ShapeShifter: Workload-Aware Adaptive Evolving Index Structures Based on Learned Models

ABSTRACT

In applications such as data management and Web search engines, indexes are key to enabling efficient data retrieval. We find that unlike standard benchmarks with uniform data distribution, index operations in real-world tasks often exhibit strong skewness. However, existing high-performance learned indexes, while proposed to enhance query and update efficiency, often fail to account for the characteristics of skewed workload access, leading to an imbalanced focus on optimizing a single performance metric at the expense of other critical aspects of overall index performance. Furthermore, the complete use of learned models in index structures can lead to increased robustness issues, making them highly vulnerable to attacks and resulting in system unavailability.

To address these challenges, we propose ShapeShifter, an adaptive evolutionary structure based on traditional indexes, capable of dynamically adjusting node structures according to the workload. ShapeShifter introduces a node evolution strategy, designed with workload-skew-aware policies, to adaptively adjust and optimize the most suitable partial index structure, leveraging a hybrid mechanism that combines traditional and learned structures for robust performance and an optimal time-space trade-off under skewed workloads and extreme data conditions. The evaluation results show that ShapeShifter achieves the optimal trade-off between performance and space efficiency while maintaining robustness.

KEYWORDS

index structure, learned model, evolving strategy, data management, web search engine

1 INTRODUCTION

In the fields of data management and Web search engines, indexes play a crucial role, enabling efficient data retrieval. For instance, in search engine technology, an efficient index structure can swiftly filter out the most relevant information from billions of Web pages. In response to the continuous growth in data volume and retrieval demands, the academic community has recently proposed some learned index architectures[9, 14, 19, 20, 25, 43]. The aim of this architecture is to minimize space usage[20] or maximize the efficiency of processing queries and update tasks[9, 43]. Research has shown that its experimental performance substantially exceeds that of traditional tree-based indexes in certain specific dimensions[13, 42].

The "RUM Conjecture"[2] posits that no data structure can achieve perfect optimization among reads, updates, and space overhead. However, unlike standard uniformly distributed data benchmarks such as TPC-H, index operations in scenarios frequently demonstrate significant skew, especially in processing real-time hot information like Web search and retrieval. In scenarios with skewed workloads, the constraints of the "RUM Conjecture" may be mitigated, allowing designers to compress cold nodes in an index while fine-tuning the data structure through adapting to the specific cumulative distribution function (CDF) of hot nodes, thereby enhancing read-write performance while reducing space overhead.

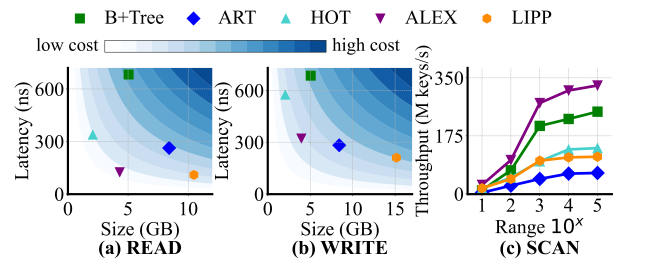


Figure 1: (a) (b) illustrate the trade-off between point search and insert performance and index size under skewed conditions on the LIBIO dataset (200M)[28]. The point search workload follows a Zipfian[7] distribution, with a write skew affecting 30% of the key range. (c) presents the index performance under varying scan numbers.

Consequently, we argue that **designing an index structure with workload-aware and adaptive adjustment capabilities not only improves system performance but also maintains stability** in dynamically changing data access environments.

However, we observe that **numerous contemporary learned index structures are developed with a predominant emphasis on enhancing a specific performance metric, frequently to the detriment of other vital performance dimensions.** For instance, while some structures achieve near-optimal space utilization in theory, they significantly compromise read performance or update capabilities. Early learned indexes, i.e., RMI[20] and RS[19], sacrifice update functionality in pursuit of high space efficiency and read performance. Similarly, the PGM-index[11] trades off read performance for improved space efficiency and write performance. Furthermore, most database management systems prioritize read and write performance in their index structure designs, which often results in higher space overhead. For example, the LIPP[43], while enhancing read and write performance, severely compromises space and scan efficiency, rendering its trade-off unacceptable. The ALEX[9] attempts to strike a balance between read, write, and space overhead, seemingly achieving a more optimal trade-off. However, its reliance on a purely learned model makes it vulnerable to poisoning attacks[45], leading to structural failures. This issue stems from the fact that **the complexity of the CDF of real-world data far exceeds the approximation capabilities of learned models**, making it difficult to ensure the robustness of index structures when balancing optimizations for read, write, and space overhead.

To address the aforementioned challenges, we prioritize adaptive adjustment capabilities as "first-class citizens" and design a workload-aware, adaptive evolving index framework ShapeShifter. This framework can flexibly self-adjust its form to accommodate varying skewed workloads and the characteristics of CDFs. Initially, ShapeShifter is based on a traditional tree-like index, selected for its robustness to meet the complex CDF challenges found in real scenarios. Based on this foundation, ShapeShifter develops a node evolving strategy capable of sensing the degree of workload skew and adaptively selecting the most appropriate traditional/learned structure, progressively optimizing hot nodes to enhance system performance. By employing a hybrid approach that combines traditional and learned structures, ShapeShifter notably enhances performance while ensuring robustness. Concurrently, ShapeShifter compresses cold node to achieve an optimal balance between performance and space. Additionally, the node evolving strategy of ShapeShifter designs dynamic learned structures tailored to different workloads for hot nodes, allowing them to self-adjust based on workload changes, thus maintaining superior performance. Ultimately, ShapeShifter also develops a lightweight node hot-cold sensing model that allows to adaptively and precisely detect local structural "temperature changes" without significantly sacrificing performance, guiding itself timely evolving.

Our microbenchmark with real-world datasets demonstrates that, under skewed workloads, ShapeShifter achieves up to a 6.34× increase in throughput and a 51.79% reduction in space usage compared to the state-of-the-art (SOTA) traditional tree-based indexes. When compared to the SOTA learned indexes, ShapeShifter shows up to a 1.26× improvement in throughput and an 824.19% reduction in space usage, while also effectively mitigating poisoning attacks. These results provide compelling evidence that ShapeShifter achieves an optimal balance between space and performance.

This paper makes three primary contributions: **(1)** we introduce ShapeShifter, a workload-aware adaptive index framework based on learned models, designed to achieve an optimal trade-off between performance and space utilization while ensuring robustness of the index. **(2)** this study integrates traditional and learned models for the first time, defining a series of node evolution strategies. These strategies enable the index to adapt to various skewed workload, effectively balancing the robustness of traditional models with the high performance characteristics of learned models, thereby achieving an effective trade-off between the two. **(3)** we design a lightweight model for determining hot/cold nodes, which identifies the optimal timing for triggering index evolving without compromising performance.

The rest of the paper is organized as follows. Section 2 discusses existing index limitations and the objectives of this paper. Section 3 outlines the architecture of ShapeShifter, framework evolution, and node classification algorithm. Section 4 shows experimental results from real-world datasets and skewed workloads. Section 5 reviews related work, and Section 6 concludes the paper.

2 INDEX FRAMEWORK INVESTIGATION

In real-world tasks, index operations demonstrate significant skewness across multiple dimensions, including query patterns, key value ranges, and the structure of the accessed index space [6]. Thus, in environments with highly skewed workloads, designers can compress cold nodes and adaptively adjust the index structure based on the data distribution of hot nodes, enhancing performance and reducing space overhead [1].

Observation 1: In workload-skewed scenarios, designing index structures with adaptive capabilities is essential to achieve a more optimal trade-off between time and space.

Figure 1 (a) and (b) display the performance of various index structures in point queries/insertion and space utilization under skewed conditions. We observe that the time-space trade-off for B+Tree[4] is primarily limited due to poor performance in point queries and insertions, while the ART[24] index suffers due to its high space overhead. The learned index LIPP, which exhibits the best performance in point queries and insertions, also incurs unacceptably high costs when handling large-scale data due to its substantial space requirements.

	Time-Space Tradeoff			Scan	Poisoning Attacks
	Point Search	Insert	Size		
B+Tree[4]	×	×	✓	✓	✓
ART[24]	✓	✓	×	×	✓
HOT[5]	✓	×	✓	×	✓
ALEX[9]	✓	✓	✓	✓	×
LIPP[43]	✓	✓	×	×	✓
ShapeShifter	✓	✓	✓	✓	✓

Table 1: Aspects of challenges in SOTA index structures.

Observation 2: Index structural design frequently prioritizes a single performance metric at the detriment of others. Furthermore, existing indexes fail to adequately consider workload-skewed accesses, hindering the achievement of an optimal balance and enhancement of overall performance.

Although Figure 1 (a) and (b) appear to show the ALEX exhibiting a favorable time-space trade-off, experiments by Yang et al.[45] reveal and highlight that since the structure of the ALEX index is entirely based on a learned model, it is susceptible to poisoning attacks, which can lead to structural collapse.

Observation 3: In practical applications, the complexity of the CDF of stored data surpasses the approximation abilities of current learned models. As a result, when indexes depend solely on learned models to balance time and space optimally, ensuring their robustness becomes challenging.

Figure 1 (c) illustrates the scan performance of various indexes. Notably, some indexes, such as HOT, exhibit generally poor performance. Considering the frequency of scan operations in real-world applications, such inadequate performance can significantly diminish the practical utility of indexes in databases.

Observation 4: Some indexes exhibit a favorable trade-off between time-space. However, its scan performance is poor.

In summary, our analysis in Table 1 highlights the limitations of current SOTA index structures. We propose that more effective indexes should incorporate workload-sensitive adaptive features to manage workload skewness and optimize the balance between time and space, rather than focusing solely on maximizing performance in one aspect while neglecting others. Additionally, the current piecewise linear models used in learned indexes have limited fitting capabilities, making it difficult for indexes that depend entirely on these models to maintain robustness and achieve an optimal time-space trade-off. Effective index designs should ensure enhanced scan performance to support prevalent range queries in databases; thus, our research excludes index structures that lack this capability.

3 THE SHAPESHIFTER DESIGN

This section provides a detailed introduction to the ShapeShifter, which aims to address the key issues discussed in Sections 1 and 2. By ensuring the robustness of the index, ShapeShifter incorporates learned models to assist in dynamic evolving operations of the index structure, thereby improving performance and reducing space overhead.

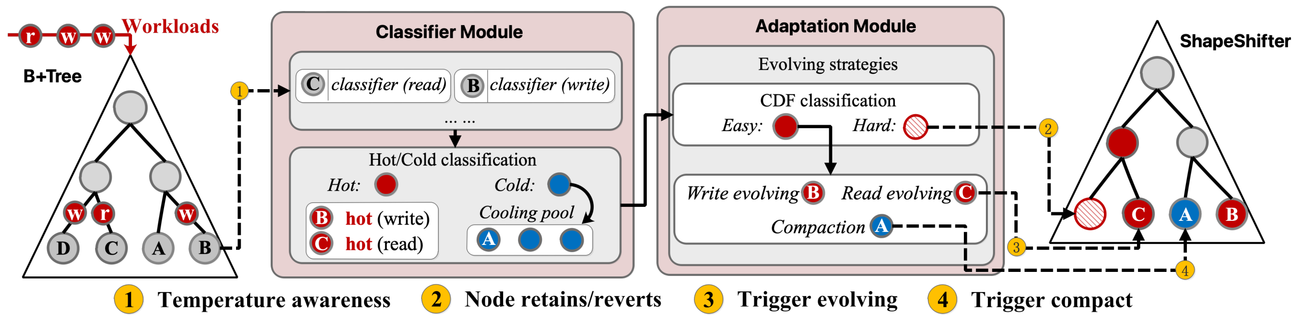


Figure 2: The structure of ShapeShifter.

In particular, Section 3.1 presents an overview of the ShapeShifter architecture and describes the different operational processes within the framework. In Section 3.2, we propose a strategy framework based on adaptive evolving operation, which further enhances the performance and robustness of the index under skewed workloads with the assistance of learned models. Finally, Section 3.3 introduces a hot-and-cold node classification framework that maintains statistical information for different node roles at minimal cost, thereby optimizing the management of the index.

3.1 Overview

3.1.1 The structure of ShapeShifter. The ShapeShifter framework, as illustrated in Figure 2, consists of two main modules: the classifier module and the adaptation module. In the first phase, during each lookup or insertion operation, ShapeShifter utilizes the classifier within the classification module to determine the "temperature state" ①, i.e., hot or cold, of the target node. In the second phase, if a node is classified as either hot ② ③ or cold ④, an evolving operation is triggered; otherwise, the structure remains unchanged.

Specifically, ShapeShifter evaluates whether the CDF of the keys stored in hot nodes is conducive to adopting a high-performance learned model in lieu of traditional index structures. If the learned model can provide stable performance improvements, the node evolves into a learned model ③; otherwise, it retains the traditional structure ② to ensure system robustness. For nodes that have evolved into learned structures, if insertion operations render their CDFs difficult to approximate effectively or even unusable, ShapeShifter provides a mechanism to revert to the traditional structure ②, ensuring system stability. Note that our evolving strategy encompasses the functionality of traditional structure modification operations (SMOs).

3.1.2 Operations of ShapeShifter.

(1) Lookup operation. Algorithm 1 delineates the query process for ShapeShifter. Due to the evolutionary adaptation of certain nodes within ShapeShifter into learned model structures, it is imperative to first ascertain the type of the current node during a query. In detail, the process begins with locating the leaf node corresponding to the target key (lines 1-11). When querying an internal node, if the node is a learned model, the model predicts the target position, followed by comparison operations to determine the precise location (lines 3-5). If the internal node is a B+Tree node, a binary search algorithm is employed to locate the target position (lines 6-8). Upon locating the leaf node where the target key resides, it is then necessary to ascertain its exact position within the leaf node (lines 14-23). If the leaf node incorporates a learned model, the target position is predicted and a last mile search is conducted to find the precise location (lines 14-16). If the leaf node adheres to a B+Tree structure, the exact position is located through binary search (lines 17-19). If the retrieved key does not match the target key, the query fails; otherwise, the query succeeds (lines 20-23).

Algorithm 1: ShapeShifter Lookup Operation

```
   Input: $key$
   Output: target$\langle key, value\rangle$
1  Function Traverse_To_Leaf($key$):
2      for $Node.type \neq Leaf$ do
3          if $Inner\_node.model$ is $linear$ then
4              $predicted\_slot \leftarrow Node.linear(key)$
5              $precise\_slot \leftarrow Range\_Compare(predicted\_slot)$
6          else if $Inner\_node.model$ is $B+Tree$ then
7              $precise\_slot \leftarrow Node.binary\_search(key)$
8          end
9          $Sub\_node \leftarrow Inner\_node[precise\_slot]$
10     end
11     return $Leaf\_node$
12 Function Lookup_Operation($key$):
13     $Leaf\_node \leftarrow$ Traverse_To_Leaf($key$)
14     if $Leaf\_node.model$ is $linear$ then
15         $predicted\_slot \leftarrow Leaf\_node.linear(key)$
16         $precise\_slot \leftarrow Last\_Mile(predicted\_slot)$
17     else if $Leaf\_node.model$ is $B+Tree$ then
18         $precise\_slot \leftarrow Leaf\_node.binary\_search(key)$
19     end
20     if $Leaf\_node[precise\_slot].key \neq target\ key$ then
21         return $\langle Null, precise\_slot\rangle$
22     end
23     return target$\langle key, value\rangle$
```

(2) Insert operation. Before inserting a new key, ShapeShifter uses the *Travel_To_Leaf* function in Algorithm 2 to locate the leaf node where the new key will be inserted and to determine its type (lines 2-3, 15). If the target leaf node is a learned model (i.e., a linear model), the target key-value pair can be calculated (lines 4-6). If the target key is Null, the new key-value pair is directly inserted (lines 7-8). Else, it must be determined whether the target key matches the new key. If they match, the target value is directly updated (lines 9-10). Otherwise, insertion conflicts must be resolved (by shifting or chaining), and then insert the new key-value pair (lines 11-14). If the target is a B+Tree node, the key-value pair is inserted or updated using the B+Tree method (lines 15-17).

(3) Other operations. For the algorithmic description of the evolving (SMO), please refer to Section 3.2. The descriptions of the Scan and Delete algorithms can be found in Appendix A.1 and A.2.

Algorithm 2: ShapeShifter Insert Operation

```
Input: $\langle key, value \rangle$
Function Insert_Operation($key$):
    $Leaf\_node \leftarrow$ Traverse_To_Leaf($key$)
    if $Leaf\_node.model$ is $linear$ then
        $predicted\_slot \leftarrow Leaf\_node.linear(key)$
        $precise\_slot \leftarrow Last\_Mile(predicted\_slot)$
        tar$\langle key, value \rangle \leftarrow Leaf\_node[precise\_slot].data$
        if $tar\ key = Null$ then
            insert $\langle key, value \rangle$ to precise_slot
        else if $tar\ key = key$ then
            $Update$(tar $value \leftarrow value$)
        else if $tar\ key \neq key$ then
            $new\_slot \leftarrow Shift\_Method\ or\ Chain\_Method$
            insert $\langle key, value \rangle$ to new_slot
        end
    else if $Leaf\_node.model$ is $B+Tree$ then
        insert $\langle key, value \rangle$ uses B+ methods
    end
```

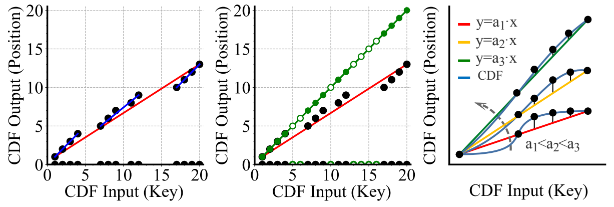
3.2 Evolving Strategies

This section is dedicated to a detailed exposition of the evolving strategies of ShapeShifter. Section 3.2.1 delves into the design of the learned models employed by ShapeShifter. Section 3.2.2 thoroughly explains the evolutionary strategies for hot nodes, where different types of hot nodes evolve into distinct learned models tailored to their specific demands, thereby optimizing the performance of ShapeShifter. Section 3.2.3 discusses the evolving strategy for cold nodes, with the main aim of reducing space overhead.

3.2.1 Learned model. Existing learned structures in indexes primarily utilize piecewise linear functions, as shown in Figure 3 (a). Considering that the distribution of stored keys often exhibits non-linear characteristics (the black dots), a single linear approximation (the red line) can result in significant errors when approximating the CDF of stored keys. In such cases, employing piecewise linear functions (the blue line) to approximate each local key segment can effectively reduce these errors. This method of piecewise linear approximation is what learned indexes initially adopted. However, as the number of segments increases, this method leads to an increased height of the tree. Essentially, this approach represents a trade-off strategy that sacrifices tree height to reduce approximation errors, generally resulting in moderate performance.

Figure 3 (b) illustrates a SOTA method that maintains minimal errors. To be specific, by "stretching" the shape of the cumulative distribution function (CDF) to more closely approximate linear characteristics without segmentation[1], effectively reducing the approximation errors of the linear model. For example, in Figure 3(b), predicting with the red line for $key = 4$ results in errors, whereas predictions using the green line achieve complete accuracy. This accuracy is due to the increased slope of the red line transitioning to the green line, necessitating a recalculation of storage position for the key based on the new green line, which alters the original shape of the CDF of the key (the green dots) to closely resemble linear characteristics. This method reduces prediction errors without the

[1]Figure 3(b) is merely illustrative; actual applications still require segmentation, considerably less than the method shown in Figure 3(a).



(a) Piecewise **(b) Increase slope** **(c) Learned model**

Figure 3: The principles of learned models.

need for segmentation. However, this "stretching" process creates gaps (the green hollow circles), thereby increasing space overhead. Therefore, this approach essentially involves a time-space trade-off strategy. Nevertheless, these gaps provide space for subsequent insertion operations, ensuring that space is not wasted in nodes with frequent insertions (hot).

Intuitively, inspired by Figure 3 (b), ShapeShifter decides to employ a learned structure with gaps as the evolving architecture for its hot nodes. Figure 3 (c) illustrates the extent to which the CDF of local storage keys is "stretched" under linear models with varying slopes. The red line fits the original CDF. The yellow line, with an increased slope, stretches the original CDF to make it more linear and further reduces the prediction error. The green line, with the highest slope, further stretches the original CDF towards linearity, also reducing the prediction error even more. Therefore, the critical task now is how the nodes of ShapeShifter, when evolving into learned models, can select the appropriate slope to balance the trade-off between performance and space. The most suitable approach is for the write hot nodes to adaptively select the appropriate slope based on the insertion rate, reserving appropriate gaps. On the other hand, read hot nodes should not reserve too many gaps to achieve the optimal time-space trade-off. We will discuss this strategy in detail in Section 3.2.2.

During the insertion process within the learned structure, if the targeted insertion position is a gap, direct insertion is feasible. However, if the designated insertion position is already occupied due to an imprecise fit, leading to an insertion conflict (similar to a hash collision), specific strategies must be adopted to resolve this. Currently, there are two main strategies to address such conflicts: one is the shift method[9], which involves rearranging nodes by moving existing keys; the other is the chain method[43], which employs a linked structure to create a new node for the targeted slot, transforming the "last mile" search into a sub-tree traversal. Among these conflict resolution methods, we find that the chain method outperforms the shift method in terms of efficiency and performance, although it incurs a relatively higher spatial overhead. Therefore, within ShapeShifter, for hot insertion nodes, we utilize the chain method to support efficient insertion operations, aiming for superior performance. Conversely, for hot read nodes, the shift method is employed to handle minimal insertions, seeking to minimize space utilization.

3.2.2 Hot node evolving strategies. ShapeShifter categorizes hot nodes into hot read nodes and hot write nodes, each following distinct evolving paths. It is important to emphasize that in practical applications, learned models may fail to fit the CDF of stored data accurately, i.e., resulting in significant errors, or may even fail under extreme conditions. Therefore, learned models should be used with caution in these instances to enhance the robustness of the index. To address these challenges, we define two modes for the learned model in this section: "easy" mode and "hard" mode. In "easy" mode, ShapeShifter evolves into different learned structures based on the specific needs of hot read and hot write nodes. In "hard" mode, to ensure the stability and reliability of the system, ShapeShifter should retain or revert to a traditional B+Tree structure. For details on how to determine the hot or cold status of nodes, please refer to Section 3.3.

The method for determining the hard mode is by multiplying operational performance and space overhead defined by Zhang et al.[47], i.e., $COST = Perf^r \cdot Spac$.

$Perf$ refers to the performance estimate of a learned model or a B+Tree, i.e., the average query latency, which is calculated based on the query time complexity of each node during index construction or node SMO processes. More precisely, we compute the average query cost for the keys stored in a node under both the learned model and the B+Tree structure. $Spac$. denotes the space overhead, while r represents the weight between performance and space overhead. When $r > 1$, performance is prioritized; when $r < 1$, space overhead is given more importance. Let $COST_1$ represent the cost of the learned model, and $COST_2$ represent the cost of the B+Tree. When $COST_1 \geq COST_2$, the system is considered to be in hard mode, during which the node will not evolve into a learned model, and the learned model also reverts to a B+Tree structure.

$$n.size = \beta \times \min(1, \frac{n.insert\_rate_t}{n.insert\_rate_{t-1}}) \times n.slot\_num \tag{1}$$

For nodes identified as operating in "easy" mode during hot insertion processes, ShapeShifter dynamically reserves varying numbers of gaps based on insertion rates during the node splitting phase, as illustrated in Equation 1. This ensures that leaf nodes with higher insertion rates are allocated more space. Here, $insert\_rate_t$ represents the insertion rate at time t. For details on how insertion rates are determined, refer to Section 3.3. The ratio $\frac{n.insert\_rate_t}{n.insert\_rate_{t-1}}$ indicates the growth rate of the insertion rate. The parameter β serves as the space factor for reserving gaps. For the determination of weight values, see Equation 2.

$$\beta = \begin{cases} \gamma, & n.slot\_num \geq 1M \\ 3\gamma, & n.slot\_num \geq 500K \\ 6\gamma, & n.slot\_num < 100K \end{cases} \tag{2}$$

As indicated by Equation 2, nodes of varying sizes should be equipped with corresponding space factors, denoted as β. Equation 1 demonstrates that smaller nodes require larger space factors to ensure adequate space reservation. For example, suppose two hot nodes have slot capacities of 8 and 16, respectively; they would require expansion factors of 6 and 3 to achieve a total expansion of 48 units. The weight value of the space factor β can be dynamically adjusted based on different workloads. In our evaluation, we set the default value of γ to 1.

Within the learned structure, as the number of insertions increases, the quantity of gaps gradually diminishes, requiring an expansion of space to accommodate insertion demands. In ShapeShifter, the classifier employs a probabilistic model to determine when to expand nodes in the learned structure; for specific strategies, please refer to Section 3.3. To mitigate extreme situations, a safeguard mechanism is incorporated into the learned models: when the number of gaps reaches a σ fraction[2] of the total number of slots within the node, a forced automatic node expansion is triggered. The specific scale of this expansion is defined in Equation 1.

For hot read nodes identified as operating in an "easy" mode, during their evolving process, the primary purpose of the reserved gaps is to reduce prediction errors (see Figure 3); thus, an excess of gaps should not be maintained. In ShapeShifter, a spatial expansion parameter[3] δ has been established to determine the extent of space expansion for read-hot nodes, aiming to achieve the optimal trade-off between query performance and space overhead.

Regarding internal nodes evolving, refer to Appendix A.3.

3.2.3 *Cold node evolving strategies.*

If a leaf node is identified as a cold node, it will not evolve into a learned structure but will retain the structure of a B+Tree leaf node. This is because the gaps present within the learned structure increase the space overhead of the index, which does not benefit cold nodes. Additionally, the reserved space (B+Tree space) within nodes identified as cold will also be correspondingly reduced, namely to a fraction σ of the total number of keys within the node.

3.3 Classifier Model

3.3.1 *The method for determining hot write nodes.*

In the index, to determine whether a node qualifies as a "hot write" node, it is necessary not only to assess whether the cumulative number of inserted keys reaches a certain threshold but also to consider the speed of key insertion. Therefore, in ShapeShifter, the insertion rate of a node is used to determine whether it is a hot node, as detailed in Equation 3. Let $n.key\_num_t$ represent the total number of keys in node n at time t. We assume that t and $t-1$ are the timestamps of the current and previous constructions or SMOs of the node. Therefore, the key insertion rate at the current moment t can be estimated by dividing the difference in the number of keys inserted between two timestamps by the time interval.

$$n.insert\_rate_t = \frac{n.key\_num_t - n.key\_num_{t-1}}{n.time_t - n.time_{t-1}} \tag{3}$$

Next, we estimate the number of keys newly inserted from the last SMO to the current moment using the insertion rate $n.insert\_rate_t$. The calculation is expressed as $n.insert\_rate_t \times (n.current\_time - n.time_t)$. By employing Equation 4, we can assess whether the number of newly inserted keys reaches a sufficient level. Here, θ represents the threshold for key growth. If the number of newly inserted keys exceeds the threshold θ, it is considered that there are sufficient keys, and the node is classified as a hot node.

$$n.insert\_rate_t \times (n.current\_time - n.time_t) \geq \theta \tag{4}$$

Based on Equation 1, the expanded insertion space (gap) since the last SMO is calculated as $\beta \times \min(1, \frac{n.insert\_rate_t}{n.insert\_rate_{t-1}}) - 1$. Consequently, we set Equation 5 as the criterion to satisfy θ. Here, ϵ denotes the fill factor, which is the multiple of the reserved space that is to be filled.

$$\theta = \epsilon \times [\beta \times \min(1, \frac{n.insert\_rate_t}{n.insert\_rate_{t-1}}) - 1], \quad (0 < \epsilon \leq 1) \tag{5}$$

[2]In ShapeShifter, σ is set to 0.2.

[3]We set δ to 1.6, i.e., the expansion of space is 1.6 times the original size.

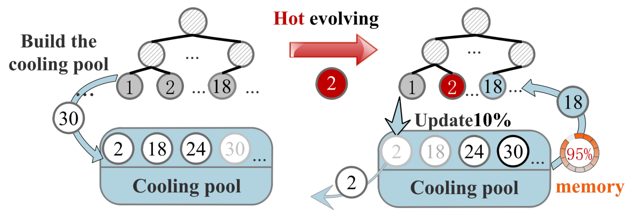


Figure 4: Cold node identification framework.

Substituting Equation 5 into Equation 4 and through equivalent transformations, we derive the probability model P_{write}, as shown in Equation 6. In ShapeShifter, under specific insertion conditions[4], it is necessary to determine whether the P_{write} within a node is triggered. When $P_{write} < 1$, a Bernoulli probability experiment is conducted to decide whether to trigger node evolving for hot nodes. When $P_{write} = 1$, the node is directly classified as a hot node and undergoes evolving. Please refer to Appendix A.4 for details.

$$P_{write} = \frac{n.insert\_rate_t \times (n.current\_time - n.time_t)]}{\epsilon \times [\beta \times \min(1, \frac{n.insert\_rate_t}{n.insert\_rate_{t-1}}) - 1]} \quad (6)$$

3.3.2 The method for determining hot read nodes. We define the probability that a target node is identified as a hot read node due to search operations, denoted as P_{read}. This hyperparameter can be adjusted based on actual conditions. Each time a leaf node is searched, ShapeShifter calculates whether this node meets the triggering criteria for P_{read}. If it does, ShapeShifter then considers this node as a hot read node.

Note that if the most recent evolving operation of a leaf node is triggered by a hot read, it indicates that no evolution of the node has been triggered by insertion operations since then, i.e., the read performance of the node has not noticeably deteriorated, and the number of new keys inserted might be limited. In such cases, we may consider adjusting P_{read} to a smaller value to avoid overly frequent evolving operations. In particular, this adjustment can be made as $P_{read} = P_{read} \times \lambda$, where $\lambda \in (0, 1)$ is a penalty coefficient.

3.3.3 The method for determining cold nodes. We propose a cold node compression evolving strategy aimed at optimizing space utilization under skewed workloads for ShapeShifter. Initially, when creating the ShapeShifter index structure, we also introduce a cooling pool space[23], as shown in Figure 4. During the construction or each evolving operation of ShapeShifter, each leaf node in the index has a 10% probability of being selected into the cooling pool. When a node undergoes an evolving operation, it is removed from the cooling pool. At this stage, nodes remaining in the cooling pool are considered temporary cold nodes.

ShapeShifter supports user customization of compression frequency settings. When the user specifies a compression time, ShapeShifter will check if the size of the index exceeds the user-defined upper limit. Should this condition be met, ShapeShifter will select the earliest added nodes from the cooling pool for compression and deletion, continuing until the index size is reduced to a range acceptable to the user.

[4]In B+Tree, P_{write} is calculated with each insertion. In learned structures, P_{write} is only calculated when insertions conflict; otherwise, due to efficient node insertion performance, node evolution is not necessary.

4 EVALUATION

4.1 Experimental Setup

All experiments are conducted on a two-socket equipped with two 16-core Intel Xeon Silver 4216 @ 2.10GHz CPUs and 503GB of DRAM. We implement ShapeShifter[5] with ~4k LOC of C++.

4.1.1 Baselines. We benchmark ShapeShifter against five baselines. They are all top-performing traditional/learned index structures as evaluated by the GRE[42] benchmarking tool: (1) B+Tree[4], a highly robust tree structure widely used in databases and file systems; (2) ART[24], an efficient adaptive radix tree structure that dynamically adjusts node sizes to optimize memory use and accelerate look-up operations; (3) HOT[5], a height-optimized in-memory index structure that flexibly adjusts node bit widths to reduce trie tree height; (4) ALEX[9], an updateable learned index offering optimal time-space trade-off; (5) LIPP[43], a high-performance learned index utilizing a model-based insert strategy for precise look-up.

4.1.2 Datasets and workloads. We select several real datasets from the GRE benchmarking tool to evaluate the performance of indexes. (1) COVID[29]: Tweet ID with tag COVID-19. (2) LIBIO[28]: Repository ID from libraries.io. (3) GENOME[36]: Pairs of locations on human chromosomes. (4) OSM[3]: OpenStreetMap locations.

We design skewed workloads to evaluate index performance using the aforementioned datasets. Specifically, the point search workload employs a Zipfian distribution, with the skewness controlled by adjusting the parameter α. For instance, WR_1 denotes a scenario where $\alpha = 1$. Please refer to Appendix A.5 for further details. For the insertion workload, skewness is indicated by defining the percentage of the data range in which insertions occur. For example, WW_{10} signifies that insertions are confined to 10% of the data locally. Note that all WW also randomly write 1% of the data without the range of hot nodes. Additionally, all keys are randomized in their order before evaluation.

4.2 Index Performance Evaluation

In Figure 5, we assess six indexes on the COVID/GENOME datasets for average latency and index size across different workload skews, recording space overhead in the WR scenarios for clarity. In read-only scenarios, workload skew ranges from WR_0 to $WR_{1.6}$, with $WR_{1.6}$ being the most skewed; in write-only scenarios, the skew ranges from WW_{100} to WW_1.

In the WR scenario of the COVID dataset, the evolving begins at $WR_{0.2}$ because the uniformity of the queries, i.e., WR_0, prevents hot nodes from being detectable.

At WR_1, ShapeShifter reduces the average query latency by 534.23% and 3.85% compared to B+Tree and SOTA, i.e., ALEX, respectively, and achieves a reduction in index size by 48.15% and 26.13%. When α exceeds 1, the performance of ShapeShifter stabilizes, indicating that the evolution of local hot read nodes is complete. In WW, the evloving starts at WW_{70}. At WW_{30}, ShapeShifter reduces the average query latency by 219.44% and 9.01% compared to B+Tree and SOTA, i.e., LIPP, respectively, while matching B+Tree in index size and achieving a 204.91% reduction compared to LIPP. HOT shows the optimal performance in terms of space overhead.

In the GENOME dataset WR_1, ShapeShifter demonstrates reductions in average query latency by 175.26% and 8.85% compared to

[5]https://anonymous.4open.science/r/Shapeshifter-35F3.

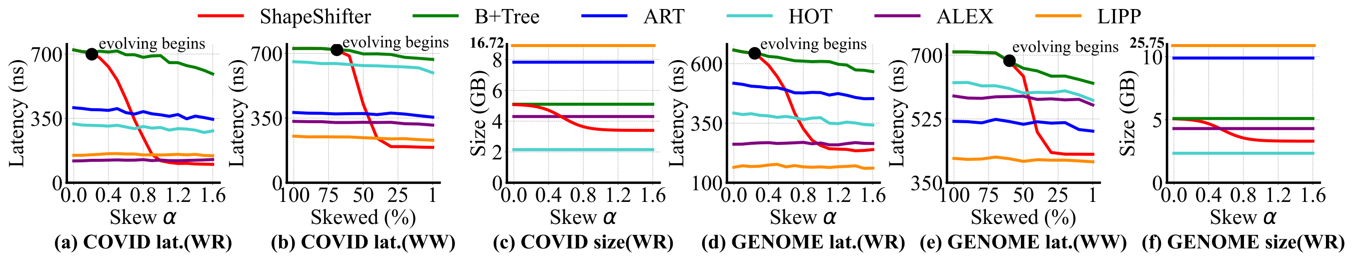


Figure 5: Average latency/space overhead of various indexes on COVID/GENOME under workloads with varying skew degrees. Extended plots are available in Appendix A.6 and A.8.

B+Tree and ALEX, respectively. The index size decreases by 51.79% and 27.98%, respectively. Although the average query latency of ShapeShifter is slightly higher than SOTA, i.e., LIPP, it achieves a significant reduction in index size, with a decrease of 666.96%. In the WW_{30} scenario, ShapeShifter reduces the average latency by 54.67% compared to the B+Tree and matches the performance of SOTA, i.e., LIPP. Concurrently, in terms of index size, ShapeShifter compares favorably with the B+Tree and achieves a substantial reduction of 306.51% compared to LIPP.

Insight 1: As workload skewness increases, ShapeShifter sees more evolved nodes, leading to ongoing decreases in average latency and space overhead, especially in WR$_1$ and WW$_{30}$ and other more skewed scenarios, where it shows superior time-space trade-off. Although ALEX and HOT seem optimal in terms of latency or space, respectively, their balance between time and space requires deeper evaluation.

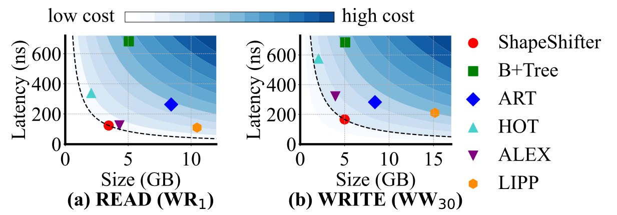


Figure 6: Time-Space trade-off evaluation on the LIBIO dataset. Extended plots are available in Appendix A.7.

4.3 Time-Space Trade-off Evaluation

To better illustrate the trade-off between performance and space, we utilize the $COST = Perf^r \cdot Spac$ function[47], as detailed in Section 3.2.2. Figure 6 illustrates the COST comparison among six indexes under WR_1 and WW_{30} workloads in the LIBIO dataset. Using a blue curve to visualize the scenario where $r = 1$, indicating that space and performance are considered equally important, indexes located on the same curve are deemed "equivalent" in terms of their time-space trade-off. Notably, under both WR_1 and WW_{30} workloads, ShapeShifter consistently achieves the optimal trade-off. Indeed, starting from the more skewed workloads of $WR_{0.9}$ and WW_{40}, ShapeShifter continues to exhibit the best trade-off, with other datasets showing similar performance. Secondly, ALEX and HOT demonstrate the second-best trade-off. Meanwhile, LIPP, which performs best in terms of latency, has a COST comparable to that of the B+Tree, indicating that the design of LIPP fundamentally prioritizes trading space for time performance. This is consistent with the findings of Wongkham et al[42]. Due to space limitations, the relevant evaluation can be found in the Appendix A.9.

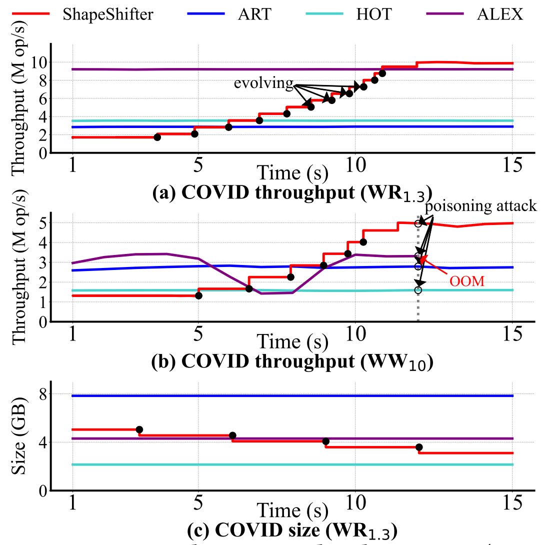


Figure 7: Latency and space overhead over time (COVID).

Insight 2: ShapeShifter demonstrates the optimal performance and space trade-off under skewed workloads. ALEX and HOT also exhibit commendable trade-off.

4.4 Time-Varying Performance Evaluation

In Figure 7, we present the changes in throughput and index size over time for indexes which are similar to ShapeShifter in terms of time-space trade-off. We select more skewed scenarios, $WR_{1.3}$ and WW_{10}, to more vividly demonstrate the performance improvements brought about by a limited number of evolutions. In the $WR_{1.3}$ scenario, as shown in Figure 7 (a), ShapeShifter initiates its first evolving process at the 3.5th second, marking the start of local nodes being identified as hot read nodes. As time progresses, the pace of evolving accelerates, culminating in complete evolving by the 12th second, thereby achieving the highest throughput.

Note that during the initial evolving, the increase in throughput is equal to the gain from evolving minus the cost of evolving, i.e., SMO. For the intermediate evolving, the increase in throughput is the gain from evolving minus the cost of the previous evolution plus the cost of the subsequent one, effectively neutralizing the performance loss due to costs, thus making the performance improvement more apparent than the first. Following the final phase of evolving, there is a marked improvement in performance, attributable to the absence of costs associated with evolving that need to be amortized during this period.

In the WW_{10} scenario, as shown in Figure 7 (b), ShapeShifter undergoes its first evolution at the 5th second. As time progresses,

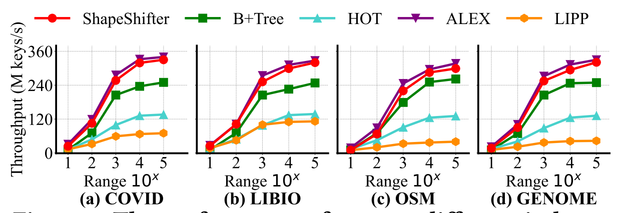


Figure 8: The performance of scan on different indexes.

the speed of evolution gradually increases, completing by the 11.4th second, at which point throughput peaks. In $WR_{1.3}$, as shown in Figure 7 (c), changes in index size are achieved through periodic compression operations. In this experiment, we set the system to perform a compression every 3 seconds to control and optimize the space occupancy of the index. The results indicate that compressing cold nodes considerably reduces the space overhead of ShapeShifter, thereby achieving the optimal time-space trade-off.

Insight 3: Over time, the number of evolving operations in ShapeShifter progressively increases, resulting in continual improvements in performance and reductions in space overhead, ultimately achieving the optimal time-space trade-off.

In Figure 7 (b), at the 12th second, we subject all indexes to a poisoning attack designed by Yang et al.[45], following which ALEX experiences an out-of-memory (OOM) error and crashes. ShapeShifter and traditional indexes are resilient to this attack, showing strong robustness. Although ShapeShifter experiences a temporary decline in throughput when some poisoned data is not added to hot nodes, performance swiftly rebounds once data insertion resumes in hot nodes.

Insight 4: The poisoning attack reveals the high vulnerability of ALEX and the significant robustness of ShapeShifter which benefits from a hybrid architecture that combines traditional/learned models without relying solely on the latter.

4.5 Index Range Query Evaluation

This experiment is designed to evaluate the performance of range queries. Initially, we batch load a complete dataset containing 200 million keys into each index, and then initiate a read-only scan workload. Each query starts from a randomly selected starting key K and retrieves a fixed number of keys. Throughout the experiment, a total of 10 million range queries are issued, and throughput is assessed by calculating the number of keys accessed per second.

Figure 8 shows the performance of five different indexes across a range of query sizes, i.e., from 10 to 100,000. The performance of ShapeShifter is comparable to that of ALEX, both achieving the current best performance level. This indicates that ShapeShifter also maintains high throughput and excellent performance when handling large-scale range queries. This is largely due to the leaf nodes of ShapeShifter utilizing the same linked-list connection method as B+Tree, and the adoption of bitmap structures in its learned architecture to skip gaps, thereby accelerating scan performance.

HOT and LIPP exhibit relatively poorer performance, primarily because HOT utilizes a Trie structure, while LIPP employs a structure similar to a B-Tree. Both require frequent recursive querying of parent nodes during range queries. Note that since ART is based on a Trie tree structure as well, its scan performance is even worse than that of LIPP, and hence it is not displayed here.

Insight 5: ShapeShifter exhibits superior scan performance, whereas HOT shows poorer scan capabilities.

4.6 Summary

In this section, our evaluation confirms that ShapeShifter offers the best time-space trade-off, and scanning performance near SOTA i.e., ALEX. While ALEX also performs well in time-space trade-off (hereinafter referred to as "performance"), its susceptibility leads to an inferior performance-robustness trade-off. This is largely because a sole reliance on learned models cannot adequately address complex CDF variations in real scenarios. In contrast, ShapeShifter uses a hybrid of traditional and learned models, enhancing performance through learned models while maintaining robustness with traditional models, thus achieving a better performance-robustness trade-off. Despite the good time-space trade-off of HOT, its poor scanning performance reduces competitiveness against ShapeShifter. Overall, ShapeShifter exhibits the best performance when simultaneously considering time-space trade-off, performance-robustness trade-off, and scanning capabilities.

5 RELATED WORK

Kraska et al.[20] introduced a read-only learned index that reduces index size and query latency using learned models. To make learned indexes more practical, several updatable versions have been developed[14, 16, 21, 22, 30, 31, 33, 34, 37, 40, 41, 44, 46, 48, 50]. Galakatos et al.[12] pioneered the FITing-tree, an updatable learned index with segment-specific buffers. Ferragina et al.[11] applied LSM-Tree[35] insertion concepts to create the updatable PGM-Index. Ding et al.[9] developed ALEX, employing a model-based insertion strategy to improve time-space trade-off. Wu et al.[43] designed the LIPP, which delivers optimal search and insertion performance. Li et al.[27] combined ALEX and LIPP to design the data distribution-aware DILI. Tang et al.[39] introduced XIndex, the first learned index supporting concurrent write operations. Li et al.[25] proposed FINEdex with fine-grained insertions. Zhang et al.[49] designed Hyper for superior time-space trade-off. Chameleon, by Guo et al.[17], is an adaptive learned index optimized for skewed scenarios with frequent updates.

Note that this paper compares ALEX and LIPP as public benchmarking tools[13, 18, 32, 38, 42] indicate their SOTA performance. Given that architecture of DILI is similar to ALEX and equally susceptible to poisoning attacks, it is not compared separately. Hyper is not open-sourced and is also not compared in this study. Chameleon does not aim for optimal time-space trade-off, and its local hash structure does not support scan operations; therefore, it is not compared with ShapeShifter. There are also some excellent works on SOTA multidimensional learned indexes[8, 10, 15, 26]. However, these works are beyond the scope of our discussion.

6 CONCLUSION

This paper proposes ShapeShifter, an adaptive index aware of workload dynamics, blending traditional and learned models to maintain robustness while dynamically optimizing its structure for skewed workloads, targeting optimal time-space trade-off. Experimental results demonstrate that under skewed workloads, ShapeShifter increases throughput by up to 6.34× and reduces space usage by 51.79% compared to traditional indexes. Against SOTA learned indexes, it achieves up to 1.26× higher throughput and decreases space usage by 824.19%. Moreover, ShapeShifter effectively balances time and space while successfully preventing poisoning attacks.

REFERENCES

[1] Christoph Anneser, Andreas Kipf, Huanchen Zhang, Thomas Neumann, and Alfons Kemper. 2022. Adaptive Hybrid Indexes. In *Proceedings of the 2022 International Conference on Management of Data*. 1626–1639.
[2] Manos Athanassoulis, Michael S Kester, Lukas M Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture.. In *EDBT*, Vol. 2016. 461–466.
[3] AWS. 2010. OpenStreetMap database. https://aws.amazon.com/public-datasets/osm.
[4] Timo Bingmann. 2013. STX B+ Tree 0.9. https://panthema.net/2007/stx-btree/, retrieved Sep. 1, 2021..
[5] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 521–534.
[6] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.
[7] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
[8] Angjela Davitkova, Evica Milchevski, and Sebastian Michel. 2020. The ML-Index: A Multidimensional, Learned Index for Point, Range, and Nearest-Neighbor Queries.. In *EDBT*. 407–410.
[9] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.
[10] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: a learned multi-dimensional index for correlated data and skewed workloads. *Proceedings of the VLDB Endowment* 14, 2 (2020), 74–86.
[11] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.
[12] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*. 1189–1206.
[13] Jiake Ge, Boyu Shi, Yanfeng Chai, Yuanhui Luo, Yunda Guo, Yinxuan He, and Yunpeng Chai. 2023. Cutting Learned Index into Pieces: An In-depth Inquiry into Updatable Learned Indexes. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 315–327.
[14] Jiake Ge, Huanchen Zhang, Boyu Shi, Yuanhui Luo, Yunda Guo, Yunpeng Chai, Yuxing Chen, and Anqun Pan. 2023. SALI: A Scalable Adaptive Learned Index Framework based on Probability Models. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.
[15] Behzad Ghaffari, Ali Hadian, and Thomas Heinis. 2020. Leveraging soft functional dependencies for indexing multi-dimensional data. *arXiv preprint arXiv:2006.16393* (2020).
[16] Na Guo, Yaqi Wang, Haonan Jiang, Xiufeng Xia, and Yu Gu. 2022. TALI: An Update-Distribution-Aware Learned Index for Social Media Data. *Mathematics* 10, 23 (2022), 4507.
[17] Na Guo, Yaqi Wang, Wenli Sun, Yu Gu, Jianzhong Qi, Zhenghao Liu, Xiufeng Xia, and Ge Yu. 2024. Chameleon: Towards Update-Efficient Learned Indexing for Locally Skewed Data. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4316–4328.
[18] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A benchmark for learned indexes. *NeurIPS Workshop on Learned Systems* (2019).
[19] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–5.
[20] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.
[21] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The case for a learned sorting algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1001–1016.
[22] Hai Lan, Zhifeng Bao, J Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS-From Evaluations to Design Choices. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–22.
[23] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 185–196.
[24] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.
[25] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.
[26] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2119–2133.
[27] Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. 2023. DILI: A Distribution-Driven Learned Index. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2212–2224.
[28] Libraries.io. 2017. Repository ID. (2017). https://libraries.io/data.
[29] Christian E Lopez and Caleb Gallemore. 2021. An augmented multilingual Twitter dataset for studying the COVID-19 infodemic. *Social Network Analysis and Mining* 11, 1 (2021), 102.
[30] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: a high-performance learned index on persistent memory. *Proceedings of the VLDB Endowment* 15, 3 (2021), 597–610.
[31] Chaohong Ma, Xiaohui Yu, Yifan Li, Xiaofeng Meng, and Aishan Maoliniyazi. 2022. FILM: A Fully Learned Index for Larger-Than-Memory Databases. *Proceedings of the VLDB Endowment* 16, 3 (2022), 561–573.
[32] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* (2020).
[33] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1705–1718.
[34] Mayank Mishra and Rekha Singhal. 2021. RUSLI: real-time updatable spline learned index. In *Proceedings of the Fourth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–8.
[35] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.
[36] Suhas SP Rao, Miriam H Huntley, Neva C Durand, Elena K Stamenova, Ivan D Bochkov, James T Robinson, Adrian L Sanborn, Ido Machol, Arina D Omer, Eric S Lander, et al. 2014. A 3D map of the human genome at kilobase resolution reveals principles of chromatin looping. *Cell* 159, 7 (2014), 1665–1680.
[37] Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. Towards practical learned indexing. *arXiv preprint arXiv:2108.05117* (2021).
[38] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned index: A comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.
[39] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.
[40] Yulai Tong, Jiazhen Liu, Hua Wang, Ke Zhou, Rongfeng He, Qin Zhang, and Cheng Wang. 2023. Sieve: A Learned Data-Skipping Index for Data Analytics. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3214–3226.
[41] Zhaoguo Wang, Haibo Chen, Youyun Wang, Chuzhe Tang, and Huan Wang. 2022. The concurrent learned indexes for multicore data storage. *ACM Transactions on Storage (TOS)* 18, 1 (2022), 1–35.
[42] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proceedings of the VLDB Endowment* (2022).
[43] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.
[44] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: robust learned index via distribution transformation. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2188–2200.
[45] Rui Yang, Evgenios M Kornaropoulos, and Yue Cheng. 2024. Algorithmic Complexity Attacks on Dynamic Learned Indexes. *arXiv preprint arXiv:2403.12433* (2024).
[46] Tong Yu, Guanfeng Liu, An Liu, Zhixu Li, and Lei Zhao. 2023. LIFOSS: a learned index scheme for streaming scenarios. *World Wide Web* 26, 1 (2023), 501–518.
[47] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.
[48] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: a cache-aware learned index with a cost-based construction algorithm. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2679–2691.
[49] Shunkang Zhang, Ji Qi, Xin Yao, and André Brinkmann. 2024. Hyper: A High-Performance and Memory-Efficient Learned Index via Hybrid Construction. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.
[50] Yong Zhang, Xinran Xiong, and Oana Balmau. 2022. TONE: cutting tail-latency in learned indexes. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. 16–23.

A APPENDIX

A.1 The Scan Operation Algorithm in ShapeShifter

Algorithm 3 comprehensively delineates the range query mechanism of ShapeShifter. Initially, the algorithm identifies the leaf node containing the target key and retrieves its storage location, thereby initiating the query process (lines 2-3). Subsequently, the system traverses this leaf node, counting the number of keys scanned until the traversal of the entire linked list is complete or the cumulative count of keys meets the preset target (lines 4-11). During this process, depending on the type of leaf node, appropriate scanning strategies are employed: for leaf nodes utilizing learned models, scanning begins from the target value and employs bitmap technology to efficiently skip gaps, thereby accelerating the scanning process (lines 5-6). for leaf nodes structured as B+Trees, the standard scanning is implemented (lines 7-9). After traversing the current leaf node, the algorithm updates the node pointer to its subsequent node (line 10). Ultimately, the total number of keys scanned is returned at the end of the process (line 12).

Algorithm 3: ShapeShifter Range Query Operation

```
   Input: $key$, $target\_num$
   Output: $scan\_num$
1  Function Range_Query_Operation($key$, $target\_num$):
2      $Leaf\_node \leftarrow$ Traverse_To_Leaf($key$)
3      tar$\langle key, value\rangle \leftarrow$ Lookup_Operation($key$)
4      while $Leaf\_node \neq Null\ \ \&\ \ scan\_num < target\_num$ do
5          if $Leaf\_node.model\ \ is\ \ linear$ then
6              $scan\_num$ += $ScanBitmap(Leaf\_node,$ tar $value)$
7          else if $Leaf\_node.model\ \ is\ \ B+Tree$ then
8              $scan\_num$ += $ScanB{+}node(Leaf\_node,$ tar $value)$
9          end
10         $Leaf\_node \leftarrow Leaf\_node.nextnode$
11     end
12     return $scan\_num$
```

A.2 The Delete Operation Algorithm in ShapeShifter

Algorithm 4 fully describes the deletion process for ShapeShifter. Initially, the algorithm locates the leaf node containing the key and retrieves its associated storage location (lines 2-3). For leaf nodes employing a learned model, the model-specific deletion method is used to remove the key and its corresponding payload (lines 4-5); for leaf nodes that are B+Tree nodes, the standard deletion procedure of the B+Tree is implemented (lines 6-8).

A.3 Internal Node Evolving of ShapeShifter

Section 3.2.2 provides a detailed discussion of the evolving strategies for leaf nodes. Specifically, based on the COST model proposed by Zhang et al.[47], ShapeShifter assesses whether leaf nodes meet the evolutionary criteria. Consequently, internal nodes, when meeting the COST criteria, should also evolve into learned structures to enhance the overall system performance. The parent and ancestor nodes of hot leaf nodes involved in read/write operations are invariably hot nodes, however, their all child nodes may not necessarily be hot. Therefore, this paper stipulates that an internal node should evolve into a learned structure when the proportion of its hot child nodes exceeds the ϕ threshold of all its children. Before evolving, it is necessary to determine whether the node satisfies the COST criteria. If compliant, the node will evolve into a hot read structure; if not, it will retain its existing structure, i.e., traditional model. Nodes that preserve the traditional structure will continue to carry the hot label, facilitating their recognition as hot nodes by their parent nodes during the evolution of the latter.

Algorithm 4: ShapeShifter Delete Operation

```
   Input: $key$
1  Function Delete_Operation($key$):
2      $Leaf\_node \leftarrow$ Traverse_To_Leaf($key$)
3      tar$\langle key, value\rangle \leftarrow$ Lookup_Operation($key$)
4      if $Leaf\_node.model\ \ is\ \ linear$ then
5          delete tar$\langle key, value\rangle$ uses linear methods
6      else if $Leaf\_node.model\ \ is\ \ B+Tree$ then
7          delete tar$\langle key, value\rangle$ uses B+ methods
8      end
```

Note that the evolving of internal nodes solely results in their transformation into hot read structures within a learned framework. This configuration maximizes read performance while ensuring minimal space overhead. Additionally, write operations on internal nodes are confined to instances of leaf node splitting. Consequently, compared to leaf nodes, the frequency of updates in internal nodes is considerably lower.

In the experimental of this paper, we stipulate that if any child node is identified as a hot node, its parent node must evolve. This rule is based on the fact that under conditions satisfying the COST criteria, the performance improvements from evolving a parent node far outweigh its spatial costs. Furthermore, under skewed workloads, we believe that evolving parent nodes offers a high benefit-cost ratio. Experimental evaluations indicate that the space overhead incurred by the evolving of a single parent node is negligible, and all experimental results presented support this conclusion.

A.4 Supplementary Conditions for Determining Hot Write Nodes

The computation of Equation 3 may result in a zero value for $n.insert\_rate_t$. In such instances, the cumulative probability computed by Equation 6 will invariably be zero, thereby precluding any further alterations in $n.insert\_rate_t$. To address this issue, a corrective factor ζ is introduced into the numerator of Equation 6. Specifically, ζ is defined as $\zeta = path\_size/10000$, where $path\_size$ denotes the path length from the root node to the current node. The refined cumulative probability model, encapsulated in Equation 7, determines whether a node has accommodated a sufficient number of new insertions.

$$P_{write} = \frac{n.insert\_rate_t \times (n.current\_time - n.time_t)] + \zeta}{\epsilon \times [\beta \times \min\left(1, \frac{n.insert\_rate_t}{n.insert\_rate_{t-1}+\zeta}\right) - 1]} \quad (7)$$

A.5 Supplementary Notes on Zipfian

The Zipfian distribution[7], proposed by George Zipf, is a discrete probability distribution used to describe the distribution patterns

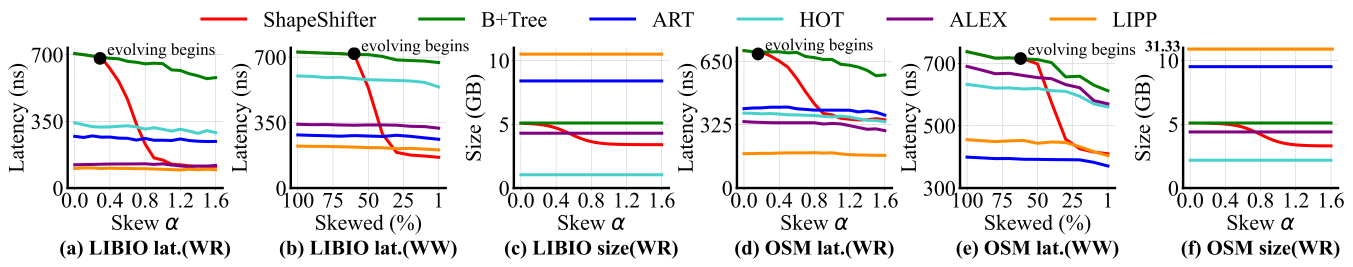


Figure 9: Average latency/space overhead of various indexes on LIBIO/OSM under workloads with varying skew degrees.

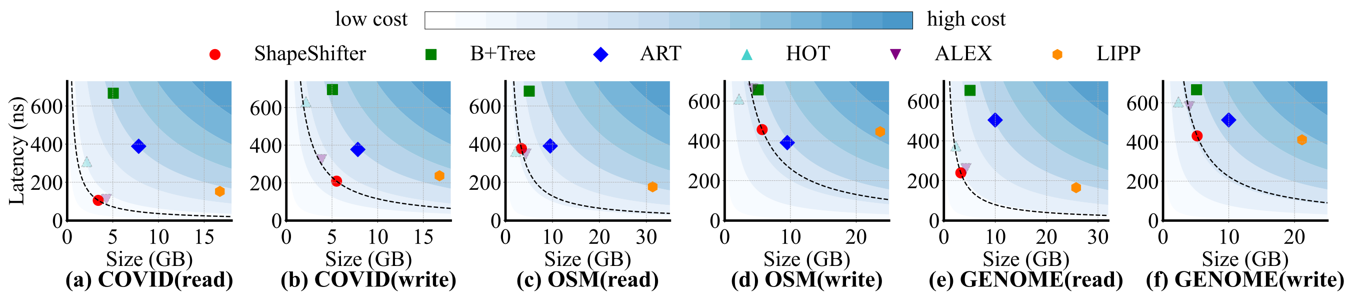


Figure 10: Time-Space trade-off evaluation of different indexes on the COVID/OSM/GENOME datasets. The reads follow a standard Zipfian distribution, i.e., $\alpha = 1$ (WR_1). Writes employ a skewed workload of 30%, i.e., WW_{30}. The "points" for Alex and Hot are rendered "transparent" because their performance in terms of robustness and scan capabilities is poor, making them incomparable to ShapeShifter.

like word frequency, population distribution across cities, and internet traffic. It follows a power-law, where a few events are highly frequent and most are rare, resulting a "long-tail" feature. This distribution effectively models real-world phenomena where a small number of occurrences dominate, while the majority are less common.

The probability mass function (PMF) of the Zipfian distribution is given by the following Equation 8:

$$P(k; N, \alpha) = \frac{1/k^\alpha}{\sum_{n=1}^{N} \frac{1}{n^\alpha}} \quad (8)$$

where: k is the event rank (positive integer, from highest to lowest frequency), N is the total number of events, α is the parameter of the distribution, known as the "exponent" or "scaling parameter", The denominator $\sum_{n=1}^{N} \frac{1}{n^\alpha}$ is the normalization constant, ensuring that the total probability sums to 1.

The parameter α determines the shape and steepness of the distribution. Specifically, α controls the relative difference between high-frequency and low-frequency events:

When $\alpha = 1$, the Zipfian distribution is referred to as the Zipf distribution, which corresponds to the case originally proposed by Zipf. In this case, the event frequency is inversely proportional to its rank, i.e., $P(k) \propto \frac{1}{k}$.

When $\alpha > 1$, the probability of high-ranked events increases sharply, and the probability of low-ranked events decreases rapidly, making the distribution steeper, i.e., the workload is more skewed..

When $0 < \alpha < 1$, the probability of low-ranked events becomes larger, and the tail of the distribution flattens, i.e., the workload is more uniform.

Larger values of α generally indicate a more "concentrated" distribution, where high-frequency events dominate substantially over low-frequency events. Conversely, smaller values of α imply a more "flattened" distribution, with higher probabilities for tail events.

A.6 Supplementary Illustrations for Index Performance Evaluation

In the Figure 9, we evaluate six indexes in terms of average latency and index size within the LIBIO and OSM datasets. In the *WR* scenario of the LIBIO dataset, the evolving begins point is set at $WR_{0.3}$. At WR_1, ShapeShifter reduces the average query latency by 448.98% compared to the B+Tree, performs on par with ALEX, and achieves a reduction in index size of 46.67%/24.64%. Although ShapeShifter exhibits a higher average latency than the SOTA, i.e., LIPP, it significantly reduces the index size by 203.48%, presenting the best trade-off.

In the *WW* scenario, the evolving begins at WW_{60}. Under the WW_{30} scenario, ShapeShifter decreases the average latency by 312.33%/26.47% compared to the B+Tree/SOTA, i.e., LIPP, respectively, while maintaining parity in index size with the B+Tree and achieving a 203.01% reduction compared to LIPP.

In the WR_1 scenario of the OSM dataset, ShapeShifter reduces the average query latency by 80.27% compared to the B+Tree, while also decreasing the index size by 49.85%. Although ShapeShifter exhibits a higher average latency than ALEX and the SOTA, i.e., LIPP, it reduces the index size by 28.62% and 824.19% respectively, thus maintaining a favorable trade-off.

In the WW_{30} scenario, ShapeShifter reduces average latency by 45.39% compared to the B+Tree and is comparable to LIPP. In terms of index size, ShapeShifter is on par with the B+Tree, but achieves a significant reduction of 314.91% compared to LIPP. Even though the average latency of ShapeShifter is higher than the SOTA, i.e., ART,

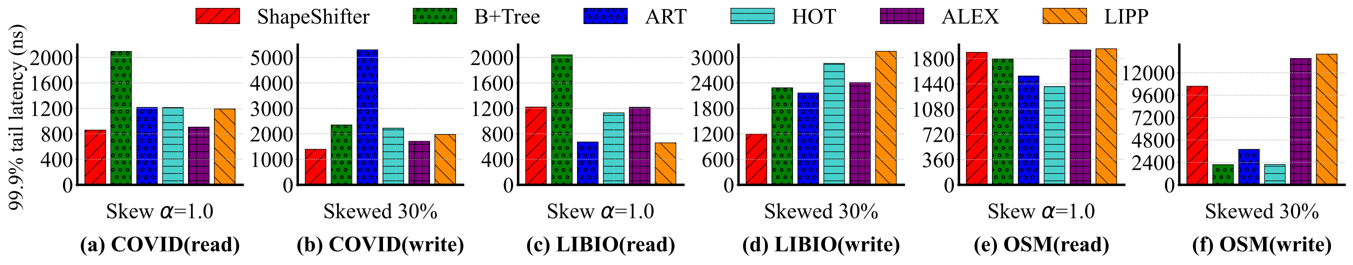


Figure 11: Demonstrating latency performance of different indexes under skewed workloads. The reads follow a standard Zipfian distribution, i.e., $\alpha = 1$ (WR_1). Writes employ a skewed workload of 30%, i.e., WW_{30}.

its reduction in index size by 67.02% still ensures a more favorable trade-off.

It should be noted that in the OSM dataset, the performance of ShapeShifter is not SOTA, primarily due to the pronounced nonlinear characteristics of OSM, which make it challenging to fit with learned models based on piecewise linear functions[42]. Despite this, ShapeShifter still manages to demonstrate a favorable trade-off between time and space under these circumstances.

A.7 Supplementary Illustrations for Time-Space Trade-off Evaluation

The Figure 10 displays a cost comparison among six indexes in WR_1 and WW_{30} scenarios across three datasets: COVID, OSM, and GENOME. During poisoning attacks, ALEX exhibits out-of-memory (OOM) issues leading to system crashes, revealing its high susceptibility to attacks, as shown in Figure 7. Additionally, HOT performs poorly in scanning tasks, as shown in Figure 8. Consequently, these two indexes are not considered in our analysis. The Figure 10 clearly shows that ShapeShifter consistently demonstrates the best trade-off between time and space in all WW and WR scenarios.

A.8 Evaluation of Tail Latencies in Various Indexes

The Figure 11 illustrates the performance of six indexes at the 99.9% tail latency across the COVID, LIBIO, and OSM datasets. In the COVID dataset, ShapeShifter exhibits the best tail latency for WR and WW operations, demonstrating high performance stability with rare occurrences of high latency. In the WW scenarios of the LIBIO dataset, ShapeShifter shows the lowest tail latency and the fastest response times; in the WR scenarios, its tail latency is comparable to ALEX. In the OSM dataset, which is characterized by pronounced non-linear features, ShapeShifter performs best among learned indexes in terms of WR and WW tail latency, yet its overall tail latency remains higher than that of traditional indexes. This underscores the limitations of learned models based on piecewise linear functions and suggests that traditional indexes might provide greater stability under complex datasets.

A.9 Evaluation under Changing Workload Skewness Distribution

Figure 12 (a) and (b) respectively illustrate the dynamic evolving process of ShapeShifter under the same skewness level but different skewness ranges. The skewness ranges before and after the workload change do not overlap, i.e., the key for reads/inserts are

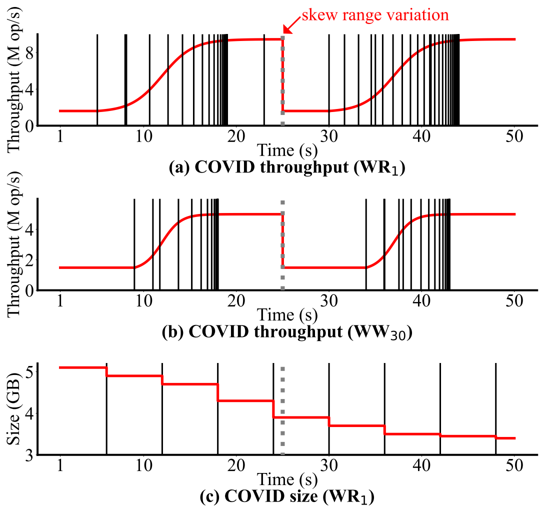


Figure 12: The schematic of ShapeShifter adapting to changes in workload distribution is presented. The reads follow a standard Zipfian distribution, i.e., $\alpha = 1$ (WR_1). Writes employ a skewed workload of 30%, i.e., WW_{30}. The space overhead is demonstrated under the WR_1 workload. The black vertical lines represent evolving points.

entirely different. Before the distribution change, the performance of ShapeShifter is consistent with the results shown in Figure 7 (a) and (b). After the distribution change, ShapeShifter experiences an immediate performance drop due to the fact that the nodes have not yet evolved. However, as the evolving progresses, the performance gradually recovers, eventually reaching the same level as before the distribution change. The same dynamic pattern is observed under both WR_1 and WW_{30} workloads, fully demonstrating that ShapeShifter can quickly adapt to changes in workload skewness and promptly adjust to achieve optimal performance.

Similar to Figure 7 (c), Figure 12 (c) illustrates the dynamic changes in space compression under the WR_1 scenario. We continue to trigger compression tasks at fixed time intervals. It can be observed that space compression is minimal during the evolving phase, while it significantly increases at the end of the evolving process, which is due to the additional space overhead introduced during the evolution. Moreover, after 40 seconds, the compression ratio gradually decreases because there are no longer sufficient cold nodes in the cooling pool for compression, indicating that the space overhead of ShapeShifter has reached its optimal state.