Plush: A Write-Optimized Persistent Log-Structured Hash-Table

Lukas Vogel
Technische Universität München
lukas.vogel@in.tum.de

Alexander van Renen
FAU Erlangen-Nürnberg
alexander.van.renen@fau.de

Satoshi Imamura
Fujitsu Limited
s-imamura@fujitsu.com

Jana Giceva
Technische Universität München
jana.giceva@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

Alfons Kemper
Technische Universität München
kemper@in.tum.de

ABSTRACT

Persistent memory (PMem) promised DRAM-like performance, byte addressability, and the persistency guarantees of conventional block storage. With the release of Intel Optane DCPMM, those expectations were dampened. While its write latency competes with DRAM, its read latency, write endurance, and especially bandwidth fall behind by up to an order of magnitude.

Established PMem index structures mostly focus on lookups and cannot leverage PMem's low write latency. For inserts, DRAM-optimized index structures are still an order of magnitude faster than their PMem counterparts despite the similar write latency. We identify the combination of PMem's low write bandwidth and the existing solutions' high media write amplification as the culprit.

We present Plush, a write-optimized, hybrid hash table for PMem with support for variable-length keys and values. It minimizes media write and read amplification while exploiting PMem's unique advantages, namely its low write latency and full bandwidth even for small reads and writes. On a 24-core server with 768 GB of Intel Optane DPCMM, Plush outperforms state-of-the-art PMem-optimized hash tables by up to 2.44× for inserts while only using a tiny amount of DRAM. It achieves this speedup by reducing write amplification by 80%. For lookups, its throughput is similar to that of established PMem-optimized tree-like index structures.

PVLDB Reference Format:
Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. PVLDB, 15(11): 2895-2907, 2022.
doi:10.14778/3551793.3551839

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tum-db/Plush.

1 INTRODUCTION

Persistent memory (PMem) promised low latency, random access performance and throughput comparable to DRAM, as well as data persistency, while being a drop-in replacement of DRAM. PMem-optimized data structures could theoretically achieve the same throughput as their DRAM counterparts while offering granular persistency guarantees without needing additional write-ahead logging. However, in practice, Intel's Optane Persistent Memory Modules (DPCMM) perform considerably worse than DRAM.[1] For example, their read latency is three times higher [42]. While its write latency is comparable to that of DRAM, its write bandwidth is lower by an order of magnitude, as shown in Table 1. Furthermore, PMem's internal smallest unit of access is 256-byte blocks, leading to significant read and write amplification for small accesses.

Table 1: Write access characteristics for different media.

Medium	Latency	Bandwidth	Price	Access unit
DRAM	90 ns	80 GB/s	7.2 $/GB	cache line (64B)
PMem	130 ns	6.5 GB/s	4.0 $/GB	block (256B)
SSD	40000 ns	3.2 GB/s	0.3 $/GB	page (4KiB)

Prior work addresses PMem's high read latency, but often fails to fully leverage its low write latency and mitigate its low write bandwidth. Many current PMem-optimized data structures, for example, follow a hybrid design, storing the recoverable part of their data (e.g., the inner nodes of a B-Tree) on faster DRAM [3, 9, 37]. This approach is excellent for lookups but it does not solve the issues of inserts as every insert has to store its payload on PMem to guarantee data persistency. State-of-the-art PMem-optimized data structures minimize the number of writes per insert to mitigate latency. However, because of PMem's low bandwidth, write amplification is just as much of a problem. PMem's internal 256-byte block structure further exacerbates this problem for small random writes: It internally amplifies each write to update a 256-byte block. This amplification leads to spurious writes saturating PMem's internal buffer, which drives up write latency and thus indirectly lowers throughput even if little actual payload is being stored [15].

Workloads consisting of small writes are a common use case in key-value stores [2]. Yahoo!, for example, states that their typical low latency workloads have more than 50% inserts [40]. We target this use case: We exploit PMem's low write latency by optimizing for small writes while mitigating its low write bandwidth by reducing write amplification. Our approach adapts the write amplification-reducing techniques of log-structured merge-trees (LSM trees) to PMem using hash tables.

LSM trees reduce write amplification for SSDs and HDDs [36] but do not leverage PMem's low latency for smaller writes. They buffer records on DRAM before merging them into consecutively larger sorted runs on SSD or HDD. This buffering reduces write

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551839

[1] From here on, we use the terms PMem and Intel Optane DCPMM interchangeably, as it is the only commercially available product on which we also base our evaluation.

amplification as LSM trees can fill each 4 KiB OS page before writing it back. They accept some overhead incurred by keeping their data sorted to ensure access patterns to SSDs/HDDs are favorable, i.e., by enforcing that merges are sequential reads and writes.

PMem-optimized LSM trees like NoveLSM [25] or SLM-DB [24] adapt existing LSM trees like LevelDB [13] and replace some components with PMem-optimized counterparts but keep their overall architecture. However, modern NVMe SSDs are often more attractive for workloads with large writes as they already offer half of PMem's throughput at a tenth of its cost. Therefore, porting LSM trees directly to PMem is hard to justify if one does not need the properties offered by sorting data (i.e., range queries) as their design does not take advantage of PMem's superior random write latency.

In contrast, we exploit that on PMem, it is unnecessary to generate large sequential runs by sorting records as it already reaches full bandwidth with just 256-byte writes. We instead propose gathering records in a list of unsorted 256-byte buckets, which we address through a hash table. Whenever such a bucket list is full, we propose re-hashing its contents and recursively merging them into a bigger hash table (the "next level"). We thus adapt the LSM tree's merging approach to a PMem-aware hash table. This was previously not workable, as the throughput of conventional block storage devices is too low for writes that small. Larger buckets to accommodate 4 KiB pages were also infeasible as search time would dominate for such large unsorted buckets. We call our approach **Persistent Log-strUctured haSH-table**, or **Plush** for short. In all other aspects, Plush takes proven approaches by LSM trees but adapts them for a PMem-*native* data structure:

Batch writes to PMem. Like an LSM tree, Plush gathers new data in DRAM before moving it to PMem in batches, minimizing write amplification so that PMem does not have to deal with spurious writes which would increase latency and lower throughput. However, it uses a hash table instead of a skip list. Plush allows a *configurable* amount of DRAM buffer. Established hybrid data structures, in contrast, cannot control their DRAM consumption as it grows with the record count. This puts a limit on record count before running out of DRAM. Plush has no such limit.

Store large records out of place. In contrast to many other PMem data structures, Plush supports variable-length keys and values. It employs a similar approach to LSM trees like WiscKey [33], which stores values in a separate log that periodically collects garbage. This approach reduces write amplification, since Plush does not have to copy values when merging them into the next level.

In summary, our main contributions are:

(1) We explore how approaches to reduce write amplification developed for LSM trees can be adapted to PMem with the help of hash tables.
(2) We propose Plush, a write-optimized hash table for PMem with *bounded* DRAM usage and low write amplification.
(3) We evaluate Plush with fixed and variable-length records and show that it outperforms state-of-the-art PMem data structures for inserts.

To our knowledge, Plush is the first PMem-optimized data structure to take this approach.

2 BACKGROUND

Plush is a *hash table* for *persistent memory* inspired by *LSM trees*. This section introduces each aspect.

2.1 Persistent Memory

PMem positions itself as a drop-in replacement of DRAM as it has the same load and store interface: One allocates a chunk of it and treats the memory region just like any allocated memory on DRAM. There are, however, some particularities unique to PMem:

Performance. We measured PMem performance in our earlier work [42]. We found that one can expect a 3.2× higher read latency for PMem compared to DRAM but similar write latencies. The write latencies are similar as a store does not have to reach the physical medium but just the CPU's ADR domain which already guarantees persistency [23], while reads have to go all the way to the physical medium. For throughput, PMem falls off more: 2.6× for reads and 7.5× for writes. Thus, a significant advantage of PMem over conventional block storage is its low latency. A persistent data structure should therefore exploit this low write latency.

Persistency barriers. Even on PMem, a system crash can lead to data loss: For example, the CPU might cache dirty data and delay flushing it to PMem. On a crash, all dirty cache lines are lost as caches are not persistent. Therefore, one has to use *persistency barriers* to ensure that the persisted data is always in a *consistent* state. A persistency barrier is a `clwb`, `clflush`, or a non-temporal write followed by an `sfence`. Flushing the cache line guarantees that the data reaches PMem. The store fence forbids the CPU to re-order any writes before or after. Such a barrier is expensive as it blocks until the CPU evicts the cache line to its ADR domain and the store fence prevents the CPU from concealing this stall by reordering other stores. One should therefore use as few persistency barriers as possible. If a barrier is required, one should ensure that the whole flushed cache line consists of payload to keep write amplification low. Intel's extended ADR (eADR), available since Ice Lake, solves this issue by including the CPU cache in the persistency domain, obsoleting persistency barriers. Since eADR needs a special power supply and not all CPUs supporting PMem support eADR, persistency barriers are still required for backward compatibility.

Torn writes. The system might crash while flushing data to PMem, resulting in a *torn write*. PMem guarantees that writes up to 8 bytes are atomic, larger writes might only be persisted partially after a crash. Torn writes are usually prevented by first writing a record to PMem followed by a persistency barrier. Afterward, this record can be "armed" by atomically writing an at most 8-byte header containing a `valid` bit followed by another persistence barrier. Having to use two persistence barriers makes this approach expensive. We have previously researched and tested approaches requiring only one persistence barrier [42], which we also employ in Plush.

256-byte blocks. Internally, PMem operates on 256-byte blocks, just like CPUs use 64-byte cache lines. The same principles apply: To reduce write amplification, programs should write and read data in 256-byte blocks. When this is not feasible, writes should be sequential so that PMem's write combining buffer can merge multiple writes which also enhances PMem's limited write endurance [22].

2.2 LSM Trees

Write amplification is an issue with data structures operating on background storage. Since the smallest unit of access is often larger than the record to be stored (256 B for PMem, 4 KiB for HDD/SSD), it is desirable to batch multiple writes, which reduces write amplification. LSM trees, therefore, buffer new records in DRAM. The LSM tree recursively merges the DRAM buffer into consecutively larger layers of background storage whenever it reaches a size limit. LSM trees keep the records sorted, usually using a skip list on the DRAM layer to make this merge process efficient. A k-way merge then comprises k sequential reads and one sequential write. This merge fits the access patterns of HDDs and SSDs as they benefit from sequential access, but keeping data sorted is expensive: Inserts cannot happen in constant time, and skip lists offer poor cache locality and can suffer from write contention. If, however, records are only stored on DRAM and PMem, there is no performance benefit for keeping them sorted: PMem has exceptional random write latency as long as writes are grouped into 256-byte blocks. Existing approaches to adapting LSM trees for PMem do not leverage this advantage but replace or improve just a few core components of already established LSM trees. NoveLSM, for example, adds persistent skip lists and mutable memtables [25]. SLM-DB only employs a single level and additionally keeps a persistent B-Tree index [24].

2.3 Hashing on PMem

Most bleeding-edge PMem-based hash tables use extendible hashing [12] or a variant thereof. Extendible hashing splits the hash table into a set of fixed-size buckets and a hash-addressable directory whose entries point to those buckets. The buckets store the actual records, consisting of a key and its value. To accommodate skew, n directory entries, with $n = 2^k, k \geq 0$ may point to the same bucket. That way, underutilized directory entries do not need their own (nearly empty) bucket taking up unnecessary space. Whenever a bucket is full, it is split into two buckets. All records of the old bucket are re-hashed with one additional bit of the hash function discriminating in which new bucket they belong. Afterward, $n/2$ directory entries of those that pointed to the old bucket point to each of the two new entries. If a bucket was already pointed to by just one directory entry before the split, we cannot discriminate further. In that case, the whole directory is doubled. Afterward, every bucket is pointed to by at least two directory entries and the bucket can be split. This is called a *structural modification operation (SMO)* which is very expensive and hard to do concurrently and with consistency guarantees.

Modern PMem-based hash tables like CCEH [35] or Dash [32] group multiple buckets into a segment to better optimize for PMem block size. They split a segment when *any* of its buckets is full. Therefore, hash tables take great care to improve the segment load factor to reduce the number of splits and SMOs arising from them. Level hashing employs a second level with standby buckets [53], Dash uses stash buckets.

3 OVERARCHING DESIGN

Plush combines the highlights of LSM Trees with the highlights of hash tables. As PMem does not depend on sequential accesses as long as the accesses are grouped in 256-byte blocks, we propose doing away with sorting and replacing the LSM tree's layers with hash tables. We still keep a DRAM buffer to group the records into 256-byte blocks, reducing write amplification.

Plush also builds on the foundation of extendible hashing. Like CCEH and Dash, it groups multiple buckets per directory entry. However, when a directory entry would need to be split, Plush does not split in place, but merges its records into a hash table with a bigger directory a level below. We call this a migration. This leveling approach allows Plush to skip expensive SMOs altogether. Unlike CCEH and Dash, Plush can also insert a record into any bucket of a directory entry instead of a specific one determined by the record's hash. While this slows down lookups, it speeds up inserts, as the load factor is higher: A directory entry is only migrated if it is full, resulting in low write amplification. To speed up lookups, Plush uses positive bloom filters in the directory entries which indicate whether a specific bucket contains a record with the requested key.

Plush is, to our knowledge, the first PMem-optimized data structure combining LSM trees and hash tables in this way. Plush exploits PMem's low random write latency for 256-byte blocks. It mitigates PMem's low write bandwidth by reducing its write amplification.

This approach helps Plush to achieve the following design goals:

Hybrid architecture. Plush uses a small but configurable amount of DRAM to increase throughput. Here, DRAM acts as a buffer whose size is configurable. In contrast, other hybrid PMem data structures cannot provide an upper bound for DRAM consumption.

Low write amplification. Plush avoids expensive random writes to PMem and instead groups data into 256-byte chunks in a DRAM buffer to write at once. It keeps a write-ahead log of not yet persisted records to still guarantee per-record persistency. Since Plush writes this log sequentially, PMem can use its write combining buffer for increased throughput. Reducing write amplification also conserves PMem's limited endurance.

Reduce persistency barriers without relaxing persistency guarantees. We tolerate small time windows where data is duplicated to ensure that Plush is always consistent. This concession allows us to reduce the number of persistency barriers. In the event of a crash, Plush amortizes data deduplication over runtime after recovery.

Concurrency without persistent locks. PMem-optimized hash tables often need locks stored on PMem for structural modification operations to reconstruct the current state during recovery. Since Plush is always in a consistent state, we can forgo such persistent locks. Inserts use fine-grained locking on DRAM, while lookups only use optimistic locking [5, 26].

Efficient bulk loading. Plush guarantees persistency with the help of a PMem write-ahead log. For bulk loading, relaxed persistency guarantees (i.e., manual checkpoints) are often sufficient. The user can temporarily turn off logging for increased throughput.

First-class support for variable-length records. In contrast to other PMem data structures, Plush supports both variable-length keys *and* values. Unlike some prior work like Dash, Plush always persists the payload itself, i.e., does *not* treat keys or values as pointers to data managed by a separate data structure like a write-ahead log. No separate write-ahead log is therefore required when using Plush.

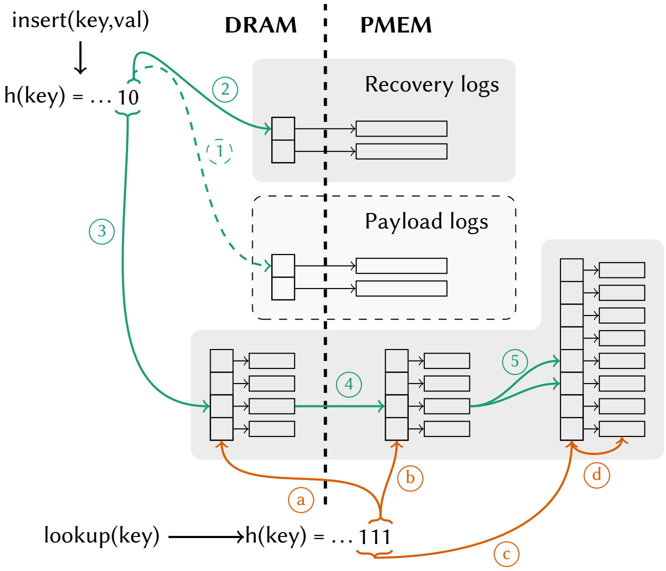


Figure 1: Plush's component overview with illustrated insert (① - ⑤) and lookup algorithms (ⓐ - ⓓ). The colored lines represent steps in the algorithms, the black lines pointers of the data structure.

4 ARCHITECTURE

Figure 1 shows Plush's architecture. It consists of three components: A multi-leveled hash table stores the (fixed-size) records. A table of write-ahead recovery logs ensures that records that have not yet reached PMem are recoverable. An optional table of payload logs stores variable-length records.

4.1 Multi-leveled Hash Table

The core of Plush is its hash table. It stores all records. Figure 1 shows such a hash table with fanout 2.

Levels. The hash table consists of multiple levels, with the lowest level residing in DRAM. All higher levels are stored on PMem, with each level's directory multiplying in size by a configurable power of 2, the fanout, except for the first PMem level, which may have a smaller fanout. The user can thus adjust the DRAM consumption by varying the size of the DRAM level compared to the first PMem level without modifying the fanout on PMem. Like extendible hashing, each level's hash table consists of a directory whose entries contain a bucket list of up to fanout buckets. A bucket holds up to 16 records consisting of 8-byte keys and values. It thus has a total size of 256 bytes which is the same size as a PMem block.

Migration. When all buckets for an entry are full, Plush re-hashes their records using $\log_2(\texttt{fanout})$ additional bits of the hash function. It then distributes the records onto fanout directory entries on the next level (④), recursively if necessary (⑤). Figure 2 illustrates the migration process. Plush appends records to the end of the last non-full bucket for the corresponding directory entry on the next level. If that bucket overflows, it allocates a new bucket (🟩). Plush then clears the old buckets in the original level, making space for newly inserted records. Records thus slowly move to larger levels like in an LSM tree. Assuming a uniform hash function, we expect fanout full buckets in level n to distribute evenly onto the fanout directory entries on level $n + 1$ where each directory entry receives $\approx$ 1 bucket of new records. Migrating a bucket to the next level, therefore, results in fanout PMem block reads and one write in the best case (🟧). In most cases, however, there is not exactly one bucket's worth of entries to insert, or the current bucket already contains some elements from a previous migration forcing some records to spill into the next bucket (🟩). We, therefore, have to expect two block writes per migrated bucket on average. Since most records will be on the highest level, the average per-record write amplification grows linearly with the number of levels (and thus logarithmically with the record count). This, however, is also true for conventional PMem data structures. Hash tables have to deal with SMOs and segment splits, trees with (leaf-) node splits.

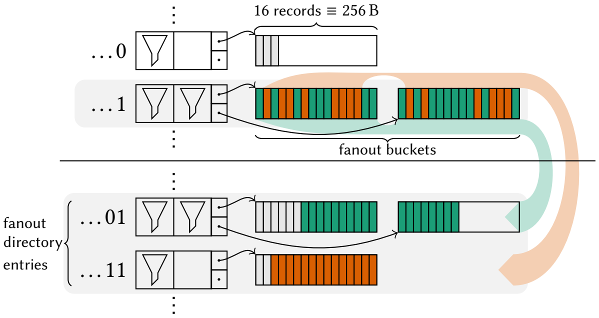


Figure 2: Migrating the buckets of a directory entry.

DRAM buffer. Plush always inserts new elements into DRAM (③). Buffering records in DRAM guarantees that Plush never stores single records on PMem by itself, which would amplify writes by 16 (updating a 256-byte block for each 16-byte record). The buffering approach is an advantage over pure PMem data structures which cannot buffer and combine writes in DRAM by design.

Lookups. Lookups search for the key by probing a filter in the directory entry at each level consecutively (ⓐ - ⓒ). Only on a filter hit is the actual bucket accessed and searched as well (ⓓ).

Figure 3 shows the layout of the hash table's directory and buckets. Each directory entry consists of:

Filter. Plush uses per-directory entry filters to efficiently check whether a bucket contains a key. In contrast to prior work like the FPTree using fingerprints [37], Plush uses a partitioned bloom filter [38, 39] where each partition covers the keys of one bucket. By forgoing fingerprints for bloom filters, we sacrifice the ability to find the offset of potential hits within the bucket upon a lookup (ⓐ) but achieve a far lower false-positive rate. Plush stores all keys within a bucket sequentially, so that it can compare all of them with the search key with two AVX-512 SIMD instructions. Since the PMem block read latency dominates the bucket lookup, this does not lose us significant performance at the benefit of a far lower false-positive rate compared to fingerprinting. A negative lookup on a hash table level thus only has to load the bloom filter, which fits into a single PMem block. When doing inserts, we bulk insert multiple elements into the same directory entry, updating the filter

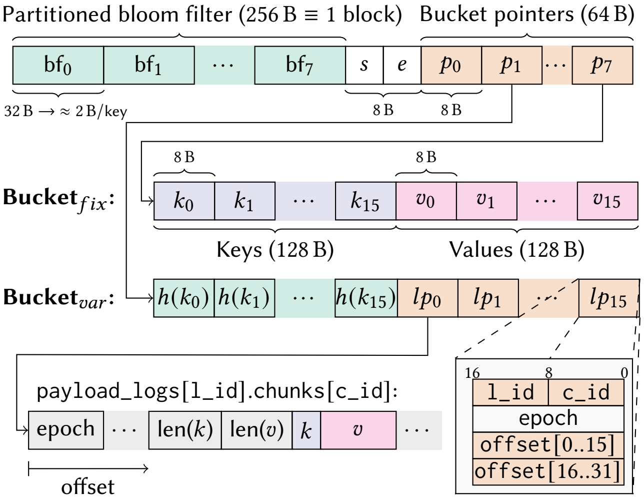


Figure 3: Layout of a directory entry, its buckets (for fixed and variable-length keys), and the payload log for fanout 8.

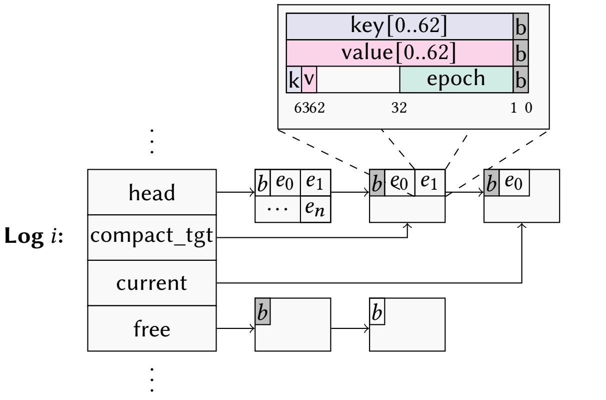


Figure 4: Layout of a recovery log. Each log consists of a set of chunks, split into in-use chunks and a free list.

in bulk. An insert of 16 elements thus only requires us to update the filter once (≡ 1 PMem block write).

Size and epoch. Since Plush fills buckets sequentially, a single size field in the directory entry suffices to calculate into which bucket and position a new element has to be inserted. The recovery log (see Section 4.2) uses the epoch field to determine which entries Plush has already migrated to a persistent hash table level on PMem. Plush increments it after every migration.

Bucket pointers. The directory entry stores an array of pointers to the buckets containing the records. Plush updates the array whenever it allocates a new bucket. Plush distinguishes between two kinds of buckets: Buckets for fixed-size records (up to 8B keys and 8B values) and buckets for variable-length records. We discuss the second kind in Section 4.3. Fixed-size buckets are the size of a single PMem block and can thus store 16 records. Plush can optionally drop the bucket pointers and pre-allocate all buckets for all directory entries of the first *k* PMem levels at pre-defined offsets as an optional optimization. Pre-allocating buckets incurs space overhead, especially if the utilization of the hash table is low. On the flip side, it speeds up lookups and inserts as the pointer dereference (a PMem read) is replaced by an unconditional, pre-calculated jump. Since we expect all but the last level to be full, pre-allocating buckets for the upper levels will not impact space consumption in the long run but considerably speed up lookups and inserts. Section 6.5 evaluates the impact of this optimization.

Plush can also store 16-, 32-, and 64-byte keys and values in its buckets. This is disabled by default as it reduces per-level capacity and requires coarser locks for synchronization as C++ atomics cannot be larger than 8 bytes.

4.2 Recovery Log

Plush guarantees that it persists records permanently when the insert method returns. Since Plush stores the newest entries on the DRAM part of its leveled hash table, it has to keep a persistent log of all entries that it has not yet migrated to a PMem hash table. Before it inserts a record into the hash table, it persists the record in such a log (②). Unfortunately, this is a “law of nature” we cannot get around: To guarantee persistency for each 16-byte record, we have to persist each record in the log with an expensive persistence barrier and high write amplification (since 16 ≪ 256). However, PMem’s write combining buffer saves us: Since we write to each log sequentially, it can combine multiple writes into a larger write without weakening persistence guarantees. Plush’s design thus prevents small random writes to PMem: Instead, it either batches writes (hash table bucket migration) or issues sequential writes (recovery logs) which the write combining buffer batches internally.

Plush employs multiple logs, which it partitions by the key’s hash. Partitioning is a trade-off: The more logs Plush uses, the lower the write contention on each log. However, more logs also mean the write combining buffer will not be as effective since more random accesses hit PMem. Plush thus allows a configurable number of logs independent of the DRAM hash table size. It uses 64 logs by default. Since Plush partitions the logs by key, it can still ensure a global ordering of updates on the same key, as it persists updates to the same key in the same (sequential) log.

Figure 4 shows the layout of a log. It consists of a configurable number of fixed-size chunks arranged in a linked list and a free list of chunks currently not in use. A `current` pointer points to the chunk that is currently being filled. Each log entry consists of the key, value, and epoch counter. The epoch counter signifies during which epoch of the DRAM hash table’s directory entry this record was inserted. Plush increments the DRAM epoch when it persists the corresponding buckets on PMem. It uses this information during log compaction and recovery to determine whether it has already moved the entry in the leveled hash table to PMem and can thus discard it.

The log has to guarantee persistency and prevent torn writes. Plush achieves both through the RAWL approach invented by Mnemosyne [43]: PMem guarantees atomicity for up to 8-byte integers. We thus distribute our data between 3 integers, each having 63 bits of payload and a flag bit *b*. Plush considers an entry valid if its flag bits are equal to the flag in the chunks header. Since each integer was stored atomically, it is guaranteed that all data is persisted if all flags match. This approach needs just one persistence barrier after writing all three integers. Invalidating a chunk's entries is cheap: Plush flips the flag bit in the chunk header.

When the free list is empty, Plush compacts the oldest log chunk by iterating over its entries. When an entry is obsolete (i.e., its epoch is smaller than that of the DRAM hash table entry it belongs to), it skips the entry. Otherwise, it copies the entry to the chunk pointed at by `compact_tgt` preserving order. Last, Plush invalidates all chunk log entries by flipping the flag bit *b* in the chunk header and inserting it into the free list. Since Plush uses a fixed number of fixed-size chunks per log, the log space consumption is bounded.

Plush also exploits that the capacity of its DRAM hash table is fixed. It sets the log's total capacity so that all logs together can hold more records than the DRAM hash table. This way, Plush will probably already have migrated the corresponding records of the oldest chunk's log entries to PMem by the time it compacts the chunk. In this case, compaction is just a sequential read without writes as all entries can be discarded. Plush additionally stores the highest epoch per chunk in DRAM. If that is lower than the epoch of all DRAM directory entries belonging to the chunk's log partition, it can even skip this read. By choosing the right log size, compaction is thus free. By default, Plush employs 64 logs with six chunks of 5 MiB. This configuration achieved free compaction at Plush's default DRAM directory size of 2^{16} in our tests.

Plush using a DRAM buffer with a separate PMem log has a distinct advantage over other PMem data structures. Unlike Plush, they trigger a PMem block write with high write amplification for every insert as they do not write to PMem sequentially and thus cannot exploit the write combining buffer.

4.3 Payload Log and Variable-Length Records

Plush also supports variable-length keys and values. It stores keys and values larger than 8 bytes out of place in a separate payload log. This separation keeps cache locality high, lookups inside buckets fast, and write amplification low. When storing records out of place, Plush does not have to copy them whenever it migrates their bucket to the next level. The payload log is similar to the recovery log except that log entries inside chunks can have arbitrary sizes, and each chunk has a log epoch counter (cf. Figure 3). Whenever Plush inserts a variable-length record, it first inserts the record into the payload log (①) followed by a persistence barrier. Only then is the record inserted into the recovery log and hash table. Plush uses special buckets for variable-length values. Instead of a key and value, they store a hash of the key and a pointer to the record's position in the payload logs. Plush pre-faults the recovery- and payload logs at startup to reduce page fault overhead at runtime [10].

5 OPERATIONS

In this section, we take a closer look at Plush's three operations: *upsert*, *lookup*, and *delete*. We then explore how Plush's design guarantees that it is always in a *consistent state* and ensures it *scales* with multiple threads. We finally explore how Plush handles recovery and how it can further speed up inserts for bulk loading.

5.1 Upsert

Like an LSM tree, Plush is an append-only data structure. It, therefore, combines inserts and updates into a single *upsert* operation.

```
1  void upsert(KeyType key, ValType val) {
2    uint64_t hash = hash(key);
3    DirectoryEntry& e = dram_directory[hash % dram_size];
4    lock(e);
5    if (e.size == fanout * 16) {
6      migrate(e); ++e.epoch;
7    }
8    Log& log = logs[hash % log_num];
9    LogEntry& entry = log.entries[log.size++]; // Entries
       are on PMEM, metadata (i.e, size field) is on DRAM.
10   entry.store(key, val, e.epoch); //Using RAWL
11   persist(entry);
12   e.buckets[e.size / 16].keys[e.size % 16] = key;
13   e.buckets[e.size / 16].vals[e.size % 16] = val;
14   ++e.size;
15   unlock(e);
16 }
```

Listing 1: Insertion algorithm for a fixed-size KV pair.

```
void migrate(DirectoryEntry& e) {
  List<Record> rehashed[fanout];
  for (int idx = e.size - 1; idx >= 0; --idx) {
    KeyType key = e.buckets[idx / 16].keys[idx % 16];
    ValType val = e.buckets[idx / 16].vals[idx % 16];
    List<Record> tgt =
          rehashed[(hash >> lvl_bits) % fanout];
    if (!tgt.contains(key)) {  tgt.append((key, val)) }
  }
  for (int e_idx = 0; e_idx < fanout; ++e_idx) {
    DirectoryEntry tgt = get_entry(lvl+1, index_of(e) *
     fanout + e_idx);
    for (int pos=0; pos < rehashed[e_idx].size; ++pos) {
      if (tgt.size == fanout * 16) { // Target full?
        migrate(tgt); // Recursive
      }
      update_filter(tgt, rehashed[e_idx][pos].key)
      insert(tgt, rehashed[e_idx][pos]);
    }
    // Nothing is guaranteed to be persisted yet!
    persist([tgt.keys, tgt.values, tgt.filters]);
    // If crashing here: Filter has false-positives
    tgt.size += elems_inserted;
    tgt.epoch = e.epoch;
    persist([tgt.size, tgt.epoch]);
    // If crashing here: Duplicates! -> will be cleaned
     up with the next migration
  }
  e.size = 0; e.filters = 0;
  persist([e.filters, e.size]); // Consistent again!
}
```

Listing 2: Migration algorithm for a directory entry.

Internally, an update is just an insert. The *lookup* operation has to ensure that it returns the latest value inserted for the specified key.

Listing 1 shows the upsert algorithm for a fixed-size key-value pair. Plush first determines the target directory entry in DRAM (2-3) and locks it (4). Note that this is a regular lock in DRAM. If there is no space for the record, it migrates the entries' contents to PMem (5-7). Note that the directory entries on the level to which Plush migrates the records only accept records from the current entry and are, therefore, indirectly locked. The migration thus only needs to consider concurrent *lookups*, but no inserts. Plush then inserts the record into the recovery log (8-11), persists it using RAWL (cf. Section 4.2), and finally inserts it into DRAM (12-15).

Listing 2 shows the migration process. First, Plush allocates a temporary buffer for re-hashing keys for the next level (2). It then iterates over the records of the old level from *back to front*, inserting them into the correct buffer (3-9). It skips records with keys that have already been inserted into the buffer before and thus prunes older versions of updated records. While this check has a runtime complexity of $O(n^2)$ in theory, n is only 16 on average in practice leading to negligible constant overhead. Afterward, it inserts the contents of the buffers into the correct bucket on the next level (10-26), with a recursive migration (14) if necessary. The order in which it updates and persists the data is critical: Plush first persists new keys, values, and filters (20). Since it has not yet updated the size field, the new values are not reachable yet and Plush will just overwrite them after a crash and recovery. The filters, however, are not protected: If the system crashes after Plush has persisted the filters at least partially, they may yield false positives after a restart. Inconsistent filters are not an issue as they are probabilistic. Plush has to deal with false positives anyway, leading to slightly degraded performance for the partially migrated bucket after recovery at worst. Plush automatically fixes this as it resets the filters during migration. After it has updated and persisted the size (22-24), the migrated records are visible but are shadowed by their still valid old version on the previous level. If a crash occurs now, those values will be duplicated until they are migrated and merged into the next level. While this leads to some potential data overhead after recovery, Plush can migrate data holding no locks on PMem, keeping read- and write amplification low. In the last step, Plush marks the old entry as empty by zeroing and persisting its size and filters (27-28).

For variable-length records, migrations have to consider additional issues: As newly inserted records shadow old records with the same key, the payload log may contain old, no longer reachable entries. When a migration merges bucket entries, it does not purge their records from the payload log leading to two inconsistencies Plush has to consider:

(1) Plush updated a record in the hash table, but its old value still lives in the log. Plush solves this through periodic garbage collection. Whenever it runs out of memory, it compacts the oldest chunk of each payload log to a new chunk. Plush checks whether each entry is still reachable, i.e., a lookup with the logged key will yield a pointer to the current log entry. If that is not the case, the entry is stale, and Plush garbage collects it. Otherwise, it moves the record and updates the log pointer. Like with the recovery logs, we assume that the oldest entries are stale most of the time so that Plush does not have to move many records on garbage collection.

(2) During a migration, Plush might discover that a pointer in the hash table points to a log entry that no longer exists. Pointers dangle if the user updated a record, and Plush then garbage collected its old version but has not yet merged the old pointer on the higher hash table level. For this reason, the log pointer contains the epoch of the chunk it is pointing to. When Plush migrates a chunk, it increments its epoch. It thus can detect a dangling pointer by comparing the epochs and excluding them from migration to the next level.

In contrast to the recovery log, the payload log does not need to detect torn writes: Plush deems log entries valid if a reachable pointer points to them. It only persists pointers after it flushed the log entry to PMem. As torn log entries were never valid, no pointer will point to them, and Plush will garbage collect them.

```
1  ValType lookup(KeyType key) {
2    uint64_t hash = hash(key);
3    DirectoryEntry& e = dram_directory[hash]; //Try DRAM
4    ValType val = lookup_in_bucket(key);
5    if (val) { return val;}
6    //Iterate over PMem until hit or reached last level
7    for (int lvl = 0; !val && lvl < pmem_levels; ++lvl) {
8      DirectoryEntry& e = get_entry(lvl, hash);
9      val = lookup_in_lvl(e, key);
10   }
11   return val;
12 }
13 ValType lookup_in_lvl(DirectoryEntry& e, KeyType key) {
14 RETRY:
15   int b_id = e.filter.get_bucket(key);
16   ValType result = nullptr;
17   if (b_id != -1) {
18     int epoch = e.epoch;
19     result = lookup_in_bucket(e.buckets[b_id]);
20     if (e.epoch != epoch) { goto RETRY; }
21   }
22   return result;
23 }
24 ValType lookup_in_bucket(Bucket& b) {
25   for (int i = get_size_of(b) - 1; i >= 0; --i) {
26         if (b.keys[i] == key) { return b.values[i]; }
27     }
28   return nullptr;
29 }
```

Listing 3: Lookup algorithm for a fixed-size KV pair.

5.2 Lookup

Listing 3 shows the lookup algorithm. Since updates are just inserts, multiple versions of a record can co-exist on different levels of the hash table. Plush thus needs to ensure *lookup* returns the latest version of a record. To guarantee that, it searches all levels, beginning with DRAM (3-5) and ending with the highest PMem level (7-12), and buckets from back to front (25-27). It only checks a bucket if its bloom filter cannot rule out a hit (15-17). This operation is expensive as we can expect most data to live in the last PMem level, leading to negative lookups at all lower levels. Plush mitigates the issue by optionally storing the directory filters in DRAM instead of PMem. Storing filters in DRAM does not weaken Plush's persistency guarantees as it can recover filters from the records stored on PMem but lengthens recovery time. Sections 6.5 and 6.7 evaluate the trade-off. DRAM consumption stays configurable as the user can decide which level's filters Plush should store on DRAM.

5.3 Delete

Plush uses tombstone records for deletion. Deletes are thus just inserts where the value equals a pre-defined tombstone marker. During migrations, tombstone markers "cancel out" records with the same key, i.e., drops the record.

5.4 Recovery

When Plush crashes, it loses the contents of its DRAM hash table and has to rebuild it from the recovery logs. Plush iterates over every log and compares each log entry's epoch with the epoch of its target bucket on the lowest PMem level. If the bucket's epoch is higher than that of the recovery log entry, Plush had already persisted that entry before the crash and skips it. Otherwise, Plush reinserts it into the DRAM hash table. As the recovery logs are partitioned, Plush can trivially recover them in parallel.

If Plush crashes during compaction of a log (i.e., while copying a log entry to a new chunk), there is no special case needed: Either it did not wholly persist the new version of the copied log entry and therefore recognizes the new value as torn, or both versions are valid. If both versions are valid, Plush will discard the older entry as it does a duplicate check when inserting recovered entries. Since all recovered entries fit into DRAM by definition (they would not need to be recovered otherwise), no migration is necessary during recovery. Plush's recovery is, therefore, idempotent. It can just restart a crashed recovery. The payload log for variable-length entries does not need a special recovery mechanism. An entry is, by definition, valid if a value in the hashtable is pointing to it. If Plush recovered no such pointer while iterating over the recovery logs, it will drop the entry in the next garbage collection run.

5.5 Concurrency

Upserts lock each bucket non-persistently. Assuming a fanout of 16, in 255 out of 256 cases, no migration is required, and the lock is thus just held for a short time. Since Plush uses hashing, inserts are ideally uniformly distributed over the directory. This uniform distribution is an advantage over LSM trees which have to keep data sorted, which leads to higher contention. Logging is the bottleneck here, as multiple directory entries share the same log. Even though the logs use a lock-free atomic counter to assign slots, the CPU's cache coherency protocol adds some overhead. We have found that for our system with 24 cores the best trade-off between synchronization overhead and space consumption is 64 logs.

Lookups do not acquire any locks but use optimistic concurrency control. Therefore, values, keys, epoch, and size have to be atomic variables. Plush reads a bucket's epoch before searching for a record's key. After it finds the key, it re-reads and compares the epoch . If the epochs match, Plush can be sure that the bucket has not been migrated and overwritten during *lookup*. This design allows for multiple lookups and up to one insert to operate on a directory entry concurrently.

5.6 Crash Consistency

As long as the records are still on DRAM, crash consistency is guaranteed: Write-ahead logging ensures that entries are recoverable before Plush inserts them into DRAM. If another thread sees the key in DRAM, it is thus guaranteed to be recoverable.

It is more complex during migration. Plush ensures that there is no inconsistent state being read by making migrated values visible after it persisted them and *only then* removing the old version at the previous level. This approach saves a lock and a persistency barrier but allows for a small time window where an entry is valid on both levels simultaneously. If the system crashes here, records are duplicated. This duplication is, however, not an issue. Since Plush's append-only architecture forces it to deal with and merge multiple records having the same key anyway, it will merge the duplicates when it migrates them to the next level. We thus trade off cheap consistency guarantees for the small risk of some temporary data duplication after a crash. Since *inserts* guarantee that Plush is always in a valid state and data is duplicated at worst, Plush does not need any additional consistency checks during *lookups*.

5.7 Bulk Loading

Some workloads do not need the strict persistency guarantees given by Plush but would benefit from increased throughput. Take bulk loading as an example: A user has to insert some existing large data set, but a (rare) crash is not world-ending as they can restart the bulk loading process. Here, it would be sufficient that the user can define checkpoints after which all records that a user has inserted before are guaranteed to be persisted (i.e. at the end of bulk loading) instead of having a per-insert persistency guarantee.

Plush can increase insert throughput by disabling write-ahead logging. When disabled, the persistency guarantees weaken as records living in the DRAM hash table are now lost on a crash. Plush mitigates this by allowing the user to create checkpoints manually. Checkpointing moves all records from the DRAM table to the first PMem level. It even supports mixing regular and relaxed persistency inserts: The user can decide upon each insertion if Plush should log the record. Section 6.5 evaluates bulk loading.

6 EVALUATION

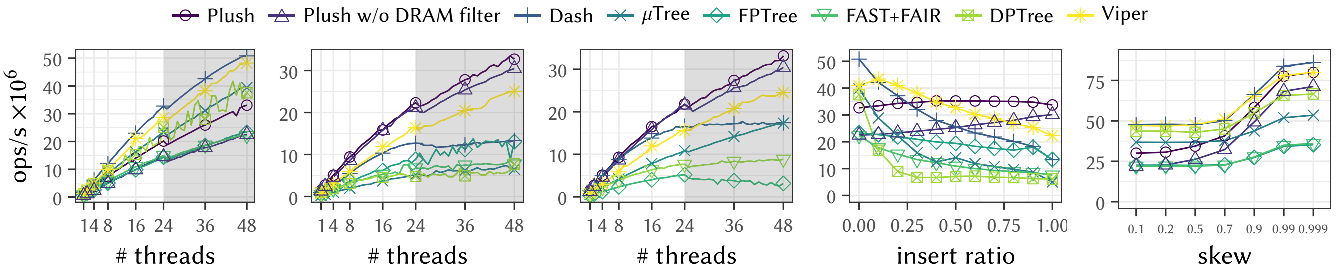
In this section, we compare Plush against other persistent trees and hash tables optimized for fixed- and variable-length records. We also evaluate the impact of different optimizations. Finally, we examine Plush's space utilization and recovery performance.

Our evaluation system is equipped with an Intel Xeon Gold 6212U CPU, with 24 physical (48 logical) cores and clocks at a 2.4 GHz base clock. The system has access to 192 GB (6×32 GB) DRAM and 768 GB (6×128 GB) of Intel's first-generation DCPMM DIMMs. It runs Ubuntu 20.04 LTS with kernel version 5.4.0. We configure the DCPMM in *AppDirect* mode and provision it in *fsdax* mode with an *ext4* filesystem. We manage via load and store instructions on an *mmap*'d memory regions and do not use any additional libraries.

We implement Plush in C++20 and make use of AVX-512 instructions. To our knowledge, all CPUs supporting PMem also support the AVX-512 instruction set. We compile Plush and all other data structures with GCC 11.1.0 with the `-O3` flag. If not otherwise mentioned, we use the PiBench benchmarking suite by Lersch et al. [27]. It was designed for persistent *tree* indexes but also supports all workloads applicable to Plush. Using a third-party benchmarking tool ensures our benchmarks do not accidentally favor Plush.

6.1 Plush Configuration

Let us first discuss how to best configure Plush depending on workload characteristics. Assume we have N records and a fanout f. Each directory entry thus holds up to $16f$ records. Further, assume that the hashing is not perfectly uniform, so migrating a batch of $16f$ records to the next level writes to two buckets on average (cf. Section 4.1). Thus a migration incurs $2 \cdot 16f$ bucket writes and $16f$ filter updates, so a media write amplification of 3 per record. Due to the write combining buffer, we get away with a write amplification of 1 for the log. Since log compaction is free (cf. Section 4.2), there is no additional amplification. We thus expect a per-record write amplification of $3 \cdot \log_f N + 1$. We thus want to choose the fanout as large as possible to minimize level count and thus write amplification while still having acceptable directory size spikes when Plush adds a new level. A high fanout also reduces read amplification as fewer levels have to be searched per lookup.



(a) 100% lookup **(b) 100% insert** **(c) 100% delete** **(d) Mixed workload** **(e) Lookups under skew**

Figure 5: Throughput of core operations under varying thread count for fixed-size records (8-byte keys, 8-byte values).

Table 2: Investigated Data Structures.

Name	Type	Var-length records	DRAM	Range qs
Plush	LSM+ht	✓	$O(1)$	✓[2]
μTree	tree	just values	$O(n)$	✗
FPTree	tree	✗	$O(n)$	✓
FAST+FAIR	tree	✗	$O(n)$	✓
DPTree	tree	✗	$O(n)$	✓
Dash	ht	just keys	-	✗
PmemKV	ht	✓	-	✗
Viper	ht	✓	$O(n)$	✗
RocksDB	LSM	✓	$O(1)$	✓
FASTER	ht+log	✓	$O(1)$	✗

We choose $f = 16$ resulting in 2^{16} DRAM directory entries and the same amount of PMem directory entries at level 1. With this configuration, Plush has a directory sized 32 MiB, 544 MiB, 8.7 GiB, and 147 GiB at 1, 2, 3, and 4 levels holding $3.3{\times}10^7$, $3.0{\times}10^8$, $4.5{\times}10^9$, and 7.3×10^{10} records, respectively. Plush uses 64 recovery logs, each with six chunks of 5 MiB, the empirically determined sweet spot between minimizing contention (more logs are better) and maximizing write combining buffer usage (fewer logs are better).

6.2 Comparison to Other Data Structures

We compare Plush against nine indexes listed in Table 2 which can be grouped into *hash tables* and *tree-like data structures*. Plush combines aspects of all those approaches allowing us to compare different trade-offs made by each approach. We also compare two versions of Plush. One, where the filters for the top two levels are stored in DRAM (cf. Section 5.2), and one, where all filters are stored on PMem. As the DRAM overhead of the filters is just 256 MiB, we treat this case as the default case if not otherwise mentioned. Dash [32] and PMemKV[3] with the cmap backend are PMem-only hash tables. Both support the same operations as Plush but do not use any DRAM. PMemKV also supports variable-length records, while Dash only supports (pointers to) variable-length keys. Viper [3] stores its records in PMem but keeps a hash table with fingerprints of all keys in DRAM, thus requiring large amounts of DRAM linearly growing with the number of records stored.

FPTree[4] [37], FAST+FAIR [21], DPTree [52], and μTree [9] are persistent B-Trees, storing inner nodes (e.g., FPTree) or keys (e.g., μTree) on DRAM. As they sort their records at the cost of some insert throughput, most support range queries in contrast to the hash tables. The DRAM consumption of all trees and Viper grows with the record count. The DRAM consumption of all other hash tables and Plush is either constant in the record count or zero for the PMem-only indexes.

FASTER [7] is a log-based kv-store designed for SSD and gives weaker persistency guarantees. We place the persistent part of its log on PMem. We configure FASTER to have the same amount of DRAM available as Plush, with half reserved for its hash index and the other half for the mutable part of its log. PMem-RocksDB[5] is a fork of *RocksDB* optimized for PMem. It serves as a representation of all established LSM trees that were adapted to PMem.

6.3 Fixed-Size Records

First, we evaluate how Plush deals with fixed-size records. We preload 100 million 16-byte records consisting of 8-byte keys and values. Keys are uniformly distributed. We then execute 100 million operations for varying thread counts. Figure 5 shows the results.

Throughput. For lookups (a), Viper and Dash win over Plush and the tree data structures as both have just a single level. Plush has to look up the key at every level on average as it stores most records on the last level and therefore cannot end the search early. For lookups, it thus behaves like a tree. It still has an advantage (×1.41) over FPTree but is beaten by μTree and DPTree which store a copy of all keys in DRAM. Plush beats the other trees because of its hybrid design where it optionally stores bloom filters in DRAM (cf. Section 6.5), speeding up negative level lookups. Without this optimization, Plush has a similar lookup performance to those trees.

Plush's tiered design benefits from *temporal skew*: If Plush looks up a key that was recently inserted, it will probably still be in DRAM or a low PMem level: It can end the search early. Figure 5e illustrates this advantage. Here, lookups are Zipf-distributed [14]. The higher the skew factor, the more likely a lookup is for a record that was inserted just recently. While all data structures profit from skew due to caching effects, Plush profits disproportionally more, catching up with Viper and nearly reaching Dash.

Plush outperforms all data structures for inserts (b), scaling nearly linearly up to 48 threads. It improves over Viper by 1.44×, Dash by 2.44×, and FPTree, the fastest tree, by 3.31×. The picture

[2] In range partitioning mode, see Section 6.6.
[3] https://pmem.io/pmemkv/
[4] We use SFU's open-source re-implementation: https://github.com/sfu-dis/fptree
[5] https://github.com/pmem/pmem-rocksdb

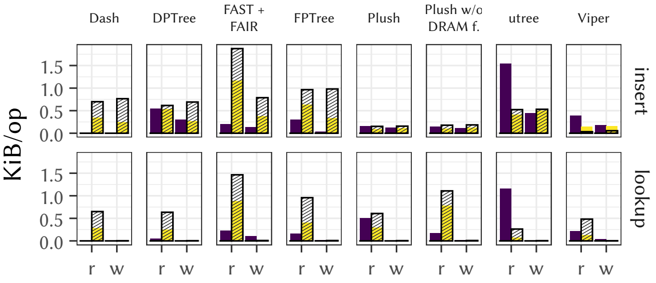


Figure 6: Read and write amplification for 16-byte records on DRAM (▮) and PMem (🟨). Overlayed hatched columns (▨) show PMem media amplification.

looks similar for deletes (c), where Plush's performance is identical to inserts as a delete is just an insert of a tombstone record.

Figure 5d shows how the data structures behave for mixed workloads on 48 threads ranging from only lookups (left) to only inserts (right). For insert ratios below 30%, Viper has higher throughput than Plush as it can leverage its superior lookup throughput below those insert ratios. Above that, Plush's insert throughput dominates.

Overall, Plush is very *predictable*: It shows stable throughput at all insert ratios, scales well for all operations due to its partitioning design, low write amplification, lock-free lookups, and profits disproportionally from temporal skew.

Read/write amplification. Figure 6 explains the throughput differences. It shows how much data is read/written per operation by the CPU at *cache line granularity* (64 bytes) on average. The overlayed hatched columns show data reaching the physical storage medium at *PMem block granularity* (256 bytes). The difference between the yellow and the hatched column is thus the overhead incurred by not packing writes and reads into 256B-blocks perfectly.

Plush's batching of reads and writes with a DRAM buffer decreases amplification on inserts: Its read and write amplification is just 63% resp. 70% of Viper, the runner-up. Viper, however, has *lower* end-to-end write amplification, as it *always* writes to PMem sequentially. This is the optimal write pattern for PMem, but it comes at a cost: Viper needs to keep a large hash table in DRAM that indexes all records, significantly increasing DRAM read- and write amplification. This experiment confirms that DRAM latency is not negligible for inserts, supporting our earlier observation that PMem's write latency is similar to DRAM's. Plush, Viper, DPTree, and Dash have similar PMem read amplifications for lookups, but Plush is at a disadvantage as it also has to read from DRAM. Even though μTree reads 56% less from PMem than Plush, its throughput is only 18% higher as it reads over twice as much data from DRAM.

This experiment confirms our initial assumption that the throughput of PMem data structures is heavily influenced by their write amplification, as write bandwidth is a bottleneck of PMem.

Latency. Plush's weakness is its migration step. To increase throughput by reducing communication overhead, Plush does *not* employ separate background migrator threads. Thus, a single "unlucky" insert might have to move a lot of data since a migration can start a chain of recursive migrations leading to higher tail latencies. We investigate the impact of this design decision on latency. For this, we run our workload on 23 threads (and one monitor thread) to rule out any SMT effects. Figure 7 shows the latency at different percentiles. Plushs insert latency is close to Dashs at lower percentiles. At the 99.9-percentile, Plush's latency increases sharply. This increase is expected, as on average, every 256th insert triggers a migration. However, Plush's latency does not worsen significantly at higher percentiles, and it even outperforms Dash again. Plush's and the other tree-like data structures' advantage is that they do not have to do any rare but costly structural modification operations like directory splits. For lookups, all data structures behave similarly.

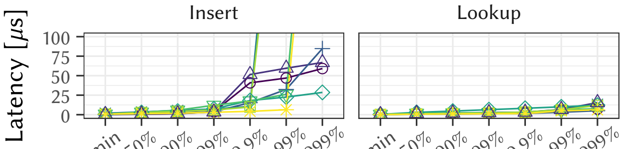


Figure 7: Tail latencies for 16-byte records with 23 threads.

6.4 Space Utilization

We fill all data structures with 16-byte records until we run out of storage space on our PMem partition , out of DRAM, or just crash. We then plot the total data set size (i.e., record count × 16 bytes) against the actual space consumption. Figure 8 shows the results for DRAM consumption (left) and overall space consumption (right).

DRAM consumption for both Plush variants and Dash is constant, with Plush using below 1 GB of DRAM and Dash not using DRAM at all. Viper, μTree, and DPTree store the bulk of their data in DRAM and are thus limited by DRAM capacity and cannot scale with the amount of installed PMem. They instead run out of DRAM or crash at $\approx$ 70 GiB inserted records, which can also be seen on the right.

Regarding overall space consumption, both Plush variants show a huge increase at $\approx$ 50 GiB when they create the directory for PMem level 4 (directories for levels 1-3 are too small to be visible here). The space consumption then shows a stair pattern: Plush migrates all buckets to the next level whenever a directory entry is full. Since the hash function is uniform, this happens roughly at the same time for all directory entries. This leads to many new bucket allocations at the last PMem level at once. After Plush has migrated most entries of a level to the next level, the previous level's buckets are empty again. Since each PMem level is 16 times the size of the previous level, the stair pattern repeats 16 times before Plush starts a new level. While the trees show a more linear growth, Dash has similar jumps in size as it doubles the directory whenever full.

6.5 Plush Tuning

As explained in Sections 4.1, 5.2, and 5.7, Plush supports multiple optimizations trading off DRAM consumption, PMem consumption, or persistency guarantees for throughput. Figure 9 compares the impact of the different optimizations compared to the baseline.

Pre-allocation. When enabled (`+prealloc`), all buckets up to the second PMem level are pre-allocated upon initialization. This setting does not increase memory consumption when enough records are inserted (as all levels except the last level are full anyway). However, it increases insert throughput (×1.36) and marginally increases

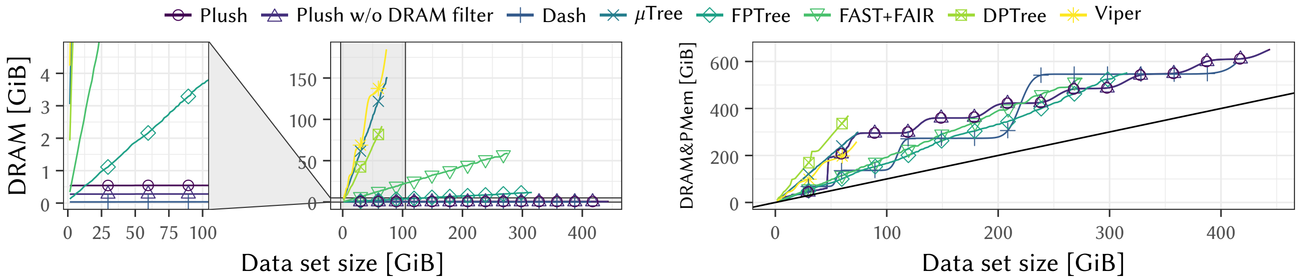


Figure 8: Storage consumption compared to data set size. Left: Just DRAM, Right: DRAM and PMem combined.

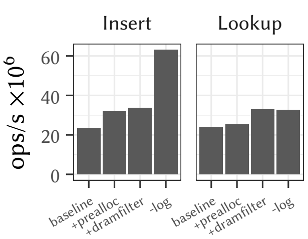


Figure 9: Throughput gained by optimizations.

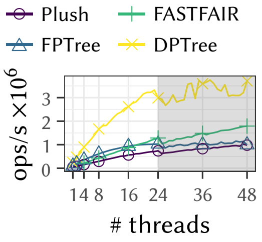


Figure 10: Throughput for range queries.

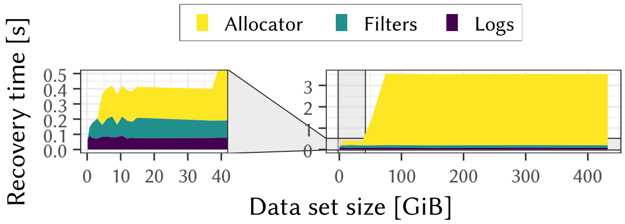


Figure 11: Recovery time vs. data set size for filters and logs.

lookup throughput (×1.05). Inserts are disproportionally faster as every migration saves `fanout` pointer dereferences while reading the records and ≈ fanout · 2 pointer dereferences for storing records on the next level. For our experiments, we pre-allocate up to the second PMem level which reserves 4.26 GiB PMem for the buckets.

Filters on DRAM. When additionally storing the filters of the first two levels in DRAM (`+dramfilter`), insert throughput improves slightly (×1.05) while lookup throughput improves considerably (×1.30). Here, lookups improve disproportionally as checking if a level contains a key no longer involves a costly PMem block read. As write latency is lower and inserts batch writes to PMem filters anyway, inserts do not benefit.

Skipping logging. When additionally not logging inserts (`-logs`), insert throughput increases dramatically (×1.89) while lookup throughput stagnates as lookups log nothing. As explained in Section 5.7, disabling logging is helpful for bulk loading. Creating a checkpoint with a single thread takes ≈ 840 ms. Plush also supports creating checkpoints concurrently. This is useful when Plush currently does not run other operations, e.g., after a bulk load before accepting requests. With 32 threads, checkpoint creation takes ≈ 69 ms.

6.6 Range Queries

By default, Plush partitions the key space into disjunct directory entries *by hash* to avoid skew. However, it supports arbitrary partitioning functions. When choosing a range partitioning function, Plush supports range queries: While records within a directory entry's buckets are still unsorted, it can use *divide and conquer* to only iterate over a few such entries. This is prone to skew but is an advantage over pure hash tables. Figure 10 shows Plush's range query performance in range partition mode. The workload consists of a data set with 100 million records and 100 million lookups of random keys for which the next 100 larger records are returned in order. Plush keeps up with FPTree but is outperformed by the other trees as they can scan their sorted leaf nodes while Plush still has to check all unsorted buckets with potential candidates.

6.7 Recovery

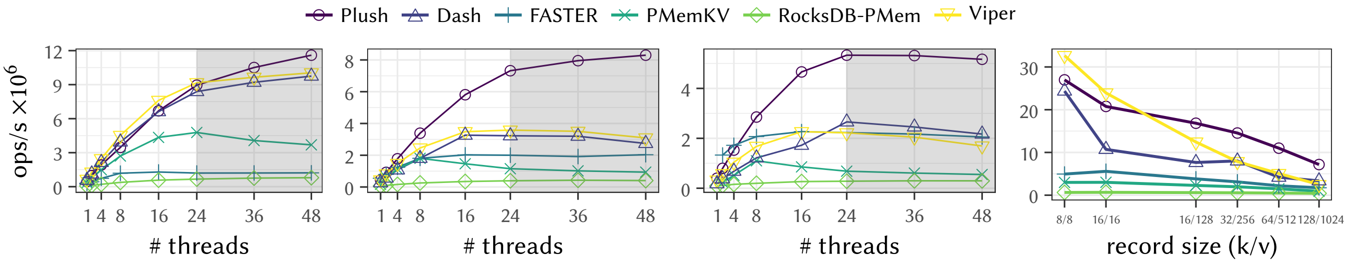
While crashes in a production system should be rare, Plush should still recover quickly to keep downtime low. We load an increasing number of 16-byte records, forcibly terminate Plush and then measure the recovery time. Figure 11 shows the results.

Plush recovers three types of data: (1) Records in the PMem log that had not been persisted in the hash table, (2) bloom filters that had been stored on DRAM (cf. Section 6.5), and (3) the status of the allocator (i.e., until which address it already had allocated buckets).

(1) is constant as the log has a fixed size, (2) is constant as Plush only allows DRAM filters up to a given level (2, in this case) and (3) is linear in the number of levels (so logarithmic in the number of records). Note that this is the worst case as recovery time only grows with the number of records, not the data set size: If storing records larger than 16 bytes, Plush does not read the keys and values themselves during recovery, but just the pointers referencing them.

6.8 Variable-Length Records

Figure 12 shows throughput for records having 16-byte keys and 1000-byte values. We evaluate a read-heavy (a), a mixed (b), and a write-heavy workload (c) with varying thread count. Plush scales well for read-heavy workloads but does not scale beyond 24 threads for write-heavy workloads. This observation aligns with our earlier findings that PMem's write bandwidth can be saturated by just a few threads. The more inserts we issue, the more bandwidth-starved all data structures become. For read-heavy workloads, Viper and Dash keep up with Plush (a). For low thread counts on an insert-heavy workload, FASTER overtakes Plush. Since FASTER gives



(a) Read-heavy (10% **inserts**) **(b) Mixed** (50% **inserts**) **(c) Write-heavy** (90% **inserts**) **(d) Varying record size**

Figure 12: Throughput for variable-length entries. (a) - (c) show throughput for 3 different workloads with 16-byte keys and 1000-byte values, (d) shows throughput for varying record sizes under a mixed workload (50% **inserts).**

fewer persistency guarantees, it may store records only in DRAM and flush them in bulk. This is an advantage over Plush, which keeps a write-ahead log. All data structures beat the RocksDB fork showing the advantage of designing a data structure optimized for PMem from the ground up instead of adding it as an afterthought. Figure 12d shows how throughput for the mixed workload changes with varying record sizes. Plush is latency-bound for smaller sizes but becomes bandwidth-limited as the record size increases. Plush profits from its low write amplification as it stores records out of place and from pre-faulting the payload log. Out-of-place storage is also the reason for the drop in throughput from 16-byte records to 32-byte records as Plush stores the latter records out of place, leading to an additional level of indirection.

7 RELATED WORK

PMem's low bandwidth (compared to DRAM) has been researched extensively [4, 11, 20, 23, 29, 44]. Plush specifically addresses issues raised by Gugnani et al. [15] and Woo et al. [44]. Most data structures mitigate this by adapting a hybrid design where reconstructible data resides in DRAM [3, 9, 19, 37, 49, 52]. Such data structures are usually B-Trees that store inner nodes inside DRAM (e.g., NV-Tree [49] or FPTree [37]), or even store (fingerprints of) keys in DRAM (e.g., μTree [9]). Viper [3] and HiKV [46] are hybrid hash tables. Viper logs records to PMem and stores their fingerprints in a DRAM hash table for efficient lookups and synchronization between threads. HiKV combines a PMem hash index with a DRAM B-Tree. These approaches do not have control over how much DRAM they consume as it depends on the data set size. LibreKV [30] uses a fixed-size hash table on DRAM like Plush but does not optimize for write amplification. Multi-tiered buffer managers [28, 41, 51] also make use of DRAM. PMem-only data structures do not consume any DRAM [1, 8, 18, 32, 34, 35, 53]. They are great for lookups but suffer from high write amplification during inserts. LB+-Tree [31] and write-optimized skip lists [47] minimize write amplification, but are PMem-only and thus have only limited possibilities to reach that goal. Plush straddles the line by using a *fixed* amount of DRAM.

Most approaches to making LSM trees PMem-aware use an existing LSM tree like LevelDB as a foundation [6, 24, 25, 48]. They are, therefore, limited by LevelDB's architecture which was not designed with PMem in mind. NVLSM [50] is built for PMem from the ground up. Plush is also inspired by approaches employed by LSM trees not originally meant for PMem: Its value separation approach was initially proposed by WiscKey [33] and has since been adapted by other LSM trees like Parallax [45] or HashKV [6]. HashKV also inspired Plush's approach to partition its logs by hash. Haubenschild et al. propose global sequence numbers [16].

Plush's logging approach was invented by Mnemosyne [43] and further refined by our previous work [42]. PiBench [27] is a benchmarking framework for persistent indexes that we also used for our evaluation. Hu et al. adapted it for hash indexes [17].

8 CONCLUSION

We have presented Plush, a write-optimized data structure for persistent memory. It employs techniques popularized by LSM trees to minimize write amplification and adapts them to PMem by replacing sorted runs with leveled hash tables and optimizing for 256-byte blocks. Because of Plush's low write amplification, it has higher insert throughput than existing PMem-optimized hash tables (2.44× of Dash) while having a lookup throughput comparable to fast tree-like PMem data structures. This confirmed our initial hypothesis that PMem data structures are often bandwidth-limited. Plush profits from temporal skew and excels at write-heavy workloads as its throughput stays constant even at high insert ratios.

9 FUTURE WORK

As Plush's DRAM consumption is independent of the data set size, Plush is arbitrarily scalable and only limited by PMem capacity. Since only 3 TiB of PMem can be installed per socket, scaling is limited in practice. We intend to extend Plush to scale to SSD by putting higher levels of the hash table on SSD. Each directory entry currently points to up to 4 KiB of unsorted records. Plush could sort those during migration and write them to an SSD page. During future migrations, Plush could merge values into this sorted run which is the root of a conventional LSM tree. With growing data set size, more and more data would live in this LSM forest with performance gracefully declining to SSD speed.

ACKNOWLEDGMENTS

We thank Kaisong Huang who helped us with debugging performance issues with FPTree. This work was partially funded by the German Research Foundation (DFG) in the context of the project "Scaling Beyond DRAM with PMem without Compromising Performance" (KE401/22-2) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037).

REFERENCES

[1] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB* 11, 5 (2018), 553–565.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *SIGMETRICS*. ACM, 53–64.

[3] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *PVLDB* 14, 9 (2021), 1544–1556.

[4] Maximilian Böther, Otto Kißig, Lawrence Benson, and Tilmann Rabl. 2021. Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs. In *DaMoN*. ACM, 7:1–7:8.

[5] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*. ACM, 2:1–2:8.

[6] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *USENIX Annual Technical Conference*. USENIX Association, 1007–1019.

[7] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD Conference*. ACM, 275–290.

[8] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797.

[9] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. ?Tree: a Persistent B+-Tree with Low Tail Latency. *PVLDB* 13, 11 (2020), 2634–2648.

[10] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *HotStorage*. USENIX Association.

[11] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *SIGMOD Conference*. ACM, 339–351.

[12] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.* 4, 3 (1979), 315–344.

[13] Sanjay Ghemawat and Jeff Dean. 2011. *LevelDB*. Retrieved 2022-07-13 from https://github.com/google/leveldb

[14] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD Conference*. ACM Press, 243–252.

[15] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *PVLDB* 14, 4 (2020), 626–639.

[16] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD Conference*. ACM, 877–892.

[17] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *PVLDB* 14, 5 (2021), 785–798.

[18] Jing Hu, Jianxi Chen, Yifeng Zhu, Qing Yang, Zhouxuan Peng, and Ya Yu. 2021. Parallel Multi-split Extendible Hashing for Persistent Memory. In *ICPP*. ACM, 48:1–48:10.

[19] Chenchen Huang, Huiqi Hu, and Aoying Zhou. 2021. BPTree: An Optimized Index with Batch Persistence on Optane DC PM. In *DASFAA (3) (Lecture Notes in Computer Science)*, Vol. 12683. Springer, 478–486.

[20] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications on Persistent Indexes. In *11th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, California, January 9-12, 2022, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2022/papers/p64-huang.pdf

[21] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *FAST*. USENIX Association, 187–200.

[22] Intel. 2022. *Intel Optane DC persIstent Memory Data sheet*. Retrieved 2022-07-13 from https://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf

[23] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019).

[24] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *FAST*. USENIX Association, 191–205.

[25] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *USENIX Annual Technical Conference*. USENIX Association, 993–1005.

[26] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[27] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *PVLDB* 13, 4 (2019), 574–587.

[28] Lucas Lersch, Wolfgang Lehner, and Ismail Oukid. 2019. Persistent Buffer Management with Optimistic Consistency. In *DaMoN*. ACM, 14:1–14:3.

[29] Haikun Liu, Di Chen, Hai Jin, Xiaofei Liao, Binsheng He, Kan Hu, and Yu Zhang. 2021. A Survey of Non-Volatile Main Memory Technologies: State-of-the-Arts, Practices, and Future Directions. *J. Comput. Sci. Technol.* 36, 1 (2021), 4–32.

[30] Hao Liu, Linpeng Huang, Yanmin Zhu, and Yanyan Shen. 2020. LibreKV: A Persistent in-Memory Key-Value Store. *IEEE Trans. Emerg. Top. Comput.* 8, 4 (2020), 916–927.

[31] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *PVLDB* 13, 7 (2020), 1078–1090.

[32] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *PVLDB* 13, 8 (2020), 1147–1161.

[33] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1 (2017), 5:1–5:28.

[34] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *FAST*. USENIX Association, 1–16.

[35] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *FAST*. USENIX Association, 31–44.

[36] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

[37] 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 *SIGMOD Conference*. ACM, 371–386.

[38] Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM J. Exp. Algorithmics* 14 (2009).

[39] Tobias Schmidt, Maximilian Bandle, and Jana Giceva. 2021. A four-dimensional Analysis of Partitioned Approximate Filters. *PVLDB* 14, 11 (2021), 2355–2368.

[40] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *SIGMOD Conference*. ACM, 217–228.

[41] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD Conference*. ACM, 1541–1555.

[42] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *VLDB J.* 29, 6 (2020), 1223–1241.

[43] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *ASPLOS*. ACM, 91–104.

[44] Youngjoo Woo, Taesoo Kim, Sungin Jung, and Euiseong Seo. 2021. Analysis and Optimization of Persistent Memory Index Structures' Write Amplification. *IEEE Access* 9 (2021), 167687–167698.

[45] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores. In *SoCC*. ACM, 305–318.

[46] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX Annual Technical Conference*. USENIX Association, 349–362.

[47] Renzhi Xiao, Dan Feng, Yuchong Hu, Fang Wang, Xueliang Wei, Xiaomin Zou, and Mengya Lei. 2021. Write-Optimized and Consistent Skiplists for Non-Volatile Memory. *IEEE Access* 9 (2021), 69850–69859.

[48] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Kenry Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *PVLDB* 14, 10 (2021), 1872–1885.

[49] Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. 2016. NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory. *IEEE Trans. Computers* 65, 7 (2016), 2169–2183.

[50] Baoquan Zhang and David H. C. Du. 2021. NVLSM: A Persistent Memory Key-Value Store Using Log-Structured Merge Tree with Accumulative Compaction. *ACM Trans. Storage* 17, 3 (2021), 23:1–23:26.

[51] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD Conference*. ACM, 2195–2207.

[52] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *PVLDB* 13, 4 (2019), 421–434.

[53] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *OSDI*. USENIX Association, 461–476.