Chameleon: Towards Update-Efficient Learned Indexing for Locally Skewed Data

Na Guo[1,2] Yaqi Wang[2] Wenli Sun[2] Yu Gu[1*] Jianzhong Qi[3] Zhenghao Liu[1] Xiufeng Xia[2] Ge Yu[1]
[1]Northeastern University, [2]Shenyang Aerospace University, [3]The University of Melbourne
{1710582@stu, {guyu, liuzhenghao, yuge}@mail}.neu.edu.cn
{{wangyaqi, sunwenli}@stu, xiaxiufeng@mail}.sau.edu.cn, jianzhong.qi@unimelb.edu.au

***Abstract*—Recently, learned indexes are assisting and are being adopted to replace traditional indexes for their low memory usage and high query performance. However, existing learned indexes suffer in query efficiency when dealing with locally skewed data distributions which may be caused or exacerbated by ubiquitous updates. Frequent model retraining and reconstruction is required under this circumstance. To address this issue, we present `Chameleon`, an adaptive learned index for locally skewed data especially in the context of frequent updates. We propose a metric for measuring local skewness, based on which we employ Multi-Agent Reinforcement Learning to assist in locating locally skewed regions and optimizing index structures. Additionally, to reduce the blocking time caused by index model retraining, we propose a lightweight lock named the Interval Lock to achieve a non-blocking retraining. Extensive experiments demonstrate that, without costing more memory, `Chameleon` outperforms the state-of-the-art learned indexes by up to 3.75× and 4.37× in lookup times for read-only and mixed workloads, respectively, and it accelerates update processing by up to 2.92×.**

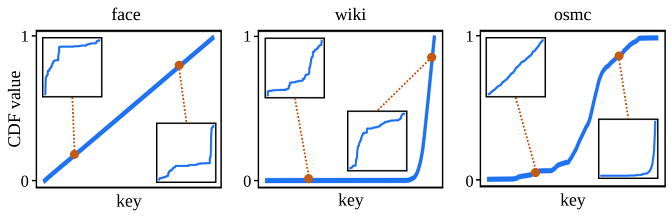
***Index Terms*—Learned Index, Locally Skewed Data, Frequent Updates, Multi-Agent Reinforcement Learning**

I. Introduction

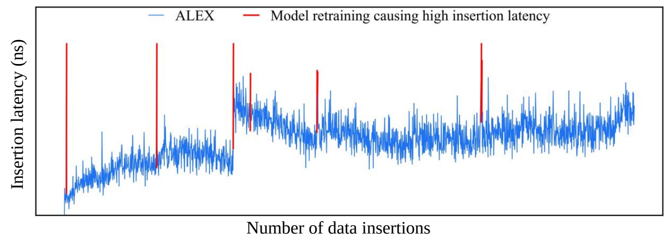
Indexes play a crucial role in databases to speed up query processing by reducing data scans. They are increasingly important in the big data era, where datasets have become too large to be scanned for each single query. Given a dataset, a learned index utilizes machine learning (ML) techniques [1] to learn the mapping from index keys to data storage positions. Recent studies [2]–[4] demonstrate that learned indexes outperform traditional indexes such as B+Tree [5] in both query time and memory costs [6]–[9].

While learned indexes have efficient query procedures, two significant challenges remain: **(i) It is challenging to construct a learned index with ML models that closely fit the distributions of datasets with varying distributions at local regions.** Fig. 1(a) shows the global data distributions and some local data distributions of three real datasets. There are different degrees of skewness at different regions in the datasets. We refer to the region where the local data distribution is skewed as the "locally skewed region". Frequent updates of data can lead to or aggravate local data skewness. Existing solutions [7], [8], [10], [11] typically apply greedy construction strategy to construct index structures, which may not accommodate well the local data skewness. **(ii) Reconstruction and retraining due to frequent updates have a significant impact on query latency.** The query efficiency of learned indexes is update-sensitive. Learned indexes are typically composed of a series of models. With more data insertions, the models which are constructed to fit the initial data distributions may not fit the updated distributions. To maintain validity, a learned index needs to split/merge nodes or to retrain (some of) the models. Frequent retraining leads to a higher maintenance cost. Taking ALEX – the learned index that has reported the best overall query performance across different datasets – as an example, Fig. 1(b) shows that the insertion latency of the index fluctuates constantly. Frequent retraining leads to a significant increase in insertion latency, as indicated by the red peaks in Fig. 1(b).

* Corresponding author



(a) The CDFs of three commonly used datasets



(b) Oscillations of insertion delays caused by data updates

Fig. 1. Motivating example

In this paper, we propose a learned index to address the challenge of measuring and processing locally skewed data distribution (**`Chameleon`**), which can withstand frequent updates of data acting like a chameleon. We present an Error Bounded Hashing (EBH) as the model of leaf nodes to address local skewness, which leverages the hashing capability to flatten locally skewed data. We then introduce a metric to effectively measure the local skewness of data distributions, based on which Deep Reinforcement Learning (DRL) is used to effectively locate locally skewed regions. Subsequently,

TABLE I
COMPARISON OF REPRESENTATIVE INDEX STRUCTURES

Index	Construction[1]		Point Query[2]		Insertion		Local Skewness[3]		
	Direction	Strategy	Inner Node	Leaf Node	Strategy	Retraining	Strategy	Weakness	Support
B+Tree	TD	Greedy	BS	BS	In-place	Blocking	Keep balance	Imprecise inner nodes	✓✓
DIC	TD	RL	BS / Hash	BS / Hash	In-place	Blocking	Keep balance	Imprecise inner nodes	✓✓
RS	TD	Greedy	RT	LIM+BS	In-place	Blocking	—	—	×
PGM	BU	Greedy	PLM+BS	PLM+BS	Out-of-place	Blocking	Rebuild balance	Imprecise inner nodes	✓
ALEX	TD	Cost-based	LIM	LRM+ES	In-place	Blocking	—	—	×
LIPP	TD	Greedy	KLM	—	In-place	Blocking	—	—	×
DILI	BU+TD	Greedy	LIM	—	In-place	Blocking	—	—	×
FINEdex	TD	Greedy	LIM	LRM+BS+LS	Out-of-place	non-Blocking	Use Level Bin	Level Bin scan [12]	✓
`Chameleon`	TD	MARL	LIM	Hash+LS	In-place	non-Blocking	Use Hash	—	✓✓✓✓

[1] TD, BU, RL, and MARL denote top-down, bottom-up, reinforcement learning and multi-agent reinforcement learning, respectively.
[2] BS, ES, PLM, RT, LIM, LRM, KLM, and LS denote binary search, exponential search, piecewise linear model, radix table, linear interpolation model, linear regression model, kernelized linear model and linear scan, respectively.
[3] "Strategy" represents how the index handles local skewness; "Weakness" represents extra operations compared to other indexes; × represents the inability to handle local skewness, while ✓represents support. The number of ✓indicates the degree of support.

we propose an index construction algorithm based on Multi-Agent RL (MARL). Additionally, to handle frequent data updates, we propose a non-blocking retraining mechanism via a lightweight lock and RL.

In summary, our contributions are as follows:

1) We design a memory-efficient learned index structure named **`Chameleon`** for one-dimensional data that can adapt to data with local skew distributions, via a combination of linear models for the non-leaf nodes and Error Bounded Hashing (EBH) models for the leaf nodes (Section III);

2) We propose a metric for measuring local skewness, and a novel approach based on MARL to construct index structures, effectively locating locally skewed regions (Section IV);

3) We define a lightweight lock, called Interval Lock, and a non-blocking retraining mechanism based on Interval Lock to address online retraining caused by frequent data updates (Section V);

4) Extensive experiments on both real and synthetic datasets demonstrate that **`Chameleon`** outperforms state-of-the-art learned indexes, achieving up to 3.75× and 4.37× speedups in lookup times for various workloads, respectively, and 2.92× speedups in update processing – without taking up more memory (Section VI).

II. RELATED WORK

We focus on one-dimensional learned indexes, which are a basic component in most learned multi-dimensional indexes [13]–[15]. The core idea of learned indexes is to design a mapping from an index key to the storage position of the corresponding data record, and such mappings can be learned with a mathematical model (i.e., an index model). When the data records are stored by the order of their index keys, the index models essentially learn the Cumulative Distribution Function (CDF) of index keys. We compare and summarize traditional indexes and representative learned indexes in Table I, which will be discussed in detail next. It can be observed that, compared to traditional indexes and other advanced learned indexes, **`Chameleon`** can better address local skewness.

A. Index Construction Algorithms

Different index construction strategies have been proposed for learned indexes, resulting in various data segmentation results, to fit the diverse data distributions. Fig. 2 illustrates three representative learned index segmentation strategies, where the blue lines represent the CDF of the dataset to be indexed, and the red lines represent the CDFs learned by the index models.

Bottom-Up. FITing-Tree [10] proposes a bottom-up greedy algorithm to construct indexes based on a given error threshold. An index model is used for a continuous data segment until its approximation error exceeds the error threshold. Another typical example is PGM [8], which constructs optimal piecewise linear models in $O(n)$ time by minimizing convex hulls, as plotted in Fig. 2(a). DILI [16] considers both tree height and leaf model accuracy in its construction process.

Top-Down. Top-down approaches avoid a secondary search in inner nodes at query time using inner nodes with precise index models. RMI [6], RS [9] and RUSLI [17] partition intervals based on the linear model or radix table. The performance of these indexes relies on the selection of hyper-parameters (e.g., the number of levels). PLEX [18] improves the robustness of RS by replacing the radix table with a radix tree. LIPP [11] addresses conflicts caused by skewed data through node splitting, as plotted in Fig. 2(b). ALEX [7] and CARMI [19] select the fanout based on a cost model to indirectly adapt to varying local data distributions, as plotted in Fig. 2(c). Additionally, Pluggable [20] accelerates index construction and querying using sampling and model-driven approaches.

Other Frameworks and Strategies. A series of studies [21]–[27] optimise learned indexes under various hardware and workload settings. Polyfit [28] is optimized for approximate range aggregate queries. MR-FT [29] pre-trains a plug-in for index construction on a set of synthetic datasets. NFL [30] first transforms the input data distribution into a near-uniform distribution for batched queries. [31] summarizes methods for expanding and optimizing learned indexes on disk. SALI [32] uses evolutionary strategies and lightweight statistical methods

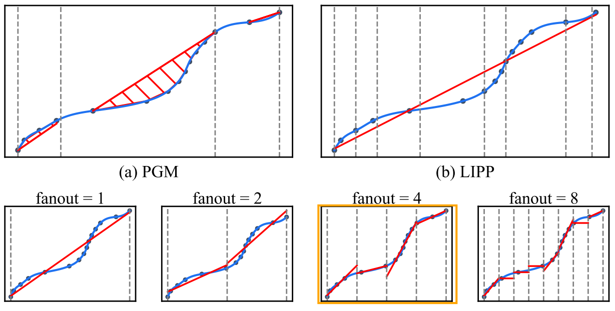


Fig. 2. Comparison of segmentation strategies of learned index structures

to enhance the concurrency performance of learned indexes. Sieve [33] optimizes key-value storage in disk and distributed scenarios. Tailored [34] employs logarithmic error optimization for linear regression problems. RL techniques [35]–[39] have been used in Database regions. For example, DIC [40] finds an approximately optimal combination of existing traditional indexes.

Discussion. As summarized in Table I, top-down algorithms achieve better query performance due to avoiding searches in the inner nodes, but existing top-down construction algorithms are unable to construct the optimal index structure due to their greedy or heuristic strategies. On the other hand, bottom-up methods suffer from poor query performance due to errors in inner nodes. Locally skewed data distribution will lead to an increase in the model prediction error or search depth, both of which may cause performance degradation. There is no index structure can adapt to data with locally skewed distributions.

B. Update Strategy

Existing update strategies [41], [42] for learned indexes can be categorized into two categories: *in-place* and *out-of-place*.

In-Place. ALEX [7], CARMI [19], and TALI [43] use a "gap array" for their leaf nodes to support data insertions. When few keys are inserted, only several key sequences are affected. As the insertion frequency increases, the data distribution may become skewed. TALI [43] adjusts the index structure based on the update-distribution. However, the update distribution is not always known.

Out-of-Place. FITing-Tree [10], PGM [8], and RUSLI [17] use delta-buffer to process insertions. A key is inserted if and only if the buffer it belongs to is not full. Compared to the *in-place* approach, this approach avoids shifting massive keys. However, a delta-buffer impacts the query performance of the index and cannot effectively support unique keys.

Other Algorithms. XIndex [44] handles concurrent writes without affecting the query performance by leveraging fine-grained synchronization and a two-phase compaction scheme. FINEdex [12] utilizes a flattened data structure to construct independent models and process queries concurrently.

Discussion. Since most learned indexes utilize linear models, they are unable to closely fit the data distributions with locally skewed regions. Although using buffers or gap arrays can store the updated data, they can only alleviate the local adjustments required by frequent updates to a certain extent. Updates accumulated in a longer time may cause the actual data distribution to deviate from the initial data distributions, based on which the index models have been constructed. The index models must be retrained to maintain the query efficiency. Moreover, it is difficult to determine the number of gaps and where to set the gaps.

III. Overview of **`Chameleon`**

This section presents our index structure, starting with definitions of a few basic concepts and a problem statement. We summarize the symbols used in Table II.

TABLE II
NOTATIONS

Notation	Description
$\mathcal{D}, \lvert\mathcal{D}\rvert$	The dataset to be bulk loaded and the size of $\mathcal{D}$
$\mathcal{N}_{ij}$	The node at the ith level and jth position
$\mathcal{N}_{ij}.f$	The fanout of $\mathcal{N}_{ij}$
$\mathcal{N}_{ij}.n, \mathcal{N}_{ij}.c$	The quantity of keys and the capacity of $\mathcal{N}_{ij}$
$\mathcal{N}_{ij}.lk, \mathcal{N}_{ij}.uk, \mathcal{N}_{ij}.cd$	The minimum / maximum key and maximum offset of $\mathcal{N}_{ij}$

Definition 1. ***Offset.*** *Offset refers to the distance between the predicted storage position $\hat{\mathcal{P}}$ of a key and its actual position $\mathcal{P}$, i.e., $offset = |\hat{\mathcal{P}} - \mathcal{P}|$.*

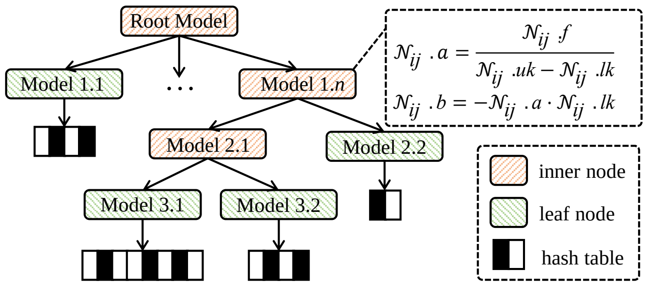
Definition 2. ***Conflict Degree.*** *Given a function $\mathcal{P}(k)$, two keys k_i and k_j conflict if and only if $\mathcal{P}(k_i)$ equals to $\mathcal{P}(k_j)$. For a node $\mathcal{N}_{ij}$ with $\mathcal{N}_{ij}.c$ slots and a function $\mathcal{P}(k)$, the conflict degree $cd = \max_{i \in [0, \mathcal{N}_{ij}.c-1]} \{0, |\{k \in \mathcal{D} | \mathcal{P}(k) = i\}| - 1\}$, i.e., maximum offset.*

Consider a mapping function $\mathcal{P}(k) = 131 \cdot (10/8 \cdot (k - 3)) \bmod 10$, the dataset $\mathcal{D}$={3, 4, 5, 6, 7, 9, 11} and the capacity of the node is 10. For each key, the predicted positions $\hat{\mathcal{P}}$ by $\mathcal{P}(k)$ are 0, 3, 7, 1, 5, 2, 7. The quantity of keys falling into each slot is 1, 1, 1, 1, 0, 1, 0, 2, 0, 0. When only one key exists in a slot, there is no conflict. Therefore, we subtract 1 from the quantity of keys in each slot. This may lead to negative values, so we further compare the result of subtracting 1 from each slot's key count with 0 and take the maximum value. Hence, the conflicts in each slot are 0, 0, 0, 0, 0, 0, 0, 1, 0, 0. Finally, the maximum conflict among all slots is the node's conflict degree. Then the conflict degree is 1.

A. The Structure of ***`Chameleon`***

The overall structure of **`Chameleon`** is illustrated in Fig. 3. The index consists of two types of nodes: inner nodes and leaf nodes. Each inner node contains a linear model corresponding to a data interval, and each leaf node contains an Error Bounded Hashing (EBH) as the model, which aims to address local skewness.

Inner Nodes. The inner nodes are used for mapping keys to child nodes, i.e., each node represents a non-overlapping data partition. The inner nodes store pointers to child nodes and



Root Model
Model 1.1
Model 1.n
Model 2.1
Model 2.2
Model 3.1
Model 3.2
$\mathcal{N}_{ij}.a = \frac{\mathcal{N}_{ij}.f}{\mathcal{N}_{ij}.uk - \mathcal{N}_{ij}.lk}$
$\mathcal{N}_{ij}.b = -\mathcal{N}_{ij}.a \cdot \mathcal{N}_{ij}.lk$
inner node
leaf node
hash table

Fig. 3. The **`Chameleon`** index structure

map keys to the corresponding child nodes. The fanout $\mathcal{N}_{ij}.f$ of an inner node $\mathcal{N}_{ij}$ is determined via RL agents (detailed in Section IV). Since we partition by key intervals, the predictions of the inner node models are precise, eliminating the need for secondary search inside an inner node. The index model used by an inner node $\mathcal{N}_{ij}$ is defined as follows:

$$\mathcal{ID}(k) = \frac{\mathcal{N}_{ij}.f}{\mathcal{N}_{ij}.uk - \mathcal{N}_{ij}.lk} \cdot (k - \mathcal{N}_{ij}.lk) \tag{1}$$

where k represents an arbitrary key, and $\mathcal{ID}$ denotes the rank of the child node of inner node $\mathcal{N}_{ij}$ corresponding to k.

Leaf Nodes. Each leaf node has an EBH with an adaptive-sized $\mathcal{N}_{ij}.c$ to hash dense data more evenly, which distinguishes our leaf nodes from those used in existing works. When performing queries, a hash function is used to efficiently predict the slot where the query key k is stored. Due to potential hash collisions, the hash function has prediction errors. Therefore, each leaf node stores a maximum offset $\mathcal{N}_{ij}.cd$ of the EBH to accelerate query processing. Specially, if the linear scanning process in $\mathcal{N}_{ij}$ exceeds $[\hat{\mathcal{P}} - \mathcal{N}_{ij}.cd, \hat{\mathcal{P}} + \mathcal{N}_{ij}.cd]$, then k is not in $\mathcal{N}_{ij}$. Given a hash factor α, the hash function for a leaf node $\mathcal{N}_{ij}$ is defined as follows:

$$\mathcal{P}(k) = \alpha \cdot (\frac{\mathcal{N}_{ij}.c}{\mathcal{N}_{ij}.uk - \mathcal{N}_{ij}.lk} \cdot (k - \mathcal{N}_{ij}.lk)) \bmod \mathcal{N}_{ij}.c \tag{2}$$

where $\hat{\mathcal{P}}$ represents the position of k in EBH mapped by the hash function $\mathcal{P}(k)$.

B. The Challenge of **`Chameleon`**

Local skewness typically occurs at local regions, leading to lop-sided partition issues. Unlike existing index structures that address local skewness by splitting downwards, **`Chameleon`** first employs RL to locate segments with local skewness and uses an EBH as the leaf node models to hash the locally skewed data. This is because minor changes in the input can lead to substantial changes in the corresponding hash values, allowing the hash function to disperse dense data, thereby effectively reducing the height of the index tree.

A key challenge in addressing the issue of local skewness is how to measure and identify it, which is particularly important when there are data updates that may cause such the locally skewed regions to change. Next, we present a metric to measure the degree of local skewness as defined in Definition 3.

Definition 3. ***Local Skewness.*** *Local Skewness is a statistic describing the skewness of local data distribution. Given a sorted dataset $\mathcal{D}$, local skewness $lsn = arctan(\frac{1}{(|\mathcal{D}|-1)^2}\sum_{i=1}^{|\mathcal{D}|-1}\frac{Mk-mk}{k_i-k_{i-1}})$, $\frac{\pi}{4} \leq lsn < \frac{\pi}{2}$, where k_i denotes the key located at position i in $\mathcal{D}$. Mk and mk represent the maximum and minimum keys in $\mathcal{D}$, respectively.*

Problem Statement. Given a dataset being bulk loaded $\mathcal{D}$, our goal is to construct an learned index based on data distributions (e.g., reducing the height of tree and $offset$). We also aim to address local skewness problem based on lsn and cd.

Next, with Fig. 4, we describe the construction process of **`Chameleon`**, using two modules DARE and TSMDP based on RL (detailed in Section IV), followed by a retraining module (detailed in Section V). DARE and TSMDP are RL agents responsible for optimizing different parts of the index respectively, and collaborate to optimize the entire index structure. The retraining module triggers TSMDP to refine the local structure in response to data updates. Specifically, given a dataset $\mathcal{D}$, we first extract the data distribution information of the entire dataset (i.e., Probability Density Function (PDF) of $\mathcal{D}$, $|\mathcal{D}|$, and lsn of $\mathcal{D}$) and input it into DARE. The DARE agent outputs a matrix of fixed size to determine the fanout of nodes in the upper h-1 levels, and thus the upper h-levels of the index is constructed. The fanouts of the non-root inner nodes for most one-dimension learned indexes are set between 1 and 2^{10}, while the fanouts of their root nodes are set between 1 and 2^{20} in order to explore larger space. Similarly, our model aims to learn the fanouts with 2^{10} and 2^{20} as the upperbounds of non-root inner nodes and root nodes, respectively. In this case, the lowerbound of the tree height can be derived to be $\lceil \log_{2^{10}}(|\mathcal{D}|) \rceil$, which is defined as the value of h. Next, we extract the data distribution information corresponding to each node at the h-th level and feed such information into TSMDP. The TSMDP agent outputs the fanout (may equal 1) of each node. After the entire index structure is constructed, a retraining thread periodically invokes the TSMDP agent to adjust the local structure without blocking queries when there are data updates that change the local data distributions.

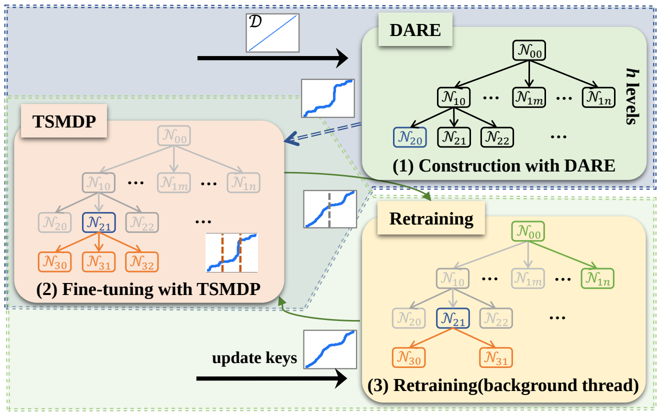


Fig. 4. The **`Chameleon`** framework

IV. Chameleon Index Construction

In this section, we firstly present the construction process of leaf nodes to resolve local skewness, followed by a tree-based index construction method called TSMDP based on RL. We further add a second RL agent named DARE to help reduce the index construction and model training time through MARL.

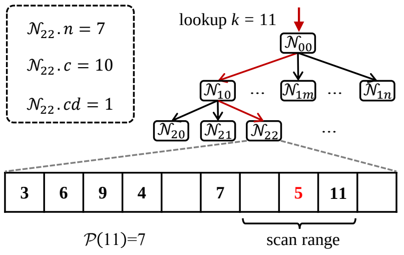
A. Resolving local skewness

The index construction process can be considered as a data partitioning process, and the core problem is to determine the fanout $\mathcal{N}_{ij}.f$ of each node $\mathcal{N}_{ij}$. When $\mathcal{N}_{ij}.f > 1$, data partitioning will be recursively executed, i.e., $\mathcal{N}_{ij}$ forms an inner node. If $\mathcal{N}_{ij}.f = 1$, the partitioning process ends, i.e., $\mathcal{N}_{ij}$ is now a leaf node, we insert the data into $\mathcal{N}_{ij}$.

Theorem 1. *Given the desired collision probabilities τ and the quantity of keys in leaf nodes $\mathcal{N}_{ij}.n$, satisfying τ requires that the node capacity $\mathcal{N}_{ij}.c \geq \frac{\mathcal{N}_{ij}.n-1}{-\ln(1-\tau)}$.*

Proof 1. *The probability of node $\mathcal{N}_{ij}$ not causing a collision is: $p(\bar{\mathcal{N}_{ij}.n}) = 1 \cdot (1 - \frac{1}{\mathcal{N}_{ij}.c})(1 - \frac{2}{\mathcal{N}_{ij}.c}) \dots (1 - \frac{\mathcal{N}_{ij}.n-1}{\mathcal{N}_{ij}.c})$. According to the Taylor series expansion and the limit, it can be inferred that: $\lim_{-1/\mathcal{N}_{ij}.c \to 0} e^{-\frac{1}{\mathcal{N}_{ij}.c}} = 1 - \frac{1}{\mathcal{N}_{ij}.c}$. Since $1 - \frac{1}{\mathcal{N}_{ij}.c} \geq 1 - \frac{2}{\mathcal{N}_{ij}.c} \geq \dots 1 - \frac{\mathcal{N}_{ij}.n-1}{\mathcal{N}_{ij}.c}$, we have: $p(\bar{\mathcal{N}_{ij}.n}) \leq (1 - \frac{1}{\mathcal{N}_{ij}.c})^{\mathcal{N}_{ij}.n-1} \leq e^{-\frac{\mathcal{N}_{ij}.n-1}{\mathcal{N}_{ij}.c}}$. Therefore, the probability of at least two keys colliding is: $p(\mathcal{N}_{ij}.n) = 1 - p(\bar{\mathcal{N}_{ij}.n}) \geq 1 - e^{-\frac{\mathcal{N}_{ij}.n-1}{\mathcal{N}_{ij}.c}}$. Assuming that the desired collision probability is τ, i.e., $p(\mathcal{N}_{ij}.n) \leq \tau$. Then, $1 - e^{-\frac{\mathcal{N}_{ij}.n-1}{\mathcal{N}_{ij}.c}} \leq \tau$, and $\mathcal{N}_{ij}.c \geq \frac{\mathcal{N}_{ij}.n-1}{-\ln(1-\tau)}$.* ■

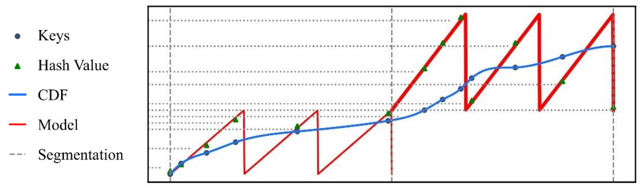
To effectively address the problem of local skewness, we adaptively adjust the capacity of each leaf node $\mathcal{N}_{ij}.c$ based on Theorem 1 to ensure that each node satisfies the desired τ. For example, as shown in Fig. 5(a)(b), assuming that $\mathcal{N}_{22}.n$ is 7, τ is 0.45. According to Theorem 1, to satisfy $\tau \leq 0.45$, $\mathcal{N}_{22}.c$ needs to be at least 10.



(a) Error bounded hash table

k	$\widehat{\mathcal{P}}$	$\mathcal{P}$	*offset*
3	0	0	0
4	3	3	0
5	7	7	0
6	1	1	0
7	5	5	0
9	2	2	0
11	7	8	1

(b) Cost computation



(c) The leaf node of Chameleon index

Fig. 5. Leaf node structure

Once the leaf node capacity $\mathcal{N}_{22}.c$ is determined, the hash function can then be set to $\mathcal{P}(k) = 131 \cdot (10/8 \cdot (k-3)) \bmod 10$, where 10 is the node capacity. As shown by the bold red line in Fig. 5(c), the hash function can effectively scatter the densely populated keys, thus addressing the issue of local skewness.

B. Locating the Locally Skewed Regions

*1) **Motivation**:* In this section, we use RL to locally skewed regions. During the index construction process, each inner node impacts all of its child nodes. Existing Markov Decision Process (MDP)-based RL algorithms focuses on sequential decision problems and are not directly applicable to our tree-based decision process. To address this issue, we propose a Tree-Structured Markov Decision Process (TSMDP) following the idea of the Deep-Q-Network (DQN) [45]. Given the dataset features corresponding to a node $\mathcal{N}_{ij}$ as input, TSMDP outputs the fanout $\mathcal{N}_{ij}.f$ of $\mathcal{N}_{ij}$. Next, we formulate the TSMDP process and then introduce the training process.

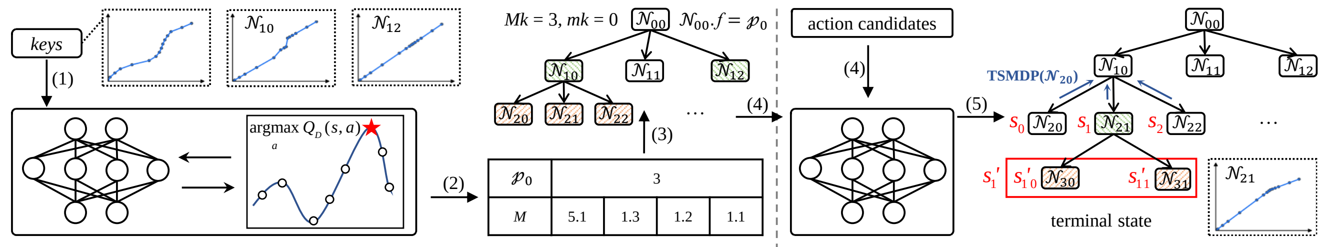
*2) **TSMDP Formulation**:* We proceed to explain how states and actions are represented in our model, and then present the reward signal design, which is particularly challenging.

State space. A state s is the information of a node before partitioning, which contains PDF, the quantity of keys, and lsn. Here PDF is represented by buckets of size b_T. Therefore, the size of the state space is $b_T + 2$.

Action space. An action a refers to the node fanout to be assigned to the node corresponding to the current state. The action space is a predefined set of discrete values $\{\xi_0, \xi_1, \dots, \xi_n\}$.

Transition. Given a state s and an action a, if $a = 1$, TSMDP has reached a terminal state and a leaf node will be constructed. It is worth noting that, due to the characteristics of the tree-based index structure, the current state s can lead to multiple next states. Hence, the next state is formed by the features of the dataset corresponding to all the child nodes of $\mathcal{N}_{ij}$. Consider the example in Fig. 6 in step (4). TSMDP gives $\mathcal{N}_{20}$ a fanout of 1, and s_0 is a termination state. The construction algorithm then runs on node $\mathcal{N}_{21}$. For $\mathcal{N}_{21}$, the fanout is 2, and its next state s'_1 is composed of a set of states $\{s'_{10}, s'_{11}\}$. s'_{10} and s'_{11} coexist and are both the next states of s'_1, which need to be decided by TSMDP.

Reward function. The reward function evaluates the query cost and the memory cost of a constructed index given a state and an action. The reward value indicates the quality of the selected action in the current state. A reward consists of two parts: query time reward and memory reward. It can be expressed as $r = -w_t \cdot R_t - w_m \cdot R_m$. Here, w_t and w_m represent the weight coefficients for query time reward and memory reward, respectively. We use R_t to represent the cost of traversing the tree and secondary searches within leaf nodes, while R_m represents the memory usage of the nodes after taking actions. We note that other factors such as the query distribution can be added to the reward function according to application requirements.


keys
(1)
$\mathcal{N}_{10}$
$\mathcal{N}_{12}$
argmax $Q_D(s,a)$
a
(2)
Mk = 3, mk = 0
$\mathcal{N}_{00}$
$\mathcal{N}_{00}.f = p_0$
$\mathcal{N}_{10}$ $\mathcal{N}_{11}$ $\mathcal{N}_{12}$
$\mathcal{N}_{20}$ $\mathcal{N}_{21}$ $\mathcal{N}_{22}$
(3)
p_0 3
M 5.1 1.3 1.2 1.1
(4)
action candidates
(5)
TSMDP($\mathcal{N}_{20}$)
s_0 s_1 s_2
s_1' s_{10}' s_{11}'
$\mathcal{N}_{30}$ $\mathcal{N}_{31}$
terminal state
$\mathcal{N}_{21}$

(a) DARE

(b) TSMDP

Fig. 6. The **`Chameleon`** index construction process via MARL

*3) **TSMDP Training**:* We adopt DQN with a technique known as experience replay where the RL agent's experience state, action, reward, and next state (s_t, a_t, r_t, s_{t+1}) is stored at each time-step t. In each step, the *boltzman* exploration strategy [46] is used to choose actions. We implement two networks, a policy network Q_T with parameter θ and a target network $\hat{Q_T}$ with parameter θ^-, respectively, which makes TSMDP more stable than using a single network.

Given a batch of transitions (s, a, r, s'), the policy network parameters θ are updated with a gradient descent step by minimizing the Mean Absolute Error (MAE) loss. Specifically, the loss function is:

$$\mathcal{L}_T(\theta) = \sum_{s,a,r,s'} \left(\left|r+\gamma\sum_{z=0}^{a-1} w_z \max_{a'_z} \hat{Q_T}(s'_z, a'_z; \theta^-) - Q_T(s,a;\theta)\right|\right) \tag{3}$$

where $\gamma \in (0,1)$ denotes a discount factor determining the importance of future rewards, a'_z represents the optimal action taken by s'_z. w_z is the weight for s'_z, which is the ratio of key quantity in s'_z to key quantity in s.

Note that the target network parameters θ^- are only synchronized with the policy network parameters θ every K (a system parameter) steps and are held fixed at other times.

*4) **Discussion**:* Although TSMDP aims to achieve the global optimal solution, it still has three limitations: (1) Slow index construction. During the index construction process, we need to recursively call the TSMDP model to make decisions for each node, which is time-consuming. (2) High training overhead. In the training process, each experience includes PDF for each node and its all child nodes. The process of calculating $\hat{Q_T}(s', a')$ one by one for child nodes using $\hat{Q_T}$ is time-consuming. Meanwhile, the use of a discrete action space limits our ability to explore a larger action space. (3) When application requirements change (e.g., the reward function has changed to prioritize query time over index size, or vice versa), TSMDP needs to be retrained. In the next section, we present an optimized **`Chameleon`** index to address these issues.

C. Enhancing TSMDP with DARE

We further improve the construction algorithm through MARL with two RL agents, DARE and TSMDP, where DARE (Dynamic Reward RL) makes single-step decisions based on the global data distribution (i.e., the PDF of $\mathcal{D}$, the lsn of $\mathcal{D}$, and $|\mathcal{D}|$), and then TSMDP fine-tunes based on the local data distribution, as shown in Fig. 6. To address Limitation (1), **`Chameleon`** only needs to invoke DARE once to construct the upper h levels of the index, with only local calls of TSMDP for fine-tuning the lower levels. To address Limitation (2), DARE uses a single-step decision RL model, which has lower training overhead, allowing us to explore a large and continuous action space to find the optimal action. To address Limitation (3), we propose a dynamic reward function to adapt to dynamic system constraints without retraining RL agent.

Specially, given a dataset $\mathcal{D}$ with the maximum and minimum keys Mk and mk, DARE firstly extracts the global data distribution features. Based on these features, it outputs a fixed-size parameter matrix $M(h-2, L)$ and the root node fanout p_0 (the fanout of the root node is a separate output parameter due to its uniqueness in an index). Here, h denotes the number of levels, and each row of the matrix M represents the parameters $p_{i,0}, p_{i,1}, ..., p_{i,L-1}$ for each non-root inner node at the i-th level. After obtaining M and p_0, we first calculate the mapping of $\mathcal{N}_{ij}$ to position x in M, where $x = \frac{(\mathcal{N}_{ij}.lk+\mathcal{N}_{ij}.uk)/2-mk}{Mk-mk} \cdot (L-1)$. Following that, the interval $[p_{i,l}, p_{i,l+1}]$ in M enclosing x is calculated, where $l = \lfloor x \rfloor$. Then, we employ piecewise linear interpolation functions to covert the discrete parameters to continuous values as:

$$\mathcal{N}_{ij}.f = \lfloor (x-l)\cdot p_{i,l+1} + (l+1-x)\cdot p_{i,l} \rceil \tag{4}$$

After the upper h-level index is constructed, based on the index constructed by DARE, TSMDP refines the lower levels, deciding whether to continue splitting. Here, the index construction is completed.

As shown in Fig. 6, assuming $h = 3, L = 4$, the minimum key $mk = 0$ and maximum key $Mk = 3$. In step (1), the dataset feature is extracted and used as the state for DARE. In step (2), DARE outputs the root node fanout p_0 and a 1×4 parameter matrix M. The root node has three child nodes due to $p_0 = 3$, where $\mathcal{N}_{10}.lk = 0$ and $\mathcal{N}_{10}.uk = 1$. Thus, $x = \frac{(0+1)/2-0}{3-0} \cdot 3 = 0.5$, and $l = \lfloor x \rfloor = 0$. Finally, $\mathcal{N}_{10}.f = \lfloor (0.5-0)\cdot 1.3 + (1-0.5)\cdot 5.1 = 3.2 \rceil = 3$. We formulate an MDP process for DARE as follows:

Experience. As DARE deals with a single step RL problem, the experience only consists of state, action and reward.

State space. We also use PDF (represented by a vector of size b_D), $|\mathcal{D}|$, and local skewness lsn to represent a state s_D. Therefore, the size of state space is b_D + 2.

Reward. To adapt to changing application requirements, we propose a Dynamic Reward Function (DRF), expressed as $r_D = \sum_{i=1}^{n} w_i \cdot cost_i$.

Suppose a DQN has been trained to project the high-dimensional state space and action space to low-dimension cost space. Given coefficient weights w_i (according to some specific requirements, s.t., $\sum_{i=1}^{n} w_i = 1$) and the costs of different application specific metrics $cost_1, cost_2, \cdots, cost_n$, DRF can predict the reward r_D. The Q-function remains valid while these weights change.

Action space. Actions a_D represent a set of parameters that determine the fanout of nodes in the upper $h-1$ levels. A larger fanout of the root node is beneficial to improve the query performance of the learned index. The fanouts of the root nodes for most existing 1-dimensional learned indexes are set between 2^0 and 2^{20} such as ALEX [7]. Therefore, the fanout of the root node is searched in the range of $[2^0, 2^{20}]$. For the inner nodes, the fanout range is set as $[2^0, 2^{10}]$ according to the existing works. For the $(h-2) \cdot L$ parameter matrix of DARE, the size L of each row vector is set to 256. In order to provide a stable strategy for action selection, DARE utilizes Genetic Algorithm (GA) [47] as the actor and DQN as the critic. In this approach, GA uses actions as genes and $Q_D(s_D, a_D)$ as the fitness function for a given state. By iteratively improving the fitness through numerical mutations and crossovers of action values, it gradually identifies actions that are closest to the maximum value, which serves as the optimal policy. Algorithm 1 outlines the process of outputting the optimal parameters using GA.

- Population. Since the values in M are finite float values, we can intuitively treat each value as a chromosome. Each individual has $1 + (h-2) \cdot L$ chromosomes.
- Mutation (Lines 3–4) can be divided into two types. The first type adds a set of random fanout to generate entirely new genotypes within the population. The second type allows GA to make better use of existing high-quality genes through slight mutations.
- Crossover (Line 5) also can be divided into two types. The former is multi-point crossover, where a configuration contains $1 + (h-2) \cdot L$ chromosomes, and DARE randomly selects certain chromosomes to pass them to the offspring. The latter involves performing numerical crossover within the same chromosome.
- Fitness (Line 6). The GA algorithm provides a root node parameter p_0 and a parameter matrix M for other inner nodes. Based on these parameters, the query cost and memory cost of the index are predicted by Q-network. Subsequently, the predicted value calculated based on DRF is used as the fitness value.
- Selection (Lines 7–8) process involves retaining a portion of the highest-fitness genes.

Algorithm 1: GetOptimizedParameters

```
input : $s_D, \boldsymbol{w}$
output: $a_D$
1  $gens \leftarrow \emptyset$;
2  for $i \leftarrow 1$ to $K$ do
3      $gens \leftarrow gens \cup X$ random individual;
4      $gens \leftarrow gens \cup$ variations($gens$);
5      $gens \leftarrow gens \cup$ crossed($gens$);
6      $rewards \leftarrow$ evaluate($gens, s_D, \boldsymbol{w}$);
7      $gens \leftarrow$ sort($gens, rewards$);
8      $gens \leftarrow gens[: X]$;
9      if converged then
10         return $gens[0]$;
11 return $gens[0]$;
```

D. Training **Chameleon**

Algorithm 2 summarizes the training process of the whole **Chameleon**. Firstly, we initialize the TSMDP policy network Q_T, target network $\hat{Q_T}$, DARE policy network Q_D, exploration probability er, and exploration termination probability ϵ (Lines 1–2). er determines the tendency between exploration and exploitation used in the selection of a_D. We use a large collection of both real and synthetic datasets as the training set. Subsequently, we randomly select a dataset $\mathcal{D}$ from training set and extract its features as s_D in each episode (Lines 3–6). We randomly generate weights $\boldsymbol{w}$ for the DRF (Line 7), and get the optimal parameters a_{best} for s_D and $\boldsymbol{w}$ based on Algorithm 1 (Line 8). The parameters a_{random} are randomly generated (Line 9). To trade-off between exploration and exploitation, DARE selects a set of parameters a_D (Line 10) based on er which is gradually reduced from 1 to 0 during the training process. The **Chameleon**-Index is instantiated via a_D and Q_T (Lines 11–12). We then train TSMDP Q_T and DARE Q_D (Lines 13–14). Finally, we decrease er and repeat the above process until ϵ is reached (Line 15). The loss function is:

$$\mathcal{L}_D(\theta_D) = \sum_{s_D, a_D, r_D} |Q_D(s_D, a_D; \theta_D) - r_D| \tag{5}$$

where θ_D represents the parameter of Q_D.

Algorithm 2: Train **Chameleon**

```
input : Environment of Chameleon-Index
output: $Q_T$ and $Q_D$
1  Initialize $Q_T, \hat{Q_T}$ and $Q_D$;
2  $er$, $\epsilon \leftarrow 1$, Exploration termination probability;
3  while $er > \epsilon$ do
4      for $i = 0$ to $K$ do
5          $\mathcal{D} \leftarrow$ A random dataset;
6          $s_D \leftarrow$ Extract features of $\mathcal{D}$;
7          $\boldsymbol{w} \leftarrow$ Generate random weights of DRF;
8          $a_{best} \leftarrow GetOptimizedParameters(s_D, \boldsymbol{w})$;
9          $a_{random} \leftarrow$ Generate random parameters;
10         $a_D = (1 - er) \cdot a_{best} + er \cdot a_{random}$;
11         $r_D \leftarrow$ Instantiate Chameleon-Index($h$-1, $a_D$);
12         Refine Chameleon-Index using $Q_T$ ;
13     $Q_T \leftarrow$ Train $Q_T$ with $\hat{Q_T}$;
14     $Q_D \leftarrow$ Train $Q_D$;
15     Decrease $er$;
16 return $Q_T, Q_D$
```

V. Online Retraining for Update

Invoking RL models online can result in expensive retraining overhead. To avoid online retraining when the data distribution is changed by data updates, an intuitive approach is to use an additional thread to retrain the index without blocking queries. This is achieved by employing a locking mechanism to ensure that the query thread and the retraining thread do not access the same node simultaneously. However, node locks require locking and unlocking all nodes along the query path, significantly reducing query performance.

A. Retraining for Update via RL

To overcome the above issues, we propose a mechanism for retraining the **`Chameleon`**-Index using an interval lock, which is defined as follows.

Definition 4. ***Interval Lock.*** *Given the interval $[\mathcal{N}_{ij}.lk, \mathcal{N}_{ij}.uk)$ of nodes $\mathcal{N}_{ij}$ at the h-th level, an interval lock is used to ensure that at any given moment, only one thread can access this interval.*

Since the sibling nodes of **`Chameleon`** are non-overlapping, a node can be uniquely represented by a path from the root node to the node. Based on this observation, we determine whether an interval is accessed or not using $\mathcal{ID}s$ ($\mathcal{ID}$ is computed using Eq 1, where $\mathcal{ID}s$ represents the set of $\mathcal{ID}$ along the query path), without checking for overlap.

When keys are inserted or deleted, the bottom-level leaf nodes usually do not require retraining; they only need to expand their capacity to handle the changes. If we retraining the nodes in the index tree that are too close to the bottom-level, it will inversely increase the cost of retraining. On the other hand, retraining may be required when a subtree has a significant change in data distribution. Therefore, we keep the structure of the upper h-1 levels and only lock the node of the h-th level (i.e., as plotted in Fig. 7, the green and blue dashed boxes). Since each interval corresponds to a set of $\mathcal{ID}s$, we only need to check whether two threads are accessing the same interval by comparing their $\mathcal{ID}s$. Considering TSMDP performs well on small-scale datasets, we retrain the local structure by employing TSMDP as the background thread.

The retraining is periodically executed (every 10s in our evaluation) and can be blocked by query/update workloads. Note that we only focus on the concurrency issue between workloads and the background retraining in this paper. Query/update workloads are still handled sequentially.

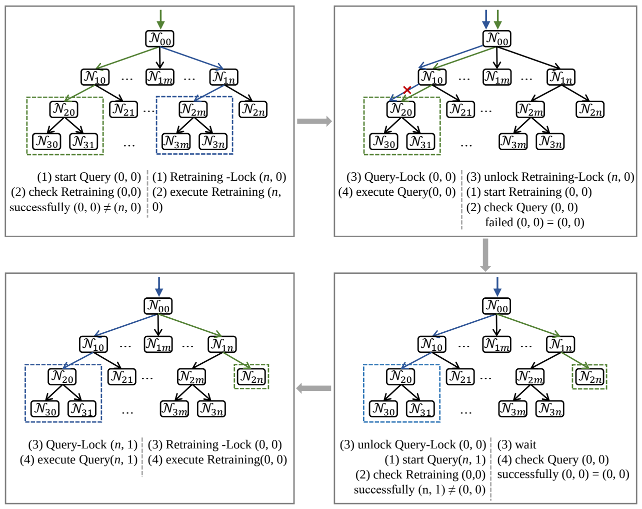
As shown in Fig. 7, node $\mathcal{N}_{20}$ (i.e., $\mathcal{ID}s$(0,0)) is to be retrained when h=3. Firstly, we check if the interval $[\mathcal{N}_{20}.lk, \mathcal{N}_{20}.uk]$ has been accessed by a Query(0,0) thread. At this point, a Query(0,0) thread is indeed accessing this interval, so a Query-Lock has been placed on the interval $[\mathcal{N}_{20}.lk, \mathcal{N}_{20}.uk]$, and the Retraining(0,0) thread's access request is denied. The retraining thread waits for the query thread to finish and unlock the Query-Lock. The retraining thread then checks whether the interval $[\mathcal{N}_{20}.lk, \mathcal{N}_{20}.uk]$ is still being accessed by the query thread. Now, the query thread accesses the interval $[\mathcal{N}_{2n}.lk, \mathcal{N}_{2n}.uk]$ (i.e., $\mathcal{ID}s(n,1)$) and is no longer accessing the interval $[\mathcal{N}_{20}.lk, \mathcal{N}_{20}.uk]$, so the retraining thread can execute. At this point, the retraining thread adds a Retraining-Lock to the interval $[\mathcal{N}_{20}.lk, \mathcal{N}_{20}.uk]$ and performs the retraining operation. After the retraining is completed, it unlocks the Retraining-Lock. Since the $\mathcal{ID}s$ of the query thread and the retraining thread are different at this point, indicating that they are not accessing the same interval, both threads can proceed simultaneously.

Limitations and Assumptions. (1) As the data continues to update, the index structure gradually deviates from the optimum. However, when the number of updated data reaches a certain threshold, any learned index faces complete reconstruction. At this point, our DARE is triggered to reconstruct the overall index structure. (2) Since uniform data distribution is not our design optimization target, we assume that data distribution of the real datasets always exhibits either large or small local skewness. These assumptions are reasonable for most real-world scenarios.

B. Complexity Analysis

Lookup Analysis. The lookup cost is only related to the height of **`Chameleon`** H_C and the predicted error of the leaf nodes. We first analyze the tree height. Since we use MARL to construct a learned index structure (i.e., reducing H_C and τ), H_C is typically 2 to 3 (for datasets of 200M points). The predicted error of leaf nodes (the error of EBH) can be adjusted through Theorem 1. The leaf nodes of **`Chameleon`** have an average querying time complexity of $O(1)$. Therefore, the time complexity of lookup operations is $O(H_C + 1)$, as shown in Table III.

B+Tree, PGM, RS require binary search at inner nodes to locate the next level node. ALEX, FINEdex adopt a greedy strategy in constructing indexes, and their tree heights are higher for locally skewed data. LIPP and DILI perform downward splitting for updates, and their empirical tree heights are also larger than that of **`Chameleon`** for locally skewed data, as shown in the experimental results in the next section.



$\mathcal{N}_{00}$
$\mathcal{N}_{10}$
$\mathcal{N}_{1m}$
$\mathcal{N}_{1n}$
$\mathcal{N}_{20}$
$\mathcal{N}_{21}$
$\mathcal{N}_{2m}$
$\mathcal{N}_{2n}$
$\mathcal{N}_{30}$
$\mathcal{N}_{31}$
$\mathcal{N}_{3m}$
$\mathcal{N}_{3n}$
(1) start Query (0, 0)
(2) check Retraining (0,0) successfully (0, 0) ≠ (n, 0)
(1) Retraining -Lock (n, 0)
(2) execute Retraining (n, 0)
(3) Query-Lock (0, 0)
(4) execute Query(0, 0)
(3) unlock Retraining-Lock (n, 0)
(1) start Retraining (0, 0)
(2) check Query (0, 0) failed (0, 0) = (0, 0)
(3) unlock Query-Lock (0, 0)
(1) start Query(n, 1)
(2) check Retraining (0,0) successfully (n, 1) ≠ (0, 0)
(3) wait
(4) check Query (0, 0) successfully (0, 0) = (0, 0)
(3) Query-Lock (n, 1)
(4) execute Query(n, 1)
(3) Retraining -Lock (0, 0)
(4) execute Retraining(0, 0)

Fig. 7. The concurrent process of executing queries and index reconstruction

TABLE III
TIME COMPLEXITY ANALYSIS

Index	Inner node	Leaf node	Update
`Chameleon`[1]	$\boldsymbol{O(H_C)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(m \cdot \tau)}$
B+Tree	$O(\log \vert\mathcal{D}\vert \cdot \log m)$	$O(\log m)$	$O(m)$
PGM[1]	$O(H_P \cdot \log m)$	$O(\log m + b)$	$O(\log \vert\mathcal{D}\vert)$
RS	$O(H_R + \log T)$	$O(\log m)$	$O(\vert\mathcal{D}\vert)$
ALEX	$O(H_A)$	$O(\log m)$	$O(m)$
LIPP[2]	$O(H_L)$	$\boldsymbol{O(1)}$	$O(\log^2 \vert\mathcal{D}\vert)$
DILI[2]	$O(H_D)$	$\boldsymbol{O(1)}$	$O(\log^2 \vert\mathcal{D}\vert)$
FINEdex	$O(H_F)$	$O(\log m + b)$	$O(m + b)$

[1] $|\mathcal{D}|$, m, b, and T represent the dataset size, leaf node capacity, delta-buffer size, and radix table size, respectively. τ represents conflict rate. H represents the height of the index.
[2] $\log |\mathcal{D}| \gg H_L > H_D > H_P > H_F > H_A > H_C > H_R$.

Update Analysis. We focus on the cost of index updates, which incurs extra overhead for insertions (deletions). Let m be the leaf node capacity, and each shift has a time complexity of $O(m)$. Since shift is not triggered by each insertion (deletion), and shifts only occur if there is a conflict. The probability of conflict is τ, and thus the average update overhead is $O(m \cdot \tau)$. In comparison, for LIPP, the height for the adjusted tree is $O(\log |\mathcal{D}|)$, and each level would cost $O(|\mathcal{D}|)$ time to retrain the index models. Thus, the average update time cost of LIPP is $O(\log^2 |\mathcal{D}|)$. **`Chameleon`** achieves lower complexity for updates than other competitors as summarized in Table III.

VI. EXPERIMENTS

In this section, we present detailed results of our experiments. We implement **`Chameleon`** and other indexes using C++. All experiments are conducted on a 64-bit Ubuntu 22.04 LTS machine equipped with an AMD 7900X CPU, 128GB DDR5 5200Mhz RAM and an NVIDIA 4070 12GB GPU. The source code of **`Chameleon`** is available[1].

A. Experimental Setup

Our experimental setup follows that of the SOSD benchmark [2]. We provide a detailed description of the parameters, datasets, workloads, and baselines.

1) Datasets: Following SOSD, we use two real datasets (OSMC and FACE) and two synthetic datasets (UDEN and LOGN). Each dataset consists of 200 million 64-bit keys.

- UDEN is a uniform distribution dataset with a local skewness of $\frac{\pi}{4}$;
- OSMC is uniformly sampled from OpenStreetMap with a local skewness of $\frac{2\pi}{5}$;
- LOGN is a lognormal distribution dataset with a local skewness of $\frac{12\pi}{25}$;
- FACE is an upsampled version of a Facebook user ID dataset [2] with a local skewness of $\frac{99\pi}{200}$.

2) Workloads: Following other baseline learned indexes, we use two types of workloads:

- Read-only workloads: We bulk load 50M, 100M, 150M and 200M keys, execute point queries, and report the average query latency and index size.
- Mixed workloads: we test the update performance with varying read-write and insert-delete ratios. For instance, for read-write workloads, we interleave operations to simulate real-time usage. Specifically, for a workload with a read-write ratio of 0.2, we perform 8 reads followed by 1 insertion and 1 deletion, and then repeat this cycle.

3) Parameters: The hyper-parameters used in this paper are shown in Table IV.

TABLE IV
PARAMETERS AND THEIR SETTINGS

Parameters	Settings
TSMDP bucket size b_T	256
DARE bucket size b_D	16384
TSMDP action space $\{\xi_0, \xi_1, \cdots, \xi_n\}$	$\{2^0, 2^0, \cdots, 2^{10}\}$
The coefficients w_t and w_m	0.5, 0.5
DARE's parameter matrix $\|M\|$	1 × 256
Discount factor γ	0.9
Learning rate η	10^{-4}
Exploration termination probability ϵ	10^{-3}

4) Baselines: We compare with the following baseline indexes, using their published source code and default configurations.

- B+Tree is implemented as the current version of STX B+Tree [48].
- DIC [40] uses RL to construct hybrid indexes.
- RS [9] uses a linear spline to approximate the CDF of the keys and a radix table to index spline points.
- PGM [8] uses piecewise linear functions as the models following an error threshold.
- ALEX [7] is an updatable adaptive learned index. It reserves gaps in data array support efficient insertions.
- LIPP [11] extends the tree structure with models in inner nodes and exact leaf nodes.
- DILI [16] computes fanout via a bottom-up method and then constructs index via a top-down method.
- FINEdex [12] employs independent models, linear interpolation optimization, and non-blocking retraining to enhance query performance.

B. Performance with Read-only Workloads

We report query latency and index size under read-only workloads of varying cardinalities.

1) Scalability: Fig. 8 reports the query latency and index size on four datasets with increasing local skewness. With similar index sizes, **`Chameleon`** achieves more stable query performance across different datasets with different local skewness compared to other indexes. Particularly, on the FACE dataset with largest lsn, the query latency of **`Chameleon`** is 3.82×, 2.08×, 2.62×, 3.75×, 3.51×, 2.10×, 4.28×, and 2.78× lower compared to B+Tree, RS, PGM, ALEX, LIPP,

[1]https://github.com/ai4db-study/Chameleon

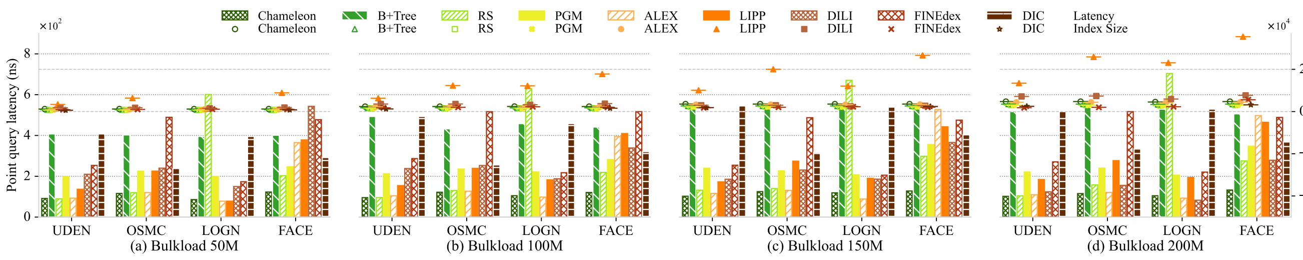


Fig. 8. Comparison of query latency and index size on static workload

DILI, FINEdex, and DIC, respectively. This advantages attributes to the index structure learned by **Chameleon** that better fits the underlying data distribution. **Chameleon** uses an EBH as leaf node to address local skewness and limits hash conflicts based on Theorem 1. On the uniform dataset UDEN, **Chameleon** has similar query performance to RS and ALEX. This is because when the local data distribution is uniform, the effect of EBH is the same as that of linear functions used by existing indexes, without particular advantage - we note that this is not our design optimization target.

2) Effect of Local Skewness: To observe the change in query performance with respect to local skewness, we generate uniform datasets and add locally skewed data around cluster centers with a normal distribution. Using different variance values, we obtain data with varying levels of local skewness. Fig. 9 illustrates the query latency of different indexes relative to that of the B+Tree as local skewness increases. We observe that, as local skewness grows, the query latency of **Chameleon** shows minor change, while that of the other indexes increase notably. This is because **Chameleon** uses EBH to reduce conflicts caused by local skewness and adaptively adjusts the EBH capacity to mitigate skewness, ensuring query performance under different scenarios.

3) Index Construction Time: Fig. 10 shows the Index construction times of different indexes on two real datasets. It can be observed that the index construction time of **Chameleon** is higher than those of most indexes. This is because running the RL agents is time-consuming. DIC and DILI are even slower to construct. DIC is also RL based, it invokes an RL agent for each index node, while the DARE agent in our algorithm reduces the training time by constructing the first h levels in incrementally. DILI needs to first construct a PGM-like index structure from bottom to top, and then construct the final index from top to bottom based on the fanout. Additionally, as the data size increases and the number of nodes grows, the construction time of different indexes increases accordingly.

4) Analysis of Index Structures: To validate the effectiveness of each module proposed in this paper, we evaluate the indexes equipped with different modules: (1) ChaB uses EBH to resolve local skewness (but no TSMDP and DARE), (2) ChaDA – ChaB enhanced with DARE (but no TSMDP), and (3) ChaDATS – ChaDA with TSMDP. Here, due to space limit, we only show DILI and ALEX as they have the best query performance among the baselines as reported in the previous subsections on locally skewed data. Table V summarizes the structures of DILI and ALEX, ChaB, ChaDA, and ChaDATS after bulk loading 200M keys. Since the leaf node of DILI adopts the structure of LIPP accurate prediction, the maximum prediction error (i.e., MaxError) and average prediction error (i.e., AvgError) of its leaf node are 0. It can be observed that even without TSMDP and DARE, the maximum height (i.e., MaxHeight) of ChaB is significantly lower than that of ALEX and DILI on datasets with higher local skewness, while the average height (i.e., AvgHeight) is similar to ALEX across all datasets. This is because DILI always splits downward to handle the local skewness, resulting in a higher tree height. ALEX leaf nodes cannot handle the local skewness using the linear regression model, so as the local skewness increases, ALEX's tree height also increases. **Chameleon** uses EBH to effectively solve the local skewness and achieve a lower tree height. The MaxError and AvgError of ChaB are up to two orders of magnitude smaller than those of ALEX. This is because as the local skewness increases, the number of data conflicts in the slots of ALEX leaf nodes increases, leading to a larger MaxError and AvgError.

Additionally, with the increase of modules, both the num-

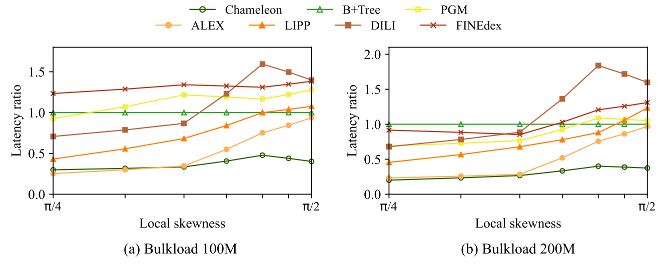


Fig. 9. Latency ratio varying local skewness

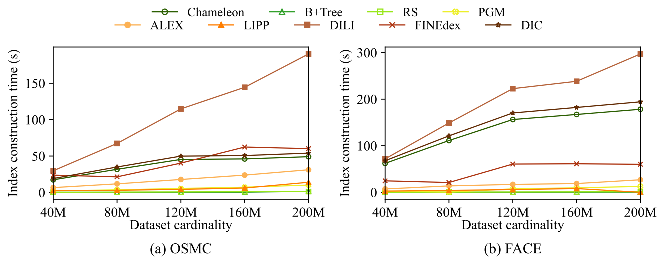


Fig. 10. Index construction time

TABLE V
Analysis of Index Structures

Dataset	Index	MaxHeight	MaxError	AvgHeight	AvgError	#Nodes
UDEN	DILI	7	0	3.1	0	30899734
	ALEX	3	2048	3	7.8	774
	ChaB	3	237	3	1.18	3030001
	ChaDA	3	203	3	0.56	98865
	ChaDATS	3	172	3	0.55	101068
OSMC	DILI	8	0	3.1	0	31170708
	ALEX	4	2048	2.14	5.91	33232
	ChaB	3	95	3	0.71	3030001
	ChaDA	3	84	3	0.59	97695
	ChaDATS	4	37	3	0.53	9935
LOGN	DILI	6	0	3	0	328290
	ALEX	7	16384	2.55	407.03	2098304
	ChaB	3	237	3	0.56	3030001
	ChaDA	3	211	3	0.52	101213
	ChaDATS	4	211	3.01	0.49	97774
FACE	DILI	15	0	4.3	0	68041530
	ALEX	7	16384	3.23	385.72	611316
	ChaB	3	237	3	0.58	3030001
	ChaDA	3	203	3	0.57	95002
	ChaDATS	4	152	3	0.54	98445

ber of nodes (i.e., #Nodes), MaxError, and AvgError of the index decrease, demonstrating the effectiveness of DARE and TSMDP. These observations demonstrate the efficiency and robustness of **Chameleon** across different data distributions, while validating its ability to handle local skewness.

C. Performance with Mixed Workloads

For mixed workloads, we initialize 40M keys and record the throughput as the dataset cardinality grows. DIC and RS are designed for static workloads, and they are highly ineffective in handling dynamic updates. Therefore, we do not include them for the following experiments.

1) Effect of Different Read/Write Ratios: Fig. 11 reports the throughput under different insertion ratios (#writes / (#reads + #writes)). **Chameleon** has the highest throughput on all datasets, and its performance is not impacted by the insertion ratio. For example, on the FACE dataset, **Chameleon** achieves 2.36×, 2.94×, 2.08×, 2.08×, 1.91× and 3.46× higher throughput than those of B+Tree, PGM, ALEX, LIPP, DILI, and FINEdex. On the OSMC and UDEN datasets, the throughput of **Chameleon** are closer to those of ALEX. For these two datasets, they have local distributions close to uniform distributions, making further partitioning impractical, which is not a setting for which **Chameleon** is optimized for. Still, with the increase in insertion ratio, the local skewness of OSMC and UDEN increases, and the throughput of **Chameleon** increases accordingly. This further validates the ability of **Chameleon** to handle dynamic local skewness.

2) Effect of Different Update Ratio: Fig. 12 reports the throughput under varying update ratios (#insertions / (#insertions + #deletions)). There is some improvement in the performance of **Chameleon** and ALEX as the update ratio increases from 0 to 0.25. This is because deletions may lead to

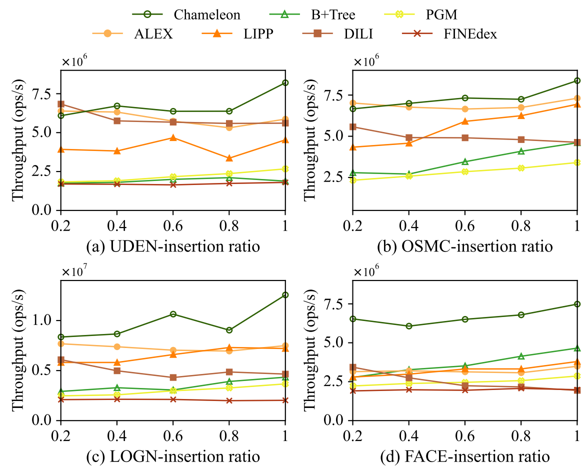


Fig. 11. Throughput comparisons varying read-write ratios

gaps that are just enough to hold a portion of the inserted data, thereby avoiding frequent rebuilding of the index structure. However, as the update rate continues to increase, performance gradually decreases. This is because the updated data distribution starts to deviates from the learned distribution, and the skewness of the data distribution increases. Importantly, other index structures are significantly affected by the changing local skewness of data distribution, while **Chameleon** is more adaptable under mixed workloads.

3) Scalability: Fig. 13 presents point query and insertion (and deletion) latency while batch inserting (deleting) keys of various sizes. To assess the stability of query performance after data updates, we use batched workloads to simulate continuous dense arrival of data updates. The batched workloads involve inserting 1/4 of the key first, followed by executing point queries, and this process is repeated until all keys are inserted. Then, it begins deleting 1/4 of the key, followed by executing point queries, and this process is repeated until all keys are deleted. It can be observed that **Chameleon** maintains a

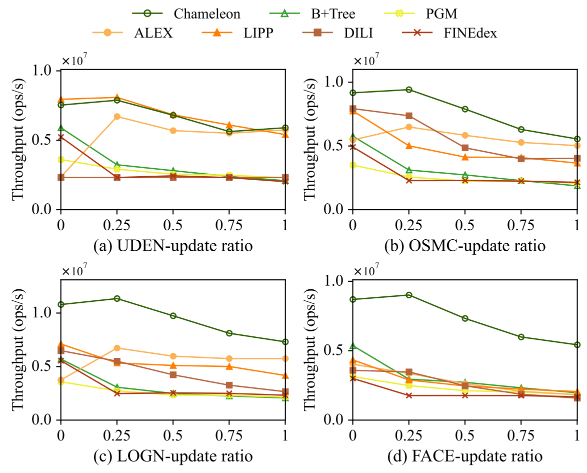


Fig. 12. Throughput comparisons varying insert-delete ratios

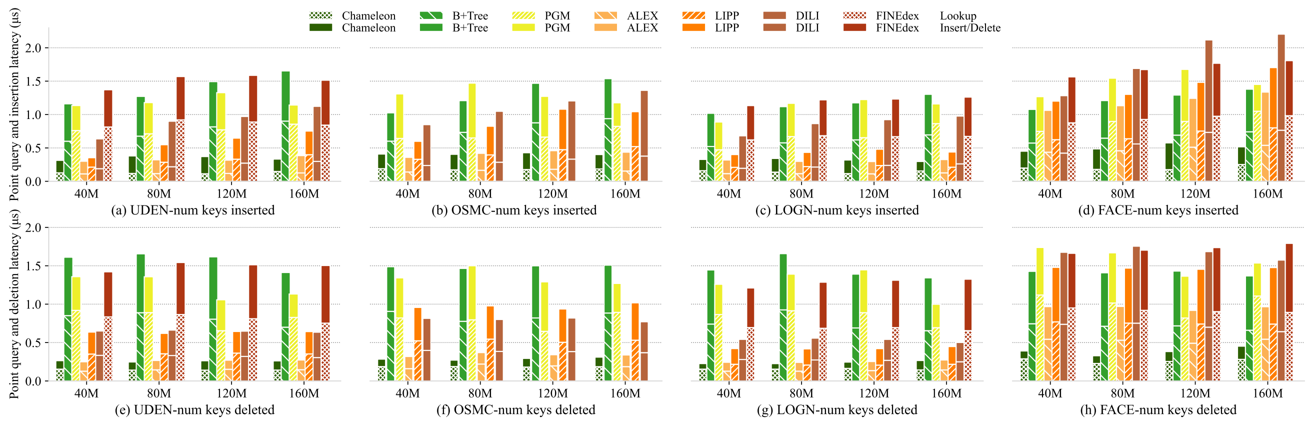


Fig. 13. The indexes scalability (read and write latency) comparisons on batched workloads

consistently more stable average point query, insertion and deletion latency compared to the other indexes and is less sensitive to the frequency of data updates. This is because **Chameleon** has a retraining thread that keeps adjusting the index structure. Moreover, on FACE dataset, **Chameleon** achieves significantly lower query latency compared to the other indexes. This is because FACE exhibits pronounced local skewness, as discussed earlier.

4) Retraining Time: Fig. 14 illustrates the average insertion time and average retraining time within it after bulk loading 20M and inserting 180M keys for different indexes. It can be observed that **Chameleon** achieves the lowest insertion and retraining times across all datasets. This attributes to the scalability of our index structure built with MARL. Additionally, the unordered EBH eliminates sorting operations during retraining, further reducing the retraining overhead.

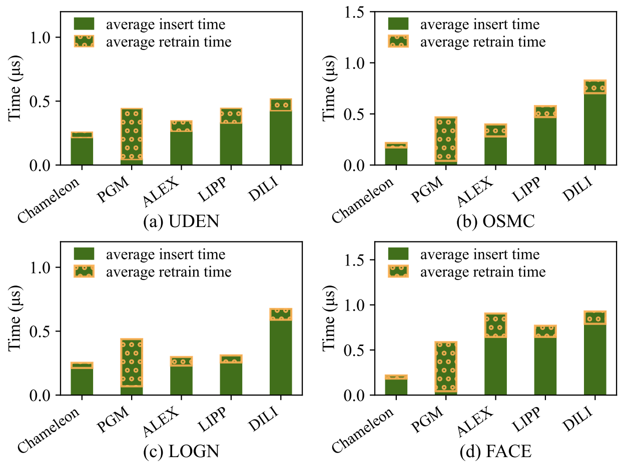


Fig. 14. Retraining time comparison and analysis of various indexes

5) Impact of the Retraining Thread: Fig. 15 shows the impact of the retraining thread on index performance. The retraining is executed every 10s. It can be observed that, compared to not using a retraining thread, **Chameleon** with a retraining thread achieves an average query latency reduction of approximately 100ns, for the following reasons. Firstly, the retraining thread continually adjusts the local structure and maintains a relatively stable leaf node density as the number of insertions increases. Secondly, the retraining process for non-blocking queries avoids a significant number of online adjustments. Thirdly, our interval locking minimizes the wait time of the retraining and queries.

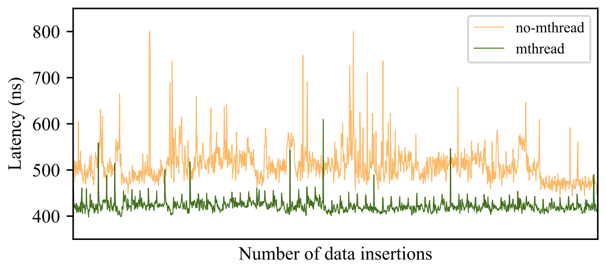


Fig. 15. Query latency comparison with and without multi-threads

VII. Conclusions

We propose a dynamic learned index for skewed data based on multi-agent reinforcement learning (MARL) named **Chameleon**, which combines tree-based learned index and hashing to form an adaptive structure. A MARL algorithm is used to construct of two substructures in a coordinated manner, which builds an overall highly optimized index structure. A non-blocking retraining thread is further employed to adjust the index structure to accommodate frequent data updates. **Chameleon** is evaluated on both real and synthetic datasets. Extensive experimental results verify that **Chameleon** excels in both query time efficiency and scalability when faced with dynamic and locally skewed data distributions.

Acknowledgement. This work is supported by the National Natural Science Foundation of China (U23B2019, 62072083) and the Fundamental Research Funds of the Central Universities (N2216017). Jianzhong Qi is partially supported by Australian Research Council (ARC) Discovery Project DP230101534.

References

[1] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognit. Lett.*, vol. 27, no. 8, pp. 861–874, 2006.

[2] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "SOSD: A benchmark for learned indexes," *NeurIPS Workshop*, 2019.

[3] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska, "Benchmarking learned indexes," *Proc. VLDB Endow.*, vol. 14, no. 1, pp. 1–13, 2020.

[4] Z. Sun, X. Zhou, and G. Li, "Learned index: A comprehensive experimental evaluation," *Proc. VLDB Endow.*, vol. 16, no. 8, pp. 1992–2004, 2023.

[5] D. Comer, "The ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.

[6] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *SIGMOD*, 2018, pp. 489–504.

[7] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska, "ALEX: An updatable adaptive learned index," in *SIGMOD*, 2020, pp. 969–984.

[8] P. Ferragina and G. Vinciguerra, "The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1162–1175, 2020.

[9] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "RadixSpline: A single-pass learned index," in *aiDM@SIGMOD*, 2020, pp. 5:1–5:5.

[10] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "FITing-Tree: A data-aware index structure," in *SIGMOD*, 2019, pp. 1189–1206.

[11] J. Wu, Y. Zhang, S. Chen, Y. Chen, J. Wang, and C. Xing, "Updatable learned index with precise positions," *VLDBJ*, vol. 14, no. 8, pp. 1276–1288, 2021.

[12] P. Li, Y. Hua, J. Jia, and P. Zuo, "Finedex: A fine-grained learned index scheme for scalable and concurrent memory systems," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 321–334, 2021.

[13] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang, "The RLR-Tree: A reinforcement learning based R-Tree for spatial data," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 63:1–63:26, 2023.

[14] H. Y. Dalsu Choi, H. Lee, and Y. D. Chung, "Waffle: In-memory grid index for moving objects with reinforcement learning-based configuration tuning system," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2375–2388, 2022.

[15] Y. Sheng, X. Cao, Y. Fang, K. Zhao, J. Qi, G. Cong, and W. Zhang, "WISK: A workload-aware learned index for spatial keyword queries," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 187:1–187:27, 2023.

[16] P. Li, H. Lu, R. Zhu, B. Ding, L. Yang, and G. Pan, "DILI: A distribution-driven learned index," *Proc. VLDB Endow.*, vol. 16, no. 9, pp. 2212–2224, 2023.

[17] M. Mishra and R. Singhal, "RUSLI: Real-time updatable spline learned index," in *aiDM*, 2021, pp. 1–8.

[18] M. Stoian, A. Kipf, R. Marcus, and T. Kraska, "PLEX: Towards practical learned indexing," in *PMLR*, vol. 229, 2023, pp. 2624–2641.

[19] J. Zhang and Y. Gao, "CARMI: A cache-aware learned index with a cost-based construction algorithm," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2679–2691, 2022.

[20] Y. Li, D. Chen, B. Ding, K. Zeng, and J. Zhou, "A pluggable learned index method via sampling and gap insertion," *CoRR*, vol. abs/2101.00808, 2021.

[21] X. Zhong, Y. Zhang, Y. Chen, C. Li, and C. Xing, "Learned index on GPU," in *ICDE*, 2022, pp. 117–122.

[22] H. Lan, Z. Bao, J. S. Culpepper, and R. Borovica-Gajic, "Updatable learned indexes meet disk-resident DBMS - from evaluations to design choices," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 139:1–139:22, 2023.

[23] G. Yang, L. Liang, A. Hadian, and T. Heinis, "FLIRT: A fast learned index for rolling time frames," in *EDBT*, 2023, pp. 234–246.

[24] T. Yu, G. Liu, A. Liu, Z. Li, and L. Zhao, "LIFOSS: a learned index scheme for streaming scenarios," *World Wide Web*, vol. 26, no. 1, pp. 501–518, 2023.

[25] D. Chen, W. Li, Y. Li, B. Ding, K. Zeng, D. Lian, and J. Zhou, "Learned index with dynamic ϵ," in *ICLR*, 2023.

[26] H. Hang and J. Sun, "Enhancing online index tuning with a learned tuning diagnostic," in *DEXA*, vol. 14146, 2023, pp. 197–212.

[27] M. Matczak and T. Czochanski, "Intelligent index tuning using reinforcement learning," in *ADBIS*, vol. 1850, 2023, pp. 523–534.

[28] Z. Li, T. N. Chan, M. L. Yiu, and C. S. Jensen, "PolyFit: Polynomial-based indexing approach for fast approximate range aggregate queries," in *EDBT*, 2021, pp. 241–252.

[29] G. Liu, J. Qi, L. Kulik, K. Soga, R. Borovica-Gajic, and B. I. P. Rubinstein, "Efficient index learning via model reuse and fine-tuning," in *ICDEW*, 2023, pp. 60–66.

[30] S. Wu, Y. Cui, J. Yu, X. Sun, T. Kuo, and C. J. Xue, "NFL: Robust learned index via distribution transformation," *Proc. VLDB Endow.*, vol. 15, no. 10, pp. 2188–2200, 2022.

[31] H. Lan, Z. Bao, J. S. Culpepper, and R. Borovica-Gajic, "Updatable learned indexes meet disk-resident DBMS - from evaluations to design choices," *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 139:1–139:22, 2023.

[32] J. Ge, H. Zhang, B. Shi, Y. Luo, Y. Guo, Y. Chai, Y. Chen, and A. Pan, "SALI: A scalable adaptive learned index framework based on probability models," *Proc. ACM Manag. Data*, vol. 1, no. 4, pp. 258:1–258:25, 2023.

[33] Y. Tong, J. Liu, H. Wang, K. Zhou, R. He, Q. Zhang, and C. Wang, "Sieve: A learned data-skipping index for data analytics," *Proc. VLDB Endow.*, vol. 16, no. 11, pp. 3214–3226, 2023.

[34] M. Eppert, P. Fent, and T. Neumann, "A tailored regression for learned indexes: Logarithmic error regression," in *aiDM*, 2021, pp. 9–15.

[35] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *AAAI*, 2016, pp. 2094–2100.

[36] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. A. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, vol. 32, 2014, pp. 387–395.

[37] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *ICLR*, 2016.

[38] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, 2015, pp. 448–456.

[39] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," in *NIPS*, 2016, p. 901.

[40] Y. L. Sai Wu, H. Zhu, J. Zhao, and G. Chen, "Dynamic index construction with deep reinforcement learning," *Data Sci. Eng.*, vol. 7, no. 2, pp. 87–101, 2022.

[41] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 3004–3017, 2022.

[42] J. Ge, B. Shi, Y. Chai, Y. Luo, Y. Guo, Y. He, and Y. Chai, "Cutting learned index into pieces: An in-depth inquiry into updatable learned indexes," in *ICDE*, 2023, pp. 315–327.

[43] N. Guo, Y. Wang, H. Jiang, X. Xia, and Y. Gu, "TALI: An update-distribution-aware learned index for social media data," *Mathematics*, vol. 10, no. 23, 2022.

[44] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, "Xindex: A scalable learned index for multicore data storage," in *PPoPP*, 2020, pp. 308–320.

[45] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[46] Y. Achbany, F. Fouss, L. Yen, A. Pirotte, and M. Saerens, "Tuning continual exploration in reinforcement learning: An optimality property of the boltzmann strategy," *Neurocomputing*, vol. 71, no. 13-15, pp. 2507–2520, 2008.

[47] L. Fenaux, T. Humphries, and F. Kerschbaum, "Gaggle: Genetic algorithms on the GPU using pytorch," in *GECCO*, 2023, pp. 2358–2361.

[48] T. Bingmann, "Stx b+ tree," https://panthema.net/2007/stx-btree/.