The Concurrent Learned Indexes for Multicore Data Storage

ZHAOGUO WANG, HAIBO CHEN, YOUYUN WANG, CHUZHE TANG, and HUAN WANG, Shanghai Jiao Tong University, China, Shanghai AI Laboratory, China, and Ministry of Education of the People's Republic of China, China

We present XIndex, which is a concurrent index library and designed for fast queries. It includes a concurrent ordered index (XIndex-R) and a concurrent hash index (XIndex-H). Similar to a recent proposal of the learned index, the indexes in XIndex use learned models to optimize index efficiency. Compared with the learned index, for the ordered index, XIndex-R is able to handle concurrent writes effectively and adapts its structure according to runtime workload characteristics. For the hash index, XIndex-H is able to avoid the resize operation blocking concurrent writes. Furthermore, the indexes in XIndex can index string keys much more efficiently than the learned index. We demonstrate the advantages of XIndex with YCSB, TPC-C (KV), which is a TPC-C-inspired benchmark for key-value stores, and micro-benchmarks. Compared with ordered indexes of Masstree and Wormhole, XIndex-R achieves up to 3.2× and 4.4× performance improvement on a 24-core machine. Compared with hash indexes of Intel TBB HashMap, XIndex-H achieves up to 3.1× speedup. The performance further improves by 91% after adding the optimizations on indexing string keys. The library is open-sourced[1].

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

Additional Key Words and Phrases: learned index structures, concurrent data structures

ACM Reference Format:
Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. 2022. The Concurrent Learned Indexes for Multicore Data Storage. *ACM Trans. Storage* 18, 8, Article 8 (February 2022), 36 pages. https://doi.org/10.1145/3478289

1 INTRODUCTION

The pioneering study on the learned index [35] opens up a new perspective on how machine learning (ML) 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. The learned index shows the cases of learning both ordered index and hash index.[2]

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 (i.e., the 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

[1] https://ipads.se.sjtu.edu.cn:1312/opensource/xindex.git
[2] We refer to the ML-based ordered index and hash index proposed in [35] as *the learned range index* and *the learned hash index* seperately.

Authors' address: Zhaoguo Wang; Haibo Chen, haibochen@sjtu.edu.cn; Youyun Wang; Chuzhe Tang; Huan Wang, Shanghai Jiao Tong University, Institute of Parallel and Distributed Systems, 800 Dongchuan Road, Minhang District, Shanghai, 200240, China , Shanghai AI Laboratory, Shanghai, China , Ministry of Education of the People's Republic of China, Engineering Research Center for Domain-specific Operating Systems, Shanghai, China.

contiguous, so the key-position mapping is easier to learn. Using simple learned models, the learned range index performs 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 real-world scenarios (Section 2.2). First, the learned range index does not support any modifications, including inserts, updates, or removes. It also assumes the workload has a relative static query distribution[3] — it assumes all data are uniformly accessed. Second, the learned hash index has to block all update requests during resizing to avoid losing concurrent updates. Last, when indexing string keys, both the learned range index and the learned hash index suffer large performance degradation compared with indexing integer keys.

In this paper, we present XIndex, a concurrent index library inspired by the learned index. It includes two basic indexes: XIndex for range indexes (XIndex-R) and XIndex for hash indexes (XIndex-H). XIndex-R leverages learned models to speed up the lookup. It can handle concurrent writes efficiently with good scalability. Moreover, it is designed to adapt its structure deterministically at runtime and decouple its efficiency from runtime workload characteristics. XIndex-H leverages learned models to reduce the hash conflict ratio comparing with traditional hash tables. It can concurrently process update requests during resizing. Besides, XIndex also integrates methodologies that can index string keys with better performance than any existing system. Data in XIndex is kept in memory for fast queries and updates, and persistence is achieved by logging and checkpointing. Specifically, this paper makes the following contributions:

A scalable and concurrent learned range index, XIndex-R. Compared with the learned range index, XIndex-R is able to efficiently handle concurrent writes without affecting the query performance by leveraging fine-grained synchronization [8–10, 45] and a new compaction scheme, Two-Phase Compaction. Furthermore, XIndex-R adapts its structure according to runtime workload characteristics to support dynamic workload. We demonstrate the advantages of XIndex-R with both YCSB and TPC-C (KV), a TPC-C-inspired benchmark for key-value stores. XIndex-R achieves up to 3.2× and 4.4× performance improvement comparing with Masstree and Wormhole, respectively, on a 24-core machine.

A non-blocking learned hash index, XIndex-H. Compared with the learned hash index, XIndex-H introduces non-blocking writes by performing the resize operation asynchronously combined with Two-Phase Compaction. We demonstrate the advantages of XIndex-H with micro-benchmarks. XIndex-H achieves up to 3.1× performance improvement compared with Intel TBB HashMap [51].

A technique that can index string keys efficiently. Inspired by existing string key indexes [4, 6, 38, 45], XIndex takes advantage of common prefixes to reduce lookup latency. Specifically, it greedily clusters keys with common prefixes into groups. Then, in each group, it uses the unique part of each key, the *partial key*, to train the model and index the data. Experimental results show that with these design choices, XIndex has up to 91% better performance over other state-of-the-art index structures when indexing string keys.

The rest of the paper is organized as follows: Section 2 describes the background and motivation; Section 3 gives the design of XIndex-R, including the support for string keys; Section 4 provides the design of XIndex-H; Section 5 presents the design for persistence; Section 6 shows the evaluation results; Section 7 summarizes related works; Section 8 concludes this paper.

2 BACKGROUND AND MOTIVATION

[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.

2.1 The Learned Index

The learned index proposed by [35] views indexes as functions to be learned that map keys to data positions. For common index types, including range indexes and hash indexes, these functions can be easily obtained by transforming the cumulative distribution function (CDF). For example, for fixed-length key-value pairs sorted by keys and stored in a continuous array, the index function is effectively the CDF of the key distribution multiplied by the total number of records. The core idea behind the learned index is to approximate CDFs with ML models such as deep neural networks (Section 2.1.1), and use the learned CDFs to enhance and even replace these traditional indexes (Sections 2.1.2 and 2.1.3).

2.1.1 Learning CDFs with the Recursive Model Index. To build learned indexes, we need first to learn the CDF of the data indexed. Using a single ML model to learn the entire CDF falls short in prediction accuracy due to the complexity of CDFs. To improve the prediction accuracy while keeping the model efficient to execute, [35] propose a staged model architecture, termed the Recursive Model Index (RMI).

An RMI contains multiple stages of models, which resemble the multi-level structure of B-tree. In an RMI, one model (an analogy for B-tree nodes) is located at the first stage (an analogy for B-tree levels), and each subsequent stage contains more models increasingly. To predict a CDF value for a given key, the key is first used as input to activate the model at the first stage. The model outputs an index number for finding a next-stage model to be activated. This process repeats until the model at the last stage is activated. The final model produces the predicted CDF value of the given key. The idea behind this architecture is to distribute regions of the complete CDF function to different models, so that each model can learn from a much smaller training set and yield better accuracy. Since upper-level models decide how keys should be distributed to lower-level models, an RMI is trained stage by stage starting from the first stage. The true CDF values are used as labels in the training set, and the next-model index number is obtained by multiplying CDF values with the number of models at the next stage.

RMI has a rigid architecture that is configurable only through a few parameters: the number of stages, the number of models at each stage, and the type of models. Unlike the proposal of Data Alchemist [30] that envisions an automatic approach for synthesizing new data structures from basic design elements [29, 31], RMI is, in essence, a new data structure that builds upon ML models.

2.1.2 The Learned Range Index. In the learned range index, records are sorted by keys and stored in a continuous in-memory array. Learned CDFs are used to predict record positions in sorted data arrays. Given the CDF F, the position of a record is $\lfloor F(key) \times N \rfloor$, where N is the total number of records. However, learned CDFs have errors. In order to provide the correctness guarantee, the learned range index stores the maximal and minimal prediction errors of the model. After training the model, the learned range 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 with key k, its position must fall in $[\text{pred}(k) + err_{min}, \text{pred}(k) + err_{max}]$, where $\text{pred}(k)$ is the predicted position. Therefore, to lookup a record, the learned range index uses binary search within such range. We refer to the logarithmic span of the binary search range, $\log_2(max\_err - min\_err + 1)$, as the *error bound*.

2.1.3 The Learned Hash Index. The learned hash index is a separate chaining hash index with linked lists. Instead of maximizing the randomness of the hash function, the learned hash index places records according to the learned CDFs to minimize conflicts. Given the CDF F, a learned hash function h can be expressed as $F(key) \times M$, where M is the size of the bucket array. If the CDF

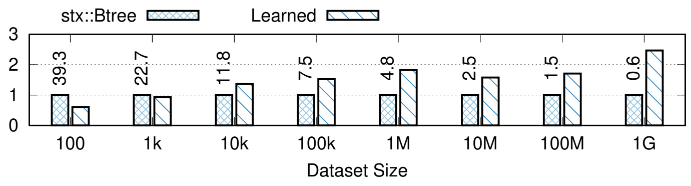


Fig. 1. The learned range index throughput normalized to stx::Btree. The numbers are stx::Btree's absolute throughputs in MOPS. The learned range index uses 10k models in 2-staged RMI where both stages use linear models. stx::Btree uses the default fanout, 16.

is perfectly learned, no conflict exists, and each key is hashed to positions in sorted order. However, learned CDFs have errors. Conflicting records are stored in the linked lists associated with buckets.

2.1.4 Advantages of the Learned Index. To demonstrate their advantages, we evaluate both the learned range index and the learned hash index. For the learned range index, we evaluated different dataset sizes under a *normal* distribution and compared it against stx::Btree [3], an open-sourced B-tree (Figure 1). The learned range index can outperform stx::Btree with large datasets ($\geq$ 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 range 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 range 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 range 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% (207 ns to 399ns). Therefore, the learned range index has better performance than stx::Btree with larger datasets. For the learned hash index, we evaluated with *linear* (Table 3) dataset and compared it against std::unordered_map, the hash map in C++ Standard Library. The learned hash index can efficiently reduce the conflict rate by 50× (from 35% to 0.7%). As a result, for 8-byte value, the learned hash index can reduce the memory footprint by 26% (3.7GB v.s. 5.0GB).

The performance advantage of the learned index is tightly coupled with the data distribution and the type of learning models. Together they decide the accuracy of learned CDFs and model computation cost. When fast and small models can fit the distribution with high accuracy, the learned indexes can enjoy a significant reduction in search space and I/Os with little model computation cost, with little memory overhead for storing models. Otherwise, the accuracy can be low even with large and costly models. Such characteristics differ from classic optimization techniques such as Bloom filter [5, 7, 22, 52], which uses additional bit arrays marked by elements' hash values, and fractional cascading [11, 12], which uses additional pointers to shorten subsequent lookup ranges. In these techniques, the performance gain and space overhead are less related to the underlying data distribution and can be probabilistically estimated. More similar to the learned index are the trie index [24] and its variants [4, 27, 61] as they leverage the data distribution through delicate encoding to save space and search time. In contrast, the learned index uses ML models to capture the data distribution. However, the precise relationship between the data distribution and model accuracy for different model types is yet unknown and requires further research.

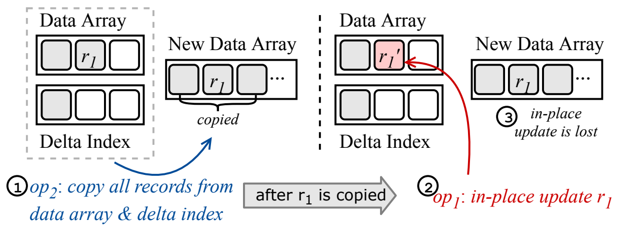


Fig. 2. Consistency issue with concurrent operations.

2.2 The Issues

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

2.2.1 Handling Updates. The learned range 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 range 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 range index. When building a learned range index with 200M records and using Masstree as 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 range 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 range 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 range 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.

2.2.2 Dependence on Workload Characteristics. The learned range 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 range index can have worse performance than B-tree with certain workloads. Table 1 shows the performance of the learned range index and stx::Btree under both uniform and skewed query distributions on the *osm* dataset (details in Section 6). 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

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.

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 range 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 range index has much higher average error bounds on the frequently accessed records, which hinders the query performance. The underlying cause is that the learned range 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 6.2.3).

2.2.3 Supporting String Keys. When indexing string keys, The learned index suffers large performance degradation compared with indexing integer keys. The performance [35] is even worse (0.78×) than traditional index structure such as B-tree. Several challenges arise for model-based indexes under string keys. First, the model computation cost increases dramatically along with the key length. For instance, the linear model's computation time increases from 16 ns to 400 ns when the key length grows from 8 bytes to 128 bytes. Second, the model errors increase with the increasing of key length. With the random dataset, the error bound of a linear model increases from 24 to 67 when the key length increases from 8 bytes to 128 bytes. Although using complicated models such as neural networks can reduce the errors, this method has high inference and training costs because of the complexity. Third, it also takes a cost to perform the search on string keys. This is because the comparison cost of two keys is proportional to the key length. With the above example, it takes 1370 ns for the 128-byte key while only 590 ns for the 8-byte key.

2.2.4 Resizing the Learned Hash Index. The learned hash index has to block all update requests during resizing, which significantly degrades its performance. To resize the index, the learned hash index retrains the model with current data and then rehashes the data using the new model into a new bucket array. To avoid losing concurrent updates due to data races, the learned hash index needs to block all updates during resizing. To evaluate blocking cost, we resize a learned hash index for linear dataset from 100M buckets to 150M buckets. The total blocking time is 160s. 33% of the cost comes from model training, while 67% comes from the rehashing.

3 XINDEX FOR RANGE INDEXES

XIndex-R 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-R performs updates in-place on existing records, and associates each group with a delta index to buffer insertions.

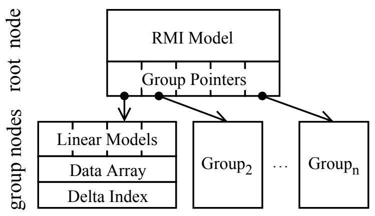


Fig. 3. The architecture of XIndex-R.

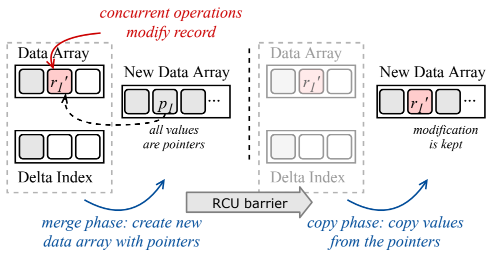

concurrent operations modify record
Data Array
New Data Array
Delta Index
all values are pointers
merge phase: create new data array with pointers
RCU barrier
Data Array
New Data Array
Delta Index
modification is kept
copy phase: copy values from the pointers

Fig. 4. Two-Phase Compaction prevents concurrent operations being lost.

XIndex-R introduces a new compaction scheme, Two-Phase Compaction (Section 3.1.3), to compact the delta index conditionally. The compaction is performed one group at a time in the background and does not block any concurrent operations. The compaction has two phases: the *merge phase* and the *copy phase*. In the merge phase, XIndex-R merges the current data array and delta index into a new data array. Instead of directly copying the data, XIndex-R 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-R performs the copy phase. It replaces each reference in the new data array with the real value. Considering the 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-R replaces p_1 with r_1' in the copy phase.

XIndex-R is able to adjust its structure according to runtime workload characteristics (Section 3.3). At runtime, if some group incurs high prediction error, XIndex-R 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-R performs *group split* — replacing the group with two new groups, each containing the half data of the old group. XIndex-R also performs *model merge* and *group merge*, if the merging does not affect the prediction accuracy. Furthermore, if there are too many groups, XIndex-R retrains the RMI model of the root node and may adjust its structure to improve the accuracy.

3.1 The Basic Algorithm

3.1.1 The Data Structure. XIndex-R 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-R uses a two-stage RMI architecture solely consisting of linear models. The number of models in its second stage is adjustable at runtime (Section 3.3).

Algorithm 1: The structure of XIndex-R.

```
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 8 bytes */
        is_ptr : 1 bit, removed : 1 bit
        lock : 1 bit, version : 61 bits;

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;
```

Algorithm 2: Get and put operations in XIndex-R.

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

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 3.1.4). For optimization purposes, the *group_t* maintains its smallest key in a separate variable, *pivot*.

3.1.2 *Basic Operations.* XIndex-R 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 14), then look up the position of the requested record in *data_array* with the given key (Lines 3 and 15). After then, their procedures diverge.

To find the corresponding group (*get_group*), XIndex-R 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

Algorithm 3: Two-Phase Compaction

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

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 3.1.4).

After finding the group *group*, XIndex-R 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.

After looking up *data_array*, the procedures diverge. For *get*, if XIndex-R 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*, XIndex-R first tries to update/remove the record in-place (Line 16). If XIndex-R cannot perform update/remove in-place, then it proceeds to perform an *upsert* on *buf* (Line 19) and optionally *tmp_buf* (Line 25) only if the *frozen_buf* flag is true (Line 18). The upsert operation updates the record in-place if a previous version exists, and otherwise inserts a new record. For *scan*, XIndex-R 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 3.2 since most subtleties stem from consistency consideration.

3.1.3 Compaction. XIndex-R uses dedicated non-stop background threads to compact data arrays with delta indexes[4]. Compaction is performed one group at a time to isolate the performance impact to other groups. To ensure consistency in face of concurrent operations (Section 2.2), XIndex-R divides the compaction into two phases, *merge phase* and *copy phase* (Algorithm 3).

In the merge phase, XIndex-R merges a group's *data_array* and *buf* into a new sorted array where values are pointers to existing records. XIndex-R first sets the old group's *buf_frozen* flag to stop newly issued *put*s inserting to *buf* (Line 2). Then XIndex-R initializes *tmp_buf* to buffer insertions during compaction (Line 4). Afterward, it creates a new group (Line 5) and merges the old group's *data_array* and *buf* into the new group's *data_array* (Line 6). In *new_group.data_array*,

[4]This might lead to data arrays being compacted with relatively small delta indexes. In this case, compaction is still performed as it helps keep the delta index consistently small, which is beneficial for lookup performance.

Algorithm 4: The group split operation in XIndex-R.

```
1  split(group):
       /* step 1 */
2      $g'_a$, $g'_b$ ← allocate 2 new group
3      $g'_a$.{data_array, buf} ← group.{data_array, buf}
4      $g'_b$..{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
       /* step 2.1, merge phase */
14     $g_a$, $g_b$ ← allocate 2 new groups
15     tmp_array ← merge(old_group.data_array,
         old_group.buf)
16     {$g_a$, $g_b$}.data_array ← split(tmp_array, $g'_b$.pivot)
17     {$g_a$, $g_b$}.buf ← {$g'_a$, $g'_b$}.tmp_buf
18     train {$g_a$, $g_b$}'s models with {$g_a$, $g_b$}.data_array
19     {$g_a$, $g_b$}.pivot ← {$g'_a$, $g'_b$}.pivot
20     $g_a$.next ← $g_b$
21     init $g_a$'s and $g_b$'s other fields
22     atomic_update_reference(group, $g_a$)
23     rcu_barrier()
       /* step 2.2, copy phase */
24     foreach record in {$g_a$, $g_b$}.data_array do
25         replace_pointer(record)
26     rcu_barrier()
27     reclaim {old_group, $g'_a$, $g'_b$}'s memory
```

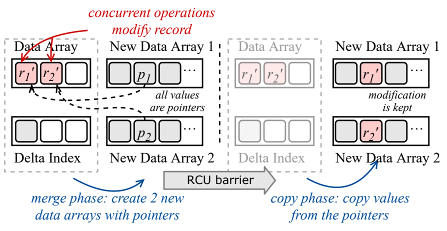
the value of each record is the reference to the corresponding record in either *group.data_array* or *group.buf*, and the *is_ptr* flag of each record is set to be TRUE. During merging, XIndex-R skips the logically removed records. The old group's *tmp_buf* is reused as the new group's *buf* (Line 7). After training linear models (Line 8) and initializing the remaining metadata of the new group (Line 9), XIndex-R atomically replaces the old group with the new one by changing the group reference in root's *groups* (Line 11).

In the copy phase, XIndex-R replaces each reference in the new group's *data_array* with the latest record value (Line 14). The replacement is performed atomically with helper function *replace_pointer* (Algorithm 5). XIndex-R uses *rcu_barrier* (Line 15) 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 16). To implement *rcu_barrier*, each worker thread is required to maintain a version that gets incremented after finishing each operation. Invoking *rcu_barrier* causes the background thread to scan through these version twice, checking that each has been incremented which indicates concurrent operations active at the invocation time have finished.

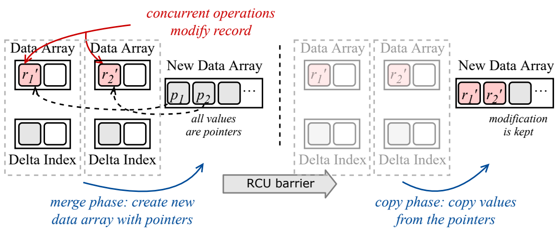
Though the idea of compacting buffered writes in background commonly seen in LSM tree-based systems is not new [1, 32, 44, 49, 54, 57, 59], our compaction scheme is fundamentally different from theirs. Unlike LSM tree-based systems where compacted data is read-only, we allow concurrent in-place updates during compaction, so that read operations do not need always to visit the delta index (an analogy for smaller levels in LSM tree). This design decision creates challenges of ensuring immediate visibility of committed puts and preserving them after compaction, which are addressed by Two-Phase Compaction.

3.1.4 Structure Updates. XIndex-R adapts its structure to dynamic workloads at runtime (Section 3.3) with model split/merge, group split/merge, and root update operations.

Model split and model merge. XIndex-R supports splitting and merging models within a group to improve lookup efficiency. For model split, XIndex-R 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-R atomically



(a) Group split.



(b) Group merge.

Fig. 5. Group split and group merge operations are performed similarly as Two-Phase Compaction.

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-R uses two steps to split a group's data evenly into two groups (Algorithm 4). It uses a similar scheme as Two-Phase Compaction to allow concurrent modification during performing group split, which is shown in Figure 5a. In step 1, XIndex-R 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-R creates g'_a and g'_b (Line 2). They share the same *data_array* and *buf* with the old group (Line 3) but have different *pivot* keys (Line 5-6). XIndex-R 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-R sets the *buf_frozen* flag (Line 11) and allocates *tmp_buf* for g'_a and g'_b (Line 13). In step 2, XIndex-R physically divides the data into two groups. Similar to compaction, this step has two phases. In the merge phase, XIndex-R first merges the old group's *data_array* and *buf* into *tmp_array* (Line 15). 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 16). It also reuses the *tmp_buf* of g'_a (g'_b) as the *buf* of g_a (g_b)[Line 17]. In the copy phase, for each group, the references in *data_array* are replaced with real values (Line 25). Last, XIndex-R links g_b at g_a*.next* (Line 20) and replaces g'_a with g_a in root's *groups* (Line 22).

Group merge. XIndex-R merges two consecutive groups' data into one new group to reduce the cost to lookup groups. As shown in Figure 5b, similar to 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-R flattens root's *groups* to reduce pointer access cost, retrains, and conditionally adjusts the RMI model to improve prediction accuracy. For root update, XIndex-R creates a new root node with a flattened *groups* and retrains the RMI model. After a new root is initialized, XIndex-R replaces the global root pointer atomically.

3.2 Handling Concurrency

XIndex-R achieves high scalability on the multicore platform using Two-Phase Compaction, along with classic techniques such as fine-grained locking [4, 45, 58], optimistic concurrency control [8–10, 45], and RCU [46]. We first discuss the coordination between writers that ensures execution exclusiveness (Section 3.2.1), then discuss how readers can always fetch a consistent result with concurrent writers (Section 3.2.2). Afterward, we discuss the interleaving with concurrent background

Algorithm 5: Helper functions.

```
1  read_record(rec):
2     while TRUE do
3        ver ← rec.ver
4        removed, is_ptr, val ← rec.{removed, is_ptr, val}
5        if ¬rec.lock ∧ rec.version = ver then
6           if removed then
7              val ← empty
8           else if is_ptr then
9              val ← read_record(DEREF(val))
10          return val
11 update_record(rec, val):
12    lock(rec.lock)
13    succ ← FALSE
14    if rec.is_ptr then
15       succ ← update_record(DEREF(rec.val), val)
16    else if ¬rec.removed then
17       rec.val ← val; succ ← TRUE
18    rec.version++
19    unlock(rec.lock)
20    return succ

21 replace_pointer(rec):
22    lock(rec.lock)
23    ref.val ← read_record(DEREF(rec.val))
24    if ref.val = empty then
25       rec.removed ← TRUE
26    rec.is_ptr ← FALSE
27    rec.version++
28    unlock(rec.lock)
```

operations (Section 3.2.3) and provide a proof sketch of the correctness condition (Section 3.2.4). The formal proof can be found in the extended version[5].

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-R, compaction and structure updates are performed by dedicated background threads sharing no conflicts, thus avoiding concurrency issues due to their interleavings.

3.2.1 Writer-Writer Coordination. XIndex-R 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 16, Algorithm 2), and the per-record lock is acquired to prevent interleaving with concurrent writers (Line 12, Algorithm 5). If updating *data_array* is not feasible, writers then operate on the delta index (Line 19, 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 3.5).

3.2.2 Writer-Reader Coordination. XIndex-R ensures readers can always fetch a consistent result in the 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). This reader-retry synchronization method is introduced in [21], and similar ideas are exploited in [36, 37]. The concurrent delta index with a single read-write lock ensures the fetched result is consistent.

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

3.2.3 Interleaving with Background Operations. With the presence of background operations, XIndex-R 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[6].

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 2, Algorithm 3 and Line 11, Algorithm 4). In the 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 22, Algorithm 5). To ensure all concurrent writers use the same lock, XIndex-R places *rcu_barrier* before the copy phase (Line 12, Algorithm 3 and Line 23, 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 7, Algorithm 3 and Line 17, Algorithm 4).

To ensure no duplicate records, XIndex-R avoids insertions to different delta indexes, namely *buf* and *tmp_buf*. XIndex-R only initializes *tmp_buf* until all writers observe a frozen *buf* using the *rcu_barrier* (Line 2-4, 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.

3.2.4 Proof Sketch of Linearizability. The correctness condition of XIndex-R can be described as "a *get(k)* must observe the latest committed *put(k, v)*," namely, linearizability [28]. XIndex-R 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-R; (I_2) If there is a record with key k in XIndex-R, 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-R 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 3.2.3, XIndex-R 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 19 and 25, 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 3.2.3. 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.

3.3 Adjusting XIndex-R at Runtime

To reduce the performance variation, XIndex-R 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 6).

First, XIndex-R 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

[6]Duplicate records do not directly violate correctness, as long as XIndex-R 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.

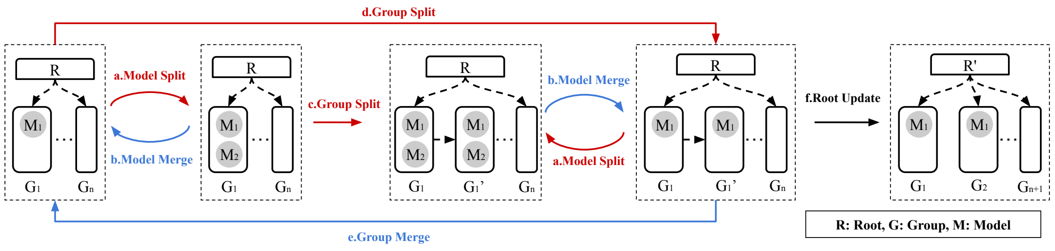


Fig. 6. Dynamic adjustment procedure illustration.

Operations	Trigger Condition
a. Model Split	Error bound $> e$ **and** the number of models $< m$
b. Model Merge	Error bound $\leq e \times f$ **and** the number of models > 1
c. Group Split	Error bound $> e$ **and** the number of models $= m$
d. Group Split	Buffer size $> s$
e. Group Merge	The number of models $= 1$ **and** error bound $\leq e \times f$ **and** buf $\leq s \times f$
f. Root Update	When groups are created and/or removed

Table 2. Conditions for structure update operations.

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-R will do model split (Figure 6-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-R will perform model merge (Figure 6-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-R will perform group split (Figure 6-c). Besides, if a group's delta index size is greater than the delta index size threshold (s, specified by the user), XIndex-R will also split the group (Figure 6-d). To reduce the cost of locating a group in the root, XIndex-R performs group merge (Figure 6-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-R periodically updates the root to reduce the access cost. Specifically, XIndex-R first checks all groups and perform model split/merge and group split/merge accordingly. If there is any group created or removed, XIndex-R then performs root update (Figure 6-f). During root update, if the average error bound is greater than e, XIndex-R will increase the number of 2nd stage models of root's 2-stage RMI[7]. If the average error bound is less than or equal to $e \times f$, XIndex-R reduces models.

3.4 Supporting String Keys

To solve the performance issue under string keys, XIndex-R adopts three important design choices tailored for string keys. First, XIndex-R uses partial keys to reduce both model computation and comparison costs (Section 3.4.1). The partial keys are order-preserving substrings of original keys that uniquely identify each record within a group. An efficient algorithm is used to compute partial keys. Second, XIndex-R leverages a greedy grouping strategy to adaptively partition keys into

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

Algorithm 6: Computing partial keys.

```
1  partial_key(keys):
2      pl ← cpl(0, keys[0], keys[0].len, keys[1],
         keys[1].len)
3      max_prefix ← pl
4      for i ← 2 to keys.size do
5          pl ← cpl(0, keys[i-1], pl, keys[i], keys[i].len)
6          l ← cpl(pl, keys[i-1], keys[i-1].len, keys[i],
             keys[i].len)
7          max_prefix ← max(max_prefix, pl + l)
8      end for
9      el ← max_prefix - pl
10     return pl, el

11 cpl(s, k1, len1, k2, len2): /* common prefix length */
12     for i ← s to min(len1, len2) do
13         if k1[i] ≠ k2[i] then
14             return i - s
15     end for
16     return min(len1, len2) - s
```

different groups (Section 3.4.2). After partitioning, each group contains a maximal number of keys, which keeps the group's model error and partial key length under specified thresholds. This partition scheme improves XIndex-R by reducing the number of groups and hence the indexing burden of the root. Third, XIndex-R uses piecewise linear models instead of an RMI model in the root node to index groups (Section 3.4.3). Using piecewise linear models helps XIndex-R gain control on indexing the key range of each model. As a result, partial keys and greedy grouping strategy can be reused for the root.

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

Algorithm 6 summarizes the procedure for identifying the partial key of a sorted key array. Given a sorted key array, XIndex-R first initializes the prefix length as the common prefix length of the first two keys, using the helper function *common_prefix_len* (Line 2). *common_prefix_len* returns the common prefix length of the substrings of input keys $k1$ and $k2$, starting at s (Line 1-16). Afterward, XIndex-R iterates over the sorted key array with a for loop to calculate the longest common prefix of all the keys, pl, and the longest common prefix length of two adjacent keys, *max_prefix*. Within a loop, let k be the key being examined, XIndex-R updates pl to be the prefix length between the existing common prefix among already examined keys and k (Line 5). *max_prefix* is updated to be the common prefix length between the current key and its previous key, $pl + l$, if this value is larger than the previous one (Line 6-7). Finally, the effective length is calculated by $max\_prefix - pl$ (Line 9).

XIndex-R benefits from partial keys in two ways. First, partial keys improve the efficiency of both model computation and model training: XIndex-R directly tokenizes partial keys into feature vectors, resulting in shorter feature length and hence reduced arithmetic computation. Second, the binary search overhead is reduced as well. XIndex-R uses the partial key for key comparison because of the uniqueness and order preservation of each partial key. Hence, the comparison cost is, to a great extent, diminished.

Other systems have also exploited the idea of indexing using substrings of keys. [6] uses a similar partial key design to reduce the comparison cost of B-tree. However, its partial keys are computed

Algorithm 7: The greedy grouping strategy.

```
1  greedy_grouping(keys, et, pt, fs, bs): /* fs > bs */
2    groups ← EMPTY
3    while $i < keys.size$ do
4      cur_grp ← EMPTY; err ← 0; par_len ← 0
5      while $err < et$ and $par\_len < pt$ do
6        records ← retrieve next fs records
7        $i \leftarrow i + fs$
8        add records to cur_grp
9        train models on cur_grp
10       err ← average_error(cur_grp)
11       par_len ← partial_key(cur_grp.key_array)
12     end while
13     while $err > et$ or $par\_len > pt$ do
14       $i \leftarrow i - bs$
15       remove last bs records from cur_grp
16       train model on cur_grp
17       err ← average_error(cur_grp)
18       par_len ← partial_key(cur_grp.key_array)
19     end while
20     add cur_grp to groups
21   end while
22   return groups
```

and stored for each consecutive key pairs, while we only compute and store two parameters, *pl* and *el*, for each group, which usually serves hundreds to thousands of keys. Furthermore, we adjust the grouping with heuristics to fully utilize the computed partial key information (Section 3.4.2). Meanwhile, storage systems commonly use fixed-length prefixes of keys for indexing to save space and reduce I/O cost [39, 50].

3.4.2 *A Greedy Grouping Strategy.*

XIndex-R greedily range-partitions data into different groups to ensure that the partial key length and model errors are under specified thresholds. Algorithm 7 depicts this greedy strategy of grouping data in XIndex-R. User-specified parameters are used to fine-tune XIndex-R: the error threshold *et*, the partial key length threshold *pt*, the forward step size *fs*, and the backward step size *bs*. XIndex-R iterates over all the records. During the iteration, it determines whether to add *fs* records into the current group or remove *bs* records from the current group according to the two thresholds, *et* and *pt*. Specifically, each time XIndex-R moves forwards, it first adds the next *fs* records to the current group (Line 6-8). Then XIndex-R trains a linear model on the current group to get the average error (Line 9-10) and uses Algorithm 6 to compute the length of the partial key (Line 11). It continues to move forwards as long as both the model error and partial key length are smaller than their respective thresholds. The last forward step can cause the violation of thresholds; therefore XIndex-R uses backward steps for alleviation. For each backward step, XIndex-R removes the most recently added *bs* records from the current group (Line 15). It repeats backward steps until both the model error restriction and partial key length restriction are restored (Line 13).

3.4.3 *Using Piecewise Linear Model in the Root.*

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

XIndex-R chooses piecewise linear models instead of the RMI model for two reasons. First, in RMI, two pivots distant with each other can be assigned to the same leaf model, which results in a large span in the key range of the leaf model. The larger span the key range has, the less chance XIndex-R has to remove common prefix for the model, and hence less performance improvement from partial keys. Second, in practice, multiple stages of linear models in RMI causes a significant increase in computational overhead for string keys. This increase cancels off the benefits of reduced model error brought by RMI, leading to no improvement for the overall performance. For a specific case, after adding another layer, binary search time is reduced from 1291 ns to 1091 ns, but the inference time increases from 227 ns to 616 ns.

3.5 Optimizations

3.5.1 A Scalable Delta Index. In the basic version, XIndex-R uses stx::Btree protected by a global read-write lock as its delta index. This limits the scalability when concurrent threads perform operations (e.g., *get* and *put*) on the same group. We did not adopt the designs of existing concurrent index structures such as Bayer and Schkolnick's B-tree [2], Foster B-tree [26], and Masstree [45] due to performance consideration. Bayer and Schkolnick's B-tree and Foster B-tree use fine-grained read-write locks to protect each node, introducing performance overhead on index lookup. For the Masstree, it employs additional mechanisms to support multiple variable length values for each key, which incurs performance penalty. Moreover, the epoch-based memory reclamation in Masstree leads to high tail latency. Thus, we implement a more light-weight and scalable structure as the delta index: it uses optimistic concurrency control to protect read accesses on each node, and fine-grained locks for write. It doesn't physically remove deleted nodes from the structure. Instead, it reclaims the memory of the entire structure after the compaction.

3.5.2 Fast Sequential Insertions. Sequential insertion is a common pattern in real-world workloads, such as periodically checkpointing. For such cases, the user can provide hints to XIndex-R so that XIndex-R 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-R 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-R is intact. Since many sequential insertion workloads have relatively static data distribution, XIndex-R only retrains models when the current model cannot generalize to newly appended data, namely, when the error bound exceeds the threshold.

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

4 XINDEX FOR HASH INDEXES

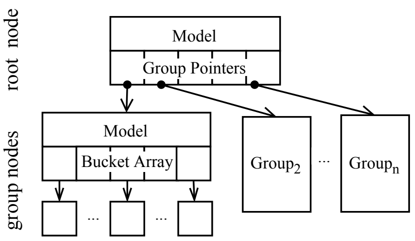


Fig. 7. The architecture of XIndex-H.

Algorithm 8: The structure of XIndex-H.

```
struct root_t:
    model_t root_model;
    uint32_t group_n;
    group_t* groups[];

struct group_t:
    bool_t frozen;
    uint32_t bucket_num;
    model_t model;
    bucket_t bucket_arr[];
    group_t* tmp_group;

struct bucket_t:
    chain_t* chain;
    key_t key;
    val_t val;
    uint64_t /* composite 8 bytes */
        uninitialized : 1 bit
        is_ptr : 1 bit, removed : 1 bit
        lock : 1 bit, version : 60 bits;
```

To improve the efficiency of the learned hash index, we propose the design of XIndex-H. Figure 7 gives the architecture of XIndex-H. XIndex-H adopts a two-layer design, which is similar to XIndex-R. The top layer contains a single *root* node that uses an ML model to hash keys into different *groups* in the bottom layer. Each group contains a bucket array, and each bucket has a linked list to solve the hash conflicts. The groups hash keys into their buckets with linear models. Each group is resized independently by background threads.

XIndex-H realizes non-blocking resize operations with Two-Phase Compaction. However, compared with XIndex-R, there is no “temporary delta index” in XIndex-H. Instead, during resizing, XIndex-H directly routes insertions to the new group. XIndex-H also incorporates the designs for string key support in XIndex-R, such as the partial key design (section 3.4.1) and the SIMD optimization (section 3.5.3).

4.1 The Basic Algorithm

4.1.1 The Data Structure. Algorithm 8 shows the layout of XIndex-H. The root node (*root_t*) includes a group pointer array *groups* and a linear model *root_model*. The *root_model* learns the CDF of the key distribution and serves as the hash function to route data into different groups. Each group node (*group_t*) contains a bucket array *bucket_arr*, a linear model *model*, a *frozen* flag, and a pointer for the temporary group *tmp_group*. The *model* hashes data into different buckets in the group. The *frozen* flag and the *tmp_group* pointer are for the non-blocking resize operation. Every bucket (*bucket_t*) in the bucket array contains the basic record data (*key*, *value*), the compact 8-byte metadata (*is_ptr*, *removed*, *uninitialized*, *lock*, *version*), and a pointer to its chain (*chain*). The *uninitialized* flag is set to *true* when the bucket array is allocated, which indicates a new record can be inserted into the bucket. The chain (*chain_t*) is used to store conflicting data.

4.1.2 Get and Put Operations. Algorithm 9 shows the pseudocode of *get* and *put* in XIndex-H. Both of them need to locate the corresponding group and bucket through *locate_group* and *locate_bucket*. The *locate_group* uses *root_model* as a hash function to decide the group that serves the request, while *locate_bucket* calculates the *bucket* for the key using the group’s *model*.

After locating the bucket, the *get* operation uses *get_from_bucket* and *get_from_chain* to fetch the data from the bucket and its chain (Line 4-6). The *get_from_bucket* function works similarly as *read_record* (algorithm 5), except that it (1) additionally verifies that the stored key matches the given key, and (2) returns EMPTY if the bucket is marked *uninitialized*. If the target is not found, it will further check the *tmp_group*’s *bucket* and *chain* (Line 7-11).

Algorithm 9: Get and put operations in XIndex-H.

```
1  get(key) :
2      group ← locate_group(root, key)
3      bucket ← locate_bucket(group, key)
4      val ← get_from_bucket(bucket, key)
5      if val = EMPTY then
6          val ← get_from_chain(bucket.chain, key)
7      if val = EMPTY ∧ tmp_group ≠ NULL then
8          bucket ← locate_bucket(tmp_group, key)
9          val ← get_from_bucket(bucket, key)
10         if val = EMPTY then
11             val ← get_from_chain(bucket.chain, key)
12     return val
13 put(key, val):
14 retry:
15     group ← locate_group(root, key)
16     bucket ← locate_bucket(group, key)
17     if group.frozen = FALSE then
18         if ¬ upsert_to_bucket(bucket, key, val) then
19             upsert_to_chain(bucket.chain, key, val)
20     else
21         if update_bucket(bucket, key, val) ∨
            update_chain(bucket.chain, key, val) then
22             return
23         if tmp_group = NULL then
24             goto retry
25         bucket ← locate_bucket(tmp_group, key)
26         if ¬ upsert_to_bucket(bucket, key, val) then
27             upsert_to_chain(bucket.chain, key, val)
```

For the *put* operation, XIndex-H checks the *frozen* flag after locating the target bucket (Line 17). This flag indicates whether some background threads are trying to resize the hash table. If not, XIndex-H will directly upsert the record to the *bucket* or its *chain* (Lines 18 and 19). An upsert operation updates the record in-place if a previous version exists, and otherwise inserts a new record. The *upsert_to_bucket* function works similarly as *update_record* (algorithm 5), except that it (1) additionally verifies that the stored key matches the given key before update, and (2) only inserts a new record if the bucket is marked *uninitialized*. If *frozen* is true, the *put* operation only performs an in-place update to the *bucket* or its *chain* (Line 21). If the in-place update fails, indicating that a previous version of the record does not exist, it then inserts the record into *tmp_group* (Line 23-27).

4.1.3 The Resizing. XIndex-H performs the resize operation asynchronously to avoid blocking *get* and *put* operations. Several background threads periodically check the load factor of each group and perform the resize if necessary. To resize a group, XIndex-H increases the bucket array size when the load factor is larger than a given threshold *upper_lf*, and shrinks the array size if the load factor is smaller than another threshold *lower_lf*.

To avoid blocking the concurrent updates, XIndex-H leverages the Two-Phase Compaction to resize the hash table in the background. Algorithm 10 gives the resize algorithm in detail. Resizing a given group includes two phases. In the first phase, it creates a new group with all existing records. However, each record's value is the pointer instead of real content, which points to the content in the old group. In detail, XIndex-H first retrieves all keys to train a new model (Line 2). The model is trained at the beginning because rehashing data into the new group depends on the new model, whereas XIndex-R trains the new model after the new group's data is ready in order to calculate the precise error bounds. Then, it enables the *frozen* flag to stop inserting new data into the old group (Line 3 in Algorithm 10, Line 17 in Algorithm 9). However, the old group can still serve read and in-place update requests (Line 21 in Algorithm 9). After an RCU barrier, XIndex-H allocates a new group with a bucket array of proper size (Lines 5 and 6). The new group also uses the previously trained model (Line 7). Afterward, XIndex-H links the new group to the old *group* via its *tmp_group* pointer. All new insertions will be routed to the *tmp_group*. XIndex-H then starts to migrate the data from the old group to the new group. Instead of having the real content, the new group maintains references of record values in the old group. After initializing the new group with all existing records, XIndex-H atomically replaces the old group with the new group (Line 13).

Algorithm 10: XIndex-H resizes with Two-Phase Compaction.

```
1  resize(group, sz):
       /* phase 1 */
2      model ← train a model with group's keys
3      group.frozen ← TRUE
4      rcu_barrier()
5      new_group ← allocate a new group
6      new_group.bucket_arr ← allocate sz buckets
7      new_group.model ← model
8      init new_group's other fields
9      group.tmp_group ← new_group
10     foreach record r in group do
11         insert <r.key, r.addr> into new_group
12     old_group ← group
13     atomic_update_reference(group, new_group)
14     rcu_barrier()
       /* phase 2 */
15     foreach record r in new_group do
16         replace_pointer(r)
17     rcu_barrier()
18     reclaim old_group's memory
```

In the second phase, XIndex-H replaces all value pointers in the new group with concrete values (Lines 15 and 16). The barrier at the start of the 2nd phase ensures that all workers access the data through the reference of the new group. Value references are replaced with real value content one by one, protected by fine-grained locks. In the end, XIndex-H safely reclaims the old group with the help of RCU (Line 18).

It should be noted that this two-phase resizing mechanism is general and applicable to other hash indexes, as evidenced in [13]. Nevertheless, it can be more beneficial in the context of the learned hash index as its additional model retraining prolongs the resize process (Section 2.2.4), making non-blocking resize more attractive.

4.2 Handling Concurrency

XIndex-H combines Two-Phase Compaction with fine-grained locking and optimistic concurrency control to handle concurrency correctly and efficiently. First, it uses fine-grained locks to coordinate writers (Section 4.2.1). Second, it leverages optimistic concurrency control [8–10, 45] to guarantee the consistent result of readers (Section 4.2.2). Moreover, Two-Phase Compaction ensures writers and readers' correctness when interleaving with background resize operations (Section 4.2.3). The proof sketch is provided in Section 4.2.4.

4.2.1 Writer-Writer Coordination. *Put* requests that operate on the same key will be protected by per-bucket lock in the bucket array, and XIndex-H uses a concurrent linked list for each bucket's chain. All writers will first try to acquire the corresponding bucket lock via the *update_bucket* and *upsert_bucket* function. If XIndex-H fails to update or insert into the bucket, writers operate on the bucket's chain. The chain itself is a concurrent data structure, which ensures the correctness under concurrent writers.

4.2.2 Writer-Reader Coordination. The optimistic concurrency control, based on versions and locks, guarantees that readers can always fetch a consistent result regardless of concurrent writers. A *get* request first tries to read the target record from the bucket through the helper function *get_from_bucket*, which is similar to *read_record*. It preserves the current version number first and then copies the bucket content to its local memory. On validation, the reader ensures the consistency of the result by checking the lock and version. Only when the lock is free and the version remains consistent with its local copy will the reader consider the result consistent. Otherwise, it will retry. If the target record does not exist in the bucket, the *get* will proceed to read from the chain. The concurrent chain ensures the consistent results of *get* requests in the face of concurrent *put*s.

4.2.3 Interleaving with Resize Operations. During resizing, Two-Phase Compaction ensures that (1) the effects of the latest write are correctly preserved, and (2) readers can always fetch the preserved records.

Similar to XIndex-R, XIndex-H achieves the first one by ensuring the following conditions: no successful *put* will be lost, and *no duplicate record* will be created. *No lost put* is ensured through the use of references during record movement. In the first phase of resizing, the old group is read- and update-only by setting the *frozen* flag (Line 3). The new group contains all the records' references (Line 11), which means the effect of a *put* request is visible to both the old group (by direct accessing) and the new group (through references). In the second phase, these references are atomically replaced with real values by *replace_pointer* (Line 16). The *rcu_barrier* before the second phase (Line 14) guarantees that all writers will only operate on the new group at the time of replacing pointers. The writer-writer coordination (Section 4.2.1) can correctly preserve the effects of concurrent writers on the same group. To ensure *no duplicate record*, the key is to initialize *tmp_group* after setting *frozen* flag and waiting for the first additional *rcu_barrier* (Line 4). After observing the *frozen* flag, all writers that need to insert new records will retry until the *tmp_group* is initialized (Line 24), which means no two writers will insert into the old group and the new group at the same time.

For readers, they can always fetch the target record from either the old or the new group during resizing. If it reads from the old group, the *tmp_group* pointer ensures that the inserted records are visible. If it reads from the new group, the references in the bucket array help the reader to get the latest value via the *ger_from_bucket* helper function.

4.2.4 Proof Sketch of Linearizability. XIndex-H is a linearizable data structure, which means "a *get*(*key*) must observe the latest committed *put(key,val)*." To provide correctness, XIndex-H follows the same three invariants as in XIndex-R: (I_1) If there is a *put(k, v)* committed, then there is exactly one record with key k in XIndex-H; (I_2) If there is a record with key k in XIndex-H, 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-H when a *get* commits, then the *get* returns the value of the record.

The carefully crafted initialization of *tmp_group* ensures *no duplicate record* (Section 4.2.3), thus further guarantees the exactly-one-record semantics (I_1). The *no lost put* property elaborated in Section 4.2.3 directly provides I_2. For I_3, in addition to the property that readers can always fetch the preserved records during resizing mentioned in Section 4.2.3, XIndex-H also ensures that *get* and *put* have the same lookup order (*bucket_arr* → *chain* → *tmp_group*). Since only the last place is insertable at the same time (*bucket* when *bucket* is uninitialized; *chain* when *bucket* contains a conflicting record and *tmp_group* is null; otherwise *tmp_group*), a *get* will not return EMPTY if another writer has successfully inserted a record with the same key.

5 PERSISTENCE AND CRASH RECOVERY

To persist data and recover from unexpected crashes, XIndex logs operations to persistent storage and periodically creates checkpoints to speed up recovery. Our design is similar to those of Masstree [45], Silo [56], and SiloR [62]. To avoid blocking, logs are flushed asynchronously, and checkpoints are created in parallel with put operations. This design avoids blocking, as seen in the write-ahead logging approach [47].

Each worker thread is associated with a log buffer and a log file. A background logging thread, co-located with the worker thread, periodically flushes log entries from the buffer to the file. Specifically, each worker thread puts its log buffer an entry containing a key, a value, and a timestamp before finishing the put operation. Timestamps are taken after put's linearization point, which is the release of the record lock (Line 19) when modifying the sorted array. Logging threads periodically wake up and flush buffered log entries in chronological order to log files. Checksums are used to guard against crashes during flushing. To reduce logging overhead, log files can be distributed on different disks, SSDs, or non-volatile memory.

Checkpoints are created by scanning all group nodes by background checkpoint threads. For XIndex-R, the scan is performed by the existing scan interface. For XIndex-H, we use a similar procedure that iterates group nodes one by one, reading all records in the bucket array. Two timestamps, $\mathsf{t}_{\mathsf{begin}}$ and $\mathsf{t}_{\mathsf{end}}$, are recorded before and after the scan, respectively. An RCU barrier is used before taking $\mathsf{t}_{\mathsf{end}}$ to ensure all committed puts whose values are scanned have log timestamps smaller than $\mathsf{t}_{\mathsf{end}}$. The scanned records and these timestamps are saved to persistent storage.

To reconstruct XIndex, we first compute the largest *safe timestamp* $\mathsf{t}_{\mathsf{safe}}$, i.e., a timestamp before which all committed puts are guaranteed to be persisted. $\mathsf{t}_{\mathsf{safe}}$ is computed by taking the minimum among the timestamps of each log file's last entry. Then, we find the latest *safe checkpoint*, i.e., a checkpoint that does not contains values committed after $\mathsf{t}_{\mathsf{safe}}$. This checkpoint is the one with the largest $\mathsf{t}_{\mathsf{end}}$ that is smaller than $\mathsf{t}_{\mathsf{safe}}$. Finally, we chronologically replay log entries created between the checkpoint's $\mathsf{t}_{\mathsf{begin}}$ and $\mathsf{t}_{\mathsf{safe}}$, since all puts committed before are guaranteed to be visible to the scan (Sections 3.2.4 and 4.2.4). Additionally, we can maintain the *version* (Algorithm 1) of each record in the log entry, so that we can skip log entries with *version* smaller than the one in the checkpoint. After replaying these log entries, a snapshot at $\mathsf{t}_{\mathsf{safe}}$ of the previous XIndex instance is restored.

6 EVALUATION

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

6.1 Setup

6.1.1 Benchmarks. We evaluated XIndex-R with complex workloads, including TPC-C (KV), YCSB, and micro-benchmarks (Section 6.2). TPC-C (KV) is a TPC-C-inspired benchmark for key-value stores. Same as [56], it uses different XIndex-R instances to represent different TPC-C tables and uses *get*, *put*, and *remove* to simulate TPC-C operations. Since XIndex-R does not support transactions, to avoid transaction aborts due to conflicts, we eliminate conflicts by manipulating each thread to execute remote transactions on one of their local warehouses. We assign eight distinct warehouses to each thread as their local warehouses for evaluation. 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 [16]. For microbenchmarks, we evaluate the performance under workloads with fixed read-write ratios (Section 6.2.2), under dynamic workloads (Section 6.2.3) and string keys (Section 6.2.4). We also analyze different factors that affect the performance in Section 6.2.5.

For XIndex-H, we tested it under read-only workloads (Section 6.3.1), and read-write workloads (Section 6.3.2) with different insert ratio to see the effectiveness of non-blocking resizing.

6.1.2 Datasets. All datasets used are listed in Table 3. For integer dataset, we use 8-byte keys generated from different distribution (*linear*, *normal* and *lognormal*) and also a real-world dataset (*osm* [16]). For string dataset, one is a synthetic random dataset (*random*), the other is a real-world URL dataset (*quote*). The maximum key length of *quote* is 128 bytes, and for URLs with length $n < 128$, we set $x_i = 0$ for $i > n$. Unless otherwise noted, the default dataset size is 200M and the value size is 8 bytes.

Name	Type	Description
linear	integer	Linear dataset with added noises
normal	integer	Normal distribution ($\mu = 0$, $\sigma = 1$), scaled to $[0, 1 \times 10^{12}]$
lognormal	integer	Lognormal distribution ($\mu = 0$, $\sigma = 2$), scaled to $[0, 1 \times 10^{12}]$
osm	integer	Longitude values of OpenStreetMap locations scaled to $[0, 3.6 \times 10^{9}]$
random	string	Randomly generated byte strings, denote as "r[key length]-[dataset size]"
quote	string	92 M URLs of quotes from Memetracker [40]. Average length is 62 bytes.

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

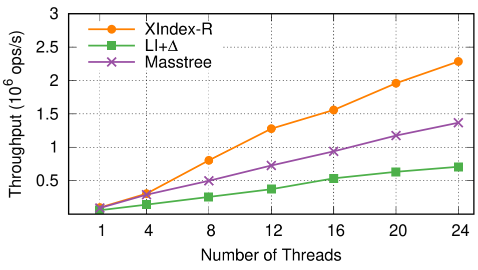


Fig. 8. TPC-C (KV) throughput.

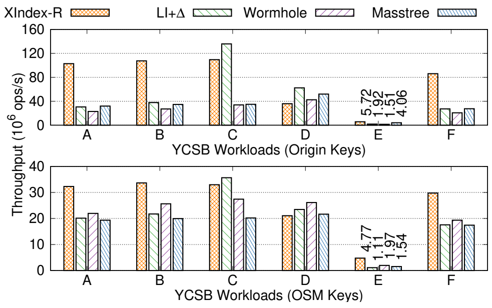


Fig. 9. YCSB throughput.

6.1.3 *Counterparts.* We compared XIndex-R with stx::Btree, Masstree, Wormhole, the learned index, "LI+Δ", and Alex. stx::Btree [3] is an efficient, but thread-unsafe B-tree implementation. Masstree [45] 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 [58] is a concurrent hybrid index structure that replaces B-tree's inner nodes with a hash-table encoded Trie. The learned index is the original learned index [35]. "LI+Δ" is the learned index attached with a Masstree as delta index, which buffers all writes. Alex [20] is a learned index that supports writes and can dynamically adjust itself to data distribution changes. Since it does not support concurrent accesses, we only compare it in dynamic workloads.

For XIndex-H, we compared it with the learned hash index, std::unordered_map, TBB::HashMap and "LH+Block". The learned hash index is the original learned hash index [35]. The std::unordered_map is a single-thread hash map implementation in C++ Standard Library. TBB::HashMap is a chaining-based concurrent unordered hash map provided by Intel Threading Building Blocks [51] library. "LH+Block" performs resizing by foreground threads and blocks all write requests while resizing.

6.1.4 *Configurations and the Test Bed.* We implement XIndex in C++, and configure 1 out of 12 threads as dedicated background threads. Background thread(s) repeatedly, with one second interval between runs, check all groups and the root to perform compaction and structure update (for XIndex-R) or resizing (for XIndex-H) accordingly. For XIndex-R, 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 string key support, the grouping error threshold et and the partial key length threshold pl are set to 50, 4 for *random* dataset and 500, 40 for *quote*. When persistence support is enabled, a checkpoint is created every 30s. The per-thread log buffer is flushed to storage every 200ms for safety. For XIndex-H, the thresholds for resizing, $upper\_lf$ and $lower\_lf$ is set to 1.5 and

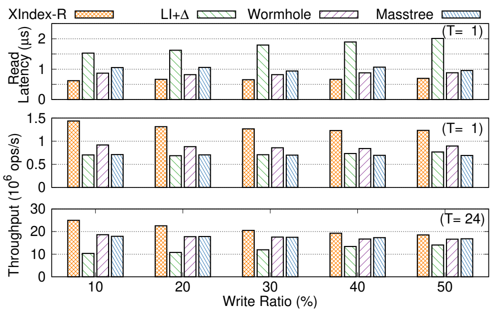


Fig. 10. Read-write throughput and read latency with the *normal* dataset. T indicates the number of threads.

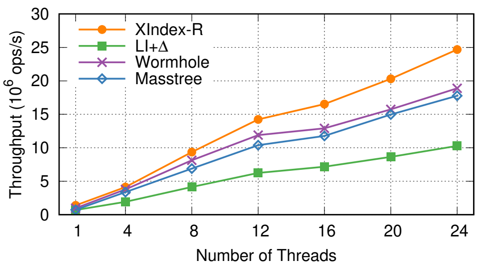


Fig. 11. Read-write throughput.

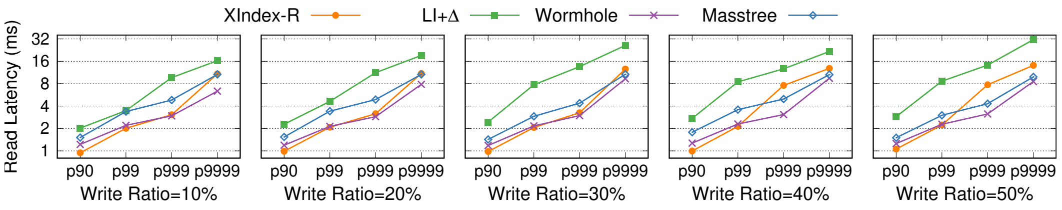


Fig. 12. Tail read latency under various write ratios with the *normal* dataset.

0.5, respectively. For the learned index, we test different configurations and pick the best one — 250k models in the 2nd stage.[8] For "LI+Δ", we use the same background threads as XIndex-R for compaction. For other indexes, 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 experiments run on a server with two 12-core Intel Xeon E5-2650 v4 CPUs, 128 GiB DDR4 memory, and one 300 GB SCSI hard drive (DELL PERC H330 Mini). The hyperthreading is disabled during evaluation.

6.2 XIndex for Range Index

6.2.1 Performance Overview.

TPC-C (KV). Figure 8 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-R 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 3.5).

YCSB. We use both the default data distribution as well as the *osm* dataset, and 24 threads for the experiment. The results are shown in Figure 9. For workload A, XIndex-R can achieve up to 3.2× and 4.4× performance improvement comparing with Masstree and Wormhole, respectively. For workload B, E, and F, XIndex-R also demonstrates superior performance advantage. This is because these workloads are read- and update-intensive. For workload C, which is read-only, XIndex-R is worse than "LI+Δ" by 19% because XIndex-R has model computation cost both in the root and

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

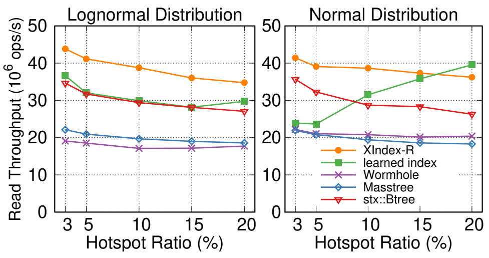


Fig. 13. 24-thread read throughput in skewed query distribution.

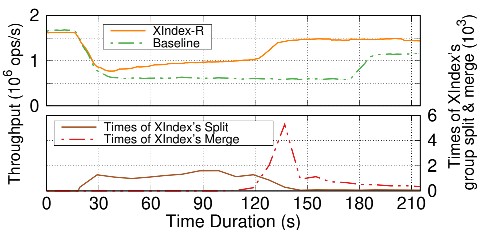


Fig. 14. Read-write throughput and group split/merge frequency when shifting data distribution from *normal* to *linear*.

groups. For workload D, XIndex-R 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-R is reduced.

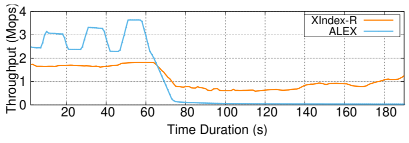
6.2.2 Performance with Writes. To further evaluate the performance of writes, we configure workloads with different read-write ratios. The ratio among different type of writes are constant (1:1:2 for insert, remove, and update) to keep the dataset size stable.

Scalability. Figure 11 shows the scalability with 10% writes using the *normal* dataset. Overall, XIndex-R achieves the best performance among all systems. With 24 threads, XIndex-R scales to 17.6× of its single-thread performance, which is 30% higher than Wormhole. "LI+Δ" has the worst performance because of its inefficient compaction, which severely degrades the read performance.

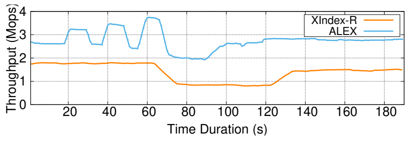
Varying write ratios. Figure 10 shows both throughputs and median latency with different write ratios with a single thread and 24 threads. XIndex-R has the best performance for all the listed write ratios, though the advantage tends to diminish with larger write ratios. For latency, XIndex-R achieves the lowest median latency as most requests (80%) can be served without accessing delta indexes. Tail latency is shown in Figure 12. In all write ratios, XIndex-R exhibits best read latency at the 90th and 99th percentile. However, as writes become dominant, more reads are likely to content with writes and background compaction, which causes p999 and p9999 read latency to increase. Nevertheless, XIndex-R always outperforms "LI+Δ."

6.2.3 Performance with Dynamic Workloads.

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 13 shows the throughput with different skewness levels under the *normal* and *lognormal* datasets. All systems except for the learned range 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 range index's large error bound in the hotspot, the learned range 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 range index due to improved locality.



(a) From *normal* to *lognormal*.



(b) From *linear* to *normal*.

Fig. 15. Read-write performance comparison under different data distribution shifts.

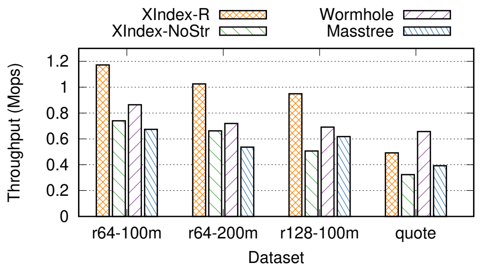


Fig. 16. Read-only single-thread performance with different string datasets.

Data distribution. We then evaluate XIndex-R under data distribution shifts. There are three stages in data distribution shift. First, we initialize each system with 50M records of one distribution and run a workload with a read-write ratio of 90:10. Then, we start shifting the data distribution by removing all existing keys while inserting 50M keys of another distribution. In this stage, the write ratio is 100% (half inserts and half removes). Finally, after keys of the new distribution are inserted, and old ones are removed, we resume the read-write workload (90:10) under the new dataset.

We first evaluate the performance of XIndex-R with and without group split/merge to demonstrate the effectiveness of the group split/merge under dynamic workloads. Figure 14 shows XIndex-R's throughput and the number of group split/merge. We use one worker thread and one background thread. XIndex-R without group split/merge is denoted as Baseline. Both XIndex-R and Baseline have similar performance in the first stage (0-20s). At the 20th second, they enter the second stage. Both throughputs begin to degrade due to the increase of the write ratio and the data distribution changes. While for XIndex-R, background threads begin to perform group split to reduce groups' error and delta index size, so we can see the throughput starts to increase at the 30th second. XIndex-R and Baseline enter the third stage at the 120th second and the 170th second, respectively. XIndex-R'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-R shows up to 140% performance improvement during and after the change of data distribution.

We then compare XIndex-R with ALEX, a non-concurrent updatable learned index that can adjust to data distribution changes. Overall, we find that XIndex-R exhibits better robustness than ALEX as it achieves similar throughput under various workloads, though ALEX can faster than XIndex in certain workloads. Figure 15 shows two workloads where XIndex-R's performance is stable while ALEX's varies significantly. In Figure 15a, when the data distribution begins to change (at the 60th second), the performance of ALEX degrades significantly (throughput drops from 2.2M to less than 100K), while XIndex-R can sustain around half of its previous throughput. This is because ALEX's RMI design allows unlimited stage depth, which can severely degrade lookup performance in the case of significant data distribution changes. Meanwhile, XIndex-R always has a two-stage architecture. It takes 75s to change its dataset, and ALEX takes more than 1400s. In Figure 15b, where the data distribution change is less drastic, ALEX can finish sooner than XIndex-R. Besides, in both workloads, ALEX is faster before the shift in data distribution for two reasons. First, XIndex-R has overhead for supporting concurrency while ALEX does not. Second,

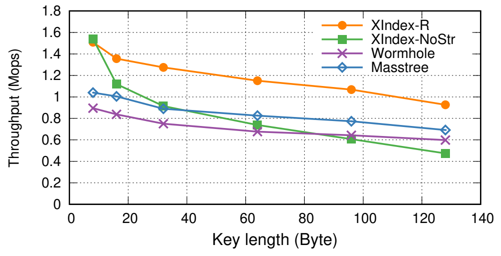


Fig. 17. Read-only single-thread performance with different string key length using *random* datasets.

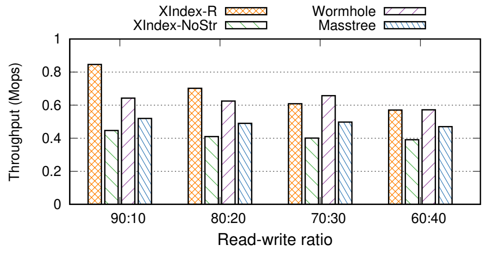


Fig. 18. Read-write single-thread performance with different read-write ratios using R128-100M string dataset.

ALEX uses an improved RMI designed while XIndex-R uses the original proposal[9]. It should be noted that in Figure 15a, XIndex-R's performance is not immediately recovered as background structure updates are still ongoing, and a trend of increasing throughout can be observed.

6.2.4 Performance with String Keys. We integrate the mechanisms for string keys (Section 3.4 and Section 3.5.3) into XIndex-R. Then we evaluate the performance of XIndex-R under string keys using read-only and read-write-mixed workloads. XIndex-R without string key support is denoted as XIndex-NoStr. Our current implementation does not support enabling runtime adjustment (Section 3.3) with the greedy grouping strategy (Section 3.4.2). Thus we disabled the runtime adjustment for these workloads. However, since there is no change in both data and query distribution, enabling runtime adjustment should not greatly affect the results.

String read-only. We first measure the single-thread throughputs of XIndex-R with read-only workloads using different datasets. As shown in Figure 16, XIndex-R achieves the best performance among all the indexes under *random* datasets but has fewer performance advantages under *quote* dataset. XIndex-R outperforms XIndex-NoStr, Masstree and Wormhole by up to 88%, 91% and 43% respectively under *random* datasets. Under *quote* dataset, XIndex-R still shows relatively good performance compared with XIndex-NoStr (1.5×) and Masstree (1.3×). However, XIndex-R's performance is worse than Wormhole by 25%. This is because *quote* dataset has more complex data distribution, leading to larger errors — 4.5× larger than the average error of *random* datasets. Also, the partial key length is 39 bytes for *quote* dataset, which is nearly 10× larger than that of *random* datasets.

Various key length. We then evaluate XIndex-R with various key lengths. Figure 17 shows the results under 100M *random* dataset, with key length ranging from 8 bytes to 128 bytes. XIndex-R shows considerable performance advantages for all key lengths. Its performance shows similar scalability in key length as Masstree and Wormhole. XIndex-NoStr only achieves good performance with short keys but suffers large performance degradation when keys are larger than 8 bytes: the throughout with 128-byte keys is only 30% of that with 8-byte keys. Compared with XIndex-NoStr, with 128-byte keys, XIndex-R still maintains 62% of its 8-byte throughout and outperforms XIndex-NoStr by 91%.

String read-write. We then investigate the performance under read-write workloads with different read-write ratios. We experiment with *random* dataset of 100M 128-byte keys. The write requests include in-place updates, inserts, and deletes, with a ratio of 2:1:1. As shown in Figure 18, XIndex-R

[9]The improved RMI can be used as a drop-in replacement for the model for both the group nodes and the root node.

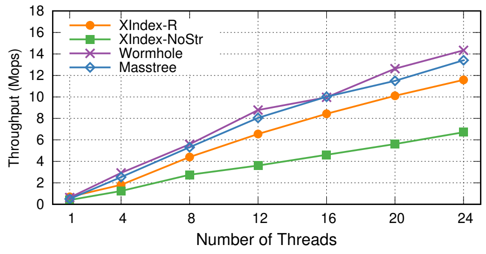


Fig. 19. Read-write performance scalability using R128-100M string dataset. The workload has a read-write ratio of 9:1.

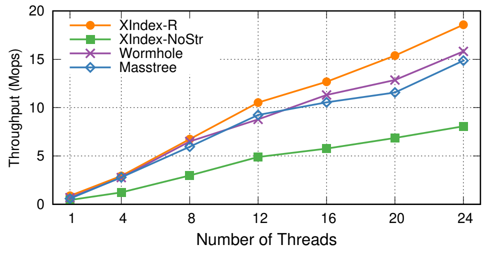


Fig. 20. Read-write performance scalability using R128-100M string dataset. The workload has a read-write ratio of 9:1 and all writes are in-place updates.

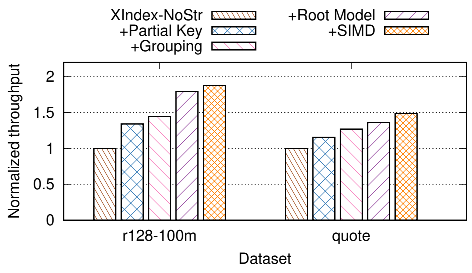


Fig. 21. Performance breakdown under string keys.

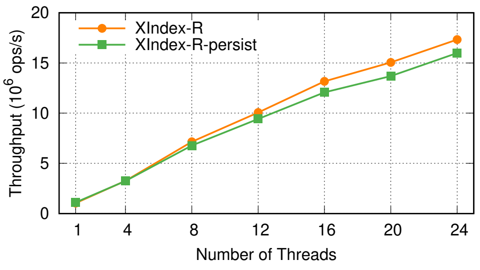


Fig. 22. Throughput of XIndex-R with and without persistence. The read-write ratio is 5:5.

maintains relatively good performance under read-write workloads. When there are 10% writes, XIndex-R has 89% better performance than XIndex-NoStr. When there are 40% writes, XIndex-R still has comparable performance as Wormhole, while being 45% better than XIndex-NoStr and 21% better than Masstree.

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

We also evaluate the scalability of XIndex-R with the same read-write ratio, but all writes are in-place updates. Figure 20 shows the results. XIndex-R exhibits good scalability in this workload since there is no need to perform compactions. The 24-thread performance can achieve 21× of its single-thread performance, which is 17% better than Wormhole.

Factor analysis. We then analyze the performance improvement brought by each design decision for string keys. We start from XIndex-NoStr where none of the proposed techniques is used, and then incrementally apply the partial key, the greedy grouping, piecewise linear models, and the SIMD optimization. Figure 21 shows the throughputs of applying each of the techniques under *random* and *quote* datasets. Throughputs are normalized to XIndex-NoStr.

We first apply the partial key in each group. With the partial key, XIndex-R improves 35% and 15% under *random* and *quote* datasets, respectively. For *random* dataset, the average key length for

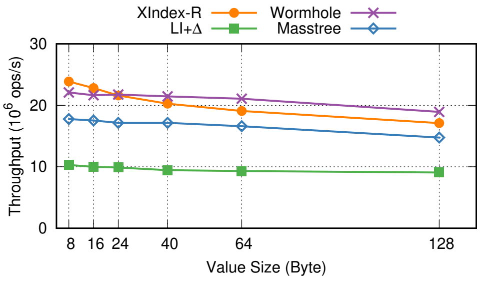


Fig. 23. Read-write throughput of various value sizes.

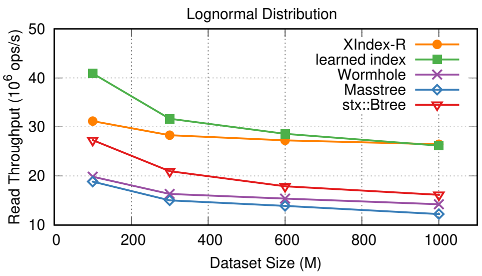


Fig. 24. Read throughput of various dataset sizes.

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

6.2.5 Other Factors.

Cost of Persistence. Figure 22 compares the performance of XIndex-R with and without persistence support, i.e., background logging and checkpointing, using the *normal* dataset and a various number of worker threads. Overall, persistence support does not incur significant performance penalties. On average, persistent XIndex-R achieves 92% of the non-persistent version's throughput. It takes XIndex-R about 14 seconds to create a checkpoint of 200 million key-value pairs (3.2 GB of data in total) for checkpointing. It takes 24 seconds to rebuild a XIndex-R instance from such a checkpoint and log files, each containing 7 million log entries on average for recovery.

Value size. We evaluate the performance of XIndex-R 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 23. With the increase of value size, the performance of all systems is reduced due to the large memory consumption. Nevertheless, XIndex-R 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 24 shows the performance of XIndex-R with different dataset sizes under the *lognormal* dataset using 24 threads. As dataset size increases, both the learned range index and XIndex-R show a large performance advantage over other systems. However, the performance of the learned range index degrades significantly because its error grows as the size increases. In contrast, XIndex-R adjusts its structure to maintain small model error bounds. Therefore, for large dataset sizes, XIndex-R can achieve similar performance with the learned range index.

6.3 XIndex for Hash Index

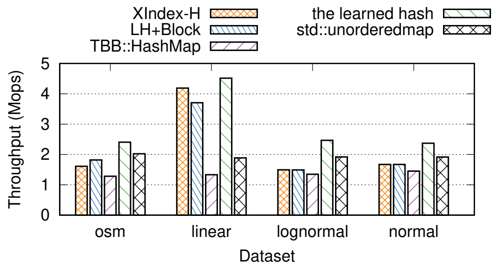


Fig. 25. Single-thread read-only throughput of various datasets.

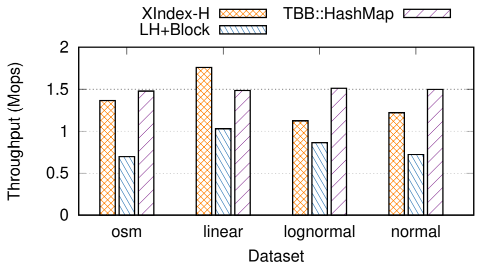


Fig. 26. Single-thread throughput under a workload with a read-insert ratio of 90:10. All indexes are initialized with 100M data, and are inserted with another 100M data.

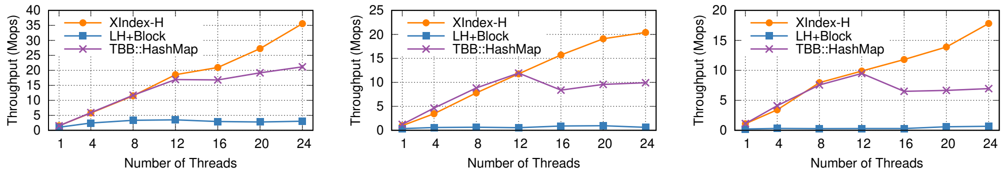
6.3.1 Performance with Reads. We first evaluate the read-only performance of XIndex-H under different datasets. The results are shown in Figure 25. XIndex-H's performance ties closely to the underlying datasets. It shows excellent performance when the model is well-learned and has less performance advantage otherwise. Under *linear* dataset, XIndex-H can achieve up to 3.1× and 2.2× performance advantage over TBB::HashMap and std::unordered_map, respectively. The conflict rate is significantly reduced to 0.7% thanks to learned models. While the conflict rate of std::unordered_map is up to 35%. Under the other datasets, the learned hash index does not show much performance improvement compared with std::unordered_map because the model cannot effectively reduce the conflict rate,[10] which is about 35% among all systems. Since XIndex-H has an additional cost for concurrency support, it performs slightly worse than the learned hash index and std::unordered_map. However, XIndex-H still shows slightly better performance compared with TBB::HashMap, which adopts the design of the split-ordered list [55]. The split-ordered list requires more pointer chasing during lookup, and a minimum of one pointer access is still needed even if there is no conflict. Furthermore, XIndex-H performs slightly worse than the learned hash index and std::unordered_map because of the concurrency support.

6.3.2 Performance with Writes.

Varying datasets. Figure 26 shows the single-thread read-insert throughput under various datasets. XIndex-H shows comparable performance compared with TBB::HashMap while achieving up-to 96% performance improvement compared with "LH+Block". With the help of non-blocking resizing, XIndex-H can continuously serve requests regardless of resizing. As a comparison, "LH+Block" blocks foreground operations for up to 160s (52s for model training and 108s for rehashing data). TBB::HashMap exhibits steady performance among different datasets because it adopts split-ordered list [55] to amortize the cost of resizing.

Scalability. We also investigate the scalability of XIndex-H with different insert ratios. Figure 27 presents the evaluation results. XIndex-H exhibits good scalability under all insert ratios. Under read-heavy workload (Figure 27a), XIndex-H can achieve 23× over its single-thread performance under 24 threads. It still scales well under write-heavy workload (Figure 27c), outperforming TBB::HashMap by 1.5× and "LH+Block" by 25× under 24 threads. While for TBB::HashMap, it scales only for the read-heavy workload. "LH+Block" has extremely poor scalability because of the blocking resizing.

[10]We didn't managed to reproduce the conflict rate in [35]. We have emailed the authors but received no response.



(a) 10% insert. (b) 50% insert. (c) 90% insert.

Fig. 27. Scalability under *linear* dataset with different read-insert ratio.

7 RELATED WORKS

There have been works extending or building systems upon the learned index for supporting writes and dynamic workloads. ALEX [20] supports writes by reserving slots in a sorted array for inserting new data. It allocates a new array and retrains the model synchronously when node fullness reaches the specified upper limit. Compare with its design, XIndex can have worse performance since it maintains standalone delta indexes. Since it might move a large nondeterministic portion of keys during insert, fine-grained concurrency control is thus challenging to implement, and ALEX does not support concurrent accesses. Furthermore, ALEX has a dynamic RMI structure that splits nodes according to its cost model, resulting in varying depth among model nodes. This design makes its adjustment in some cases more flexible than XIndex when data distribution changes. However, the RMI depth is not bounded, which might significantly impact lookups, writes, and structure adjustment, as confirmed by our evaluation AIDEL [42] 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. Same as ALEX, AIDEL does not support concurrency either, and operations are blocked during rearranging data and retraining models. It introduces a greedy strategy to partition data for models inside RMI according to the specific data distribution at bulk loading time. However, its structure is fixed during runtime even when data distribution changes, while XIndex can continuously adjust its structure according to runtime workloads. PGM-index [23] optimizes 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 [34] is a database that proposes to leverage the learned index for data indexing as well as for speeding up sorting and join. Bourbon [17] is a key–value store based on LSM-tree that leverages the learned index to improve efficiency for accessing SSTables. FITing-Tree [25] 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. For better build efficiency, Kipf et al. propose RadixSpline [33], which employs a bottom-up build method that only needs a single pass over the dataset. Flood [48] is a learned multi-dimensional index. It exploits ML methods to learn an optimal layout that divides d-dimensional data space into a grid of contiguous cells and uses linear models to speed up queries. LISA [43] is a learned index structure designed for disk-resident spatial data. It leverages learned models to generate data layouts in the disk. Compared with these works, XIndex is the only learned index structure that is optimized for string keys.

Classic concurrency techniques have long been used in concurrent data structures. Masstree [45] 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 [58] 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 [14, 15] 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 [4] 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 (COW) to allow reads to proceed with no synchronization overhead. The Bw-Tree [19, 41] is a completely lock-free B-tree and achieves its lock-freedom via COW and compare-and-swap (CAS) techniques. The Intel Threading Building Blocks [51] provides a chaining-based concurrent hash map. It leverages per-bucket lock to coordinate concurrent operations and exploits split-ordered list [55] to handle resizing. Libcuckoo [51] is a concurrent cuckoo hash table that adopts an optimistic design to minimize the locked critical section during writes. Clevel Hashing [13] is a lock-free concurrent level hashing for persistent memory. It leverages atomic primitives to enable lock-free operations and performs resizing in the background. Building upon existing works, XIndex leverages fine-grained locking and optimistic concurrency control to coordinate to individual records and combines Two-Phase Compaction with RCU mechanism to eliminate interference with queries and writes due to background operations.

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 [60] uses different storage schemes for hot keys and cold keys. Storage systems such as H-Store [18], COLT [53] are designed to detect the hotness and manage data accordingly in a self-tuning process.

8 CONCLUSION

In this paper, we introduced XIndex, a concurrent learned index library for both range index (XIndex-R) and hash index (XIndex-H). XIndex exploits machine learning models for fast indexing. It achieves high performance on the multicore platform via a combination of the innovative Two-Phase Compaction and a number of classical concurrency techniques. Extensive evaluations demonstrate that XIndex performs well for both integer key and string key workloads. For the range index, XIndex-R has a performance advantage by up to 3.2× and 4.4×, compared with Masstree and Wormhole, respectively. For the hash index, XIndex-H achieves up to 3.1× performance improvement compared with Intel TBB HashMap. XIndex is publicly available at https://ipads.se.sjtu.edu.cn:1312/opensource/xindex.git.

ACKNOWLEDGMENTS

This work is supported in part by the National Key Research and Development Program of China under grand No. 2020AAA0108500, the National Natural Science Foundation of China under grand No. No. 61902242, and the HighTech Support Program from Shanghai Committee of Science and Technology under grand No. No. 20ZR1428100. Haibo Chen (haibochen@sjtu.edu.cn) is the corresponding author.

REFERENCES

[1] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 363–375. https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[2] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Informatica* 9 (1977), 1–21. https://doi.org/10.1007/BF00263762

[3] Timo Bingmann. 2008. *STX B+ Tree C++ Template Classes*. http://panthema.net/2007/stx-btree

[4] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 521–534. https://doi.org/10.1145/3183713.3196896

[5] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[6] Philip Bohannon, Peter Mcllroy, and Rajeev Rastogi. 2001. Main-Memory Index Structures with Fixed-Size Partial Keys. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data* (Santa Barbara, California, USA) *(SIGMOD '01)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/375663.375681

[7] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters. In *Algorithms – ESA 2006*, Yossi Azar and Thomas Erlebach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 684–695. https://doi.org/10.1007/11841036_61

[8] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/1693453.1693488

[9] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. *SIGPLAN Not.* 45, 5 (Jan. 2010), 257–268. https://doi.org/10.1145/1837853.1693488

[10] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 181–190.

[11] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional Cascading: I. A Data Structuring Technique. *Algorithmica* 1, 2 (1986), 133–162. https://doi.org/10.1007/BF01840440

[12] Bernard Chazelle and Leonidas J. Guibas. 1986. Fractional Cascading: II. Applications. *Algorithmica* 1, 2 (1986), 163–191. https://doi.org/10.1007/BF01840441

[13] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *Proceedigns of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 799–812. https://www.usenix.org/conference/atc20/presentation/chen

[14] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 199–210. https://doi.org/10.1145/2150976.2150998

[15] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. *SIGPLAN Not.* 47, 4 (March 2012), 199–210. https://doi.org/10.1145/2248487.2150998

[16] OpenStreetMap contributors. 2019. *OpenStreetMap*. https://aws.amazon.com/public-datasets/osm Accessed: 2019-4-24.

[17] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (Virtual Event) *(OSDI '20)*. USENIX Association, 155–171. https://www.usenix.org/conference/osdi20/presentation/dai

[18] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1942–1953. https://doi.org/10.14778/2556549.2556575

[19] 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* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[20] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 969–984. https://doi.org/10.1145/3318464.3389711

[21] William B. Easton. 1971. Process Synchronization without Long-Term Interlock. In *Proceedings of the Third ACM Symposium on Operating Systems Principles* (Palo Alto, California, USA) *(SOSP '71)*. Association for Computing Machinery, New York, NY, USA, 95–100. https://doi.org/10.1145/800212.806505

[22] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 1998. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Vancouver, British Columbia, Canada) *(SIGCOMM '98)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/285237.285287

[23] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[24] Edward Fredkin. 1960. Trie Memory. *Commun. ACM* 3, 9 (Sept. 1960), 490–499. https://doi.org/10.1145/367390.367400

[25] 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* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1189–1206. https://doi.org/10.1145/3299869.3319860

[26] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. 2012. Foster B-Trees. *ACM Trans. Database Syst.* 37, 3, Article 17 (Sept. 2012), 29 pages. https://doi.org/10.1145/2338626.2338630

[27] Roberto Grossi and Giuseppe Ottaviano. 2015. Fast Compressed Tries through Path Decompositions. *ACM J. Exp. Algorithmics* 19, Article 3.4 (Jan. 2015), 20 pages. https://doi.org/10.1145/2656332

[28] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

[29] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.* 41, 3 (2018), 64–75. http://sites.computer.org/debull/A18sept/p64.pdf

[30] Stratos Idreos, Kostas Zoumpatianos, Subarna Chatterjee, Wilson Qin, Abdul Wasay, Brian Hentschel, Mike S. Kester, Niv Dayan, Demi Guo, Minseo Kang, and Yiyou Sun. 2019. Learning Data Structure Alchemy. *IEEE Data Eng. Bull.* 42, 2 (2019), 47–58. http://sites.computer.org/debull/A19june/p47.pdf

[31] Stratos Idreos, Kostas Zoumpatianos, Brian Hentschel, Michael S. Kester, and Demi Guo. 2018. The Data Calculator: Data Structure Design and Cost Synthesis from First Principles and Learned Cost Models. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 535–550. https://doi.org/10.1145/3183713.3199671

[32] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan

[33] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Portland, Oregon) *(aiDM '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3401071.3401659

[34] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research* (Asilomar, CA, USA) *(CIDR '19)*. http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf

[35] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 489–504. https://doi.org/10.1145/3183713.3196909

[36] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[37] Leslie Lamport. 1977. Concurrent Reading and Writing. *Commun. ACM* 20, 11 (Nov. 1977), 806–811. https://doi.org/10.1145/359863.359878

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

[39] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 447–461. https://doi.org/10.1145/3341301.3359628

[40] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. 2009. Meme-Tracking and the Dynamics of the News Cycle. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09)*. Association for Computing Machinery, New York, NY, USA, 497–506. https://doi.org/10.1145/1557019.1557077

[41] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *Proceedings of the 29th IEEE International Conference on Data Engineering* (Brisbane, Australia) *(ICDE '13)*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

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

[43] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2119–2133. https://doi.org/10.1145/3318464.3389703

[44] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, USA) *(FAST '16)*, Angela Demke Brown and Florentina I. Popovici (Eds.). USENIX Association, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[45] 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* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[46] Paul E. McKenney and John D. Slingwine. 1998. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*. Las Vegas, NV, 509–518.

[47] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (March 1992), 94–162. https://doi.org/10.1145/128765.128770

[48] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985–1000. https://doi.org/10.1145/3318464.3380579

[49] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[50] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2018. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 490–502. https://doi.org/10.1145/3267809.3267824

[51] Chuck Pheatt. 2008. Intel® threading building blocks. *Journal of Computing Sciences in Colleges* 23, 4 (2008), 298–298.

[52] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, Hash- and Space-Efficient Bloom Filters. In *Experimental Algorithms* (Rome, Italy) *(Lecture Notes in Computer Science, Vol. 4525)*, Camil Demetrescu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 108–121. https://doi.org/10.1007/978-3-540-72845-0_9

[53] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2007. On-Line Index Selection for Shifting Workloads. In *Proceedings of the 23rd International Conference on Data Engineering Workshops* (Istanbul, Turkey) *(ICDE '07)*. IEEE Computer Society, 459–468. https://doi.org/10.1109/ICDEW.2007.4401029

[54] Russell Sears and Raghu Ramakrishnan. 2012. BLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 217–228. https://doi.org/10.1145/2213836.2213862

[55] Ori Shalev and Nir Shavit. 2006. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *J. ACM* 53, 3 (May 2006), 379–405. https://doi.org/10.1145/1147954.1147958

[56] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[57] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2592798.2592804

[58] 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* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 18, 16 pages. https://doi.org/10.1145/3302424.3303955

[59] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[60] 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* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1567–1581. https://doi.org/10.1145/2882903.2915222

[61] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336. https://doi.org/10.1145/3183713.3196931

[62] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, CO, USA) *(OSDI '14)*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 465–477. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting