size is 90 as ROR employs sequential cache flushing and uses less memory barriers than random writs. The throughput improvement slows down when the batch size is larger than 90 as the the hardware has been saturated, where the P99 latency increases by 1.7x but the throughput gains by only 1.4x when the batch size increases from 50 to 90. The performance drops when the batch size is larger than 90 since large batches may block serial commits.

Impact of Thread Number. In Figure 16(b), We fix the batch size to 50 and vary the thread number from 1 to 64. The results show that the throughput scales almost linearly when the thread number increases from 1 to 16. When increasing the thread number from 16 to 64, the throughput improves by only 1.1x but the P99 latency increases significantly by 2.9x. The reason is that the high resource contention inside DCPMM limits its ability to handle accesses from multiple threads simultaneously. The evaluation shows that the configuration with batch size equal to 50 and threads number less

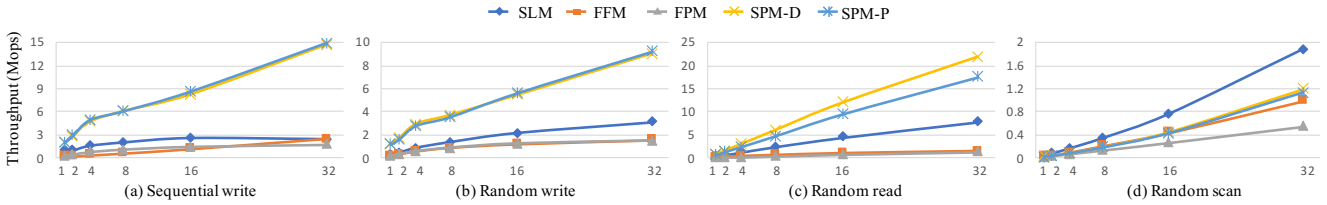

Figure 14: Performance evaluation of memtables with threads increasing from 1 to 32.

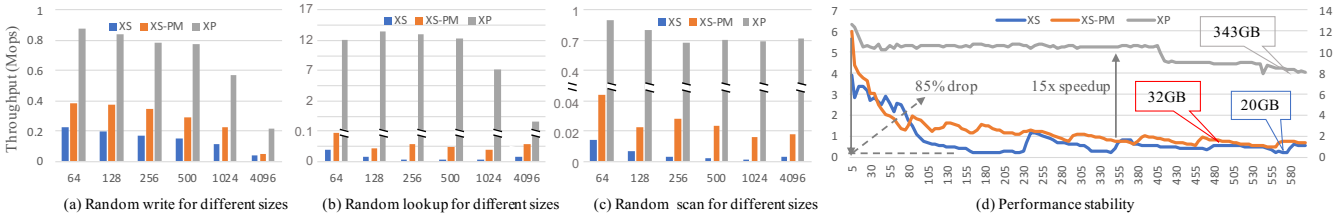

Figure 15: Evaluation for Global Index, where Figure (a-c) depicts the read and write throughout under different record size and (d) shows the performance stability over time.

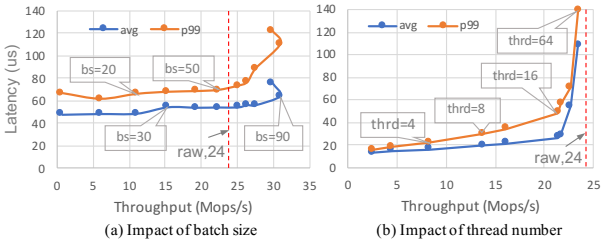

Figure 16: Throughput-Latency Correlation for ROR, (a): impact of batch size and (b): impact of the number of threads.

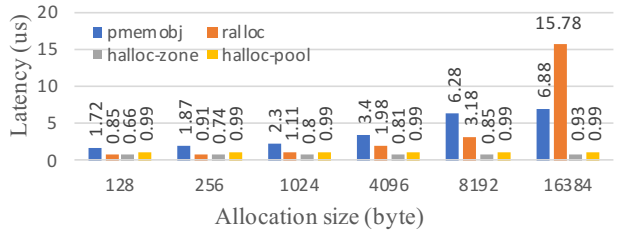

Figure 17: Evaluation of Halloc for persistent performance.

than 16 is a good choice for ROR algorithm in DCPMM for 24-byte records. However it depends largely on machines and workloads.

4.4 Evaluation of Global Index

To evaluate the Global Index (GI), we use single LSM-tree and insert data with size less than the size limit of GI to ensure that the GI is not compacted to disk. We choose the XS and XS-PM for comparisons and disable compaction from L_0 to L_1 to leave all data blocks in L_0. We reserve 10GB block cache and 5GB row cache.

System Performance. We insert 50GB records with 8-byte key and value size increasing from 64 bytes to 4KB bytes and measure the performance of random write, random read and random scan under different KV sizes. Figure 15(a) shows that the XP outperforms XS and XS-PM for all cases because XP avoids WAL and flushes. In Figure 15(b) and 15(c), XP largely outperforms XS and XS-PM in random read and random scan workloads, more than 113x for random read and 21x for random scan respectively. This is

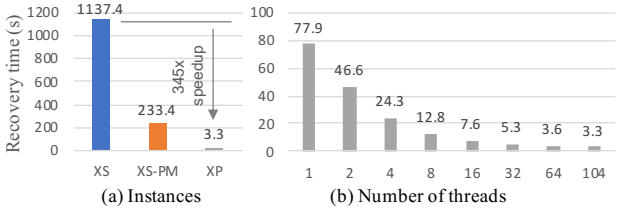

Figure 18: Recovery time, where Figure (a) depicts recovery performance for different instances and (b) shows the multi-thread scalability for XP.

because we disable the $L_0 \rightarrow L_1$ compaction in XS, thereby causing large read amplification due to plenty of unordered data blocks in L_0 (over 58 files). The XP avoids the problem thanks to the GI.

Performance Degradation. In the experiment, we run random reads/writes (1:1) with 32 threads for 10 minutes. Figure 15(d) illustrates that XP is faster (up to 15x speedup) and more stable (less than 35% performance drop) than XS and XS-PM while XS and XS-PM suffer from more than 85% performance drop. For the XS, the accumulated unsorted L_0 results in large overheads for reads (more than 30 unordered data blocks). Since XS may merge multiple unordered data blocks in L_0 into a large one to reduce read amplification, the performance of XS increases when time=230 and time=355. However, it is still significantly slower compared to XP, even XP writes about 345GB records in total while XS/XS-PM only write 30GB/32GB records in the test cycle. In contrast, XP persists data in PM with no needs of WAL and performs in-memory compaction at record granularity, which almost avoids the read amplification. The feature benefits significantly in case of the write-heavy workload where too many data blocks may pile in L_0.

4.5 Evaluation of Halloc

We compare Halloc with the state-of-the-art representative persistent allocators including Ralloc[11] and pmemobj v1.4[5] to evaluate its persistent performance. We exclude transient performance of Halloc with libmemkind[12] since both of them are the wrapper of jemalloc. More performance evaluations about such kind of transient allocators can be found in Waddington et al.[48].

We test the latency for memory allocation by measuring the duration of performing 1 million allocations of objects with size varying from 128 bytes to 16KB. Figure 17 shows that Halloc-zone consistently has the lowest latency across all allocation sizes. Furthermore, Halloc-pool has latency less than 1 us for all test cases. Although ralloc has a little shorter latency than Halloc-pool for allocation size of 128 bytes or 256 bytes, its performance decreases significantly when the allocation size grows larger. For example, for allocation size of 16KB, the latency of ralloc is around 15x of Halloc-zone or Halloc-pool. The pmemobj allocator consistently has the largest latency for allocation size less than 16K. The performance advantage of Halloc is largely due to the fact that it issues only one cache flush and one fence for objects of any size.

4.6 Recovery Time

In the experiment, we compare our solution (XP) with XS and XS-PM to evaluate the recovery performance. We randomly insert data of around 32GB with 8-byte key and 500-byte value, which is around 70 million records in total. The size limit of Global Index is set to 32GB, which enforces that all of the 32GB records are only stored in L_0 (Global Index) for XP. For XS and XS-PM, we maintain 32GB data in memory with no flushes and compactions, which results in at least 32GB valid WAL at runtime.

We set the number of recovery threads varying from 1 to 104 for XP. Figure 18(a) shows that our solution achieves nearly instant recovery, i.e., 3.6s for 64 threads. However both the XS and the XS-PM take hundreds of seconds. The original design of XS only uses single thread to replay the WAL upon startup time, while our solution utilizes all CPU resources to simultaneously rebuild the index nodes. The recovery time is almost reduced linearly with the increasing number of threads (Figure 18(b)).

5 RELATED WORK

One of the major benefits of building databases on the PM is the potential to achieve durable transactions without using disk-based log files (e.g., WALs, redos, etc.). While directly porting the WAL from the slower SSD to the PM helps [36, 52, 53], the latency of writing logs remains in the critical path of transactions. To make it even more challenging, current PM hardware only supports 8-byte atomic writes [53], and further requries expensive memory fences and flushes to achieve the atomicity and persistency. Thus, some existiing work replaces database logs with internal logs to overcome this limitation [5, 47]. Such logging mechanisms may still cause up to around 35% overhead [50]. PMwCAS[49], CoW[16] and NAW[13, 37] enable log-free data structures for the design of range indexes in PM. However, the NAW and CoW significantly rely on expensive general purpose PM allocators and the PMwCAS incurs extra overhead for persisting the multi-word descriptor buffers[37]. They have been found to be inefficient for large objects (>100 bytes)[35] and only work for single KV operations. In this work, we propose a transaction processing mechanisms and data structures on the PM that support multiple KVs and OLTP workloads.

There are many interesting byproducts after making transactions persistent on the PM in this work. Firstly, in-memory data structures (e.g., memtables, indexes) can resume working at their normal performance very quickly after a recovery or reboot, because their major parts are natively persistent. This feature has great potentials for the HA (high availability) of databases. For example, we may no longer need to maintain fresh and full-fledged database replicas for HA purposes. Secondly, regarding of LSM-tree based storage engines, moving structures from the disk into the PM (e.g., L_0) natively reduces the I/Os and CPUs that are previously required for flushes and compactions, introducing new perspectives to address LSM-tree challenges. Both the overall performance and the endurance of the storage benefit from such design.

Bortnikov et al. [10] and Balmau et al. [8] have explored in-memory compactions such as merging hot records in the memory in advance. On the DRAM, however, such optimizations may increase pressures on the memory footprints, which is less of an issue in the PM. In this work, we strike at making PM-native data structures along with their access methods (e.g., compactions). On PM-native data structures, approaches[36] that directly put the disk-resident levels into PM to speedup the I/O in compaction suffer from high overheads in data copying between levels. TLSM uses the persistent skiplist as levels in PM[32], but the persistent skiplist shows poor merge performance. MatrixKV[54] proposes the matrix container to maintain the L_0 in PM, but still shows inefficiency in range query as data records in matrix container are not globally sorted. SLMDB[30] maintains a persistent mutable B+tree in LSM-tree to improve read performance and compacts LSM-tree with persistent B+tree. However, the solution suffers from large overheads in maintaining the consistency between the B+tree and LSM-tree. FPTree uses fingerprints to accelerate point lookups and exploits Hardware Transactional Memory (HTM) for concurrency control [44]. In this work, we are looking at general-purpose data structures that perform reasonably well for a wide range of workloads including range lookups, inserts, and deletes, and can be integrated into existing LSM-tree backed OLTP storage engines with acceptable engineering efforts.

6 CONCLUSION

In this work, we investigate how to leverage PMs to revisit the conventional LSM-tree based OLTP storage engines. Specifically, we (1) propose a light-weight PM allocator called Halloc optimized for LSM-tree, (2) design a high-performance semi-persistent Memtable utilizing the persistent in-memory writes of PM, (3) design the concurrent Reorder Ring algorithm to achieve log-free transactions for OLTP workloads and (4) present a Global Index as the new globally sorted persistent level with non-blocking in-memory compaction. Our evaluation shows that these key designs can unleash the power of PM-SSD storage architecture to significantly improve the performance of LSM-tree based OLTP storage engines. The overall evaluation shows that the performance of our proposal over PM-SSD storage hierarchy outperforms the baseline over DRAM-SSD storage hierarchy by up to 3.8x in YCSB benchmark and by 2x in TPC-C benchmark.

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China (project No.61772056) and the National Key R&D Program of China under Grant 2019YFB2102400.

REFERENCES

[1] Intel 2015. *Intel and Micron Produce Breakthrough Memory Technology.* Intel. Retrieved May 29, 2021 from https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology

[2] Intel 2019. *The Challenge of Keeping Up with Data.* Intel. Retrieved May 29, 2021 from https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html

[3] Alibaba Cloud 2021. *Enhanced SSDs, Alibaba Cloud.* Alibaba Cloud. Retrieved May 29, 2021 from https://www.alibabacloud.com/help/doc-detail/122389.html

[4] Apache 2021. *HBase, a distributed, scalable, big data store.* Apache. Retrieved May 29, 2021 from https://github.com/google/leveldb

[5] Intel 2021. *PMDK: Persistent Memory Programming.* Intel. Retrieved May 29, 2021 from https://pmem.io/pmdk/

[6] Facebook 2021. *Rocksdb, a persistent key-value store for fast storage enviroments.* Facebook. Retrieved May 29, 2021 from https://rocksdb.org/

[7] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Vol. Part F127746. Association for Computing Machinery, New York, New York, USA, 1753–1758. https://doi.org/10.1145/3035918.3054780

[8] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. {TRIAD}: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 363–375.

[9] Matias Bjørling. 2019. From open-channel SSDs to zoned namespaces. In *Linux Storage and Filesystems Conference (Vault 19)*. 1.

[10] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better memory organization for LSM key value stores. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1863–1875. https://doi.org/10.14778/3229863.3229873

[11] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and optimizing persistent memory allocation. *International Symposium on Memory Management, ISMM* (2020), 60–73. https://doi.org/10.1145/3381898.3397212 arXiv:2003.06718

[12] Christopher Cantalupo, Vishwanath Venkatesan, Jeff R. Hammond, Krzysztof Czury lo, and Simon Hammond. 2015. *User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies.* Intel. Retrieved May 29, 2021 from http://memkind.github.io/memkind/

[13] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[14] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2020), 1077–1091. https://doi.org/10.1145/3373376.3378515

[15] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[16] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 133–146.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010.*

[18] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.

[19] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Vol. 4. Association for Computing Machinery, New York, New York, USA, 449–466. https://doi.org/10.1145/3299869.3319903

[20] Alexander Driskill-Smith. 2010. Latest advances and future prospects of STT-RAM. In *Non-Volatile Memories Workshop*. 11–13.

[21] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. *Proceedings of the 13th EuroSys Conference, EuroSys 2018* 2018-Janua (2018). https://doi.org/10.1145/3190508.3190524

[22] Jason Evans. 2006. A scalable concurrent malloc implementation for FreeBSD. In *Proceedings of the bsdcan conference, ottawa, canada.*

[23] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

[24] Sanjay Ghemawat and Jeff Dean. 2011. *LevelDB.* Google. Retrieved May 29, 2021 from https://github.com/google/leveldb

[25] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)* 15, 4 (dec 1983), 287–317. https://doi.org/10.1145/289.291

[26] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 651–665. https://doi.org/10.1145/3299869.3314041

[27] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. *Endurable Transient Inconsistency in Byte- Addressable Persistent B+-Tree.* 187–200 pages. https://www.usenix.org/conference/fast18/presentation/hwang

[28] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-atomic persistent memory updates via JUSTDO logging. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 427–442.

[29] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[30] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-level key-value store with persistent memory. *Proceedings of the 17th USENIX Conference on File and Storage Technologies, FAST 2019* (2019), 191–205.

[31] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Georgia Tech, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *USENIX Annual Technical Conference*. 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan

[32] Jihwan Lee, Won Gi Choi, Doyoung Kim, Hanseung Sung, and Sanghyun Park. 2020. TLSM: Tiered Log-Structured Merge-Tree Utilizing Non-Volatile Memory. *IEEE Access* 8 (2020), 100948–100962. https://doi.org/10.1109/ACCESS.2020.2985407

[33] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[34] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 1–8.

[35] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019), 574–587. https://doi.org/10.14778/3372716.3372728

[36] Jianhong Li, Andrew Pavlo, and Siying Dong. 2017. NVMRocks: RocksDB on non-volatile memory systems.

[37] Jihang Liu, Shimin Chen, and Lujun Wang. 2019. LB + -Trees : Optimizing Persistent Index Performance on 3DXPoint Memory. 13, 7 (2019), 1078–1090.

[38] Chen Luo and Michael J. Carey. 2020. LSM-based storage techniques: a survey. *VLDB Journal* 29, 1 (2020), 393–418. https://doi.org/10.1007/s00778-019-00555-y

[39] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks : LSM-Tree Database Storage Engine Serving Facebook ' s Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.

[40] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (jan 1992), 94–162. https://doi.org/10.1145/128765.128770

[41] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the 1st ACM SIGOPS Conference on Timely Results in Operating Systems, TRIOS 2013*. Association for Computing Machinery, Inc, New York, New York, USA, 1–17. https://doi.org/10.1145/2524211.2524216

[42] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[43] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory management techniques for large-scale persistent-main-memory systems. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1166–1177. https://doi.org/10.14778/3137628.3137629

[44] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[45] John D Valois. 1994. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*. 64–69.

[46] Alexander Van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (2019). https://doi.org/10.1145/3329785.3329930 arXiv:1904.01614

[47] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[48] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. 2019. Evaluation of intel 3D-xpoint NVDIMM technology for memory-intensive genomic workloads. In *Proceedings of the International Symposium on Memory Systems*. 277–287.

[49] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[50] William Wang and Stephan Diestelhorst. 2018. Quantify the performance overheads of PMDK. *ACM International Conference Proceeding Series* (2018), 7–9. https://doi.org/10.1145/3240302.3240423

[51] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a BW-tree takes more than just BuzzWords. *Proceedings of the ACM SIGMOD International Conference on Management of Data* 1 (2018), 473–488. https://doi.org/10.1145/3183713.3196895

[52] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2019), 427–439. https://doi.org/10.1145/3297858.3304077

[53] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[54] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao