APEX: A High-Performance Learned Index on Persistent Memory (Extended Version)

Baotong Lu  
The Chinese University of Hong Kong  
bblu@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 $\sim 15\times$ better than existing PM indexes and can recover from failures in $\sim 42$ms.

1 INTRODUCTION
Modern data systems use fast memory-optimized indexes (e.g., B+-trees) \[4, 30, 33, 38, 44\] for high performance. As data size grows, however, scalability is limited by DRAM’s high cost and low capacity: OLTP indexes alone can occupy $> 55\%$ of total memory \[60\]. Byte-addressable persistent memory (PM) \[10, 47, 55\] offers persistence, high capacity, and lower cost compared to DRAM. The recently released Intel Optane DCPMM \[21\] is available in 128–512GB DIMMs, yet 128GB DRAM DIMMs are rare and priced $\sim 5–7\times$ 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 have led to numerous PM-optimized indexes \[2, 7, 8, 12, 20, 28, 35–37, 40, 42, 51, 58, 61, 62\] 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 \[11, 14, 15, 17, 19, 25, 26, 34, 41, 43, 48, 52, 59\], 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 $O(1)$ complexity and the entire index is as simple as a linear function. Some learned indexes (e.g., ALEX \[14\]) also support updates and inserts. They typically use a hierarchy of models \[14, 17, 26\] that form a tree-like structure to improve accuracy. However, individual nodes could be much bigger (e.g., 16MB in ALEX \[14\]), leading to very high fanout (e.g., $2^{15}$) 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 $\sim 3–14\times$ lower bandwidth than DRAM \[32, 57\], 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 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 \[14\]—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 $3–4\times$ lower,}

\(^1\)Original ALEX does not support concurrency. On each core we run an ALEX instance that works on a data partition.
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, 35, 42, 58] 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 [22, 29] 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, PNIDK [22]—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. 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. 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. 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, 20, 35, 42, 58, 61] 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 [27], 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, 47, 55], 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 [57] 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 [23], which includes a write buffer and a write pending queue with persistence guarantees upon failures [57]. Once in ADR, not necessarily in PM media, data is considered persisted.

Although writing DCPMM media exhibits higher latency than reads, recent work [50, 57] 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 [57]. 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”) [57]. 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, 20, 35–37, 40, 42, 58, 61, 62] have been proposed based on B+-trees and hash tables. They 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, 35, 42, 58] 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, FP-Tree [42] 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 [36, 62] 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 [36].

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, 20, 36, 62], reducing service down time. Lazy recovery [20, 36] is a well-known technique to realize this. Here we describe a recent approach [36]. The index maintains a global version number $G$ 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 [35, 42, 61] 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, 53] has demonstrated the feasibility of building PM-based lock-free indexes more easily, but the overhead is not negligible [32]. 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) [35, 42] to reduce PM writes. FP-Tree [42] 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 [32]. Some proposals [36, 61] 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 \( \text{pos} = [a \times \text{key} + b] \) where $a$ and $b$ 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, 14, 15, 17, 19, 25, 26, 34, 41, 43, 48, 52, 56, 59] 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 [48].

Figure 2: ALEX structure: in addition to models, inner/leaf (data) nodes store pointers/records in gapped arrays [14].

Design Overview. We build APEX on ALEX [14], 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 $m$ child pointers ($m = 4$ 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 $m$ subspaces until the keys in each subspace can be modeled well by a linear model. As Figure 2 shows, since subspace \( [1/2, 1] \) is non-linear, another node is created hoping that the new subspaces are "linear" enough, while \( [1/2, 1] \) is already linear, so ALEX uses one data node for it.
**Insert.** An insert in a data node first uses the model to predict the insert position in a 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 \( d_L (0.6 \text{ by default}) \) and upper density \( d_u (0.8 \text{ by default}) \). Once the node’s density is \( d > d_u \), 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 [14]. 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.

- **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 [14], 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 [14] 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 [36] (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 [14, 17, 25, 48], APEX stores fixed-length numeric keys that are at most 8-bytes and 8-byte payloads (either inline or a pointer). Like previous work [36, 56], 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 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 [35, 42], we place inner nodes in PM, which also enables instant recovery (Section 5).

To support model-based insert operations, each data node consists of (1) a primary array and a stash array (and in

---

**Figure 3:** APEX data node layout and DRAM-resident stash data.
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 \((K, P)\), 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 [5]). In Section 6 we show its impact using realistic datasets. 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 (>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 XPLines 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 [42]. For a range query where \(i\) is the predicted positions of the start/end keys in a data node, APEX first collects the records between \(PA[i]\) and \(PA[j + 15]\), and finds the remaining ones from stash. Indexes that use unsorted nodes [2, 35], 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 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 (>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.

### 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.

**Uniqueness Check.** After obtaining a predicted position \(k\), 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
We alleviate the impact of extended stash by carefully setting SA with an invalid key. The validity bitmap in DRAM is also updated when we locate the record. APEX simply replaces the target key in the slot contents. We indicate free slots by storing in them an invalid key that is out of the node’s key range \([\text{min, max}]\). Then we must not place \(\text{min \text{Int64}}\) and \(\text{max \text{Int64}}\) in the same node. This is done by ensuring the initialization/bulk loading algorithm always generates at least two data nodes, one with range \([\text{min \text{Int64}}, \text{max}]\), and the other with range \([\text{min}, \text{max \text{Int64}}]\). Then we use \(\text{max} + 1\) and \(\text{min} - 1\) 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 \([22]\) which ensures safe PM ownership transfer between the allocator and PM to avoid permanently leaking PM.

**Locating a Free Slot.** PM indexes often use bitmaps to indicate free space \([7, 35, 42, 61]\), 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 \([\text{min, max}]\). To prevent the index from becoming unmanageable, APEX triggers a node split whenever the number of cached free slots drops below a threshold.

**Crash-Consistent Insert.** As described in Section 2, ALEX \([14]\) 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, and if the key is found, it atomically updates the payload with one XPLLine write. Same as inserts, the lookup process in deletes and updates uses PA’s fingerprints to reduce PM accesses.

**Delete & Update**

APEX implements deletes 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 XPLLine write. Same as inserts, the lookup process in deletes and updates uses PA’s fingerprints to reduce PM accesses.

**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 \(d_l\) because insert performance degrades beyond that. To achieve an average memory utilization of \(~70\%\) (same as a B+-tree), APEX uses 0.6 as the lower density limit \(d_u\). 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 \(A\) in three steps: (1) allocate and initialize a new node \(B\); (2) retrain or re-scale the model and insert records from \(A\) to \(B\) using the new model; (3) attach \(B\) to the parent node, update the sibling pointers, and reclaim \(A\).

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 \(⟨\text{old, new, key, stage}⟩\), where \(\text{old}\) is a pointer to the old node, \(\text{key}\) is the insert that triggered the node expansion (for locating the parent node of the old node), and \(\text{stage}\) is set to UNDO. Next, we allocate a new node using PMDK which atomically stores the new node’s address in \(\text{new}\) and initialize the new node (setting all keys in the PA/SA to invalid). Step 2 then updates and persists the \(\text{stage}\) 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 \(\text{stage}\) value reset to NO SMO.

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 \(\text{stage}\) 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 \(S\), 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 [36, 62] 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 \( O \) of a data node with \( N_d \) keys is the fraction of the number of overflowed keys \( N_o \) in that data node, i.e., \( O = \frac{N_o}{N_d} \). 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 \( S \) 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 \( S \) empirically. Based on extensive experiments using realistic datasets (details in Section 6), we find setting \( S \) within the range \([0.05, 0.3]\) well balances model-based search and insert performance. APEX thus bounds \( S \) in this range. Then, we set \( S \) via simulation: Given \( N_d \) keys to insert to a node, we first assume the node will reach the upper density limit \( d_u \) 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 \( N_o \) overflowed to SA without actually carrying out any inserts (thus a “simulation”). With the overflow ratio \( O = \frac{N_o}{N_d} \), we calculate \( S \) based on two intuitions: (1) the higher the overflow ratio \( O \), the bigger the stash ratio \( S \); (2) \( S \) should be greater than \( O \) as \( O \) 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 \( S \) using \( O \). For simplicity, we set \( S \) to be a multiple \((n)\) of \( O \), i.e., \( S = n \times O \), and empirically determined \( n \)’s value to be 1.5 via experiments (not shown here for space limitation); we call \( n \) 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, \( S = \max(0.05, \min(0.3, 1.5 \times O)) \).

Overall, our method gives reasonable performance and is simple to implement/calulate. 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 \( S \) 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 [31]. 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 [18] 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 \( T \) takes the node’s lock in shared (reader) mode. Note that since \( T \) 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
We implemented APEX in C++ and compare it with recent PM indexes. 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 (<1s), although it uses DRAM, in contrast to prior work that trades off instant recovery for performance.

6 Evaluation

We implemented APEX in C++ and compare it with recent PM indexes: BzTree [2], LB+Tree [35], FAST+FAIR [20], DPTree [61], FPTree [42] 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 [22] 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 not 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 [35] 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 [16] 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 log-normal distribution. YCSB contains user IDs in YCSB [9]. SOSD [39] 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-fit among SOSD datasets. We use it to stress test the indexes. We also run the TPC-E [49] 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 [14] we randomly shuffle them to simulate a uniform distribution over time. All the datasets use 8-byte keys and 8-byte payloads. Except Longitudetudes 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 Longitudetudes datasets), recovery and delete operation, so we do not include it in the corresponding experiments.

6.3 Single-thread Performance

We first run single-thread experiments to compare the indexes without contentation. 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×/2.7×/3.8×/1.66×/6.8×/3.7×, for five 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

---

4Details at https://github.com/schencoding/lbtree/pull/6.
5Details at https://github.com/baotonglu/apex.
6Code for all indexes is summarized in our repo: https://github.com/baotonglu/apex.

For example, uTree was designed to self-manage PM space, but the open-sourced code does not implement recycling. So we use the PMDK allocator.
Table 1: APEX statistics after bulk loading.

| Metric                     | Longitudes | Longlat | 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   | 21296     | 81556| 16384| 145353|
| Minimum inner node size    | 16B        | 16B     | 16B       | 16B  | 16B  | 16B   |
| Median inner node size     | 16B        | 32B     | 64B       | 16B  | 16B  | 16B   |
| Maximum data node size     | 512KB      | 4MB     | 16MB      | 512KB| 64KB | 4KB   |
| Minimum data node size     | 496B       | 496B    | 496B      | 113KB| 1056B|
| Median data node size      | 139.5KB    | 29.03KB | 168.3KB   | 21.5KB| 138KB| 256KB |
| Maximum data node size     | 256KB      | 256KB   | 256KB     | 256KB| 143KB| 149KB |
| Average stash ratio        | 0.05       | 0.118   | 0.05      | 0.142| 0.06 | 0.06  |
| Average overflow ratio     | 0.006      | 0.065   | 0.0008    | 0.048| 0.0014| 0.064 |

Figure 4: Single-thread throughput. APEX performs the best on five 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.

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 five 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 [32]. 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. Figure 7–9 shows that update, erase and scan scale well as they all benefit from efficient model-based search. The one PM-write design in update and delete operation especially make APEX scale better than other indexes because other indexes require more PM writes. Since we reduce most unnecessary PM reads and writes in those operations, they could further leverage the hyperthreading to exhaust the remaining bandwidth.

Mixed Operations. Real-world applications usually involve a mix of operations. We evaluate each index with two mixed workloads: (1) a read-heavy workload that consists of 20% inserts and 80% search, and (2) a balanced workload with 50% inserts and 50% search. Both workload issue 100/90 million inserts from Longlat/Lognormal, leading to a total of 500/450 million and 200/180 million operations, respectively. In Figure 10, we show that APEX scales the best, and
the other indexes show similar trends as before. As the workload becomes more write-dominant, the throughput of all indexes drops. But APEX is still up to 1.69× better than the next best performing index, LB+Tree.

**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 11, compared to the SMO-intensive cases, APEX and LB+Tree perform 5.0×/2.1× better under their “normal” cases. Under 24 threads, LB+Tree outperforms APEX by 1.3× 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 12(a–b), with a single thread, with higher skewness (theta), 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 smaller because they have to flush records to PM. Under 24 threads in Figures 12(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-fit 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 13(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 13(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× lower search throughput. Both insert and search performance peak with maximum 256KB data nodes (APEX’s default). Then we fix the maximum node data size as 256KB and vary inner node maximum sizes. In Figure 13(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 13(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 \( D \) impact performance in Figure 14. For easy-to-fit Longlat, increasing \( D \) barely impacts search since the model fits well. But for hard-to-fit Longlat, increasing \( D \) from 8 to 128 lowers search performance by 26.5%. Although a larger \( D \) reduces SA accesses, the average PA probing distance increases (e.g., from 3.6 to 6.5 when \( D \) 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 \( D \) also mandates more probing in PA before accessing SA. As Figure 14(b) shows, insert performance grows by 8% when \( D \) grows from 8 to 32 which more efficiently resolves collisions in PA, thus reducing SA and extended stash accesses. However, a large \( D \) like 128 reduces insert performance due to the higher cost of uniqueness check in PA. A higher \( D \) also reduces PA’s sortedness, lowering scan performance. APEX therefore uses \( D = 16 \) 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

**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 larger improvement ratio than Longlat. We start from the 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 16 shows the results. APEX with overflow buckets outperforms the baseline version with linear search in SA by 1.09× by reducing the cost of stash accesses. The improvement for inserts is 2.78×, 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× because of the reduced PM
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 then kill the process during an insert workload. After recovery, we issue lookups and observe throughput. In Figure 17, 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 1.96/0.15s 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 FP Tree; using 24 threads further reduces it to < 1s.

### 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 FP Tree 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. FP Tree 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 cases are very rare. BzTree uses PMwCAS to transparently recover by scanning a constant-size descriptor pool [53]. 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 17, 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 1.96/0.15s 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 FP Tree; using 24 threads further reduces it to < 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× higher throughput compared to recent PM-based indexes, and can instantly recover in ∼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). [dimm-pricing-performance,39007.html, last accessed on 13/11/2021].
[2] Joy Arulraj, Justin J. Levandowski, Umar Farooq Minhas, and Per-Åke Larsson. 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. [https://registry.opendata.aws/osm, last accessed on 13/11/2021].
[4] Robert Bunna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HQT: 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 Chandramowoli and Jonathan Goldstein. 2014. Patience is a Virtue: Revisiting Merge and Sort on Modern Processors. In SIGMOD. 731–742.
[6] Leping Chen and Shimin Chen. 2021. How Does Updatable Learned Index Perform on Non-Volatile Main Memory?. In 57th IEEE International Conference on Data Engineering Workshops, ICDE Workshops. IEEE.

---

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 |
|-------|------|---------|-----------|--------|-------|--------|
| 5M    | 0.042 (1.94/0.18) | 3.62 | 0.042 | 0.098 | 21.49 | 1.63 |
| 10M   | 0.041 (3.74/0.26) | 7.29 | 0.041 | 0.109 | 43.27 | 3.26 |
| 15M   | 0.042 (5.24/0.32) | 10.77 | 0.041 | 0.097 | 65.03 | 4.90 |

---

Figure 16: Factor analysis for APEX under 24 threads. Features are added from left to right and are cumulative.

Figure 17: APEX’s throughput over time upon restart.
Figure 18: Search throughput under 24 threads with different payload sizes on Lognormal and Longlat.

APPENDIX

A COMPLEXITY ANALYSIS

Now we analyze the complexity of APEX operations. Let \( n \) and \( m \) be the maximum number of slots in inner and data nodes (with PA and SA), respectively. Although data nodes may allocate slots in the extended stash, the maximum number of records that can be stored in a data node is fixed \((m \times d_u)\) where \(d_u\) is the upper bound density since an SMO will start if the number of records would exceed \(m \times d_u\). Following ALEX’s proof [14], traversing from root to leaf takes \(\log_2 p\) time, where \(p\) is the minimum number of partitions such that when the key space \(s\) is divided into \(p\) partitions of equal width, every partition contains no more than \(m \times d_u\) keys. We omit the proof details here and interested readers could refer to [14].

In the best case, inserting or searching a data node takes \(O(1)\) time if the model fits well. If the predicted PA position for insert is close to a free slot within the bounded probing distance, the record could be inserted in \(O(1)\) complexity; Future search operation may also directly hit the target record with linear probing in \(O(1)\) time. If the collisions cannot be addressed in the PA, a linear search in the SA may be required to find a free slot, or an extended stash will be allocated. The worst-case cost for insert is bounded by \(O(m)\) since the extended stash space could not be infinitely allocated given the SMO condition introduced earlier. Searching a data node may also need accessing the stash area by linearly traversing the overflow buckets, which again has a bounded cost of \(O(m)\).

B IMPACT OF PAYLOAD SIZE

In this appendix, we study the impact of payload size on the lookup performance of the evaluated trees. Using an 8-byte pointer as the payload in index leaf (data) nodes is common in main-memory indexes [13]. The open-source implementations of other trees under comparison [2, 8, 20, 35, 42, 61] also all assume their payloads are 8-byte pointers. Therefore, for all indexes more cache misses are likely to happen when the external payload is accessed.

To explore how such cache misses impact each index, we conducted an experiment to show how indexes perform when the payload is externally stored with varying sizes (from 8-byte to 128-byte). We also include the performance when the payload is an inlined 8-byte value for reference (named 8B(in) in Figure 18). As shown in Figure 18, compared to using 8-byte inline payloads, APEX’s search throughput drops by 44%/32% on Lognormal and Longlat respectively when accessing external payloads. Beyond
16-byte payloads, the performance in general drops with bigger payload sizes. Our profiling results show that the number of cache misses per search operation of APEX increases from 3.36 to 4.51 on Lognormal (i.e., 34% more cache misses) when the 8-byte payload is changed from inline-stored to external-stored. The numbers for FPtree increases from 9.41 to 10.39 (i.e., 10% more cache misses), i.e., its performance degradation is not as obvious as APEX’s. However, even in the condition of external payload accesses (from 8-byte to 128-byte), APEX’s search performance still achieves much higher throughput due to the efficient model-based search.

C IMPACT OF STASH COEFFICIENT
As stated in Section 4.5, we set stash ratio $S$ to be a multiple ($n$) of overflow ratio $O$, i.e., $S = \max(0.05, \min(0.3, n\times O))$ and empirically determined stash coefficient $n$’s value via experiments. We show the experimental results in Figure 19. As the figure shows, setting $n$ as 1.5 achieves both good search and insert performance. In general, a larger stash coefficient means potentially more keys are stored in the stash areas. When varying $n$ from 1 to 5, the average stash ratio increases from 0.09 to 0.22 (i.e., 22% of the data node’s space is allocated to the stash array). As the figure shows, the impact is more pronounced for search operations: a larger $n$ (e.g., 3 and 4) means more search operations are routed into stash array but no insert performance improvement. We therefore chose 1.5 as the default stash coefficient and overall this gives reasonable performance and is simple to implement/calculate. We hope to explore ways to improve on space allocation between primary/stash arrays (e.g., fine-tuning the stash coefficient) to further optimize performance in future work.

D COMPARISON WITH FULL-DRAM APEX
We study the impact of PM by comparing with the full-DRAM APEX. Specifically, we ran APEX on DRAM by pointing PMDK to use DRAM, same as in prior work [32] (Lersch et al.). As Figure 20 shows, the DRAM version performs $2.4 \times / 2.3 \times$ faster on Longitudes and Longlat for search operations. The numbers for inserts are $3.3 \times / 2.7 \times$. APEX on DRAM also achieves near-linear scalability, thanks to DRAM’s ~4× lower latency and ~3-14× higher bandwidth than PM’s. Also, DRAM’s adequate bandwidth allows APEX to benefit from hyperthreading which helps hide CPU stalls. Since future PM is expected to have higher bandwidth (e.g., the recent Optane 200 Series already delivers 32% more bandwidth on average [24]), this experiment also give hints how APEX might perform on newer devices.

E MEMORY (PM/DRAM) CONSUMPTION
For PM/DRAM consumption, we measure the memory consumption of all indexes after bulk loading 100 million records on different datasets (~1.5GB of raw data), shown in Table 4. Except for APEX, the memory consumption of other indexes does not vary a lot on different data sets so we only report their memory consumption on Lognormal. All indexes require extra space than raw data sizes because they reserve empty slots to absorb future inserts. Compared to APEX, the other indexes require more PM space because they either store extra metadata in PM (e.g., fingerprints in FPtree) or put the whole tree in PM (e.g., BzTree and FAST+FAIR). APEX consumes more DRAM than FPtree and DPtree but its DRAM consumption is always smaller than uTree (up to 13× smaller) which duplicates the whole index in DRAM. In most datasets, APEX has a smaller or similar DRAM consumption as LB+Tree. The only exception is FB (0.676GB DRAM of APEX vs. 0.265GB of LB+Tree), where APEX has to allocate more DRAM-resident overflow buckets for stash due to FB’s extremely non-linear distribution.

F BULK-LOADING TIME
In this section, we study the cost of initial bulk loading. We conducted an experiment to show the time for bulk loading 100 million records on various data sets. Among the other indexes under comparison, only LB+Tree provides a fast bulk loading algorithm while others all require inserting keys from scratch. For fair comparison,
Table 4: PM and DRAM space consumption (GB). Except for APEX, other indexes only present the memory consumption on Lognormal data set.

|                  | APEX (Longitudes) | APEX (Longlat) | APEX (Lognormal) | APEX (FB) | APEX (YCSB) | APEX (TPC-E) | LB+Tree | uTree | DPTree | FPTree | FAST+FAIR | BzTree |
|------------------|-------------------|----------------|------------------|-----------|-------------|--------------|---------|-------|--------|--------|-----------|--------|
| PM               | 2.139             | 2.159          | 2.152            | 2.29      | 2.138       | 2.244        | 2.649   | 2.98  | 3.03   | 2.542  | 2.47      | 3.46   |
| DRAM             | 0.232             | 0.311          | 0.226            | 0.676     | 0.227       | 0.265        | 0.265   | 2.396 | 0.055  | 0.055  | 0         | 0      |

we test bulk loading time of APEX and LB+Tree. As shown in Figure 21, both trees may perform differently depending on the dataset. On average APEX takes 28% less time. In the worst case (TPC-E and FB), APEX is 2.13\times slower than LB+Tree because APEX takes more time to search the best node fanout in its fanout space. We hope to further investigate on improving for such cases in future work.