WALDEN: Workload-Aware Learned Tree Index

Abstract. The rapid development of learned index brings a surge in index performance of database management systems. It treats an index as a learned model and manages to learn a regression between keys and their values' positions in the database. Existing works mainly focus on improving the regression accuracy of the model itself. However, we observe that the real-world query workload is biased and keys have different access frequencies, which indicates that keys are not equally important. Simply optimizing the regression accuracy by treating keys equivalently may not be the most effective regarding the workload. We propose the novel Workload-Aware Learned Tree Index (WALDEN) which sets weights to keys and arranges important keys into the shallower levels of the tree index. Specifically, an algorithm which minimizes keys' weighted collision degree is devised to build this workload-aware tree index efficiently. Furthermore, we also design an index update mechanism to monitor and respond to any significant shift of the query workload distribution. Comprehensive evaluations are conducted on synthetic workloads on real datasets w.r.t. common settings, which shows that WALDEN outperforms the state-of-the-art learned indexes with an improvement between 9.7%-500% in terms of the average throughput.

Keywords: Learned Index · AI4DB .

1 Introduction

B+tree is currently widely used in mainstream databases like Oracle[1], MYSQL[2], and PostgreSQL[3] as an index for accessing data. However, with the exponentially growing amount of data, the performance of B+ tree can be insufficient. It has been shown that the B+ tree-based index can consume up to 55% of the total memory when running typical OLTP workloads (TPC-C) in a state-of-the-art in-memory DBMS [46]. Many new models are thus proposed to address this challenge.

Kraska et al. [28] propose a new index concept *Learned Index* which brings in the power of Machine Learning (ML) models to exploit the distribution of data, expecting the index to directly learn the relation between the keys and their values' positions. Given a key and returning a predicted position p, the ML model in learned index is indeed a location mapping function $p = F(key) * N$, where $F(\cdot)$ is its learned cumulative distribution of the keys. Observing a surprising improvement in the data lookup performance from this new index

[1] https://www.oracle.com/
[2] https://www.mysql.com/
[3] https://www.postgresql.org/

structure, a significant amount of follow-up works emerged, with the majority directly focusing on optimizing the location mapping accuracy of the model itself [13, 16, 44].

The PGM index [16] uses an error-bounded piece-wise linear regression method to fit the keys' distribution ($F(\cdot)$) and builds the index in a bottom-up way. ALEX [13] adopts the standard least square algorithm [1] and introduces gaps among the key entries in the tree nodes [19]. LIPP [44] defines the concept of "key collision degree" (identifying the number of keys falling in the same entry inside a node), by minimizing which they reduce the average node depth of keys. NFL [45] uses the Normalizing Flow method [14, 40] to transform the original complex key distribution into a near-uniform one. DILI [31] builds the index in a bottom-up way with linear regression models, then adjusts the tree structure in a top-down manner. SALI [20] proposes a set of node-evolving strategies and a lightweight statistical information maintenance strategy to cope with multi-core conditions.

All the methods above treat all the keys equally important and devise tree structures with regression models to minimize the average access time of a key. However, in practical applications, it is not uncommon that some keys are accessed more frequently than others. For example, Figure 1 shows the check-in location distribution in the dataset Gowalla [12], which records users' check-in locations on a social network application. The horizontal axis lists the latitudes of locations (keys), and the vertical axis represents the check-in frequencies (query workload). We can see that the access frequency of different keys varies greatly, and there are two key clusters ($25 \sim 50$ and $50 \sim 75$) that take apparently higher workloads. In terms of the on-the-fly query performance, keys with higher key frequency are more important than the others and thus should be given more concern.

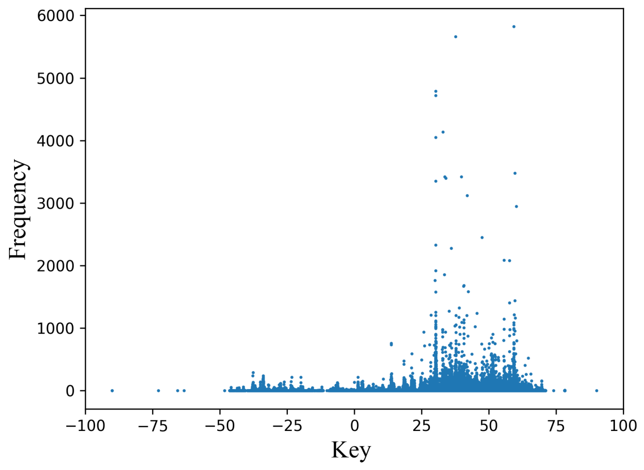


Fig. 1: The check-in workload in Gowalla

The prior knowledge about the query workload can be explored to optimize the index structure by giving higher priority to the more frequently visited keys. For the example presented in Figure 2, there is a workload of 15 queries, among which 1 for keys A, B, C, D, and E and 10 for the key F. Figure 2a shows the workload-blind tree index, where all keys are equivalently placed at the nodes on the second level, and thereby the resultant query cost for the workload is 30. Note that the cost is computed as the length of the traversal path in the tree. In Figure 2b, the key F is given higher priority and moved to one level higher. This tree structure yields a query cost of 22, much smaller than the previous one.

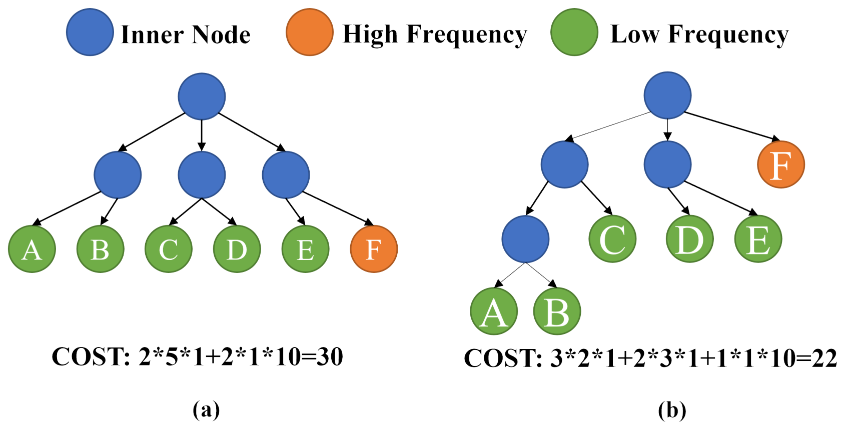


Fig. 2: Why does priority matters

In this work, we devise our method WALDEN, which upgrades the state-of-the-art learned index LIPP [44] by assigning weights to keys and utilizing them to optimize the tree. WALDEN generalizes the collision degree model in LIPP into a weighted one. Subsequently, for each tree node, a regression model is learned by minimizing the weighted collision degree. In this way, we expect that fewer location collisions would happen for the important keys so that they can be placed at nodes of lower depth, costing less query time. In addition, we also propose a mechanism to monitor any significant workload change and perform index regulations if necessary. The experiments show that WALDEN is overall more effective and robust than the state-of-art methods. In particular, for Read-Write Workloads, WALDEN achieves 150%, 25.9%, and 26.1% improvements on the average throughput over NFL [45], ALEX [13], and LIPP [44], respectively, and has a shorter bulk loading time.

The rest of the paper is organized as follows. Section 2 introduces the background of the learned index. Section 3 describes the details and construction of WALDEN. Section 4 shows the index operation of WALDEN. Section 5 presents the experimental results. Section 6 presents the literature review. Section 7 concludes this paper.

2 BACKGROUND

Previous works [20, 43] both acknowledge LIPP's index design and agree on LIPP's state-of-art performance. Regarding the outstanding performance and the sound theoretical basis, we choose LIPP as the basis for proposing WALDEN. We briefly introduce the state-of-the-art model index LIPP in this section, which is composed of segments in a hierarchy structure as shown in Figure 3. A segment comprises an array of nodes $\mathcal{N}$ to store data or pointers to child nodes, a bit vector to record the nodes' content, and a model $\mathcal{M}$ to predict keys' located nodes. One key feature of LIPP is that the model always predicts the keys' exact nodes, which does not require an additional exhaustive search around the predictions as in some existing works [13, 16].

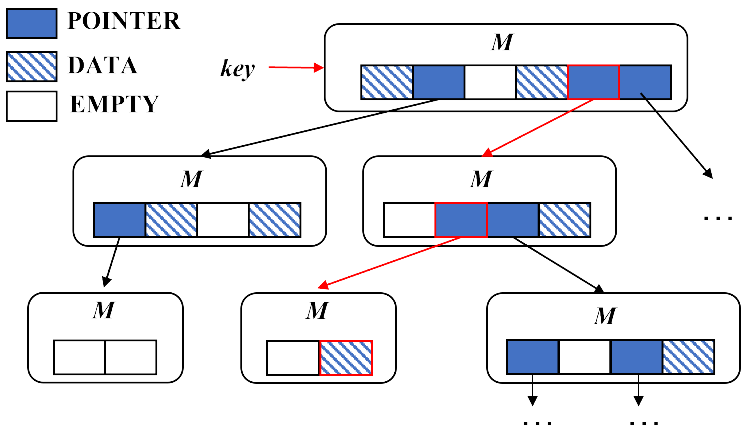


Fig. 3: LIPP model

Given a key as input, the model $\mathcal{M}$ outputs its located node in the segment per level from root to leaf until it eventually reaches a node that exactly stores the key. The model $\mathcal{M}$ is devised to be monotonically increasing, that is, giving two keys k_i, k_j satisfying $k_i \leq k_j$, then $\mathcal{M}(k_i) \leq \mathcal{M}(k_j)$. Given the key k and a node list of length L, model $\mathcal{M}$ computes the key's position at the list according to Equation (1):

$$\begin{cases} \text{if } \lfloor A \cdot \mathcal{F}(k) + b \rfloor < 0\,, & \mathcal{M}(k) = 0 \\ \text{if } \lfloor A \cdot \mathcal{F}(k) + b \rfloor \geq L\,, & \mathcal{M}(k) = L - 1 \\ \qquad \text{otherwise}\ , & \mathcal{M}(k) = \lfloor A \cdot \mathcal{F}(k) + b \rfloor \end{cases} , \tag{1}$$

where $\mathcal{F}$ is a monotonically increasing kernel function; A and b are two learnable parameters. LIPP simply sets the kernel function as $F(k) = k$. Each model $\mathcal{M}$ at a particular segment minimizes its *conflict degree*, as defined below.

Definition 21 ([44],Definition 3.1) *Given a segment having L nodes, a learned model $\mathcal{M}\,(key)$ and a key set $\mathcal{K}$, the conflict degree $T_{\mathcal{M}}$ for this segment is*

$$T_{\mathcal{M}} = max\ number\ of\ keys\ k\ with\ \mathcal{M}(k) == p(p \in [0, L-1]). \tag{2}$$

There are three types of nodes in a segment:

– **EMPTY NODE** : This node stores nothing and is a void space in memory. All nodes are initialized as *EMPTY NODE* and can transform to *DATA NODE* or *POINTER NODE* under certain operations.
– **DATA NODE** : When an empty node writes in data, then it transforms to *DATA NODE*. One *DATA NODE* stores one key and corresponding payload (or pointer to the payload).
– **POINTER NODE** : *POINTER NODE* points to segment in next level. When multiple keys are inserted into the same *DATA NODE*, it will become a *POINTER NODE* and store a pointer to the newly generated segment that holds former conflict keys.

2.1 LIPP Deviation

Regarding the above structure, the main contribution of LIPP is its deviation to compute the parameters A's and b's of model $\mathcal{M}$.

$T_{\mathcal{M}}$ depicts the maximum number of keys that map to the same position. The key idea is to make the model $\mathcal{M}$ evenly map keys into the positions, so LIPP targets minimizing $T_{\mathcal{M}}$, by which they can shrink the search range of the slope and intercept for their linear regression model $\mathcal{M}$. Assuming that the minimum max-conflict-degree viable with $\mathcal{M}$ is T, the model should satisfy Condition (3), which means no more than T elements are mapped to position 0 or $L-1$ in $\mathcal{M}$.

$$\text{“}\begin{cases} A \cdot \mathcal{F}(k_i) + b \geq 1\,, & \exists i \leq T \\ A \cdot \mathcal{F}(k_{N-1-j}) + b < L-1\,, & \exists j \leq T \end{cases}.\text{”} \tag{3}$$

Regarding Condition (3), LIPP further devise Condition (4):

$$\text{“}\ A \leq \max_{i,j} \frac{L-2}{\mathcal{F}(k_{N-1-j}) - \mathcal{F}(k_i)} = \frac{L-2}{\mathcal{F}(k_{N-1-T}) - \mathcal{F}(k_T)}\,,\text{ ”} \tag{4}$$

which gives an upper bound for A.

The above discusses the case that keys are mapped to 0 or $L-1$. Next, they discuss the case that keys are mapped to positions $[1, L-2]$. They start with Lemma 22 to judge whether two keys are mapped to the same position.

Lemma 22 *$\forall key,\ key'$, for model $\mathcal{M}$ with kernel function F, they would not be mapped to the same position as long as $A \geq |(F(key) - F(key'))|^{-1}$.*

With Lemma 22, they deduct Condition (5) which deducts the condition for the slope A, in order to realize the maximum conflict degree not larger than T.

$$\text{“ } A \geq \max_{i \in [0, N-1-T]} \frac{1}{\mathcal{F}(k_{i+T}) - \mathcal{F}(k_i)} \text{ . ”} \tag{5}$$

Finally, based on Condition (4)(5), they conclude the value of T satisfies Condition (6) and b follows Condition (7):

$$\text{“ } \frac{L-2}{\mathcal{F}(k_{N-1-T}) - \mathcal{F}(k_T)} \geq \max_{i \in [0, N-1-T]} \frac{1}{\mathcal{F}(k_{i+T}) - \mathcal{F}(k_i)} \text{ , ”} \tag{6}$$

$$\text{“ } L - 1 - A \cdot \mathcal{F}(k_{N-1-T}) \geq b \geq 1 - A \cdot \mathcal{F}(k_T) \text{ . ”} \tag{7}$$

For the computation simplicity, they transform Condition (6) and get Condition (8) as follows:

$$\text{“ } \forall i \in [0, N-1-T] : \ \mathcal{F}(k_{i+T}) - \mathcal{F}(k_i) \geq U_T \text{ , ”} \tag{8}$$

where " $U_T = \frac{\mathcal{F}(k_{N-1-T}) - \mathcal{F}(k_T)}{L-2}$ " . The conditions devised above are then used to search optimal values for A and b for M per segment in the tree.

3 Our method WALDEN

In this section, based on LIPP, we propose a new design, the workdload-aware learned index (WALDEN) that distinguishes keys by considering their workloads.

3.1 Workload-aware Learned Index.

In practice, we can see that not only the data (keys) follow a certain distribution, but also the requests to data (keys) may have a preference and follow some patterns. By collecting the historical queries of the users, we can get a *Query workload* as in Definition 31.

Definition 31 *A Query workload $\mathcal{Q}$ is the record of the user's request to the database, which contains the query frequency c_i for each key k_i.*

Directly using c_i's to build our index is unwise since it may have sharp values for some popular keys while 0 value for keys that happen not to be in $\mathcal{Q}$. For smoothing purposes, we use a *probabilistic distribution model* $\mathcal{G}$ to fit the normalized $\mathcal{Q}$ as in Definition 32.

Definition 32 *Given the query workload $\mathcal{Q}$, the probabilistic distribution model $\mathcal{G}$ is trained to fit keys' access ratios:*

$$\frac{c_i}{\sum_i c_i} \leftarrow \mathcal{G}(k_i) \, .$$

$\mathcal{G}$ can be any suitable probability distribution model. In our work, we use the Gaussian Mixture Model (GMM) [39] to implement $\mathcal{G}$, which performs well in our experiments. Based on $\mathcal{G}$, we define keys' weights.

Definition 33 (Key weights) *Given the learned query probabilistic distribution model $\mathcal{G}$, the keys' weights are defined as:*

$$w_i = 1 + \lambda \cdot \mathcal{G}(k_i)\,,$$

where λ is an adjustable scaling parameter, and the addition of 1 is for smoothing purposes. This correction can avoid the problem of insufficient training samples leading to $w_i \approx 0$ for some keys.

3.2 Model Parameter Properties

We use the linear model $\mathcal{M}$ to predict keys as well as construct the index tree. Thus, we need to specify the parameters of $\mathcal{M}$, i.e., the slope a and intercept b. We wish to find their values such that the minimum collision degree $\mathcal{C}$ could be realized. A, b and $\mathcal{C}$ are all initially unknown. To find the minimum $\mathcal{C}$, we search from 0 to inf, and for each specific $\mathcal{C}$ value, we find whether there are A and b that can realize this collision degree. If not, we increment $\mathcal{C}$ by 1, easing the difficulty and searching for proper A and b again. Searching A blindly is very time-consuming. Thus, we deduct such upper/lower bounds for A so that we can shrink the search range. Note that the upper/lower bounds are all based on $\mathcal{C}$. With this process, we jointly deduct A, b, and $\mathcal{C}$.

First of all, $\mathcal{C}$ of WALDEN is the "weighted maximum collision degree" among nodes.

Definition 34 *Assuming a segment has L nodes and the learned model $\mathcal{M}$ (key) maps key k_i with weight w_i to position p (from 0 to $L-1$), then the weighted maximum collision degree $\mathcal{C}$ is*

$$\mathcal{C} = \max_{p \in [0, L-1]} \sum_{k_i \in \mathcal{K}} w_i \text{ } \textit{if}\, \mathcal{M}(k_i) == p\,. \tag{9}$$

With this weighted definition version, $\mathcal{C}$ would be high if collisions happen for high-weight keys. By minimizing $\mathcal{C}$, we aim to reduce collisions, especially for those high-weight keys.

To analyze C, we first have the keys sorted in the ascending order. That is, $\forall i$, $key_{i+1} \geq key_i$. $\mathcal{C}$ is determined by the model $\mathcal{M}(key) = A \cdot \mathcal{F}(key) + b$, since $\mathcal{M}$ decides keys' positions. For arbitrary keys in $\mathcal{K}$, model $\mathcal{M}$ maps it to a position p. The weighted collision degree at any position p should be less than or equal to collision degree $\mathcal{C}$ (which is the largest). Note that keys with value $\mathcal{M}(key)$ out of the range of 0 to $L-1$ (border cases) would be truncated to either 0 or $L-1$. Nevertheless, even for this border case, its resultant collision degree at position 0 is also bounded by $\mathcal{C}$. Then we have Condition (10) as below, which is the weighted version of Condition (3) of LIPP.

$$\exists i\,, A \cdot \mathcal{F}(k_i) + b \geq 1 \text{ and } \textstyle\sum_0^{i-1} w_i \leq \mathcal{C}\,. \tag{10}$$

Condition (10) focuses on the collision at the position "0" at a segment (the starting location). First the keys are sorted in ascending order. If we have the first i keys mapped to position “0" whose sum of key weights do not exceed C, then we know that the “$(i+1)$-th " key should be mapped to the position $\geq$ “1", which leads to Condition (10). Similarly, for another border case (position $L-1$), we have Condition (11), which means the last j keys mapped to position $L-1$ should not have accumulated weight more than $\mathcal{C}$.

$$\exists\, j, A \cdot \mathcal{F}(k_{N-1-j}) + b < L-1 \text{ and } \textstyle\sum_{N-j}^{N-1} w_j \leq \mathcal{C}\,. \tag{11}$$

Combining Condition (10)(11), we can easily obtain an upper bound for A as below, which upgrades Condition (4) of LIPP by considering the first and last locations x and y with weights (rather than simply a step length T), for the two aforementioned two border cases.

$$A \leq \frac{L-2}{\mathcal{F}\left(k_{N-1-y}\right) - \mathcal{F}\left(k_x\right)}\,, \tag{12}$$

where $x = \max(i)\ s.t. \sum_0^i w_i \leq \mathcal{C}$ and $y = \max(j)\ s.t. \sum_{N-1-j}^{N-1} w_j \leq \mathcal{C}$. Indeed, Condition (12) states a ceiling for the parameter A in order to prevent keys with accumulated weight more than C from colliding on the position either 0 or $L-1$. It contains constraint information acquired from the head and the tail of the position list. The middle locations of the position list can also deduce a constraint for A. To realize that, we first define $L(i)$ in Definition 35, which is the maximum length starting from key_i that has the accumulated weight no more than $\mathcal{C}$.

Definition 35 *Assume a segment has input keys set $\mathcal{K}$ where key_i have weight w_i and the maximum collision degree is $\mathcal{C}$, then $L(i)$ is the maximum number satisfying:*

$$w_i + w_{i+1} + \ldots + w_{i+L(i)-1} \leq \mathcal{C}\,. \tag{13}$$

Based on Definition 35, for any arbitrary key_i, any key starting from the subscript $i + L(i)$ should not be mapped to the same position as key_i. We then have: for arbitrary key_i, $A \geq |(\mathcal{F}(k_{i+L(i)}) - \mathcal{F}(k_i))|^{-1}$. That is,

$$A \geq \max_{i \in [0, N-1-y]} \frac{1}{\mathcal{F}(k_{i+L(i)}) - \mathcal{F}(k_i)}\,, \tag{14}$$

which is a generalization of Condition (5) of LIPP, for general keys with weights. We can search A by considering the lower and upper bound derived above. Once A is determined, we then can derive from Condition (10)(11) to get bounds for b as below (generalizing LIPP's Condition (7)):

$$L - 1 - A \cdot \mathcal{F}(k_{N-1-y}) \geq b \geq 1 - A \cdot \mathcal{F}(k_x)\,. \tag{15}$$

Combing Condition (12)(14), we can also obtain Condition (16) (generalizing LIPP's Condition (6)) as

$$\frac{L-2}{\mathcal{F}(k_{N-1-y}) - \mathcal{F}(k_x)} \geq \max_{i \in [0, N-1-y]} \frac{1}{\mathcal{F}(k_{i+L(i)}) - \mathcal{F}(k_i)}\,. \tag{16}$$

3.3 Compute Index Structure

Based on the previous discussions, we need to first find $\mathcal{C}$ so as to decide A and b. Following LIPP, we transform Condition (16) into a validation for each k_i, thus we get Condition (17):

$$\forall i \in [0, N-1-y] : \ \mathcal{F}(k_{i+L(i)}) - \mathcal{F}(k_i) \geq U_{\mathcal{C}} \ , \tag{17}$$

where $U_{\mathcal{C}} = \frac{\mathcal{F}(k_{N-1-y})-\mathcal{F}(k_x)}{L-2}$, which is monotonically decreasing as $\mathcal{C}$ increases. We then can find the minimum $\mathcal{C}$ that still holds this inequality. Simply enumerating the value of $\mathcal{C}$ starting from 0 and checking validity for every k_i by Condition (17) can theoretically get the minimum eligible $\mathcal{C}$. However, this naïve algorithm is time-consuming and cost $O(N^2)$ time complexity.

We propose a Weighted Minimum Collision Degree (WMCD) algorithm to solve this problem, which is the generalization of LIPP's model specification method (FMCD), for weighted keys.

As details shown in Algorithm 1, we first set the value of $\mathcal{C}$ as 1 and initialize $x, y, U_{\mathcal{C}}$ (line 1-3). Then we linearly traverse key set $\mathcal{K}$ from the start k_0, if $\mathcal{F}(k_{i+L(i)}) - \mathcal{F}(k_i) \geq U_{\mathcal{C}}$ is satisfied, we move to the next key, update $L(i)$, and repeat the loop (line 6-7). If unsatisfied, we break the verification loop and increase $\mathcal{C}$ (line 12). Correspondingly, we update the values of $x, y, U_{\mathcal{C}}$ (line 13-14) and back to the verification loop starting from the current k_i. The insight is that *keys* (validated before) ahead of k_i do not cause inequality violation for the previous smaller $\mathcal{C}$, then they must be safe for the current $\mathcal{C}$. This escape from repeating starting from $i = 0$ saves much time. If all the keys in range $[0, N-1-y]$ are checked, then the verification is over, and we skip the loop (line 10-11). Finally, we determine A and b according to the found $\mathcal{C}$ (Lines $15 \sim 16$).

Complexity Analysis: By using WMCD(Algorithm 1), we can improve computing efficiency. Here we analyze its time and space complexity. We only need to store temperate parameters like $\mathcal{C}, x, y, U_{\mathcal{C}}$ (line 2-3), then the space complexity is $O(1+1+1+1) \rightarrow O(1)$.

For time complexity, the loop (line 4-14) will be visited a maximum of N times (visited once for each key).

If the condition check succeeds, we need to update the value of $L(i)$ which can be done in constant time. We denote c as the times of computation to update $L(i)$, then the complexity is $O(cN)$. If the condition check fails, the algorithm recomputes the value of $\mathcal{C}, x, y, U_{\mathcal{C}}$. We denote C as the upper bound for the collision degree $\mathcal{C}$, thus, the complexity for update $\mathcal{C}$ is $O(C)$. With a total of N keys, the updates for $x, y, U_{\mathcal{C}}$ cost $O(N)$. In conclusion, the overall time complexity is $O(cN + C + N) \rightarrow O(cN)$.

4 On-the-fly tree adjustment

In this section, we address index degeneration problems which may be caused by either query operations or workload shifts. For the former, query operations like insertion and deletion may change the tree structure and thus undermine

Algorithm 1: WMCD $(\mathcal{K}, L, w)$

```
Input: $\mathcal{K}$: the key set, $L$: the length of nodes list, $w_i$: the key weight set
Output: $C$: the collision degree, $\mathcal{F}$: the model
1  $i = 0; \mathcal{C} = 1; N = |\mathcal{K}|$;
2  $x = max\left\{i|\sum_0^i w_i \le \mathcal{C}\right\}$; $y = max\left\{j|\sum_{N-1-j}^{N-1} w_j \le \mathcal{C}\right\}$;
3  $U_\mathcal{C} = \frac{\mathcal{F}(k_{N-1-y})-\mathcal{F}(k_x)}{L-2}$;
4  while $i \le N-1-y$ do
5      while $i + L(i) < N$ do
6          if $\mathcal{F}(k_{i+L(i)}) - \mathcal{F}(k_i) \ge U_\mathcal{C}$ then
7              $i++$;
8          else
9              break;
10     if $i + y \ge N-1$ then
11         break;
12     $\mathcal{C} = \mathcal{C} + 1$;
13     $x = max\left\{i|\sum_0^i w_i \le \mathcal{C}\right\}$;$y = max\left\{j|\sum_{N-1-j}^{N-1} w_j \le \mathcal{C}\right\}$;
14     $U_\mathcal{C} = \frac{\mathcal{F}(k_{N-1-y})-\mathcal{F}(k_x)}{L-2}$;
15 $\mathcal{M}.A = \frac{1}{U_\mathcal{C}}$;
16 $\mathcal{M}.b = \frac{L-\mathcal{M}.A\cdot\mathcal{F}(k_{N-1-y})-\mathcal{M}.A\cdot\mathcal{F}(k_x)}{2}$;
17 return $\{\mathcal{C}, \mathcal{M}\}$;
```

its performance. We thereby propose real-time tree adjustment strategies to prevent such degeneration (Section 4.2). For the latter, workload shifts change key weights, which can deviate from those used to build the tree. We thereby propose a method to detect and handle it (Section 4.3). Besides, We introduce some basic index operations available in Section 4.1, 4.4.

4.1 Lookup, Range query, and Insert

Lookup, Range query, and Insert operations are conducted in the same way as LIPP [44]. We simply describe them here:

- **Lookup Query:** The lookup operation starts at the root segment of the index and traverses the index tree based on a prediction by the current segment' model. If the current node type is *DATA NODE*, we must check if the key in the current node is consistent with the lookup key instead of returning the current node directly. No further extra search step in *DATA NODE* is needed because we use the in-place insert strategy as LIPP does (all the keys will be placed at the position where the model $\mathcal{M}$ predicts it to be).
- **Range Query:** For example, given range $[a, b]$, we first use the lookup operation to locate the position of key a. Due to the keys being stored monotonically, we can simply scan forward until we find the end key b. We may

traverse through segments and pointer nodes, in other words, index tree to scan for the nodes.

- **Insert:** We first locate the exact node where the insert key points by lookup operation. If the node type is *EMPTY NODE*, then just change the node type into *DATA NODE* and write in the insert key and payload. If the node type is *DATA NODE*, we generate a new segment to store these two keys, then we change the node type into *POINTER NODE* and store the pointer to the new segment. For example, in Figure 4 we insert the new keys '56' and '81'. The predicted positions for keys are marked with colored rectangles. The insertion key will first use the lookup operation. The found position in the first layer is POINTER NODE, then it follows the pointer to the segment in the second layer. It repeats the above operations until it reaches the leaf segment. At last, we can see that the current node type in the leaf segment (third layer) is EMPTY NODE, thus, it changes the node type into DATA NODE and writes in the data.

4.2 Weighted Adjustment Strategy

Successive insertion operations may significantly change the tree structure, resulting in performance degradation. We propose rules on whether a segment should be adjusted, which proceed considering the following parameters: *build_num* is the number of keys when the segment is created, *key_num* is the number of keys of the current segment, *build_weit* is the total weight of keys when the segment is created, *key_weit* is the total weight of keys of the current segment, *collision_num* is the number of conflict keys which means two keys are mapped to the same position, *collision_weit* is the total weight of conflict keys. WALDEN uses rules 1-5 for adjustment.

Next, we first introduce three rules derived from LIPP [44]:

Adjustment Rules

- **Rule 1**: *the number of inserted keys indexed by segment s reaches at least* β *times the number of keys indexed by s in the last adjustment*, i.e. $s.key\_num \geq \beta \cdot s.build\_num$. β is set to 2 by default.
- **Rule 2**: *the ratio between the number of conflicts and insertions in a segment s exceeds a given threshold* α, i.e. $\frac{s.collision\_num}{s.key\_num - s.build\_num} \geq \alpha$. α is set to 0.1 by default.
- **Rule 3**: *segment with too many or too few keys will not trigger adjustment*, i.e. $\gamma \leq s.key\_num \leq \zeta$. γ and ζ are set to 64 and 10^6 by default.

The former three rules target the number of keys during insertions, which means these rules treat all keys equally. However, under the condition that this paper targets, keys, in fact, have priority. We need rules to give more attention to the keys with higher weights, monitoring whether their positions gather too many keys. Thus, we have the following rules to trigger tree adjustment.

Algorithm 2: Adjust($\mathcal{I}, s, w_i$)

```
Input: $\mathcal{I}$: the index, $s$: the segment to adjust, $w_i$: the key weight of $key_i$
1  $s.key\_num = s.key\_num + 1$;
2  $s.key\_weit = s.key\_weit + w_i$;
3  if the insertion conflicts then
4      $s.collision\_num = s.collision\_num + 1$;
5      $s.collision\_weit = s.collision\_weit + w_i$;
6  if $\gamma \leq s.key\_num \leq \zeta$ then
7      if $(s.key\_num \geq \beta \cdot s.build\_num$ and $\frac{s.collision\_num}{s.key\_num - s.build\_num} \geq \alpha)$
8      or $(s.key\_weit \geq \beta \cdot s.build\_weit$ and $\frac{s.collision\_weit}{s.key\_weit - s.build\_weit} \geq \alpha)$ then
9          $\mathcal{K} \leftarrow$ the key set indexed by segment $s$;
10         $s' \leftarrow$ RebuildSubtree($\mathcal{K}$);
11         Replace segment $s$ with $s'$ in $\mathcal{I}$;
```

Rule 4: *the total weight of inserted keys indexed by segment s reaches at least β times the total weights of keys indexed by s in the last adjustment.* i.e. $s.key\_weit \geq \beta \cdot s.build\_weit$. β is set to 3 by default.

Rule 5: *the ratio between the total weight of conflicted keys and the total weight of inserted keys in a segment s exceeds a given threshold α*, i.e. $\frac{s.collision\_weit}{s.key\_weit - s.build\_weit} \geq \alpha$. α is set to 0.1 by default.

Adjustment Algorithms In Algorithm 2, after a key inserts to the index, we first update the statistic of segments in the traversal path (line 1-5). Then for each segment, we check if the rules above are satisfied (line 6-7). If the segment s triggers the adjustment operation, we conduct a sequential traversal to collect the key set $\mathcal{K}$ indexed by segment s (line 8). Regarding $\mathcal{K}$, we build a subtree with a compact structure (line 9). Finally, we change the pointer to a new segment to replace the old index subtree (line 10). Figure 4 shows the adjustment process. The main idea is to replace divergent subtree structures with newly built, compacted, and optimal tree structures. Based on the conclusion we have in Section3.3, the new subtree has minimum collision degree, thus effectively reducing the tree height.

RebuildSubtree Algorithm Receiving $\mathcal{K}$, **RebuildSubtree** first initializes a segment s with L nodes. L is δ (e.g. δ =2) times of the number of keys in $\mathcal{K}$ to create enough *EMPTY NODE* to absorb future insertion. L also has a predefined upper bound (1M) to restrain tree size. Then it computes the key weight for each key prepared for the next step. It uses the WMCD algorithm to compute the model for the new segment, regarding which we put the keys into their positions. If position p has only one key, then it simply inserts relevant data and change node type to *DATA NODE*. Otherwise, if multiple keys are inserted into p, it recursively builds a new subtree until no conflict happens,

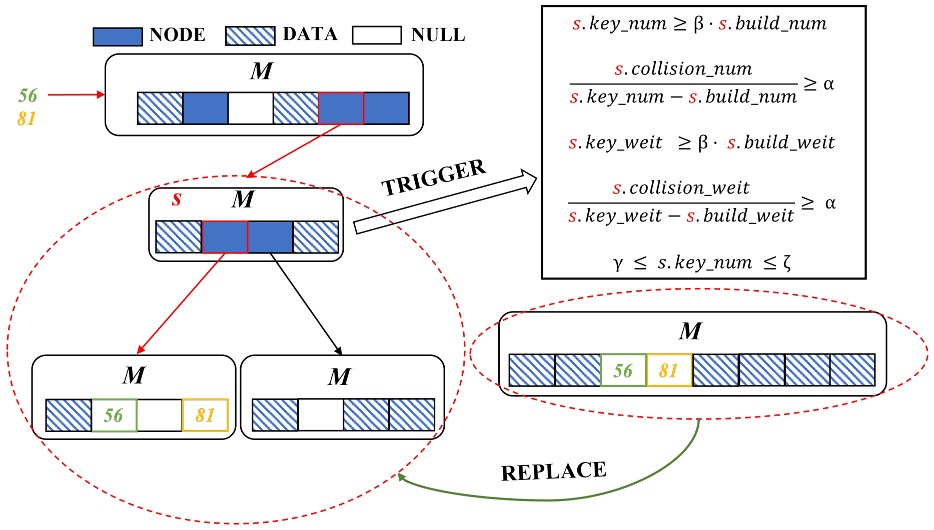

NODE
DATA
NULL
M
56
81
s
TRIGGER
REPLACE
$s.key\_num \geq \beta \cdot s.build\_num$
$\frac{s.collision\_num}{s.key\_num - s.build\_num} \geq \alpha$
$s.key\_weit \geq \beta \cdot s.build\_weit$
$\frac{s.collision\_weit}{s.key\_weit - s.build\_weit} \geq \alpha$
$\gamma \leq s.key\_num \leq \zeta$

Fig. 4: Segment adjustment

then it changes the key type to POINTER NODE and redirect the pointer to the new segment.

4.3 Detect and Handle Workload Change

Our tree model is built according to the recent workloads to prioritize those important keys. However, it is not uncommon that there may be a shift in the workload over the data as time goes by. The shift of the query workload will cause performance degradation of our built index due to its deviation from the current workload. The shifted workload may bring many operations for the keys, which are rarely accessed in the workload for building the tree and are placed at deep levels, leading to long query costs. Let us look back at the example in Figure 2, and consider the tree built in Figure 2b, regarding the training workload where query 1 time for keys “A, B, C, D, E" and query 10 times for key “F", then consider a new workload where keys “B, C, D, E, F" are queried for one time and keys “A" are queried for ten times, which leads to an unbearable cost at 40, even worse than the ordinary one in Figure 2a.

Therefore WALDEN proposes Algorithm 3, which will be called each time after τ times operations have been executed to monitor whether the workload has a significant shift from the old one. In our experiment, we set this number to 10% the size of the dataset. This algorithm would be executed in the background as another thread and thus would not affect the normal operation of the index tree.

Considering a sequence of workload over time shown as the long black box in Figure 5, where the dashed rectangles are our sliding windows. Each window is of the same length $\mathcal{S}$ = 6M, and the windows are obtained via moving with

Algorithm 3: Distribution Shift($\mathcal{G}, \mathcal{K}_0, \mathcal{S}$)

```
Input: $\mathcal{K}_0$: the original key workload, $\mathcal{G}$: the original probabilistic distribution
       model of $\mathcal{K}_0$, $\mathcal{S}$: the size of sliding window
1  for every $\tau$ times of operations do
2      $\mathcal{K}' \leftarrow$key set collected from the new key workload;
3      $\mathcal{G}' \leftarrow$compute probabilistic distribution model for $\mathcal{K}'$;
4      $kl \leftarrow$compute KL divergence between $\mathcal{G}$ and $\mathcal{G}'$;
5      if $kl < 1$ then
6          $shift\_mark = 0$ ;
7      else
8          $\mathcal{G} \leftarrow \theta \cdot \mathcal{G} + (1-\theta) \cdot \mathcal{G}'$;
9          $shift\_mark++$ ;
10     if $shift\_mark \geq \eta$ then
11         $\mathcal{K}_s \leftarrow$ key set collected form the most recent sliding window $\mathcal{S}$;
12         $\mathcal{G} \leftarrow$ retrain probabilistic distribution model for $\mathcal{K}_s$;
13         $shift\_mark = 0$;
14         RebuildSubtree($\mathcal{K}_0 + \mathcal{K}'$);
```

step length. We set step length equal to τ (10% size of the workload, e.g. 20M). Algorithm 3 will be called after every τ times of operations (line 1). $\mathcal{K}_0$ is the original key workload, $\mathcal{G}$ is the original probabilistic distribution model of $\mathcal{K}_0$, $\mathcal{K}'$ is the key set collected from the new key workload and $\mathcal{G}'$ is the probabilistic distribution model for $\mathcal{K}'$. Here, $\mathcal{K}_0$ stands for the key set of the historical workload used to train the model, while $\mathcal{K}'$ represents the key set from the newest window. Then, we judge the shift by comparing $\mathcal{K}'$ and $\mathcal{K}_0$. For $\mathcal{K}'$, we use probabilistic distribution models $\mathcal{G}'$ to fit $\mathcal{K}'$, and use the Kullback-Leibler divergence to judge the deviation (line 2-4). If $kl < 1$, which means the new workload's distribution resembles the historical workload, so the condition of distribution shift does not happen, we set the $shift\_mark$ variable to 0 (lines 5-6). If $kl \geq 1$, the starting distribution G would be updated as the weighted combination of the G and G', i.e., $\theta \cdot \mathcal{G} + (1-\theta) \cdot \mathcal{G}'$ (line 7-9). That is, once the first shift occurs, we also update the G to consider both the start-up workload and the recent one when comparing to the next workload G''.

The tree rebuilding procedure would be triggered only if sustaining shifts happen, i.e., the workload of the successive recent windows are all significantly different from the historical workload. This is modeled by whether the mark $shift\_mark$ is greater than a particular threshold value η (e.g. $\eta = 3$), then the tree rebuilding procedure is called (line 10). First, we use the keys $\mathcal{K}_s$ from the most recent window to learn a new workload distribution $\mathcal{G}_s$ (line 11-12), regarding which the tree is rebuilt via calling the function **RebuildSubtree** (line 14). Recall that RebuildSubtree renews the segment into a larger size, by re-invoking the mapping function WMCD, described in Section 11. Finally, it resets the $shift\_mark$ variable to 0 (line 13).

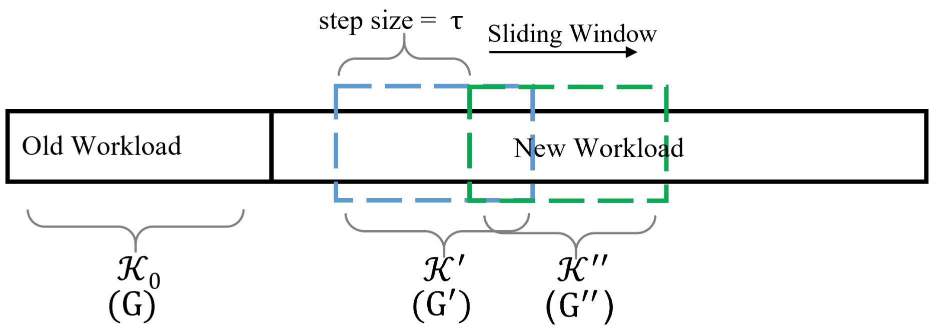


Fig. 5: Distribution shift monitor demonstration

In our implementation, this detection and rebuilding procedure is realized by another thread that runs parallel to the index procedure. Thus, it would not cause any significant delay for the normal operations. Whenever a tree rebuilding operation occurs, a lazy update is realized. That is, during the rebuilding procedure, the old index still runs for new queries. We use a similar approach as in the database cluster master-slave replication scenario. In MySQL, data backup and synchronization are realized in two steps: full replication and incremental replication. Full replication refers to copying all the data of the entire data table. In this step, the system would record the operations occurring during this step. Then, the incremental replication copies only the data that has changed since the last backup, i.e., w.r.t. the recorded operations. WALDEN first builds a new index tree and then synchronizes the operations recorded during the reconstruction phase.

Once the rebuilding procedure is completed, the old index is replaced immediately by the new one.

4.4 Other Operations

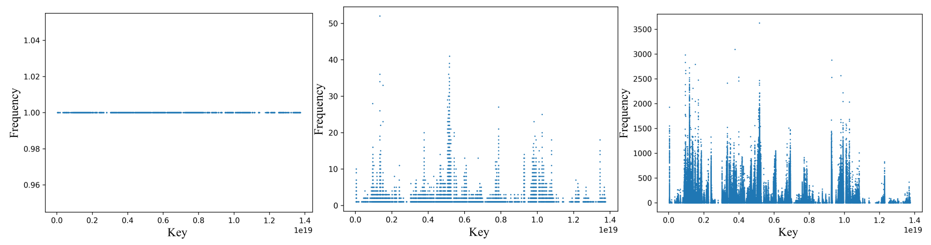
- **Delete:** The deletion can be implemented by locating the indexing segment and changing the corresponding bit map type into *EMPTY NODE*. Then we update the parameter values that monitor the tree index status, such as parameters mentioned in Algorithm 2.
- **Update:** The update operation can be implemented by a lookup operation combined with an in-place payload update or a delete operation combined with an insert operation, depending on whether it updates the key or payload.
- **Bulkload:** The bulk load operation follows the same procedure as the subtree modeling operation in Section 11, which returns the root node as a result.

5 EXPERIMENTAL STUDIES

In this section, we present the experiment settings (datasets, baselines, and parameters) in Section 5.1. In addition, we present the experiment results with the following layout: • The overall results of comparing WALDEN to existing methods in Section 5.2 • The results of dynamic workloads in Section 5.3 • The average/maximum heights of index trees in Section 5.4 • The results of synthetic Zipfian and Normal Distribution Workload in Section 5.5 • More comprehensive experiments results in Section 5.6.

5.1 Experimental Settings

Datasets We evaluate our method with five popular real-world benchmarks widely used in previous works [13, 25, 33], with the data statistics shown in Table 1: (1) LONGLAT (LLT) dataset consists of the longitudes of locations around the world from Open Street Maps [5]. (2) LONGITUDES (LTD) dataset consists of compound keys that combine longitudes and latitudes from Open Street Maps by applying the transformation to these pairs. The resulting distribution of key is highly nonlinear [13]. (3) GOWALLA (GOW) dataset contains location information that users share by checking in a location-based social networking website [12]. (4) OSM dataset is uniformly sampled OpenStreetMap locations [37] represented as Google S2 CellIds [41]. (5) BOOK dataset represents Amazon sales rank data for print and Kindle books [2].



(a) OSM-r0 (b) OSM-r5 (c) OSM-r10

Fig. 6: Frequency of rouned OSM dataset

To generate workloads for these datasets, we do round operations to the data (keys) so that data (key) with the same prefix will be rounded to that prefix number. Then, the prefix number becomes a new key, whose access frequency is the number of original keys having this prefix. For example, 1.67, 1.68, and 1.63 can be rounded to 1.6. If so, the frequency of 1.6 is 3. For each dataset, we create multiple copies with different rounded bits, which leads to different key frequency distributions. These heterogeneous key distributions help to evaluate the methods' robustness. For example, in Figure 6, we show the diverse distributions of the OSM dataset with different number of rounded bits (0, 5 and 10). We eventually get datasets LLT-r0, LLT-r2, GOW-r0, GOW-r2, LTD-r0, LTD-r1, OSM-r0, OSM-r5, OSM-r10, BOOK-r0, BOOK-r5 and BOOK-r10, where the number behind ‘r’ stands for the number of rounded decimal digits. For example, in the LTD-r1 dataset, the number 1.6215445 would be rounded to 1.621545. In the OSM-r10 dataset, the number 9740915555863771079 would be rounded to 9740915560000000000. Note that LLT-r0, GOW-r0, LTD-r0, OSM-r0, and BOOK-r0 are the original datasets without any rounding operation. We also randomly shuffle the keys in these datasets to simulate the workloads, following the setting used by [44]. Augmenting the datasets with rounded bits aims to test the methods'performances over different workload distributions, to test their robustness. Indeed, rounding the tail bits for the keys imitates users' coarse queries. This would be useful when users do not need querying on a precise level. For example, we have keys 55.367 and 55.364, and users care for only the first two digits after the decimal, i.e., his query is 55.xx. Then, rounding the keys to having only two decimal bits as a priori can save much query time.

Table 1: Statistics of datasets

	LLT	GOW	LTD	OSM	BOOK
Num Keys	200M	6.4M	200M	400M	800M
Key Type	double	double	double	unint	unint
Payload Size	8byte	8byte	8byte	8byte	8byte

Baselines and Parameters We compare our WALDEN with existing state-of-the-art base-lines: (1) RMI [28], using a two-level recursive model index with a linear model at each node and an eventual binary search to locate the position [8]. (2) B+Tree [11], an efficient B-Tree implemented by Google. (3) ALEX [13], an in-memory, updatable learned index, which introduces gaps to handle new insertions. (4) LIPP [44], an updatable learned index that creates new nodes to handle conflicted keys to minimize the maximum conflict degree. (5) PGM [16], builds a tree in a bottom-up way, with each node representing an error-bounded piece-wise linear regression model. (6) NFL [45], A two-stage Normalizing-Flow-based Learned index, which transforms the original complex key distribution into a near-uniform distribution. (7) SALI [20], an index that proposes strategies optimized for multi-core conditions of great scalability and efficiency and we use the single-thread version in the following experiments. Note that, we have evaluated various methods proposed in recent years and compared WALDEN to all baselines (as above) which we successfully run with their codes.

These source codes are publicly available, and we run the experiments with their default parameter settings.

Parameter determination. For our method, we set the scale parameter λ to 10, the sliding window length $\mathcal{S}$ to 6M (referring to the number of queries), η and θ in Algorithm 3 to 2 and 0.5. For the adjustment rules in Section 4.2, α and β are set to 0.1 and 3, γ and ζ are set to 64 and 10^6, respectively. They are selected according to the parameter sensitivity results in Figure 7, Figure 8, and Figure 9. Parameters including the scale parameter λ, window size $\mathcal{S}$, the threshold η for judging the distribution shift, α and β for the adjustment rules were investigated, leading to a total of 60 (16*3+12) sets of experiment results reported. For each parameter, WALDEN's performances with its different values were presented, w.r.t. different workload types and datasets are presented. We can see that the influence of parameters is minor and the deviation of the throughput is within 15% under most conditions. The results also demonstrate WALDEN's robustness on its default parameter values.

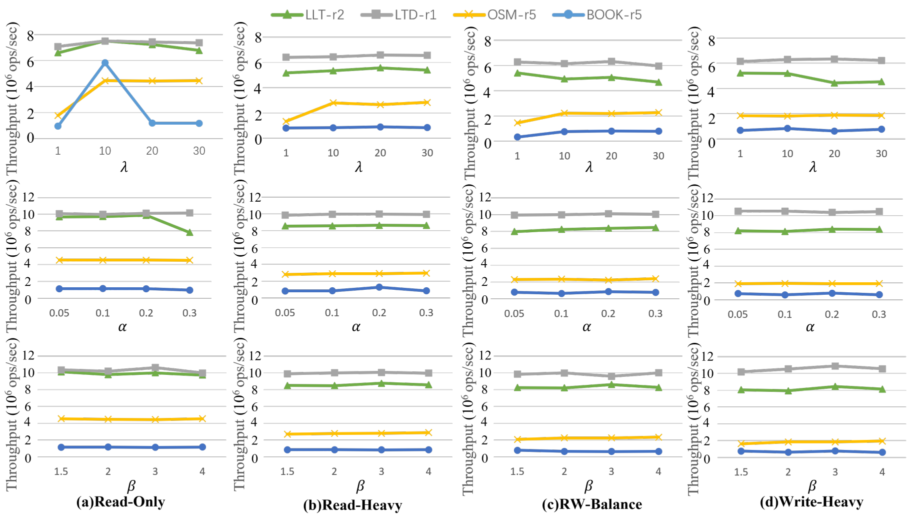


Fig. 7: The effect of λ, α, β

Workloads and Evaluation Metrics We evaluate the methods according to their query throughput. To test their robustness, we evaluate the methods with four workloads following the setting of previous works [13, 44, 45]: (1) The Read-Only workload, which only executes the lookup operation. (2) The Read-Heavy workload, which contains 70% lookup operation and 30% insert operation. (3) The RW-Balance workload, which contains 50% lookup operations and 50% insert operations. (4) The Write-Heavy workload, which contains 30% lookup operations and 70% insert operations.

We use the bulk-loading operation to initialize the index with half randomly selected keys of a dataset following previous works [13, 31, 45]. The lookup keys

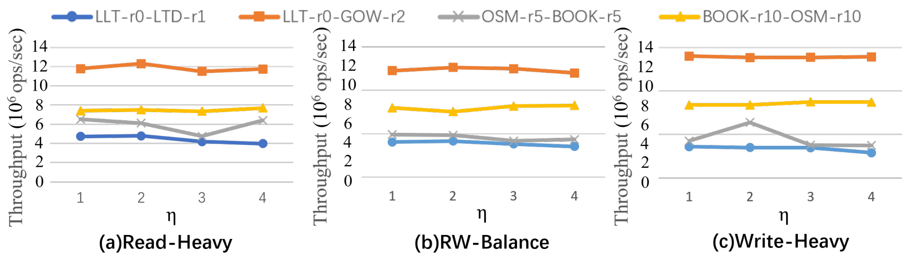


Fig. 8: The effect of η

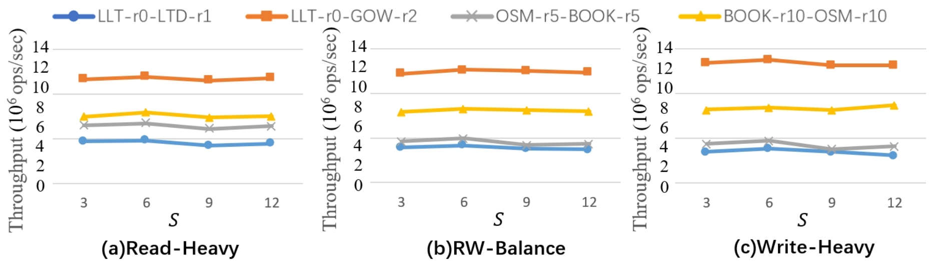


Fig. 9: The effect of $\mathcal{S}$

are generated with weighted random sampling from the whole dataset, where the weight is the key's frequency in the dataset. The insertion keys are randomly selected from datasets, excluding existing ones in the index. We then run the workload above several times to obtain the average throughput of operations and the index size, etc.

We implement WALDEN in C++ and the code is made available[4]. We perform our evaluation on a machine running Ubuntu 20.04 with Intel Xeon Gold 6248R 3.00GHz CPU and 256GB RAM.

5.2 Overall Result

We compare WALDEN with state-of-the-art methods. Table 2 shows the average throughput of WALDEN and other baselines on different workloads. We chose LIPP as a performance reference as it is relatively good among all competitors. The color-encoding indicates how much faster (green) or slower (red) a model is against the reference.

The Read-Only Workload For read-only workloads, WALDEN achieves up to 490%, 260%, 110%, 290%, 18.9%, 408%, 9.7%, improvements on the throughput on average than RMI, PGM, NFL, B+Tree, ALEX, SALI, and LIPP respectively. WALDEN is overall the best and attains the highest throughput on most datasets.

[4] https://github.com/sunpeter2019/WALDEN

Table 2: Lookup and insert throughput

	Methods	LLT-r0	LLT-r2	GOW-r2	GOW-r0	LTD-r0	LTD-r1	OSM-r0	OSM-r5	OSM-r10	BOOK-r0	BOOK-r5	BOOK-r10
Read-Only	RMI	1.3(-80.4%)	1.3(-86.6%)	2(-90.4%)	2(-89.2%)	1.6(-83.2%)	1.6(-84.3%)	1.1(-75.5%)	1.1(-75.2%)	1.1(-85.9%)	2.3(-52.0%)	2.3(-55.5%)	2.3(-67.5%)
	B+Tree	1.9(-71.1%)	2.2(-77.5%)	6.4(-69.3%)	6.3(-66.0%)	2(-78.8%)	2.2(-77.7%)	1.7(-61.2%)	1.7(-61.0%)	2.6(-66.2%)	1.4(-71.1%)	1.5(-71.3%)	1.7(-76.2%)
	ALEX	6.1(-8.6%)	9.2(-6.7%)	14.9(-28.6%)	15.3(-17.8%)	9.7(5.5%)	3.11(32.1%)	2.8(-36.0%)	2.8(-35.6%)	8.1(6.0%)	6(21.9%)	5.9(12.5%)	8.1(12.6%)
	NFL	8.9(32.5%)	10.3(4.4%)	3.1(-84.9%)	3.1(-83.1%)	9(-2.6%)	9.8(-1.3%)	3.2(-27.6%)	3.3(-23.8%)	9.6(26.1%)	4.8(-1.3%)	4.2(-18.6%)	7.7(6.4%)
	PGM	3.2(-51.9%)	3.5(-64.7%)	2.4(-88.4%)	2.1(-88.6%)	3.5(-62.3%)	3.8(-61.4%)	2.5(-44.2%)	2.5(-42.3%)	3.5(-53.6%)	2.2(-55.5%)	2.2(-57.1%)	3.1(-56.7%)
	SALI	1.2(-82.7%)	1.9(-80.5%)	3.3(-84.3%)	3.2(-82.8%)	1.7(-81.6%)	1.8(-81.6%)	1.3(-70.5%)	1.3(-69.8%)	1.9(-75.3%)	0.5(-90.5%)	0.5(-90.7%)	1.8(-75.6%)
	LIPP	6.7(1.0x)	9.8(1.0x)	20.9(1.0x)	18.6(1.0x)	9.2(1.0x)	9.9(1.0x)	4.4(1.0x)	4.4(1.0x)	7.6(1.0x)	4.9(1.0x)	5.2(1.0x)	7.2(1.0x)
	WALDEN	6.1(-8.8%)	10.1(2.8%)	21.1(0.8%)	19.9(6.9%)	9.9(7.2%)	10.6(6.5%)	4.7(6.1%)	4.6(5.6%)	8.6(12.9%)	6.1(24.2%)	6.1(17.5%)	9.7(34.3%)
Read-Heavy	B+Tree	1.6(-59.6%)	2.4(-72.2%)	8.3(-61.9%)	7.3(-62.1%)	1.6(-67.1%)	2.2(-74.8%)	1.4(-51.4%)	1.4(-51.3%)	3.4(-66.5%)	1.1(-68.3%)	1.1(-51.7%)	1.9(-78.2%)
	ALEX	3.8(-2.4%)	7.9(-8.6%)	16.9(-22.9%)	18.3(-5.5%)	6.1(26.8%)	10.8(24.5%)	2(-32.3%)	2(-33.1%)	10(-0.5%)	3.2(-5.5%)	3.1(40.5%)	8.1(-6.0%)
	NFL	4.6(17.3%)	7.9(-9.0%)	3.1(-86.0%)	2.8(-85.3%)	4.7(-2.6%)	7.8(-10.5%)	2.6(-12.0%)	2.7(-7.6%)	10.2(1.3%)	3.1(-8.2%)	3(35.6%)	7.4(-13.8%)
	PGM	0.9(-77.8%)	1.1(-87.2%)	1.8(-91.9%)	1.6(-91.6%)	0.9(-81.4%)	1.1(-87.2%)	0.7(-75.0%)	0.8(-72.9%)	1.3(-87.4%)	0.7(-78.3%)	0.7(-66.4%)	0.9(-89.0%)
	SALI	1(-74.7%)	2.3(-73.6%)	2.2(-90.2%)	4.1(-78.8%)	1.6(-67.2%)	2.1(-75.8%)	1.1(-61.4%)	1.1(-61.5%)	2.6(-74.4%)	0.4(-88.2%)	0.4(-81.7%)	2(-76.1%)
	LIPP	3.9(1.0x)	8.6(1.0x)	21.9(1.0x)	19.4(1.0x)	4.8(1.0x)	8.7(1.0x)	2.9(1.0x)	2.9(1.0x)	10(1.0x)	3.4(1.0x)	2.2(1.0x)	8.6(1.0x)
	WALDEN	3.9(-1.4%)	9.1(5.6%)	23.3(6.4%)	21.7(11.9%)	5.7(17.9%)	10.2(17.8%)	3(2.0%)	3(1.1%)	11.2(11.4%)	3.9(15.7%)	4(77.6%)	10.6(24.0%)
Write-Heavy	B+Tree	1.4(-46.3%)	3.2(-60.8%)	11.2(-60.4%)	10.8(-58.1%)	1.4(-55.8%)	3(-64.7%)	1.3(-36.2%)	1.3(-37.2%)	6.6(-61.5%)	1(-4.3%)	1(1.9%)	2.6(-78.2%)
	ALEX	2.6(1.7%)	7.6(-8.4%)	21.2(-25.0%)	20.6(-20.2%)	3.7(15.9%)	10.2(22.1%)	1.4(-30.3%)	1.5(-27.8%)	16.7(-2.1%)	1.9(92.5%)	1.9(102.5%)	7.7(-34.6%)
	NFL	3(19.2%)	7.3(-11.1%)	2.8(-90.0%)	2.8(-89.0%)	3.1(-2.2%)	7(-15.9%)	2(0.2%)	2.1(2.2%)	13.3(-22.0%)	2.1(110.3%)	2.1(122.5%)	7.3(-37.5%)
	PGM	1.5(-43.0%)	2.3(-72.4%)	3.5(-87.6%)	3.4(-87.0%)	1.6(-50.7%)	2.1(-74.4%)	1.3(-34.9%)	1.4(-30.0%)	2.7(-83.9%)	1.3(31.3%)	1.3(41.4%)	1.9(-83.8%)
	SALI	1.3(-48.0%)	3.1(-62.2%)	6.9(-75.5%)	6.4(-75.1%)	1.4(-57.1%)	2.8(-66.3%)	0.6(-70.9%)	0.6(-71.7%)	5(-70.9%)	0.4(-61.0x)	0.1(-86.0%)	3(-74.6%)
	LIPP	2.6(1.0x)	8.3(1.0x)	28.3(1.0x)	25.8(1.0x)	3.2(1.0x)	8.4(1.0x)	2(1.0x)	2(1.0x)	17(1.0x)	1(1.0x)	1(1.0x)	11.7(1.0x)
	WALDEN	2.6(0.5%)	8.8(7.1%)	31.2(10.3%)	29.7(15.3%)	3.5(10.3%)	10.9(29.7%)	2(-3.2%)	2(-0.5%)	19.4(14.2%)	2.7(168.4%)	2.6(176.9%)	12.8(9.1%)
RW-Balance	B+Tree	1.5(-53.1%)	2.7(-67.1%)	9.1(-62.1%)	8.9(-59.9%)	1.5(-61.3%)	2.5(-70.5%)	1.3(-44.2%)	1.3(-44.1%)	4.3(-65.0%)	1(-61.9%)	1(-5.1%)	2.2(-78.3%)
	ALEX	3(-3.8%)	7.7(-6.7%)	19.5(-18.6%)	18.5(-16.9%)	4.4(16.6%)	10.4(25.0%)	1.7(-29.7%)	1.7(-29.1%)	12.5(0.6%)	1.8(-31.1%)	2.3(117.8%)	8(-20.2%)
	NFL	3.5(12.4%)	7.6(-8.1%)	2.9(-87.7%)	2.9(-86.9%)	3.6(-3.8%)	7.3(-12.9%)	2.3(-4.9%)	2.4(0.2%)	11.4(-8.1%)	2.6(-2.6%)	2.6(145.9%)	7.3(-26.7%)
	PGM	1.2(-62.1%)	1.3(-83.8%)	2.2(-91.0%)	2.1(-90.5%)	1.2(-67.2%)	1.5(-81.8%)	1(-59.3%)	0.9(-62.8%)	1.6(-87.1%)	1(-61.1%)	1(-1.6%)	1.4(-86.3%)
	SALI	1(-68.7%)	2.7(-67.4%)	5.1(-78.5%)	5.1(-77.0%)	1.5(-60.5%)	2.4(-71.1%)	1.1(-56.3%)	0.6(-74.0%)	3.4(-73.0%)	0.4(-84.4%)	0.4(-59.7%)	2.4(-75.6%)
	LIPP	3.2(1.0x)	8.3(1.0x)	23.9(1.0x)	22.2(1.0x)	3.8(1.0x)	8.3(1.0x)	2.4(1.0x)	2.4(1.0x)	12.4(1.0x)	2.6(1.0x)	1.1(1.0x)	10(1.0x)
	WALDEN	3.2(0.1%)	8.9(7.6%)	25.5(6.6%)	24.8(11.7%)	4.3(13.3%)	10.3(23.0%)	2.4(0.0%)	2.4(-1.0%)	14.2(13.8%)	3.2(21.3%)	3.2(201.1%)	11.5(14.9%)

More specifically, compared with LIPP, WALDEN put keys with higher weights into a shallower layer of the index tree so that they can be found with a shorter traversal path. This way, the average traversal path of WALDEN is shorter than LIPP, therefore achieving higher throughput. Compared with ALEX, PGM, and RMI, WALDEN inherits the merit of LIPP, which is free of an exhaustive search step in leaf nodes. Thus, it saves a large time overhead on the leaf nodes. We can see that ALEX's performance is particularly good on LTD and BOOK datasets, which may be because its index design well fits these two datasets. Nevertheless, its throughput suffers decay on other datasets, where WALDEN still has a clear advantage. NFL transforms the original keys into another near-uniform distributed key space to enable a better fitting for the index. However, we can see that its performance is not stable, without even mentioning its expensive overhead training process even under the speedup of GPU equipment. Even within the several rounded dataset instances of BOOK, its performance is diverse.

Meanwhile, we can see that the performance of WALDEN on the LLT-r0 dataset is defeated by the NFL, and worse than the baseline LIPP. After some investigations, we found that this is caused by the weak performance of the Gaussian mixture model we used to fit the key distribution on LLT. LLT is a highly non-uniform dataset [45], which brings more difficulty for distribution fitting. What's more, the LLT-r0 dataset does not contain duplicate elements, thus keys are not distinguished via their frequencies. This holds back WALDEN's advantage. Nevertheless, when it comes to LLT-r2 with key frequency, the performance of WALDEN becomes comparable with NFL or even better on read-write workloads.

One interesting observation is that the results of WALDEN differ from LIPP on the “r0" datasets where the keys are unique (i.e., frequency is 1), though WALDEN is a generalization of LIPP. This is because even on “r0" datasets, the unique keys are not uniformly distributed. Instead of directly using the key frequencies, WALDEN uses GMM to fit the keys first, which still can make different key weights as long as the keys are non-uniformly distributed.

The Read-Write Workload For read-write workloads (Read-Heavy, Write-Heavy, and RW-Balance), WALDEN is overall most competitive, achieving top throughputs on most datesets. On overage, shown by Table 2, WALDEN achieves up to 500%, 150%, 210%, 25.9%, 287%, 26.1% improvements on the throughput than PGM, NFL, B+Tree, ALEX, SALI, and LIPP, respectively. Note that we exclude RMI from the comparisons here because it does not support write operation. Compared with the read-only workload, WALDEN suffers some decay in throughput. The performance degradation is caused by the insertion operations, which are more costly than the lookup operation. After locating the key position, WALDEN needs to write in the new key-weight pair. If there’s collision happens, it needs to generate a new node to accommodate the new data. Besides, the adjustment process may be triggered, which also takes time. For the results of GOW-r0 and GOW-r2, WALDEN’s results on the former are better than the latter. This is because the GOW dataset is special compared with others, its raw dataset already has frequency for keys, making non-uniform workloads. GOW-r0 already suits WALDEN and leads to satisfactory performance.

Compared with LIPP, the node adjustment process is further optimized in WALDEN (see Section 4.2), thus achieving even higher throughput improvement on read-write workloads. PGM’s insertion process is motivated by LSM-tree [35, 36], which is expensive for insertion operations. It needs $O(logN)$ trees to support insertions, and the lookup operation needs to scan all the trees. NFL’s bucket design may degrade its performance when the buffer space is full. The linear search on the bucket can cause large overheads, and the NFL has to adjust its index tree structure. Compared with ALEX, WALDEN can avoid the element shifting in the node and has a simpler way to adjust the nodes. Although ALEX performs particularly well on the LTD-r0, as mentioned previously, WALDEN still reaches comparable throughput at most times on LTD-r0. WALDEN achieves up to 408% and 287% improvements in the throughput than SALI for read-only and read-write workloads. We can see that WALDEN is comparatively better in this case. Nevertheless, SALI is designed and optimized for multi-thread conditions (64 threads or more) thus would be better when multiple threads become available.

5.3 Workload Shift

In this section, we test WALDEN’s performance under workload-shifting conditions. First, we test the throughput with dynamic workloads. Then we do an ablation experiment to test the effectiveness of our method. Finally, we explore WALDEN’s performance with different workload-changing frequencies.

Dynamic Workload In this experiment, to simulate a dynamic workload on the query keys, we randomly select 100M keys from two different datasets (of the same data type, both Double or both Unint) and get $\mathbf{P}_1$, $\mathbf{P}_2$. We initialize the evaluated methods with $\mathbf{P}_1$ by bulk loading operation, then we use $\mathbf{P}_2$ to generate four types of workload with different lookup-insert ratios. The lookup keys are sampled weightedly from $\mathbf{P}_2$, and the insertion keys are randomly selected from those in $\mathbf{P}_2$ but do not exist in $\mathbf{P}_1$. We then run the workload 10 times to obtain the average throughput of operations. We compute the KL-divergence, Pearson correlation of $\mathbf{P}_1$ and $\mathbf{P}_2$, while yields scores of over 4.08 and under 0.23. This suggests that they share significantly different patterns. As RMI does not support insertion operation, RMI is excluded from the experiments here. Table 3 shows the experimental results, where the first column indicates the two datasets where $\mathbf{P}_1$, $\mathbf{P}_2$ are sampled, and the workload type (Read-Heavy, etc.). Note that the method WALDENnoD refers to WALDEN without the mechanism for detecting and handling shifts, which will be analyzed in the ablation experiment in the next section.

Overall, WALDEN achieves persistently competitive throughput over different settings. Even under the workload shift condition, WALDEN can still achieve 59.3%, 13.1%, 55.5%, 78.9%, and 21.8% throughput improvements over B+Tree, ALEX, NFL, PGM, and LIPP, respectively. This demonstrates our proposed methods for detecting and handling workload change. It has a mechanism to monitor and handle the distribution shift of workload. If it happens, WALDEN would update the distribution model $\mathcal{G}$ accordingly and rebuild the index if necessary.

We can see that ALEX performs well on certain datasets (LLT), which is consistent with previous results in Table 2. We can also see that the performance of the NFL is not as good as it was with previous results in Table 2, which is in expectation. NFL devises a key transformation process that also relies on the knowledge of key distribution. However, it does not have a mechanism to address a shift as WALDEN does.

Ablation Experiment WALDENnoD does not apply the workload shift detection and handling mechanism in Section 4.3, which is treated as a comparison method. WALDENnoD has a strong assumption on the workload; once the distribution has deviated from the original distribution, the throughput of WALDENnoD can be severely affected. As shown in Table 3, the throughput of WALDENnoD is far behind WALDEN when the shift occurs. This demonstrates the effectiveness of the proposed detection mechanism.

Workload Changing Frequency In this section, we add an experiment to show WALDEN's performance when the workload changes with different frequencies. In particular, we change the query workload after 10M, 30M, and 50M operations on the old workload. With a total of 200M operations, the operations tested for the new workload for the above three cases are 190 M, 170 M, and 150 M respectively. We can see from Figure 10 that WALDEN WALDEN achieves an

Table 3: Dynamic workload composed of different patterns

Read-Heavy	B+Tree	ALEX	NFL	PGM	LIPP	WALDENnoD	WALDEN
OSM-r5-BOOK-r5	1.8(-62.6%)	3.7(-22.4%)	2.2(-54.6%)	0.9(-81.6%)	4.8(1.0x)	4.9(1.8%)	6.3(32.2%)
LLT-r0-LTD-r1	2.1(-48.4%)	7.5(84.5%)	4.5(11.0%)	0.8(-79.3%)	4.1(1.0x)	3.6(-11.3%)	4.6(13.6%)
LLT-r0-GOW-r2	6.5(-10.6%)	11.6(59.9%)	1.1(-84.2%)	1.7(-77.0%)	7.3(1.0x)	7.9(8.7%)	12.5(72.6%)
OSM-r5-BOOK-r10	1.9(-64.3%)	4(-24.6%)	1.9(-63.6%)	1(-80.7%)	5.3(1.0x)	6(13.8%)	9.8(83.9%)
BOOK-r10-OSM-r10	2.8(-55.7%)	6.4(0.4%)	4.1(-34.8%)	1.1(-82.1%)	6.3(1.0x)	4.8(-24.2%)	7.4(16.8%)
Write-Heavy	**B+Tree**	**ALEX**	**NFL**	**PGM**	**LIPP**	**WALDENnoD**	**WALDEN**
OSM-r5-BOOK-r5	1.4(-61.3%)	2.4(-34.7%)	1.7(-53.0%)	1.4(-61.5%)	3.7(1.0x)	2.9(-21.3%)	3.9(4.4%)
LLT-r0-LTD-r1	2.1(-38.2%)	5.4(60.3%)	4(19.4%)	1.7(-49.6%)	3.4(1.0x)	2.7(-20.6%)	3.8(12.9%)
LLT-r0-GOW-r2	9.3(-12.4%)	14.1(32.1%)	1.1(-89.7%)	3.1(-71.1%)	10.7(1.0x)	9.8(-8.4%)	14.1(32.0%)
OSM-r5-BOOK-r10	1.9(-67.1%)	3.2(-43.5%)	1.7(-69.1%)	1.6(-71.2%)	5.6(1.0x)	4.8(-14.9%)	7.6(34.4%)
BOOK-r10-OSM-r10	3.9(-51.0%)	6.6(-15.6%)	4.8(-39.1%)	2.1(-73.5%)	7.9(1.0x)	5.6(-28.5%)	8.5(7.9%)
RW-Balance	**B+Tree**	**ALEX**	**NFL**	**PGM**	**LIPP**	**WALDENnoD**	**WALDEN**
OSM-r5-BOOK-r5	1.6(-61.0%)	3(-26.3%)	1.9(-53.5%)	1.1(-71.9%)	4(1.0x)	3.3(-18.9%)	4.9(21.7%)
LLT-r0-LTD-r1	2(-43.4%)	6.5(83.1%)	4(14.0%)	1.1(-68.5%)	3.5(1.0x)	3.6(0.9%)	4.5(26.1%)
LLT-r0-GOW-r2	7.2(-14.5%)	12.7(50.0%)	1.2(-86.4%)	2.3(-73.2%)	8.4(1.0x)	8.3(-1.8%)	13.1(55.1%)
OSM-r5-BOOK-r10	1.9(-65.8%)	3.6(-34.8%)	1.8(-67.9%)	1.2(-77.6%)	5.5(1.0x)	5(-8.0%)	8.1(47.3%)
BOOK-r10-OSM-r10	3.3(-53.6%)	6.9(-3.0%)	4.4(-38.3%)	1.4(-79.9%)	7.1(1.0x)	5.2(-26.7%)	7.9(10.8%)

average of 36.4%, 17.7%, and 24.6% improvements on the throughput with Read-heavy,write-heavy and rw-balance conditions. The frequently changing workload can obviously affect WALDEN, LIPP, and ALEX which share a similar model design.

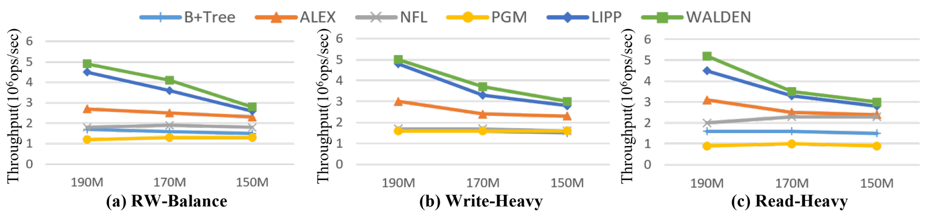


Fig. 10: Workload changing frequency

5.4 Average and Maximum tree height

Table 4 shows the average/maximum heights of index trees after the entire work-loads from different datasets, respectively. ALEX has a smaller maximum height because it avoids increasing nodes when dealing with collide keys, but it will cost extra time to correct and find the real position in the leaf node. NFL uses a bucket structure that stores collide keys in one node to avoid the growth of tree height. Also, the NFL needs extra time to search within the bucket. For our method WALDEN, it has a larger maximum tree, but its average tree height remains small, remaining 0.94 $\sim$ 1.09 times as large as that of LIPP. For the maximum tree height, WALEND is 1.17 $\sim$ 2.67 times that of LIPP. This means that in the worst case, the time of accessing a key in WALDEN is bounded by $1.17 \sim 2.67$ times that for LIPP.

Table 4: Avg/Max height of indexes

	ALEX	NFL	LIPP	WALDEN
LLT-r2	2.31/4	1.72/6	2.02/6	2.00/7
GOW-r0	2.14/3	1.74/4	1.86/5	1.77/6
LTD-r1	2.04/3	1.91/2	1.85/3	1.74/8
OSM-r5	2.81/6	3.10/8	2.62/8	2.87/11
BOOK-r5	2.33/4	2.12/8	2.03/7	1.91/12

5.5 Zipfian and Normal Distribution Workload

To show the ability of WALDEN under various workload distributions, we add two widely accepted query distributions Zipfian distribution and Normal distribution [13, 25, 33]. We can see from Figure 11 that WALDEN still maintains its advantages in lookup under the Zipfian distribution. WALDEN achieves an average 45.1% improvement in the throughput than LIPP. ALEX performs better under Read-Only and Read-Heavy workloads. The reason is that ALEX's design of RMI structure realizes a good fitting of the data. With more insertion operations (Write-Heavy, and RW-Balance), ALEX's node expansion and split mechanism may be frequently triggered and subsequently lose its advantage. In contrast, WALDEN is more robust and behaves better in the majority of settings.

As shown in Figure 12, we can see that WALDEN still achieves the highest throughput on workloads of Normal distribution, which is consistent with the results in Table 2.

5.6 More Results

In this section, we show some supplementary results. (1) We test WALDEN's performance with bulk Load operation. (2) We test the throughput with range query operation. (3) We analyze the index size between different methods. (4) We Analyze the index performance with a hotkey workload. (6) We explore the effect of different fitting models.

Bulk Load In Figure 13, we present the time cost of loading half the size of datasets at once to build the index. WALDEN only takes less than half of the time to bulk load compared with ALEX, RMI, and NFL, and WALDEN

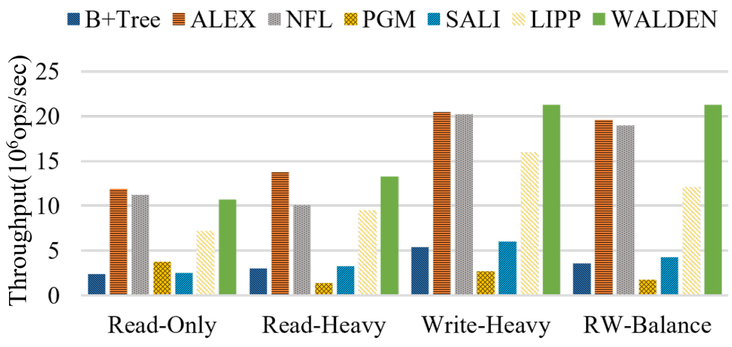


Fig. 11: Zipfian workload

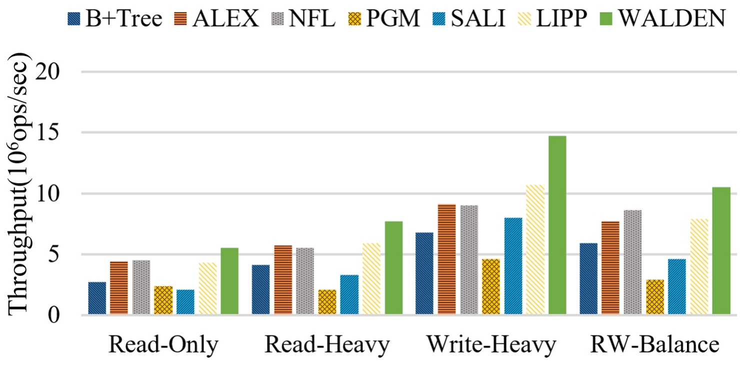


Fig. 12: Normal workload

is faster than LIPP due to its optimized construction and adjustment algorithm. The bulk-loading time cost of WALDEN is larger than that of PGM and B+Tree. However, the operation cost of WALDEN is much smaller than PGM and B+Tree, as demonstrated in Table 2.

The high efficiency of the bulk load operation of WALDEN also demonstrates the liability of our mechanism in dealing with workload shifting. WALDEN will rebuild itself to cope with the new distribution of data, and its efficient bulk load process can make sure the reconstruction of the index will not lead to too much overhead.

Range Query We follow the experiment setting for range queries of previous works [13, 44]. We restrict the number of scanned keys varied from 50 to 10000 for range queries, as set up by [13]. Figure 14 shows that WALDEN achieves higher throughput. Results on other datasets share a similar tendency, we omit them here.

The mean performance improvement (over the datasets) of WALDEN drops gradually from 8.1% to 2.1% when the scan length is increased from 50 to 10000.

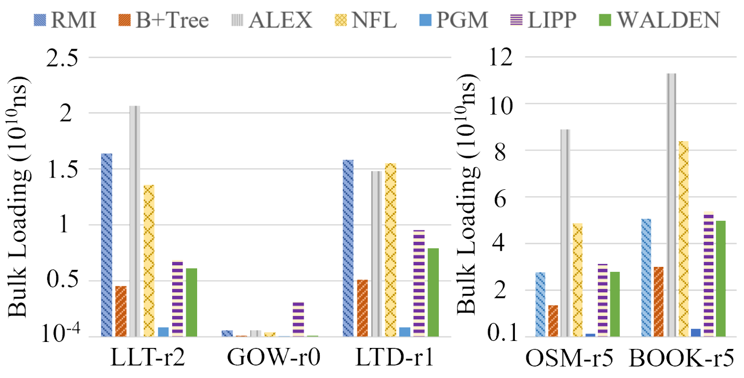


Fig. 13: Bulk loading time

The advantage of WALDEN on lookups becomes less remarkable due to the scanning time begins to dominate in the overall query time cost, which grows with the scan length. But still, WALDEN maintains competitive performance.

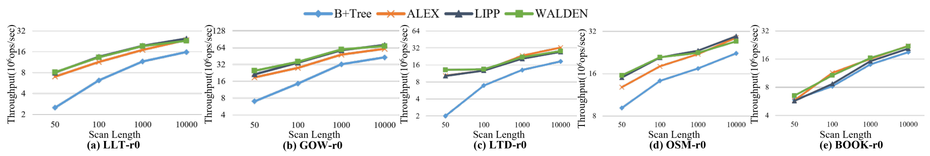


Fig. 14: Range query

Index Size Figure 15 shows the index size after the read-heavy and write-heavy workloads (the results on other workloads are similar and thus omitted). We evaluate the overall index size, including the early allocated empty entries inside the tree. The index size distinguishes the method into two groups, one with B+Tree, ALEX & PGM, and another one with WALDEN, LIPP & NFL which have larger sizes. WALDEN and LIPP are more costly as they maintain more auxiliary information for the tree. WALDEN even costs slightly more space since it maintains keys' weights. Nevertheless, its space consumption is still acceptable and can be easily admitted by practical servers. Note that all of our experiments are run on a machine with 256GB RAM.

Hotkey Workload We generate a new workload in which the lookup keys only include the top 1000 high-access frequency keys. Results are shown in Figure 16. This specially generated workload does fit our method WALDEN and thus achieves great throughput gains.

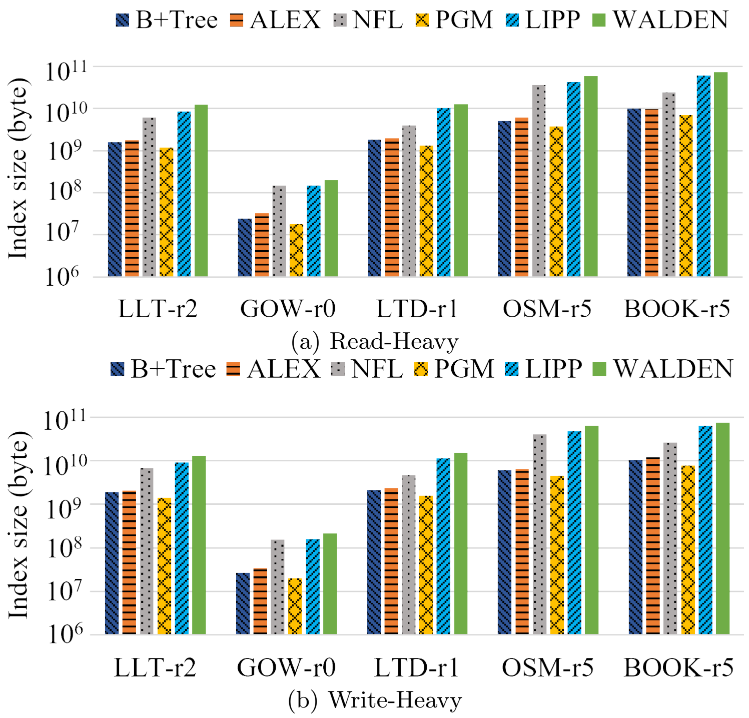


Fig. 15: Index size

Fitting Models In this section, we investigate the effect of implementing $\mathcal{G}$ with different models. We choose three classic probability distribution models Gaussian Mixture Model (GMM), Normal distribution (Norm), and Polynomial distribution (Polynomial), with the experiment results as Figure 17, showing that GMM is the most effective.

6 Related work

B-tree Variants: B+-trees [7] is the most popular B-tree variant designed for disk-based systems. Balanced/red-black trees [6,10] and T-trees [29] are designed for in-memory systems. CSB+-tree [38] is proposed to address the poor cache behavior of B+-tree. Other methods like FAST [24] or HB+-tree [23, 42] take advantage of SIMD instructions or GPUs to improve performance on the cache. MassTree [32] inspired by trie [30] uses the concatenation of B-trees to improve cache awareness. Tries/radix-trees, B-trie or Kiss-tree [3,9,17,27] which combine ideas from B-Trees and tries [30] are designed for indexing text. There are many variant targets to reduce memory cost using techniques like prefix/suffix truncation, dictionary compression, key normalization [21, 22, 34], or hybrid hot/cold indexes [46]. A BF-tree [4] uses bloom filters to substantially reduce the index

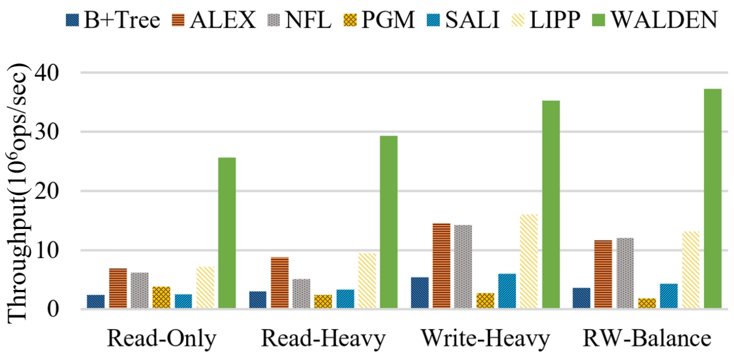


Fig. 16: Hotkey workload

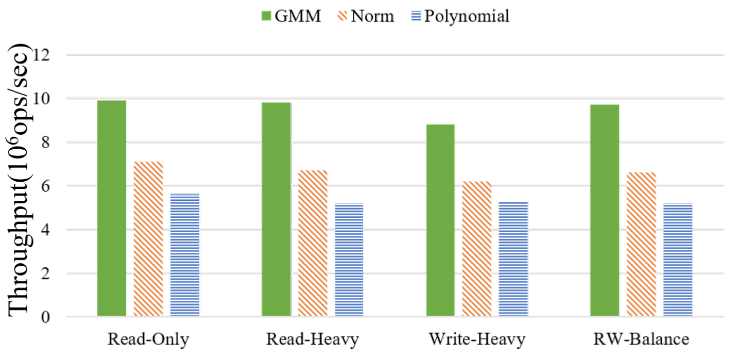


Fig. 17: Fitting Models

size. Siberia [15] distinguish hot keys and cold keys and store them in different storage mediums while we improve the existing index's performance by distinguishing keys' popularity and changing the node depth in index tree. Though some principles are similar, the technique designs are significantly different.

Learned Index: RMI [28] is composed of a hierarchy of models where the upper level takes the key as an input and outputs the model we choose in the next level until we reach the final level and get position by scanning small target payload restricted by an error bound. RMI has inspired a series of learned indexes. FITing-Tree [18] uses linear models to replace the leaf nodes of a B-Tree to reduce memory cost. CARMI [47] based on RMI further improves the construction process by applying data partition and supports data update operation. RadixSpline [26] builds the index by scanning once over the sorted data with splines functions as the learned model. PGM [16] employs piecewise linear models to approximate the distribution of keys in the dataset with error bound and support insert operation using the LSM-tree-inspired logarithmic method. ALEX [13] improves the approximation ability of linear regression models by introducing a gapped array for each leaf node where exponential search to locate the key and use dynamic node expansion and split to support update. LIPP [44] minimizes the conflict degree, which stands for approximation quality with kernelized linear functions and generates new nodes for conflicted keys to further improve the performance. NFL [45] first uses Normalizing Flow to transform the original complex key distribution into a near-uniform distribution, then designs an index structure based on the characteristics of the transformed keys. DILI [31] first builds the index bottom-up way with linear regression models, then uses the top-down manner to adjust the tree structure to further improve performance. SALI [20] targeting multi-core data storage conditions and propose fine-grained locks to convert the index to a multi-thread version as well as improve its performance. Previous works try different ways to improve the index performances: (1) Optimize the index structure, adopting different ways to build the index [16, 28, 31]. (2) Use different strategies to generate the data layout [13, 45]. (3) Optimize the mapping model [16, 28, 44]. (4) Changing the ways of index operation [13, 20, 44]. In contrast, WALDEN targets the frequency of the lookup keys and exploits the corresponding workload to optimize the index.

Based on the idea that different keys have different access frequencies, we wish to design an index that can capture this information and optimize performance. We first use a probabilistic distribution model to model the access frequency information, and we set a new attribute, the key weight to every key, which shows the priority difference between keys. Based on the state-of-the-art (SOTA) model LIPP, we further modify the tree construction process to take advantage of key weight so as to put the key with higher query frequency into the shallower level of the index tree. In this way, the index tree we built will consider the priority of keys and thus be aware of the query workload. What's more, we also propose an index regulation mechanism to monitor the workload change and perform necessary regulations.

7 CONCLUSION AND FUTURE WORK

In this work, we propose workload-aware learned tree index WALDEN supports a full set of index operations for 1D keys. We improve the performance of the previously learned index by giving keys priority and arranging keys with higher priority into the shallower layer of the tree index to improve overall throughput. We introduce metrics and algorithms to evaluate and construct tree index structures. We also present an adjustment strategy to support efficient insert operation. We further propose a mechanism to monitor and handle workload distribution shift conditions. Extensive experiment results on popular datasets demonstrate the advantage of our proposed method. For future work, we plan to investigate a multi-thread realization of WALDEN, inspired by SALI [20] and [43], perhaps we can use item-level optimistic locks for WALDEN to support the concurrent index operations from multiple threads.

References

1. Abdi, H., et al.: The method of least squares. Encyclopedia of measurement and statistics **1**, 530–532 (2007)
2. Amazon sales rank data for print and kindle books.: (2019), `https://www.kaggle.com/ucffool/amazon-sales-rank-data-for-print-and-kindle-books.`
3. Askitis, N., Zobel, J.: B-tries for disk-based string management. The VLDB Journal **18**(1), 157–179 (2009)
4. Athanassoulis, M., Ailamaki, A.: Bf-tree: approximate tree indexing. In: Proceedings of the 40th International Conference on Very Large Databases. No. CONF (2014)
5. AWS, A.: Openstreetmap on aws (2021), `https://registry.opendata.aws/osm.`
6. Bayer, R.: Symmetric binary b-trees: Data structure and maintenance algorithms. Acta informatica **1**, 290–306 (1972)
7. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. In: Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control. pp. 107–141 (1970)
8. benchmark, R.: A repository to test pgms and rmis on different platforms using a much simpler benchmark harness than sosd (2020), `https://github.com/RyanMarcus/rmi_pgm`
9. Boehm, M., Schlegel, B., Volk, P.B., Fischer, U., Habich, D., Lehner, W.: Efficient in-memory indexing with generalized prefix trees. Datenbanksysteme für Business, Technologie und Web (BTW) (2011)
10. Boyar, J., Larsen, K.S.: Efficient rebalancing of chromatic search trees. Journal of Computer and System Sciences **49**(3), 667–682 (1994)
11. cpp btree: The C++ B-Tree library implemented by Google. (2011), `https://code.google.com/archive/p/cpp-btree`
12. Cho, E., Myers, S.A., Leskovec, J.: Friendship and mobility: user movement in location-based social networks. In: Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 1082–1090 (2011)
13. Ding, J., Minhas, U.F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., et al.: Alex: an updatable adaptive learned index. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. pp. 969–984 (2020)
14. Dinh, L., Krueger, D., Bengio, Y.: NICE: non-linear independent components estimation. In: Bengio, Y., LeCun, Y. (eds.) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings (2015), `http://arxiv.org/abs/1410.8516`
15. Eldawy, A., Levandoski, J., Larson, P.Å.: Trekking through siberia: Managing cold data in a memory-optimized database. Proceedings of the VLDB Endowment **7**(11), 931–942 (2014)
16. Ferragina, P., Vinciguerra, G.: The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. Proceedings of the VLDB Endowment **13**(8), 1162–1175 (2020)
17. Fredkin, E.: Trie memory. Communications of the ACM **3**(9), 490–499 (1960)
18. Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., Kraska, T.: Fiting-tree: A data-aware index structure. In: Proceedings of the 2019 international conference on management of data. pp. 1189–1206 (2019)

19. Ge, J., Shi, B., Chai, Y., Luo, Y., Guo, Y., He, Y., Chai, Y.: Cutting learned index into pieces: An in-depth inquiry into updatable learned indexes. In: 2023 IEEE 39th International Conference on Data Engineering (ICDE). pp. 315–327. IEEE (2023)
20. Ge, J., Zhang, H., Shi, B., Luo, Y., Guo, Y., Chai, Y., Chen, Y., Pan, A.: Sali: A scalable adaptive learned index framework based on probability models. Proceedings of the ACM on Management of Data **1**(4), 1–25 (2023)
21. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: Proceedings 14th International Conference on Data Engineering. pp. 370–379. IEEE (1998)
22. Graefe, G., Larson, P.A.: B-tree indexes and cpu caches. In: Proceedings 17th International Conference on Data Engineering. pp. 349–358. IEEE (2001)
23. Kaczmarski, K.: B+-tree optimized for gpgpu. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 843–854. Springer (2012)
24. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: Fast: fast architecture sensitive tree search on modern cpus and gpus. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 339–350 (2010)
25. Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., Neumann, T.: Sosd: A benchmark for learned indexes. NeurIPS Workshop on Machine Learning for Systems (2019)
26. Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., Neumann, T.: Radixspline: a single-pass learned index. In: Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management. pp. 1–5 (2020)
27. Kissinger, T., Schlegel, B., Habich, D., Lehner, W.: Kiss-tree: smart latch-free in-memory indexing on modern architectures. In: Proceedings of the Eighth International Workshop on Data Management on New Hardware. pp. 16–23 (2012)
28. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: Proceedings of the 2018 international conference on management of data. pp. 489–504 (2018)
29. Lehman, T.J., Carey, M.J.: A study of index structures for main memory database management systems. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (1985)
30. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: Artful indexing for main-memory databases. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE). pp. 38–49. IEEE (2013)
31. Li, P., Lu, H., Zhu, R., Ding, B., Yang, L., Pan, G.: DILI: A distribution-driven learned index. Proc. VLDB Endow. **16**(9), 2212–2224 (2023). `https://doi.org/10.14778/3598581.3598593`, `https://www.vldb.org/pvldb/vol16/p2212-li.pdf`
32. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proceedings of the 7th ACM european conference on Computer Systems. pp. 183–196 (2012)
33. Marcus, R., Kipf, A., van Renen, A., Stoian, M., Misra, S., Kemper, A., Neumann, T., Kraska, T.: Benchmarking learned indexes. Proc. VLDB Endow. **14**(1), 1–13 (2020)
34. Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for rdf. Proceedings of the VLDB Endowment **1**(1), 647–659 (2008)

35. Overmars, M.H.: The design of dynamic data structures, vol. 156. Springer Science & Business Media (1983)
36. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (lsm-tree). Acta Informatica **33**, 351–385 (1996)
37. Pandey, V., Kipf, A., Neumann, T., Kemper, A.: How good are modern spatial analytics systems? Proceedings of the VLDB Endowment **11**(11), 1661–1673 (2018)
38. Rao, J., Ross, K.A.: Making b+-trees cache conscious in main memory. In: Proceedings of the 2000 ACM SIGMOD international conference on Management of data. pp. 475–486 (2000)
39. Reynolds, D.A., et al.: Gaussian mixture models. Encyclopedia of biometrics **741**(659-663) (2009)
40. Rezende, D., Mohamed, S.: Variational inference with normalizing flows. In: International conference on machine learning. pp. 1530–1538. PMLR (2015)
41. S2 Geometry.: (2019), `https://s2geometry.io/`.
42. Shahvarani, A., Jacobsen, H.A.: A hybrid b+-tree as solution for in-memory indexing on cpu-gpu heterogeneous computing platforms. In: Proceedings of the 2016 International Conference on Management of Data. pp. 1523–1538 (2016)
43. Wongkham, C., Lu, B., Liu, C., Zhong, Z., Lo, E., Wang, T.: Are updatable learned indexes ready? Proc. VLDB Endow. **15**(11), 3004–3017 (2022). `https://doi.org/10.14778/3551793.3551848`, `https://www.vldb.org/pvldb/vol15/p3004-wongkham.pdf`
44. Wu, J., Zhang, Y., Chen, S., Chen, Y., Wang, J., Xing, C.: Updatable learned index with precise positions. Proc. VLDB Endow. **14**(8), 1276–1288 (2021). `https://doi.org/10.14778/3457390.3457393`, `http://www.vldb.org/pvldb/vol14/p1276-wu.pdf`
45. Wu, S., Cui, Y., Yu, J., Sun, X., Kuo, T., Xue, C.J.: NFL: robust learned index via distribution transformation. Proc. VLDB Endow. **15**(10), 2188–2200 (2022). `https://doi.org/10.14778/3547305.3547322`, `https://www.vldb.org/pvldb/vol15/p2188-wu.pdf`
46. Zhang, H., Andersen, D.G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R.: Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In: Proceedings of the 2016 International Conference on Management of Data. pp. 1567–1581 (2016)
47. Zhang, J., Gao, Y.: CARMI: A cache-aware learned index with a cost-based construction algorithm. Proc. VLDB Endow. **15**(11), 2679–2691 (2022). `https://doi.org/10.14778/3551793.3551823`, `https://www.vldb.org/pvldb/vol15/p2679-gao.pdf`