PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery

Zhou Zhang
University of Science of Technology of China
zzwolf@mail.ustc.edu.cn

Zhaole Chu
University of Science of Technology of China
czle@mail.ustc.edu.cn

Peiquan Jin*
University of Science of Technology of China
jpq@ustc.edu.cn

Yongping Luo
University of Science of Technology of China
yoluo@mail.ustc.edu.cn

Xike Xie
University of Science of Technology of China
xkxie@ustc.edu.cn

Shouhong Wan
University of Science of Technology of China
wansh@ustc.edu.cn

Yun Luo
Tencent
cloudluo@tencent.com

Xufei Wu
Tencent
feiwu@tencent.com

Peng Zou
Tencent
lynzou@tencent.com

Chunyang Zheng
Intel Corporation
james.zheng@intel.com

Guoan Wu
Intel Corporation
dennis.wu@intel.com

Andy Rudoff
Intel Corporation
andy.rudoff@intel.com

ABSTRACT

Non-Volatile Memory (NVM) has emerged as an alternative to next-generation main memories. Although many tree indices have been proposed for NVM, they generally use B+-tree-like structures. To further improve the performance of NVM-aware indices, we consider integrating learned indexes into NVM. The challenges of such an integration are two fold: (1) existing NVM indices rely on small nodes to accelerate insertions with crash consistency, but learned indices use huge nodes to obtain a flat structure. (2) the node structure of learned indices is not NVM friendly, meaning that accessing a learned node will cause multiple NVM block misses. Thus, in this paper, we propose a new persistent learned index called PLIN. The novelty of PLIN lies in four aspects: an NVM-aware data placement strategy, locally unordered and globally ordered leaf nodes, a model copy mechanism, and a hierarchical insertion strategy. In addition, PLIN is proposed for the NVM-only architecture, which can support instant recovery. We also present optimistic concurrency control and fine-grained locking mechanisms to make PLIN scalable to concurrent requests. We conduct experiments on real persistent memory with various workloads and compare PLIN with APEX, PACtree, ROART, TLBtree, and Fast&Fair. The results show that PLIN achieves 2.08x higher insertion performance and 4.42x higher query performance than its competitors on average. Meanwhile, PLIN only needs ~30 μs to recover from a system crash.

PVLDB Reference Format:
Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, Andy Rudoff. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. PVLDB, 16(2): 243 - 255, 2022.
doi:10.14778/3565816.3565826

Proceedings of the VLDB Endowment, Vol. 16, No. 2 ISSN 2150-8097.
doi:10.14778/3565816.3565826

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tawnysky/PLIN.

1 INTRODUCTION

1.1 Background

The advance in main memory databases and near-memory big data processing calls for new memory technologies. Recently, Non-Volatile Memory (NVM), such as Phase Change Memory (PCM) [19], STT-MRAM [29], ReRAM [46], and Optane Persistent Memory [9, 13, 40], has offered an alternative to the next-generation main memories. Accordingly, many studies have proposed to revisit the key modules in database engines to make them NVM-aware, including indexing [7, 15, 24–26, 28], join processing [30, 37], and buffer management [43, 44].

Regarding NVM-aware indices, most of existing studies proposed to improve the conventional B+-tree to make it NVM friendly. One approach is to adopt an unordered node design in the B+-tree [5] and use data structures such as bitmap, slot array [6], or fingerprint [33] in nodes to mark the validity of slots and the order of keys. With this strategy, a newly inserted key to the B+-tree can be written to any free slot without moving other data in the node, thus reducing the number of `CLWB` and `SFENCE` instructions. Another approach is to keep the data ordered and move the data in nodes one by one when inserting a key. It does not need `CLWB` and `SFENCE` instructions when moving data in the same cacheline [14]. So far, most existing NVM-aware indices use small nodes, e.g., 256 bytes, because the current commercial NVM product, Intel Optane

DC persistent memory, accesses data at a block granularity of 256 bytes [24, 40]. Thus, the height of a tree index will become high when facing large datasets, which will worsen the read and write performance. For example, the height of the B+-tree with the 256-byte node size will be over 8 when 100 million keys are loaded. Since the current read latency of NVM is still higher than that of DRAM [9, 40], a high tree height will lead to a long traversal path and extra NVM accesses.

Recently, an emerging index structure, learned index [18], promises better read/write performance and space efficiency than conventional tree indices. A learned index can sense the distribution pattern of data and locate the storage position of a key by model inference inside nodes. The node size of a learned index is much larger than that of B+-tree, and each node can store thousands to hundreds of thousands of keys [42]. Therefore, the shape of a learned index is much flat, which usually has only two or three layers. Although the nodes of a learned index are large, they can quickly narrow down the search to tens of slots by model inference. Because of high space efficiency and search performance, the learned index has been one of the hottest research fields in the database community in recent years [10–12, 31, 38, 42].

1.2 Motivation

Therefore, an intuitive idea is to employ the learned-index approach to further improve the performance of NVM-aware indices. However, current NVM-aware indices do not consider machine learning. In addition, the existing crash-consistency schemes for NVM-aware indices are efficient only for small nodes but have high write costs for huge nodes that are common in a learned index. On the other hand, existing learned indices are not NVM friendly. Currently, learned indices mainly run on DRAM, but porting a learned index to NVM will degrade write performance. For example, as reported by previous work [4], simply implementing the state-of-the-art learned index ALEX [10] on NVM will lower the write performance by more than 80%. This is mainly because the node structure of existing learned indices is not NVM friendly. For instance, accessing a node of a learned index requires first fetching the model parameters in the node header, which needs an extra NVM block access; next, a local search is performed based on the predicted position, which will also cause multiple NVM block accesses. Therefore, current learned indices usually have low efficiency on NVM.

To the best of our knowledge, there is only one recent work called APEX [25] that has noticed the necessity of re-visiting learned indices for NVM. APEX is toward the hybrid memory architecture composed of NVM and DRAM. It uses several DRAM structures to maintain metadata information of the index. We noted that such a design would have high latency of crash recovery because all DRAM structures are required to be re-built during a crash. We implemented APEX and measured its recovery time, and the results are shown in Fig. 1. We can see that APEX needs about 1,700 ms for resuming throughput from a system crash in the single-thread environment. While evaluated in the multi-thread environment, APEX also costs about 160 ms for recovery.

Therefore, to offer instant recovery for learned indices on NVM, we turn to the NVM-only memory structure. Although current NVM products show a bit worse read/write performance than

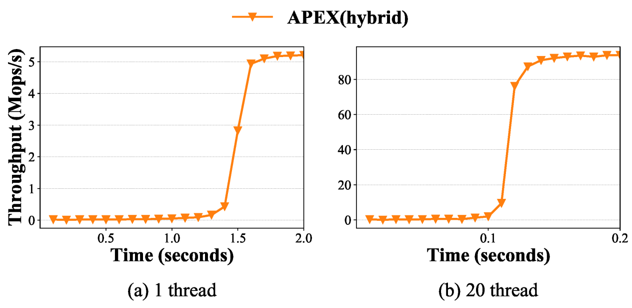


Figure 1: The recovery performance of APEX.

DRAM, it is reported that the next-generation NVM will deliver a high bandwidth comparable to DRAM, e.g., Intel has announced that its Optane 300 series has a 3-4 GB/s random I/O bandwidth and a 6-8 GB/s sequential I/O bandwidth. On the other hand, how to use NVM in memory hierarchy is still an open issue. While some indices were toward the DRAM+NVM-based hybrid architecture [7, 24, 33, 45], others focused on the NVM-only architecture [1, 6, 14, 15, 26, 28]. So far, both research directions are worth studying, and there is no evidence that one will dominate the future memory architecture. Notably, our study is the first one addressing the learned indices on the NVM-only architecture.

1.3 Our Contributions

In this paper, we propose a persistent learned index called **PLIN** (Persistent Learned INdex), which is toward the NVM-only architecture. PLIN aims to reduce the crash consistency cost of write operations and improve the efficiency of NVM block usage while maintaining the advantages of learned indices. It is also designed to offer instant recovery support for a system crash. Specifically, PLIN adopts several new designs, including NVM-aware data placement, locally unordered and globally ordered leaf nodes, model copy, and hierarchical insertion. It also uses optimistic concurrency control and fine-grained locking to support concurrent operations. Briefly, we make the following contributions in this paper.

- We devise a novel learned index structure for PLIN to make it NVM friendly. First, PLIN uses an NVM-aware data placement strategy in the nodes to ensure that only one NVM block is accessed for each request. Second, PLIN adopts the locally unordered and globally ordered policy to organize leaf nodes. Thus, it can avoid additional data movements caused by insertions while maintaining the high search performance of learned indices. Third, PLIN maintains the model parameters of a node in the parent node to avoid extra NVM block accesses. (Section 3)
- We detail the algorithms of the operations on PLIN, including query, upsert, delete, bulk load, and recovery. We demonstrate that the algorithms can reduce the number of NVM block accesses and ensure instant recovery. Particularly, we propose a hierarchical insertion strategy for PLIN, including deterministic insertion for leaf nodes, opportunistic insertion for bottom inner nodes, and no insertion for other inner nodes. The opportunistic insertion strategy is implemented with the bulk rebuilding of inner nodes, which can avoid costly data movement. (Section 4)

- We implement concurrency control to ensure the scalability of PLIN. To be more specific, we adopt an optimistic concurrency-control scheme to achieve reader-writer coordination and a fine-grained locking technique to achieve writer-writer coordination. We also use a multi-level locking mechanism to support leaf-node splitting and inner-node rebuilding. (Section 5)
- We conduct experiments on a server with real Intel Optane DC persistent memory and compare PLIN with state-of-the-art NVM-oriented indices, including APEX [25], PACtree [15], ROART [28], TLBtree [26, 27], and Fast&Fair [14], on various workloads. The experimental results show that PLIN has higher read and write performance than its competitors. Particularly, the read and write performance of PLIN is 4.49 times and 2.40 times higher than compared indices on average. Meanwhile, PLIN only needs ~30 μs to recover from a system crash. (Section 6)

2 RELATED WORK

2.1 NVM-Oriented Tree Indices

The Intel Optane DC persistent memory is the first commercial NVM. It has the following important characteristics that influence the design of indices on it. First, since NVM exchanges data directly with the CPU cache, crash consistency for data needs to be reconsidered [36]. Second, the Intel Optane DC persistent memory is accessed at a block granularity of 256 bytes [24, 40]. Third, it has a higher read latency and a lower bandwidth than DRAM [40].

To ensure crash consistency, a commonly used solution is to use a `CLWB` instruction to write CPU cachelines back to the NVM after each write instruction and then to use an `SFENCE` instruction to isolate subsequent write instructions [20, 36]. However, the insert operations in the B+-tree will cause data movements, which require multiple NVM write instructions invoking many `CLWB` and `SFENCE` instructions, which becomes the main cost of the crash consistency in B+-tree-like indices. Recent work has proposed a variety of approaches to reduce this cost, such as selective persistence [41], unordered leaf nodes [5], slot arrays [6], fingerprints [33], tolerating transient inconsistent states [14], batch updating [45], and speculative data movements [24].

For example, accessing an unordered node requires scanning the entire node, while tolerating transient inconsistent states requires moving data one by one within the node when handling insert operations, both of which are positively related in cost to the node size. Recent studies have shown that setting the node size of the B+-tree to 256 bytes, which is the access granularity of the Intel Optane DC persistent memory, is a good choice [24, 27]. Therefore, existing NVM-oriented B+-trees usually have a high tree height and a long traversal path.

So far, NVM-oriented indices can be classified into two types. The first type is for the NVM+DRAM hybrid memory architecture [7, 24, 33, 45], while the second type is for the NVM-only architecture. The indices for hybrid memory place the inner nodes in DRAM and the leaf nodes in NVM. However, since the inner nodes are not persistent, they need to rebuild the inner nodes during recovery. On the other hand, the indices for the NVM-only architecture place the entire index in NVM [1, 3, 14–16, 26, 28], which can offer instant recovery. At present, both directions are worth investigating, depending on the advance of the future NVM.

2.2 Learned Indices

The idea of the learned index is to use machine learning models instead of traditional index structures. The first learned index is RMI (*Recursive Model Index*) [18], which is a hierarchical model structure consisting of multiple models [34]. Each model in the RMI takes a key as input and returns a position. The output of the upper layer model is used to select the model of the next layer, and the output of the last layer model is used as the output of RMI. Compared with traditional indices, learned indices have lower space costs and higher query performance [31].

However, RMI has some drawbacks. First, RMI uses a simple uniform data partition strategy that cannot sense the data distribution during the learning phase. In the worst case, the data range cannot be fitted well by the given model, which will lead to a large maximum error of the model and increase the local search cost. Some recently learned indices use a bottom-up data segmentation strategy [11, 12, 17, 42]. This strategy performs a segmented fitting algorithm, which can complete data partitioning and model fitting by scanning the dataset in only one trip in a sequential manner. For example, PGM-index [11] uses a piecewise linear approximation algorithm called OptimalPLR [39], which guarantees the maximum error of each model is less than a user-specified threshold. In addition, unlike RMI which only stores keys in leaf nodes, most existing learned indices also store keys in inner nodes to ensure that the predictions of the upper layer models are immediately corrected and to avoid errors being amplified at the bottom layer [10, 12, 38, 42]. The learned index discussed in this paper also stores keys at each layer. Second, RMI does not support insertions because: (1) model retraining is costly; (2) insertions lead to model failure; and (3) the nodes in a learned index have a large amount of data, and insertions can lead to expensive data movement costs.

To solve problem (1), a common solution is to use simple models such as linear regression. To solve problem (2) and problem (3), existing solutions are mainly divided into out-of-place and in-place insertions. The out-of-place insertion is to write the new keys into a buffer and perform a bulk merge when the buffer is full [12, 35]. The performance of the buffer strategy is affected by the buffer size [42]. Another out-of-place insertion strategy is to use multi-level global buffers [11], which is similar to LSM-tree [32]. This strategy has good write performance but poor read performance because multiple buffers must be traversed for each query. The in-place insertion is to reduce the cost of data movement by redesigning the node structure. For example, ALEX [10] uses gapped arrays [2] to reduce the data movements during insertions.

2.3 NVM-oriented Learned Indices

Most of the existing learned indices were proposed for working on DRAM. It is promising to design an NVM-oriented learned index. On the one hand, the learned indices have flat structures, which can solve the problem of the high tree heights of existing NVM-oriented indices. On the other hand, NVM can make the learned indices persistent. However, designing an NVM-oriented learned index requires solving the crash consistency problem and reducing the high cost of NVM write instructions. In particular, the node size of a learned index is much larger than the NVM block granularity, and the data movement cost on NVM is higher than that of DRAM [4].

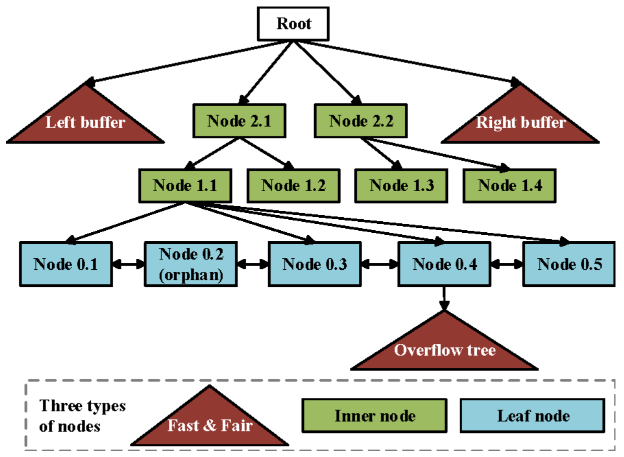


Figure 2: The overall index structure of PLIN.

The existing crash consistency solutions for B+-trees, such as the unordered nodes of wB+-tree [6] and the transient-inconsistency tolerance of Fast&Fair [14], are based on the assumption of small nodes, which cannot be directly applied to learned indices.

To the best of our knowledge, APEX [25] is the only NVM-oriented learned index. It places ALEX [10] on NVM and uses a probe-and-stash mechanism to reduce the cost of insert operations. It introduces accelerators that reside in DRAM for storing metadata, locks, bitmaps, and fingerprints. However, as APEX uses several DRAM structures to maintain metadata information of the index, it will lead to high latency of crash recovery because all DRAM structures are required to be re-built. Unlike APEX, in this paper, we present the first learned index for the NVM-only architecture, which can offer instant recovery and reduce DRAM usage.

3 INDEX STRUCTURE OF PLIN

3.1 Overall Structure

PLIN is a learned index designed for NVM-only architecture. All components of PLIN are preserved in NVM. The index structure of PLIN is shown in Fig. 2. The leaf and inner nodes are learned nodes, i.e., a machine learning model is used to accelerate the node search. PLIN uses a bottom-up recursive approach to build the index, so it is a balanced tree. PLIN fits the dataset using the OptimalPLR algorithm [39], and each node holds a linear regression model. All models are monotonically increasing and are guaranteed to train with a maximum error less than a user-given threshold ϵ. Each model includes two parameters *slope* and *intercept*. for a given *key*, PLIN uses (1) to predict the position of the key.

$$position(key) = slope \times key + intercept \tag{1}$$

In addition to the learned nodes, PLIN contains other data structures, including the root node, the left/right buffers, and the overflow trees. Among them, the root node and the left/right buffers are unique. The size of the root node is 256 bytes, which includes index metadata and a set of node pointers. The index metadata includes the minimum and maximum keys of the piecewise linear model and a pointer to the left/right buffers. PLIN decides whether to go to the learned nodes or the left/right buffers to lookup based on which range the key falls in. The left/right buffers are used to support the insertion of keys outside the coverage of the piecewise linear model. PLIN uses a well-known NVM-oriented B+-tree called Fast&Fair [14] as the left/right buffers and the overflow trees. Any other NVM-oriented data structure can be used as an alternative. The structure of the leaf and inner nodes will be described below.

3.2 Key Designs

To improve the read and write performance, PLIN is designed to reduce the number of NVM blocks to be accessed for index operations and the number of `CLWB` and `SFENCE` instructions required for insert operations. The novel designs of PLIN are as follows.

NVM-aware data placement. A search on a conventional learned index might read many slots around the predicted slot because the predicted position may not be exact as expected. Thus, a search operation will read multiple NVM blocks. To solve this problem, we propose an NVM-aware data placement strategy, ensuring most searches only need to read one NVM block. The core idea of the NVM-aware data placement is to adjust the data slot when building nodes. (1) Why is the adjustment allowed? This is because PLIN always remains some empty slots in a node. (2) How to adjust the data slot when placing data? We first place the data in the position obtained from the model calculation. If the position has been used, we place the data in another slot within an NVM block size (256 bytes). (3) How to deal with conflicts? We divide the node space into NVM blocks and allow data to be offset within the blocks. If the NVM block is full, an overflow tree will be used. (4) What are the benefits of NVM-aware data placement? We decouple the local search range from the maximum error of the model. Thus, a local search need not access multiple NVM blocks.

Locally unordered, globally ordered leaf nodes. Inserting a key using an in-place insertion strategy may cause the movement of other keys, resulting in multiple write instructions. To ensure crash consistency, these write instructions need to be isolated using the `SFENCE` instructions and guaranteed to be persisted to NVM using the `CLWB` instructions, which results in very costly data movement. Unordered nodes are usually used to avoid data movement, but each lookup requires scanning the entire node, so they are not suitable for the large nodes of a learned index. Therefore, we propose locally unordered and globally ordered leaf nodes, which can reduce extra data movements and maintain high lookup performance simultaneously. Specifically, PLIN uses an unordered design within each block and keeps the order between blocks. As each model is monotonically increasing, the NVM-aware data placement ensures the order between blocks. With this approach, each insertion only needs to find an empty slot in the block to execute the write instruction, which does not cause the movement of other keys.

Storing a model copy in the parent node. In a learned index, when accessing a node, the model parameters need to be fetched from the node header first. PLIN uses a piecewise linear approximation to fit the dataset, with each node corresponding to a linear regression model. The parameter size of a linear regression model is only a dozen bytes. However, an additional NVM block needs to be accessed to obtain the model parameters, resulting in inefficient use of the NVM block. To solve this problem, PLIN generates a copy of

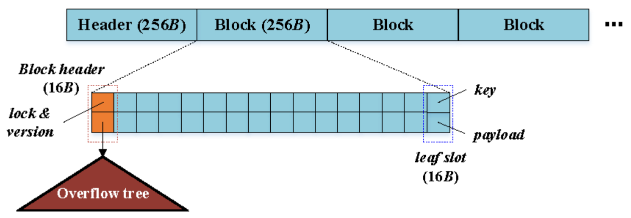


Figure 3: The structure of leaf nodes.

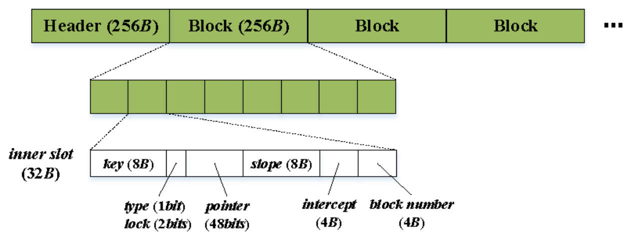


Figure 4: The structure of inner nodes.

the parameters needed to perform the prediction, packed with the key and pointer of the node in the parent node. This approach can skip accessing the node header when accessing the node, improving the efficiency of the NVM block usage.

Hierarchical insertion. The unordered node design is not suitable for inner nodes. This is because most operations in inner nodes are lookup operations based on a lower bound, which can only be performed in ordered arrays. Thus, the keys in an inner node of PLIN are ordered. In the worst case, performing an insertion may cause all the keys in the node to move position, which is very bad for large nodes. PLIN uses a hierarchical insertion strategy to avoid this case. In leaf nodes, PLIN guarantees to perform insertion every time. Insertion may lead to splitting of a leaf node, which results in insertions in an inner node. PLIN adopts an opportunistic insertion strategy in inner nodes. Specifically, insertion is executed only when an empty slot exists in the target block of the key to be inserted. If an empty slot does not exist in the target block, this insertion is stopped. The leaf node that is not inserted into the parent node is called an orphan node. PLIN uses a bidirectional pointer to connect all the leaf nodes to ensure that the keys in the orphan nodes can also be found. When the number of orphan nodes is large, PLIN rebuilds all inner nodes.

3.3 Structure of Leaf Nodes

The structure of leaf nodes in PLIN is shown in Fig. 3. Since the access granularity of Intel Optane DC persistent memory is 256 bytes [40], we align the node space by 256 bytes and divide it into blocks of 256 bytes size. The first block is used as the node header to store node metadata, including locks, model parameters, number of blocks, number of overflow keys, and the sibling pointers to the left and right sibling nodes. The later blocks are used to store data. Each block contains 15 leaf slots, and each leaf slot has a size of 16 bytes and consists of an 8-byte key and an 8-byte payload. A payload can either store a value whose size is not over eight bytes or a pointer to a value larger than eight bytes. The first 16 bytes of each block are used as the block header. Among these, eight bytes are used to store a lock, a block version, and a global version, and the other eight bytes store a pointer to the overflow tree. The overflow tree is constructed only when necessary, and a null pointer is stored if the overflow tree does not exist.

The leaf nodes of PLIN use the in-place insertion strategy and the NVM-aware data placement strategy. When building a leaf node, we allocate twice as much space (same as B+-tree) to support insertions and extend the mapping of the model to the scaled node space. Then, we use the model to predict the position of each key and place the key and the corresponding payload into any free leaf slot in the block predicted by the model. If the block is full, the data is inserted into the overflow tree. The leaf nodes are partially ordered, i.e., keys within a block are unordered, but all blocks are ordered. The idea of making keys unordered in a block can avoid additional data movements during insertions which generate additional write instructions. However, as the models are monotonically increasing, it is natural to keep order between blocks. In a leaf node, we do not use a bitmap to mark whether the leaf slot is occupied or not. Instead, since the model coverage is limited, we use a key outside the model coverage as a free flag to fill the free slots in a node. Specifically, PLIN uses the largest possible value of the key's type (e.g., 65535 for *uint16_t*) as the free flag and stores the keys with the largest possible value in the right buffer to avoid collisions. Thus, the free flag is consistent across nodes and needs not to be updated when rebuilding a node. The delete flag is implemented similarly.

3.4 Structure of Inner Nodes

We also make the space of an inner node 256-byte aligned and divide it into 256-byte blocks. The first block stores metadata and the others store data. Each block contains eight 32-byte inner slots.

As shown in Fig. 4, an inner slot contains an eight-byte key and a pointer to a child node. An inner slot also contains a copy of the parameters of the child node, including the model parameters *slope* and *intercept* and the number of blocks in the child node. Using these three parameters, we can predict the position of the key in the child node before accessing the child node without having to access the header of the child node. When performing the prediction, the *intercept* does not need too much precision since it needs to be rounded at the end, and we allocate four bytes for it. We allocate four bytes for the number of blocks, which can support nodes with up to 64 billion slots and cope with almost all cases. An inner slot also contains some special bits, where *type* is used to mark whether the child node is a leaf node or an inner node, and *lock* is used to mark whether the child node is splitting, which contains a read lock and a write lock. On today's AMD64 and Intel® 64 architectures, only 48 bits of the pointer are valid, and we integrate these two special bits into the eight bytes of the pointer. A key outside the coverage of the model is also used as a free flag in an inner node to fill all free inner slots.

All slots in an inner node are stored in order, and we also use the NVM-aware data placement for inner nodes. When building an inner node, we prioritize the placement of a key in the block predicted by the model. Unlike the leaf nodes, we allow the key using the space of the subsequent blocks if the block is full. Therefore, no overflow block exists in an inner node. Instead, we allocate additional ϵ/N blocks for each inner node to ensure that the overflow data in the last block can be loaded, where ϵ is the error threshold of the model and N is the number of inner slots in each block.

4 OPERATIONS OF PLIN

4.1 Query

PLIN supports point and range queries. The input of a point query is a key, and the output is the payload corresponding to the key. In the index metadata, we reserve the coverage of the piecewise linear model, i.e., a minimum key and a maximum key. The query first visits the metadata and determines whether the target key is within the coverage of the model. If it is, we need to access the root node; otherwise, we access the left/right buffer. When accessing each node of PLIN except the root node, we use a copy of that node's model parameters in the parent node to predict the position of the target key in the node. Since the inner nodes of PLIN use an opportunistic insertion strategy, there are a small number of orphan nodes in the leaf nodes. An orphan node can only be found through its left sibling node. Since no parent node exists for an orphan node, it is necessary to access the node header to obtain the model parameters. Then, we perform a local search based on the predicted position. If the node is an inner node, we search left or right until we find the largest key in the node that is less than or equal to the target key. If the node is a leaf node, we compute the starting address of the block where the predicted position is located and scan the entire block. If the target key is found, the corresponding payload is returned. Otherwise, the current block is checked to see if there is an overflow tree. If it exists, the overflow tree is searched. Since we use a piecewise linear approximation algorithm to train the model, we can guarantee the maximum error of the model is less than a threshold ϵ.

The input of a range query is a lower bound and an upper bound, and the output is all the indexed keys covered by the range as well as the corresponding payloads of the keys. The first half of the range query proceeds the same as the point query, where we find the leaf node and the block where the smallest key greater than or equal to the lower bound is located. Then, since all leaf nodes of PLIN are ordered, and all blocks in a leaf node are also ordered, we perform a block-grained scan, which processes one block and its corresponding overflow tree until a key larger than the upper bound is found. Since the leaf nodes are linked by bidirectional pointers, the scanning process does not need to backtrack the inner nodes.

4.2 Upsert

PLIN requires the key in the index to be unique and supports upsert operations. When a key and its payload need to be inserted into PLIN, we need not perform a lookup to determine whether the target key already exists in the index. The upsert interface of PLIN can make this judgment automatically. If the target key does not exist, the key and payload are inserted into the index. If the target key exists, the payload of the key is changed to the newer one. The first half of the upsert process is the same as the point query, which requires first finding the leaf node where the target key is located. Then, we use the model to predict a block in the leaf node and scan this block. If there is a target key in the predicted block, the update is executed. Otherwise, the algorithm tries to perform an insertion. We scan the predicted block again and look for a free slot, i.e., the key of the slot equals the free flag. If a free slot is found, we insert the key and payload into the slot. Specifically, we first write the eight-byte payload, and then write the eight-byte key.

The upsert operation of PLIN is crash consistent. After an update or insertion, we use a `CLWB` instruction to write the cacheline back to NVM and an `SFENCE` instruction to isolate the subsequent write instructions. In the case of insertion, we need to write two eight-byte contents, i.e., two CPU write instructions are required. However, we only need to use one `CLWB` instruction and one `SFENCE` instruction. Since the node space is aligned by 256 bytes, the contents of the same leaf slot must be in the same cacheline. The CPU can guarantee that the write operations in the same cacheline will be executed in order. Therefore, the write of the payload and the key will not be disordered. One problem is that the writing payload is completed, but the key has not been written into the node. If the system crashes between a key write and a payload write, since the key of the leaf slot still equals the free flag, it is recognized as a free slot, and the status of the index remains unchanged, meaning that the insertion is automatically aborted. In the inner nodes, we also use a similar method to insert an inner slot.

If a free slot is not found in the predicted block, the upsert of the overflow tree needs to be executed. We have slightly modified the source code of Fast&Fair to make it support the upsert interface. Its upsert algorithm returns a flag that tells PLIN whether the execution is an update or an insertion. If it is an insert, we add one parameter $overflow\_number$ stored in the node header. A leaf node uses this parameter to count the amount of overflow data in the node and to decide whether to perform a split. If the overflow data exceeds a percentage of the node size, the algorithm returns a split flag, which informs PLIN of performing a split operation. The split operation is done by a background thread, and the thread responsible for the upsert operation does not need to wait for it.

4.3 Structure-Modified Operations

When a certain upsert returns a split flag, PLIN performs a split operation for the leaf node. First, the left and right siblings of the target node need to be recorded. Then, we scan all the data in the target node, retrain the model, and build a new set of leaf nodes. Unlike the B+-trees, a split operation in PLIN may yield more than two new nodes or only one node. The number of new nodes depends on the number of segments output by the piecewise linear approximation algorithm [39]. We use the REDO logging to guarantee the crash consistency of split operations. Although some NVM-oriented B+-trees proposed to use a log-free manner [1, 14, 24], it is not costly to use logs in PLIN because the nodes of PLIN are large, and split operations will not be triggered frequently. Relatively, each split is more expensive, so using logs for split operations does not add many extra costs. We use the REDO logging to ensure that all pointer-modification operations generated by a split operation are done. The pointer modification operations include sibling pointer updates in left/right sibling nodes and pointer updates and insertions in the parent node.

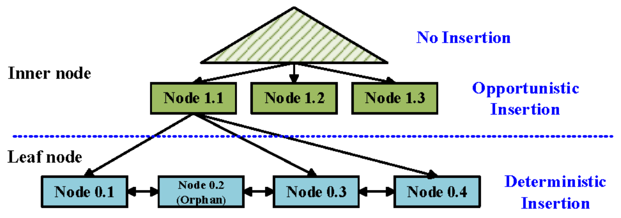


Figure 5: The hierarchical insertion strategy.

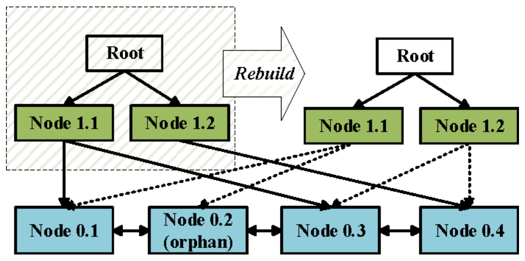


Figure 6: Bulk rebuilding of inner nodes.

As shown in Fig. 5, PLIN uses different insertion strategies for nodes at different layers. For leaf nodes, PLIN uses a deterministic insertion strategy to ensure that the new key can be inserted into the index. Note that PLIN's overflow tree is a B+-tree that can grow without limit. Therefore, any insert operation in a leaf node will not fail even if the leaf node is full. For the parent nodes of leaf nodes, PLIN uses an opportunistic insertion strategy to insert the new leaf nodes generated by a split into the parent node. First, we use the model to locate a block in the node and then look for a free inner slot inside. If found, we perform the insertion. If not found, the insertion is aborted, but the splitting will not be canceled. Then, the leaf node produced by the splitting becomes an orphan node, which is linked to its left sibling node. Since all the slots in an inner node are ordered, the opportunistic insertion strategy prevents a single insertion from causing many data movements. The opportunistic insertion strategy guarantees that only the parent nodes of leaf nodes will perform insertions. Since no splitting is triggered in the parent nodes, no insertion will occur in the upper layer of the parent nodes. PLIN monitors the number of orphan nodes. When the number of orphan nodes exceeds a certain percentage of the total number of leaf nodes, PLIN rebuilds all inner nodes, as shown in Fig. 6. The rebuild operation eventually modifies the root node pointer and ensures crash consistency.

4.4 Delete

When a record needs to be deleted, PLIN finds the target key in the block and updates it to a delete flag. Similar to the free flag, the delete flag is also a key outside the coverage of the model, but is not equal to the free flag. The query operations will skip the slot with a delete flag, while the upsert operations treats it as a free slot. Similar to the state-of-the-art indices [10, 38], the delete operation in PLIN does not result in node merging because real-world data tends to grow over time.

4.5 Bulk Load

PLIN requires learning the distribution of the dataset, so it does not support the case of inserting from zero. We propose to use a B+-tree to handle this case and convert the index to PLIN when the dataset size increases. A bulk load interface is provided by PLIN to load the dataset. The bulk load requires the input data to be arranged in the order of the key. We use the OptimalPLR algorithm [39] to scan a trip to the dataset and get a set of models. After that, leaf nodes are built using each model and the corresponding piece of data in turn. We use the minimum key of each node as a new dataset to construct the inner nodes and recursively execute this process until the root node can load the output nodes. When building a leaf node or an inner node, we first allocate extra space to the node proportionally and adjust the mapping range of the model to the whole node space. Then, we use the free flag to fill all leaf/inner slots and use the NVM-aware data placement strategy to place data into the slots.

4.6 Instant Recovery

Since NVM is a persistent storage device, the data structures on NVM need to be guaranteed to recover quickly from a system crash. As all the structures of PLIN are reserved in NVM and all upsert operations are atomic, PLIN can ensure instant recovery. During recovery, we only need to check the REDO log and re-execute the pointer update operations in the log, which is updated during structure-modified operations. Note that APEX [25] uses many DRAM structures in its implementation. Thus, it needs to rebuild the structures in DRAM during a recovery process, which is time-consuming and will lower the throughput.

5 CONCURRENCY

Using node-level locking mechanisms is inefficient because learned indices use huge nodes and result in expensive costs in node splitting. Some learned indices on DRAM use a two-phase compaction mechanism based on read-copy-update (RCU) to support lock-free node splitting [23, 35]. However, the two-phase compaction mechanism causes write amplification and is unsuitable for use on NVM. PLIN adopts optimistic concurrency control [21, 22] and fine-grained locking with the following design principles: (1) Minimize the time to lock the entire node. (2) Accessing additional NVM blocks due to lock acquisition should be avoided unless a structure-modified operation occurs.

PLIN uses multiple granularities of locks and versions. In the index header, PLIN reserves a global lock $lock_g$ and a global version $version_g$. In the node header in an inner node or a leaf node, PLIN reserves a node-level lock $lock_i$ or $lock_l$, respectively. In the block header of a block in a leaf node, PLIN reserves a 1-bit block-level lock $lock_b$, a 31-bit global version $version_g$, and a 32-bit block-level version $version_b$. In the following, we describe how PLIN uses these locks and versions for concurrency control.

Reader-writer coordination. PLIN adopts optimistic concurrency control for reader-writer coordination. Query operations need to check $lock_b$ but do not need to modify it. Upsert operations and delete operations need to acquire $lock_b$ and update $version_b$ before writing in a slot. The query operations check $version_b$ once when checking $lock_b$ and after fetching the record. If $version_b$ obtained twice are equal, the read is successful; otherwise, the record must be fetched again. Since both $lock_b$ and $version_b$ are located inside the block, we need not access an additional NVM block.

Writer-writer coordination. Upsert operations and delete operations require acquiring $lock_b$ and updating $version_b$. $lock_b$ is used to block other write operations, so a block (containing the overflow tree) allows only one thread to perform a write operation at a time. Since PLIN uses block-level locks instead of node-level locks, the blocking probability is substantially reduced. Note that the use of fine-grained locks presupposes that PLIN uses an NVM-aware data placement policy. Write operations in other learned indices (e.g., ALEX [10]) involve the entire memory space of the node, so fine-grained locks cannot be used.

Splitting of leaf nodes. A split operation of a leaf node requires using node-level locks. $lock_l$ contains three status bits, which are used to indicate blocking split operations, blocking write operations, and blocking read operations, respectively. When a node splitting occurs, (1) acquire $lock_l$ in the splitting node and the left and right sibling nodes, set the lock status of the splitting node to block split operations and write operations, and set the lock status of the left and right sibling nodes to block split operations. (2) Get the data in the splitting node, train new models, and build new leaf nodes. (3) Acquire $lock_i$ in the parent node to block other write operations and add blocking read operations to the lock status of $lock_l$ in the splitting node. (4) Update the left and right sibling node pointers and update the pointers in the parent node. (5) Release all locks. In step (1), if any lock acquisition fails, this split operation will be abandoned immediately without blocking the subsequent operations of the thread. Since the overflow trees in PLIN have no capacity limit, giving up a split operation does not cause the node to be unable to insert new records. The next insert operation on that node will cause PLIN to try a split operation again. In addition, most of the time in a split operation is spent on step (2), which does not block read operations. Finally, since read/write operations require checking the read/write status of $lock_l$, PLIN reserve a copy of these two status bits in the inner slot in the parent node. Therefore, read and write operations do not need to access the node header to check the lock status.

Rebuilding of inner nodes. A split operation is not allowed during the rebuilding of inner nodes in PLIN. $lock_g$ is used to avoid this kind of conflict. $lock_g$ contains a status bit and a 31-bit reference count. When a node splitting occurs, the reference count will be increased. When a rebuild operation is triggered, the status bit of $lock_g$ is set to block split operations. After this, no new split operations are allowed. PLIN waits for the ongoing split operations to complete and begins the rebuild operation. Similarly, a blocked split operation does not need to wait but is aborted immediately.

Avoiding deadlocks. Since NVM is a persistent storage device, deadlocks caused by crash recovery need to be avoided. The global lock and node-level locks are used only when a structure-modified operation occurs, so we can complete the operation and release the locks by checking REDO logs. For the block-level locks, PLIN uses $version_g$ to avoid deadlocks. PLIN reserves $version_g$ in the index header and updates the version number after a recovery. When a write operation acquires $lock_b$, it checks $version_g$ in the block, and only a lock with matching $version_g$ is valid. If $version_g$ does not match, which means $lock_b$ is out-of-date, $version_g$ is updated to the new one when $lock_b$ is fetched.

Table 1: Server configuration.

Component	Description
CPU	Intel® Xeon® Gold 6242 CPU Dual-socket with 40 cores at 3.1GHz
L1 cache	32 KB iCache & 32 KB dCache (per-core)
L2 cache	1 MB (per core)
L3 cache	36 MB (per socket)
Total DRAM	256 GB (2 (socket) x 4 (channel) x 32 GB)
Total NVM	2,048 GB (2 (socket) x 4 (channel) x 256 GB)

6 PERFORMANCE EVALUATION

In this section, we first detail the experimental settings and then report the comparative results between PLIN and other indices.

6.1 Experimental Setup

We conduct experiments on a server equipped with real Intel Optane DC persistent memory. Table 1 shows the environment configuration for the experiments. The server contains 256 GB DRAM and 2,048 GB Intel Optane DC persistent memory, equally distributed over two Sockets. We configure all Optane modules to `App-Direct` mode and create a DAX-aware ext4 file system. Then, we mount the file system using the DAX option.

Datasets. We use three datasets in our experiments, namely, the Normal dataset, the Lognormal dataset, and the OSM dataset. In the first two datasets, the keys are randomly generated numbers. The keys in the OSM dataset are the earth longitude attribute of the entity object extracted from the OpenStreetMap dataset[1]. Each dataset has 200M key-value pairs, where both the key and value size is 8 bytes. The keys in each dataset are unique.

Competitors. We compare PLIN with five NVM-oriented indices and use their open-source codes in our evaluation, including APEX[2], TLBtree[3], Fast&Fair[4], ROART[5], and PACTree[6]. TLBtree [26, 27] and Fast&Fair [14] are two state-of-the-art NVM-oriented B+-tree-like indices. PACTree [15] and ROART [28] are two state-of-the-art NVM-oriented trie-like indices. APEX [25] is the state-of-the-art learned index designed for the NVM+DRAM hybrid memory architecture. It maintains sophisticated data structures on DRAM to accelerate the query and write process. As APEX is the only NVM-aware learned index, we also include it in the comparison. However, to make the comparison fair and without changing the key idea of APEX, we place all its data structures on NVM to make it suitable for the NVM-only architecture. Specifically, when APEX needs to allocate memory for a DRAM structure, we will allocate a piece of persistent memory for it. The DRAM structures include locks, metadata, stash bitmaps, accelerators, and overflow buckets [25]. Write operations on these structures do not use the `CLWB`

[1]The OpenStreetMap dataset. https://registry.opendata.aws/osm.
[2]APEX source code. https://github.com/baotonglu/apex.
[3]TLBtree source code. https://github.com/ypluo/TLBtree.
[4]Fast&Fair source code. https://github.com/DICL/FAST_FAIR.
[5]ROART source code. https://github.com/madsys-dev/ROART.
[6]PACtree source code. https://github.com/cosmoss-vt/pactree.

and SFENCE instructions because they are not required to guarantee crash consistency. Note that we also include the experimental comparison between PLIN and the original APEX version running on DRAM and NVM in Section 6.5. To make the presentation clear, in the following text, we use the notation "**APEX**" to represent the APEX implementation for the NVM-only architecture and use "**APEX(hybrid)**" to indicate the original APEX implementation for the DRAM-NVM hybrid architecture.

We implemented PLIN using C++. We use PMDK to handle large chunks of persistent memory allocation and use the memory allocator offered by TLBtree [27] to allocate small chunks for nodes efficiently. TLBtree and Fast&Fair also use the same interface. For APEX, PACtree, and ROART, we use their own persistent memory allocators. The leaf node size of TLBtree/Fast&Fair/ROART/PACtree is set to 256/512/1024/1024 bytes. All competitors use CLWB and SFENCE instructions to ensure crash consistency. Because we do not focus on the optimization of the NUMA effect, all the persistent memory is allocated on one NUMA node, and all tasks are running on the same node. Each set of experiments uses the "-O3" optimization. Before each experiment, we execute 100 million query operations as a warm-up to ensure that the CPU cache is used efficiently. Note that all indices (except the original APEX discussed in Section 6.5) only use the NVM capacity, i.e., all the structures and data of an index are placed on NVM.

Default parameters. Some adjustable parameters are used in the PLIN implementation. If we do not talk about them specifically, we use the default values of the parameters in our experiments. We set the block size in the leaf/inner nodes to 256 bytes to match the NVM block granularity for a good locality. The OptimalPLR algorithm [39] requires a maximum error ϵ for the input. If we use a larger ϵ, each model can cover more data, thus reducing the number of nodes and decreasing the tree height. However, using a larger ϵ may lead to more overflow data. We use a large ϵ, i.e., 256, in the leaf nodes to ensure that the tree is flat enough. And we use a small ϵ, i.e., 16, in the inner nodes to reduce overflow data in the inner nodes. Since PLIN can significantly reduce the number of leaf nodes, using inner nodes with loose density will not incur high space costs. Therefore, we set the initial fill rate of inner nodes to 20% to avoid frequent triggering of rebuilding. Moreover, we set the maximum overflow rate to 30% and trigger node splitting if the amount of the overflow data exceeds this rate. Also, we set the maximum orphan rate to 10% and start rebuilding inner nodes if the number of the orphan nodes exceeds this rate.

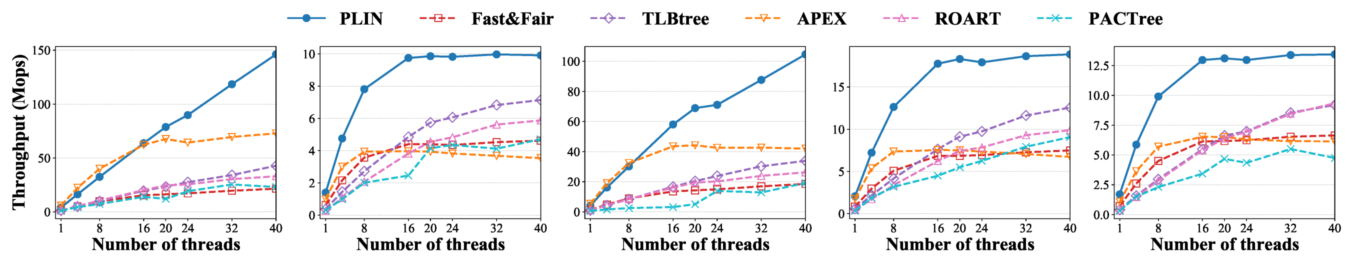
6.2 Read and Write Performance

In this experiment, we run all indices on five workloads that have different read and write ratios, namely *write-only*, *upsert*, *write-heavy* (50% write and 50% read), *read-heavy* (5% write and 95% read), and *read-only*. The last three workloads correspond to YCSB A, B, and C workloads [8], respectively. Each write operation except *upsert* inserts a new key into the index. The workload *upsert* consists of 50% insertions and 50% updates. The read and update operations in all workloads follow the Zipfian ($zipf$=0.99) distribution. The experiments scale from one thread to 40 threads. When the number of threads is less than or equal to 20, each thread uses a separate physical core. When the number of threads exceeds 20, the exceeding threads will use hyper-threading. Figures (7-9) show the throughput of PLIN and the other five indices on the three datasets. The unit of the Y-axis is *Mops*, which means one million read/write operations per second. We observe that PLIN exhibits much better performance than the other indices on all workloads and datasets. Note that the running time of PLIN includes all inner-node rebuilding time.

Figures (7-9)(a) show the throughput of the six indices on the read-only workload, PLIN scales nearly linearly to the thread number. When the thread number varies from 1 to 8, the read throughput of PLIN is slightly lower than that of APEX. However, when the thread number increases from 16 to 40, APEX does not get many benefits, but PLIN shows an explicit throughput increase and eventually achieves up to 2.01 times higher throughput than APEX. We can see that APEX has a performance drop when the number of threads is increased from 20 to 24, this is because of the impact of hyper-threading. In contrast, PLIN is not affected significantly. By analyzing the runtime statistics of APEX and PLIN, we find that the query process of APEX requires accessing more NVM blocks. Specifically, for a query operation, the model, locks, and data of the leaf node are located in different NVM blocks. Moreover, each query of APEX needs to update the metadata. Therefore, the contention for NVM bandwidth limits its scalability. In PLIN, accessing the model and locks of a leaf node does not result in additional NVM block accesses. However, APEX usually needs to scan only one cacheline in a leaf node to find the target key, while PLIN usually needs to scan more than two cachelines. Therefore, the single-threaded read performance of APEX is slightly higher than that of PLIN.

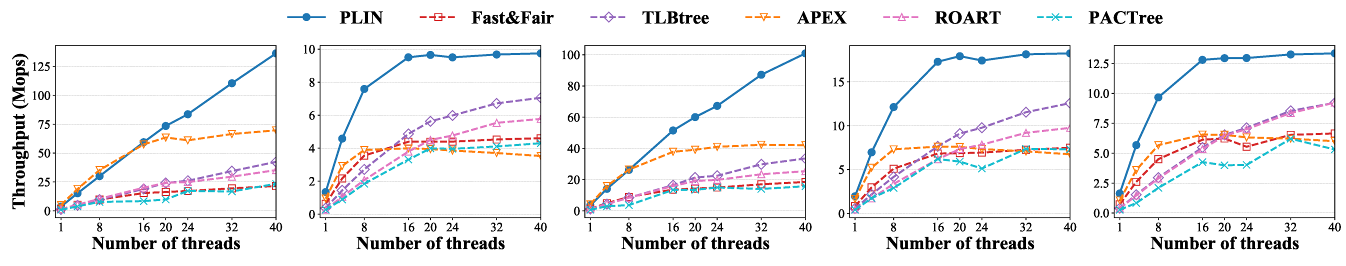
Figures (7-9)(b) show the throughput of the six indices on the write-only workload. The figures show that PLIN achieves the best performance and scalability compared to other indices. More specifically, when the thread number varies from one to eight, PLIN scales nearly linearly due to the fine-grained lock mechanism. While the thread number increases from 16 to 40, the growth of PLIN's throughput becomes slow. PLIN achieves 1.38-1.41 times higher throughput than the second place under 40 threads. We can see that APEX also scales well when the thread number varies from one to eight. While the thread number ranges from 16 to 40, the throughput of APEX is even decreasing, and it gets the worst performance under 40 threads. This is because APEX requires metadata and auxiliary data structures to be updated during insertions, resulting in multiple NVM writes. It can cause severe NVM bandwidth contention in multi-thread concurrent environments.

Figures (7-9)(c) and (d) show the throughput of the six indices on the mixed workload. We can see that PLIN achieves the best performance again. More specially, all the indices exhibit a similar performance trend on the read-heavy workload and the read-only workload, and PLIN achieves 2.22-2.49 times higher throughput than the second place under 40 threads. For the write-heavy workload, all the indices have the same trend as the write-only workload, and PLIN achieves 1.45-1.50 times higher throughput than the second place under 40 threads. We can see that PLIN performs better when the workload has a higher read ratio. This is because the performance improvement of PLIN is mainly owing to the reduction of the index height and the number of NVM blocks accessed.



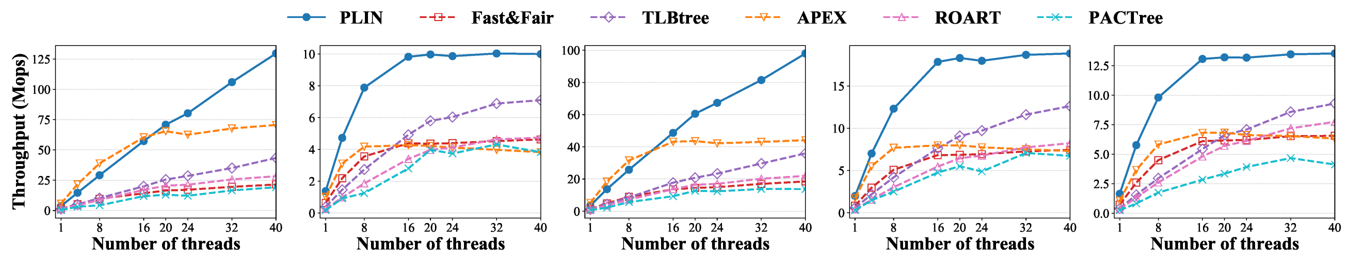
(a) Read-only workload (b) Write-only workload (c) Read-heavy workload (d) Write-heavy workload (e) Upsert workload

Figure 7: Throughput on the Normal dataset.



(a) Read-only workload (b) Write-only workload (c) Read-heavy workload (d) Write-heavy workload (e) Upsert workload

Figure 8: Throughput on the Lognormal dataset.



(a) Read-only workload (b) Write-only workload (c) Read-heavy workload (d) Write-heavy workload (e) Upsert workload

Figure 9: Throughput on the OSM dataset.

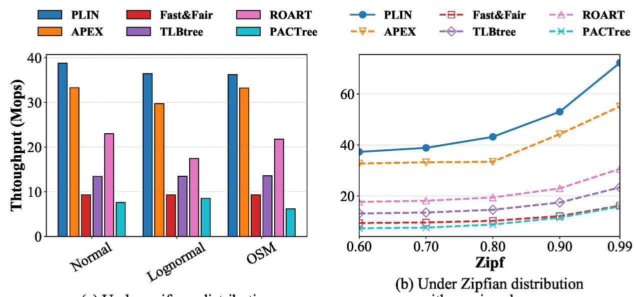
Figures (7-9)(e) show the performance of the six indices on the workload *upsert*. We observe that the performance of upsert operations follows a similar trend as it exhibits on the write-only workload. PLIN also scales well when the thread number ranges from one to eight, while the performance stops scaling beyond 16 threads because of the impact of synchronization. However, PLIN still achieves the best performance, and its throughput on the upsert workload is about 1.46 times higher than the second place under 40 threads.

We observe that the read and write performance of PLIN and APEX is affected by the data distribution. PLIN achieves the best read performance on the Normal dataset and the best write performance on the OSM dataset. APEX performs best on the OSM dataset for both read and write. Although both PLIN and APEX aim to fit the data distribution, there are still some data that are hard to be fitted, which will cause performance degradation. The performance of ROART and PACTree is also influenced by the data distribution because they both adopt the trie structure. Hence, different datasets may produce different index structures, which impact the performance. In summary, PLIN can achieve high and scalable performance on read-intensive and write-intensive workloads.

6.3 Performance with Varying Access Patterns

In this experiment, we evaluate the query performance when varying the access pattern. We main vary the skewness of accesses to see whether PLIN can maintain the performance superiority over other indices.

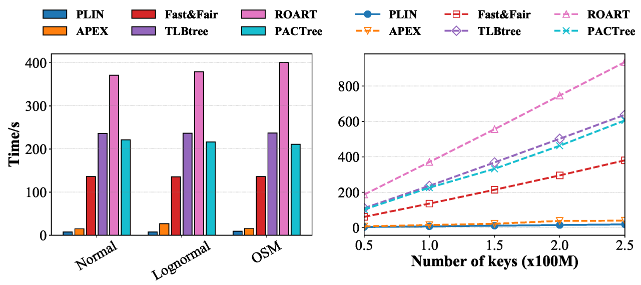
Figure 10(a) shows the query throughput on the three datasets under 20 threads with random accessing patterns. It can be seen from the figure that PLIN is still the fastest one among the six indices. Figure 10(b) shows the query throughput on the Lognormal dataset under 20 threads with varying skewness, indicating that all the indices achieve better performance with higher skewness. High skewness means that most accesses focus on a small set of hot keys; therefore, the index can utilize the cache better. In addition, PLIN achieves the highest performance with all skewness, meaning that it can maintain stable performance for different access patterns.



(a) Under uniform distribution

(b) Under Zipfian distribution with varying skewness

Figure 10: Throughput under the uniform and Zipfian distributions with varying skewness.



(a) On various datasets

(b) On various data scales

Figure 11: Comparison of the bulk load time.

6.4 Bulk-Load Performance

Figure 11(a) shows the bulk load times of PLIN and competitors on different datasets, with 100 million keys loaded for each set of experiments. Among the six indices, only PLIN and APEX provide a bulk load interface. Since the other four indices do not provide a bulk load interface, we call their insert interfaces to insert keys one by one. We can see that the bulk load speeds of PLIN and APEX are much faster than other indices. On all datasets, PLIN consumes less than 15 seconds, while APEX consumes 2x more time than PLIN. Moreover, the other four indices consume at least 100 seconds. This is because the fitting algorithm used by PLIN requires only one scan of the dataset, i.e., it has an $O(n)$ time complexity. Also, since the node number and the tree height of PLIN are much less than those of other competitors, PLIN builds fewer nodes and processes fewer layers recursively, helping accelerate the bulk loading process. Figure 11(b) shows the bulk load time with different dataset sizes. The bulk load time for PLIN and other indices grows linearly with the dataset size. PLIN is consistently faster than the other competitors, suggesting its high performance in training models and building the index structure.

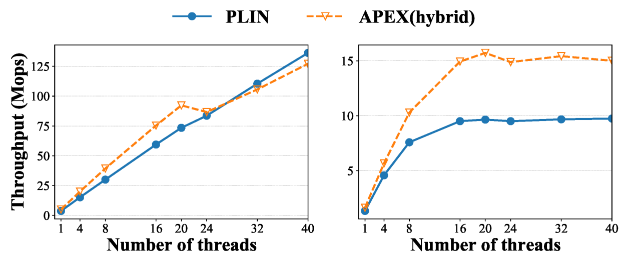
6.5 Comparison with the Original APEX

In this section, we compare PLIN with the original APEX, which is denoted as "APEX(hybrid)".

Read and write performance. This experiment was performed on the Lognormal dataset. As shown in Fig. 12(a), APEX(hybrid) achieves higher read performance when the thread number scales from 1 to 20. However, its performance decreases when the number of threads is increased from 20 to 24. This is due to the impact of hyper-threading. PLIN is only slightly affected by hyper-threading and outperforms APEX(hybrid) in read performance when the number of threads is greater than 32. Figure 12(b) shows that the write throughput of APEX(hybrid) is higher than that of PLIN. This is because that APEX(hybrid) has many accelerating structures residing in DRAM, ensuring that it can have fewer NVM writes than PLIN.

Space efficiency. In this experiment, we build PLIN and APEX(hybrid) on the three datasets and measure their space efficiency. For each dataset, we first load half of the data using the bulk load interface and then insert the other half. Table 2 shows the runtime states of PLIN and APEX(hybrid). We can see that PLIN has a flat structure on all datasets. On both the Normal and Lognormal datasets, PLIN has only hundreds of leaf nodes and is only three layers high. While APEX(hybrid) has tens of thousands of leaf nodes on all three datasets.

Compared with APEX(hybrid), PLIN has higher space efficiency. We divide the space cost into two parts, namely the data fill rate and the index size. The index size is composed of the DRAM size and the PM size. The data fill rate refers to the percentage of filled data-slots in leaf nodes. Table 2 shows that the fill rate of APEX(hybrid) is ~80%, while PLIN has a higher fill rate up to ~90%. This is because the node splits in PLIN are not triggered by the data filling of nodes but are only invoked when the percentage of the overflow data reaches 30%. We observe that the overflow data percentage in PLIN is ~10%. The second part of the space cost is the index size without data slots, including inner nodes and the metadata of leaf nodes. As shown in Table 2, PLIN only takes less than 1 MB PM spaces, while APEX(hybrid) takes over 10 MB PM spaces on average. In addition, PLIN does not use any DRAM space, but APEX(hybrid) consumes up to 700 MB of DRAM space because it maintains many data structures in DRAM.



(a) Read-only workload

(b) Write-only workload

Figure 12: Throughput comparison with APEX(hybrid).

Table 2: Runtime states of PLIN and APEX(hybrid).

Index	Dataset	#inner nodes	#leaf nodes	Fill rate in leaf nodes	Overflow rate in leaf nodes	DRAM size	PM size (without data)
PLIN	Normal	5	589	89.69%	10.30%	0 MB	0.13 MB
	Lognormal	12	776	89.70%	10.29%	0 MB	0.18 MB
	OSM	30	4224	90.81%	9.08%	0 MB	0.92 MB
APEX (hybrid)	Normal	1	23450	80.28%	2.91%	759.66 MB	9.23 MB
	Lognormal	587	22689	80.30%	2.87%	756.71 MB	24.35 MB
	OSM	557	25814	80.63%	2.37%	653.56 MB	11.11 MB

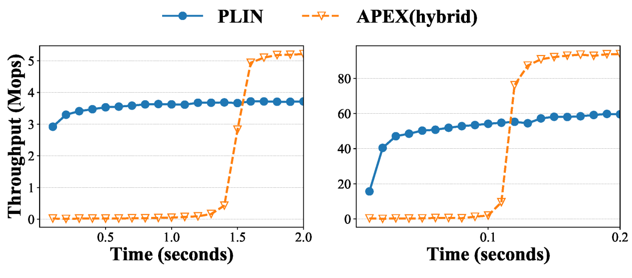


Figure 13: Comparison of the recovery performance.

Recovery. This experiment aims to evaluate the recovery time. First, we use the Lognormal dataset and load 100 million keys. Then, we start another process to recover the index and execute queries simultaneously. Because all the structures of PLIN are placed on persistent memory, PLIN can achieve instant recovery. The experiment shows that PLIN's recovery takes 15.387 ms on average. APEX(hybrid) achieves a similar result (16.146 ms) because it defers the real recovery work to runtime [25]. We also test the recovery time of PLIN in the worst case, i.e., the recovery process is killed during node splits or rebuilding, and the result shows both cases only incur an overhead of ~30 μs.

Figure 13 shows the trend of throughput since the beginning of recovery. APEX(hybrid) exhibits a low throughput at the beginning of recovery because it has to recover the DRAM structures. As a result, APEX(hybrid) takes about 1,700 ms (one thread) or 160 ms (20 threads) to resume its normal throughput, while PLIN can instantly resume a high throughput at the very beginning of the recovery.

6.6 Overhead of Structure-Modified Operations

In this experiment, we study the overhead of structure-modified operations during insertions. We use the OSM dataset and the write-only workload and measure the overhead of splitting a leaf node, rebuilding inner nodes, and writing logs during splitting. The results show that the overhead of splitting a leaf node is 1.40% of the total overhead, the overhead of rebuilding inner nodes is 0.007% of the total overhead, and the overhead of writing logs is 0.0006% of the total overhead and 0.046% of the splitting overhead. The average overhead of each splitting is 793 μs, and the average overhead of each rebuilding is 2.258 ms. Although the overhead of a single structure-modified operation is a bit high, the total overhead of structure-modified operations is low because such operations are not triggered frequently. In addition, the percentage of the writing logs overhead among all overheads is quite low.

6.7 Impact of PLIN Design Choices

In this section, we study the impact of PLIN design choices on read and write performance and latency.

Locally unordered, globally ordered leaf nodes. This experiment aims to reveal the impact of the design of locally unordered and globally ordered leaf nodes in PLIN. Thus, we change the leaf nodes of PLIN to be ordered and keep the other designs unchanged. We find that it causes a dramatic drop in write performance. In particular, in the single-thread environment, the write throughput of PLIN is 0.29 *Mops* (78% drop). In the 20-thread environment, the write throughput of PLIN is 3.68 *Mops* (62% drop). Such performance drops are caused by the additional data movements in the ordered leaf nodes when inserting keys, which introduce many extra CLWB and SFENCE instructions.

Storing a model copy in the parent node. To measure the impact of storing a model copy in the parent node, we remove the model copy mechanism and leave the other designs unchanged. When removing the model copy, we can infer that each operation has to access the node header to obtain the model parameters and node-level locks, which needs additional NVM block accesses, causing the degradation in performance. The experimental result shows that in the single-thread environment, the read throughput of PLIN is 2.29 *Mops* (38% drop) and the write throughput is 1.10 *Mops* (18% drop). In the 20-threaded environment, the read throughput of PLIN is 45.44 *Mops* (37% drop) and the write throughput is 8.28 *Mops* (15% drop). All the results demonstrate the efficiency of the model copy mechanism in PLIN.

Other design choices of structure-modified operations. The performance of structure-modified operations in PLIN are impacted by background threads and orphan nodes, which are used to reduce the tail latency. In this experiment, we disable all background threads and let structure-modified operations be done in the foreground. The result shows that the 99.999% tail latency increases from 25.94 μs to 757.39 μs (29.20 times increase). Next, we disable all orphan nodes, which will cause the frequent rebuilding of inner nodes. In this case, the 99.9999% tail latency increases from 636.97 μs to 2222.32 μs (3.49 times increase). Thus, we conclude that the design of background threads and orphan nodes in PLIN is efficient.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we presented PLIN, a persistent learned index for the NVM-only memory architecture. The main contribution of PLIN is that it lowers the height of NVM-oriented tree indices by incorporating the learned index structure. We proposed several new designs in PLIN, including NVM-aware data placement, locally unordered and globally ordered leaf nodes, storing a model copy in the parent node, and a hierarchical insertion strategy. We also adopted optimistic concurrency control and fine-grained locking mechanisms to improve the scalability of PLIN. The experimental results showed that PLIN achieved higher performance and faster recovery than several NVM-oriented indices and the state-of-the-art persistent learned index APEX. In the future, we will focus on the adaptivity of PLIN to the NVM-only and NVM-DRAM-based hybrid architecture. We notice that both architectures have their own advantages, but neither of them can satisfy various demands. Therefore, a better solution is to find out some adaptive approach to ensure that the index can run on both architectures. In addition, we will devise new techniques to support duplicated keys in PLIN.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation of China (No. 62072419) and Tencent. We also thank the KeeWiDB group in Tencent for their valuable suggestions on improving the paper. Peiquan Jin is the corresponding author.

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. *Proc. VLDB Endow.* 11, 5 (2018), 553–565.

[2] Michael A. Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Trans. Database Syst.* 32, 4 (2007), 26.

[3] Hokeun Cha, Moohyeon Nam, Kibeom Jin, Jiwon Seo, and Beomseok Nam. 2020. B^3-tree: Byte-addressable binary B-tree for persistent memory. *ACM Trans. Storage* 16, 3 (2020), 17:1–17:27.

[4] Leying Chen and Shimin Chen. 2021. How does updatable learned index perform on non-volatile main memory?. In *HardBD@ICDE*. IEEE Computer Society, Chania, Greece, 66–71.

[5] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking database algorithms for phase change memory. In *CIDR*. www.cidrdb.org, Asilomar, CA, USA, 21–31.

[6] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.* 8, 7 (2015), 786–797.

[7] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. μTree: A persistent B+-tree with low tail latency. *Proc. VLDB Endow.* 13, 11 (2020), 2634–2648.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, Indianapolis, Indiana, 143–154.

[9] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *SIGMOD*. ACM, Virtual Event, China, 339–351.

[10] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An updatable adaptive learned index. In *SIGMOD*. ACM, Portland, Oregon, USA, 969–984.

[11] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175.

[12] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A data-aware index structure. In *SIGMOD*. ACM, Amsterdam, The Netherlands, 1189–1206.

[13] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.* 14, 4 (2020), 626–639.

[14] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *FAST*. USENIX Association, Oakland, CA, USA, 187–200.

[15] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *SOSP*. ACM, Virtual Event / Koblenz, Germany, 424–439.

[16] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Trans. Storage* 14, 1 (2018), 5:1–5:17.

[17] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A single-pass learned index. In *aiDM@SIGMOD*. ACM, Portland, Oregon, USA, 5:1–5:5.

[18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. ACM, Houston, TX, USA, 489–504.

[19] Taehyun Kwon, Muhammad Imran, and Joon-Sung Yang. 2021. Reliability Enhanced Heterogeneous Phase Change Memory Architecture for Performance and Energy Efficiency. *IEEE Trans. Computers* 70, 9 (2021), 1388–1400.

[20] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *SOSP*. ACM, Huntsville, ON, Canada, 462–477.

[21] 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.

[22] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*. ACM, San Francisco, CA, USA, 3:1–3:8.

[23] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems. *Proc. VLDB Endow.* 15, 2 (2021), 321–334.

[24] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-trees: Optimizing persistent index performance on 3DXPoint memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.

[25] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A high-performance learned index on persistent memory. *Proc. VLDB Endow.* 15, 3 (2021), 597–610.

[26] Yongping Luo, Peiquan Jin, Qinglin Zhang, and Bin Cheng. 2021. TLBtree: A read/write-optimized tree index for non-volatile memory. In *ICDE*. IEEE Computer Society, Chania, Greece, 1889–1894.

[27] Yongping Luo, Peiquan Jin, Zhou Zhang, Junchen Zhang, Bin Cheng, and Qinglin Zhang. 2021. Two birds with one stone: Boosting both search and write performance for tree indices on persistent memory. *ACM Trans. Embed. Comput. Syst.* 20, 5s (2021), 1–25.

[28] 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, Indianapolis, Indiana, 1–16.

[29] Nooshin Mahdavi, Farhad Razaghian, and Hamed Farbeh. 2022. Data block manipulation for error rate reduction in STT-MRAM based main memory. *J. Supercomput.* 78, 11 (2022), 13342–13372.

[30] Tobias Maltenberger, Till Lehmann, Lawrence Benson, and Tilmann Rabl. 2022. Evaluating In-Memory Hash Joins on Persistent Memory. In *EDBT*. 2:368–2:372.

[31] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[32] 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.

[33] 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*. ACM, San Francisco, CA, USA, 371–386.

[34] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR*. OpenReview.net, Toulon, France.

[35] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A scalable learned index for multicore data storage. In *PPoPP*. ACM, San Diego, California, USA, 308–320.

[36] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*. USENIX Association, San Jose, CA, USA, 61–75.

[37] Stratis Viglas. 2014. Write-limited sorts and joins for persistent memory. *Proc. VLDB Endow.* 7, 5 (2014), 413–424.

[38] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable learned index with precise positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288.

[39] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded Piecewise Linear Representation for online stream approximation. *VLDB J.* 23, 6 (2014), 915–937.

[40] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *FAST*. USENIX Association, Santa Clara, CA, USA, 169–182.

[41] 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.

[42] Zhou Zhang, Peiquan Jin, Xiao-Liang Wang, Yan-Qi Lv, Shouhong Wan, and Xike Xie. 2021. COLIN: A cache-conscious dynamic learned index with high read/write performance. *J. Comput. Sci. Technol.* 36, 4 (2021), 721–740.

[43] Zhiyong Zhang, Zhaoyan Shen, Zhiping Jia, and Zili Shao. 2020. UniBuffer: Optimizing Journaling Overhead With Unified DRAM and NVM Hybrid Buffer Cache. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39, 9 (2020), 1792–1805.

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

[45] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential indexing for persistent memory. *Proc. VLDB Endow.* 13, 4 (2019), 421–434.

[46] Farzaneh Zokaee, Mingzhe Zhang, Xiaochun Ye, Dongrui Fan, and Lei Jiang. 2019. Magma: A Monolithic 3D Vertical Heterogeneous ReRAM-based Main Memory Architecture. In *DAC*. 115.