LIVAK: A High-Performance In-Memory Learned Index for Variable-Length Keys

Zhaole Chu[†], Zhou Zhang[†], Peiquan Jin[*†], Xiaoliang Wang[†], Yongping Luo[†], Xujian Zhao[‡]
[†]University of Science and Technology of China, Hefei, China
[‡]Southwest University of Science and Technology, Mianyang, China
{czle,zzwolf,wxl147,ypluo}@mail.ustc.edu.cn,[*]jpq@ustc.edu.cn,jasonzhaoxj@swust.edu.cn

ABSTRACT

In-memory learned index has been an efficient approach supporting in-memory fast data access. However, existing learned indexes are inefficient in supporting variable-length keys. To address this issue, we propose a new in-memory learned index called *LIVAK* that adopts a hybrid structure involving trie, learned index, and B+-tree. Each node indexes an 8-byte slice of keys, and we use learned indexes for large nodes but B+-trees for small nodes. Also, LIVAK presents a character re-encoding mechanism to avoid performance degradation. We compare LIVAK with B+-tree, Masstree, and SIndex on various datasets and workloads, and the results suggest the efficiency of LIVAK.

CCS CONCEPTS

• **Information systems → Data structures**.

KEYWORDS

In-Memory index, Learned index, Variable-length key

ACM Reference Format:
Zhaole Chu[†], Zhou Zhang[†], Peiquan Jin[*†], Xiaoliang Wang[†], Yongping Luo[†], Xujian Zhao[‡]. 2024. LIVAK: A High-Performance In-Memory Learned Index for Variable-Length Keys. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657385

1 INTRODUCTION

Learned indexes [5, 12] aim to build an in-memory machine-learning model to find the memory address of the requested key quickly, which can support in-memory data processing efficiently. In general, a learned index has much larger nodes than B+-tree, yielding a much flatter structure [3]. Each node of a learned index can store thousands to hundreds of data elements, yielding a low index height. In addition, through model inference, a learned index can quickly narrow down the search to tens of elements.

However, existing learned indexes are inefficient in handling variable-length string keys [13]. The main reason is that existing model-fitting algorithms can only support numeric keys and need to locate the precise address of each element in the dataset. This drawback limits the application scenarios of learned indexes because variable-length keys exist in many real applications. In addition, long string keys will increase the comparison overhead and reduce the caching efficiency of learned indexes. Therefore, making in-memory learned indexes support variable-length string keys has been a key challenge.

At present, SIndex [13] is the only work that can support variable-length keys. It reduces the key-comparison overhead during query processes by partitioning keys based on common prefixes and using partial keys [2] for comparison. However, SIndex slices a key into at most three parts, namely a prefix for partitioning, an infix with uniqueness, and an unused suffix, which causes its performance to degrade when the key length grows. Specifically, when the key length exceeds 128 bytes, the performance of SIndex is inferior to that of the trie variants [13].

This paper proposes a new high-performance in-memory learned index called **LIVAK** (**L**eaned **I**ndex for **VA**riable-length **K**eys). LIVAK adopts a trie-like tree structure, and each of its nodes is responsible for indexing 8-byte slices of keys. Further, LIVAK constructs each trie node and uses two types of indexes, i.e., learned indexes and B+-trees. Also, LIVAK supports bulk data loading and uses path compression techniques to handle longer keys. To avoid the performance degradation caused by distribution discontinuities of keys, we propose a new character re-encoding mechanism, which can improve the performance of LIVAK further.

Briefly, we make the following contributions in this paper:

(1) We present a trie-like hybrid index structure in LIVAK, which uses a mixture of learned indexes and B+-trees as trie nodes and supports path compression. Particularly, we use learned indexes for nodes with a large data size and B+-trees for those with a small data size. We demonstrate that such a hybrid node structure can support variable-length string keys and achieve high performance.

(2) To address the performance degradation caused by discontinuous key distributions when dealing with string keys, we propose an efficient character re-encoding technique to further improve the performance of LIVAK.

(3) We compare LIVAK with B+-tree, Masstree (an in-memory index used in the industry), and SIndex (the state-of-the-art learned index supporting variable-length keys). The results on three datasets and four workloads show that LIVAK outperforms all competitors' read and write performance.

2 RELATED WORK

Kraska et al. proposed the first learned index called RMI (Recursive Model Index) [5] in 2018. Although RMI does not support insertions, this problem has been addressed by subsequent work [3, 4, 16,

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3657385

17]. Currently, for numeric keys, updatable learned indexes, e.g., ALEX [3], outperform traditional indexes in terms of read and write performance and space efficiency. However, another shortcoming of current learned indexes is that they can only support fixed-length numeric keys. Since variable-length string keys are common in real production environments, this shortcoming severely limits the application scenarios and practicality of learned indexes.

So far, SIndex [13] is the only learned index that supports variable-length string keys. It slightly modifies the fitting algorithm and index structure and uses a greedy algorithm to group datasets by common prefixes. Then, SIndex uses partial keys in each group to construct learned nodes, solving the problem of long common prefixes for the partial keys approach. However, since SIndex only slices a key into three parts, it has limited performance when dealing with long keys. RSS [11] uses fixed-length slicing of strings to train the model and moves keys with many common prefixes to the child nodes to ensure that the keys in each node are within a given maximum error. However, it does not support insertions. In summary, it is still challenging for learned indexes to support variable-length string keys.

Existing in-memory indexes can support variable-length string keys, and most of them are variants of tries, such as Masstree [7], ART [6], HOT [1], Hyperion [9], Wormhole [14], HydraList [10], and Cuckoo Trie [15]. Among them, Masstree [7] has been used in many industrial products. Masstree slices keys according to an 8-byte span, creates B+-trees for the slices, and concatenates B+-trees in a trie-like form. Although this paper aims to optimize learned indexes, in the experiment, we will compare our design with Masstree, which is used as the representative of existing in-memory string indexes.

3 DESIGN OF LIVAK

3.1 Motivation

Inspired by Masstree [7], we propose a hybrid index structure that combines learned indexes and tries. The hybrid index structure can enjoy the advantages of both index structures, i.e., the flat structure and the ability to perceive data distribution of the learned index, and the natural advantage and path compression ability of the trie in handling variable-length string keys. Our initial idea is to use learned indexes to replace all the B+-trees in Masstree.

However, building a trie structure concatenated by learned indexes is not trivial, and the following problems need to be solved:

(1) Learned indexes are suitable for large datasets. However, in the trie-like hybrid index structure, there are many trie nodes with only a small number of keys. For such sparse nodes, learned indexes do not perform as well as B+-trees due to the additional training costs, model computation costs, and more metadata.

(2) Learned indexes usually do not support inserting data starting from an empty dataset. Tries and their variants usually do not support bulk loading and only allow inserting keys one by one. Thus, the bulk loading algorithm must be redesigned to build the hybrid index structure.

(3) Tries need to combine path compression techniques to improve the efficiency of handling long keys. Previous work SIndex [13] and RSS [11] only support suffix path compression. The trie-like hybrid index structure requires a new algorithm to support infix and suffix path compression.

(4) The presence of special characters and invalid values in the character encoding leads to discontinuous key distributions. However, learned indexes usually use piecewise linear approximation algorithms to train the models. Such algorithms use continuous line segments to fit the CDF of the keys, so the discontinuous key distribution leads to an increase in the model's error, which results in a degradation of the index performance.

To solve the above problems, we improve the hybrid index structure and propose LIVAK, which takes advantage of learned indexes and B+ Trees simultaneously. The basic idea of LIVAK is to use learned indexes as trie nodes with large data sizes and B+ trees as trie nodes with small data sizes.

3.2 Trie-like Hybrid Index Structure

The overall structure of LIVAK is a trie, and each node is responsible for indexing an 8-byte slice of the keys. As shown in Fig. 1, LIVAK constructs an index for each trie node and uses multiple types of index structures. In brief, LIVAK is a trie-like concatenation with mixed types of indexes. The structure of LIVAK is generic and can integrate any state-of-the-art index structure and enjoy the advantages of each index.

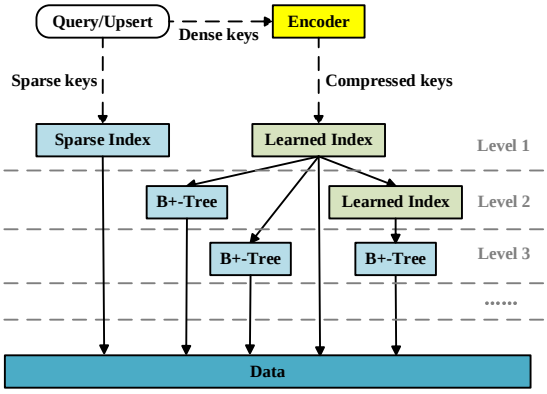


Figure 1: The index structure of LIVAK.

LIVAK uses path compression techniques, including infix path compression and suffix path compression. When there is only one key slice in a trie node, instead of constructing this node, LIVAK uses the unique child node pointer or data pointer it holds to replace its pointer in the parent node. For example, in Fig. 1, the parent node of B+-tree on the left side of the third level is discarded due to path compression, so the root node holds its pointer directly. This is a case of infix path compression. In addition, the B+-tree on the left side of the third level may hold a pointer to a key of length 40 bytes. Since no key in the dataset is the same as its first 24 bytes, LIVAK does not store its trailing 16 bytes but stores its data pointer in the third level. This is a case of suffix path compression, which is very common in LIVAK. Since LIVAK uses multiple types of indexes as trie nodes, it is more complicated to implement path compression in LIVAK than in a conventional trie.

LIVAK vs. B+-tree and Learned Indexes. Compared with B+-trees and learned indexes, LIVAK has the following advantages:

(1) For the common prefixes in the dataset, LIVAK only needs to store them once, while B+-trees and learned indexes need to store them repeatedly. So LIVAK has an advantage in space efficiency.

(2) When the keys are long, B+-trees and learned indexes lose their cache efficiency because they store the complete keys in the nodes. Since LIVAK only stores 8-byte slices of keys in the nodes, it does not lose cache efficiency.

(3) During a local search within the nodes of B+-trees and learned indexes, each comparison operation requires comparing the complete key. When the keys are long, the cost of the comparison operation rises accordingly. In contrast, in LIVAK, each comparison operation involves only 8-byte key slices. Also, LIVAK uses path compression techniques to skip some unnecessary comparisons.

LIVAK vs. Trie-like Indexes. LIVAK has the following advantages over the traditional trie and its variants:

(1) Traditional trie constructs one node for every byte and reserves pointer space for all possible values, leading to low cache and space efficiency. LIVAK constructs one node for every 8 bytes and uses B+-trees or learned indexes as trie nodes, reserving pointer space only for the existing values, which can achieve higher cache and space efficiency.

(2) Compared with Masstree [7], LIVAK uses learned indexes to handle trie nodes with large data sizes to utilize the advantages of learned indexes. Therefore, LIVAK expects higher performance and higher space efficiency than Masstree.

3.3 Node Structure

Each LIVAK node stores 8-byte key slices. To enable key slices to be computed in the model, LIVAK converts them to unsigned long integer types (uint64_t). The node structure of LIVAK is defined as a C++ Struct *livak_node*. Here, *type* indicates the node type. *middle_path_compression* indicates whether the node's parent node is discarded due to path compression. *prefix* is used to store the node's prefix. If a key exactly equals the node's prefix, the data pointer corresponding to that key is stored in *data_ptr*. *index_ptr* stores an index pointer to the index instance corresponding to the node. LIVAK dynamically interprets the type of the index pointer according to *type*. The index instance stores all the key slices in the corresponding trie node.

```
Struct livak_node {
    bool type;
    bool middle_path_compression;
    uint32_t depth;
    std::string prefix;
    void * data_ptr;
    index_type<uint64_t, void *> * index_ptr;}
```

3.4 Index Operations

In this section, we describe the operations of LIVAK.

Point Query. Algorithm 1 shows the point query operation of LIVAK. If the target key is found, the function *Find* returns a pointer to the memory address of the data. If the target key is not found, a null pointer is returned. The function *Get_payload* is responsible for searching inside the LIVAK node. The function *Get_type* resolves the free bits of the pointer and returns the type of the pointer, which has three possible return values: NODE means the pointer points to a LIVAK node, DATA means the pointer points to a data memory address without suffix path compression, and PATH_COMPRESSION means the pointer points to a data memory address with suffix path compression.

Algorithm 1: Point query

```
1  Function Find(key)
2      pointer ← Get_payload(root, key);
3      while pointer and Get_type(pointer) = NODE do
4          pointer ← Get_payload(*pointer, key);
5      end
6      if pointer then
7          if Get_type(pointer) = PATH_COMPRESSION and *pointer != key
             then
8              return NULL;
9          return pointer;
10     return NULL;
11 Function Get_payload(node, key)
12     if node.middle_path_compression and !Match(key, node.prefix) then
13         return NULL;
14     if key.length() = node.depth then
15         return node.deta_ptr;
16     else if node.type = LEARNED_INDEX then
17         return (learned_index*)node.index_ptr->
             Find(Get_keyslice(key, node.depth));
18     return (btree*)node.index_ptr-> Find(Get_keyslice(key, node.depth));
```

Range Query. Similar to tries, range queries of LIVAK are based on the backtracking method. First, the lower bound of the range is found using the point query algorithm. Then, the key slices are scanned sequentially in the LIVAK node. If the pointer corresponding to a key slice points to a node, the lower LIVAK node is scanned first. After finishing the scan of the lower node, return to the upper node and continue the previous scanning process. When a key equal to or greater than the upper bound of the range is found, it means that the range query process is completed.

Insertion and Update. LIVAK provides an upsert interface to support insert and update operations. This interface takes a key-value pair as input. If the target key exists in the index, its value is updated. Otherwise, an insertion will be performed. Since LIVAK uses both infix and suffix path compression in its structure, one upsert operation may lead to four cases, as shown in Algorithm 2.

Deletion. The deletion operation of LIVAK takes a key as input. The target key is first found in the index by a point query process, and then the key slice is deleted in the lowermost LIVAK node. After that, it is necessary to determine whether there is only one pointer left in that node. If yes, we perform path compression on that node. Note that there are at least two pointers in any LIVAK node.

Bulk Loading. LIVAK builds the index by a bulk-loading interface. First, a hash table is built, where the key of the hash table is a string and the value is an array. The array stores all 8-byte key slices prefixed with that string. Then, we slice all the keys in the dataset by 8 bytes and insert the slices into the hash table. We use an additional array to store the key slices prefixed by null as the slices in the root node. For the key slices that are less than 8 bytes, we use the null character as the suffix. The hash table can be considered as a trie. Its keys are the prefixes of trie nodes, and the values store the key slices. We traverse the trie by post-order traversal and transform each trie node into a LIVAK node.

Algorithm 2: Upsert

```
1  Function Upsert(key, value)
2      parent_node ← root;
3      current_node ← root;
4      pointer ← Get_payload(root, key);
5      while pointer and Get_type(pointer) = NODE do
6          current_node ← pointer;
7          pointer ← Get_payload(*pointer, key);
8          if pointer then
9              parent_node ← current_node;
10         end
11     end
12     if pointer then
13         if Get_type(pointer) = PATH_COMPRESSION and *pointer != key
             then
14             new_node ← Key_comparison(key, *pointer);
               /* Case 1: Create a new LIVAK node using the found key and
                  the target key                                          */
15             current_node.Update_node(new_node);
16             return;
17         end
18         current_node.Update(key, value) ;
           /* Case 2: The keys match, so an update is performed.        */
19         return;
20     end
21     if current_node.middle_path_compression and !Match(key,
         current_node.prefix) then
22         new_node ← Key_comparison(key, current_node.prefix) ;
           /* Case 3: Create a new LIVAK node using the node prefix and
              the target key                                             */
23         parent_node.Update_node(new_node);
24         return;
25     end
26     current_node.Insert(key, value) ;
       /* Case 4: The node prefix is matched, so the key is inserted.   */
27     return;
```

3.5 Character Re-Encoding

We observe that in real-world datasets, the range of values per character (i.e., radix) is much smaller than the express ability of a single character (i.e., 256). This is because common character encodings (e.g., ASCII) have only a few dozen displayable characters, with the rest being control characters, communication-specific characters, and extended characters. A smaller radix may lead to a decrease in the performance and space efficiency of a learned index because it will make the CDF of the dataset discontinuous and exhibit a step-shaped distribution, which is not well supported by piecewise linear models used in learned indexes [8].

LIVAK uses an optional character encoder to handle the problem of small character bases. When building the index, LIVAK samples the characters in the dataset and estimates the character radix and its distribution. LIVAK enables the character encoder when the majority of characters are distributed over less than or equal to 64 characters. In our implementation, this percentage is set to 99% by default. As shown in Fig. 1, the encoder is responsible for re-encoding the keys. LIVAK samples the 64 characters with the largest estimated character distribution, calling them *dense* characters and the rest *sparse* characters. For each key, LIVAK first determines whether it contains sparse characters when processing it. If it does, it is called a *sparse* key; if it does not, it is called a *dense* key. LIVAK re-encodes dense keys while keeping the original order of the keys. Specifically, LIVAK slices the key into 10-byte key slices and re-encodes the slices into an unsigned long integer, with each character occupying 6 bits. This process does not disrupt the order of the key slices. LIVAK uses a trie-like concatenation structure with mixed indexes to index the re-encoded dense keys, and an additional sparse index (B+-tree) to index sparse keys.

The character encoder of LIVAK focuses on encoding/decoding speed rather than compression ratio. Turning on the character encoder requires only a very low additional space cost. Specifically, we use a radix table of length 256 as the encoding table and a radix table of length 64 as the decoding table. By accessing the radix tables, LIVAK can encode and decode quickly.

4 PERFORMANCE EVALUATION

All experiments were conducted on a Linux server configured with an Intel Xeon Gold 6240 CPU 2.6 GHz and 256 GB DDR4. The Linux kernel version is 5.4.0.

Datasets. We use a synthetic dataset (*Random*) and two real-world datasets (*Titles* and *Urls*). The *Random* dataset is an algorithm-generated random-length string containing English upper and lower case letters, numbers, and common symbols. *Titles* is all page titles in Wikipedia. *Urls* contains over 100 million web addresses. Table 1 shows the characteristics of the three datasets.

Table 1: Characteristics of the datasets.

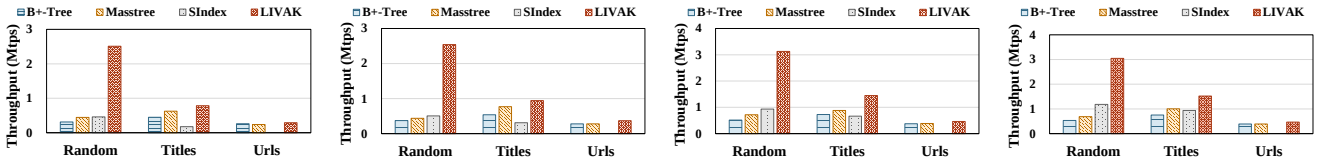
Dataset	Number of keys	Average key size	Maximum key size
Random	100M	34 bytes	64 bytes
Titles	41.4M	19 bytes	255 bytes
Urls	106M	104 bytes	2040 bytes

Competitors. We compare LIVAK with three representative in-memory indexes that support variable-length keys, including B+-tree, Masstree [7], and SIndex [13]. Each set of experiments is optimized using `-O3` and run by a single thread. Before each set of experiments, 100 million query operations are executed as a warm-up to ensure the CPU cache is used efficiently.

4.1 Read and Write Performance

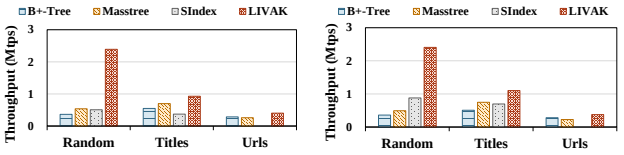
In this experiment, we test all indexes on four workloads with different read/write ratios, i.e., write-only, write-heavy (50% read and 50% write), read-heavy (95% read and 5% write), and read-only workloads. The read operations for all workloads follow the Zipfian distribution. The write operations all insert a new key to the index.

Figure 2 shows the experimental results for read and write performance. The results of SIndex on *Urls* are missing in Fig. 2 because SIndex can only support keys whose length is less than 256 bytes but the longest key length in the dataset is about 2KB. LIVAK achieves the best performance on all settings. Particularly, on the *Random* dataset, LIVAK achieves 5.47x, 5.04x, 3.36x, and 2.58x higher performance on the four workloads than the second place. On the *Titles* dataset, LIVAK outperforms the second place on four workloads by 25.46%, 22.48%, 64.00%, and 51.10%, respectively. On the *Urls* dataset, LIVAK outperforms the second place on four workloads by 10.83%, 32.02%, 20.38%, and 19.68%, respectively.



(a) On the write-only workload. (b) On the write-heavy workload. (c) On the read-heavy workload. (d) On the read-only workload.

Figure 2: Read and write performance.



(a) On the upsert workload. (b) On the read-modify-write workload.

Figure 3: Update performance.

4.2 Update Performance

In this subsection, we focus on the performance of the update operation. Two workloads are used in the experiments in this subsection. The first one is the upsert workload, which consists of 50% of insert operations and 50% of update operations. Each operation first looks up a key and then writes a new payload back to the index. Figure 3 shows the experimental results of the update performance, where LIVAK outperforms the other indexes for all datasets and workloads. In addition, we observe that Masstree outperforms SIndex on the upsert workload, and B+-tree consistently outperforms Masstree on *Urls*.

4.3 Impact of the Key Length

In this experiment, we vary the key length in the *Random* dataset to study the effect of key length on performance. As shown in Fig. 4, we set the length of the longest key to 32, 64, and 128, and the length of each key in the dataset is a random number between 1 and the longest key length. We observe that the read-and-write performance of all four indexes decreases as the key length grows. The performance degradation of LIVAK is the smallest among all the indexes for both write-only and write-heavy workloads, and the performance degradation of B+-tree and Masstree is severe for both workloads, with Masstree showing the highest degradation of 57.82%. The performance degradation of SIndex is small for read-only and write-only workloads but severe for mixed read and write workloads. All indexes show similar performance degradation on read-only and read-heavy workloads.

4.4 Impact of the Character Re-encoding

In this experiment, we manually turn on/off the character re-encoding mechanism of LIVAK and test read/write performance. The results are shown in Fig. 5. The first three datasets are synthetic datasets with the longest key length of 32, 64, and 128, and the last two datasets are real-world datasets. We can see that turning on the character re-encoding significantly improves the performance on all synthetic datasets but leads to performance degradation on the *Titles* dataset. The performance on the *Urls* dataset remains approximately the same, except for slight performance degradation on the read-only workload. This is because the *Random* dataset contains only letters, numbers, and common symbols, and the radix of characters is small; thus, the character re-encoding mechanism can effectively reduce the discontinuity of key distribution. However, the *Titles* dataset contains many extended characters with a large character radix, while the *Urls* dataset has a high percentage of special symbols. Using character re-encoding leads to a large amount of data being identified as sparse keys that are put into the sparse index, worsening the performance. To this end, the character re-encoding mechanism works better for datasets where most characters are distributed among the first few frequently appearing characters.

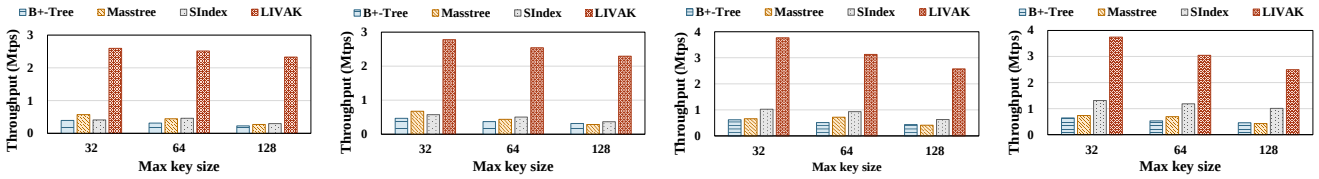
4.5 Bulk Loading

LIVAK provides a bulk loading interface to load data and build the index. Figure 6(a) shows the bulk-loading time for LIVAK and other indexes, where the *Random* dataset and the *Urls* dataset each bulk loads 50 million key-value pairs, and the *Titles* dataset bulk loads 20 million key-value pairs. Since Masstree does not provide a bulk-loading interface, we call its insert interface to insert all keys one by one. When building LIVAK, a trie needs to be built first, and then a post-order traversal is used to convert the trie nodes into learned indexes or B+-trees in turn. Therefore, the bulk loading cost of LIVAK is higher than Masstree. In addition, B+-tree has the fastest bulk loading speed because it does not require key slicing when building the index. SIndex has a fast bulk loading speed on the *Random* dataset but the slowest speed on the *Titles* dataset. This is because SIndex needs to specify the length of the longest key when building the index. When the length of the longest key is large, the bulk loading time of SIndex increases dramatically. After the index is built, SIndex does not support inserting keys longer than the preset longest key length.

Figure 6(b) shows the effect of manually turning on or off the character re-encoding mechanism on bulk loading time. Contrary to our intuition, the time spent on bulk loading decreases when the character re-encoding mechanism is turned on. The first reason is that the character re-encoding based on the radix table takes very little time. The second reason is that the character re-encoding mechanism shortens the length of the keys.

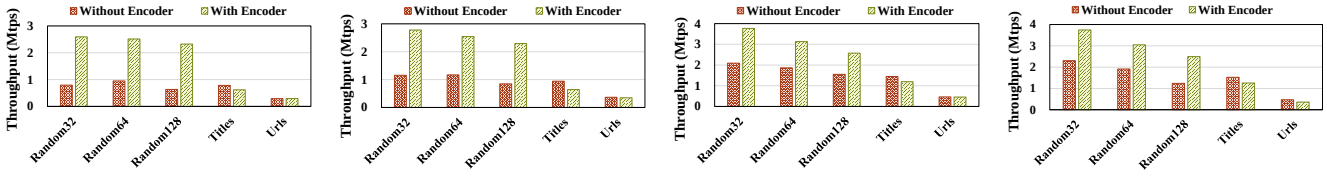
4.6 Memory Cost

In this experiment, we measure the memory cost of LIVAK, B+-tree, Masstree, and SIndex. We first use half of the data to construct the index and then insert all the keys in the dataset. Then, we calculate



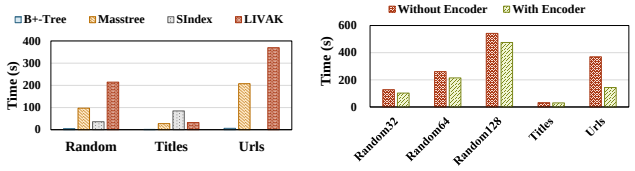
(a) On the write-only workload. (b) On the write-heavy workload. (c) On the read-heavy workload. (d) On the read-only workload.

Figure 4: Impact of the key length.



(a) On the write-only workload. (b) On the write-heavy workload. (c) On the read-heavy workload. (d) On the read-only workload.

Figure 5: Impact of the character re-encoding mechanism.



(a) Bulk-loading time of all indexes on the three datasets. (b) Impact of the character re-encoding on bulk-loading time.

Figure 6: Comparison of the bulk-loading time.

the memory usage subsequently. Table 2 provides an overview of the memory consumption of each index. Notably, LIVAK demonstrates the highest memory efficiency across all datasets. This superiority is attributed to the combination of the two memory-efficient structures: the trie index and the learned index.

Table 2: Comparison of memory cost (GB).

	B+-Tree	Masstree	SIndex	LIVAK
Random	15.43	113.37	15.92	**3.05**
Titles	4.68	10.68	20.42	**4.67**
Urls	26.81	98.53	N/A	**22.88**

5 CONCLUSIONS

In this paper, we proposed LIVAK to enable learned indexes to support variable-length keys efficiently. LIVAK adopts a hybrid structure, which takes advantage of the friendliness of tries to variable-length keys, the superiority of learned indexes in handling large-scale data, and the simplicity of B+-tree. We presented the overall structure of LIVAK and the internal design of nodes. In addition, we proposed a character re-encoding mechanism to address the problem that learned indexes suffered from discontinuous key distributions that might cause performance degradation. The experimental results on various datasets and workloads showed that LIVAK outperformed B+-tree, Masstree, and SIndex.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation of China (No. 62072419) and CCF-Huawei Populus Grove Challenge Fund. Peiquan Jin is the corresponding author.

REFERENCES

[1] Robert Binna, Eva Zangerle, and et al. 2018. HOT: A height optimized trie index for main-memory database systems. In *SIGMOD*. 521–534.
[2] Philip Bohannon, Peter McIlroy, and et al. 2001. Main-memory index structures with fixed-size partial keys. In *SIGMOD*. 163–174.
[3] Jialin Ding, Umar Farooq Minhas, and et al. 2020. ALEX: An updatable adaptive learned index. In *SIGMOD*. 969–984.
[4] 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.
[5] Tim Kraska, Alex Beutel, and et al. 2018. The case for learned index structures. In *SIGMOD*. 489–504.
[6] Viktor Leis, Alfons Kemper, and et al. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.
[7] Yandong Mao, Eddie Kohler, and et al. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.
[8] Ryan Marcus, Andreas Kipf, and et al. 2020. Benchmarking learned indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.
[9] Markus Mäsker, Tim Süß, and et al. 2019. Hyperion: Building the largest in-memory search tree. In *SIGMOD*. 1207–1222.
[10] Ajit Mathew and Changwoo Min. 2020. HydraList: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.* 13, 9 (2020), 1332–1345.
[11] Benjamin Spector, Andreas Kipf, and et al. 2021. Bounding the last mile: Efficient learned string indexing. In *AIDB@VLDB*.
[12] Zhaoyan Sun, Xuanhe Zhou, and et al. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 16, 8 (2023), 1992–2004.
[13] Youyun Wang, Chuzhe Tang, and et al. 2020. SIndex: A scalable learned index for string keys. In *APSys*. 17–24.
[14] Xingbo Wu, Fan Ni, and et al. 2019. Wormhole: A fast ordered index for in-memory data management. In *EuroSys*. 18:1–18:16.
[15] Adar Zeitak and Adam Morrison. 2021. Cuckoo trie: Exploiting memory-level parallelism for efficient DRAM indexing. In *SOSP*. 147–162.
[16] Zhou Zhang, Zhaole Chu, and et al. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proc. VLDB Endow.* 16, 2 (2022), 243–255.
[17] Zhou Zhang, Peiquan Jin, and et al. 2021. COLIN: A cache-conscious dynamic learned index with high read/write performance. *J. Comput. Sci. Technol.* 36, 4 (2021), 721–740.