XIndex: A Scalable Learned Index for Multicore Data Storage

Chuzhe Tang†‡, Youyun Wang†‡, Zhiyuan Dong†‡, Gansen Hu†‡
Zhaoguo Wang†‡, Minjie Wang⋄, Haibo Chen†‡
† Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
‡ Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
⋄ Department of Computer Science, New York University

Abstract

We present XIndex, a concurrent ordered index designed for fast queries. Similar to a recent proposal of the learned index, XIndex uses learned models to optimize index efficiency. Comparing with the learned index, XIndex is able to effectively handle concurrent writes without affecting the query performance by leveraging fine-grained synchronization and a new compaction scheme, Two-Phase Compaction. Furthermore, XIndex adapts its structure according to runtime workload characteristics to support dynamic workload. We demonstrate the advantages of XIndex with both YCSB and TPC-C (KV), a TPC-C variant for key-value stores. XIndex achieves up to 3.2× and 4.4× performance improvement comparing with Masstree and Wormhole, respectively, on a 24-core machine, and it is open-sourced[1].

CCS Concepts • **Information systems** → **Data structures**; • **Theory of computation** → **Concurrent algorithms**.

1 Introduction

The pioneering study on the learned index [15] opens up a new perspective on how machine learning can re-sculpt the decades-old system component, indexing structure. The key idea of the learned index is to use learned models to approximate indexes. It trains the model with records' keys and their positions, then uses the model to predict the position with the given key.

To deliver high lookup performance, the learned index uses simple learned models such as linear models or single-layer neural networks. To accommodate to the limited capacity of simple models (inability to well fit complex functions), the learned index adds extra requirements on the data layout. For instance, to learn an ordered index, it requires the data to be both ordered and contiguous, so the key-position mapping is easier to learn. Using simple learned models, the learned index[2] performs 1.5-3× better than B-tree [15].

However, the current study of the learned index is still preliminary and lacks practicability in a broad class of real-world scenarios because of two limitations. First, it does not support any modifications, including inserts, updates, or removes. Second, it assumes the workload has a relative static query distribution[3] — it assumes all data are uniformly accessed. Nevertheless, making the learned index practical for dynamic workloads with writes is not an easy task, because its high performance is tied closely to both data distribution and query distribution, especially for ordered index. First, the learned index requires simple data distribution by enforcing ordered and contiguous data layout. Thus, it needs to reconstruct the layout and retrain the model to handle every write request. Although there are proposals [9, 18] which try to handle writes for the learned index efficiently, none of them can ensure the correctness in the face of concurrent operations. Second, the learned index is sensitive to the changes of the data and query distribution at runtime. Its current design employs several learned models, and each is in charge of a portion of data. However, the prediction error of every model varies. At the same time, queries in real-world workloads tend to be skew, where some "hot" keys are much more frequently queried than others [7, 10, 16, 25]. As a result, when the models that index those hot keys have large errors, queries can incur high overhead (Section 2.2).

In this paper, we present XIndex, a new fully-fledged concurrent index structure inspired by the learned index. While XIndex leverages learned models to speed up the lookup, it can handle concurrent writes efficiently with good scalability.

[1] https://ipads.se.sjtu.edu.cn:1312/opensource/xindex.git

[2] In following paragraphs, we refer to the learned range index proposed in [15] as "the learned index".

[3] The query distribution describes the access frequencies of keys among queries within a specific workload. By contrast, data distribution describes the keys and their lookup positions within a dataset.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374547

Moreover, it is designed to adapt its structure deterministically at runtime and decouple its efficiency from runtime workload characteristics. Specifically, this paper makes the following contributions:

A scalable and concurrent learned index, XIndex. With understandings of the learned index's pros and cons, XIndex range-partitions data into different groups, each attached with learned models for lookup and a delta index to handle inserts. To achieve high performance, XIndex exploits a combination of innovative methods (e.g., Two-Phase Compaction) and classic techniques (e.g., read-copy-update (RCU) [21], optimistic concurrency control [3, 4, 20]).

Data structure adjustment according to runtime workload characteristics. Unlike B-tree, whose structure is decided by the fanout, XIndex adapts its structure to runtime workload characteristics through structure update operations, such as group split and group merge. Users can configure the expected behaviors through parameters such as error bound threshold and delta index size threshold.

Implementation and evaluation with macro and micro benchmarks. We implement XIndex and compare it against state-of-the-art structures (The learned index [15], Wormhole [24], Masstree [20] and stx::Btree [1]). The benchmarks we used include different microbenchmarks, the YCSB benchmark, and the TPC-C (KV) benchmark, which is a TPC-C variant for key-value stores. The experimental results show that, with 24 CPU cores, XIndex achieves up to 3.2× and 4.4× performance improvement comparing with Masstree and Wormhole, respectively.

The rest of the paper is organized as follows: Section 2 describes the background and motivation; Section 3 gives the design of XIndex; Section 4, 5, 6 present the concurrent support, the runtime structure adjustment strategy, and optimizations accordingly; Section 7 shows the evaluation results; Section 8 provides a short discussion on alternative design choices and limitations; Section 9 summarizes related works; Section 10 concludes this paper.

2 Background & Motivation

2.1 The learned index

The insight of the learned index is that range indexes can be viewed as functions mapping keys to data positions. For fixed-length key-value records, assuming they are sorted in an array, this function is effectively a cumulative distribution function (CDF) of keys' distribution. Given the CDF F, the position of a record is $\lfloor F(key) \times N \rfloor$, where N is the total number of records.

The core idea behind the learned index is to approximate the CDF with machine learning models, such as deep neural networks, and predict record positions using models. In order to provide the correctness guarantee despite prediction errors, the learned index stores the maximal and minimal prediction errors of the model. After training the model, the learned index calculates the errors by taking the difference between the actual and predicted positions of each key and stores the maximum and minimum. For a record in the sorted array, its actual position must fall in $[pred(key) + min\_err, pred(key) + max\_err]$, where $pred(key)$ is the predicted position. Therefore, the learned index uses binary search within the range to locate the record. We use error bound, $log_2(max\_err - min\_err + 1)$, to express the cost of lookup. The learned index will be more effective with a smaller error bound. In contrast with common machine learning practices where model generalization matters, the learned index expects the model to overfit to reduce errors over existing data.

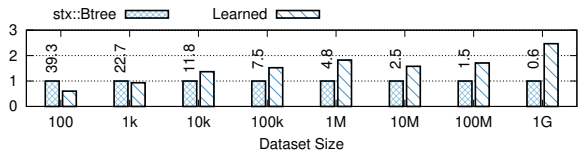

Figure 1. The learned index throughput **normalized to** stx::Btree. The numbers are stx::Btree's absolute throughputs in MOPS. The learned index uses 10k models in 2-staged RMI where both stages use linear models. stx::Btree uses the default fanout, 16.

However, using a single model to learn the entire CDF falls short in prediction accuracy due to the complexity of CDFs. To improve the prediction accuracy and reduce the error bound, the paper proposes a *staged model architecture*, termed Recursive Model Indexes (RMI). RMI contains multiple stages of models, which resemble the multi-level structure of B-tree. The model at each internal stage predicts which model to be activated at the next stage; the model in the leaf stage directly predicts the CDF values. With RMI architecture, each leaf stage model only approximates a small part of the complete CDF, a much simpler function to learn, which in turn reduces the model error bounds.

We evaluated the learned index with different dataset sizes under a *normal* distribution and compared it against stx::Btree [1], an open-sourced B-tree (Figure 1). The learned index can outperform stx::Btree with large datasets (≥ 10k) due to small binary search costs, while with small datasets, its performance is limited by the model computation cost. For example, when the dataset size is 100, the learned index spends much time on model computation (20 ns out of 42 ns). In contrast, stx::Btree only needs 25 ns to traverse two nodes for each query. When the dataset size increases, the learned index's binary search cost increases much slower than stx::Btree's query time and its model computation cost is constant. For example, when dataset size increases from 1M to 10M, the learned index's binary search time only grows 37% (68 ns to 94 ns) and the error bound increases from 4.7 to 6.6. However, stx::Btree's query time increases by 92%

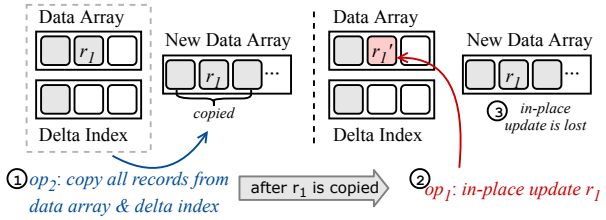

Figure 2. Consistency issue with concurrent operations.

Systems	Workloads			
	Skewed 1	Skewed 2	Skewed 3	Uniform
stx::Btree	1.84	1.86	1.83	1.16
learned index	1.57	3.71	1.41	2.38
Error bound	15.71	5.87	19.52	6.95

Table 1. Performance of stx::Btree and the learned index under different query distributions on the *osm* dataset. Throughputs are shown in MOPS. Error bound refers to the average error bound weighted by models' access frequencies.

(207 ns to 399ns). Therefore, the learned index has better performance than stx::Btree with larger datasets.

2.2 The issues

Despite of the performance advantage of the learned index, there are two issues that limit its practicability.

First, the learned index does not provide an efficient method to handle writes, especially under concurrent scenarios. Based on the current design, an intuitive solution is to buffer all writes in a delta index, then periodically compact it with the learned index. The compaction includes merging the data into a new sorted data array and retraining the models. Though straightforward, this method suffers from severe slowdown for queries. One reason is that each request has to first go through the delta index before looking up the learned index. Considering building the learned index with 200M records, and using Masstree to be the delta index, with a workload of 10% writes, the query latency increases from 530ns to 1557ns due to the cost of searching Masstree. Another reason is that concurrent requests are blocked by the compaction, which is time-consuming. It takes up to 30 seconds to compact a delta index of 100k records with the learned index with 200M records.

A possible improvement for the above method is performing updates in-place with a non-blocking compaction scheme. When we perform updates to existing records in-place, then only newly inserted records are in the delta index. Thus, a query can only lookup the delta index when it fails to find a matching record in the learned index. Meanwhile, to avoid blocking query requests, we can compact the data asynchronously with background threads. However, the correctness issue arises if we simply use these two methods together — the effect of updates might be lost due to the data race with background compaction. Let us consider this example (Figure 2), where operation op_1 updates record r_1 in-place and operation op_2 concurrently merges the delta index with the learned index into a new data array. With the following interleaving, op_1's update to r_1 will be lost due to the concurrent compaction: 1) op_2 starts the compaction and copies r_1 to the new array; 2) op_1 updates r_1 in the old array; 3) op_2 finishes the compaction, updates the data array, and retrains the model.

Second, the learned index's performance is tied closely to workload characteristics, including both data and query distributions. This is because the lookup efficiency depends on the error bounds of specific leaf stage models activated for the queries. Meanwhile, the error bounds of different models vary. As a result, the learned index can have worse performance than B-tree with certain workloads. Table 1 shows the performance of the learned index and stx::Btree under both uniform and skewed query distributions on the *osm* dataset (details in Section 7). Under the uniform query distribution, all keys have the same chance to be accessed. Under the skewed query distribution, 95% queries access 5% hot records, and the hot records of each workload reside in different ranges. "Skewed 1" chooses hot keys from the 94th to 99th percentiles of the sorted data array. "Skewed 2" chooses from the 35th to 40th, while "Skewed 3" chooses from the 95th to 100th.

The learned index has better performance than stx::Btree under the workloads of "Skewed 2" and "Uniform", but is outperformed under "Skewed 1" and "Skewed 3". This is because under workload "Skewed 1" and "Skewed 3", the learned index has much higher average error bounds on the frequently accessed records, which hinders the query performance. The underlying cause is that the learned index only minimizes each model's error individually, lacking the consideration for model accuracy differences. Similar results can be observed in other workloads as well (Section 7.3).

3 XIndex Data Structure

3.1 Overview

XIndex adopts a two-layer architecture design (Figure 3). The top layer contains a *root* node which indexes all *group* nodes in the bottom layer. The data is divided into groups by range partitioning. The root node uses a learned RMI model to index the groups. Each group node uses learned linear models to index its data. For writes, XIndex performs updates in-place on existing records, and associates each group with a delta index to buffer insertions.

XIndex introduces a new compaction scheme, Two-Phase Compaction (Section 3.4), to compact the delta index conditionally. The compaction is performed in the background and does not block any concurrent operations. The compaction

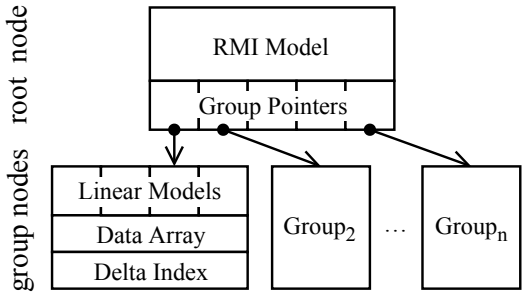

Figure 3. The architecture of XIndex.

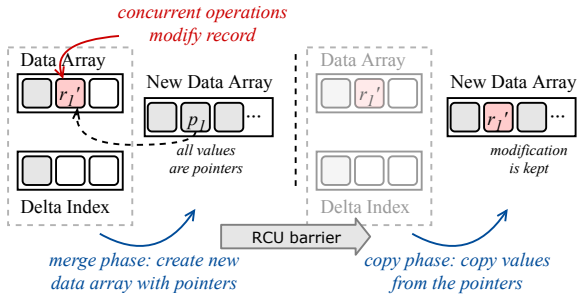

Figure 4. Two-phase compaction prevents concurrent operations being lost.

has two phases: the *merge phase* and the *copy phase*. In the merge phase, XIndex merges the current data array and delta index into a new data array. Instead of directly copying the data, XIndex maintains data references in the new data array. Each reference points to records being compacted, residing in either the old data array or the delta index. After ensuring no accesses on the old data array through an RCU barrier, XIndex performs the copy phase. It replaces each reference in the new data array with the real value. Considering previous example (Figure 2) with Two-Phase Compaction in Figure 4, after the merge phase, the new data array contains references (e.g., p_1) to each record (e.g., r_1). If there is a concurrent writer which updates r_1 to r_1', the writer can safely proceed as the record is already referenced in the new data array. After an RCU barrier, no thread will access the old data array anymore. XIndex replaces p_1 with r_1' in the copy phase.

XIndex is able to adjust its structure according to runtime workload characteristics (Section 5). At runtime, if some group incurs high prediction error, XIndex adds more linear models in that group with "*model split*" to improve the inference accuracy. If a group has too many models or its delta index is too large, XIndex performs *group split* — replacing the group with two new groups, each containing the half data of the old group. XIndex also performs *model merge* and *group merge*, if the merging does not affect the prediction accuracy. Furthermore, if there are too many groups, XIndex retrains the RMI model of the root node and may adjust its structure to improve the accuracy.

Algorithm 1: Structures

```
1  struct root_t:
2    rmi_t rmi;
3    uint32_t group_n;
4    key_t pivots[];
5    group_t* groups[];
6
7  struct record_t:
8    key_t key;
9    val_t val;
10   uint64_t // composite 8B
11     is_ptr : 1, removed : 1
12     lock : 1, version : 61;
13 struct group_t:
14   key_t pivot;
15   bool_t buf_frozen;
16   uint16_t model_n;
17   uint32_t array_size;
18   model_t
       models[MAX_MODEL_N];
19   record_t data_array[];
20   buffer_t* buf;
21   buffer_t* tmp_buf;
22   group_t* next;
```

3.2 Layout

XIndex maintains three basic structures — *record_t*, *root_t*, and *group_t*, for the record, the root node, and the group node, respectively (Algorithm 1).

The *record_t* is the basic representation of the data. It includes the key (*key*), the record data (*val*), and some metadata. The *is_ptr* flag indicates whether *val* is the actual value or a memory reference. The *removed* flag is set when a record is logically removed. The *lock* and *version* are concurrency control information, which ensures execution exclusiveness of concurrent operations.

The *root_t* contains the groups' information and an RMI model. The group information includes each group's address (*groups*), their smallest keys (*pivots*), and the total number of groups (*group_n*). The RMI model (*rmi*) is used to predict the group with a given key. It is trained with elements in *pivots* and their indexes, $\{(pivots[i], i) \mid i = 0, \ldots, group\_n - 1\}$. In the current design, XIndex uses a two-stage RMI architecture solely consisting of linear models. The number of models in its second stage is adjustable at runtime (Section 5).

The *group_t* has three basic components: the data, the models, and the delta index. For the data, all records indexed by the group is continuously stored in *data_array*. Each group uses at least one linear model to index the record in *data_array*, and the models are maintained in *models*. The *model_t* includes parameters of the linear model and the smallest key of the model's belonging data range. The *buf* is the delta index, which buffers all insertions. During compaction, *buf_frozen* is set to be true and *buf* is frozen. The *tmp_buf* serves as a temporary delta index, which buffers all insertions temporarily during the compaction. The *next* pointer is used by group split operation (Section 3.5). For optimization purposes, the *group_t* maintains its smallest key in a separate variable, *pivot*.

Algorithm 2: Get and Put

```
1  get(root, key):
2    group ← get_group(root, key)
3    pos ← get_position(group, key)
4    val ← empty
5    if pos ≠ empty
6      val ← read_record(group.data_array[pos])
7    if val = empty
8      val ← get_from_buffer(group.buf, key)
9    if val = empty && group.tmp_buf ≠ null
10     val ← get_from_buffer(group.tmp_buf, key)
11   return val
12
13 put(root, key, val):
14 retry:
15   group ← get_group(root, key)
16   pos ← get_position(group, key)
17   if pos ≠ empty
18     if update_record(group.data_array[pos], val) = true
19       return
20   if group.buf_frozen = false
21     insert_buffer(group.buf, key, val)
22   else
23     if update_in_buffer(group.buf, key, val) = true
24       return
25     if group.tmp_buf = null
26       goto retry
27     insert_buffer(group.tmp_buf, key, val)
```

3.3 Basic operations

XIndex provides basic index interfaces — *get*, *put*, *remove*, and *scan* (Algorithm 2). All operations first use the root to find the corresponding group (Lines 2 and 15), then look up the position of the requested record in *data_array* with the given key (Lines 3 and 16). After then, their procedures diverge.

To find the corresponding group (*get_group*), XIndex first predicts a group number with the RMI model in the root node (*root.rmi*). Then, it corrects the group number with binary search the *root.pivots* within an error-bounded range. After finding a candidate group, it needs to check the group's *next* pointer further. If the pointer is not *null*, it follows the pointer to find the corresponding group by comparing *group.pivot* with the target key. Checking the *next* is necessary, this is because some newly created group may be linked to a group's *next*, and not indexed by the root yet (Section 3.5).

After finding the group *group*, XIndex tries to look up the record within its *data_array*. It first finds the correct linear model for the prediction. It scans the *group.models* and uses the first model whose smallest key is not larger than the target key. Then, it uses the model to predict a position in the *group.data_array*. Last, it corrects the position with binary search in a range bounded by the model's error.

Algorithm 3: Two-Phase Compaction

```
1  compact(group):
2    /* phase 1 */
3    group.buf_frozen ← true
4    rcu_barrier()
5    group.tmp_buf ← allocate new delta index
6    new_group ← allocate new group
7    new_group.data_array ← merge(group.data_array,
     group.buf)
8    new_group.buf ← group.tmp_buf
9    train new_group's models with new_group.data_array
10   init new_group's other fields
11   old_group ← group
12   atomic_update_reference(group, new_group)
13   rcu_barrier()
14   /* phase 2 */
15   for each record in new_group.data_array
16     replace_pointer(record)
17   rcu_barrier()
18   reclaim old_group's memory
```

After looking up *data_array*, the procedures diverge. For *get*, if XIndex finds a record matching the requested key (Line 5) in *data_array*, then it tries to read a consistent value with helper function *read_record* (Line 6). An *empty* result indicates a logically removed record. In this case, the *get* proceeds to search *buf* (Line 7-8), then search the temporary delta index if *tmp_buf* is not *null* (Line 9-10). A *get* request returns as soon as a non-*empty* result is fetched, otherwise it returns *empty*.

For *put* and *remove*, similar to *get*, if a matching record is found inside *data_array* (Line 17), XIndex first tries to update/remove the record in-place (Line 18). If XIndex cannot perform update/remove in-place, then it proceeds to operate on *buf* (Line 21) and optionally *tmp_buf* (Line 27) only if the *frozen_buf* flag is true (Line 20). For *scan*, XIndex first locates the smallest record that is ≥ requested key, and then consistently reads *n* consecutive records.

We elaborate on the details of *put*, *remove*, and helper functions in conjunction with concurrent background operations in Section 4 since most subtleties stem from consistency consideration.

3.4 Compaction

To ensure consistency in face of concurrent operations (Section 2.2), XIndex divides the compaction into two phases, *merge phase* and *copy phase* (Algorithm 3).

In the merge phase, XIndex merges a group's *data_array* and *buf* into a new sorted array where values are pointers to existing records. XIndex first sets the old group's *buf_frozen* flag to stop newly issued *put*s inserting to *buf* (Line 3). Then

Algorithm 4: Group Split

```
1  split(group):
2    /* step 1 */
3    g'_a, g'_b ← allocate 2 new group
4    {g'_a, g'_b}.{data_array, buf} ← group.{data_array, buf}
5    g'_a.pivot ← group.pivot
6    g'_b.pivot ← group.data_array[group.array_size / 2]
7    g'_a.next ← g'_b
8    init other fields of g'_a and g'_b
9    old_group ← group
10   atomic_update_reference(group, g'_a)
11   {g'_a, g'_b}.buf_frozen ← true
12   rcu_barrier()
13   {g'_a, g'_b}.tmp_buf ← allocate new delta indexes
14   /* step 2.1, merge phase */
15   g_a, g_b ← allocate 2 new groups
16   tmp_array ← merge(old_group.data_array,
       old_group.buf)
17   {g_a, g_b}.data_array ← split(tmp_array, g'_b.pivot)
18   {g_a, g_b}.buf ← {g'_a,g'_b}.tmp_buf
19   train {g_a, g_b}'s models with {g_a, g_b}.data rray
20   {g_a, g_b}.pivot ← {g'_a, g'_b}.pivot
21   g_a.next ← g_b
22   init g_a's and g_b's other fields
23   atomic_update_reference(group, g_a)
24   rcu_barrier()
25   /* step 2.2, copy phase */
26   for each record in {g_a, g_b}.data_array
27     replace_pointer(record)
28   rcu_barrier()
29   reclaim {old_group, g'_a, g'_b}'s memory
```

XIndex initializes *tmp_buf* to buffer insertions during compaction (Line 5). Afterwards, it creates a new group (Line 6) and merges the old group's *data_array* and *buf* into the new group's *data_array* (Line 7). In *new_group.data_array*, the value of each record is the reference to the corresponding record in either *group.data_array* or *group.buf*, and the *is_ptr* flag of each record is set to be *true*. During merging, XIndex skips the logically removed records. The old group's *tmp_buf* is reused as the new group's *buf* (Line 8). After training linear models (Line 9) and initializing the remaining metadata of the new group (Line 10), XIndex atomically replaces the old group with the new one by changing the group reference in root's *groups* (Line 12).

In the copy phase, XIndex replaces each reference in the new group's *data_array* with the latest record value (Line 16). The replacement is performed atomically with helper function *replace_pointer* (Algorithm 5). XIndex uses *rcu_barrier* (Line 17) to wait for each worker to process one request, so the old group will not be accessed after the barrier. Then it can safely reclaim the old group's memory resources (Line 18).

3.5 Structure update

XIndex adapts its structure to dynamic workloads at runtime (Section 5) with model split/merge, group split/merge, and root update operations.

Model split/merge. XIndex supports splitting and merging models within a group to improve lookup efficiency. For model split, XIndex first clones the group node. Both group nodes reference the same *data_array* and *buf*. Then, it increments the new node's *model_n*, evenly reassigns the group's data to each model, and retrains all models. At last, XIndex atomically updates the group reference in root's *groups* to the new group. For model merge, it essentially performs a reverse procedure of model split.

Group split. To avoid blocking other operations, XIndex uses two steps to split a group's data evenly into two groups (Algorithm 4).

In step 1, XIndex creates two logical groups. They share the data and delta index, but each has its own temporary delta index. As a result, both groups can buffer the insertion in their temporary delta indexes during the split. In detail, XIndex creates g'_a and g'_b (Line 3). They share the same *data_array* and *buf* with the old group (Line 4) but have different *pivot* keys (Line 5-6). XIndex links g'_b to g'_a's *next* field (Line 7) and replaces the old group with g'_a in root's *groups* (Line 10). Last, XIndex sets the *buf_frozen* flag (Line 11) and allocates *tmp_buf* for g'_a and g'_b (Line 13).

In step 2, XIndex physically divides the data into two groups. Similar to the compaction, this step has two phases. In the merge phase, XIndex first merges the old group's *data_array* and *buf* into *tmp_array* (Line 16). Then, it splits the *tmp_array* with the key of g'_b.*pivot*, and initializes two new groups, g_a and g_b accordingly (Line 17). It also reuses the *tmp_buf* of g'_a (g'_b) as the *buf* of g_a (g_b)[Line 18]. In the copy phase, for each group, the references in *data_array* are replaced with real values (Line 27). Last, XIndex links g_b at g_a.*next* (Line 21) and replaces g'_a with g_a in root's *groups* (Line 23).

Group merge. XIndex merges two consecutive groups' data into one new group to reduce the cost to lookup groups. Similar to the group split, data is merged in two phases. In the merge phase, both groups' *data_array*s and *buf*s are merged together while inserts are buffered in a single shared *tmp_buf*. In the copy phase, the merged references are replaced with concrete values. Finally, among the two consecutive groups in root's *groups*, the former is replaced with the new group and the latter is marked as *null*, which will be skipped by *get_group*. For brevity, we omit the pseudocode for group merge.

Root update. XIndex flattens root's *groups* to reduce pointer access cost, retrains, and conditionally adjusts the RMI model to improve prediction accuracy. For root update, XIndex creates a new root node with a flattened *groups* and

Algorithm 5: Helper functions

```
1  read_record(rec):
2  while true
3      ver ← rec.ver
4      removed, is_ptr, val ← rec.{removed, is_ptr, val}
5      if !rec.lock && rec.version = ver
6          if removed
7              val ← empty
8          else if is_ptr
9              val ← read_record(DEREF(val))
10         return val
11
12 update_record(rec, val):
13     lock(rec.lock)
14     succ ← false
15     if rec.is_ptr
16         succ ← update_record(DEREF(rec.val), val)
17     else if !rec.removed
18         rec.val ← val
19         succ ← true
20     rec.version ++
21     unlock(rec.lock)
22     return succ
23
24 replace_pointer(rec):
25     lock(rec.lock)
26     ref.val ← read_record(DEREF(rec.val))
27     if ref.val = empty
28         rec.removed ← true
29     rec.is_ptr ← false
30     rec.version ++
31     unlock(rec.lock)
```

retrains the RMI model. After a new root is initialized, XIndex replaces the global root pointer atomically.

4 Concurrency

XIndex achieves high scalability on the multicore platform using Two-Phase Compaction, along with classic techniques such as fine-grained locking [2, 20, 24], optimistic concurrency control [3, 4, 20], and RCU [21]. We first discuss the coordination between writers that ensures execution exclusiveness (Section 4.1), then discuss how readers can always fetch a consistent result with concurrent writers (Section 4.2). Afterwards, we discuss the interleaving with concurrent background operations (Section 4.3) and provide a proof sketch of the correctness condition (Section 4.4). The formal proof can be found in the extended version[4].

For brevity, we treat *remove* as a special *put*, which updates existing records' *removed* flag. We further omit group merge and root update in the discussion, as the reasoning resembles compaction's and group split's. In XIndex, compaction and structure updates are performed by dedicated background threads sharing no conflicts, thus avoiding concurrency issues due to their interleavings.

4.1 Writer-writer coordination

XIndex ensures that conflicting writers, *put*/*remove*s with the same key, will execute exclusively with the per-record lock in *data_array* and the concurrent delta index. All writers first try to update a matching record in *data_array* (Line 18, Algorithm 2), and the per-record lock is acquired to prevent interleaving with concurrent writers (Line 13, Algorithm 5). If updating *data_array* is not feasible, writers then operate on the delta index (Line 21, Algorithm 2), protected by a single read-write lock in the basic version. We improve its scalability with fine-grained concurrency control as an optimization (Section 6).

4.2 Writer-reader coordination

XIndex ensures readers can always fetch a consistent result in face of concurrent writers with locks and versions in *data_array* and the concurrent delta index. A *get* first tries to read a value from *data_array* (Line 6, Algorithm 2). It snapshots the version number before reading the value (Line 3, Algorithm 5). After the value is fetched (Line 4, Algorithm 5), *get* validates if the lock is being held (to detect concurrent writer) *and* if the current version number matches the snapshot (to detect inconsistent or stale result)[Line 5, Algorithm 5]. If the validation fails, the *get* repeats the procedure until a successful validation, so the result is consistent and the latest. If reading from *data_array* is not feasible, it then tries to read from the delta index (Line 8, Algorithm 2). The concurrent delta index with a single read-write lock ensures the fetched result is consistent.

4.3 Interleaving with background operations

With the presence of background operations, XIndex ensures that the effects of writers are preserved and can always be correctly observed by readers. Space constraints preclude a full discussion, but we mention two important conditions: 1) no successful *put* will be lost, and 2) no duplicate records (records with the same key) will be created[5].

To ensure no lost *put*, the key is to perform data movement in two phases, the merge phase and the copy phase, to preserve concurrent modifications. During the merge phase, *all* records in the old group's *data_array* and *buf* can be correctly referenced in the new group's *data_array*. This is because both the *data_array* and *buf* of the old group are read- and update-only, as the *buf_frozen* flag forbids insertions (Line 3, Algorithm 3 and Line 11, Algorithm 4). In the

[4] https://ipads.se.sjtu.edu.cn/_media/publications/xindex_extended.pdf

[5] Duplicate records do not directly violate correctness, as long as XIndex enforces a freshness ordering, $data\_array \geqslant buf \geqslant tmp\_buf$, where *data_array* has the latest version. However, doing so requires non-trivial efforts.

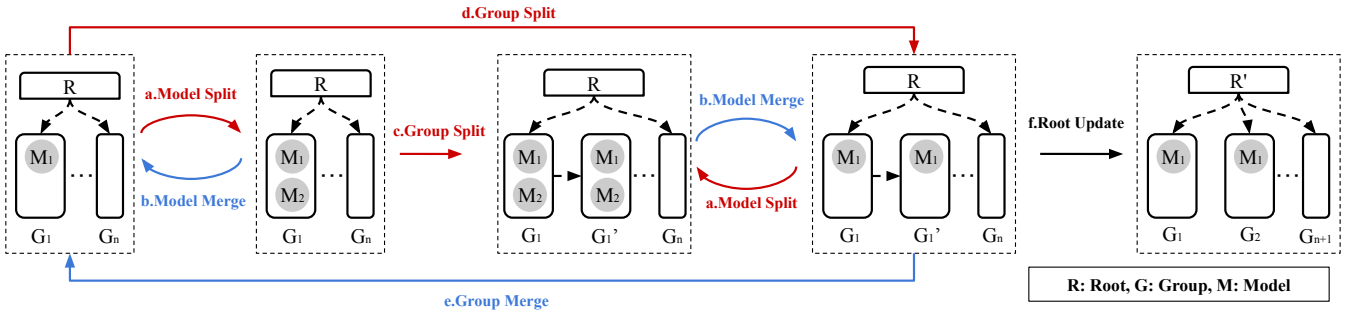

Figure 5. Dynamic adjustment procedure illustration.

copy phase, those references can be atomically replaced with latest values by *replace_pointer*, since it uses the per-record lock to coordinate with concurrent writers (Line 25, Algorithm 5). To ensure all concurrent writers use the same lock, XIndex places *rcu_barrier* before the copy phase (Line 13, Algorithm 3 and Line 24, Algorithm 4), which waits for each writer (and reader) to process one request. Therefore, later conflicting *put*s will not reference the old group's *data_array*. In addition, concurrent inserts are preserved in the shared temporary delta index (Line 8, Algorithm 3 and Line 18, Algorithm 4).

To ensure no duplicate records, XIndex avoids insertions to different delta indexes, namely *buf* and *tmp_buf*. XIndex only initializes *tmp_buf* until all writers observe a frozen *buf* using the *rcu_barrier* (Line 3-5, Algorithm 3 and Line 11-13, Algorithm 4). Therefore, whenever *tmp_buf* is used to serve requests, the *buf* is sure to be read- and update-only.

4.4 Proof sketch

The correctness condition of XIndex can be described as "a *get(k)* must observe the latest committed *put(k, v)*", namely, linearizability [13]. XIndex formally provides the correctness condition by ensuring the following inductive invariants. I_1) If there is a *put(k, v)* committed, then there is exactly one record with key k in XIndex; I_2) If there is a record with key k in XIndex, then its value equals the value of the last committed *put(k, v)*; and I_3) If there is a record with key k in XIndex when a *get* commits, then the *get* returns the value of the record.

For I_1, in addition to *no duplicate records* guarantee we discussed in Section 4.3, XIndex ensures that a new record will be created by *put* if no record currently exists yet. This is obvious as such *put* will invoke *insert_buffer* (Lines 21 and 27, Algorithm 2), and the concurrent delta index will handle the record creation. For I_2, the key is to ensure that no *put* will be lost, as we discussed in Section 4.3. For I_3, the key is to let *get* and *put* have the same lookup order (*data_array*→*buf*→*tmp_buf*). Since only the last place (*buf* when *tmp_buf* is *null*, otherwise *tmp_buf*) is insertable, a *get* returning an empty result only indicates that the *put* that creates the record has not yet finished. Therefore, a *get* can fetch the value correctly.

Operations	Trigger Condition
a. Model Split	error bound > e *and* # of models < m
b. Model Merge	error bound ≤ $e \times f$ *and* # of models > 1
c. Group Split	error bound > e *and* # of models = m
d. Group Split	buf > s
e. Group Merge	# of models = 1 *and* error bound ≤ $e \times f$ *and* buf ≤ $s \times f$
f. Root Update	when groups are created and/or removed

Table 2. Conditions for structure update operations.

5 Adjusting XIndex at Runtime

To reduce the performance variation, XIndex adjusts its structure according to runtime workload characteristics. The basic idea is to keep both error bound and delta index size small with structure update operations (model split/merge, group split/merge, and root update). Several background threads periodically check error bound and delta index size of each group and perform corresponding operations accordingly (Table 2 and Figure 5).

First, XIndex leverages model split to lower the error bound and model merge to reduce the cost of traversing a group's *models* array. Specifically, when a model's error bound is greater than the error bound threshold (e, specified by the user) *and* the model number of the corresponding group is less than the model number threshold (m, specified by the user), XIndex will do model split (Figure 5-a). When a model's error bound is less than or equal to $e \times f$ and the model number of the corresponding group is greater than one, XIndex will perform model merge (Figure 5-b). $f \in (0, 1)$ is a tolerance factor specified by the user.

When a model's error bound is greater than e, but the model number of the corresponding group equals m, XIndex will perform group split (Figure 5-c). Besides, if a group's delta index size is greater than the delta index size threshold (s, specified by the user), XIndex will also split the group (Figure 5-d). To reduce the cost of locating a group in the

root, XIndex performs group merge (Figure 5-e) when the following conditions hold — for two neighboring groups, 1) they both only have one model and the model's error bounds are less than or equal to $e \times f$; *and* 2) their delta index sizes are both less than or equal to $s \times f$.

XIndex periodically updates the root to reduce the access cost. Specifically, XIndex first checks all groups and perform model split/merge and group split/merge accordingly. If there is any group created or removed, XIndex then performs root update (Figure 5-f). During root update, if the average error bound is greater than e, XIndex will increase the number of 2nd stage models of root's 2-stage RMI[6]. If the average error bound is less than or equal to $e \times f$, XIndex reduces models.

6 Optimization

Scalable delta index. In the basic version, XIndex uses stx::Btree protected by a global read-write lock as its delta index. This limits the scalability when concurrent writers insert records to the same group. One possible solution is directly using Masstree as the delta index. However, Masstree provides unnecessary functionalities such as supporting variable length, multi-column values, and epoch-based memory reclamation. Thus, we implement a scalable delta index with a simplified design — each index node has a version to ensure that a *get* request can always fetch consistent content of the node and a lock to protect node update and split.

Sequential insertion. Sequential insertion is a common pattern in real-world workloads, such as periodically checkpointing. For such cases, the user can provide hints to XIndex so that XIndex can pre-allocate space to allow appending records directly to *data_array* and conditionally retrain models. Specifically, each group maintains an additional *capacity* field and a per-group lock. Only when XIndex detects the sequential insertion pattern, will it use the lock to coordinate concurrent sequential insertions. Otherwise, the lock is not used, so the scalability of XIndex is intact. Since many sequential insertion workloads have relatively static data distribution, XIndex only retrains models when the current model cannot generalize to newly appended data, namely, when the error bound exceeds the threshold.

7 Evaluation

We evaluate XIndex with complex workloads as well as micro-benchmarks of different characteristics and compare it against state-of-the-art systems.

Benchmarks. We develop a TPC-C (KV) benchmark by implementing TPC-C benchmark with only *get* and *put* operations, which is the same as [20]. We assign 8 distinct warehouses to each thread as their local warehouses for evaluation. Since XIndex does not support transactions, in order to avoid the impact of transaction abortions due to conflicts, we eliminate the conflicts by manipulating each thread to execute remote transactions on one of their own local warehouses. TPC-C (KV) benchmark can evaluate index systems under data and query distribution of real-world database workload while not requiring transaction support. YCSB includes six representative workloads (A-F) with different access patterns: update heavy (A), read mostly (B), read-only (C), read latest (D), short ranges (E) and, read-modify-write (F). For YCSB, besides its default data distribution, we also evaluate with a real-world dataset OpenStreetMap [6]. For microbenchmarks, we evaluate the performance under workloads with fixed read-write ratios (Section 7.2), under dynamic workloads (Section 7.3). We also analyze different factors that affect the performance in Section 7.4. All datasets used are listed in Table 3. The default dataset size is 200M unless otherwise noted, and each record has 8 bytes key and 8 bytes value by default.

Name	Description
linear	Linear dataset with added noises
normal	Normal distribution ($\mu = 0$, $\sigma = 1$), scaled to $[0, 1 \times 10^{12}]$
lognormal	Lognormal distribution ($\mu = 0$, $\sigma = 2$), scaled to $[0, 1 \times 10^{12}]$
osm	Longitude values of OpenStreetMap locations scaled to $[0, 3.6 \times 10^{9}]$

Table 3. Datasets. For the *linear* dataset, we first generate keys $\{i \times A \mid i = 1, 2, \dots\}$, then add a uniform random bias ranging in $[-A/2, A/2]$ for each key, where $A = 1 \times 10^{14}/\textit{dataset size}$. All keys are integers.

Counterparts. stx::Btree [1] is an efficient, but thread-unsafe B-tree implementation. Masstree [20] is a concurrent index structure that hybrids B-tree and Trie. When the key size is 8 bytes, Masstree can be regarded as a scalable concurrent B-tree. Wormhole [24] is a concurrent hybrid index structure that replaces B-tree's inner nodes with a hash-table encoded Trie. The learned index [15] is the original learned index. "learned+Δ" is the learned index attached with a Masstree as delta index, which buffers all writes.

Configuration & Testbed. We implement XIndex in C++, and configure 1 out 12 threads as dedicated background threads. Background thread(s) sleeps one second after it has checked all groups and root to perform compaction and structure update accordingly. Throughout all experiments, the error bound threshold (e) is 32, the delta index size threshold (s) is 256, the tolerance factor (f) is $\frac{1}{4}$, and the model number threshold (m) is 4. For the learned index, we test different configurations and pick the best one — 250k models in the 2nd stage.[7] For "learned+Δ", we use the same background threads as XIndex for compaction. For stx::Btree, Masstree, and Wormhole, we directly run their source code with the default setting. For each experiment, we first warmup all the

[6]The number of models stops increasing when it reaches a given limit.

[7]The candidates' model number ranges from 50k to 500k (step is 50k).

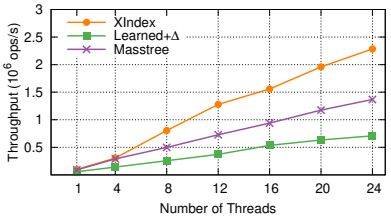

Figure 6. TPC-C (KV) throughput.

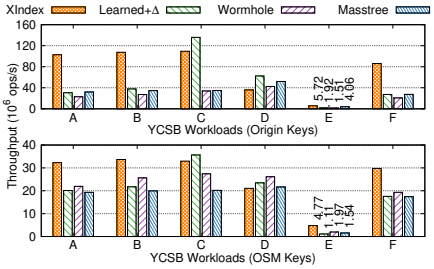

Figure 7. YCSB throughput.

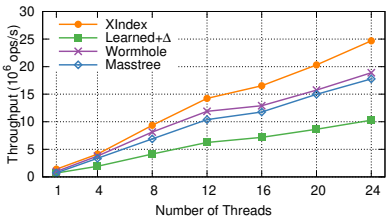

Figure 8. Read-write throughput.

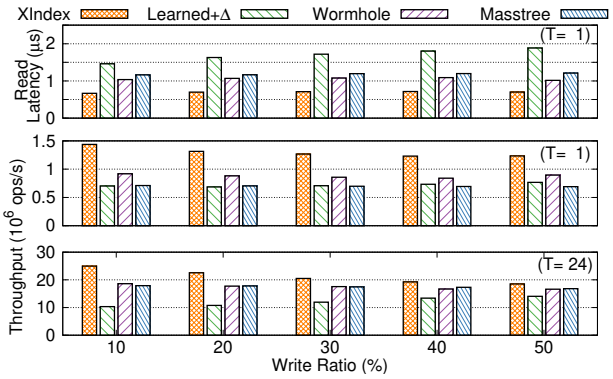

Figure 9. Read-write throughput and read latency with the *normal* dataset. T indicates the number of threads.

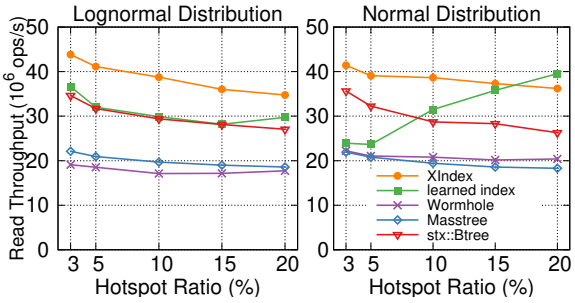

Figure 10. 24-thread read throughput in skewed query distribution.

systems and present steady-state results. The experiments run on a server with two Intel Xeon E5-2650 v4 CPUs, and each CPU has 12 cores. The hyperthreading is disabled during evaluation.

7.1 Performance Overview

TPC-C (KV). Figure 6 shows the performance comparison with different numbers of threads. Wormhole is excluded because the Wormhole implementation we use does not support multiple tables, while TPC-C (KV) requires them. XIndex outperforms Masstree by up to 67% with 24 threads. First, the data generated in TPC-C (KV) are multidimensional linear mappings. Therefore, the learned models can obtain a good approximation. Second, 63% of the write operations update existing records. Thus they can be efficiently executed in-place. Lastly, 34% of the write operations perform sequential insertion, which can be improved by the optimization (Section 6).

YCSB. We use both the default data distribution as well as the *osm* dataset, and 24 threads for the experiment. As shown in Figure 7, for workload A, B, E, and F, XIndex demonstrates superior performance advantage. This is because these workloads are read- and update-intensive. For workload C, which is read-only, XIndex is worse than "learned+Δ" by 19% because XIndex has model computation cost both in the root and groups. For workload D, XIndex performance is worse than the other systems by up to 30%. The reason is that workload D tends to read recently inserted records that might not have been compacted, which brings overheads for read operations. With the *osm* dataset, the results are similar. However, because of the complex real-world data distribution, the advantage of XIndex is reduced.

7.2 Performance with writes

To further evaluate the performance of writes, we configure workloads with different read-write ratios. The ratio among different type of writes are constant (1:1:2 for insert, remove, and update) to keep the dataset size stable.

Scalability. Figure 8 shows the scalability with 10% writes using the *normal* dataset. Overall, XIndex achieves the best performance among all systems. With 24 threads, XIndex scales to 17.6× of its single-thread performance, which is 30% higher than Wormhole. "learned+Δ" has the worst performance because of its inefficient compaction, which severely degrades the read performance.

Varying write ratios. Figure 9 shows both throughput and latency with different write ratios with a single thread and 24 threads. XIndex has the best performance for all the listed write ratios, though the advantage tends to diminish with larger write ratios. For latency, XIndex achieves the lowest latency as most requests (80%) can be served without accessing delta indexes.

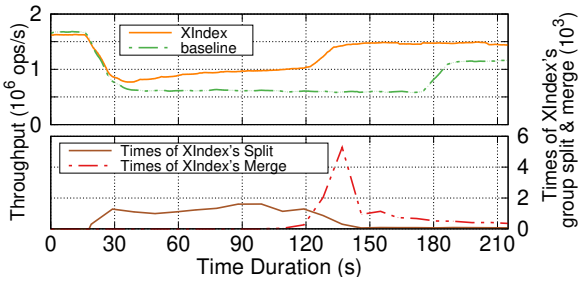

Figure 11. Read-write throughput and group split/merge frequency under dynamic workload.

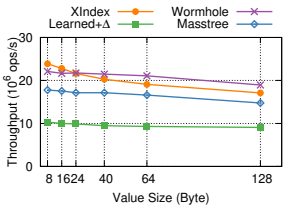

Figure 12. Read-write throughput of various value sizes.

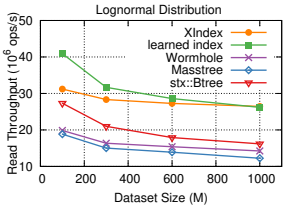

Figure 13. Read throughput of various dataset sizes.

7.3 Performance of dynamic workload

Query distribution. For dynamic workload, we first evaluate the performance when the query distribution is non-uniform. To control the skewness, we make the workload's 90% queries access hotspot of different sizes (the hotspot ratio). All hotspots are ranges that start from the same key but end differently. The smaller the hotspot is, the more skewed the query distribution is. Figure 10 shows the throughput with different skewness levels under the *normal* and *lognormal* datasets. All systems except for the learned index see a performance improvement when the skewness level rises since the skewed query distribution brings a more friendly memory access locality. However, due to the learned index's large error bound in the hotspot, the learned index can perform even worse than stx::Btree and Wormhole. For the *lognormal* dataset, when the hotspot ratio decreases under 5%, the increase of hot models' error bounds slows down, thus we can observe a slight performance improvement of the learned index due to improved locality.

Data distribution. We then evaluate XIndex under the workload, whose data distribution and read-write ratio will be dynamically changed at runtime. As a baseline, we also run XIndex without background group split and merge. Figure 11 shows the throughput and the number of XIndex's group split/merge under this workload using one worker thread and one background thread.

Both XIndex and baseline are initialized with 50M *normal* dataset, and the initial read-write ratio is 90:10. In the beginning, they have similar performance. At the 20th second, the write ratio becomes 100% (half inserts and half removes), and we remove all existing keys and insert new keys with 50M *linear* dataset. From this point, both throughput of XIndex and baseline begin to degrade due to the increase of write ratio and the dramatical changes of data distribution. While for XIndex, background threads begin to do group split to reduce the error of the group and delta index size, so we can see the throughput starts to increase at the 30th second.

At the 120th second, XIndex finishes dataset shifting, and at the 170th second, baseline ends shifting. Afterwards, the read-write ratio is 90:10, and keys follow the *linear* distribution. XIndex's background thread detects that both the delta index size and the error bound of groups are small after the shifting, so it invokes lots of group merge operations to reduce the number of groups. Overall, XIndex shows up to 140% performance improvement during and after the change of workload.

7.4 Other factors

Value size. We evalute the performance of XIndex with different value sizes under the *normal* dataset with 24 threads. The read-write ratio is 90:10, and the value contains 8-128 random generated bytes. The result is shown in Figure 12. With the increase of value size, the performance of all systems is reduced due to the large memory consumption. Nevertheless, XIndex has the largest performance drop. This is because the overhead of data copying during compaction (128B's overhead is 13.5× larger than 8B's).

Dataset size. Figure 13 shows the performance of XIndex with different dataset sizes under the *lognormal* dataset using 24 threads. As dataset size increases, both the learned index and XIndex show a large performance advantage over other systems. However, the performance of the learned index degrades significantly because its error grows as the size increases. In contrast, XIndex adjusts its structure to maintain small model error bounds. Therefore, for large dataset sizes, XIndex can achieve similar performance with the learned index.

8 Discussion

Inline values v.s. separated values. XIndex contiguously stores keys and values in *data_array* (inline values). Another popular approach [19, 22] is to store values in separate storage and only the pointers along with keys (separated values). For small values, our approach has an advantage in reducing DRAM accesses, since both the key and value can reside in one cache line. For large values, separating value can reduce compaction cost, since only pointers will be copied. However, both approaches require Two-Phase Compaction as the record's metadata, such as *removed* and *lock*, should be inline and might be changed during compaction. As separating the metadata may incur high compaction cost, since it needs

additional memory accesses for each record to check if the value has been logically removed.

Limitations. First, when the dataset is small, other index structures (e.g., stx::Btree) can outperform XIndex for the model computation cost and the cost to traverse the two-layer structure. Second, with long keys, the overhead of model training and inference will increase significantly, which can affect the efficiency of XIndex. For 64-byte keys, there can be up to 50% performance degradation compared with 8-byte keys. We leave the design of a more flexible structure for variable-scale workloads and reducing the cost of long keys as future work.

9 Related Works

There have been works that extend or build systems upon the learned index. Many works extend the learned index to support writes. ALEX [9] achieves this by reserving slots in a sorted array for inserting new data. It allocates a new array and retrains the model synchronously when there are not enough reserved slots. AIDEL [18] handles insertions by attaching a sorted list for each record in the sorted array. When the list is too long, it copies the data into a new sorted array and retrains the model synchronously. Both data structures are not designed for concurrent scenarios and operations are blocked during rearranging data and retraining models. PGM-index [11] extends the learned index to optimize the structure with respect to given space-time trade-offs. It recursively constructs the index structure and provides an optimal number of linear models. Comparing with PGM-index, XIndex adjusts its structure at runtime, does not assume an already known query distribution. SageDB [14] is a database that proposes to leverage the learned index for data indexing as well as for speeding up sorting and join. FITing-Tree [12] indexes data with a hybrid of B-tree and piece-wise linear function, making it a variant of the learned index. It supports insertions and provides strict error guarantees. Comparing with FITing-Tree, XIndex is a fully-fledged concurrent index structure and adapts its structure to both data and query distribution at runtime.

Classic concurrency techniques have long been used in concurrent data structures. Masstree [20] is a trie-like concatenation of B-trees and uses fine-grained locking and optimistic concurrency control to achieve high performance under multi-core scenarios. It carefully crafts its protocol to improve efficiency for reader-writer coordination. Wormhole [24] is a variant of B-tree that replaces B-tree's inner nodes with a hash-table encoded Trie. It uses per-node read-write locks to coordinate accesses to leaf nodes and uses a combination of locking and the RCU mechanism to perform internal node updates. Bonsai tree [5] is a concurrent balanced tree. It allows reads to proceed without locks in parallel with writes by using RCU mechanism, though a single write lock is still required to coordinate writes. HOT [2] is a trie-based index structure which aiming to reduce the height of the trie. It uses per-node locks to coordinate writes and uses copy-on-write (COW) to allow reads to proceed with no synchronization overhead. The Bw-Tree [8, 17] is a completely lock-free B-tree and achieves its lock-freedom via COW and compare-and-swap (CAS) techniques. Building upon existing works, XIndex leverages fine-grained locking and optimistic concurrency control to coordinate to individual records and uses the RCU mechanism to eliminate interference with queries and writes due to background compaction and structures updates.

Dynamic data and query distributions are common in real-world workloads. While XIndex strikes to reduce performance variation between records, many works distinguish hot and cold data and further optimize the performance for hot data. Hybrid index structure [25] uses different storage schemes for hot keys and cold keys. Storage systems such as H-Store [7], COLT [23] are designed to detect the hotness and manage data accordingly in a self-tuning process.

10 Conclusion

In this paper, we introduced XIndex, a concurrent and flexible index structure based on the learned index. XIndex achieves high performance on the multicore platform via a combination of the innovative Two-Phase Compaction and a number of classical concurrency techniques. Furthermore, it can dynamically adjust its structure according to the runtime workloads to maintain competitive performance. Extensive evaluations demonstrate that XIndex can have a performance advantage by up to 3.2× and 4.4×, compared with Masstree and Wormhole, respectively. XIndex is publicly available at https://ipads.se.sjtu.edu.cn:1312/opensource/xindex.git.

Acknowledgments

We thank Jinyang Li, Yueying Li, and 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, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and a grant from Huawei Technologies. Zhaoguo Wang (zhaoguowang@sjtu.edu.cn) is the corresponding author.

References

[1] Timo Bingmann. 2008. STX B+ tree C++ template classes.
[2] 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*. ACM, 521–534.
[3] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *ACM Sigplan Notices*, Vol. 45. ACM, 257–268.
[4] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, Vol. 1. 181–190.

[5] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable address spaces using RCU balanced trees. *ACM SIGPLAN Notices* 47, 4 (2012), 199–210.

[6] OpenStreetMap contributors. [n. d.]. OpenStreetMap database. https://aws.amazon.com/public-datasets/osm. Accessed: 2019-4-24.

[7] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-caching: A new approach to database management system architecture. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1942–1953.

[8] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 1243–1254.

[9] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David Lomet. 2019. ALEX: An Updatable Adaptive Learned Index. *arXiv preprint arXiv:1905.08898* (2019).

[10] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through siberia: Managing cold data in a memory-optimized database. *Proceedings of the VLDB Endowment* 7, 11 (2014), 931–942.

[11] Paolo Ferragina and Giorgio Vinciguerra. 2019. The PGM-index: a multicriteria, compressed and learned approach to data indexing. arXiv:cs.DS/1910.06169 https://arxiv.org/abs/1910.06169

[12] 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*. ACM, 1189–1206.

[13] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[14] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system.

[15] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 489–504.

[16] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 26–37.

[17] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 302–313.

[18] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. 2019. A Scalable Learned Index Scheme in Storage Systems. *arXiv preprint arXiv:1905.06256* (2019).

[19] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 5.

[20] 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*. ACM, 183–196.

[21] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2001. Read-copy update. In *AUUG Conference Proceedings*. AUUG, Inc., 175.

[22] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1994. AlphaSort: A RISC machine sort. In *ACM SIGMOD Record*, Vol. 23. ACM, 233–242.

[23] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2007. On-line index selection for shifting workloads. In *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop*. IEEE Computer Society, 459–468.

[24] 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*. ACM, 18.

[25] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1567–1581.