LiBox: A Learned Index as an Array to Minimize Last-Mile Search

Jian Zhou
University of Texas at Arlington
Arlington, USA
jxz9486@mavs.uta.edu

Luna Wang
Cupertino High School
Cupertino, USA
lunawang257@gmail.com

Shuaihua Zhao
University of Texas at Arlington
Arlington, USA
sxz6329@mavs.uta.edu

Chen Zhong
University of Texas at Arlington
Arlington, USA
chen.zhong@mavs.uta.edu

Song Jiang
University of Texas at Arlington
Arlington, USA
song.jiang@uta.edu

ABSTRACT

Learned indexes have attracted significant attention for their potential to deliver substantial performance and space savings over traditional index structures. Their advantage lies in replacing explicit key comparisons with model-based computation that predicts the position of a search key in a sorted array. However, prediction errors prevent models from precisely locating keys, requiring a last-mile search over a candidate range. Both model evaluation and last-mile search can be expensive, limiting the performance.

We propose LiBox, a hierarchical, box-based learned index that overcomes these limitations. LiBox partitions a sorted key array into "boxes" such that: (1) the box containing a search key can be identified with zero error using a simple linear regression function, and (2) the last-mile search within a box requires only a single AVX-512 instruction. This design yields a highly predictable and efficient lookup, with each query involving a fixed, minimal number of instructions and memory accesses. By allocating modest extra space within each box to handle irregular key distributions, LiBox supports both read and write queries at near array-access speed. Furthermore, its reorganization operations can be aligned with workload read/write intensity, enabling high-performance reads while hiding structural modification costs.

We propose LiBox, a hierarchical box-based learned index that addresses these limitations. LiBox partitions a sorted key array into disjoint "boxes" such that: (1) the box containing a search key can be identified without error using a simple linear regression function, and (2) the last-mile search within a box usually requires only a single AVX-512 vector instruction. This design yields a highly predictable and efficient lookup process, with each query involving a fixed and minimal number of instructions and memory accesses. By allocating a modest amount of additional space within each box to accommodate irregular key distributions, LiBox supports both read and write queries at near–array-access speed.

We implemented LiBox and conducted an extensive experimental evaluation. The results demonstrate that it significantly outperforms both state-of-the-art learned indexes (e.g., ALEX, LIPP) and non-learned indexes (e.g., ART) while achieving comparable or better space efficiency.

PVLDB Reference Format:
Jian Zhou, Luna Wang, Shuaihua Zhao, Chen Zhong, and Song Jiang. LiBox: A Learned Index as an Array to Minimize Last-Mile Search. PVLDB, 19(5): 836 - 848, 2026.
doi:10.14778/3796195.3796199

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/strivesnail/Libox.git.

1 INTRODUCTION

Learned indexes have attracted significant attention in data management research [15]. In principle, they can achieve O(1) indexing by computing the position of a key in a sorted array. While hash tables also compute positions and provide O(1) lookup, learned indexes are more powerful because they maintain a sorted key layout, enabling efficient range queries. By storing key-value (KV) pairs in sorted order, learned indexes have the potential to supersede traditional comparison-based indexes such as B^+-trees, which incur O(log n) time complexity. The originally proposed Recursive Model Index (RMI) [15] did not support insertions or deletions, but subsequent studies have extended learned indexes to efficiently handle updates [7, 8, 25, 32].

The effectiveness of a learned index depends heavily on the quality of the model(s)—or prediction functions—used to compute key locations. Quality is measured by the prediction accuracy when mapping each key to its position in the sorted key array. Constructing a single model that is both efficient and accurate across millions of keys with diverse distributions is generally infeasible. A common solution is to partition the key space into multiple segments, each modeled independently according to its local distribution. Because determining the correct segment for a search key can be expensive, higher-level models are often introduced to direct this process. For very large key sets, this approach can be applied recursively, forming a hierarchy of models. The seminal learned index design follows this principle, resulting in the Recursive Model Index (RMI) [15].

Even a model covering only a small segment of keys can struggle to accurately compute the position of a search key (k) in a sorted key array for every key in the segment. Therefore, the model produces a prediction of a key's location rather than an exact result. When a misprediction occurs, there is a non-zero distance between the predicted position ($pos_{\text{model}}(k)$) and the true

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 5 ISSN 2150-8097.
doi:10.14778/3796195.3796199

key position ($pos_{\text{real}}(k)$) in the array. This distance is referred to as the prediction error ($e(k)$) for key k. This prediction error can be corrected by performing a localized search within the range $[pos_{\text{model}}(k) - e(k), pos_{\text{model}}(k) + e(k)]$. Because it is impractical to individually record the errors for every key, the search range is instead defined by the maximum error across all keys ($E = \max_k e(k)$), which serves as the error bound for the model. The resulting local search can substantially offset the performance benefits of a learned index. First, a single large error inflates the search range for all mispredictions, forcing searches over a broad scope. Second, as long as the prediction is not exact, the penalty is not correlated to the accuracy, and is maximized to the full search cost defined by the error bound.

In summary, existing learned index designs suffer from one or more of the following limitations. (1) Even relatively simple models, such as ReLU-based neural networks, can be too costly for index operations that demand extremely low latency. (2) A hierarchical model structure with more than two to three levels may substantially degrade search performance due to random memory access. (3) The last-mile search over hundreds of sorted keys can be too expensive in a highly optimized hierarchical structure. It is noted that these issues are often interrelated. For example, reducing the number of levels or using more efficient but less accurate models may increase the error bound, thereby expanding the last-mile search range. Consequently, it remains challenging to design a learned index that simultaneously uses only ultra-simple models, such as linear regression functions, employs very few hierarchical levels, and minimizes the cost of last-mile searches.

In this paper, we propose a learned index that addresses all these issues in one design. It aims to run a simple linear regression function only two to three times to compute the location of a search key without any prediction errors. The key technique enabling this goal is to fit a straight line over fixed key ranges, referred to as *windows*, rather than over individual keys, without any errors. Keys within each window are grouped into a box. The number of keys in a box is bounded so that a single SIMD instruction, such as AVX-512, can process them at once.

In conclusion, we make three key contributions in this work:

- We propose a technique that transforms irregularities in key distributions, which is the root cause of mispredictions in model-based computation of key positions, into regularity in box distributions, enabling fully accurate and high-speed computation of the box position for a search key.
- Leveraging SIMD instructions, which are widely available in modern processors, the search key within a box can be rapidly located using a single instruction.
- We design a box-based learned index, named *LiBox*, to address several practical challenges, including adaptively determining window size and box capacity, as well as efficiently accommodating new keys into the index.

2 THE DESIGN OF LIBOX

The core technique introduced in the design of LiBox is to transform an irregular key distribution into a fully linear window distribution, where a window is a fixed-size unit of key space. The number of keys in each window may vary. When the key space is divided into multiple fixed-size windows, the mapping from a key to its corresponding window is simply a linear function. While a window is a logical concept, it is mapped to a box in an array physically storing the keys. A box consists of a fixed number of contiguous slots in the array. For a key space, the ith window is mapped to the ith box in the array, and all keys covered by a window are stored in its corresponding box. Accordingly, the number of keys in a box may also vary, even though the box's capacity is fixed.

2.1 Opportunity and Challenges

As an example, Figure 1 shows the position of each key in an array, or its index in the array, for all keys from a virtual disk trace, where each key represents a disk address (LBA)[1]. This is essentially a CDF (Cumulative Distribution Function) graph illustrating the keys' distribution. From a high-level perspective, the CDF curve appears smooth and regular. However, at the level of individual keys, their distribution is highly irregular, and it is difficult to find an efficient function that maps a key precisely to its position. Therefore, an additional step is required to map a key to its exact position in the array, known as the local last-mile search. These two steps pose potential performance issues. Using a powerful function, the index may become CPU-intensive, while relying on the last-mile search may incur multiple cache misses, both of which can slow down index lookup. In contrast, LiBox implements its two-step process much more efficiently. First, it uses the simplest and most efficient linear function to precisely map a key to its box. Second, it locates the key within the box using a single SIMD instruction, assuming the box size has been appropriately pre-determined.

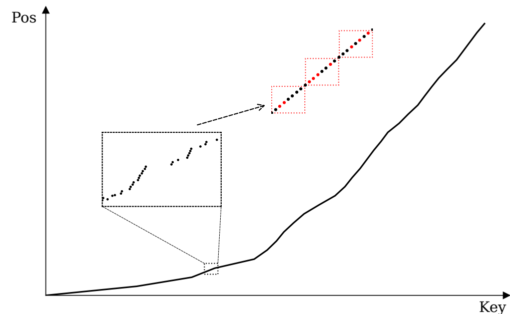


Figure 1: The CDF curve illustrates the distribution of keys in a key space. A zoomed-in view of a sampled key region reveals a highly irregular key distribution. By using fixed-size boxes that store keys (shown in red), a simple linear relationship is introduced between keys and boxes.

In this box-structured learned index framework, the design of LiBox must address several challenges. First, to enable a linear mapping between windows and boxes, each box must have a fixed size, as illustrated in the second zoomed-in plot in Figure 1. However, each box is designed to store only the keys covered by its corresponding window. To ensure efficient search within a box, the box size must remain constant (e.g., 64 keys). At the same time, if the window size is too small, many empty slots must be allocated in the

[1]It is the VMWare W018 trace containing 3,068,126 unique keys [26].

box, wasting memory. Conversely, if the window size is too large, the corresponding box may not be able to accommodate all keys in the window. Therefore, LiBox must carefully select the window size to balance these design parameters. Second, using a uniform window size across the entire key space with highly variable key distributions makes maintaining a fixed box size less feasible. Accordingly, LiBox may need to partition the key space into multiple segments and apply different window sizes. This necessitates a higher-level index structure. Third, inserting new keys from write requests into a box may cause it to overflow, requiring a space- and time-efficient strategy to handle the issue. Fourth, when writes and deletes accumulate over time in a limited key region, LiBox requires efficient Structural Modification Operations (SMOs) that minimally disrupt ongoing user requests.

2.2 Segments and Boxes

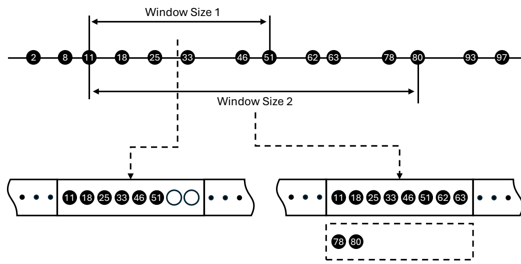


Figure 2: Illustration of the mapping from a window to a box, where all keys within a window in the key space are stored in a corresponding box in the array. A small window size, as shown in "Window Size 1", can cause box underutilization (underflow), whereas a large window size, as shown in "Window Size 2", can lead to box overflow.

A segment is an array of boxes that share a common window size and cover a contiguous region of the key space. A window size that is well suited for one key region may be inappropriate for an adjacent region, where it can cause excessive overflow (i.e., the number of keys in a window exceeds the box capacity) or severe underflow (i.e., the number of keys in a window is far less than the box capacity), as illustrated in Figure 2. This behavior indicates a significant change in key distribution. Without adjusting the window size accordingly, such changes lead to either time inefficiency (due to additional searches caused by overflow) or space inefficiency (due to unused capacity in underflowed boxes). To address this issue, a new window size must be adopted to adapt to the changed distribution to initiate a new segment. The remaining challenge is how to bound the underflow and overflow ratios during segment construction, where the underflow ratio is defined as the fraction of empty slots in a box relative to the box capacity, and the overflow ratio is defined as the fraction of keys exceeding the box capacity.

As all keys within a window are mapped into a single box, the number of keys that can be stored in a box is upper bounded by the box size. To generate a segment, starting from a given key position in the key space, the maximum window size is defined as the key-space distance that covers exactly c keys, where c denotes the capacity of a box. At different key positions, this maximum window size may vary due to differences in key density. In particular, a sparser key distribution results in a larger maximum window size.

Algorithm 1: Construction of a Segment

```
1  Function Segmentation(KeyArray, cur_pos, α, β):
       /* Randomly sample k key positions and compute candidate
          window sizes */
2      sample_key_positions[] = RandomSample(KeyArray, cur_pos, expected_end_pos);
3      candidate_win_sizes[] =
         FindCanWinSizes(sample_key_positions[]);
4      sorted_win_sizes[] = Sort(candidate_win_sizes[]);
5      cur_win = FindMedianWinSize(sorted_win_sizes[]);
6      max_segment = EmptySegment();
7      while cur_win != NULL do
8          [segment, break_condition] = CreateOneSegment(KeyArray, cur_pos, cur_win, α, β);
9          if segment.length > max_segment.length then
10             max_segment = segment;
           /* Select next candidate: smaller if overflow, larger
              if underflow */
11         if break_condition == OVERFLOW then
12             cur_win = FindNextSmallerWinSize(sorted_win_sizes[], cur_win);
13         else if break_condition == UNDERFLOW then
14             cur_win =
                 FindNextLargerWinSize(sorted_win_sizes[],
                 cur_win);
15         else
16             cur_win = NULL;
17     return max_segment;
```

The algorithm for generating a segment starting at a given key position in the key space takes two parameters: the thresholds on the cumulative overflow and underflow ratios for a segment, denoted by α and β, respectively. These cumulative ratios (α and β) are defined as the ratio of the total number of overflow keys (or empty slots, respectively) within a segment to the segment capacity, i.e., the total number of key slots across all boxes in the segment.

With the given α and β thresholds, the segmentation algorithm aims to identify a window size that maximizes the length of the target segment. For a fixed window size, the algorithm scans the sorted key array starting from the initial key, partitioning the keys into contiguous boxes sequentially. As the algorithm advances to each new box, it computes the cumulative overflow and underflow ratios, accumulated from the first box to the current one. If the overflow ratio exceeds α, or if the underflow ratio exceeds β, the constraint condition is violated and the segment can no longer be extended to include the current box. Consequently, the segment constructed under the given window size spans from the first box to the box immediately preceding the current one.

Starting from a given key position in the array, the algorithm may generate segments of varying lengths (in terms of the number of keys covered) under different window sizes. While LiBox strives to minimize the total number of segments, the algorithm aims to select a window size that maximizes segment length. To this end, the algorithm constructs a set of representative candidate window sizes and evaluates them iteratively to obtain the longest possible

segment. Specifically, it randomly samples k key positions in the key space, ranging from the start key to an expected end key for the next segment. The expected end key is determined by doubling the length of the largest recent segment, allowing room for segment growth. The algorithm computes the maximum window size, which covers exactly 64 keys, at each sampled key position, yielding k candidate window sizes, which are then sorted in ascending order to form a window size list.

The algorithm initially selects the median window size from the list. During segment construction, if the overflow constraint condition (defined by the α threshold) is violated, the algorithm reduces the window size by selecting the next smaller candidate. Conversely, if the underflow constraint condition (defined by the β threshold) is violated, the algorithm increases the window size by selecting the next larger candidate. The process terminates when one of the following conditions holds: (1) the newly constructed segment is no longer than the previous one; (2) a different condition than in the previous iteration is violated; or (3) the smallest or largest window size in the candidate list is reached.

For efficiency, the algorithm selects a moderate value of k (by default, $k = 5$), which has been empirically shown to provide a good balance between runtime overhead and segment length. Moreover, instead of scanning every key in the array to locate the next box boundary, the algorithm jumps ahead by the number of keys in the previous box and then identifies the boundary by searching only among nearby keys.

The pseudocode for generating a segment is described in Algorithm 1. This segmentation algorithm is executed either after bulk loading the initial set of keys to initialize a LiBox index, or after a large number of insertions and deletions occur within a segment, triggering re-segmentation.

2.3 The Hierarchical LiBox

All keys belonging to a segment are placed in an array, on which boxes are defined. If LiBox can identify the segment in which a search key resides, it can directly compute the corresponding box number for the key. However, a hierarchical index structure is required to locate the correct segment for a search key. At a minimum, LiBox requires one additional level, referred to as the top level, above the level where the keys are stored, which is called the bottom level. The first key of each segment in the bottom level is stored in sorted order in an array, which is then organized into one or more segments of boxes, enabling similarly fast lookup at the top level. If the number of segments at the top level becomes very large, an additional upper level may be required. However, empirical evidence from publicly available real-world datasets suggests that such cases are rare in practice.

With appropriately chosen α and β values (e.g., 10% and 50%, respectively) at the bottom level, a segment can contain a large number of boxes, resulting in a moderate total number of segments. Consequently, the top level contains a relatively small number of keys. At the top level, we can adopt a larger β value to further reduce the segment count while maintaining low memory overhead.

The search for a key in the index begins by locating the segment in the top level (if there is more than one segment in that level). To facilitate this, the index maintains a root segment, which stores the start keys of the top-level segments along with pointers to them. By setting the underflow ratio β to 100%, LiBox ensures that only a single root segment exists. Within this segment, LiBox uses the segment's simple linear function ($pos = a \cdot key + b$) to locate the key's box, where a search using AVX instructions, such as $_mm512_cmpge_epi8_mask$ or $_mm512_cmple_epi8_mask$, is performed to locate the segment in the bottom level. Once the top-level segment is located, LiBox repeats the same search process to obtain the pointer to the corresponding lower-level segment, where its linear function is then used to precisely identify the box containing the key. LiBox subsequently uses an SIMD instruction to locate the key within the box. Details of the in-box search operation are described later. Assuming the key is found in the box, if the request is a read, the value is returned. Otherwise, if it is an update, the value is modified accordingly.

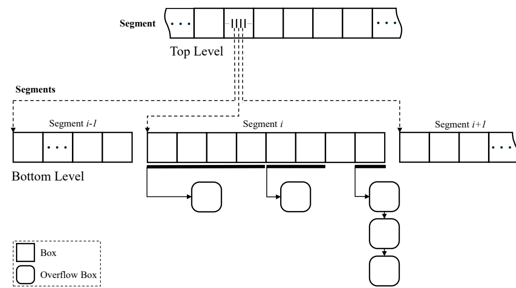


Figure 3: The hierarchical LiBox structure uses the top-level segment to locate the corresponding bottom-level segment during a key search. Within each segment the box containing the key can be precisely identified using a simple linear function. Overflow keys are stored in overflow boxes, which can be shared among a subset of boxes within the segment.

If a write request occurs and the key is not found in the box, and the box still has empty slots, the key is inserted into any available slot. Since LiBox uses a bitmap to track slot occupancy in each box, an empty slot can be located instantly. However, if the box is full, the key becomes an overflow one. When a segment contains overflow key(s), it allocates overflow box(es) that share the same structure and size as regular boxes. To maximize space efficiency, LiBox attempts to share overflow boxes among multiple regular boxes whenever possible.

As illustrated in Figure 3, initially, when the number of overflow keys is less than a box's capacity, the entire segment shares a single overflow box. When the box becomes full and a new overflow key must be inserted, the overflow box is split, meaning a new overflow box is allocated. All keys overflowing from the first $\lceil box_num/2 \rceil$ regular boxes remain in the initial overflow box, while the remaining keys are moved to the new overflow box. The two overflow boxes are then linked to the first boxes of the first and second halves of the segment, respectively. If the overflow box intended for a new key is still full, the box continues to split, allowing the corresponding two $\lceil box_num/4 \rceil$ sub-segments to each have their own overflow box. As more keys are inserted into a segment, overflow boxes become progressively less shared among regular boxes. Specifically, an overflow box may be shared among $\lceil box_num/2^n \rceil$

Algorithm 2: Search for a Given Key

```
1  Function Search(key):
2      top_segment = FindTopLevelSegment(key);
3      top_box = top_segment.FindBox(key); // use this
         segment's linear function to compute the box whose key
         range covers the key
4      bottom_segment = top_box.SIMDSearch(key); // find
         the bottom-level segment whose key range covers the key
5      bottom_box = bottom_segment.FindBox(key);
6      kv_pair = NULL;
7      kv_pair = bottom_box.AVXLookup(key); // use AVX
         instruction to locate the key
8      if kv_pair != NULL then
9          return [bottom_box, kv_pair];
       // not found in the regular box
10     overflow_box = bottom_box.next_overflow;
11     while overflow_box != NULL do
12         kv_pair = overflow_box.AVXLookup(key);
13         if kv_pair != NULL then
14             return [overflow_box, kv_pair];
15         overflow_box = overflow_box.next_overflow;
16     return [bottom_box, NULL];
```

Algorithm 3: Read and Write a Key-value Pair

```
1  Function Read(key):
2      [box, kv_pair] = Search(key);
3      if kv_pair == NULL then
4          return NULL;
5      return kv_pair.value;
6  Function Write(key, value):
7      [box, kv_pair] = Search(key);
8      if kv_pair != NULL then
           // this is an update
9          kv_pair.value = value;
10         return kv_pair;
       /* Key does not exist, insert into box or overflow boxes */
11     if box.isEmptySlotsAvailable then
12         kv_pair = box.InsertKV(key, value);
13         return kv_pair;
       /* Look for an empty slot in overflow boxes */
14     overflow_box = box.next_overflow;
15     while overflow_box != NULL do
16         if overflow_box.isEmptySlotsAvailable then
17             kv_pair = overflow_box.InsertKV(key, value);
18             return kv_pair;
19         overflow_box = overflow_box.next_overflow;
       /* A new overflow box is required */
20     overflow_box = box.AllocAndLinkOverflow;
21     kv_pair = overflow_box.InsertKV(key, value);
22     return kv_pair;
```

regular boxes, where n increases over time. Eventually, each regular box may have one or more dedicated overflow boxes, which are organized as a linked list after many new keys are mapped to the regular box for insertion. The LiBox structure with overflow boxes is illustrated in Figure 3.

2.4 Search in a Box

After LiBox uses simple linear functions to locate the box in the bottom level, it has a limited search space equal to the box size. LiBox does not rely on search algorithms that require sorted key placement, such as binary search or exponential search. Consequently, a new key can be inserted into any empty slot, leaving the keys in the box unsorted. This approach ensures that insertions and deletions in a box do not require moving multiple keys, which would be an expensive operation. At the same time, LiBox does not need to scan the array of keys individually for comparisons. Instead, it uses a single SIMD instruction to compare all keys in the box simultaneously.

Specifically, in the current LiBox implementation, the box size c is set at 64 (holding up to 64 keys), and the instruction is AVX-512 [30], which is widely supported in various mainstream processors such as Intel Xeon Phi, Intel Core X-Series, and recent Intel Core Desktop CPUs as well as AMD's Zen 4 and Zen 5 CPUs. To maximize the number of keys that can be compared in one AVX-512 instruction (ideally 64 keys), LiBox chooses only one byte, named the comparison byte, from each key in the comparison. In the meantime, to minimize the number of comparisons (ideally only one AVX-512 comparison), LiBox applies a simple hash function on each 8-byte user key to generate a comparison byte to avoid collision of comparison bytes for keys in a box (i.e., multiple keys share a common comparison byte). Specifically, LiBox collects the comparison bytes of all keys in the box into a byte array and applies the instruction to compare each of them with the comparison byte of the search key simultaneously. If there isn't a match of the byte in the array, the search key doesn't exist in the box. Otherwise, for each of the matched bytes in the array, LiBox compares each of their corresponding full keys with the search key. This seems to be an expensive operation. However, with a limited number of keys in a small key range (the window size) and use of the hash function, the probability of collision (i.e., with more than one matching from AVX-512) is very rare. LiBox has a bitmap to track empty array slots in the box. Any matching(s) on the empty slots are disregarded. Therefore, for a search key existing in a box, it almost always executes only two compare instructions (one SIMD instruction and one for full-key comparison). The pseudocode for searching a key as well as reading/writing a key-value pair is described in Algorithms 2 and 3, respectively.

While AVX-512 instructions are ubiquitous in today's mainstream CPUs to support widely used workloads such as multimedia processing, scientific computation, AI/ML, and gaming, they may not be available on some lower-end consumer computer CPUs. In this case, a binary search is required within a box. To enable the binary search, keys in a box must be sorted, which further increases the key insert cost. These have been challenges for long time in various index designs. Therefore, AVX-512 instructions are essential for LiBox to keep its performance advantage.

2.5 Support of Range Search

The design of a range search operation is usually straightforward in a fully sorted index, such as a B^+-tree. It simply locates the first key in the range and scans through the sorted list of keys to the last key in the range. However, if the keys are not fully sorted, such as in an

LSM-tree where keys are stored across multiple sorted lists, range search can become very expensive [37]. In LiBox, keys within a box may be unsorted to allow faster insertions, and keys in regular boxes and their overflow boxes are not sorted together. Maintaining a fully sorted order at all times would be too costly to support range search. Accordingly, LiBox does not modify the design for range search. Instead, it identifies all boxes whose key ranges overlap with the search range. For each boundary box, whose keys are partially covered by the search range, LiBox uses AVX512 instructions to perform parallel range comparisons, using operations such as "$_mm512_cmpge_epi8_mask$" and "$_mm512_cmple_epi8_mask$" within each box to extract keys falling within the search range. These operations can be performed without requiring the keys in the boxes to be sorted. Because the vectorized filtering efficiently identifies keys in the range, LiBox does not need to individually compare each key to check whether it is in the range. The resulting keys, or those extracted from partially overlapped boxes and all keys in boxes fully contained in the search range, constitute the set of keys returned by the search query.

Note that in this case, the AVX512 instruction is used for range search rather than exact key matching. Therefore, LiBox cannot use a single byte from a key as input, meaning the instruction must be applied multiple times for each boundary box. To reduce the number of executions, LiBox sorts the keys within the box beforehand, allowing the search to terminate early once it is determined that not all keys in an AVX512 vector are smaller (or larger) than the boundary key of the query range, because under this condition LiBox can immediately identify which keys in the box should be included in the query and which should not.

2.6 Re-segmentation

With the presence of overflow boxes, key searches may extend into one or more of these boxes, which can slow down both read and write operations. To address this, LiBox removes overflow boxes, which serve as temporary buffers for new keys from write requests. When the overflow ratio of a segment (i.e., the ratio of overflow boxes to regular boxes in the segment) exceeds a threshold γ, or when the length of the overflow list for any regular box in the segment exceeds a threshold L, a re-segmentation operation is triggered. Similarly, underflow boxes may exist, i.e., boxes whose underflow ratio exceeds β due to key deletions. When the underflow ratio of a segment(i.e., the ratio of underflow boxes to total boxes in the segment) exceeds γ, re-segmentation is also triggered. During re-segmentation, keys from both regular and overflow boxes are merge-sorted into a new array, and new segment(s) and boxes are generated as described in Section 2.2.

If there is more than one segment after re-segmentation at the low level, new key(s) are inserted into the corresponding top-level segment(s). With a large overflow ratio α and infrequent re-segmentation operations, there are typically sufficient empty slots in the top-level boxes to accommodate the new keys. Otherwise, a top-level re-segmentation is immediately triggered to create space for the new keys. While a top-level re-segmentation depends on its causal low-level re-segmentation and cannot be performed concurrently with it, it can run concurrently with other low-level re-segmentation operations. Additionally, low-level re-segmentation operations for different segments can be executed in parallel.

The performance of the index is sensitive to top-level search time, as every search passes through this level. Therefore, overflow boxes are not allowed at the top level. If overflow boxes were permitted, pointer-chasing operations would occur, resulting in CPU cache misses. Furthermore, a re-segmentation at the low level may increase the number of segments in the bottom level, which adds more keys to the top level and can trigger top-level re-segmentation, significantly impacting overall index performance. To postpone or reduce top-level re-segmentation, LiBox monitors the occupancy of top-level boxes whenever a new key is inserted. A box whose occupancy exceeds a threshold (80% by default) is flagged as compaction-ready. When the index is not actively servicing user requests, the system may instruct LiBox to perform a re-segmentation that includes all low-level segments covered by a compaction-ready top-level box. This opportunistic approach improves index performance in the background.

2.7 Support of Concurrency

LiBox is inherently scalable due to its near-flat structure. Except during a re-segmentation operation, which occurs relatively infrequently, exclusive access is enforced at the individual box level. Each segment maintains a shared-exclusive lock. Importantly, this lock does not affect readers, which are always lock-free at the segment level and can enter a segment without requesting any locks.

The lock is shared among writers, allowing multiple writers to enter a segment concurrently, but it is exclusive for re-segmentation operations. When a segment is selected for re-segmentation, it acquires the lock exclusively, preventing any writers from entering until the operation completes and the lock is released. Writers check the lock before entering a segment but do not acquire it exclusively, allowing them to proceed concurrently with other writers unless a re-segmentation is in progress.

Each box within a segment also maintains a shared-exclusive lock, ensuring that at most one writer can operate on the box at a time, while multiple readers may access the box concurrently as long as the lock is not held by a writer. If a writer triggers the creation of a new overflow box due to the full occupancy of an existing overflow box, it retains its acquired lock on the original box until the new overflow box is allocated and linked. This prevents other read or write threads from entering the overflow box from a regular box during this operation.

A reader can access a segment lock-free, even when the segment is undergoing re-segmentation, because re-segmentation is an out-of-place write operation. Data in the original segment remain available to readers, allowing re-segmentation to be initiated without blocking read requests. When the new segment(s) are generated, their first key(s) quickly replace the key associated with the original segment or are inserted into unused key slots in a box of the top-level segment, under the protection of a box-level lock. If this replacement causes overflow in the box, the top-level segment undergoes re-segmentation immediately, with the β value increased by a step percentage (20% by default), up to a large upper bound (90%). The α value is always 0% for top-level segments, and the

β value of the root segment is always 100%. The lock protection mechanism for top-level re-segmentation is the same as that for bottom-level segments.

In the concurrency control design, Epoch-Based Reclamation (EBR) [9] is applied in three scenarios to enable safe lock-free access before a write-exclusive lock is acquired. This ensures that shared data can be safely modified without other threads concurrently accessing it. The first scenario is to guarantee that no writers are in a segment before its re-segmentation thread acquires an exclusive lock. The second scenario ensures that when a new overflow box is created due to the full occupancy of an existing overflow box, no readers or writers are accessing the existing overflow box. The third scenario occurs at the end of a re-segmentation, when the memory occupied by the old segment needs to be reclaimed. LiBox employs EBR to ensure that no readers are accessing the old segment when reclaiming its memory space.

3 PERFORMANCE EVALUATION

We have implemented a prototype of LiBox. The goal of the evaluation is to assess how effectively its algorithmic advantages, including highly efficient model functions, minimal last-mile search, and fine-grained locking, translate into real-world performance and scalability improvements. To this end, we extensively compare LiBox with several state-of-the-art learned and non-learned index structures. Specifically, the learned indexes used for comparison are ALEX [7], LIPP [32], XIndex [25], and SALI [11]. The non-learned indexes include the B$^+$-tree [5] and ART [16]. To isolate the performance impact of AVX-512, we also develop a variant of LiBox without AVX-512 support, named LiBox_noAVX. In this variant, keys within each box are maintained in sorted order, and binary search is used to locate a key.

- ALEX is a learned index designed to support read–write workloads. It employs gapped arrays, in which empty slots are strategically reserved and later populated to accommodate newly inserted keys at locations close to those predicted by the learned model. When combined with exponential search, ALEX effectively reduces the cost of local search. In this work, we use its multi-threaded implementation, denoted ALEX+ [31], enabling a fair comparison of scalability.
- LIPP is a learned index that provides precise position prediction. By using its model to directly determine key placement, LIPP eliminates in-node search. When key conflicts occur at a predicted position, the index expands its structure to accommodate the conflicting keys. Similar to ALEX, we use its multi-threaded implementation, denoted LIPP+ [31], in our evaluation.
- XIndex [25] is a learned index designed to improve write efficiency by associating each key-range group with a delta buffer for newly written keys. It employs a two-phase compaction mechanism that merges buffered data into the main tree-based index structure in the background, without blocking concurrent user requests.
- SALI [11] is a framework designed to improve the scalability of learned indexes. It lightweightly tracks access statistics and adaptively evolves the index structure (node evolution) according to workload skewness, achieving higher performance and improved scalability. SALI uses LIPP as its underlying base index.
- B$^+$-tree is one of the most widely used traditional index structures, featuring a balanced hierarchical organization of internal and leaf nodes. It supports dynamic updates through node splits and merges while preserving logarithmic search complexity. We use a multi-threaded B$^+$-tree implementation [29]
- ART [16] is a non-learned index built on an adaptive radix tree. It is highly optimized and widely recognized for its strong performance [1, 3, 6, 33]. We use its multi-threaded implementation, ART+ [17], in the evaluation.

The source code of the above indexes is obtained from a comprehensive index comparison study [31] and is publicly available [4]. We use the default configuration settings provided by each index in our experiments. For LiBox, the default values of its accumulative overflow ratio α and underflow ratio β per segment are 10% and 50%, respectively. Additionally, the thresholds for the overflow-box ratio γ and the maximum overflow-box list length L, which trigger re-segmentation, are set to 20% and 2, respectively. These default settings are used unless stated otherwise.

All experiments are conducted on a Supermicro ASG-1115S-NE3X12R server equipped with a single-socket AMD EPYC 9634 processor featuring 84 cores with a maximum frequency of 3.70 GHz and 168 GB of DRAM. Simultaneous multithreading (SMT) is disabled, and all experiments are restricted to a single NUMA node. The operating system is Ubuntu Linux 24.10.

3.1 The Datasets

In the evaluation, we selected datasets from diverse sources with varying key distributions, representing different levels of *hardness* for a learned index to model the distribution and organize keys effectively. The datasets and their descriptions are listed in Table 1. Among them, W048 and Msr_web are derived from I/O traces collected from a production VMware environment by CloudPhysics's caching analytics service [26] and an enterprise server workload collected by Microsoft Research Cambridge [22], respectively. In these I/O traces, keys correspond to (virtual) disk block addresses, also known as logical block addresses (LBAs). File systems and database systems commonly maintain in-memory indexes over such keys to enable fast data access in buffer caches. Because accessed LBAs are often contiguous, these datasets exhibit highly sequential access patterns with long runs of consecutive keys. This behavior is especially pronounced in W048, where over 95% of keys belong to sequences longer than 1,000 consecutive LBAs. Msr_web is less sequential, exhibiting frequent gaps where LBAs are absent from the trace. Overall, both datasets are relatively friendly to accurate modeling by learned indexes.

The remaining three datasets (Longitudes, Genome, and OSM) represent significantly harder workloads for learned indexes. The notion of *hardness*, introduced in prior comparative studies of learned indexes [31], quantitatively captures the non-linearity of a dataset's key-distribution CDF, at both global and local scales.

Table 1: Datasets used in experiments

Dataset	Description	# of Unique Keys
W048	Set of virtual disk LBAs collected by CloudPhysics in production VMware environments[26]	5,458,867
Msr_web	Set of disk LBAs on an enterprise server collected by Microsoft Research Cambridge [22].	8,974,377
Longitudes	The longitudes of locations around the world from OpenStreet Maps[2].	200,000,000
Genome	The loci pairs in human chromosomes[24]	200,000,000
OSM	The uniformly sampled OpenStreetMap locations [13]	200,000,000

For example, the Longitudes dataset uses longitude values of geographic locations from OpenStreetMap [2] as its key set. Points of interest in geographic data are often distributed highly irregularly, making accurate modeling difficult. Representing an even more challenging case, the OSM dataset projects multi-dimensional spatial data into a one-dimensional key space, thereby destroying any regularity present in individual dimensions [13]. This dataset is notoriously difficult for learned indexes: prior studies show that traditional index structures frequently outperform learned indexes on OSM [21]. Including such a worst-case dataset is essential. If a learned index cannot consistently match or exceed the performance of traditional indexes, the resulting performance uncertainty would make practitioners reluctant to deploy learned indexes in production systems.

Except for W048 and Msr_web, which contain 4-byte unsigned integer keys, all other datasets use 8-byte unsigned integer keys. For uniformity, we store all keys using 8 bytes, regardless of their original size. The value size is fixed at 8 bytes across all experiments.

3.2 The Workloads

The workloads are designed to capture varying read-write ratios in order to evaluate how index performance responds to changing workload compositions. For each dataset, we first randomly shuffle all keys to generate an unordered key sequence. We then construct a reference trace based on this sequence.

- Read-Only (100% Read): Keys are first bulk-loaded into the index, after which each key is read once in a random order.
- Read–Write Balanced (50% Read, 50% Write): Half of the unique keys are randomly selected for bulk loading. After bulk loading, the remaining 50% of the keys are inserted in a random order. Each write operation is immediately followed by a read, where the read key is randomly chosen from the keys currently present in the index.
- Write-Only (0% Read): Half of the unique keys are randomly selected for bulk loading. After bulk loading, the remaining 50% of the keys are inserted in a random order.

We make a minor modification to the *Balanced* workload to generate additional read-write ratios. Instead of following each write with a single read, we issue four read operations after each write to create an 80% read and 20% write workload. Conversely, by issuing one read after every four writes, we obtain a 20% read and 80% write workload. Other read-write ratios are generated similarly.

3.3 Throughput and Scalability

Figure 4 shows the single-thread throughput of the indexes under different workloads. As observed, LiBox achieves higher throughput than the other indexes in most experiments. Figure 5 presents the throughput under varying numbers of threads, with each thread runs on a separate core. A striking observation across these results is that LiBox substantially outperforms the other indexes, particularly at higher thread counts, and that its throughput scales well with increasing parallelism.

In particular, we make three observations. First, as expected, harder data sets make learned indexes less competitive. We use the RO (read-only) workload as an example. Except for the hardest data set, OSM, there is always at least one learned index other than LiBox that outperforms ART+, a highly optimized traditional index. Under OSM, however, LiBox is the only learned index that outperforms ART+. Notably, the B^{+}-tree, which otherwise exhibits the lowest throughput among all indexes, outperforms XIndex on OSM, further highlighting the challenge this data set poses to learned indexes.

Second, servicing write requests substantially offsets the performance advantages of learned indexes. As the workload shifts from RO (read-only) to BAL (read–write balanced) and then to WO (write-only), the relative performance advantage of learned indexes consistently diminishes. Across all data sets, ART+ steadily rises in the throughput ranking, surpassing nearly all learned indexes except LiBox. Under WO workloads, even the B^{+}-tree consistently outperforms XIndex and LIPP+. Figure 6 further corroborates this observation by showing how throughput varies with increasing read ratios using 40 threads. Compared with traditional indexes, all learned indexes exhibit a common trend of improved performance as the read ratio increases, whereas traditional indexes are less sensitive to workload mix. As a result, traditional indexes often outperform learned indexes under write-dominant workloads.

Third, LiBox exhibits better and more consistent scalability than the other indexes. For example, even under the WO workload, when the number of threads doubles from 40 to 80, LiBox's throughput increases by 69%, 57%, 37%, 47%, and 62% for the W048, Msr_web, Longitudes, Genome, and OSM data sets, respectively. Compared with all other indexes, LiBox demonstrates impressive scalability across all workloads. Other learned indexes tend to scale less effectively. In particular, write-intensive workloads severely limit LIPP+'s scalability. Under both BAL and WO workloads, LIPP+'s throughput shows little to no improvement as the number of threads increases. This behavior stems from its design: each node maintains its own model computed using the Fastest Minimum Conflict Degree (FMCD) algorithm and can grow to a very large size (up to one million keys). Consequently, locks must be applied at coarse granularity, and once a writer acquires a lock on a node, all concurrent accesses to that node are blocked. SALI improves LIPP+ by adaptively restructuring (sub)trees based on access intensity, thereby increasing concurrency and throughput. Nevertheless, its

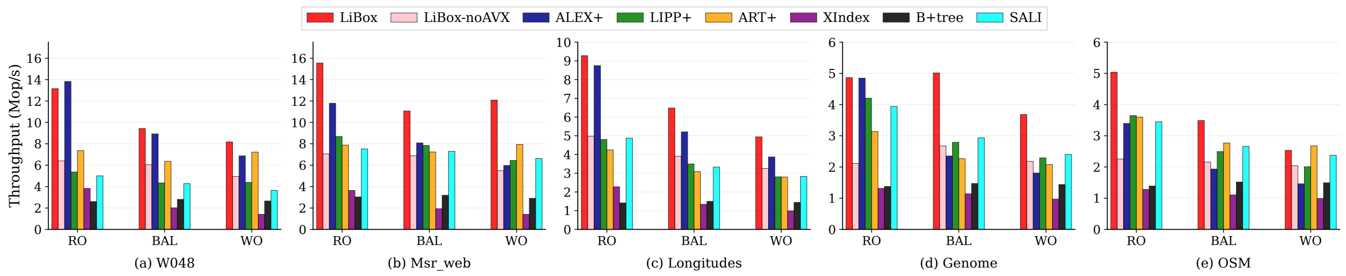


Figure 4: Single-threaded throughput of read-only (RO), read–write balanced (BAL), and write-only (WO) workloads across different index structures.

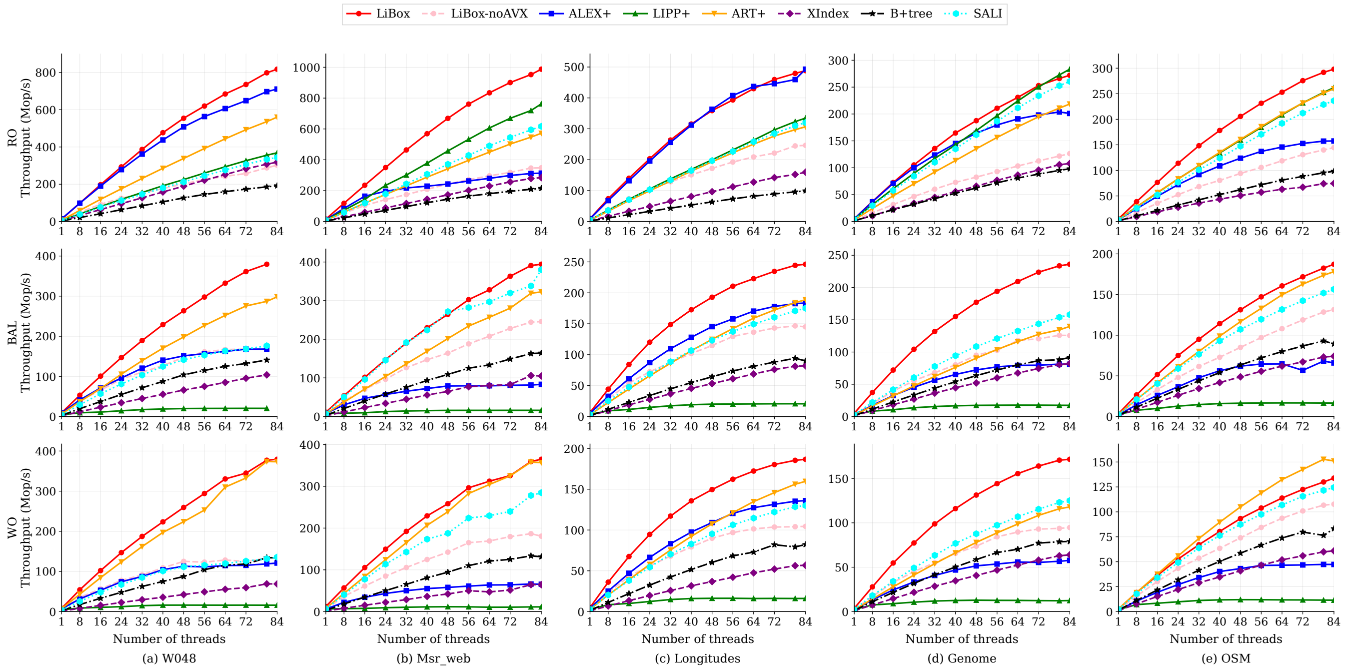


Figure 5: Throughput under read-only (RO), read–write balanced (BAL), and write-only (WO) workloads across different index structures with varying numbers of threads.

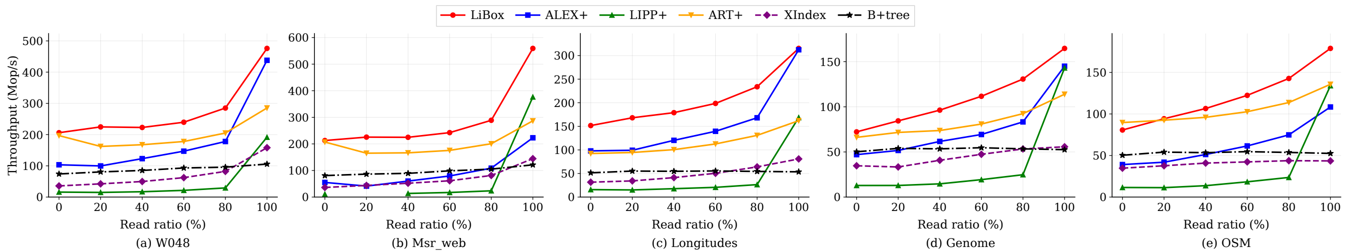


Figure 6: Throughput of the indexes under workloads with different read ratios. The thread count is fixed at 40.

throughput remains significantly lower than that of LiBox in most cases. LiBox, in contrast, applies write locks at the box granularity, where each box holds only 64 keys. Even during structure modification operations (SMOs) such as re-segmentation, read threads are allowed to continue accessing the original segment. With its

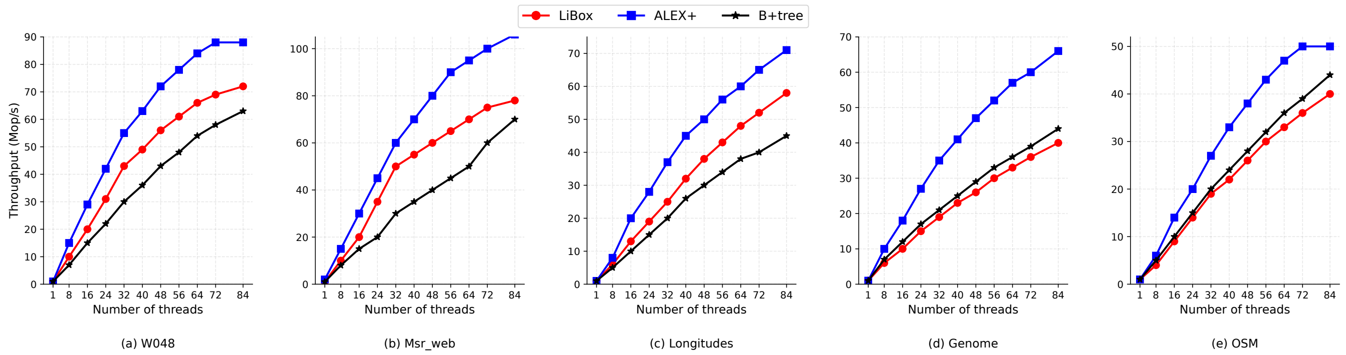


Figure 7: Throughput of range search queries. The query size is 100 keys.

box-based design, the overhead traditionally associated with writes, concurrency control, and retraining, often regarded as the Achilles' heel of learned indexes, is substantially reduced. In fact, LiBox provides stronger support for write-heavy and highly concurrent workloads than even traditional index structures.

Fourth, the throughput of LiBox_noAVX is substantially lower than that of LiBox, although in some cases it still demonstrates competitive performance compared with several other indexes. These results indicate that the use of SIMD instructions is essential to LiBox's performance advantage.

Table 2: Number of segments in each level after a bulk loading in the read-only workload.

Data Sets	# of Segments	
	Top Level	Bottom Level
W048	1	57
Msr_web	2	81
Longitudes	2	2,369
Genome	1,700	1,022,494
OSM	4,331	839,869

To better understand how LiBox achieves its performance advantage, Table 2 reports the number of segments at the top and bottom levels of the index after fully bulk loading each data set. As shown, for data sets with more regular key distributions, the top level contains only a small number of segments. Notably, the Longitudes data set exhibits little global regularity, such as linear CDF curves. However, LiBox does not evaluate or exploit regularity at the key granularity. Instead, it operates at the box granularity. Combined with its tolerance for underflow and overflow, LiBox effectively absorbs local irregularities and transforms them into strong linearity at a coarser granularity. Even for hard data sets such as Genome and OSM, which contain around one million bottom-level segments, the first keys of these segments can be indexed by only a few thousand top-level segments. The auxiliary data structure used to index and locate top-level segments is therefore very small and can reside entirely in the CPU cache. As a result, when overflow boxes are not involved, a lookup requires memory access to only two boxes – one at the top level and one at the bottom level.

LiBox's outperforming over traditional indexes across diverse data sets, workloads with varying read/write ratios, and different levels of concurrency is critically important for the widespread adoption of learned index techniques in industry. Because indexes are among the most performance-critical components in data management systems, any uncertainty in performance behavior or potential weaknesses relative to traditional indexes can discourage practitioners from replacing existing index structures in production systems with learned alternatives.

Range Search. As a sorted index, LiBox supports range search operation. To evaluate its range search performance, we revise the RO (read-only) workload by replacing each point lookup with a range query that retrieves 100 consecutive keys following the queried key. We include two learned indexes (LiBox and ALEX+) and one traditional index (B^+-tree) in this evaluation, as their source code readily supports range queries. The performance results are shown in Figure 7. Unlike the results for point-key operations, LiBox does not exhibit a clear performance advantage in range search. One of LiBox's primary design strengths is its ability to perform (mostly) exact key matching within a box using a single SIMD instruction for point lookups. Range search, however, does not focus on locating the existence of a single key in a box. Instead, it requires LiBox to perform local sorting and multiple AVX-512 operations to determine whether keys fall within the query range for the first and last boxes that partially overlap with the specified range. These additional operations reduce LiBox's relative advantage. As future work, we plan to explore opportunities to pre-sort keys in boxes in read-only segments, enabling binary search to accelerate range queries.

Memory Space. One major advantage often claimed by learned indexes is their potentially much smaller memory footprint, as they can compute the location of a key rather than maintaining an explicit internal index and traversing a path to a leaf node where the key-value pair is stored. In practice, however, irregular key distributions usually necessitate a hierarchical, tree-structured model to guide the search to the correct region of the key space. Each internal node in such a hierarchy requires memory to store routing keys and pointers to lower-level nodes. Moreover, supporting dynamic updates further increases memory consumption. Learned indexes typically allocate additional space, such as delta buffers or unoccupied slots in key arrays, to accommodate new insertions. LiBox follows a similar principle: it deliberately allows a certain percentage of slots within each segment to remain empty. This reserved

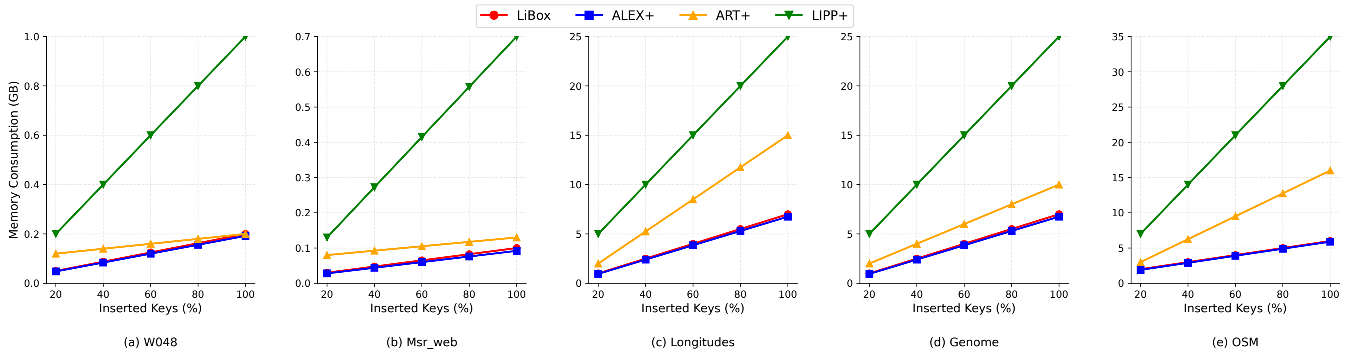


Figure 8: The memory footprint size with the growth of an index

space enables segments to grow smoothly, facilitates fast insertions, and reduces the frequency of re-segmentation operations.

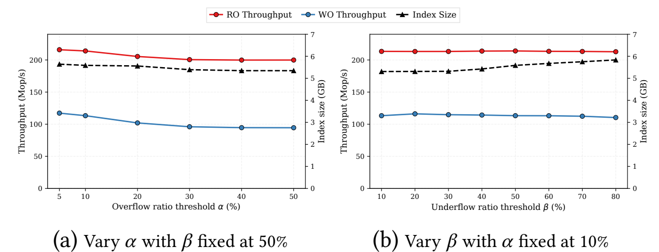
Figure 8 shows the memory footprint of each index as new keys are continuously inserted until all keys from a data set are loaded. For this experiment, we include only indexes that explicitly report their memory consumption: ALEX+, ART+, and LIPP+. As shown in the figure, updatable learned indexes do not consistently outperform traditional indexes in terms of space efficiency. In particular, LIPP+ more than doubles the memory footprint of ART+, reflecting its design choice to aggressively trade space for performance. LIPP+ allocates a large number of empty slots in its nodes to support model-guided key placement while minimizing collisions. While such space-performance trade-offs are often necessary in learned index designs, the resulting memory overhead must remain well bounded to be practical. In contrast, the memory footprints of LiBox and ALEX+ are consistently smaller than that of the widely used traditional index ART+. Although both ALEX+ and LiBox deliberately reserve empty space to facilitate efficient updates (using gapped arrays in ALEX+ and controlled box underflow in LiBox) and therefore consume a comparable amount of memory, LiBox achieves higher throughput (see Figure 5). This result demonstrates that the presence of empty slots in LiBox segments is well justified, as it enables significant performance gains without incurring excessive memory overhead.

Impact of Underflow and Overflow Thresholds. As explained earlier, the thresholds of the accumulative overflow and underflow ratios for a segment (α and β, respectively) directly affect how segments are formed and, consequently, the overall structure of the index. These parameters also have important implications for read and write performance, and for the memory footprint of the index.

To understand these impacts, we select Genome, which lies in the middle of the *hardness* spectrum among the evaluated data sets, and vary its α and β values. As shown in Table 3, when β is fixed at 50% and the overflow ratio threshold α is varied from 5% to 50%, a larger α allows LiBox to adopt a larger window size and thus place more keys into each box. With a fixed box capacity, this increases the likelihood that keys overflow beyond regular boxes. Accordingly, the number of overflow boxes increases with α, as shown in the table. As more keys reside in overflow boxes, accessing them incurs additional indirections, leading to longer access times. Figure 9(a) shows the throughput of the index under read-only (RO) and write-only (WO) workloads for different α values. For both workloads, throughput decreases as α increases, since searches must traverse more overflow boxes. However, the performance degradation is moderate because overflow boxes are relatively evenly distributed across regular boxes. While increasing α leads to more overflow boxes, it also reduces the number of regular boxes by compacting more keys into each box. As a result, the total number of boxes decreases only slightly, and the overall index size (i.e., memory footprint) is reduced moderately. Based on this trade-off, we set the default α value to 10%, which effectively limits the number of overflow boxes while maintaining good performance.

Table 3: LiBox structure with different overflow threshold α (underflow ratio threshold β is 50%)

α (%)	# Seg in the Top Level	# Seg in the Bottom Level	# Regular Boxes	# Overflow Boxes
5	1,218	1,282,918	4,136,374	450,080
10	1,700	1,022,494	3,925,924	687,118
20	2,443	626,247	3,600,130	1,097,906
30	1,922	369,454	3,260,593	1,358,690
40	1,771	332,630	3,200,209	1,389,183
50	1,768	332,393	3,199,737	1,389,417



(a) Vary α with β fixed at 50% (b) Vary β with α fixed at 10%

Figure 9: Impacts of the α and β thresholds on performance and index size (40 threads)

Table 4 shows the impact of varying the underflow ratio threshold β while fixing α at 10%. As expected, a larger β allows regular boxes to be more sparsely populated. Given a fixed number of keys in the data set, this requires more regular boxes to store the data. Interestingly, the number of overflow boxes also increases. This

Table 4: LiBox structure with different underflow threshold β (overflow ratio threshold α is 10%)

β (%)	# Seg in the Top Level	# Seg in the Bottom Level	# Regular Boxes	# Overflow Boxes
10	246	1,528,519	3,697,200	494,643
20	345	1,492,537	3,696,284	510,690
30	1,114	1,234,610	3,674,653	623,854
40	1,631	1,076,766	3,753,397	692,731
50	1,700	1,022,494	3,925,924	687,118
60	1,752	997,314	3,994,514	706,523
70	1,786	979,110	4,061,750	713,709
80	1,837	967,224	4,131,951	721,685

behavior arises because the α and β thresholds are not designed to bound the number of keys or empty slots in individual boxes. Instead, both thresholds are defined over accumulative ratios at the segment level, bounding the total amount of overflow and underflow across all boxes in a segment. By setting a larger β, LiBox tends to admit sparser windows into a segment, which lowers the accumulative overflow ratio. This, in turn, allows denser windows to be admitted later without violating the overflow threshold. However, as a side effect, individual boxes become more likely to overflow. Because the increase in overflow boxes is moderate, the resulting throughput reduction and index size increase are also small, as shown in Figure 9(b). Based on this trade-off, we set the default β value to 50%, which does not significantly increase the memory footprint while leaving more free space in boxes to better accommodate future highly skewed write workloads.

4 RELATED WORKS

Learned indexes, first introduced by Kraska et al. [15], leverage machine learning models to augment or replace traditional index structures, such as B$^+$-trees, achieving substantial improvements in performance and space efficiency. Since then, numerous studies have extended this paradigm to support dynamic workloads, concurrent access, and multi-dimensional datasets.

Traditional and Model-based Learned Indexes. Classical database indexing has long relied on tree-based data structures, including B$^+$-trees [5] and Adaptive Radix Trees (ART) [16]. These structures provide robust and predictable performance, but do not explicitly exploit underlying data distribution patterns. The emergence of learned indexes [15] marked a fundamental shift, framing indexing as a model-based prediction problem that estimates key positions in sorted data. The Recursive Model Index (RMI) [15] demonstrated up to a 3× improvement in lookup performance over B$^+$-trees while reducing memory usage by an order of magnitude.

A fundamental challenge in learned index design is the trade-off between model complexity and prediction accuracy. Early designs, such as the Learning Index Framework (LIF) [15], explored complex models, including neural networks that may require GPU acceleration, or deep hierarchies of simpler models to incrementally refine predictions. Both approaches increase computational overhead and memory accesses, potentially eroding the performance advantages over traditional indexes. Consequently, subsequent work, such as PGM [8] and RadixSpline [14], introduced efficient construction techniques that fit piecewise linear models in a single pass, making learned indexes more practical. However, even a large number of piecewise models may fail to accurately capture highly complex key distributions. LiBox addresses this challenge by fitting ultra-simple linear models over boxes rather than individual keys, intentionally confining distribution irregularity within boxes.

Updatable Learned Indexes. The original RMI design is limited to static datasets and read-only workloads. To overcome these limitations, ALEX [7] introduced an architecture that combines learned models with gapped arrays to support efficient insertions. LIPP [32] further refined this approach by maintaining precise position predictions in leaf nodes, thereby reducing update overhead. Other notable designs include CARMI [36], which employs cost-based node-type selection, XIndex [25], which targets efficient concurrent writes, and FITing-Tree [10], which proposes an index structure that balances model complexity and prediction accuracy. A persistent challenge in these designs is the need to maintain newly inserted keys in sorted order alongside existing keys, which is necessary for efficient lookup but incurs high update costs. By using SIMD instructions for in-box search, LiBox doesn't need to keep keys in the same box sorted.

Using overflow buffers is a common technique for accommodating newly inserted keys in updatable learned indexes, such as ALEX [7], APEX [20], and LIPP [32]. These indexes extend their structures by creating child nodes at lower levels of the tree to absorb insertions. LiBox similarly employs overflow boxes to store keys that cannot be accommodated in a regular box. Conceptually, a segment in LiBox corresponds to a node in the tree-based structures of these indexes. Because a segment typically has a much larger capacity than a tree node, overflows occur less frequently, thereby reducing their impact on performance.

Specialized Applications and Extensions. Learned indexes have been extended to a wide range of data modalities. In spatial indexing, models such as LISA [18] and related approaches [23, 27] have demonstrated promising performance on multi-dimensional datasets. For string and text data, SIndex [28], LITS [34], and more recent work such as Kim et al. [12] report substantial performance gains. Multi-dimensional learned indexing has also been systematically evaluated in [19]. On the theoretical side, studies such as [35] establish complexity bounds for learned-index query time under various data distributions. The capability introduced by LiBox is orthogonal to these efforts, and as future work, LiBox can be extended to support additional key types and data modalities.

5 CONCLUSIONS

In this paper, we present LiBox, a novel learned index that reduces the overhead of model evaluation and last-mile search by optimizing data placement and organization to handle irregular key distributions, while leveraging SIMD-based acceleration. Our comprehensive experimental evaluation shows that LiBox consistently outperforms state-of-the-art learned indexes and traditional index structures across a wide range of workloads, while scaling efficiently and maintaining a low memory footprint.

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] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. 2015. A comparison of adaptive radix trees and hash tables. In *2015 IEEE 31st International Conference on Data Engineering*. 1227–1238. https://doi.org/10.1109/ICDE.2015.7113370

[2] Amazon Web Services. 2025. OpenStreetMap on AWS. https://registry.opendata.aws/osm. Accessed: May 2025.

[3] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 521–534.

[4] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. GRE Index Benchmark Suite. https://github.com/gre4index/GRE. Accessed: May 2025.

[5] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[6] Dean De Leo and Peter Boncz. 2019. Packed Memory Arrays - Rewired. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 830–841. https://doi.org/10.1109/ICDE.2019.00079

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[8] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.

[9] Keir Fraser. [n.d.]. Practical lock-freedom. *PhD thesis, Cambridge University Computer Laboratory* ([n. d.]). https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[10] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 international conference on management of data*. 1189–1206.

[11] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALI: A Scalable Adaptive Learned Index Framework based on Probability Models. *Proc. ACM Manag. Data* 1, 4, Article 258 (Dec. 2023), 25 pages. https://doi.org/10.1145/3626752

[12] Minsu Kim, Jinwoo Hwang, Guseul Heo, Seiyeon Cho, Divya Mahajan, and Jongse Park. 2024. Accelerating String-Key Learned Index Structures via Memoization-Based Incremental Training. *Proc. VLDB Endow.* 17, 8 (April 2024), 1802–1815. https://doi.org/10.14778/3659437.3659439

[13] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014* (2019).

[14] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management*. 1–5.

[15] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[16] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[17] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 1–8.

[18] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2119–2133.

[19] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. 2025. How good are multi-dimensional learned indexes? An experimental survey. *The VLDB Journal* 34, 2 (2025), 1–29.

[20] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: a high-performance learned index on persistent memory. *Proc. VLDB Endow.* 15, 3 (Nov. 2021), 597–610. https://doi.org/10.14778/3494124.3494141

[21] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13. https://doi.org/10.14778/3421424.3421425

[22] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Trans. Storage* 4, 3, Article 10 (Nov. 2008), 23 pages. https://doi.org/10.1145/1416944.1416949

[23] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.

[24] Suhas SP Rao, Miriam H Huntley, Neva C Durand, Elena K Stamenova, Ivan D Bochkov, James T Robinson, Adrian L Sanborn, Ido Machol, Arina D Omer, Eric S Lander, et al. 2014. A 3D map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell* 159, 7 (2014), 1665–1680.

[25] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*. 308–320.

[26] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 487–498. https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

[27] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned index for spatial queries. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 569–574.

[28] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 17–24.

[29] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. 473–488.

[30] Wikipedia. 2025. AVX-512. https://en.wikipedia.org/wiki/AVX-512. Accessed: May 2025.

[31] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proc. VLDB Endow.* 15, 11 (July 2022), 3004–3017. https://doi.org/10.14778/3551793.3551848

[32] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable learned index with precise positions. *Proc. VLDB Endow.* 14, 8 (April 2021), 1276–1288. https://doi.org/10.14778/3457390.3457393

[33] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A comprehensive performance evaluation of modern in-memory indices. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 641–652.

[34] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3415–3427.

[35] Sepanta Zeighami and Cyrus Shahabi. 2023. On distribution dependent sub-logarithmic query time of learned indexing. In *International Conference on Machine Learning*. PMLR, 40669–40680.

[36] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: a cache-aware learned index with a cost-based construction algorithm. *Proc. VLDB Endow.* 15, 11 (July 2022), 2679–2691. https://doi.org/10.14778/3551793.3551823

[37] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 51–64. https://www.usenix.org/conference/fast21/presentation/zhong