XIndex: A Scalable Learned Index for Multicore Data Storage

Chuzhe Tang†, Youyun Wang†, Zhiyuan Dong†, Gansen Hu†
Zhaoguo Wang†‡, Minjie Wang⋄, Haibo Chen†‡
zhaoguowang@sjtu.edu.cn
‡ Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University
† Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
⋄ Amazon

Abstract

We present XIndex, a concurrent ordered index designed for fast queries. Similar with 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 store. On a 24-core machine, XIndex achieves up to 3.3× and 4.4× performance improvement comparing with Masstree and Wormhole respectively. XIndex 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. More specifically, it first trains the model with each key and its position, then uses the model to predict the data position with the given key.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374547

To deliver high lookup performance, the learned index uses simple learned models such as linear regression or simple layer neural networks. To enable the simplity on learned models, it adds extra requirements on the data layout. For instance, to learn ordered index, it requires the data to be both ordered and contiguous. As a result, the data distribution can be simulated with linear models, and the performance is 1.5-3× better than B-tree.

However, the current study of the learned index is still preliminary, and lacks practicability in a broad class of read-world scenarios because of two limitations: first, it does not support of any modifications including inserts, updates or deletes; Second, it assumes the workload has a relative static query distribution[2]. More specifically, it requires all data are uniformly accessed. But, making the learned index practical for dynamic workloads with updates is not a 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 data to be contiguous for using simple models. Thus, it needs to construct new dataset and retrain the model to handle every update events. Although, there are proposals [9, 18] of handling updates for the learned index more efficiently. But none of them is able to ensure correctness in face of concurrent operations [**ZG: need to discuss it in detail in related work section**]. 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 frequenty queried than others [7, 10, 16, 25]. As a result, when the model that indexes those hot keys has large errors, queries can incur high overheads (sec:issues).

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

[2]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.

Moreover, it is designed to adapt its structure deterministically at runtime and decouple its efficiency from the 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] and optimistic concurrency control [3, 4, 20]).

Deterministic data structure adjustment according to runtime workload characteristics. Unlike B-tree whose structure is decided by the fanout, XIndex adapt its structure to runtime workload characteristics through structure update operations, such as group split and group merge. Users can configure the expected behaviours through parameters such as error bound threshold (*e*) and delta index size threshold (*s*).

Implementation and evaluation with both 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, Wormhole respectively.

The rest of the paper is organized as follows. We first describe the background and motivation in Section 2. Section 3 gives the design of XIndex. Afterwards, we presents the concurrent support (Section 4), the runtime structure adjustment strategy (Section 5) and optimizations (Section 6). Section 7 shows evaluation results. We discuss alternative design choice and limitations of current design in Section 8 and summarize related works in Section 9. 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[3], 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 of prediction errors, the learned index stores the maximal and minimal prediction errors of the model. After training the model, the errors are calculated by taking the difference between the predicted and the actual positions of each key and taking 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.

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 accuarcy and therefore reduce the error bound, the paper proposes a *staged model architecture*, termed Recursive Model Indexes (RMI). RMI contains multiple stages of models, which resembles 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.

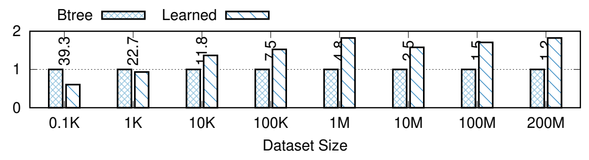


Figure 1. Learned index throughput **normalized to** stx::Btree. The numbers indicate the absolute throughput of stx::Btree in Mops/s.

The evaluation of the paper [15] shows that the learned index can achieve better (1.5-3×) performance than B-tree across several datasets. We further evaluate the learned index with different dataset size. We find that, the learned index has better performance than stx::Btree with large datasets, but gets outperformed by stx::Btree with small dataset. Figure 1 shows the performance of the learned index under different dataset sizes with *normal* distribution. The learned index is configured with a 2-stage RMI architecture and 2nd stage has 10K linear models. We compare the learned index with stx::Btree [1] and configure stx::Btree with its default fanout (16). With small datasets (less than 10K), the learned index's performance is limited by the model computation cost. Most of the learned index's query time (47% of 42ns) is spent on model computation when the dataset size is 0.1K. While

[3]We refer to this array as *sorted arrays*.

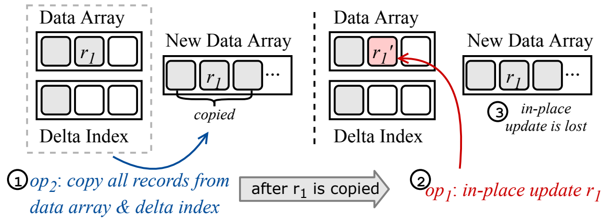


Figure 2. Concurrency issue in face of concurrent operations.

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

Table 1. stx::Btree's and the learned index's throughput(Mps) & the learned index's error bound in uniform and skewed query workloads of OSM dataset. The last row gives the average error bound weighted by the models' access frequency.

stx::Btree has 2 layers and only needs 25ns to finish one query. When the dataset size increases, the learned index shows large performance advantages over stx::Btree. The reason is that, the increase in the overhead of binary search for the learned index is much slower than that of node traversal for stx::Btree. Meanwhile, the model computation cost of the learned index is constant. For instance, with dataset size increasing from 1M to 10M, the error bound increases from 4.7 to 6.6 and the time of binary search in the learned index has a growth of 38% (from 68ns to 94ns). However, the query time of stx::Btree only has a increase of 92% (from 207ns to 399ns).

2.2 The issues

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

First, the learned index does not provide an efficient method to handle writes, especially under concurrent scenarios. Based on 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 both merging the data into a new sorted array and retrain the models. Though straightforward, this method suffers from severe slow down for queries. One reason is that each request has to first go through the delta index(es) before looking up the the learned index. Considering building a 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 30s to compact a delta index of 100K records with a the learned index with 200M records.

A possible improvement based on above method is employing in-place updates with nonblocking compaction scheme. We can perform writes to existing records in-place, and only inserting new records to the delta index. Thus, a query only looks up 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, simply using these two methods together may arise correctness issue — the effect of in-place writes might be lost due to the data race with background compaction. Let's consider this example (Figure 2) where operation op_1 in-place updates record r_1 and operation op_2 concurrently merges the delta index with the learned index into a new sorted record 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 in-place updates r_1 in the old array; 3) op_2 finishes the compaction, update 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 bound of different model varies. 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 *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 key 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 The learned index have much higher average error bounds on the frequently accessed records under workload "Skewed 1" and "Skewed 3", which hinders the query performance. The underlying cause is that the learned index only tries to minimize the error of each model individually, lacking the consideration for accuracy differences between models. 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

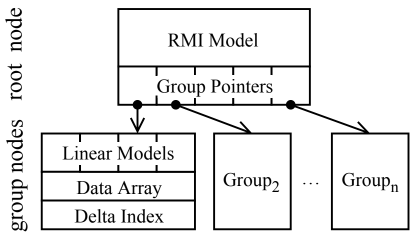


Figure 3. The architecture of XIndex.

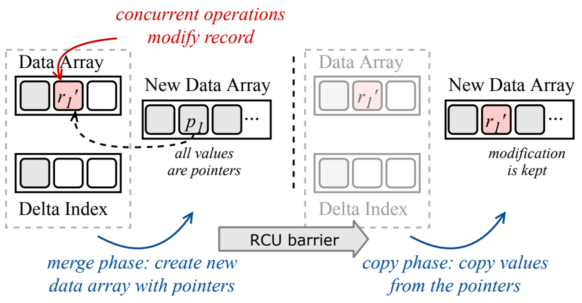


Figure 4. Two-phase compaction prevents concurrent operations being lost.

in the bottom layer. The data is divided into groups by range partitioning. The *root* uses a learned RMI model to index the groups. Each *group* node uses a learned linear model to index its belonged data. For writes, XIndex performs in-place updates on existing records, and associates each group with a delta index to buffer insertions.

XIndex first introduces a new compaction scheme, Two-Phase Compaction (Section 3.4), to compact the new data conditionally. The compaction is performed at background and does not block any concurrent operations. The compaction has two phases: *merge phase* and *copy phase*. In merge phase, XIndex merges current data array and delta index into a new data array. Instead of directly copying the data, XIndex maintains the data references in the new data array. Each reference points to the records being compacted, residing in either the old data array or the delta index. After ensuing there will be 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 above exmple (Figure 2) with Two-Phase Compaction in Figure 4, after the merge phase, the new data array contains the pointer (e.g., p_1) of each record (e.g., r_1). If there exist a concurrent write which updates r_1 to be r_1', it can safely proceed as it is already referenced by the new data array. After a RCU barrier, no thread will access the old data array anymore. p_1 is replaced with r_1' in the copy phase.

XIndex is able to adjust its structure according to runtime workload characteristics. At runtime, if some group incurs

Algorithm 1: Structures

```
struct root_t:
    rmi_t rmi;
    uint32_t group_n;
    key_t pivots[];
    group_t* groups[];

struct record_t:
    key_t key;
    val_t val;
    uint64_t // composite 8B
        is_ptr : 1, removed : 1
        lock : 1, version : 61;
struct group_t:
    key_t pivot;
    bool_t buf_frozen;
    uint16_t model_n;
    uint32_t array_size;
    model_t
  models[MAX_MODEL_N];
    record_t data_array[];
    buffer_t* buf;
    buffer_t* tmp_buf;
    group_t* next;
```

high prediction error, XIndex adds more linear models in that group by "*model split*" to increase the inference accuracy. If a group has too many models or its delta index is too large, XIndex splits the group — replacing the group with two new groups each containing half data of the old group. XIndex also performs *model merge* and *group merge*, if the merging does not affect the prediction accuracy. Furhtermore, if there are too many groups, XIndex retrains the RMI model of the root node, and may adjust the model's structure to improve the accuracy.

3.2 Layout

XIndex maintains three basic structures of *record_t*, *root_t* and *group_t* according to the root node, the group node and the record (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* is concurrency control flags which ensure concurrent operations are exclusively executed.

The *root_t* contains the group's information and an RMI model. The group information includes each group's smallest key (*pivots_n*) and address (*groups*), and the total number of groups (*groups_n*). The RMI model (*rmi*) is used to predict the group with a given key. It is trained with *pivots* and *groups*. In current design, XIndex uses two-stage RMI architecture solely 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, models and delta index. For the data, all records indexed by the group is continuously stored in *data_array*. Each group uses at least one linear models to index the record in *data_array*, and the models are maintained in *models*. The *mode_t* includes the parameters of the linear models, and the data range belonged to each model. The *buf* is the delta index which buffers all insertions. During compaction, *buf* is frozen and (*buf_frozen*

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)
```

is set to be true. The *tmp_buf* serves as a temporary delta index which buffers all insertions termporarily during the compaction. For optimization purpose, the *group_t* maintains its smallest key in a seperate variable, *pivot*.

3.3 Basic operations

XIndex provides basic index interfaces — *get*, *put*, *remove* and *scan* (Algorithm 2). All the operations share the same logic for searching the correct group and lookup the requested key in *data_array*, after then, the procedures diverge.

XIndex first uses the root to find the group the requested key belongs to (Line 2, 15) assisted by the root's RMI model. The models only provide predicted positions and we need to furthur binary search within a error-bounded range to actually find the correct position (Section 2.1). In the case when a group's *next* pointer is not *null*, which indicates previously a group had been split into two groups, XIndex needs to follow the pointer to find the correct group. When looking up the key within a group's *data_array* (Line 3), XIndex also leverages the corresponding learned model.

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

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
```

with *read_record* helper function (Line 6). An *empty* result indicates a logically removed record. In this case, *get* proceed to search *buf* (Line 7-8), and 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 in-place update/remove, then it proceeds to operate on *buf* and optionally *tmp_buf*. Unlike *get* operations, *put* requests only access *buf* when *frozen_buf* flag is not set. For *scan*, XIndex first locates the smallest record that is ≥ requested key, and then consistently reads *n* consecutive records.

We elaborate 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* a new sorted array in which 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 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). Furthermore, the values in the new group's *data_array* are references to records in

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    /* step 2.1, merge phase */
12    {g_a′, g_b′}.buf_frozen ← true
13    rcu_barrier()
14    {g_a′, g_b′}.tmp_buf ← allocate new delta indexes
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_array
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
```

either *data_array* or *buf* of the old group. The *is_ptr* flags are set to *true*. During merging, XIndex skips the logically removed records. The old group's *tmp_buf* is re-used 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), we atomically replace the old group with the new one by changing the group references 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 *replace_pointer* helper function (Section 4). 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 adjusts its structure at runtime with model split/merge, group split/merge and root update operations. XIndex leverages these operations to adapt to dynamic workloads (Section 5).

Model split/merge updates a group with a increased or decreased number of models to index *data_array*. For model split, XIndex first creates a new group with all the fields copied from the orginal group except *model_n* and *models*. Then, it increments *model_n* by one, reassigns data to each model and retrains them. 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 replaces a group with two new groups each containing half data of the old group (Algorithm 4). There are two steps in group split. In step 1, XIndex creates two intermediate groups, *g_a′* and *g_b′* (Line 3), to logically split the old group. *g_a′* and *g_b′* share the same *data_array* and *buf* with the old group (Line 4). However, they have different *pivot* keys (Line 5-6) and *g_b′* is linked to *g_a′*'s *next* field (Line 7). Therefore, requests can be directed to and served by different groups accordingly though the data is shared. At the end of step 1, XIndex replaces the old group with *g_a′* (Line 10). In step 2, XIndex creates two final groups, *g_a* and *g_b*, to physically splits the shared data. Since naively copying the data can easily cause inconsistency (Section 2.2), XIndex employ a procedure similar to compaction which includes a merge phase and a copy phase. In the merge phase, XIndex merges old group's *data_array* and *buf* into a temporary array of references (Line 16) while buffers inserts in *tmp_buf*s (Line 14). Then XIndex splits the temporary array by *g_b′.pivot* and attach the results to *g_a* and *g_b* (Line 17). In the copy phase, the references are replaced with concrete values (Line 27). Finally, XIndex replaces *g_a′* with *g_a* whose *next* points to *g_b* (Line 10). The new groups are linked to batch changes to the root's *groups* array, so that XIndex can reduce the retraining frequency of root's model.

Group merge replaces two consecutive groups with one new groups containing both groups' data. Similar to group split, group merge includes a merge phase and a copy phase. 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. For brevity, we omit the pseudocode for group merge.

Root update flattens root's *groups* to reduce pointer access cost, retrains and conditionally adjusts the RMI model to improve prediction error. During root update, XIndex creates a new root node with a flattened *groups* and retrains the RMI model. After a new root is initialzied, XIndex replaces the global root pointer atomically and reclaims the memory of the old root with *rcu_barrier*.

4 Concurrency

XIndex achieves high scalability on multicore platform using Two-Phase Compaction, along with classic techniques such

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)
```

as fine-grained locking [2, 20, 24], optimistic concurrency control [3, 4, 20] and RCU [21].

The correctness condition of XIndex can be described as "a *get(k)* must observe the latest committed[4] *put(k, v)*", namely, linearizability [13]. A *get* should always returns a correct value regardless of concurrent *put*s. When there is a concurrent *put*, the *get* can returns either the old value or the new value, indicating the *put* commits after or before the *get* respectively. We first discuss the coordination between readers and writers without concurrent background operations, then discuss interleaving with concurrent background operations, and finally provide a proof sketch of the correctness condition. The formal proof can be found in the extended version [XXX].

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]A *put* commits when its effect becomes visible to others.

4.1 Writer-writer coordination

XIndex ensures that conflicting writers, *put*/*remove*s with the same key, are atomic 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), and the concurrent data structure coordinates concurrent writers to achieve atomicity. In the basic version, we use stx::Btree protected by a single read-write lock as delta index. We improve its scalability with fine-grained concurrency control as an optimization (Section 6).

4.2 Writer-reader coordination

XIndex ensures reader atomicity in face of concurrent writers with locks and versions in *data_array* and the concurrent delta index. A *get* first try to read a value from *data_array* (Line 6, Algorithm 2). It first snapshots the version number before reading the value (Line 3-4, Algorithm 5). After the value is fetched, *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 latest. If reading from *data_array* is not feasible, it then try to read from the delta index (Line 8, Algorithm 2). The concurrent delta index ensures the atomicity of concurrent operations.

4.3 Interleving 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, the *all* records in old group's *data_array* and *buf* can be correctly referenced in new group's *data_array* (Line 7, Algorithm 3 and Line 16, Algorithm 4). This is because insertions to old group's *buf* is forbitted by the *buf_frozen* flag (Line 3, Algorithm 3 and Line 12, Algorithm 4). In the copy phase, those

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

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 uses the same lock, XIndex places *rcu_barrier* before copy phase (Line 13, Algorithm 3 and Line 24, Algorithm 4), which waits for all writers (and readers) to each process one request. Therefore, later conflicting *put*s are sure to observe the new group and use the same lock in new 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 12-14, 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

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) The 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) The 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 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 latter 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.

5 Adjusting XIndex at runtime

To reduce the performance variation, XIndex adjusts its structure according to runtime workload charactersitics. 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 theads periodically checks error bound and delta index size of each group and perform corresponding operations accordingly (Figure 5).

First, XIndex leverages model split to lower the error bound and model merge to reduce the cost of finding the right model in a group. Specifically, when a group's model error bound $> e$ (error bound threshold) *and* its number of models $< m$ (model number threshold), XIndex will do model split. When a group's model error bound $\leq e \times f$ and the number of models > 1, XIndex will perform model merge. e, m and $f \in (0, 1)$ are parameters specified by users.

To keep the delta index size small and further reduce the error bound, XIndex perform group split; meanwhile, group merge is used to reduce the cost of locating groups. Specifically, if a group's delta index size $> s$ (delta index size threashold), XIndex will split the group. When a group's model error bound $> e$, but the number of models $= m$, XIndex will also perform group split. Group merge is performed when both the following two conditions hold: 1) two neighboring groups' error bounds $\leq e \times f$; *and* 2) their delta index sizes $\leq s \times f$. s is a parameter specified by users.

XIndex periodically updates the root to reduce the access cost. Newly created groups are linked to *next* pointers of the previous group, which increases the overhead of locating groups. XIndex first creates a new root with a flattened *groups* array and then retrains its 2-stage RMI model. During root update, if the error bound $> e$, XIndex will increases the number of 2nd stage models[6]. If the error bound $\leq e \times f$, 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 delta index. However, Masstree provides unnecessary functionalities such as supporting variable length, multi-colunm values and echo-based memory reclaimation. 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, 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 thus the scalability of XIndex is intact. Since many sequential insertion workloads have relatively static data distribution, XIndex only retrains models when current model cannot generalize to newly appended data, namely the error bound exceeds the threshold.

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

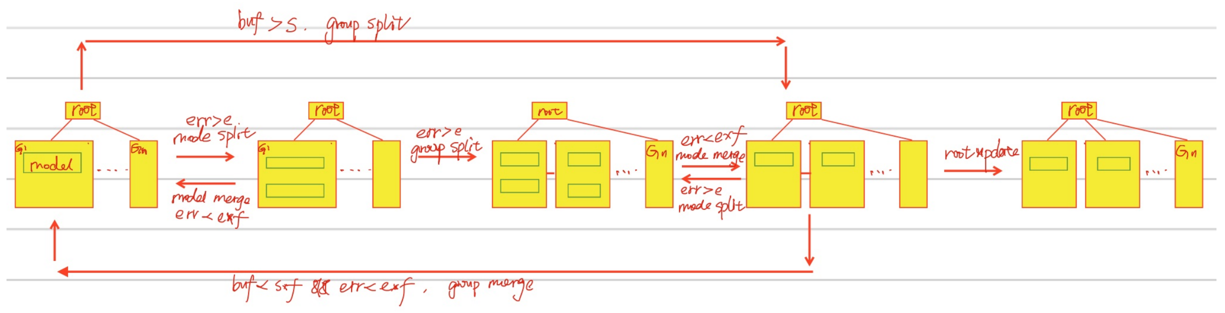


Figure 5. Dynamic adjustment procedure illustration

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 with only *get* and *put* operations, which is the same as [20]. We assign 8 distinct warehouses to each thread as their local warehouse 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 micro-benchmarks, 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 2. The default dataset size is 200M unless otherwise noted, and each record has 8 bytes key and 8 bytes value by default.

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. For evaluation, we set the error bound threshold (e) to be 32 and the delta index size threshold (s) to be 256. The f parameter is set to $\frac{1}{4}$ throughout all the experiments.

Name	Description
linear	Linear dataset with added noises
normal	Normal distribution ($\mu = 0$, $\sigma = 1$), scaled to $[0, 10^{12}]$
lognormal	Lognormal distribution ($\mu = 0$, $\sigma = 2$), scaled to $[0, 10^{12}]$
osm	Longitude values of OpenStreetMap locations scaled to [0, 3.6e9]

Table 2. Datasets. For *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 $A = 1e14/\text{dataset size}$. All keys are integers.

For the learned index, we test different configurations and picked 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 systems and present steady state results. The experiment runs on a server with two Intel Xeon E5-2650 v4 CPU, 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 used 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 a multidimensional linear mappings. Therefore, the learned models could obtain a good approximation. Second, 64% 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 *osm* dataset, and 24 threads for the experiment. As shown in

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

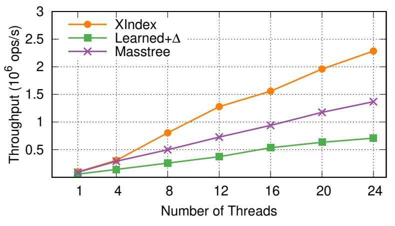


Figure 6. TPC-C (KV) throughput.

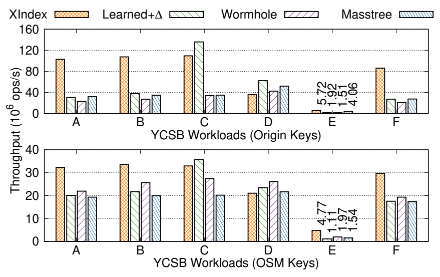


Figure 7. YCSB throughput.

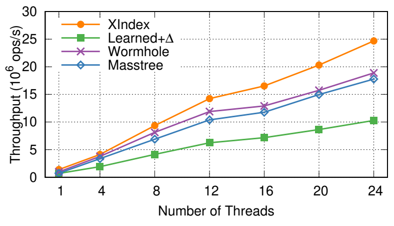


Figure 8. Read-write throughput.

Figure 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 the learned index by 19% because of the design to support writes. 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 *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 the workloads with different read-write ratio. 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 *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.

Performance with different write ratio. 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 the delta indexes.

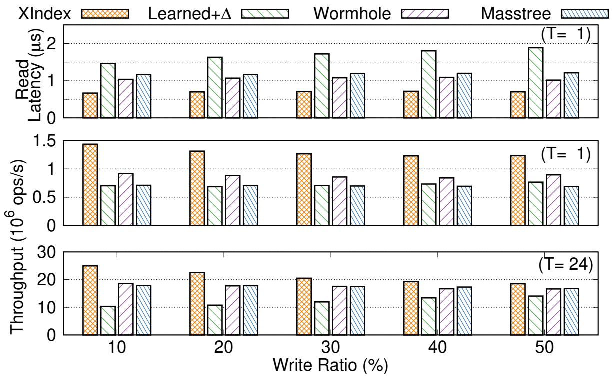


Figure 9. Read-write throughput and read latency with normal distribution. (T means the number of threads)

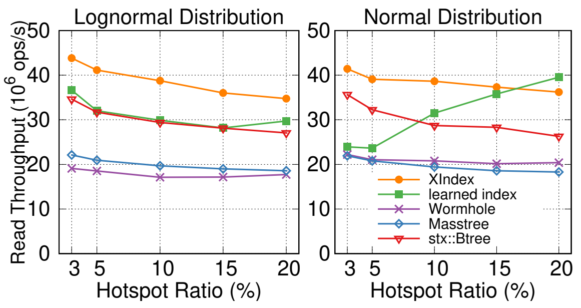


Figure 10. 24-thread read throughput in skewed query distribution.

7.3 Performance of dynamic workload

Query distribution. For dynamic workload, we first evaluate 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 ends differently. The smaller the hotspot is, the more skewed the query distribution is. Figure 10 shows the throughput with different skewness level under *normal* and *lognormal* dataset. 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 could perform even worse than stx::Btree and Wormhole. For *lognormal* dataset, when the hotspot ratio decreases under 5%, the increase of hot models' error bound slows down, thus we can observe a slight performacne improvement of the learned index due to improved locality.

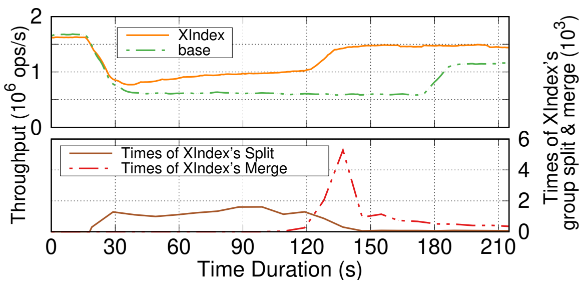
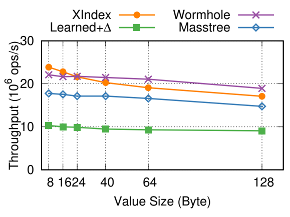


Figure 11. Read-write throughput and times of XIndex's group split and merge under dynamic workload.



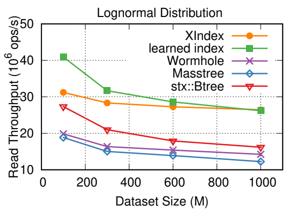


Figure 12. Read-write throughput of varying value size.

Figure 13. Read throughput of varying dataset size.

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 and 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 20-th second, the write ratio becomes 100% (half inserts and half removes), we remove all existing keys and insert with 50M *linear* dataset. From this point, both throughput of baseline and XIndex 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 size of delta index, so we can see the throughput starts to increase at the 30-th second.

At the 120-th second, XIndex finishes dataset shifting, and at the 170-th second, baseline ends shifting. Afterward, the read-write ratio is 90:10 and keys follow *linear* distribution. XIndex's background thread detects that both the size of delta index and the error bound of group is small after the shifting, so it invokes lots of group merge 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 size under *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 beacuse the overhead of data copying during compaction (128B's overhead is 13.5x larger than 8B's).

Dataset size. Figure 13 shows the performance of XIndex with different dataset size under *lognormal* dataset using 24 threads. As dataset size increases, both the learned index and XIndex show 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

Inlined values vs. separated values. XIndex directly stores raw values in *data_array*, i.e., inline values. Compared with another popular approach [19, 22] where values are stored in a separate storage (separated values) and only the pointers are inline, our approach reduces costly raw DRAM latency for queries. On the one hand, separating values can reduce compaction cost for large values since only pointers need copying, and reduce the complexity of compaction scheme since stale copy of values is eliminated by design. On the other hand, it incurs forbiddingly high compaction cost since XIndex then needs additional memory accesses to check if the value has been logically removed and is ready for garbage collection. To avoid such cost, we can store the status such as the *removed* flag beside the value pointer in *data_array*. Then, Two-Phase Compaction is required to ensure that no change to status will be lost.

Limitations. First, when the dataset is small, other index structures (e.g., stx::Btree) can outperformance 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 64B key, there can be up to 50% performance degradaion compared with 8B key. We leave the design of a more flexible strucutre for variable-scale workloads and reducing the cost of long keys as future work.

9 Related Works

There has been works that extend or build system upon the learned index. Many works extend the learned index to support writes requests. ALEX [9] achieves this by leaving gaps in a sorted array for inserting new data. AIDEL [18] handles insertions by attaching a sorted list for each record in the sorted array. Nevertheless, these works lacks the support for concurrency. 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 a already known query distribution. SageDB [14] is a database that propose 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 structures and adapts the 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 craft 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 coordination accesses to leaf nodes and uses a combination of locking and 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 aims to reduce the height of the trie. It uses per-node locks to coordinate writes and uses copy-on-write 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 copy-on-write (COW) and compare-and-swap (CAS) techniques. Building up on existing works, XIndex leverages fine-grained locking and optimistic concurrency control to coordination accesses to individual records and uses RCU mechanism to eliminates 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 multi-core platform via a combination of the innovative Two-Phase Compaction and a number of classical concurrency techniques. Futhermore, it can dynamically adjust its structure according to the runtime workloads to maintain competitive performance. Extensive evaluation demonstrates that XIndex can have a performance advantage by up to 3.3× and 4.4×, compared with Masstree and Wormhole, respectively.

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] T. Bingmann. Stx b+ tree c++ template classes, 2008.

[2] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. Hot: a height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 521–534. ACM, 2018.

[3] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45, pages 257–268. ACM, 2010.

[4] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, volume 1, pages 181–190, 2001.

[5] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. *ACM SIGPLAN Notices*, 47(4):199–210, 2012.

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

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

[8] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[9] J. Ding, U. F. Minhas, H. Zhang, Y. Li, C. Wang, B. Chandramouli, J. Gehrke, D. Kossmann, and D. Lomet. Alex: An updatable adaptive learned index. *arXiv preprint arXiv:1905.08898*, 2019.

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

[11] P. Ferragina and G. Vinciguerra. The pgm-index: a multicriteria, compressed and learned approach to data indexing, 10 2019.

[12] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1189–1206. ACM, 2019.

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

[14] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. 2019.

[15] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

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

[17] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[18] P. Li, Y. Hua, P. Zuo, and J. Jia. A scalable learned index scheme in storage systems. *arXiv preprint arXiv:1905.06256*, 2019.

[19] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.

[20] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[21] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[22] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A risc machine sort. In *ACM SIGMOD Record*, volume 23, pages 233–242. ACM, 1994.

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

[24] X. Wu, F. Ni, and S. Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 18. ACM, 2019.

[25] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1567–1581. ACM, 2016.