TONE: Cutting Tail-Latency in Learned Indexes

Yong Zhang
McGill University
Montreal, Canada
yong.zhang3@mail.mcgill.ca

Xinran Xiong
McGill University
Montreal, Canada
xinran.xiong@mail.mcgill.ca

Oana Balmau
McGill University
Montreal, Canada
oana.balmau@cs.mcgill.ca

Abstract

Low memory footprint and tail latency are important in indexing for data management systems. Learned indexes have been gaining popularity in recent years due to their low memory overhead, and adaptability to fluctuations in workloads. However, state-of-the-art learned indexes are optimized for read-heavy workloads where the key access distribution is relatively stable, and thus exhibit high tail latency for insertions.

Our main observation is that high insertion tail latency stems from structural modification operations in the index, such as node expansions and splits. Based on this insight, we propose a new tail-latency optimized learned index we call TONE. TONE has a novel node layout for leaves and adaptable policies for triggering structural modification operations. Moreover, TONE provides a shard-based mechanism to boost parallelism, while minimizing contention.

We evaluate the insertion tail-latency and overall throughput of TONE and provide a comprehensive comparison to state-of-the-art indexes such as skiplists (used in RocksDB) and existing learned indexes. We show that TONE reduces the 99p write tail-latency up to one order of magnitude and has a low memory use.

CCS CONCEPTS

•Information systems ~ Information storage systems ~ Storage management

KEYWORDS

Learned index, Tail latency, Indexing

ACM Reference format:

Yong Zhang, Xinran Xiong and Oana Balmau. 2022. TONE: Cutting Tail-Latency in Learned Indexes. In Proceedings of ACM Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS'22). https://doi.org/10.1145/3503646.3524295

CHEOPS '22, April 5, 2022, RENNES, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9209-9/22/04…$15.00
https://doi.org/10.1145/3503646.3524295

1 Introduction

Learned indexes [8, 9, 11, 12, 16] propose replacing classic index structures (e.g., B-tree, hash-tables, and skiplists) with *models* that predict the position of a data record in a given dataset. Their main advantage over existing indexes is adapting to common patterns that are prevalent in real world data. Learned indexes take advantage of such patterns by providing hotter items with shorter access paths, or by allowing for more entries in densely populated parts of the dataset. In addition, due to their use of small models, learned indexes have a much lower memory footprint than existing data-structures that need to maintain additional metadata to locate each record (e.g., routing nodes in B-trees [6]).

Despite their many advantages, to the best of our knowledge, learned indexes have not yet been used to replace existing indexing structures in large-scale systems. The reason is two-fold. First, every time the data distribution changes significantly, learned indexes need to be retrained to accommodate the workload change. Maybe surprisingly, we find that even though the retraining itself may not be costly if simple models are used, the node splits and data movement that occur post-retraining dominate tail latency in existing learned indexes. Second, we find similar spikes in tail latency in write-heavy workloads which also require significant structural readjustment. Therefore, we conclude that while existing indexes are excellent when the characteristics of datasets are relatively static, once this assumption breaks, they become difficult to use in practice.

In this paper, we take one more step towards making learned indexes truly practical. We introduce TONE, the first tail-latency optimized learned index. TONE reduces latency caused by structural modifications that appear in *write-intensive workloads, with fluctuating data distributions.*

The main design contribution in TONE is simple, yet effective. We designed a novel two-level leaf node which absorbs insertions, while allowing the structural modifications of the tree (e.g., node expansions and splits) to proceed in the background. The first array is allocated by default, while the second array is allocated on an as-needed basis. Intuitively, this secondary array serves as a temporary buffer for bursts of writes, postponing node expansions while maintaining model accuracy. Our experiments show that TONE's leaf structure does not add any memory overhead compared to existing learned indexes.

In summary, we make the following contributions:

(1) Analysis of tail latency causes in existing learned indexes. Contrary to popular belief, we show that node expansions and splits are more costly than training time, in dynamic workloads [3, 4, 19].
(2) Leaf layout for write-optimized learned indexes. Our two-level leaves allow TONE to decrease the average number of structural modifications, thus reducing tail latency.
(3) TONE open-source implementation. In TONE, we extend the leaf layout to accommodate parallel read, write, and scan requests.
(4) Experimental evaluation. We show that TONE provides up to one order of magnitude reduction in tail latency in dynamic benchmarks, while maintaining high throughput and low memory footprint.
Roadmap. The rest of this paper is organized as follows. Section 2 gives an overview of existing learned indexes. Section 3 provides an analysis of tail latency causes in existing learned indexes. Section 4 presents the design and implementation of TONE. Section 5 presents the experimental evaluation. Section 6 presents the related work, and Section 7 concludes, providing directions we plan to follow in future work.

2 Background

2.1 Read-only Learned Indexes

Learned indexes were introduced by Kraska et al [12]. The main idea is to "learn" the cumulative distribution function of a given key space, and then predict the position of keys within this distribution during client lookups. Similar to B+-trees, learned indexes divide the key space into smaller subranges hierarchically. Each model plays a role similar to that of an intermediate node in a B+-tree. However, the models in intermediate nodes are machine learning estimators such as linear regression, or neural networks.

On a high-level, each of the models in the hierarchy makes a prediction about the position for the searched key K. Then, this prediction is used to select the next model, which is responsible for a sub-area of the keyspace and can make a better prediction with a lower error. Eventually, one of the models predicts a location inside a leaf node. If the predicted position did not successfully locate K, a local search is carried out inside the leaf (e.g., binary search, exponential search [10]). Even though the key position may not be correct, it is certain that K is located in the leaf predicted by the model hierarchy, or it is absent from the dataset.

Such a model hierarchy is useful because it makes it possible to split complex data distributions into easier to learn shapes, and thus increases accuracy for last-mile searches inside the leaf nodes. Moreover, learned indexes remove the overhead of search inside the tree, because they simply query a sequence of models with no extra search before reaching the desired leaf.

2.2 Updating Learned Indexes

Despite their advantages, one significant drawback of classic learned indexes is that they are static. In case the data access distribution changes, it is impossible to modify the index structure after it learned the data distribution without a full reconfiguration and re-training. Thus, classic learned indexes cannot adapt to changes in the key distribution automatically. Moreover, they were not designed with write-heavy workloads in mind and do not support insertions. Unfortunately, dynamic update-heavy workloads are the norm in practice [3, 4, 14, 19].

To address these limitations, Ding et al. propose ALEX, an updatable learned index [8]. ALEX modifies the learned index leaves to support updates through a structure called a *gapped array*. Gapped arrays are arrays with pre-allocated empty slots. Similar to the empty slots provided by leaves in B+-trees, gapped arrays buffer incoming inserts. If a gapped array is full or it is not possible to fit an element in a desired slot, ALEX changes the tree structure to accommodate more insertions. Such structural changes entail data movement, as well as retraining of some of the models to accurately reflect the data distribution changes.

Even though ALEX allows for structural modifications, its current implementation is single-threaded due to the complexity of these operations. Several recent works extend the classic learned indexes with concurrency control mechanisms [13, 16, 17]. For instance, XIndex [16] adopts a two-layer design where the top layer serves as a direction layer and the bottom group layer stores the data records. Note that despite the two-layer design, XIndex uses the top layer as a metadata layer, thus significantly increasing memory overhead, as we show in Section 5.

The primary goal in prior work is to improve overall throughput. In contrast, in this work, we focus on reducing learned index tail latency in write-heavy workloads, while preserving high throughput and a low memory footprint.

3 Tail Latency Analysis in Learned Indexes

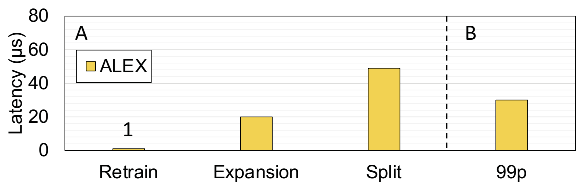

Figure 1. Write latency breakdown for ALEX. Data movement dominates the tail latency in existing learned indexes. Part A shows the latency breakdown for the key data movement operations which contribute to tail latency. Part B shows the 99p write latency.

Figure 1 illustrates the average latency breakdown of node expansions, node splits, and model rebuilding in ALEX in a 50% write, 50% read workload with 1 kilobyte (KB) key-value pairs. ALEX was run under its default configuration. Clearly, data movement caused by node expansions and splits is the main reason for high tail latency.

A node expansion happens when a data node reaches a high occupancy level (a configurable parameter, e.g., 80% of the gapped array size), but the model associated to the data node is still accurate. In other words, the data access pattern is relatively stable, thus the learned index predictions are still accurate. In this case, the data node grows, allocating more array slots and simply scaling up the linear model.

If the model predictions deviate significantly from the actual record position for significant number of queries (a configurable parameter), then the learned index may consider to perform a node split or model rebuilding depending on the learned index cost model. In ALEX, the cost model selects the option with the lowest expected cost.

The fact that the model rebuilding does not dominate the cost may be surprising. However, due to the hierarchy of models employed by learned indexes, each individual model can be simple, and thus easy to retrain. In ALEX's case, each model is a linear function whose retraining comes at a fraction of the cost of data movement.

4 TONE

In this section, we introduce the design and implementation of TONE. We first give an overview of the system and its main data structures, then describe the TONE API and implementation in detail.

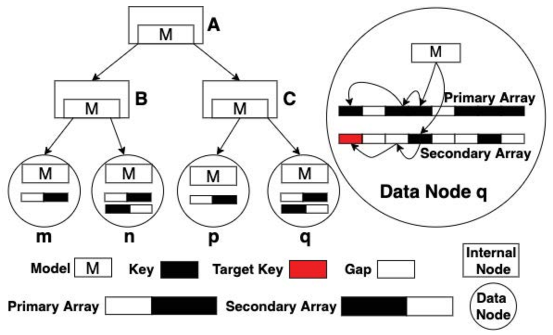

Figure 2. Two-level leaf structure in TONE. The internal structure of the data node `q` is shown on the right hand side. Exponential search for a target key is conducted in the secondary array, and then in the primary array, if necessary.

4.1 TONE in a Nutshell

In short, the basic idea of TONE is to use a two-level leaf component, where the first level—the *primary array*— absorbs incoming updates, and the second level—the *secondary array*— is created on an as-needed basis in order to postpone structural modifications in the index. Both arrays are gapped arrays, and by default have equal capacities. Note that, to maintain an equal memory footprint to a learned index that uses a single gapped array (e.g., ALEX), the maximum size of the primary array should be set to half the maximum size of the single gapped array.

When the primary array is full, TONE dynamically allocates the secondary array rather than expanding the existing gapped array. At the same time, if necessary TONE triggers model retraining, as the secondary array absorbs the extra updates. This design allows TONE to reduce tail latency, as well as to minimize memory footprint.

4.2 TONE Structure

Figure 2 shows an instance of TONE with a hierarchical depth of three models. Internal nodes `A, B,` and `C` are simple linear models. Nodes `m, n, p,` and `q` are leaves that contain linear models and the two-level gapped array. The internal nodes only contain one pointer array, either to the data nodes or the internal nodes in the next level.

Gapped array. The TONE primary and secondary arrays are implemented as gapped arrays [8]. Gaps are spread throughout the arrays to accommodate for new entries.

Dynamic allocation of secondary array. Both the primary and secondary arrays store the key-value pairs and both arrays share the linear model. The secondary array is initialized with the same capacity as the primary array by default. A scaling factor may be applied on the secondary array to allow configurable array capacity. When a new data node is allocated, the primary array is automatically initialized. The secondary array is only allocated if the primary array hits the maximum threshold (i.e., a configuration parameter empirically set to 80% in our implementation). This dynamic allocation helps to save space if certain key space is not intensively updated.

Reducing structural modifications. The two-level leaf design eliminates the node expansion operation. When the primary array is full, TONE accepts new insertions in the secondary array. When the secondary array is at the maximum threshold, these two-level arrays (i.e. the primary and secondary array) will be merged back into a single large Gapped Array through an efficient *MergeTwoArrays* operation, shown in Algorithm 1, lines 18–27. Whether TONE retrains the linear model depends on the cost model evaluation and the number of keys in the new primary array.

The only expensive structural modification operation that can arise in TONE are node splits. If TONE decides to split the data node according to the cost model, it will merge the primary and secondary arrays before re-building its structure, to ensure consistency and accuracy. Even though this operation can be expensive, in Section 5 we show that splits almost never happen, even in write heavy workloads. The triggering of model retraining and node split shares the similar mechanism with ALEX.

Algorithm 1 TONE API

```
structure Node: {model; prim_array[], sec_array[] (Gapped
    Array); $d_l$; num_keys; num_sec_keys; is_sec_active}
1:  procedure WRITE(key, value)
2:      write_at_sec ← false.
3:      /* decide the target array */
4:      if num_keys > max_threshold then
5:          if is_sec_active = false then
6:              AllocateSecArray(size_ratio)
7:              write_at_sec ← true.
8:          else
9:              if num_sec_keys > sec_max_threshold then
10:                 if model is accurate then
11:                     MergeTwoArrays(false)
12:                 else
13:                     /* retrain or node split */
14:             else
15:                 write_at_sec ← true.
16:     WriteAtGappedArray(key, value, write_at_sec)
17:
18: procedure MERGETWOARRAYS(retrain)
19:     iterator ← GetIterator()
20:     new_capacity ← (num_keys+num_sec_keys)∗1/$d_l$
21:     /* allocate new primary array */
22:     merged_array ← array(size=new_capacity)
23:     new_model ← Expand(model, retrain)
24:     while iterator.hasNext() do
25:         key ← iterator.get()
26:         ModelBasedInsert(key)
27:     prim_array ← merged_array
28:
29: procedure GET(key)
30:     predicted_pos ← model.scaledPredict(key)
31:     if is_sec_active then
32:         actual_pos ← SearchAtSec(key, predicted_pos)
33:         if actual_pos is valid then
34:             return sec_array[actual_pos]
35:     /* search at the primary array */
```

4.3 TONE API

The TONE API consists of `reads`, `writes`, and `scans`. Algorithm 1 shows the pseudo-code for read and write operations, for a single thread. We omit the range scans and data partitioning pseudo-code for conciseness.

Writes. Algorithm 1 lines 1-16 provides detailed steps for the write operation at a data node. d_l indicates the minimum array density of the primary array (set to 60% in our implementation). A write can update an existing key, or insert a new key. TONE ensures that there are no duplicate key-value pairs in the index. Thus, if the requested key is already in the index, TONE performs an in-place update to replace the old value and returns directly. In case of an insert, a pre-allocated empty array slot in the gapped array is occupied by the new key, potentially shifting elements inside the leaf node to preserve ordering. Once the number of key-value pairs inside of a data node reaches the maximum threshold of the primary array (set to 80% in our implementation), TONE allocates a secondary array dynamically. After the allocation, TONE redirects the incoming insertions to the secondary array. By default, the newly allocated array has the same capacity and the expansion threshold as the primary array.

Reads. Algorithm 1 lines 29-35 provides detailed steps for the read operation at a data node. A read operation involves traversing from the root of the tree to the corresponding leaf, and then searching within the leaf node, if necessary. The traversing path from root to leaf node is decided by the hierarchy of models inside each internal node. After reaching the correct leaf node, the record in TONE can either reside in the primary array or, potentially in the secondary array. TONE checks whether the key exists in the secondary array at first by applying the linear model to predict the position and applying exponential search if needed. This is an optimization, since the secondary array typically has fewer, more recent entries. Finally, since the number of entries is typically lower in the secondary array, TONE applies a scaled prediction, using the same linear model.

Scans. TONE supports both sorted and unsorted results. Given a lower bound key and the upper bound key, TONE locates the leaf node that might contain the lower bound key. Then, TONE creates an iterator from the first key that is no less than the lower bound key. For a sorted scan, the iterator will be able to jump between the two level arrays to ensure the returned keys are in increasing order. For an unsorted run, instead of one iterator that handle both arrays, TONE creates two different iterators that deal with each array correspondingly. Each iterator will keep the key order within each array, however, the returned keys are not necessarily sorted. In the unsorted version of scan, we avoid the additional overhead of jumping between primary and secondary arrays. The iterators stop advancing when the current key is larger than the upper bound key.

4.4 TONE Parallelism

TONE employs a sharding-based mechanism. The dataset is statically partitioned into multiple index trees and each thread takes charge of a specific key space. Given that TONE is an ordered index, the key space can be easily divided and no duplicated key-value pairs can be written. The scan across multiple key spaces are decomposed into several smaller scans to retrieve all the records within the requested range in parallel. Minimal synchronization is necessary to build the scan result, if the scan coveres multiple partitions. We leave dynamic partitioning and load balancing strategies in TONE for future work.

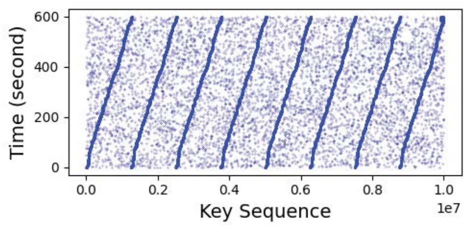

Figure 3. Evolving data access pattern under dynamic workloads. A blue dot represents a requested key.

5 Experimental Evaluation

We implement TONE in C++, as an extension of ALEX [8]. The TONE code is available at https://gitlab.cs.mcgill.ca/discs-lab/tone-cheops22. In the evaluation, we set out to answer the following questions:
(1) Tail Latency. Does TONE reduce tail latency in dynamic, write-heavy workloads?
(2) Steadiness. Does TONE avoid latency fluctuations?
(3) Throughput. Does TONE maintain good throughput in read-heavy workloads, in spite of its optimizations for write-intensive scenarios?
(4) Memory consumption. Does TONE maintain low memory consumption?

5.1 Experimental Setup

Hardware platform. We perform all measurements on an Ubuntu Linux server with Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz and 755GB RAM.

Competitor systems. We include three baseline competitors in our evaluation:
(1) ALEX: a write-optimized, single-threaded learned index, using a 1-level gapped array.
(2) XIndex: a concurrent learned index optimized for reads.
(3) RocksDB skiplist: as implemented in the RocksDB [1] memory component to offer a baseline comparison with an index used in a popular production system.
We set the maximum data node size to 32KB in all indexes. For the skiplist we use a maximum height of 12, as in the RocksDB default configuration.

Benchmarks. We use the following benchmarks:
(1) YCSB. The YCSB benchmark [7] is a popular choice for evaluating memory systems. It contains 6 default workloads: workloads A and F are write-heavy (50% read, 50% write and read-modify write respectively), workloads B and D are read-heavy (95% read, 5% write), workload C is read-only, and workload E is scan-dominated (95% scan, 5% write, with average scan size of 100 keys). We configure the benchmark with 8-byte (8B) keys and 8B values. Our value size configuration is different from the standard 1KB values for fairness, as the open-source implementation of XIndex [2] only supports 8B values. The requested keys follow a uniform distribution.

(2) Dynamic workload. To highlight TONE behavior under fluctuating conditions, we synthetically generate a dynamic workload inspired from our measurements in production systems at Nutanix. In the dynamic workload, 1% of the keys are accessed 99% of the time, according to a zipfian distribution. Moreover, the set of hot keys shifts over time, as illustrated in Figure 3. The keys are 8B and the values are 1KB. The read:write ratio is 1:1 and we initialize the index with 100K keys.

(3) Insert-only workload. To benchmark the memory consumption, we use an insert-only benchmark with 8B keys and 8B values. We vary the dataset size from 256MB to 8GB. The number of key-value pairs to insert is the quotient of the dataset size and the key-value pair size (16B).

In all experiments apart from the memory benchmarks and dynamic workloads, we initialize the index with 10M keys, selected uniformly, at random. For each experiment, we average the results over 5 runs. In the latency measurements, we log the tail latency (99^{th} percentile) every second. We use system-level monitoring commands (i.e., *top* and *htop*).

5.2 Results

5.2.1 YCSB Workloads

Figure 4 and Figure 5 show the throughput and the 99^{th} percentile latency respectively in YCSB with a single worker thread. We separate the multi-threaded and single-threaded scenarios, since ALEX does not provide a concurrency control mechanism. ALEX and TONE dominate performance across the board. In particular, in write-heavy workloads TONE outperforms ALEX by up to 11% in terms of overall throughput and outperforms XIndex and the skiplist by up to 2.6X and 3.8X respectively. Conversely, tail latency is low for TONE across the board, and on par with ALEX. This is expected, as the YCSB workloads are static. We will show how TONE outperforms ALEX in dynamic workloads in the following section.

Figure 6 and Figure 7 show the YCSB benchmark in a multithreaded scenario, with 8 worker threads. TONE's simple concurrency control allows it to scale across the board. In write-dominated workloads, TONE's throughput is up to 2x higher than that of XIndex, its best competitor. In read-dominated workloads, TONE is on par with XIndex, even though our system is not optimized for reads. Similarly, TONE shows low tail latency, in the order of a few microseconds, across all workloads.

In this section, we have shown that TONE performs well in static workloads. However, this is not where TONE shines. Our system is optimized for dynamic, write-intensive workloads, which we explore in the next section.

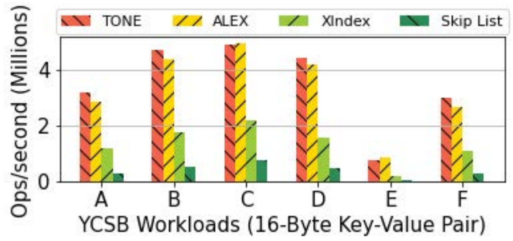

Figure 4. YCSB Throughput (single-thread). TONE outperforms or is on par with other indexes on all tasks.

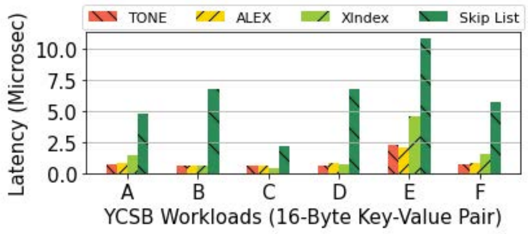

Figure 5. Overall latency on YCSB workloads (single-thread). TONE and ALEX exhibit low tail-latency for small KV pairs.

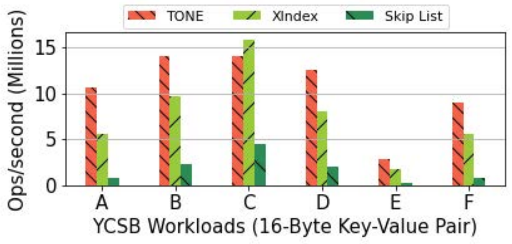

Figure 6. YCSB Throughput (8-threads). TONE outperforms XIndex and Skip List on the multi-threaded workload.

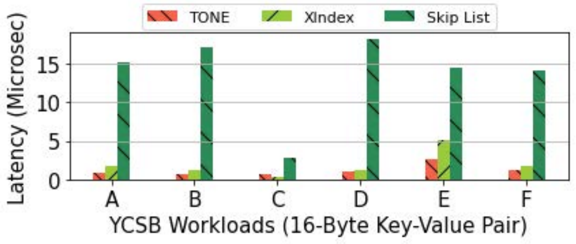

Figure 7. Overall latency in YCSB workloads with multi-threading (8-threads).

5.2.2 Dynamic Workload

Figure 8 shows the average throughput of ALEX and TONE in the dynamic workload (left-hand side), and the average tail latency (right-hand side). TONE reduces tail-latency up to one order of magnitude compared to ALEX, from 80 microseconds during ALEX's latency peaks, down to 6-8 microseconds. On average, TONE reduces tail latency by 56%.

Figure 9 shows a tail latency timeline, measured every second. ALEX struggles with the workload changes, showing significant latency spikes. In contrast, TONE's tail latency is low and steady, in both single- and multi-threaded scenarios. Whenever there is a shift of the hotness of the requested data, the tree is likely to grow rapidly within the hot key space. Since TONE does not need to do node expansions, TONE can absorb bursts of writes efficiently, which explains the low tail-latency. In the TONE 8-thread scenario, the tail latency is higher than in the single-threaded case. This latency increase is caused by slight contention on shared the resources (e.g., caches) among the worker threads.

On average, TONE needs 25% of structural modification operations compared to ALEX, under similar workloads. TONE exploits this design especially in workloads where a small key space is being queried in bursts. In this case, the array merging mechanism shines. Figure 10 shows that TONE achieves 58% lower write tail latency than ALEX by performing significantly fewer data movement operations in a write-heavy workload.

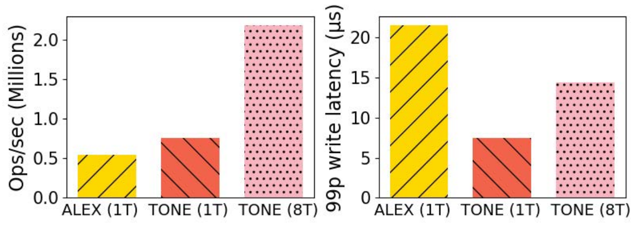

Figure 8. TONE and ALEX throughput (left) and average tail latency (right) in the dynamic workload. 1T and 8T represent single-thread and 8-threads, respectively.

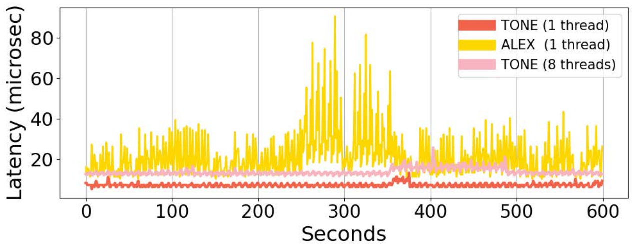

Figure 9. Write tail latency in the dynamic workload. TONE maintains low and steady tail latency of 6-8 microseconds, while ALEX can spike up to 80 microseconds.

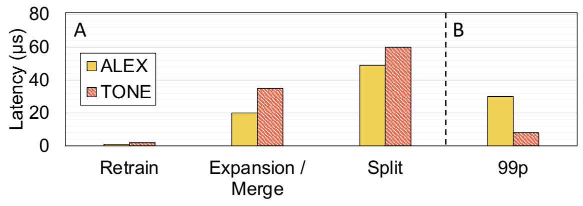

Figure 10. Write latency breakdown of TONE and ALEX. Part A shows the latency breakdown for the key data movement operations. Part B shows the 99p write tail latency. TONE removes node expansions completely, however it introduces merging which takes a similar time to expansion in ALEX. However, such operations occur rarely in TONE and thus do not affect the 99p latency.

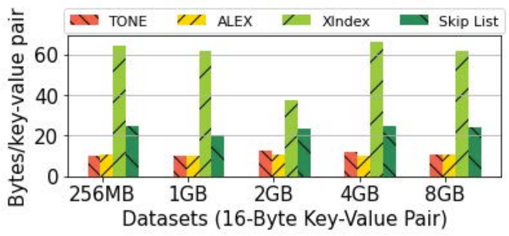

Figure 11. Memory footprint of indexes in an insert-only workload. TONE and ALEX have a similar, low memory footprint. They both outperform XIndex and Skiplist.

5.2.3 Memory Footprint

Figure 11 shows the memory overhead for datasets ranging from 256MB to 8GB. This experiment confirms that TONE maintains a memory-optimized structure. We note that TONE maintains a similar index size compared to ALEX across all datasets. They both outperform Xindex and Skiplist. On average, TONE achieves up to 6.34X and 2.56X lower memory consumption than Xindex and Skiplist, respectively. XIndex has a higher memory consumption due to its two-layer leaf node, out of which the top layer is exclusively dedicated to metadata. Furthermore, the skiplist has high memory overhead because the skiplist nodes may keep up to 12 pointers (i.e., the maximum height of the skiplist).

6 Related Work

Learned Indexes. ALEX [8] is the first updatable learned index using gapped arrays. Wu et al. [18] improve the prediction accuracy of index operations in LIPP. Flood [15] is a multi-dimensional learned index which improves throughput. XIndex [16] is designed to scale and to support concurrent updates. However, the above indexes mainly focus on improving memory consumption and lookup performance. In contrast, TONE focuses on optimizing tail latency. TONE's two-level leaf structure is general and can benefit the above-mentioned systems without affecting their primary improvements on learned index.

Tail Latency. Many designs have been proposed to alleviate high tail latency in various types of data stores. For instance, SILK [3] introduces an I/O bandwidth scheduler to reduce tail latency in key-value stores, and μTree [5] proposes latency optimizations in B+-trees. However, none of these design patterns are applied to the learned index, which is the main focus of TONE.

7 Conclusion and Future Directions

In this work, we first show that heavy structural modification operations such as data node expansions and splits are the main cause of tail latency in existing learned indexes. We then propose a two-layered gapped array design for leaf nodes, to mitigate such structural modifications. We integrate our novel leaf design in TONE, a learned index optimized for tail latency in write-heavy, dynamic workloads. Our experimental evaluation shows that TONE outperforms state-of-art learned indexes, reducing tail-latency up to one order of magnitude in write-heavy workloads. We believe that TONE is a promising step in making learned indexes practical. We are currently working on TONE integration into real datastores such as RocksDB's memory component or block cache, and using TONE as a cache replacement solution in an industry setting.

References

[1] Facebook. [n.d.]. RocksDB: A Persistent Key-value Store for Fast Storage Environments, 2019. https://rocksdb.org/.

[2] Xindex repository. https://ipads.se.sjtu.edu.cn:1312/opensource/xindex, 2021.

[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[4] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020.

[5] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *VLDB Endowment*, 13 (12), 2020.

[6] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2), 1979.

[7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[8] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[9] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *VLDB*, 13(8), 2020.

[10] Fedor V Fomin and Petteri Kaski. Exact exponential algorithms. *Communications of the ACM*, 56(3), 2013.

[11] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*, 2019.

[12] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[13] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. Apex: A high-performance learned index on persistent memory. *arXiv preprint arXiv:2105.00683*, 2021.

[14] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[15] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM*

SIGMOD International Conference on Management of Data, 2020.

[16] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. Xindex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020.

[17] Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. The concurrent learned indexes for multicore data storage. *ACM Transactions on Storage (TOS)*, 18(1), 2022.

[18] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *arXiv preprint arXiv:2104.05520*, 2021.

[19] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Characterizing facebook's memcached workload. *IEEE Internet Computing*, 18(2), 2013.