Scaling Learned Indexes on Persistent Memory *

Baotong Lu[1], Jialin Ding[2], Eric Lo[1], Umar Farooq Minhas[3], Tianzheng Wang[4]
[1]The Chinese University of Hong Kong, [2]MIT, [3]Microsoft Research, [4]Simon Fraser University
[1]{btlu, ericlo}@cse.cuhk.edu.hk, [2]jialind@mit.edu, [3]ufminhas@gmail.com, [4]tzwang@sfu.ca

1 MOTIVATION

Byte-addressable persistent memory (PM) such as Intel Optane DCPMM promises persistence, low cost, high capacity and low latency on the memory bus. This opens up new possibilities for indexes that operate and persist data directly on PM, with desirable features like instant recovery while maintaining high performance. Recent learned indexes [4] exploit data distribution and have shown great performance for some workloads. The main intuition behind learned indexes is that if keys are continuous integers (e.g., 0-100 million), the value mapped by 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. Of course, the data distribution of real world data could be much more complex so that current learned indexes [2, 4] typically use a hierarchy of models that form a tree-like structure to improve accuracy.

We observe learned indexing is a natural fit for PM: Real PM (Optane DCPMM) exhibits ~4× higher latency and ~3–14× lower bandwidth than DRAM [10], whereas model-based search is especially good at reducing memory accesses. However, learned indexes were designed based on DRAM without considering PM properties, and existing PM-based indexes [8, 9] typically evolve B+-trees or hash tables which are agnostic to data distribution. Therefore, 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 [2, 4] require key-value records be maintained in sorted order, which may cause existing records to be shifted during inserts. 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. A common solution is to use unsorted nodes [9] that accept inserts in an append-only manner at the cost of linear search. 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 [2]) with sorted data. This in turn makes structural modification operations (SMOs), such as node splits, 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. 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 writes on PM are atomic. Although recent work [3, 5] 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 due to heavyweight logging internally.

* Orignally published in VLDB 2022 [7]: https://vldb.org/pvldb/vol15/p597-lu.pdf.

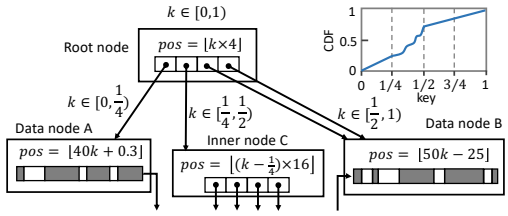

Figure 1: Overall architecture of ALEX [2], a representative updatable learned index which APEX is based upon.

2 APEX

We present *APEX*, a persistent learned index that retains the benefits of learned indexes while supporting scalable concurrency and instant recovery on PM. 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.** SMOs 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 crash consistency with low overhead. Ideally, it should support instant recovery to achieve high availability.

APEX is based on ALEX [2], a state-of-the-art updatable learned index. APEX inherits ALEX's architecture (Figure 1) to consist of inner nodes and data nodes. Each inner node uses a linear regression model to predict the next child node (model) to probe, until reaching a data node. A data node stores key-value records and supports "last-mile" search to account for any inaccuracies of the model. APEX is open-source at: https://github.com/baotonglu/apex.

2.1 Design Highlights

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**). A key observation is that each data node can be treated as a hash table which uses a model as an order-preserving hash function to predict insert locations (model-based insert). To resolve collisions without introducing excessive PM accesses, we propose a new probe-and-stash mechanism inspired by recent PM hash tables [8] (**P1**, **P2**). We set different maximum node sizes for APEX's inner and data nodes to avoid SMOs hindering scalability

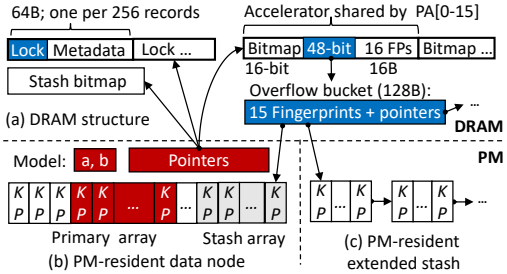

Figure 2: APEX data node layout and DRAM-resident data.

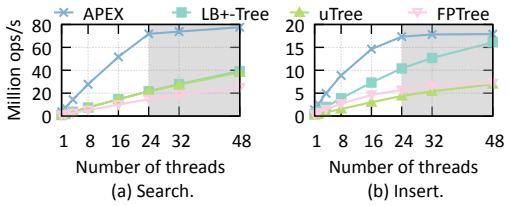

Figure 3: Throughput with a varying number of threads.

while maintaining a shallow tree (**P3**). For instant recovery, we design a log-free approach to updates with low overhead and make DRAM-resident components reconstructable on-demand (**P5**).

PM-optimized nodes with data awareness. Each inner node stores a linear model and an array of child pointers. In Figure 2, 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. We devise a probe-and-stash mechanism to avoid excessive PM accesses in data nodes. Starting from the linear model's predicted position in the primary array (PA), an insert operation linearly probes PA for a free slot. We bound the probing distance to 16, making records in PA nearly-sorted to improve (range) search performance. If no free slot is found, APEX inserts the new record to the stash array (SA). Stashing allows APEX to efficiently resolve collisions without excessive PM writes compared to ALEX's element-shift or other common techniques (e.g., chaining). We also use accelerators including fingerprints [9] to reduce the overhead of stash accesses (details in full paper [7]). Each data node has a fixed number of record slots, so APEX needs to properly divide the slots between PA and SA. Allocating more slots to PA can efficiently facilitate model-based operations, yet there must be enough stash slots in case collisions do happen. The better the model fits the underlying data, the less frequently collisions happen and thus the smaller size of SA we should allocate. APEX strikes a balance between performance and collision handling, by simulating the model's accuracy using existing data to properly sets the SA size upon node creation, making node structure data-aware.

Low-overhead data consistency with instant recovery. APEX makes index operations log-free, leveraging the fact that the integer key can be atomically updated and indicate the record's validity. Specifically, we indicate free slots by storing in them an invalid key that is out of the node's key range; upon recovery slots with invalid keys are safely ignored. Because of the low tree depth, inner nodes well uses CPU caches. Placing them in DRAM does not benefit much. Different from other PM-DRAM trees [1, 6, 9], this motivates us to place inner nodes in PM to easily enable instant recovery. APEX adopts lazy recovery [8], which serves the query requests instantly upon system restart and reconstructs the DRAM-resident metadata in data nodes on demand by the accessing threads.

Scalable concurrency. APEX adapts optimistic locking [8] for learned indexes on PM with many careful designs (e.g., every 256 records share a lock), to balance the programmability and performance. Data nodes in APEX are variable-sized, but have a *maximum* size which is set to 256KB to fully exploit model-based operations. This is larger than typical B+-tree node sizes (e.g., 256B), but small enough to efficiently implement SMOs and achieve good scalability. Since SMOs in inner nodes are relatively rare, we keep their maximum size to be 16MB, giving APEX more flexibility to select node fanout, lower tree depth and maintain good search performance.

3 EVALUATION

We conducted experiments on a server with a 24-core Xeon 6242R CPU, 768GB of DCPMM (6 DIMMs on all six channels) in AppDirect mode. We use six realistic and synthetic data sets in our workloads but only show the results on one dataset (`Longitudes`) for brevity. As shown in Figure 3, APEX scales well for lookups thanks to its lightweight concurrency control and model-based operations, being up to 3.9× faster than the state-of-the-art LB+-Tree [6]. Although no indexes scales linearly under inserts (which inherently exhibit many random PM writes), APEX is the most scalable solution.

4 CONTRIBUTION AND CONCLUSION

We make three contributions. First, APEX brings persistence to learned indexes which is a missing but a necessary feature, 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 adapt a set of techniques to implement learned indexes on real PM. APEX is based on ALEX, but our techniques (e.g., probe-and-stash) are general-purpose and applicable to other indexes. Evaluation results showed that APEX outperforms state-of-the-art by up to 3.9×.

Acknowledgements. This work is partially supported by 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.

REFERENCES

[1] Youmin Chen et al. 2020. uTree: a Persistent B+-Tree with Low Tail Latency. *PVLDB* 13, 11 (2020), 2634–2648.
[2] Jialin Ding et al. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*.
[3] Intel. 2018. PMDK. (2018). http://pmem.io/pmdk/libpmem/.
[4] Tim Kraska et al. 2018. The Case for Learned Index Structures. In *SIGMOD*.
[5] Se Kwon Lee et al. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *SOSP*.
[6] Jihang Liu et al. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.
[7] Baotong Lu et al. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proc. VLDB Endow.* 15, 3 (2021), 597–610.
[8] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (2020), 1147–1161.
[9] Ismail Oukid et al. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*.
[10] Jian Yang et al. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST*. 169–182.