# Shift-Table: A Low-latency Learned Index for Range Queries using Model Correction

Ali Hadian
Imperial College London

Thomas Heinis
Imperial College London

## ABSTRACT

Indexing large-scale databases in main memory is still challenging today. Learned index structures — in which the core components of classical indexes are replaced with machine learning models — have recently been suggested to significantly improve performance for read-only range queries.

However, a recent benchmark study shows that learned indexes only achieve limited performance improvements for real-world data on modern hardware. More specifically, a learned model cannot learn the micro-level details and fluctuations of data distributions thus resulting in poor accuracy; or it can fit to the data distribution at the cost of training a big model whose parameters cannot fit into cache. As a consequence, querying a learned index on real-world data takes a substantial number of memory lookups, thereby degrading performance.

In this paper, we adopt a different approach for modeling a data distribution that complements the model fitting approach of learned indexes. We propose *Shift-Table*, an algorithmic layer that captures the micro-level data distribution and resolves the local biases of a learned model at the cost of at most one memory lookup. Our suggested model combines the low latency of lookup tables with learned indexes and enables low-latency processing of range queries. Using Shift-Table, we achieve a speedup of 1.5X to 2X on real-world datasets compared to trained and tuned learned indexes.

## 1 INTRODUCTION

Trends in new hardware play a significant role in the way we design high-performance systems. A recent technological trend is the divergence of CPU and memory latencies, which encourages decreasing random memory access at the cost of doing more compute on cache-resident data [25, 42, 44].

A particularly interesting family of methods exploiting the memory/CPU latency gap are learned index structures. A learned index uses machine learning instead of algorithmic data structures to learn the patterns in data distribution and exploits the trained model to carry out the operations supported by an algorithmic index, e.g., determining the location of records on physical storage [7, 12, 18, 24, 25, 29]. If the learned index manages to build a model that is compact enough to fit in processor cache, then the results can ideally be fetched with a single access to main memory, hence outperforming algorithmic structures such as B-trees and hash tables.

In particular, learned index models have shown a great potential for range queries, e.g., retrieving all records where the key is in a certain range $A <$ key $< B$. To enable efficient retrieval of range queries, range indexes keep the records physically sorted. Therefore, retrieving the range query is equivalent to finding the first result and then sequentially scanning the records to retrieve the entire result set. Therefore, processing a range query $A <$ key $< B$ is equivalent to finding the first result, i.e., the smallest key in the dataset that is greater than or equal to $A$ (similar to `lower_bound(A)` in the C++ Library standard). A learned index can be built by fitting a regression model to the cumulative distribution function (CDF) of the key distribution. The learned CDF model can be used to determine the physical location where the lower-bound of the query resides, i.e., $\texttt{pos}(\text{A}) = N \times F_\theta(\text{A})$ where N is the number of keys and $F_\theta$ is the learned CDF model with model parameters $\theta$.

Learned indexes are very efficient for sequence-like data (e.g., machine-generated IDs), as well as synthetic data sampled from statistical distributions. However, a recent study using the Search-On-Sorted-Data benchmark (SOSD) [22] shows that for real-world data distributions, a learned index has the same or even poorer performance compared to algorithmic indexes. For many real-world data distributions, the CDF is too complex to be learned efficiently by a small cache-resident model. The data distribution of real-world data has "too much information" to be accurately represented by a small machine-learning model, while an accurate model is needed for an accurate prediction. One can of course use smaller models that fit in memory with the cost of lower prediction accuracy, but will end up in searching a larger set of records to find the actual result which consequently increases memory lookups and degrades performance. Alternatively, a high accuracy can be achieved by training a bigger model, but accessing the model parameters incurs multiple cache misses and also increases memory lookups, reducing the margins for performance improvement.

In this paper, we address the challenge of using learned models on real-world data and illustrate how the micro-level details (e.g., local variance) of a cumulative distribution can dramatically affect the performance of a range index. We also argue that a pure machine learning approach cannot shoulder the burden of learning the fine-grained details of an empirical data distribution and demonstrate that not much improvement can be achieved by tuning the complexity or size thresholds of the models.

We suggest that by going beyond mere machine learning models, the performance of a learned index architecture can be significantly improved using a complementary enhancement layer rather than over-emphasizing on the machine learning tasks. Our suggested layer, called *Shift-Table* is an algorithmic solution that improves the precision of a learned model and effectively accelerates the search performance. Shift-Table, targets the micro-level bias of the model and significantly improves the accuracy, at the cost of only one memory lookup. The suggested layer is optional and applied after the prediction; it can hence be switched on or off without re-training the model.

Our contributions can be summarized as follows:

- We identify the problem of learning a range index for real-world data, and illustrate the difficulty of learning from this data.

© 2021 Copyright held by the owner/author(s). Published in Proceedings of the 24th International Conference on Extending Database Technology (EDBT), March 23-26, 2021, ISBN 978-3-89318-084-4 on OpenProceedings.org.
Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

- We suggest the Shift-Table approach for correcting a learned index model, which complements a valid (monotonically increasing) CDF model by correcting its error.
- We show how, and in which circumstances, the suggested methods can be used for best performance.
- We suggest cost models that determine whether the Shift-Table layer can boost performance.
- The experimental results show that our suggested method can improve existing learned index structures and bring stable and almost-constant lookup time for real-world data distributions. Our enhancement layer achieves up to 3X performance improvement over existing learned indexes. More interestingly, we show that for non-skewed distributions, the Shift-Table layer is effective enough to help a dummy linear model outperform the state of the art learned indexes on real-world datasets

# 2 MOTIVATION

## 2.1 Lookup Cost for Learned Models

In modern hardware, the lookup times of in-memory range indexes and the binary search algorithm are mainly affected by their memory access pattern, most notably by how the algorithm uses the cache and the Last-Level-Cache (LLC) miss rate.

Processing a range query in a learned index has two stages: 1) **Prediction**: Running the learned model to predict the location of the first result for the range query; and 2) **Local search** (also known as *last-mile search*): searching around the predicted location to find the actual location of the first result. Figure 1a shows common search methods for the local search. If the learned model can determine a guaranteed range area around the predicted position, one can perform binary search. Otherwise, exponential or linear search should be used, starting from the predicted position.

A cache miss in a learned index can occur in the first stage for accessing the parameters of the model (if the model is too big to fit in cache), or in stage two for the local search. Key in understanding the cost of a learned index is that local search is done entirely over non-cached blocks of memory. A learned index built over millions of records could predict the location of records with an error of, say, 1000 records and yet achieve no performance gain over binary search algorithms or algorithmic indexes. This is because while the learned index fits the models in cache, its algorithmic competitors also fit the frequently-accessed parts of the data in cache, which limits the potential for improvement for a learned index.

## 2.2 Lookup Cost for Algorithmic Indexes

Classical algorithms, such as binary search, can be seen as a hierarchy of [non-learned] models, which take the middle-point as its parameter and predicts (accurately) which direction the search should follow. Specifically for the first few steps of binary search where the middle-points usually reside in cache, the functionality of binary search is the same as a learned model from a performance point of view.

In a pure binary search on the entire data, the first set of memory locations accessed by the algorithm (i.e., the median, quarters, etc.) will already be in the CPU cache after a few lookups. Therefore, the major bottleneck in binary search is for the latter stages of search where the middle elements are not in cache, causing last-level-cache (LLC) misses. Figure 1b shows a schematic illustration of how caching accelerates binary search.

In basic implementations of binary search, the "hot keys" are cached with their payload and nearby records in the same cache line, which wastes cache space. Binary search thus uses the cache poorly and there are more efficient algorithmic approaches whose performance is not sensitive to data distributions.

Cache-optimized versions of binary search, e.g., a binary search tree such as FAST [21], a read-only search tree that co-locates the hot keys but still follows the simple bisecting method of binary search, are up to 3X faster than binary search [22]. This is because FAST keeps more hot keys in the cache and hence it needs to scan a shorter range of records in the local search phase (cache-non-resident iterations of the search).

## 2.3 Preliminary Experimental Analysis

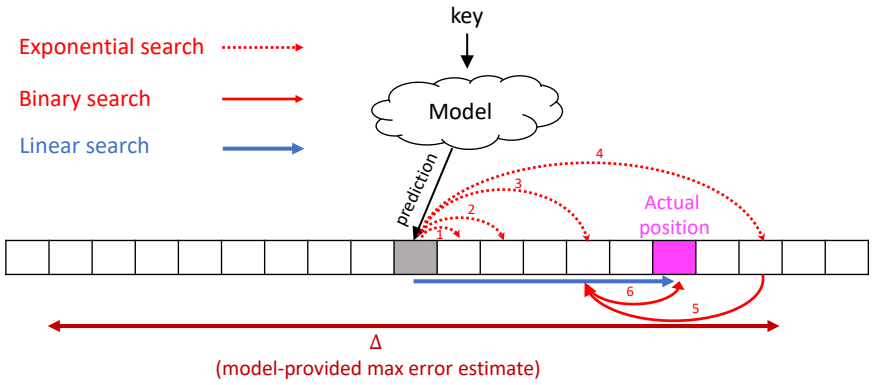
For a tangible discussion and to elaborate on the real cost of a learned model, we provide a micro-benchmark that measures the cost of errors in a learned index. We use the experimental configuration used in the SOSD benchmark [22], i.e., searching over 200M records with 32-bit keys and 64-bit payloads. Figure 2a shows the lookup time of the second phase (local search) in a learned model for different prediction errors. We include the lookup times for binary search, as well as FAST [21], over the whole array of 200M keys.

We are interested to see that if the position predicted by a learned index, say $\texttt{predicted_pos}(x)$, has an error $\Delta$, then how long does it take in the local phase to find the correct record. Thus, for each query $x_i$, we pre-compute the 'output' of the learned index with error $\Delta$, i.e., $[\texttt{predicted_pos}(x_i) \pm \Delta]$, and then run the benchmark given $\{x_i, [\texttt{predicted_pos}(x_i) \pm \Delta]\}$ tuples.

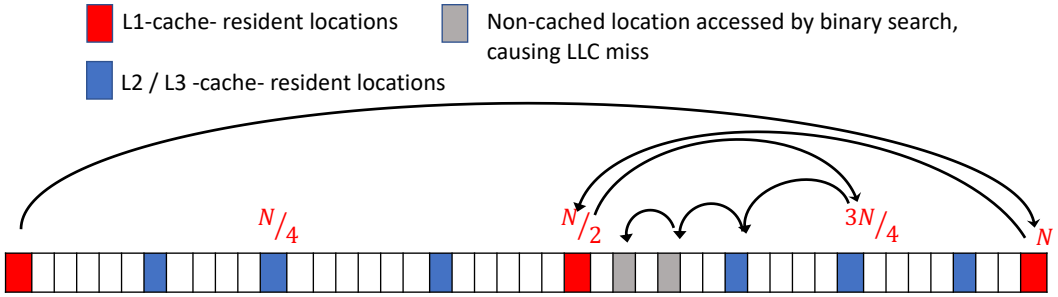
As shown in Figure 2a, if the error of the model is more than ~300 records on average, then FAST outperforms the learned model (with linear or exponential local search). Even if the learned model can give a guaranteed range around the predicted point to guide the local search and enable binary search, FAST outperforms it if the error exceeds 1000 records. The same trend can be seen for the LLC miss rates in Figure 2b.

Note that this micro-benchmark over-estimates the maximum error that the learned index can have because we only compare the time of *local search phase* in a learned index with the *total search time* of FAST and binary search. Considering the time taken to execute the model for predicting the location, a learned model needs to have a much lower error to compete with the generic, reliable, and distribution-independent algorithms such as binary search and FAST. For example, FAST takes 200 nanoseconds to search a key in the entire 200M-key dataset. If a learned index takes, say, 120 nanoseconds to run (for accessing model parameters and computing the prediction), then the local search can take at most 80 nanoseconds so that the learned index can outperform FAST, which means that the prediction error ($\Delta$) must be less than 16 records (based on Figure 2a).

Tuning the learned index for a balance of model size and accuracy is a challenging task. Improving the local search time requires using a more accurate model with a higher learning capacity and more parameters. However, accessing such a big model typically incurs further cache misses during model execution, and consequently the lookup time. Therefore, if the data distribution cannot be learned efficiently with a small memory footprint (fitting into cache), outperforming cache-efficient algorithmic indexes is very challenging. This is indeed the case for most real-world datasets that cannot be modelled accurately with a small-sized model.




**(a) Different "last-mile" search methods (performed after location prediction) in learned index. The locations predicted by the model depend on the query and are not known in advance. Since the last-mile search algorithms need to access different memory locations for each query, they cannot exploit the processor cache and the search algorithm incurs multiple cache misses**




**(b) Schematic illustration of processor caching in binary search. The locations accessed by the very early stage of binary search, such as the min, max, and the midpoint, are frequently accessed and available in the L1 cache. Further steps of binary access locations that are less frequently accessed and fit on lower levels of the memory hierarchy. Therefore, a deterministic search algorithm like binary search enjoys a high cache hit rate**

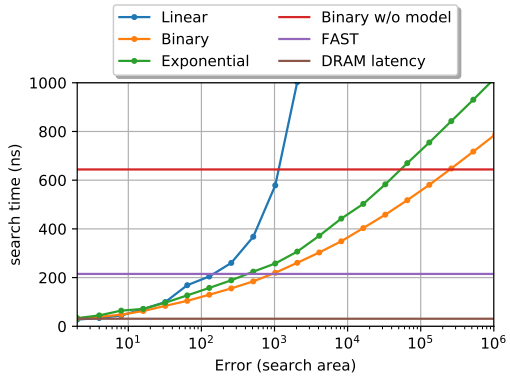
**Figure 1: Comparison of patterns in binary search (partially cached) and local search in learned indices (non-cached).**

## 2.4 Difficulty of Learning Real-world Data

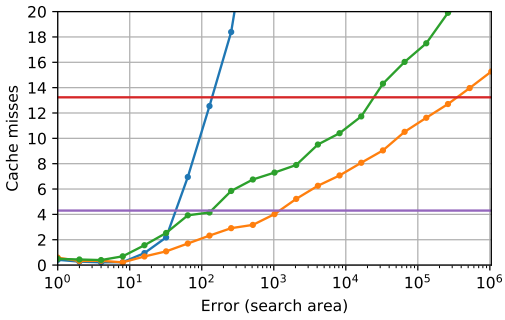
To use a learned index in a production system, it is essential to identify when learned indexes fail to achieve superior performance and what aspects of the data distribution contributes to the performance of a learned index model. We realized that a major challenge in understanding learned indexes is that the common practices of performance evaluation for indexing algorithms are misleading for learned indexes. For example, it is common to use the uniform and skewed distributions (such as log-normal) as arguably the two best- and worst-case extremes for a search task [25]. However, for evaluating search over sorted read-only data, the difficulty of the task is determined by the *unpredictability* of the data, which is not necessarily a factor of skewness or shape parameter of the data distribution. As we will show in this section, most statistical distributions are much easier to model than real-world data.

*Distributions that matter.* An interesting observation from the SOSD benchmark results is that even for datasets that have the same background distribution, e.g., both closely match a uniform distribution, the performance of a learned model can vary significantly, depending on the fine-grained details in the empirical CDFs. For example, consider Figures 3a and 3b, which represent two CDFs that are both close to uniform. The uniform data (`uden64` [22]) is comprised of dense integers that are synthetically sampled from a uniform distribution, and Facebook (`face64` [22]) is a Facebook user ID dataset. While both datasets match closely with the uniform distribution, face64 is significantly harder to model due to its fine-grained details in the CDF. The lookup time of learned indexes (both RMI and Radix-Splines) for face64 is 6-7× higher than that of uden64 (see Table 2) because there are many micro-level details (unpredictability) in the CDF, hence a huge model with a high learning capacity is needed to fit the CDF accurately. Using the RMI learned index, for example, the uden64 data is easily modelled with a simple line (two parameters) with near-zero error, while the best architecture found by the SOSD benchmark for modelling the face64 data is a hierarchy of two linear models, a huge model (136MB), with an average error of 202 records.

Generally speaking, real-world datasets are more difficult to learn compared to synthetic ones and the learned index built over them is not significantly faster than the algorithmic rivals. The main question remains what distinguishes a real-world data from a synthetic one? Consider the four distributions in Figure 3, where Figures 3a, 3c are synthetic (generated from uniform and log-normal distributions), and Figures 3b, 3d are real-world data. The mini-chart inside each CDF highlights the distribution in a small sub-range, i.e., a "zoomed-in" view of the CDF. For the synthetic data, the CDF is very smooth in any short sub-range of the whole CDF. Synthetic data (such as uniform, normal and log-normal) are built using a cumulative density function that is derivable, meaning that the at any small sub-range, the shape of the CDF is close to a straight line with a slope that is close to the derivation of the underlying CDF in that range. Such a




(a) Lookup time




(b) Cache misses

**Figure 2: Cost of local search in a learned index**

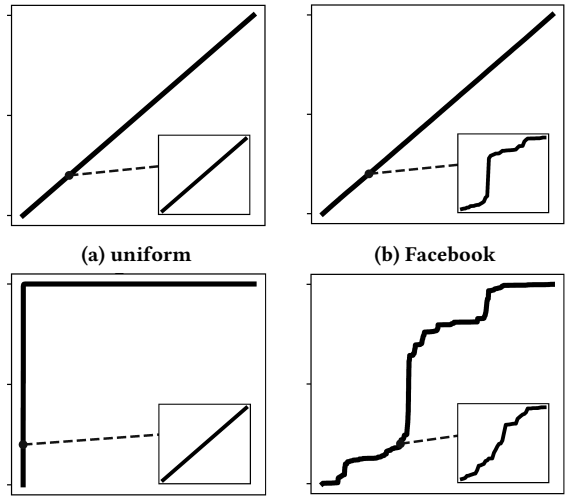
smooth CDF has less information to be compressed into a model. For example, a learned index model based on linear splines can accurately fit the whole CDF by fitting each part of the CDF to a line. Even for very skewed distributions, such as log-normal, the data is so predictable that it can be easily fitted to simple, linear models.

Real-world data, however, is much less predictable and has a much higher level of complexity in its patterns. Even if an ideal learning algorithm is used to model the real-world data, the model itself needs to be very big because the compressed version of the CDF (to be stored as a model) is still very big.

This explains why state-of-the-art learned indexes perform extremely well for datasets that are synthetically generated from a statistical distribution (such as uniform, normal, and log-normal), but perform comparably poor for real-world data that even almost match (shapewise) with those synthetic distributions [22]. On real-world datasets, learned indexes have a high cache miss rate and lookup time, contrary to their primary goal of having fewer cache misses.

Using learned models is beneficial when they are 1) accurate enough to predict a position within the same cache line that contains the data point, otherwise the lookup time will be adversely affected due to multiple cache misses, and 2) compact enough to fit in cache and not to cause LLC misses. With this in mind, we can argue that a pure machine-learning approach might fail to "learn the data perfectly" and "fit the model in cache" simultaneously, specifically in case of real-world datasets that contain a lot of underlying patterns like spikes and generally noise.

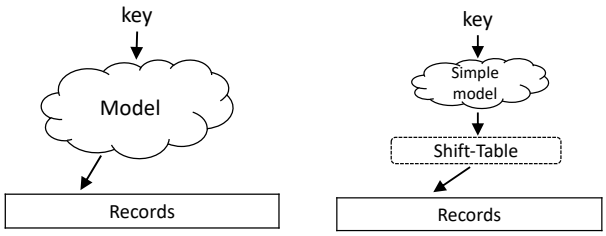
As a consequence, learned models are crucial to indexing but they cannot shoulder the burden of indexing the data alone. We hence suggest an algorithmic layer that can mitigate the difficulty



(a) uniform (b) Facebook

(c) Lognormal (d) OSMC

**Figure 3: Example distributions with different complexities in micro and macro levels**




(a) Learned index (b) Model + Shift-Table

**Figure 4: Leveraging correction layers to a learned index**

of learning the data distribution. In this approach, the learned model is allowed to learn an semi-accurate, small model that learns the holistic shape of the distribution, and the fine-tuned modelling is provided by the algorithmic layers.

## 2.5 Model Correction

While learned index models are powerful tools for describing a data distribution in a compact representation, merely focusing on learning a highly-accurate model does not necessarily lead to a high-performance index. In this paper, we suggest a new approach for boosting existing learned models with additional layers, specifically developed with hardware costs in mind.

The suggested helping layers add a small overhead when executing queries, but significantly reduce the overall lookup time of the learned index. The suggested layers are very powerful and consequently allow for using more lightweight models, yet ideally avoid computationally-expensive algorithms for training.

As Figure 4 illustrates, in addition to the learned index model we add a correction layer, an optional component, that can be added to improve the performance. We explore the potential of correction layers in the next sections.

# 3 SHIFT-TABLE

A learned model predicts a relative position $F_\theta(x)$ for a given query $x$. To calculate the position of the result, the estimated

relative position is multiplied by the number of keys, and truncated to an integer (the index), hence the predicted position is $[NF_\theta(x)]$. The actual position of the record, however, is $NF(x)$ where $F(x)$ is the empirical CDF of the data points, and $N$ is the data size. Therefore, the result is $NF(x) - [NF_\theta(x)]$ records ahead of the predicted position. We identify $NF(x) - [NF_\theta(x)]$ as the *drift* of $F_\theta$ at key $x$, which is the signed error of the prediction, as opposed to the absolute error.

The idea of the Shift-Table layer is to have a lookup table that contains the drift values so that the drift of the prediction can be corrected. Capturing the drift for every value of $x$ requires an auxiliary index, which is not feasible. However, we can use the *output* of the learned index model ($[NF_\theta(x)]$), which is in the range of $[0, N]$, and we construct a mapping from each possible output of the model, say $k$, to "how far ahead is the actual record if model predicts k's record", so that we can correct the predictions using this mapping. This means that for each prediction, we only need an extra lookup of $k$ in a fixed array of size $N$.

To build the Shift-Table layer, we first partition the keys $x_0, \cdots, x_{N-1}$ into $N$ partitions. We define $P_k$ as the set of keys for which the model predicts $k$ as the position:

$$P_k = \{x \mid [NF_\theta(x)] = k\} \quad (1)$$

Each of the indexed keys in $P_k$ has an index, say $NF(x)$ and a prediction $k = [NF_\theta(x)]$. For each partition, we extract two parameters that specify the range for local search, namely $\Delta_k$ and $C_k$. $\Delta_k$ is defined as:

$$\Delta_k = \min\left(NF(x) - k\right) \quad \forall x \in P_k \quad (2)$$

which indicates that if the predicted location is $k$, the search should be started at point $k+\Delta_k$. Also, $C_k = |P_k|$ is the cardinality of $P_k$, i.e., the number of indexed keys for which the prediction predicts the $k$'th record, in other words, the length of the area that has to be searched in the local search phase.

To correct the prediction, we first compute the predicted position $k = [NF_\theta(x)]$, and then perform local search in the range of $[k + \Delta_k, k + \Delta_k + C_K - 1]$.

The number of partitions depends on the range of the output of the learned index, which should be $0, N)$. Therefore, The $<\Delta_k, C_k>$ pairs: pairs are stored in a single array of size $N$, so that the correction can be done using a single lookup into the array of pairs.

A Shift-Table layer is depicted in Figure 5. The index contains 100 elements in range [0,999]. The CDF model is a simple model: $F_\theta(x) = x/1000$, hence the prediction is simply $k = [x/10]$. If the query is 771, for example, the prediction of the model is $k = 77$. The correction information are $\Delta_{77} = -41$ and $C_{77} = 2$, which indicates that the result is -41 records ahead of the prediction, and the search area is of length 2. Therefore, the local search is performed on the indexes of range $[36, 37]$.

Algorithm 1 shows how Shift-Table is used to accelerate query processing. The Shift-Table layer reduces the prediction error of the model, but incurs an additional memory lookup.

**Algorithm 1** Search with direct-mapped learned index

```
1:  procedure FIND_LOWER(q, model, Shift-Table)
2:      pos = model.predict(q)
3:      pos = Shift_Table.mapping[pos].startPoint
4:      range = Shift_Table.mapping[pos].range
5:      if range < linear_to_binary_threshold then
6:          pos = LinearSearch(start=data[pos],range)
7:      else
8:          pos = BinarySearch(start=data[pos],range)
9:      end if
10:     return pos
11: end procedure
```

## 3.1 Querying non-indexed keys

If the query is on the indexed keys, the result is in range $[k + \Delta_k\ ,\ k + \Delta_k + C_K - 1]$. In Figure 5, for example, querying 771 and 782 points to the correct range that contains the result. However, if the query is not among the indexed keys, then the query is either within the range, or in the position just after the range (at data$[k+\Delta_k+C_K]$. For example, in Figure 5, the record corresponding to queries 778 and 781 is the same, though the aforementioned model ($k = [q/10]$), maps 778 to range [36, 37], and 781 to [38, 39]. In both cases, however, the local search algorithm (either binary or linear search) within the range computes the correct position of the result (i.e., 38). Notably for $q = 778$, a typical local search implementation realizes that the query is greater than the largest value in range and returns the first index right after the range of [36, 37], which is 38.

Another issue that can arise for non-indexed keys is when the predicted position $P_k$ has an empty partition that none of the indexed keys belongs to. In Figure 5, if the query is 15, then the predicted position is $k = [15/10] = 1$, but $P_1$ is empty because the model does not predict position 1 for any of the indexed keys. If the query is predicted to be in an empty partition, the result is the first record in the next non-empty partition, e.g., the result of query=15 is record 3. To make the Shift-Table layer consistent for the empty partitions, we put pseudo values for $\Delta, C$ in the mapping layer such that they refer to the same range as the next existing partition. If $P_{k^\varnothing}$ is an empty partition and $P_k$ is the first non-empty partition after $P_{k^\varnothing}$, then $C_{k^\varnothing} = C_k$ and $\Delta_{k^\varnothing} = \Delta_k + (k - k^\varnothing)$. The pseudo $\Delta, C$-values are depicted using dashed arrows in Figure 5.

## 3.2 CDF and duplicate values

It should be noted that the *empirical CDF* function, i.e., $F(X) = P(X \leq x)$ does not exactly identify the result of a range query on x. In this paper, we use the CDF (F(x)) notation as the index of the result corresponding to $x$. We consider range queries of type (`key <= query`), hence the CDF for a point $x$ is the relative position of the *first* key in the indexed keys, as the range is scanned towards the right. More precisely, we assume that $NF(x_0) = 0$ and $NF(x_{N-1}) = N - 1$ (for the last key).

A range learned index built for a specific comparison operator, say $x \leq q$, can be used for other operators ($\geq, >,$, etc.) with a brief left/right scan. However, if there are too many duplicates in the indexed data, then the the performance of the learned index will be worse for queries that do not match the presumed definition of F(X). In such cases, it is more efficient to use the specific definition of $F(x)$ that reflects the position of the result of the query in the most common type of constraint in the queries. For example, if most of the queries are of type $x >= q$, then $F(x)$ should be defined such that $NF(x)$ identifies the index of the last key among the duplicate values.

## 3.3 Building the Shift-Table layer

Algorithm 2 describes how the mapping of the Shift-Table layer is built. In the first stage, it computes the $\Delta, C$ values and updates for the non-empty partitions, i.e., $P_k$s for which at least one of

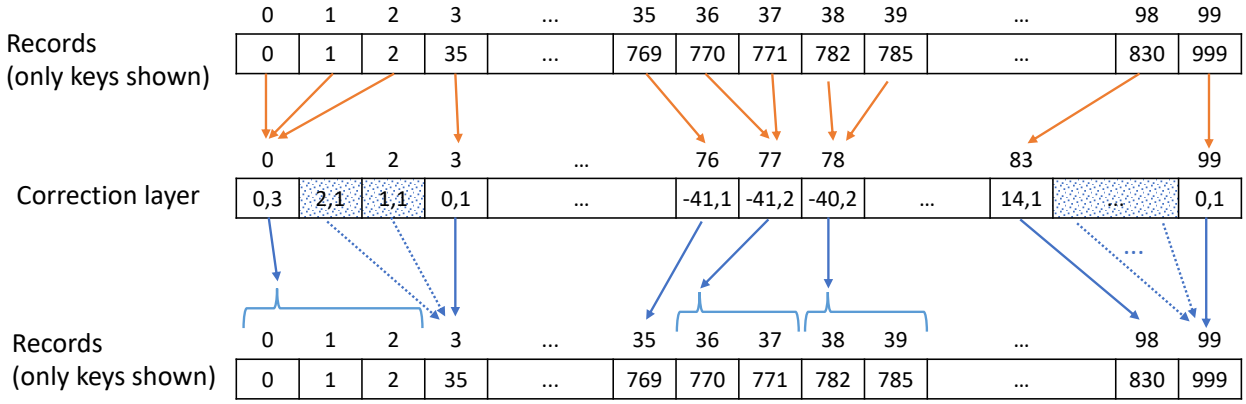



**Figure 5: Shift-Table**

the indexed keys is mapped to $k$. In the second stage, a backward traversal is performed on the Shift-Table layer and the compute the pseudo-values for the empty partitions (Algorithm 2, lines 10–14). Starting from the last entry, a pseudo-partition has the same count ($C$) as the first non-empty partition on its right side, but the shift $\Delta$ is adjusted so that they both point the the same region for local search.

The computational complexity of building the Shift-Table layer is $O(N) \times O(F_\theta)$ to compute the drifts and updating the mapping, as it only traverses the data and the Shift-Table layer once. In case that running the model is expensive, model executions can be parallelized for faster execution.

**Algorithm 2** Building the Shift-Table layer

```
1:  procedure SHIFT-TABLE_BUILD(model (F_θ), data)
2:      Shift-Table = Array of tuples <Δ, C>, all set to zero
3:      for all x ∈ data do
4:          pos = NF(x)                          ▷ Position of x (sec 3.2)
5:          k = [NF_θ(data[i])]
6:          Δ = pos − k
7:          Shift_Table[k].Δ = min(Shift_Table[k].Δ, Δ)
8:          Shift-Table[k].C += 1
9:      end for
10:     for k ← N − 1 ··· 0 do
11:         if Shift_Table[k].C = 0 then          ▷ Empty partitions
12:             Shift_Table[k].C = Shift_Table[k-1].C
13:             Shift_Table[k].Δ = Shift_Table[k-1].Δ + 1
14:         end if
15:     end for
16:     return Shift_Table
17: end procedure
```

## 3.4 Compressing the Shift-Table layer

Correcting the prediction of the model using the Shift-Table layer takes a single DRAM lookup irrespective of the size of the index. However, it might be of interest to reduce the size of the layer. The Shift-Table layer is an array of size N, containing <Δ, C> tuples. Further compression can be used to decrease the memory footprint of the Shift-Table layer.

One approach is to keep a single parameter instead of the <Δ, C> tuples. In this regard, a predicted position $k$ should be mapped to the key that is in the median point among the keys in $P_k$, which is

$$\bar{\Delta}_k = \left[\Delta_k + \frac{C_k}{2}\right] \tag{3}$$

To correct using the $\bar{\Delta}_k$ values, the final position is computed as $pos = k + \bar{\Delta}_k$, which indicates where the search should be started without specifying the guaranteed range that should be searched. Therefore, search algorithms that require the boundaries specified such as binary search cannot be used for local search. As discussed in section 2.4, linear or exponential search can be used for local search without boundaries, but they are slightly slower if the error is considerable after the correction.

A second approach that complements the first one, is to shrink the size of the Shift-Table layer by merging nearby partitions. We can extend the definition of $\mathcal{P} = \{P_1, \cdots, P_N\}$ to allow partitions that have a size of $M < N$. We define $M$ partitions $\mathcal{P}^M = \{P_1^M, \cdots, P_M^M\}$ where each partition is defined as:

$$P_k^M = \{x \mid [MF_\theta(x)] = k\} \tag{4}$$

Similarly, $\Delta_k^M$ is the minimum "move to the right" shifts that each of the keys in $P_K^M$ need:

$$\Delta_k^M = \min\left(NF(x) - [NF_\theta(x)]\right) \quad \forall x \in P_k^M \tag{5}$$

and $C_k$ should be defined such that the boundary is valid for all keys in $P_K^M$, which is:

$$C_k^M = \max(NF(x) - \underbrace{([NF_\theta(x)] + \Delta_k^M)}_{\text{start of the search window}}) \quad \forall x \in P_k^M \tag{6}$$

To combine approaches to compact the Shift-Table layer, we can use average drifts $\bar{\Delta}_k^M$ instead of the <$\Delta_k^M, C_k^M$> pairs:

$$\bar{\Delta}_k^M = \left[\frac{1}{|P_k^M|} \sum_{x \in P_k^M} (NF(x) - [NF_\theta(x)])\right] \tag{7}$$

and then use $[NF_\theta(x)] + \bar{\Delta}^M_{[MF_\theta(x)]}$ as the corrected prediction. Suppose the same data as in Figure 5, but instead of a Shift-Table layer of size N, we use only M=30 partitions. Table 1 shows how a compact Shift-Table layer is built and used for correction, on a portion of the index. We use the same model ($F_\theta = [x/1000]$), hence the prediction is $NF_\theta(x) = [0.1x]$, and the partition corresponding to a key is $NF_\theta(x) = [0.03x]$. All of the records from data[35..39] are assigned to the same partition $P_{23}^{30}$ and their predictions are shifted 40 records backwards. Note that when

$M \neq N$, a partition does not specify a single point (or range) for all of the keys in the partition. Instead, the position of a key after correction depends on both $NF_\theta(x)$ (prediction) and $MF_\theta(x)$ (partition number). For example, all keys belonging to $P_{23}^{30}$, i.e., data$[35 \cdots 39]$ have the same correction of $\bar{\Delta}_{23}^{30} = -40$, but their final predictions are different. Therefore, the correction error of a compact Shift-Table layer is less than the number of elements in the partitions.

**Table 1: Illustration of Shift-Table with $M = 30$ mapping entries on an index with $N = 100$ keys**

| Index | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 |
|---|---|---|---|---|---|---|---|---|
| key (x) | 752 | 769 | 770 | 771 | 782 | 785 | 820 | 830 |
| Predicted index= [0.1 x] | 75 | 76 | 77 | 77 | 78 | 78 | 82 | 83 |
| Error before correction | -41 | -41 | -41 | -40 | -40 | -39 | -42 | -42 |
| Partition (k) = [0.03 x] | 22 | 23 | | | | | 24 | |
| $\bar{\Delta}_k^{30}$ | -41 | -40 | | | | | -42 | |
| Prediction after correction | 34 | 36 | 37 | 37 | 38 | 38 | 40 | 41 |
| Error after correction | 0 | 1 | 1 | 0 | 0 | -1 | 0 | 0 |

The drift of $P_k^M$, namely $\bar{\Delta}_k^M$ is the index of the median key among the members of $P_k^M$. This means that if the key is predicted to be in the $k$'th partition (among the $M$ partitions), the local search is done around $[NF_\theta(x)] + \bar{\Delta}_k^M$.

Using a Shift-Table layer of size $K < N$ does not affect the complexity of building the layer, which is $O(N) \times O(F_\theta) + O(M)$. However, if the midpoint-values are used (correction without specifying the boundary), it is possible to construct the map using a sample of the indexed keys, which comes at the cost of the accuracy. Using a sample of size $S < N$, the layer can be built in $O(S) \times O(F_\theta) + O(K)$ time.

Nonetheless, keep in mind that the Shift-Table layer is designed for applications that favour latency to memory footprint, hence reducing the memory footprint of the Shift-Table layer by a large factor will limit its margin for improvement as the fine-grained details of the empirical CDF will be lost to some extent.

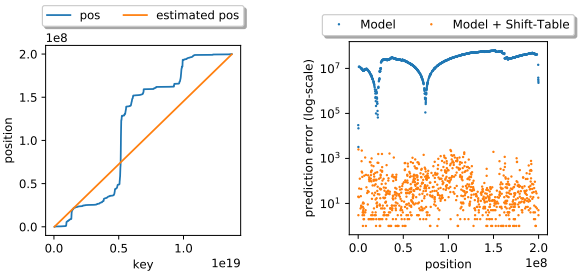
## 3.5 Measuring the error

Since the Shift-Table layer specifies a range for local search, the notion of error is not trivial. However, we can use the estimates without range ($\bar{\Delta}$), for which the correction picks the median value among the keys in the $P_k$. The error for the keys in each partition is $\left\{[\frac{C_k}{2}], \cdots, 0, \cdots, [\frac{C_k}{2}]\right\}$ if $C_k$ is odd, and $\left\{[\frac{C_k}{2}] - 1, \cdots, 0, \cdots, [\frac{C_k}{2}]\right\}$ if $C_k$ is even. The average error is approximately $C_k/4$.

In a learned index without Shift-Table, the error is the distance between $F(x)$ and $F_\theta(x)$. After correcting the model with the Shift-Table, however, the error only depends on the $C_k$ values, i.e., a prediction error only occurs when $[F_\theta(x)]$ predicts the same position for multiple keys. Therefore, the local search range and the error are combinations of multiple step functions over the $P_k$s with $C_k > 1$.

The average error depends on the data distribution in the query workload. If the queries are uniformly sampled from the keys, then the average error is:

$$\bar{e} = \frac{1}{2N} \sum_{k \in \mathcal{P}} C_k^2 \tag{8}$$




**(a) Example data & model** **(b) Error**

**Figure 6: Error correction using the Shift-Table layer**

## 3.6 Behaviour of the Shift-Table layer

Figure 6 illustrates how the Shift-Table layer corrects the error of a linear interpolation model on the OSMC data. While the model is too simple to capture the patterns in data, the Shift-Table layer alone is effective for correcting the predictions. While the average error of the model is 28 million keys, Shift-Table reduces the error to only 129 keys.

Shift-Table corrects two types of error. First, when the model has a considerable local bias, which means that $NF(x)$ diverges significantly from $NF_\theta(x)$ in a sub-range of the data distribution. The second type of error is the fluctuations of the distribution between the nearby keys, for most of which the Shift-Table layer is very effective. The only type of error that can degrade the performance of the Shift-Table layer is when there is a congestion of keys in a small sub-range of values, leading to many of the keys being classified in a single layer, and hence having some partitions with high $C_k$.

The behavior of the Shift-Table layer and its error estimate indicates that it can be effective in eliminating different types of errors that models have. One common type of error is the local bias in the model, i.e., when the error of the model, i.e., $NF_\theta(x) - NF(x)$ has a considerable *bias* in some sub-ranges of the distribution, meaning that the $F$ and $F_\theta$ diverge at some point. This happens when the model cannot capture the CDF in a local neighborhood. Table 2 shows that even if a single line is used as a model, which has a huge bias in most areas of the distribution, the Shift-Table layer can efficiently eliminate the huge bias of a fully linear model (a single line as a model), and reduces the error significantly such that the linear model outperforms all other algorithms for the real-world datasets, as well as the uspr dataset (sparse uniformly-distributed integers) which has a significantly higher variance than uniformly-distributed dense integers.

Another type of error that the Shift-Table layer eliminates is the local variance in the data, which is the fluctuations of the values between nearby keys. This type of error is very common in real-world data. For example, the *face, uspr, and uden* datasets all follow a uniform distribution, but they have different local variances, which is the amount of fluctuations in the nearby keys. The uden dataset is very easy to model using the learned indexes and does not require a helping layer such as Shift-Table. The other two datasets, however, are very hard to model using the learned index structures. This is because the Shift-Table model can easily correct the fluctuations of values (different increments between each two points), as long as the model does not predict a single record for a lot of nearby keys (resulting in a high $C_k$ value).

### 3.7 Cost model of the Shift-Table layer

The accuracy of the model after correction with Shift-Table depends on the cardinalities of the partitions ($C_i$ values). Ideally, if the records of each partition reside on a single cache line, the results will be retrieved in a single memory lookup. The cost of local search, i.e., the mapping between the accuracy in each partition and the latency to do local search depends on the hardware. As discussed in section 2.1, the latency of search for various ranges can be measured by a micro-benchmark over non-cached regions with different sizes. Let $L(s)$ be the measured latency of non-cached search over a range containing $s$ records. The latency for looking up a key in a region of size $s$ is $L(C_k)$. Assuming that the queries have the same distribution as the data points, the average lookup latency for the index is:

$$\text{Latency with Shift-Table} = \text{Latency}(F_\theta) + \frac{1}{N}\sum_{k\in\mathcal{P}} C_k L(C_k) \quad (9)$$

The cost model can also be used to estimate which of the local search algorithms should be used, by substituting in equation 9 the local search cost of each local search algorithm, i.e., $L(s)$ mappings for linear, binary, and exponential search; and for and their different implementations. Branch-optimized binary search would be the natural choice if the Shift-Table model can determine the boundary (if using the $\Delta_k, C_k$ pairs), otherwise either linear or exponential search should be chosen based on the latency estimate.

Taking the cost of running the Shift-Table layer into account, we should consider how much the correction improves the accuracy of the learned index model and hence estimate the speedup. The lookup time of the model without using the Shift-Table model can be estimated once the Shift-Table model is built, without running a speedup benchmark. The model error for each key is $\bar{\Delta}_k = \Delta_k + \frac{C_k}{2}$, therefore the estimated runtime of the index without correction is:

$$\text{Latency without Shift-Table} = \text{Latency}(F_\theta) + \frac{1}{N}\sum_{k\in\mathcal{P}} C_k L(\bar{\Delta}_k) \quad (10)$$

### 3.8 CDF model validity constraint

The correction layer requires the learned model to be a valid CDF function, i.e., $F_\theta(x)$ should be monotonically increasing: $x_i > x_j \longrightarrow F_\theta(x_i) >= F_\theta(x_j)$. Among our baselines, the RadixSplines learned index always produces a valid (increasing) CDF, but the RMI index does not always produce monotonically increasing predictions. In RMI, for example, the CDF model might decrease when using cubic models [30] or on the edge point between two models in the second-level. If $F_\theta(x)$ is not monotonically increasing, then the correction layer could identify a range that does not include the query result, because the values of $x$ for which the learned model predicts $k$'th record are not in a contagious memory block.

A learned index model that is non-monotonic can still use the Shift-Table layer, as the output of the Shift-Table layer would still predict a position but it is not guaranteed that the position is in the predicted range. Therefore, the local search algorithm should check if the query is in the predicted range and perform a search outside of the range. Another hack for non-monotonic model is to use the $\bar{\Delta}$ midpoint-values instead of the $\Delta_k, C_k$ pairs, which predicts a location (instead of a range) to start the local search.

If the Shift-Table layer uses the $<\Delta_k^M, C_k^M>$ pairs, it can determine the range for local search and we can apply either linear or binary search, depending on the error range. We do linear search if the range is smaller than a threshold (8 keys, in our experiments), otherwise a binary search is performed. However, if it only contains the average shift values ($\bar{\Delta}_k$, it predicts a position without specifying the boundaries that contain the record; hence either linear or exponential search can be performed depending on the *average* error rate and performance objectives (average or worst-case latency).

### 3.9 Tuning the system

The Shift-Table layer is optional and adds overhead to the search. Therefore, enabling Shift-Table is only worthwhile if it can eventually accelerate the original learned index structure. An effective configuration of the index is a choice between 1) Using the model alone, 2) model + Shift-Table. Note that the Shift-Table layer is optional and can be deactivated with zero cost. The output of the model and the Shift-Table layer are of the same type and both represent a prediction of the records, hence if the Shift-Table layer is disabled, we can easily use the model alone for prediction of the records.

While tuning the system, the performance of each configuration can be directly measured using performance tests, or by measuring the model error and then using the cost model of the Shift-Table model on the bottom of the architecture (section 3.7).

The parameters of the architecture, i.e., the Shift-Table array size $M$ and the parameters of the learned CDF model, can be tuned by computing the error estimate using Shift-Table's cost model, or alternatively, by running a performance tests on the built architecture. Our suggested default value for the Shift-Table layer is $M = N$, because using a mapping layer that has the same number of entries as the keys will ensures that the layer can exhibit its ultimate effect to eliminate the signed error, and does not have more latency compared to using smaller $M$ values.

An advantage of Shift-Table is that the learned model does not need to be very accurate, as a correction will be applied anyway. Therefore, a more relaxed measure can be used instead of least-square error. In this paper, however, we do not learn the model w.r.t. the Shift-Table layer, for the sake of simplicity and to keep the Shift-Table layer detacheable (optional), preserving the assumption that the Shift-Table layer can be disabled to free up memory space on run-time while the model can still be used.

The accuracy of the learned model also determines the size of the entries of the Shift-Table layer. Each mapping entry should at most fit a $\Delta$ value of $\Delta_{MAX}$, which is the maximum error of the model. If, for example, the error is smaller than $2^16/2$, then a 16-bit integer (`short` type) can be used.

## 4 EVALUATION

In this section, we compare the performance of our proposed method with the SOSD benchmark [1], which is a recent benchmark for search on sorted data. The benchmark includes learned indexes, classical indexes, and no-index search algorithms.

***Experimental Setup.*** The algorithms are implemented in C++ and compiled with GCC 9.1. The experiments are performed on a system with 16 GB of memory and Intel Core i7-6700 (Skylake), which has four cores and is running at 3.4 GHz with 32 KB L1, 256 KB L2, and 8 MB L3 caches. The operating system is Ubuntu 18.04 with kernel version 4.15.0-65. In our setup, the LLC miss

[1] https://github.com/learnedsystems/SOSD/tree/mlforsys19

penalty measured by Intel Memory Latency Checker [2] is 36 ns, which is the minimum lookup time of an ideal index.

Note that all data resides in main memory. The range index finds the first indexed key that is equal to or bigger than the lookup key. Also, the keys on the physical layout are sorted (i.e., it is a clustered index), so that the entire result set of the range query can be returned once the first key is found. Similar to [22, 25], we only report the lookup time for the first result and do not include the scan times in our experiments because all indexes use the same layout for the data records.

***Datasets.*** For the sake of reproducibility, we used the same datasets as in the SOSD benchmark, which contains four datasets synthetically generated from known distributions and four real-world ones. The synthetic data are generated from different distributions, namely *logn*: lognormal distribution $(0, 2)$, *norm*: normal distribution, *uden*: uniformly-generated dense integers, and *uspr*: uniformly-generated sparse integers. The real-world datasets are *face*: Facebook user IDs [42], *amzn*: book sale popularity from Amazon sales rank data [3], *osmc*: uniform sample of OpenStreetMap locations [4], and *wiki*: timestamps of edit actions on Wikipedia articles[5]. All datasets contain 200M unsigned integers.

***Implementation details.*** Our experiments are based on the SOSD benchmark [22]. The baseline includes two learned indexes, namely RadixSpline [33] (RS), which uses linear splines; and Recursive Model Index (RMI), which uses a hierarchy of models. Note that RMI has a choice of different models and SOSD [22] specifically handpicked the best models for each of the datasets in the benchmark [6]. SOSD also includes no-index search algorithms such as binary search (BS), linear interpolation search (IS), and the recently suggested non-linear triple-point interpolation (TIP) [42]. We also compare against algorithmic index structures such as ART: Adaptive Radix Tree [26], FAST [21], RBS (Radix Binary Search): a two-stage algorithm in which a radix structure that maps a fixed-length key prefix to the range of all keys having that prefix and then a binary search is performed on the range [22], and STX implementation of B+tree [1]. Finally, we included four *On-the-fly* search algorithms, namely BS: Binary search (STL implementation), TIP: three-point interpolation [42], Interpolation search, which is similar to binary search but uses interpolated positions in each iteration, and IM: *Interpolation as a Model*: a dummy model that interpolates the key between the minimum and maximum value of the keys and then performs exponential search around the predicted key.

The experiments use either 32- or 64-bit unsigned integer IDs for the key (depending on the dataset), and 64-bytes for the payload.

## 4.1 The SOSD benchmark

To test the effectiveness of the suggested layers compared to learned indexes, we use a simple interpolation model (IM), i.e., $F_\theta(x) = (x - minVal)/(maxVal - minVal)$. Such a dummy model is deliberately chosen to purely delegate the burden of data modelling to the correction layers.

The Shift-Table layer has the same number of entries as the actual data, i.e., $M = N$. We followed the tuning procedure discussed in section 3.9: we start from the model (IM and RS) and consequently evaluate IM+Shift-Table and RS+Shift-Table. The cost of running the Shift-Table layer is around 40ns, which pays off by reducing the prediction error and thus lookup time. Therefore, based on the cost model of the Shift-Table layer (Section 3.7) and the error-to-latency micro-benchmark (Figure 2a), we should not add the Shift-Table layer if the error before adding the configuration is less than a threshold (10 records), or 2) the error of the index after adding the Shift-Table layer does not decrease by a factor of 10 (roughly equivalent to the 50-nanoseconds latency the additional layer, according to the error-to-latency micro-benchmark).

Table 2 compares the lookup times (nanoseconds per lookup) of the baseline algorithms with our dummy interpolation model (IM), and the two corrected versions, i.e., IM+Shift-Table and RS+Shift-Table. Note that ART does not support data with duplicate keys, and FAST does not support 64-bit keys. Also, interpolation search (IS) takes too much time on some datasets, because the execution time of interpolation search highly depends on the uniformity of data distribution, varying from O(loglogN) + O(1) iterations on uniform distributions, to O(N) iterations for very skew ones [42].

For the synthetic datasets, the difficulty of the datasets for our dummy linear interpolation model varies from very easy (uden64) to extremely hard (logn64). While the Shift-Table layer significantly improves a dummy layer on non-uniform data distributions, it cannot outperform the learned index models. This is not surprising, as all synthetic datasets (uniform, lognormal, and uniform) have a pattern derived from continuously differentiable density functions, hence the distribution is similar to a straight line on smaller sub-ranges as we "zoom in" the data distribution (e.g., see Figure 3c). Therefore, a learned index structure composed of linears at the bottom (including both RMI and RS) can effectively model the distribution using a very compact representation.

For the real-world data, however, the fluctuations in data severely affect both RMI and RS learned indexes. The Shift-Table layer, effectively corrects a highly inaccurate dummy IM model, such that it outperforms the RMI learned index by 1.5X to 2X on all datasets, while RS falls behind both. Keep in mind that RMI requires to be tuned with the best architecture and parameters, while Shift-Table does not require a manual training process and can even work with a simple model such as IM that is not trained, and yet deliver a lower latency.

Figure 7 shows the average build times of the indexes, along with the standard deviation bars indicating how the build time varies for different distributions. Please note that the RMI implementation used in the SOSD benchmark needs to be compiled for faster retrievals, however we did not include RMI's extra overhead for compiling the code and only reported the build time. IM+Shift-Table, the winner method latency-wise, also takes either the same or even less build time than the competing learned indexes.

## 4.2 Explaining the performance

The latencies reported in Table 2 present the fastest configuration for each learned index. In this section, we present the details of the tuning process to see the optimum performance of each learned index.

[2] https://software.intel.com/en-us/articles/intelr-memory-latency-checker
[3] https://www.kaggle.com/ucffool/amazon-sales-rank-data-for-print-and-kindle-books
[4] https://aws.amazon.com/public-datasets/osm
[5] https://dumps.wikimedia.org
[6] The architectures and parameters of the RMI models used for each dataset is specified at https://github.com/learnedsystems/SOSD/blob/mlforsys19/scripts/build_rmis.sh

**Table 2: Comparison of lookup times (nanoseconds per lookup) with the SOSD benchmark. The red box indicates the base model (IM) and the enhanced versions.**

| | | Algorithmic indexes | | | | On-the-fly search | | | | Learned indexes | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Dataset | ART | FAST | RBS | B+tree | BS | TIP | IS | IM | IM + Shift-Table | RMI | RS | RS + Shift-Table |
| Synthetic | logn32 | N/A | 230 | 385 | 375 | 624 | 551 | N/A | 1384 | 166 | **73.9** | 83.9 | 143.5 |
| | norm32 | 173 | 197 | 267 | 390 | 655 | 671 | N/A | 1479 | 88.2 | **51.5** | 60.3 | 96.4 |
| | uden32 | 99.4 | 196 | 235 | 389 | 654 | 126 | **32.3** | 38.6 | 67.5 | 38.1 | 47.8 | 72.3 |
| | uspr32 | N/A | 198 | 230 | 390 | 654 | 298 | 321 | 425 | **89.7** | 141 | 166 | 153.5 |
| | logn64 | 238 | N/A | 622 | 427 | 674 | 377 | N/A | 1075 | 376 | 132 | **109** | 151.0 |
| | norm64 | 214 | N/A | 317 | 427 | 672 | 705 | N/A | 1615 | 88.6 | **51.7** | 61.8 | 93.2 |
| | uden64 | 104 | N/A | 255 | 428 | 670 | 142 | **34.8** | 40.4 | 67.4 | 39.8 | 47.9 | 71.8 |
| | uspr64 | 216 | N/A | 244 | 427 | 673 | 329 | 338 | **472** | **92.8** | 145 | 182 | 154.6 |
| Real-world | amzn32 | N/A | 208 | 243 | 393 | 658 | 569 | 3228 | 1524 | **99.5** | 185 | 236 | 110.8 |
| | face32 | 179 | 203 | 238 | 388 | 654 | 717 | 792 | 861 | **103** | 213 | 310 | 142.8 |
| | amzn64 | N/A | N/A | 284 | 428 | 676 | 578 | 3510 | 1575 | **105** | 189 | 238 | 119.3 |
| | face64 | 290 | N/A | 257 | 427 | 671 | 925 | 1257 | 918 | **149** | 247 | 344 | 204.1 |
| | osmc64 | N/A | N/A | 410 | 428 | 675 | 4617 | N/A | 1462 | 194 | 297 | 339 | **177.2** |
| | wiki64 | N/A | N/A | 271 | 437 | 686 | 767 | 5867 | 1687 | **94.2** | 172 | 191 | 124.1 |

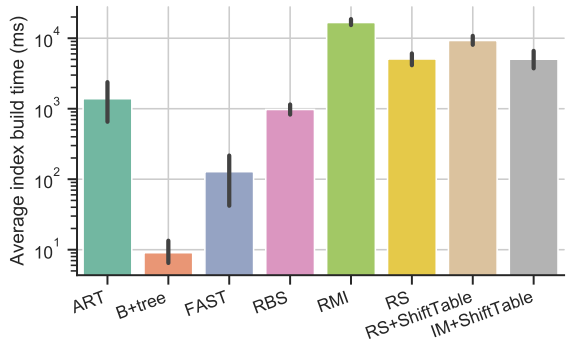



**Figure 7: Build times (average time for all datasets)**

For those indexes that have a parameter affecting the index size (such as the branching factor in B+tree, and the number of radix bits in ART, RS, and RBS), the performance can be tuned by evaluating the latency for different index sizes.

Figure 8 shows the latencies of the indexes for the face64 and osmc64 datasets, along with the average Log2 error, CPU instructions, and L1/LLC cache misses. IM+Shift-Table and RS+Shift-Table achieve faster lookup times on both datasets. For most indexes, except RMI and RBS, the latency does not improve beyond a certain optimum index size, after which the latency increases again. RBS has a much larger latency than both [IM/RS]-Shift-Table indexes of the same size, and extrapolating the RMI latencies also suggest that if we could extend RMI size to 1400MB (equal to Shift-Table's size), it could not achieve a game-changing performance on either of the datasets. Note that we could not run RMI with larger models because RMI embeds the parameters into the code, and the compile times for models larger than 400MB were astonishingly high.

Average Log2 errors indicate the average number of iterations in binary search for the last-mile search stage. Larger models result in lower Log2 errors in all indexes and lead to faster last-mile search, however, once the model exceeds the LLC cache sizes, cache-miss rate increases (when running the model), and hence the prediction time worsens. For RS, ART, and B+tree, the cache misses and extra overhead of running the models increases either the number of instruction, the cache misses, or both, enough to prevent the index from improving latency by increasing the footprint.

## 4.3 Layer size

As discussed in section 3.4, the Shift-Table layer can be compressed by merging multiple entries, hence reducing its footprint. Figure 9 shows the effect of the Shift-Table layer size on lookup time and prediction error. Shift-Table can operate in two modes: **R-1**: a full layer containing $<\Delta_k^M, C_k^M>$ pairs similar to Figure 5 that indicates the exact *range* for local search (hence enabling binary search); and **S-X**: a compressed single-entry map similar to Table 1 containing one $\bar{\Delta}_k^M$ entry per $X$ records. Thus, S-X contains $M = N/X$ entries; and the memory footprint of S-1 is half the size of R-1.

The error of the S-1 Shift-Table is slightly more than that of R-1. This is due to the fact that S-1 is designed to draw boundaries for binary search; hence it always points to the first record of each partition; while R-1 always points to the middle of the partition and almost half the error of S-1. Performance-wise, however, S-1 always has the lowest latency, because its boundaries for the last-mile search operation do not need to be discovered using additional boundary-detection algorithms such as exponential search. As expected, compressing the Shift-Table by allocating one entry per $X$ records increases the error and hence degrades

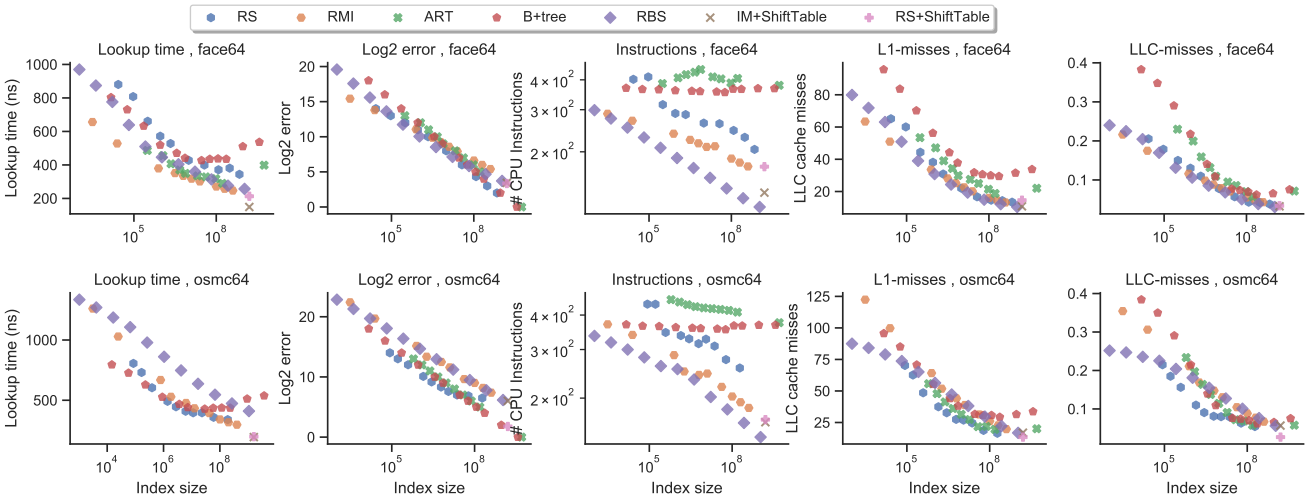



**Figure 8: Analysis of the effect of index size on performance**

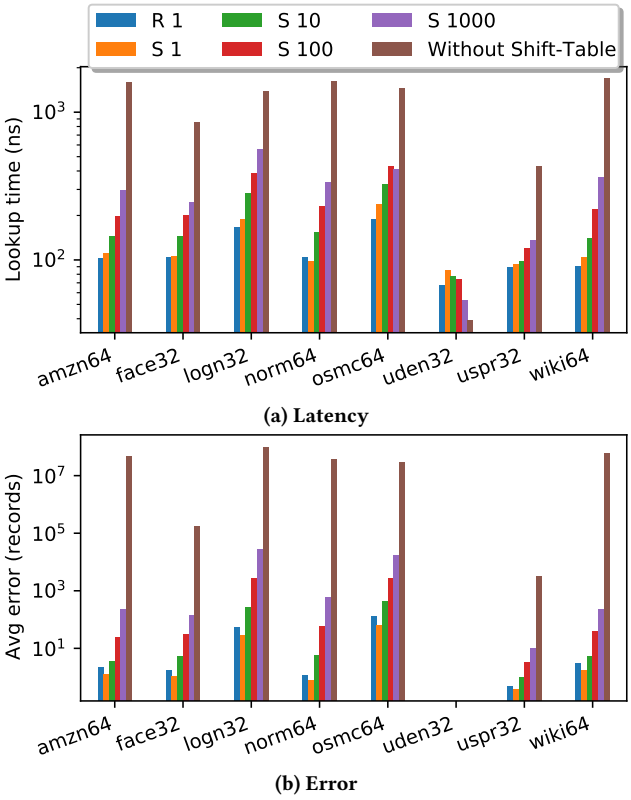



**Figure 9: Analysis of the effect of Shift-Table layer size**

the performance. This is due to the fact that with higher compression ratios, the ability of Shift-Table to "memorize" the fine-grained details of the data distribution degrades due to the loss of information after merging.

## 5 RELATED WORK

**On-the-fly search on sorted data** A fundamental problem that is studied for decades is how to find a key among a sorted list of items. The classic approach is binary search and numerous extensions have been suggested to improve it for special cases, most notably interpolation search [35] and exponential search [3]. For data distributions that are close to uniform, interpolation-search is shown to be very effective [13, 36, 42]. Due to the growing gap between CPU power and memory latency in the past decade, more advanced interpolation techniques such as three-point interpolation are becoming viable on modern hardware [42]. Exponential search enables binary search over an unbounded list. Exponential search is also extensively used in learned indexes when the key is more likely to be near a "guessed" location, but a guaranteed boundary around the guessed point that contains the data is not known [7, 25, 32].

**Range indexes** An alternative to on-the-fly binary search over sorted data is to keep the data in an index structure. Nonetheless, indexes that are built to answer range queries (such as B-trees) are similar to the binary search in that they need to keep the data sorted internally. Common index structures for range index include skiplists, B+trees, and radix-trees. The B+-tree is cache-efficient, but requires pointer chasing, which incurs multiple cache misses [14]. There has been a tremendous effort to make binary search trees and B+-trees efficient on modern hardware. For example, FAST [21] organizes tree elements efficiently to exploit modern hardware features such as the cache line and SIMD. Another common solution is to use compression techniques on the indexed keys, most notably as a radix-tree. Modern radix trees exploit hardware-efficient heuristics for fitting a distribution in memory (usually by building a heuristically-optimized compressed trie), such as adaptive radix index (ART) [5, 26], and Succinct Range Filter (SuRF) [44]. Skiplist is specifically efficient for concurrent updates workloads [41, 43].

**Learned index structures** Learned range indexes [7, 12, 25, 29, 33] have recently been suggested as an alternative to range indexes. In this approach, a model is trained from the data with the intent of capturing the data distribution and processing the queries more efficiently. We refer to the paper by Kraska et al. [25], which introduced the idea of the learned index. In a learned index, the CDF of the key distribution is learned by fitting a model, and the learned model is subsequently used as a replacement of the index (B+-trees or similar) for finding the location of the query results on the storage medium. Index learning frameworks such as the RMI model [25, 30] can learn arbitrary models [30], although a further theoretical study [9] as well as a recent experimental benchmark [22] have shown that simple

model like linear splines are very effective for datasets. Spline-based learned indexes include Piecewise Geometric Model index (PGM-index) [11], Fiting-tree [12], Model-Assisted B-tree (MAB-tree) [19], Radix-Spline [23], Interpolation-friendly B-tree (IF-Btree) [18] and some others [29, 40]. We refer to [10] for an extensive comparison of learned indexes. Recently, there has been numerous theoretical works [4, 27, 38, 39] on learned indexes. Also, numerous efforts have been made to handle practical challenges around using a learned index, including update-handling [7, 17] and designing a learned DBMS [24]. The idea of using a model of the data to boost an existing algorithmic index has been the center of focus in the past few years [14, 17, 19, 37]. In the multivariate area, learning from a sample workload has also shown interesting results [8, 20, 28, 32]. Aside from the main trend in learned indexes, which is on range indexing, machine learning has also inspired other indexing and retrieval tasks. This includes bloom filters [6, 31], multidimensional indexing on datasets with correlated attributes [15], and other applications [2, 16, 34].

## 6 CONCLUSION AND FUTURE WORK

Learning and modeling data distributions via machine learning approaches is a great idea for managing and analyzing data management systems. However, the approaches and objective functions that are common in machine learning problems are not necessarily optimal choices when the ultimate target is performance improvement. Instead of pushing machine learning model algorithm to its limits for highly accurate modeling of data distributions, it is more efficient if we only use ML models to approximate the high-level, generalizable "patterns" in data distribution (the holistic shape), and handle the fluctuations and fine-grained details of the distribution using a more hardware-efficient approach, outperforms learned models as well as algorithmic index structures even if a simple or somewhat dummy model such as min/max linear interpolation is used. The Shift-Table layer is effective in learning almost all distributions even without using models that require training from data, and takes only a single pass over the data points to build the layer. Our results show that even a simple linear model equipped with the Shift-Table enhancement layer outperforms trained and tuned learned indexes by 1.5X to 2X on real-world datasets.

Our current work only considers read-only workloads. We leave it as future work to adapt Shift-Table with workloads having updates. One idea is to capture the drifts in data distribution using update-tracking segments [17], and use Fenwick trees to estimate and correct the drifts in both the model and the Shift-Table.

## REFERENCES


[1] STX. B+Tree C++ Template Classes. http://panthema.net/2007/stx-btree.
[2] Naiyong Ao, Fan Zhang, Di Wu, Douglas S Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *VLDB Endowment* 4, 8 (2011), 470–481.
[3] Jon Louis Bentley and Andrew Chi-Chih Yao. 1976. An almost optimal algorithm for unbounded searching. *Information processing letters* 5 (1976).
[4] Rasmus Bilgram and Per Hedegaard Nielsen. 2019. *Cost Models for Learned Index with Insertions.* Technical Report. University of Aalborg.
[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 *SIGMOD*. 521–534.
[6] Zhenwei Dai and Anshumali Shrivastava. 2019. Adaptive learned Bloom filter (Ada-BF): Efficient utilization of the classifier. *arXiv:1910.09131* (2019).
[7] 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.
[8] Mohamad Dolatshah, Ali Hadian, and Behrouz Minaei-Bidgoli. 2015. Ball*-tree: Efficient Spatial Indexing for Constrained Nearest-neighbor Search in Metric Spaces. arXiv:cs.DB/1511.00628
[9] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why are learned indexes so effective?. In *ICML*, Vol. 119. PMLR.
[10] Paolo Ferragina and Giorgio Vinciguerra. 2020. Learned data structures. *Recent Trends in Learning From Data* (2020), 5–41.
[11] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *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 *SIGMOD*. 1189–1206.
[13] Goetz Graefe. 2006. B-tree indexes, interpolation search, and skew. In *DaMoN*.
[14] Goetz Graefe and Harumi Kuno. 2011. Modern B-tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.
[15] Ali Hadian, Behzad Ghaffari, Taiyi Wang, and Thomas Heinis. 2021. COAX: Correlation-Aware Indexing on Multidimensional Data with Soft Functional Dependencies. arXiv:cs.DB/2006.16393
[16] Ali Hadian and Thomas Heinis. 2018. Towards Batch-Processing on Cold Storage Devices. In *ICDEW*.
[17] Ali Hadian and Thomas Heinis. 2019. Considerations for handling updates in learned index structures. In *AIDM*.
[18] Ali Hadian and Thomas Heinis. 2019. Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes. In *EDBT*.
[19] Ali Hadian and Thomas Heinis. 2020. MADEX: Learning-augmented Algorithmic Index Structures. In *AIDB*.
[20] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *AIDB*.
[21] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*. 339–350.
[22] 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 Machine Learning for Systems* (2019).
[23] 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*.
[24] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR*.
[25] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. 489–504.
[26] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.
[27] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. 2019. A Scalable Learned Index Scheme in Storage Systems. *arXiv:1905.06256* (2019).
[28] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *SIGMOD*.
[29] Anisa Llavesh, Utku Sirin, Robert West, and Anastasia Ailamaki. 2019. Accelerating B+tree Search by Using Simple Machine Learning Techniques. In *AIDB*.
[30] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *SIGMOD*. 2789–2792.
[31] Michael Mitzenmacher. 2018. A Model for Learned Bloom Filters and Related Structures. *arXiv:1802.00884* (2018).
[32] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-dimensional Indexes. In *SIGMOD*. 985–1000.
[33] Thomas Neumann and Sebastian Michel. 2008. Smooth interpolating histograms with error guarantees. In *BNCOD*. Springer, 126–138.
[34] Harrie Oosterhuis, J Shane Culpepper, and Maarten de Rijke. 2018. The potential of learned index structures for index compression. In *ADCS*.
[35] W Wesley Peterson. 1957. Addressing for random-access storage. *IBM journal of Research and Development* 1, 2 (1957), 130–146.
[36] CE Price. 1971. Table lookup techniques. *Comput. Surveys* 3, 2 (1971), 49–64.
[37] Wenwen Qu, Xiaoling Wang, Jingdong Li, and Xin Li. 2019. Hybrid Indexes by Exploring Traditional B-Tree and Linear Regression. In *WEBIST*. 601–613.
[38] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2018. Deja Vu: an empirical evaluation of the memorization properties of ConvNets. *arXiv:1809.06396* (2018).
[39] Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. 2019. Spreading vectors for similarity search. In *ICLR*.
[40] Naufal Fikri Setiawan, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2020. Function Interpolation for Learned Index Structures. In *ADC*. 68–80.
[41] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. 2016. Cache-sensitive skip list: Efficient range queries on modern cpus. In *DaMoN*. Springer, 1–17.
[42] 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.
[43] Zhongle Xie, Qingchao Cai, HV Jagadish, Beng Chin Ooi, and Weng-Fai Wong. 2017. Parallelizing skip lists for in-memory multi-core database systems. In *ICDE*. IEEE, 119–122.
[44] 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 *SIGMOD*. 323–336.