How Does Updatable Learned Index Perform on Non-Volatile Main Memory?

Leying Chen
State Key Laboratory of Computer Architecture
Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
chenleying19z@ict.ac.cn

Shimin Chen*
State Key Laboratory of Computer Architecture
Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
chensm@ict.ac.cn

***Abstract*—Recent work on learned index opens a new research direction for index structures. By exploiting the data distribution, a read-only learned index can achieve much better time and space performance than B+-Trees. A number of recent studies propose enhanced learned index structures that provide various levels of support for insertions and deletions. Among them, ALEX, an updatable adaptive learned index, provides the most comprehensive support, allowing on-the-fly insertions and dynamically adjusting the index structure similar to the B+-Tree. In this paper, we study learned index from a new perspective, trying to understand how ALEX performs on Non-Volatile Main Memory (NVM). We analyze the insertion behaviors of ALEX and experimentally evaluate its performance on a real machine equipped with Intel Optane DC Persistent Memory. We find that learning data distributions in learned index makes it feasible to create very large leaf nodes, and reduce the tree height drastically for better search performance. However, this design choice causes a large number of NVM writes for insertion operations, even for the well optimized ALEX design.**

I. Introduction

Recent work on learned index [1] opens a new research direction for index structures. A range index structure, such as B+-Tree, is viewed as a regression tree that maps keys to the corresponding positions in the sorted array of index entries. This opens the new possibility of replacing existing index trees with learned models, which are built based on the data distribution. The proposed RM-Index [1] outperforms the traditional B+-Tree on both search performance and indexing space overhead.

As the RM-Index focuses on read-only workloads, several recent studies propose enhanced learned index structures, such as FITing-tree [2], ALEX [3], and PGM-index [4], that provide various levels of support for insertions and deletions. Among them, ALEX [3] provides the most comprehensive support, allowing on-the-fly insertions and dynamically adjusting the index structure similar to B+-Tree. ALEX employs gapped arrays in the data node and proposes mechanisms including node expansion and node splits for insertions. Experimental results based on DRAM show that ALEX achieves higher index throughput than both RM-Index and B+-Tree for a wide variety of workloads, including read-only, read-heavy, write-heavy, and write-only workloads.

*Corresponding Author

In this paper, we study learned index from a new perspective, trying to understand how ALEX performs on emerging Non-Volatile Main Memory (NVM) [5]–[8]. 3DXPoint based Intel Optane DC Persistent Memory [8] has been available in the mainstream computing market since April 2019. A dual-socket server can be equipped with up to 6TB of 3DXPoint NVM main memory, which is much higher than the DRAM size on a common server machine. The much larger main memory may bring significant benefit to database and big data systems because more applications can reduce disk I/O overhead by putting their working data sets in main memory.

NVM has different characteristics from DRAM. First, 3DX-Point NVM has (e.g., 2–3x) lower bandwidth than DRAM. Second, NVM write bandwidth is lower than NVM read bandwidth. Third, the persist operations (which use cache line flush and memory fence instructions to flush data from the volatile CPU cache to NVM for crash consistency purpose) are significantly slower than normal NVM writes. As a result, recent studies on NVM optimized B+-Trees [9]–[14] exploit DRAM for non-leaf search and reduce NVM writes for better insertion performance.

With a closer look at the learned index structures, we see that learning data distributions makes it feasible to create very large (e.g., $\sim$100KB to $\sim$MB) leaf nodes compared to B+-Trees (which often have 128–1024B nodes in main memory). This can drastically reduce the tree height, leading to substantial performance gains for search. However, this design choice has implications for insertions. We would like to better understand the insertion behavior of ALEX and experimentally study its performance on NVM.

In the rest of the paper, we begin by describing background on NVM and on learned index in Section II. Then, we study the insertion operation of ALEX in detail and describe how to support persistent ALEX on NVM in Section III. After that, we perform experiments on a real machine to study how ALEX performs on NVM in Section IV. Finally, we discuss our findings and conclude the paper in Section V.

II. Background

A. Non-Volatile Main Memory

DRAM scaling encounters significant challenges. The response from academia and industry to this challenge is a

new generation of Non-Volatile Memory (NVM) technologies, including PCM [5], STT-RAM [6], Memristor [7], and the first commercially available technology, 3DXPoint [8]. Intel has been shipping 3DXPoint based SSD products and memory DIMM products. The latter is called Intel Optane DC Persistent Memory, and has been available in the mainstream computing market since April 2019.

A dual-socket server has 6 NVM DIMM slots for either of the two CPU sockets. An NVM DIMM is of 128GB–512GB large. Therefore, a dual-socket server can be equipped easily with 6TB of NVM, roughly an order of magnitude larger than the DRAM capacity on a server with similar configurations. The benefits of NVM to database systems are twofold. First, more data can fit into main memory, leading to lower I/O overhead during normal operations. Second, data can be persisted in NVM, which permits faster crash recovery.

As discussed in Section I, NVM has different characteristics from DRAM. Recent studies have proposed various techniques to improve the performance of B+-Tree indices on NVM [9]–[14]. First, selective persistence, i.e. placing the non-leaf nodes in DRAM and leaf nodes in NVM, effectively improves search performance. As all index entries are stored in the leaf nodes, the nonleaf nodes can be rebuilt from the leaf nodes upon crash recovery. Second, unsorted leaf nodes, i.e. allowing index entries to be unsorted within a leaf node and using a bitmap to specify which index slots are taken, significantly reduce the number of NVM writes for insertions and deletions. Third, a number of techniques are based on unsorted leaf nodes, including atomic NVM writes, entry moving, and logless node splits. They further improve the performance of insertions to both leaf nodes with empty slots and full leaf nodes. It would be interesting to see if the above techniques can be applied to learned index structures.

B. Learned Index

Learned Index. Kraska et al. proposes the RM-Index in the seminal learned index paper [1]. As shown in Figure 1, index entries are stored in a sorted array. Given a search key, the learned models predict the position of the key in the sorted array. The prediction may not be exactly precise. The RM-Index then performs a local search in the sorted array to locate the desired entry. Kraska et al. find that a single learned model cannot capture the low-level details of the data distribution. Therefore, the RM-Index can consist of multiple levels of learned models. The RM-Index is designed for read-only workloads. Insertions, deletions, and updates can be incorporated by re-building the index offline.

Updatable Learned Index. Recent studies propose a number of enhanced learned index structures that provide various levels of support for insertions and deletions [2]–[4].

The FITing-tree [2] proposes an in-place strategy and a delta strategy for supporting insertions. The in-place strategy stores index entries in the middle of a page, and preserves a number of empty entry slots at the beginning and at the end of a page. For an insertion into a page, the sorted entries in the page are moved to create a new slot for the insertion. The direction of the movement depends on whether the left or the right end of the array is close to the insertion position. However, this strategy may result in large movement cost. The delta strategy maintains a delta buffer for each segment of index entries for insertions. When the delta buffer is full, the segment of data and the delta buffer are merged. Then a new learned model is computed on the newly merged segment of data.

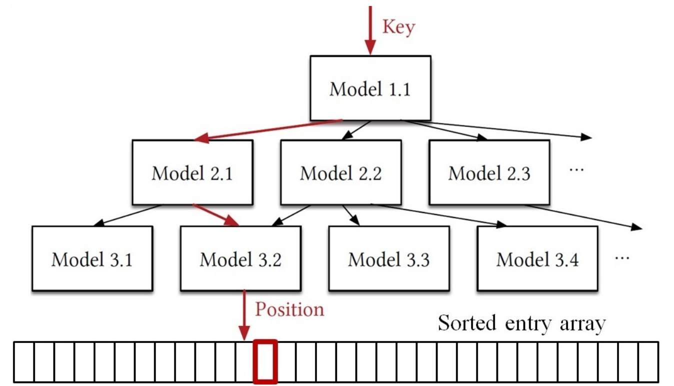

Fig. 1. Learned Index (RM-Index) proposed by Kraska et al. [1].

The PGM-index [4] employs a logarithmic method for insertions. There are a series of PGM-indexes on sets S_0, ..., S_k of keys. S_i is either empty or contains 2^i keys. For an insertion of key x, the first empty set S_l is located such that S_i ($i = 0$, ..., $l-1$) is not empty. Let S_l=$S_0 \cup \cdots \cup S_{l-1} \cup \{x\}$. The method then creates a new PGM-index on S_l and removes data sets S_0 to S_{l-1} and their corresponding PGM-indexes. In this way, the PGM-index remains read-only, while the logarithmic method gives $O(\log n)$ amortised time per insertion.

Neither the FITing-tree nor the PGM-index is a true dynamic index structure. Both the delta strategy and the logarithmic method avoid dynamically updating the index structures. This may incur significant tail latency.

In contrast, ALEX [3] is a fully dynamic index structure. It employs gapped arrays in the data nodes to reduce the entry movement cost for insertions into non-full data nodes. When a data node is close to full, ALEX first attempts to expand the node by allocating a larger gapped array. If the node is too large, ALEX splits the node into two nodes by either splitting sideways or splitting down.

As ALEX provides the most comprehensive insertion support for learned index, we focus on understanding the performance of ALEX on NVM in the following.

III. ALEX on Non-Volatile Main Memory

We describe the data structure and index insertion operations of ALEX in Section III-A. This description is based on the ALEX paper [3] and the source code on github [15]. This sets the ground for an in-depth analysis to understand the difference between ALEX and B+-Trees in Section III-B. The focus of the analysis is on whether techniques previously proposed for NVM-optimized B+-Trees can be easily employed to improve ALEX. Finally, we describe our implementation of ALEX variants for NVM memory in Section III-C.

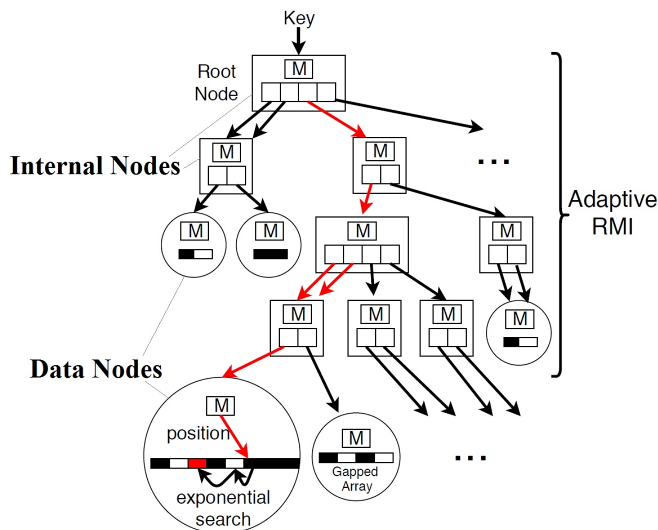

Fig. 2. ALEX proposed by Ding et al. [3].

A. ALEX Structure and Insertion Operations

Figure 2 depicts the ALEX data structure. In ALEX terminology, leaf nodes are called data nodes. Non-leaf nodes are called internal nodes. A data node (internal node) is depicted as a circle (rectangle) in the figure. The "M" in every node represents the learned model. ALEX employs the linear regression model. A model is stored as two double values for the slope and the intercept in every node. From the figure, the data nodes of ALEX may be on different levels. This is in contrast to B+-Trees, where leaf nodes are on the same level, and the RM-Index, where the number of models from the root to the sorted array is the same for all keys. This is an interesting design choice. It allows ALEX to flexibly choose how to handle the node full case for insertions. However, the downside is that the tree is no longer balanced and there can be different number of pointer chases for different keys.

An internal node contains an array of pointers to child nodes. The number of pointers in the array is always a power of 2. The maximum size of the pointer array is 16MB. A data node contains an array of keys, an array of payloads, and a bitmap array. As shown in Figure 2, there are gaps in the key and payload arrays. The bitmap array records which slot is used or empty. This gapped array design reduces the movement cost for insertions. The maximum size of a data node is also 16MB.

Insertion to Data Nodes with Empty Slots. When creating a new data node in bulkload, ALEX sets the load factor of the data node to be 0.7. That is, 70% of the slots are used to store index entries, while 30% of the slots are gaps. To insert an entry to a leaf node with empty slots, ALEX computes the predicted position for the entry using the linear model in the node. Then it inserts the entry to the predicted position in the array if the position is a gap. Otherwise, ALEX creates a gap at the predicted position for the new entry by shifting existing entries by one slot in the direction of the closest gap.

As the load factor increases, the number of shifts per insertion increases. ALEX considers that a data node is "full" if the load factor reaches 0.8. That is, when 80% of the slots in a data node are taken, ALEX considers that a node is full and it performs either node expansion or node splits for inserting into a full data node.

Node Expansion. When an insertion goes to a full node, ALEX considers expanding the node as long as the resulting node size is not greater than the 16MB size limit. The new node is set to have a load factor of 0.6. Suppose the original data node contains k slots with 80% of the slots taken. Then the new data node has $\frac{0.8k}{0.6}$=$\frac{4}{3}k$ slots, $\frac{1}{3}$ larger than the original node. ALEX either scales the linear model by a factor of $\frac{4}{3}$ or retrains the model if it finds that the lookup and insertion costs substantially deviate from expected costs.

Node Splits. When the node is too large to expand, ALEX splits the node. There are two ways to split a data node in ALEX: splitting sideways and splitting down. In splitting sideways, the data node D is split into two data nodes $D1$ and $D2$. The pointer array of the parent internal node P is updated. Because the pointer array size of an internal node is always power of 2, it is possible that multiple entries in P's pointer array point to the same child node. If this is the case for D, then the pointers to $D1$ and $D2$ can be directly updated in P's pointer array. If there is only a single pointer to D in P, then the size of the parent node P is doubled to accommodate the two child pointers.

In splitting down, ALEX also splits the data node D into two data nodes $D1$ and $D2$. However, the new pointers are not inserted into the parent internal node P. Instead, ALEX creates a new internal node P' as the parent node of $D1$ and $D2$. P' becomes a child node of P. ALEX replaces the child node pointer to D with the pointer to P' in node P.

ALEX selects the action that results in a tree with lower expected lookup and insert costs. In the case where the parent internal node P would become larger than 16MB by splitting sideways, splitting down has to be used.

B. ALEX vs. B+-Trees: Can We Easily Apply Techniques from NVM-Optimized B+-Trees?

A number of techniques have been shown to effectively improve NVM-optimized B+-Trees, as discussed previously in Section II-A. A natural question to ask is whether these techniques can be applied to ALEX.

Selective Persistence. In B+-Trees, selective persistence places nonleaf nodes in DRAM and leaf nodes in NVM. The straight-forward way to apply selective persistence to ALEX is to put internal nodes in DRAM and data nodes in NVM. However, there are two complications.

First, during crash recovery, nonleaf nodes of B+-Trees can be rebuilt by scanning leaf nodes in NVM. However, it is challenging to rebuild the internal nodes by scanning the data nodes of ALEX. The bulkload operation in learned index designs, including ALEX, are typically *top down*. It starts by computing a learned model for the entire data set. This learned model will be stored in the root level. Then the data

set is divided into smaller data sets. The process is repeated for each smaller data set to create nodes in deeper level of the tree. It is unclear how to efficiently rebuild the internal nodes *bottom up* from data nodes. One naïve idea is to collect all the data from all data nodes and bulkload a new ALEX tree. However, the resulting data nodes in the new tree will probably be different from before, and the cost of the bulkload recomputation can be quite significant.

Second, the benefit of selective persistence comes from faster search in the nonleaf levels of B+-Trees. In comparison, the nonleaf portion of a learned index is often much shallower and much smaller than that of a B+-Tree [1], [3]. For example, the nonleaf portion of ALEX is on average 1–2 levels [3]. Therefore, a large fraction (if not all) of the internal nodes can stay in the CPU cache. The benefit of placing the internal nodes into DRAM may be less significant for learned indexes, such as ALEX.

Unsorted Leaf Nodes. In B+-Trees, the keys in a leaf node are unsorted to avoid NVM writes due to shifting entries for insertions. However, This is not directly applicable to ALEX. The linear model is monotonic. That is, larger keys lead to larger predicted positions. This naturally maps to sorted keys. Moreover, if the predicted position for a search key is not accurate, ALEX has to do a local (exponential) search. The local search assumes that the keys are sorted. Otherwise, ALEX would have to scan the entire (large) data node for the key. This would be very expensive especially for negative searches (for keys not exist in the index).

The gapped array design in ALEX reduces the entry shifting cost. Suppose the load factor of the node is f. For the ideal case where the keys are independently uniformly distributed, the expected number of entries between two gaps is $\frac{f}{1-f}$. For example, when $f = 0.7$, the expected number of entries between two gaps is 2.33 for the ideal case. The number of entries to shift for an insertion is about 1.17, which is quite small. However, when the distribution is not ideal, there can be long sequence of keys without gaps, incurring high cost.

Atomic Writes, Entry Moving, and Logless Node Splits. These techniques require unsorted leaf nodes to be implemented efficiently. For example, the atomic write optimization places the newly inserted entry into an empty slot, then uses a single NVM atomic write to change the bitmap (and other relevant metadata) in the header of the node. The state of the node is consistent both before and after the NVM atomic write. However, without unsorted nodes, ALEX has to shift existing entries if the predicted position for the newly inserted entry is occupied. The shifting changes the node state. Therefore, such an insertion must be protected by more heavy-weight mechanisms, such as write-ahead logging (WAL).

Summary. From the above discussion, we see that the techniques previously proposed for improving NVM-optimized B+-Trees can NOT be easily applied to improve ALEX.

C. ALEX Implementations to Compare in Experiments

We compare three ALEX implementations in our experiments. (1) *Original ALEX on DRAM*, obtained from github [15]; (2) *ALEX on NVM without flush*, for which we allocate all the ALEX nodes in NVM by replacing the allocation calls in the ALEX code, and using in-placement constructor invocations; and (3) *ALEX on NVM with flush*, for which we protect the NVM writes with write-ahead logging, and issue `clwb` and `sfence` to achieve persistence.

IV. Experimental Study

A. Experimental Setup

We run all experiments on a Ubuntu Linux machine with 2 Intel Xeon Gold 5218 2.30GHz CPU, each equipped with 192GB (6x32GB) DRAM and 768GB (6x128GB) 3DXPoint based Intel Optane DC Persistent Memory NVDIMMs. We bind the ALEX process to CPU socket 0 and only access NVM on pmem0 to avoid any NUMA effects. We map NVM via function `pmem_map_file` in PMDK (Persistent Memory Development Kit).

Datasets. We use four datasets for index keys, whose characteristics and CDFs are shown in Table I and Figure 3. The longitudes dataset consists of the longitudes of locations in North America from Open Street Maps [16]. The longlat dataset consists of compound keys that combine each pair of longitude and latitude in North America with the transformation k=180×⌊longitude⌋+latitude. The lognormal dataset is generated according to a lognormal distribution with μ=0 and σ=2, multiplied by 10^9 and rounded down to the nearest integer. The random dataset consists of the lower 64-bits of randomly generated uuids (using Python `uuid.uuid4`).

TABLE I
DATASET CHARACTERISTICS.

	longitudes	**longlat**	**lognormal**	**random**
Num keys	1B	200M	200M	200M
Key type	double	double	64-bit int	64-bit int
Payload size	8B	8B	8B	8B

Workloads. We use five workloads in the experiments. For four of the five workloads, we bulkload 100 million initialization keys randomly chosen from a given dataset, then perform 10 million random read/write operations. We vary the read/write mix for the four workloads: (1) Read-only, (2) Read-heavy (95% reads and 5% inserts), (3) Write-heavy (50% reads and 50% writes), and (4) Write-only. For workload (5) Insertion from scratch, we bulkload 100 keys, then insert 10 million keys. This workload incurs a lot of node expansion and node splits.

B. Impact of NVM on ALEX

Workloads after Bulkloading. We study Workload (1)–(4) for all the datasets. Table II shows the statistics on ALEX structures after bulkloading. We see that the average depths (levels of internal nodes) is 1.02–3. The median data node size is 17.6KB–8.99MB. Compared to a typical main memory B+-Tree, ALEX is much shallower and the data node size is much larger. This leads to the good search performance and small indexing space overhead of ALEX.

Figure 4 compares the throughput of the three ALEX implementations for the four workloads on the four datasets.

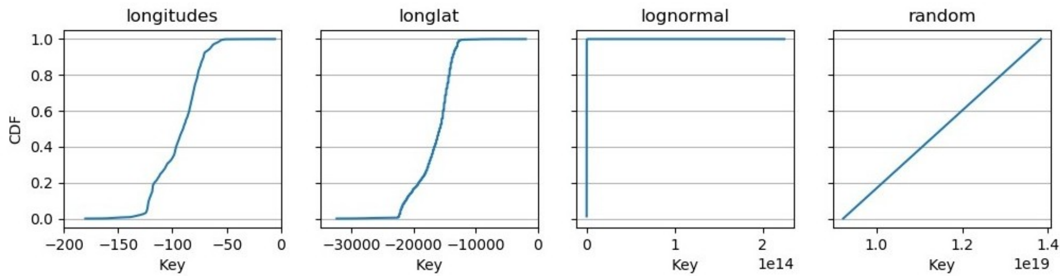

Fig. 3. CDF (Cumulative Distribution Function) of datasets.

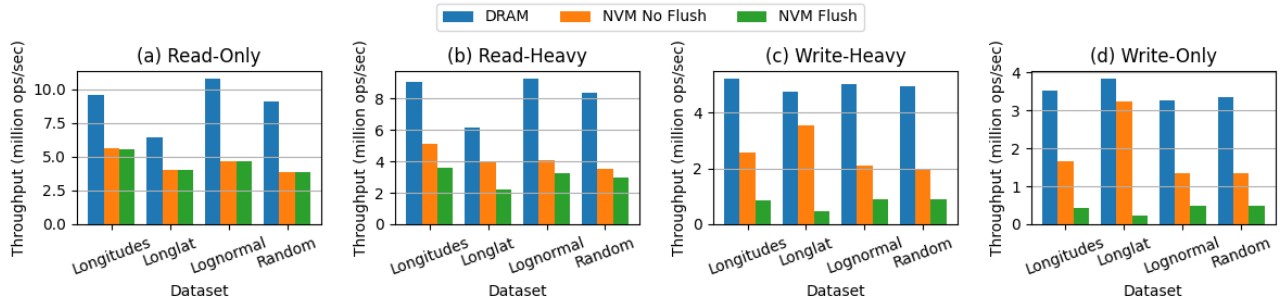

Fig. 4. Comparing three ALEX implementations using four workloads on four datasets. (ALEX is bulkloaded with 100 million keys. Then, a batch of 10 million index operations are performed in each workload with different read/write mixes.)

TABLE II
STATISTICS AFTER BULK LOAD.

	longitudes	longlat	lognormal	random
Avg depth	1.02	1.41	1.71	3
Max depth	4	4	3	3
Num inner nodes	385	5776	30	132
Num data nodes	12542	48133	1243	308
Min DN size	24B	24B	40B	2.22MB
Median DN size	151KB	17.6KB	1.38MB	8.99MB
Max DN size	2.05MB	3.27MB	14.4MB	9.04MB

TABLE III
DATA NODE STRUCTURAL CHANGES IN WRITE-HEAVY WORKLOAD.

	longitudes	longlat	lognormal	random
Expand + scale	3	4166	0	0
Expand + retrain	57	1385	0	0
Split sideways	43	1244	0	0
Split downwards	0	121	1	0
Total times full	103	6916	0	0

We see that compared to ALEX residing in DRAM, ALEX on NVM is 1.59–2.35x, 1.55–2.38x, 1.35–2.46x, and 1.19–2.51x slower for read-only, read-heavy, write-heavy, and write-only workloads, respectively. NVM persist operations further lower the throughput for workloads that perform insertions. ALEX on NVM with flush is 2.50-2.85x, 5.47-10.38x, and 6.72-16.94x slower than the ALEX on DRAM for read-heavy, write-heavy, and write-only workloads, respectively.

For the read-only workload, although there is no NVM write, the two NVM based implementations still perform worse than ALEX on DRAM. This is because NVM read performance is lower than DRAM.

For workloads that perform insertions, we see that as the percentage of writes increases, the performance difference between ALEX on NVM with flush and ALEX on DRAM increases. As shown in Figure 5, it is clear that more `clwb` and `sfence` are issued as the workload becomes more write intensive. We measure the average number of element shifts

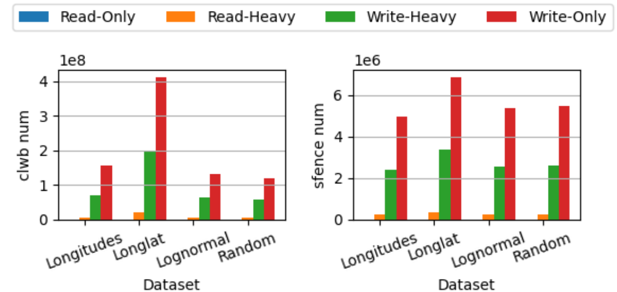

Fig. 5. Number of Flush operations for experiments in Figure 4.

for the workloads. For all datasets, the average number of shifts per insertion is 2.51–8.41. The expected number of shifts is 0.75–2 as the load factor of a data node is between 0.6 and 0.8. The actual number of shifts is higher than the expected number. This number is still reasonably low compared to the node size. This means that the gapped array design is quite effective for the workloads after bulkloading.

Interestingly, we see that longlat has the largest number of flush operations. This can be explained by Table III, which reports the number of node restructure operations for the write-heavy workload. The dataset longlat is the most complex among all the datasets, whose distribution has many detailed low level structures (as pointed out in the ALEX paper [3]). We see that there are much larger number of node expansions and node splits for the longlat dataset, incurring significantly more NVM writes. We also see that there is no node expansion or splits for the lognormal and random datasets. This means that all insertions find empty slots in their target data nodes for these two datasets.

Insertion from Scratch. The last workload stresses ALEX for inserting 10 million keys to a small tree bulkloaded with 100 keys. This workload incurs much larger number of data node structural changes, as shown in Table IV. Figure 6

TABLE IV
DATA NODE STRUCTURAL CHANGES IN INSERTION FROM SCRATCH.

	longitudes	longlat	lognormal	random
Expand + scale	20764	246769	4143	5009
Expand + retrain	1883	81256	120	175
Split sideways	4677	15784	198	171
Split downwards	0	806	2	0
Total times full	27324	344615	4463	5355

TABLE V
STATISTICS AFTER INSERTION FROM SCRATCH.

	longitudes	longlat	lognormal	random
Avg depth	1	388	1.001	1
Max depth	1	749	2	1
Num inner nodes	1	807	3	1
Num data nodes	4688	16601	388	407
Min DN size	24B	24B	136B	26.9KB
Median DN size	14.2KB	14.9KB	417KB	417KB
Max DN size	2.58MB	408KB	4.16MB	3.12MB

compares the throughput of the three ALEX implementations for insertion from scratch. We see similar trends as in Figure 4. ALEX on NVM with flush is 6.86–12.26x slower than ALEX on DRAM for longitudes, lognormal, and random datasets.

For the most complex data set, longlat, the resulting tree is poorly structured. As shown in Table V, the average depth of the tree is 388 levels, compared to 1.41 after bulkloading in Table II. This results in a large number of node restructuring operations, leading to a great amount of computation. For ALEX on DRAM, the throughput of longlat is 63x lower than that in the write-only workload. The unbalanced design in ALEX may cause very poor performance in certain situations.

Comparison with NVM Optimized B+-Tree. Finally, we compare the performnce of ALEX with NVM flush and LB+-Tree, an NVM optimized B+-Tree. As shown Figure 7, for read-only workloads, ALEX has higher throughput than LB+-Tree. This is expected as ALEX is much shallower than B+-Trees. However, for write-only workloads, ALEX is significantly slower, obtaining less than half the throughput of LB+-Tree. This is because ALEX incurs larger number of NVM write and persist operations.

V. DISCUSSION AND CONCLUSION

Both learned index and NVM have promising futures. However, when we simply move ALEX from DRAM to NVM, it performs 1.19-2.51x worse on all kinds of datasets and workloads. Flush operations make the performance drop even larger. ALEX in NVM with WAL and flush operations is 1.58-16.94x slower than the original ALEX.

Model-based insertions in ALEX often incur element shift when there are empty slots in the data node. Node expansions and splits are common in complex datasets and incur a large number of NVM writes and flush operations. WAL incurs redundant NVM writes and flushes in order to ensure that the index is consistent after crash. Unfortunately, techniques previously proposed for NVM-optimized B+-Trees, including selective persistence, unsorted nodes, and NVM atomic writes, are not easily applicable to ALEX. It is interesting yet challenging to design an efficient updatable learned index structure on NVM main memory.

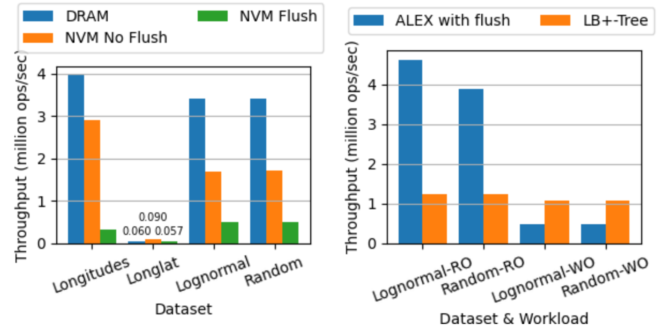

Fig. 6. Comparing ALEX implementations for insertions from scratch.

Fig. 7. Comparing ALEX with an NVM optimized B+-Tree.

VI. ACKNOWLEDGMENTS

This work is partially supported by National Key R&D Program of China (2018YFB1003303).

REFERENCES

[1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.
[2] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *SIGMOD*, 2019, pp. 1189–1206.
[3] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska, "ALEX: an updatable adaptive learned index," in *SIGMOD*, 2020, pp. 969–984.
[4] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *PVLDB*, vol. 13, no. 8, pp. 1162–1175, 2020.
[5] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. Chen, H. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4-5, pp. 465–480, 2008.
[6] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer torque magnetic random access memory (STT-MRAM)," *JETC*, vol. 9, no. 2, pp. 13:1–13:35, 2013.
[7] J. J. Yang and R. S. Williams, "Memristive devices in computing system: Promises and challenges," *JETC*, vol. 9, no. 2, pp. 11:1–11:20, 2013.
[8] D. H. Graham, "Intel Optane technology products - what's available and what's coming soon," https://software.intel.com/en-us/articles/3d-xpoint-technology-products.
[9] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 21–31.
[10] S. Chen and Q. Jin, "Persistent B+-Trees in non-volatile main memory," *PVLDB*, vol. 8, no. 7, pp. 786–797, 2015.
[11] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for nvm-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, J. Schindler and E. Zadok, Eds. USENIX Association, 2015, pp. 167–181.
[12] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 371–386.
[13] J. Arulraj, J. J. Levandoski, U. F. Minhas, and P. Larson, "BzTree: A high-performance latch-free range index for non-volatile memory," *PVLDB*, vol. 11, no. 5, pp. 553–565, 2018.
[14] J. Liu, S. Chen, and L. Wang, "LB+-Trees: Optimizing persistent index performance on 3dxpoint memory," *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 1078–1090, 2020.
[15] Microsoft, "ALEX code," https://github.com/microsoft/ALEX.
[16] "OpenStreetMap on AWS," https://registry.opendata.aws/osm/.