LINE: A Learned Index with Group-Enhanced Leaves and Cache-Optimized Inner Tree

LEYING CHEN and SHIMIN CHEN[*], SKLP, ACS, Institute of Computing Technology, CAS, China and University of Chinese Academy of Sciences, China

Compared to traditional indices, learned indices exploit the underlying data distribution to construct trees with larger node sizes and lower tree heights, thereby achieving good performance. However, the underlying data characteristics can significantly impact the index performance, and the large tree nodes incur heavy structure modification operations (SMOs). This leads to three main challenges for learned indices to achieve more stable performance: local data hardness, global data skew, and worst-case tail latency.

In this paper, we propose LINE, a learned index that consists of a set of group-enhanced leaf nodes and a cache-optimized inner tree for addressing the three challenges. First, a group-enhanced leaf node employs a linear model to map keys in the leaf node into groups. We create variable-sized groups and perform in-group hashing to handle local data hardness. Second, the bulkload data set is divided into leaf nodes and the leaf boundary keys are inserted into an inner tree during index construction. We perform cache-aware error bound selection with a segment-based PLA (piece-wise linear approximation) algorithm to reduce the inner tree size. In this way, we simplify the inner tree structure and improve its cache performance for dealing with global data skew. Finally, we decompose lengthy SMOs into group-wise migrations for reducing the worst-case tail latency. Our experiments on real-world and synthetic data sets show that compared to state-of-the-art learned indices, LINE achieves up to 5.8x performance improvement while dramatically reducing the worst-case tail latency.

CCS Concepts: • **Information systems** → **Data access methods**; **Point lookups**; **Data scans**.

Additional Key Words and Phrases: LINE, learned index, group-enhanced leaves, cache-optimized inner tree

ACM Reference Format:
Leying Chen and Shimin Chen. 2026. LINE: A Learned Index with Group-Enhanced Leaves and Cache-Optimized Inner Tree. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 203 (June 2026), 27 pages. https://doi.org/10.1145/3802080

1 Introduction

Learned indices have emerged as a prominent research topic in recent years [2, 3, 6–12, 14–17, 20–23, 27–30, 32, 33, 35–41, 44–50]. They train models that learn the cumulative distribution function (CDF) of the data, and exploit the models to predict the location of a key for fast tree node search. By exploiting the underlying data distribution, learned indices achieve superior performance compared to traditional tree-based structures. Model-based predictions enable larger nodes by improving costly comparison-based key searches with efficient model inferences, reducing both instruction counts and cache misses. Larger nodes lead to shallower trees, decreasing the number of node visits for index operations.

[*]Shimin Chen is the corresponding author.

Authors' Contact Information: Leying Chen, chenleying19z@ict.ac.cn; Shimin Chen, chensm@ict.ac.cn, SKLP, ACS, Institute of Computing Technology, CAS, Beijing, China and University of Chinese Academy of Sciences, Beijing, China.

ACM 2836-6573/2026/6-ART203
https://doi.org/10.1145/3802080

Despite these innovations, three key challenges remain in order to achieve more stable index performance for learned indices:

- *Local hardness* represents the non-linearity of the key distribution in leaf nodes. Larger local hardness of the underlying data set diminishes the effectiveness of leaf prediction models and amplifies reliance on collision-resolving structures, such as last-mile search, overflow buffers, or new child nodes, incurring performance overhead.
- *Global skew* refers to i) global hardness, i.e., the non-linearity of the global key distribution, and ii) skewed empty gaps between global key segments. The latter has not been studied before. Higher global skew often causes more complex inner structures in learned indices, deteriorating the index performance.
- *Worst-case tail latency* results from expensive SMOs in learned indices. Compared to traditional indices, larger tree nodes incur more heavy-weight SMOs. In addition, more expensive operations, such as subtree rebuilding and model retraining, are often performed to support changing data distributions.

In this work, we propose LINE, a novel in-memory updatable learned index designed to address all three challenges. First, to tolerate local hardness, LINE constructs *group-enhanced leaf nodes* that map keys to variable-sized groups using the leaf model. Groups perform hashing to handle in-group data skews, while enabling fine-grain concurrency control and leaf SMOs. Second, to tolerate global skew, LINE builds a *cache-optimized inner tree*. During index construction, LINE partitions the underlying data set into leaf nodes using our SPLA (segment-based piece-wise linear approximation) algorithm with cache-aware error bound selection. Then, it creates a skew-tolerant root for supporting skewed gaps between global key segments. Finally, LINE reduces the worst-case tail latency by decomposing lengthy leaf SMOs into *group-wise migration* steps.

We show that LINE can achieve bounded performance for normal point and scan operations regardless of data distribution. Moreover, our experiments show that compared to state-of-the-art learned indices, LINE achieves speedups of 1.02-4.0x, 1.1-5.8x, and 1.1-3.6x on read-only, read-write, and write-only workloads, respectively, while maintaining good scan performance and moderate memory consumption. Furthermore, LINE reduces the worst-case tail latency by up to 5713x, 65x, 18x, 4x, and 44x relative to PGM [9], ALEX [7], LIPP [44], DILI [22], SALI [11], respectively.

Contributions. First, we identify three design challenges, and discuss existing learned indices with regard to the challenges. Second, we propose LINE to achieve good bounded performance for normal point operations and range scans with moderate tail latency. Finally, we perform a comprehensive experimental study comparing LINE with state-of-the-art learned indices using both real-world and synthetic data sets with diverse data characteristics. Our implementation is publicly available.[1]

Outline. After the introduction, Section 2 reviews existing solutions with regard to the design challenges. Section 3 overviews LINE. Then, Section 4 and 5 present the leaf and the inner tree designs respectively. Section 6 analyzes LINE's cost. Section 7 reports the experimental results. Finally, Section 8 concludes the paper.

2 Challenges and Related Work

Learned indices have been extensively studied in the literature [2, 3, 6–12, 14–17, 20–23, 27–30, 32, 33, 35–41, 44–50]. In this paper, we focus on in-memory updatable learned indices for fixed-sized keys.

[1]https://github.com/schencoding/line

Compared to traditional index structures (e.g., B+-Trees and radix trees), learned indices exploit the distribution of the indexed data set to achieve superior performance. However, the characteristics of the data set can impact the stability of the performance of learned indices; data sets with different characteristics can lead to very different index performance. We consider local hardness and global skew as two design challenges to reflect such impact. Moreover, learned indices often create much larger nodes (e.g., up to 16MB in ALEX) than traditional indices, thereby reducing the number of tree levels. Hence, SMOs (structure modification operations) can incur much larger cost, causing large variance in the latency of index accesses. We consider tail latency as the third design challenge.

Next, we discuss each design challenge. We examine six representative updatable learned indices and discuss how well they handle the challenges. Specifically, we choose three well-studied indices (i.e., PGM [9], ALEX [7], and LIPP [44]) and three recently proposed indices (i.e., DILI [22], Hyper [48], and Chameleon [12]). (Our experiments in Section 7 compare PGM, ALEX, LIPP, and DILI. Hyper was omitted as no implementation was available, and Chameleon was excluded due to its dependence on GPU resources.)

Local Hardness. The metric of hardness was introduced in the GRE work [43]. Given a set of keys and an error bound ε, a PLA (piecewise linear approximation) model is computed such that the error of the linear approximation of each key segment is bounded by ε. Hardness is defined as the number of resulting PLA segments. Local hardness is the number of segments when ε=32. It often characterizes non-linearity of the keys *within leaf nodes*. This is because learned indices often create large leaf nodes by (explicitly or implicitly) allowing the errors of leaf models to be larger.

Data sets can have very different local hardness. Table 1 lists the characteristics of the data sets used in this study (cf. Section 7.1 for more details). We see that there is over 15x difference between the highest and the lowest local hardness values. Moreover, Table 1 also reports the 99th percentile of the conflict degrees (in ALEX's leaves), where the conflict degree [45] is the number of keys mapped to the same position. This confirms that larger local hardness results in more significant key collisions.

We examine the leaf design of the six representative indices with regard to local hardness. (The discussion is summarized in Table 2.)

- PGM [9] partitions the bulkload keys into leaf key segments by computing a PLA model. Because of the leaf error bound, in-leaf search and scan achieve good bounded performance. However, writes are handled by building a series of indices with increasing sizes. Consequently, operations (e.g., search or scan) have to process multiple trees, exacerbating the costs.
- ALEX [7] performs exponential search after prediction in leaves. However, when local hardness intensifies, the distance between predicted and actual positions increases, triggering significant local search costs. In contrast, scan performance benefits from sorted keys in leaves.
- LIPP [44] handles key collisions by creating a new node for each conflict. However, it can construct excessively deep trees under high local hardness which hurt the performance significantly.
- DILI [22] adopts a LIPP-like leaf structure, generating new nodes for key collisions, thereby inheriting similar limitations.
- Hyper [48] constructs leaf nodes similar to PGM (with default ε=128). Keys mapped to the same location are stored in a sorted overflow buffer. Note that the maximal number of conflicts per location is $2\varepsilon + 1$. Hence, search cost is bounded but insertions have to move half of the keys in the buffer on average to maintain the key order. Scan performance is good.
- Chameleon [12] employs error-bounded hashing to accommodate local skewness in leaves. This supports efficient point operations. However, small range scans can suffer significantly because entire leaves are unsorted.

Table 1. Data set characteristics.

	osm	**planet**	**fb**	**genome**	**covid**	**books**
Local Hardness	661K	614K	1055K	1290K	82.9K	263K
P99 Conflict Degree	11	14	25	62	4	9
Global Hardness	5495	2313	1687	1426	850	97
PGM	22MB/6	23MB/5	35MB/4	51MB/5	3.7MB/4	12MB/4
ALEX	21MB/2.7	5.6MB/2.2	5.5MB/6.7	630KB/2.0	256KB/2.0	320KB/2.0
LIPP	6.4GB/2.3	6.4GB/1.9	6.4GB/1.8	6.4GB/2.0	6.4GB/1.5	6.4GB/1.8
DILI	320B/3.3	676MB/3.3	142KB/4.2	731MB/4.0	1.4GB/3.4	1.2GB/3.3

Note: Each data set contains 200 million 64-bit keys.
LIPP: root size / avg depth; PGM, ALEX, and DILI: inner size / avg depth

Table 2. Comparison of learned index solutions.

Index	**Local Hardness (leaf nodes)**	
	Leaf Point Operation	**Scan**
PGM[9]	☹ multiple trees, leaves with error bound	☹
ALEX[7]	☹ gapped sorted array, exponential search	☺
LIPP[44]	☹ new child node for key collision	☹
DILI[22]	☹ new child node for key collision	☹
Hyper[48]	😐 sorted overflow buffer	☺
Chameleon[12]	☺ whole-leaf error-bounded hashing	☹
LINE (our work)	☺ group-enhanced leaf nodes	

Index	**Global Skew (inner nodes)**			**Tail Latency**
	Size	**Depth**	**Skewed Gaps**	
PGM[9]	😐	☹	☹	☹ multiple tree merging
ALEX[7]	😐	😐	☹	😐 SMOs up to root
LIPP[44]	☹	☺	☹	☹ subtree rebuild
DILI[22]	☹	😐	☹	☹ leaf node and subtree adjustment
Hyper[48]	😐	☺	☹	☹ leaf adjustment, subtree rebuild
Chameleon[12]	😐	😐	☹	☹ RL-model retraining
LINE (our work)	☺ cache-optimized inner tree			☺ group-wise migration

Global Skew. We consider two aspects of the skewness in the global key distribution. The first aspect is global hardness (ε=4096). It reflects the hardness to use a set of linear models to approximate the global key distribution. As shown in Table 1, like local hardness, global hardness can also vary drastically across data sets, with over 50x difference between the highest and the lowest global hardness values. Since global hardness tends to impact the non-leaf structure, we examine the complexity of the inner structure in terms of the inner size and the average tree depth in Table 1.

- ALEX [7] performs top-down bulkloading. Higher global hardness requires ALEX to partition the key range of an inner node into a larger number of segments, leading to more complex inner structures, as evidenced in Table 1.
- PGM [9] and Hyper [48] perform bottom-up bulkloading for the entire tree and the lower tree levels including the leaf level, respectively. The number of leaf nodes is influenced by local hardness. Hence, both global hardness and local hardness impact the complexity of PGM's and Hyper's inner structures. PGM achieves better search performance with lower leaf error bound.

(By default, ε=16 in GRE's PGM implementation [43]). However, this can lead to larger inner structures than that of ALEX. The depth of Hyper is smaller because of larger ε.

- LIPP [44] and DILI [22] both employ collision-driven nodes, which resolve key conflicts by creating new child nodes. LIPP builds a very large array in the root node in order to reduce the average depth, leading to huge roots shown in Table 1. The case of DILI is more intriguing. In Table 1, the reported inner size does not show clear trends. One explanation is that the reported size corresponds to the size of the internal nodes, but DILI's leaf nodes can also contain child pointers, playing the role of the inner structures to some degree. Nevertheless, the size can be very large compared to ALEX.
- Chameleon [12] integrates a top-down bulkloading approach with reinforcement learning, achieving compact tree structures.

Table 2 summarizes the discussion. The size and depth columns are based on the results in Table 1, and for Hyper and Chameleon based on their original papers.

For the second aspect, we observe that there can be significant gaps in the global key distribution. For example, in the fb data set, a number of keys are much larger than the rest of the keys; over 90% of the entire key range is empty. This large gap results in the large depths for fb in ALEX and DILI shown in Table 1, and many wasteful empty slots in LIPP's root. In general, large and skewed gaps can cause poor inner structures. Note that the skewed gaps are orthogonal to global hardness. The latter refers to the number of global key segments with ε=4096, while the former focuses on the empty key space between global segments. Unfortunately, skewed gaps have not been studied in previous works. In this work, we consider skewed gaps as an important factor of global skew.

Tail Latency. Compared to traditional indices, SMOs such as node expansion and node split are typically more expensive in learned indices due to their larger node sizes. Moreover, learned indices often rebuild subtrees with retrained models to accommodate changing data distributions. Such SMOs are often protected by node locks, which block normal operations to the affected nodes. As a result, the (worst-case) tail latency can be very large, making learned indices less suitable for latency-sensitive applications despite their good average-case index performance.

We consider the most severe SMOs in the representative learned indices. PGM [9] merges (multiple) trees for handling inserts, which incurs significant overhead. Since new writes conflict with ongoing tree merging, writes need to wait for tree merging to complete. In ALEX [7], structural changes can propagate up to the root. LIPP [44] initiates a subtree rebuild when data distribution deteriorates, making part of the index temporarily unavailable. DILI [22] addresses read performance degradation by adjusting its (subtrees of) leaf nodes. Hyper [48] adjusts a leaf if the collision degree exceeds a pre-defined threshold, and rebuilds a subtree if the number of leaves in the subtree has doubled. Chameleon [12] performs online retraining of the reinforcement learning models, which can be computationally expensive and exacerbate the tail latency.

Motivation. As shown in Table 2, we see that none of the existing learned indices consistently excel for the three challenges. This motivates us to pursue a design to address all the challenges.

3 LINE Overview

We propose LINE (Learned Index with group-enhaNced lEaves and cache-optimized iNner trEe). Figure 1 depicts LINE's structure. It consists of a cache-optimized inner tree and a set of group-enhanced leaf nodes. In the following, we overview LINE's data structure and operations with an emphasis on our proposed techniques to address the three design challenges.

Local Hardness: Group-Enhanced Leaf Nodes. As shown in the middle of Figure 1, each leaf node contains a linear model, which maps keys to a set of groups. High local hardness can impact

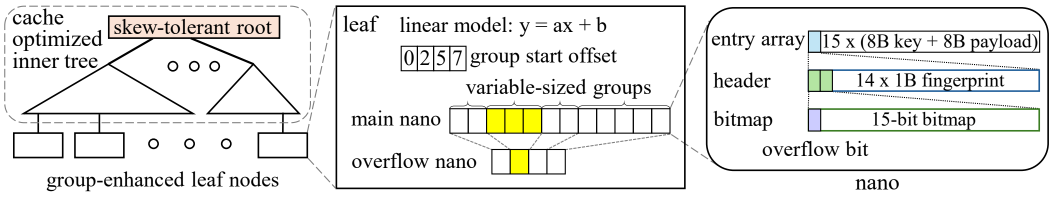

Fig. 1. LINE overview.

the groups in two ways: 1) the number of keys per group can be very skewed, and 2) the in-group key distribution can also be very skewed, and may not be well approximated with the linear model.

Our group design addresses the two problems as follows. First, to tolerate skew key distribution across groups, the group size is not fixed but *variable*. The size of each group is set to be proportional to the number of keys mapped to the group during bulkloading (cf. Equation 2 in Section 4.1). Second, to deal with local skew within a group, each group is organized as a hash table. The hash bucket is denoted as *nano* as shown in Figure 1 (cf. Section 4.1 for details of the nano structure). The resulting group design consists of a number of nanos in the main array, one nano in the overflow array, and a start offset recording the array index of the first nano in the main array. We employ a variant of two-choice hashing with stash as the hashing strategy to improve group space utilization.

There are several benefits of group-enhanced leaf nodes. First, the leaf exploits the linear model to map a key precisely to a group, thereby supporting large leaf nodes as existing learned indices. Second, the variable group size and the in-group hashing can effectively tolerate local hardness for efficient index search/insert/delete/update operations. Third, unlike the hashing-based whole leaf design in Chameleon [12], we restrict the use of hashing within a group. Since range scans visit a group at a time, we can judiciously set the average group size for good range scan performance. Finally, the group is a nice unit for concurrency control and leaf SMOs.

Global Skew: Cache-Optimized Inner Tree. During index construction, we divide the bulkload data set into leaf nodes, then insert leaf boundary keys into an inner tree. To handle data sets with high global skew, we build a cache-optimized inner tree with skew-tolerant root as depicted in the left of Figure 1.

First, to lower the complexity of the inner tree, we reduce the number of leaf nodes to make the inner tree smaller than the (last-level) CPU cache size. This is achieved with a novel SPLA (segment-based PLA) algorithm that enables efficient cache-aware error bound selection (cf. Section 5.1). The resulting inner tree is cache-optimized and can fit into the CPU cache when the index is intensively used (e.g., during index-based joins or data scans). On the other hand, when the index is used intermittently, we expect the root and its children to stay in the cache, and the rest of the inner tree to be shallow. Consequently, an inner tree search sees only a small number of cache misses.

Second, we detect large skewed gaps in the set of leaf boundary keys when building the inner tree. Based on the skewed gaps, we divide the keys into subsets and construct an inner subtree for each subset. Then, we store the boundary keys of the inner subtrees in the skew-tolerant root, which performs comparison-based search for tolerating the skewed gaps (cf. Section 5.2).

Finally, to simplify the implementation, we consider employing an existing learned index (e.g., ALEX [7], LIPP [44]) as the inner subtree. For a negative search, we modify the index to return the left-most entry to locate the associated leaf node (cf. Section 5.3).

Tail Latency: Group-Wise Migration. At the leaf level, the most costly operation is a leaf SMO. We decompose a leaf SMO into a number of steps. Each step migrates a single group from the old leaf to a new leaf. While a leaf is undergoing an SMO, a concurrent index read or write can visit the old or the new leaf node(s) depending on the migration status of its target group. Compared to locking the entire leaf node for migration in a background thread, group-wise migration can significantly reduce the latency for concurrent operations (e.g., inserts) accessing the migrating leaf (cf. Section 4.3).

For the inner tree, a write occurs only when there is a leaf SMO. Therefore, the SMOs in the inner tree are much less frequent than leaf SMOs. Moreover, since the inner tree is cache-sized, the cost of a single non-leaf expand or split operation is bounded. We disable any subtree rebuild operation in the inner tree.

Index Operations. After understanding the high-level picture of LINE, we overview the index operations in the following:

- *Bulkload*: First, given a data set D, LINE selects the cache-aware error bound ε using our SPLA algorithm, and produces a sequence of key segments. Second, for each key segment, LINE constructs a group-enhanced leaf node. Third, LINE collects the set of (left-most key, leaf pointer) as D_{inner}, detects large skewed gaps, and partitions D_{inner} into subsets based on the gaps. Finally, for each subset of D_{inner}, LINE invokes the inner subtree's bulkload algorithm, then inserts the (left-most key, inner subtree root pointer) into the skew-tolerant root.
- *Search/Insert/Delete/Update*: LINE first searches the given key in the inner tree to locate the relevant leaf, then calls the leaf search/insert/delete/update algorithm. The leaf insert performs uniqueness check[2]. The leaf insert or delete may lead to a leaf SMO. For a new leaf, the corresponding (left-most key, leaf pointer) is inserted into the inner tree. For a deleted leaf, its (left-most key, leaf pointer) entry is deleted from the inner tree.
- *Range Scan*: Given a range, LINE scans the inner tree to obtain a list of relevant leaf pointers. Then, it scans each leaf in the list. Within each leaf, LINE uses the linear model to locate groups that overlap with the range, and visits each relevant group to retrieve scan results. For groups that span the range boundaries, LINE filters out keys outside the range.

4 Group-Enhanced Leaf Nodes

We begin this section by describing the leaf structure and normal operations without SMOs in Section 4.1 and 4.2, respectively. Then, we focus our attention on leaf SMOs in Section 4.3.

4.1 Leaf Node with Variable-Sized Groups

Leaf Construction with Variable-Sized Groups. The construction procedure is provided with a key segment, consisting of an array D_{leaf} of sorted key-value entries, a linear model $Model(\cdot)$ that predicts a key's index in the entry array, and an error bound ε (cf. Section 5.1 for the key segment and leaf model computation). Table 3 lists the main terms used in this paper.

Suppose the average group size is m nanos (e.g., m=10), and the bulkload fill factor is f (e.g., 0.7). Let the number of key-value slots per nano be k_{nano} (k_{nano}=15 in our design). Then, we aim to bulkload an average $k_{group} = f \cdot m \cdot k_{nano}$ index entries per group. Hence, there are $n_{group} = \lceil \frac{|D_{leaf}|}{k_{group}} \rceil$ groups. The group ID of a given key is computed as follows:

$$GroupId(key) = \lfloor \frac{Model(key)}{k_{group}} \rfloor \tag{1}$$

We compute the group ID for each key in D_{leaf}, and count the number of keys per group kg_i, where i=0, 1, ..., n_{group}-1.

[2]We follow existing learned index implementations [7, 44] to assume unique keys. However, it is straight-forward to support duplicate keys by storing a pointer in place of the value in the leaf to point to a buffer that contains all values with the same key.

Table 3. Main terms used in this paper.

term	definition
$\lvert D_{leaf}\rvert$	number of keys at leaf construction
ε	error bound of the leaf model $Model(\cdot)$
m	(parameter) average group size in nanos, e.g., 10
f	(parameter) bulkload fill factor, e.g., 0.7
k_{nano}	number of slots per nano, i.e., 15
k_{group}	avg # keys per group at leaf construction, $k_{group} = fmk_{nano}$
$\hat{k}$	avg # keys per group in the current leaf, $\hat{k} \le mk_{nano}$
n_{group}	number of groups in the leaf, $n_{group} = \lceil \frac{\lvert D_{leaf}\rvert}{fmk_{nano}} \rceil$
kg_i	number of keys in group$_i$
mg_i	number of nanos in group$_i$, $mg_i = \lceil \frac{kg_i}{fk_{nano}} \rceil$
N_{leaf}	number of leaf nodes (number of leaf key segments)

When the local hardness is high, the distribution of kg_i can be very skewed. A naïve design with fixed-sized group would incur significant space or time cost. First, one way is to allocate a sufficient number of nanos per group to accommodate the largest group. However, this would waste space for small groups, which contain only a small number of keys. Second, another idea is to allocate additional space to the large groups, such as unsorted overflow nanos, sorted overflow buffer as in Hyper [48], or child nodes as in LIPP [44] and DILI [22]. However, these additional structures would incur significant index search and/or insert cost for large groups. Third, one can decrease the number of groups by increasing the m parameter. Large groups may be merged with nearby small groups, and the resulting distribution of keys across groups tends to be less skewed. (The hash-based whole leaf in Chameleon [12] is the extreme case where $m = \infty$.) Unfortunately, large average group size causes poor scan performance for small ranges.

To address this problem, we make the group size *variable*. Specifically, for group$_i$, we allocate mg_i-1 main nanos and one overflow nano, where mg_i is computed as follows:

$$mg_i = \lceil \frac{kg_i}{f \cdot k_{nano}} \rceil \tag{2}$$

Group$_i$'s start offset is $\sum_{j=0}^{i-1}(mg_j - 1)$. There are $\sum_{i=0}^{n_{group}-1}(mg_i - 1)$ and n_{group} nanos in the main and overflow arrays, respectively.

Then, we insert the keys into the corresponding group. Our nano design is inspired by hash buckets [25] and unsorted tree nodes [24, 27] in existing indices. As illustrated in Figure 1, a nano is an aligned 256-byte structure. It consists of a 16-byte header and 15 key-value entry slots. The header is composed of 14 fingerprints for the first 14 entries, 15 per-entry empty/valid bits, and an overflow bit indicating whether keys mapped to this nano have overflowed. Entries in a nano are unsorted. For a key search, SIMD computation can efficiently filter out most slots using the valid bits and the fingerprints. (If the last slot is used, then it is always checked.)

We limit the leaf node size to be up to 16MB. There can be at most 16MB/256= 64K nanos. Hence, the group start offset can be represented as 2-byte unsigned integers (i.e., `uint16_t`).

Upper Bound of Group Size. Since the prediction error of the leaf model is bounded, we can derive the following bound:

Theorem 4.1 (Upper Bound of Group Size). *Let m be the target average group size and ε be the error bound of the leaf model. The number of nanos per group is upper bounded by $m + \lceil \frac{2\varepsilon}{f \cdot k_{nano}} \rceil$ by the leaf construction.*

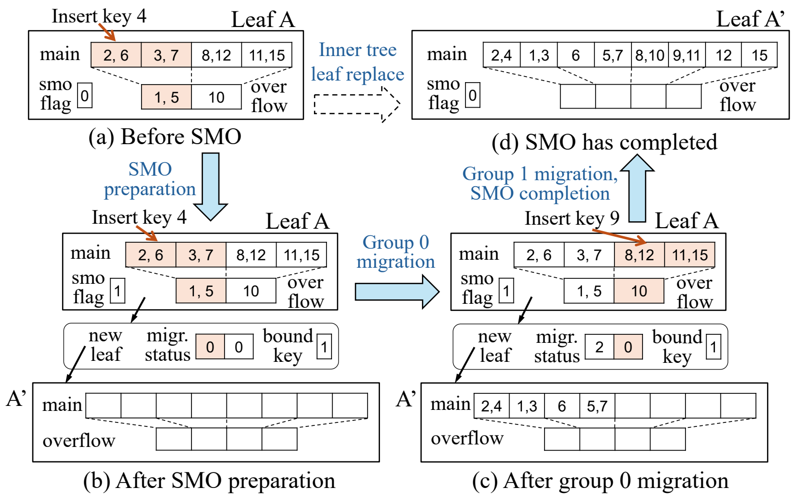

Fig. 2. An illustration of a leaf SMO with group-wise migration.

Proof. We first show the number of keys per group by the leaf construction is upper bounded by $k_{group} + 2\varepsilon$.

Without loss of generality, we consider the keys in a group$_i$. From Eqn 1, if $GroupId(key) = i$, the model output $Model(key)$ must be in $[k_{group} \cdot i, k_{group} \cdot (i + 1))$. Since the error bound of $Model(\cdot)$ is ε, the array index of the keys must be in $[k_{group} \cdot i - \varepsilon, k_{group} \cdot (i + 1) + \varepsilon)$. Thus, there are at most $(k_{group} + 2\varepsilon)$ keys in group$_i$.

Then, from Eqn 2, since $k_{group} = f \cdot m \cdot k_{nano}$, the maximum number of nanos in a group is $m + \lceil \frac{2\varepsilon}{f \cdot k_{nano}} \rceil$. □

The upper bound of group size impacts the range scan performance and the worst-case tail latency of leaf SMOs.

4.2 Leaf Index Operations without SMOs

Leaf Search. The search within a leaf begins with computing $i = GroupID(key)$ according to Eqn 1. Then, LINE hashes the key to two main nanos. If the key is found in the hashed nanos, the search returns the matching entry. Otherwise, LINE checks if the overflow flag is set in the searched nanos. If so, it proceeds to search the overflow nano in the group. If no matching key is found, the search returns not found.

Leaf Insert. To insert a key into a leaf, LINE follows the same steps as leaf search to locate the hashed nanos. It first performs uniqueness check for the key. Next, it attempts to insert the new entry in the first target nano, then the second target nano. If neither nano has empty slots, LINE inserts the new entry into the overflow nano. If the overflow nano is also full, a leaf SMO is triggered.

Leaf Delete and Update. The delete operation within a leaf proceeds by locating the key location in a nano and the specific slot within the nano through a leaf search. Then, the corresponding valid bit in the bitmap is set to 0 to signify the deletion.

Similarly, the update operation (that modifies the value for a given key) begins by identifying the key's nano and its slot via search. The update is then executed by modifying the fingerprint and value stored in the corresponding slot.

The delete/update returns failure if the key does not exist.

Leaf Range Scan. Given a key range $[key_b, key_e]$, LINE visits all groups in $[max(GroupId(key_b), 0), min(GroupId(key_e), n_{group} - 1)]$. For each such group, it retrieves all valid entries in the main nanos and the overflow nano of the group. If the group is either $GroupId(key_b)$ or $GroupId(key_e)$, the retrieved keys are compared against the key range to filter out unqualified results.

Concurrent Normal Leaf Operations. We use per-group locks to allow separate groups to be written concurrently. For efficient reads, each lock word contains a version to support optimistic locking. Group locks serve as the basis for group-wise migration.

To access a group, a write request obtains the group lock before modifying the group. Then, it increments the version and releases the group lock with a single CAS operation. A read request verifies that the target group is not locked and the group lock version stays the same during the visit. If a conflict is detected due to locking or version checking, the request is retried.

4.3 Leaf SMOs with Group-Wise Migration

A failed insertion in an overflow nano triggers a leaf SMO. The leaf SMO is carried out in three phases: 1) preparation, 2) group-wise migration, and 3) completion.

SMO Preparation. Every leaf contains an SMO flag. A LINE thread starts the SMO preparation by setting the flag from 0 to 1 with a CAS operation. This prevents other concurrent threads from starting the SMO preparation on the same leaf node.

Next, LINE decides the SMO strategy (i.e., leaf expansion vs. leaf split) based on the key distribution in the leaf. To reduce the latency for computing the distribution, LINE samples a nano per group. Because of in-group hashing, keys in any nano are representative of the group's key distribution. A linear model is computed for each group based on the keys of the sampled nano and the group size, then the SPLA algorithm (c.f. Section 5.1) is applied with the target error bound ε to derive the key segments. Note that LINE does not acquire per-group locks because the model approximation can tolerate slight inconsistency in the sampled keys.

If the result contains only one segment, the key distribution stays roughly the same, prompting node expansion. If multiple segments are identified, the leaf is split. We set the aggregate size of the new leaves to be roughly double the size of the old leaf. Then, we compute the new groups in the new leaf node(s). Each old group in the old leaf is mapped to one or multiple new groups. We ensure that the new group size still observes the upper bound in Theorem 4.1.

Finally, LINE allocates space for the new leaf node(s), and initializes three migration structures pointed from the old leaf: 1) per-group migration status, 2) pointer(s) to the new leaf node(s), and 3) new leaf boundary key(s). Both the SMO flag and the per-group migration status can take one of three values: 0) the leaf (group) has not yet migrated; 1) the leaf (group) is actively migrating; and 2) the leaf (group) has completed the migration.

Concurrent operations to the leaf that started before the SMO are not affected by the SMO preparation. Concurrent operations to the leaf that start during the SMO preparation will see the non-zero SMO flag and correctly detect the ongoing SMO.

Group-Wise Migration. The unit of migration is a group to reduce the tail latency of leaf SMOs. Groups in the same leaf can be migrated independently by different concurrent threads since the memory locations of different groups are disjoint.

To perform a leaf index operation, a thread detects an ongoing SMO by checking the SMO flag. If the flag is 0, then it performs the normal leaf operation as described in Section 4.2. If the SMO flag is non-zero, read and write threads take different actions.

A leaf read thread checks the per-group migration status before and after accessing a group. If the status remains 0, the read in the old leaf succeeds. If the status changes or begins with 1, it retries from the root. If the status is 2 at the beginning, the thread performs a normal read in the new leaf.

A leaf write thread performs eager group migration in order to accelerate the SMO. It chooses the target group of the current request to migrate if the target group is not migrated. Otherwise, an arbitrary group with status 0 is chosen. Then, the write thread uses CAS operations to modify the migration status of the chosen group so that other threads cannot concurrently migrate the same group. It also obtains the per-group lock of the chosen group in the old leaf, thereby waiting for any concurrent writes to the group that may have started before the SMO but not yet completed. Note that this group lock will not be released in the SMO to ensure that this old group will not be written any more. After completing a group migration, the thread checks the target group's status. For status 1, the target group is being migrated (by other threads). Hence, it retries the write request from the inner tree. For status 2, the thread performs the writes in the new leaf.

SMO Completion. After migrating a group, a thread sets the per-group status to 2 with a CAS operation. Then, it checks if all groups are in status 2. In such case, it starts the SMO completion processing by issuing a CAS to update the SMO flag from 1 to 2. This prevents other threads from completing the SMO on the same leaf.

For a leaf expansion, the thread issues an update request in the inner tree to replace the old leaf pointer by the new leaf pointer. For a leaf split, it inserts (leaf boundary key, pointer) of the new leaf node(s) from right to left to the inner tree except for the left-most new leaf node. Then, it replaces the old leaf pointer with the left-most new leaf's pointer in the inner tree.

In both cases, a concurrent index operation either visits the old leaf or a new leaf. The former is correct because it will detect the SMO and redirect the visit to the correct new leaf. The correctness of the latter is easily seen for the leaf expansion. However, we need further explanation for the leaf split. Suppose the old leaf covers a key range $[k_{left}, k_{right})$. Let k_{latest} be the boundary key of the latest inserted new leaf in the inner tree. If the key of the concurrent index operation is in $[k_{left}, k_{latest})$, then it goes to the old node and is correct. A new leaf is visited if the key is in $[k_{latest}, k_{right})$. Since the new leaf node(s) are inserted in the descending order of boundary keys, all new leaf node(s) in $[k_{latest}, k_{right})$ must exist in the inner tree. Hence, the operation visits the correct new node.

Deletion-Triggered SMOs. The above description focuses on insertion-triggered SMOs. Since the data size tends to grow in general, we perform simplified SMOs for deletion. That is, we only delete empty leaf nodes. When an index delete sees an empty target nano, it checks if all nanos in the group are empty. If so, it obtains all group locks to further check if all other groups in the leaf are empty. In such cases, it issues a CAS to update the SMO flag from 0 to 2, then deletes the leaf boundary key from the inner tree.

Example. Figure 2 depicts an example of a leaf SMO. In Figure 2(a), a write request attempts to insert key 4 to leaf A but fails. Both the hashed main nanos and the overflow nano of the highlighted target group are full. This triggers a leaf SMO.

In the preparation phase, LINE decides to expand the leaf. It allocates a new leaf A', records the new leaf pointer and its boundary key, and initializes the per-group migration statuses to all 0s. This transitions the leaf structures to Figure 2(b).

After the SMO preparation, the insertion of key 4 is retried. LINE detects that the target group has not yet migrated (status 0). Consequently, it migrates group 0, copying all keys in group 0 to the new leaf, and updating the migration status to 2 to indicate that the migration of group 0 completes, as depicted in Figure 2(c).

Subsequently, another write request to insert key 9 performs a similar migration of group 1.

Algorithm 1: Segment-based PLA algorithm.

```
Function SPLA(S, ε)
    $S_{out} = \{\}$; $x_0 = S[0].left\_key$; $a_{min} = 0$; $a_{max} = \infty$;
    for $i = 0$ to $|S|$ do
        $x_l = S[i].left\_key$; $x_r = S[i].right\_key$; $\Delta\varepsilon = \varepsilon - S[i].error\_bound$;
        rewrite $S[i].model$ as $y = a_i(x - x_0) + b_i$;
        $t_{min} = max(a_{min}, a_i + \frac{b_i - \Delta\varepsilon}{x_l - x_0}, a_i + \frac{b_i - \Delta\varepsilon}{x_r - x_0})$;
        $t_{max} = min(a_{max}, a_i + \frac{b_i + \Delta\varepsilon}{x_l - x_0}, a_i + \frac{b_i + \Delta\varepsilon}{x_r - x_0})$;
        if $(t_{min} \leq t_{max})$ then
            $a_{min} = t_{min}$; $a_{max} = t_{max}$;
        else
            $a = (a_{min} + a_{max})/2$;
            append $(x_0, x_r, y = a(x - x_0), \varepsilon)$ to $S_{out}$;
            $x_0 = S[i].left\_key$; $a_{min} = 0$; $a_{max} = \infty$; $i = i - 1$;
        end
    end
    return $S_{out}$;
```

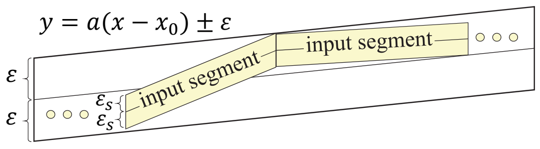

Fig. 3. Illustration of segment-based PLA algorithm.

Once this migration concludes, as shown in Figure 2(d), the SMO is finalized. LINE updates the inner tree to replace A with A'.

5 Cache-Optimized Inner Tree

In this section, we first propose two techniques, i.e., cache-aware error bound selection and skew-tolerant root, for handling global skew in Section 5.1 and 5.2, respectively. Then, we describe the operations in the inner tree in Section 5.3.

5.1 Cache-Aware Error Bound Selection

Given a data set D, LINE performs PLA (piece-wise linear approximation) to divide D into N_{leaf} key segments with per-segment linear models. An important parameter of PLA is the error bound ε. The larger the ε, the less accurate the per-segment linear model. On the other hand, N_{leaf} decreases as ε increases. We would like to choose a good ε to reduce the inner tree size.

A naïve solution is to loop over the candidate ε's, and for each ε run the PLA algorithm on D. However, each PLA run processes every key in D, taking $O(|D|)$ time. Hence, this multi-pass solution can incur significant computation overhead.

Segment-Based PLA Algorithm. To avoid this cost, our basic idea is to perform PLA with the smallest error bound candidate ε_s (e.g., 16) to generate an initial set S of key segments with linear models. Then, we design a Segment-Based PLA (SPLA) algorithm. It takes the initial segments as input, and computes key segments for the candidates with larger error bounds (i.e., $\varepsilon > \varepsilon_s$). In this way, we process D once and S multiple times. Since $|S|$ is often several orders of magnitude smaller than $|D|$, SPLA can effectively reduce the cost of error bound selection.

As shown in Figure 3, SPLA computes a linear model with error bound ε that approximates as many consecutive input segments as possible. The outer parallelogram depicts the region described by the model $y = a(x - x_0)$ with ε, where x_0 is the left-most key of the first input key segment. Each inner parallelogram covers all points in an input segment. We derive the constraints on the slope a for the outer parallelogram to cover an inner parallelogram.

For an input segment$_i$, we rewrite its model as $y = a_i(x - x_0) + b_i$. Denote the segment's left-most key as x_l, right-most key as x_r, and error bound as ε_i. ($\varepsilon_i = \varepsilon_s$ during bulkloading.) If segment$_i$ can be covered by the outer parallelogram, then we have the following:

$$\begin{cases} a_i(x - x_0) + b_i + \varepsilon_i \le a(x - x_0) + \varepsilon \\ a_i(x - x_0) + b_i - \varepsilon_i \ge a(x - x_0) - \varepsilon \end{cases}$$
$$\Rightarrow \begin{cases} a \ge a_i + \frac{b_i - \Delta\varepsilon}{x - x_0} \\ a \le a_i + \frac{b_i + \Delta\varepsilon}{x - x_0} \end{cases} \text{for } x \in [x_l, x_r] \text{ and } \Delta\varepsilon = \varepsilon - \varepsilon_i \tag{3}$$

The SPLA algorithm is listed in Algorithm 1. It processes each input segment in S (Line 4), and employ Eqn 3 to compute the lower and upper bounds of a to accommodate as many segments as possible (Line 5–11). Note that the right-hand side of Eqn 3 is a monotonic function of x, and thus only x_l and x_r need to be examined. If an input segment cannot be covered by the current slope bounds, then SPLA generates an output key segment, and starts a new output segment (Line 13–16).

Error Bound Selection. We would like to select a good error bound so that the inner tree as well as the leaf metadata can fit into the CPU cache. Suppose the average size per key of the inner tree is $size_{entry}$ and the size of a leaf's metadata is $size_{leafmeta}$. (Both parameters can be measured offline.) Moreover, we also consider the growth of the inner tree due to new insertions. Suppose the data set is expected to grow at most w (e.g., 10) times. Then, we run SPLA for multiple ε's (e.g., 32, 64, 128, 256, 512, 1024, ...) until $w \cdot |S_{out}|(size_{entry} + size_{leafmeta}) <$ CPU cache size. S_{out} is used to construct the leaf nodes. $N_{leaf} = |S_{out}|$.

5.2 Skew-Tolerant Root Node

We divide the entire key range into n_{intv} (e.g., 100) equal-sized intervals, then mark the empty intervals without any keys. A gap consists of a sequence of empty intervals between two non-empty intervals. The number of empty intervals is the gap's size.

We aim to detect highly skewed gaps. First, if a gap size $\ge s_{lg}$, the gap is considered a large skewed gap. Second, we sort the rest of the gaps in the ascending order of their sizes, and obtain the medium gap size s_{md}. Third, any gap whose size $\ge u_{skew}s_{md}$ is also considered highly skewed. We have the following guarantee.

Theorem 5.1 (Number of Detected Skewed Gaps). *The skewed gap detection procedure returns at most* $\frac{n_{intv}-1}{\min(s_{lg}+1, u_{skew}+3)}$ *gaps.*

Proof. Suppose the first step gets a gaps. In the third step, there are b gaps whose size $\ge u_{skew}s_{md}$ and c gaps whose size $\le s_{md}$.

The total number of empty intervals $\ge as_{lg} + bu_{skew}s_{md} + c$. Since the first and the last intervals are non-empty, there are more key segments than gaps. Thus, the number of key segments is at least $a + b + c + 1$. We have the following:

$$(as_{lg} + bu_{skew}s_{md} + c) + (a + b + c + 1) \le n_{intv} \tag{4}$$

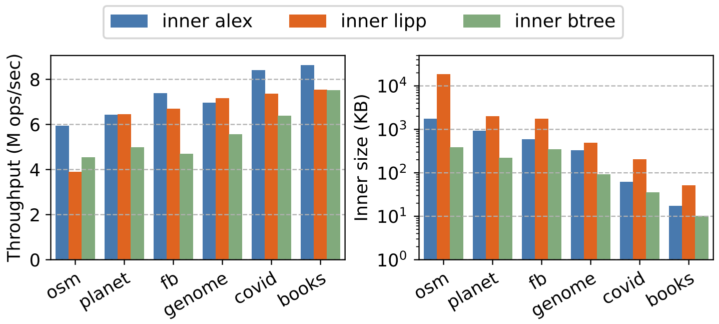

Fig. 4. Read-only throughput and size of inner subtrees.

As s_{md} is the medium gap size, $b \leq c$ and $s_{md} \geq 1$.

$$a(s_{lg} + 1) + b(u_{skew} + 3) \leq n_{intv} - 1 \tag{5}$$

Hence, the number of returned gaps= $a + b \leq \frac{n_{intv}-1}{\min(s_{lg}+1, u_{skew}+3)}$. □

We divide the leaf boundary keys into subsets at the skewed gaps, and constructs an inner subtree for each subset. Then, we put (left-most key, subtree root pointer) into the root node. It tolerates skewed gaps by performing key comparison based search rather than model-based search.

For efficient root search, we restrict the skew-tolerant root to have up to 8 child subtrees. Given $\lfloor \frac{n_{intv}-1}{\min(s_{lg}+1, u_{skew}+3)} \rfloor \leq 7$, s_{lg} and u_{skew} can be derived from n_{intv}. We set n_{intv}=100, s_{lg}=15, u_{skew}=10 by default. (cf. Section 7.3 for experiments varying n_{intv})

5.3 Index Operations in Inner Tree

For simplicity, we employ an existing index structure as the inner subtree. We examine several choices in Figure 4, which reports the read-only throughput and the size of the different inner choices. Compared to traditional indices (e.g., B+-Trees), we prefer learned indices because of their superior performance. Among learned indices, we consider ALEX and LIPP because of their simple data structures. However, the large root of LIPP consumes a lot of space. Since our goal is to reduce the inner tree size, our implementation of the inner subtree is based on ALEX.

To reduce the tail latency of the inner ALEX, we modify ALEX to preserve space in the inner tree to accommodate the expected growth of the index as described in Section 5.1. We also adjust the maximum node size in the inner ALEX (from 16MB) to 1MB to reduce the data copying cost during SMOs. As a result, we expect SMOs of the inner tree (if occurs) to be handled in the L3 cache, thus guaranteeing moderate tail latency.

Left Key Search. We modify ALEX to return the left key upon a negative search. However, when a search reaches an ALEX leaf, the search key may not reside in the leaf's key range. Hence, it is necessary to compare the leaf's minimum key with the search key and potentially search backwards to locate the first leaf whose minimum key is less than or equal to the search key.

Search/Insert/Delete/Update. For a point operation, LINE performs the left-key search in the inner tree to locate the leaf node, then invokes the relevant operation in the leaf. LINE writes to the inner tree only for leaf SMOs (c.f. Section 4.3).

Range Scan. LINE performs the range scan in the inner tree to retrieve a set of leaf nodes that overlap with the range. Then, it scans the relevant leaf nodes to retrieve the scan results.

6 Cost Analysis

Cost of Point Operation without SMOs. We consider the number of cache misses. For an inner tree search, if the index is intensively used, the inner tree fits into the CPU cache, then the cost is minimized. If the index is used intermittently, then we expect the skew-tolerant root and the root node of the inner subtrees to stay in the CPU cache. Since the inner tree is small, it is often quite shallow and incurs only a few cache misses.

For the leaf point operation, we have the following guarantee.

THEOREM 6.1 (COST OF LEAF POINT OPERATION WITHOUT SMOS). *A leaf point operation without SMOs incurs $O(1)$ cache misses.*

PROOF. For a point operation, LINE accesses the leaf metadata and reads the group start offset. This incurs at most 2 cache misses. Then, LINE visits at most 2 main nanos and 1 overflow nano in the target group because it employs two-choice hashing with stash. Hence, the total number of cache misses is $O(1)$. □

Cost of Range Scan Operation. We mainly consider scans in the leaf nodes. Suppose a scan retrieves k_{scan} keys and a group contains $\hat{k}$ keys on average. We have the following results.

THEOREM 6.2 (COST OF LEAF RANGE SCAN). *A leaf range scan retrieves $k_{scan} + \hat{k}$ keys on average. It retrieves at most $k_{scan} - 2 + 2(m + \lceil \frac{2\varepsilon}{f \cdot k_{nano}} \rceil)k_{nano}$ keys.*

PROOF. Let $p = \lfloor \frac{k_{scan}}{\hat{k}} \rfloor$ and $q = k_{scan} \mod \hat{k}$. That is, $k_{scan} = p \cdot \hat{k} + q$. We consider the group that contains the first key in the range. Suppose k_1 keys in this group fall in the range. When $k_1 = 1, ..., \hat{k} - q$, the scan visits $p + 1$ groups. When $k_1 = \hat{k} - q + 1, ..., \hat{k}$, the scan visits $p + 2$ groups. Hence, on average, the scan visits $p + 1 + \frac{q}{\hat{k}}$ groups, retrieving $k_{scan} + \hat{k}$ keys.

Moreover, in the worst case, the first and the last keys are in two large groups. There are $k_{scan} - 2$ middle keys. From Theorem 4.1, the max number of nanos in a group is $m + \lceil \frac{2\varepsilon}{f \cdot k_{nano}} \rceil$. Hence, the max number of retrieved keys is $k_{scan} - 2 + 2(m + \lceil \frac{2\varepsilon}{f \cdot k_{nano}} \rceil)k_{nano}$ □

Cost of Leaf SMO and Tail Latency. First, SMO preparation samples a nano per group and computes the key segments. The average number of keys per nano is $\frac{\hat{k}}{m}$. Hence, it processes a total $\frac{\hat{k}n_{group}}{m}$ keys, taking $O(\frac{\hat{k}n_{group}}{m})$ time. Since $\hat{k}n_{group}$ is the total number of keys in the leaf, SMO preparation processes about $1/m$ of all keys in the leaf. Second, a group migration retrieves all the keys in a group and inserts them into the new group(s), which costs $O(mk_{nano})$. Third, SMO completion updates the inner tree with the new leaf node(s), taking $O(1)$ time. Overall, the cost of leaf SMO is $O(\frac{\hat{k}n_{group}}{m} + mk_{nano})$. The tail latency is the maximum cost for preparing the SMO or migrating a group.

Cost of Index Construction. First, leaf construction steps process either every key or every group, taking $O(|D_{leaf}|)$ time. Second, cache-aware error bound selection performs a one-pass PLA on D to generate the initial set S of key segments, then runs SPLA multiple times on S. Hence, the cost is $O(|D| + (n_\varepsilon - 1)|S|)$, where n_ε is the number of tested error bounds. Since $|S| \ll |D|$, our solution is much faster than the naïve multi-pass PLA, which costs $O(n_\varepsilon|D|)$. Third, skewed gap detection marks empty intervals by searching for the interval boundaries, and checking if there are keys between consecutive boundaries. Thus, its cost is $O(n_{intv}log(N_{leaf}))$. Finally, constructing the skew-tolerant root takes $O(1)$ cost.

Space Cost. For a leaf node, the leaf metadata takes $O(1)$ space. After leaf construction, a group$_i$ contains mg_i 256-byte nanos and a 2-byte start offset. Thus, the total space for all groups is

$256 \sum_i mg_i + 2n_{group} = 256 \sum_i \lceil \frac{kg_i}{fk_{nano}} \rceil + 2\lceil \frac{|D_{leaf}|}{fmk_{nano}} \rceil$. Since both terms at the right hand side is $O(|D_{leaf}|)$, the leaf space is $O(|D_{leaf}|)$.

The space of the leaf level is $O(|D|)$, where D is the bulkload data set. The size of the inner tree is less than the CPU cache size. Hence, the total space of LINE is $O(|D|)$.

Leaf Parameter Selection. Given the above analysis, we see that the setting of m impacts both the worst-case range scan performance and the tail latency. For range scans, mk_{nano} should be close to the frequently used smallest range size. For a given tail latency target, we measure the constant factors of the SMO costs and select m so that both the SMO preparation and the group migration can satisfy the target tail latency. In our experiments, we set m=11, and the bulkload fill factor f=0.7 by default.

7 Performance Evaluation

7.1 Experimental Setup

Machine Configuration. The experimental machine is a Ubuntu 20.04 Linux server equipped with two Intel Xeon Gold 5218 CPUs (2.30GHz, 16 cores/32 threads, 22MB L3 cache) and 348GB DRAM. All experiments are run on a single CPU to avoid NUMA effects.

Solutions to Compare. By default, we perform single-threaded experiments and compare LINE with six baselines: 1) *B+-Tree* [1], 2) *ART* [18], 3) *PGM* [9], 4) *ALEX* [7], 5) *LIPP* [44], and 6) *DILI* [22].

In the scalability experiments, we run up to 32 threads and compare LINE with five baselines: 1) *BtreeOLC*, B+-Tree with optimistic lock coupling [42]; 2) *BlinkTree*, a Blink-pthread implementation with optimistic synchronization [31]; 3) *ART-ROWEX*, ART with read-optimized write exclusion synchronization [19]; 4) *ALEX+*, a multi-threaded implementation of ALEX [43]; and 5) *SALI*, a concurrency-optimized index based on LIPP that employs probability models and a novel node evolution strategy [11]. (There are no multi-threaded implementations for PGM and DILI.)

LINE and all the baselines are implemented in C/C++. We employ the GRE platform [43] to run experiments, which include implementations of all baselines except DILI and SALI. We obtain DILI and SALI from the code repositories provided by the original publications. For DILI, we modify the type of its model parameters from `double` to `long double` to prevent crashes during bulkloading. DILI supports only signed long integer keys. Thus, we adapt all keys from `uint64_t` to `int64_t`.

For all baselines, we retain their default parameters. For LINE, the default target group size m=11 and the bulkload fill factor f=0.7.

Real-World Data Sets. We use four complex real-world data sets with either high local or global hardness from GRE [43]: 1) *osm*: uniformly sampled OpenStreetMap locations [13]; 2) *planet*: planet ID in OpenStreetMap [4]; 3) *fb*: upsampled Facebook user IDs [13]; and 4) *genome*: loci pairs in human chromosomes [34]. Moreover, we incorporate two simpler data sets: 5) *covid*: uniformly sampled Tweet IDs tagged with COVID-19 [26]; and 6) *books*: Amazon book sales popularity rankings [13]. The characteristics of the data sets are summarized in Table 1. Each data set consists of 200 million 64-bit keys paired with a 64-bit value.

Synthetic Data Sets. The GRE data generation tool can set the local and global hardness of the synthetic data set to produce. We modify the tool to support skewed gaps. The gap sizes are generated to follow the Zipf distribution. We can set the Zipf factor and the percentage of the key range that is empty and occupied by gaps.

We generate two sets of synthetic data. First, to understand the impact of local and global data hardness, we generate data sets by varying the local hardness from 64K, 128K, 256K, 512K, to 1M, and global hardness from 128, 256, 512, 1024, 2048, to 4096. We choose the hardness values based on the max and min hardness values of the real-world data sets in Table 1.

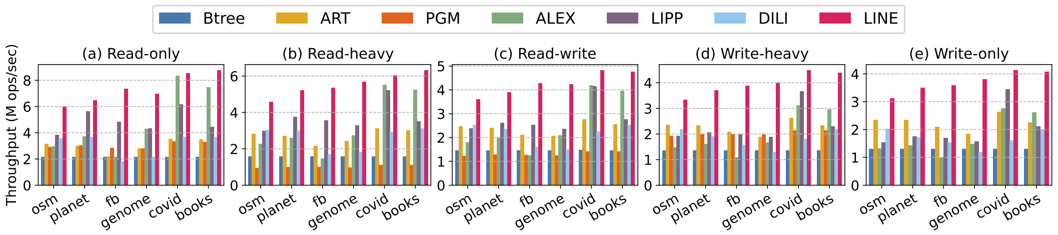

Fig. 5. Index performance varying read-write ratios on real-world data sets.

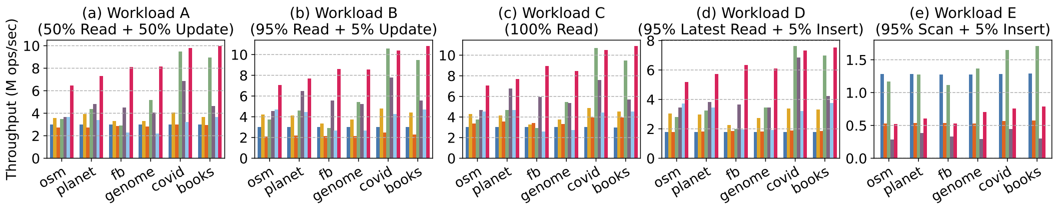

Fig. 6. Index performance running YCSB workloads on real-world data sets.

Second, to understand the influence of skewed gaps, we generate data sets with 70% and 90% gaps in the key range. The Zipf factor of the gap size distribution is set to 5.

Index Workloads. By default, we bulkload an index with 100 million random keys from a data set, then perform 100 million random index operations. We consider five workloads: 1) *read-only* (100% reads), 2) *read-heavy* (80% reads and 20% inserts), 3) *balanced read-write* (50% reads and 50% inserts), 4) *write-heavy* (20% reads and 80% inserts), and 5) *write-only* (100% inserts). The read keys are randomly chosen from the bulkload keys, and the insert keys are randomly chosen from the remaining 100 million keys in the data set. In addition, we also run the YCSB workloads [5]. We report operation throughput for the experiments.

Moreover, to measure range scan performance, we bulkload each index with all 200 million keys from a data set. For a given range size, we randomly generate $\frac{2\times10^9}{range\ size}$ ranges with start and end keys so that each experiment retrieves 2 billion keys. We report the throughput of the scans.

Furthermore, we measure latency under intensive write situations where we bulkload an index with 20 million random keys from a data set, then insert the remaining 180 million keys.

7.2 Overall Performance

Performance Varying Read-Write Ratios on Real-World Data. Figure 5 shows the index throughput of five workloads varying read-write ratios on the six real-world data sets. LINE achieves the best performance for all workloads on all six data sets. The group-enhanced leaf nodes and the cache-optimized inner tree can effectively address the challenges of local hardness and global skew.

Specifically, for the read-only workloads, LINE achieves 2.8–4.0x, 1.9–3.4x, 2.1–2.6x, 1.02–3.4x, 1.2–2.0x, 1.7–4.0x higher throughput than B+-Tree, ART, PGM, ALEX, LIPP, and DILI, respectively. For the mixed workloads in Figure 5(b)–(d), LINE attains performance improvements of 2.4–4.0x, 1.4–2.5x, 1.7–5.8x, 1.1–3.6x, 1.2–2.1x, and 1.4–3.2x over B+-Tree, ART, PGM, ALEX, LIPP, and DILI, respectively. For the write-only workloads, LINE out-performs B+-Tree, ART, ALEX, LIPP, and DILI by factors of 2.4–3.2x, 1.3–2.1x, 1.5–3.6x, 1.2–2.4x, and 1.5–3.2x, respectively.

We omit PGM from the write-only experiments. It achieves high write throughput by first storing the new inserts into a small sorted buffer and merging it into the tree when the buffer is full.

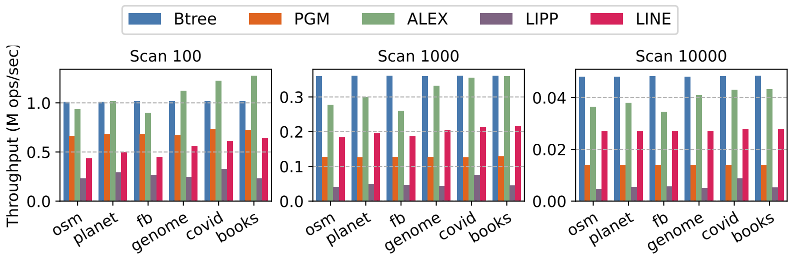

Fig. 7. Range scan performance on real-world data sets.

However, reads suffer from this solution as seen in Figure 5(b), and tree merging can incur drastic tail latency exceeding 1 second.

Performance of YCSB Workloads on Real-World Data. Figure 6 reports the performance of the YCSB workloads. For workload A to D, the results show similar trends as the performance varying read-write ratios in Figure 5. LINE achieves superior performance. Interestingly, the skewed Zipf distribution of YCSB mitigates the impact of the inner index size, which allows ALEX to surpass LINE on covid in workload B–D. For workload E, the results are similar to that of short range scans, as will be discussed below.

Range Scans on Real-World Data Sets. Figure 7 reports the throughput for scanning ranges that contain 100, 1000, and 10000 keys. We compare B+-Tree, PGM, ALEX, LIPP, and LINE. We omit ART and DILI in this set of experiments because the ART implementation does not support range scans, and DILI fails to return enough keys .

From Figure 7, we have the following observations. First, we see that among learned indices, ALEX achieves the best scan performance because of the large gapped sorted array in leaves. Scans can traverse the sorted array to efficiently retrieve keys. Second, LINE achieves the second-best performance for 1000-key and 10000-key scans among all learned indices. Compared to ALEX, LINE pays higher cost to visit the unsorted groups for retrieving keys. Finally, as the range size increases, the performance difference between LINE and ALEX decreases. As proved in Theorem 6.2, a scan retrieves $\frac{\hat{k}}{k_{scan}}$ fraction of extra keys due to the unsortedness of groups. A larger k_{scan} mitigate extra cost.

Maximum Tail Latency on Real-World Data Sets. After bulkloading an index with 10% of the keys from a data set, we randomly insert the remaining 90% of the keys, and measure the latency of each insertion. We report the maximum tail latency as an important metric as p99 or p99.9 tail latency may fail to capture the infrequent extreme events, e.g., subtree rebuilding or model retraining.

In Figure 8, each point represents an experiment of an index on one of the six real-world data sets. The x-axis reports the maximum tail latency, while the y-axis reports the average latency. Hence, a point closer to the origin indicates both superior insert performance and lower maximum tail latency. Figure 8(a) shows the results of the single-threaded experiments, while Figure 8(b) reports the performance of multi-threaded experiments with 32 threads.

From Figure 8, we see that LINE achieves the lowest average latency, which is consistent with the throughput in Figure 5. Moreover, LINE significantly reduces the maximum tail latency, achieving improvements in the single-threaded experiments of up to 5713x, 65x, 18x, and 4x compared to PGM, ALEX, LIPP, and DILI, respectively. PGM suffers from the worst tail latency, exceeding 1 second. In multi-threaded experiments, LINE decreases the maximum tail latency by factors of up to 35x, 44x and 516x compared to ALEX+, SALI, and BlinkTree, respectively. Furthermore, the size

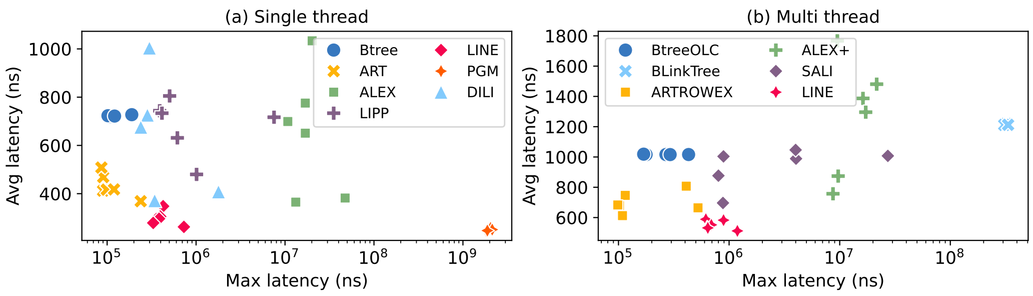

Fig. 8. Maximum latency v.s. average latency.

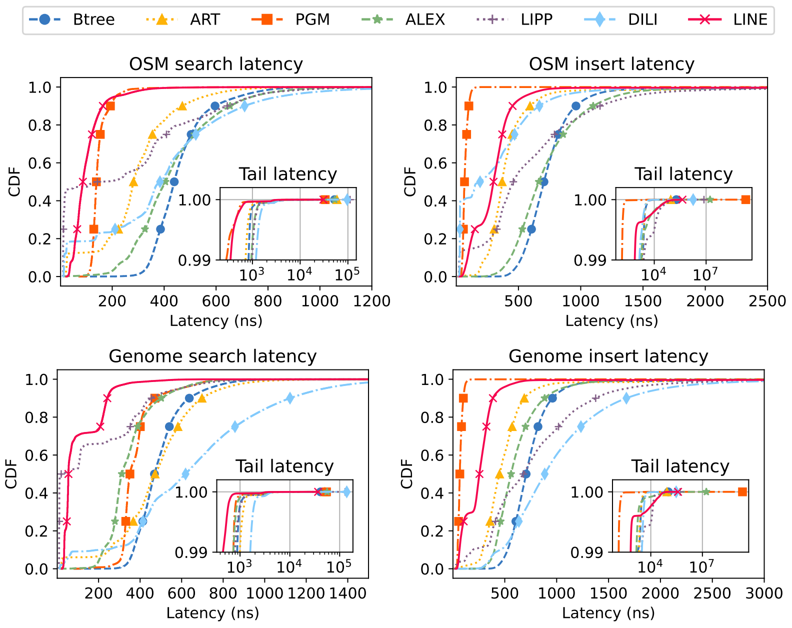

Fig. 9. Distribution of search and insert latency.

of the largest group of LINE is about 28KB, which is much larger than the node sizes in B+-Tree and ART. This leads to the higher tail latency of LINE compared to B+-Tree and ART. Nevertheless, LINE keeps the maximum tail latency to be around and most times under 1 ms. Since the maximum tail latency of B+-Tree is over 100us in all cases and up to 430us. We consider the maximum tail latency of LINE quite acceptable.

Figure 9 depicts the CDF of search and insert latency on osm, the data set with the largest global hardness, and genome, the data set with the largest local hardness. The markers indicate the p25, p50, p75, p90, and max latency. The zoom-in figures highlight the curves between 0.99 and 1. We see that LINE exhibits robust good performance in most cases and attains moderate tail latency.

Drill-Down Analysis of Insertions. Table 4 studies the different categories of insertions for the workload in Figure 8. First, normal insertions account for over 99.6% cases, which achieve low latency. Second, group-wise migration effectively divides leaf SMOs into SMO start, per-group migration, and SMO finish phases. The average latency of each phase is moderate. SMO start takes more time because it samples each group and runs SPLA to decide the SMO strategy. Third, only

Table 4. Statistics of categories of insertions in LINE.

Category	Normal	SMO start	Migration	SMO finish			
				Leaf	A	B	C
osm: Cnt	179.3M	32.8K	602.7K	31.7K	30.2K	1.48K	0
Percentage	99.629%	0.018%	0.335%	0.018%	0.017%	0.001%	0.000%
Avg lat (us)	0.152	79.4	16.6	15.5	+0.304	+4.56	-
genome: Cnt	179.3M	9.34K	648.4K	9.17K	8.58K	594	0
Percentage	99.630%	0.005%	0.360%	0.005%	0.005%	0.000%	0.000%
Avg lat (us)	0.116	76.2	18.3	19.8	+0.335	+4.77	-

Note: SMO finish modifies the leaf's migration metadata (leaf), then updates/inserts into the inner tree. There are 3 cases for inner tree modifications: A) normal leaf operation(s); B) leaf SMO; and C) leaf and inner node SMOs. A–C reports the extra inner-tree latency.

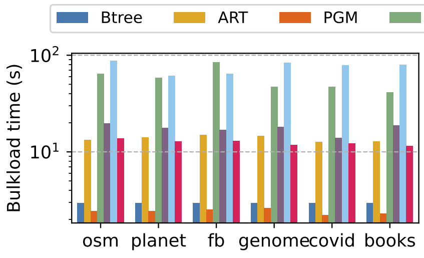

Fig. 10. Index construction.

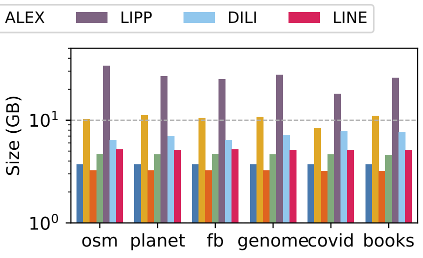

Fig. 11. Index size.

Table 5. LINE statistics after bulkloading 100M keys.

Data set	osm	planet	fb	genome	covid	books
Avg group size	11.9	11.9	11.9	12.0	12.0	12.0
Max group size	105	103	102	102	91	81
Avg n_{group}	40.3	69.8	45.5	167	436	1549
Start offset size	1.73MB	1.70MB	1.72MB	1.68MB	1.67MB	1.67MB
LINE leaf num	20911	12032	18516	5005	1913	538
Inner avg depth	1.10	1	1	1	0	0
Inner size	1.76MB	923KB	597KB	329KB	61.9KB	17.6KB
Leaf metadata size	1.84MB	1.06MB	1.63MB	440KB	168KB	47.3KB

about 0.001% of insertions incur leaf SMOs in the inner tree, and the cost of an inner leaf SMO is moderate as shown in column B. There are no non-leaf SMOs in the inner tree.

Index Construction and Index Size. Figure 10 shows the time for bulkloading 200 million keys. LINE's index construction performance is comparable to that of ART, and much better than ALEX and DILI. LINE's efficiency stems from the simplicity of its bulkload procedure.

Figure 11 compares the size of the index constructed by bulkloading 200 million keys across different data sets. We see that LINE's size is comparable to ALEX and DILI, and is smaller than ART and LIPP. LINE exhibits reasonable moderate memory consumption.

Index Structure Analysis. Table 5 shows the statistics of LINE after bulkloading 100 million keys on the six real-world data sets. (Statistics of other competitors are shown in Table 1.) We have the following observations. First, the average group size stays stable across the data sets. It is about 12

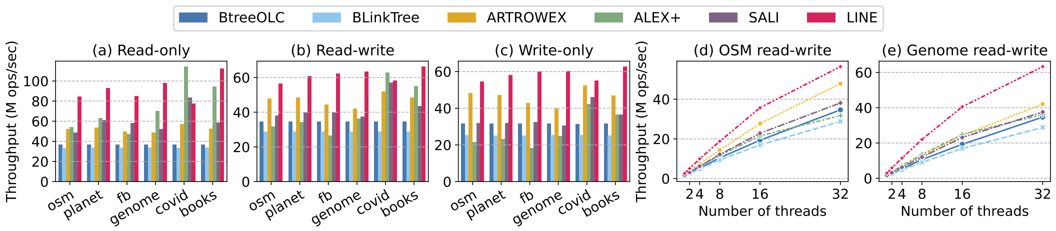

Fig. 12. Index performance varying read-write ratios with up to 32 threads.

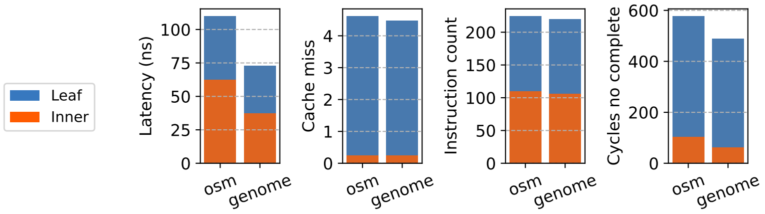

Fig. 13. Breakdown analysis for search operations in LINE.

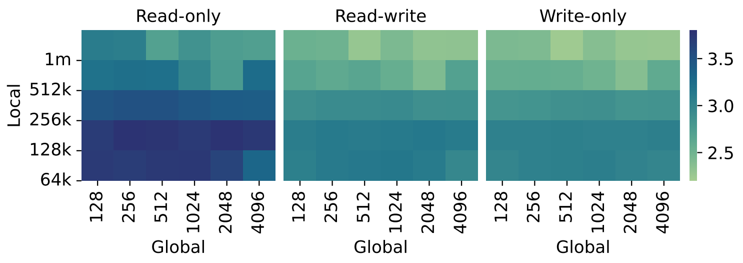

Fig. 14. Index performance normalized to that of B+-Tree across different hardness and read-write ratios.

rather than m=11 because our implementation allocates nanos even for empty groups, making the average group size slightly larger. Second, the number of leaves in LINE generally increases as the global hardness of the data set, leading to a larger inner tree. Third, the inner tree is very shallow. The average depth of the inner tree (excluding the skew-tolerant root and the inner ALEX's root node) is about 1 level for the four complex data sets and 0 level for the two simpler data sets. This means that for the complex data set, the inner subtree contains a subtree root and a single level of inner leaf nodes. For the simpler data sets, the inner subtree's root node directly stores the boundary keys of the LINE leaves. Finally, the inner subtree, the leaf metadata, and even the group start offsets can easily fit into the 22MB L3 cache, improving the performance of LINE operations.

Breakdown Analysis of Search Operations. Figure 13 shows the breakdown for the search operation into leaf and inner parts of LINE on osm and genome. To understand the time breakdown, Figure 13 also shows the breakdown of cache misses, number of executed instructions, and cycles with no instruction completion for the search. We see that the inner tree incurs almost no cache misses, but executes about 50% of the total instructions. The instruction cost contributes to the significant portion of time in the inner tree search. Moreover, the leaf search incurs about 4 cache misses which accounts for approximately one nano access on average. This proves that the search key can be found in the first hashed nano in most cases, leading to good performance. As the inner tree on osm is more complex than that on genome, osm sees a higher branch misprediction rate (8.87%) than genome (5.54%), incurring more wasted cycles with no instruction completion.

Table 6. LINE vs. LINE w/ fixed-sized groups (Mops/sec).

osm	**10m75**	**20m71**	**50m67**	**100m51**	**150m40**	**200m26**	**LINE**
Point	2.98	3.31	3.96	5.85	6.31	6.33	5.99
Range	0.32	0.31	0.22	0.15	0.11	0.09	0.44
genome	**10m71**	**20m70**	**50m63**	**100m45**	**150m25**	**200m23**	**LINE**
Point	5.68	6.30	6.66	7.33	7.45	7.64	6.97
Range	0.34	0.33	0.23	0.16	0.12	0.09	0.56

Note: *A*m*B* denotes *A* main and *B* overflow nanos per fixed-sized group.

Scalability. Figure 12 shows the multi-threaded index performance. Figure 12(a)–(c) show the index performance of read-only, balanced read-write, and write-only workloads using 32 threads across all six data sets. Figure 12(d) and (e) show the index performance for the balanced read-write workload on osm and genome while varying the number of threads from 1 to 32.

First, as shown in Figure 12(d) and (e), LINE achieves good scalability as the number of threads increases. The CPU contains 16 cores and 32 threads and cause the slight decrease of the line slope at 16. LINE scales almost linearly as the number of threads increases from 1 to 16. Second, for the read-only workload with 32 threads in Figure 12(a), LINE outperforms BtreeOLC, BlinkTree, ARTROWEX, ALEX+, and SALI by up to 3.1x, 3.3x, 2.1x, 1.8x, and 1.9x, respectively. For the write-only workloads in Figure 12(c), LINE demonstrates superior performance, outperforming BtreeOLC, BlinkTree, ARTROWEX, ALEX+, and SALI by up to 2.0x, 2.5x, 1.5x, 3.3x, and 2.0x, respectively. LINE achieves strong write scalability due to group-wise migration in leaf nodes, which enable concurrent group accesses in leaf nodes during SMOs. For the read-write workloads in Figure 12(b), LINE improves the performance of BtreeOLC, BlinkTree, ARTROWEX, ALEX+, and SALI by factors of up to 1.9x, 2.3x, 1.5x, 2.3x, and 1.7x.

Overall, LINE achieves good scalability. Multi-threaded LINE significantly out-performs the competitors in most cases.

Synthetic Data Sets Varying Local and Global Hardness. We study the impact of the local hardness and global hardness on the performance of LINE using synthetic data sets. Figure 14 shows the improvement of LINE compared to B+-Tree while varying local hardness on the y-axis, global hardness on the x-axis, and the read-write ratio in different sub-figures. We use B+-Tree as the normalization base because B+-Tree's throughput is not sensitive to the hardness of the data set when the number of keys is the same. Darker color shows higher speedup. LINE tends to perform better on simpler data sets due to simpler structures of the inner tree and fewer overflow keys in leaves. LINE exhibits lower speedups in more write-intensive workload because compared to B+-Tree, the SMOs of LINE are more costly. The results are consistent with those on real-world data sets. From the figure, we see that LINE achieves good performance for various combinations hardness and workload. This indicates that LINE is capable of effectively handling a wide range of underlying data characteristics.

7.3 Benefits of Individual Techniques

Group-Enhanced Leaf Nodes. To study the benefit of variable-sized groups, we implement a variant of LINE with fixed-sized groups. Specifically, we fix the number of main nanos per group and compute the number of overflow nanos to accommodate all keys in each group. Table 6 shows the point search and short range scan throughput varying the fixed group size. We see that as the group size increases, the search performance increases but the range scan performance degrades. This is because smaller groups require larger fractions of overflow nanos, which incur higher search

Table 7. Varying the target average group size m.

	m	6	11	21	41
osm	Search latency (ns)	176.9	165.8	163.0	160.9
	Short scan latency (ns)	1996	2295	2824	3752
	# keys accessed by short scan	249	336	479	722
	Initial load factor	0.59	0.66	0.71	0.73
genome	Search latency (ns)	147.0	142.7	143.2	143.8
	Short scan latency (ns)	1604	1777	2204	3143
	# keys accessed by short scan	197	250	371	611
	Initial load factor	0.59	0.67	0.71	0.74

Table 8. The impact of inner tree size.

	Version	Size(MB)	Throughput	Avg cache miss
osm	LINE	5.33	5.99 Mops	4.62
	Large	47.8	3.20 Mops	8.35
genome	LINE	2.45	6.97 Mops	4.47
	Large	40.6	3.56 Mops	7.83

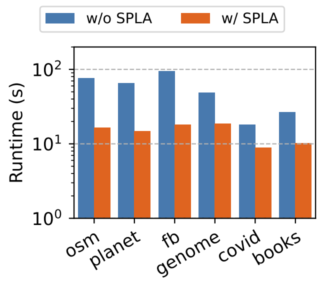

Fig. 15. Error bound selection with and without SPLA.

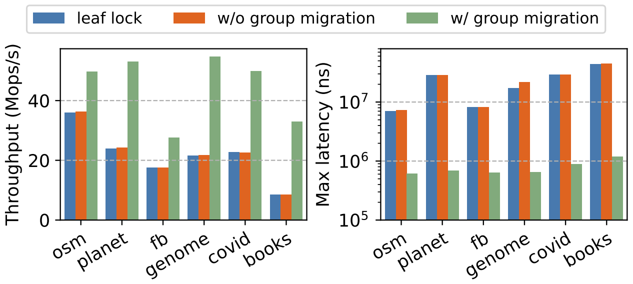

Fig. 16. Impact of group lock and group migration.

cost for visiting the overflow nanos. On the other hand, larger group size leads to more wasteful accesses for the range boundary groups. In comparison, LINE with variable-sized groups achieves both good search and good range scan performance.

We further study the influence of m (i.e., the target average group size) in Table 7. We vary the average number of main nanos per group (i.e., m-1) from 5, 10, 20, to 40. Table 7 reports the average search latency, the short scan latency, the number of keys accessed by short scans retrieving 100 keys, and the initial load factor. From the table, we see that the search latency remains stable as m increases. This is because LINE employs in-group hashing to find keys in a group. Moreover, the short range scan performance degrades as m grows because of the growth of unsorted region. LINE needs to fetch all keys in every group that overlaps with the given range. The latency and the number of keys accessed in a range scan confirm the result in Theorem 6.2. Finally, the initial load factor of small m is lower because there are more empty groups reserving a main and an overflow nanos with no keys, consuming extra space.

Cache-Optimized Inner Tree. To study the importance of cache-optimized inner tree, we set the error bound to 32 to create an index whose inner tree and leaf metadata exceed the L3 cache

Table 9. Benefit of skew-tolerant root.

	orig tree (70% gap)	**tree1**	**tree2**	**orig tree** (90% gap)	**tree1**	**tree2**
Max depth	2	1	1	12	2	1
Avg depth	1.00296	1	1	9.99	1.01	1

size. We call this variant *Large*. Table 8 compares the inner tree size, the search throughput, and the average number of cache misses per search for LINE and large. We see that compared to large, LINE with the cache-optimized inner tree incurs much fewer cache misses, achieving significantly better search throughput.

Figure 15 shows the runtime of error bound selection algorithm w/o or w/ SPLA. Using SPLA to find the cachable number of leaves instead of PLA can speedup the error bound selection by 2.1x to 5.2x. A more complex dataset with higher hardness benefits more from SPLA because it needs more rounds of segmentation.

For the skew-tolerant root, we test three sets of parameters (n_{intv}, s_{lg}, u_{skew}) = (100, 15, 10), (1000, 150, 145), and (10000, 1500, 1450) on real-world and synthetic data sets with gaps. (Note that s_{lg} and u_{skew} are derived from n_{intv} as described in Section 5.2.) We find that with different parameters, the generated subtrees on the same data set are the same. We choose n_{intv}=100 as the default setting. In all cases, the skewed gap detection takes less than 2ms.

Synthetic Data Sets Varying Global Gaps. Table 9 shows the benefit of skew-tolerant root. On a 70% gap synthetic dataset, the root divides a tree of height 2 to two trees of height 1. On a 90% gap synthetic dataset, the original inner tree is extremely high with an average depth of 9.99, the skew-tolerant root smooth this by generating two trees and reduce the height to approximately 1.

Group-Wise Migration. Figure 16 shows the impact of group lock and group-wise migration. From left to right, the three bars show leaf lock + whole leaf migration, group lock + whole leaf migration, and group lock + group-wise migration (i.e., LINE), respectively. Note that leaf locks essentially disable group-wise migration; group locks are the basis for group-wise migration. Decomposing the leaf SMO into group-wise migration leads to benefits both in improving the throughput and in reducing the maximum latency. This method accelerate the most-insert workload throughput by 1.4x to 3.8x and reduce the tail latency by up to 97%.

8 Conclusion and Discussion

In conclusion, we have presented LINE, a novel learned index for addressing the three design challenges of local hardness, global skew, and worst-case tail latency. Our comprehensive evaluation shows that LINE achieves significant performance improvement over state-of-the-art learned indices, demonstrating the effectiveness of our proposed solution.

Index Growing from an Empty Data Set. LINE requires an initial data set to learn the data characteristics during bulkloading. To support the situation where the index evolves from an empty data set, one way is to build a start index (e.g., B+-Tree) for the initial insertions. When the start index grows beyond a pre-defined size threshold (e.g., the L3 cache size), we use the keys in the start index as bulkloading keys to build the LINE. In this way, we can combine the start index and LINE to support the situation of growing from an empty data set.

Acknowledgments

This work is partially supported by National Key R&D Program of China (2023YFB4503600) and Natural Science Foundation of China (62172390). Shimin Chen is the corresponding author.

References

[1] Timo Bingmann. 2013. Stx b+ tree c++ template classes. https://panthema.net/2007/stx-btree/

[2] Supawit Chockchowwat. 2022. Tuning Hierarchical Learned Indexes on Disk and Beyond. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* ACM, 2515–2517.

[3] Zhaole Chu, Zhou Zhang, Peiquan Jin, Xiaoliang Wang, Yongping Luo, and Xujian Zhao. 2024. LIVAK: A High-Performance In-Memory Learned Index for Variable-Length Keys. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23-27, 2024.* ACM, 174:1–174:6.

[4] Google Cloud. 2017. OpenStreetMap. https://console.cloud.google.com/marketplace/details/openstreetmap/geo-openstreetmap

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010.* ACM, 143–154.

[6] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020.* OpenProceedings.org, 407–410.

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 969–984.

[8] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (2020), 74–86.

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

[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, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* ACM, 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 (2023), 258:1–258:25.

[12] Na Guo, Yaqi Wang, Wenli Sun, Yu Gu, Jianzhong Qi, Zhenghao Liu, Xiufeng Xia, and Ge Yu. 2024. Chameleon: Towards Update-Efficient Learned Indexing for Locally Skewed Data. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024.* IEEE, 4316–4328.

[13] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *CoRR* abs/1911.13014 (2019). arXiv:1911.13014 http://arxiv.org/abs/1911.13014

[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, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020.* ACM, 5:1–5:5.

[15] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings.* www.cidrdb.org.

[16] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* ACM, 489–504.

[17] Hai Lan, Zhifeng Bao, J. Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2024. A Fully On-Disk Updatable Learned Index. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024.* IEEE, 4856–4869.

[18] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013.* 38–49.

[19] 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, DaMoN 2016, San Francisco, CA, USA, June 27, 2016.* ACM, 3:1–3:8.

[20] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proc. VLDB Endow.* 15, 2 (2021), 321–334.

[21] 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 International Conference on Management of Data, SIGMOD Conference 2020, online conference*

[Portland, OR, USA], June 14-19, 2020. ACM, 2119–2133.

[22] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proc. VLDB Endow.* 16, 9 (2023), 2212–2224.

[23] Liang Liang, Guang Yang, Ali Hadian, Luis Alberto Croquevielle, and Thomas Heinis. 2024. SWIX: A Memory-efficient Sliding Window Learned Index. *Proc. ACM Manag. Data* 2, 1 (2024), 41:1–41:26.

[24] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.

[25] Zhuoxuan Liu and Shimin Chen. 2023. Pea Hash: A Performant Extendible Adaptive Hashing Index. *Proc. ACM Manag. Data* 1, 1 (2023), 108:1–108:25.

[26] Christian E. López and Caleb Gallemore. 2021. An augmented multilingual Twitter dataset for studying the COVID-19 infodemic. *Soc. Netw. Anal. Min.* 11, 1 (2021), 102. doi:10.1007/S13278-021-00825-0

[27] 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 (2021), 597–610.

[28] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: a Fully Learned Index for Larger-than-Memory Databases. *Proc. VLDB Endow.* 16, 3 (2022), 561–573.

[29] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 2789–2792.

[30] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Optimizing by Sandwiching. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada.* 462–471.

[31] Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1331–1344.

[32] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 985–1000.

[33] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proc. VLDB Endow.* 13, 11 (2020), 2341–2354.

[34] Suhas S.P. 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, and Erez Lieberman Aiden. 2014. A 3D Map of the Human Genome at Kilobase Resolution Reveals Principles of Chromatin Looping. *Cell* 159, 7 (2014), 1665–1680. doi:10.1016/j.cell.2014.11.021

[35] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, Michael Mitzenmacher, and Tim Kraska. 2022. Can Learned Models Replace Hash Functions? *Proc. VLDB Endow.* 16, 3 (2022), 532–545.

[36] Yufan Sheng, Xin Cao, Yixiang Fang, Kaiqi Zhao, Jianzhong Qi, Gao Cong, and Wenjie Zhang. 2023. WISK: A Workload-aware Learned Index for Spatial Keyword Queries. *Proc. ACM Manag. Data* 1, 2 (2023), 187:1–187:27.

[37] Yuanyuan Song, Miao Cai, Baoliu Ye, and Guo Cheng. 2024. SLIN: A CPU-efficient, Hybrid Tree and Learned Index for String Data. In *Proceedings of the 8th International Conference on Computing and Data Analysis, ICCDA 2024, Wenzhou, China, November 15-17, 2024.* ACM, 52–58.

[38] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (2023), 1992–2004.

[39] 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 *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020.* ACM, 308–320.

[40] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: a scalable learned index for string keys. In *APSys '20: 11th ACM SIGOPS Asia-Pacific Workshop on Systems, Tsukuba, Japan, August 24-25, 2020.* ACM, 17–24.

[41] Zhonghua Wang, Chen Ding, Fengguang Song, Kai Lu, Jiguang Wan, Zhihu Tan, Changsheng Xie, and Guokuan Li. 2024. WIPE: A Write-Optimized Learned Index for Persistent Memory. *ACM Trans. Archit. Code Optim.* 21, 2 (2024), 22.

[42] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 473–488.

[43] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (2022), 3004–3017.

[44] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288.

[45] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust Learned Index via Distribution Transformation. *Proc. VLDB Endow.* 15, 10 (2022), 2188–2200.

[46] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings. *Proc. VLDB Endow.* 17, 11 (2024), 3415–3427.

[47] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proc. VLDB Endow.* 15, 11 (2022), 2679–2691.

[48] Shunkang Zhang, Ji Qi, Xin Yao, and André Brinkmann. 2024. Hyper: A High-Performance and Memory-Efficient Learned Index via Hybrid Construction. *Proc. ACM Manag. Data* 2, 3 (2024), 145.

[49] Yu Zhang, Shiyu Yang, Wenlei Zhong, Guojie Ma, Jianye Yang, and Weihong Zhou. 2024. PLIS: Persistent Learned Index for Strings. In *Web Information Systems and Applications - 21st International Conference, WISA 2024, Yinchuan, China, August 2-4, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14883).* Springer, 251–263.

[50] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proc. VLDB Endow.* 16, 2 (2022), 243–255.

Received October 2025; revised January 2026; accepted February 2026