LASER: Line-Aware and Self-Balancing Learned Index for Rapid Key Lookup

Shuaihua Zhao, Jian Zhou, Song Jiang
University of Texas at Arlington, Arlington, USA
sxz6329@mavs.uta.edu, jxz9486@mavs.uta.edu, song.jiang@uta.edu

***Abstract*—Learned indexes have received significant attention for their potential to dramatically outperform traditional tree-based indexes in both speed and space efficiency. Their core strength lies in using predictive models to estimate the position of a key within a sorted array. To handle complex key distributions and support frequent insertions in dynamic workloads, learned indexes typically organize multiple models hierarchically in a tree structure. These indexes perform best when a model can accurately predict a key's location. However, existing learned indexes often require traversing several models to reach the one responsible for a target key. Moreover, if the prediction is imprecise, an additional local search (the last-mile search) is needed. These overheads before and after model execution can significantly degrade performance, sometimes approaching that of conventional indexes like B+-trees.**

In this paper, we propose LASER, a new learned index design that tackles these inefficiencies by leveraging two common patterns in real-world workloads. First, LASER exploits key access locality, where some key ranges are more likely to be accessed. Second, it detects linear key distributions, enabling precise prediction without last-mile search. LASER adaptively promotes models covering long key ranges to the top of the model tree, reducing the number of model traversals. For key ranges with perfectly linear distributions, it employs models that guarantee a direct hit (on the exact key position), eliminating the need for further search. We implemented LASER and conducted extensive evaluations. The results show that LASER outperforms state-of-the-art learned indexes such as LIPP and ALEX, as well as traditional indexes, like ART, by up to 1.6 to 5.5 times.

***Index Terms*—learned index, line aware.**

I. Introduction

Learned index has attracted considerable attention in the field of data management systems [1]. This indexing technique enables the construction of O(1) indexes by predicting the position of a key within a sorted array. Although hash tables also compute key locations to achieve O(1) lookup efficiency, learned index offers additional capabilities by maintaining a sorted key array, thereby supporting range queries. By preserving the sorted order of key-value (KV) pairs, learned index has the potential to replace traditional comparison-based structures such as B+-trees, which typically operate in O(lgn) time. Recent research has extended learned indexes to support insertions and updates [2]–[5].

A. Hierarchical Model Structure and its Performance Impact

The effectiveness of a learned index largely depends on the quality of the model(s)—or simply function(s)—used to predict key locations. The quality is typically measured by the accuracy of the model in mapping a key to its position in the sorted key array. Constructing a single model that is both accurate and efficient across millions of keys with diverse distributions is often impractical. To address this, a common approach is to build multiple specialized models, each responsible for a segment of keys exhibiting a distinct and recognizable distribution. Although selecting the appropriate model for a given key can be costly, one or more higher-level models are introduced to guide this selection process. For large datasets, this leads naturally to a recursive model hierarchy. The seminal work on learned index adopts this approach, giving rise to the Recursive Model Index (RMI) framework [1].

A widely adopted approach for constructing such a model tree is piecewise linear approximation [5]–[7], where a segmentation algorithm is employed to partition the sorted key array into segments. Each segment is approximated by a simple linear model defined by two parameters: slope and intercept. The segmentation algorithm operates under a specified error bound, ensuring that the predicted position of any key within a segment deviates from its true position by no more than the given bound. A smaller error bound results in a greater number of segments, each fitted with its own linear model. These models are organized as leaf nodes at the bottom level of a tree structure. The internal nodes of the tree serve to direct queries to the appropriate leaf node, either using standard B+-tree-style comparisons or recursively applying linear models.

While organizing multiple models into a hierarchy improves prediction accuracy, it departs from the core goal of learned indexes: computing a key's location in constant time. This hierarchical design brings performance closer to traditional tree-structured indexes such as the B+-tree. To mitigate this issue, the RadixSpline index [7] was introduced. It replaces the hierarchical model structure with a flat radix table over segments (represented by spline points), enabling direct access to the target segment using prefix bits. This approach eliminates costly pointer chasing across model levels. However, RadixSpline is limited to static datasets—it does not support key insertions, as its radix table is fixed to a single sorted key array. Furthermore, to avoid collisions (where one prefix maps to multiple segments), the radix table must remain sparsely populated.

To overcome these limitations in a writable learned index, we propose LASER (Line-Aware and Self-Balancing Learned Index), a novel design that enables the migration of segments across a tree structure. First, LASER allows the key segments

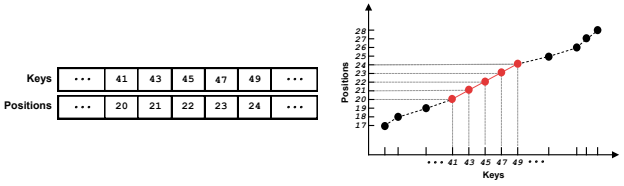
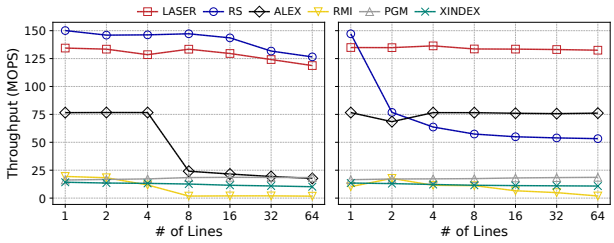


Fig. 1: Using a function to characterize the relationship between a key and its position in a sorted array.



(a) (b)

Fig. 2: (a) Fix the slope for all lines and randomly select a gap from $[2^1, 2^{10}, 2^{20}, 2^{30}]$ between two lines. (b) Randomly select a slope within [1, 100] for each line.

to be placed at any tree level, including the root node. Second, it dynamically migrates more segments with more accesses to a higher level (closer to the root) so that more accesses can be quickly completed with traversal of a few levels. Third, LASER maintains tree balance through localized compaction operations, which retrain subsets of keys into compact segments. This reduces tree height and increases the likelihood of serving lookups at the root. By adapting to workload patterns and minimizing pointer-based traversal, LASER preserves the benefits of a writable index while mitigating the performance penalties of hierarchical model structures.

B. Last-mile Search and Opportunity of its Elimination

A model in a learned index can hardly describe the mapping accurately for all keys in the key space to their respective positions in the sorted array. Instead, it only makes a prediction by computing a key's position. When the predicted position (pos_{model}) matches the actual position (pos_{real}), we call it a hit. However, in most cases, the prediction deviates from the actual position, and this deviation—denoted as the prediction error ($e(k)$) for Key k—must be corrected through a local search. To ensure correctness, the system performs a last-mile search within a bounded range: $[pos_{model} - E, pos_{model} + E]$, where $E = \max_k e(k)$ is the maximum error across all keys and serves as the model's global error bound. Since tracking the exact error for each key is impractical, this conservative bound ensures that the true position is always found—but at a cost. This local search can substantially diminish the performance benefits of a learned index. First, even a small number of large errors can inflate the error bound E, forcing broad searches for all mis-predicted keys. Second, any misprediction—no matter how minor—incurs a fixed penalty tied to E, not to the actual error size. As a result, unless the model prediction is a perfect hit, the system pays the full last-mile cost, undermining the index's efficiency.

Accurate Predictions on Straight Lines While hits are crucial for performance, achieving a high number of them in practice is challenging. Consider plotting the (key, position) points in the coordinate plane, where the X-axis represents keys and the Y-axis their positions in the sorted key array. As illustrated in Figure 1, the points are connected into a line plot. A learned index aims to fit a model to this plot. The hits take place at those points where the line plot and the model fit overlap. Most fitting strategies prioritize minimizing the maximum error E, as it defines the range of the last-mile search. As long as individual errors are within this bound, reducing smaller errors is often considered secondary. However, this approach conflicts with the goal of maximizing the number of hits, which requires precise predictions rather than just bounded ones. Moreover, even when a prediction hits the correct position, a local search may still be necessary. For example, if the key maps exactly to its true position but the slot is empty (i.e., the key doesn't exist), the index must still perform a search around that position to confirm the key's absence.

We observed that it is common in real-world datasets that many linear mappings from the keys to their positions, or straight lines in the coordinate plane, exist. Examples include student IDs assigned in series, product serial numbers, and disk blocks sequentially allocated to a file. When they are used as keys and stored in database tables, indexes to these tables will contain many segments of linear mappings, or simply lines hereafter for the sake of brevity. As an example shown in Figure 1, some contiguous keys stored in a section of the array are evenly spaced out. When we depict them in the coordinate plane, they form a line, which can be further presented as a linear function $pos = a * key + b$, where a = 0.5, b = -0.5. This function effectively serves as a model for position prediction over that key range. Identifying and isolating these linear mappings provides three appealing advantages. First, perfect accuracy—every prediction is a hit. Second, no local search needed—even for non-existent keys, absence can be confirmed directly. Third, minimal space—the mapping is compactly represented by just two parameters: the slope (a) and intercept (b).

C. Exploiting Linear Mapping

Although it is generally infeasible to fit the entire key-position mapping with a single model, learned indexes overcome this by partitioning the key space into segments, each handled by a separate model with a predefined error threshold. Given that learned indexes are designed to exploit regularities in key distribution, key sets that align well with linear patterns should, in theory, be ideal candidates—potentially delivering high read throughput. To confirm or disprove this, we construct two synthetic key sets, each consisting of one million keys forming a perfect line: $(1, 2, 3, ..., 1,000,000)$.

For the first key set, we evenly split it into n shorter lines. Between each of the two continuous lines, we insert five new keys to introduce discontinuities. The gap between each inserted key and its predecessor is randomly selected from a set of values $(2^1, 2^{10}, 2^{20}, 2^{30})$. For the second key set, we again divide the data into n shorter lines. Each line is generated using a slope randomly chosen from $(1, 2, \ldots, 100)$. Figure 2 (a) and (b) show the read throughput (in millions of operations per second) for a workload randomly reading keys in each of the two key sets, evaluated across several learned index structures: RadixSpline [7], ALEX [2], RMI [1], PGM [5], and XIndex [4].

Among the learned indexes, only RadixSpline (RS) effectively handles the key distribution patterns in the first key set. RS employs the GreedySplineCorridor algorithm [8] to identify spline segments—each modeled as a linear function within a specified error bound. With only gap variation, RS recognizes each of the lines as its spline segments and thus fully exploits the underlying linear mappings. In contrast, ALEX fails to benefit from the linearity once the number of lines exceeds four. More surprisingly, there are three learned indexes (RMI, PGM, XIndex) that derive little to no performance gains from the abundant linear mappings. Their throughput remains much lower than that of RS, a result that aligns with findings in prior comparative evaluations of learned indexes [9], [10]. For the second key set, even RS struggles to leverage linear mappings when multiple lines are present. This limitation stems from RS's design goal of bounding prediction errors within a threshold, rather than explicitly seeking zero-error segments.

To address these limitations, LASER explicitly identifies all perfect linear mappings among the keys and handles them with zero-error linear models before applying learned index models to the remaining points that are not on the lines. In this way, it maximizes hits in the key access. Unlike RS and RMI, LASER is an updatable index that supports the insertion of new keys. We have implemented LASER and extensively evaluated it across various real-world traces. Experimental results show that LASER consistently achieves up to 5.5x higher throughput.

II. The Design of LASER

The design of LASER is guided by four key objectives: (1) to fully leverage the benefits of linear mapping; (2) to enable localized retraining for dynamically capturing regularities in key distributions; (3) to optimize the best-case performance (i.e., accessing keys at the root); and (4) to bound the worst-case performance (i.e., accessing keys at the leaf level within a fixed tree height). To achieve these goals, LASER is structured as a self-balancing tree of segments, where each segment represents a sequence of keys in the key space.

A. Segments

As previously mentioned, the GreedySplineCorridor algorithm [8] generates a spline, which is a piecewise polynomial function, to interpolate keys in a sorted array. Given the array and an error bound, the algorithm outputs a list of spline segments, each approximating a subset of keys with a simple linear function whose prediction error does not exceed the specified bound. A smaller error bound results in more segments, thereby reducing the local search space but increasing the number of segments. Notably, the algorithm is designed to produce long segments rather than to minimize the prediction error for individual keys, which often leads to missed opportunities in capturing linear key patterns.

To address this limitation, LASER performs segmentation in two steps. First, it scans the sorted key array to identify lines—sequences of keys with a constant gap—each treated as a line segment. To avoid too short lines, it sets a threshold (`min_line_length`). Only a line whose length, in terms of number of keys on it, is at least as large as this threshold is identified. Second, it applies the GreedySplineCorridor algorithm to the key sequences between consecutive lines, generating spline segments for those regions. These line and spline segments are then merged into an ordered list based on their positions in the key space and organized into a self-balancing ordered tree structure.

B. The LASER Tree

As discussed, the LASER tree serves as an index structure that organizes segments rather than individual keys. By indexing at the segment level, LASER combines the strengths of learned index with the structural advantages of traditional tree-based indexes. This design preserves the efficiency of learned models within each segment while enabling dynamic operations such as segment growth, shrinkage, and reorganization. A key advantage of this segment-based structure is the significant reduction in tree size. Since each segment may represent tens or hundreds of keys, the number of nodes—and thus the tree height—is substantially reduced, leading to faster search times across the tree.

To understand the LASER tree's structure, we begin with its initialization during bulk loading. When a sorted key array is loaded into memory, LASER applies the segmentation process described earlier to partition the array into segments, which are then inserted into the tree as its initial structure.

Initially, the LASER tree contains only a single, empty root node. The generated segments are inserted into this node in sorted order. The tree is governed by a global configuration parameter `node_capacity`, which defines the maximum number of segments any node can hold. This shared limit helps control the overhead of maintaining sorted segments within nodes during insertions and deletions. If the number of initial segments exceeds the root's `node_capacity`, LASER redistributes some segments to child nodes while retaining the rest in the root. This redistribution follows a carefully designed segment placement policy based on the following considerations: (1) Access locality: Segments expected to receive more key accesses should remain in the root to improve lookup performance. While LASER does not assume a specific workload pattern, it heuristically prioritizes longer segments (those covering more keys), as they are more likely

to be accessed. (2) Routing functionality: Each segment in the root also serves as a routing element, directing queries that cannot be answered at the root to the appropriate child node. Thus, segments retained at the root must uniquely represent disjoint key ranges corresponding to the children. (3) Tree balance: To maintain a balanced structure, segments should be distributed such that each child node (and its descendants) contains approximately the same number of segments.

The segment placement policy is guided by the considerations discussed above. Let the global `node_capacity` be N, and suppose there are n segments, which are arranged in order based on their positions within the key space, to be placed in a node. LASER selects the top $T(= top\_k\% \times N)$ longest segments to remain in the current node. The $n-T$ remaining segments are then grouped into $N-T$ contiguous sequences, partitioned as evenly as possible based on their positions in the key space.

Each sequence containing more than one segment is compressed into a special placeholder segment, referred to as a dummy segment, which serves as a routing entry in the current node. The original segments within that sequence are demoted to a newly created child node, associated with the dummy segment. If a sequence contains only a single segment, it remains directly in the current node without compression.

This placement process is applied recursively to each newly created child node. As a result, bulk loading produces a segment tree that promotes long segments to the top levels while maintaining an approximately balanced structure. An illustration of the resulting LASER tree is shown in Figure 3.

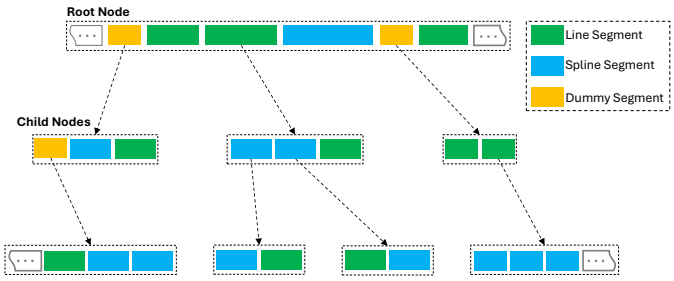


Fig. 3: The Structure of the LASER Tree

C. The Lookup Operation

Lookup is a fundamental operation that supports higher-level requests such as read, write, update, and delete. It locates the target key in the LASER tree before the corresponding action is executed. Given a search key, the lookup begins at the root node. A top-level search—accelerated by a simple linear model and an auxiliary array–is used to efficiently locate the segment whose coverage includes the key. Here, a segment's coverage refers to the key space between its start key and the start key of the next segment, while its key range spans from its start key to its end key.

Once a segment covering the key is identified, LASER checks whether the key also lies within the segment's range. If so, the segment's model—either a line model or a linear regression model, depending on whether it is a line segment—is used to search for the key directly. If the key is not found, or it lies within the segment's coverage but outside its actual range, the search proceeds recursively into the child node, as shown in Figure 3, associated with that segment. This recursive process continues until the key is found or a terminal node is reached without a matching child.

D. The Insert Request

For the insertion of a key, there are two possible options. One is to insert the key into an existing segment as a key. The second option is to insert it into the tree as a new segment. The first option modifies the segment's key distribution, which may require retraining the segment's model and potentially splitting the segment—making the insertion cost high and unpredictable. In contrast, the second option creates a new segment containing only the inserted key, referred to as a key segment, and adds it to the tree. LASER adopts this second approach by treating segments as immutable, except when updating the value of an existing key.

An insertion begins with a lookup. If the key is found, the operation is treated as an update, and the value is modified in place. If the key is not found after a recursive search, a key segment is created and inserted. The segment that was last searched during the lookup is referred to as the final segment. A final segment has no child node. If the key lies within the final segment's coverage but not its range, LASER checks whether the current node has reached its `node_capacity`. If there is room, the key segment is inserted into the node directly. Otherwise, or if the key falls within the final segment's range, a new child node is created, and the key segment is placed there. This completes the insertion process.

E. The Delete Request

A delete request begins with a lookup of the target key. If the key is found within a segment, it is logically deleted by marking it as removed. This approach preserves the key distribution and structure of the segment, so the segment's prediction model remains unchanged. If all keys in a segment are marked as deleted and the segment has no child node, the segment can be safely removed from the tree. Otherwise, the actual removal of deleted keys is deferred and handled during a subsequent tree compaction operation.

F. The Tree Compaction Operation

Although the LASER tree is balanced after the initial bulk loading, it may become unbalanced over time due to insertions and deletions. The compaction process serves two key purposes: (1) to rebalance the tree and (2) to re-organize segments in order to exploit new opportunities for applying learned indexing techniques.

1) Triggering Compaction: Compaction is triggered when the tree becomes unbalanced. We define a balanced tree as one in which each subtree rooted at a segment in the root node contains a similar number of segments. "Similar" is quantified

by comparing each subtree's average number of segments to the average number of segments per subtree. Let:

- N be the global `node_capacity` (i.e., the number of segments allowed in the root).
- n be the total number of segments in the tree.
- *AvgSize* $= n/N$ be the average number of segments per subtree.
- *MySize* be the size of the subtree rooted at a particular segment.

After each insert or delete operation that affects a root segment, LASER updates *MySize* and *AvgSize*, and checks whether the subtree is significantly imbalanced. If:

- *MySize* $>$ *Threshold* $\times$ *AvgSize* (after an insert), or
- *MySize* $<$ *AvgSize*/*Threshold* (after a delete),

then compaction is triggered. Here, *Threshold* is a constant greater than 1 that provides tolerance for temporary imbalances and limits the frequency of compaction, which can be expensive.

2) Compaction Process: When compaction is triggered by an insert at a root segment, LASER traverses the affected subtree, collects all live (non-deleted) keys, and sorts them. It then applies the bulk loading procedure (Section II-B) to this subset of keys, with two modifications: (1) only a subset of the tree is compacted. (2) The number of segments allowed at the root after compaction is set to K, where $K < N$. Initially, K = 1. If the re-segmented keys still result in a subtree size larger than *Threshold* $\times$ *AvgSize*, LASER can increase K, or use more root segments to create multiple subtrees to reduce the $MySize$s to below $AvgSize$. The final value of K is chosen to ensure this condition is met.

However, if the root node is already full, there are no free slots to accommodate the additional $K-1$ root segments. LASER then needs to free up space by reclaiming slots from smaller subtrees. It scans the root node and considers every group of three consecutive segments. For each group, it computes the average subtree size. It then selects the group with the smallest average and applies the compaction process to it. After the re-segmentation of the keys in the three segments, LASER calculates the minimal number of root segments required to keep the new subtree sizes no more than $AvgSize$. Assuming this number is k, $3-k$ segment slots can be freed by the compaction. If the freed slots are not fewer than the $K-1$ slots, the additional root node slots demanded by the initial compaction are recovered, and the compaction is done. Otherwise, additional smallest subtrees will be compacted to release more root node slots.

When compaction is triggered by a deletion that removes a segment from the root, the process is similar. However, in this case, more root slots become available, and compaction is applied to larger subtrees to consume the freed slots.

This compaction mechanism keeps the tree balanced by dynamically adjusting subtree sizes in response to significant deviations. At the same time, it enables re-application of learned index models to reorganized key distributions, thereby optimizing performance.

III. Performance Evaluation

To assess the effectiveness of the proposed LASER index design, we implemented a LASER prototype in C++. This section presents a comprehensive performance evaluation of LASER in comparison with several state-of-the-art learned indexes, including ALEX [2], LIPP [3], PGM [5], and XIndex [4], as well as traditional indexes, such as B+-tree [11] and ART [12].

All experiments were performed on a Supermicro ASG-1115S-NE3X12R server, featuring a single-socket AMD EPYC 9634 processor with 84 physical cores (up to 3.70 GHz) and 168 GB of DRAM. Hyperthreading was disabled to ensure consistent performance measurement. All workloads were restricted to a single NUMA node. The system was running Ubuntu 24.10 as the operating system.

A. The Datasets

We selected a diverse set of data sets from various sources. Table I summarizes the data sets used in the experiments along with their descriptions. Specifically, UMass_fin1 originates from the I/O trace collection of the UMass Trace Repository [13]. W048 and W054 are traces collected by CloudPhysics's caching analytics service [15] in a VMware production environment. Msr_proj and Msr_src2 are enterprise server traces provided by Microsoft Research Cambridge [14]. FIU_mail15 and FIU_mail20 are mail server traces from the Computer Science Department at Florida International University, encompassing the inboxes of mails. All data sets use 4-byte unsigned integers as keys. In all experiments, we fix the payload size to 8 bytes.

B. Workloads

To provide a comprehensive performance profile of LASER, we evaluate it using two representative workloads:

- **Read-Only (100% Read)**: The entire data set is first bulk-loaded into the index. Subsequently, the workload consists of random reads over the loaded keys.
- **Mixed (95% Read / 5% Write)**: 95% of the unique keys from each trace are initially bulk-loaded into the index. The workload then consists of 95% read operations on these existing keys and 5% write operations that insert new, previously unseen keys.

C. Throughput Results

1) Overall Read Throughput: Figure 4 presents the read throughput for each index across the different traces. The findings show that LASER outperforms other indexes in most cases, achieving up to 1.6x to 5.5x higher throughput compared to the performance of competing methods. This performance advantage is most pronounced on the UMass_fin1 trace, where LASER's design philosophy is perfectly aligned with the data's characteristics. On this trace, approximately 96% of keys are captured by line segments within the root node. Consequently, a vast majority of lookups are resolved via an extremely efficient "fast path": a single model prediction reduces the search to a specific line segment, and a simple

TABLE I: Datasets used in experiments

Dataset	Description	# of Unique Keys	Size (MB)
UMass_fin1	Set of I/O traces from the UMass Trace Repository [13]	827,801	36
Msr_proj	Set of disk LBAs on an enterprise server collected by Microsoft Research Cambridge [14]	81,374,583	2,059
Msr_src2	Set of disk LBAs on an enterprise server collected by Microsoft Research Cambridge [14]	2,667,296	97
W048	Set of virtual disk LBAs collected by CloudPhysics in production VMware environments [15]	8,974,377	866
W054	Set of virtual disk LBAs collected by CloudPhysics in production VMware environments [15]	68,984,159	2,002
FIU_mail15	Set of mail server traces collected by FIU CS department [16]	1,835,744	223
FIU_mail20	Set of mail server traces collected by FIU CS department [16]	2,638,979	215

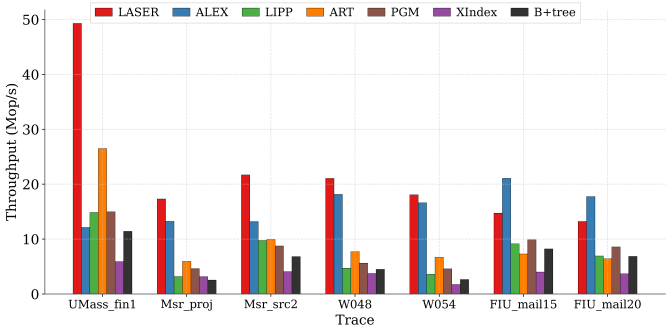


Fig. 4: Throughput of the read-only workloads on various indexes with different traces.

arithmetic offset calculation pinpoints the key's position. This approach effectively reduces query latency.

However, on the FIU_mail15 and FIU_mail20 traces, LASER's throughput, while still competitive, is slightly below that of ALEX. This is because only about 30% of keys on these traces are covered by the root node's line segments. LASER's primary architectural advantage is its "fast path" lookup for keys in these root-level lines. When the hit rate on this fast path is low, most operations must descend into child nodes. In these specific scenarios, LASER's performance relies on its underlying tree structure, where ALEX's particular implementation proves more efficient. In addition, we observe that LASER's throughput generally decreases with larger dataset sizes, as deeper trees and reduced cache locality increase lookup costs.

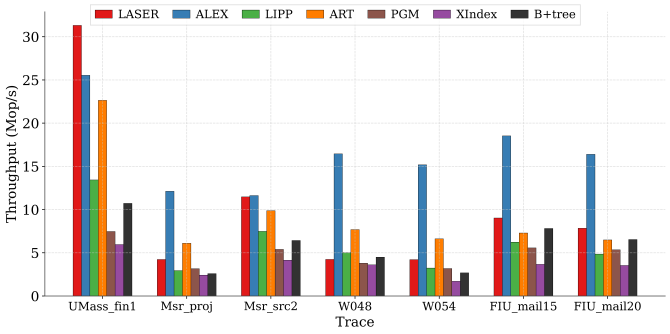


Fig. 5: Throughput of the mixed workloads on various indexes with different traces.

2) Overall Mixed Throughput: We evaluate all indexes under the mixed workload described in the Workloads section, consisting of 95% reads and 5% inserts. Figure 5 reports the throughput results across all evaluated indexes under this workload. LASER performs competitively on several traces, notably achieving the highest throughput on trace UMass_fin1, and remaining on par with ALEX on Msr_src2. This highlights LASER's strength in handling data with high locality to enable localized retraining. However, on write-sensitive traces such as Msr_proj, W048, and W054, ALEX significantly outperforms other indexes. ALEX's design favors write efficiency through the use of gapped arrays, which defer expensive node splits by absorbing insertions with lightweight shifts. This makes it particularly effective in dynamic workloads with frequent writes.

By contrast, LASER prioritizes fast read performance through carefully trained models and efficient segment routing. While this yields high read throughput, write-related workloads may trigger costly structural updates—such as node compaction—especially when the key distribution causes imbalance in the root node. LASER includes adaptive rebalancing mechanisms to mitigate these effects, but they introduce additional overhead during inserts. Overall, LASER demonstrates strong performance on mixed workloads with skew reads and moderate insert rates. With further optimization to its write path, LASER has the potential to close the gap on more dynamic traces as well.

3) Parameter Sensitivity Analysis: An index's performance is often sensitive to its configuration parameters. We analyzed the key parameters of LASER to understand their impact on read throughput and to justify our default settings.

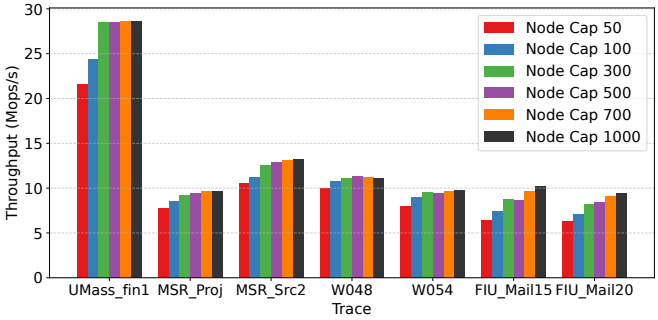


Fig. 6: Read throughput with different node capacity values

First, we examined `node_capacity`, which dictates the maximum number of segments a node can contain. As shown in Figure 6, read throughput grows as `node_capacity` increases from a small value. This is because larger nodes can cover a wider range of keys, reducing the overall depth of the tree and the number of node-to-node traversals. However,

this benefit is subject to diminishing returns. Once a node's size exceeds the capacity of the CPU's L1 or L2 caches, the time required to search within the node increases due to cache misses. This internal search latency begins to negate the advantage of a shallower tree. Our experiments show that performance gains plateau around a capacity of 300. We select `node_capacity` = 100 as the default, as it captures the majority of the throughput benefits while ensuring nodes remain compact and cache-friendly, providing a robust balance between read performance and memory hierarchy efficiency.

Next, we considered `min_line_length`, the threshold for forming a line segment. This parameter controls the trade-off between aggressively identifying linear patterns and the overhead of managing them. As Figure 7 illustrates, a higher `min_line_length` leads to a monotonic decrease in the total number of segments. While a lower threshold might seem beneficial by capturing more lines, it risks creating a fragmented index with many small segments. This fragmentation increases both memory overhead and the number of segments to check during a lookup. We set `min_line_length` = 100 as a default that effectively reduces the number of segments while still being sensitive enough to detect meaningful, contiguous lines to support efficient read access.

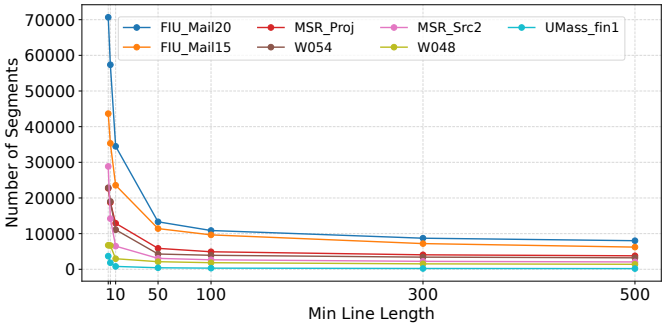


Fig. 7: Number of segments with different min line lengths

The `top_k` parameter denotes the percentage of the largest segments—ranked by the number of keys covered relative to node capacity—that are retained in the root node during bulk loading or model retraining. Conceptually, the root node acts as a specialized, high-speed cache for the most frequently accessed data patterns. Increasing `top_k`, as shown in Figure 8, populates this "cache" more aggressively, increasing the hit rate of the fast path and boosting overall throughput. However, a larger `top_k` results in a larger root node. Just as with `node_capacity`, an oversized root node can suffer from poor cache locality, making the internal search within it slower. The empirical data shows that throughput gains diminish significantly beyond `top_k` = 0.5. At this point, the latency added by scanning a larger root begins to offset the benefit of avoiding a child traversal. Thus, `top_k` = 0.5 represents the optimal value that maximizes the utility of the root node without incurring significant internal search penalties.

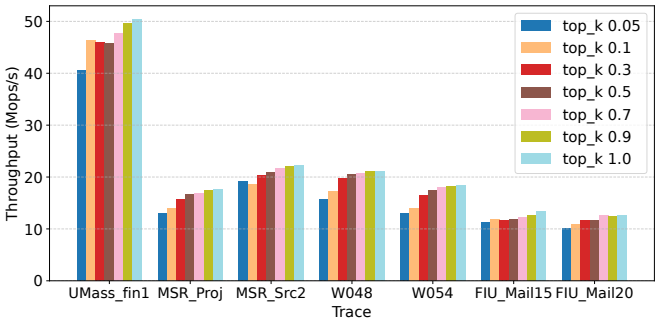


Fig. 8: Read throughput with various top_k values

4) Impact of Line Recognition: Finally, to quantify the contribution of our core architectural innovation, we conducted a comparative evaluation. We constructed two index versions: one with our standard hybrid approach (line recognition + splines) and one where all keys were modeled exclusively by the general-purpose GreedySplineCorridor algorithm. The two versions were then subjected to the same read-only workload. The results, presented in Figure 9, are clear and conclusive. The configuration that explicitly recognizes and optimizes for line segments yields dramatically higher read throughput. This confirms our central hypothesis: specializing for the common case of linear data patterns with a simple, fast arithmetic model is vastly more efficient than relying solely on a more complex, general-purpose spline model. The GreedySplineCorridor algorithm is powerful for non-linear data, but using it for simple lines introduces unnecessary computational overhead. LASER's hybrid approach ensures that the simplest, fastest model is used when possible, delivering superior performance.

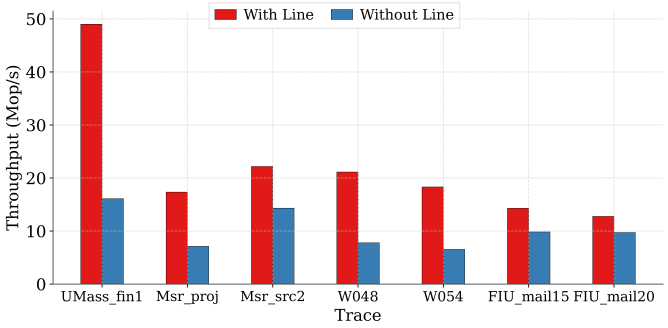


Fig. 9: Throughput comparison across different traces with and without the line optimization

IV. Related Works

The field of learned index structures was first introduced by Kraska et al. [1], who proposed that traditional indexes like B+-trees could be replaced by machine learning models. Their Recursive Model Index (RMI) uses a hierarchy of models to learn the cumulative distribution function (CDF) of keys, thereby predicting a key's position in a sorted array. This approach showed significant potential but was initially limited to static data and offered no worst-case performance guarantees, often requiring a costly last-mile search to locate a key after

an inexact prediction. Subsequent research has largely focused on addressing these initial limitations. To handle dynamic workloads with frequent insertions and updates, researchers developed structures like ALEX [2], which uses gapped arrays and adaptive node splits, and XIndex [4], which is designed for concurrency and uses delta indexes to manage writes. Other works have focused on improving prediction accuracy and providing performance guarantees. The PGM-index [5] and FITing-Tree [6] use piecewise linear functions with a bounded error, ensuring the final search is always constrained. Similarly, RadixSpline [7] and LIPP [3] offer different strategies for building accurate models efficiently. The increase of these designs led to important benchmarking efforts to standardize comparisons and assessment. Works like Marcus et al. [17], SOSD [18], and GRE [9] provided a benchmark platform to analyze the learned indexes under realistic and dynamic workloads. Beyond single-dimensional numeric keys, research has also extended the learned index to more complex data types and systems. For instance, SIndex [19] and the more recent LITS [20] and Kim et al. [21] tackle the challenges of indexing variable-length string keys. Furthermore, another line of research has focused on adapting learned indexes to modern hardware, with designs like APEX [22] for persistent memory, specific optimizations for disk-based systems [23], and GPU-accelerated implementations [24].

While these advancements have made learned indexes more practical, they have not fully resolved the core performance bottlenecks. Even with bounded errors, a final search step is almost required. Our work, LASER, directly targets these remaining inefficiencies. It recognizes the line segments, for the common case of linearly distributed keys, to eliminate the last-mile search entirely.

V. Conclusion

Learned indexes offer significant performance potential but are often hindered by the overhead of additional model traversals and costly last-mile searches. In this paper, we introduced LASER, a novel learned index designed to directly mitigate these limitations. By exploiting key access locality and linear distribution patterns, LASER adaptively restructures its model hierarchy to reduce lookups and uses calculation to completely eliminate the final search for linear key segments. Our experimental evaluation confirms that LASER achieves substantial performance gains over both state-of-the-art learned indexes and traditional index structures, advancing their practicality for high-performance index systems.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by the U.S. National Science Foundation under Grant CCF-2313146.

References

[1] 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.

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

[3] J. Wu, Y. Zhang, S. Chen, J. Wang, Y. Chen, and C. Xing, "Updatable learned index with precise positions," *Proc. VLDB Endow.*, vol. 14, no. 8, p. 1276–1288, Apr. 2021. [Online]. Available: https://doi.org/10.14778/3457390.3457393

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] T. Neumann and S. Michel, "Smooth interpolating histograms with error guarantees," in *British National Conference on Databases*. Springer, 2008, pp. 126–138.

[9] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *arXiv preprint arXiv:2207.02900*, 2022.

[10] Z. Sun, X. Zhou, and G. Li, "Learned index: A comprehensive experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 16, no. 8, pp. 1992–2004, 2023.

[11] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[12] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The art of practical synchronization," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, 2016, pp. 1–8.

[13] K. Bates and B. McNutt, "Storage - umass trace repository," https://traces.cs.umass.edu/docs/traces/storage/, 2025.

[14] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, Nov. 2008. [Online]. Available: https://doi.org/10.1145/1416944.1416949

[15] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 487–498. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

[16] SNIA IOTTA Repository, "Msr cambridge block i/o traces," http://iotta.snia.org/traces/block-io/391, 2025, accessed: July 2025.

[17] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska, "Benchmarking learned indexes," *arXiv preprint arXiv:2006.12804*, 2020.

[18] 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.

[19] Y. Wang, C. Tang, Z. Wang, and H. Chen, "Sindex: a scalable learned index for string keys," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020, pp. 17–24.

[20] Y. Yang and S. Chen, "Lits: An optimized learned index for strings," *Proceedings of the VLDB Endowment*, vol. 17, no. 11, pp. 3415–3427, 2024.

[21] M. Kim, J. Hwang, G. Heo, S. Cho, D. Mahajan, and J. Park, "Accelerating string-key learned index structures via memoization-based incremental training," *arXiv preprint arXiv:2403.11472*, 2024.

[22] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang, "Apex: a high-performance learned index on persistent memory," *arXiv preprint arXiv:2105.00683*, 2021.

[23] J. Zhang, K. Su, and H. Zhang, "Making in-memory learned indexes efficient on disk," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–26, 2024.

[24] X. Zhong, Y. Zhang, Y. Chen, C. Li, and C. Xing, "Learned index on gpu," in *2022 IEEE 38th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2022, pp. 117–122.