Learned Indexes with Distribution Smoothing via Virtual Points

Kasun Amarasinghe
The University of Melbourne
Melbourne, Australia
kasun.amarasinghe@student.unimelb.edu.au

Farhana Choudhury
The University of Melbourne
Melbourne, Australia
farhana.choudhury@unimelb.edu.au

Jianzhong Qi
The University of Melbourne
Melbourne, Australia
jianzhong.qi@unimelb.edu.au

James Bailey
The University of Melbourne
Melbourne, Australia
baileyj@unimelb.edu.au

ABSTRACT

Recent research on learned indexes has created a new perspective for indexes as models that map keys to their respective storage locations. These learned indexes are created to approximate the cumulative distribution function of the key set, where using only a single model may have limited accuracy. To overcome this limitation, a typical method is to use multiple models, arranged in a hierarchical manner, where the query performance depends on two aspects: (i) traversal time to find the correct model and (ii) search time to find the key in the selected model. Such a method may cause some key space regions that are difficult to model to be placed at deeper levels in the hierarchy. To address this issue, we propose an alternative method that modifies the key space as opposed to any structural or model modifications. This is achieved through making the key set more learnable (i.e., smoothing the distribution) by inserting virtual points. Furthermore, we develop an algorithm named CSV to integrate our virtual point insertion method into existing learned indexes, reducing both their traversal and search time. We implement CSV on state-of-the-art learned indexes and evaluate them on real-world datasets. Extensive experimental results show significant query performance improvement for the keys in deeper levels of the index structures at a low storage cost.

1 INTRODUCTION

Learned indexes [13] have reported strong query performance and are attracting much attention from both the academia and industry in recent years. The core idea of learned indexes is that an index structure can be seen as a mapping function $f(\cdot)$ from a search key k_i to the storage location (i.e., the rank $rank(k_i)$) of the corresponding data record: $rank(k_i) \approx f(k_i)$. The mapping function (a.k.a. *indexing function*) is learned and approximated by machine learning algorithms (models). To enable the learning, a storage ordering needs to be established. Typically, an ascending order based on the search keys is used, such that the mapping function is effectively the *cumulative distribution function* (CDF) of the search keys.

Different learned indexes have been proposed [2–4, 7, 10, 14–16, 20, 23, 24, 28, 30, 32, 34, 38] with a common theme to design indexing functions and structures that enable better approximation of the CDF, since approximation errors translate to mapping errors (i.e., difference between $rank(k_i)$ and $f(k_i)$) and hence extra search costs to examine data at around $f(k_i)$ and recover from errors. However, such approaches mean the use of either complex indexing functions (e.g., splines [11, 22]) or piece-wise functions with many segments [7, 8, 15], both of which could lead to sub-optimal query efficiency. This is illustrated by Fig. 1 with LIPP [34] – one of the latest learned indexes. The index has a hierarchical structure built in a top-down manner. When an index model (i.e., a node in the index) cannot achieve an overall low mapping error for all keys assigned to it for indexing, sub-index models are created recursively as a hierarchy, to accommodate for the "more difficult to learn" keys. As Fig. 1 shows, keys indexed in deeper levels (higher levels in the figure) reported higher query times on average, on all four datasets.

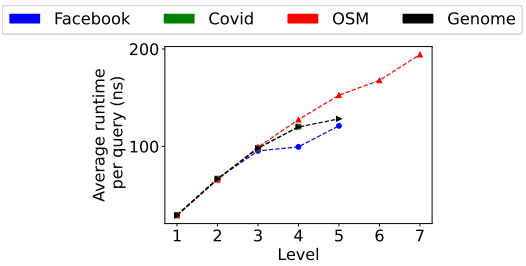


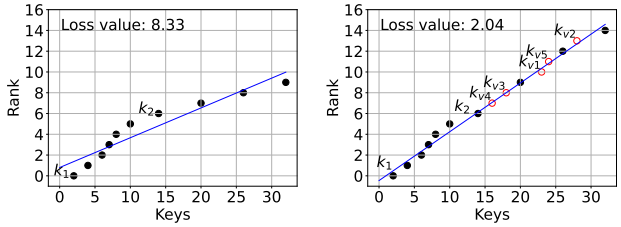
Figure 1: Query time at each level of the LIPP index for four real datasets, each with 200 million keys.

In this paper, we approach the problem from an alternate perspective – we adjust the CDF such that it becomes easier to be approximated by the indexing functions, to achieve lower approximation errors and higher query efficiency. Our core idea is to add *virtual points* to "smooth" the CDF of a dataset. Take Fig. 2a as an example, where each black dot represents a data point (i.e., its search key). Approximating the CDF of the dataset with a linear function can result in a large approximation error (and hence high search costs at query time) for keys k_1 and k_2. We sum up the squared prediction error of every point:

$$\mathcal{L}_f(K) = \sum_{i=1}^{n} \big(f(k_i) - rank(k_i)\big)^2, \tag{1}$$

where K denotes the set of keys and n is its size, $k_i \in K$ is a key, $rank(k_i)$ is its rank, and $f(\cdot)$ is the indexing function. We refer to $\mathcal{L}_f(K)$ as the *loss function*, for which we use the *sum of squared errors* (SSE). In this case, $\mathcal{L}_f(K) = 8.33$ – a large value of $\mathcal{L}_f(K)$ suggests worse prediction accuracy using f for search key mapping and hence higher query times.

As Fig. 2b shows, we can add virtual points $V = \{k_{v1}, k_{v2}, \ldots, k_{v5}\}$ represented by the red hollow dots. Here, we assume a *smoothing budget* of $0.5n = 5$, i.e., 5 virtual points are allowed. Now the original data points are spread out, and the CDF of the (original



(a) Before smoothing **(b) After smoothing**

Figure 2: Indexing data points (keys) with CDF smoothing.

and virtual) points is closer to a straight line. We refit the points with a new indexing function f', with the loss function value $\mathcal{L}_{f'}(K)$ being reduced to 2.04 (and $\mathcal{L}_{f'}(K \cup V) = 2.29$).

We show that, given a smoothing budget that constrains the additional space costs, finding the optimal placement of the virtual points to minimise the loss function $\mathcal{L}_{f'}(K)$ is NP-hard. We then propose approximation solutions for two generic scenarios: (1) smoothing the CDF for index learning with a single indexing function and (2) smoothing the CDF for a hierarchy of indexing functions, which is a common structure for existing learned indexes.

We propose an algorithm, *CDF smoothing via virtual points* (*CSV*) to smooth the CDF for optimising hierarchical learned indexes, to reduce the overall height of the structures as well as the prediction errors of each indexing function, and hence the query costs. This is performed by collecting sub-trees of the hierarchical structure and smoothing the CDF of the keys in them. As a result, the keys that were initially in lower levels of the sub-tree, could now be placed or 'promoted' into a new single node in the root of the sub-tree due to the higher learnability, provided it surpasses a cost model threshold value. Here, the cost model is used to balance the reduction in index traversal time and the potential increase in the leaf-node search time due to the increase of keys in a node.

It is important to note that our aim is *not* to propose yet another learned index but rather a technique that can be integrated with existing or emerging hierarchical learned indexes to optimise their query efficiency with controllable extra space. To show the applicability of CSV we integrate it with three recent learned indexes ALEX [2], LIPP [34], and SALI [10], which are the state-of-the-art (SOTA). To summarize, this paper makes the following contributions: (1) We propose a key space transformation technique using CDF smoothing via inserting virtual points to enhance index learnability. (2) We propose an efficient algorithm, CSV to integrate the CDF smoothing technique with hierarchical learned indexes to improve the query performance, with a controllable space overhead (3) We integrate CSV with three learned indexes ALEX, LIPP, and SALI, and we conduct experiments with four real datasets. The experimental results show that the learned indexes powered by CSV manage to promote up to 60% of the keys in lower levels to upper levels, resulting in up to 34% improvement of their query time, with less than 15% increase to the storage space overhead.

2 RELATED WORK

We first review learned indexes in general. Then, we focus on studies addressing complex distributions, which share a similar goal with us. We also cover a technique called the poisoning attacks, which motivates our CDF smoothing technique.

2.1 Learned Indexes

Learned indexes are a trending topic in the database community [2, 4, 7, 11, 13, 14, 20, 34, 36, 38]. Their key idea is to treat indexes as functions that map a search key to the storage position of the corresponding data object, which can be learned with machine learning models. A common approach is to lay out the data objects by ascending order of their search keys, such that the indexing functions are effectively (approximations of) CDFs of the keys.

To index large datasets, multiple indexing functions are used, typically organized in a hierarchy like a B-tree. The lookup performance of such a structure is then dominated by two steps: (1) the *traversal time* to find the leaf-node (every node corresponds to an indexing function) indexing the search key, and (2) the search time within the selected leaf-node (*leaf-node search time* hereafter to distinguish from the traversal time) to locate the target data object, as the indexing functions have errors and may not produce the exact storage position of the search target [29, 33]. It is a challenge to balance the query costs from the two steps above. While a deeper structure with more indexing functions may fit the data distribution better and have lower leaf-node search times, it may also have higher traversal times and larger index sizes [9, 35]. Some studies impose a maximum error bound on the indexing functions to reduce the leaf-node search times [7, 8], also at the cost of more indexing functions. Another approach is to use more complex indexing functions (as opposed to linear ones) [11, 13, 30], e.g., splines, which could better fit the CDFs. The issue with this approach is the higher inference time for the function, and hence higher query and insertion times [29]. These studies design structures and indexing functions to better fit the data distribution. We address the challenge from an alternate perspective, i.e., we adjust the data distribution such that it is easier to be fitted by the indexing functions.

2.2 Addressing Complex Data Distributions

To better index CDFs of complex data distributions, there are two common approaches. One is to use more complex functions such as splines and piece-wise linear regression models [7, 11]. The other is to use better data partitioning strategies for easier CDF learning over each partition, such as by CARMI [37] and EWALI [18]. Another study, LER [5], uses logarithmic error-based loss functions (instead of the more commonly used least squared error-based) to improve the learning of index models that better fit the CDF.

A latest development, SALI [10], identifies the most frequently accessed nodes via probability models given a query workload. The corresponding sub-trees are flattened using a segmentation approach, similar to the PGM index [7], to reduce their traversal time. However, this leads to an additional search step for queries, as we need to find the correct node from the flattened structure.

A couple of studies [17, 35] transform the input key set into a more uniform distribution to improve the CDF learnability. The NFL index [35] transforms the key distribution using a numerical normalizing flow that transforms a latent distribution to a new distribution via generative models. The distribution transformation introduces overheads, while queries also need to be transformed to use the index. Further, the transformation may increase the tail conflict degree for certain distributions, making it unsuitable in those instances. The *gap insertion* [17] technique inserts gaps between the keys (i.e., storage positions of the corresponding data objects) to straighten the CDF of the keys, by

Table 1: Comparison with Existing Works

	CSV	NFL [35]	GI [17]
No extra transformation at query time	✓	✗	✓
Low storage overhead	✓	✓	✗
Integrable into other learned indexes	✓	✓	✗
Robust across different distributions	✓	✗	✓

first identifying a better indexing mechanism and using it obtain new ranks for the key set, thereby improving its learnability. However, this is performed by manipulating the rank of each key, and as a result, multiple keys could be given the same position. An extra array is used to house such conflicting keys, which in turn introduces search overheads to locate the correct key. This method leads to a heavy storage space increase of up to 87%.

Several learned indexes [2, 20, 34] leave gaps in their storage structure (i.e., gapped arrays). While their purpose is to accommodate data insertions, a side effect is changing the data distribution, which is what we do. A core difference to note is that, they do not consider minimizing the indexes' model prediction errors when adding gaps, in contrast to our approach which does.

2.3 Poisoning CDFs

Our idea of adjusting data distribution to fit the indexing functions is rooted from *data poisoning* – a process of manipulating the training data to change the results from a predictive model [12]. Data poisoning has been introduced into learned indexes to poison the indexing functions and negatively impact their capability to approximate the CDFs [12]. The main goal of this process is to identify new points to include into the original key set that would cause the maximum increase to the loss function value (i.e., the SSE). To achieve this goal, they have developed a greedy method of identifying and inserting a poisoning point sequentially. Using the properties the loss function such as it being a composite of convex sub-sequences has led to a more computationally efficient manner of identifying poisoning keys. Since the loss function is a collection of convex sub-sequences as depicted in Fig.3, the best poisoning point can always be found in one of these endpoints, significantly reducing unnecessary calculations. This poisoning method only harms the leaf-node search time for data nodes, ignoring the traversal time needed to identify the correct leaf-node. Furthermore, recent learned indexes such as LIPP and SALI does not process data nodes as they are precise learned indexes.

Motivated by the poisoning technique, we propose a technique that smooths the data distribution by adding virtual points, to obtain CDFs that are easier to be approximated by indexing functions (models), hence leading to a structure with higher query efficiency. Since the models are built with virtual points that can be used to host data insertions, a side benefit of our structure is that it is more resilient against data insertions. Table 1 highlights the key difference between our technique CSV, NFL, and gap insertion (GI).

3 PRELIMINARIES

Problem statement. Consider a dataset D of n data records, where each record is associated with a one-dimensional value as its index key. Let K be the list of all index keys associated with D, sorted in ascending order.

Suppose that the index keys have been partitioned and indexed by a set $\mathcal{F}$ of m_l indexing functions at level l of the index. Each indexing function $f_i \in \mathcal{F}$ has some prediction error (a.k.a "loss") for a key $k \in K_i$ indexed by it. Here, $K_i \subset K$ refers to the subset of keys indexed by f_i and its sub-tree. The loss refers to the squared difference between the predicted index position $f_i(k)$ and the rank of k in K, i.e., $rank(k)$. The sum of squared errors (SSE) is one of the most commonly used metrics to represent loss in the existing studies of learned indexes. Let $\mathcal{L}_{\mathcal{F}}$ be the *total sum of squared errors* of all indexing functions in $\mathcal{F}$ for level l:

$$\mathcal{L}_{\mathcal{F}}(K) = \sum_{i=1}^{m_l} \sum_{k \in K_i} \big(f_i(k) - rank(k)\big)^2. \tag{2}$$

Due to these errors in the indexing functions, learned indexes cannot query the location of a key accurately. To overcome this, learned indexes tend to sub-divide the key set, and each subdivision is fitted by an individual indexing function to reduce the errors. This approach creates a hierarchical structure and adds extra inference time. Our approach addresses the issue by manipulating the dataset to better suit the indexing functions instead. This approach allows us to accommodate more keys with fewer indexing functions and hence creating shallower hierarchical structures. For this purpose, we propose the approach of CDF smoothing via virtual points. The goal of the approach is to insert virtual keys into the existing dataset in a manner that would minimise the loss function of the indexing function. This is illustrated by Fig. 2, where adding the virtual keys transforms the CDF of the dataset into a more linear shape.

Eq. 2 is the loss function of our optimisation problem. We aim to insert values (*virtual points*) into K while keeping it sorted, such that $\mathcal{L}_{\mathcal{F}}(K)$ is minimised, i.e., to *smooth the CDF* of K, thereby allowing more keys to be fitted with fewer indexing function at each level of the index.

A naive optimal smoothing scheme is to insert as many virtual points as needed such that every point $k \in K_i$ lies at the $f_i(k)$-th position (i.e., $rank(k) = f_i(k)$) in the list (assuming unique integer keys). This way, the loss becomes zero after smoothing. In reality, this smoothing scheme is infeasible, due to the non-uniqueness of the keys in K and the potentially high space cost. For example, if the keys are 64-bit integers, it will take $2^{64} \times 8$ bytes $\approx$ 128 exabyte (i.e., 128×10^6 TB) to achieve such an index key layout.

To balance between the space overhead and the smoothness of the CDF with inserted points, we consider a "smoothing budget" λ, i.e., the number of virtual points allowed to be inserted, such that the loss is minimised given the constraint of λ. We assume $\lambda = \alpha \cdot n$ where *the smoothing threshold* α is in $(0, 1)$, to retain a linear space overhead. Formally, we aim to solve the following problem:

Definition 3.1. [**Learned index smoothing**] Given a list of index keys K sorted in ascending order and partitioned into m segments, each of which is indexed by an indexing function $f_i \in \mathcal{F}$, the learned index smoothing problem aims to insert a set V ($|V| \leq \lambda$) of virtual points into K while keeping K in order, such that the loss as defined by Eq. 2 is minimised.

We consider linear indexing functions as they are used in most existing learned indexes and for their efficiency, although CDF smoothing can naturally extend to more complex (e.g., quadratic) functions. To simplify the discussion, we use integer index keys, while our techniques also apply to real number index keys when they can be scaled up to become integers. Similarly, Strings can

be converted into integers via an encoding such as using ASCII values prior to indexing [6, 39], hence our proposed method can be applied after this conversion. We omit duplicate keys as SOTA learned indexes such as LIPP and SALI does not support them.

NP-hardness analysis. Solving the exact CDF smoothing is NP-hard as it can be reduced from the Knapsack problem.

Lemma 3.2. *Learned index CDF smoothing is NP-hard.*

Proof. We reduce from the Knapsack problem which is NP-hard. The Knapsack problem considers a set of items S. Each item $s \in S$ has a value c_s and a weight w_s. The objective is to determine the subset $A \subseteq S$ that maximises the total value of the items in A while the total weight of the items is less than a given limit t.

CDF smoothing considers a key set K of size n. We aim to find a subset V (virtual points k_{vi}) of at most size λ from a candidate set C that would minimise the loss $\mathcal{L}$ (i.e., maximisation of loss reduction from the loss without CDF smoothing). Naively, the set C can be formed by considering λ candidates between every two adjacent keys in K, i.e., $|C| \leq \lambda \cdot (n-1)$. To reduce the Knapsack problem to our problem, the set S of items is mapped to C. We set the weight of every item (a candidate virtual point) to 1 and let t be our target number of virtual points to be added, i.e., λ. The value of an item, c_s, is mapped to the loss reduction contributed by the corresponding virtual point. Maximising the values of the items in subset A is mapped to maximising the total loss reduction of the virtual points in V. In our problem, the total loss reduction when multiple virtual points are added together varies from the sum of the loss reduction when the virtual points are added individually, represented as: $|\sum_{s\in A} c_s| = r \sum_{s\in A} |c_s|$, where $|c_s|$ is the magnitude of the value of a virtual point and $r \in \mathbb{R}$ is a parameter. Here, r is deterministic since it could be calculated based on Eq. 1. As such, the total value of the combined items can be transformed to the sum of the individual item values, and maximising the latter for the Knapsack problem can be mapped to maximising the former for our problem. The transformation between the two problems can be done in polynomial time since there is a one-to-one mapping. Thus, when our problem is solved, the Knapsack problem is also solved in polynomial time. As such, our problem is NP-hard. □

Due to the computational complexity in finding an exact optimal solution for the learned index smoothing problem over a large set of keys, next, we consider two practical variants of the problem and propose highly effective heuristic solutions: (1) smoothing the CDF for the subset of keys K_i indexed by an indexing function f_i (Section 4); and (2) smoothing the CDFs for all m subsets K_i when they are indexed under a hierarchical learned index (Section 5).

4 CDF SMOOTHING FOR A SINGLE LINEAR MODEL

We first consider a single indexing function over a segment of keys K_i. Given the smoothing budget λ, our optimisation goal is:

$$\text{argmin}_{V_i,w,b}\, \mathcal{L}_{f_{w,b}}(K_i \cup V_i), \quad s.t. |V_i| \leq \lambda \tag{3}$$

This optimisation goal varies from Eq. 2. Importantly, we now allow the slope (w) and intercept (b) of the indexing function f to be refitted based on the keys K_i (with adjusted ranks) and the inserted virtual points V_i, rather than just inserting virtual points to adjust $rank(k)$ for $k \in K_i$ to fit $f(k)$ of the original indexing function f. This way, intuitively, we could achieve a lower loss with fewer virtual points (hence reducing space overhead), as opposed to the naive optimal smoothing scheme described above that simply spreads K_i to fit $rank(k)$ to a given $f(k)$. In Eq. 3, we include V_i in the loss calculation, such that the storage space allocated to the virtual points can be used to accumulate data insertions, with minimized prediction errors when querying the inserted data.

Challenges. Allowing to refit the slope and the intercept of the indexing function makes the optimisation problem more difficult, as now the loss reduction brought by inserting a virtual point depends further on the other virtual points to be inserted. To solve the problem, a simple greedy heuristic is to iteratively select the best remaining candidate virtual point that leads to the largest loss reduction and to refit the indexing function after each virtual point selection. This process is repeated λ times to identify λ virtual points to be inserted. There are complexity issues with this simple heuristic. Challenge 1: Large size of candidate set: Given keys K_i, the candidate virtual points can be any key value in $(\min\{K_i\}, \max\{K_i\})$ (detailed in Section 4.2), which can be a large range in real datasets. Challenge 2: Repeated loss calculations: For each candidate virtual point k_v, we need to recalculate $\mathcal{L}_f(K_i \cup V_i)$, where V_i includes k_v, to help select the best candidate that leads to the largest loss reduction, which takes $O(|K_i| + \lambda)$ time. For λ iterations and $O(p)$ candidate virtual points per iteration, overall, the greedy solution takes $O((|K_i| + \lambda) \cdot \lambda \cdot p)$ time.

To overcome these challenges, we present an efficient solution below to reduce the overall time complexity through (1) reducing the number of candidate virtual points p and (2) the time cost to calculate the loss for each candidate virtual point.

Our solution. Our solution takes three steps: **1:** We propose an effective approach to reduce the number of candidate virtual points. The idea is, if there are consecutive candidate virtual points, we only need to consider the point among these consecutive points that minimises the loss, which can be identified utilising the first-order partial derivative of the loss function (addressing Challenge 1, detailed in Section 4.2). **2:** We propose an efficient algorithm to calculate the loss. The core idea is to calculate the loss incrementally, and to reuse part of the calculation from the previous iteration, as only one new candidate virtual point is included into the calculation each time (addressing Challenge 2, detailed in Section 4.1). **3:** Finally, from the reduced set of candidate virtual points, and with efficient loss calculation, we present an efficient algorithm to compute the best subset of virtual points of size λ (Section 4.3).

A running example on finding the best virtual point is shown in Fig. 3, which corresponds to the keys in Fig. 2a. In Fig. 3, each hollow dot represents a candidate virtual point. Its y-value represents the loss (i.e., SSE) if the virtual point is included into the key set. Adjacent hollow dots are linked together, forming a segment of candidate virtual point values. For example, the segment formed by 21 to 25 is between index keys 20 and 26 (the index keys themselves are *not* considered to be candidate virtual points; they correspond to the gaps between the curve segments in Fig. 3). The goal is to search for the virtual point with the smallest loss value. In the given example, point 23 is the search target.

In what follows, we present the algorithm to efficiently calculate the loss first (Section 4.1), based on which the strategy to reduce the candidate virtual points is described (Section 4.2), and our overall algorithm to compute the best λ virtual points is detailed (Section 4.3).

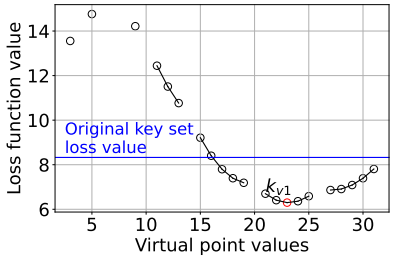


Figure 3: Loss function (SSE) value corresponding to different insertion positions for a virtual point.

4.1 Efficient Loss Calculation and Indexing Function Refitting

We require to calculate the loss after each candidate virtual point. However, doing so would be computationally expensive. As such the idea is to reuse the calculations as much as possible. For the rest of Section 4, we abuse the notation slightly and use K instead of K_i to denote the segment of keys for which CDF smoothing is to be done, since the discussion only concerns a single segment. Similarly, we use f instead of f_i to denote the indexing function, $\mathcal{L}$ instead of $\mathcal{L}_i$ to denote the loss function, and V instead V_i to denote the candidate virtual points for the segment.

Let the rank of a key k_i be y_i (assuming that the ranks start from 0) for the n keys in K, i.e., $y_i = rank(k_i)$. First, consider a virtual point k_v with its rank (after inserted into K) being y_v (we explain the case for multiple virtual points later). The loss for K and $V = \{k_v\}$ can be calculated as follows, where w and b are the parameters of the indexing function f for the existing and inserted keys:

$$\mathcal{L}(\{K \cup V\}) = \sum_{i=1}^{n} (wk_i + b - y_i)^2 + (wk_v + b - y_v)^2 \tag{4}$$

It is computationally expensive to calculate the loss for each candidate virtual point using standard Eq. 4, as w and b can change when the indexing function f is refitted for the candidate virtual point, which is governed by the following equations (derived from the first-order partial derivatives of Eq. 4):

$$w = \frac{\sum_{i=1}^{n+1}(k_i - \bar{k_v})(y_i - \bar{y_v})}{\sum_{i=1}^{n+1}(k_i - \bar{k_v})^2}, \; b = \bar{y_v} - w\bar{k_v}. \tag{5}$$

Here, $\bar{k_v}$ and $\bar{y_v}$ are the mean of the key set (i.e., $K \cup V$) and the rank set after inserting the virtual point (k_v, y_v), respectively. They can be computed by Eqs. 6 as follows:

$$\bar{k_v} = \frac{\sum_{i=1}^{n} k_i + k_v}{n+1}, \; \bar{y_v} = \frac{\sum_{i=1}^{n} y_i + y_v}{n+1}. \tag{6}$$

$\sum_{i=1}^{n} y_i$ is the summation ranks in the original key set (i.e., K) after the inclusion of the virtual point k_v.

To make the calculation efficient we rewrite Eq. 4 for calculating the loss after each candidate virtual point such that the candidate virtual point k_v is separated from the values of K. This enables us to calculate the terms in the equation related to K separately and then reuse their values for different candidate virtual points. This would reduce the time when computing the loss for different candidate virtual points.

By expanding Eq. 4 we obtain Eq. 7

$$\begin{aligned}\mathcal{L}(\{K \cup V\}) = w^2 \sum_{i=1}^{n} k_i^2 + 2wb \sum_{i=1}^{n} k_i \\ -2w \sum_{i=1}^{n} k_i y_i + nb^2 - 2b \sum_{i=1}^{n} y_i + \sum_{i=1}^{n} y_i^2 + (wk_v + b - y_v)^2\end{aligned} \tag{7}$$

By adding the new point with rank y_v, we would be increasing the rank of all keys after y_v by one, until the final key's rank is n. This results in the following Eq. 8, where $\sum_{i=1}^{n} y_{original_i}$ is the ranks of the original key set prior to the virtual key.

$$\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} y_{original_i} + n - y_v, \tag{8}$$

Similarly, we can define the means of the original key set and rank after the adding k_v as,

$$\bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}, \; \bar{k} = \frac{\sum_{i=1}^{n} k_i}{n} \tag{9}$$

Substituting Eqs. 8 to 9 into Eq. 7, we can simplify the loss function as shown in Eq. 10.

$$\begin{aligned}\mathcal{L}(\{K \cup V\}) = w^2 \sum_{i=1}^{n} k_i^2 + 2wbn\bar{k} - 2w \sum_{i=1}^{n} k_i y_i + nb^2 \\ -2nb\bar{y} + \sum_{i=1}^{n} y_{original_i}^2 + n^2 - y_v^2 + (wk_v + b - y_v)^2\end{aligned} \tag{10}$$

Similarly, we can expand the slope from Eqs. 5 to simplify the calculation of the indexing function.

$$w = \frac{\sum_{i=1}^{n}(k_i y_i) + k_v y_v - (n+1)\bar{k_v}\bar{y_v}}{\sum_{i=1}^{n} k_i^2 + k_v^2 - (n+1)\bar{k_v}^2} \tag{11}$$

Adjustment for multiple virtual points. In the case of inserting λ virtual points, after inserting one virtual point (k_{v1}, y_{v1}), to find the next virtual point (k_{v2}, y_{v2}), the original key set terms will be adjusted to include the newly added virtual point (k_{v1}, y_{v1}). As a result, the loss for the next candidate virtual point k_{v2} can also be efficiently calculated by considering only the changes induced by k_{v2} and its corresponding y_{v2}.

In the derived equations above, the terms are separated based on whether they belong to the original key set or dependent on the candidate virtual key to be inserted. Doing so enables efficient calculations of the loss by reusing the terms of the original key set after calculating them just once.

4.2 Filtering Virtual Point Candidates

Next, we present an efficient approach to identify the candidate virtual points that can potentially reduce the loss, thus providing a much smaller set of candidate virtual points to consider. This step is important because while using the equations above helps improve the efficiency of processing one candidate virtual point, the number of candidate virtual points to be processed has a multiplicative impact to the overall algorithm time efficiency.

To reduce the search space for the candidate virtual points, we bound it in $(\min\{K\}, \max\{K\})$. This is because any virtual points added prior to $\min\{K\}$ would cause all keys' ranks to increase at the same time, while adding virtual points after $\max\{K\}$ would not impact any key's rank. As such, neither would help achieve a better-fitted indexing function. We also skip the index keys already in K, such that our solution can be compatible with learned indexes that do not support duplicate keys [29].

Below, we present an approach based on the derivative of the loss function to further reduce the set of candidate virtual points. Our idea is illustrated using Fig. 4, which plots the partial derivative of the loss function with respect to a candidate virtual point k_v. Each sub-sequence (depicted as lines or dots) corresponds to the partial derivative of the loss of a sub-sequence of key values of a candidate virtual point (which can also be seen as a sub-sequence of candidate virtual points). The sub-sequences that cross the zero-value line (i.e., the x-axis) imply that there is a minimal loss point within the sub-sequence. Otherwise, the minimal

point for the sub-sequence must be at one of two endpoints of the sub-sequence, since each of such sub-sequences has been shown to be convex [12]. Intuitively, this convex property is because the loss function is a summation of n quadratic terms ($(wk_i+b-y_i)^2$) and only one non-quadratic term ($(wk_v+b-y_v)^2$), and the wk_v term is nonlinear variable. We exploit this property to streamline the selection of candidate virtual points, i.e., to select the best virtual point from each sub-sequence.

Following the idea above, we propose to filter the candidate virtual points as follows: (1) For each sub-sequence of candidate points (that is, where the candidate virtual point values are continuous), if the length of the sub-sequence is greater than 2, there can be a candidate virtual point within the sub-sequences with a local minima of the loss. We compute the partial derivative of the loss function, which includes the previously added virtual points ($\mathcal{L}(\{K \cup V\})$ shown in Eq. 4) with respect to candidate virtual point k_v, i.e., $\mathcal{L}(\{K \cup V\})'$ of the two endpoints of such sub-sequence (we present efficient ways to compute the partial derivative of a candidate later). (a) If the sign of $\mathcal{L}(\{K \cup V\})'$ of the two endpoints are the same, it means that there is no point with the local minima within this sub-sequence (i.e.,, the sub-sequence does not cross the zero value in y-axis as shown in Fig. 4). In that case, we only need to consider the two endpoints of the sub-sequence as the candidate virtual points, and we can safely discard all the candidate virtual points in between. (b) Otherwise, if the sign of $\mathcal{L}(\{K \cup V\})'$ of the two endpoints are opposite, it means that there is a point with a local minima within the sub-sequence. The minimal point can be calculated by using the two partial derivative values to find their intersection with the x-axis. As this minimal point is guaranteed to have a smaller loss than all the other points in that sub-sequence, only the point is considered as a candidate virtual point. All other points in the sub-sequence can be safely discarded.

(2) If the length of the sub-sequence is less than or equal to 2, we need to consider all points in the sub-sequence as the candidate virtual points.

For datasets that allow duplicate keys, potential virtual points would not be in sub-sequences divided by the existing points. Instead, now all key values become candidate virtual points. We could use a gradient descent approach to find the local optima for virtual point insertion.

Efficient computation of the first-order derivative of the loss function value. Similar to the computation of the loss function, we present an efficient way to calculate the first-order partial derivative of the loss function with respect to candidate virtual point k_v, where k_v and its related terms are separated from the other terms to enable reusing the terms that require information from K only. This is achieved by the following equation:

$$\mathcal{L}(\{K \cup V\})' = 2(w'(w\sum_{i=1}^{n} k_i^2 + nb\bar{k} - \sum_{i=1}^{n} k_i y_i) + nb'(w\bar{k} + b - \bar{y}) + (wk_v + b - y_v)(w'k_v + w + b')). \quad (12)$$

Here, w' and b' refer to the partial derivatives of w and b with respect to k_v, respectively. They can be computed by Eqs. 13 as follows.

$$w' = \frac{A(n(y_v - \bar{y})) - B(2n(k_v - \bar{k}))}{A^2}, \; b' = -\frac{(w + (n+1)\bar{k_v}w')}{n+1} \quad (13)$$

$$A = (n+1)(\sum_{i=1}^{n} k_i^2 + k_v^2) - ((n+1)\bar{k_v})^2, \quad (14)$$

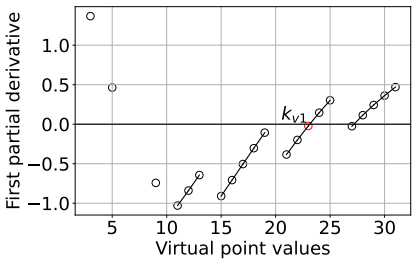


Figure 4: First-order partial derivatives of the loss (Eq. 12) with respect to the key value of a virtual point k_v

$$B = (n+1)(\sum_{i=1}^{n} k_i y_i + k_v y_v) - (n+1)^2 \bar{k_v}\bar{y_v}. \quad (15)$$

A and B are intermediary for the partial derivatives calculation.

4.3 Algorithm for Inserting λ Virtual Points

After filtering the candidate virtual points, among the remaining ones, we present an efficient algorithm to find the best subset of candidate virtual points of size λ. When there are λ virtual points to insert, the optimal solution would require computing the loss for every size-λ subset of the candidate virtual points in the range of $(\min\{K\}, \max\{K\})$. If there are p possible insertion positions for the virtual points, the time complexity will be $O({}^pC_\lambda \cdot n \cdot p)$, where ${}^pC_\lambda$ is the combination of every size-λ subset from p. As this will be prohibitively expensive for a large dataset, we propose a greedy algorithm that inserts individual virtual points iteratively.

Our core idea is to identify the virtual point that would minimise the loss for each sub-sequence (i.e., local minima for the sub-sequences) and select the one that reduces the loss the most (i.e., global minimum). This process needs to be performed λ times. The algorithm for CDF smoothing by inserting λ virtual points is summarised in Algorithm 1 and described below.

Algorithm. Our algorithm takes as input a key set K and a smoothing threshold α (or a smoothing budget $\lambda = \alpha n$). We use G to denote a set that stores the potential sub-sequences, where there can be a candidate virtual point within the endpoints of the sub-sequences with a local minima of the loss. The endpoints of the candidate virtual points contributing to this local minima are stored in an array M, while C stores the set of candidate keys for the virtual points. We use U to hold the loss for each candidate virtual point and vector V to store the final optimal virtual points.

First, the algorithm identifies the sub-sequences of candidate virtual points that could have their minimal loss at the endpoints or in-between the sequence. This is shown in Lines 4 to 12. If there are more than two points in a sub-sequence, the candidate virtual point with the minimal loss can be within that sub-sequence. As such, the two endpoints of the sub-sequence are saved in array G for calculating the partial derivatives.

Afterwards, in Lines 13 to 22, the partial derivative of the loss function with respect to the candidate virtual points is calculated for all point pairs in G using the equations derived above. If the signs of the partial derivatives corresponding to the two endpoints of a sub-sequence are different (i.e., on opposite sides of the x-axis), the two endpoints are added to M for calculating the minimum point. As shown in Fig. 4, for the sub-sequences that contain candidate virtual points with minimal loss, the partial

Algorithm 1 CDF_smoothing

Require: Key set: K, loss function with new virtual point: $\mathcal{L}(K \cup k_v)$, smoothing threshold: α

1: U = [ ], C = [ ], V = [ ]
2: G = [ ], M = [ ] ▷ Arrays of point pairs
3: $\mathcal{L}'(K \cup k_v) = \frac{\partial L(K \cup k_v)}{\partial k_v}$, $\lambda = \alpha \cdot K.size$, $\mathcal{L}_{previous} = \mathcal{L}(K \cup \varnothing)$
4: Find the endpoint pairs, E, for each sub-sequence
5: **while** $V.size < \lambda$ **do**
6: **for** i from 1 to $E.size$ **do** ▷ Separate sub-sequences
7: **if** $E[i].sec$ - $E[i].fir \leq 1$ **then**
8: Append $E[i].fir$ and $E[i].sec$ to C
9: **else** ▷ There are more than 2 points
10: Append $E[i].fir$ to $G.fir$ and $E[i].sec$ to $G.sec$
11: **end if**
12: **end for**
13: **for** i from 1 to $G.size$ **do** ▷ Calculate the partial derivatives
14: **if** $\mathcal{L}'(K \cup G[i].fir) \cdot \mathcal{L}'(K \cup G[i].sec) < 0$ **then**
15: Append $G[i].fir$ to $M.fir$ and $G[i].sec$ to $M.sec$
16: **else**
17: Append $G[i].fir$ and $G[i].sec$ to C
18: **end if**
19: **end for**
20: **for** i from 1 to $M.size$ **do** ▷ Calculate minimum point
21: Append $minimum\_point(M[i].fir, M[i].sec)$ to C
22: **end for**
23: **for** i from 1 to $C.size$ **do** ▷ Calculate loss value
24: $U[i] = \mathcal{L}(K \cup C[i])$
25: **end for**
26: Find index i of minimum $\mathcal{L}$
27: **if** $\mathcal{L}_{previous} \leq U[i]$ **then**
28: break
29: **end if**
30: Append $C[i]$ to V, Append $C[i]$ to K, $\mathcal{L}_{previous} = U[i]$
31: **end while**
32: **return** C

derivatives of the endpoints will appear on the two sides of the x-axis. These minimal points are added to C after they are calculated. If the partial derivatives of the two endpoints are on the same side of x-axis, the minima is at one of the endpoints, so the two endpoints are added to C.

Finally, Lines 23 to 31 compute the loss for each candidate virtual point in C and select the point with the minimum loss, as long as the new loss is smaller than the loss obtained so far.This process is repeated until at most λ virtual points are inserted, or when the loss is not reduced any further. When the algorithm terminates, the final virtual points in V are returned.

Complexity analysis. CDF reduces the computation of loss over K to just once, which takes $O(n)$ time. This process is repeated to find λ optimal candidate virtual points. However, there is no need to recalculate the loss function after adding a virtual point, as we could treat the key set with the previous virtual point inserted as the new original or base key set for a constant time calculation. Thereby, giving a time complexity of $O(\lambda + n)$.

4.4 Approximation Quality Analysis

The effectiveness of our proposed greedy method of iteratively identifying the λ virtual points as opposed to the exhaustive manner of comparing all λ subsets, is demonstrated via experimentation in this section. The key set of 10 keys given in Fig. 2 was subjected to CDF smoothing with a smoothing threshold (α) of 0.5 (smoothing budget of 5) via both methods. Here, the greedy method improves the loss by 72.34%, while the exhaustive method improves it by 74.44%. However, the time taken by the exhaustive method is nearly 3 orders of magnitude more than the greedy method. This results show that the effectiveness of the greedy method is similar to the exhaustive method, and the exhaustive method is impractical to use in real datasets.

5 CDF SMOOTHING FOR HIERARCHICAL INDEXES

In this section, we present the CDF smoothing to a hierarchical learned index to improve the performance of queried keys. A direct application to individual nodes would help reduce leaf-node search time by improving the learnability of the models but fail to address traversal time. Therefore, a method for addressing both traversal and leaf-node search is required. As such we present *CSV* to smooth segments of the CDF for different sub-trees in the hierarchical structure of a learned index in order to merge and reduce the overall structure height. A major challenge is the balancing between the improvement of traversal time due to the reduction of the index height and the increase in leaf-node search time due to more keys being merged into single nodes. To address this, we present a cost model that takes both of these factors into consideration.

The core idea is to start from the bottom most level of the index that contains parent nodes of leaf-nodes and select those nodes. Then for each of these parents nodes, the keys in the node and its child nodes are collected, which are then subjected to smoothing using Algorithm 1. If the minimum cost threshold is satisfied (more details below), then the sub-tree and the node are reconstructed to merge the collected nodes. The merging is performed by creating a new leaf-node in place of the parent node and placing the keys from the collected nodes. By doing so, more keys would be placed in upper level nodes of the index as the indexing functions of these nodes would be improved by the CDF smoothing, but the cost models would limit the number of keys as to not offset the performance gain by the increase in the leaf-node search time. Further details regarding this is given in Section 5.1. This process is performed until the root node depicted as level 1 is reached, thus reducing the loss ($\mathcal{L}$).

As shown in Fig. 5, the original keys from the selected node and its sub-tree are not fitting well to the indexing function used. However, after the inclusion of the virtual points, the indexing function becomes more accurate. Due to this, the selected node manages to accommodate more keys in the node. As such, the keys that were previously in the level below have the potential to be placed or promoted to a higher level.

This process is applied to a constructed learned index structure as the purpose of the method is to enhance the structure of the index. Further, it is computationally expensive to perform the smoothing operation on the full key set. As such it is more reasonable to handle subsets of the key set in the constructed hierarchical index. For this purpose, unbalanced learned index structures are better suited as it gives the ability to reduce the height of taller branches without affecting the rest.

The algorithm for CDF smoothing of a hierarchical learned index structure is given in Algorithm 2 and described below. First the algorithm starting from the maximum level of the index, identifies all nodes with sub-trees in the level, which is shown in Line 5. Afterwards, as shown in Lines 7-8, each of the node's and its sub-tree's keys are collected and subjected to the CDF smoothing. Provided that they meet the minimum cost threshold selected, the node and its sub-tree are reconstructed to promote

Algorithm 2 CSV

Require: Nodes with sub trees : $Nodes$, smoothing threshold : α, cost threshold : c

1: $Nodes = [\,], keyset = [\,], keyset\_smooth = [\,]$
2: $max\_level \leftarrow$ maximum level of index with sub trees
3: $current\_level \leftarrow max\_level$
4: **while** $current\_level > 1$ **do**
5: $Nodes \leftarrow$ all nodes with sub trees
6: **for** i from 1 to $Nodes.size$ **do**
7: $keyset \leftarrow$ collect all keys in the node and its sub tree
8: $keyset\_smooth \leftarrow CDF\_smoothing(keyset, \alpha)$ ▷ Using Algorithm 1
9: **if** $cost < c$ **then**
10: Reconstruct the sub-tree and node with $keyset\_smooth$
11: **end if**
12: **end for**
13: $current\_level \leftarrow current\_level - 1$
14: **end while**

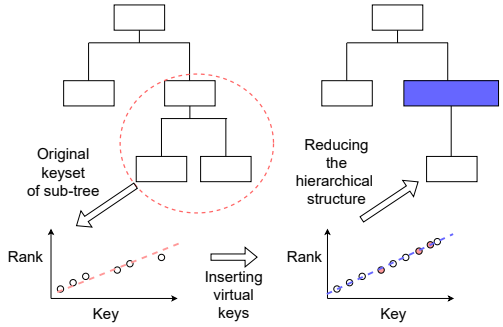


Figure 5: The CSV method

as many keys to upper levels as possible as shown in Lines 9-11. This process is iteratively performed in a bottom up manner for other levels of the index.

The CSV method is applied after the construction of the initial learned index structure for two reasons: (1) this makes our method more versatile and applicable to different learned indexes; (2) this avoids the high cost of repeating the optimal virtual point calculation process every time a new node is created (recall the NP-hardness of the problem). It would make an interesting future work to further explore how to efficiently run virtual point insertion during index construction, which could open further optimisation opportunities.

5.1 Cost Conditions

For indexes that do not contain any searching component such as LIPP and SALI, their loss function values can be taken as the cost conditions. This is because if the new model could hold more keys than before, then it does not have any other component (that is, leaf-node search time) that would negatively affect the performance. However, for the indexes with leaf-node search components like ALEX, there must be a trade-off between the increase of leaf-node search time over the reduction of traversal time. The reason is, introducing new keys into the node would require more time to locate the key. For this purpose, we develop the following cost model, where reconstruction is performed only if the cost is less than a specified threshold value, c.

$$cost = search\_constant \times expected\_number\_of\_searches + traversal\_constant \times index\_level \quad (16)$$

To make the implementation hardware independent, the constants can be measured by sampling queries to measure the time spent per leaf-node search for $search\_constant$ and the traversal time spent per level for $traversal\_constant$. The $expected\_number\_of\_searches$ can be calculated via the inbuilt function in ALEX that uses the log_2 error to estimate it. Considering the cost model depicts the expected query time for the node, the cost threshold, c should be set below 0 to identify an improvement. Setting a lower value would result in fewer keys being able to be promoted to upper levels but the expected query time improvement will be greater.

Complexity analysis. For a key set of size n, a smoothing budget λ and an index with m non-leaf nodes, the complexity of the algorithm can be calculated as follows. The complexity for $node_i$ with n_i keys and a smoothing budget of λ_i is $O(\lambda_i + n_i)$. Similarly, for the m nodes, we would get a complexity of $O(\lambda_1 + n_1 + \lambda_2 + n_2 + \cdots + \lambda_m + n_m)$. This can be simplified to $O(\lambda + n)$.

Choice of smoothing threshold. Increasing the smoothing threshold would make the algorithm insert more virtual points, thus reducing the loss function value of the newly fitted model more. As a result these indexing functions would be able to accommodate more keys which were originally in lower levels of the index and improve their query time. However, that is a trade-off between the query time improvement and the higher space cost for increasing this threshold. Further, key sets that cause the original index structure to construct poorly should benefit more from a higher smoothing threshold as shown in the experiments 6.2.1.

6 EXPERIMENTAL RESULTS

Next, we report experimental results. The implementation of our evaluation framework is based on an existing benchmark [1] implementation. All experiments were run on an Ubuntu 20.04.5 virtual machine with an AMD EPYC 7763 64-Core CPU and 128 GB RAM. The CSV pre-processing was performed in parallel with 32 threads, while all other aspects of the experiments were performed in a single threaded environment.

6.1 Experimental Settings

Competitors. To show the general applicability of our proposed techniques, we integrate **CSV** with recent learned indexes, including **ALEX** [2], **LIPP** [34], and **SALI** [10] (SOTA). These three indexes were chosen because they are the latest and among the most widely used benchmark learned indexes. They have reported strong empirical performance, outperforming both traditional indexes such as the B$^+$-tree and learned ones [29, 33] such as the PGM index [7], XIndex [30], and FINEdex [15]. For simplicity, we do not repeat the comparison results with these other indexes. We also compare our method with the two most comparable baselines, NFL and Gap insertion method. The implementation for NFL was available, however, Gap insertion method had to be implemented, according to the descriptions given in the paper.

ALEX is a learned index with two types of nodes, i.e., internal nodes and data nodes. An internal node contains an indexing function and pointers to its children nodes, while a data node contains an indexing function and data points. It utilises gapped

arrays within each data node to accommodate future data insertions. ALEX uses a cost model to calculate the accuracy of an indexing function. If the accuracy is high, a data node is formed with the function, otherwise an internal node is formed and the corresponding data subset is split further.

Similarly, LIPP also uses gapped arrays for data nodes to support insertions. However, LIPP does not use special internal nodes i.e., all nodes in LIPP contain data points. Further, there is no searching within each node at query time, as LIPP resorts to perfect predictions. This is achieve by collecting keys that are predicted to the same location in a node and creating a new data node at the next level of the index to host such keys. This process is repeated until there are no collisions. SALI is based on LIPP but modified to support concurrent queries and updates.

All three learned indexes resort to hierarchical structures due to the errors in the indexing functions, while our CDF smoothing technique can help reducing the depth of the structures by reducing the errors.

Datasets. We run experiments with six datasets from two benchmark works [21, 33]: (1) **Facebook** contains 200 million integer Facebook user IDs [31]; (2) **Covid** contains 200 million integer tweet IDs randomly sampled from tweets tagged with "Covid-19" [19]; (3) **OSM** contains 200 million locations randomly sampled from OpenStreetMap and represented as Google S2 [27] cell IDs [25]; and (4) **Genome** contains 200 million entries of loci pairs in human chromosomes represented as integers [26]. (5) **Books** contains 200 million book sale popularity data as integers [21]. (6) **Uniform** contains 200 million uniformly distributed sparse integers [21]. For all datasets, duplicate keys were removed to suit LIPP and SALI's requirements. Other datasets that were available in the SOSD benchmark such as WIKI, lognormal and normal datasets were excluded from the experiments due to their key distribution being fairly uniformly distributed in each subtree and as such does not contain many keys in lower levels for our CSV method to promote. Furthermore, in the ALEX index Books and Uniform manage to place nearly all data in level 1 or level 2, as such they were omitted from the ALEX results.

Out of the datasets, OSM and Genome are considered more difficult for learned indexes [33] (hard datasets), while Facebook, Covid, Books and Uniform are easier (easy datasets). All datasets except OSM have almost globally linear CDFs with more variability in the local distribution patterns. Except for Covid, all datasets deviate from linear CDFs at local level [21, 33].

Workloads. We use the following two types of workloads: (1) *Read-only workload.* The learned indexes ALEX, LIPP, and SALI are constructed over the full datasets. Afterwards, our CSV algorithm is applied to optimise their structures. Then, the queries (detailed below) are run. (2) *Read-write workload.* The learned indexes are constructed over a random half of each dataset and then CSV is applied. The other half of the dataset is inserted in random batches of size $0.1n$. Queries are run after each batch insertion without using CSV again for each batch.

Queries. Considering the main objective of the developed method is to improve the performance of keys in lower levels, the experiments are focused on them. Specifically, we report results for the **promoted data**, which includes every key that has been promoted to upper levels in the index by our algorithm.

Parameters. We vary the smoothing threshold, α, from 0.05 to 0.8, with a default value of 0.1. To show the scalability of our algorithm by varying the dataset size, the original datasets were down sampled by eliminating every j-th key from the sorted datasets in order to remove n/j data points and create smaller datasets of size 12.5 million, 25 million, 50 million, and 100 million, respectively. The default datasets are the original ones with 200 million points.

For each queried key, the query time was recorded by repeating the query 25 times and taking its average.

For LIPP and SALI, they can create nodes that are indexing only a few keys [34]. For these two indexes, CSV is run starting at the second level of the index structures, such that each smoothing step can benefit more points. This is not an issue for ALEX, and CSV is run starting at the bottom level. Further, since the query times of the keys in the top two levels of the index structures are very close, CSV stops at the second level from the top (i.e., the root).

Evaluation metrics. We report: (1) the **total query times saved** by the CSV-enhanced indexed compared with those of the original indexes; (2) the **query time improvement**, which is the average query time (over all queried keys) of the CSV-enhanced indexes and that of the original ones (depicted as $\alpha = 0$); (3) the **promoted data**, which is the number of keys promoted to upper levels in the index structure among all keys that can be promoted (i.e., keys at levels 3 or below of the original indexes); (4) the **storage space increase** , which is the index size overhead of the CSV-enhanced indexes kain bytes; (5) the **node reduction**, which is the number of index models reduced by the CSV-enhanced indexes over the original ones; and (6) the **insert time increase**, which is the increase in the average time per insertion required by the CSV-enhanced indexes compared to the original ones.

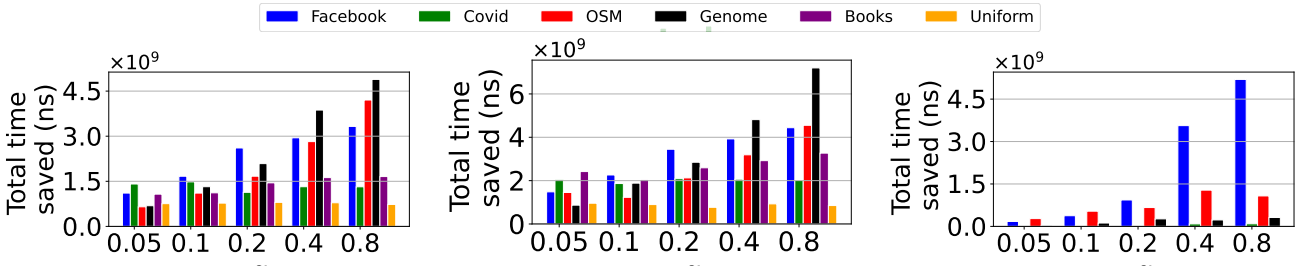
6.2 Results on Read-only Workloads

6.2.1 Impact of Smoothing Threshold. We vary the smoothing threshold from 0.05 to 0.8 to quantify its impact.

Query time (for promoted data). Here, we report the query time improvement by CSV for the 'promoted keys' (i.e., the keys that is promoted to an upper level of the index by CSV), compared to the original index. We depict the total time saved due to the method in Fig. 6. The general trend is that adding more virtual points (i.e., increasing the smoothing budget, α) saves more query times. LIPP and SALI tend to perform quite similarly due to SALI using LIPP as the base index. For LIPP and SALI indexes, the easy to learn datasets (Facebook, Covid, Books and Uniform) stabilise after a certain number of virtual points are inserted. This is because the original datasets' CDFs are already quite linear. The same pattern was not observed for ALEX, this is because ALEX has an additional leaf-node search step not required by LIPP and SALI (CSV forms larger nodes that could lead to longer leaf-node search times).

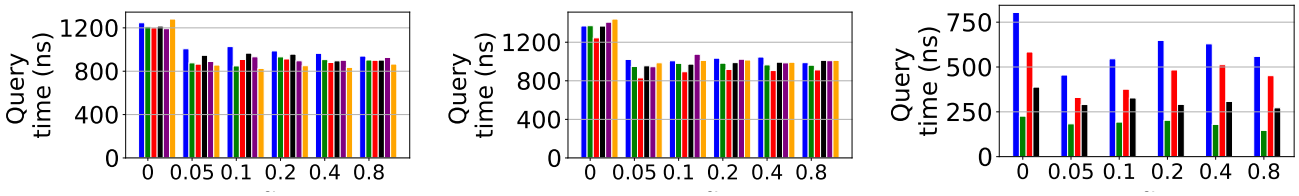
Fig. 7 reports the average query over promoted data. It shows that applying CSV yields a query time improvement of up to 34%, with stronger benefits over the two SOTA indexes, LIPP and SALI. Smaller performance gain is observed over ALEX due to its leaf-node search process. It is important to note that CSV still yields consistent query time improvements in this case. In ALEX, there are small fluctuations due to the cost model obtaining its constants via runtime. Since there is no such cost model in LIPP and SALI, their query performance is stagnant and the improvement represents the reduction in the index traversal time for query processing.

Fig. 8 shows the average query time for randomly selected data from the entire key set. The *CSV* optimised indexes demonstrate similar performance to the original index structure (shown as $\alpha = 0$). This is due to the main goal of the proposed method



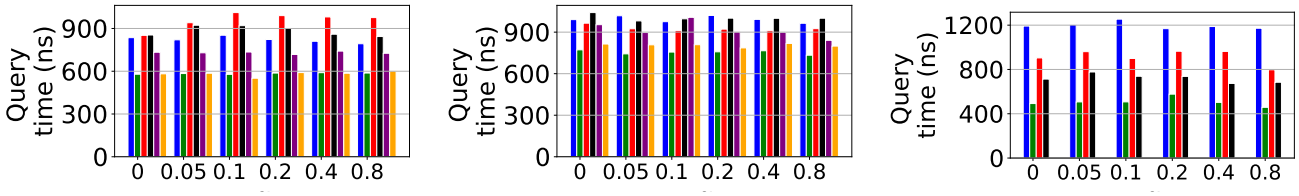
(a) LIPP (b) SALI (c) ALEX

Figure 6: Total time saved vs. smoothing threshold α



(a) LIPP (b) SALI (c) ALEX

Figure 7: Query time improvement vs. smoothing threshold α



(a) LIPP (b) SALI (c) ALEX

Figure 8: Average query time for random data vs. smoothing threshold α

of improving keys that perform worse. However, due to many of the keys being in level 1 and 2 of the index, majority of the queried keys tend to be selected from these levels, as such the impact of improving worse case keys cannot be seen in this kind of experiment.

Size of the promoted data. Due to the similar trends between LIPP and SALI indexes, SALI is omitted from the results below for brevity.

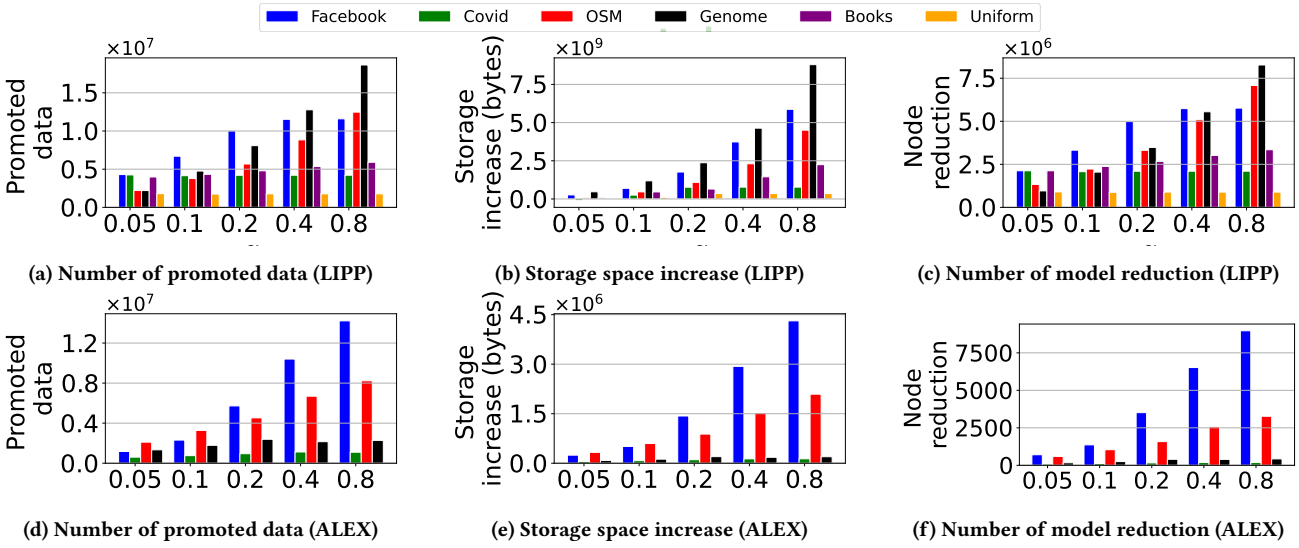
Figs. 9a and 9d show the percentage of keys promoted. For the Facebook dataset, CSV can promote around 60% of all promotable data (i.e., keys at level 3 or below of the original index), while for the Covid dataset, CSV promotes around 30% of the promotable data. For the harder to learn datasets, OSM and Genome, CSV also manages to promote up to 27% and 57% of the promotable data, respectively. Books and Uniform promoted 15% and 30% respectively. The datasets with the most promoted data are again different for ALEX, due to its structural difference. Overall, as the smoothing threshold increases, more keys get promoted to upper levels. This is consistent with the theoretical analysis as adding more virtual points would allow more keys to be placed in nodes in upper levels. The difficult to learn key sets (OSM and Genome) demonstrate this property the most. This can also be matched with Fig. 6, where the total time saved for OSM and Genome is higher due to more data being promoted for those datasets, compared to the other datasets.

Index size. Due to the addition of virtual points, we expect the storage space consumption to increase. This is reflected in Figs. 9b and 9e. In most cases, less than 10% of additional storage space is required by the CSV-enhanced indexes compared to the original structures, and in the worst case, less than 31%. The space cost overhead is proportional to α, which is also intuitive. The storage space increase is balanced by the removal of unnecessary nodes, whose data is promoted to higher levels. Figs. 9c and 9f report the node reductions achieved by CSV. They follow similar patterns to the percentage of promoted data as expected.

Pre-processing time for CSV. The times taken to run CSV to optimise the learned index structures are summarised in Tables 2 and 3 for LIPP and ALEX, along with the original index construction times in 'Original' column.

CSV takes more time to run as α grows, which is consistent with our time complexity analysis. The algorithm running times vary across different datasets, again because the datasets have different difficulties in index learning.

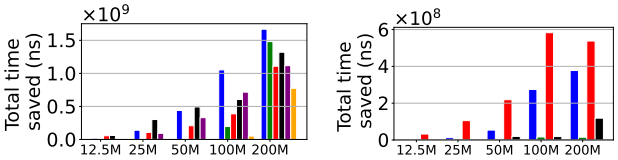
To amortize extra construction times with the benefits in query times, it takes about a billion queries. This number sounds huge, but it is mainly because learned indexes are already extremely fast for a single query (e.g., 844 ns per query on the Facebook dataset). We emphasize that the extra construction times are one-off pre-processing costs, and that our technique can benefit query-time sensitive applications. One may further mitigate the



(a) Number of promoted data (LIPP) (b) Storage space increase (LIPP) (c) Number of model reduction (LIPP)

(d) Number of promoted data (ALEX) (e) Storage space increase (ALEX) (f) Number of model reduction (ALEX)

Figure 9: Space cost vs. smoothing threshold α



(a) LIPP (b) ALEX

Figure 10: Total time saved vs. dataset cardinality

impact of extra construction times by using original indexes for queries while constructing a parallel index structure with CSV. Once the CSV optimised structure is ready, it is switched on for query processing.

6.2.2 Impact of Dataset Cardinality. The default smoothing threshold of 0.1 was used for these experiments. To demonstrate the scalability of CSV against the dataset cardinality, we repeat the experiments on datasets of 12.5 million to 200 million data points. Fig. 10 shows the total query times saved by applying CSV. For all datasets, the times saved grow with the dataset cardinality, with faster growth being observed on the easier datasets (Facebook and Covid) grows faster. This is because there are not many keys in the lower levels for these datasets when the dataset cardinality is small. These results confirm the scalability of CSV towards dataset cardinality.

Table 2: CSV Pre-processing Time (s) for LIPP

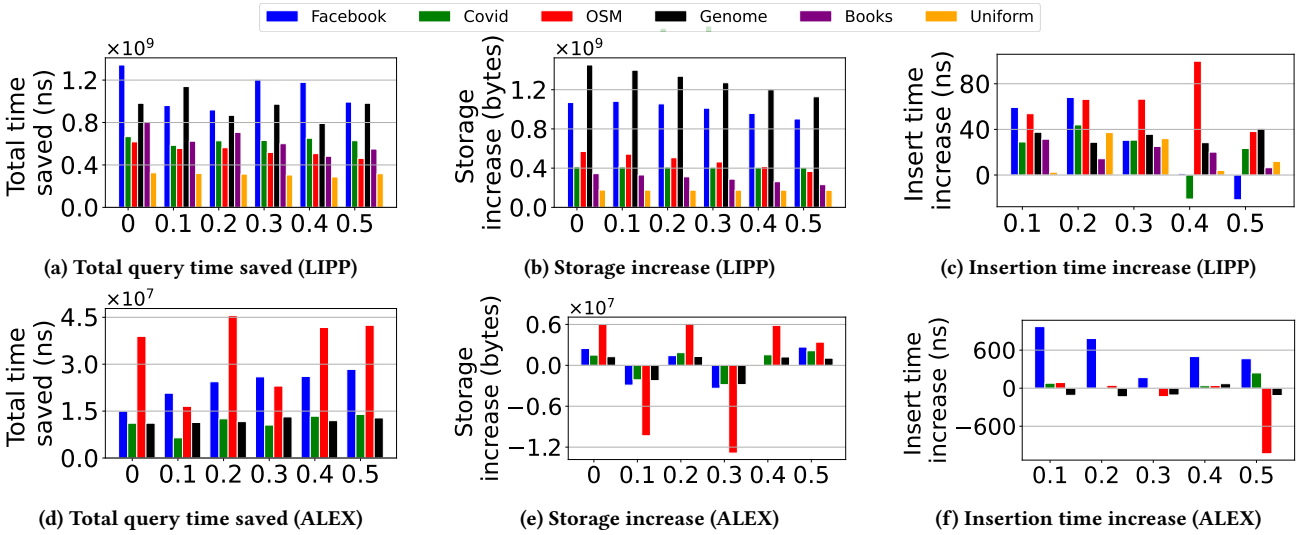
	Original	0.05	0.1	0.2	0.4	0.8
Facebook	52	76	90	132	269	636
Covid	53	101	103	103	152	346
OSM	46	124	166	236	414	870
Genome	52	114	149	221	401	956
Books	63	152	158	149	182	327
Uniform	25	45	46	47	74	191

Table 3: CSV Pre-processing Time (s) for ALEX

	Original	0.05	0.1	0.2	0.4	0.8
Facebook	546	110	206	421	890	1823
Covid	373	123	224	311	383	392
OSM	389	1265	1972	5400	9684	20158
Genome	366	225	402	615	696	713

6.2.3 Comparison with baselines. Fig. 12 shows the average query time for randomly selected queries. The LIPP, SALI and ALEX depicts the performance of the CSV enhanced indexes while GI indicates the use of the Gap insertion method instead of the virtual points insertion. Gap insertion method was designed to address the leaf-node search time, in data nodes. But considering LIPP and SALI does not have any data nodes but many small nodes, the Gap insertion was performed in a similar manner to our CSV method, that collects nodes with child nodes and attempts to create one singular node. In the case of ALEX, the new results from Gap insertion method was used for all data nodes. The average query performance from this method was comparable to or worse than the original index structure performances. This is due to the introduction of linking nodes that handle any conflicting key ranks as a result changes the index structure and query processing methods. Any keys that are placed into a linking array would initiate a linear search to find the correct key, resulting in large query times.

In NFL, of the experimented datasets (Facebook, Covid, OSM, Genome, Books and Uniform) only Facebook dataset was selected by NFL to perform the numerical flow distribution transformation that converts the original key set into a more uniform distribution via generative models. The average query time performance for Facebook dataset was improved to 275 ns. All other datasets would have decreased performance if the transformation was performed instead of using the original key set. As such these datasets used the original keys to bulk load the NFL index structure. However, the performance of the datasets that failed to perform the transformation were similar to that of ALEX.



(a) Total query time saved (LIPP) (b) Storage increase (LIPP) (c) Insertion time increase (LIPP)

(d) Total query time saved (ALEX) (e) Storage increase (ALEX) (f) Insertion time increase (ALEX)

Figure 11: Performance results vs. data insertions (× n)

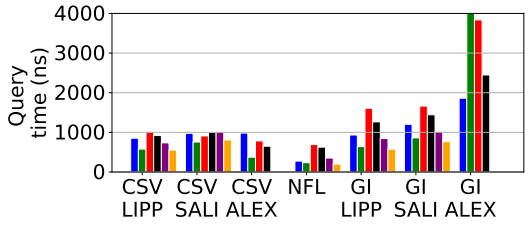


Figure 12: Comparison withe baselines

6.3 Results on Read-write Workloads

Query time (promoted data). Figs. 11a and 11d show the total query times saved by CSV for LIPP and ALEX, respectively, compared to the original index structures, as more batches of data are inserted (recall that each batch consists of $0.1n$ data points). Here, the query times saved are decreasing slightly as more data points are inserted for LIPP, because the inserted data points have a higher chance of colliding with the promoted data points which are now in the upper levels, compared to when they are in lower levels as in the original index structure. For ALEX, the trend is quite similar except for on the OSM dataset, where there are two drops after one and three insertion batches (i.e., $0.1n$ and $0.3n$ data points are inserted). This is because the original index structure's query times happen to be slightly lower in these two cases.

Index size. The index size overhead decreases after each batch of insertions, as shown in Figs. 11b and 11e, because the initial gaps left by the virtual points are gradually filled up by inserted points, hence improving the overall space use. Index size overhead is at or below 10%, emphasising the space efficiency of CSV. For ALEX, the storage increase is negligible ($< 0.5\%$). In some cases, the storage size of the CSV-enhanced ALEX is lower than the original index, as the original ALEX may need to create more new nodes to host the insertions which outweighs the space overhead of CSV.

Insertion time. Figs. 11c and 11f show the average insertion times of CSV-enhanced indexes compared to the original indexes. CSV helps improving the insertion times in some cases as the gaps left by the virtual points are reused for insertions. CSV could also lead to higher insertion times in other cases. The reason is, there are more keys at the upper levels of the CSV-enhanced indexes which may lead to more collisions with the insertions, which requires new index node creation. Overall, the insertion times of the CSV-enhanced indexes are on par to the original indexes.

7 CONCLUSION

We addressed the issue of index learning over data of complex distributions by a CDF smoothing technique to modify the key set, instead of developing yet another indexing function or structure. We proposed an algorithm named CSV to utilize this technique on existing hierarchical learned index structures, to improve the query time for the keys in lower levels of these index structures. The proposed algorithm is implemented on three recent learned indexes, which are evaluated on real-world datasets. The results show significant query performance improvements, i.e., up to 34%, with a controllable and low storage space overhead.

ACKNOWLEDGMENTS

This work is in part supported by the Australian Research Council (ARC) via Discovery Project DP230101534. Jianzhong Qi is supported by ARC Future Fellowship FT240100170. The authors thank the four anonymous reviewers for their insightful and constructive comments that helped improve the paper.

REFERENCES

[1] Matthias Bachfischer, Renata Borovica-Gajic, and Benjamin IP Rubinstein. 2022. Testing the Robustness of Learned Index Structures. *arXiv preprint arXiv:2207.11575* (2022). https://doi.org/10.48550/arXiv.2207.11575

[2] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *SIGMOD*. 969–984.

[3] 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.

[4] Yuquan Ding, Xujian Zhao, and Peiquan Jin. 2022. An Error-Bounded Space-Efficient Hybrid Learned Index with High Lookup Performance. In *DEXA*. 216–228.

[5] Martin Eppert, Philipp Fent, and Thomas Neumann. 2021. A Tailored Regression for Learned Indexes: Logarithmic Error Regression. In *aiDM*. 9–15.

[6] Paolo Ferragina, Marco Frasca, Giosuè Cataldo Marinò, and Giorgio Vinciguerra. 2023. On nonlinear learned string indexing. *IEEE Access* (2023).

[7] 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.

[8] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-aware Index Structure. In *SIGMOD*. 1189–1206.

[9] 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 *ICDE*. 315–327.

[10] 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), 258:1–258:25.

[11] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *aiDM*. 5:1–5:5.

[12] Evgenios M. Kornaropoulos, Silei Ren, and Roberto Tamassia. 2022. The Price of Tailoring the Index to Your Data: Poisoning Attacks on Learned Index Structures. In *SIGMOD*. 1331–1344.

[13] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *SIGMOD*. 489–504.

[14] Hai Lan, Zhifeng Bao, J Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2023. A Simple Yet High-Performing On-disk Learned Index: Can We Have Our Cake and Eat it Too? *arXiv preprint arXiv:2306.02604* (2023). https://doi.org/10.48550/arXiv.2306.02604

[15] 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.

[16] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*. 2119–2133.

[17] Yaliang Li, Daoyuan Chen, Bolin Ding, Kai Zeng, and Jingren Zhou. 2021. A pluggable learned index method via sampling and gap insertion. *arXiv preprint arXiv:2101.00808* (2021). https://doi.org/10.48550/arXiv.2101.00808

[18] Li Liu, Chunhua Li, Zhou Zhang, Yuhan Liu, Ke Zhou, and Ji Zhang. 2023. A Data-aware Learned Index Scheme for Efficient Writes. In *ICPP*. 28:1–28:11.

[19] 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.

[20] 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.

[21] 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* 14, 1 (2020), 1–13.

[22] Mayank Mishra and Rekha Singhal. 2021. RUSLI: Real-time Updatable Spline Learned Index. In *aiDM*. 1–8.

[23] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *SIGMOD*. 985–1000.

[24] Sachith Pai, Michael Mathioudakis, and Yanhao Wang. 2024. WaZI: A Learned and Workload-aware Z-Index. In *EDBT*. 559–571.

[25] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment* 11, 11 (2018), 1661–1673.

[26] 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.

[27] S2Geometry. 2024. *The S2 Geometry Library*. http://s2geometry.io/

[28] Yufan Sheng, Xin Cao, Yixiang Fang, Kaiqi Zhao, Jianzhong Qi, Gao Cong, and Wenjie Zhang. 2023. WISK: A Workload-aware Learned Index for Spatial Keyword Queries. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 187:1–187:27.

[29] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.

[30] 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 *PPoPP*. 308–320.

[31] Peter Van Sandt, Yannis Chronis, and Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *SIGMOD*. 36–53.

[32] Zhonghua Wang, Chen Ding, Fengguang Song, Kai Lu, Jiguang Wan, Zhihu Tan, Changsheng Xie, and Guokuan Li. 2024. WIPE: A Write-Optimized Learned Index for Persistent Memory. *ACM Transactions on Architecture and Code Optimization,* 21, 2 (2024), 22:1–22:25.

[33] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are updatable learned indexes ready? *Proceedings of the VLDB Endowment* 15, 11 (2022), 3004–3017.

[34] 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.

[35] 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.

[36] Guang Yang, Liang Liang, Ali Hadian, and Thomas Heinis. 2023. FLIRT: A Fast Learned Index for Rolling Time frames.. In *EDBT*. 234–246.

[37] 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.

[38] Zhou Zhang, Pei-Quan Jin, Xiao-Liang Wang, Yan-Qi Lv, Shou-Hong Wan, and Xi-Ke Xie. 2021. COLIN: A cache-conscious dynamic learned index with high read/write performance. *Journal of Computer Science and Technology* 36 (2021), 721–740.

[39] Weihong Zhou and Shiyu Yang. 2024. SLIPP: a space-efficient learned index for string keys. In *Proceedings of the 2024 6th International Conference on Big-data Service and Intelligent Computation*. 69–77.