Tair-PMem: A Fully Durable Non-Volatile Memory Database

Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Long Qian, Rui Chen, Jiang Qi, Ruo Wang, Guoyun Zhu, Chenghu Yang, Wei Zhang, Feifei Li
{caixin.gcx, tianchengjin.tcj, zhengheng.wzh, sh.wang, xiyu.wxy, mobing.fql, qinwu.qw, qianlong.ql, chenrui_c.pt, qijiang.qj, wangruo.wr, zongdai, yexiang.ych, zwei, lifeifei}@alibaba-inc.com
Alibaba Group

ABSTRACT

In-memory databases (IMDBs) have been the backbone of modern systems that demand high throughput and low latency. Because of the cost and volatility of DRAM, IMDBs become incompetent when dealing with workloads that require large data volume and strict durability. The emergence of non-volatile memory (NVM) brings new opportunities for IMDBs to tackle this situation. However, it is non-trivial to build an NVM-based IMDB, due to performance degradation, NVM programming complexity, and other challenges. In this paper, we present *Tair-PMem*, an NVM-based enterprise-strength database atop Redis, the most popular IMDB. *Tair-PMem* adopts a well-controlled data layout and a *log-as-user-data* design to mitigate NVM overheads. It eases the NVM programming complexity by providing a hybrid memory programming toolkit. To better leverage the enterprise-strength features and implementations from Redis, *Tair-PMem* retrofits it in a less intrusive way to achieve full compatibility and stability, while retaining its advanced features. With all of the above techniques elaborately implemented, *Tair-PMem* satisfies full durability, high throughput, and low latency at the same time. *Tair-PMem* has now been publicly available as a cloud service on Alibaba Cloud. To the best of our knowledge, *Tair-PMem* is the first cloud service that makes good use of the persistence capability of NVM.

PVLDB Reference Format:
Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Long Qian, Rui Chen, Jiang Qi, Ruo Wang, Guoyun Zhu, Chenghu Yang, Wei Zhang, Feifei Li. Tair-PMem: A Fully Durable Non-Volatile Memory Database. PVLDB, 15(12): 3346 - 3358, 2022.
doi:10.14778/3554821.3554827

1 INTRODUCTION

In-memory databases (IMDBs) [4, 26, 32, 37] have been playing a vital role in various applications, such as e-commerce services, web services, and advertisements. They help to accelerate data access by caching frequently accessed data in memory. Among different IMDBs, Redis [37] is one of the canonical choices in most scenarios, due to its high performance, simplicity, and ease of use. To further meet enterprise customers' needs for IMDBs, Alibaba Cloud offers the enterprise-strength in-memory database service, called Tair [10], which is initially compatible with both Redis and Memcached [32] and has now been extended with graph and relational interfaces. The Tair-for-Redis service enhances the original Redis with many advanced features, such as read-write separated architecture and real-time hotspot diagnostics [5]. Tair for Redis has been extensively used by both cloud customers and Alibaba's internal businesses. For instance, it helps Alibaba's e-commerce platform to sustain sub-millisecond access latency and billions-of-QPS peak throughput during the Double 11 Global Shopping Festival, offering smooth shopping experiences to hundreds of millions of consumers.

However, since Redis and all other IMDBs are built on top of DRAM, they inherently suffer from several limitations that hinder their broader usage beyond a caching system. First, DRAM is the most expensive storage medium in terms of per GB price. The cost of DRAM makes it expensive to expand the capacity of IMDBs for the ever-growing volumes of application data. Second, due to the volatility of DRAM, IMDBs can hardly by themselves handle those scenarios demanding strict data durability. Though some IMDBs, such as Redis, do provide basic data persistence capability (*e.g.*, via log and checkpoint), the performance usually slumps when the full durability option is on. Hence, IMDBs are usually backed by a separate persistent storage system, such as MySQL [34] and Cassandra [2], when limited budget and full durability are considered. In this case, the overall performance deteriorates significantly by heavy disk I/Os from the back-end system.

Fortunately, the emergence of byte-addressable non-volatile memory (NVM) brings new opportunities for IMDBs to resolve the above issues. From academia, many NVM-oriented designs have been proposed to IMDBs [16, 20, 30, 44, 49] as well as other systems [14, 21, 23, 45, 46, 48]. From industry, the first commercial NVM product, Intel® Optane™ Persistent Memory (Optane PMem) [18], already offers appealing characteristics such as high performance, large capacity, and low cost. Optane PMem stimulates database practitioners to build and offer enterprise-strength NVM-backed IMDB services.

In this paper, we target the problem of enhancing Redis towards an enterprise-strength NVM-backed IMDB. We observe that it is far from trivial to build such a system, and the major challenges are three-fold: (1) *Performance degradation.* Optane PMem has 3× higher read latency and more than 10× lower bandwidth compared with DRAM [47]. It is challenging to support full durability while sustaining comparable performance against original Redis. (2) *NVM programming complexity.* NVM programming is hard and error-prone [13, 24]. Prior works [15, 24, 25, 31, 42] propose many libraries or frameworks to convert the volatile data structures to NVM-backed ones. They help to reduce the engineering complexity but introduce either high overheads (on running time and memory footprint) [15, 24, 31] or consistency compromises [15, 25, 42]. It is

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554827

challenging to hide NVM programming complexities while preserving similar behaviors as on DRAM. (3) *Stability with full Redis compatibility.* Existing works often consider one specific index structure on NVM, *e.g.*, a hash [8, 30, 33, 39, 50] or a tree [3, 6, 17, 29, 35, 41, 49]. In practice, Redis provides a set of rich data models with more than 300 APIs [38] in total. For example, it supports index structures like hashes, lists, skip-lists, and trees, as well as database features like checkpoint and transaction. An exhaustive NVM-oriented retrofit for all of them would be error-prone and time-consuming. Maintaining high stability by reducing engineering complexity during the retrofitting process is a practical challenge.

To address the above challenges, we propose *Tair-PMem*, an enterprise-strength NVM-based IMDB that achieves both full durability and full Redis compatibility. It has been commercialized on Alibaba Cloud [9]. From its core, *Tair-PMem* adopts the following major designs.

(1) A well-controlled data layout that mitigates NVM overheads. To reduce the access and persistence overhead of NVM, *Tair-PMem* avoids placing all Redis' data into NVM. Instead, it applies a DRAM-NVM hybrid data layout, following a carefully weighed data placement strategy. This strategy determines the location (*i.e.*, DRAM or NVM) of distinct data categories (*e.g.*, user data, index, metadata) according to their different demands on durability, access frequency, and volume. For example, as the durability of user data is a must, we persist it in NVM. In contrast, since the allocator metadata is frequently accessed and small in size, we pin it in DRAM. To better fit the access characteristics of NVM, we adopt the *log-as-user-data* design on it. In a nutshell, the log entries on NVM also play the role of user data, which are directly accessed by user queries through upper-layer indexes. Consequently, the user data only needs to be written once, lowering the occupation of NVM bandwidth.

(2) A lightweight hybrid memory programming toolkit that encapsulates NVM programming complexity. The toolkit is designed specifically for performance-critical IMDBs with lightweight and efficient features, which contains two components, *allocator* and *log & data pool*. The former manages both DRAM and NVM, and enables developers to allocate space following the classic malloc-and-free style. It supports recovery but does not request persisting any data, thus obtaining efficient performance. The latter provides lightweight transaction semantics that ensures all data stored is durable and atomic, thus hiding the complex NVM programming details for data persistence.

(3) A set of database components that achieve full Redis compatibility and advanced features in a low-intrusive manner. To provide an enterprise-strength product, we prefer to utilize outstanding designs and implementations from Redis as much as we can. *Tair-PMem*'s retrofits are done in modulized database components, such as *codec*, *checkpoint* and *recovery* modules, with low code intrusion. Hence, Redis' index structures are completely untouched, making it extremely simple to be compatible with various data models. Furthermore, *Tair-PMem* utilizes NVM's persistence capability to significantly enhance fundamental database features. For instance, by abandoning Redis' log, *i.e.* append only file (AOF), and alternatively adopting an instant checkpoint mechanism, *Tair-PMem* resolves the periodic latency spike issue in Redis. Based on this mechanism, *Tair-PMem* further provides an extremely fast recovery process after a normal shutdown.

To sum up, our main contributions are as follows:

- We pioneer the problem of enhancing Redis with NVM capabilities and build an enterprise-strength NVM-backed IMDB service called *Tair-PMem* on Alibaba Cloud. To the best of our knowledge, *Tair-PMem* is the first commercial in-memory cloud database service that makes good use of the NVM's persistence capability.
- We propose a suite of vital designs to address the three major challenges (*i.e.*, performance degradation, NVM programming complexity, and stability with full Redis compatibility) encountered during the development of *Tair-PMem*.
- Compared to Redis, we show in experimental evaluation that *Tair-PMem* sustains comparable throughput and avoids periodic latency spikes. To illustrate, the maximum latency of fully durable *Tair-PMem* is 219× lower than that of fully durable Redis and 38× than that of partially durably one.

The rest of this paper is organized as follows. Section 2 discusses background and challenges. Section 3 overviews *Tair-PMem*'s core design concepts and architecture. The hybrid memory programming toolkit is covered in Section 4 and 5, and the core database components are covered in Section 6. Section 7 discusses NVM programming skills used during *Tair-PMem*'s development. Section 8 evaluates *Tair-PMem* against other baselines. Section 9 summarizes the related work and Section 10 concludes the paper.

2 BACKGROUND

In this section, we first briefly introduce Redis and NVM, then discuss the challenges of integrating NVM into Redis.

2.1 Redis

Redis (Remote Dictionary Server) [37] is a widely used IMDB that provides high performance, advanced key-value abstraction, and optional data durability. It adopts a single-threaded processing model where all requests from clients are queued and executed sequentially. This model avoids complex concurrency controls and hence utilizes the CPU resources more efficiently. It allows Redis to easily support rich data models, *i.e.*, keys in Redis are always string objects while values could be complex models such as String, Hash, List, Set, and ZSet (Sorted Set). Redis supports atomic operations on all these data types, *e.g.*, inserting many elements into a set.

Redis supports different levels of data durability via command logging or point-in-time checkpoints. They are implemented by the following two approaches, respectively. (1) Append only file (AOF). It records write commands in an append-only manner, and data can be recovered by replaying the log. When it is oversized, the log is converted to a compact one from the latest snapshot. (2) Redis database (RDB). It is a very compact file generated by serializing the latest snapshot. Note that both approaches rely on the snapshot obtained by the `fork` [27] system call. The forked background process generates a compact AOF or RDB, while the original process continues to serve requests from the foreground users. During this period, newly arrived requests trigger copy-on-write operations for the modified memory pages, hence the generated snapshot is unaffected. Thus, `fork` is universal enough

to make a snapshot for any structure. For generating a checkpoint for new structures, Redis only needs to develop new serialization methods. It is noteworthy that `fork` is a heavy operation that causes significant latency spikes, which will be verified in our experiments (see Section 8.2.2). Moreover, since Redis is not designed as a durable database, achieving full durability will suffer in both throughput and latency. As a result, by default, `fsync` is only called once per second to persist writes to AOF.

2.2 Non-Volatile Memory

The first commercially available NVM product, Intel® Optane™ Persistent Memory (Optane PMem), supports byte-addressability, high density, and direct persistence. However, despite these advanced features, Optane PMem has much lower bandwidth and increased latency compared to DRAM. According to experimental results [47], the peak read and write bandwidth are 6.6GB/s and 2.3GB/s respectively in a single Optane PMem DIMM setup. The read latency is about 3× higher than that of DDR4 DRAM (*e.g.*, 305ns and 101ns for random reads on Optane PMem and DRAM, respectively). Optane PMem supports the following two working modes: (1) Memory mode. Optane PMem acts as the addressable volatile main memory providing large capacity, and DRAM is used as an upper-layer cache to hide the higher latency of Optane PMem. (2) App Direct mode. Optane PMem is treated as a separate memory from DRAM. Applications can directly access it using load/store instructions. To leverage the persistence of Optane PMem, we focus on App Direct mode in this paper. Note that at runtime, data might be buffered at many places (*e.g.*, store buffers and CPU caches) before reaching NVM. Hence, some instructions, *e.g.*, `SFENSE` and `CLWB`, are provided to guarantee execution order and durability. In addition, since the granularity of atomic writes is only 8B on 64-bit CPUs, writing with a larger payload might result in an unexpected state from a system crash. Therefore, careful designs are required to safely use Optane PMem as a non-volatile medium.

2.3 Challenges

Scaling DRAM-based Redis to a larger capacity is costly, and the application scenarios of Redis are limited due to the volatile nature of DRAM. NVM is less expensive on capacity and provides persistence support. A natural question is how to introduce NVM into the Redis design space. TieredMemDB [40] is a Redis branch that can utilize the large capacity of NVM with a few code changes to Redis. Its main idea is to provide Redis with a configurable memory allocation policy that enables it to allocate data in DRAM or NVM according to the predefined configuration. However, TieredMemDB does not take advantage of the persistence of NVM. It still relies on AOF and RDB to support persistence. We now discuss the challenges of leveraging NVM to build a cost-efficient Redis system with full transaction durability.

Challenge 1: performance degradation. Putting all of Redis' data into NVM will bring significant performance degradation. First of all, the high access latency and low bandwidth of NVM will significantly degrade system performance. In addition, Redis commands and their operating parameters, *i.e.* user data, need to be persisted in AOF, which also causes significant overhead.

Table 1: The characteristics of different data.

Data Type	size	persistent	hot	location
User data	large	yes	-	NVM
MetaData of Allocator	small	no	yes	DRAM
Indexes	large	no	-	DRAM/NVM
Runtime variables	small	no	-	DRAM/NVM

Challenge 2: complexity of NVM programming. NVM programming is hard and error-prone [13, 24] as even missing a single `CLWB` or `SFENSE` instruction may lead to inconsistent and irrecoverable data damage. Prior works [15, 24, 25, 31, 42] propose libraries or conversion frameworks to convert volatile indexes to NVM, but they all come with various limitations such as additional space cost, performance penalties [15, 24, 31], or consistency compromises [15, 25, 42].

Challenge 3: stability with full Redis compatibility. Redis provides various data models and many database features to accommodate a wide spectrum of application scenarios. Being fully compatible with Redis while making the new product stable is complicated. Besides, some database features (*e.g.*, checkpoint and transaction) need to be redesigned or reimplemented due to the introduction of NVM. For example, Redis' checkpointing relies on the `fork` system call. However, the `fork` may lead to serious latency spikes, not to mention that existing techniques do not support a similar approach for data on NVM. Furthermore, Redis supports transactions by specifying a group of commands to be atomically processed. The introduction of full transaction durability on NVM by design complicates the compatibility of transaction atomicity.

3 OVERVIEW OF *TAIR-PMEM*

In this section, we first introduce the core design concepts behind *Tair-PMem*, and then discuss the overall architecture.

3.1 Core Design Concepts

We aim to design an enterprise-strength IMDB, which can provide high stability, high performance, and full durability simultaneously atop Redis. With these goals in mind, we design *Tair-PMem* following the guidelines below.

Providing DRAM-like performance. To hide most of the slower NVM accesses, *Tair-PMem* adopts a DRAM-NVM hybrid structure, which is based on a well-controlled data placement strategy. This strategy determines the location of data according to the perspectives of durability, access frequency, and volume, as shown in Table 1. The persistence of user data needs to be guaranteed, so it needs to be placed in NVM. The metadata of the allocator is frequently accessed during writes, and its size is small. Thus, we pin it in DRAM. Because the size of the indexes and runtime variables may be large, *Tair-PMem* does not force all of them to be pinned in DRAM, the location of which is mainly determined by the space usage of both DRAM and NVM. To further reduce persistence overhead, we prefer the *log-as-user-data* design. In *Tair-PMem*, log entries will be accessed by user queries through indexes, playing the role of user data and thus reducing the occupation of NVM bandwidth.

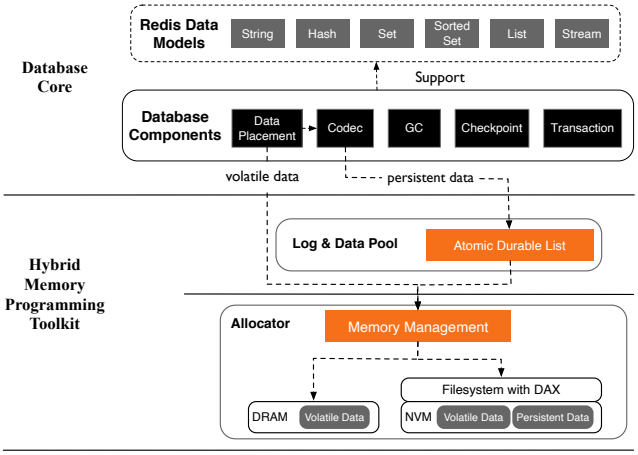

Figure 1: The Architecture of *Tair-PMem*.

Hiding NVM programming details in unified APIs. NVM programming is complex and error-prone when trying to utilize its persistence property. *Tair-PMem* provides a toolkit to efficiently manage both DRAM and NVM in the classic malloc-and-free style. In addition, the toolkit abstracts unified APIs to guarantee durability, atomicity, and consistency, hiding complicated NVM programming skills, such as avoiding partial writes and memory leaks.

Modularizing retrofit to keep code intrusions low for stability and compatibility. Redis supports abundant data models and dazzling APIs through many different index structures. To keep code intrusions low, durable, atomic, and consistent writes rely on modularized components, such as *log & data pool*, *codec*, etc. The original index implementation can be untouched, leaving the determination of the data location of indexes to the allocator. As a result, the implementation of the read operations requires no modification.

Leveraging NVM to enhance fundamental database features. The byte-addressable and durable characteristics of NVM can be exploited to further enhance Redis. Specifically, the AOF, the rewriting of which may incur latency spikes, is removed. The instant checkpoint mechanism further eliminates the heavy `fork` method, significantly mitigating latency spikes. To accelerate the recovery procedure, *Tair-PMem* supports fast recovery by backing up the volatile DRAM data (*e.g.*, allocator metadata and indexes).

3.2 Architecture Overview

Figure 1 shows the architecture of *Tair-PMem*. It consists of two components: *hybrid memory programming toolkit* and *database core*.

Hybrid Memory Programming Toolkit. The toolkit is designed specifically for performance-critical IMDBs and aims to be lightweight and efficient. It provides unified APIs to manage different storage media and hide complicated NVM programming details by two subcomponents: *Allocator* and *Log & Data Pool*. *Allocator* (see Section 4.2) is responsible for managing the spaces of both DRAM and NVM. To support the data placement strategy in DRAM-NVM hybrid data layout, it divides the storage space into three regions, *i.e.*, volatile space on DRAM, volatile space on NVM, and persistent space on NVM. For excellent performance, the allocator metadata always resides in DRAM. Given the start address and allocated size, the metadata of the allocated space can be recovered. *Log & Data Pool* (see Section 5) organizes all persistent user data by a linked list, which provides the start addresses and sizes of all allocated spaces to the allocator for recovery. It offers lightweight transaction semantics, which is the core component to support the *log-as-user-data* design and hide complex NVM programming details.

Database Core. Database core (see Section 6) contains several modularized subcomponents to support rich Redis data models. Utilizing the toolkit and *codec* component, *Tair-PMem* can place different kinds of data according to the strategy shown in Table 1. User data is encoded as entires of the durable *log & data pool* and plays the role of the log, thus removing Redis' original log (AOF). The *codec* component is deliberately designed so that the original indexes can point directly to the user data encoded in the entry, and thus the implementation of original indexes is untouched. The untouched volatile indexes can be recovered by scanning the log entries to achieve durability. Furthermore, the *garbage collection (GC)* component is designed for log order maintenance and efficient entry deletion from the data pool. Due to the removal of the original log (AOF), *checkpoint* and *recovery* components are redesigned. Their capabilities are further enhanced with the help of NVM's persistent property. Besides, the *transaction* component provides transaction properties, such as atomicity and full durability.

4 MEMORY PROGRAMMING TOOLKIT

The *hybrid memory programming toolkit* includes two components, *allocator* and *log & data pool*. The former is used to manage the memory spaces of both DRAM and NVM. The latter is designed to ease the use of NVM's persistence property by hiding complex NVM programming tricks, avoiding partial writes and memory leaks, and obtaining persistent atomic writes. This section introduces the toolkit's APIs that we export to developers, followed by the design of the *allocator* component.

4.1 The APIs of the Toolkit

The APIs exported by the toolkit are shown in Table 2, including the following two parts.

The APIs of Allocator. The allocator manages two storage media, DRAM and NVM, through three memory spaces. By assigning the `type` parameter of `malloc`, the user can specify the memory space to be allocated. For volatile data, the user can leave the decision to the allocator for ease of use. Given the allocated address, the `free` function figures out which space it belongs to and then deallocates it correspondingly. The `mark_as_allocated`, an interface not provided by the classic allocator, can recover the metadata of an allocation given its start address and allocated size.

The APIs of Log & Data Pool. These APIs serve to store persistent data and guarantee durability, atomicity, and consistency. The data is organized in a linked list in the order of the creation time, which allows the database to employ the list as a log. The `txn_begin`, `entry_append` and `txn_end` functions enable the log to support atomic commits, avoid partial-writing, and guarantee persistence. The database can also use this linked list as the user-data pool. With `entry_append` and `entry_free`, data can be inserted into and freed

	API	Description

Allocator	`void* malloc(size_t size, MemType type);`
	`enum MemType {`
	`  VOLATILE_DRAM, VOLATILE_NVM,`
	`  PERSISTENT_NVM, VOLATILE`
	`};`
		Allocates memory with a `size` located in different memory spaces which are divided into three kinds, the volatile DRAM, volatile NVM, and persistent NVM.
		`VOLATILE`, the default value of `type`, means the location of an object is determined internally by the allocator.

	`void free(void* ptr);`	Deallocates the memory space pointed by `ptr`.
	`void mark_as_allocated(void* ptr, size_t size);`	Recovers allocation metadata of an object given by the `ptr` and `size`.
Log & Data Pool	`log* txn_begin();`	Marks the beginning of a transaction and then returns a newly created log for the transaction.
	`entry* entry_append(log* txn, codec* method);`	Allocates an entry from persistent NVM space, then encodes it by the codec `method`, and finally atomically appends it to the `txn` log.
	`void txn_end(log* txn, log* global);`	Commits the transaction by atomically appending the `txn` log to the `global` log.
	`void entry_free(entry*);`	Atomically removes the `entry` from its corresponding log, and then deallocates the entry by `free()` function.
	`void recover(log* global);`	Recovers the `global` log by scanning it to obtain each entry's pointer and size, thus the overall allocation metadata can be recovered by the `mark_as_allocated()` function.

Table 2: The APIs of *hybrid memory programming toolkit.*

from the pool atomically. The `recover` method scans all persistent entries stored in the linked list and restores the metadata of the memory allocations for these entries.

4.2 Allocator

The design of the allocator follows the high-level idea that all the metadata is stored in DRAM for achieving high performance. The allocator structure and allocation process remain almost unchanged from classical allocators, making the allocator as stable as the classical ones. The key problem to be addressed is how to recover metadata after the database crashes or exits.

4.2.1 Allocator Structure. The allocator extends the classical slab allocator jemalloc [19]. A slab is a contiguous memory region of DRAM or NVM with a 4KB-aligned start address and a space size that is a multiple of 4KB. As shown in Figure 2, the allocator contains a set of *size_class* objects, each of which manages a set of slabs. All slabs from a *size_class* object manage a number of allocation units of the same size, denoted by s. The size s is recorded in *size_class*, such as 32B, 48B in the figure. Slab metadata, as the primary metadata of the allocator, contains the start address of the slab and a bitmap that marks allocated units. The slab size is set to the least common multiple (LCM) of 4KB and s, denoted by $LCM(4KB, s)$. This slab size setting was originally introduced to mitigate space waste and now becomes a key for rebuilding allocator metadata after the database crashes, which will be discussed later.

4.2.2 Allocation Process. Recall that the memory space is divided into three types. By passing `MemType` (see Table 2) to `malloc`, the allocation will happen in the corresponding memory space. For data that does not require persistence and does not need to be pinned in DRAM, the user can leave the `MemType` unspecified. In this case, `VOLATILE`, the default value of `MemType`, is used to allocate space fairly from `VOLATILE_DRAM` and `VOLATILE_NVM`, according to a predefined DRAM-NVM usage ratio.

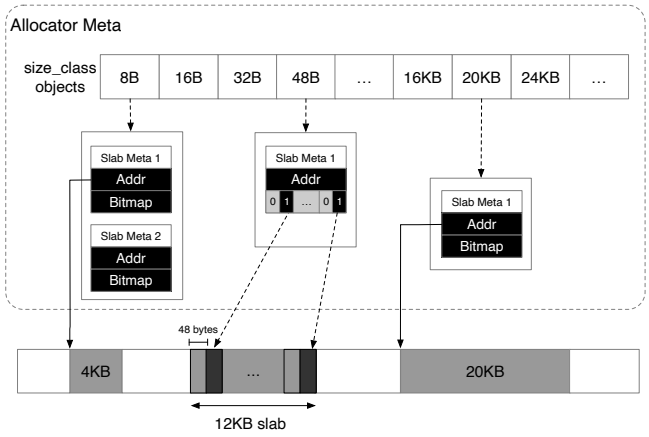

Figure 2: The simplified structure of an allocator.

After assigning `MemType`, the allocation size x needs to be further specified. The allocator finds the *size_class* object with the smallest managed size among all those objects with size greater than x. For example, when requesting 40B, the 48B *size_class* object will be returned. After finding the suitable *size_class* object, the allocator returns the address of a free space unit and marks it as allocated in the bitmap. If no free space is left, a new slab will be created.

4.2.3 Metadata Rebuilding. There are two methods to rebuild allocator metadata for two distinct recovery scenarios.

Rebuilding after a normal shutdown. During a normal shutdown, the allocator backups the metadata by scanning all the metadata shown in Figure 2 and serializing them into a persistent NVM region. During a restart, the allocator just deserializes them.

Rebuilding after a crash. During the recovery after a crash, the allocator metadata that manages persistent data should be recovered. Notably, the user of the allocator, *i.e. log & data pool* to be

discussed in section 5.3, should provide the start addresses and allocated sizes of all allocations. With the start address `ptr` and the allocated size `size`, `mark_as_allocated` reconstructs the metadata of an allocation, *i.e.*, corresponding slab metadata.

The implementation of `mark_as_allocated`. The *size_class* objects are predefined and will be automatically initialized at startup. According to the start addresses and allocated sizes of allocations, `mark_as_allocated` reconstructs the metadata of slabs, *i.e.*, start addresses of slabs and the bitmaps of slabs, by the following three steps. (1) According to the allocated size, calculate which *size_class* object the allocation belongs to, similar to the allocation process above. (2) The recovery of the start address of a slab is based on the following proposition (proved in the next paragraph) — *among all the allocation units in a slab, only the start address of the first unit, which is also the start address of the slab, can be divisible by 4KB.* Based on the proposition, keep subtracting the start address of the allocation by s until the address becomes divisible by 4KB. Then, the start address of the first unit, *i.e.* the start address of the slab, can be obtained. (3) Mark the allocation in the bitmap similar to the allocation process.

The Proof of the Proposition. Assume that the start address of a non-first unit in a slab is divisible by 4KB. Since each unit is s-aligned, the start address of the non-first unit is a common multiple of 4KB and s, denoted by $CM(4KB, s)$. Since the start address of a slab is always 4KB aligned, the address of the first unit is also a $CM(4KB, s)$. A slab contains two addresses that are $CM(4KB, s)$, which conflicts with that the slab size is $LCM(4KB, s)$.

5 LOG & DATA POOL

Log & data pool is another component of the *hybrid memory programming toolkit*. It serves the persistent data, which hides the complicated NVM programming skills and guarantees durability, atomicity, and consistency. We first explain why we choose a linked list structure to organize the data in *log & data pool*. Then, we describe the reason for employing the S-Linked list [1] as the list implementation, followed by the enhancements on it. At last, we present the API implementation of transaction semantics.

5.1 Determination of Data Structure

Sequential File or Linked List. Since we have chosen the design that log entries will be accessed by user queries through indexes to reduce the occupation of NVM bandwidth, we need to pick a structure, *i.e.*, a traditional sequential file or linked list, to organize the log. If we select sequential files, we need to find a way to compact several files and generate new files for fast access and garbage collection, as the LSM-tree [36] does. The data will be repositioned after compaction, and the complex and varied indexes of Redis should be reset, which complicates the retrofit. In contrast, if we select linked lists, useless entries can easily be removed without repositioning the data, thus the indexes need not be reset. Hence, we favor the linked list for organizing entries of *log & data pool*.

5.2 The Variant of the S-Linked List

For the classic singly linked list, an entry cannot be effectively removed. For the classic doubly linked list, it takes extra two pointers (16B) for each entry, which is somewhat wasteful of space. We adopt the variant of S-Linked list [1], called the ES-Linked list (the Enhanced S-linked list), which reduces the extra pointers to nearly one per entry and optimizes it for effective entry removal.

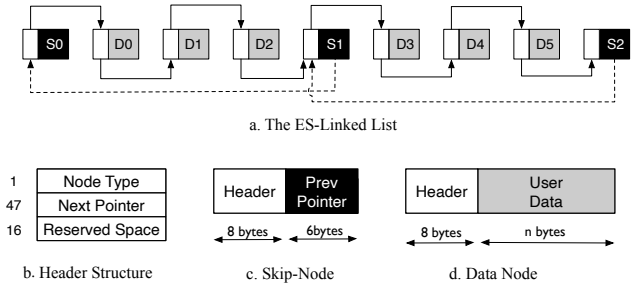

Figure 3: The structure of the ES-linked list.

5.2.1 Structure of the ES-Linked List. As shown in Figure 3(a), an entry of the list includes a header and a remainder. The former contains node type (1 bit), next pointer (47 bits), and reserved space (16 bits), shown in Figure 3(b). The node type is used to identify the type of node, *i.e.*, data node or skip-node. Since all the nodes are persisted in the address space created by mapping an NVM file, a 47-bit pointer (or offset) is sufficient, which can index more than 100TB of data. The reserved space is used by the *database core* which will be described in section 6.

The structure is similar to the singly linked list, whose nodes only point to the next node. The difference is that there are special nodes called skip-nodes containing a backward pointer to another skip-node, as shown in Figure 3(c). A skip-node will be generated in every k nodes. k is 16 in *Tair-PMem*. A backward pointer pointing to another skip-node makes the sub singly linked list be a circular linked list. Note that the deletion of a node needs to change the next pointer of the previous node, which can be located by traversing the circular linked list from the node to be deleted. The data nodes, shown in Figure 3(d), store the user data.

Compared to the original S-linked list, skip-nodes of the ES-linked list do not hold any user data. As a result, the deleting and adding of skip-nodes need not modify database index structures. Thus it's easier and more effective to keep a proper number of the data nodes between two skip-nodes, lowering the space consumption from skip-nodes. Besides, the ES-linked list utilizes the following technique to optimize deletion.

5.2.2 Group Deletions with Prefetching. The original S-linked list saves the extra space cost but introduces many more entry accesses when deleting an entry. Specifically, each deletion needs to search 17 entries for locating the previous one. The time to search the previous node is spent primarily on memory accesses rather than CPU computations. In addition, Optane PMem has higher latency than DRAM, making memory accesses take longer.

To hide most of the access latency, every m (say 32) log entries of the ES-linked list will be deleted in a group to utilize memory prefetching. Specifically, the m nodes to be deleted will be asynchronously prefetched first and then accessed one by one. Thus, the last *m-1* NVM accesses are expected to hit the cache, making the

actual number of NVM accesses be 1. Following the same way, the next node will be accessed until the previous node is found through the circular linked list.

5.3 Implementation of Operations

Atomic and Durable Log Appending of Multiple Entries. Inserting, updating, or deleting an element in *Tair-PMem* appends one redo entry in the log. A command may operate multiple elements, and users can further specify multiple commands to be atomic and durable. Thus, we need to ensure the appending of multiple entries be atomic and durable. *Tair-PMem* contains two kinds of logs, the global log and the detached uncommitted transaction log, both of which are implemented by the ES-linked list. The `txn_begin` function will create a detached list. When inserting data, *Tair-PMem* first records the entries in the detached list by the `entry_append` function. When committed by the `txn_end`, all the entries of the detached list will be first persistent in NVM by calling `CLWB` instructions for them, followed by an `SFENCE` call. Next, the next pointer of the tail entry on the global list point to the detached list. Lastly, `CLWB` and `SFENCE` are called for the pointer that was just set. Because of the 8-bytes atomic writing, the pointer setting is atomic. As a result, log appending of multiple entries is atomic and durable.

Recovery of the List. The `recover` function can recover the allocations of all entries by traversing the persistent linked list. It calls the `mark_as_allocated` to recover the allocation for an entry by using its start address and size. Note that, the size information is provided by the user of the toolkit, *i.e.*, the *codec* component of *database core* (see Section 6.6.1), which obtains the size by the entry decoding function.

Atomic Deletion of One Entry. Deleting or updating will reclaim an entry from the data pool via the `entry_free` function. The deletion of an entry should be atomic. An entry removal will change the next pointer of the previous entry, and subsequently call `CLWB` and `SFENCE` to persist the pointer. However, the actual freeing of the target entry from the allocator may not be completed before a crash. In such a case, since the allocator is recovered by scanning the list and the target entry is not in the entry list, the recovered allocator metadata will not involve it. Thus, it can still be freed after recovery, avoiding memory leaks. For the atomic deletion of multiple entries, it is implemented by appending tombstone entries before the actual deletions, which is done in the *database core*.

6 DATABASE CORE

One of our primary design goals is to keep intrusive Redis modifications low through modularized retrofits. Specifically, Redis' multifarious index implementations are aimed to be untouched. Therefore, it can be much easier to ensure the compatibility and stability of Redis' APIs. To achieve this, *Tair-PMem* divides Redis' data into two types: volatile and persistent. All volatile data maintains its original implementation, and only the placement of volatile data is further optimized via the allocator API. The persistent data (*i.e.*, user data) is encoded by the *codec* component and then persisted to the *log & data pool*.

This section details the major retrofits, including managing volatile and persistent data in the database, leveraging the toolkit for transaction atomicity and durability, pursuing advanced feature benefits from the NVM-based durability.

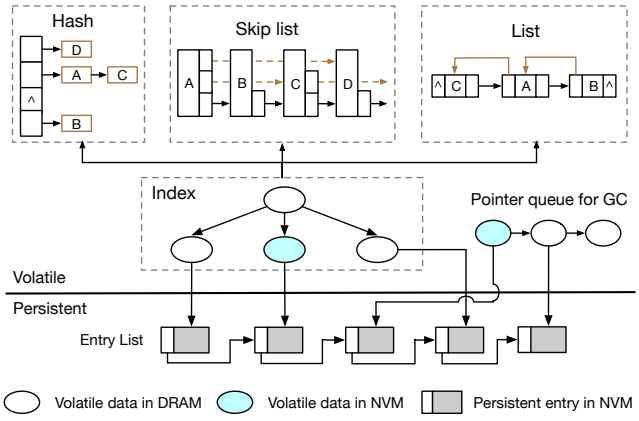

Figure 4: The basic structure of *Tair-PMem*.

6.1 Volatile Data Placement

Redis' volatile data primarily contains indexes, various immediate runtime variables, etc. As shown in Figure 4, a data model may be indexed by one or two index structures, such as hashes, skip lists, lists, etc. For example, a hash model is directly indexed by a hash structure, and a sorted set is indexed by both skip lists and hashes. With the help of abundant indexes, Redis supports many data models and dazzling APIs. For easier API compatibility, we barely modify indexes and keep the original implementation intact.

As described above, volatile data could be presented in a wide range of formats. The medium on which volatile data is stored depends on the default behavior of the `malloc` described in Section 4.2.2. Specifically, when the volatile data stored in DRAM exceeds a certain ratio, say 1/5, an NVM address will be returned. Otherwise, a DRAM address is returned. By specifying the `VOLATILE_DRAM` type for `malloc`, *Tair-PMem* pins some hot data structures in DRAM to optimize performance, such as an array of bucket pointers for the global hash index.

6.2 Persistent Data Encoding

By providing encoding functions for *codec* component, user data of different data models can be persistent as entries of the ES-linked list in *log & data pool*. In addition, the encoded user data should be able to be pointed to directly by the volatile index to maintain the original index structures. As Redis has varied data models, it is tedious and unnecessary to explain all detailed entry layouts here. Instead, we show the layouts of two representative kinds of models, *i.e.*, the string and KVs models. The string model represents the simplest case that the value is a string, and the KVs model represents the case that the value is a complex structure (*e.g.*, hash, set, sorted set, list) that further contains a number of key-value pairs. *Tair-PMem* also supports some other structures, but we omit them here for clarity.

All entries store two fields, *i.e.*, opcode (1B) and database id (1B), in the reserved space of the entry header shown in Figure 3(b). These fields are used for recovery, making entries function as a log.

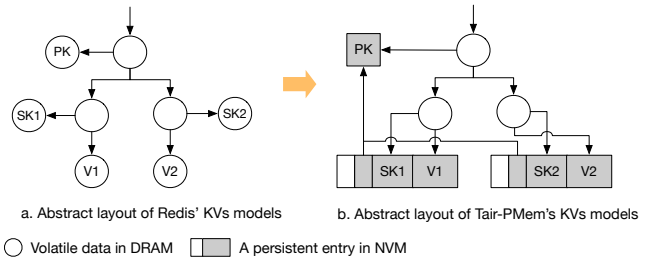

Figure 5: The layout transformation of KVs model. The value of the primary key (PK) is a complicated structure containing some KVs, that is <SK1, V1>, <SK2, V2> in this figure.

The opcode is used to identify write operations, such as inserting a string, deleting a key, inserting an element into a set, etc. The database id is used to differentiate the internal databases in Redis.

6.2.1 Entry Layout of KVs Models. Although the indexes are distinct between different KVs models, Figure 5 abstracts them and shows a common sketch. Redis locates the primary key through several volatile objects. For brevity, we abstract these objects into one, as shown in Figure 5(a). In Redis, the volatile indexes of different data models keep three pointers to the primary key (PK), secondary key (SK), and value, respectively. The PK is shared by all its KVs. After encoding an element of the KVs models, the persistent entry contains the entry header, PK, SK, and value, as shown in Figure 5(b). We extract the PK from the entry with only a key pointer saved, and thus the PK can be shared by all KVs. The extracted PK holds the reference counter and will be reclaimed when the counter decreases to zero. The volatile indexes keep the original pointers to the PK, SK, and value. Because of the unchanged implementations of all volatile objects, the implementations of all read logic remain unchanged too.

6.2.2 Entry Layout of String Model. When the value is a string, *Tair-PMem* puts both key and value to the same entry, preceded by the header. The volatile object also retains the original behavior that it maintains two pointers to key and value, respectively. The key and value are stored together for better data locality.

6.2.3 Idempotent Log Entries. An operation encoded by an opcode should be idempotent. Idempotency means an operation can be applied several times without changing the result after the first run. Without idempotency, the database might be corrupt when a recovery is interrupted and restarts. For example, Redis' `ZREMRANGEBYRANK` command removes all elements in the sorted set with rank between a certain range. If the command is redone many times, the sorted set may be changed after each redoing. Thus, the command is not idempotent and cannot be encoded in the log directly. A command transform is necessary. For example, we transform it to the `ZREMRANGEBYSCORE` command that removes all elements in the sorted set with a score in a specified range.

6.3 Transaction Atomicity

The data should be consistent after redoing logs in recovery. Therefore, modifications of the log should be atomic from the database transaction perspective. Otherwise, the database will go into an inconsistent state after the recovery. Making the persistent log play the role of user data minimizes writes to NVM but complicates the implementation of transaction atomicity.

6.3.1 Redo Log for User Operations. *Tair-PMem* enables the *log & data pool* to function as a redo log to achieve transaction atomicity. An uncommitted log implemented by an ES-linked list is first created via the `txn_begin` method.

- When inserting, a new entry is atomically appended to the uncommitted log by the `entry_append`, and the volatile index is set to point to that entry.
- When updating, copy-on-write is used instead of in-place-update. Specifically, the original log entry is first located by the index, then a copied entry is updated and appended to the uncommitted log via the `entry_append`. After that, the volatile index is reset to point to the new entry.
- When deleting, a tombstone entry is appended to the uncommitted log by the `entry_append`. The tombstone entry only stores the key and contains no value.

Finally, once the transaction is committed, the uncommitted log will be atomically appended to the global one by the `txn_end` method.

6.3.2 Order Maintenance of Log Entry. Since the log plays the role of user data, internal entries of the log may be deleted via the `entry_free`. Maintaining the correct order of the entries is crucial for recovery. To achieve this, the implementation of user operations should conform to the following two guidelines:

(1) The original entries should not be deleted until the updates or deletes are committed successfully.
(2) For user deletions, the tombstone should be removed after the original entry has been deleted.

If a transaction is committed, but the deletion of entries is not completed before a crash, the corresponding updating entries or tombstone entries must be in the global log. The incomplete deletions will continue to be executed when these entries are read and decoded in the recovery process, thus avoiding NVM leaks.

6.4 Garbage Collection for Entries

To better modularize *Tair-PMem*, we add the *garbage collection (GC)* component to reclaim the discarded persistent entries. It serves two purposes, *i.e.*, pursuing better performance and serving the *checkpoint* component (Section 6.5.1). In *Tair-PMem*, Redis' original threads issue the deletions in an ordered fashion to the *GC* component, which is responsible for removing them in the order expected by the threads.

Lock-free GC. To offload the deletion of a large data model, Redis' main thread delegates it to Redis' GC thread and only processes the deletion of the small one. As a result, deletions will be performed by two threads. These two threads will not conflict when deleting the DRAM data of different keys, but they may conflict in *Tair-PMem* because removing two different linked-list entries may operate the same entry. To avoid such conflicts, we assign all entry deletions to a new background thread, *i.e.* the entry-GC thread. The producers are Redis' two original threads that generate the deletion jobs for log entries. The entry-GC thread is the only consumer, orderly

consuming the jobs. As shown in Figure 4, the volatile lock-free queue records pointers to the log entries freed by the producers.

Lock-free Linked List. Entry deletion may still conflict with entry appending. To avoid that, the entry-GC thread does not reclaim the list tail entry, which is the only entry that will be modified by the append operation. Thus, no conflicts exist and no locks are needed.

6.5 Checkpoint

Checkpoints are an important mechanism used in many scenarios, such as backups and replications.

6.5.1 Instant Checkpoint of Tair-PMem. The database can recover from the *log & data pool* (Section 6.6.1). Therefore, a sublist of the global log, starting from the first entry and ending with an entry that was the tail entry of a committed transaction, is a checkpoint. All the matters to create a checkpoint are recording the ending entry and closing the GC for the sublist range. The latter relies on the *GC* component. This component supports disabling GC for a sublist range while the log entries remain in the correct order. We omit the details for clarity. Nevertheless, it can be simply implemented by stopping consuming all deletion jobs. As a result, a checkpoint can be generated instantly without negative impacts on performance. After the checkpoint is released, the GC thread starts consuming deletion works again.

6.5.2 Compatibility. Since the above checkpoint is not compatible with Redis' RDB, we further support RDB generation to maintain compatibility with other systems in the cloud. A snapshot of persistent user data pointed by volatile indexes can be taken by the above method. A snapshot of volatile data on DRAM is created by `fork` as usual. To reuse Redis's original implementation, we should further support `fork`-like capabilities for volatile data on NVM. However, there is no existing technology that can be directly exploited.

Note that volatile data on NVM is stored in a memory-mapped file that resides on a DAX (direct access) aware file system. Thus, we try to utilize the `reflink` [28] system call to take a snapshot on this region, similar to what the `fork` does in memory. The community filesystem does not support the `reflink` function on the DAX mode. We extend the ability on XFS [43] to achieve that.

6.6 Recovery

By utilizing the durable log appending from *log & data pool*, all entries of a transaction are persisted once committed, giving *Tair-PMem* full transaction durability. *Tair-PMem* redos the data operations encoded in the log entries for disaster recovery. Furthermore, special backups will be made to speed up recovery after normal shutdowns.

6.6.1 Disaster Recovery. The recovery process consists of two stages, the recovery of the *log & data pool* and that of data indexes. The former is performed via the `recover` function, as discussed in section 5.3. The latter will be discussed below. By traversing the log, each entry can be parsed to extract the opcode and command arguments. Redoing the parsed operations will rebuild the volatile indexes pointing to the existing entries. For example, the opcode may indicate that it is an operation to insert a string with arguments of key and value. These operations may insert, update or delete data for any kind of data model.

Table 3: The address spaces managed by the allocator. Each type of data resides in either one or two spaces depending on whether it is stored in two media.

Managed Data Type	Managed Media (address space)	Recovery Needed	Data Persistent
Log Entries	NVM (space A)	**yes**	yes
Indexes	DRAM (space B) NVM (space C)	**yes**	no
Runtime variables	DRAM (space D) NVM (space E)	no	no

Experiment Results. For 16GB user data of string model, Redis takes **306s** or **239s** to recover depending on whether AOF is periodically rewritten with a compact AOF header or RDB header, respectively. *Tair-PMem* takes **286s** to complete recovery, avoiding the re-generation of user data.

6.6.2 Recovery after a Normal Shutdown. For fast recovery, two kinds of backups are made before a normal shutdown: one for the metadata of the allocator that manages both log entries and indexes and the other for indexes themselves. With these backups, normal recovery becomes fast, which is often used in routine instance maintenance, such as version upgrades.

Fast Recovery of Allocator Metadata. We rebuild the allocator metadata by backing up it, as described in Section 4.2.3. To back up only the allocator metadata of log entries and indexes, we partition five memory spaces as shown in Table 3 and enable the backups in units of address spaces. *Tair-PMem* separately back up the metadata of the address spaces of log entries, DRAM indexes, and NVM indexes, *i.e.*, space *A*, *B*, and *C*. Since the size of the metadata is small, the backup and recovery are fast.

Fast Recovery of Indexes and Log Entries. After the allocator metadata of space *A*, *B*, and *C* are rebuilt, these spaces themselves should be recovered. Log entries are persisted in space *A* and thus need not be touched. For DRAM indexes, before a normal shutdown, *Tair-PMem* backs up it by mirror copying the memory space *B* to a new persistent NVM space B', which avoids time-consuming serialization. During recovery, B' is mirror copied back to *B*. For NVM indexes, it is very efficient, since the recovery process reuses the original space *C* which will be persisted by `CLWB` instruction before a normal shutdown.

Experiment Results. For 16GB user data of string model, Redis takes **164s** to generate an RDB and **96s** to recover, while *Tair-PMem* takes only **4.9s** to make the backups and **5.4s** to recover. This verifies that *Tair-PMem* can significantly improve the user experience.

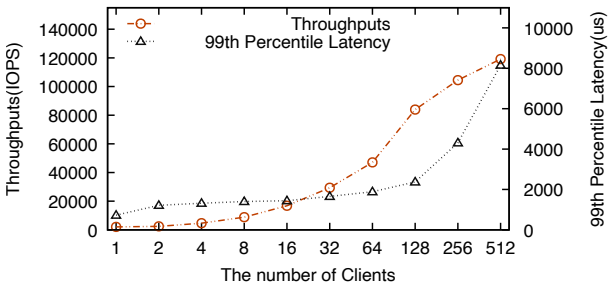

Figure 6: The throughputs and 99th percentile latencies of string model for fully durable Redis.

7 TYPICAL NVM PROGRAMMING SKILLS

Tair-PMem employs a number of NVM programming skills to achieve performance similar to that of original Redis. The typical of these are introduced here.

7.1 Breaking Large Values into Shards for COW

When updating a log entry, the copy-on-write mechanism should be utilized instead of the in-place updating. Hence, setting a bit for it will rewrite the whole entry. For example, Redis' `SETBIT` command sets bits for a string model. *Tair-PMem* breaks a large string into many shards, each of which is a 256B log entry. These entries are organized via an index of volatile pointer array, serving as a large string. To reduce the bandwidth occupation of Optane PMem, a bit set will copy-on-write only a single shard.

7.2 Single Tombstone Entry When Possible

In Redis, a primary key deletion removes all elements of a KVs model. Deleting a large string removes all entries of the shards. If a tombstone entry is generated for each log entry, the deletion performance will be unsatisfactory. *Tair-PMem* generates only one tombstone entry for the above scenarios while encountering the following challenge. If only partial entries are removed before a crash, it should ensure that no entries are leaked after recovery.

By redoing the global log during recovery, the unreclaimed elements will be put into a KVs model. As discussed in Section 6.3.2, the tombstone is always the last one to be reclaimed. Thus, redoing the tombstone log will remove all existing elements of a KVs model, ensuring no entries are leaked. Similarly, one tombstone for a sharded string also guarantees no entry leaks.

7.3 Prefetching

We utilize prefetching to hide the memory access latency, especially for NVM accesses, which is used for ES-linked list entry removal and Redis' index accesses. The former has been described in Section 5.2.2. Since Redis executes commands in groups, the latter can be implemented similarly. Specifically, the data for commands in the same group will be prefetched first before a command is executed.

8 EXPERIMENTS

Since Redis has abundant models and numerous APIs, it's cumbersome and unnecessary to evaluate the detailed performances of every model and API. To illustrate that *Tair-PMem* achieves the desired goals, we pick two representative models, string model and hash model, to evaluate. The former represents a simple model, and the latter represents a KVs model. The two models are also the most frequently used models in cloud services.

8.1 Experiment Setup

Our experimental platform is based on the Alibaba cloud server of *ecs.ebmre6p.26xlarge* instance type. The server is equipped with 2.50 GHz Intel Xeon Platinum 8269 processors, 384 GB DRAM, 1536 GB Intel® Optane™ Persistent Memory, 2 TB Alibaba cloud Enhanced SSD (ESSD) of Performance Level 1 (PL1), just the same as the deployments of our cloud service. The operating system is Alibaba Cloud Linux 2, and the file system for ESSD is ext4.

The evaluation is based on Yahoo! Cloud Serving Benchmark (YCSB) [11], including loading and transaction phases. The latter contains six workloads by default, namely workload A, B, C, D, E, and F. We evaluate all of them except workload E which contains short scan operations not supported by Redis.

We choose Redis and TieredMemDB [40] for comparison. The latter extends Redis with Optane PMem only to save costs. Like Redis, it supports the optional durability by utilizing the AOF and RDB. This means that when durability is requested, the data space needs to be doubled to store one additional copy of data stored in persistent media. By default, Redis with AOF on flushes the buffered data to the persistent storage device every second. In cloud services, the persistent medium is always an SSD for economic and technical reasons. Both Redis and TieredMemDB are evaluated with two configurations, *i.e.*, AOF synchronized per second and AOF synchronized per transaction. The former configuration may lose the data written in the last second, and the latter achieves full transaction durability. All the tests are evaluated by loading 16GB of user data with a value of 128B.

8.2 Results

Before discussing *Tair-PMem*'s performance, we evaluated the insertion performance of fully durable Redis. As shown in Figure 6, the throughput increases with the number of clients, while the latency also increases. As Redis serves as an IMDB, users are sensitive to latency. We evaluate the fully durable Redis and TieredMemDB with 128 clients, which allows for the best balance between throughput and tail latency, rather than blindly pursuing the highest throughputs but suffering very high latencies. In all the other tests, the number of clients is set to 48, which is the minimum number of clients to achieve the highest throughput.

8.2.1 *The Throughput Results.*

Loading Throughputs. As shown in Figure 7, *Tair-PMem* is fully durable (FD) natively and thus is denoted by FD *Tair-PMem* in the figures. For the loading phase, *Tair-PMem* achieves 1.8× and 2.2× throughputs compared to the FD Redis and the FD TieredMemDB, respectively. The improvement attributes to the removal of writes to the traditional persistent media, here PL1 ESSDs. It is worth noting that the removal also eliminates an additional copy of data. Compared with the default partially durable (PD) Redis, *Tair-PMem* achieves 81% of its write performances. The performance loss is

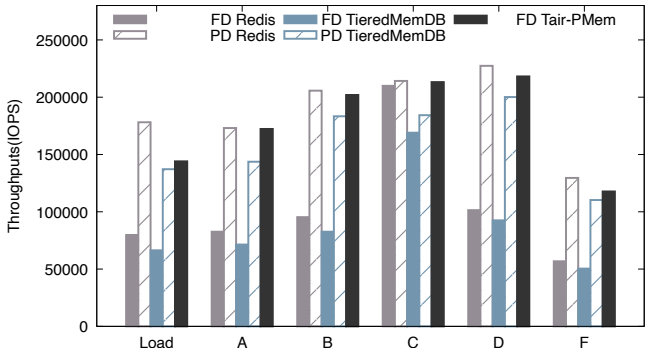

Figure 7: The throughputs of string model.

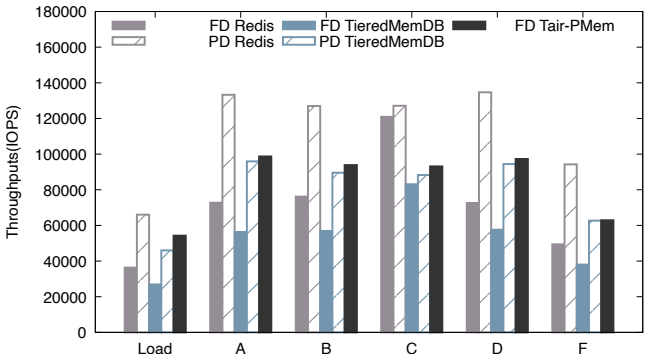

Figure 8: The throughputs of hash model.

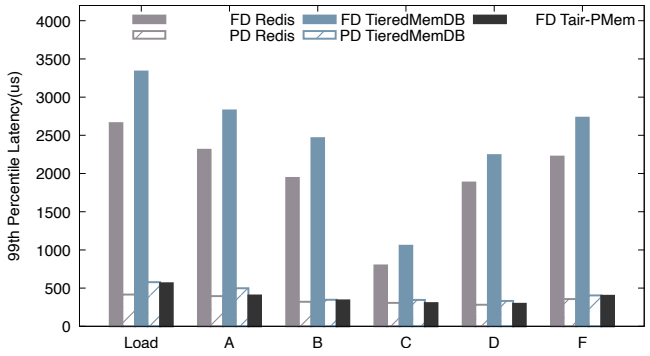

Figure 9: The 99 percentile latencies of string model.

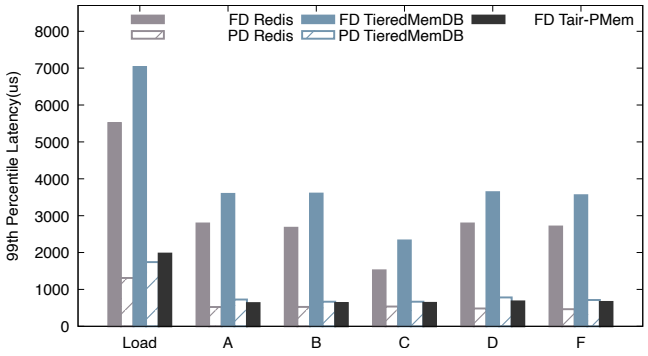

Figure 10: The 99 percentile latencies of hash model.

mainly due to the persistent writes to *log & data pool* for each transaction with the `CLWB` and `SFENCE` instructions. Interestingly, *Tair-PMem* achieves 5% enhancements over PD TieredMemDB. Figure 8 shows the results of the hash model serving as a representative of KVs models. The write performance is similar to that of the string model.

Read Throughputs. Figure 7 shows that *Tair-PMem* obtains similar read performance as Redis in the string model, illustrated by the results of read-only workload C. However, TieredMemDB loses 14% of its throughput. The performance loss is due to the access of NVM data, for both volatile and persistent ones. *Tair-PMem* has the same challenge, but the prefetching technique tackles it.

For the results of the hash model shown in Figure 8, a single read request gets all 10 KVs of the model with 73% of the throughput of the original Redis. The throughput for reading a single KV from the hash increases to 90% of that of the original Redis (not shown in the figure). Prefetching techniques are more difficult to effect for complex queries. Anyway, *Tair-PMem* still performs better than TieredMemDB.

Mixed-Workload Throughputs. For other workloads, it is a mix of reads and writes. For workloads containing many writes, namely workload A and F, it is similar to the loading performance. For the workloads containing mostly reads, namely workload B and D, it is similar to the read-only workload C. No other special observation worth mentioning further.

8.2.2 *The Tail-Latency Results.*

The 99th Percentile Latency. Figure 9 and Figure 10 show the 99th percentile latency. In general, *Tair-PMem* performs similar with PD Redis and TieredMemDB, but much better than FD Redis and TieredMemDB. For example, the FD Redis and TieredMemDB suffer 4.7× and 5.6× longer 99th percentile latencies than *Tair-PMem* for the loading string workload, respectively.

The Maximum Latency. The maximum latency is another important indicator of system stability. Many cloud users are very sensitive to it, especially for IMDB. In our cloud service, a high percentage of tickets raise that their systems suffer from occasional or periodic high latency spikes of hundreds of milliseconds or more.

As shown in Figure 11 and Figure 12, *Tair-PMem* obtains very small maximum latency, achieving an extremely stable performance. Specifically, it is less than 22ms for write-heavy workloads, and less than 9ms for read-heavy workloads. For the write-heavy scenarios, *i.e.*, the loading phase and transaction phase of workload A and F, both Redis and TieredMemDB suffer serious latency spikes. Specifically, for the global maximum latency incurred in all the write-heavy scenarios, FD Redis and TieredMemDB suffer 219× and 403× longer latency, and PD ones suffer 38× and 67× worse

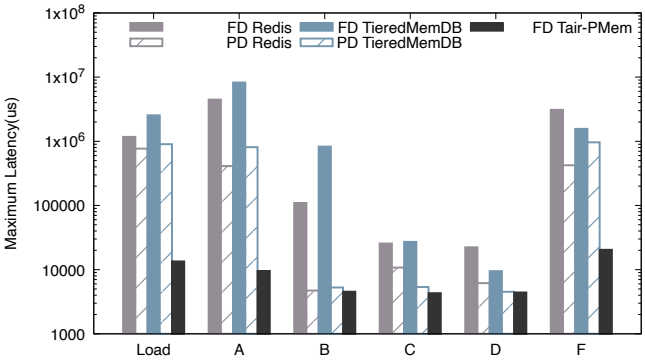

Figure 11: The maximum latencies of string model.

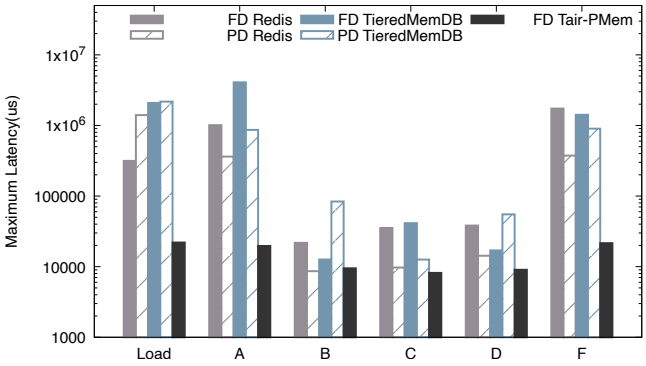

Figure 12: The maximum latencies of hash model.

performance, respectively. The results of the hash model are similar. For reading mostly or only workloads, namely workload B, C, and D, *Tair-PMem* and TieredMemDB achieve more stable performance than that of write-heavy workloads, but are still unsatisfactory.

By checking the runtime log, we find the latency spikes are caused by AOF rewrites. The latency spikes specifically come from calling `fork` and writing the log commands accumulated in the checkpoint phase to the new AOF. In contrast, since no AOF exists in *Tair-PMem*, extremely stable performance is obtained.

9 RELATED WORK

There are a large body of works on designing high-performance persistent NVM indexes, including NVM-based hash tables [8, 30, 33, 39, 50], and NVM-based tree structures [3, 6, 17, 29, 35, 41, 49]. These works mainly focus on reducing the overhead of crash consistency and improving concurrency. Among them, several recent studies [12, 15, 24, 25, 31, 42] aim at converting volatile indexes into persistent and crash-consistent NVM counterparts. Recipe [25] introduces a set of conditions specifying what kind of DRAM indexes can be converted using the Recipe approach. PRONTO [31] introduces asynchronous semantic logging to convert each operation of the volatile index into a failure-atomic operation. TIPS [24] proposes a near black-box conversion strategy, which leverages a hybrid logging technique to guarantee crash consistency, prevent memory leaks, and promise durable linearizability. These works reduce the engineering complexity but introduce either high overheads (on runtime or memory space) or consistency compromises. *Tair-PMem* addresses these issues by providing a lightweight and high-performance programming toolkit, especially for building NVM-backed IMDBs.

Other works aim to exploit DRAM-NVM hybrid architectures for modern data-intensive systems, such as database systems, key-value stores, and file systems [7, 14, 21–23, 45, 46, 48]. Yan et al. [46] leverages NVM to revisit the conventional LSM-tree, which eliminates the WAL and proposes several log-free designs to further mitigate the persistence overhead of NVM. SLM-DB [22] is a key-value store that achieves high read performance by maintaining a B+-tree index in DRAM and reduces write amplification by adopting a single-level LSM-tree in NVM. NOVA [45] is a log-structured file system that stores each inode to a separate linked-list log to improve concurrency, and stores file data outside the log to minimize the log size and reduce garbage collection overheads. These works utilize NVM to propose new designs like eliminating WAL and reducing log size, therefore to better adapt NVM to existing systems. Similar to these works, *Tair-PMem* introduces design decisions like *log-as-user-data* to eliminate the logging overhead of Redis' AOF mechanism, so as to fully exploit the performance of NVM.

10 CONCLUSION

As the cloud service provider, we see NVM as a game-changer for IMDBs. Alibaba Cloud supported Redis-6.0 in May 2020 and provided *Tair-PMem* which is compatible with the former four months later. Till now, 5× more data is stored in *Tair-PMem* than Redis-6.0. We analyzed the major scenarios in which customers choose NVM services, including: the enterprise cloud migrations to reduce the Total Cost of Ownership (TCO); the advertisement systems requiring extremely low latency and hardly tolerating any latency spikes; the fintech replacing the combination of Redis and MySql with NVM services only; the online feature stores demanding high capacity and performance, and so on. Improving the durability, latency, data volume, and TCO while obtaining system stability makes the service attractive.

This paper shows how we leverage NVM to design a fully durable and enterprise-strength IMDB. *Tair-PMem* is the first cloud service that makes good use of the persistence capability of NVM. Specifically, *Tair-PMem* (1) provides the *hybrid memory programming toolkit* to hide complicated NVM programming details, (2) adopts the DRAM-NVM hybrid design according to a well-controlled data placement strategy, thus reducing the NVM access and persistence overhead. (3) develops a set of database components that achieve full Redis compatibility and advanced features in a low-intrusive manner for high stability. Our evaluation shows that, compared to Redis, *Tair-PMem* obtains full transaction durability, comparable throughput, an extremely fast recovery process after a normal shutdown, and avoids periodic latency spikes.

REFERENCES

[1] O Akinde Aderonke, O Okolie Samuel, and O Kuyoro'Shade. 2013. The S-Linked List–A Variant Of The Linked List Data Structure. *Journal of Emerging Trends in Computing and Information Sciences* 4, 6 (2013).

[2] Apache. 2008. Cassandra. Retrieved January 25, 2022 from http://cassandra.apache.org

[3] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[4] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*. 275–290.

[5] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 239–252.

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

[7] 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. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.

[8] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 799–812.

[9] Alibaba Cloud. 2021. Persistent memory-optimized instances. Retrieved January 25, 2022 from https://www.alibabacloud.com/help/en/doc-detail/183956.html

[10] Alibaba Cloud. 2022. ApsaraDB for Redis. Retrieved January 25, 2022 from https://www.alibabacloud.com/product/apsaradb-for-redis

[11] 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*. 143–154.

[12] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 377–392.

[13] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Changwoo Min, and Dongyoon Lee. 2020. WITCHER: Detecting Crash Consistency Bugs in Non-volatile Memory Programs. *arXiv preprint arXiv:2012.06086* (2020).

[14] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single machine graph analytics on massive datasets using Intel optane DC persistent memory. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1304–1318.

[15] Swapnil Haria, Mark D Hill, and Michael M Swift. 2020. MOD: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 775–788.

[16] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 967–979.

[17] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 187–200.

[18] Intel. 2018. Intel® Optane™ Memory - Responsive Memory, Accelerated Performance. Retrieved January 25, 2022 from https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html

[19] jemalloc. 2005. jemalloc memory allocator. Retrieved January 25, 2022 from http://jemalloc.net/

[20] Hai Jin, Zhiwei Li, Haikun Liu, Xiaofei Liao, and Yu Zhang. 2019. Hotspot-aware hybrid memory management for in-memory key-value stores. *IEEE Transactions on Parallel and Distributed Systems* 31, 4 (2019), 779–792.

[21] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 494–508.

[22] 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 *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 191–205.

[23] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 993–1005.

[24] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 773–787.

[25] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 462–477.

[26] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 429–444.

[27] Linux. 2021. fork(2) — Linux manual page. Retrieved January 25, 2022 from https://man7.org/linux/man-pages/man2/fork.2.html

[28] Linux. 2021. ioctl_ficlonerange(2) — Linux manual page. Retrieved January 25, 2022 from https://man7.org/linux/man-pages/man2/ioctl_ficlone.2.html

[29] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. Lb+ trees: optimizing persistent index performance on 3dxpoint memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[30] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1147–1161.

[31] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 789–806.

[32] Memcached. 2012. Memcached. Retrieved January 25, 2022 from https://memcached.org

[33] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.

[34] Oracle. 2009. MySQL. Retrieved January 25, 2022 from https://www.mysql.com/

[35] 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*. 371–386.

[36] 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.

[37] Redis. 2009. Redis. Retrieved January 25, 2022 from https://redis.io

[38] Redis. 2022. Redis Command. Retrieved January 25, 2022 from https://redis.io/commands

[39] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. 1–8.

[40] TieredMemDB. 2022. TieredMemDB. Retrieved January 25, 2022 from https://tieredmemdb.github.io/TieredMemDB/

[41] 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.

[42] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 623–637.

[43] xfs.org. 2013. Main Page. Retrieved January 25, 2022 from https://xfs.org/index.php/Main_Page

[44] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 349–362.

[45] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 323–338.

[46] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the design of LSM-tree Based OLTP storage engine with persistent memory. *Proceedings of the VLDB Endowment* 14, 10 (2021), 1872–1885.

[47] 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)*. 169–182.

[48] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 207–219.

[49] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment* 13, 4 (2019), 421–434.

[50] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 461–476.