LITS: An Optimized Learned Index for Strings

Yifan Yang
SKLP, ACS, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
yangyifan22z@ict.ac.cn

Shimin Chen[*]
SKLP, ACS, Institute of Computing Technology, CAS
University of Chinese Academy of Sciences
chensm@ict.ac.cn

ABSTRACT

Index is an important component in database systems. Learned indexes have been shown to outperform traditional tree-based index structures for fixed-sized integer or floating point keys. However, the application of the learned solution to variable-length string keys is under-researched. Our experiments show that existing learned indexes for strings fail to outperform traditional string indexes, such as HOT and ART. String keys are long and variable sized, and often contain skewed prefixes, which make the last-mile search expensive, and adversely impact the capability of learned models to capture the skewed distribution of string keys.

In this paper, we propose a novel learned index for string keys, LITS (Learned Index with Hash-enhanced Prefix Table and Subtries). We propose an optimized learned model, combining a global Hash-enhanced Prefix Table (HPT) and a per-node local linear model to better distinguish string keys. Moreover, LITS exploits compact leaf nodes and hybrid structures with a PMSS model for efficient point and range operations. Our experimental results using eleven string data sets show that LITS achieves up to 2.43x and 2.27x improvement over HOT and ART for point operations, and attains comparable scan performance.

PVLDB Reference Format:
Yifan Yang and Shimin Chen. LITS: An Optimized Learned Index for Strings. PVLDB, 17(11): 3415-3427, 2024.
doi:10.14778/3681954.3682010

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

1 INTRODUCTION

Indexes play an essential role in modern database engines to accelerate transaction and query processing. Learned indexes have been shown to outperform traditional tree-based index structures for fixed-sized integer or floating point keys [13, 15–17, 20, 21, 23, 24, 27, 28]. However, this is hardly the case for variable-length string keys, which are common in the real world [9, 11].

While learned indexes have been extensively studied for fixed-sized integer or floating point keys in recent years, the application of the learned solution to variable-length string keys is under-researched with only a couple of studies [22, 26]. We experimentally compare the existing learned indexes for strings, i.e. SIndex [26] and RSS [22], with state-of-the-art traditional string indexes, i.e., ART [19] and HOT [10]. We find that existing learned indexes fail to outperform traditional indexes. In fact, traditional string indexes win by a large margin.

By examining real-world string data sets, we observe two distinct features of string keys that differ significantly from fixed-sized integer or floating point keys. First, string keys are often *long and variable sized*, making the key access and comparison more expensive. Second, string data sets see *skewed prefixes* among string keys. Popular prefixes shared by multiple strings make it difficult for learned models to distinguish individual string keys.

The two distinct features impact the tree height, the node search, and the last-mile search in index structures. For example, the last-mile search often requires expensive key comparisons, and therefore should be avoided as much as possible. Recent studies attempt to adapt CDF models for fixed sized keys (e.g. RMI [17], Radix Spline [16], and piece-wise linear models) to string data sets. However, the resulting learned models work poorly for capturing the skewed distribution of string keys, leading to large tree heights that degrade index performance.

In this paper, we propose a novel learned index for string keys, LITS (Learned Index with Hash-enhanced Prefix Table and Subtries). First, LITS employs the collision-driven design of LIPP [28] to avoid the last-mile search by creating a child node to store the keys that are mapped to the same slot. Second, we propose an optimized learned model, combining a global Hash-enhanced Prefix Table (HPT) and a per-node local linear model. The HPT approximates the conditional probability of the next character given a prefix in the string key. Compared with existing learned models, HPT can better distinguish string keys. Third, the collision-driven design can result in a large number of small leaf nodes containing two or only a few keys. Consequently, a scan may have to traverse many small nodes, incurring expensive cache misses and node jump overhead. We introduce the compact leaf node, which replaces a group of small nodes with a single node. Finally, we observe that trie-based index, such as HOT, is very efficient for highly skewed string data sets. Therefore, we combine our learned index and HOT using a performance model (PMSS) to determine whether a subtrie is more beneficial to be used in the place of a child node.

LITS supports common index operations on string keys, including bulkload, search, insert, delete, update, and range scans. It is specifically optimized for point operations. We conduct extensive experiments using seven real-world string data sets and four synthetic data sets. Our experimental results show that LITS achieves up to 2.06x and 2.14x improvement over HOT and ART for point operations, respectively. For the scan-heavy workload, LITS's performance is comparable with HOT and better than ART.

---

[*]Shimin Chen is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682010

Table 1: String data sets used in this work. (cf. Section 4.1)

Dataset	Min Len	Max Len	Avg Len	Number of Keys	Total Size
address	4B	133B	24B	34M	802MB
dblp	2B	255B†	76B	7M	506MB
geoname	2B	152B	13B	7M	106MB
imdb	2B	106B	13B	9M	132MB
reddit	3B	26B	11B	26M	292MB
url	12B	255B†	64B	63M	4.6GB
wiki	2B	255B†	15B	43M	870MB
email*	11B	47B	23B	45M	1.1GB
idcard*	18B	18B	18B	63M	1.2GB
phone*	11B	23B	17B	50M	819MB
rands*	2B	61B	32B	50M	1.6GB

†: The data set is processed to remove strings longer than 255B. The maximum key length of the unprocessed dblp is up to 1461B.
*: The data set is synthetically generated.

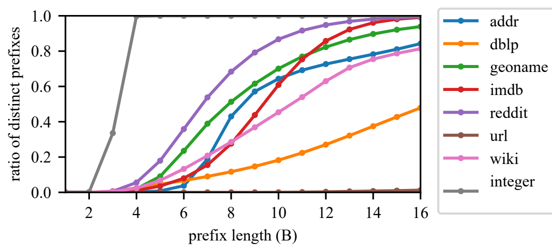


Figure 1: Prefix skewness of string keys.

Contributions. The contributions of the paper are as follows. First, we propose a novel HPT-based CDF model that exhibits strong discriminative power for string keys. Second, we propose LITS, a novel learned index for strings that exploits the HPT-based model, compact leaf nodes, and hybrid structures with a PMSS model for efficient point and range operations. Finally, we perform extensive experiments to compare our proposed LITS with five state-of-the-art string indexes using eleven string data sets. Our experiments show that LITS achieves the overall best performance.

Organization. The rest of the paper is organized as follows. Section 2 studies the characteristics of string keys and examine existing string indexes to motivate our study. Then, Section 3 presents the LITS design. Section 4 experimentally compares LITS with state-of-the-art string indexes. Finally, Section 5 concludes the paper.

2 BACKGROUND AND MOTIVATION

Learned indexes have been shown to outperform traditional tree-based index structures for fixed-sized integer or floating point keys [13, 15–17, 20, 21, 23, 24, 27, 28]. However, this is hardly the case for string keys. In the following, we study the characteristics of string keys in Section 2.1, then examine existing index structures optimized for strings to motivate our study in Section 2.2.

2.1 Characteristics of String Keys

Table 1 summarizes 7 real-world and 4 synthetic string data sets used in this work. (Please see detailed description in Section 4.1.) Focusing on the real-world data sets, we observe two features that are distinct from fixed-sized integer or floating point keys.

Long and Variable Sized Keys. Integer or floating point keys are typically of 4B or 8B large. In comparison, the string keys in the real-world data sets are much more complex. They can vary from 2B to over 1KB. The average key length of the real-world data sets is from 11B to 26B, which is much longer than 4B/8B keys. Consequently, storing entire string keys in index (inner) nodes can significantly reduce node fanouts, degrading index performance. On the other hand, storing pointers to string keys in index nodes causes pointer dereferences, incurring CPU cache misses. Moreover, the comparison of long keys is also more expensive.

Skewed Prefixes. The prefixes of string keys are often quite skewed. Figure 1 compares the prefix skewness of the real-world string data sets and a uniformly generated integer data set. For each prefix length k, we compute the ratio of distinct prefixes of a data set as the number of distinct k-byte prefixes divided by the total number of keys in the data set. This ratio is between 0 and 1. If it is closer to 1, then the data set is more evenly distributed. If it is closer to 0, then a few prefixes are very popular. A large number of keys share the same prefixes. The data set is more skewed. In Figure 1, we consider the prefix length when the ratio of a data set is over 0.99. For the integer data set, all keys can be distinguished by four bytes. In contrast, all real-world string data sets have very low ratios of distinct prefixes at 4B prefixes. For reddit, the ratio reaches 0.99 at 16B prefixes. The ratio of url gets to 0.99 at 154B prefixes. Consequently, it is necessary to examine a much larger number of bytes for distinguishing string keys, adversely impacting the effectiveness of learned models in learned indexes.

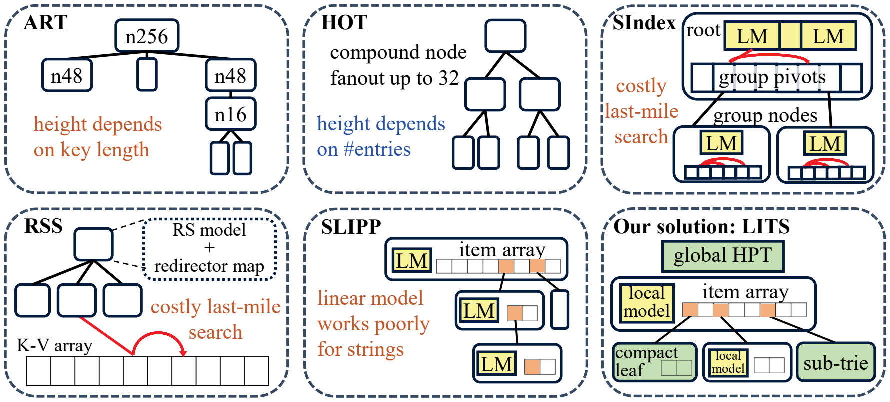
2.2 Existing Indexes Optimized for Strings

We focus on ordered indexes for strings in this paper. Figure 8 compares the search performance of five state-of-the-art index structures optimized for strings, including two trie-based indexes (i.e., ART [19] and HOT [10]), and three learned index based structures (i.e., SIndex [26], RSS [22], SLIPP, which is based on LIPP [28])[1]. From the figure, it is clear that existing learned indexes work poorly compared to traditional trie-based indexes. In the following, we examine the index design choices to understand the pros and cons of the existing index designs.

Index Performance Factors. Ordered indexes are typically organized as a tree. All the five state-of-the-art string indexes are essentially trees consisting of inner nodes and leaf nodes. A search often starts from the root of a tree, visits several inner nodes at different tree levels, and finally reaches a leaf node in the tree. Therefore, the number of tree levels from the root to the leaf and the search procedures at inner and leaf nodes are main factors influencing the index search performance:

- *Tree height*: The (average) tree height indicates the expected number of nodes accessed by an index search. Each node access often incurs an expensive CPU cache miss. These cache misses are dependent on each other since the memory address of the node at the next level is known only after searching the node at the previous level. Therefore, the tree height is an important performance factor. We see four cases for tree heights. First, the tree height is determined by (the logarithm of) the number of index entries (e.g., in B+-Trees). Second, the tree height is

[1]We do not include B+-Trees in the comparison because previous work has shown that trie-based indexes significantly outperform B+-Trees for string data sets [10]. Recent work on Extendible Radix Tree (ERT) enhances trie nodes with extendible hashing [25]. While the idea could potentially support string keys, the original paper and its code focus only on integer keys. Therefore, we do not consider ERT in this work.



	tree height	node search	last-mile search
ART	x	√	n/a
HOT	–	√	n/a
Sindex	√	–	x
RSS	√	√	x
SLIPP	x	√	n/a
LITS	√	√	n/a

x: poor; √: good; –: medium;
n/a: no last-mile search

Figure 2: Comparing index structures optimized for strings.

determined by the length of the index keys (e.g., in ART or RSS). Third, learned indexes introduce CDF models to predict the key positions in a node in order to increase node sizes and reduce the tree height. The tree height is *model-based*. Finally, SIndex [26] constructs a two-level tree with a root node and a number of group nodes.

- *Node search*: Search in an inner node narrows down the search scope to a subtree of the node. Search in a leaf node locates the target index entry. Both often follow similar search procedures. There are mainly three ways to support node search. First, it can be based on *key comparisons* over the full keys or on part of the keys, e.g., binary search in a sorted B+Tree node. Second, it can perform an *array lookup*, e.g., in a common trie node, such as node256 in ART. Finally, learned indexes often conduct *model-based* search, which employs a learned model to predict the location of the search key in the key array.
- *Last-mile search*: In learned indexes, model prediction in inner nodes is often accurate by construction. Index keys are mapped to subtrees of an inner node using the associated model during bulkload and write operations. However, the models in leaf nodes may not predict the correct key positions. Therefore, learned indexes often have to do extra work, a.k.a. *last-mile search*, to locate the key around the predicted position in leaf nodes. The last-mile search often performs key comparisons (e.g., with exponential or binary search), and can incur poor performance for learned indexes with fixed-sized keys [27]. The situation is even worse for long and variable sized keys because of pointer dereferences and higher key comparison costs.

Pros and Cons of Existing Indexes. We examine the three performance factors of the five string indexes. Figure 2 compares the five indexes and our proposed solution.

- *ART*: Adaptive Radix Tree (ART) [19] is a compressed trie. Each level of an ART uses a byte in the keys. Therefore, the height of an ART is determined by the key lengths, which can be quite large for string data sets. ART compresses the trie node to four types of nodes (i.e., Node4/16/48/256) to reflect the effective node fanouts. The node search in Node48 and Node256 performs array lookups, while Node4 and Node16 employ key comparison based search. Both procedures are fast because ART searches 1B in every node. There is no last-mile search.
- *HOT*: Height Optimized Trie (HOT) [10] optimizes ART by reducing the tree height. Each inner node of a HOT is a compound node that represents a Patricia trie with a fanout of up to 32. This is achieved by carefully storing only a subset of distinct key bits (a.k.a. partial keys) in each node. As a result, HOT often reduces the tree height significantly for long string keys. Its height can be viewed as roughly determined by the number of index entries. Moreover, the node search compares partial keys with efficient SIMD operations. There is no last-mile search.
- *SIndex*: SIndex [26] is a two-level tree consisting of a root node and a level of group nodes. The root node employs a piece-wise linear model (PLM) and divides the key space into key groups. Then, each group node uses a linear model (LM) to locate an index entry. Since the model prediction is not fully accurate, SIndex performs a last-mile search, which is a binary search within the error bound around the predicted location, in both the root and the group nodes. The last-mile search incurs significant performance overhead.
- *RSS*: Radix String Spline (RSS) [22] is a trie. Each trie node uses 8B or 16B of the keys, and computes a Radix Spline (RS) model [16] for them. The RS model is a piece-wise linear model that provides monotonic CDF prediction with a given error bound. For keys dissatisfying the error bound (e.g., because of shared prefixes), RSS stores the key in the redirector map and creates a new child node for the key. RSS compares 8B/16B portions of keys, which has lower cost than full key comparison. However, the last-mile search is still very costly. In the search experiments, RSS spends over 70% of the time in the last-mile search.
- *SLIPP*. LIPP [28] is an interesting learned index with fixed-sized keys because it avoids the costly last-mile search. In each inner node, LIPP trains a linear model. If multiple keys are mapped to the same entry slot by the linear model, LIPP creates a new child node for the collision keys. This design is known as the collision-driven approach. It essentially converts the last-mile search into a sub-tree search. We implement a variant of LIPP, called *SLIPP*, to support string keys. At each node, SLIPP computes a numeric representation of each string key (after excluding the common prefix of the keys in the node) using a straight-forward formula: $y = \frac{s_1}{256} + \cdots + \frac{s_{len}}{256^{len}}$, where s_i is the i-th byte of the key. Then, it computes the linear model based on the numeric representation.

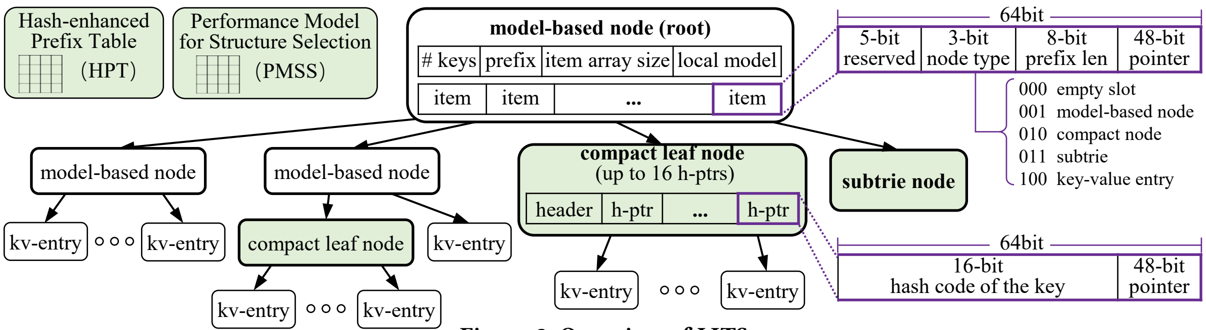


Figure 3: Overview of LITS.

The other design of LIPP is kept unchanged. While SLIPP avoids the last-mile search, the computed model can hardly distinguish keys with skewed prefixes, resulting in many collisions and large tree heights.

Motivation. The above discussion focuses on search, which is representative of point operations. The comparison of the five string index structures motivate us to design an optimized learned index for strings that avoids the last-mile search and improves the effectiveness of the learned models for reducing the tree height.

3 LITS

We propose LITS (Learned Index with Hash-enhanced Prefix Table and Sub-tries) for string keys in this section. Section 3.1 overviews the structure and operations of LITS. Then, Section 3.2, 3.3, and 3.4 present the three main techniques of LITS, optimizing the learned model, accelerating scans with compact leaf nodes, and exploiting subtries to further improve performance, respectively. Finally, Section 3.5 describes the time and space cost of LITS.

3.1 Overview of LITS

Figure 3 depicts the structures in LITS. We describe the distinct features of LITS (highlighted in the figure) in the following.

- *Model-based node*: To deal with the problem of the last-mile search, as discussed in Section 2.2, we employ the collision-driven design of LIPP to avoid the last-mile search. Specifically, a model-based node consists of a header and an item array. The header contains metadata, such as the number of keys, the key prefix, the size of the item array, and a local linear model. Each slot in the item array is a 64-bit pointer. We store additional information in the upper bits of the pointers, which are otherwise unused in current machines. Keys are mapped to slots in the array with an optimized learned model (discussed in more detail below). There are three cases. First, a slot is empty. Then, it contains a NULL item. Second, only a single key is mapped to a slot. Then, the slot holds a pointer to the key-value entry. Third, multiple keys are mapped to the same slot. Then, LITS creates a child node to store the keys to avoid the last-mile search. The node type field in the 64-bit item indicates the different child node types.
- *Optimized global HPT and local models*: As discussed in Section 2.2, due to skewed prefixes and long keys, existing learned models work poorly for string keys. This results in the large tree height in SLIPP, lowering index performance. We propose an optimized learned model, combining a global Hash-enhanced Prefix Table (HPT) and a per-node local linear model to effectively distinguish string keys. (cf. Section 3.2)
- *Compact leaf node*: The collision-driven design in model-based nodes lead to a large number of small leaf nodes that contain two or only a few kv-pointers. However, a scan has to traverse many such small leaf nodes, and suffers from expensive cache misses and node jump overhead. We introduce the compact leaf node to make the design scan-friendly. A compact node contains a header and an array of h-pointers sorted in the key order. An h-pointer consists of a 16-bit computed hash of the key and a 48-bit pointer to the kv-entry. In this way, we replace a number of small leaf nodes with a single compact leaf node, thereby reducing the number of node visits in scans. (cf. Section 3.3)
- *Subtrie node and PMSS*: We call the resulting index with the above techniques, LIT. Our experiments show that LIT outperforms all the five existing indexes for most data sets, but it is slightly slower than the trie-based index (i.e., HOT) for a couple of data sets. Therefore, we propose to combine LIT and trie-based indexes. We build a performance model (i.e., PMSS) to determine whether a subtrie is more beneficial to be used in the place of a child node. The combined structure of LIT with subtries is our final proposed solution, LITS. (cf. Section 3.4)

After overviewing the structures of LITS, we describe the common index operations in the following.

Search. A search goes from the root node to a leaf node. Based on the node type, LITS performs different search procedures.

First, in a model-based node, LITS compares the common prefix recorded in the node header with the search key. Most commonly, the prefixes match. Then, LITS skips the prefix and uses the remaining substring of the search key to predict the slot position based on the global HPT and the local model. The position is between 1 and $ItemArraySize - 2$. In rare situations, the prefixes do not match. We preserve the first (the last) item of the item array for the case where the search key prefix is less (greater) than the recorded prefix. Then, LITS gets the target item according to the search key. If the item is NULL, the search key does not exist. Otherwise, LITS dereferences the pointer to visit the child node / kv-entry.

Second, in a compact leaf node, LITS performs a key comparison based search. It dereferences an h-pointer only if the hash of the search key matches the hash in the h-pointer.

Third, if the node is a subtrie node, LITS calls the search procedure of the subtrie (e.g., HOT) to continue the search.

Finally, upon reaching a kv-entry, LITS compares the search key with the key in the kv-entry. The search succeeds if it is a match.

Insert/Delete/Update. An insert first performs the search for the input key. If the search gets to an empty item, LITS simply inserts the new kv-entry to the empty slot. If the search gets to a single-entry item (i.e., a pointer to a kv-entry), LITS builds a new compact leaf node to contain the new key and the existing key. If the search gets to a compact leaf node, then LITS inserts the key to the compact leaf node if there are less than 16 entries. When the compact node already contains 16 entries, LITS performs the PMSS-based decision and replaces the compact node with either a model-based node or a subtrie. If the search gets to a subtrie, then LITS calls the insert procedure of the subtrie to complete the insertion.

The delete or update procedures work similarly. For an update, LITS searches the key and either modifies the value in the kv-entry or changes the item or h-pointer to point to a new kv-entry. For a delete, LITS removes the key if it is found. This clears the single-entry item, or reduces the number of keys in a compact node. The situation is a little complicated for subtrie deletion because HOT does not implement the delete function. We employ a delete list to hold the deleted keys associated with a subtrie. If the number of deleted keys is beyond a predefined ratio of the keys in the subtrie, we reconstruct the subtrie.

When a model-based node contains too many (or too few) keys, LITS follows a procedure similar to LIPP to perform node resizing operations. Moreover, when inserting to a compact leaf node with 16 entries, LITS uses the PMSS to determine if a model-based node or a subtrie should be constructed. (Please see Section 3.4 for all cases where PMSS-based decision is performed.) Finally, we adapt the classic method of optimistic locking, which protects each node with a lock and allows reading a node without locking it, to support concurrent threads.

Scan. Given a begin key, a scan searches the begin key and constructs an iterator. Using the iterator, one can obtain a list of kv-pointers sorted by the key order by repeatedly calling the iterator's next method. Internally, the scan maintains a stack of pointers to nodes from the root to the current leaf node. Both the items in model-based nodes and the h-pointers in compact leaf nodes are sorted in the key order. Therefore, the scan can easily traverse the nodes in the tree using the stack.

Bulkload. At the beginning, LITS samples a subset of keys and compute the global HPT model. Then, LITS bulkloads the tree in a similar fashion as LIPP. There are two main differences. First, for a sub range of data, LITS chooses which node type to build. If the number of keys is at most 16, then LITS builds a compact node. If there are more keys, LITS uses the PMSS to choose between a model-based node and a subtrie. Second, when constructing a model-based node, LITS uses the global HPT to compute the local linear model for the keys in the node.

3.2 Hash-enhanced Prefix Table (HPT)

We would like to design a good learned model to better distinguish string keys. In the following, we begin by deriving a recursive formula for CDF computation. Using the formula, we explain why previous linear models work poorly. Then, we propose our HPT-based model to better approximate the CDF. Finally, we describe the training and computation procedure using the HPT.

Recursive Formula for CDF Computation. Given a string data set and a string S=$s_1 ... s_n$, we would like to compute $cdf(S)$. For brevity of presentation, we prepend a special (non-existent) beginning character s_0 to every string. Hence, S=$s_0 s_1 ... s_n$. We denote the k-byte prefix of the string as $\mathcal{P}_k$=$s_0 s_1 ... s_k$. Therefore, $S \equiv \mathcal{P}_n$. Then, we have the following recursive formula:

$$\begin{aligned} &cdf(\mathcal{P}_0) = 0 \\ &cdf(\mathcal{P}_{k+1}) = cdf(\mathcal{P}_k) + prob(\mathcal{P}_k) \times \Sigma_{c=0}^{s_{k+1}-1} prob(c|\mathcal{P}_k) \end{aligned} \tag{1}$$

Here, $prob(\mathcal{P}_k)$ represents the probability of prefix $\mathcal{P}_k$ in the string data set. $prob(c|\mathcal{P}_k)$ stands for the conditional probability of the next character being c given the prefix $\mathcal{P}_k$.

We can also derive a recursive formula for $prob(\mathcal{P}_k)$ as follows:

$$\begin{aligned} &prob(\mathcal{P}_0) = 1 \\ &prob(\mathcal{P}_{k+1}) = prob(\mathcal{P}_k) \times prob(s_{k+1}|\mathcal{P}_k) \end{aligned} \tag{2}$$

From Eqn 1 and 2, it is clear that obtaining $prob(c|\mathcal{P}_k)$ for any prefix $\mathcal{P}_k$ and any character c is crucial for computing $cdf(S)$.

Problem of Existing Linear Models. Existing linear models predict the position of a string S=$s_1 ... s_n$ as a linear function: $y(S) = \alpha \times x + \beta$, where $x = \Sigma_{k=1}^{m} \frac{s_k}{256^k}$. SLIPP computes x based on the full string, and hence m=n. RSS uses an 8B or 16B portion of the keys in each node in the model prediction. Therefore, m= 8 or 16 in RSS. We can rewrite the formula in a recursive fashion as follows:

$$y(\mathcal{P}_{k+1}) = y(\mathcal{P}_k) + \frac{\alpha}{256^k} \times \frac{s_{k+1}}{256} \tag{3}$$

Since $y(S)$ is a scaled version of $cdf(S)$, we can compare Eqn 1 and 3. $\frac{\alpha}{256^k}$ corresponds to a scaled version of $prob(\mathcal{P}_k)$, and $\frac{s_{k+1}}{256}$ corresponds to $\Sigma_{c=0}^{s_{k+1}-1} prob(c|\mathcal{P}_k)$. $\frac{s_{k+1}}{256}$ implies that any 8-bit character appears uniformly at random. Therefore, the existing linear models essentially assume that the distribution of the next character following any given prefix is uniform. This estimation can hardly reflect the true distribution in a string data set, which often contains highly skewed prefixes.

Our Solution: HPT. We would like to better approximate the conditional probability $prob(c|\mathcal{P})$. Existing neural network-based nonlinear CDF models [8, 14] are more accurate than linear models. However, these models are complex. The model training and model prediction are time consuming. It would be an over-kill for our goal of designing efficient string indexes.

A naïve idea is to record the conditional probabilities for all possible (prefix, character) pairs in the string data set. However, such an approach would require prohibitively large space to store the conditional probabilities.

To reduce the space overhead, we propose the Hash-enhanced Prefix Table (HPT). As illustrated in Figure 4, the HPT is a table (i.e., 2D array). For any prefix, we map the prefix to a row in the table using a hash function. (We set the hash of the empty prefix, $hash(s_0)$=0.) Then, each column corresponds to a character in the character set. We approximate the conditional probability $prob(c|\mathcal{P})$ with table lookups as HPT[$hash(\mathcal{P})$][$c + 1$].cdf - HPT[$hash(\mathcal{P})$][c].cdf. We have stored this value in the model as HPT[$hash(\mathcal{P})$][c].prob. Note that HPT[$hash(\mathcal{P})$][c].cdf approximates $cdf(c|\mathcal{P})$, which is $\Sigma_0^{c-1} prob(i|\mathcal{P})$. This reduces the complexity for the CDF computation.

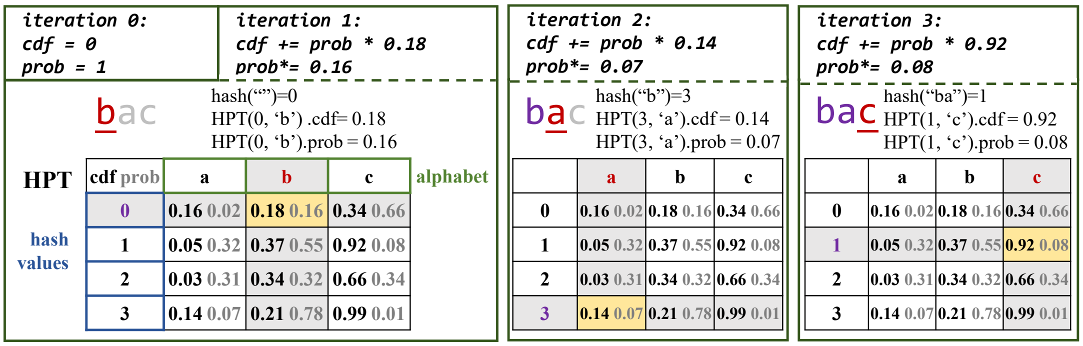


Figure 4: An illustration of the CDF computation using the HPT for string “bac”. (purple: prefix; red: current character)

Algorithm 1 HPT-based CDF computation.

```
1: procedure GetCDF(HPT, string S)
2:     cdf = 0, prob = 1
3:     for k = 0 to len(S) − 1 do
4:         hashval = (k == 0 ? 0 : hash(S[0:k-1]))
5:         c = S[k]
6:         cdf += prob * HPT[hashval][c].cdf
7:         prob *= HPT[hashval][c].prob
8:     return cdf
```

HPT Construction. The construction of the HPT is simple. We randomly sample a small fraction (e.g., 1%) of the string data set during bulkloading, and compute the HPT using the sample. First, we initialize the HPT table to all 0s. Second, we iterate through all the string keys in the sample. For each string, we extract all (prefix $\mathcal{P}$, character c) pairs, and increment the corresponding cell HPT[$hash(\mathcal{P})$][c]. After the processing, each cell contains the frequency of ($hash(\mathcal{P})$, c). Finally, we process each row in the HPT. We compute the accumulate frequencies, and divide them by the total frequencies in the row to obtain $cdf(c|hash(\mathcal{P}))$. We store $cdf(c|hash(\mathcal{P}))$ in HPT[$hash(\mathcal{P})$][c].cdf and $cdf(c+1|hash(\mathcal{P}))$ - $cdf(c|hash(\mathcal{P}))$ in HPT[$hash(\mathcal{P})$][c].prob.

Model Prediction. Algorithm 1 shows the computation of $cdf(S)$ using the HPT. The initialization in Line 2 corresponds to $cdf(\mathcal{P}_0)$ and $prob(\mathcal{P}_0)$. Then, we use Eqn 1 (Line 6) and Eqn 2 (Line 7) to iteratively compute the CDF and probability of the current prefix, respectively. To reduce the cost of the hash computation in Line 4, we keep an internal state and incrementally update the state with the next character in the string. Then, we can compute the hash value of the prefix with $O(1)$ cost. The loop proceeds until the CDF of the string S is computed. Figure 4 shows an example computation of $cdf(bac)$.

LITS combines the global HPT and the per-node linear model in model prediction. In a model-based node, the predicted position for string S is computed as $y(S) = \alpha \times x + \beta$, where $x = GetCDF$(HPT, S). Note that we exclude the common prefix in this computation.

Benefits of the HPT-Based Model. First, compared to the uniform assumption in the existing linear models, our HPT-based model better captures the distribution of the string data set. Therefore, it can distinguish string keys more effectively. Second, the hashing design in the HPT reduces the space overhead for recording the conditional probability distributions. One can adjust the number of HPT rows to balance the space cost and the estimation quality. The larger the HPT table, the higher the estimation quality. However, a very large HPT table not only causes significant space overhead, but also incurs random memory accesses and CPU cache misses for HPT lookups. Therefore, we set the HPT table size (e.g., 2MB in our experiments) to be small enough to fit in the CPU cache. Finally, our design is computationally efficient. The HPT construction using a sample of string keys is simple and fast. Storing the conditional CDFs in the HPT reduces the cost for computing the sum term in Eqn 1. Hence, model prediction takes $O(len(S))$ time.

Analysis of HPT Accuracy. We have the following theorem for the accuracy of approximating the conditional probability. (Please refer to the extended version of the paper [29] for the proof.)

Theorem 3.1. *If prefix $\mathcal{P}$ appears $n_{\mathcal{P}}$ times in the string data set, and the HPT[hash($\mathcal{P}$)] row sees d occurrences of other prefixes, then*

$$|HPT[hash(\mathcal{P})][c].prob - prob(c|\mathcal{P})| \le \frac{1}{\frac{n_{\mathcal{P}}}{d}+1}.$$

For a popular prefix $\mathcal{P}$, we expect $n_{\mathcal{P}} \gg d$ with a reasonable sized HPT (e.g., 2MB). In such cases, the absolute error of the HPT approximation is small. Our experiments confirm this result. For the string data sets in our experiments, the average absolute error of the conditional probability is 0.0006–0.006 for popular prefixes that appear at least 10,000 times.

Dealing with Data Distribution Changes. If the data distribution changes, HPT may become less accurate, leading to degraded index performance. To handle data changes, LITS can sample the index performance (e.g., for 1% of the queries). If it observes that the index performance falls below a pre-defined water mark (e.g., 50% of the average performance after bulkloading), LITS can judiciously retrain the HPT model and rebuild the entire index.

3.3 Compact Leaf Node

The collision-drive design in the model-based nodes avoids the last-mile search by creating new child nodes. However, we observe that it can result in small nodes with only two or a few keys, as illustrated in Figure 5. The figure depicts a subtree with four model-based nodes, i.e., $N0$–$N3$. The four nodes are in three different tree levels, while the entire subtree rooted at $N0$ contains only five kv-entries. This subtree structure is sub-optimal for the following reasons. First, it degrades scan performance. Suppose $kv1$–$kv5$ are retrieved by a scan operation. Then the scan has to traverse four nodes in three levels, incurring expensive CPU cache misses and significant book-keeping overhead for entering nodes and backtracing. Second, the small nodes tend to increase the tree height, adversely impacting point operations. As shown in the example, $kv1$–$kv4$ are located

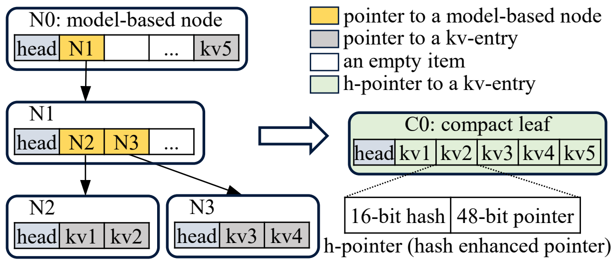


Figure 5: Replacing multiple nodes with a compact leaf node.

two levels deeper than $kv5$. A search for $kv1$ is more costly than $kv5$. Finally, the small nodes increase the space cost for storing the per-node headers and the child node pointers.

To address this problem, we replace a number of small nodes with a single compact leaf node. As illustrated in Figure 5, the four nodes are replaced with a compact leaf node, holding the five kv-entries. Each h-pointer stores a 16-bit hash of the key for better search performance. A search in a compact node sequentially compares the hash of the search key with the hash in every h-pointer. Only when there is a match does LITS dereference the pointer to visit the kv-entry. The false positive rate with the 16-bit hash is 0.0015%. Compared to the common binary search, the h-pointer based search can effectively avoid the high cost of unnecessary kv-entry dereference and key comparison. Moreover, the h-pointer array is sorted in the key order so that the scan iterator avoids the cost of sorting the keys in the compact node.

We discuss two important design choices of the compact node.

- *Size threshold w of compact nodes*: A compact node can hold up to w keys. If w is too small, compact nodes may not effectively reduce the small nodes in the index. On the other hand, if w is too large, the search performance suffers because it takes $O(w)$ time to sequentially examine the h-pointers. In Section 4, we study the impact of w on LITS performance and set w=16 based on the experiments.
- *Method to support inserts*: We consider two methods to support inserts. First, each compact node contains an array of w slots. If there are k keys, then $w-k$ slots are empty. An insert operation places the new key into the existing array. It moves existing elements to keep the sort order. Second, an alternative method is to make the node compact. A compact node with k keys is stored in an array of k slots. No space is reserved for empty slots. Then an insert operation creates a new compact node with one more slot to hold both the existing keys and the new key. Our experiments find that the first method sees substantial space waste because of the reserved empty slots, and both methods have similar performance. Therefore, we choose the second method as the default design for the compact node.

3.4 LIT Enhanced with Subtries

We call the learned index using HPT-based models and compact nodes, LIT (Learned Index with Hash-enhanced Prefix Table). In the following, we present a hardness metric, GPKL, for string data sets. We compare LIT and trie-based indexes experimentally, and combine LIT with HOT using a GPKL-based performance model to further improve index performance.

Hardness of String Data Sets. Previous study on learned indexes defined a hardness metric for data sets with integer or floating point keys [27]. The metric reflects the difficulty of applying linear models to approximate the CDF of the data set. However, this metric cannot be directly applied to string data sets because linear models hardly capture the properties of string data sets, as shown in Section 3.2. In this work, we propose a new hardness metric, GPKL (Group Partial Key Length), for strings.

Definition 3.1 (Common Prefix Length). *The common prefix length of a list $\mathcal{L}$ of strings, denoted as $cpl(\mathcal{L})$, is the length of the longest prefix shared by all strings in $\mathcal{L}$.*

Definition 3.2 (Partial Key Length). *Given a sorted list $\mathcal{L}$ of strings, the partial key of the i-th string S_i in $\mathcal{L}$ is the shortest substring of S_i that distinguishes S_i from S_{i-1} and S_{i+1} after removing the common prefix of $\mathcal{L}$. The partial key length of S_i, denoted as $pkl(\mathcal{L}, S_i)$, is the length of S_i's partial key.*

$pkl(\mathcal{L}, S_i)$ can be computed with common prefix lengths as follows:

$$pkl(\mathcal{L}, S_i) = max(cpl(\{S_{i-1}, S_i\}), cpl(\{S_i, S_{i+1}\})) + 1 - cpl(\mathcal{L}) \quad (4)$$

$cpl(\{S_a, S_b\}) + 1$ gives the smallest prefix length to distinguish S_a and S_b. Hence, the max term plus 1 shows the length of the shortest prefix that distinguishes S_i from S_{i-1} and S_{i+1}. Then, $pkl(\mathcal{L}, S_i)$ is obtained by subtracting the common prefix length of all strings in $\mathcal{L}$ from this shortest prefix length.

Definition 3.3 (Group Partial Key Length). *The group partial key length (GPKL) of a sorted list $\mathcal{L}$ of strings is the average of the partial key lengths of strings in $\mathcal{L}$: $gpkl(\mathcal{L}) = \frac{1}{|\mathcal{L}|}\sum_{S\in\mathcal{L}} pkl(\mathcal{L}, S)$.*

We choose GPKL as the hardness metric for strings for the following reasons. First, GPKL measures the difficulty of distinguishing keys in a string data set. The larger the GPKL, the more key bytes are necessary to distinguish the strings. Therefore, the metric reflects the hardness of modeling the string data set. Second, GPKL skips the common prefix of strings. This behavior mimics the design of inner nodes in most string indexes, including HOT, ART, Sindex, RSS, and LIT. Finally, GPKL can be computed efficiently by reading the sorted list of strings in one pass. This makes it possible to compute the GPKL online for structure selection decisions.

We define both a global GPKL and a local GPKL metric. Given a sorted list of strings, the global GPKL is the GPKL of the entire list. To compute the local GPKL, we divide the sorted list into disjoint sublists containing g consecutive strings in the list. We obtain the GPKL for each sublist, then compute the average of the sublist GPKLs as the local GPKL. We set g = 32 in the following.

Impact of Hardness on Index Performance. HOT and ART significantly outperform existing learned indexes for strings, as shown in Section 2.2. Hence, we are interested in comparing LIT with HOT and ART. Table 2 reports the index throughput for both a read-only workload and a write-only workload. For the read-only workload, we randomly search 20 million keys after bulkloading an index with all keys. For the write-only workload, we bulkload an index with 50% of the keys, and then we measure the throughput of randomly inserting the rest of the keys into the index.

As shown in Table 2, the data sets are arranged in the order of increasing global GPKLs. We see that LIT achieves the best read

Table 2: **Impact of hardness on index performance (Mops).**

String Dataset	Global GPKL	Local GPKL	Read-Only LIT	Read-Only HOT	Read-Only ART	Write-Only LIT	Write-Only HOT	Write-Only ART
`rands*`	6.12	2.42	**3.37**	2.62	3.24	**2.41**	1.03	1.47
`reddit`	8.24	3.48	**3.01**	1.90	2.39	**1.74**	1.17	1.52
`geoname`	10.36	4.75	**2.88**	2.27	2.27	**1.62**	1.26	1.45
`imdb`	10.51	3.79	**2.63**	2.00	1.97	**1.88**	1.23	1.35
`phone*`	10.84	4.01	**2.92**	2.01	2.38	**1.53**	1.18	1.43
`address`	12.61	6.55	**2.23**	2.08	1.83	**1.52**	0.94	1.19
`idcard*`	12.89	5.04	**3.19**	1.92	1.62	**2.01**	1.03	1.02
`wiki`	14.32	6.23	**1.94**	1.68	1.36	**1.17**	0.98	1.10
`email*`	15.32	5.86	1.88	**1.89**	1.11	**1.06**	0.92	1.00
`dblp`	20.79	10.19	1.55	**1.93**	1.30	**0.88**	0.72	0.83
`url`	47.61	17.79	0.83	**1.27**	0.78	0.54	**0.68**	0.58

note: * indicates that the data set is synthetically generated.

throughput for 8 data sets and the best write throughput for 10 out of the 11 data sets. However, for the datasets with the highest hardness values, trie-based indexes have higher performance. Specifically, HOT has the best read performance for `email`, `dblp`, and `url`, and the best write performance for `url`. This finding motivates us to combine the strengths of LIT and HOT.

Performance Model for Structure Selection (PMSS). To combine LIT and HOT, our basic idea is to make a decision to choose from either LIT or HOT when creating a node for a subset of string keys. Obviously, it would be too costly to experimentally compare the two choices online. Therefore, we develop a performance model (PMSS) to make quick and accurate online decisions.

The PMSS model works as follows. We choose GPKL and the number (n) of strings as two important metrics to characterize a subset of strings. For a given index, the PMSS model provides two functions, $readlat(gpkl, n)$ and $writelat(gpkl, n)$, which estimate the index search latency and the index insert latency, respectively. A target workload is specified as containing f_r fraction of reads and f_w fraction of writes, where $f_r + f_w = 1$. (Operation statistics can be updated online to estimate the f_r/f_w parameters.) Then, we estimate the average latency of index operations in the target workload as follows:

$$latency = f_r \cdot readlat(gpkl, n) + f_w \cdot writelat(gpkl, n) \quad (5)$$

We estimate the latency for each index, and select the design with the lowest latency for the given subset of strings. Figure 6 illustrates this decision process using the PMSS.

To obtain $readlat(gpkl, n)$ and $writelat(gpkl, n)$, we perform a set of offline benchmarking tests using synthetically generated data for various combinations of $gpkl$ and n, and populate a $readlat$ table and a $writelat$ table for each index (i.e., LIT and HOT). In our experiments, we populate the tables for $gpkl$=3, 5, ..., 21, and n= 2^4, 2^5, ..., 2^{25}. The total size of latency tables for LIT and HOT is less than 10KB. Then, for a specific ($gpkl$, n), we can use the latency tables to easily compute $readlat(gpkl, n)$ and $writelat(gpkl, n)$.

Figure 7 displays the results of offline benchmarking tests for the read-only workload. Figure 7(a) shows a heat map. We divide the $readlat(gpkl, n)$ of HOT by that of LIT and then use different colors to represent the speedup. The darker color shows where HOT wins, while the brighter color shows where LIT wins. We see that for a fixed $gpkl$, LIT exhibits a leading advantage as the data size increases. This can be explained by Figure 7(b). As the number (n) of keys increases, the height of HOT increases significantly,

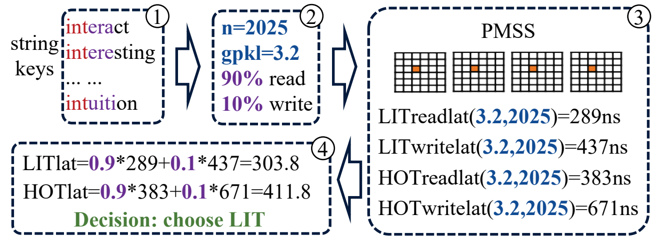
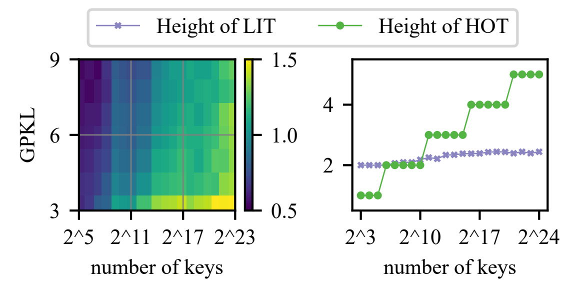


Figure 6: The decision process with PMSS.



(a) speedup of LIT over HOT (b) tree height (GPKL=5)

Figure 7: Offline benchmark tests for read-only workload.

roughly following $\log_{32}(n)$, while the height of LIT only changes slightly. As a result, LIT outperforms HOT.

One interesting detail is how to generate a synthetic string data set with specific $gpkl$=l and n. First, we generate a random dictionary to contain 10000 random strings that are 2B–6B long. The strings are used as prefixes. Second, we generate a set $\mathcal{L}$ of n random strings, sort the strings, and compute the initial $gpkl_0$ for $\mathcal{L}$, which is typically small for randomly generated strings. Third, we increase the $gpkl$ of $\mathcal{L}$ as follows. We randomly select k adjacent strings $S_{a1}, S_{a2}, ..., S_{ak}$ in the sorted list, and compute the common prefix length cpl for the k strings. Then, we randomly pick a string S_p from the dictionary, generate a random insert position $j \in [0, cpl]$, and insert S_p into each S_{ai} at the j-th byte. In this way, the $gpkl$ of $\mathcal{L}$ increases by at most $\frac{k \cdot len(S_p)}{|\mathcal{L}|}$. We adjust the location of the k strings to keep the sort order of $\mathcal{L}$. Finally, we repeat the third step until the $gpkl$ reaches the target l.

Structure Selection Scenarios. LITS performs PMSS-based decisions in three main scenarios: 1) Bulkload: if a node corresponds to over 16 kv-entries, LITS uses PMSS to decide whether to build a subtrie or a model-based node in the bulkload operation; 2) Insert into a full compact node: When an insert sees a full compact node (with 16 keys), it replaces the compact node with either a model-based node or a subtrie; 3) Resize a model-based node: Like LIPP, LITS performs node resizing if there are too many or too few keys in a model-based node N_r. The resizing process rebuilds the subtree rooted at node N_r, and uses the PMSS to decide whether a model-based node or a subtrie should be constructed.

Moreover, LITS detects the case where over 50% of the keys are mapped to an index slot in a model-based node. In such a case, LITS builds a subtrie for the child node corresponding to the index slot. This restriction ensures that the non-subtrie part of the tree is at most $O(logN)$ high. Note that the 50% restriction is actually quite weak, and it has not been triggered in our experiments.

Implementation Consideration. It should be noted that careful design is required at the connection point of different structures to avoid potentially unnecessary cost. For example, the root node of HOT contains a single pointer to the actual first level node. Therefore, when creating a HOT subtrie, we do not simply set the child pointer in the item to point to the root node of HOT. Instead, we directly replace that item with the root node of HOT (while also handling the flag bits of both LIT and HOT correctly). In this way, we save a pointer dereference for accessing the HOT subtrie.

3.5 Cost Analysis

The time and space cost of tries (e.g., ART [19] and HOT [10]) have been extensively measured and studied. In the following, we mainly summarize the time and space cost of the non-subtrie part of LITS. (Please see the full description in the extended version of the paper [29].) Suppose there are N keys in the index. Then, in the worst case, (1) the height of the non-subtrie part of LITS is $O(logN)$, (2) the search/update cost is $O(logN)$, (3) the amortized insert/delete cost is $O(log^2 N)$, and (4) the space cost is $O(NlogN)$.

4 EVALUATION

In this section, we compare the performance of LITS with state-of-the-art string indexes, and study the performance benefits of our proposed techniques in LITS.

4.1 Experimental Setup

Machine Configuration. All experiments are conducted on a machine equipped with two 3.4GHz Intel Xeon Platinum 8380 CPUs (with 40 cores / 80 threads per CPU and 60MB L3 cache) and 256GB memory. The machine runs the standard Ubuntu 20.04 Linux. All programs are compiled with GCC 9.4.0 using the O3 optimization level. To avoid NUMA effects, we perform the experiments using a single CPU in the machine.

Solutions to Compare. We compare LITS with five state-of-the-art traditional and learned indexes for strings:

- *ART (Adaptive Radix Tree)* [19]: We find multiple ART implementations on github, and choose the one with the highest stars (https://github.com/armon/libart.git). The implementation does not support scans. Therefore, we add a scan procedure for ART.
- *HOT (Height Optimized Trie)* [10]: We obtain the code written by the HOT authors (https://github.com/speedskater/hot.git).
- *SIndex* [26]: We use the implementation provided by the authors (https://github.com/curtis-sun/TLI.git). SIndex requires all strings to be padded to a uniform length. Therefore, for SIndex experiments, we pad the strings in each data set to the length of the longest string in the data set.
- *RSS (Radix String Spline)* [22]: For RSS, we cannot find publicly available code and write our own implementation in C++. We follow the RSS paper to employ the two-gram compression of HOPE [30] to encode string keys. This improves RSS's search performance. The reported index performance includes both the encoding of the query key and the actual index operation in RSS. However, RSS does not support insertions because it stores sorted key-value data in an array, and uses the array indexes to indicate the key ranges in tree nodes. As a result, we omit RSS for all experiments that perform insertions.
- *SLIPP*: We obtain the LIPP [28] code provided the LIPP authors (https://github.com/Jiacheng-WU/lipp). Then, we modify LIPP to support strings as described in Section 2.2. We implement the bulkload and the search operations. We find that SLIPP has much worse search performance than HOT, ART, and RSS. Since SLIPP is clearly less competitive, we choose not to implement the other operations for SLIPP and omit SLIPP for the rest of the experiments. The implementation is written in C++.
- *LITS*: We implement LITS in C++. The size of HPT is 2MB (with 1024 rows, 128 columns, and 16B per cell). A compact leaf node has a maximum capacity of 16 elements. The index used for constructing a hybrid structure with LIT is HOT, because HOT demonstrates better overall performance compared to ART.
- *Variants of LITS*: To understand the benefit of our proposed techniques, we also implement several variants of LITS. LIT is the learned index without subtries. Moreover, we change the learned model in LIT and implement several LIT(model) variants, as will be described in Section 4.3. Furthermore, we study the combination of LIT with different trie indexes. LITS-A is LIT enhanced with ART as the subtrie of choice. (LITS-H is LIT enhanced with HOT, which is another name for LITS.)

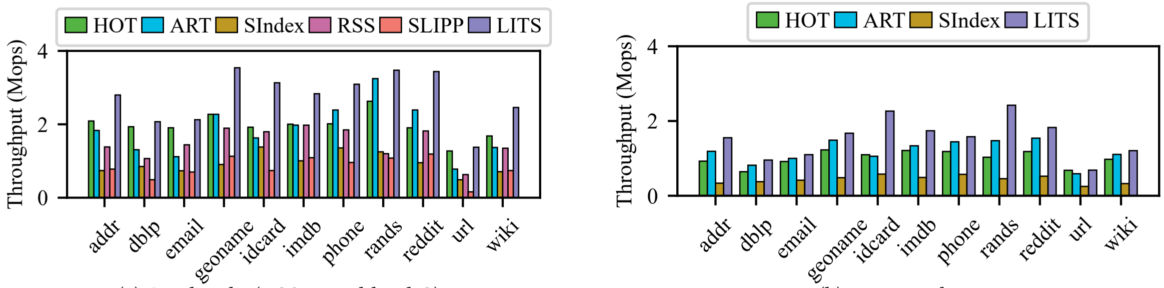
All experiments are conducted using a single thread except for the scalability experiments in Section 4.2. For the scalability experiments, we compare LITS with the most competitive solution, HOT. The HOT code supports multiple threads. We implement optimistic locking for LITS to support concurrent threads.

Datasets. We use seven real-world string data sets in our experiments, as listed in Table 1. (1) `address` contains 34M addresses in the form of unit-street-city in the US West [3]. (2) `dblp` contains 7M paper titles in dblp [4]. (3) `geoname` contains 7M geographical names, such as "Pic des Langounelles" [2]. (4) `imdb` contains 9M actor names in imdb [5]. (5) `reddit` contains the user names of 26M reddit accounts that have commented since Dec 2017 [1]. (6) `url` contains 63M urls from the CommonCrawl [6]. An example is "http://1000rosanegra.com.ar/index.html". (7) `wiki` contains 43M wiki titles [7]. An example is "1980-81_Mersin_Idmanyurdu_season".

Moreover, we generate four synthetic data sets. (8) `email` contains 45M synthetic email addresses generated by the Faker 14.2.1 package using Python 3.6.9. (9) `idcard` contains 63M synthetic Chinese id-card numbers. A id-card number is a 18-byte string. The first 6B represents a region, such as a city or a county. The next 8B is the birthday in the form of "yyyymmdd". Then the remaining 4B assigns a unique code to distinguish ids with the same 14B prefix. (10) `phone` contains 50M synthesis phone numbers generated by the Faker package. (11) `rands` contains 50M randomly generated strings. The characters are selected uniformly from `a` to `z`.

For the experiments, all data sets have been processed to remove duplicate strings, strings containing non-ASCII characters, and strings longer than 255 characters. Then, the value for each string key is a randomly generated 64-bit integer.

Workload. We use six YCSB core workloads: A (50% read, 50% update), B (95% read, 5% update), C (100% read), D (95% latest-read, 5% insert), E (95% short range scan, 5% insert), and F (50% read, 50% read-modify-write) [12]. For all YCSB workloads except the read-only workload C, we bulkload the indexes with 80% of the keys in a data set. For the read-only workload, we bulkload 100%



(a) Read-only (YCSB workload C)

(b) Insert-only

Figure 8: Index performance for read-only and insert-only workloads.

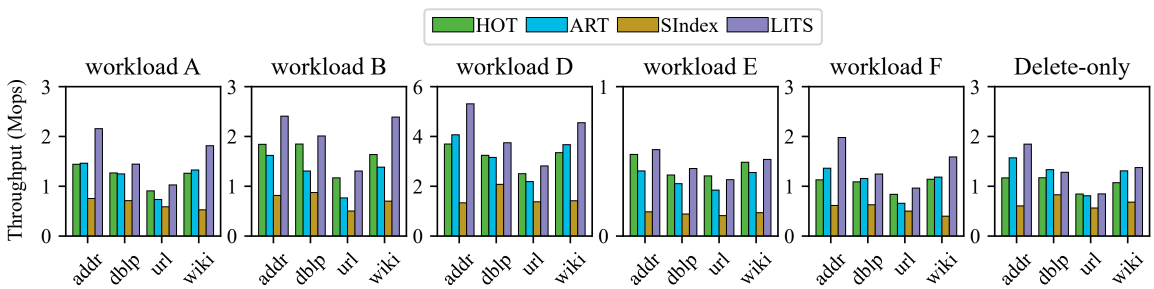


Figure 9: Index performance for YCSB workloads.

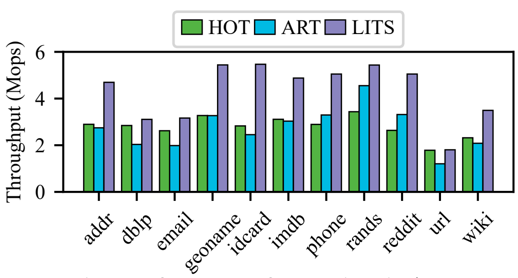


Figure 10: Index performance for read-only (YCSB workload C) with Zipf distribution.

of the keys. Then, we perform 20M random operations. Search keys are randomly selected from the bulkload keys. Insert keys are randomly selected from the 20% keys that are new. Update keys are randomly selected from the entire data set. For an existing key, the entry is modified. For a new key, we will perform an insert operation. Unless otherwise noted, the random keys are chosen uniformly at random. We also perform a set of experiments where the chosen keys follow the zipf distribution with zipf factor = 1.

Apart from the YCSB workloads, we test insert-only and delete-only workloads. For the insert-only workload, we bulkload the indexes with 50% keys in a data set, then measure the performance of randomly inserting all the remaining keys. For the delete-only workload, we bulkload the indexes with 100% keys, then measure the performance of randomly deleting 50% existing keys.

4.2 Overall Performance

Figure 8 and 9 show the overall index performance. We report the results of all 11 data sets for the read-only (YCSB workload C) and insert-only workloads in Figure 8. Due to space limitations, we report the results of the four largest real-world data sets, i.e., `address`, `dblp`, `url`, `wiki`, for the YCSB workload A, B, D, E, F, and the delete-only workload in Figure 9. Experiments on the other data sets show similar trends.

Figure 10 shows the performance of the read-only workload under the zipf distribution with zipf factor = 1. Due to space limitations, we report more experimental results under the zipf distribution in the extended version of the paper [29].

From the figures, we see that LITS achieves the best performance for most workloads and data sets. (LITS is slightly slower than HOT for workload E on `url`). For the read-only workload, LITS achieves up to 1.93x and 2.23x improvement over HOT and ART, respectively. Compared to SIndex, LITS demonstrates a performance advantage of 2.26x-3.91x. LITS also exhibits excellent performance for insert operations. Compared to HOT, ART, and SIndex, LITS attains up to 2.06x, 2.14x, and 5.31x improvement for the insert-only workload, respectively. Similarly, LITS achieves up to 2.43x, 2.27x, and 3.99x improvement over HOT, ART, and SIndex for workload A, B, D, and F. For the scan-heavy workload E, LITS's performance is comparable with HOT, and better than ART and SIndex. Finally, the zipf results show similar trends. Interestingly, under the zipf distribution, nodes that contain popular keys tend to stay in the CPU cache, leading to higher index performance than that with the uniform distribution.

Index Height. Table 3 compares the height of different indexes after bulkloading. The height of LITS is composed of two parts: LITS (base), which is the height of the LIT structure including model-based nodes and compact leaf nodes, and LITS (hot), which is the height of the subtries. From Table 3, we see that the height of LITS is significantly smaller than HOT, ART, and SLIPP. This partially explains the good performance of LITS. Note that RSS achieves good tree heights. However, RSS suffers from expensive local search, and for popular duplicate key prefixes, it has to visit and compare the string keys.

Bulkload Time. The left figure in Figure 11 compares the bulkload time of LITS, HOT, and ART. We bulkload all the keys in each data set. For HOT and ART, we sort the keys, then insert all the strings into the index in the sorted order. From the figure, we see that the

Table 3: Comparing the height of index solutions.

data set	LITS (base)	LITS (hot)	HOT	ART	SIndex	RSS	SLIPP
addr	2.7	0.2	7.0	10.2	2	2.0	4.9
dblp	2.7	1.7	6.8	14.1	2	2.2	7.3
url	3.0	2.1	7.8	16.1	2	3.7	9.1
wiki	2.9	1.0	7.8	11.6	2	2.1	5.7

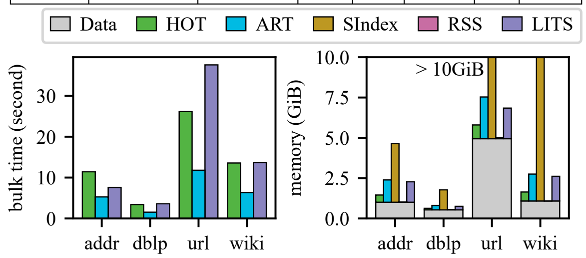


Figure 11: Bulkload time and memory space consumption

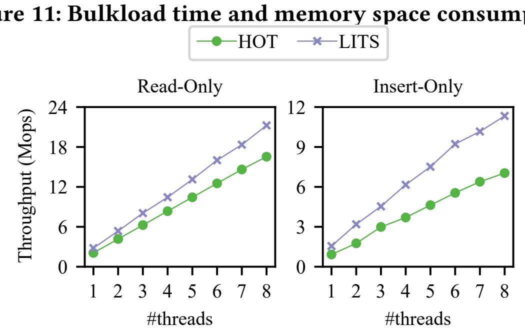


Figure 12: Scalability on the 34M address dataset.

bulkload time of LITS is comparable to that of HOT.

Space Cost. The right figure in Figure 11 compares the space cost of LITS, HOT, ART, Sindex, and RSS. The figure does not display the space cost of SLIPP; it exceeds 10GB in all four datasets. The grey bar shows the raw data size. From the figure, we see that LITS consumes lower space than ART and SIndex. SIndex consumes a lot of space for padding all strings to the maximal length in a data set. Interestingly, the read-only RSS has the lowest space cost, which is consistent with the RSS paper [22]. (Please see the in-depth analysis of the space consumption of learned indexes in the extended version of the paper [29].)

Scalability. We compare LITS and HOT in scalability experiments. For each data set, we bulkload the indexes with 50% keys in the data set. Then, the insert-only workload measures the performance of randomly inserting the remaining 50% keys. After that, the read-only workload measures the performance of 10M search operations for keys randomly distributed in the data set.

Figure 12 reports the index throughput for LITS and HOT varying the number of threads for `address` data sets. From the figure, we see that both LITS and HOT achieve nearly linear scalability. Compared to HOT, LITS achieves 1.19x – 1.31x and 1.52x – 1.64x improvement for the read-only and insert-only workloads, respectively.

4.3 Benefit of HPT

To understand the benefit of the HPT-based model in LITS, we compare HPT with existing learned models for strings:

- *Simple Model (SM)*: The simple method to calculate the CDF of a string is to use the equation $x = \frac{c_1}{256} + \ldots \frac{c_n}{256^n}$ to get a monotonic value based on the characters in the string. SM is used in SLIPP.

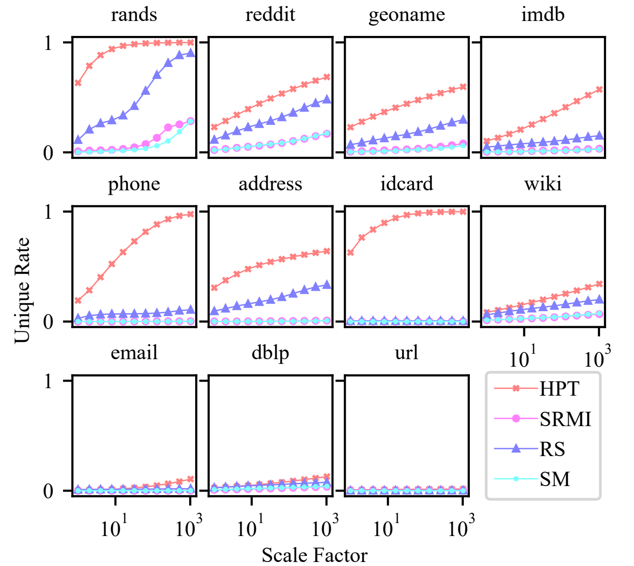


Figure 13: Unique rate of learned models.

- *Radix Spline (RS)*: RS is the default CDF model used in Radix String Spline (RSS) [22]. In each inner node of RSS, a K-byte substring of the key string is converted into an integer, and a RS model is used to compute the CDF. We use the same configuration as the RSS paper [22]. In the experiments, K is set to 8 and the error-bound in Radix Spline is set to 127.
- *SRMI*: SRMI is a string CDF model mentioned in the learned sort paper [18]. SRMI first converts a string into a floating point number using $x = \frac{c_1}{256} + \ldots + \frac{c_n}{256^n}$, then employs a two-layer RMI to compute the CDF from the coded floating point x.

Effectiveness for Distinguishing Strings. We would like to compare the effectiveness of the learned models for distinguishing strings in the data sets. For this purpose, we define and measure a unique rate metric.

We use a learned model to map a set $\mathcal{S}$ of unique strings to an item array of size $SF \cdot |\mathcal{S}|$, where $SF \geq 1$ is the scale factor. In the ideal situation, a perfect learned model will map every string in $\mathcal{S}$ to a separate location in the array. However, in the common case, there can be collisions. That is, two or more strings are mapped to the same item. The total number of occupied item slots after mapping, denoted as $NumValidSlots$, is always less than or equal to $|\mathcal{S}|$. We define UR_{SF} for scale factor SF as follows:

$$UR_{SF} = \frac{NumValidSlots}{|\mathcal{S}|} \tag{6}$$

UR_{SF} is between 0 and 1. The larger the UR_{SF}, the more effective that the learned model distinguishes keys in the string data set.

Figure 13 shows the unique rates of the four learned models varying the scale factor from 1 to 1000 for all the 11 data sets. We see that HPT achieves the best unique rate for all data sets and under all the scale factors. Compared to SM, RS, and SRMI, HPT is more powerful in distinguishing strings. For the three data sets with the highest GPKL, i.e., `email`, `dblp`, and `url`, all the learned models work quite poorly. These data sets require larger number of bytes to discern one string from the adjacent string in the sort order, making it hard for the learned models to separate the strings.

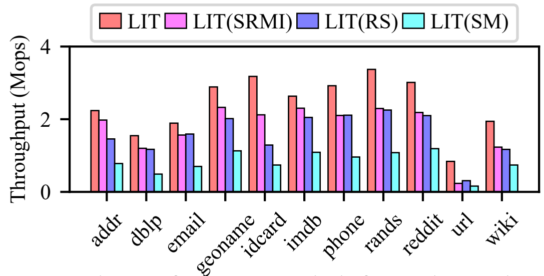


Figure 14: Index performance with different learned models.

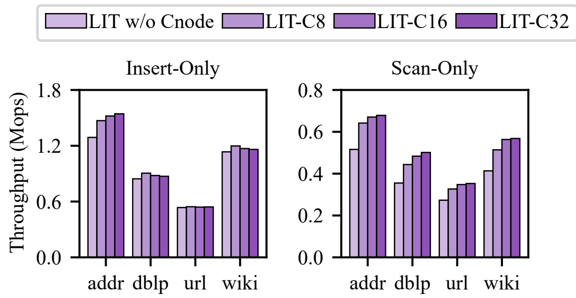


Figure 15: Comparing LIT with compact node designs.

Index Performance with Different Learned Models. Figure 14 compares the performance of LIT with different learned models. We choose to compare LIT instead of LITS because the hybrid structure of LITS could mask the performance impact of the learned model. The experiments run the read-only workload (YCSB workload C). LIT with the HPT model achieves the best index performance. It is 1.14x – 3.65x better than the second best, i.e., LIT(SRMI).

HPT Space and Time Cost. The HPT is lightweight. In our experiments, it takes 2MB, which is orders of magnitude smaller than the data set. HPT can easily fit into the CPU cache, and the model prediction using HPT is fast. It takes 20–50ns to compute the HPT-based model for an 8-byte substring.

4.4 Benefit of Compact Leaf Node

We study the benefit of the compact node in this subsection.

Performance Benefit. For the same reason as in Section 4.3, we conduct experiments using LIT rather than LITS. We compare LIT without compact nodes, and LIT with compact nodes whose size limit is set to 8, 16, and 32. Figure 15 reports the insert-only and scan-only throughput for the four LIT variants. From the figure, we see that the introduction of compact leaf nodes not only improves the performance of the scan operations but also enhances the performance of the insert operations. Scan is improved due to the fact that compact nodes place kv-pointers contiguously, thereby reducing cache misses for visiting many small nodes during the scan process. Insertion is improved because compact nodes tend to reduce the tree height and avoid extra cache misses caused by visiting small nodes in deeper levels of the index.

Moreover, we see that the scan throughput increases as the size limit increases, but becomes relatively flat beyond 16. The insert throughput may even suffer when the size limit exceeds 16. Therefore, we set the default size limit of compact nodes to 16 in all the other experiments.

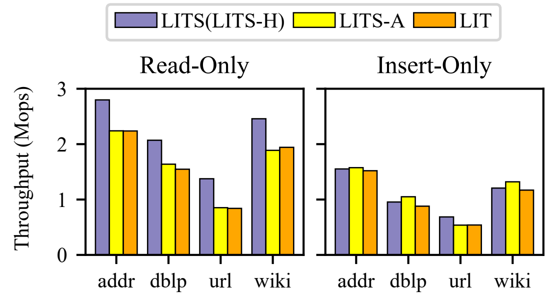


Figure 16: Comparing LITS-H, LITS-A, and LIT.

Variants of Compact Node Implementation. We consider two variants: 1) pre-allocating 16 entries in compact nodes; 2) exploiting SIMD for cnode search. However, our experimental results show that preallocation incurs up to 93% extra space overhead without significant performance benefits. The performance improvement with SIMD is less significant since cnode search is only a small part of the search procedure. Please see the extended version of the paper [29] for more details.

4.5 Benefit of LIT Enhanced with Subtries

The LITS mentioned in the above subsections are all LITS-H (i.e., the hybrid structure of LIT and HOT). We prefer this combination because the performance characteristics of LIT and HOT are complementary as shown in Figure 7. HOT performs well for data sets with large GPKLs and relatively small data sizes (e.g. `dblp`). In comparison, LIT demonstrates better performance for data sets with small GPKLs and large data sizes (e.g., `reddit`).

In this subsection, we consider the alternative design of combining LIT and ART, i.e., LITS-A, and LIT without subtries. Figure 16 compares the read-only and the insert-only throughput of LITS-H (i.e., LITS), LITS-A, and LIT. From the figure, we see that LITS-H achieves better performance than LIT, confirming that the hybrid structure indeed improves index performance. For data sets with large GPKLs (e.g., `url`), LITS-H brings up to 50% improvement for search performance. Moreover, compared to LITS-A, LITS-H has higher performance improvement for search. For the insert-only workload, LITS-A and LITS-H show comparable performance.

GPKL Computation Cost. In the insert-only workload, a single invocation of the GPKL computation takes 0.8–1.7us. The total GPKL computation time contributes to 0.5%–1.3% of the total insert time across all data sets.

5 CONCLUSION

In conclusion, we have presented a novel string index called LITS (Learned Index with Hash-enhanced Prefix Table and Sub-tries). Our experimental results show that compared to HOT and ART, LITS achieves up to 2.43x and 2.27x improvement for point operations and comparable scan performance.

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] 2017. reddit. https://www.kaggle.com/datasets/colinmorris/reddit-usernames
[2] 2018. geoname. https://www.kaggle.com/datasets/geonames/geonames-database
[3] 2023. address. https://v2.openaddresses.io/batch-prod/collection-us-west.zip
[4] 2023. dblp. https://dblp.org/xml/dblp.xml.gz
[5] 2023. imdb. https://datasets.imdbws.com/name.basics.tsv.gz
[6] 2023. url. https://data.commoncrawl.org/CC-MAIN-2023-14/
[7] 2023. wiki. https://dumps.wikimedia.org/enwiki/
[8] Domenico Amato, Giosuè Lo Bosco, and Raffaele Giancarlo. 2023. Neural networks as building blocks for the design of efficient learned indexes. *Neural Comput. Appl.* 35, 29 (2023), 21399–21414.
[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012.* ACM, 53–64.
[10] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* 521–534.
[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020.* USENIX Association, 209–223.
[12] 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.* 143–154.
[13] 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.* 969–984.
[14] Paolo Ferragina, Marco Frasca, Giosuè Cataldo Marinò, and Giorgio Vinciguerra. 2023. On Nonlinear Learned String Indexing. *IEEE Access* 11 (2023), 74021–74034.
[15] Ali Hadian and Thomas Heinis. 2020. MADEX: Learning-augmented Algorithmic Index Structures. In *AIDB@VLDB 2020, 2nd International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2020, Monday, August 31, 2020, Online Event / Tokyo, Japan.*
[16] 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.* 5:1–5:5.
[17] 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.* 489–504.
[18] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* 1001–1016.
[19] 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.
[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, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proc. VLDB Endow.* 16, 9 (2023), 2212–2224.
[22] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. 2021. Bounding the Last Mile: Efficient Learned String Indexing. *CoRR* abs/2111.14905 (2021). arXiv:2111.14905
[23] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. PLEX: Towards Practical Learned Indexing. *CoRR* abs/2108.05117 (2021). arXiv:2108.05117
[24] 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.* 308–320.
[25] Ke Wang, Guanqun Yang, Yiwei Li, Huanchen Zhang, and Mingyu Gao. 2023. When Tree Meets Hash: Reducing Random Reads for Index Structures on Persistent Memories. *Proc. ACM Manag. Data* 1, 1 (2023), 105:1–105:26.
[26] 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*, Taesoo Kim and Patrick P. C. Lee (Eds.). 17–24.
[27] 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.
[28] 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.
[29] Yifan Yang and Shimin Chen. 2024. LITS: An Optimized Learned Index for Strings (An Extended Version). In *Technical Report.* arXiv:2407.11556
[30] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* 1601–1615.