SIndex: A Scalable Learned Index for String Keys

Youyun Wang, Chuzhe Tang, Zhaoguo Wang, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

ABSTRACT

The learned index structures have reshaped our perspectives on the design of traditional data structures. With machine learning (ML) techniques, they can achieve better lookup performance than existing indexes. However, current learned indexes primarily focus on integer-key workloads and failed to efficiently index variable-length string keys. We introduce SIndex, a concurrent learned index specialized in variable-length string key workloads. To reduce the cost of model inference and data accesses, SIndex groups keys with shared prefixes and use each key's unique part for model training. We evaluate SIndex with both real-world and synthesized datasets. The result shows that SIndex can achieve up to 91% better performance compared with other state-of-the-art index structures. We have open-sourced our implementation[1].

CCS CONCEPTS

• **Information systems → Data structures**.

ACM Reference Format:
Youyun Wang, Chuzhe Tang, Zhaoguo Wang, Haibo Chen. 2020. SIndex: A Scalable Learned Index for String Keys. In *11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), August 24–25, 2020, Tsukuba, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409963.3410496

1 INTRODUCTION

The recent work by Kraska et al. [12] has started an era of *learned index structures*, which use ML models to index data [7, 10, 15, 17, 18]. First, learned index structures ask models to learn the distribution of keys. Then, with trained models, they can predict the position of a given key. The predicted positions are close to actual positions, so that the requested data can be found efficiently. Results show that leveraging ML techniques can improve the lookup performance by up to 3× while reducing the memory consumption by more than 90%.

However, the efficiency of learned indexes comes with several drawbacks: high update latency, restricted key types and poor scalability. Successive works have targeted some of the limitations [7, 10, 15, 17, 18]. But none of them can efficiently support workloads with variable-length string keys which are common in the real world [1, 2, 6]. The major challenge of supporting such workloads is that the cost of both model inference and data accesses increases with the increasing key length. This is caused by inflated arithmetic computation and memory accesses. Furthermore, the model accuracy tends to decrease[2], which results in more data accesses during lookups. An intuitive solution is to use more sophisticated models such as deep neural networks to improve accuracy and hence reduce the amount of data accessed. Unfortunately, this method incurs significantly higher inference and training costs and therefore fails to improve the overall performance.

This paper introduces SIndex, a scalable learned index that can efficiently support string keys. Inspired by existing string key indexes [3–5, 13, 16], SIndex takes advantage of common prefixes to improve lookup latency. Specifically, SIndex greedily clusters keys with common prefixes into groups (Section 3.2). Then, in each group, SIndex uses the unique part of each key, the *partial key*, to train the model and index the data (Section 3.1). Experimental results show that SIndex has up to 91% better performance over other state-of-the-art index structures.

In summary, this paper makes the following contributions.

- An empirical evaluation that manifests the challenges of supporting variable-length string keys in learned index structures.
- A novel learned index design, SIndex, that exploits common prefixes to achieve efficient model inference and data accesses.
- A set of experimental results that demonstrates the effectiveness of SIndex with both real-world and synthesized datasets.

[1] https://ipads.se.sjtu.edu.cn:1312/opensource/xindex/-/tree/sindex

[2] Although the model accuracy highly depends on the datasets, we have witnessed such phenomenon with synthesized datasets of various distributions.

APSys '20, August 24–25, 2020, Tsukuba, Japan
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8069-0/20/08…$15.00
https://doi.org/10.1145/3409963.3410496

2 BACKGROUND AND MOTIVATION

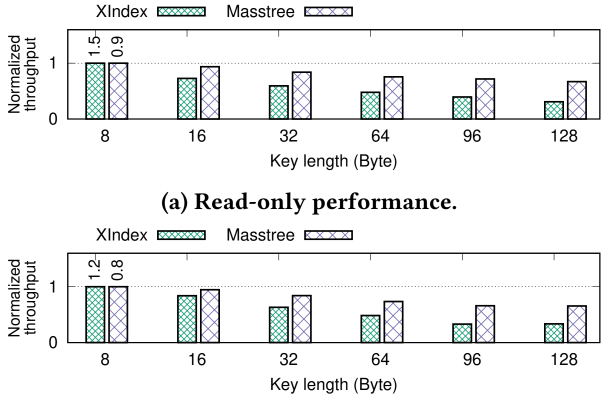
2.1 Learned Indexes

Learned indexes use ML models to index data, key-value records [7, 10, 12, 15, 17, 18]. Specifically, indexes are considered as functions that map keys to the positions of associated records. Thus, the core idea of learned indexes is using the ML model to predict the record positions of the given keys. In the range index setting, assuming the records are continuously sorted, the index function is effectively the cumulative distribution function (CDF) of keys [12]. After using ML models, such as neural networks, to learn the CDF F, record positions can be calculated by $\lfloor F(key) \times N \rfloor$, where N is the number of records.

The original the learned index[3] [12] adopts two key mechanisms for high performance. First, it introduces a hierarchical model architecture, named *Recursive Model Indexes* (RMI), to improve prediction accuracy. Since the prediction error of a single model increases with the increase of dataset size, the learned index uses a multi-stage architecture to reduce the error. In RMI, each model at higher stages dispatch requests to the models at lower stages; the last stage models predict the positions of requested records with given keys. Second, each last stage model memorizes the maximal prediction error e_{max} and the minimal error e_{min}. After getting the predicted position p, the learned index performs a binary search within the range $[p + e_{min}, p + e_{max}]$, which is guaranteed to contain the record if it exists.

To handle concurrent update operations efficiently, Tang et al. proposed a technique called *Two-Phace Compaction* and integrated it in XIndex, an updatable concurrent learned index [18]. XIndex is a two-layer index structure. The top layer contains a single node called *root node*, and the bottom layer includes several nodes called *group nodes*. XIndex range-partitions records into different groups. An RMI model is used in the root node to index those groups. Each group is responsible for indexing a range of data using multiple linear models and a delta index. The data indexed by a group's linear models are contiguous, sorted in an array. The delta index is used to buffer inserted records, and it is conditionally compacted with the sorted data indexed by models.

To process a request, XIndex first locates the target group through the root node's RMI model. Then it finds the corresponding model in the group node that manages the given key. For a read request, XIndex uses the model to search the target record within the sorted data array. If the record is not found, XIndex continues to search the delta index. For a write request, XIndex first tries to perform an update in-place in the sorted data array. If a qualified record does not exist, XIndex then inserts a new record into the delta index.



(a) Read-only performance.

(b) Read-write performance.

Figure 1: Normalized performance of XIndex and Masstree with various key lengths under 100M randomly generated datasets. Throughputs are normalized to their respective throughputs under 8-byte string keys. The numbers above the bars show the absolute throughput in MOPS. XIndex tokenizes a string into a feature vector $x \in \mathbb{R}^n$ as model features, where n is the string length, and x_i is the ASCII value of i-th character. The read-write ratio is 9:1 for the read-write workload, where writes comprise of inserts, deletes and updates (1:1:2).

Comparing with the learned index, XIndex has 2.5× better performance with the workload of 10% updates, and 1.7× better with the read-only workload of skew accesses. For the read-only workload with uniform access distribution, the performance of XIndex is similar to the learned index.

2.2 Problem Statement

Existing learned indexes have shown superior performance on integer keys. However, they suffer significant performance degradation under string keys. Figures 1a and 1b show the read-only and read-write performance of XIndex, in comparison with Masstree, with various key lengths. Both read and write performance of XIndex drops dramatically with the increase of key length. For the read-only workload, when the key length is 128 bytes, XIndex only has 30% of its 8-byte key performance. For the read-write workload, XIndex's performance drops more than 66%, while Masstree's only drops 34%.

There are three reasons for the poor performance. First, the model computation cost increases dramatically along with the key length. Even for the simplest model, namely the linear model, model computation still costs 400 ns when the key length is 128 bytes, 23× higher than that of an 8-byte key. Second, the model errors also increase. For the random

[3] We refer to the learned index design proposed by Kraska et al. as "the learned index".

dataset, the average error grows from 24 to 68 as the key length grows from 8 to 128 bytes. Increased errors indicate that more records are accessed during lookups, which increases the latency. More complicated models, such as neural networks, can help reduce errors [12]. But they come with unacceptable prediction cost and training cost (1400 ns of prediction for 128-byte key), leading to no improvement for the overall performance. Third, the binary search also incurs large overhead under string keys — it takes 1370 ns for 128-byte keys and only 590 ns for 8-byte keys. This is because the comparison cost is proportional to the key length.

To summarize, the key length is critical for performance as it equals both model feature length and the number of comparisons in bytes. Our key insight is that *some comparisons and model computations are unnecessary* — in many cases, substrings of keys are sufficient to uniquely identify each record. By eliminating these overheads, we can improve the performance with string keys.

3 SINDEX

Figure 2 shows the architecture of SIndex. Similar to XIndex, SIndex also adopts a two-layer design — one root node in the top layer and several group nodes in the bottom layer. Each group is responsible for storing records of a specific key range. Records are stored either in a sorted data array, indexed by a linear model, or a delta index if introduced by insertions. For indexing groups, the root node stores each group's pivot keys, the lower bound of a group's responsible range. The root leverages multiple linear models to index groups. SIndex continuously checks all groups in the background. If a group's delta index size is larger than the size threshold given by the user, SIndex performs compaction. During compaction, SIndex merges the delta index with the old sorted array into a new sorted array and retrains models.

SIndex makes three important design choices that are tailored for string keys. First, SIndex uses partial keys to reduce both model computation cost and comparison cost (section 3.1). The partial keys are order-preserving substrings of original keys that uniquely identify each record within a group. An efficient algorithm is used to compute partial keys. Second, SIndex leverages a greedy grouping strategy to adaptively partition keys into different groups (section 3.2). After partitioning, each group contains a maximal number of keys, which keeps the group's model error and partial key length under specified thresholds. This partition scheme improves SIndex by reducing the number of groups and hence the indexing burden of the root. Third, SIndex uses a piecewise linear model instead of an RMI model in the root node to index groups (section 3.3). Using a piecewise linear model helps SIndex gain control on indexing the key range of each model. As a result, partial keys and greedy grouping strategy can be reused for the root.

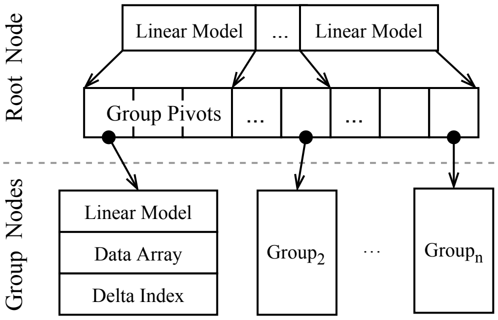


Figure 2: The architecture of SIndex.

3.1 Partial Key

In each group, SIndex extracts partial keys for model computation and comparisons. Informally, partial keys are the shortest fixed-length substrings of the keys within the group, which (1) have stripped off common prefixes, and (2) remain distinguishable from each other with respect to their original ordering. Specifically, for any key in a group, denoted as k, the partial key is $k[pl{:}pl+el]$, which is an *el*-length substring, starting at the zero-based index *pl*. The *pl* is the longest common prefix length among keys in the group, and the *el* is the *effective key length* defined below. The effective key length is the shortest prefix length among the stripped keys, keys with first *pl* characters removed, that still maintains uniqueness within one group.

Algorithm 1 summarizes the procedure for identifying the partial key of a sorted key array. Given a sorted key array, SIndex first initializes the prefix length as the common prefix length of the first two keys, using the helper function *CommonPrefixLen* (Line 1). *CommonPrefixLen* returns the common prefix length of the substrings of input keys *k1* and *k2*, starting at *s* (Line 9-13). Afterward, SIndex iterates over the sorted key array with a for loop to calculate the longest common prefix of all the keys, *pl*, and the longest common prefix length of two adjacent keys, *max_prefix*. Within a loop, let k be the key being examined, SIndex updates *pl* to be the prefix length between the existing common prefix among already examined keys and k (Line 4). *max_prefix* is updated to be the common prefix length between the current key and its previous key, $pl + l$, if this value is larger than the previous one (Line 5-6). Finally, the effective length is calculated by $max\_prefix - pl$ (Line 7).

SIndex benefits from partial keys in two ways. First, partial keys improve the efficiency of both model computation and model training: SIndex directly tokenizes partial keys into feature vectors, resulting in shorter feature length and hence

Algorithm 1: Compute Partial Key

```
In  : A sorted key array $K$, a key length $kl$.
Out : The prefix length $pl$, the effective key length $el$.
1  $pl \leftarrow CommonPrefixLen(0, K[0], kl, K[1], kl)$
2  $max\_prefix \leftarrow pl$
3  for $i \leftarrow 2$ to $K.size$ do
4      $pl \leftarrow CommonPrefixLen(0, K[i-1], pl, K[i], kl)$
5      $l \leftarrow CommonPrefixLen(pl, K[i-1], kl, K[i], kl)$
6      $max\_prefix \leftarrow \max(max\_prefix, pl + l)$
7  $el \leftarrow max\_prefix - pl$
8  return $pl, el$

9  Function $CommonPrefixLen(s, k1, len1, k2, len2)$:
10     for $i \leftarrow s$ to $\min(len1, len2) - 1$ do
11         if $k1[i] \neq k2[i]$ then
12             return $i - s$
13     return $\min(len1, len2) - s$
```

reduced arithmetic computation. Second, the binary search overhead is reduced as well. SIndex uses the partial key for key comparison because of the uniqueness and order preservation of each partial key. Hence, the comparison cost is, to a great extent, diminished.

3.2 Greedy Grouping

SIndex greedily range-partitions data into different groups to ensure that the partial key length and model errors are under specified thresholds. Algorithm 2 depicts this greedy strategy of grouping data in SIndex. User-specified parameters are used to fine-tune SIndex: the error threshold *et*, the partial key length threshold *pt*, the forward step size *fs*, and the backward step size *bs*. SIndex iterates over all the records. During the iteration, it determines whether to add *fs* records into the current group or remove *bs* records from the current group according to the two thresholds, *et* and *pt*. Specifically, each time SIndex moves forwards, it first adds the next *fs* records to the current group (Line 20-21). Then SIndex trains a linear model on the current group to get the average error (Line 22-23) and uses Algorithm 1 to compute the length of the partial key (Line 24). It continues to move forwards as long as both the model error and partial key length are smaller than their respective thresholds. The last forward step can cause a violation of thresholds, therefore SIndex uses backward steps for alleviation. For each backward step, SIndex removes the most recently added *bs* records from the current group (Line 27). It repeats backward steps until both the model error restriction and partial key length restriction are restored (Line 25).

Algorithm 2: Greedy Grouping

```
In  : A sorted data array $D$, an error threshold $et$, a
      partial key length threshold $pt$, a forward step
      size $fs$, a backward step size $bs$ ($< fs$).
Out : Groups of records $G$.
14 $G \leftarrow \emptyset$
15 while $i < D.size$ do
16     $cur\_grp \leftarrow$ new empty group
17     $err \leftarrow 0; pl \leftarrow 0$
18     while $err < et$ and $pl < pt$ do
19         $i \leftarrow i + fs$
20         $records \leftarrow$ retrieve next $fs$ records
21         add $records$ to $cur\_grp$
22         train a linear model on $cur\_grp$
23         $err \leftarrow$ get average model error for $cur\_grp$
24         $pl \leftarrow$ get partial key length for $cur\_grp.Keys$
25     while $err \geq et$ or $pl \geq pt$ do
26         $i \leftarrow i - bs$
27         remove last $bs$ records from $cur\_grp$
28         train model on $cur\_grp$
29         $err \leftarrow$ get average model error for $cur\_grp$
30         $pl \leftarrow$ get partial key length for $cur\_grp.Keys$
31     add $cur\_grp$ to $G$
32 return $G$
```

3.3 Piecewise Linear Model for the Root

SIndex uses a piecewise linear model in the root node to index groups. A piecewise linear model consists of a series of linear segments, each of which is represented as a linear model. Each model is responsible for serving requests of non-overlapping key ranges. At training time, these linear models are trained using group pivots. SIndex exploits the same greedy grouping algorithm to determine assignment of group pivots to each model, which in turn determines the key range of each model. Then it applies the partial key in each model. At inference time, SIndex uses binary search to find responsible linear models for requests.

SIndex chooses the piecewise linear model instead of the RMI model, which is used in the root of XIndex, for two reasons. First, in RMI, two pivots distant with each other can be assigned to the same leaf model, which results in a large span in the key range of the leaf model. The larger span the key range has, the less chance SIndex has to remove common prefix for the model, and hence less performance improvement from partial keys. Second, in practice, multiple stages of linear models in RMI causes a significant increase in computational overhead for string keys. This increase cancels off the benefits of reduced model error brought by

RMI, leading to no improvement for the overall performance. For a specific case, after adding another layer, binary search time is reduced from 1291 ns to 1091 ns, but the inference time increases from 227 ns to 616 ns.

3.4 Optimization

To accelerate model inference, SIndex exploits SIMD instructions to perform model computation. Specifically, SIndex uses _mm256_fmadd_pd in FMA to perform the fused multiply-add operation of every four 8-byte floating-point numbers in one instruction. SIndex will fall back to the conventional way — one multiplication at a time — for dot product when the feature length is less than four.

4 EVALUATION

In this section, we experimentally evaluated the performance of SIndex and compared it with other state-of-the-art index structures.

Implementation. We implement SIndex in C++. We store metadata of partial key as part of groups' metadata, including the common prefix length and the effective key length. The partial key is recalculated each time after compaction. SIndex applies the greedy grouping algorithm at initialization.

Dataset. We use two families of datasets throughout experiments. One comprises synthetic datasets where all keys are randomly generated, denote as "R[key length]-[dataset size]". The other is a real-world dataset containing 92M URLs of quotes from Memetracker [14]. The maximum length of URLs is 128 bytes and the average length is 62 bytes. We tokenize URLs into 128-length feature vectors. For URLs with length $n < 128$, we set $x_i = 0$ for $i > n$.

Counterparts. We compare SIndex with XIndex [18], Masstree [16], and Wormhole [19]. XIndex is a concurrent index based on learned models. Masstree is a concurrent index that layers B-tree over Trie. Wormhole is an index that replaces the internal nodes of B-tree with Trie.

Configuration. The error threshold, *et*, and the partial key length threshold, *pt*, are set to 50 and 4 for the random dataset, and 500 and 40 for the URL dataset. The forward step size, *fs*, and the backward step size *bs* are set to 500 and 50, respectively. One thread is configured for background compaction, which is accounted for in the total thread count. We use the default configurations for other indexes. For each experiment, we report the steady-state performance after warming up. The tests are run on a server with two 12-core Intel Xeon E5-2650 v4 CPU, each with 30 MB LLC.

4.1 Overall Performance

In this section, we measure the overall performance of SIndex, including the performance under both read-only workloads and read-write workloads.

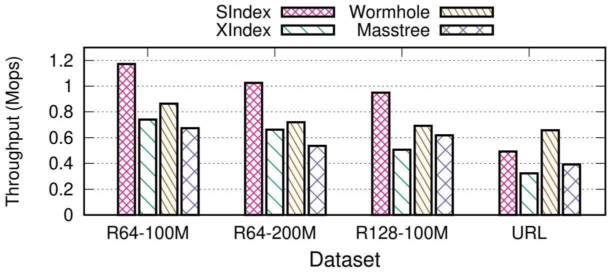


Figure 3: Read-only single-thread performance with different datasets.

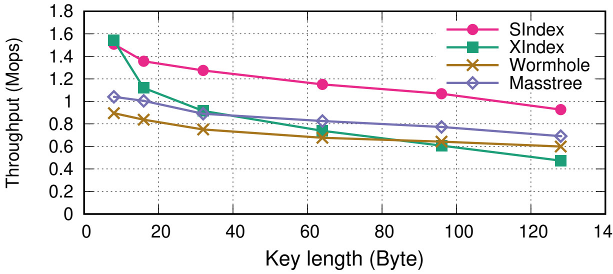


Figure 4: Read-only single-thread performance with different key lengths using random datasets.

Read-only performance. We first measure the single-thread throughputs of SIndex with read-only workloads using different datasets. As shown in Figure 3, SIndex achieves the best performance among all the indexes under random datasets but has fewer performance advantages under the URL dataset. SIndex outperforms XIndex, Masstree and Wormhole by up to 88%, 91% and 43% respectively under random datasets. Under the URL dataset, SIndex still shows relatively good performance compared with XIndex (1.5×) and Masstree (1.3×). However, SIndex's performance is worse than Wormhole by 25%. This is because the URL dataset has more complex data distribution, leading to larger errors — 4.5× larger than the average error of the random datasets. Also, the partial key length is 39 bytes for the URL dataset, which is nearly 10× larger than that of the random datasets.

We then evaluate SIndex with various key lengths. Figure 4 shows the results under 100M random dataset, with key length ranging from 8 bytes to 128 bytes. SIndex shows considerable performance advantages for all key lengths. Its performance shows similar scalability in key length as Masstree and Wormhole. XIndex only achieves good performance with short keys but suffers large performance degradation when keys are larger than 8 bytes: the throughout with 128-byte keys is only 30% of that with 8-byte keys. Compared with XIndex, with 128-byte keys, SIndex maintains 62% of its 8-byte throughout and outperforms XIndex by 91%.

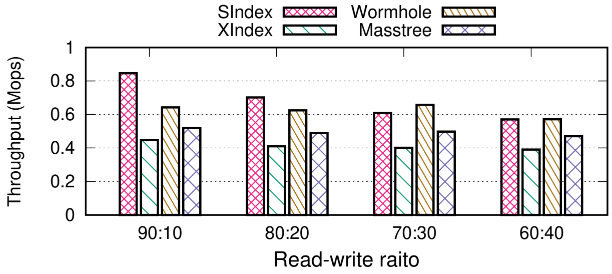


Figure 5: Read-write single-thread performance with different read-write raitos using R128-100M dataset.

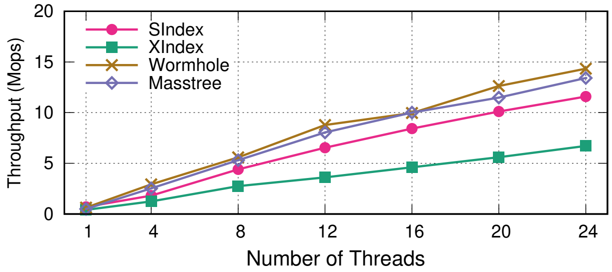


Figure 6: Read-write performance scalability using R128-100M dataset. The worklaod has a read-write ratio of 9:1.

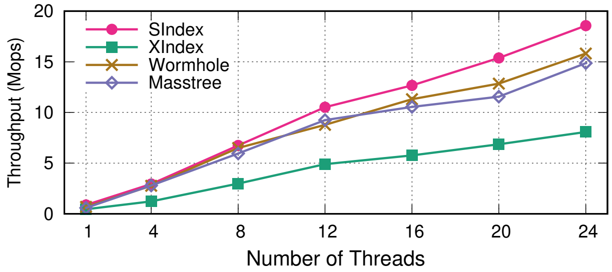


Figure 7: Read-write performance scalability using R128-100M dataset. The worklaod has a read-write ratio of 9:1 and all writes are in-place updates.

Read-write performance. We then investigate the performance of SIndex with read-write workloads using different read-write ratios. We experiment with the random dataset of 100M 128-byte keys. The write requests include in-place updates, inserts, and deletes, with a ratio of 2:1:1. As shown in Figure 5, SIndex maintains relatively good performance under read-write workloads. When there are 10% writes, SIndex has 89% better performance than XIndex. When there are 40% writes, SIndex still has comparable performance as Wormhole, while being 45% better than XIndex and 21% better than Masstree.

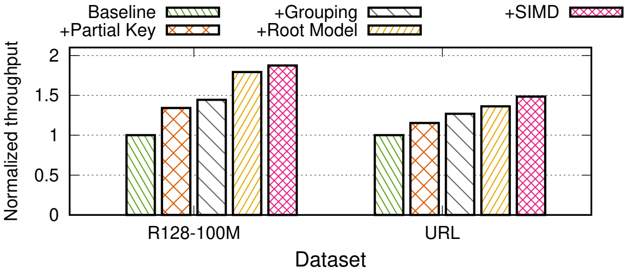


Figure 8: Performance breakdown of SIndex.

Figure 6 shows the scalability in threads under a workload with a 90:10 read-write ratio. SIndex shows up to 72% performance advantages compared with XIndex. However, SIndex is worse than Wormhole and Masstree under multi-thread read-write scenarios (19% and 13% worse under 24-thread). This is mainly because the compaction process is not quick enough to merge data in delta indexes. Despite the use of the partial key, the model retraining is still time-consuming in SIndex— 300 μs to train a linear model with 1400 keys. This leads to a large delta index size — up to 650 on average under 24 threads.

We also evaluate the scalability of SIndex with the same read-write ratio, but all writes are in-place updates. Figure 7 shows the results. SIndex exhibits good scalability in this workload since there is no need to perform compactions. The 24-thread performance can achieve 21× of its single-thread performance, which is 17% better than Wormhole.

4.2 Performance Breakdown

In this subsection, we analyze the performance improvement brought by each design. We start from a baseline design where none of the proposed techniques is used, denoted as "Baseline". Then we incrementally apply our design decisions, namely the partial key, the greedy grouping, the piecewise linear model, and the SIMD optimization, and finally obtain the full SIndex design. Figure 8 shows the throughputs of applying each of the techniques under random and URL datasets. Throughputs are normalized to the baseline.

We first apply the partial key in each group. With the partial key, SIndex improves 35% and 15% under random and URL datasets, respectively. For the random dataset, the average key length for model and comparison of all groups is significantly reduced from 128 bytes to 4 bytes. For the URL dataset, it is reduced from 128 to 80, with an average prefix length of 14 bytes. Next, we apply the greedy grouping strategy to range-partition data for group nodes. SIndex has an improvement of 7% and 9% for the two datasets, respectively. The average partial key length for groups further decreases to 39 bytes for the URL dataset. Afterward, the use of the piecewise linear model in the root node reduces the root model inference time, bringing 24% and 7% overall improvement. For the random dataset, the root model inference time decreases from 640 ns to 70 ns. For the URL, it decreases from 640 ns to 354 ns, with a partial key length of 105 bytes for root models. Finally, we adopt SIMD for model inference, which gives SIndex another improvement of 4% for random and 9% for URL.

5 RELATED WORKS

Learned indexes. The proposal of learned index structures [12] has boosted a rich corpus of index structures based on learned models. RMI by Kraska et al. [12] uses a hierarchy of simple ML models to achieve adequate accuracy with small model inference cost. For better space efficiency, Ferragina and Vinciguerra propose PGM-index [8], a learned index that achieves a provably efficient time and space bounds. It extends the RMI design by adapting to both the distribution of keys and their access frequencies. For better build efficiency, Kipf et al. propose RadixSpline [11], which employs a bottom-up build method that only needs a single pass over the dataset. FITing-Tree [10] is a data-aware index structure, which can be seen as a specialization of RMI. It approximates data distribution using piecewise linear functions and provides bounded error specified by users to balance lookup performance and space consumption. ALEX [7] is an updatable learned index that proposes a careful space-time trade-off. It dynamically adapts its RMI structure based on the workload and offers a Gapped Array for model-based insertion. Tang et al. propose XIndex [18], a concurrent learned index that can handle concurrent update operations. XIndex uses a two-phase compaction scheme to merge buffered data into existing sorted data efficiently. Flood [17] is a learned multi-dimensional index. It exploits ML methods to learn an optimal layout that divides d-dimensional data space into a grid of contiguous cells and uses linear models to speed up queries. LISA [15] is a learned index structure designed for disk-resident spatial data. It leverages learned models to generate data layouts in the disk. Compared with these works, SIndex is the only learned index structure that is optimized for string keys.

Conventional indexes. A number of conventional index structures are extensively studied for string keys, most of which are variants of trie. Trie [9] is a tree-based index structure where each node represents a slice of the string key. Keys with common prefixes can be served with a small set of nodes. The lookup performance is bounded by the length of the string. Masstree [16] partitions key into 8-byte segments and index them with a trie structure. Within each trie node, a concurrent B-tree is used to index the segments. Height Optimized Trie (HOT) [3] maximizes the node fanout to reduce the height of trie. It stores sparse partial keys in each node for compactness and is carefully engineered for fast SIMD operations. Wormhole [19] is an index structure that hybrids B-tree, trie, and hash table. It replaces the internal nodes of B-tree with a trie, which is then encoded with a hash table for efficient access. The Adaptive Radix Tree (ART) [13] adaptively chooses the most compact and efficient data layout for internal nodes, thus achieving both space and time efficiency. Compared with them, SIndex takes advantage of ML models to achieve fast indexing.

6 CONCLUSION

In this paper, we present SIndex, the first concurrent learned index for string keys. To mitigate the unique challenges introduced by the combination of string key workloads and ML methods, we propose three innovative designs: *partial keys* for reducing the cost of model inference and data access, *a greedy grouping strategy* for limiting the length of partial keys in each group, and *a piecewise linear model for the root* for reusing the partial key and the greedy grouping strategy. Our evaluation shows that SIndex is able to maintain competitive performance under string workloads using both synthetic and real-world datasets. SIndex is publicly available at https://ipads.se.sjtu.edu.cn:1312/opensource/xindex/-/tree/sindex.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 61902242), and the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Zhaoguo Wang (zhaoguowang@sjtu.edu.cn) is the corresponding author.

REFERENCES

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. *SIGMETRICS Perform. Eval. Rev.* 40, 1, 53âĂŞ64. https://doi.org/10.1145/2318857.2254766

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53âĂŞ64. https://doi.org/10.1145/2254756.2254766

[3] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 521âĂŞ534. https://doi.org/10.1145/3183713.3196896

[4] Philip Bohannon, Peter Mcllroy, and Rajeev Rastogi. 2001. Main-Memory Index Structures with Fixed-Size Partial Keys. In *Proceedings*

of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD âĂŽ01). Association for Computing Machinery, New York, NY, USA, 163âĂŞ174. https://doi.org/10.1145/375663.375681

[5] Philip Bohannon, Peter Mcllroy, and Rajeev Rastogi. 2001. Main-Memory Index Structures with Fixed-Size Partial Keys. *SIGMOD Rec.* 30, 2 (May 2001), 163âĂŞ174. https://doi.org/10.1145/376284.375681

[6] 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 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 969âĂŞ984. https://doi.org/10.1145/3318464.3389711

[8] 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 (April 2020), 1162âĂŞ1175. https://doi.org/10.14778/3389133.3389135

[9] Edward Fredkin. 1960. Trie Memory. *Commun. ACM* 3, 9 (Sept. 1960), 490âĂŞ499. https://doi.org/10.1145/367390.367400

[10] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1189âĂŞ1206. https://doi.org/10.1145/3299869.3319860

[11] 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 '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3401071.3401659

[12] 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 '18)*. Association for Computing Machinery, New York, NY, USA, 489âĂŞ504. https://doi.org/10.1145/3183713.3196909

[13] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812 ISSN: 1063-6382.

[14] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. 2009. Meme-Tracking and the Dynamics of the News Cycle. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '09)*. Association for Computing Machinery, New York, NY, USA, 497âĂŞ506. https://doi.org/10.1145/1557019.1557077

[15] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2119âĂŞ2133. https://doi.org/10.1145/3318464.3389703

[16] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183âĂŞ196. https://doi.org/10.1145/2168836.2168855

[17] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985âĂŞ1000. https://doi.org/10.1145/3318464.3380579

[18] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 308âĂŞ320. https://doi.org/10.1145/3332466.3374547

[19] Xingbo Wu, Fan Ni, and Song Jiang. 2019. Wormhole: A Fast Ordered Index for In-Memory Data Management. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 18, 16 pages. https://doi.org/10.1145/3302424.3303955