Zhang Z, Jin PQ, Wang XL *et al.* COLIN: A cache-conscious dynamic learned index with high read/write performance. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(4): 721–740 July 2021. DOI 10.1007/s11390-021-1348-2

COLIN: A Cache-Conscious Dynamic Learned Index with High Read/Write Performance

Zhou Zhang[1,2], Pei-Quan Jin[1,2,*], *Senior Member*, *CCF*, *Member*, *ACM*, *IEEE*, Xiao-Liang Wang[1,2]
Yan-Qi Lv[1,2], Shou-Hong Wan[1,2], *Member*, *ACM*, *IEEE*, and Xi-Ke Xie[1], *Member*, *ACM*, *IEEE*

[1] *School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China*
[2] *Key Laboratory of Electromagnetic Space Information, Chinese Academy of Sciences, Hefei 230026, China*

E-mail: zzwolf@mail.ustc.edu.cn; jpq@ustc.edu.cn; {wxl147, lvyanqi}@mail.ustc.edu.cn
{wansh, xkxie}@ustc.edu.cn

Received February 1, 2021; accepted June 13, 2021.

Abstract The recently proposed learned index has higher query performance and space efficiency than the conventional B+-tree. However, the original learned index has the problems of insertion failure and unbounded query complexity, meaning that it supports neither insertions nor bounded query complexity. Some variants of the learned index use an out-of-place strategy and a bottom-up build strategy to accelerate insertions and support bounded query complexity, but introduce additional query costs and frequent node splitting operations. Moreover, none of the existing learned indices are cache-friendly. In this paper, aiming to not only support efficient queries and insertions but also offer bounded query complexity, we propose a new learned index called COLIN (Cache-cOnscious Learned INdex). Unlike previous solutions using an out-of-place strategy, COLIN adopts an in-place approach to support insertions and reserves some empty slots in a node to optimize the node's data placement. In particular, through model-based data placement and cache-conscious data layout, COLIN decouples the local-search boundary from the maximum error of the model. The experimental results on five workloads and three datasets show that COLIN achieves the best read/write performance among all compared indices and outperforms the second best index by 18.4%, 6.2%, and 32.9% on the three datasets, respectively.

Keywords learned index, cache-conscious, insertion, dynamic index, read/write performance

1 Introduction

The index is an essential component of database systems, which can speed up data access. In 2018, Kraska *et al.*[1] proposed an innovative indexing approach called "learned index" in SIGMOD to integrate machine learning models with database index structures, which has attracted much attention from the database community. The learned index is an in-memory index used to retrieve in-memory data. Like the famous B+-tree, the learned index aims to support point queries and range queries efficiently. However, it uses machine learning models to learn data distribution and find the indexed items, instead of traversing the multi-level structure of the B+-tree. Notably, it uses a machine-learning model to predict the data position according to a given query. It then performs a local search, e.g., a binary search, based on the predicted position. Compared with the B+-tree, the learned index has higher query performance and lower space cost[1].

However, the original learned index has two shortcomings.

1) *Insertion Failure.* First, it fails to support insertions and can only work on static datasets. There are two reasons. One is that insertions can invalidate the existing learned models. The current models were built based on the old data distribution, and insertions will probably change the distribution feature of data and make the models invalid. The other is that, in the learned index, a learned model covers much more keys

Regular Paper
Special Section on AI4DB and DB4AI
The work was supported by the National Natural Science Foundation of China under Grant No. 62072419 and the Huawei-USTC Joint Innovation Project on Fundamental System Software.
*Corresponding Author

than a traditional B+-tree node. Thus, insertions with new keys will cause a lot of data shifts.

2) *Unbounded Query Complexity.* Another problem is that it lacks the guarantee of query complexity. The original learned index cannot ensure that each model can work well on a data segment because it does not limit the error boundary. Therefore, in the worst case that a model cannot accurately predict the data position, the learned index requires a local search over a large area, which will worsen the index's query performance.

A few previous studies have been presented to solve the insertion-failure problem, such as FITing-Tree [2] and PGM-index [3], which adopt some out-of-place insertion strategies but fail to offer high performance for both read and write requests. Moreover, none of the existing learned indices consider optimizing the use of the CPU cache, i.e., they are not cache-friendly. Learned indices use a model to predict the position of data in a node. Based on the predicted position, it performs a local search that ranges to the maximum error of the model, which usually covers multiple cachelines.

In this paper, we revisit the learned index structure and propose a new learned index called COLIN (CacheOnscious Learned INdex). COLIN aims to solve the two problems of the existing learned index, namely "insertion failure" and "unbounded query complexity". Unlike previous solutions that use an out-of-place insertion strategy, COLIN adopts an in-place approach to support insertions. It reserves some empty slots in data nodes and uses these slots to optimize the nodes' data placement. By using model-based data placement and cache-conscious data layout, the range of local searches and data shifts is limited within a cacheline. Briefly, we make the following contributions in this paper.

• We propose a new learned index called COLIN. COLIN has two new designs. First, COLIN uses a heterogeneous node structure containing learned nodes and simple nodes. Specially, we use learned nodes to offer high query performance and simple nodes to support out-of-bounds insertions. Second, we propose model-based data placement and cache-conscious date layout for learned nodes to decouple the local-search boundary from the model's maximum error, guaranteeing that the actual data position and the data position predicted by the model must be within the same cacheline (see Section 3).

• We design efficient algorithms for the operations in COLIN, including query, upsert, deletion, and bulk loading. We also present the structure-modified algorithms, including splitting/expanding a learned node and transforming a simple node into a learned node (see Section 4).

• We theoretically analyze the model's maximum error and the worst-case query cost of COLIN (see Section 5).

• We compare COLIN with the B+-tree and three state-of-the-art dynamic learned indices, namely FITing-Tree [2], PGM-index [3], and ALEX [4], on two tailor-made datasets and one real dataset. The results show that COLIN achieves better read/write performance than its competitors (see Section 6).

The remainder of the paper is structured as follows. Section 2 introduces the background and motivation of this paper. Section 3 presents the index structure and the node structure of COLIN. Section 4 describes the algorithms of COLIN. Section 5 analyzes the model's maximum error and the worst-case query cost of COLIN. Section 6 reports the experimental results. Finally, in Section 7, we conclude the entire paper.

2 Background

2.1 Learned Index

An index takes the keys as the input and outputs the position of the payloads corresponding to the keys in the storage medium. It seems that an index is similar to a machine learning model. The core idea of the learned index is to use machine learning models to replace the traditional index structure. To improve the accuracy and reduce the execution cost of the models, the learned index proposes a multi-layer model structure called RMI (recursive model index) [1, 5]. The structure of RMI is shown in Fig.1. At the top layer, RMI uses a neural network model to fit the overall distribution of the dataset, which is used to predict the approximate range of a given key. At the lower layers, RMI divides the dataset into many disjoint subsets and trains a linear regression model on each subset. RMI selects a lower model according to the output of the upper model and uses it to predict a more accurate position of the key. Finally, the learned index performs a local search at the predicted position to find the exact position of the key. RMI saves the maximum error for the models and performs binary searches within the error range. The learned index takes advantage of the characteristics of data distribution, which is ignored by traditional indices. As a result, it has better query performance and lower space cost.

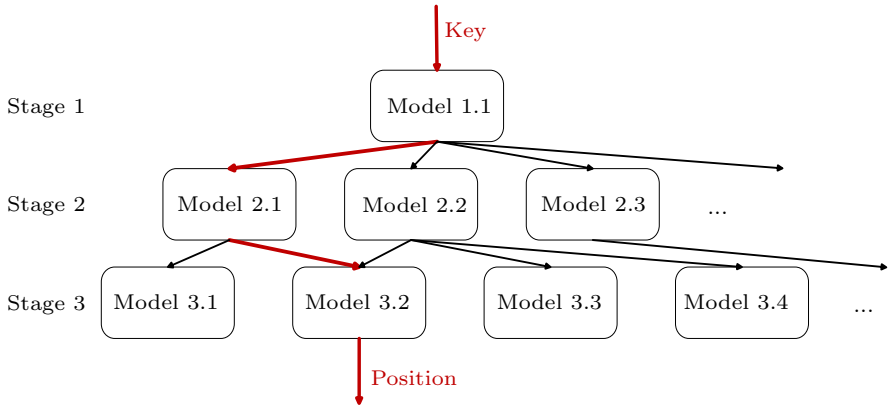


Fig.1. Structure of RMI.

However, the learned index can only work on static datasets. It is not easy for support insertions. First, the insertion can lead to the failure of the error bound of the model, resulting in the need for the model to be retrained. However, retraining a model is expensive, especially for neural networks. Second, a model in the learned index covers a large number of keys, which is far more than that in a B+-tree node. If a new key is inserted directly into the data with a dense layout, the cost of data shifting can be very high. Third, each model in the learned index can only cover a limited range, thereby it is necessary to consider the case that the inserted key is outside the index range.

Another problem with the learned index is that it cannot guarantee the worst-case query complexity. The learned index uses a naive partitioning strategy, which divides the dataset evenly according to the number of keys. Therefore, it cannot determine the maximum error of the model on each data segment. If the maximum error of a model is large, the cost of local search on its corresponding data segment is relatively high.

2.2 Dynamic Learned Indices

Dynamic learned indices aim to support insertions. Current representative dynamic learned indices include FITing-Tree [2], PGM-index [3], and ALEX [4]. Among them, FITing-Tree and PGM-index have bounded query complexity, while ALEX cannot provide a theoretical guarantee of query complexity. In terms of insertion strategy, FITing-Tree and PGM-index adopt out-of-place strategies, while ALEX adopts an in-place strategy.

To provide a theoretical guarantee of bounded query complexity, the key is to ensure that the height of the tree is bounded and that the maximum error of each model is bounded. FITing-Tree [2] and PGM-index [3] use the PLR (Piecewise Linear Representation) algorithm [6–8] to fit the dataset. The PLR algorithm can fit the dataset using a piecewise linear model in $O(n)$ time and ensures that the maximum error of each model is less than a user-given threshold. A difference between FITing-Tree and PGM-index is the type of inner nodes. FITing-Tree only uses learned nodes as data nodes. At the upper layers of the index, it returns to the traditional index method and employs a B+-tree as the index of data nodes. In contrast, PGM-index recursively executes the PLR algorithm, which also uses piecewise linear models at the upper layers of the index. Both of these methods ensure that the tree is a balanced tree, with a height of $O(\log n)$. In a nutshell, this type of learned indices can guarantee the worst-case query complexity.

ALEX [4] uses a cost model to divide the dataset based on the key space and obtain an unbalanced tree. Inside the node, ALEX does not save the maximum error of the model but adopts exponential search [9] without bounds. Therefore, ALEX does not have bounded query complexity.

To support insertions, FITing-Tree [2] adopts two strategies. The first strategy is similar to B+-tree, which inserts a new key directly into an ordered position. To avoid invalidating the model, each insertion increases the maximum error of the model [2, 10]. However, since the number of keys in a learned node is much more than that in a B+-tree node, each insertion can result in a high cost of data shifting. The second strategy is to use node-level buffers [2, 11]. Specifically, it provides an insert buffer for each data node. A new key is first inserted into a buffer. When a buffer is full, the keys in the buffer are merged with the keys in the learned node, and FITing-Tree uses the PLR algorithm [6–8] to

fit and partition these data again.

PGM-index [3] uses global multi-level buffers similar to LSM-Tree [12]. In detail, it builds multiple learned indices, whose capacities are different integer powers of 2. The data with new keys will be inserted into a global buffer. When the buffer is full, it will be merged with the smallest index. When the index size of one layer reaches the threshold, it will be merged with the index of the next layer. Each merge will result in a model retraining.

However, the out-of-place insertion strategies are problematic for learned indices. For FITing-Tree, there are problems with using either smaller or larger buffers. If the smaller buffers are used, the buffers will be filled quickly and model retraining and node splitting will occur frequently. If the larger buffers are used, the read and write costs of the buffers are higher. Fig.2 shows the impact of the buffer size on the insertion performance of FITing-Tree. Here, the maximum error of the model (ε) is set to 8, which is the optimal value of the parameter for insertion performance. As shown in Fig.2, regardless of the size of the buffer, the insertion performance of FITing-Tree is worse than that of B+-tree. For PGM-index, each query requires lookups to be performed in multiple indices, which can significantly reduce query performance.

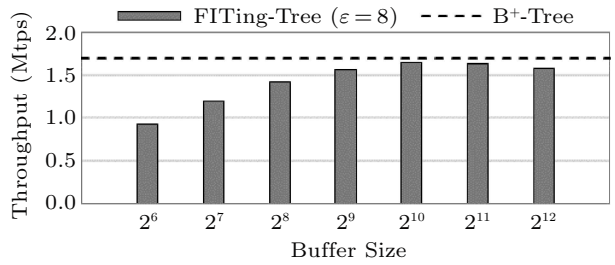


Fig.2. Impact of the buffer size on the insertion performance of FITing-Tree.

ALEX adopts an in-place insertion strategy. It uses gapped arrays [13] to reduce the cost of data shifting. When a node is built, it pre-allocates empty slots in the data slots and inserts the empty slots evenly into the data slots. However, gap arrays increase the prediction error of the model, and inserting new keys may make things worse. Therefore, ALEX uses exponential search [9] and does not use the model's maximum error for searching, i.e., it does not guarantee bounded query complexity. Our approach is also inspired by the idea of the gapped array. When building nodes, we place the data based on the position predicted by the model and limit the range of places where the data can be placed. This strategy allows our local search to be carried out within a bounded range.

Moreover, the existing learned indices are in-memory indices, but they are not cache-friendly. Previous work has shown that the latency of accessing data with CPU cache hits is much lower than that of accessing data with CPU cache misses [14]. However, the existing learned indices need to perform local searches larger than the size of a cacheline, which may cause multiple cache misses. For example, the recommended error threshold for PGM-index is 64, which means a 512-byte binary search needs to be performed. The value is a compromise. Because with smaller error thresholds, the height of the index may increase, which also increases the query cost. In this paper, we decouple the boundary of local search from the maximum error of models. Therefore, we can use a larger error threshold and perform the local search with the cacheline granularity.

2.3 Other Types of Learned Indices

There are also some other types of learned indices. Hadian and Heinis [15] proposed to use an auxiliary data structure called Shift-Table between the model and the data array, which stores the mapping of the predicted position of the key to the real position and can speed up the local search. However, Shift-Table does not support update operations and requires a lot of extra space. XIndex [16] is a concurrent learned index structure that uses a two-phase compaction algorithm to help unlocked node splitting. RadixSpline [17] uses spline interpolation [18] to fit the data and uses a radix tree to index the splines. Bilgram proposed a cost model to determine when to retrain the models [19]. SIndex [20] is specially optimized for string-type keys.

Another kind of learned indices is model-assisted indices, which aim to use machine learning models to assist rather than to replace traditional index structures, such as B+-tree. Llaveshi *et al.* proposed to use linear regression models to speed up the search within B+-tree nodes [21]. IFB-Tree [22] determines when building nodes whether each node is interpolation-friendly, i.e., the maximum error when using interpolation search is below a given threshold. If the node is marked as interpolation-friendly, the interpolation search is used in the node. MADEX [23] further reduces the error of interpolation search. However, these model-assisted indices are limited by the performance of the B+-tree itself, such as the high tree height and the large index size.

In addition to one-dimensional range indices, the idea of the learned index is also applied to spatial indices [24, 25], multi-dimensional indices [26, 27], and existence indices (e.g., Bloom Filters [1]). Furthermore, the learned index is only the tip of the iceberg of "AI4DB" [28]. Machine learning techniques are also applied to many other essential components of the database, such as query optimization [29], buffer management [30], workload forecast [31], and data synthesis [32].

3 Structure of COLIN

3.1 Overview

COLIN is a multi-tier learned index that uses models in both data and inner nodes to speed up searches. To ensure the retraining speed of the model, it only uses the linear regression model. Like PGM-index [3], COLIN calls the PLR algorithm to partition the dataset, train the models, and recursively perform this process to obtain the models of the upper layers. It adopts the OptimalPLR algorithm [8], which has $O(n)$ time complexity and can get the optimal partition result. In COLIN, the nodes in the upper layer are used to index the nodes in the lower layer, and the bottom nodes are responsible for storing data. We call them inner nodes and data nodes, respectively. Like most learned indices, COLIN stores data in an orderly manner to support range queries.

To solve poor insertion performance and low cache efficiency of the existing learned indices, COLIN mainly adopts the following three designs. First, COLIN uses an in-place insertion strategy and leaves some empty slots in the data slots. Using the enlarged data space, COLIN uses a model-based data placement strategy to arrange the data (see Subsection 3.3). This strategy is used not only when new data is inserted but also when the node is (re)built. All data needs to be rearranged. Second, COLIN uses a cache-conscious data layout to decouple the range of local searches to the maximum error of the model (see Subsection 3.3). With this strategy, COLIN limits the range of both local searches and data shifts to cacheline size. At the same time, a large model error threshold is maintained to ensure that the index structure is sufficiently flat. Third, COLIN uses a heterogeneous index structure (see Subsection 3.2). In addition to using model-accelerated learned nodes, it also introduces simple nodes, which are mainly used to solve out-of-bounds insertions and data overflow problems.

3.2 Heterogeneous Index Structure

COLIN involves two kinds of nodes: "learned nodes" and "simple nodes". The learned nodes use models to predict the position of data and perform local searches based on the predicted position. Each learned node corresponds to a bounded range. Simple nodes do not use models, thereby there is no bounded limit.

Different types of nodes play different roles in COLIN. The learned data nodes store most of the indexed data, namely the keys and the corresponding payloads. The payloads can be the actual value or a pointer to the value's position. The learned inner nodes are used to index the nodes in the lower layer. It stores keys and pointers to nodes, where a key represents the smallest key in the corresponding children.

We use simple nodes to support efficient out-of-bounds insertions in COLIN. As the coverage of a learned node is fixed, once the training of a learned node is completed, the range of keys covered by the node cannot be changed until a new training is performed. Therefore, the keys outside the range cannot be inserted into a learned node. Thus, we use simple nodes to cache out-of-bounds insertions. Besides, since COLIN uses a cache-conscious data layout, there may be data overflows. Simple nodes are also used to store overflowed data. Specifically, there are five use-cases that we need to allocate a simple node in COLIN, as shown in Fig.3.

Use Case 1: *As the Left or Right Buffer* (*Simple Data Node*). The left and the right buffers are unique components resident in COLIN, storing insertions smaller than the minimum key or larger than the maximum key of the learned nodes. They are accessed only when inserting and querying an out-of-bounds key.

Use Case 2: *As the Root Node* (*Simple Inner Node*). When the left/right buffer becomes full, it is converted into a learned data node. COLIN inserts the new keys directly into the root node. COLIN uses a simple node as the root node to support this kind of out-of-bounds insertions.

Use Case 3: *As a Temporary Child of the Root Node* (*Simple Inner Node*). When the root node is filled, we first merge the children with fewer layers to ensure the balance of the index tree. These children will be put in a simple node first, e.g., "node 1.3" in Fig.3. When the number of children in the simple node reaches a threshold, it will evolve into a learned node.

Use Case 4: *As an Overflow Block of the Learned Data Node* (*Simple Data Node*). There may be overflowed data in a learned data node. COLIN uses simple

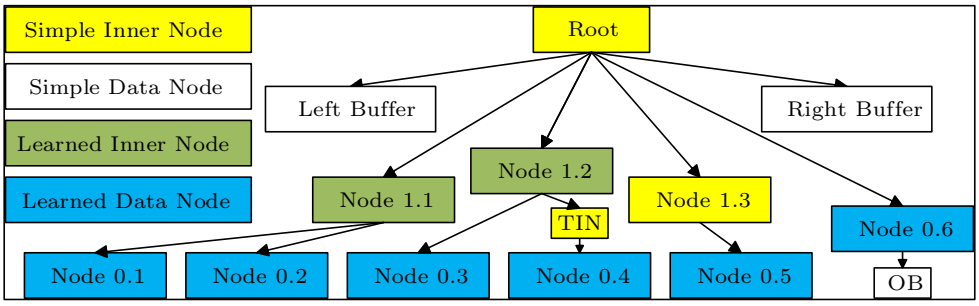


Fig.3. Index structure of COLIN. "TIN" represents a temporary inner node and "OB" represents an overflow block.

nodes to store overflowed data, e.g., the "OB" node in Fig.3. This component is accessed only when the overflowed data is inserted or queried, and there is only a small amount of overflowed data in the index.

Use Case 5: *As a Temporary Inner Node* (*Simple Inner Node*). Similarly, there may be overflowed data in a learned inner node. COLIN uses simple nodes as the temporary children of learned inner nodes to store overflowed data, e.g., the "TIN" node in Fig.3.

3.3 Cache-Conscious Learned Node

The learned nodes in COLIN use a model-based data placement strategy and cache-conscious data layout to decouple the local-search boundary from the model's maximum error. To make learned nodes cache-conscious, we enforce that the actual slot of a key to be in the same cacheline as the slot predicted by the model, which can guarantee that both local searches and data shifts in a node will involve only one cacheline.

3.3.1 *Learned Data Node*

The learned data node consists of metadata, a key array, a payload array, and an overflow block pointer array. Model is stored in the metadata and contains two member variables, *slope* and *intercept*, which are both double values. For a given key, COLIN uses (1) to calculate the predicted position.

$$pos = slope \times \text{key} + intercept. \tag{1}$$

The key array is an array of key types. When building a node, we multiply the number of keys by an amplification factor as the size of the key array. As shown in Fig.4, the key array is divided into multiple blocks, and each block is divided into multiple lines. In our implementation, we set each block to contain eight lines. Each line is the same size as a cacheline. We assume that the key size is 8 bytes and a cacheline size is 64 bytes, thereby each line contains eight data slots. There are some free slots in a learned data node. Since the coverage of each learned node is limited, we use an out-of-bounds key as a token to fill all free slots. This design can save the overhead of bitmap.

In the learned data node, we use a model-based data placement strategy to arrange the keys. When building a node, we use this strategy to place all the keys in the node. This strategy is also used when a new key is inserted into the node. For a key, we use the model to predict its position and place it on the predicted slot. If the slot is already occupied, we allow the key to deviate from the predicted slot, but not from the current line. Meanwhile, all keys in a line remain in order. If there is no space in the line, we insert the key into the overflow block. Using this strategy, the actual deviation

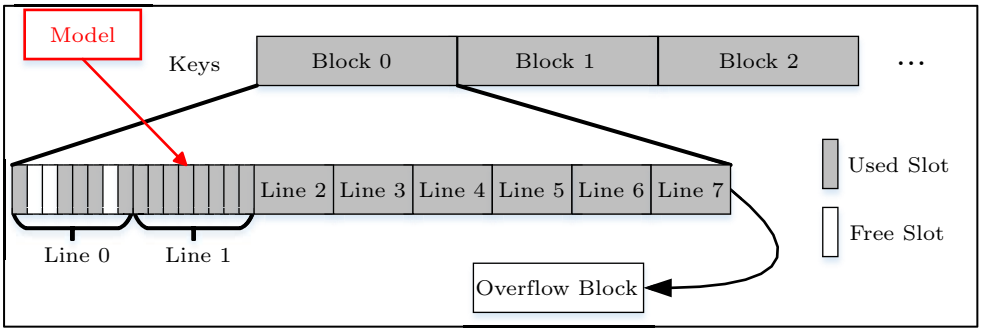


Fig.4. Data layout of a learned data node.

of the data position in the node is no more than the size of one line. Therefore, the range of local searches and data shifts corresponds to the size of the cacheline. However, due to the large error thresholds used when training the model, a learned node can still hold a large amount of data. Therefore, our strategy decouples the local-search boundary from the maximum error of the model.

In the learned data node, the payload offset in the array is the same as the key. The memory addresses of the key array and the payload array are aligned with the size of a cacheline, thereby we can identify which slots are in the same cacheline by the offset. Since the size of each line is limited, there may be overflowed keys. We use simple data nodes as overflow blocks. If a line is filled, subsequent keys inserted into the line will be inserted into an overflow block. In the learned data node, we reserve a pointer to the overflow block for each block. That is, lines in a block share an overflow block, which is to save the space cost of the pointers.

3.3.2 *Learned Inner Node*

A learned inner node is used to index multiple underlying nodes, which includes metadata, a key array, a node pointer array, and a bitmap. COLIN implements a base class of nodes and four node types as derived classes. The pointer array of inner nodes stores pointers of base class type, which can point to any type of nodes. In the key array, the stored key is the lower bound of the key range in the corresponding children. Similarly, in learned inner nodes, we also divide the key array into multiple lines, the size of which is a cacheline size, and align the array with the size of a cacheline. The difference is that there is no concept of the block in learned inner nodes.

A model-based placement strategy is also used in learned inner nodes. The difference is that there are no free slots in the key array of the learned inner node. We use the nearest key and pointer on the left to fill all the free slots. Fig.5(a) gives an example of an unfilled line. There are two free slots between the first key and the second key, which be filled with the first key and the first pointer. Note that the redundant filling can involve subsequent lines. Using this mechanism, local searches for lower bound lookups, which happen frequently at an inner node, can be limited to one cacheline.

When there is no enough space in a line, we use a simple inner node as a temporary inner node, as shown in Fig.5(b). Only one temporary inner node can be included in a line, and the node's position is on the leftmost side of the line. We use a bitmap to record which lines contain a temporary inner node. Note that the

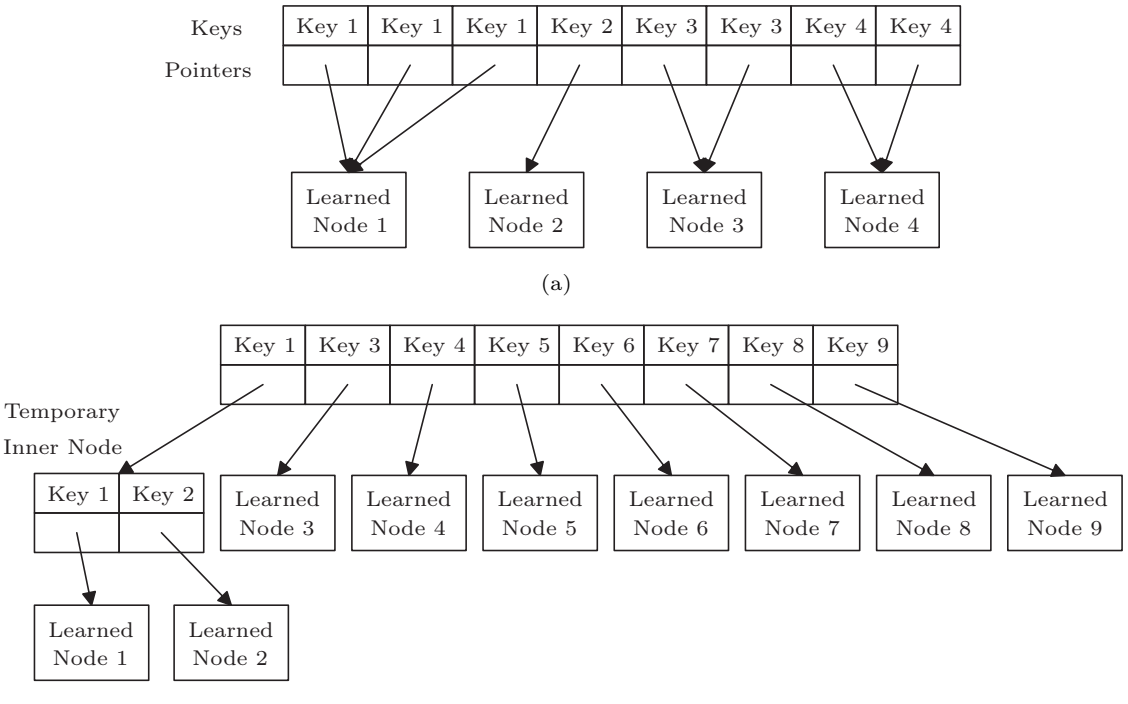


Fig.5. Example lines in learned inner nodes. (a) An unfilled line. (b) A filled line with overflowed keys.

bitmap is only used to recycle temporary inner nodes when the learned inner node is abandoned, and queries and insertions do not require querying the bitmap.

3.4 Simple Node

Models are not allowed in simple nodes. A naive approach is to allocate a fixed size of contiguous storage space and use binary lookup, as with B+-tree nodes. However, a simple node must guarantee enough space to store the worst-case data overflow. As a result, allocating a fixed size of space wastes space and leads to high read and write overhead for simple nodes. One solution is to use a linked list structure to support capacity scaling. However, the overhead of a linked list structure increases linearly with the amount of data and results in the index with unbounded query complexity.

The simple nodes in COLIN adopt a scalable structure, as shown in Fig.6(a). Nodes will expand from a one-level structure to a three-level structure according to the number of keys. When the node has only one level, the keys are placed in the L1 array, which together with the metadata takes up the size of two cachelines. In our implementation, assuming the key size is 8 bytes, the capacity of the L1 array is 12. When the node expands into a two-level structure, we request a new memory space as the L2 array. In this case, the L1 array is the index of the L2 array. In our implementation, the size of the L2 array is eight times that of the L1 array. The i-th element in the L1 array equals the $(8 \times i)$-th element in the L2 array. The same is true of the three-level structure, as shown in Fig.6(b).

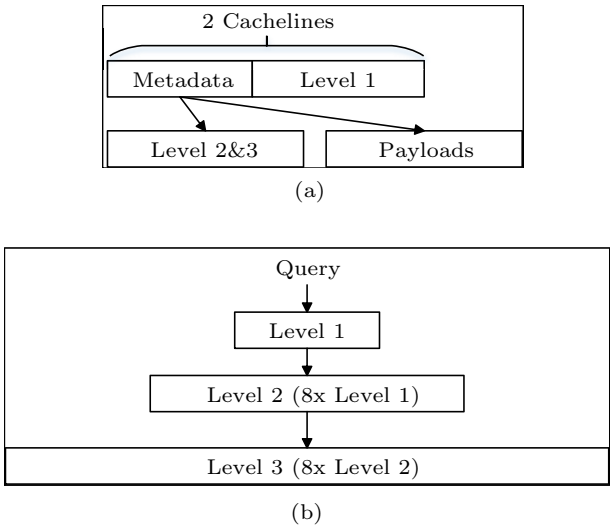


Fig.6. Structure of a simple node. (a) Structure of a simple data node. (b) Query in a three-level simple node.

The keys in any level are ordered, and a key outside the range is used to represent the free slot. To reduce the data shifting cost during insertion, we allow the L1 array and the L2 array to be filled, but the L3 array is only allowed to be filled to 75%. L2 array, L3 array, and payload array are aligned with the size of a cacheline. The structure of a simple inner node is similar to that of a simple data node, except that the payload array is replaced by a pointer array.

COLIN's simple nodes have good performance and space efficiency when dealing with a variety of situations. For example, when we use a simple node as an overflow block and there is only a small amount of overflowed data, COLIN will cost very little space. If there is a large amount of overflowed data, COLIN can also provide enough space. At the same time, COLIN can also maintain high query efficiency. Because the upper-level data is used as the index, only one cacheline in L2 and L3 is searched.

4 Operations in COLIN

4.1 Query Operations

Point Query. We reserve the minimum key and the maximum key of the index (not including the left and right buffers) in COLIN's metadata. When a query request arrives, the index first determines whether the target key is in the index's coverage. If so, the *Find* algorithm of the root node is called; otherwise, the *Find* algorithm of the left/right buffer is called. The *Find* algorithm is implemented by a virtual function. The algorithm to be called will be selected according to the type of nodes.

Algorithm 1 is the *Find* algorithm of learned inner nodes. We first use the model to calculate the predicted position of the key and confirm the line on which the key is located. Then, we compare the target key with the key at the predicted position. If the target key is smaller, we search to the left, and vice versa. When a key smaller than the target key is encountered, or when the line boundary is searched, the search stops, and the children's *Find* algorithm is invoked. In any of COLIN's learned nodes, the *Find* algorithm only needs to search the range of a cacheline. In addition, as shown in Fig.5(b), the child of a learned inner node may be either a learned node or a simple node (acting as a temporary inner node).

Algorithm 1. Find (Learned Inner Node)

```
Input: key
Output: payload
1:  pos ← predictPos(key)
2:  line ← [pos/8]
3:  if key_array[pos] > key /* Search to the left */
4:      while pos × 8 = line /* Search in the line */
5:          if key_array[pos] ⩽ key
6:              return node_array[pos].find(key)
7:          end if
8:          pos = pos − 1
9:      end while /* Arrive at the boundary of the line */
10:     return node_array[pos + 1].find(key)
11: else /* Search to the right */
12:     ···
13: end if
```

Algorithm 2 shows the *Find* algorithm of learned data nodes. Similarly, we calculate the predicted position, locate the line, and search in the line. If the target key is smaller than the key at the predicted position, we search to the left. If a free slot is found, it means the target key does not exist. This is because COLIN uses a model-based data placement strategy, which ensures that if a key exists in the node, there must be no free slot between its real position and the position predicted by the model. If the target key is searched, then the payload corresponding to the target key is returned. If a key smaller than the target key or a line boundary is found, then the *Find* algorithm of the overflow block corresponding to the current block needs to be executed if any.

Algorithm 2. Find (Learned Data Node)

```
Input: key
Output: payload
1:  pos ← predictPos(key)
2:  line ← [pos/8]
3:  if key_array[pos] > key /* Search to the left */
4:      while pos × 8 = line /* Search in the line */
5:          if key_array[pos] is free
6:              return null
7:          else if key_array[pos] = key
8:              return payload_array[pos]
9:          else if key_array[pos] < key
10:             break
11:         end if
12:         pos = pos − 1
13:     end while /* Arrive at the boundary of the line */
14:     overflow_block ← getOverflowBlock(pos + 1)
15:     return overflow_block.find(key)
16: else /* Search to the right */
17:     ···
18: end if
```

Algorithm 3 shows the *Find* algorithm on a three-level simple data node. The L1 array is first searched sequentially until a key larger than the target key is found. Since the keys in the upper layer are a sample of the keys in the next layer, we can calculate the key's position in the next layer, as shown in line 3. A similar search is then performed in the L2 array to get the search starting point for the L3 array. Finally, a sequential search is performed in the L3 array. If the target key is found, the corresponding payload is returned; otherwise, the algorithm returns that the key does not exist. The *Find* algorithm for simple inner nodes is similar to that for simple data nodes except that the algorithm's return is changed to perform the *Find* algorithm on the child.

Algorithm 3. Find (Three-Level Simple Data Node)

```
Input: key
Output: payload
1:  for pos ← 0 to L1_size − 1 /* Search in L1 */
2:      if L1_array[pos] > key
3:          pos ← (pos − 1) × 8 /* Calculate the start pos in L2 */
4:          break
5:      end if
6:  end for
7:  for pos ← pos to pos + 7 /* Search in L2 */
8:      if L2_array[pos] > key
9:          pos ← (pos − 1) × 8 /* Calculate the start pos in L3 */
10:         break
11:     end if
12: end for
13: for pos ← pos to pos + 7 /* Search in L3 */
14:     if L3_array[pos] = key
15:         return payload_array[pos]
16:     else if L3_array[pos] > key or L3_array[pos] is free
17:         return null
18:     end if
19: end for
```

Range Query. All keys in COLIN are ordered in nodes. For range queries, COLIN takes two parameters, lower bound and upper bound. It first finds the position of the lower bound, similar to point queries. Then, it scans the data nodes sequentially until the key is larger than the upper bound. In learned data nodes, we scan the range of one block at a time. If the block has a corresponding overflow block, we scan both the block and the overflow block. When the scan operation has processed the last block in the node, we continue to scan the block in the next node. We maintain pointers to the left and right brothers in the learned data nodes to support cross-node scanning.

4.2 Upsert

COLIN provides an *Upsert* interface, which accepts a key and a payload as inputs. If the key already exists in the index, COLIN updates the payload to the new payload. Otherwise, the key and the payload will be inserted into the index. First, we determine if the key belongs to the coverage of the index. If so, we invoke the *Upsert* algorithm of the root node; otherwise, we invoke the *Upsert* algorithm of the left/right buffer. The *Upsert* algorithm for inner nodes is the same as the *Find* algorithm. We find the data node corresponding to the key through the root node and inner nodes, and then execute the *Upsert* algorithm on the learned data node.

In a learned data node, a model-based insertion is performed. The key is preferred to be inserted into the predicted position of the model. As shown in Algorithm 4, first, the model is used to predict the position of the key. Then, if the position is not a free slot, a local search is performed to find the real position of the key in the ordered array. Note that if the position is a free slot, there is no need to perform the local search. The local search stops when a free slot or line boundary is encountered. If the target key is found in the array, the corresponding payload is updated and the algorithm is terminated, as shown in lines 4–6. Otherwise, we look for a free slot in the array. We start with the target position and search both left and right sides at the same time. The search range is the current line and the return is the position of the nearest free slot. If there is a free slot in the current line, we empty the slot at the target position by shifting the data, and insert the key and payload, as shown in lines 10–13. Otherwise, we perform the *Upsert* algorithm on the overflow block.

The *Upsert* algorithm of simple data nodes is described below with a two-level structure as an example. First, the search starting point for the L2 array *pos_l2* is obtained from the L1 array. If there is a free slot in range [*pos_l2*, *pos_l2*+7], we insert the key and keep it in order. Otherwise, we look farther for a free slot. In this case, some keys at the $8k$ (k is an integer) position in the L2 array are changed, thereby the keys in the L1 array need to be updated. If the L2 array is filled, the node is expanded to a three-level structure. To avoid excessive data shifting costs, we allow up to 75% padding for the L3 arrays.

In this subsection, we only describe the *Upsert* algorithm without structure-modified operations. The structure-modified operations resulting from insertions are described in Subsection 4.4.

Algorithm 4. Upsert (Learned Data Node)

Input: *key*, *payload*

```
1:  pos ← predictPos(key)
2:  if key_array[pos] is not free
3:    pos ← local_search(pos, key) /* Find the real pos */
4:    if key_array[pos] = key
5:        payload_array[pos] ← payload /* Update */
6:        return
7:    end if
8:  end if
9:  free_pos ← getFreePos(pos) /* Find a free slot */
10: if free_pos ⩾ 0 /* A free slot exists */
11:   shift keys and payload from pos to free_pos
12:   key_array[pos] ← key /* Insert */
13:   payload_array[pos] ← payload
14: else /* The line is full */
15:   overflow_block ← getOverflowBlock(pos)
16:   overflow_block.insert(key, payload)
17: end if
```

4.3 Deletion

To delete a key, we first search the inner nodes to find the data node on which the key resides. If the key is in a simple data node, then the key is simply removed, i.e., the slot in which the key resides is marked free. If the key is in a learned data node, we use a deletion token instead of the key. Like the free token, the deletion token is an out-of-bounds key that is not equal to the free token. During a local search, if a deletion token is found, the slot is skipped, and the search continues. When looking for a free slot in the *Upsert* algorithm, the deletion token is treated as a free token, allowing new keys to be inserted into the slot.

If many keys are deleted from a node, space can be wasted. In particular, there may be many keys stored in overflow blocks. These keys should be moved into the free slots in the node to make full use of the node space. We record the number of deletion tokens in the header of the node. If the number of deletion tokens exceeds a threshold and the amount of overflowed data is also greater than a threshold, we perform a batch replacement of the data in the overflow blocks. Specifically, we scan keys in both lines and overflow blocks, and if there is a deletion token in a line and there is an overflowed key belonging to that line in the overflow block, we insert the key from the overflow block into the line. In our implementation, the deletion token threshold and the overflow data threshold are both set to 10% of the node size.

4.4 Bulk Loading

Like FITing-Tree [2] and PGM-index [3], COLIN does not support zero-based insertions. When building an index, users need to provide enough data to train a piecewise linear model. COLIN uses the OptimalPLR algorithm [8] to train the models and partition the dataset. Then, the models and data are used to build the learned data nodes.

Algorithm 5 shows the *Build* algorithm of the learned data node. First, we use a parameter to enlarge the size of the array to provide some free slots to support subsequent insertions. Then, the two parameters of the model, which are *slope* and *intercept*, are enlarged at the same scale. After that, we need to perform model correction to avoid wasting space. The model correction algorithm will be described later. Then, all slots in the key array are filled with a key that is not within the scope of the current node. Finally, model-based insertions are performed for all keys and payloads (see Algorithm 4).

Algorithm 5. Build (Learned Data Node)

Input: *slope*, *intercept*, *keys*, *payloads*, *number*

1: $size \leftarrow number/min\_fill\_rate$
2: $slope \leftarrow slope/min\_fill\_rate$
3: $intercept \leftarrow intercept/min\_fill\_rate$
4: $model\_correction(slope, intercept, keys)$
5: **for** $i \leftarrow 0$ to $number - 1$
6: $\quad key\_array[pos] \leftarrow free\_token$
7: **end for**
8: **for** $i \leftarrow 0$ to $number - 1$
9: $\quad insert(keys[i], payloads[i])$
10: **end for**

When using the *OptimalPLR* algorithm [8] to partition datasets, parameter ε is required to represent the maximum error of the model. In COLIN, this value can be set to a large one, such as 256. Sometimes, using a model provided by *OptimalPLR* directly can result in a waste of space. For example, the predicted position of the minimum key in a node is 256, which achieves the maximum error. Then, since COLIN uses a model-based data placement strategy, the slot before 256 will always be free. We use a model correction as shown in Algorithm 6 to avoid this space waste. First, the predicted positions of the minimum key and the maximum key are obtained, and their coordinates on the two-dimensional plane are obtained. Then, if the position of the minimum key is to the right of the left end of the array, we move its position to the left end of the array and update the coordinate. The coordinate of the maximum key is treated the same. Finally, using the new two coordinate points, we can calculate a new linear model. Note that the slope of the corrected model can only increase, thereby there is no line in the node that will become more crowded.

Algorithm 6. Model Correction

Input: $min\_key, max\_key, slope, intercept, size$

1: $min\_pos \leftarrow predictPos(min\_key)$
2: $max\_pos \leftarrow predictPos(max\_key)$
3: **if** $min\_pos > 0$ and $max\_pos < size - 1$
4: $\quad slope \leftarrow (size - 1)/(max\_key - min\_key)$
5: $\quad intercept \leftarrow 0 - slope \times min\_key$
6: **else if** $min\_pos > 0$
7: $\quad slope \leftarrow max\_pos/(max\_key - min\_key)$
8: $\quad intercept \leftarrow 0 - slope \times min\_key$
9: **else if** $max\_pos < size - 1$
10: $\quad slope \leftarrow (size - 1 - min\_pos)/(max\_key - min\_key)$
11: $\quad intercept \leftarrow size - 1 - slope \times max\_key$
12: **end if**

After building the learned data nodes, we continue to use the *OptimalPLR* algorithm [8] to train a piecewise linear model using the minimum key of each node as a new dataset and build learned inner nodes. This process is performed recursively until the number of nodes is less than the number of a two-level simple node. Finally, these nodes are used to build the root node. The first half of the *Build* algorithm for learned inner nodes is the same as lines 1–4 in Algorithm 5. In the latter part, the key is placed using the model-based placement strategy described in Subsection 3.3.

4.5 Structure-Modified Operations

The structure-modified operations in COLIN will be triggered when a node becomes full. For learned nodes, there are two structure-modified operations: node splitting and node expansion. The simple nodes used as left and right buffers will evolve into learned nodes. For the root node, which is also a simple node, we perform a root neatening. Note that the simple nodes used as overflow blocks or temporary inner nodes are the components of learned nodes. They will be reclaimed when a learned node triggers a structure-modified operation.

Learned Nodes. For learned data nodes, we record the number of keys and the number of overflowed keys. We set a maximum fill ratio and determine whether the node reached that ratio after each insertion. If achieved, then we decide whether to trigger node splitting or node expansion based on the proportion of overflowed keys.

If a lot of keys are overflowed, we think that the current model has poor ability to fit the data in the node and needs to be updated, thereby node splitting is performed. If there are fewer keys overflowed, we think the current model can continue to work and therefore perform node expansion. Besides, if an overflow block is filled, a node splitting will also be triggered.

For node expansion, we execute Algorithm 5 on the model and data in the node without retraining the model. Algorithm 7 describes the splitting of learned data nodes. When COLIN receives the signal of node splitting, it traverses the inserted node again and records the traversed path. Then, the *OptimalPLR* algorithm is used to retrain the data in the node and build new nodes. After that, we upsert the new nodes into the parent node. The node splitting may lead to cascade splitting. The splitting algorithm of learned inner nodes is similar to that of learned data nodes, as shown in lines 7–9. Inserting into the root node may trigger root neatening, which will be described later.

Algorithm 7. Node Split

```
Input: key
1:  traversal_path ← getTraversalPath(key)
2:  data_node ← traversal_path.pop_back()
3:  models ← OptimalPLR(data_node)
4:  nodes ← build(models, data_node)
5:  while inner_node ← traversal_path.pop_back()
6:      split_flag ← inner_node.upsert(nodes)
7:      if split_flag /* Trigger cascade splitting */
8:          models ← OptimalPLR(inner_node)
9:          nodes ← build(models, inner_node)
10:     else /* Split end */
11:         return
12:     end if
13: end while /* Split involves the root node */
14: neaten_flag ← root.upsert(nodes)
15: if neaten_flag
16:   neaten_root()
17: end if
```

Upsert of learned inner nodes can only be caused by splitting on the children (line 6), which has two cases. Assuming that node A splits into A_1 and A_2, the key of A_1 must equal that of A. First, we update all slots occupied by A to A_1. *Case* 1. If A occupies multiple slots in the node, such as five slots, and A_2 is predicted to be in the third slot, then the third to the fifth slots will be updated to A_2. *Case* 2. Node A only has one slot, thereby we need to look for free slots in the line, i.e., other keys that occupy multiple slots. If there are no free slots, we move the keys in the line to the left and squeeze the leftmost key into the temporary inner node. Simple inner nodes have the same *Upsert* algorithm as simple data nodes.

Left/Right Buffer. If the left/right buffer is filled, node evolution will be triggered. We train the data in the buffer and build learned data nodes. Then, we insert the new node into the root node, as shown in case 1 in Fig.7.

Root Node. When the root node is filled, we need to perform a root neatening. Since a node with out-of-bounds insertions is merged into the root node, COLIN may be in an unbalanced state. The purpose of root neatening is to maintain the balance of COLIN and make enough free slots. In the root node, we maintain the maximum depth of the index. We scan from the left and the right to the middle and merge the nodes whose depth is below the maximum depth into a simple inner node, as shown in case 2 in Fig.7. Note that only the nodes with the same depth will be merged, and there is only one simple inner node for a specific depth. When a simple inner node is full, we evolve it into a learned inner node, as shown in case 3 of Fig.7. Finally, if the root node becomes full and there are no nodes to merge, we merge all the nodes with the maximum depth in the root node into a new learned inner node and increase the maximum depth by 1, as shown in case 4 of Fig.7.

5 Analysis of COLIN

5.1 Maximum Error of Models

When performing bulk loading or structure-modified operations, COLIN uses the OptimalPLR algorithm [8] to train the models. The algorithm guarantees that the maximum error of the models is less than parameter ε. By using a larger ε, a model can cover more data. Since we have decoupled the local-search boundary with the maximum error of the model, in order to reduce the height of the index tree, it seems that the maximum error threshold ε should be set as big as possible. However, a larger ε may result in more overflowed data. To make COLIN work properly, we need to ensure that the size of the overflow block is large enough when building the node. Since the structure of the overflow block has an upper limit of capacity, we can get the maximum value of ε by calculating the amount of overflowed data in the worst case.

Assume that the initial fill rate is fr when building nodes. The size of block B is b, the size of a line is l, and the starting offset of B is sp. When building nodes,

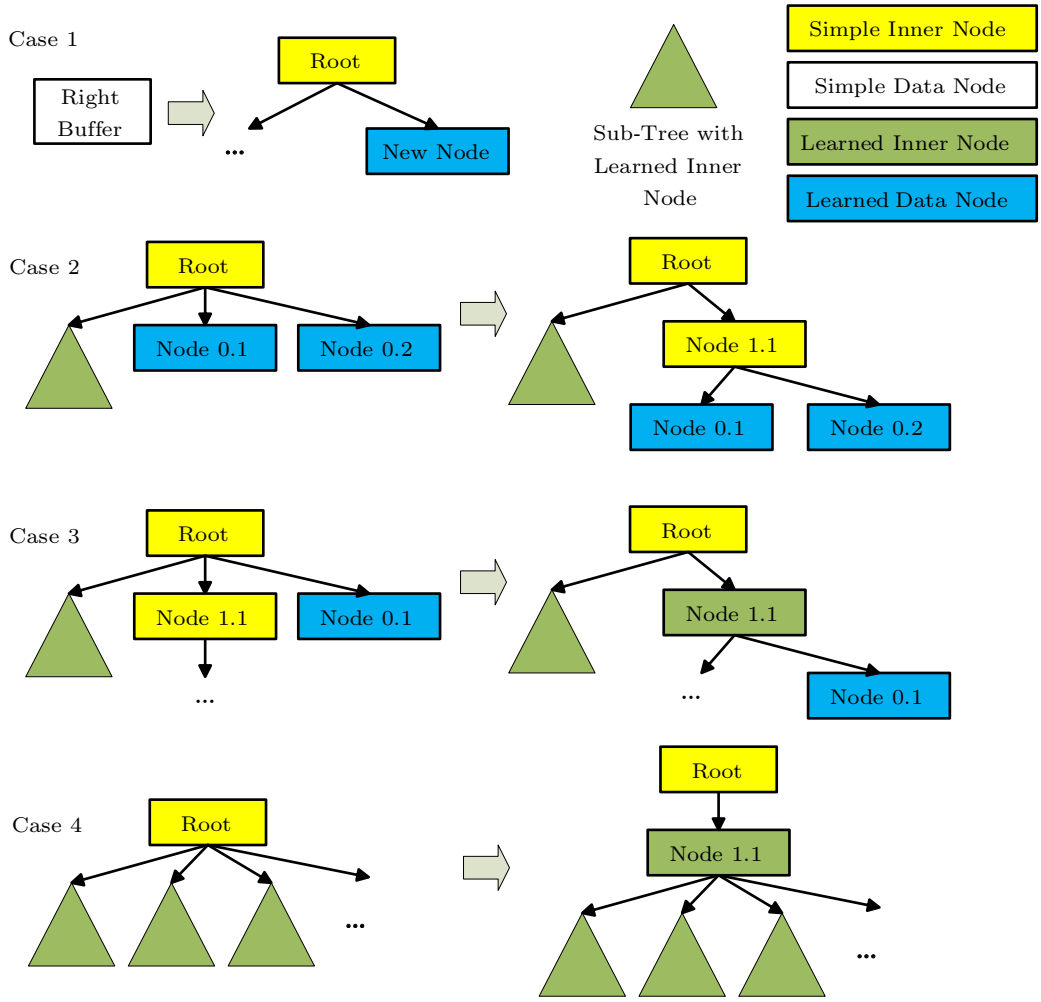


Fig.7. Root neatening caused by merging buffers.

the minimum and the maximum offsets of the keys assigned to B in the original dataset are represented by (2) and (3) respectively.

$$offset_{\min} = \frac{sp \times fr - \varepsilon}{fr}. \tag{2}$$

$$offset_{\max} = \frac{(sp + b - 1) \times fr + \varepsilon}{fr}. \tag{3}$$

In the worst case, all keys are concentrated in the first and the last two cachelines of the block. In this case, the amount of data overflowed can be estimated by (4).

$$\begin{aligned} overflow_{\max} &= offset_{\max} - offset_{\min} - 2l \\ &= \frac{2\varepsilon}{fr} + b - 1 - 2l. \end{aligned} \tag{4}$$

In our implementation, the maximum capacity of a three-level simple node is $\text{L1\_}size \times 8 \times 8 \times 75\% = 576$, $fr = 60\%$, $b = 64$, and $l = 8$. Therefore, the maximum value of ε is 260.5. We suggest setting ε to 256. The experimental results in Subsection 6.6 show that COLIN has better read/write performance and space efficiency when this value is used.

5.2 Cost of the Worst-Case Query

The query cost in COLIN includes two parts, the model calculation cost and the data access cost. We divide the data access cost into two types: cache hit and cache miss. We use C_{model}, C_{hit}, and C_{miss} to represent the model calculation cost, the data access cost with a cache hit, and the data access cost with a cache miss respectively.

The *Find* algorithm of learned data nodes needs to access the metadata to get the model and the array pointers, perform model calculation, search in a cacheline, and get payloads. The *Find* algorithm of learned inner nodes is similar. Therefore, the cost of the worst-

case query in learned nodes can be estimated by (5).

$$C_{\text{learned}} = C_{\text{model}} + 3 \times C_{\text{miss}} + 10 \times C_{\text{hit}}. \quad (5)$$

The *Find* algorithm of simple nodes needs to access metadata to get array pointers, search in the L1, L2, and L3 arrays, and get the payload or the pointer. The worst-case query cost of a three-level simple node is shown in (6).

$$C_{\text{simple}} = 5 \times C_{\text{miss}} + 27 \times C_{\text{hit}}. \quad (6)$$

For learned data nodes, if the data is in an overflow block, additional access to the overflow block pointer array is required, which requires an additional C_{miss} and C_{hit}.

To sum up, there is a constant upper bound for the query cost of any node in COLIN. Therefore, the query complexity of COLIN is $O(\log n)$. Our design significantly reduces the number of C_{miss}. On the current computer architecture, $C_{\text{miss}} \gg C_{\text{hit}}$ can be assumed, thereby our design can have an excellent query performance.

6 Performance Evaluation

6.1 Experimental Setup

Competitors. We compare COLIN with three learned indices and a traditional index. The learned indices are FITing-Tree [2], PGM-index [3], and ALEX [4]. The source code of PGM-index is provide by the PGM-index group①, and the implementation of ALEX is based on the open-source code in Github②. For the traditional B+-tree index, we use a popular in-memory B+-tree implementation③. For FITing-Tree, we implement it by ourselves according to the technical details in [2]. Note that ALEX does not have provably bounded query complexity, which differs from COLIN and the other competitors.

Parameters. The default parameter settings of COLIN are shown in Table 1. For FITing-Tree, we select a set of parameters with the best read and write performance, where the size of ε is 8, and the buffer size (or the reserved space size for in-place insertions) is 256. For PGM-index, we use the recommended parameter settings [3], where the size of ε is 64. For B+-tree, we set the node size to 256 bytes.

Environment. The experiments run on an Ubuntu Linux machine with Intel Core i7-7700 CPU and 64 GB RAM. COLIN and all of the competitors are implemented in C++. Each set of experiments uses "-O3" optimization and runs with a single thread.

Datasets. Three datasets with different characteristics are used in the experiments, namely, a normal dataset, a lognormal dataset, and an OSM dataset. The first two datasets include generated random numbers, and the OSM dataset involves the longitude attributes of the map data extracted from OpenStreetMap④. Table 2 shows the amount of data and key information for each dataset. All keys in each dataset are unique. Fig.8 shows the CDFs (cumulative distribution functions) of the three datasets, from which we can see that the three datasets have very different data distributions.

Table 1. Default Parameter Settings of COLIN

Parameter	Default	Description
ε	256	Maximum error of model
Line size	64 bytes	The size of a line in learned nodes
Block size	512 bytes	The size of a block in learned data nodes
Minimum fill rate	60%	Used to expand the array size when initializing the learned node
Maximum fill rate	90%	Used to determine whether to perform a structure modified operation
Maximum overflow rate	30%	Used to determine whether to perform a node splitting or a node expansion
Maximum capacity	2^{20}	The maximum allowed capacity of a learned node
L1 size	96 bytes	The L1 array size of simple nodes
L2/L1	8	The expansion ratio for L1 array to L2 array
L3/L2	8	The expansion ratio for L2 array to L3 array
L3 fill rate	75%	The maximum fill rate of L3 array

①https://pgm.di.unipi.it/, July 2021.
②https://github.com/microsoft/ALEX/, July 2021.
③https://panthema.net/2007/stx-btree/, July 2021.
④https://registry.opendata.aws/osm/, July 2021.

Table 2. Dataset Characteristics

Dataset	Number of Keys ($\times 10^6$)	Key Type	Key Size (Byte)	Payload Size (Byte)
Normal	200	Double	8	8
Lognormal	200	Double	8	8
OSM	180	Double	8	8

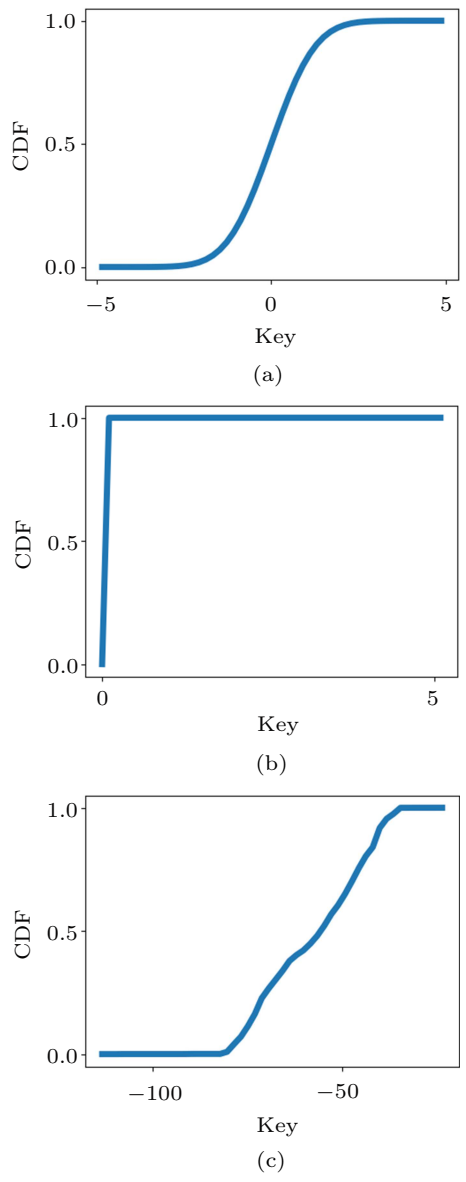


Fig.8. CDFs of the datasets. (a) Normal. (b) Lognormal. (c) OSM.

Workloads. We use five workloads: write-only, write-heavy (50% write and 50% read), read-heavy (5% write and 95% read), read-only, and read-modify-write. The last four workloads correspond to the YCSB A, B, C, and F workloads, respectively [33]. Except for the last workload, for all writes, we insert a new key instead of updating an existing one. For the read-modify-write workload, we read a key, modify its payload, and update it in the index. The read operations in all workloads follow the Zipfian distribution [34]. For each set of experiments, we first load half of the data in the dataset in bulk and then write the other half (if necessary). Before running each set of workloads, we run 10^8 queries as a pre-warming of the index.

Runtime State of COLIN. Table 3 shows the runtime states of COLIN after all keys have been inserted. Because a large ε is used, COLIN has a flat structure on any dataset.

6.2 Read and Write Performance

Figs.9(a)–9(e) show the throughput of all indices on the five workloads. The Y-axis is in Mtps (millions of transactions per second). For the first four workloads, a transaction represents a read or write operation. For the read-modify-write workload, one transaction includes a read operation and a write operation.

Overall, COLIN shows the best performance on all workloads. In particular, on the real dataset OSM, COLIN outperforms the second place on five workloads by 31%, 34%, 40%, 37%, and 22%, respectively. This is because COLIN adopts a model-based data placement strategy and a cache-conscious data layout, which can reduce the cost of local searches and data shifts in the process of read and write. ALEX's performance is second only to COLIN's. For the lognormal dataset, its performance is comparable to COLIN's. For the normal dataset, its performance on the first two workloads is close to COLIN's, and the performance on the last three workloads is worse than COLIN's. For the OSM dataset, it is inferior to COLIN on all workloads. Moreover, COLIN is well ahead of the other four competitors, except that PGM-Index performs well on write-

Table 3. Runtime States of COLIN

Dataset	Maximum Depth (Including a Root Node)	Number of Learned Inner Nodes	Number of Learned Data Nodes	Number of Keys per Node
Normal	3	1	599	333 890
Lognormal	3	4	777	257 400
OSM	3	4	3 164	56 890

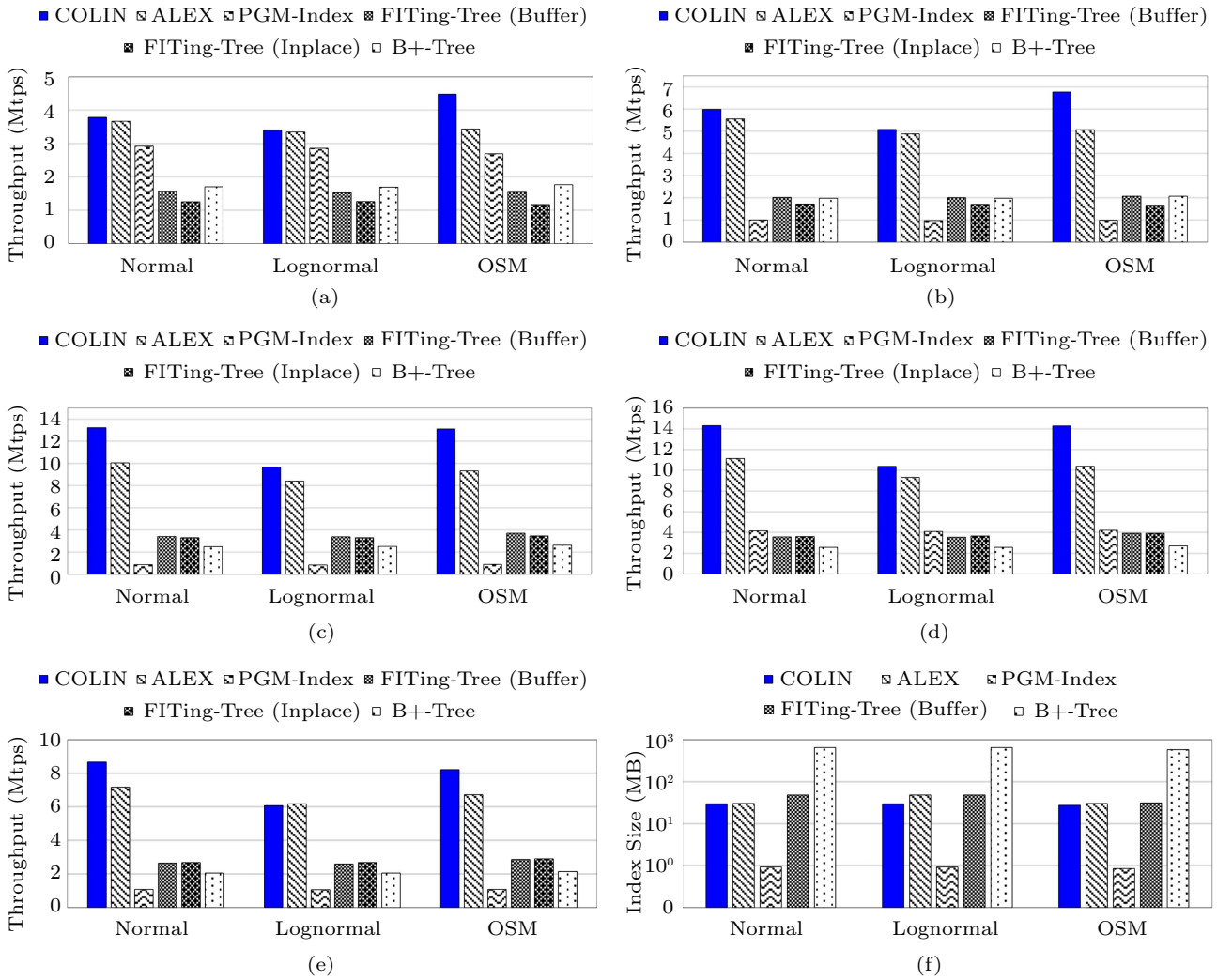


Fig.9. Experimental results on various workloads and datasets. (a) Write-only workload. (b) Write-heavy workload. (c) Read-heavy workload. (d) Read-only workload. (e) Read-modify-write workload. (f) Index size.

only workload, on which COLIN is 19%–66% better than PGM-index. On the other four workloads, COLIN performs more than twice as well as PGM-index. For the other three competitors, COLIN performs more than twice as well on all workloads.

Note that there are some differences in COLIN's performance between different datasets. COLIN's performance is better on the normal and OSM datasets, and poorer on the lognormal dataset. We think this is because COLIN uses a large ε and is more affected by the distribution of datasets. FITing-Tree and PGM-index have similar phenomena, which are much smaller than COLIN because they use smaller ε, which means they use shorter and more segments to fit the dataset.

We observe some interesting results of PGM-index. It performs well on both read-only and write-only workloads but poorly on read-write mixed workloads. Once the new key is inserted, PGM-index needs to use $\log n$ global buffers, which can dramatically degrade query performance. There is no query on the write-only workload after a write operation, which yields the high performance of PGM-index. However, on the read-write mixed workloads, the problem of read amplification makes PGM-index perform poorly.

6.3 Index Size

Fig.9(f) shows the index size of the competitors. The index size contains the inner nodes (including metadata, key arrays, and pointer arrays) and the metadata of the data nodes (including models and overflow block pointer arrays, etc.). We load half of the dataset in bulk, insert the other half, and count the index size. For reference, all keys in the normal or log-

normal datasets are 1.49 GB in size, while in the OSM dataset they are 1.34 GB. As a baseline, B+-tree is 650 MB in size on the first two datasets and 585 MB in size on the OSM dataset. PGM-index has the smallest index size, which is 688 times smaller than B+-tree. This is because PGM-index abandons pointers at inner nodes, which finds children by offset. COLIN is the second most space-efficient, occupying 20 times less memory than B+-tree. ALEX and FITing-Tree have a larger size than COLIN. In addition, ALEX shows different space efficiency on different datasets. On the normal dataset, it takes up about the same amount of space as COLIN. On the lognormal dataset, it takes up 61% more space than COLIN, and it is close to the size of FITing-Tree.

6.4 Bulk Loading

Fig.10 shows the bulk loading performance of competitors. In Fig.10(a), for each dataset, 100M keys are loaded, and the loading time is shown. ALEX has the worst bulk loading performance because it computes a cost model to determine the shape of the index tree during the building process. We observe that the bulk loading speed of ALEX is correlated with data distribution. On the normal dataset, it takes twice as long as B+-tree. But on the lognormal dataset, it takes five times as long as B+-tree. The bulk loading speeds of other competitors are consistent across the three datasets, indicating that their bulk loading performance is independent of data distribution. Surprisingly, all the other three learned indices outperform B+-tree. Although learned indices require training models, the time complexity of the algorithm is $O(n)$ [6–8]. The index size and the number of the nodes are much smaller than those of B+-tree, which means much less data needs to be organized when building the index. Compared with PGM-index and FITing-Tree, COLIN needs to perform model-based insertions during bulk loading, instead of replicating data in bulk. The results show that the bulk loading time of COLIN is 27%–32% longer than that of PGM-index, and is similar to that of FITing-Tree, which means the cost of insertions is not particularly expensive.

In Fig.10(b), we show how the bulk loading time changes as the data volume expands. The results show that the time of all competitors is linearly related to the amount of data. Furthermore, although the experiments are performed on a single thread, all the competitors except ALEX can easily use multithreading for bulk loading by evenly dividing the datasets in advance [3].

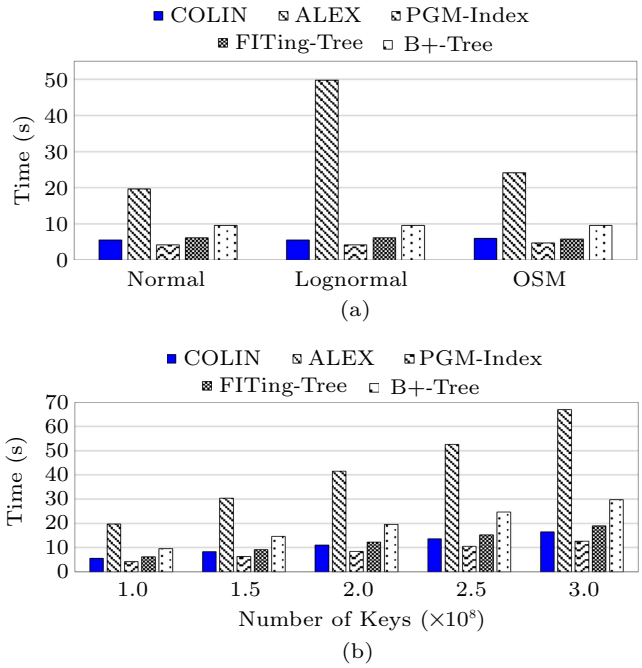


Fig.10. Time of bulk loading. (a) On different datasets. (b) On the normal dataset with scaling.

6.5 Fill Rate and Overflow Rate

Fig.11 shows the fill rate and overflow rate of COLIN. The fill rate refers to the proportion of slots used in all learned data nodes. The overflow rate refers to the proportion of data in the overflow blocks. We bulk load 100M keys first, then insert 200M keys, counting every 10M inserted. The results show that the fill rate of COLIN fluctuates from 60% to 80%, and the mean value is about 71.3%. The overflow rate fluctuates between 2% and 10%, with an average of about 5.7%.

Fig. 11. Fill rate and overflow rate of COLIN on the normal dataset.

6.6 Maximum Error of Models

Fig.12 shows the impact of the maximum error ε on the performance of COLIN, where the normal dataset is used. Among them, Fig.12(a) shows the size of the index, Fig.12(b) shows the performance on the write-heavy workload, and Fig.12(c) shows the performance on the read-heavy workload. With the increase of ε, COLIN's index size, read and write performance are all getting better. This is because COLIN uses a model-based data placement strategy and a cache-conscious data layout to decouple the search boundary from the maximum error of models. In this way, COLIN can utilize a large ε to reduce the number of nodes and index height, and ensure that local search performance does not degrade.

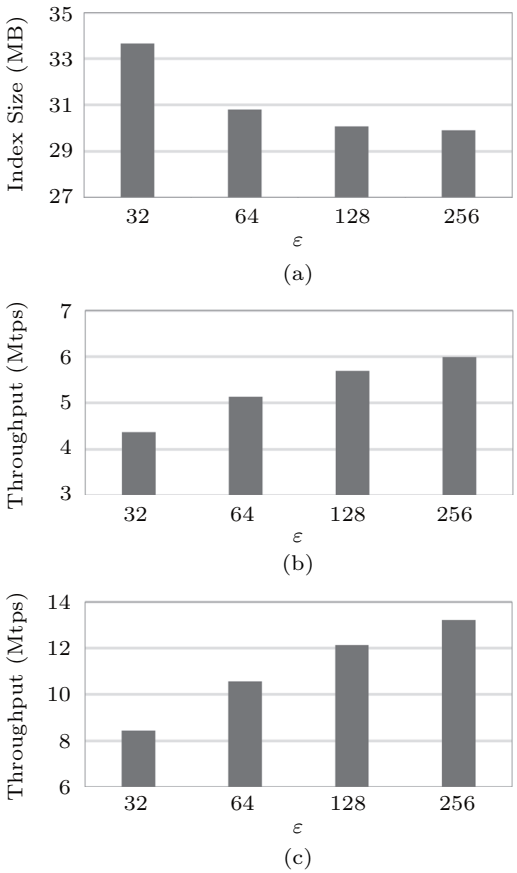


Fig.12. Impact of the maximum error ε on the performance of COLIN. (a) Index size. (b) Write-heavy workload. (c) Read-heavy workload.

7 Conclusions

In this paper, we presented COLIN, a cache-conscious dynamic learned index with bounded query complexity. It has a higher read and write performance than existing dynamic learned indices. COLIN uses a model-based data placement policy and a cache-conscious data layout to decouple the local-search range from the maximum error of the model. Moreover, it uses a heterogeneous index structure consisting of learned nodes and simple nodes to support out-of-bounds insertions and data overflows. The operations for COLIN, including query, upsert, deletion, bulk loading, and structure-modified operations, are detailed in the paper. We analyzed the model's maximum error and the cost of the worst-case query for COLIN. We conducted extensive experiments to compare COLIN with the traditional B+-tree and three state-of-the-art dynamic learned indices, namely FITing-Tree, PGM-index, and ALEX. The results showed that COLIN outperforms all competitors in terms of read and write performance on various datasets and workloads.

Although the current version of COLIN has shown performance improvement over existing approaches, the complex index structure will introduce extra work of implementing the index into a DBMS kernel. Thus, in the future, we will consider the simplification of the structure of COLIN, e.g., replacing simple nodes with B+-tree nodes. Another future work is how to adjust COLIN according to the workload type so that COLIN can adapt to different kinds of workloads. In addition, we will conduct additional experiments to verify the performance of COLIN when varying the size of the cacheline and the CPU cache.

References

[1] Kraska T, Beutel A, Chi E H, Dean J, Polyzotis N. The case for learned index structures. In *Proc. the 2018 International Conference on Management of Data*, Jun. 2018, pp.489-504. DOI: 10.1145/3183713.3196909.

[2] Galakatos A, Markovitch M, Binnig C, Fonseca R, Kraska T. FITing-Tree: A data-aware index structure. In *Proc. the 2019 International Conference on Management of Data*, Jun. 2019, pp.1189-1206. DOI: 10.1145/3299869.3319860.

[3] Ferragina P, Vinciguerra G. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 2020, 13(8): 1162-1175. DOI: 10.14778/3389133.3389135.

[4] Ding J, Minhas U F, Yu J *et al.* ALEX: An updatable adaptive learned index. In *Proc. the 2020 ACM International Conference on Management of Data*, Jun. 2020, pp.969-984. DOI: 10.1145/3318464.3389711.

[5] Shazeer N, Mirhoseini A, Maziarz K, Davis A, Le Q V, Hinton G E, Dean J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *Proc. the 5th International Conference on Learning Representations*, April 2017.

[6] Liu X, Lin Z, Wang H. Novel online methods for time series segmentation. *IEEE Transactions on Knowledge and Data Engineering*, 2008, 20(12): 1616-1626. DOI: 10.1109/TKDE.2008.29.

[7] Xu Z, Zhang R, Ramamohanarao K, Parampalli U. An adaptive algorithm for online time series segmentation with error bound guarantee. In *Proc. the 15th International Conference on Extending Database Technology*, Mar. 2012, pp.192-203. DOI: 10.1145/2247596.2247620.

[8] Xie Q, Pang C, Zhou X, Zhang X, Deng K. Maximum error-bounded piecewise linear representation for online stream approximation. *The VLDB Journal*, 2014, 23(6): 915-937. DOI: 10.1007/s00778-014-0355-0.

[9] Bentley J L, Yao A C. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 1976, 5(3): 82-87. DOI: 10.1016/0020-0190(76)90071-5.

[10] Hadian A, Heinis T. Considerations for handling updates in learned index structures. In *Proc. the 2nd International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, Jul. 2019, Article No. 3. DOI: 10.1145/3329859.3329874.

[11] Li X, Li J, Wang X. ASLM: Adaptive single layer model for learned index. In *Proc. the 2019 International Conference on Database Systems for Advanced Applications*, Apr. 2019, pp.80-95. DOI: 10.1007/978-3-030-18590-9_6.

[12] O'Neil P, Cheng E Y, Gawlick D, Oneil E. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 1996, 33(4): 351-385. DOI: 10.1007/s002360050048.

[13] Bender M A, Hu H. An adaptive packed-memory array. *ACM Transactions on Database Systems*, 2007, 32(4): Article No. 26. DOI: 10.1145/1292609.1292616.

[14] Ailamaki A, DeWitt D, Hill M, Wood D. DBMSs on a modern processor: Where does time go? In *Proc. the 25th International Conference on Very Large Data Bases*, Sept. 1999, pp.266-277.

[15] Hadian A, Heinis T. Shift-Table: A low-latency learned index for range queries using model correction. In *Proc. the 24th International Conference on Extending Database Technology*, Mar. 2021, pp.253-264. DOI: 10.5441/002/edbt.2021.23.

[16] Tang C, Wang Y, Hu G *et al.* XIndex: A scalable learned index for multicore data storage. In *Proc. the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2020, pp.308-320. DOI: 10.1145/3332466.3374547.

[17] Kipf A, Marcus R, van Renen A, Stoian M, Kemper A, Kraska T, Neumann T. RadixSpline: A single-pass learned index. In *Proc. the 3rd International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, Jun. 2020, Article No. 5. DOI: 10.1145/3401071.3401659.

[18] Neumann T, Michel S. Smooth interpolating histograms with error guarantees. In *Proc. the 25th British National Conference on Databases*, July 2008, pp.126-138. DOI: 10.1007/978-3-540-70504-8_12.

[19] Bilgram R. Cost models for learned index with insertions [Master Thesis]. Department of Computer Science, Aalborg University, 2019.

[20] Wang Y, Tang C, Wang Z, Chen H. SIndex: A scalable learned index for string keys. In *Proc. the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, Aug. 2020, pp.17-24. DOI: 10.1145/3409963.3410496.

[21] Llaveshi A, Sirin U, Ailamaki A, West R. Accelerating B+-tree search by using simple machine learning techniques. In *Proc. the 1st International Workshop on Applied AI for Database Systems and Applications*, Aug. 2019.

[22] Hadian A, Heinis T. Interpolation-friendly B-trees: Bridging the gap between algorithmic and learned indexes. In *Proc. the 22nd International Conference on Extending Database Technology*, Mar. 2019, pp.710-713. DOI: 10.5441/002/edbt.2019.93.

[23] Hadian A, Heinis T. MADEX: Learning-augmented algorithmic index structures. In *Proc. the 2nd International Workshop on Applied AI for Database Systems and Applications*, Aug. 2020.

[24] Li P, Lu H, Zheng Q, Yang L, Pan G. LISA: A learned index structure for spatial data. In *Proc. the 2020 International Conference on Management of Data*, Jun. 2020, pp.2119-2133. DOI: 10.1145/3318464.3389703.

[25] Qi J, Liu G, Jensen C S, Kulik L. Effectively learning spatial indices. *Proceedings of the VLDB Endowment*, 2020, 13(11): 2341-2354. DOI: 10.14778/3407790.3407829.

[26] Nathan V, Ding J, Alizadeh M, Kraska T. Learning multi-dimensional indexes. In *Proc. the 2020 International Conference on Management of Data*, Jun. 2020, pp.985-1000. DOI: 10.1145/3318464.3380579.

[27] Ding J, Nathan V, Alizadeh M, Kraska T. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment*, 2020, 14(2): 74-86. DOI: 10.14778/3425879.3425880.

[28] Zhou X, Chai C, Li G, Sun J. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*. DOI: 10.1109/TKDE.2020.2994641.

[29] Sun J, Li G. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment*, 2019, 13(3): 307-319. DOI: 10.14778/3368289.3368296.

[30] Rodriguez L V, Yusuf F, Lyons S, Paz E, Rangaswami R, Liu J, Zhao M, Narasimhan G. Learning cache replacement with CACHEUS. In *Proc. the 19th USENIX Conference on File and Storage Technologies*, Feb. 2021, pp.341-354.

[31] Zhou X, Sun J, Li G, Feng J. Query performance prediction for concurrent queries using graph embedding. *Proceedings of the VLDB Endowment*, 2020, 13(9): 1416-1428. DOI: 10.14778/3397230.3397238.

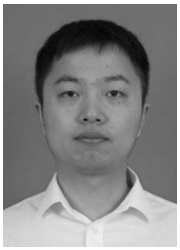
[32] Fan J, Liu T, Li G, Chen J, Shen Y, Du X. Relational data synthesis using generative adversarial networks: A design space exploration. *Proceedings of the VLDB Endowment*, 2020, 13(11): 1962-1975. DOI: 10.14778/3407790.3407802.

[33] Cooper B F, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In *Proc. the 1st ACM Symposium on Cloud Computing*, Jun. 2010, pp.143-154. DOI: 10.1145/1807128.1807152.

[34] Jin P, Ou Y, Härder T, Li Z. AD-LRU: An efficient buffer replacement algorithm for flash-based databases. *Data & Knowledge Engineering*, 2012, 72: 83-102. DOI: 10.1016/j.datak.2011.09.007.

Zhou Zhang received his B.S. degree in computer science and technology from the University of Science and Technology of China, Hefei, in 2016. He is a Ph.D. candidate of the School of Computer Science and Technology, University of Science and Technology of China, Hefei. His current research interests include database index, non-volatile memory, and stream processing systems.

Pei-Quan Jin received his Ph.D. degree in computer science and technology from the University of Science and Technology of China, Hefei, in 2003. He is currently an associate professor in the School of Computer Science and Technology, University of Science and Technology of China, Hefei. He is a member of ACM and IEEE, and a senior member of CCF. His research interests focus on big data management, databases on new storage, and information retrieval.



Xiao-Liang Wang received his B.S. degree in computer science and technology from Nanjing University of Aeronautics and Astronautics, Nanjing, in 2015. He is currently a Ph.D. candidate of the School of Computer Science and Technology, University of Science and Technology of China, Hefei. His research interests focus on buffer management systems and key-value storage engines.

Yan-Qi Lv received his B.S. degree in computer science and technology from the University of Science and Technology of China, Hefei, in 2019. He is a Master's candidate of the School of Computer Science and Technology, University of Science and Technology of China, Hefei. His current research interests include time-series database and micro-batch processing.

Shou-Hong Wan received her Ph.D. degree in computer science and technology from the University of Science and Technology of China, Hefei. She is currently an associate professor in the School of Computer Science and Technology, University of Science and Technology of China, Hefei. She is a member of ACM and IEEE. Her research interests focus on big data management, image processing, and information retrieval.

Xi-Ke Xie received his Ph.D. degree in computer science and technology from the University of Hong Kong, Hong Kong. He is currently a professor in the School of Computer Science and Technology, University of Science and Technology of China, Hefei. He is a member of ACM and IEEE. His research interests include distributed databases, spatiotemporal databases, and mobile computing.