

APEX: A High-Performance Learned Index on Persistent Memory

Baotong Lu

The Chinese University of Hong Kong
btlu@cse.cuhk.edu.hk

Jialin Ding

Massachusetts Institute of Technology
jialind@mit.edu

Eric Lo

The Chinese University of Hong Kong
ericlo@cse.cuhk.edu.hk

Umar Farooq Minhas

Microsoft Research
ufminhas@microsoft.com

Tianzheng Wang

Simon Fraser University
tzwang@sfu.ca

ABSTRACT

The recently released persistent memory (PM) offers high performance, persistence, and is cheaper than DRAM. This opens up new possibilities for indexes that operate and persist data directly on the memory bus. Recent learned indexes exploit data distribution and have shown great potential for some workloads. However, none support persistence or instant recovery, and existing PM-based indexes typically evolve B+-trees without considering learned indexes.

This paper proposes APEX, a new PM-optimized learned index that offers high performance, persistence, concurrency, and instant recovery. APEX is based on ALEX, a state-of-the-art updatable learned index, to combine and adapt the best of past PM optimizations and learned indexes, allowing it to reduce PM accesses while still exploiting machine learning. Our evaluation on Intel DCPMM shows that APEX can perform up to $15\times$ better than existing PM indexes and can recover from failures in 42ms .

PVLDB Reference Format:

Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas and Tianzheng Wang. APEX: A High-Performance Learned Index on Persistent Memory. PVLDB, 15(3): 597–610, 2022.
doi:10.14778/3494124.3494141

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/baotonglu/apex>.

1 INTRODUCTION

Modern data systems use fast memory-optimized indexes (e.g., B+-trees) [4, 28, 31, 36, 42] for high performance. As data size grows, however, scalability is limited by DRAM’s high cost and low capacity: OLTP indexes alone can occupy $\geq 55\%$ of total memory [58]. Byte-addressable persistent memory (PM) [10, 45, 53] offers persistence, high capacity, and lower cost compared to DRAM. The recently released Intel Optane DCPMM [20] is available in 128–512GB DIMMs, yet 128GB DRAM DIMMs are rare and priced $5\text{--}7\times$ higher than 128GB DCPMM [1]. Although PM is more expensive than SSDs, it offers better performance, making it an attractive option to complement limited/expensive DRAM. These features

Figure 1: Insert scalability (left) and single-thread throughput (right) of ALEX [13] on PM. Naively using PMDK (Naive) limits performance due to PM’s limited bandwidth. Directly running it on PM (Unsafe) further loses crash consistency.

have led to numerous PM-optimized indexes [2, 7, 8, 12, 19, 26, 33–35, 38, 40, 49, 56, 59, 60] that directly persist and operate on PM. Some also support instant recovery to reduce down time.

Most (if not all) existing PM indexes are based on B+-trees or hash tables which are agnostic to data distribution. As demonstrated by recent learned indexes [11, 13, 14, 16, 18, 23, 24, 32, 39, 41, 46, 50, 57], indexes can be implemented as machine learning models that predict the location of target values given a search key. Suppose all values are stored in an array sorted by key order, a linear model trained from the data can directly output the value’s position in the array. If keys are continuous integers (e.g., 0–100 million), the value mapped to key k can be accessed by `array[k]`. Such model-based search gives $\mathcal{O}(1)$ complexity and the entire index is as simple as a linear function. Some learned indexes (e.g., ALEX [13]) also support updates and inserts. They typically use a hierarchy of models [13, 16, 24] that form a tree-like structure to improve accuracy. However, individual nodes could be much bigger (e.g., 16MB in ALEX [13]), leading to very high fanout (e.g., 2^{16}) and low tree depth (e.g., 2), making search operations lightweight even for very large data sizes.

1.1 When Learned Indexing Meets Persistent Memory: The Old Tricks No Longer Work!

We observe learned indexing is a natural fit for PM: Real PM (Optane DCPMM) exhibits $3\text{--}14\times$ lower bandwidth than DRAM [30, 55], whereas model-based search is especially good at reducing memory accesses. But learned indexes were designed based on DRAM without considering PM properties, and prior PM indexes did not leverage machine learning. It remains challenging for learned indexes to work well on PM.

Challenge 1: Scalability and Throughput. Although learned indexes are frugal in bandwidth usage for lookups, they still exhibit excessive PM accesses for inserts. This is because learned indexes

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494141

require data (key-value pairs) be maintained in sorted order, which may require shifting records for inserts. Figure 1 shows its impact by running ALEX [13]—a state-of-the-art learned index—on PM without any optimizations (denoted as Unsafe).¹ Since PM exhibits asymmetric read/write bandwidth with writes being 3–4× lower, frequent record shifting can easily exhaust write bandwidth and eventually limit insert scalability and throughput. This problem was confirmed by recent work [6]. Similar issues were also found in B+-trees. A common solution is to use unsorted nodes [2, 33, 40, 56] that accept inserts in an append-only manner, but require linear search for lookups. This is reasonable for small B+-tree nodes (e.g., 256B–1KB), but for model-based operations to work well, it is critical to use large nodes (e.g., up to 16MB in ALEX) with sorted data. Structural modification operations (SMOs) become more expensive with more PM accesses and higher synchronization cost: typically only one thread can work on a node during an SMO.

Challenge 2: Persistence and Crash Consistency. A key feature of persistent indexes is to ensure correct recovery across restarts and power cycles. Prior learned indexes were mostly based on DRAM and did not consider persistence issues. Simply running a learned index on PM does not guarantee consistency. Any operations that involve writing more than eight bytes could result in inconsistencies as currently only 8-byte PM writes are atomic. Although recent work [21, 27] provides easy ways to convert DRAM indexes to work on PM with crash consistency, they are either not general-purpose, or incur very high overhead. For example, PMDK [21]—the de-facto standard PM library—allows developers to wrap operations in transactions to easily achieve crash consistency. As shown in Figure 1, compared to the Unsafe variant, this approach (Naive) scales poorly with low single-thread throughput because it uses heavyweight logging which incurs write amplification and extra persistence overhead, depleting the scarce PM bandwidth.

1.2 APEX

This paper presents *APEX*, a persistent learned index that retains the benefits of learned indexes while guaranteeing crash consistency on PM and supporting instant recovery and scalable concurrency.

APEX is carefully designed to address the challenges. ❶ We observe a data (leaf) node in a learned index can be regarded as a hash table where a linear model is effectively used as an order-preserving hash function. A collision results when the model predicts the same position for multiple keys. Based on this observation, we develop a collision-resolving mechanism, probe-and-stash, to retain efficient model-based search while avoiding excessive PM writes. APEX also achieves crash consistency for all operations with low overhead. ❷ To reduce synchronization and SMO overheads, APEX adopts variable node sizes with smaller data nodes (256KB) and larger inner nodes (up to 16MB) as SMOs on inner nodes are relatively rare. The former allows lightweight SMOs; the latter allows shallower trees with high fanout. We also design a lightweight concurrency control protocol to reduce synchronization overhead. ❸ Similar to prior work, APEX stores certain frequently used metadata in DRAM to reduce the impact of PM’s higher latency and lower bandwidth. Unlike many other PM indexes, however, APEX does

so while providing instant recovery. The key is to ensure DRAM-resident metadata can be re-constructed quickly and most recovery work can be deferred.

Our evaluation using realistic workloads shows that APEX is up to 15× faster as compared to the state-of-the-art PM indexes [2, 8, 19, 33, 40, 59] while achieving high scalability and instant recovery. We made APEX open-source at <https://github.com/baotonglu/apex>.

We make four contributions. First, APEX brings persistence to learned indexes which is a missing but a necessary feature [25], bringing learned indexing another step closer to practical adoption. Second, APEX combines the best of PM and machine learning (high performance with a small storage footprint). Third, we propose a set of techniques to implement learned indexes on real PM. APEX is based on ALEX, but our techniques (e.g., probe-and-stash and judicious use of DRAM) are general-purpose and applicable to other indexes. Last, we provide a comprehensive evaluation and compare APEX with prior PM indexes to validate our design decisions.

2 BACKGROUND AND RELATED WORK

In this section, we provide background on PM hardware, existing techniques in PM-optimized indexes, and learned indexes.

2.1 Intel Optane DC Persistent Memory

Among various scalable PM types [10, 45, 53], only Intel Optane DCPMM based on 3D XPoint is commercially available; so we target it in this paper. DCPMM can run in *Memory* or *App Direct* [55] modes. The former leverages PM’s high capacity to present bigger but slower volatile memory with DRAM as a hardware-managed cache. The latter allows software to judiciously use DRAM and PM with persistence. We leverage PM’s persistence using the App Direct mode and frugally use DRAM to boost performance. Both DRAM and PM are behind volatile CPU caches, and the CPU may reorder writes to PM. For correctness, software must explicitly issue cacheline flushes (CLWB/CLFLUSH) and fences to force data to the ADR domain [22], which includes a write buffer and a write pending queue with persistence guarantees upon failures [55]. Once in ADR, not necessarily in PM media, data is considered persisted.

Although writing DCPMM media exhibits higher latency than reads, recent work [48, 55] showed that end-to-end read latency is often higher as a write commits once it reaches ADR while PM reads often require fetching data from raw media unless cached. DCPMM exhibits 300ns random read latency, 4× slower than DRAM’s; the end-to-end write latency can be lower than 100ns [55]. DCPMM bandwidth is also lower than DRAM’s. Compared to DRAM, it exhibits 3×/8× lower sequential/random read bandwidth and 11×/14× lower sequential/random write bandwidth. Its write bandwidth is also 3–4× lower than read bandwidth. DCPMM’s internal access granularity is 256 bytes (one “XPLine”) [55]. To serve a 64-byte cacheline read, it internally loads 256B and returns the requested 64B. DCPMM also writes in 256B units. Thus, 256B accesses lead to read and write amplification that wastes bandwidth. For high performance, software should consider PM access in 256B units.

2.2 PM-Optimized Indexing Techniques

Numerous PM-optimized indexes [2, 7, 8, 19, 33–35, 38, 40, 56, 59, 60] have been proposed based on B+-trees and hash tables. They

¹Original ALEX does not support concurrency. On each core we run an ALEX instance that works on a data partition.

mainly optimize for crash consistency and performance. We give an overview of the key techniques proposed by prior PM indexes which APEX adapts for learned indexing in later sections.

Reducing PM Accesses. Many PM B+-trees [2, 7, 33, 40, 56] use unsorted leaf nodes to avoid shifting records upon inserts. A record can be inserted into any free slot in a node; free space is tracked by a bitmap. This reduces PM writes but requires linear scan for point queries. To alleviate such cost, FPTree [40] accompanies each key with a fingerprint (a one-byte hash of the key) to predict if a key possibly exists. Lookups then only access records with matching fingerprints, removing unnecessary PM accesses. Some hash tables [34, 60] use additional stash buckets to handle collisions. This reduces expensive PM accesses in the main table that would otherwise be necessary (e.g., chaining requires more dynamic PM allocations and linear probing may issue many reads). The tradeoff is lookups may need to check stashes in addition to the main table, but this can be largely alleviated using fingerprints for stashes [34].

Instant Recovery. PM’s byte-addressability and persistence allow placing the entire index on the memory bus and recover from failures without much work [2, 7, 19, 34, 60], reducing service down time. Lazy recovery [19, 34] is a well-known technique to realize this. Here we describe a recent approach [34]. The index maintains a global version number in PM and each PM block (e.g., an inner or leaf node) is associated with a local version number l . Upon restart, g is incremented by one, after which the system is ready to serve requests. Individual nodes are only recovered later by the accessing threads if g and l do not match. This way, the “real” recovery work is amortized over runtime, in exchange for instant and bounded recovery time (incrementing one integer).

(Selective) Persistence. To overcome PM’s lower performance, some PM indexes [33, 40, 59] leverage DRAM by placing reconstructable data (e.g., B+-tree inner nodes) in DRAM for fast search. Upon restart, the DRAM-resident data must be reconstructed from data in PM, before the system can start to serve requests. This is doable for B+-trees using bulk loading algorithms. The downside is recovery time scales with data size, sacrificing instant recovery.

Concurrency Control. Both lock-free and lock-based designs have been proposed for PM indexes. Traditionally, lock-free programming has been difficult on PM. Recent work [2, 51] has demonstrated the feasibility of building PM-based lock-free indexes more easily, but the overhead is not negligible [30]. Traditional node-level locking causes exclusive accesses and incur more PM writes when acquiring/releasing read locks. So lock-based designs are often combined with lock-free read and/or hardware transactional memory (HTM) [33, 40] to reduce PM writes. FPTree [40] uses HTM for inner nodes and locking for leaf nodes. HTM performs well under low contention, but is not robust due to issues like spurious aborts [30]. Some proposals [34, 59] use optimistic locking that requires locking for writes, and reads can proceed without holding a lock but must verify the read data is consistent. This is usually done by checking a version number associated with the data item did not change, which if happened, would cause the read operation to be retried.

2.3 Learned Indexes

Learned indexes build machine learning models that predict the position of a given key. For example, one may train a linear model

Figure 2: ALEX structure: in addition to models, inner/leaf (data) nodes store pointers/records in gapped arrays [13].

$y = O \times x + 1$ where O and 1 are parameters learned from prediction accuracy. A learned index may use more complex models (e.g., neural networks), build a hierarchy of simple models, or both. Many learned indexes [11, 13, 14, 16, 18, 23, 24, 32, 39, 41, 46, 50, 54, 57] are based on this idea, but most are read-only. We focus on updatable OLTP learned indexes, but to the best of our knowledge, none support persistence and very few support concurrency [46].

Design Overview. We build APEX on ALEX [13], a fast, updatable learned index. We use ALEX to present our approach to crash consistency and concurrency on PM. Many of our techniques are applicable to other learned indexes; doing so is interesting future work. ALEX uses a hierarchy of simple linear regression models. The structure of ALEX is also called recursive model index (RMI). It uses gapped arrays (i.e., arrays that leave empty slots between records to efficiently absorb future inserts) to store fixed-size keys and payloads sorted by keys. As shown in Figure 2, each inner node stores a linear model with $<$ child pointers ($< = 4$ in Figure 2). Traversal starts from the root node which uses its linear model to predict the next child node (model) to probe, until reaching a leaf (data) node. Each data node stores a linear model and two aligned gapped arrays (GAs), one for keys and one for payloads to reduce search distance and cache misses (Figure 2 shows one for brevity).

Models and Root-to-Leaf Traversals. ALEX uses ordinary least squares linear regression with closed-form formula to train data node models. We find that linear regression models work well in most datasets except one extremely non-linear data set (FB in Section 6). In such highly non-linear cases, complex models can possibly provide higher accuracy (and thus fewer PM accesses) but they also come with higher overhead for training and inference. How to balance model accuracy and the extra cost of complex models is an interesting future direction. Inner node models can partition key space flexibly. For example, in Figure 2 the root node model divides the key space $[0, 1]$ into four equally-sized subspaces, and each subspace is assigned to a child node; all keys in $[0, 1/4]$ are placed in data node A. In other words, inner node models do not *predict* which child node a key falls in, but *guide* how ALEX places keys in child nodes. Thus, model “predictions” in inner nodes during traversal are accurate by construction.

Search. To probe a data node, ALEX uses the stored model to predict a position into the GA. The search succeeds if the predicted position contains the target key. Otherwise, ALEX uses exponential search to find the key. If the keys are uniformly distributed (easy to fit by the model), and the number of keys is smaller than the

maximum data node size, one may use a (large) data node to accommodate all records to drastically reduce inner node size. Otherwise, ALEX recursively partitions the key space to $<$ subspaces until the keys in each subspace can be modeled well by a linear model. As Figure 2 shows, since subspace $[\frac{1}{4}, \frac{1}{2}]$ is non-linear, another node is created hoping that the new subspaces are “linear” enough, while $[\frac{1}{2}, 1]$ is already linear, so ALEX uses one data node for it.

Insert. An insert in data node first uses the model to predict the insert position in gapped array and may employ the exponential search to locate the proper position. Two cases are possible upon insert: (1) insert into the dense region, or (2) insert to a gap. Case (1) requires the elements shifts while case (2) needs to fill all consecutive gaps with the adjacent keys to enable exponential search; Both cases incur excessive PM writes. Such write amplification can easily saturate PM write bandwidth, limit the performance and make efficient crash consistency (without logging) impossible.

SMOs. ALEX defines node density as the fraction of filled GA slots, and further defines lower density 3_l (0.6 by default) and upper density 3_u (0.8 by default). Once the node’s density is $> 3_u$, an SMO is triggered, because insert performance will deteriorate with fewer gaps. An SMO can expand or split a node. An expansion enlarges the node’s GA. So data nodes in ALEX are variable-sized. A split is carried out like in a B+-tree. For example, when data node B in Figure 2 is split, two new nodes are allocated and trained with data partitioned across these two nodes. Then the two rightmost pointers in the root node which originally point to B will respectively point to the two new data nodes. If node A is also split, there is no spare pointer in the root node. ALEX may double the root node’s size or create a new inner node with two child data nodes (split downwards), each contains a split of A. Deletion is simple because it can just leave a new gap. ALEX may perform node contraction and merge to improve space utilization. ALEX uses built-in cost models to make SMO decisions using various statistics. More details about the cost model can be found elsewhere [13]. Both inner and data nodes are variable-sized and can be much larger (e.g., up to 16MB) than nodes in a B+-tree. Using large nodes is important for reducing tree depth, but may significantly slow down SMOs as model retraining takes more time and more data needs to be inserted to the new nodes. On multicore CPUs, this could present a scalability bottleneck as an SMO will block concurrent accesses to the node. Inner node SMO does not require model retraining. As noted earlier, inner node models are only used for space partition and are always accurate so that we only need to scale the model by doing simple multiplication.

3 APEX OVERVIEW

We design APEX with a set of principles distilled from the unique properties of PM and learned indexes:

- **P1 - Avoid Excessive PM Reads and Writes.** A practical PM index must scale well on multicore machines. Given the limited and asymmetric bandwidth of PM, APEX must reduce unnecessary PM accesses and avoid write amplification.
- **P2 - Model-based Operations.** Data-awareness and model-based operations uniquely make search operations efficient. A persistent learned index such as APEX must retain this benefit.

Figure 3: APEX data node layout and DRAM-resident data.

- **P3 - Lightweight SMOs.** Structural modification operations in learned indexes can be heavyweight and eventually limit scalability. APEX should be designed to reduce such overheads.
- **P4 - Judicious Use of DRAM.** APEX can use DRAM for performance, but should use it frugally to reduce cost.
- **P5 - Crash Consistency.** APEX operations must be carefully designed to guarantee correct recovery. Ideally, it should support instant recovery to achieve high availability.

3.1 Design Highlights

APEX combines new and existing techniques based on the above design principles. Similar to ALEX [13], APEX consists of inner nodes and data nodes. APEX places all node contents in PM except a small amount of metadata and accelerators in DRAM to improve performance and reduce PM writes (**P1**, **P4**). APEX employs model-based insert [13] where each data node can be treated as a hash table that uses a model as an order-preserving hash function to predict insert location. To resolve collisions without introducing unnecessary PM accesses, we propose a new probe-and-stash mechanism (Section 4.2) inspired by recent PM hash tables [34] (**P1**, **P2**). We set different maximum node sizes for APEX’s inner and data nodes to ensure most SMOs do not hinder scalability while maintaining a shallow tree (**P3**). For instant recovery, we design DRAM-resident components to be reconstructable on-demand (**P5**).

3.2 Node Structure

Each node in APEX contains a linear model consisting of two double-precision floating point values (slope and intercept) stored in node header, e.g., a, b in the data node in Figure 3(b). Each inner node also contains an array of child pointers. Data nodes also store key-payload pairs as records. Same as other learned indexes [13, 16, 23, 46], APEX stores fixed-length² numeric keys that are at most 8-bytes and 8-byte payloads (either inlined or a pointer). Like previous work [34, 54], we assume unique keys, but non-unique keys can be supported by storing a pointer to a linked list of records as payload.

Data nodes in APEX are variable-sized, but have a fixed *maximum* size which is set to 256KB to fully exploit models. This is larger than the typical size (256 bytes – 1KB) in B+-trees, but small enough to efficiently implement SMOs and achieve good scalability. Since SMOs in inner nodes are relatively rare and exhibit low SMO

²There has been initial work supporting variable-length keys [44, 52]. As future work, we hope to explore how APEX could adopt these techniques.

cost (Section 6), we keep the maximum size of inner nodes to be 16MB. This gives APEX more flexibility to select node fanout, lower tree depth and maintain good search performance. Because of the low tree depth, inner nodes exhibit good CPU cache utilization. Placing them in DRAM does not benefit much. Thus, different from PM-DRAM B+-trees [33, 40], we place inner nodes in PM, which also enables instant recovery (Section 5).

To support model-based lookups and hash-based inserts, each data node consists of (1) a primary array and a stash array (and in case of overflows, extended stash blocks), which store records and are PM-resident, and (2) reconstructable metadata stored in DRAM to accelerate various operations and to support concurrency.

PM-Resident Primary and Stash Arrays. As shown in Figure 3(b), both arrays store data records in record-sized slots. The linear model predicts a position in the primary array (PA) for a given key. To insert a new record k , if the predicted position in the PA is not free, APEX linearly probes the PA and inserts the record into the first free slot. We limit the probing distance to a constant $d = 16$ so that the thread would probe no more than two XPLines (512 bytes). Bounding the probing distance also allows records in PA to be nearly-sorted [5], improving search performance. If no free slot is found, APEX inserts the new record to a free slot in the stash array (SA), which acts as an overflow area. Stashing allows APEX to efficiently resolve collisions without excessive PM writes compared to using ALEX’s gapped array (element shifts) or other common techniques (e.g., probing a large number of slots with excessive PM accesses). We also use accelerators to reduce the overhead of stash accesses. We present more details in Sections 4.1 and 4.2.

Each node has a determined size and number of record slots (computed using the number of keys in the node and lower density), so APEX needs to properly divide the slots between the PA and SA. Allocating more slots to PA can lower collision rate (faster search and inserts), yet there must be enough stash slots in case collisions do happen. Therefore, APEX needs to strike a balance between insert/probe performance and collision handling. APEX leverages data distribution to solve this problem (Sections 4.4 and 4.5).

DRAM-Resident Metadata and Accelerators. APEX places in DRAM certain structures that are (1) easy to reconstruct in case of failures yet are (2) very critical to performance at runtime. As Figure 3(a) shows, we store metadata, locks and accelerators in DRAM, accessible via the pointers stored in the PM data node’s header. Metadata includes basic information about the node, e.g., number of records. As we discuss later, locks do not need to survive power cycles for recovery, and placing them in DRAM can avoid excessive PM accesses. Accelerators are compact data structures to enable fast record access with reduced PM accesses. The key is to use fingerprints [40] to quickly determine if a key possibly exists (often without even reading the whole key). Bitmap indicating slot status is also used for inserts to quickly locate a free PA slot. Finally, to reduce storage overhead we share an accelerator for every 16 records in the PA (pa), e.g., in Figure 3, $pa[0] \dots pa[15]$ share the first accelerator. One accelerator (24-byte) includes 16-byte fingerprints, a 16-bit free-slot bitmap, and one 48-bit pointer³. A stash bitmap is used to indicate the empty slots in the stash array.

4 APEX OPERATIONS

Now we present APEX operations in a single-thread setting with crash-consistency. Section 5 discusses concurrency and recovery.

4.1 Search & Range Query

To search for a key, we start at the root node and use its model to predict which child node to go to, until we reach a leaf node. There is no search within inner nodes as by construction the model’s prediction is always accurate. All operations that require a traversal share this logic; for now we focus on data node operations. Within a given data node, we devise a probe-and-stash mechanism to reduce unnecessary PM accesses during key lookups. As Figure 3 shows, all memory accesses by a lookup are highlighted in red (blue) for PM (DRAM). We first “probe” using the node’s model to predict the position of a key k in the PA. Suppose the predicted position is 3 in Figure 3(b). APEX directly returns the record if the key is found in the slot. Otherwise, it linearly probes from $pa[3]$ till $pa[3 + d - 1] = pa[18]$, where d is the probing distance (16) described in Section 3.2. APEX returns the record if k is found; otherwise it continues to check the SA. Note that for all operations linear probing always proceeds in the same direction (conceptually from “left to right”) because APEX does not shift data during inserts and so a key cannot be stored in a slot before its predicted position, simplifying concurrency control (details later).

To accelerate the lookup in stashes, APEX creates an overflow bucket in DRAM for every 16 PA records if a key is originally predicted within the 16 records but overflowed to SA or extended stash; the overflow bucket’s address is stored in the DRAM accelerator. We find the value of 16 balances memory consumption and performance: having an overflow bucket per record needs a 48-bit pointer (next to the bitmap in Figure 3(a) in the accelerator) to point to it, adding non-trivial overhead ($d/37\%$). Using one overflow bucket per 16 records amortizes this cost. Continuing with the example, before accessing the stash, APEX first checks the corresponding overflow bucket, which holds up to 15 pointers for indexing overflow keys in the stash. A new overflow bucket is dynamically allocated and linked if there are more than 15 overflow keys.

Each pointer to a stash record in the overflow bucket inlines a fingerprint of that stash record in the most significant byte of the pointer. Only keys in the stash with matching fingerprints are accessed from PM. A negative search will issue no PM reads on the stash, but up to two XPLine accesses in the PA. For a positive search, fingerprints in the overflow buckets pinpoint the target stash entries, reducing the expected number of PM reads to one [40].

For a range query where l/r are the predicted positions of the start/end keys in a data node, APEX first collects the records between $pa[l]$ and $pa[r + 15]$ and finds the remaining ones from stash. Indexes that use unsorted nodes [2, 33], often require sorting full nodes, which adds overhead and requires using smaller nodes to alleviate. APEX only needs to sort the final result set. This is efficient as APEX maintains nearly-sorted order, sorting which is faster [5]. In Section 6 we show its impact using realistic datasets.

4.2 Insert

To insert a record with key k , APEX ensures k is not already in the index, and if so, locates and inserts the record to a free slot.

³Modern x86 processors use the least significant 48 bits for addressing [22]

Uniqueness Check. After obtaining a predicted position p , instead of directly probing PA, we first check key existence using the fingerprints in the accelerator. This can potentially save PM accesses: a negative result indicates k is definitely not in PA. Fingerprints are usually co-located in the same cacheline with the 16-bit bitmap which must be brought to the CPU cache to find a free slot for PA insertion, accessing fingerprints incurs little overhead and is practically “free” without additional memory accesses. In Figure 3, assume the model predicted position 3 for the new key, APEX first checks existence using the 4th to 16th fingerprints from the first accelerator, and the first three fingerprints from the second accelerator. We access PA only if there is a matching fingerprint. Note that lookup (Section 4.1) does not use PA’s fingerprint because first accessing the accelerator can incur extra cache misses if the key is stored in PA (which is the common case). The uniqueness check then continues with the fingerprints in the overflow bucket(s) and (if needed) stash slots in PM, same as a regular key search.

Locating a Free Slot. PM indexes often use bitmaps to indicate free space [7, 33, 40, 59], but persisting them for each insert/delete on PM adds non-trivial overhead. Thus, APEX includes the bitmaps in the DRAM accelerators that are rebuilt on-demand by reading slot contents. We indicate free slots by storing in them an invalid key that is out of the node’s key range $[<B, <OG]$. Then we must not place $<B = C64$ and $<OG = C64$ in the same node. This is done by ensuring the initialization/bulk loading algorithm always generates at least two data nodes, one with range $[<B = C64, <OG]$ and the other with range $[<B, <OG = C64]$. Then we use $<OG + 1$ and $<B - 1$ as the invalid key in the two nodes, respectively. As a result, both insert and delete only require one PM write (updating the record) and one DRAM write (flipping a bitmap entry).

SA is statically allocated during node creation. When it is full (although rare), APEX dynamically allocates a new 256-byte extended stash in PM and atomically stores a pointer to the newly allocated block in PM. Extended stash blocks are linked together and reachable via a pointer in the data node. Crash consistency is guaranteed by the PM allocator [21] which ensures safe PM ownership transfer between the allocator and PM to avoid permanently leaking PM.

Crash-Consistent Insert. As described in Section 2, ALEX [13] uses GA with exponential search, which incurs excessive PM writes by shifting records or filling consecutive gaps with adjacent keys. To save PM bandwidth, APEX neither shifts records nor fills gaps. This is possible since APEX uses linear probing instead of exponential search with two careful designs: (1) APEX co-locates the key and payload, so an insert requires only a 16-byte PM write. (2) APEX writes the payload *prior* to writing the key and persists them in PM using one flush and fence, leveraging the fact that modern x86 CPUs do not reorder writes to the same cacheline [43]. In case of a crash, APEX simply discards records with invalid keys. Thus, unless a new extended stash block is needed, an insertion only writes one XPLine. We alleviate the impact of extended stash by carefully setting SA size based on data distribution (Sections 4.4–4.5).

4.3 Delete & Update

APEX implements delete as lookup followed by invalidation. Once we locate the record, APEX simply replaces the target key in the slot with an invalid key. The validity bitmap in DRAM is also updated

to reflect this change. APEX updates records in place. If the key is found, we atomically update and flush the payload with one XPLine write. Same as inserts, the lookup process in deletes and updates uses PA’s fingerprints to reduce PM accesses.

4.4 Structural Modification Operations

Similar to ALEX, APEX uses node density to decide when to trigger an SMO. ALEX uses 0.8 as the upper density limit 3_D because insert performance degrades beyond that. To achieve an average memory utilization of 70% (same as a B+-tree), ALEX uses 0.6 as the lower density limit 3_L . In APEX, however, such a tight bound would trigger many SMOs, incur excessive PM writes (e.g., moving data to a new node for node splits) and hurt performance. Since inserts in APEX incur little write amplification, APEX can tolerate a higher upper density limit to reduce SMOs. Based on empirical evaluation, we use 0.5–0.9 with the same 70% average memory utilization.

Node Expansion vs. Split. Once an SMO is triggered, we use cost models to choose between node *expansion* and *split* like ALEX does. APEX follows the same model as ALEX’s but uses different statistics. To quantify the cost of a search, APEX uses the average number of cache misses in probe-and-stash instead of ALEX’s average number of iterations of exponential search. Insert cost is the average number of overflow buckets allocated plus the search cost.

Data Node Expansion. APEX expands a node in three steps: (1) allocate and initialize a new node; (2) retrain or re-scale the model and insert records from old to using the new model; (3) attach to the parent node, update the sibling pointers, and reclaim old.

This multi-step process needs to be implemented carefully. For example, there will be a PM leak if a crash happens before step 3. Also, the index would be inconsistent if a crash happens during step 3. APEX achieves lightweight crash-consistency via *hybrid logging*. We make a key observation: only step 2 is relatively long running. Hence, different from prior work which only uses redo-logging to possibly redo a *whole* SMO, APEX uses undo logging before step 3 and redo-logging after step 2. If a crash happens after step 2, APEX would waste less work and can resume step 3 upon restart.

A naïve logging approach, such as PMDK’s physical undo logging logs all data records and so incurs excessive PM writes. In APEX, node expansions have only three steps, so we can use logical logging with a small log area in PM. Upon step 1, we initialize a “node-expand” log entry (one cacheline) in PM with the format of $>3 = 4F : 4 - BC064$, where >3 is a pointer to the old node, $: 4 -$ is the insert that triggered the node expansion (for locating the parent node of the old node), and $BC064$ is set to UNDO. Next, we allocate a new node using PMDK which atomically stores the new node’s address in $=4F$, and initialize the new node (setting all keys in the PA/SA to invalid). Step 2 then updates and persists the $BC064$ of the node-expand log entry to REDO. Now we switch from undo logging to redo logging and start step 3. Finally, we persist the node-expand log entry in the PM with a $BC064$ value reset to NoSMO.

This approach gives low overhead (only three PM log writes) and fast recovery. If the system fails before step 3, we discard the new node to undo the incomplete expansion. If it fails after step 2, the SMO can resume from step 3 by observing $BC064$ in the log.

Data Node Split and Inner Node Expansion. APEX uses the same data node split and inner node expansion logic as ALEX

(Section 2). APEX also handles these SMOs in a very similar way to node expansions explained above: Each SMO has its own log entry in PM. We use logical undo-redo logging to ensure once a heavyweight step (e.g., record copying) is done, APEX would only redo the lightweight step (e.g., switching pointers) upon recovery.

Stash Ratio. The ratio between the sizes of the stash and primary array is governed by a *stash ratio* (ζ), defined as the fraction of stash array size to the sum of the primary array and stash array sizes. Setting a reasonable stash ratio is needed to avoid excessive collisions or the overhead of extra stash block allocations. Previous PM hash tables [34, 60] allocate a fixed-size stash array based on a predefined collision probability. This is not ideal for learned indexes since the collision probability of the model depends on how well the model fits the data. APEX automatically configures the stash ratio when creating a new data node, based on the overflow ratio of the old node. Specifically, the overflow ratio $\$$ of a data node with $\#_3$ keys is the fraction of the number of overflowed keys $\#_>$ in that data node, i.e., $\$ = \frac{\#_>}{\#_3}$. These simple statistics are all part of the metadata in DRAM and APEX maintains a set of them per 256 records to amortize the cost. APEX assumes that the expanded or split nodes follow the same distribution from the old node. Hence, the stash ratio of new node created by an SMO is set to be the overflow ratio. This strategy ensures that the stash ratio of a data node is adaptive to the actual data distribution.

4.5 Bulk Loading

Like ALEX, during bulk loading APEX grows the RMI greedily, but uses different cost models described in Section 4.4 and must also determine PA and SA sizes. Ideally the stash ratio (ζ) should match the percentage of records overflowed to SA during real inserts to reduce extended stash use and balance insert (which prefers larger SA) and lookup (which prefers smaller SA) speeds. This requires knowing data distribution which is unavailable upon bulk loading.

We estimate a reasonable ζ empirically. Based on extensive experiments using realistic datasets (details in Section 6), we find setting ζ within the range [0.05, 0.3] well balances model-based search and insert performance. APEX thus bounds ζ in this range. Then, we set ζ (via simulation: Given $\#_3$ keys to insert to a node, we first assume the node will reach the upper density limit 3_D and all free slots are allocated to PA. We then compute the predicted positions in PA for each key and probe-and-stash to collect the number of keys $\#_>$ overflowed to SA without actually carrying out any inserts (thus a “simulation”). With the overflow ratio $\$ = \frac{\#_>}{\#_3}$, we calculate ζ based on two intuitions: (1) the higher the overflow ratio $\$$, the bigger the stash ratio (ζ); (2) ζ should be greater than $\$$ as $\$$ was determined by assuming all slots are in PA (i.e., real inserts should exhibit more collisions than simulation). There could be many ways to determine ζ using $\$$. For simplicity, we set ζ to be a multiple ($=$) of $\$$, i.e., $\zeta = \times \$$, and empirically determined \times ’s value to be 1.5 via experiments (not shown here for space limitation); we call \times the “stash coefficient.” In general, a higher stash coefficient means potentially more keys are stored in the stash areas. Taking the aforementioned bound into account, $\zeta = \min(0.05, \times \$)$.

Overall, our method gives reasonable performance and is simple to implement/calculate. The upper limit ensures most records stay in PA; the lower limit is a safety net to absorb collisions (e.g., due

to a distribution shift) before an SMO reorganizes the node. In practice, we do not expect stashing to be the main storage as APEX recursively partitions the key space so that each subspace can be modeled well. Note that bulk loading still succeeds even if ζ is inaccurate: more keys will be stored in SA and/or the extended stash. Our evaluation in Section 6 shows that in practice stash ratio is low in most cases and extended stash blocks are rarely used as stash ratio is reasonably set based on data distribution. Further optimizations are interesting future work.

5 CONCURRENCY AND RECOVERY

As Section 2.2 describes, compared to traditional node-level locking and lock-free approaches, optimistic locking is usually a better fit for PM and balances programmability and performance. APEX further adapts optimistic locking for learned indexes on PM.

Inner Node Accesses. APEX uses different maximum sizes for inner and data nodes (16MB vs. 256KB). Inner nodes typically have less contention so we pick a larger node size. Each inner node carries a reader-writer lock for SMO, compared to traditional optimistic locking with mutual exclusion locks [29]. Reading an inner node (e.g., lookup) is lock-free. In traditional optimistic locking, the thread retries traversal if inconsistencies caused by modifications on the node are detected, wasting CPU cycles. APEX avoids such aborts by an out-of-place-based SMO design, described later.

Data Node Accesses. Data nodes may see many concurrent accesses, so using a smaller node size can help reduce contention. In addition to a node-level lock to ensure only one thread can conduct SMO on the node, we allocate one optimistic lock per 256 records in PA to isolate non-SMO updates. This design balances the synchronization and lock acquisition overhead during SMOs.

To read a data node, the thread keeps traversing down until reaching the target data node, without holding any locks. However, upon reading data records, it uses the version in the optimistic lock to guarantee the read correctness and restarts the search if the version changed. Like many prior approaches, we use epoch-based memory reclamation [17] for safe memory management.

To insert a key, the thread first traverses to the target data node using lock-free read. To find the key in the node, the thread may need to use linear probing to access multiple slots, which requires acquiring the corresponding lock(s) that cover(s) the probing slots. More than one lock may be acquired if the predicted position plus probing distance crosses lock boundary. Unlike ALEX, since in APEX all threads linearly probe in the same direction (described in Section 4.1), it is guaranteed that deadlocks will not happen. To update a data node, the thread first acquires the lock that protects the record, and then continues to hold it if it needs to update the stash array. Multiple threads can race to install new records into the stash array while holding different locks (i.e., in two different 256-record blocks). Therefore, threads must first allocate a free stash slot using the stash bitmap, which is done by using the compare-and-swap (CAS) instructions to atomically set the “next free” bit in the bitmap (and retry if the CAS failed); after that each thread can continue to insert the record to its own stash array slot.

SMOs. Data node expansion and split require more care to work under optimistic locking. Expanding a data node is done in an out-of-place manner that always allocates a new node and updates

the parent node to point to the new node. Meanwhile, the parent node may be undergoing an expansion. The aforementioned reader-writer lock in inner nodes is for handling such cases. Upon updating the inner node, the thread γ takes the node's lock in shared (reader) mode. Note that since γ already holds the lock for the data node, it is safe to directly use an atomic write to update the pointer in the parent node. This allows multiple threads to proceed and expand different data nodes in parallel. If a data node split causes the parent node to expand, the inserter thread locks the inner node in exclusive (writer) mode. In theory, it is possible for splits to propagate up and grow the RMI by acquiring locks bottom-up. This can incur non-trivial overhead [13]. Our implementation therefore follows prior work [13] to disallow inner node split and only allow expansion. This limits the number of acquired locks to three (data node, parent and grandparent levels). Note that throughout this process, readers proceed without taking any locks but must verify version numbers. Using out-of-place SMO and updates without shifting in inner nodes allow the traversals to data nodes without retries. The thread may see an obsolete data node due to concurrent SMOs (although the key range is correct). It detects this case by checking data nodes's lock status and retries from root if it is set.

Instant Recovery. APEX adopts lazy recovery in Section 2.2. It needs to undo in-flight SMOs (if any) by deallocating PM blocks and switching pointers (Section 4.4); both are lightweight and after that APEX can start to handle requests. Since each thread has at most one in-flight SMO upon crash, recovery time scales with thread count, instead of data size. Modern OLTP systems usually limit thread count, making APEX recovery practically instant.

6 EVALUATION

We now present a comprehensive evaluation of APEX including comparisons against the state-of-the-art PM indexes. We show that:

- APEX retains the benefits of model-based search and achieves high throughput and good scalability.
- APEX's individual design principles and choices are effective, collectively allowing APEX to perform and scale well.
- APEX instant recovers (γ 's), although it uses DRAM, in contrast to prior work that trades off instant recovery for performance.

6.1 Index Implementations

We implemented APEX in C++ and compare it with recent PM B+-trees: BzTree [2], LB+Tree [33], FAST+FAIR [19], DPTree [59], FPTree [40] and uTree [8]. BzTree and FAST+FAIR are PM-only indexes and do not use DRAM. LB+Tree and FPTree are hybrid PM-DRAM indexes that place inner/leaf nodes in DRAM/PM. They combine HTM and locking for synchronization. uTree puts both inner and leaf nodes in DRAM and relies on a linked list of records in PM for persistence. DPTree batches modifications in DRAM buffer and merges with the background PM-DRAM tree to reduce persistence overhead. Except BzTree and FPTree (which were not originally open-sourced), we use the original authors' open-sourced code and add the necessary but missing functionality in best effort.⁴

Persistence. We use PMDK [21] to support persistence in all indexes and verify they do not incur unnecessary overheads. We

modified LB+Tree, FAST+FAIR, uTree and DPTree to use PMDK because they either were proposed based on DRAM emulation or did not implement certain necessary PM-related functionality due to the lack of a full-fledged PM allocator.⁵ We also fixed LB+Tree to provide correct read committed isolation level like other indexes.⁶

Operations. We did best-effort implementations of the missing range scan in FAST+FAIR, LB+Tree and uTree. We faithfully implemented recovery for LB+Tree and uTree. For multi-threaded recovery, LB+Tree requires statistics [33] that are currently not being collected. We therefore implemented a single-threaded version.

6.2 Experimental Setup

We run experiments on a server with a 24-core (48-hyperthread), 2.1GHz Intel Xeon Gold 6252 CPU, 768GB Optane DCPMM (6 × 128GB DIMMs on all six channels) and 192GB DRAM (6 × 32GB DIMMs). The CPU has 35.75MB of L3 cache. The server runs Arch Linux (kernel 5.10.11). We use PMDK/jemalloc [15] to allocate PM/DRAM. All code is compiled with GCC 10.2 with all optimizations. For fair comparison, we set each index to use the parameters used in its original paper. LB+Tree/FAST+FAIR/uTree/BzTree use 256B/512B/512B/1KB node. FPTree uses 28/64-record inner/leaf nodes. APEX uses maximum 16MB/256KB inner/data nodes.

Datasets. We use six synthetic and realistic datasets to test all the indexes. Longitudes is extracted from Open Street Maps (OSM) [3]. Longlat is also from OSM but is transformed to become highly non-linear to stress learned indexes. Lognormal represents the lognormal distribution. YCSB contains user IDs in YCSB [9]. SOSD [37] includes four realistic datasets. Due to space limits, we focus on the Facebook (FB) dataset containing randomly sampled Facebook user IDs. FB is extremely non-linear and the hardest-to-test among SOSD datasets. We use it to stress test the indexes. We also run the TPC-E [47] benchmark and collect three datasets (trade, settlement, cash transaction) by loading the database with 15000 customers and 300 initial trading days. APEX performs similarly under them, so we only report results from the trade dataset.

All keys are unique in these datasets. Same as previous work [13] we randomly shuffle them to simulate a uniform distribution over time. All the datasets use 8-byte keys and 8-byte payloads. Except Longitudes and Longlat whose key type is double, all the other datasets consist of 8-byte integer keys. Lognormal contains 190 million keys (2.83GB), whereas TPC-E (trade) contains 259 million keys (3.86GB); other datasets include 200 million keys (2.98GB).

Benchmarks. We stress test each index using microbenchmarks. For all runs, we bulk load the index with 100 million records, and then test individual operations. Since only LB+Tree supports node merge (when the node is empty) which may reduce its performance, we run 90 million deletes to avoid triggering merges for fair comparison. Other workloads issue 100 million requests. Range scans start at a random key and scan 100 records. Lognormal only has 190 million keys, so for its insert test we issue 90 million requests. The source code of DPTree does not support double key type (used in Longlat and Longitudes datasets), recovery and delete operation, so we do not include it in the corresponding experiments.

⁵For example, uTree was designed to self-manage PM space, but the open-sourced code does not implement recycling. So we use the PMDK allocator.

⁶Details at <https://github.com/schencoding/lbtree/pull/6>.

⁴Code for all indexes is summarized in our repo: <https://github.com/baotonglu/apex>.

Figure 4: Single-thread throughput. APEX performs the best on live datasets for inserts and range scans, and remains competitive for the worst-case FB. For search, update and delete APEX performs the best across all cases.

Table 1: APEX statistics after bulk loading.

Metric	Longi tudes	Longi lat	Lognormal	FB	YCSB	TPC-E
Average depth	1.10	1.64	1.95	3.13	2	3.43
Maximum depth	2	3	3	6	2	9
Number of inner nodes	541	3628	374	6279	8193	20879
Number of data nodes	17438	44071	12696	81856	16384	143035
Minimum inner node size	16B	16B	16B	16B	16B	16B
Median inner node size	16B	32B	64B	32B	16B	16B
Maximum inner node size	512KB	4MB	16MB	512KB	64KB	4KB
Minimum data node size	496B	496B	496B	496B	131KB	1056B
Median data node size	139.5KB	29.05KB	168.75KB	21.5KB	136KB	2.56KB
Maximum data node size	256KB	256KB	256KB	256KB	142KB	149KB
Average stash ratio	0.05	0.118	0.05	0.299	0.05	0.06
Average overflow ratio	0.006	0.065	0.0008	0.48	0.0014	0.034

6.3 Single-thread Performance

We first run single-thread experiments to compare the indexes without contention. Note that we do not remove concurrency support; with one thread the overhead is minimal. To better understand the results, we also show the basic statistics of APEX after bulk loading in Table 1. For inserts, as shown in Figure 4(a), APEX outperforms BzTree/LB+Tree/FAST+FAIR/DPTree/uTree/FPTree by up to $15\times/2.7\times/3.8\times/1.66\times/6.8\times/3.7\times$, for live datasets. The advantage mainly comes from APEX’s model-based insert and probe-and-stash. In most cases, APEX only issues one PM write per insert, whereas other indexes often issue more (e.g., $j=10$ for BzTree [2]). For FB, APEX is $1.18\times/5.16\times/2.3\times/1.26\times$ faster than FAST+FAIR/BzTree/uTree/FPTree, but achieves a lower (5.77%/42%) throughput when compared to LB+Tree/DPTree. As Table 1 shows, FB exhibits the highest average overflow ratio. Thus, FB incurs many collisions in PA with more inserts routed to SA and extended stash. This requires more CPU cycles to find a free slot and allocate overflow buckets. Note that FB is the hardest-to-t (the worst case). Overall, APEX remains competitive under worst-case scenarios and outperforms other indexes by up to $15\times$ in common cases.

Figure 4(b) shows the result for search operations. APEX performs up to $7.1\times/3.9\times/4.1\times/3.2\times/3.2\times/5.8\times$ higher than BzTree/LB+Tree/FAST+FAIR/DPTree/uTree/FPTree. Notably, APEX’s throughput is 60%/37% higher than that of LB+Tree and DPTree under the hardest-to-t FB dataset. The performance differences across all datasets are due to how well APEX could t the data. For datasets that are easy to t by linear models, APEX exhibits much higher search throughput, for two reasons: (1) APEX’s index layer is much smaller and shallow, resulting in much better CPU cache efficiency. For example, in Table 1, the average tree depth on Longi tudes is 1.1. (2) For easy-to-t data, most records are stored in PA instead of

SA (e.g., overflow ratio is 0.006 on Longi tudes), so most lookups only need model-based search without probing stashes. For hard-to-t data, APEX tree depth can be higher (e.g., 3.13 for FB), reducing traversal efficiency. The probing distance and overflow ratio are also higher, adding more overhead during key lookup.

Search performance also affects update/delete. APEX only issues one PM write per update/delete, and consistently outperforms other indexes in Figures 4(c)–(d). Thanks to the nearly-sorted order using probe-and-stash, APEX scans up to $7.6\times/1.45\times/1.46\times/1.83\times/16.38\times/3.1\times$ faster than BzTree/LB+Tree/FAST+FAIR/DPTree/uTree/FPTree on live datasets in Figure 4(e). FB presents the worst case for APEX, due to long overflow bucket lengths. Scans incur more cache misses to traverse the overflow buckets. FAST-FAIR does not need sorting. LB+Tree’s sorting overhead is very small for its small data nodes (256B). BzTree often has to sort much larger (1KB) leaf nodes, adding much more overhead and so performs $3\times$ lower than APEX even on FB. uTree performs the worst among all indexes since traversing the PM linked list incurs lots of cache misses.

6.4 Scalability with Number of Threads

Now we examine how each index scales with an increasing number of threads under various operations and workloads.

Individual Operations. For inserts (Figure 5), APEX scales better than others before hyperthreading on live datasets and is competitive under FB. In most cases, APEX incurs one XPLine write per insert and its adaptive node sizes help reduce synchronization overhead. In contrast, FAST+FAIR needs to shift existing records while uTree needs to frequently allocate PM records. BzTree needs to update much metadata used by itself [30]. LB+Tree scales well with help of DRAM and small 256B nodes, but with lower throughput in most cases. Note that APEX also scales under FB, although with lower numbers vs. other workloads. This shows APEX’s concurrency control protocol is lightweight. APEX benefits from hyperthreading, but not as significantly as LB+Tree since APEX has much higher single-thread throughput, exhausting PM bandwidth earlier with fewer threads. LB+Tree uses hyperthreading to better use PM bandwidth, whereas APEX use less resource (threads) to fully exploit PM bandwidth and maintain high performance.

APEX scales nearly linearly for lookups before hyperthreading (Figure 6). It consistently outperforms other indexes for all datasets benefiting from efficient model-based search which better utilizes CPU caches. We observed similar trends for update, delete and scan (not shown for limited space). For updates and deletes, APEX only requires one PM write, while other indexes may require more.

Figure 5: Insert scalability. APEX and LB+Tree scale better than others over all datasets. LB+Tree performs better than APEX on FB, but is limited by its lower single-thread throughput for other datasets.

Figure 6: Search scalability. APEX scales nearly linearly before hyperthreading, as model-based search leverages CPU caches more effectively than data-agnostic designs where tree traversal incurs many more cache misses.

Table 2: SMO statistics of insert-only workloads (24 threads).

	Longitudes	Longlat	Lognormal	FB	YCSB	TPC-E
Inner node expansions	105	2057	296	5	1	2169
Data node expansions	15786	35277	3003	73377	16383	139621
Data node splits (sideway)	1653	5327	9499	440	1	6075
Data node splits (downwards)	0	0	193	0	0	0
Average inner node SMO time (ms)	0.11	0.09	0.66	0.94	0.01	0.03
Average data node SMO time (ms)	1.74	0.78	2.1	0.49	1.54	0.21
Lock % during data node SMO	0.03%	0.03%	0.03%	0.03%	0.02%	0.04%

SMO Costs. Table 2 lists SMO statistics under insert-only workloads. Many more SMOs happen in data nodes than in inner nodes where SMOs are lightweight as shown by the average SMO times. This justifies our adaptive node size design. Although we use more locks per node, the overhead is very small (0.02%-0.04% time of an SMO) relative to other SMO work (e.g., node allocations).

We further stress test APEX and LB+Tree on SMO-intensive workloads. We bulk load APEX/LB+Tree with 100 million records with upper density limit 0.9/1.0 (SMO-intensive) and lower density limits 0.5/0.5 (normal) and then run 10 million inserts. In Figure 7, compared to the SMO-intensive cases, APEX and LB+Tree perform $5.0\times/2.1\times$ better under their “normal” cases. Under 24 threads, LB+Tree outperforms APEX by $1.3\times$ with intensive SMOs as APEX SMO needs to do more work, e.g., model retraining and making decisions based on the cost-models. We believe APEX’s much higher improvement in the common cases outweighs such degradation on corner cases; we leave more optimizations as future work.

Skewed Workloads. We evaluate search and update operations with 1 and 24 threads under varying skewness. As shown in Figures 8(a–b), with a single thread, with higher skewness (θ), all indexes perform better because the accesses are focused on a smaller set of hot keys, better utilizing the CPU cache and less impacted by PM’s high latency. The improvement for updates is

Figure 7: Insert with SMO-intensive vs. normal cases.

smaller because they have to push records to PM. Under 24 threads in Figures 8(c–d), the indexes maintain their relative merits to each other and achieve higher search throughput because of reduced PM accesses. However, under very high skew ($\theta=0.99$), contention increases and synchronization becomes the major bottleneck.

6.5 Effect of Individual APEX Design Choices

We quantify the impact of APEX’s design choices, including node size, probing distance, stash ratio, density bound and accelerators.

Maximum Node Size. We start with the impact of maximum node sizes using the easy-to-t Lognormal as it generates larger nodes close to size limits. We first set the maximum inner node size as 16MB and vary maximum data node sizes. In Figure 9(a), APEX with 16MB maximum data node size achieves lower performance compared to using maximum 256KB nodes due to higher SMO costs. However, using very small data nodes (e.g., 4KB) also leads to lower performance since more inner nodes are needed to index the data, increasing tree depth in Figure 9(b). This in turn causes more cache misses during traversal, e.g., compared to using maximum 16MB nodes, using 4KB maximum nodes increases the average tree depth from 1.54 to 3.38 with $2.8\times$ lower search throughput. Both insert and search performance peak with maximum 256KB data

Figure 8: Throughput under varying skewness of Zipfian distribution with one (a–b) and 24 (c–d) threads (Longlat).

Figure 9: Impact of APEX node sizes (24 threads, Lognormal).

nodes (APEX’s default). Then we fix the maximum data node size as 256KB and vary inner node maximum sizes. In Figure 9(d), although tree depth increases with smaller inner nodes, search performance is barely affected. The reason is the inner nodes can all fit in the CPU cache, so a deeper tree only needs more computation without much data movement. Since SMOs on inner nodes are rare, we also observe little impact on insert performance in Figure 9(c). Thus, we set 16MB as the default maximum inner node size to lower tree depth and maintain good search performance.

PA Probing Distance. We study how different bounded (maximum) probing distances impact performance in Figure 10. For easy-to-fit Longlat, increasing δ barely impacts search since the model fits well. But for hard-to-fit Longlat, increasing δ from 8 to 128 lowers search performance by 26.5%. Although a larger δ reduces SA accesses, the average PA probing distance increases (e.g., from 3.6 to 6.5 when δ increases from 8 to 128) as collisions are more common in hard-to-fit datasets, pushing records far away from the predicted position. Using a larger δ also mandates more probing in PA before accessing SA. As Figure 10(b) shows, insert performance grows by 8% when δ grows from 8 to 32 which more efficiently resolves collisions in PA, thus reducing SA and extended stash accesses. However, a large δ like 128 reduces insert performance due to the higher cost of uniqueness check in PA. A higher

Figure 10: Impact of PA probing distances (24 threads).

Figure 11: APEX 24-thread throughput under different stash ratios; da indicates APEX’s distribution-aware approach.

also reduces PA’s sortedness, lowering scan performance. APEX therefore uses $\delta = 16$ to balance search/scan/insert performance.

Stash Ratio. APEX sets SA size based on data distribution and bounds the stash ratio in the $[0.05, 0.3]$ range. Now we explore more options by directly setting the stash ratio between 0 and 0.9. As Figures 11(a)–(b) shows, search performance drops by 66%/59% on Longlat/Longlat when stash ratio grows from 0 to 0.9. Recall that a larger SA leads to a smaller PA, so a higher stash ratio leads to more collisions in the smaller PA, routing more lookups in SA/extended stash. However, insert performance improves by 8%/11% on Longlat/Longlat when stash ratio increases from 0 to 0.2 in Figures 11(c)–(d) as a relatively larger SA can absorb more inserts and reduce PM allocation costs for extended stash. When stash ratio is ≥ 0.3 , insert becomes slower as PA collision overhead cancels out SA’s collision-resolving benefits. Our distribution-aware approach (da) gives nearly the best performance for both inserts and lookups. As Table 1 shows, the average stash ratios set by da are 0.05/0.118 on Longlat/Longlat. In general, hard-to-fit data like Longlat should have a higher stash ratio to efficiently absorb inserts. Easy-to-fit data like Longlat can use a lower stash ratio to retain the efficiency of model-based search on PA. Very low stash ratios will degrade insert performance while high stash ratios (> 0.3) will negatively impact both search and insert performance; this leads to our decision to bound the stash ratio in range of $[0.05, 0.3]$.

Effect of Accelerators, DRAM and Density Bound. With the recommended parameters fixed, now we explore how each other design choices affect APEX’s performance by conducting a factor analysis on one hard-to-fit dataset: Longlat. Our results on hardest-to-fit FB (not shown for space limitation) has a similar trend but

Figure 12: Factor analysis for APEX under 24 threads. Features are added from left to right and are cumulative.

larger improvement ratio than Longlat. We start from the a baseline version that comes with no accelerators, stores the whole tree in PM, and use the tight density bound [0.6,0.8] as ALEX does. We then add additional APEX features and observe throughput under 24 threads. Figure 12 shows the results. APEX with overflow buckets outperforms the baseline version with linear search in SA by 1.09 \times by reducing the cost of stash accesses. The improvement for inserts is 2.78 \times , which is more significant because the overflow bucket effectively accelerates uniqueness check by avoiding scanning the whole SA. PA fingerprints also accelerates uniqueness check during inserts and traversals during update/delete. This further improves insert performance by 5%. Note that the accelerators added above are still kept in PM, therefore leading to limited improvement.

After placing the accelerators and metadata in DRAM, APEX’s search performance increases by 1.76 \times because of the reduced PM accesses. The increase for inserts is 2.21 \times . Note that the amount of DRAM-resident data used by APEX is small in most cases. For example, APEX consumes 0.23/2.14GB of DRAM/PM after loading 100 million records from Longlatudes. It only consumes more DRAM in FB (0.68/2.29GB of DRAM/PM) since more DRAM-resident overflow buckets are created for stash. This shows that APEX can leverage PM’s high capacity and potentially reduce total system cost, by requiring less or even no DRAM if the user desires. FPTree may consume less DRAM [40] than APEX while APEX always has less DRAM consumption than uTree and is competitive with LB+Tree. Finally, using a loose density bound ([0.5,0.9]) further improves performance by 1.21 \times , because the loose bound only incurs half of SMOs than using the tight bound, thus issuing less PM writes.

6.6 Recovery

We now evaluate how quickly the indexes recover. We (1) load a certain number of records, (2) kill the process to emulate a crash and (3) measure the time needed for the index to start accepting requests. Table 3 shows the recovery times for each index on Longlat. As expected, the recovery time of LB+Tree, uTree and FPTree scales with data size as their in-DRAM inner nodes need to be rebuilt. The main difference between them lies in the leaf-level traversal speed, which is determined by leaf level layout. uTree exhibits the longest recovery time as its leaf level is organized in a linked list with one record per node, traversing which incurs many more cache misses than LB+Tree’s 256B leaf nodes. FPTree recovers faster than uTree and LB+Tree as its leaf node size is even bigger (1KB) incurring fewer cache misses. The other indexes achieve instant recovery (<1s). At restart, APEX needs to redo/undo in-flight SMOs; such

Table 3: Recovery time (s). APEX can recover instantly with a short warm-up time under 1/24 thread(s) (in parenthesis).

#Keys	APEX	LB+Tree	FAST+FAIR	BzTree	uTree	FPTree
50M	0.042 (1.94/0.18)	3.62	0.042	0.098	21.49	1.63
100M	0.041 (3.74/0.26)	7.20	0.041	0.109	43.27	3.26
150M	0.042 (5.24/0.32)	10.77	0.041	0.097	65.032	4.90

Figure 13: APEX’s throughput over time upon restart.

cases are very rare. BzTree uses PMwCAS to transparently recover by scanning a constant-size descriptor pool [51]. Both APEX and FAST+FAIR instantly recover with lazy recovery.

Since APEX defers the “real” recovery work to runtime, we evaluate how much warm-up time APEX needs before its throughput peaks. We load 50 million records from Longlat and then kill the process during an insert workload. After recovery, we issue lookups and observe throughput. In Figure 13, red arrows indicate when APEX is ready to accept requests. APEX initially achieves relatively low throughput: 0.01-0.03 Mops/s with one thread and 0.1-0.4Mops/s with 24 threads. It takes 1.9s/0.15s for throughput peaks with 1/24 thread(s). Using more threads helps as they can recover different data nodes in parallel. The warm-up time scales with data size, but as Table 3 shows, it is still faster than uTree and LB+Tree and close to FPTree; using 24 threads further reduces it to \sim 1s.

7 CONCLUSION

PM offers high performance, cheap persistence and possibility of instant recovery. Prior work either does not exploit the advantages of learned indexes or PM. Yet naively porting a learned index to PM results in low performance. In this paper, we distill several general design principles for adapting the best of PM and learned indexes. We apply those principles to the design and implementation of APEX, a concurrent and persistent learned index with instant recovery. Our in-depth evaluation on Intel Optane DCPMM shows that APEX achieves up to 15 \times higher throughput compared to recent PM-based indexes, and can instantly recover in \sim 42ms.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. We also thank Weiran Huang who helped with figure plotting. This work is partially supported by an NSERC Discovery Grant, a Canada Foundation for Innovation John R. Evans Leaders Fund, Hong Kong General Research Fund (14200817), Hong Kong AoE/P-404/18, Innovation and Technology Fund (ITS/310/18, ITP/047/19LP) and Centre for Perceptual and Interactive Intelligence (CPII) Limited under the Innovation and Technology Fund, MIT Data Systems and AI Lab (DSAIL), NSF IIS 1900933.

REFERENCES

- [1] Paul Alcorn. 2019. Intel Optane DIMM Pricing: \$695 for 128GB, \$2595 for 256GB, \$7816 for 512GB (Update). [h ps://www.tomshardware.com/news/intel-optane-dimm-pricing-performance-39007.html](https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance-39007.html), last accessed on 13/11/2021.
- [2] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-ikje Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB* 11, 5 (2018), 553–565.
- [3] AWS. 2021. OpenStreetMap on AWS. [h ps://registry.opendata.aws/osm](https://registry.opendata.aws/osm), last accessed on 13/11/2021.
- [4] Robert Binna, Eva Zangerle, Martin Pichl, Gtzather Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) (*SIGMOD '18*). Association for Computing Machinery, New York, NY, USA, 521–534.
- [5] Badrish Chandramouli and Jonathan Goldstein. 2014. Patience is a Virtue: Revisiting Merge and Sort on Modern Processors. In *SIGMOD*. 731–742.
- [6] Leying Chen and Shimin Chen. 2021. How Does Updatable Learned Index Perform on Non-Volatile Main Memory?. In *37th IEEE International Conference on Data Engineering Workshops, ICDE Workshops*. IEEE.
- [7] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797.
- [8] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a Persistent B+-Tree with Low Tail Latency. *PVLDB* 13, 11 (2020), 2634–2648.
- [9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB (*SoCC '10*). Association for Computing Machinery, New York, NY, USA, 143–154. [h ps://doi.org/10.1145/1807128.1807152](https://doi.org/10.1145/1807128.1807152), last accessed on 13/11/2021.
- [10] Rob Crooke and Mark Durcan. 2015. A Revolutionary Breakthrough in Memory Technology. *3D XPoint Launch Keynote* (2015).
- [11] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Bzlam, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 407–410. [h ps://doi.org/10.5441/002/edbt.2020.44](https://doi.org/10.5441/002/edbt.2020.44), last accessed on 13/11/2021.
- [12] Biplob Debnath, Alireza Haghdost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. 2015. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, INFLOW 2015, Monterey, California, USA, October 4, 2015*. 1:1–1:9.
- [13] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yanan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (*SIGMOD '20*). Association for Computing Machinery, New York, NY, USA, 969–984. [h ps://doi.org/10.1145/3318464.3389711](https://doi.org/10.1145/3318464.3389711), last accessed on 13/11/2021.
- [14] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *PVLDB* 14, 2 (2020), 74–86.
- [15] Jason Evans. 2006. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of the BSDCan Conference*.
- [16] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *PVLDB* 13, 8 (2020), 1162–1175.
- [17] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge, UK.
- [18] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) (*SIGMOD '19*). Association for Computing Machinery, New York, NY, USA, 1189–1206. [h ps://doi.org/10.1145/3299869.3319860](https://doi.org/10.1145/3299869.3319860), last accessed on 13/11/2021.
- [19] 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.
- [20] Intel. 2021. Intel Optane Persistent Memory (PMem). [h ps://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-dc-persistent-memory.html](https://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-dc-persistent-memory.html), last accessed on 13/11/2021.
- [21] Intel. 2021. Persistent Memory Development Kit. (2021). [h ps://pmem.io/pmdk/](https://pmem.io/pmdk/), last accessed on 13/11/2021.
- [22] Intel Corporation. 2021. Intel 64 and IA-32 Architectures Software Developer's Manual. (2021). [h ps://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html](https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html), last accessed on 13/11/2021.
- [23] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned
- [24] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) (*SIGMOD '18*). Association for Computing Machinery, New York, NY, USA, 489–504. [h ps://doi.org/10.1145/3183713.3196909](https://doi.org/10.1145/3183713.3196909), last accessed on 13/11/2021.
- [25] Tim Kraska, Umar Farooq Minhas, Thomas Neumann, Olga Papaemmanouil, Jignesh M. Patel, Chris Rizk, and Michael Stonebraker. 2021. ML-In-Databases: Assessment and Prognosis. *IEEE Data Engineering Bulletin* 44, 1 (2021), 3.
- [26] Se Kwon Lee, K. Hyun Lim, Hyunsu Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 257–270.
- [27] 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* (Huntsville, Ontario, Canada) (*SOSP '19*). Association for Computing Machinery, New York, NY, USA, 462–477. [h ps://doi.org/10.1145/3341301.3359635](https://doi.org/10.1145/3341301.3359635), last accessed on 13/11/2021.
- [28] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, USA, 38–49.
- [29] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. Article 3, 8 pages.
- [30] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *PVLDB* 13, 4 (2019), 574–587.
- [31] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. 302–313.
- [32] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (*SIGMOD '20*). Association for Computing Machinery, New York, NY, USA, 2119–2133. [h ps://doi.org/10.1145/3318464.3389703](https://doi.org/10.1145/3318464.3389703), last accessed on 13/11/2021.
- [33] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *PVLDB* 13, 7 (2020), 1078–1090.
- [34] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *PVLDB* 13, 8 (2020), 1147–1161.
- [35] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 1–16.
- [36] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.
- [37] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *PVLDB* 14, 1 (2020), 1–13.
- [38] Moohyeon Nam, Hokeun Cha, Youngri 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.
- [39] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (*SIGMOD '20*). Association for Computing Machinery, New York, NY, USA, 985–1000. [h ps://doi.org/10.1145/3318464.3380579](https://doi.org/10.1145/3318464.3380579), last accessed on 13/11/2021.
- [40] 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*

- [45] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80–83.
- [46] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A Scalable Learned Index for Multi-core Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) (PPoPP '20). Association for Computing Machinery, New York, NY, USA, 308–320. <https://doi.org/10.1145/3332466.3374547>, last accessed on 13/11/2021.
- [47] Transaction Processing Performance Council (TPC). 2015. TPC Benchmark E Standard Specification, revision 1.14.0. <http://www.tpc.org/tpce>, last accessed on 13/11/2021.
- [48] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019*. 12:1–12:7.
- [49] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, Vol. 11. USENIX Association, 61–75.
- [50] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. 569–574.
- [51] Tianzheng Wang, Justin Levandoski, and Per-ikke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 461–472.
- [52] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. [n.d.]. SIndex: a scalable learned index for string keys. In *APSys '20: 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, Tsukuba, Japan, August 24-25, 2020. 17–24.
- [53] H. S. P. Wong, S. Raoux, Sangbum Kim, Jiale Liang, John P. Reifenberg, B. Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase Change Memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.
- [54] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. [arXiv:2104.05520 \[cs.DB\]](https://arxiv.org/abs/2104.05520)
- [55] 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 *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, February 24-27*.
- [56] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 167–181.
- [57] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-ikke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 193–208. <https://doi.org/10.1145/3318464.3389770>, last accessed on 13/11/2021.
- [58] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) (SIGMOD '16). ACM, New York, NY, USA, 1567–1581. <https://doi.acm.org/10.1145/2882903.2915222>, last accessed on 13/11/2021.
- [59] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *PVLDB* 13, 4 (2019), 421–434.
- [60] 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.