DILI: A Distribution-Driven Learned Index (Extended Version)

Pengfei Li[1], Hua Lu[2], Rong Zhu[1], Bolin Ding[1], Long Yang[3] and Gang Pan[4]
[1]Alibaba Group, China, [2]Roskilde University, Denmark, [3]Peking University, China, [4]Zhejiang University, China
[1]{lpf367135, red.zr, bolin.ding}@alibaba-inc.com, [2]luhua@ruc.dk, [3]yanglong001@pku.edu.cn, [4]gpan@zju.edu.cn

ABSTRACT

Targeting in-memory one-dimensional search keys, we propose a novel DIstribution-driven Learned Index tree (**DILI**), where a concise and computation-efficient linear regression model is used for each node. An internal node's key range is equally divided by its child nodes such that a key search enjoys perfect model prediction accuracy to find the relevant leaf node. A leaf node uses machine learning models to generate searchable data layout and thus accurately predicts the data record position for a key. To construct DILI, we first build a bottom-up tree with linear regression models according to global and local key distributions. Using the bottom-up tree, we build DILI in a top-down manner, individualizing the fanouts for internal nodes according to local distributions. DILI strikes a good balance between the number of leaf nodes and the height of the tree, two critical factors of key search time. Moreover, we design flexible algorithms for DILI to efficiently insert and delete keys and automatically adjust the tree structure when necessary. Extensive experimental results show that DILI outperforms the state-of-the-art alternatives on different kinds of workloads.

1 INTRODUCTION

Recently, the learned index [29] is proposed to replace B+Tree [15] in database search. It stages machine learning models into a hierarchy called Recursive Model Index (RMI). Given a search key x, RMI predicts, with some error bound, where x's data is positioned in a memory-resident dense array. Compared to B+Tree, RMI achieves comparable and even better search performance. However, the layout of RMI, *i.e.*, the number of stages and the number of models at each stage, must be fixed before the models are created. Also, RMI fails to support key insertions and deletions.

To support data updates, ALEX [18] extends RMI by using a gapped array layout for the leaf level models. Moreover, ALEX uses cost models to initiate the RMI structure and to dynamically adapt the structure to updates. However, the stage layout of ALEX is not flexible enough as its fanout, *i.e.*, the number of a node's child models, is stipulated to a power of 2. This renders ALEX's internal nodes' key ranges relatively static, which may result in node layout not good for particular key distributions, *e.g.*, lognormal distribution. Also, ALEX's leaf level learned models do not guarantee accurate predictions. Thus, extra local search is needed to locate the required data, which downgrades the search performance. More recently, LIPP [43] trains learned models for the whole dataset and places data at the predicted positions. When multiple data records are assigned to the same position, a new node is created at the position to hold them. However, this simple strategy ignores the data distribution and often results in long traversal paths. Also, compared to B+Tree and ALEX, LIPP consumes much more memory.

In this paper, we design a novel index tree—DIstribution-driven Learned Index (DILI). Its each node features an individualized fanout and a model created for a data portion whose key sequence is covered by the node's range. For an internal node, its child nodes equally divide its range. Thus, the cost is minimized to locate the relevant leaf node in a key search. In a leaf node, an entry array $\boldsymbol{V}$ holds the keys in the node's range and the pointers to the corresponding data records. In addition, a leaf node uses an efficient linear regression model to map its keys to the positions in $\boldsymbol{V}$.

A critical issue for constructing DILI is to determine its node layout that is able to achieve good search performance. We design a sophisticated approach aware of data distributions and search costs. A key search in DILI involves two steps: 1) finding the leaf node covering the given key and 2) local search inside the leaf node. Accordingly, the general search performance depends on two factors: leaf nodes' depths and linear regression models' accuracy in the leaf nodes. Both factors should be considered in DILI construction.

To this end, we propose a two-phase bulk loading approach. The first phase creates a distribution-driven bottom-up tree (BU-Tree), whose node layout is determined by a greedy merging algorithm that considers both aforementioned factors. The merging creates linear regression models, starting at the bottom level to fully utilize the known key distribution. As a result, the models in the BU-Tree's leaf nodes guarantee good accuracy. Basically, we build DILI by making its node layout overall similar to that of the BU-Tree. However, a BU internal node's range is not necessarily equally divided by its child nodes. Therefore, search in the BU-Tree's internal nodes can incur extra time to decide which child node to visit. To this end, the second phase converts the BU-Tree to a DILI by redistributing keys among sibling nodes. When doing so, we carefully set different fanouts for DILI's different internal nodes according to their local key distributions, such that each internal node is equally divided by its child nodes. Meanwhile, we retain good model accuracy in DILI's leaf nodes and keep them at the same level as the counterparts in the BU-Tree. As a result, we obtain a DILI that is efficient at finding leaf nodes and have linear regression models of high accuracy in leaf nodes. In other words, we first create a mirror model (BU-Tree) that exhibits a good node layout but cannot guarantee perfect accuracy, and then we create another similar model (DILI) that avoids the mirror model's drawbacks but maintains its advantages.

It is noteworthy that we build the BU-Tree and DILI according to detailed analyses of search costs, which consider caching effects in the main-memory context. Though DILI and ALEX share some structural similarities, they are constructed according to different perspectives and principles. ALEX is built top down, while the BU-Tree is bottom up and initially deals with all the keys. Thus, the BU-Tree understands the key distribution better and partitions them into leaf nodes more reasonably. This make the 'mirrored' DILI achieve good local search performance. Also, our proposed cost function makes the BU-Tree (and DILI) have a suitable height. In general, DILI has fewer levels than ALEX and B+Tree. As a result, finding leaf nodes in DILI consumes less time.

Our bulk loading approach makes the linear regression models

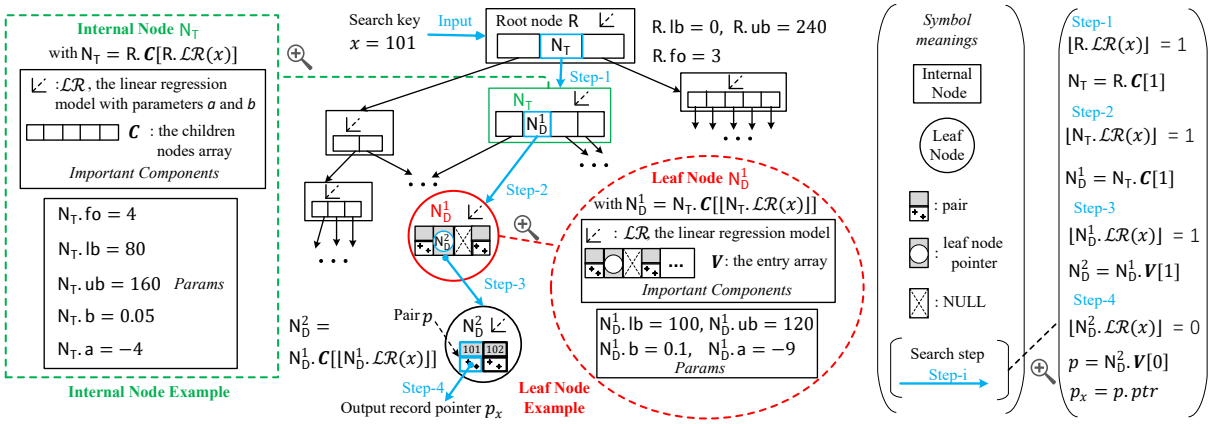


Figure 1: The Structure of DILI

in DILI's leaf nodes have high but not 100% accuracy. We find that the 'last-mile' local search in the leaf nodes is often the bottleneck of an entire query. To this end, we conduct a local optimization at each leaf node after the bulk loading, forcibly making the key-to-position mapping precise. If multiple keys are mapped into the same position, a new child node is created to hold them. Experimental results show that local optimizations improve the query performance of DILI by avoiding the local search inside the leaf nodes.

Our local optimization is inspired by LIPP [43] and LISA [31]. However, unlike LIPP, DILI's local optimization applies to leaf nodes only. Also, the two phase bulk loading approach makes DILI reasonably partitions data such that the keys covered by leaf nodes are almost linearly distributed. Compared to LIPP, the linear regression models in DILI's leaf nodes assign fewer keys the same slots. Thus, DILI encounters less conflicts and achieves better search performance and lower memory consumption.

Furthermore, unlike RMI [29], DILI supports data updates. When an inserted key conflicts with an existing key at a data slot of DILI's leaf node, our insertion algorithm creates a new leaf node to cover the conflicting data. Also, DILI flexibly redistributes data covered by a leaf node in a balanced way, when insertions generate too many nodes and degrades the query performance. Meanwhile, it allocates more data slots for those leaf nodes that encounter more frequent conflicts. In this way, DILI's height is bounded and the query performance downgrades only slightly even for many insertions. In addition, when a leaf node covers only one key after some deletions, this node will be trimmed to improve query performance and save memory consumption.

We make the following major contributions in this paper.

- We design a distribution-driven learned index DILI for in-memory 1D keys, together with algorithms and cost analysis.
- Accordingly, we design a distribution-driven BU-Tree as a node layout reference for DILI, and after more specific cost analyses we devise an algorithm to construct DILI based on BU-Tree.
- We propose a local optimization on DILI's leaf nodes to avoid the local search and improve query performance.
- To update DILI for key insertions and deletions, we devise efficient algorithms that retain search performance.
- We experimentally validate DILI's performance advantage over state-of-the-art alternatives on synthetic and real datasets.

The rest of the paper is organized as follows. Section 2 gives an overview of DILI. Section 3 analyses its search cost. Sections 4, 5 and 6 elaborate on DILI's construction, local optimization and updates, respectively. Section 7 reports on the experimental studies. Section 8 reviews the related work. Section 9 concludes the paper.

2 OVERVIEW OF DILI

Table 1 lists the important notations used in the paper. R4.R5

Table 1: Notations

N.fo	Fanout of the node N. N can be an internal or a leaf node
N.$\mathcal{LR}$	Linear regression model of the node N
$N_T.C$	The child node array of the internal node N_T
$N_D.V$	The entry array of the leaf node N_D
$N_D.\Omega$	Number of pairs covered by the leaf node N_D
$N_D.\Delta$	Total number of entries to be accessed to search for all keys covered by N_D, starting from N_D
$N_D.\kappa$	Average number of entries to be accessed to search for a key, starting from N_D, after the last local optimization to N_D
$N_D.\alpha$	Number of adjustments of N_D so far
$T_s(x)$	Search cost of key x in DILI without local optimization.
$T_{ns}^B(N, x, h)$	Search cost of key x *w.r.t.* a BU node N at height h
$T_{ea}^B(X_h, X)$	Estimated accumulated search cost of the break points list X_h for the key set X in the BU-Tree

Definition 1 (Pair). A pair is a 2-tuple $p = (key, ptr)$, where ptr is a pointer to the data record identified by key.

Let $\boldsymbol{P} = [p_0, p_1, \cdots, p_{|P|-1}]$ be an array of pairs, and KEYS($\boldsymbol{P}$) = $[p_0.key, p_1.key, \cdots, p_{|P|-1}.key]$ the key sequence from $\boldsymbol{P}$.

Definition 2 (Least square estimator). Given $I \subseteq [\tilde{n}] = \{0, 1, \cdots, n-1\}$, two sequences $\boldsymbol{X} = [x_0, \cdots, x_{n-1}]$ and $\boldsymbol{Y} = [y_0, \cdots, y_{n-1}]$, the least square estimator restricted to I is the linear function that minimizes $\sum_{i \in I}(y_i - f(x_i))^2$ over any linear function f. We use LEASTSQUARES($\boldsymbol{X}, \boldsymbol{Y}, I$) to denote an algorithm that finds the least square estimator for the data points restricted to I. When $I = [\tilde{n}]$, we simplify LEASTSQUARES($\boldsymbol{X}, \boldsymbol{Y}, I$) to LEASTSQUARES($\boldsymbol{X}, \boldsymbol{Y}$).

Fig. 1 illustrates the structure of DILI. The depths of its leaf nodes may be different, *i.e.*, DILI is an unbalanced tree. Instead of having key-pointer pairs, a node in DILI contains a model for indexing purpose. Specifically, a node N keeps two numbers N.lb and N.ub such that [N.lb, N.ub) forms N's **range**, *i.e.*, the key sequence covered by N. A node N, be internal or not, also stores a linear regression model N.$\mathcal{LR}$ parameterized by its intercept a and slope b, *i.e.*, N.$\mathcal{LR}(x) = a + bx$. Such models serve different purposes in internal and leaf nodes. R1.R1

Internal Nodes. DILI's internal nodes are represented as red dotted boxes in the bottom-middle part of Fig. 1. An internal node N_T stores a linear regression model $N_T.\mathcal{LR}$ and an array $N_T.C$ of pointers to N_T's child nodes. Note that $N_T.C$'s each element is a simple pointer, without any keys. In Fig. 1, an internal node's child nodes are represented as small **equal sized** rectangles. This is R1.R1

because N_T's child nodes equally divide N_T's range. Unlike B+Tree, we impose no constraints on N_T's fanout, *i.e.*, the length of $N_T.C$. Also, an internal node N_T in DILI does not need to store an additional ordered set of elements to describe the children's ranges, because they are clearly described by the linear regression model. Given a key x, we can easily know which child node covers x with a few simple calculations. When a search goes downward in the tree, N_T uses $N_T.\mathcal{LR}$ to 'compute' the location (in $N_T.C$) of the pointer to the next child node to visit. Let $N_T.\texttt{fo}$ denote the fanout of N_T. The intercept a and the slope b of $N_T.\mathcal{LR}$ are calculated as follows:

$$b = N_T.\texttt{fo}/(N_T.\texttt{ub} - N_T.\texttt{lb}), a = -b \times N_T.\texttt{lb} \quad (1)$$

Accordingly, N_T's ith child node's range is $[N_T.\texttt{lb} + \frac{i}{b}, N_T.\texttt{lb} + \frac{i+1}{b})$, *i.e.*, $[N_T.\mathcal{LR}^{-1}(i), N_T.\mathcal{LR}^{-1}(i+1))$. All N_T's child nodes' ranges are of equal length. For example, in Fig. 1, the internal node N_T has four children and its range is $[80, 160)$. N_T's second child node N_D^1 is assigned a range of $[N_T.\texttt{lb} + \frac{1}{N_T.b}, N_T.\texttt{lb} + \frac{2}{N_T.b}) = [100, 120)$.

Leaf Nodes. The leaf nodes in DILI are represented by circles in Fig. 1, where the ellipse in the bottom-middle part gives the details. A leaf node N_D stores an **entry** array $\boldsymbol{V}$ and a linear regression model $\mathcal{LR}$. Each entry is a pair, a pointer to another leaf node or a NULL flag indicating the corresponding slot of $\boldsymbol{V}$ is empty. $N_D.\boldsymbol{V}$ may cover leaf node pointers or NULL flags due to the ***local optimization*** strategy applied to the leaf nodes, which will be introduced later in this section and Section 5. The learned model $\mathcal{LR}$ maps a key to a position in $\boldsymbol{V}$. Unlike those models in internal nodes, $\mathcal{LR}$ is the solver to the mean squared error minimization problem, whose input is the keys of $\boldsymbol{P}_L$, *i.e.*, the pairs covered by N_D's range, and ground truth is the corresponding indices in $\boldsymbol{P}_L$. Specifically, $N_D.\mathcal{LR} \triangleq \textsc{leastSquares}(\textsc{keys}(\boldsymbol{P}_L), [|\tilde{\boldsymbol{P}_L}|])$. At present, we simply assume $N_D.\boldsymbol{V} = \boldsymbol{P}_L$. In other words, $\boldsymbol{V}$ only stores the key-pointer pairs, which are tightly arranged in $\boldsymbol{V}$. Note that each internal or leaf node only needs to store two parameters a and b for its linear regression model.

Search without Optimization. To search for a key x, we first find the leaf node whose range covers x, using the function LocateLeafNode (lines 6–10 in Algorithm 1). This function starts at DILI's root, iteratively uses the linear regression model in the current internal node to 'compute' a location in the node's pointer array C. It follows the pointers to child nodes iteratively until reaching a leaf node N_D. As the internal nodes' linear regression models have perfect accuracy, *i.e.*, they always choose the child nodes covering x. Thus, no local search is needed inside an internal node.

Algorithm 1 Search(Root, x)

```
1:  N_D ← locateLeafNode(Root, x)
2:  pos' ← ⌊N_D.LR(x)⌋
3:  pos ← exponentialSearch(N_D.V, x, pos')
4:  p ← N_D.V[pos]
5:  return (p.key = x ? p.ptr : NULL)
6:  function locateLeafNode(Root, x)
7:      while N points to an internal node do
8:          pos ← ⌊N.LR(x)⌋
9:          N ← N.C[pos]
10:     return N
```

After finding the leaf node N_D, we search the pair array $N_D.\boldsymbol{V}$ for the pair whose key is x (lines 2–5). Suppose the pair $p \in N_D.\boldsymbol{V}$ is the *least upper bound*[1] of x in $N_D.\boldsymbol{V}$. We use the model $N_D.\mathcal{LR}$ to estimate p's position pos' in $N_D.\boldsymbol{V}$ (line 2). From the position pos', an exponential search (line 3) is performed to find the actual position of p. At the returned position pos is the pair p (line 4). If $p.key$ is not x, we return NULL as no data record contains the key x. Otherwise, we return $p.ptr$ that points to x's data (line 5).

Construction. DILI is built by considering the keys' distribution to reduce the expected lookup time of queries. We propose a novel bulk loading algorithm to build DILI in Section 4. The algorithm 'learns' a good node layout for DILI from a given pair set $\boldsymbol{P}$.

Local Optimization strategy. In the experiments, we find that the 'last-mile search' in the leaf nodes (line 3 in Algorithm 1) is usually a bottleneck of the entire query. While our bulk loading algorithm make keys covered by a leaf node N_D almost linearly distributed, $N_D.\mathcal{LR}$ cannot guarantee perfect accuracy. To address this issue, we put a local optimization on each leaf node after its range and linear regression model is determined. Inspired by the novel idea of LISA [31] that "**using ML models to directly determine keys' storage positions instead of approximating them**, the local optimization makes DILI avoid local search by forcibly placing pairs at the returned position by the linear regression model. In overview, if $pos = N_D.\mathcal{LR}(p.key)$, the pair p will be put at $N_D.\boldsymbol{V}[pos]$. When the predictions of multiple pairs by $N_D.\mathcal{LR}$ are the same, *i.e.*, they *conflict*, a new leaf node will be created to deal with them.

It is noteworthy that Algorithm 1 is only used in DILI's bulk loading stage. In practice, a search algorithm with the local optimization will be adopted. The details of the local optimization as well as the optimized search algorithm are to be introduced in Section 5.

Updates. DILI supports data updates. Our insertion algorithm will create new leaf nodes to cover conflicting pairs if insertions incur conflicts. Meanwhile, pairs covered by a leaf node will be redistributed when too many node creations degrades the search performance. The details are to be given in Section 6.

Discussion. As described above and illustrated in Fig. 1, DILI's internal and leaf nodes feature different structures. Their local arrays keep different types of elements, due to their different roles in search. Internal nodes' role in the search process is to efficiently locate the leaf node covering the search key. To fulfill this, an internal node N_T's children nodes are assigned equal-size ranges through its model $N_T.\mathcal{LR}$. Thus, $N_T.\boldsymbol{V}$ arranges N_T's child nodes tightly. In contrast, the search process needs to find the pair from a leaf node N_D's entry array. However, $N_D.\mathcal{LR}$ may predict for multiple keys the same position in $N_D.\boldsymbol{V}$. To process the conflicts, the local optimization is adopted to create new leaf nodes to store the conflicting keys. Since for each pair at least one slot is preserved but multiple keys may conflict at the same position, it is possible that some slots in $N_D.\boldsymbol{V}$ have no contents. Thus, $N_D.\boldsymbol{V}$ consists of different kinds of elements from $N_D.C$.

3 SEARCH COST ANALYSIS

This section is a preparation for the bulk loading algorithm in Section 4. We make a detailed cache-aware cost analysis of Algorithm 1. Note that no local optimization is assumed at present. In other words, given any leaf node N_D, $N_D.\boldsymbol{V}$ contains no leaf node pointer and $N_D.\mathcal{LR}$ does not guarantee perfect prediction accuracy.

Cost Analysis. Algorithm 1 consists of two steps: 1) finding the leaf node covering the search key and 2) local search inside the leaf node. Given a pair p with key x, suppose N_D (with depth D)

[1] In a pair array $\boldsymbol{P}$, a key x's **least upper bound (LUB)** is a pair $p \in \boldsymbol{P}$ satisfying two conditions: 1) $p.key \geq x$ and 2) If $\exists p' \in \boldsymbol{P}$ *s.t.* $p'.key \geq x$, then $p'.key \geq p.key$.

is the leaf node covering x. Let $\mathsf{N_D}.\mathcal{LR}$'s prediction error for x be $\epsilon_x = |\mathsf{N_D}.\mathcal{LR}(x) - pos|$ where pos is p's position in $\mathsf{N_D}.\boldsymbol{V}$. The estimated search cost of key x is denoted by $\mathbf{T_s}(x)$ as follows.

$$\mathbf{T_s}(x) \approx \big((D-1) \times \mathbf{T_{is}}(x)\big) + \mathbf{T_{ds}}(\mathsf{N_D}, x) \quad (2)$$
$$\mathbf{T_{is}}(x) = (\theta_\mathsf{N} + \eta + \theta_\mathsf{C}), \mathbf{T_{ds}}(\mathsf{N_D}, x) = \theta_\mathsf{N} + \eta + \mathbf{t_E}(\mathsf{N_D}, x)$$

where $\mathbf{T_{is}}(x)$ and $\mathbf{T_{ds}}(\mathsf{N_D}, x)$ denote the time spent in an internal node and the leaf node $\mathsf{N_D}$ covering x respectively; θ_N and η are the estimated time of executing a linear function (lines 2 and 8 in Algorithm 1) and loading a DILI's node from the main memory respectively; θ_C is the estimated time of accessing the address of an internal node N's child node. In particular, after calculating $pos = \lfloor \mathsf{N}.\mathcal{LR}(x) \rfloor$ (line 2 in Algorithm 1), we need to get the pos-th element from $\mathsf{N}.\boldsymbol{C}$, and θ_C is the time of getting the corresponding pointer. Usually, both θ_N and θ_C equal the time of loading a cache line sized block from the main memory to the cache.

An exponential search needs about $2\log_2 \epsilon_x$ iterations. Each iteration consists of the calculation of the middle position, an operation of pair addressing and a comparison of two keys. Thus, the estimated time of the local search in $\mathsf{N_D}$ is $\mathbf{t_E}(\mathsf{N_D}, x) = 2\log_2 \epsilon_x \times (\mu_\mathsf{E} + \theta_\mathsf{E})$, where θ_E and μ_E are the average time of accessing a pair and executing the other operations in one iteration, respectively. An exponential search fetches pairs mostly stored separately.

Discussion. In practice, pair access is much slower than other operations. Due to limited cache size, a new node or pair often triggers a cache miss, which entails addressing in the heap. Addressing takes two steps: finding the leaf node and local search inside the node. However, less local search time in a leaf node often means the node stores fewer pairs, which tends to increase the number of leaf nodes. All this means more and deeper leaf nodes, which in turn incurs more time cost for finding the correct leaf node. To strike a trade-off between DILI's leaf node depth and number of leaf nodes, we proceed to design a bulk loading method to construct DILI by taking into account the time cost of both steps together.

4 CONSTRUCTION OF DILI

For a pair array $\boldsymbol{P}$ sorted on all keys, we want to build DILI with a good node layout having fast lookups for arbitrary search keys.

4.1 Motivation and Overall Idea of BU-Tree

As described in Section 2, DILI's linear regression model in an internal node $\mathsf{N_T}$ has perfect accuracy because its children equally divide its range by design. This design gives rise to a unique critical problem in constructing DILI: deciding the suitable fanout for $\mathsf{N_T}$.

One idea is to follow the top-down construction of ALEX [18], using a power of 2 for a internal node's fanout. If the whole key range is $[0, 1)$, the length of a leaf node's range must be $\frac{1}{2^k}$ for some integer k. For a complex key distribution (*e.g.*, a long-tail type), k must be large in order to ensure high accuracy of the linear model in the leaf node. This tends to result in many leaf nodes and thus a high tree, making it slow to find the leaf node for a given key.

Another idea comes from the bottom-up bulk loading of B+Tree. First, we partition all pairs in $\boldsymbol{P}$ into pieces and store each piece in a leaf node $\mathsf{N_D}$. In each $\mathsf{N_D}$, we build a linear model to map $\mathsf{N_D}$'s keys to their positions in $\mathsf{N_D}$'s piece. With an appropriate algorithm, we can ensure a relatively low total loss of all linear models in all leaf nodes. Then, we partition the boundaries of the leaf nodes using the same algorithm. We create internal nodes at height level 1 to save the boundaries, use the boundaries as the separation values to group the leaf nodes, and make each group of leaf nodes the children of their corresponding parent node at level 1. Likewise, we create internal nodes at height level i based on those at level $i-1$, and repeat the process until we reach a level with only one node. This approach reduces the local search time in the leaf nodes but it does not guarantee that child nodes equally divide their parent node's range, and thus the parent node's linear regression model fail to give perfect prediction accuracy. Rather, from the predicted position, extra operations must be performed to find the child node covering a given key, making the overall lookup time longer.

To build DILI that incurs low overall lookup time for arbitrary keys, we combine both ideas in a two-phase bulk loading algorithm. First, we create a bottom-up tree (BU-Tree), starting from leaf nodes and growing the tree upwards. Second, we reuse the BU-Tree's node layout to build DILI, and improve the latter's internal linear models such that they also obtain perfect prediction accuracy. The built DILI is able to find the leaf node covering a search key with only a few calculations of linear functions. Also, the local search time cost in the leaf nodes is small as DILI has a leaf node layout similar to the BU-Tree. Thus, DILI is built in a novel paradigm—we first design a mirror model that finds leaf nodes efficiently, and then create a similar model that further optimizes local search in a leaf node. Fig. 2 illustrates the procedure of our bulk loading algorithm.

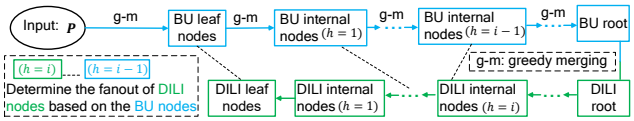


Figure 2: Framework of the bulk loading algorithm

We use *BU internal node* or *BU leaf node* to refer to an internal or leaf node in the BU-tree. A BU internal node $\mathsf{N_T}$ is structurally the same as that in DILI, except that $\mathsf{N_T}$ stores an additional array $\boldsymbol{B}$ to record the ranges of all its child nodes. Specifically, $\mathsf{N_T}.\boldsymbol{C}[i-1].\mathsf{ub} = \mathsf{N_T}.\boldsymbol{C}[i].\mathsf{lb} = \mathsf{N_T}.\boldsymbol{B}[i]$. Note that the child nodes may not equally divide $\mathsf{N_T}$'s range. All BU leaf nodes are at the same height level and they are reused as the basis of the leaf nodes in DILI.

Key search in a BU-Tree is different from DILI. Finding the child node covering key x in a BU internal node $\mathsf{N_T}$ involves two steps. The first step computes $j = \mathsf{N_T}.\mathcal{LR}(x)$ and the second step searches $\mathsf{N_T}.\boldsymbol{B}$ from position j to find the index i such that $\mathsf{N_T}.\boldsymbol{B}[i] \le x < \mathsf{N_T}.\boldsymbol{B}[i+1]$. As a result, $\mathsf{N_T}.\mathbf{C}[i]$ points to the correct child node.

4.2 Building BU-Tree

For a given $\boldsymbol{P}$, BU-Tree is built by Algorithm 2. After initialization (line 1), it calls the function GREEDYMERGING (Algorithm 3 to be detailed in Section 4.2.2) to generate all leaf nodes (line 2). Subsequently, Algorithm 2 creates all BU internal nodes (lines 3–11) in a bottom-up way, until an appropriate root node is found (lines 7–9). At each height h, we independently decide if the nodes at the current height should be the children of an immediate root node or not. For both cases (lines 5 and 6), we calculate the average *estimated accumulated search cost* (to be detailed in Section 4.2.2). It is an estimate of the lookup time of the corresponding DILI from its root node to the node at height h that covers the search key. If having

an immediate root node implies a smaller cost, we create a root node and set its child nodes to be $\mathbf{N}^h$, the BU nodes at height h (lines 7–9). Otherwise, the BU-Tree grows to height $h+1$ (line 10).

Algorithm 2 BuildBUTree($\boldsymbol{P}$)

1: $N \leftarrow |\boldsymbol{P}|, \boldsymbol{X} \leftarrow [x_0, \cdots, x_{N-1}]$ where $x_i = \boldsymbol{P}[i].key$
▷ Generate BU leaf nodes
2: $n_0, \boldsymbol{X}_0, \mathbf{N}^0, \varepsilon^1 \leftarrow$ greedyMerging(NULL, $\boldsymbol{X}$)
3: $h \leftarrow 0$
4: **while** $n_h > 1$ **do**
5: $\mathsf{N}^r, \varepsilon^0 \leftarrow$ generateRoot($\mathbf{N}^h, \boldsymbol{X}_h, \boldsymbol{X}$)
6: $n_{h+1}, \boldsymbol{X}_{h+1}, \mathbf{N}^{h+1}, \varepsilon^1 \leftarrow$ greedyMerging($\mathbf{N}^h, \boldsymbol{X}_h$)
7: **if** $\varepsilon^0 < \varepsilon^1$ **then** ▷ Growing DILI will result in larger cost
8: Set N^r to be the root node of the BU-Tree
9: **break**
10: **else** $h \leftarrow h+1$
11: **return** the root node
12: **function** generateRoot($\mathbf{N}^{h-1}, \boldsymbol{X}_{h-1}, \boldsymbol{X}$)
13: $Y_{h-1} \leftarrow I, I \leftarrow [\tilde{n_{h-1}}], n_{h-1} \leftarrow |\boldsymbol{X}_{h-1}|$
14: $\mathcal{F} \leftarrow$ leastSquares($\boldsymbol{X}_{h-1}, Y_{h-1}, I$)
15: $\mathsf{R} \leftarrow$ an empty BU internal node
16: $\mathsf{R}.\mathcal{LR} \leftarrow \mathcal{F}, \mathsf{R}.\mathtt{fo} \leftarrow n_{h-1}, \mathsf{R}.\boldsymbol{C} \leftarrow \mathbf{N}^{h-1}$
17: $\varepsilon \leftarrow \frac{1}{N}\sum_{i=0}^{N-1} \mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}, x_i)$ ▷ Calculate the search cost
18: **return** R, ε

4.2.1 Bottom-up Node and Model Creation. Given the pair set $\boldsymbol{P}$, we have $\boldsymbol{X} = \text{keys}(\boldsymbol{P}) = [x_0, \cdots, x_{N-1}]$ and $Y = [\tilde{N}] = [0, \cdots, N-1]$ where $x_i = \boldsymbol{P}[i].key$. We first find a suitable integer n_0 and $n_0 - 1$ break points $[\beta^0_1, \cdots, \beta^0_{n_0-1}]$ to partition the key space keys($\boldsymbol{P}$) into n_0 pieces. The ith piece's range is equal to $[\beta^0_i, \beta^0_{i+1})$ where $\beta^0_0 = \inf \text{keys}(\boldsymbol{P})$ and $\beta^0_{n_0} = \sup \text{keys}(\boldsymbol{P})$. For the ith piece, supposing $\beta^0_i \le x_l < \cdots < x_r < \beta^0_{i+1}$, we train a linear regression model $\mathcal{F}^0_i$ with input $[x_l, \cdots, x_r]$ and $[l, \cdots, r]$. Then, n_0 BU leaf nodes are created. The ith node N^0_i is described as follows.

$$\mathsf{N}^0_i.\mathtt{lb} = \beta^0_i, \mathsf{N}^0_i.\mathtt{ub} = \beta^0_{i+1}, \mathsf{N}^0_i.\mathcal{LR}(x) = \mathcal{F}^0_i(x) - l, \mathsf{N}^0_i.\boldsymbol{V} = \boldsymbol{P}[l:r] \quad (3)$$

Suppose that the BU nodes at height $h-1$ have been created. We define two lists $\boldsymbol{X}_{h-1} = [\mathsf{N}^{h-1}_0.\mathtt{lb} \cdots, \mathsf{N}^{h-1}_{n_{h-1}-1}.\mathtt{lb}]$ and $Y_{h-1} = [0, \cdots, n_{h-1}-1]$, where N^{h-1}_i is the ith node at height $h-1$ and n_{h-1} is the number of the nodes at height $h-1$. Similarly, we generate $n_h - 1$ break points $[\beta^h_1, \cdots, \beta^h_{n_h-1}]$, partition the space into n_h pieces, and build n_h linear regression models. Given a key x, suppose $\mathsf{N}^{h-1}_l.\mathtt{lb} \le x < \mathsf{N}^{h-1}_l.\mathtt{ub}$. We define a function $\zeta^{h-1}(x) = l$. The ith node N^h_i at height h is described as follows.

$$\mathsf{N}^h_i.\mathtt{lb} = \beta^h_i, \mathsf{N}^h_i.\mathtt{ub} = \beta^h_{i+1}, \mathsf{N}^h_i.\mathtt{fo} = n_h, \quad (4)$$
$$\mathsf{N}^h_i.\mathcal{LR}(x) = \mathcal{F}^h_i(x) - \zeta^{h-1}(x), \mathsf{N}^h_i.\boldsymbol{C}[j] = \mathsf{N}^{h-1}_q,$$
$$\mathsf{N}^h_i.\boldsymbol{B}[j] = \mathsf{N}^{h-1}_q.\mathtt{lb}, \text{where } q = \zeta^{h-1}(\beta^h_i) + j$$

A key challenge here is to decide n_h (the number of nodes at height h) and $[\beta^h_1, \cdots, \beta^h_{n_h-1}]$ (the break points for these n_h nodes), as they determine the node layout at height h.

4.2.2 Determining Node Layout at A Height. We want to have a suitable BU node layout such that the corresponding DILI will have a good node layout to minimize the average search time. As the DILI has a similar node layout with the BU-Tree, we simulate the DILI's querying process in the BU-Tree. To search for a key in the BU-Tree, we observe which nodes are accessed as well as the losses of their linear regression models. Based on those observations, we estimate the cost of searching for a key in the DILI.

Given a key x, we define the *estimated search cost* $\mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}, x, h)$ *w.r.t.* a BU node N and a height h as follows.

$$\mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}, x, h) = \theta_{\mathsf{N}} + \eta + \rho^h \times \mathbf{t}^{\mathbf{B}}_{\mathbf{E}}(\mathsf{N}, x), \text{where } \rho \in (0,1)$$
$$\mathbf{t}^{\mathbf{B}}_{\mathbf{E}}(\mathsf{N}, x) = \log_2 |\mathsf{N}.\mathcal{LR}(x) - i| \times (\mu_{\mathsf{E}} + \theta_{\mathsf{E}}) \quad (5)$$
$$i = \begin{cases} \mathsf{N}.\boldsymbol{B}[i] \le x < \mathsf{N}.\boldsymbol{B}[i+1], & \mathsf{N} \text{ is a BU internal node} \\ \mathsf{N}.\boldsymbol{V}[i].key \le x < \mathsf{N}.\boldsymbol{V}[i+1].key, & \text{otherwise.} \end{cases}$$

Here, θ_{N}, η, μ_{E} and θ_{E} carry the same meanings as those in Eq. 2; $\mathbf{t}^{\mathbf{B}}_{\mathbf{E}}$ is a simple extension of $\mathbf{t}_{\mathbf{E}}$ on BU nodes. If the node height is irrelevant, we simplify $\mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}, x, h)$ to $\mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}, x)$.

Given a key x, suppose the search in BU-Tree visits the complete node path $\mathsf{N}_k, \mathsf{N}_{k-1}, \cdots, \mathsf{N}_0$, where N_k is the root node and N_0 is the leaf node covering x. We define the *accumulated search cost till height* h $\mathbf{T}^{\mathbf{B}}_{\mathbf{al}}(h, x)$ and the *complete search cost* $\mathbf{T}^{\mathbf{B}}_{\mathbf{s}}(x)$ as follows.

$$\mathbf{T}^{\mathbf{B}}_{\mathbf{al}}(h, x) = \sum_{j=h}^{k} \mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}_j, x), \mathbf{T}^{\mathbf{B}}_{\mathbf{s}}(x) = \mathbf{T}^{\mathbf{B}}_{\mathbf{al}}(0, x) = \sum_{i=0}^{k} \mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}_i, x)$$

To minimize the average lookup time of the search key, we try to minimize $\frac{1}{N}\sum_{i=0}^{N-1} \mathbf{T}^{\mathbf{B}}_{\mathbf{s}}(x_i)$, *i.e.*, the average complete search cost for all keys. If the nodes under height h have been created, minimizing $\frac{1}{N}\sum_{i=0}^{N-1} \mathbf{T}^{\mathbf{B}}_{\mathbf{s}}(x_i)$ is equivalent to minimizing $\frac{1}{N}\sum_{i=0}^{N-1} \mathbf{T}^{\mathbf{B}}_{\mathbf{al}}(h, x_i)$. However, as a BU-Tree is grown upwards, we do not even know the height of the BU-Tree when creating nodes at height h, let alone estimating the search cost of a key in a node above height h. To this end, we introduce the *estimated accumulated search cost of the break points list* $\boldsymbol{X}_h$ for the key set $\boldsymbol{X}$, termed as $\mathbf{T}^{\mathbf{B}}_{\mathbf{ea}}(\boldsymbol{X}_h, \boldsymbol{X})$. It measures the quality of the node layout generated from the break points list $\boldsymbol{X}_h$. For simplicity, we assume that each BU internal node has the same number of child nodes, and define $\mathbf{T}^{\mathbf{B}}_{\mathbf{ea}}(\boldsymbol{X}_h, \boldsymbol{X})$ as follows.

$$\mathbf{T}^{\mathbf{B}}_{\mathbf{ea}}(\boldsymbol{X}_h, \boldsymbol{X}) = \frac{1}{N}\sum_{i=0}^{N-1} \sum_{h'=h}^{\lceil\delta\rceil} \min(1, \delta + 1 - h') \times \mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}^h_{t_i}, x_i, h')$$
$$\text{where } t_i = \zeta^h(x_i), \delta = \log_{\frac{n_{h-1}}{n_h}} n_{h-1}, n_{-1} = N$$

Above, $\mathsf{N}^h_{t_i}$ is the node at height h whose range covers x_i. In other words, $\boldsymbol{X}_h[t_i] \le x < \boldsymbol{X}_h[t_i+1]$. Moreover, δ is the estimated depth of the nodes at height h. We explain δ by an example. Suppose $n_{h-1} = 1000$ and $n_h = 100$, *i.e.*, the number of nodes at height $h-1$ and h are 1000 and 100, respectively. A node at height h has $\frac{1000}{100} = 10$ child nodes on average. The estimated $n_{h+1} = \frac{100}{10} = 10, n_{h+2} = 1$ and thus the root node's height is $h+2$. Because δ is the estimated depth of the nodes at height h, $\delta = (h+2) - h + 1 = 3 = \log_{\frac{1000}{100}} 1000$. As δ may not exactly be an integer, we add a multiplication factor $\delta + 1 - h' = \delta - \lfloor\delta\rfloor$ before $\mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}^h_{t_i}, x_i, h')$ when $h' = \lceil\delta\rceil$. Here, $\mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}^h_{t_i}, x_i, h')$ is an estimate of $\mathbf{T}^{\mathbf{B}}_{\mathbf{ns}}(\mathsf{N}^{h'}_{t_i}, x_i)$.

Given $\boldsymbol{X}_{h-1}$ and Y_{h-1}, to find the best n_h and $\boldsymbol{X}_h$, a straightforward approach is to set n_h to be different values. For each specific n_h, we solve a n_h-piecewise linear regression problem with input $\boldsymbol{X}_{h-1}$ and Y_{h-1}, and compute the estimated accumulated search cost of this configuration. Then, we choose the configuration with the smallest accumulated search cost. However, it is costly to directly solve a number of k-piecewise linear regression problems.

Instead, we adapt an efficient greedy merging algorithm [11] to iteratively solve a series of k-piecewise linear regression problems with input $\boldsymbol{X}_{h-1}$ and Y_{h-1}. At each iteration, we generate $k-1$ break points and calculate the estimated accumulated search cost for them. Those break points induce the smallest cost form $\boldsymbol{X}_h$, the basis for creating the nodes at height h. Algorithm 3 finds the suitable n_h and $\boldsymbol{X}_h$ and generates the nodes at height h. In lines 6–7, ω is a pre-defined average maximum fanout for DILI's nodes. In

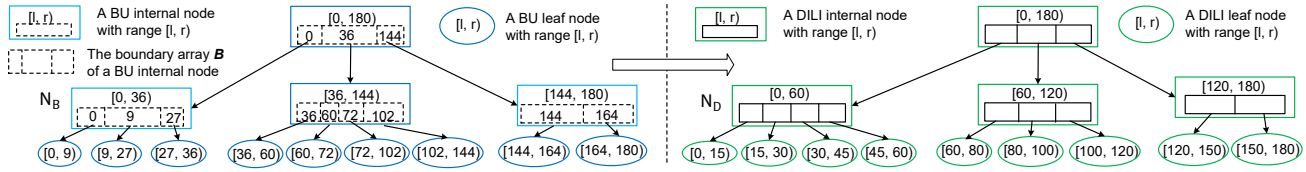


Figure 3: Building DILI based on the BU-Tree

our implementation, we set ω to 4096 as a DILI with good search performance cannot have too many nodes.

Algorithm 3 GREEDYMERGING($\mathbf{N}^{h-1}, \boldsymbol{X}_{h-1}$)

1: $k \leftarrow \frac{n_{h-1}}{2}$, $\boldsymbol{Y}_{h-1} \leftarrow [\tilde{n_{h-1}}]$, $n_{h-1} \leftarrow |\boldsymbol{X}_{h-1}|$
▷ The last set may contain 3 elements
2: $\mathcal{I}^k \leftarrow \{\{0,1\},\{2,3\},\cdots,\{2k-2,(2k-1),n_{h-1}-1\}\}$
3: $\forall k$ and $0 \le i < k$, let I_i^k denote the ith element of $\mathcal{I}^k$
4: For any indices set I, let $\gamma(I) \leftarrow \text{RMSE}(\boldsymbol{X}_{h-1}, \boldsymbol{Y}_{h-1}, I)$
5: $\forall k, i$, let $s_i^k \leftarrow \gamma(I_i^k), m_i^k \leftarrow \gamma(I_i^k \bigcup I_{i+1}^k)$
6: Set ω to be a large number and $k_{min} \leftarrow \frac{n_h}{\omega}$ ▷ In practice, we set $\omega = 2,048$.
7: **while** $k \ge k_{min}$ **do** ▷ Iterative greedy merging
▷ $m_i^k = \gamma(I_j^k \bigcup I_{j+1}^k), s_i^k = \gamma(I_j^k), s_{i+1}^k = \gamma(I_{j+1}^k)$
8: $u = \text{argmin}_i\, m_i^k - s_i^k - s_{i+1}^k$
9: $\mathcal{I}^{k-1} \leftarrow \{I_0^k, I_1^k, \cdots, I_u^k \bigcup I_{u+1}^k, I_{u+2}^k, \cdots, I_{k-1}^k\}$
10: $s_u^{k-1} \leftarrow m_u^k$ and calculate m_{u-1}^{k-1} and m_u^{k-1}
11: $\forall u < i < k-1, s_i^{k-1} \leftarrow s_{i+1}^k, m_i^{k-1} \leftarrow m_{i+1}^k$
12: $k \leftarrow k-1$
▷ Generate new break points
13: $\forall 0 \le i < k, q_i \leftarrow \inf I_i^k$
14: $\boldsymbol{B}_k = [\boldsymbol{X}_{h-1}[q_0], \boldsymbol{X}_{h-1}[q_1], \cdots, \boldsymbol{X}_{h-1}[q_{k-1}]]$
15: $\varepsilon_k \leftarrow \mathbf{T}_{\mathbf{ea}}^{\mathbf{B}}(\boldsymbol{B}_k, \boldsymbol{X})$ ▷ Get $\boldsymbol{B}_k$'s estimated accumulated search cost
16: $\boldsymbol{X}_h \leftarrow \boldsymbol{B}_{n_h}, n_h \leftarrow \text{argmin}_k\, \varepsilon_k$
17: $\forall 0 \le i < n_h, \mathcal{F}_i^h \leftarrow \text{LEASTSQUARES}(\boldsymbol{X}_{h-1}, \boldsymbol{Y}_{h-1}, I_i^{n_h})$
18: $\mathbf{N}^h \leftarrow$ the BU nodes at height h described in Eq. 3 or Eq. 4
19: **return** $n_h, \boldsymbol{X}_h, \mathbf{N}^h, \varepsilon$

The value of k is decided as follows. Initially, k is set to be $\frac{n_{h-1}}{2}$ and the list $\boldsymbol{X}_{h-1}$ is partitioned into k pieces (lines 1-2). At each iteration (lines 7–15), we merge two continuous pieces that result in the most linear loss increase and decrease the value of k by 1 (lines 8-12). We do not need to calculate the linear loss *w.r.t.* every piece. Instead, we maintain two number s_i^k and m_i^k for the ith piece I_i^k. They equal the linear loss *w.r.t.* I_i^k and $I_i^k \bigcup I_{i+1}^k$, respectively. Each iteration involves the change of one piece only. Thus, after deciding merging I_u^k and I_{u+1}^k, we only need to update the values of s_u^{k-1} and m_u^{k-1} *w.r.t.* I_u^{k-1} (*i.e.*, $I_u^k \bigcup I_{u+1}^k$) and m_{u-1}^{k-1} *w.r.t.* I_{u-1}^{k-1}. Clearly, $s_u^{k-1} = m_u^k$ and thus only two calculations of m_{u-1}^{k-1} and m_u^{k-1} are needed. Meanwhile, the break points of the k-piecewise linear function $\boldsymbol{B}_k$ are generated (lines 13-14). We compute $\mathbf{T}_{\mathbf{ea}}^{\mathbf{B}}(\boldsymbol{B}_k, \boldsymbol{X})$ for each k (line 13). After the iterations, we set $\boldsymbol{X}_h$ to $\boldsymbol{B}_{n_h}$ and n_h to the k with the smallest $\mathbf{T}_{\mathbf{ea}}^{\mathbf{B}}(\boldsymbol{B}_k, \boldsymbol{X})$ (line 16). The nodes at height h are then created (lines 17–18) according to Eq. 3 and Eq. 4.

At each iteration, we calculate m_{u-1}^{k-1} and m_u^{k-1} to estimate the linear losses *w.r.t.* two pieces. In implementation, we make the number of items in each piece smaller than a pre-defined threshold of 2ω. Thus, both calculations run in time $O(1)$. We use a priority queue to store $d_u^k = m_i^k - s_i^k - s_{i+1}^k$ for all i. Thus, the time complexity of selecting u (line 9) is $O(k) = O(n_h)$. Besides, the calculation of the estimated accumulated search cost of $\boldsymbol{B}_k$ runs in time $O(1)$. In summary, the time complexity of Algorithm 3 is $O(n_h \log_2 n_h)$.

4.3 BU-Tree based Bulk Loading for DILI

Algorithm 4 formalizes bulk loading for DILI. In line 1, the BU-Tree is created by BUILDBUTREE (Algorithm 2). Let H be the BU-Tree's height. At any height $h \le H$, DILI and the BU-Tree have the same number of nodes, but the node layouts may be different. Based on the BU-Tree, we grow DILI top down (lines 3–7). The range of DILI's root node Root is set to the counterpart in the BU-Tree. Root is created by the recursive function CREATEINTERNAL (line 7).

Algorithm 4 BULKLOADING($\boldsymbol{P}$)

1: BURoot $\leftarrow$ BUILDBUTREE($\boldsymbol{P}$)
2: $H \leftarrow$ the height of BURoot
3: Get $\mathbf{N}^0, \mathbf{N}^1, \cdots, \mathbf{N}^{H-1}$ from N′
4: **for** $i \in \{0, 1, \cdots, H-1\}$ **do**
5: $\boldsymbol{\theta}^i = [\mathbf{N}^i[0].\text{lb}, \mathbf{N}^i[1].\text{lb}, \cdots, \mathbf{N}^i[|\mathbf{N}^i|-1].\text{lb}]$
6: $\boldsymbol{\Theta} \leftarrow [\boldsymbol{P}, \boldsymbol{\theta}^0, \boldsymbol{\theta}^1, \cdots, \boldsymbol{\theta}^{H-1}]$
7: Root $\leftarrow$ CREATEINTERNAL(BURoot.lb, BURoot.ub, H, $\boldsymbol{\Theta}$)
8: **return** Root
9: **function** CREATEINTERNAL(lb, ub, h, $\boldsymbol{\Theta}$)
10: $\text{N}_\text{T} \leftarrow$ an empty DILI internal node
11: $\text{N}_\text{T}.\text{lb} \leftarrow$ lb, $\text{N}_\text{T}.\text{ub} \leftarrow$ ub
12: $\text{N}_\text{T}.\texttt{fo} \leftarrow \#\{x | x \in \boldsymbol{\theta}, \text{lb} \le x < \text{ub}\}, \boldsymbol{\theta} \leftarrow \boldsymbol{\Theta}[h-1]$
13: $\text{N}_\text{T}.\mathcal{LR}(x) = a + bx, a \leftarrow -b \times \text{lb}, b \leftarrow \frac{\text{N}_\text{T}.\texttt{fo}}{\text{ub}-\text{lb}}$
14: **for** $i \in \{0, 1, \cdots, \text{N}_\text{T}.\texttt{fo}-1\}$ **do**
15: $l \leftarrow \text{lb} + \frac{i}{b}, u \leftarrow \text{lb} + \frac{i+1}{b}$ ▷ The lower/upper bound
16: **if** h is 1 **then** ▷ Child nodes are leaf nodes
17: $\text{N}_\text{T}.\boldsymbol{C}(i) = \text{CREATELEAFNODE}(l, u, \boldsymbol{\Theta}[0])$ ▷ $\boldsymbol{\Theta}[0]$ is $\boldsymbol{P}$
18: **else** $\text{N}_\text{T}.\boldsymbol{C}(i) = \text{CREATEINTERNAL}(l, u, h-1, \boldsymbol{\Theta})$
19: **return** N_T
20: **function** CREATELEAFNODE(lb, ub, $\boldsymbol{P}$)
21: $l \leftarrow \text{argmin}_i\, \boldsymbol{P}[i].key \ge \text{lb}$, $u \leftarrow \text{argmin}_i\, \boldsymbol{P}[i].key \ge \text{ub}$
22: $M \leftarrow |\boldsymbol{P}_\text{D}|, \boldsymbol{P}_\text{D} \leftarrow \boldsymbol{P}[l:u]$
23: $\text{N}_\text{D} \leftarrow$ an empty DILI leaf node, $\text{N}_\text{D}.\Omega \leftarrow M$
24: $\text{N}_\text{D}.\mathcal{LR} \leftarrow \text{LEASTSQUARES}(\text{KEYS}(\boldsymbol{P}_\text{D}), [\tilde{M}])$
25: LOCALOPT($\text{N}_\text{D}, \boldsymbol{P}_\text{D}$)
26: **return** N_D

To create an internal node N_T (lines 9–19), we set its range according to the input bounds (line 11), its fanout to the number of BU nodes at height $h-1$ whose range is covered by N_T's range (line 12), and its linear regression model accordingly (line 13). We recursively create $\text{N}_\text{T}.\texttt{fo}$ nodes and make them equally divide N_T's range (lines 14–18). These nodes compose $\text{N}_\text{T}.\boldsymbol{C}$.

When creating a leaf node N_D (lines 20–26), we include in $\boldsymbol{P}_\text{D}$ the pairs with keys from N_D's range (lines 21–22). The model $\text{N}_\text{D}.\mathcal{LR}$ is trained with the input KEYS($\boldsymbol{P}_\text{D}$) (lines 24). The function LOCALOPT distributes the pairs to the entry array $\text{N}_\text{D}.\boldsymbol{V}$ (line 25), performing a local optimization on N_D. The details will be given in Section 5.

Fig. 3 exemplifies building DILI. The ith internal nodes of BU-Tree and DILI at height h may have different fanouts. For example, when $h = 1$, node $\text{N}_\text{T}^\text{B}$ (in the BU-Tree) has 3 child nodes but the DILI node $\text{N}_\text{T}^\text{D}$'s fanout is 4, because $\text{N}_\text{T}^\text{D}$'s range $[0, 60)$ covers the left boundaries of the first four BU leaf nodes' ranges.

4.4 Remarks

In Section 4.2.2, $\mathbf{T}_{\mathbf{ns}}^{\mathbf{B}}(\text{N}_\text{i}, x)$ estimates $\mathbf{T}_{\mathbf{is}}(x)$ or $\mathbf{T}_{\mathbf{ds}}(\text{N}_\text{i}, x)$ in Eq. 2, depending on if N_i is internal or not. If N_i is a BU leaf node (*i.e.*, $h = 0$), $\mathbf{T}_{\mathbf{ns}}^{\mathbf{B}}(\text{N}_\text{i}, x)$ and $\mathbf{T}_{\mathbf{ds}}(\text{N}_\text{i}, x)$ are the same as long as N_i.ifL is true. Otherwise, $\mathbf{T}_{\mathbf{ns}}^{\mathbf{B}}(\text{N}_\text{i}, x)$ is not the same as $\mathbf{T}_{\mathbf{is}}(x)$. Our DILI bulk loading (Algorithm 4) makes the leaf node layouts of the BU-Tree and DILI as alike as possible. However, lower accuracy of BU internal nodes' linear regression models would cause the two layouts to be more different. To this end, we modify $\mathbf{T}_{\mathbf{ns}}^{\mathbf{B}}(\text{N}_\text{i}, x)$ to strike

a trade-off between the BU-Tree's height and the accuracy of its models. In addition, internal nodes at a higher height tend to have less impact on the layout of DILI's leaf nodes. Thus, we multiply $\mathbf{t}_{\mathbf{E}}^{\mathbf{B}}(\mathsf{N}_\mathsf{i}, x)$ by a factor ρ^h in Eq. 5 to reflect this effect.

Our cost models of BU-Tree and DILI consider the effect of the cache locality and attempt to strike a good balance between the node fanouts and the tree height. In contrast, the B+Tree restricts its node fanout to a pre-defined range $[\frac{m}{2}, m]$. Thus, to find the next child node to visit in an internal node's child array, the B+Tree binary search needs to access the array $\lceil log_2 m \rceil$ times in the worst case. As m is relatively large, the local search inside a B+Tree node often triggers many cache misses. In contrast, finding the required child node in DILI triggers only one cache miss. Also, compared to B+Tree, DILI's internal nodes have larger average fanouts. Thus, it usually has a wider and shallower structure such that a search needs to traverse fewer nodes to locate the relevant leaf nodes.

Note that ALEX partitions the key space in a relatively static way as its node fanout is always a power of 2. This causes its linear models to have relatively low accuracy. To this end, ALEX adopts a gapped array to increase the model accuracy in its leaf nodes. Still, this cannot guarantee optimal model accuracy—ALEX needs more time and triggers more cache misses than DILI, as shown in Section 7. Unlike ALEX, our local optimization (to be detailed in Section 5) makes the models in DILI's leaf nodes perfectly accurate and thus shortens DILI's search time. Nevertheless, DILI still outperforms ALEX even without the local optimization.

5 LOCAL OPTIMIZATION OF DILI

A leaf node $\mathsf{N_D}$'s linear model $\mathsf{N_D}.\mathcal{LR}$ approximates the relation between keys and their positions in the array $\mathsf{N_D}.\boldsymbol{V}$. However, no model can always make perfect predictions. Our experiments imply that the exponential search in the leaf nodes often forms a bottleneck. Also, the leaf node structure does not consider insertions. On average, half pairs covered by a leaf node needs to be shifted for a single insertion. To address these issues, we propose a local optimization for DILI's leaf nodes. It helps the linear models make 100% accurate predictions and avoid element shifting in insertions.

Algorithm 5 LocalOpt($\mathsf{N_D}, P_\mathsf{D}$)

1: $N \leftarrow |P_\mathsf{D}|, \boldsymbol{X} \leftarrow [x_0, \cdots, x_{N-1}]$ where $x_i = P_\mathsf{D}[i].key$
2: $\mathsf{N_D}.\Delta \leftarrow 0$, $\mathsf{N_D}.\mathtt{fo} \leftarrow \eta \mathsf{N_D}.\Omega (\eta > 1)$
3: $\mathtt{fo} \leftarrow \mathsf{N_D}.\mathtt{fo}$ and allocate $\mathsf{N_D}.\boldsymbol{V}$ with $\mathtt{fo}$ NULLs
4: Define $f_\mathsf{D}(x) \triangleq \min(\mathtt{fo}-1, \max(0, \mathsf{N_D}.\mathcal{LR}(x)))$
5: $\forall\, i = 0$ to $N-1$, set $\boldsymbol{P}_\mathsf{D}^i$ an empty list
6: **for** $i \in \{0, 1, \cdots, N-1\}$ **do**
7: $\quad t \leftarrow f_\mathsf{D}(x_i)$, $\boldsymbol{P}_\mathsf{D}^t$.append($P_\mathsf{D}[i]$)
8: **for** $t \in \{0, 1, \cdots, \mathtt{fo}-1\}$ **do**
9: $\quad$ **if** $|\boldsymbol{P}_\mathsf{D}^t| = 1$ **then**
10: $\quad\quad \mathsf{N_D}.\boldsymbol{V}[t] \leftarrow \boldsymbol{P}_\mathsf{D}^t[0]$, $\mathsf{N_D}.\Delta$+= 1
11: $\quad$ **else if** $|\boldsymbol{P}_\mathsf{D}^t| > 1$ **then**
12: $\quad\quad$ Create a new leaf node N' and train $\mathsf{N}'.\mathcal{LR}$ with the input $\boldsymbol{P}_\mathsf{D}^t$
13: $\quad\quad$ LocalOpt($\boldsymbol{P}_\mathsf{D}^t, \mathsf{N}'$)
14: $\quad\quad \mathsf{N_D}.\boldsymbol{V}[t] \leftarrow$ the pointer to N', $\mathsf{N_D}.\Delta$ += $|\boldsymbol{P}_\mathsf{D}^t| + \mathsf{N}'.\Delta$
15: $\quad$ **else** $\mathsf{N_D}.\boldsymbol{V}[t] \leftarrow$ NULL
16: $\mathsf{N_D}.\kappa = \frac{\mathsf{N_D}.\Delta}{\mathsf{N_D}.\Omega}$

Our local optimization (Algorithm 5) starts at the final step in the procedure of creating a leaf node $\mathsf{N_D}$ (line 25 in Algorithm 4). Suppose there is one and only one pair whose predicted position by $\mathsf{N_D}.\mathcal{LR}$ is t, we set $\mathsf{N_D}.\boldsymbol{V}[t]$ to be this pair (lines 6-7, 9-10). If multiple pairs conflict at position t, we create a new leaf node N' to cover them (lines 11-12). The LocalOpt function is recursively called such that the entry array $\mathsf{N}'.\boldsymbol{V}$ is created to organize the conflicting pairs (line 13). After that, the pointer to N' is assigned to $\mathsf{N_D}.\boldsymbol{V}[t]$. Those slots of $\mathsf{N_D}.\boldsymbol{V}$ without a pair are set to NULL (lines 15). In practice, we set $\mathsf{N_D}.\mathtt{fo}$ to $\eta \cdot \mathsf{N_D}.\Omega$, where η is an enlarging ratio, such that continuous keys are more likely assigned in different slots (line 2) and conflicts are reduced. Fig. 4 illustrates the structure of a typical leaf node.

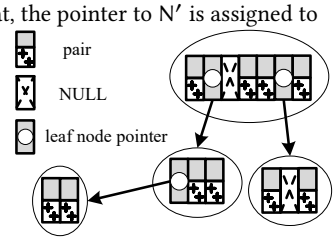


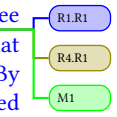
Figure 4: Leaf node structure

The local optimization renders search via DILI slightly different from Algorithm 1, as shown in Algorithm 6. After finding the highest leaf node $\mathsf{N_D}$ covering x (line 1), the loop starts from obtaining the value of the returned element p in $\mathsf{N_D}.\boldsymbol{V}$ through $\mathsf{N_D}.\mathcal{LR}$. If p points to another leaf node N', the loop continues by setting $\mathsf{N_D}$ to N' (lines 4-5). Otherwise, the loop ends and returns the right result, depending on if p is a pair with key equal to x (lines 6-8).

Algorithm 6 SearchWOpt(Root, x)

1: $\mathsf{N_D} \leftarrow$ locateLeafNode(Root, x)
2: **while** True **do**
3: $\quad pos \leftarrow \lfloor \mathsf{N_D}.\mathcal{LR}(x) \rfloor$, $p \leftarrow \mathsf{N_D}.\boldsymbol{V}[pos]$
4: $\quad$ **if** p points to a leaf node N' **then**
5: $\quad\quad \mathsf{N_D} \leftarrow \mathsf{N}'$
6: $\quad$ **else if** $p \neq$ NULL and $p.key == x$ **then**
7: $\quad\quad$ **return** $p.key$
8: $\quad$ **else return** NULL

Fig. 1 gives an example of search via locally optimized DILI for a key $x = 101$. The root node R's key range is $[0, 240)$. R has three child nodes. It is easy to derive that $\mathsf{R}.a = 0$, $\mathsf{R}.b = \frac{1}{80}$ such that R's linear model equally divides its key range into three parts. By a simple calculation $\lfloor \mathsf{R}.a + \mathsf{R}.b \times x \rfloor = 1$, we know x is covered by R's second child node, namely $\mathsf{N_T}$ in Fig. 1 (Step-1). $\mathsf{N_T}$ equally divides its range $[80, 160)$ to its four children with $\mathsf{N_T}.a = -4$ and $\mathsf{N_T}.b = 0.05$. The search goes into $\mathsf{N_T}$'s second child node $\mathsf{N_D^1}$ whose range is $[100, 120)$ (Step-2). $\mathsf{N_D^1}$ is a leaf node whose linear model is trained with the keys covered by its range, in a different way from that of R and $\mathsf{N_T}$. At the predicted slot position $\lfloor \mathsf{N_D^1}.a + \mathsf{N_D^1}.b \times x \rfloor = \lfloor -9 + 0.1 \times 101 \rfloor = 1$ is another leaf node $\mathsf{N_D^2}$ (Step-3). Note that two keys 101 and 102 conflict at the same slot in $\mathsf{N_D^1}.\boldsymbol{V}$. Thus, $\mathsf{N_D^2}$ is generated to store them as the local optimization. Finally, $\mathsf{N_D^2}.\mathcal{LR}$ predicts for x a pair p at position 0 in $\mathsf{N_D^2}.\boldsymbol{V}$. The output pointer $p_x = p.ptr$ points to the data record identified by x.



6 DATA UPDATES IN DILI

6.1 Insertions

Insertions via DILI are logically simple and efficient. Our insertion algorithm avoids element shifting that happen to B+Tree and ALEX, and it redistributes pairs when insertions degrade the search performance. The details are shown in Algorithm 7.

To insert a pair p to DILI, the first step calls the function locateLeafNode (defined in Algorithm 1) to find the leaf node $\mathsf{N_D}$ that supposedly covers $p.key$ (line 1). Next, the algorithm inserts p into $\mathsf{N_D}$ by calling the recursive function insertToLeafNode (line 2). We use the model $\mathsf{N_D}.\mathcal{LR}$ to calculate the position pos in $\mathsf{N_D}.\boldsymbol{V}$ for

p. Suppose that at position pos is the element p' (line 4). If p' is NULL, the pos-th slot of $N_D.\mathcal{LR}$ is empty. We simply place p at the slot. Then, the cost of searching for all pairs except for p covered by N_D does not change. Searching for p from N_D needs only one extra entry access, so we simply add one to $N_D.\Delta$ (lines 6-7). Here, $N_D.\Delta$ denotes the total number of entries to be accessed to search for all keys covered by N_D, starting from N_D. If p' points to another leaf node N', we insert p into N''s entry array (line 10). This time the change of $N_D.\Delta$ is related to those pairs in N'. Thus, we record $N'.\Delta$ before the insertion (line 9). The increment of $N_D.\Delta$ is the change of $N'.\Delta$ plus 1 (line 11). If p exists (line 12), we do nothing (line 13). If a conflict happens (line 14), we need to replace the pair p' with a new leaf node covering p and p' at the pos-th position of $N_D.\boldsymbol{V}$ (lines 15-17). In this case, $N_D.\Delta$ is increased by three (line 18): one for searching for p' and two for p.

Algorithm 7 INSERT(Root, p)

```
1:  N_D ← LOCATELEAFNODE(Root, p.key)
2:  return INSERTTOLEAFNODE(N_D, p)
3:  function INSERTTOLEAFNODE(N_D, p)
4:      p' ← N_D.V[pos], pos ← N_D.LR(p.key)
5:      notExist ← True
6:      if p' = NULL then
7:          N_D.V[pos] ← p, N_D.Δ += 1                ▷ insert p to an empty slot
8:      else if p' points to another leaf node N' then
9:          Δ' ← N'.Δ
10:         notExist ← INSERTTOLEAFNODE(N', p)
11:         N_D.Δ += 1 + N'.Δ − Δ'
12:     else if p'.key = p.key then
13:         notExist ← True                           ▷ p exists
14:     else
15:         create a new leaf node N' to cover p and p'
16:         N'.Δ ← 2, N'.Ω ← 2 and train N'.LR
17:         N_D.V[pos] ← the pointer to N'
18:         N_D.Δ += 1 + N'.Δ
19:     N_D.Ω += (notExist = True ? 1 : 0)
20:     if notExist = True and N_D.Δ / N_D.Ω > λ × N_D.κ then
21:         collect all pairs covered by N_D and store them in list P_D
22:         N_D.fo ← N_D.Ω × r, r ← φ(N_D.α), N_D.α += 1
23:         N_D.LR ← LEASTSQUARES(KEYS(P_D), [N_D.Ω])
24:         N_D.LR.a ← N_D.LR.a × r, N_D.LR.b ← N_D.LR.b × r
25:         LOCALOPT(N_D, P_D)
26:     N_D.κ = N_D.Δ / N_D.Ω
27:     return notExist
```

However, many new nodes for conflicting keys may increase the depth of leaf nodes wildly. Thus, it is necessary to adjust the layout of some leaf nodes when the search performance degrades. We observe the relationship among $N_D.\Delta$, $N_D.\Omega$ and $N_D.\kappa$, and uses a flexible strategy to decide if a leaf node N_D should be adjusted. When inserting a pair p to N_D, from this node, if the average number of entries need to be accessed to search for a pair (*i.e.*, $\frac{N_D.\Delta}{N_D.\Omega}$) is larger than a pre-defined threshold (line 20), we think the insertions degrade the search performance. Thus, N_D is adjusted, which starts by collecting all pairs covered by N_D (line 21). Then, the capacity of $N_D.\boldsymbol{V}$ is enlarged and we train N_D's linear model accordingly such that conflicts will happen more rarely (lines 22-24). Finally, we redistribute the pairs with the local optimization (line 25).

The pre-defined threshold is set to $\lambda \cdot N_D.\kappa$ ($\lambda > 1$) where $N_D.\kappa$ is the average number of accessed entries in search for a pair covered by N_D **after executing the last local optimization at** N_D. In our experiments, λ is set to 2. As a result, if the average cost per search *w.r.t.* N_D doubles after a series of insertions, we deem some nodes under N_D become too deep and the performance of searching for relevant keys degrades dramatically. In this case, it is better to collect all pairs that N_D covers, retrain $N_D.\mathcal{LR}$ and redistribute those pairs. Finally, the value of $N_D.\kappa$ will be updated (line 26).

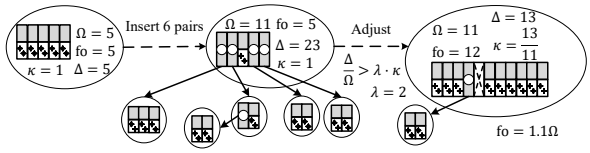


Figure 5: Adjusting a leaf node after insertions

When adjusting the leaf node N_D, we also set $N_D.\texttt{fo}$ to be larger than $N_D.\Omega$. The gap between them grows with more adjustments (line 22). A simple yet reasonable assumption is that the more adjustments, the more frequently relevant pairs are accessed. Also, more adjustments usually mean more conflicts. Thus, to reduce the number of conflicts at N_D, we enlarge the capacity of $N_D.\boldsymbol{V}$, making more slots for pairs. In our experiments, the enlarging ratio $\varphi(N_D.\alpha) \triangleq \min(\eta + 0.1 \times N_D.\alpha, 4)$, where η carries the same meaning with that in Algorithm 5. $\varphi(\cdot)$ can be any monotonically increasing function and its derivative should consider the memory usage. Our strategy of having redundancy in frequently adjusted nodes is similar to but more flexible than the usage of gapped array in ALEX [18]. Fig. 5 gives an example of adjusting a leaf node.

6.2 Deletions

To delete a pair with the key x, we call LOCATELEAFNODE to find the highest leaf node N_D covering x (line 1). Next, we use DELETEFROMLEAFNODE to delete the pair from N_D (line 2). It recursively checks if there is a pair, covered by N_D or a leaf node underneath, having the key x (lines 4-14). If the pair is found, we remove it by setting the corresponding slot in the leaf node's entry array to NULL (lines 5-6). Otherwise, DELETEFROMLEAFNODE simply returns False (lines 7-8). After the removal, the values of N_D's some fields will change, like the number of pairs contained in N_D and the average cost of searching keys from N_D. Thus, we update the values of $N_D.\Delta$, $N_D.\Omega$ and $N_D.\kappa$ (lines 6, 10-12, 16). If N_D's child leaf node N' contains only one pair p'' after the removal, we simply delete N' and replace the pointer to it with p'' in $N_D.\boldsymbol{V}$ (lines 13-15).

Algorithm 8 DELETE(Root, x) ▷ x is the key to be deleted

```
1:  N_D ← LOCATELEAFNODE(Root, x)
2:  return DELETEFROMLEAFNODE(N_D, x)
3:  function DELETEFROMLEAFNODE(N_D, x)
4:      p' ← N_D.V[pos], pos ← f_D(x), exist ← True
5:      if p'.key = x then
6:          N_D.V[pos] ← NULL, N_D.Δ -= 1             ▷ delete p' from N_D.V
7:      else if p' = NULL then
8:          exist ← False                             ▷ corresponding pair does not exsit
9:      else if p' points to another leaf node N' then
10:         Δ' ← N'.Δ
11:         exist ← DELETEFROMLEAFNODE(N', x)
12:         N_D.Δ -= 1 + Δ' − N'.Δ
13:         if N'.Ω = 1 then                          ▷ N' covers only one pair p''
14:             N_D.V[pos] ← the remaining one pair p'' contained in N'
15:             N_D.Δ -= 1 and delete N'
16:     N_D.Ω -= (exist = True ? 1 : 0), N_D.κ = N_D.Δ / N_D.Ω
17:     return exist
```

7 EXPERIMENTAL STUDIES

DILI and all competitors are implemented in C++ [1] and evaluated using a single thread on a Ubuntu server with a 96-core Xeon(R) Platinum 8163 CPU and 376 GB memory.

7.1 Experimental Settings

Datasets. We use four real datasets from the SOSD benchmark [35] and one synthetic dataset.

- FB [2] contains 200M Facebook user ids.
- WikiTS [3] contains 200M unique request timestamps (in integers) of log entries of the Wikipedia web-site.
- OSM [4] contains 800M ids of OpenStreetMap cells.
- Books [5] contains 800M ids of books in Amazon.
- Logn contains 200M unique values sampled from a heavy-tail log-normal distribution with $\mu = 0$ and $\sigma = 1$.

For each key, we associate it with a random integer number and pack them as a simulated record. The records are stored in an data array. For each record, its key and address together form a pair. For the pairs for index's bulk loading, we sort them according to their keys and feed them to index's bulk loading algorithm.

Competitors. We compare DILI with the following methods:

- **BinS** does a binary search over the whole sorted key set to find the position of the given search key.
- **B+Tree** [15]: We use a production quality B+Tree implementation stx::btree for comparison [6].
- **MassTree** [34] is a variant of B-Tree which improves cache-awareness by employing a trie-like [12] structure.
- **RMI** [29] is built through a linear stage and a cubic stage.
- **ALEX** [18] is an in-memory learned index which partition keys into leaf nodes in a relatively static way [7].
- **RS** (RadixSpline) [26] uses a linear spline to approximate the CDF of the data and a radix table to index spline points.
- **PGM** (PGM-index) [20] contains multiple levels, each representing an error-bounded piece-wise linear regression [8].
- **LIPP** [43] can be seen as a special RMI. Its root node uses a linear regression model with the range of $[0, N)$, where N is the dataset cardinality. At lower levels, LIPP recursively uses linear regression models to partition search keys until each key's position is accurately predicted. LIPP aims to predict as many keys' position as possible with only one model [9].

For RMI and RS, we adopt the implementations in SOSD [10, 35].

Table 2 summarizes the properties of all indexes. The better performance is indicated in bold.

Table 2: Properties of different methods

Method	Support update	Consider data distribution	Extra local search	Tree height	Memory cost
B+Tree	✓	×	✓	medium	medium
RMI	×	×	✓	**low**	**small**
RS	×	✓	✓	**low**	**small**
PGM	✓	×	✓	high	medium
Masstree	✓	×	✓	medium	medium
ALEX	✓	✓	✓	medium	medium
LPP	✓	×	×	medium	large
DILI	✓	✓	×	**low**	medium

Evaluation Metrics. We use two performance metrics: **Lookup** is the average lookup time per query, including the time spent in the index and in finding the records in the data array. **Throughtput** is the number of operations, including query, insertion and deletion, that of a method can handle per second.

Parameter Settings. Table 3 lists the parameter settings for B+Tree and ALEX. They are built with bulk loading for better lookup and throughput performance. For RMI and RS, we follow [35] to use two settings with the largest (L) and smallest (S) memory costs.

In our machine, an LL-cache line is of 64 bytes and fetching a cache line from the memory costs 130 CPU cycles at worst [16, 21, 40]. A DILI (internal or leaf) node can be held in a single cache line. Therefore, we set $\theta_N = \theta_C = 130$. Executing a linear function as well as type casting cost about $\eta = 25$ cycles. Moreover, $\mu_L = 5$ and $\mu_E = 17$ cycles are spent on executing operations except accessing pairs in linear search and exponential search, respectively. The decaying rate ρ in Eq. 5, enlarging ratio η in Algorithm 5 and maximum fanout ω in Algorithm 3 are set to 0.2, 2 and 4,096, respectively.

Table 3: Parameter settings in experiments

Param	Description	Setting
Ω	Node fanout of a B+Tree	16, 32, 64, 128, 256, 512
Γ	Max node size of ALEX	16KB, 64KB, 1MB, 16MB, 64MB

7.2 Overall Query Performance

For each dataset, we build all indexes using the whole dataset P and randomly select 100M keys in KEYS(P) to form point queries. All competitors are built with their preferred parameter settings. Table 4 reports on the overall performance results of all methods on point queries. To investigate the effects of the local optimization, we also include the experimental results of DILI-LO, a DILI variant by disabling the local optimization in its leaf nodes, tightly arranging pairs in the entry arrays, and making search process follows Algorithm 1. We choose the LIPP as the fixed reference point as it is the best among all competitors. The color-encoding indicates how much faster or slower a model is against the reference point.

Table 4: Lookup time (ns) of all methods after bulk loading

Model	Config	FB	WikiTS	OSM	Books	Logn
BinS		819	822	839	844	817
B+Tree	Ω=16	629	633	578	584	624
	Ω=32	620	616	589	611	629
	Ω=64	658	649	641	651	653
	Ω=128	722	719	693	699	725
	Ω=256	794	790	776	775	790
	Ω=512	995	980	979	982	984
ALEX	Γ=16KB	655	580	544	509	463
	Γ=64KB	573	465	419	382	398
	Γ=1MB	490	248	281	274	259
	Γ=16MB	476	236	223	221	170
	Γ=64MB	462	252	234	203	161
RMI	(S)	833	806	1255	540	907
	(L)	215	175	166	221	208
RS	(S)	398	313	358	355	172
	(L)	305	264	211	217	132
MassTree		1245	1238	1500	1492	1220
PGM		483	468	474	457	453
LIPP		197	152	178	182	173
DILI-LO		240	168			142
DILI		**150**	**139**	**117**	**148**	**116**

DILI has clear advantages over other state-of-the-art methods. Compared to LIPP, DILI saves about 9% to 34% lookup time. The design of DILI's bulk loading algorithm make the keys in DILI's leaf nodes almost linearly distributed and the linear regression models well describe these distributions. Thus, conflicts happens more rarely in DILI. The traversal path of DILI is shorter than that of LIPP and other competitors, which results in DILI has better performance. Compared to ALEX and PGM, besides the shorter traversal path, DILI is able to avoid the search inside the leaf nodes and have clearer advantages. BinS, MassTree and all variants of B+Tree even needs to take 4–10 times of lookup time to search for a key on average. Also, DILI clearly outperforms RMI and RS on processing point queries. And it is noteworthy that RMI and RS do not support updates. These results illustrate DILI achieves large lookup

superiority over other alternatives. In addition, DILI consumes less lookup time than DILI-LO over all the five datasets. This verifies the effectiveness of the local optimization in DILI's leaf nodes.

As B+Tree with $\Omega = 32$, ALEX with $\Gamma = 16$MB and the large RMI and RS perform best among their variants, we will choose them as representatives and omit the evaluation of the other variants in the following sections. The parameter settings of these methods will also be omitted when we refer to them. Also, to save space, we will omit the comparable results on the datasets OSM and Books. The results on both datasets are similar to that on other datasets.

Cache Misses. DILI's advantage is partly due to that the design of DILI's structure makes DILI triggers fewer cache misses. A single LL-cache miss incurs 50-200 additional cycles [16, 29, 40]. In contrast, register operations like addition and multiplication cost 1-3 cycles only. Avoiding cache misses clearly speeds up query processing for DILI. Table 5 reports the average number of LL-cache misses for all methods. In particular, compared with ALEX and LIPP, DILI avoids up to 9.7 and 3.5 LL-cache misses per query.

R1.D2 M2

Table 5: #LL-cache misses of methods per point query

Dataset	B+Tree	RMI	RS	PGM	MassTree	ALEX	LIPP	DILI
FB	10.27	5.25	8.43	10.73	9.84	14.91	7.94	**4.88**
WikiTS	10.51	**4.60**	5.68	11.56	9.24	7.36	5.86	4.78
Logn	10.19	5.28	**3.22**	9.88	9.48	4.47	7.17	3.80
OSM	10.47	**3.89**	4.50	7.42	12.85	5.86	7.13	4.08
Books	10.46	6.02	4.74	7.38	13.02	**4.27**	7.81	4.31

Offline Construction Time. On a 100M dataset, the bulk loading of B+Tree, ALEX, LIPP and DILI takes less than 1, 2, 1 and 6 minutes, respectively. Each construction time grows almost linearly with the increase of the data size. The most time-consuming step in DILI's construction is the greedy merging algorithm to get the BU nodes at the bottom layer. A direct yet effective approach to make this step more efficient is sampling. When a piece I_u^k (in Algorithm 3) covers many keys, we could randomly or selectively sample part of the keys, *e.g.,* select one key out of two, to get the linear regression model and calculate the cost. The sampling strategy makes little influence on the whole BU-tree node layout and the performance of the generated DILI. However, it will clearly reduce the construction time of the BU-tree and DILI. Also, the bulk loading is one-time, a couple of more minutes of DILI highly pay off.

R4.R3 M2

Analysis of DILI's Construction. Table 6 shows DILI's minimum/maximum/average heights and the number of conflicts in DILI's construction for different datasets. Apparently, DILI has a shallow structure. The slot assignments for most pairs cause no conflicts. The average heights of the DILIs built on the Logn and WikiTS dataset are smaller than that of others. The reason is that the keys in both datasets are more linearly or piecewise linearly distributed. Thus, the linear regression models in the leaf nodes are able to make more accurate predictions and thus result in less conflicts.

R1.D8 M2

Table 6: The statistics of DILI

Dataset	Minimum height	Maximum height	Average height	# of conflicts per 1K keys
FB	3	8	3.45	227.1
WikiTS	3	6	3.09	44.4
Logn	3	4	3.01	1.2
OSM	3	9	3.26	117.7
Books	3	8	3.44	220.4

Index Size. Fig. 6 (a) displays the memory cost of different methods. RMI and RS consume the least memory. However, they do not need to store pairs in their structures and do not support data updates. DILI consumes more memory than B+Tree, PGM and ALEX due to the local optimization in the leaf nodes. A conflict will result in a new leaf node creation and an empty slot in the entry array. Nevertheless, our design strikes a trade-off between the memory cost and the query efficiency. Considering modern computers usually have huge memory, it is acceptable to improve the efficiency at the expense of some memory. Although LIPP also tries to strike a memory-efficiency trade-off and adopts a similar strategy for conflicts, its node layout is not so optimized as ours. Thus, LIPP results in more conflicts and memory costs. Its memory cost is at least one order of magnitude larger than others. On the other hand, after disabling the local optimization, the memory costs of the modified DILI-LO will become comparable with B+Tree. Meanwhile, the query performance does not degrade much.

R1.R3 M2 M5

Range Query Performance. The range query via DILI is implemented by searching for the lower bound key followed by a scan for the subsequent keys. Fig. 6 (b) reports on the average response time of B+Tree, PGM, ALEX, LIPP, DILI and DILI-LO on short range queries. Following the settings in [18], we restrict the number of keys covered by a range less than 100. All methods are built with bulk loading on a full $\boldsymbol{P}$. On each dataset, 10M random range queries are generated and executed. The advantage of DILI over the alternatives is less apparent than that in the point query performance comparison. This is attributed to that the pairs are not densely stored in the entry arrays in DILI's leaf nodes and DILI needs to distinguish between different entry types. Nevertheless, DILI achieves higher throughput than LIPP on all cases, and it is comparable to other competitors. Also, the performance of DILI could be improved by disabling the local optimization and tightly arrange pairs in the leaf nodes' entry arrays. In this case, the leaf nodes' entry arrays only cover pairs. Inside a leaf node, we only need to perform a sequential scan over the whole of part of the entry array and do not need to consider the entries of leaf node pointers and NULL flags. Referring to Fig. 6 (b), the average range query response time of DILI-LO gets improved over DILI.

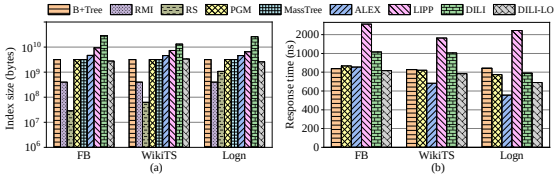


Figure 6: (a) Index size; (b) Range query results.

7.3 Performance on Different Workloads

We conduct experiments to compare DILI and the alternatives on different types of workloads: (1) The Read-only workload contains 100M point queries. (2) The Read-Heavy workload contains 50M insertions and 100M point queries. (3) The Write-Heavy workload contains 100M insertions and 50M point queries. (4) The Write-only workload contains 100M insertions. In each dataset $\boldsymbol{P}$, we randomly select 50% of the pairs as the initial dataset $\boldsymbol{P}_0$. The other 50% of $\boldsymbol{P}$ is named $\boldsymbol{P}_1$. All workloads are tested on an index with bulk loading of $\boldsymbol{P}_0$. Besides, the query keys are randomly selected from the KEYS($\boldsymbol{P}$), and the pairs to be inserted are randomly chosen from $\boldsymbol{P}_1$. Each workload is a random mix of queries and insertions. We run the workloads on different indexes for five times and obtain their average throughput. As RMI and RS do not support updates,

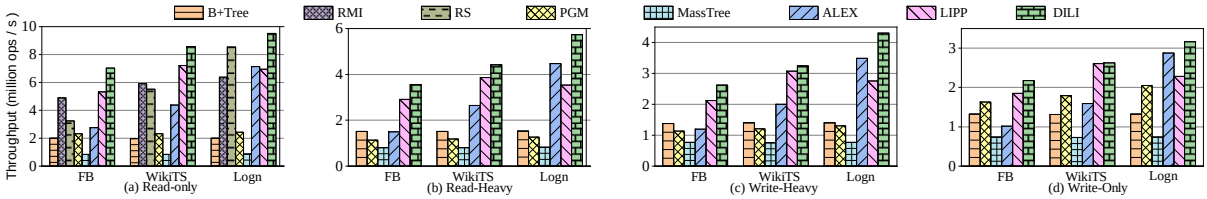


Figure 7: DILI vs. State-of-the-art methods: throughput comparisons on four workloads

they are excluded from the experiments involving insertions. The experimental results are shown in Fig. 7. Overall, DILI achieves the highest throughput on all workloads.

For the read-only workloads, compared to the others, DILI achieves shorter average search path. In particular, DILI accesses only 0.2-1 node per point query on average. This indicates that DILI utilizes the data distribution well and thus the learned models in its leaf nodes incur few conflicts. The alternatives need longer search paths queries and extra steps to carry out local search. RMI and RS achieves comparably long search paths with DILI. However, the effort of correcting their prediction results in lower throughput.

When more insertions are in the workloads, we see that DILI still outperforms others though its performance also degrades. The reason is that an insertion not only includes searching for a key but also writing a pair to an entry array. Also, new node creations are required to process conflicts. Moreover, adjustments occasionally happen to bound DILI's height. Even though insertions on DILI requires more time than queries, DILI is still able to deal with index structure change well and more efficient at insertions than others. PGM performs worst in these workloads as it needs $O(\log N)$ trees to support insertions and each query will search in all these trees. Compared to B+Tree and ALEX, DILI can avoid element shifting. Also, the new node creation in DILI is light-weight. In addition, compared to LIPP, DILI has shorter traversal path for insertions. Our strategy of setting more slot redundancy for leaf nodes more frequently accessed also avoid unnecessary node adjustments.

7.4 Effect of Many Deletions

We also experimentally investigate the effect of deletions on DILI, B+Tree, PGM MassTree and ALEX. LIPP is excluded as it does not support deletions. We first build each of them with bulk loading of the whole $\boldsymbol{P}$. Then we repeatedly delete/search for random keys from $\boldsymbol{P}$ via all methods and observe their changing throughput on three workloads: (1) Read-Heavy workload, which contains 100M lookups and 50M deletions; (2) Deletion-Heavy workload, which contains 100M deletions and 50M loopups. Fig. 8 shows the results.

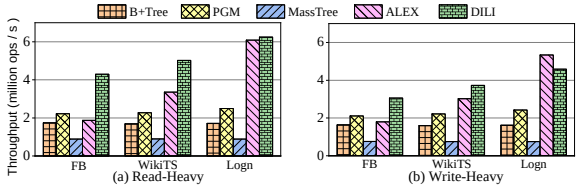


Figure 8: Performance after deletions

Referring to Figure 8, on Read-Heavy workload, DILI achieves up to 3.6×, 2.3×, 7.0× and 2.3× higher throughput than B+Tree, PGM, MassTree and ALEX, respectively. This illustrates that DILI maintains high performance on queries with deletions happening. On Deletion-Heavy workload, only ALEX performs a little better than DILI on Logn dataset. As ALEX almost adopts lazy deletion strategy, deleting a pair from ALEX almost equals searching for it. However, this strategy will cause its lookup time not decrease even through it index a small amount of data only. Actually, DILI performs much better than ALEX when the workload consists of more queries.

7.5 Hyperparameter Studies

To investigate the effects of hyperparameters on DILI, we use different hyperparameter values to build different versions of DILI and observe their performance on the FB dataset. We omit the results on other datasets as they are similar with those on the FB dataset.

Effects of ω. ω is used to control the least number of nodes at DILI's each level. According to our observations, the generated DILIs with the value of ω varied from 1,024 to 8,192 have the same node layout. DILI tends to have a wide structure. Thus, as long as the value of ω is large enough, it slightly influences the performance.

Effects of ρ. Table 7 shows the lookup time and memory costs of DILI with different values of ρ, which is the decaying rate of the impact of BU internal nodes at higher level on the layout of DILI's leaf nodes. Apparently, the value of ρ has little influence on DILI's overall structure and query performance. When the value of ρ is set to around 0.1, DILI performs the best.

Table 7: The effects of the hyperparameter ρ

Param	lookup time (ns)	Memory cost (10^9 bytes)	Average height
$\rho = 0.05$	154	9.327	3.441
$\rho = 0.1$	151	9.325	3.439
$\rho = 0.2$	153	9.328	3.441
$\rho = 0.5$	162	9.369	3.442

Effects of λ. Also, we investigate how λ influences the insertion performance of DILI, where λ is the pre-defined enlarging ratio for the fanouts of DILI's leaf node if a node adjustment is performed. First, the DILI is built with half of the dataset. Then, we vary the value of λ and insert 100M keys to DILI. After that, search operations are conducted. Table 8 shows the average insertion time, the average results with different λ values on the FB dataset. Results on other dtaasets are similarly and thus omitted. The insertion performance of DILI is almost not influenced by λ. When the value of λ is set to 2, DILI achieves the best lookup time and the shortest tree structure.

Table 8: The effects of the hyperparameter λ

Param	Insertion time (ns)	lookup time (ns)	Memory cost (10^9 bytes)	Average height
$\lambda = 1.5$	483	182	8.924	3.866
$\lambda = 2$	487	180	8.911	3.865
$\lambda = 4$	488	184	8.912	3.868
$\lambda = 8$	485	185	8.913	3.869

7.6 DILI vs RMI and BU-Tree

Our last experiment investigates why DILI outperforms RMI and the BU-Tree (Section 4.1) on point queries. Note that DILI and BU-Tree process point queries in the same two-step fashion: finding the relevant leaf node (`Step-1`), followed by search inside the leaf node (`Step-2`). Similarly, the search process with RMI can be decomposed into two steps: the computation of the predicted position (`Step-1`) and the local search around the prediction (`Step-2`). We created the BU-Tree using the whole dataset $\boldsymbol{P}$ with Algorithm 2. DILI is

built based on it. Then, we compare the breakdown time cost of the three models. Table 9 reports on the experimental results.

Table 9: DILI vs RMI & BU-Tree

Dataset	Model	Step-1 (ns)	Step-2 (ns)	Total (ns)
FB	RMI	139	76	215
	BU-Tree	386	210	596
	DILI	**75**	**75**	**150**
WikiTS	RMI	138	**37**	175
	BU-Tree	377	110	487
	DILI	**75**	**64**	**139**
Logn	RMI	135	73	208
	BU-Tree	273	57	330
	DILI	**73**	**43**	**116**

R1.R2 M2 M3

DILI vs RMI DILI has clear advantage over RMI at Step-1. RMI is built through a linear stage and a cubic stage. Also, RMI needs to calculate the error bound when making predictions. In contrast, all DILI versions have two layers of internal nodes and the step of locating the leaf node requires the calculations of two linear models only. Thus, DILI requires less calculations.

At Step-2, RMI requires a local search to access the true position. The time cost of this step is correlated to the gap between the predicted position and the true position. In our experiments, the average gaps for the three datasets are 4.93, 2.97 and 53.9, respectively. DILI needs to access one or more leaf nodes and do a calculations by the linear model per leaf node. Thus, the time cost of DILI's Step-2 depends on the number of leaf nodes traversed by the searches. According to our statistics, the average numbers of leaf nodes accessed per search for the three datasets are only 1.16, 1.05 and 1.02, respectively. Thus, DILI is also more efficient at Step-2 on most cases.

DILI vs BU-Tree Compared to the BU-Tree, DILI incurs less time cost on both steps. In particular at Step-2, the time gap between both trees is much larger. All in all, DILI incurs considerably less total time as it is more efficient at finding the leaf nodes. This verifies the effectiveness of the design of DILI's internal nodes and the algorithm of building DILI from the BU-Tree.

R2.D6 M2 M5

The BU-Tree's construction time is about 5 minutes. It takes less than 1 minute only to build DILI based on the BU-Tree. The memory costs of the BU-Tree on the three datasets are 1.90-1.98$\times 10^9$ bytes, 1.40-3.17$\times 10^9$ bytes smaller than that of DILI. However, DILI is much more efficient at search.

8 RELATED WORK

B-tree variants. A B+Tree [15] is the most popular B-tree variant in which each internal node contains only keys, and the leaf nodes are chained with extra links. Digital B-trees [32] allows a node to use two pages via a hashing-like technique. The B-trie [12] combines B-Tree and trie [27] to index strings stored in external memory. MassTree [34] employs a trie-like concatenation of B-trees to improve cache-awareness in indexing key-value pairs. A BF-tree [13] replaces B-Tree leaf nodes with bloom filters to substantially reduce the index size. Unlike all B-tree variants, our DILI stores models instead of pointers in the nodes for indexing purpose.

Learned indexes for 1D keys. The recursive model index (RMI) [29] uses staged models. An internal model directs a key search to one of its child models and a bottom-level model predicts an error-bounded position in the database. RMI has inspired a number of learned indexes. To reduce index memory footprint, a FITing-Tree [22] uses linear models to replace the leaf nodes of a B-Tree. CARMI [46] applies data partitioning to RMI construction and supports data update. NFL [44] uses a normalizing flow techniques [39] to transform the key space for better approximation on the CDF. PGM-index [20] employs piecewise linear models to approximate the relationship between search keys and their positions in a database. Hermit [45] creates a succinct tiered regression search tree (TRS-tree) which passes a search query to an existing index for correlated columns. RadixSpline [26] uses a set of spline functions as the learned index that can be built in a single pass over sorted data. ALEX [18] trains accurate linear regression models to split the key space, organizes all models also in a tree-like structure, and uses a gapped array for each leaf node. ALEX supports updates. LIPP [43] uses kernelized linear functions as learned models that make perfect predictions. However, it does not make use of the information of data distribution. SOSD [35] is a preliminary benchmark for 1D learned indexes. FINEdex [30] is a fine-grained learned index scheme, which constructs independent models with a flattened data structure to process concurrent requests. APEX [33] combines the recently released persistent memory optimization [24] and ALEX to support persistence and instant recovery. The on-disk learned index prototype AirIndex [14] uses with a storage-aware auto-tuning method to minimize accesses to the external memory. To validate the effectivenesss of the existing updatable learned indexes, Wongkham et al. [42] conduct a comprehensive evaluation.

Learned indexes for multidimensional data. SageDB [28] extends RMI to index multidimensional data in a transformed 1D space. ZM-index [41] applies RMI to the Z-order curve [36] to process spatial point and range queries. ML-index [17] applies RMI to iDistance [25] to support queries on multidimensional data. Flood [37] and Tsunami [19] are learned indexes for in-memory multidimensional data, whereas LISA [31] and RSMI [38] are for disk-resident dynamic spatial data. In contrast, our DILI focuses on 1D data.

9 CONCLUSION AND FUTURE WORK

In this work, we design for in-memory 1D keys a distribution-driven learned tree DILI. Its nodes use linear regression models to map keys to corresponding children or records. An internal node's key range is equally divided by all its children, endowing internal models with perfect accuracy for finding the leaf node covering a key. We optimize DILI's node layout by a two-phase bulk loading approach. First, we create a bottom-up tree that balances the number of leaf nodes and tree height. Based on that, we determine for each DILI internal node its best fanout and local model. Also, we design algorithms for DILI for data updates. Extensive experimental results show that DILI clearly expand the state of the art.

For future research, it is relevant to adapt DILI to disk-resident data. To this end, the cost model in the BU-Tree's construction process should consider the expected IOs, striking a trade-off between the IO cost and the computational overhead. Also, the local optimization should be disabled as it may create leaf nodes covering few keys. Also, it is interesting to consider concurrent data updates with DILI. Note that an insertion or deletion operation in DILIinvolves only one leaf node. The node adjustment of DILI is much simpler than the rebalance operation of the B+Tree. Theoretically, the lock-free and lock-crabbing [23] approaches can also be applied to DILI, in the same way as how they are applied to the B+Tree.

R2.D2 M4 R4.R6

REFERENCES

[1] online. https://github.com/lpf367135/DILI.
[2] online. https://doi.org/10.7910/DVN/JGVF9A/Y54SI9.
[3] online. https://doi.org/10.7910/DVN/JGVF9A/SVN8PI.
[4] online. https://www.dropbox.com/s/j1d4ufn4fyb4po2/osm_cellids_800M_uint64.zst?dl=1.
[5] online. https://www.dropbox.com/s/y2u3nbanbnbmg7n/books_800M_uint64.zst?dl=1.
[6] online. https://panthema.net/2007/stx-btree.
[7] online. https://github.com/microsoft/ALEX.
[8] online. https://github.com/gvinciguerra/PGM-index.
[9] online. https://github.com/Jiacheng-WU/lipp.
[10] online. https://github.com/learnedsystems/SOSD.
[11] Jayadev Acharya, Ilias Diakonikolas, Jerry Li, and Ludwig Schmidt. 2016. Fast Algorithms for Segmented Regression. In *ICML*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. 2878–2886.
[12] Nikolas Askitis and Justin Zobel. 2009. B-tries for disk-based string management. *VLDB J.* 18, 1 (2009), 157–179.
[13] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proc. VLDB Endow.* 7, 14 (2014), 1881–1892.
[14] Supawit Chockchowwat. 2022. Tuning Hierarchical Learned Indexes on Disk and Beyond. In *SIGMOD*. 2515–2517.
[15] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.
[16] Intel Corporporation. 2018. Intel 64 and ia-32 architectures software developer manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.
[17] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries. In *EDBT*. OpenProceedings.org, 407–410.
[18] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. 969–984.
[19] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *CoRR* abs/2006.13282 (2020). arXiv:2006.13282 https://arxiv.org/abs/2006.13282
[20] 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 (2020), 1162–1175.
[21] Agner Fog. 2018. Lists of Instruction Latencies, Throughputs and Micro-operation Breakdowns for Intel, AMD and VIA CPUs, Technical University of Denmar, Last updated 2021-01-31. http://www.agner.org/optimize/instruction_tables.pdf,DoA. (2018).
[22] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*. 1189–1206.
[23] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Trans. Database Syst.* 35, 3 (2010), 16:1–16:26.
[24] Intel. 2021. Intel Optane Persistent Memory (PMem), Last updated 2021-11-13. https://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-dcpersistent-memory.html.
[25] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.* 30, 2 (2005), 364–397.
[26] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *aiDM@SIGMOD*. 5:1–5:5.
[27] Donald Ervin Knuth. 1997. *The art of computer programming*. Vol. 3. Pearson Education.
[28] 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 *CIDR*.
[29] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). 489–504.
[30] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proc. VLDB Endow.* 15, 2 (2021), 321–334.
[31] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*. 2119–2133.
[32] David B. Lomet. 1981. Digital B-Trees. In *VLDB*. 333–344.
[33] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proc. VLDB Endow.* 15, 3 (2021), 597–610.
[34] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.
[35] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.
[36] Guy M Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. (1966).
[37] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.
[38] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. 2020. Effectively Learning Spatial Indices. *Proc. VLDB Endow.* 13, 11 (2020), 2341–2354.
[39] Esteban G Tabak and Cristina V Turner. 2013. A family of nonparametric density estimation algorithms. *Communications on Pure and Applied Mathematics* 66, 2 (2013), 145–164.
[40] Vladimir Tsymbal. 2019. Tuning Guides and Performance Analysis Papers, Last updated 2020-12-15. https://software.intel.com/content/www/us/en/develop/articles/processor-specific-performance-analysis-papers.html.
[41] Haixin Wang, Xiaoyi Fu, Jianliang Xu, and Hua Lu. 2019. Learned Index for Spatial Queries. In *MDM*. 569–574.
[42] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proc. VLDB Endow.* 15, 11 (2022), 3004–3017.
[43] Jiacheng Wu, Yong Zhang, Shimin Chen, Yu Chen, Jin Wang, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proc. VLDB Endow.* 14, 8 (2021), 1276–1288.
[44] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust Learned Index via Distribution Transformation. *Proc. VLDB Endow.* 15, 10 (2022), 2188–2200.
[45] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. In *SIGMOD*. 1223–1240.
[46] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-based Construction Algorithm. *Proc. VLDB Endow.* 15, 11 (2022), 2679–2691.

A APPENDIX

Due to space limit, we omit some contents in the main body of the paper. In this section, we will show more experimental results and analysis.

A.1 Scalability of Indexes

To further investigate how different methods perform in terms of scalability, we again run the read-only workloads, initializing each index with 50M, 100M, 150M and 200M keys, respectively. As the results on all datasets are similar, we only report the results on FB in Fig. 9 (a). As the number of indexed keys increases, DILI maintains higher throughput than the alternatives. This indicates that DILI is more adaptive to the size of data.

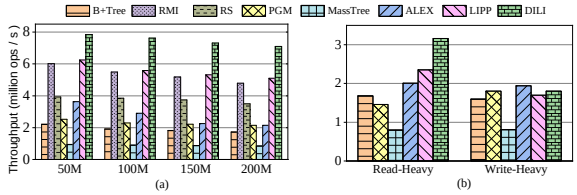


Figure 9: (a) Throughput on FB dataset with varying data cardinalities; (b) Performance with distribution shifting.

A.2 Effect of Distribution Shifting

To investigate if DILI still works well when keys from a different distribution are inserted, we design an experiment as follows. Suppose P_A is another pair set whose keys are in a different distribution from P. First, we build DILI and other indexes supporting inserts using their bulk loading algorithms on the whole P. Then, we repeatedly insert random keys from P_A via all indexes. Meanwhile, search operations are alternatively conducted. We tested a representative combinations of P and P_A from different distributions: FB and Logn. We observe each method's throughput on Read-Heavy and Write-Heavy workloads. Fig. 9 (b) reports that DILI performs a bit worse than ALEX on Write-Heavy workload. The reason behind is that DILI is built to grasp the distribution characteristics of specific dataset. Thus, inserting pairs from different distribution will incur more conflicts and result in adjustments happen more frequently. However, on Read-Heavy workload, DILI still has clear advantages over other alternatives. Also, in real-life scenarios, queries are often more common than insertions. Thus, DILI is supposed to have better performance than other state-of-the-art methods in practice.

A.3 Effect of Distribution Shifting—More Extreme Cases

R2.D3
M2

To investigate the performance of DILI and other alternatives with extremely skewed writes, we design an experiment as follows. Suppose P is the pair set used in the bulk loading stage, the range of $\text{KEYS}(P)$, *i.e.,* the keys of P, is $[A, A + 10\delta)$ and Q is another pair set whose keys are in a different distribution from P. We first make a compression to the keys of $\text{KEYS}(Q)$ by mapping them to the range $[A, A + \delta)$. The mapped keys associated with the original record pointers comprise a new pair set Q'. The sizes of the set P and Q' are both set to 100M. We build DILI, B+Tree, ALEX and LIPP using their bulk loading algorithms on the P. Afterwards, random pairs from the skewed set Q' are repeatedly inserted and then search operations are conducted via all indexes. We test all possible combinations of P and Q' from three different distributions: FB, WikiTS and Logn. We observe each method's throughput on read and write operations. Fig. 10 reports on the comparison results between DILI and other competitors.

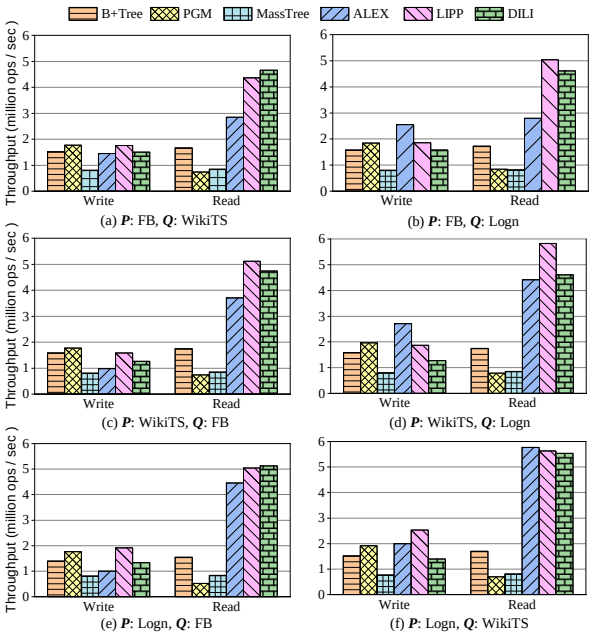


Figure 10: The effects of skewed insertions

Referring to Fig. 10, DILI does not have clear advantages over other competitors as before. It is because that DILI is a distribution-driven learned index which has customized node structure for each dataset. When the distribution of the inserted keys is skewed, more conflicts and node adjustments tend to happen, which result in a higher tree. For example, after inserting 100M skewed keys from the WikiTS dataset to the DILI built with the FB dataset, the average height of the DILI is 4.22. In contrast, if the inserted keys are not out of the initial distribution, the height is 3.86. Nevertheless, DILI still achieves comparable results with LIPP and ALEX.

A.4 Detailed Memory Cost Analysis

R1.R3
M2
M5

Table 10 also shows more direct comparison results of different methods on memory costs.

Referring to Table 10, RMI and RS consume the least memory. Their memory costs are at least 1 order of magnitude smaller than others. However, they do not need to store keys and record pointers in their structures and do not support data updates. LIPP is the most memory-consuming method. Its memory cost is one order of magnitude larger than that of B+Tree, PGM, ALEX and DILI.

DILI consumes more memory than B+Tree, PGM and ALEX due to the local optimization strategy adopted in the leaf nodes. A conflict will result in a new leaf node creation and an empty slot in the entry array. Nevertheless, our design strikes a trade-off between the memory cost and the query efficiency. Although LIPP also adopts a similar strategy to process conflicts, its node

Table 10: The memory cost (10^9 bytes) of different methods

Method	FB	WikiTS	Logn
RMI	0.40	0.40	0.40
RS	0.029	0.063	1.07
B+Tree	1.53	1.53	1.53
MassTree	1.53	1.53	1.53
PGM	3.21	3.20	3.20
ALEX	4.67	4.62	4.61
LIPP	25.61	15.62	28.92
DILI	9.32	7.27	6.48

layout can not be so optimized as ours. Thus, LIPP results in much more conflicts and memory costs than DILI. Considering modern computers and servers usually have huge memory, it is acceptable to improve the efficiency at the expense of some memory. Moreover, it is easy to disable the local optimization strategy, tightly arrange the pairs sorted by their keys in the entry array in DILI's leaf nodes, and make search process follows Algorithm 1. In this case, the memory costs of the modified DILI on the three datasets will become 3.68×10^9 bytes, 3.60×10^9 bytes and 3.27×10^9 bytes, respectively, all comparable with B+Tree. Meanwhile, the query performance will degrade slightly. Also, the range query performance of DILI will get greatly improved. But the efficiency of the insertions will be affected greatly because the pairs are tightly arranged in the entry array and an insertion at position i will usually result in the movement of the pairs whose positions are larger than i, like the B+Tree. At the worse case, we need to move the entire entry array.

A.5 Memory Cost Analysis in Write-heavy Workloads

R2.D5 M2 M5 M7

We also conduct experiments to investigate the effects of insertions on the memory costs of DILI and the alternatives. Similarly, we build all indexes with half of the datasets and use them to carry out 100M insertions. Table 11 shows the memory cost comparisons among all indexes.

Table 11: The memory costs (10^9 bytes) of different indexes in write-heavy workloads

Dataset	Before/after insertions	B+Tree Ω=32	MassTree	PGM	ALEX Γ=16MB	LIPP	DILI
FB	Before	1.56	1.56	1.60	2.35	12.68	5.07
	After	3.12	3.12	3.21	4.59	22.60	8.91
WikiTS	Before	1.56	1.56	1.60	2.31	10.11	3.95
	After	3.12	3.12	3.20	4.31	21.22	5.79
Logn	Before	1.56	1.56	1.60	2.30	14.53	3.38
	After	3.12	3.12	3.20	4.10	18.04	3.67

As shown in Table 11, the memory costs of B+Tree, MassTree and PGM are smaller than that of ALEX, DILI and LIPP. DILI achieves comparable results with ALEX. After the same 100M insertions from the Logn dataset, DILI uses less memory than ALEX. Comparing to DILI, the memory costs of LIPP on all of the three datasets are much larger. This indicates DILI is able to avoid much more conflicts as well as the empty slots in the entry array.

A.6 DILI vs RS

R1.R4 M2

We also conduct experiments to investigate why DILI outperforms RS. Similarly, the search with RS are broken down into two steps. At the first step, RS predicts for the search key a position through a linear spline and a radix table. Then, at the second step, RS conducts a local linear search to find the true position. The average response times of the first step with RS are 264 ns, 212 ns and 101 ns for the three datasets, respectively. Referring to Table 9, those times with DILI are 75, 64 and 43 ns, respectively. Apparently, DILI is much more efficient at the first step. On the other hand, at the second step, RS additionally needs 41 ns, 52 ns, 31 ns on average to carry out the local search for the three dataset, respectively. Thus, RS's query performance is overall worse than DILI.

A.7 Effects of Node Adjustments

R2.D4 M2

To investigate the effects of the adjusting strategy, we conduct experiments using DILI and its variant DILI-AD which does not adopt the adjusting strategies when performing insertions. Following the Write-only workload evaluation process in Section 7.2, both indexes are first built with half of the datasets. Then, for each dataset, we use both indexes to sequentially execute a Write-only workload containing 100M insertions and a Read-only workload containing 100M queries. The comparisons of their response times per insertion, memory costs, average heights, lookup time after insertions are shown in Table 12. Also, this table shows the average number of insertions per adjustment with DILI.

Table 12: The effects of the adjusting strategy

Dataset	Model	Avg # of insertions per adjustment	Insertion time (ns)	Memory cost (10^9 bytes)	Avg height	Lookup time (ns)
FB	DILI-AD	-	**442**	9.31	3.91	187
	DILI	229.1	486	**8.91**	**3.86**	**180**
WikiTS	DILI-AD	-	**349**	5.86	3.58	172
	DILI	758.4	361	**5.78**	**3.55**	**168**
Logn	DILI-AD	-	**304**	3.71	3.11	130
	DILI	1304.6	310	**3.67**	**3.09**	**126**

Apparently, the adjusting strategy in DILI's insertions will result in longer insertion time as it sometimes requires extra operations to collect all pairs covered by a node and train a new linear regression model. However, the adjusting strategy makes DILI avoid more conflicts such that DILI will have a shorter structure and achieves better lookup time and memory costs.

A.8 A Possible Way to Improve DILI's Construction Time

R4.R3 M2

The most time-consuming step in DILI's construction is the greedy merging algorithm for getting the BU nodes at the bottom layer. Because we need to solve a MSE minimization problem for the union of each continuous two pieces, *i.e.* $\mathcal{I}_u^k \bigcup \mathcal{I}_{u+1}^k$ and calculate the MSE to select the next two pieces to merge. Also, after merging $\mathcal{I}_u^k$ and $\mathcal{I}_{u+1}^k$ into a new piece, namely $\tilde{\mathcal{I}}_u^k$, we also need to conduct similar operations for $\mathcal{I}_{u-1}^k \bigcup \tilde{\mathcal{I}}_u^k$ (Algorithm 3).

A direct yet effective approach to make the BU node generation more efficient is sampling. When a piece covers many keys, we could randomly or selectively sample part of the keys, *e.g.*, select one key out of two, to get the linear regression model and calculate the cost. The sampling strategy makes little influence on the whole BU-tree node layout and the performance of the generated DILI. However, it will clearly reduce the construction time of the BU-tree and DILI. An experiment is conducted to show this.

We apply the sampling strategy on the BU-tree's construction as follows. When a piece $\mathcal{I}_u^k$ covers more than 8 keys, we only use half keys get the linear regression models. A DILI is then built based on this BU-tree. As a comparison, we build another DILI without adopting the sampling strategy. According to our statistics, the construction time of the former DILI is 1 minute more less than that of the latter. Their comparison results on the lookup time are shown in Table 13. Apparently, the lookup time of the DILI with sampling is only slightly larger than that of the ordinary DILI.

Table 13: The lookup time of DILI variants

Method	FB	WikiTS	OSM	Books	Logn
DILI-W-Sampling	160	145	121	154	117
DILI	150	139	117	148	116

Considering the construction process of a DILI will be executed once only, a slightly higher construction time is acceptable.

A.9 Concurrent Insertion with DILI

R4.R6

It is possible that DILI support concurrent data updates. It is noteworthy that either an insertion or deletion operation involves only one leaf node. The node adjustment of DILI is much simpler than the rebalance operation of the B+Tree. Theoretically, the lock-free and lock-crabbing approaches can also be applied to DILI, in the same way as how they are applied to the B+Tree. For example, the lock-crabbing protocal can be simply applied to DILI as follows:

(I) Get the lock for the lowest leaf node, namely N_D, covering the pair to be inserted.

(II) If N_D has an empty slot for the pair, put the pair here and release the lock for N_D.

(III) If a conflict happens, the following steps are sequentially executed:

1) If a node adjustment is required, carry out the adjustment and exit. Otherwise, execute the following operations.
2) Create a new leaf node N'_D and put the node at the conflicted slot.
3) Get the lock for N'_D.
4) Release the lock for N_D.
5) Store the conflicting pairs in N'_D.
6) Release the lock for N'_D.