From LeanStore to LearnedStore: Using a Learned Index to Improve Database Index Search

Sujit Maharjan, Shuaihua Zhao, Chen Zhong, Song Jiang*
University of Texas at Arlington
Arlington, TX
{sxm5754,sxz6329,chen.zhong}@mavs.uta.edu, song.jiang@uta.edu

***Abstract*—In the realm of database systems, optimizing B+-tree index performance is of paramount importance to overall database performance. LeanStore, a high-performance OLTP storage engine, has extensively optimized its in-memory B+-tree component as well as its B+-tree-indexed database on the disk. However, B+-tree's lookup time increases linearly with the tree height. This is especially problematic when all or part of its lookup path is on the disk. Recently proposed learned index technique has the potential to significantly improve the performance of the B+-tree-based index by predicting location of the search key, instead of the level-by-level path walk. However, this machine-learning-model-based index has prediction errors that require a local search within the sorted keys. When the keys are on the disk, this search can be very expensive. The errors increase with write requests, which makes the search increasingly more expensive.**

We propose LearnedStore, in which the learned index technique is leveraged to improve the LeanStore database while the deterioration of learned-index's search performance is curbed. Instead of replacing the B+-tree index in LeanStore, LearnedStore opportunistically employs a learned index when it offers a performance advantage over the B+-tree index. Otherwise, it reverts to using the B+-tree index. By seamlessly integrating the learned index with LeanStore's B+-tree structure, LearnedStore takes advantage of the learned index while effectively addressing its challenges. We have implemented LearnedStore and extensively evaluated its performance. Experiment results show that LearnedStore improves throughput by up to 2.29 times for read-only workloads when the index and data set are all in memory. It reduces tail latency by up to 6.84 times when the index and data set are partially in the memory. Even when the entire index and data set are on the disk, LearnedStore can improve startup time by 3.05 times.

***Index Terms*—Learned Index Structures, Machine Learning, Indexing Techniques, Performance Optimization, key-value database, LeanStore, B+-tree**

I. Introduction

With the exponential growth of data, it has become imperative for database systems to efficiently manage a large volume of data. Databases serve as the backbone of data storage and analysis, enabling efficient storage, organization, and information retrieval. Indexing data structure plays a crucial role in achieving optimal query performance of the systems. Common database systems use tree-based data structures for indexing their data, such as B+-tree [1] and LSM tree [2]. The index structures are critical to the system's performance. However, the tree-based index requires key lookup at multiple levels of the tree to reach the leaf node containing the search key. As the size of the data set increases, there are more levels in the index, and more time is spent on the index during data access. Moreover, in a real-world database system, it is often impractical to always keep the entire database index in the memory due to the limited memory space. Some of the index components, such as some less-accessed inner nodes on the index tree, could be replaced out of the memory and be only available on the disk. In this case, a key lookup along an index tree path would incur disk I/O, making the search at the excessively low disk speed.

Learned index [3], like a hash index, can predict a range of key space where the search key can be found (a predicted key location with a tolerable error), thereby avoiding the need to traverse multiple levels of an index structure. Instead of relying on a predefined hash function, a learned index learns the mapping from keys to their positions in the array of sorted keys (the learned index model). When the predicted location is wrong, a local search on the sorted array within a predetermined error is conducted. If the error is small, this approach enables constant lookup time, similar to hash indexes, and supports range queries.

However, the learned index model's prediction error increases with the write operations, degrading the performance of the learned index. The model needs to be retrained so that the newly inserted keys can be accurately predicted. A number of updatable learned indexes, such as ALEX [4], XIndex [5], LIPP [6], FITing-tree [7], and PGM [8], have been proposed. However, when used for disk-resident indexes, these learned index designs often perform worse than the B+-tree index [9]. A fundamental reason is that search with a learned index is not deterministic. It uses a trained model to predict the location. This prediction becomes less accurate with the change of data set due to insert/delete operations. A misprediction leads to a search within a local key range defined by the model's error. This search of on-disk data may cause multiple disk accesses in the worst case. In contrast, the number of steps in the walk of a B+-tree is well bounded. When the learned index is unable to accurately locate the search key, the B+-tree index often performs better and should be used instead.

Furthermore, when the data set receives constant insertions and deletions, the learned model is required to be retrained to maintain its effectiveness. The retraining has to be infrequent as it is expensive due to its accessing the entire data set.

* Corresponding author.

979-8-3503-9416-0/23/31.00$2023 IEEE

The original learned index technique uses stochastic gradient descent for learning a model, which requires multiple passes through the data set [3]. This retraining is especially expensive with a large on-disk data set. RadixSpline [10] simplifies the training method by using a greedy spline generation algorithm. However, it still requires reading the entire data set for once. When a significant portion of the data set is on the disk, the training time can be excessively long, significantly consuming the disk bandwidth and leading to much increased high search latency during the retraining period [11].

LeanStore [12], a B+-tree-based key-value database, has been designed to optimize its in-memory index and data management, and reduce the number of disk I/O with an optimized management of the buffer cache. In this paper, we propose LearnedStore, which introduces the learned index technique into LeanStore to opportunistically accelerate search on its B+-tree index. By using the model-based index, LearnedStore improves in-memory performance and reduces the disk I/O by bypassing multiple levels in a tree-based index and directly reaching the leaf node that contains the search key. In the meantime, LearnedStore addresses the issues with the learned index by treating the model-based search as an opportunistic accelerator of the B+-tree index. By conditionally using learned index, LearnedStore can effectively address (or avoid) the issues with the method of relying on the learned index as its sole index structure.

This paper makes the following contributions.

- We examine the performance improvement potentials in terms of lookup time, tail latency, and database startup time when using a learned index in a database in various scenarios, including its index being all in the memory, partially in the memory, or entirely on the disk.
- We design LearnedStore that can adapt its search dynamically between learned index and the B+-tree index to maximize the performance.
- We implement and extensively evaluate LearnedStore with various workloads. The experiment results show that LearnedStore has up to 2.28 times improvement for read-only workloads. Further, the tail latency can be reduced by 6.8 times when the index and data set are partially in memory for a read-only workload. Experiments also show that LearnedStore achieves 1.13 times, 1.18 times and 1.31 times throughput improvement for 100% insert, 50% insert, and 5% insert workloads, respectively.

II. Background and Related Work

In this section, we first describe the model structure of learned index and how it learns the mapping function. Then we introduce LeanStore and some of its key techniques that make it one of the state-of-the-art B+-tree-based key-value databases and the reason why we choose it as a baseline B+-tree database to develop our LearnedStore system.

A. Learned Index

Learned index [3], unlike traditional index structures, learns a function (or a model) that maps a key to its location in a sorted array. The learned index's model emulates the Cumulative Distribution Function (CDF) of the data set with some misprediction error. Recursive Model Index (RMI) [3], a simplified version of a neural network that consists of a hierarchy of simple linear regression models, was proposed to remove the neural network model's inference overhead. However, RMI is trained through a stochastic gradient descent algorithm, which requires multiple passes through the data set to train the model. It also requires trial-and-error hyper-parameter tuning and an additional error evaluation cycle to find the maximum misprediction error of the trained model by examining the error of all the keys in the data set.

TABLE I: Time to iterate through the randomgen data set (31GB, see Table II) on the disk for generating linear splines and the deterioration rate over the single-pass in-memory time.

	Time (s)	**Deterioration by**
Single pass in memory	5	
Double pass in memory	10	2x
Single pass on disk	931	186x

RadixSpline [10] simplifies the model further and develops the model as a collection of linear splines approximating the CDF curve with some acceptable error. RadixSpline's model inference consists of a radix table lookup to find a set of candidate linear splines followed with a binary search in the candidate set to find the correct spline and finally a prediction using the spline. Unlike RMI, RadixSpline generates a collection of linear splines after a single pass through the data set using a greedy spline generation algorithm [13] to represent the mapping function with a predetermined acceptable error. Furthermore, it reduces the number of hyperparameters to just two (the number of radix bits and maximum error) to ease the tuning process. Although it considerably reduces the training time of the learned index, it is still not practical for on-disk data sets as even a single pass over the data set on the disk is slower than that in the memory by orders of magnitude, as revealed in our experiment results shown in Table I.

RMI and RadixSpline do not support write operations. Some updatable learned index models, including ALEX [4], XIndex [5], LIPP [6], FITing-tree [7], PGM [8], use delta index or gaps in the array to support insertion. However, they assume that all the data sets will be in the memory, and the nodes are of variable size. Therefore, page-sized buffer cache management in LeanStore, including its replacement policy, cannot be directly used to manage the index nodes. An experiment-based study has found that for on-disk data sets these updatable learned index models are not competitive with the B+-tree index [9].

B. LeanStore

LeanStore [12] is a B+-tree-based key-value database that manages a B+-tree index across the memory and the disk. It can achieve in-memory B+-tree performance when the data set fits in the memory. A basic method for creating a B+-tree index for a large data set involves converting nodes to disk blocks and accessing disk blocks during the tree traversal.

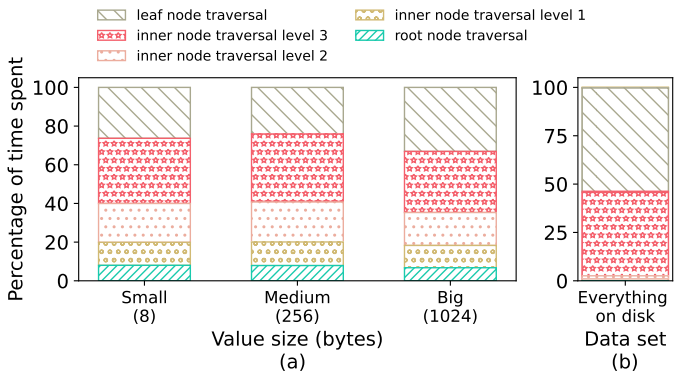

Fig. 1: Breakdown of LeanStore's lookup latency in two scenarios about the index placement: (a) all in-memory and (b) all on-disk

It has been found that relying on the OS page cache for loading the page in the memory is inefficient [14]. Instead, a database system usually bypasses the file system for disk access and has its own buffer manager for data pages in the buffer pool. The buffer manager provides buffer search, allocation of page frame, and page replacement functionalities. Even though the buffer manager uses a hash table to efficiently search the buffer frame associated with a page ID, it still adds considerable overhead as buffer search is required at every level of the B+-tree when it traverses from the root to the leaf node. This substantially increases the search time.

LeanStore optimizes the in-memory B+-tree search performance by using the pointer swizzling technique [15] that allows efficient traversal of B+-tree nodes that have been loaded into the buffer pool without using a buffer frame hash table. Additionally, it optimizes its page replacement policy. This makes LeanStore a state-of-the-art B+-tree-based key-value database for the cases where the data set and index are either completely or partially cached in the main memory. Accordingly, we choose it as the base system on which LearnedStore is developed.

III. Learned Index for LeanStore

In this section, we investigate potential performance improvements that can be achieved by applying the learned index technique. This provides a guideline for the design of LearnedStore.

In order to gauge the potential benefit of directly reaching a leaf node with a model-based prediction, we first measure the time spent on each of the B+-tree levels in a LeanStore's lookup operation. We then analyze LeanStore's lookup latency for the following scenarios: (a) the in-memory case, where the entire data set can fit in the memory under different value sizes (8, 256, or 1024 bytes); and (b) when the entire data set resides on disk.

We generate a B+-tree of five levels by inserting 500 million 8-byte keys for small 8-byte values, 40 million 8-byte keys for medium 256-byte values, or 20 million keys for large 1024-byte values for the in-memory cases. For the all-on-disk case,

TABLE II: Description of data sets used

Data set	Description
lineargen	data set consisting of 200 million 8-byte key with a 64-byte value whose keys are generated using a linear function
piecewise	data set consisting of 200 million 8-byte key with a 64-byte value whose keys are generated using a number of linear function
randomgen	data set consisting of a 200 million 8-byte key with a 64-byte value generated from a random distribution
amzn64	book sale popularity data from SOSD [16]. Consists of a 200 million 8-byte key with a 64-byte value
face64	unsampled version of Facebook user ID data set from SOSD [16]. Consist of a 200 million 8-byte key with a 64-byte value
logn64	key sampled from logn distribution from SOSD [16]. Consist of a 200 million 8-byte key with a 64-byte value
norm64	key sampled from norm distribution from SOSD [16]. Consist of a 200 million 8-byte key with a 64-byte value
largerandomgen	200 million randomly generated unsigned integer keys of size 8 bytes with randomly generated values of size 512 bytes which occupies 194 GB on disk.

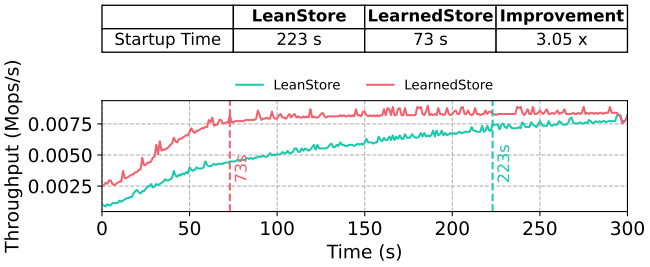

Fig. 2: Comparison of throughput of LeanStore and LearnedStore with the largerandomgen data set. The lower graph shows increasing throughput starting the lookup service with a cold buffer pool.

we create a larger data set with 200 million 8-byte keys and 512-byte values (the largerandomgen data set in Table II). For the in-memory case, we measure the average time spent on each level of the tree when all the keys are looked up. For the all-on-disk case, we measure the latency breakdown during a 20-second service of lookup requests, assuming that the entire B+-tree initially resides on the disk.

The experiment results are shown in Figure 1. As shown, over 60% of the lookup latency is spent on inner node traversal for the in-memory case. For the all-on-disk case, 50% of the time is spent on the inner nodes in the third level, which are less likely to be all cached in the buffer pool.

In addition to average lookup latency, startup time and tail latency are two additional important metrics that are critical for real-time and mission-critical applications, where consistent performance and availability are essential. They measure a system's ability to quickly resume its execution (after a restart or recovery from a failure) and performance outliers, respectively. LeanStore uses a buffer pool to optimize its performance. Indexes are known to take a significant amount of memory, and it can take a long time to load them into the buffer pool.

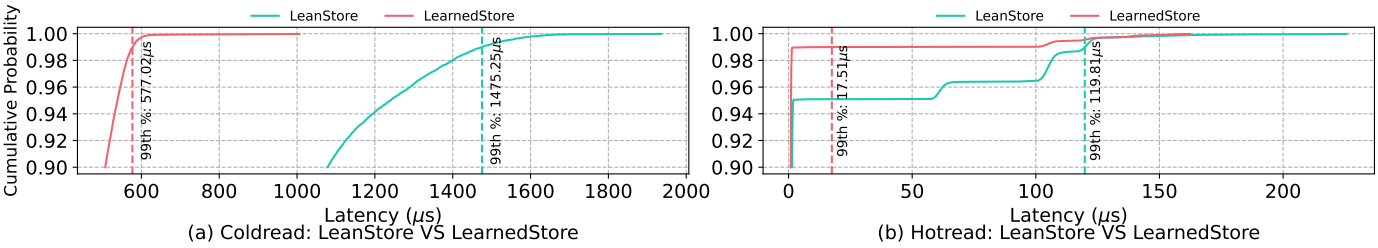

Fig. 3: CDF curve of lookup latency for LeanStore and LearnedStore for (a) cold buffer pool read for 20 seconds and (b) warm buffer pool with 1% index nodes missing.

During the startup phase or while accessing the cold index, the database has to load the indexes from the disk to service queries. As a result, users may experience longer waiting time and slow system responsiveness. Specifically, we define the startup time as the time required for the system to reach 90% of the peak performance when the index is initially all on the disk. To assess the startup time we prepare a data set of 200 million key-value pairs, each of 8-byte key and 512-byte value (the largerandomgen data set described in Table II). The data set is of size 194GB and the buffer pool is of size 100GB. As shown in Figure 2, the startup time of LeanStore is 223 seconds. This is a long time period during which most of the I/O bandwidth is used to warm up the buffer pool and only very limited bandwidth is available for servicing front-end queries. Learned index could reach the leaf nodes immediately without waiting for the inner nodes to be loaded, thus reducing the startup time.

LeanStore's buffer pool is designed to cache as many pages/nodes in the memory as possible. However, LeanStore cannot guarantee that all the nodes are in the buffer pool. Some inner nodes may be evicted by the background page-replacement thread. This leads to a lookup on a B+-tree path whose nodes are not all in the memory. Lookup on the 'bumpy' path causes long latency as it needs to load node(s) from the disk. While this may not happen often on a sufficiently large memory, it does cause long tail latency and compromises user experience. To assess its impact on the tail latency, we create a 200 million 8-byte key with a 512-byte value data set (the largerandomgen data set described in Table II) and experiment with two setups. In the first setup, the buffer pool is initially empty. We issue 20 seconds of random lookups of different keys. As shown in Figure 3, the 99th percentile latency of LeanStore is 1475.25 μs, which is 2.27 times of the average latency (647.77 μs). In the second setup, we make the inner nodes out of memory with a probability of 0.01. That is, the buffer pool has cached almost all the index and the data set. As shown in Figure 3, the 99th percentile latency of the LeanStore was 119.8 μs, much higher than the average latency (6.09 μs). Learned index can skip the bumpy path traversal and improve the tail latency.

IV. The Design of LearnedStore

In this section, we present LearnedStore's design to realize the performance improvement potential enabled by learned index. LearnedStore uses learned index to bypass inner nodes during index lookups. In the meantime, it keeps the existing B+-tree index of LeanStore intact for lookups with high local search costs in the learned index.

A. The Training Phase

Instead of predicting the location of the search key in the sorted array [3], LearnedStore predicts the location of the B+-tree leaf node containing the key. This is because the B+-tree organizes key-value pairs in pages in the memory and in blocks on the disk. As the disk is a block device, the prediction of a search key's location within a block does not help with the access efficiency. More importantly, a learned index model becomes less accurate with updates in its indexed data. With insert/delete requests, the positions of key-value pairs in a block may frequently change. However, the change of a leaf node (via node split/merge) is much less frequent. Predicting the leaf nodes helps maintain the accuracy of the learned model for a longer time.

To this end, we use the largest key of a leaf node, which is also the boundary key in the leaf node's parent node, to represent the leaf node in the learned index. This also helps reduce training time as all the leaf node pages do not need to be loaded in the buffer pool during the training phase. We only need access to the parent nodes of the leaf nodes to obtain all the maximum keys of the leaf node. As the index (except the leaf nodes where key-value pairs are stored) is most likely in the memory, a training phase can therefore almost avoid disk access. To enable the training over the maximum keys, we order and store the maximum keys in an array where each slot of the array is mapped to a leaf node. This array is named *leaf node mapping table* and is always resident in the memory. The leaf node mapping table can be populated with keys (one boundary key per leaf node) either directly from the leaf nodes or from their parent nodes. Because each parent node stores multiple (around 60-80) boundary keys together and it's almost always in the memory, it is more efficient to obtain keys from the parent nodes (especially when leaf nodes are on the disk). As shown in Table III, using boundary keys of inner nodes to represent the leaf nodes for training reduces the training time by about 4.5 times when the leaf nodes are in the memory and about 62 times when they are on the disk. Therefore, LearnedStore always uses the parent nodes to populate the mapping table.

LearnedStore uses a list of linear spline segments to represent the CDF of the maximum keys of all the leaf nodes using

TABLE III: Training time of LearnedStore when retrieving keys from the leaf node vs. from the boundary key of the leaf node's parent into the leaf node mapping table.

Training	With leaf node	With boundary key	Improvement
In-memory	0.28 (s)	0.062 (s)	4.5 times
On-disk	915.17 (s)	14.71 (s)	62 times

the Greedy Spline Corridor algorithm [13] with a maximum error threshold as its hyperparameter. This enables us to find linear segments within the data set and controls the error correction overhead using a single max error hyperparameter. This further aids in reducing training time as we can easily train the model on a single pass through the data set and does not require a time-consuming hyperparameter tuning. LearnedStore also uses a simple linear regression model on leaf nodes to predict the location of keys within the leaf node. In order to avoid any disk access while training the linear regression model at the leaf node, LearnedStore only trains the model for the leaf node resident in the buffer pool.

B. The Index Lookup

LearnedStore first tries to use the learned index to directly reach the leaf node. To this end, it looks for the correct spline segment for the search key with a binary search. The model for the spline segment is then used to determine the leaf node ID (a prediction of the target leaf node), which points to a slot in the leaf node mapping table. Each slot stores the page ID of a leaf node, a pointer to the page frame in the buffer pool, and the maximum key in the leaf node. Note that the prediction of the learned index model, which is a leaf node ID corresponding to a slot in the leaf node mapping table, may not be accurate. If the search key is not in the range covered by the slot (by checking the maximum keys in this and its previous slot), an exponential search within an error bound is performed on the table to find the correct slot.

While the pointers in the table to the buffer pool frames can be used to conveniently reach the page frames in the buffer pool, keeping them up to date can be expensive. A leaf node can be split into two leaf nodes or merge with another leaf node. Accordingly, its pointed page frame may store different set of keys. Furthermore, a page frame may be reclaimed for caching another leaf node when its page is replaced out of the memory. It's expensive to keep monitoring the page frame changes and conduct real-time updates of the pointers.

Our solution is to use a lazy update technique, in which the pointers are not immediately updated. We know that the pointers are always valid because the page frames they point to stay in the buffer. However, these frames can be assigned to store a different leaf node. When LearnedStore follows a pointer to obtain a page frame, it needs to compare the page ID associated with the pointer in the leaf node mapping table to the page ID in the page frame. If they do not match, we resort to the buffer frame hash table to find the page frame in the buffer pool. As the leaf node split/merge operations are much less frequent than key insert/delete, most of the time the correct page frames can be quickly found via the leaf node

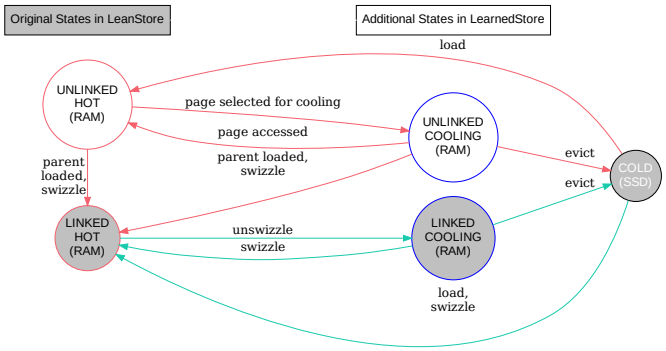

Fig. 4: Possible states of a page

mapping table without using the hash table. For queries that can be serviced all in the memory, this short-cut access further reduces the time of reaching the search key.

Because key-value pairs in a leaf node may be updated, LearnedStore needs to further check if the search key is between the smallest and the largest keys of the leaf node to determine if LearnedStore reaches the correct leaf node. The most likely scenario is that the leaf node is in the buffer pool and the search key is in the range, and the learned index helps quickly find the search key. However, there are two cases that challenge the use of learned index. In the first case, the leaf node is in the memory, but the search key isn't in its range. In this case, we can resort to the LeanStore's B+-tree index to carry out the lookup operation. This design choice reveals one of the LearnedStore's unique advantages of being a hybrid index. In contrast, other updatable learned index designs would have to rely on complicated and expensive strategies to handle the consequences of model misprediction. The second case is a more subtle one in which the leaf node is not yet in the buffer pool. A straightforward approach is to load it from the disk and then check if the key is in the range. However, if the key is not in the range, a wrong leaf node would be loaded and an unnecessary and costly I/O operation was conducted. The impact of loading a wrong leaf node is too high on the lookup performance. To address this issue, LearnedStore retains the largest and smallest keys of a leaf node evicted to the disk in memory. This metadata is used to promptly verify whether the search key falls within this range, and the B+-tree is employed if the key is not in the range. As the B+-tree index always leads the lookup to the correct leaf node, there won't be any unnecessary I/O. The pointer in the leaf node mapping table is updated if the learned index predicts the correct leaf node and it is loaded from the disk.

LearnedStore uses the leaf node's simple linear regression model to predict the location of the key within the leaf node. Exponential search is used to find the correct position of the key if the initial prediction is incorrect.

C. Page Eviction

LeanStore uses a page eviction algorithm. Rather than immediately evicting the buffer frame from the memory, the page is instead moved to the cooling queue (from the HOT

state to the COOLING state), as shown in Figure 4. This allows the buffer pool to maintain a higher hit rate, as the page can still be accessed from the memory. Once a page has been in the cooling queue for a certain amount of time, it can then be safely evicted from the buffer pool. This process reduces the overhead of the page eviction process by allowing pages to remain in memory for a longer period of time and decreasing the frequency of page evictions, which can be resource-intensive and time-consuming.

LeanStore cannot load the leaf node directly without loading the parent node as every node/page in the buffer pool must have a single owning pointer (swip) in the buffer pool. This is a major hurdle for LearnedStore as it prevents it from using its learned index to reach and load the leaf node. Therefore, LearnedStore modifies the page eviction algorithm to track nodes/pages that do not have parents loaded in the buffer pool and designate them as in the "UNLINKED" state. Those with their parent nodes in the buffer pool are in the "LINKED" state. Similar to LeanStore, recently loaded nodes are kept in the HOT state, and a certain percentage of the HOT nodes/pages are transferred to the COOLING state. However, LearnedStore independently tracks whether the parent node is loaded and whether a swip needs to be unswizzled (swip associated with a page ID) using a separate state. Moreover, the pages in the UNLINKED state do not require the swip in the parent node to be unswizzled and thus can be more efficiently evicted to the disk.

D. Serving Write Requests

Serving a write request starts with an index lookup to find the correct leaf node as just described. Insertion/deletion of a key-value pair into a leaf node is the same as that of LeanStore. However, when insertion/deletion causes leaf node split/merge, it not only causes adjustment of the B+-tree index but also leads to increasingly more mispredictions for the learned index model. LearnedStore uses two background threads to address the issue. One retrains the learned index model used for leaf node prediction, while the other trains the linear regression model within individual leaf nodes. LearnedStore tracks the number of leaf node mispredictions and the number of leaf node splits/merges. It computes the ratio of the two numbers each time when the learned index is used. The ratio quantifies the impact of leaf node splits/merges on the model prediction accuracy. When the ratio is greater than a misprediction tolerance threshold, the background threads are activated for retaining.

V. Evaluation

To understand the performance of LearnedStore, we first evaluate it with a read-only workload and then with a workload with writes. We randomize the sequence in which the keys are read or written so that the measurement does not overfit a specific access pattern. We use various synthetic and real-world data sets for the experiments, as shown in Table II. We use Intel Xeon CPU E5-2683 v4 with 220 GB memory and 458 GB SATA SSD, with the execution pinned to a specific NUMA node, to measure the performance metrics.

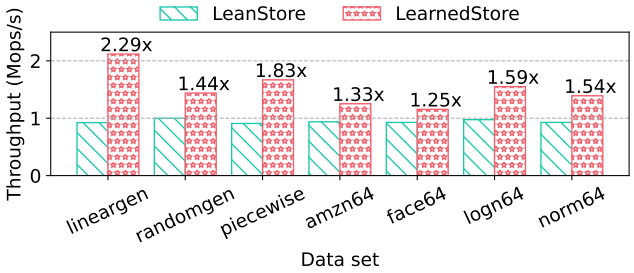

Fig. 5: Lookup throughput of LeanStore and LearnedStore for the in-memory case

A. Read-only Workload

As described in Section III, we evaluate the read-only throughput in three cases: when the index and data set are entirely in the memory; partially in memory and partially on the disk, or entirely on the disk. The three different scenarios have different numbers of disk access during a read-only lookup and thus different performance results.

1) The All-in-memory Case: We run LeanStore and LearnedStore under various data sets that can fit in a 100GB buffer pool to measure the throughput for the in-memory case. As shown in Figure 5, LearnedStore achieves up to 2.29 times throughput improvement for the lineargen data set which consists of a 200 million 8-byte key with a 512-byte value, as described in Table II, while the lowest improvement (1.25 times) is seen in face64 [16]. The performance improvements are almost proportional to the accuracy of the model's predictions. Among the data sets, lineargen is the easiest to learn for a highly accurate model and hence produces the highest performance improvement.

2) The Case of Partially in Memory and Partially on the Disk: To assess the LearnedStore's performance when the data set is large enough to be only partially in the memory, we assume that a tree node could be out of memory with a probability of 1% and the system experiences a sequence of reads with a hot buffer pool, as described in Section III. We use the rangdomgen data set in Table II with a 200 million KV pairs (8-byte key and a 64-byte) to measure the latency of each lookup when the buffer pool has cached almost all the pages. We assume that the system experiences a 300-second burst of random lookup of existing keys. As shown in Figure 3, the 99th percentile latencies are 119.81μs and 17.5μs for LeanStore and LearnedStores, respectively. Thus, our approach was able to improve the 99 percentile latency by 6.84 times. Meanwhile, the average latency of LearnedStore is measured to be 2.03μs, which is 2.99 times LearnedStore's average latency of 6.09μs, as shown in Figure 6.

These results show that with only 1% of nodes out of memory, the 99th percentile tail latency is way higher the

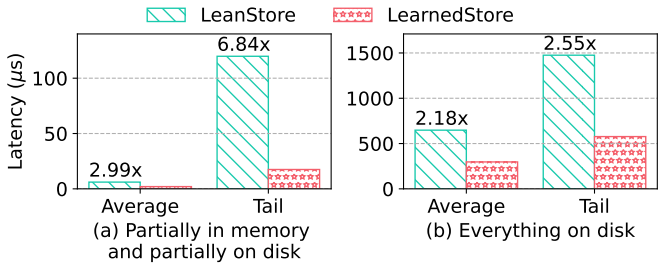

Fig. 6: Average and tail latency of LeanStore and LearnedStore under the read workload when (a) the data set and index are partially in memory and partially on disk and (b) all is on disk

TABLE IV: Throughput of LeanStore and LearnedStore for the write workload

Experiment	LeanStore (MOPS/s)	LearnedStore (MOPS/s)	Improvement
100% Write	0.6397	0.7273	1.13 times
YCSB-A	0.7374	0.8760	1.18 times
YCSB-B	0.9035	1.1838	1.31 times

average latency. LearnedStore can avoid almost all the non-leaf-node disk accesses and thus deliver more consistent performance. In the case, the number of disk access that can be saved determines the amount of improvement of the tail latency by LearnedStore.

3) The On-disk Case: To demonstrate the performance of LearnedStore when the data set and its index are initially on the disk, we use the largerandomgen data set with 200 million KV pairs (8-byte key and a 512-byte value) to measure the throughput and latency of the LeanStore and LearnedStore starting with a cold buffer pool of 100 GB.

LearnedStore attempts to directly reach the leaf node, avoiding the need to load inner nodes. This significantly reduces the startup time and improves the system's overall performance. As shown in Figure 2, the time at which LearnedStore first achieves 90% of its maximum throughput is around 73 seconds, while the time at which LeanStore achieves 90% of its maximum throughput is around 223 seconds. That is, LearnedStore reduces the startup time by 3.05 times.

We also measure the latency of each lookup in a workload of a 20-second random lookup of existing keys on the disk. As shown in Figure 3, the 99th percentile latency of the LeanStore is 1475.25μs while LearnedStore's is 577.02μs. LearnedStore improves the 99th percentile latency by 2.55 times. Meanwhile, the average latency of the LeanStore is 647.77μs, while the average latency of LearnedStore is 297.11μs, which is 2.18 times the average latency of the LeanStore (as illustrated in Figure 6).

The results indicate that LearnedStore can efficiently handle lookup requests without loading the inner nodes. The high startup time of LeanStore demonstrates that the number of inner nodes that are required to be loaded can be significant and becomes a major hurdle for achieving high throughput when a relatively cold index is being accessed.

B. Workload with Writes

We further evaluate LearnedStore when it is subjected to 100% write workload and then on a mixed workload with 50% write (YCSB-A [17]) and 5% write (YCSB-B [17]). We start with a B+-tree consisting of 100 million randomly generated KV pairs (4-byte keys and 64-byte values). Then, we experiment with the mixed workloads until another 100 million randomly generated keys written.

As shown in Figure 7 and Table IV, LearnedStore achieves 1.13 times improvement for 100% insert, 1.18 times improvement for YCSB-A workload, and 1.31 times improvement for YCSB-B workload. The experiment shows that by reaching directly the leaf nodes during the write workload, the speed of write operations can be substantially increased. With more intensive writes the improvement becomes smaller as more frequent background model retraining is required. However, even with the 100%-write workload, there is still 13% improvement. This is because most writes do not cause any structural changes in the index. As the model is built on the leaf nodes, rather than on individual keys, the impact of writes on the model prediction accuracy is alleviated.

VI. More Related Work

There have been many works that apply the learned index technique to accelerate data search. Bourbon [18] uses learned indexes to accelerate the LSM-tree-based KV store [19]. In particular, Bourbon accelerates the KV database Wiskey [20] that organizes the data in the sorted order instead of B+-tree. It modifies the immutable SSTable Index of BigTable [21] to use a simple linear regression model with model-based writes so that the model can accurately predict the correct block.

AULID [22] and FILM [23] explore the use of learned index instead of B+-tree-based index in the disk-resident database. AULID proposes a learned index-based index structure as an alternative to the B+-tree. However, AULID assumes that all data is on the disk and does not consider a buffer pool in its design. FILM also replaces the B+-tree index with the learned index that is designed for append-only workloads but supports out-of-order writes in the leaf nodes. Unlike LearnedStore which has a fixed size leaf node equal to the block size of the disk, the leaf node of FILM is of variable size. FILM's buffer manager maintains an LRU list over all the existing records rather than leaf pages and evicts key-value pairs instead of leaf nodes. Thus, FILM's buffer manager has more implementation overhead. Llaveshi et al. propose to use machine learning model to accelerate B+-tree traversal [24]. However, its model-based prediction is limited to selection of a child node in the next level, rather than directly a leaf node. It also assumes that the data set is all in the memory. Therefore, its performance improvement potential is very limited.

VII. Conclusions

This paper explores the performance potential of using the learned index technique to accelerate disk-based B+-tree-indexed key-value databases. Our investigation revealed that the most time-consuming aspect of the lookup operation

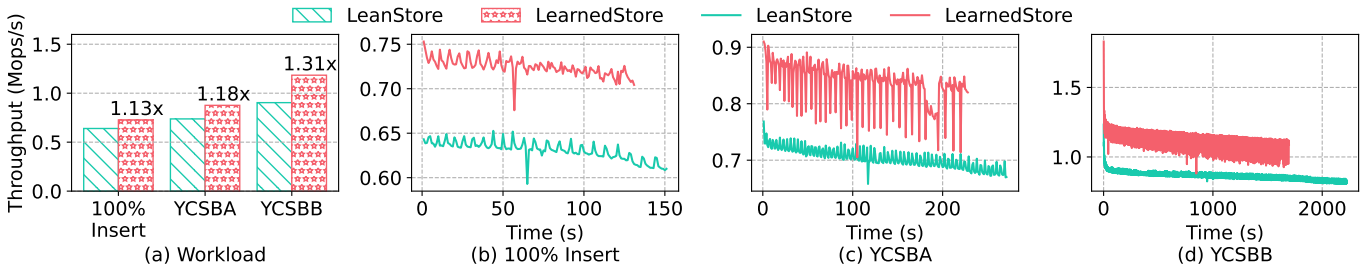

Fig. 7: Performance of LeanStore and LearnedStore with the write workload

occurs when traversing inner nodes, particularly when some of the B+-tree nodes are not in the memory, whereas disk I/O becomes the primary bottleneck. We designed and implemented LearnedStore on top of a high-performance B+-tree-based database, LeanStore. Instead of using learned index to replace the B+-tree index, LearnedStore adds learned index as an accelerator to the B+-tree index. In this way, the B+-tree index becomes faster and the shortcomings of learned index with index updates can be overcome by the B+-tree index. Our experiments demonstrate that LearnedStore can yield an impressive improvement of up to 2.29 times in scenarios where the data set resides in the memory while delivering a remarkable 6.84 times reduction in tail latency for data sets that do not fit in the memory. Further, we introduced an auto-training method for LearnedStore to have consistent performance improvements even with write workloads.

References

[1] D. Comer, "Ubiquitous B-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979, publisher: ACM New York, NY, USA.

[2] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996, publisher: Springer.

[3] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[4] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, and D. Kossmann, "ALEX: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969–984.

[5] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. 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, pp. 308–320.

[6] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *arXiv preprint arXiv:2104.05520*, 2021.

[7] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 1189–1206.

[8] P. Ferragina and G. Vinciguerra, "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, 2020, publisher: VLDB Endowment.

[9] H. Lan, Z. Bao, J. S. Culpepper, and R. Borovica-Gajic, "Updatable Learned Indexes Meet Disk-Resident DBMS-From Evaluations to Design Choices," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–22, 2023, publisher: ACM New York, NY, USA.

[10] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "RadixSpline: a single-pass learned index," in *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management*, 2020, pp. 1–5.

[11] Y. Zhang, X. Xiong, and O. Balmau, "TONE: cutting tail-latency in learned indexes," in *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, ser. CHEOPS '22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 16–23. [Online]. Available: https://dl.acm.org/doi/10.1145/3503646.3524295

[12] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann, "LeanStore: In-memory data management beyond main memory," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 185–196.

[13] T. Neumann and S. Michel, "Smooth interpolating histograms with error guarantees," in *Sharing Data, Information and Knowledge: 25th British National Conference on Databases, BNCOD 25, Cardiff, UK, July 7-10, 2008. Proceedings 25*. Springer, 2008, pp. 126–138.

[14] M. Stonebraker, "Operating system support for database management," *Communications of the ACM*, vol. 24, no. 7, pp. 412–418, Jul. 1981. [Online]. Available: https://dl.acm.org/doi/10.1145/358699.358703

[15] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch, "In-Memory Performance for Big Data," *Proceedings of the VLDB Endowment*, vol. 8, pp. 37–48, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2735465

[16] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "SOSD: A benchmark for learned indexes," *arXiv preprint arXiv:1911.13014*, 2019.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. Indianapolis Indiana USA: ACM, Jun. 2010, pp. 143–154. [Online]. Available: https://dl.acm.org/doi/10.1145/1807128.1807152

[18] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "From {WiscKey} to Bourbon: A Learned Index for {Log-Structured} Merge Trees," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 155–171.

[19] H. Abu-Libdeh, D. Altınbüken, A. Beutel, E. H. Chi, L. Doshi, T. Kraska, A. Ly, and C. Olston, "Learned indexes for a google-scale disk-based database," *arXiv preprint arXiv:2012.12501*, 2020.

[20] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017, publisher: ACM New York, NY, USA.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008, publisher: ACM New York, NY, USA.

[22] H. Lan, Z. Bao, J. S. Culpepper, R. Borovica-Gajic, and Y. Dong, "A Simple Yet High-Performing On-disk Learned Index: Can We Have Our Cake and Eat it Too?" *arXiv preprint arXiv:2306.02604*, 2023.

[23] C. Ma, X. Yu, Y. Li, X. Meng, and A. Maoliniyazi, "FILM: A Fully Learned Index for Larger-Than-Memory Databases," *Proceedings of the VLDB Endowment*, vol. 16, no. 3, pp. 561–573, Nov. 2022. [Online]. Available: https://dl.acm.org/doi/10.14778/3570690.3570704

[24] A. Llaveshi, U. Sirin, A. Ailamaki, and R. West, "Accelerating B+ tree search by using simple machine learning techniques," in *Proceedings of the 1st International Workshop on Applied AI for Database Systems and Applications*, 2019.