Constant-space tunable-accuracy insertion-robust learned range indices

Askar Safipour Afshar [1] **Nikhil Kannan** [1] **Scott C Wang** [1]

Abstract

One of the main limitations of the original learned index framework as proposed in (Kraska et al., 2018) is that it assumes the underlying data distribution (key space) to be static, which implies that a learned range index does not support any form of insertions or deletions in the key space. We address this challenge by extending the learned range index to accommodate a dynamic data distribution. Our learned index architecture accounts for various insertion and lookup patterns, by using a mixture-of-experts model trained on a training cache of recently accessed keys, and a feedback mechanism that allows the learned range index to optimize for different data access patterns. Furthermore, we analyze various conditions and parameter settings that affect the accuracy of our learned range index architecture, and show that our architecture has a tunable accuracy that allows for trading off space consumption.

1. Introduction

A range index is a data structure that permits mapping a key to its position in a sorted array of keys. It is a cumulative distribution function (CDF) of a sample space of keys.

Hitherto, the predominant range index in production database systems is the B-tree, a data structure that maintains pointers that permit logarithmic-time access into a sorted array of keys. The pointers stored by a B-tree occupy space linear in the number of keys; (Zhang et al., 2016) finds that B-tree indices occupy 55% of the space of a modern production database. Consequently, there is much interest in making range indices more compact.

The learned range index (Kraska et al., 2018) is a step in this direction; it proposes the use of a machine learning model to approximate the CDF of the key space. Its critical observation is that it is acceptable for the index to return an inexact position for a particular key; if the sorted array of keys is stored on disk, the page containing that key would have to be read from disk in its entirety anyway, and then the cost of an in-memory scan or search would be minimal. Accordingly, the learned index has the potential to be tuned for accuracy by adjusting the architecture and the hyperparameters of the underlying machine learning model.

One of the main challenges in designing a learned index framework that supports insertions is approximating the CDF of the constantly changing underlying data distribution. Furthermore, insertions of new keys into the key space constantly changes the existing key-position mappings, making it difficult to train the machine learning model accurately. To overcome these challenges, we design a novel architecture for our learned index framework, which includes the following components:

1. Mixture-of-experts neural network: comprises an ensemble of expert neural networks controlled by a gating function. This neural network architecture accurately learns the CDF of a simple underlying data distribution. Furthermore, it has tunable hyperparameters, such as the number of experts and number of hidden units per expert, that control the tradeoff between the accuracy of the model and its space consumption.

2. Feedback mechanism: trains the mixture-of-experts model. Once the true position for the given key is found in the underlying array, the key-position mapping is used as training data to train the index model. This helps the learned index to quickly learn the positions of frequently looked up keys, as well as capture the variations in the data distribution that occur due to key insertions.

3. Training cache: maintains robustness of the index to insertion by storing the most recently accessed keys and their respective true positions, accounting for keys that have been inserted. Each time a new key is inserted into the underlying array, the original key-position mappings maintained within the cache are updated to reflect the true mappings after the insertion operation. These updated mappings are then used to retrain the model.

[1]Department of Computer Sciences, University of Wisconsin–Madison, Madison, WI, USA. Correspondence to: Askar Safipour Afshar <safipourafsh@wisc.edu>, Nikhil Kannan <nkannan2@wisc.edu>, Scott C Wang <wangsc@cs.wisc.edu>.

Proceedings of the 36^{th} International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019.

In summary, we contribute the following:

1. We design a novel architecture for a learned range index that has
   (a) constant space and time overhead in the number of keys
   (b) flexibility to model any distribution function by adopting a mixture-of-experts neural network.
   (c) sensitivity to lookup frequency distribution: greater accuracy for more frequently looked-up keys due to a feedback mechanism that retrains the model once the true key position is found
   (d) robustness to drift from inserting key k due to maintaining a randomly-replaced cache of key-position mappings, allowing retraining with updated positions for all keys greater than k
2. We implement the learned range index architecture, and show that it provides accuracy that can be tuned for space consumption and that adapts to various synthetic data distributions and access patterns, including:
   (a) uniformly distributed, Gaussian distributed, and mixture-of-Gaussian distributed data
   (b) insertion patterns following defined distributions (sequential, random uniform)
   (c) interspersed insertions and lookups (i.e., a given ratio between lookups and insertions)
   (d) lookup patterns following defined distributions (sequential, random uniform, loose modal distribution, tight modal distribution, most recently inserted)

2. Related work

Database indices For decades, indices such as B-trees and Bloom filters have been studied and implemented in production systems thanks to their significant query processing speed. However, for large datasets, indices incur a high overhead in I/O throughput and space (Zhang et al., 2016). Consequently, it is paramount to minimize index size (Goldstein et al., 1998; Graefe & Larson, 2001) and adopt techniques to adjust the index structure to be compatible with the characteristics of modern hardware (Kim et al., 2010; Leis et al., 2013; Rao & Ross, 2000). Recently, some studies have proposed the possibility of replacing traditional indices with learned models such as neural networks (Kraska et al., 2018; Galakatos et al., 2019; Wu et al., 2019; Tang et al., 2020).

We build our study upon (Kraska et al., 2018), addressing its limitations. The learned index introduces a new perspective on how machine learning can augment index structures in database systems to accelerate lookups and reduce space consumption. The learned index builds on the idea that indices such as B-trees and Bloom filters can be considered as key-value mapping functions, and uses machine learning models to approximate the function. In the case of an underlying sorted array of keys, traditionally indexed using a B-tree, this function is the CDF of the key space distribution. Imagine a database with N records, within which we wish to locate a key k; the position of k may be represented as

$$\text{pos}(k) = N \cdot \mathsf{Pr}(x \leq k)$$

We can replace the range index function of a B-tree with a machine learning model, $f(k) = N \cdot \mathsf{Pr}(x \leq k)$, that provides the approximate position of k in the sorted array. Even if the approximate position is not exactly accurate, knowing it still allows reducing the scans needed within the underlying array.

Consequently, the main goal of the learned range index is to train machine learning models, such as deep neural networks, to approximate the CDF of the key space for accurate prediction of keys. Despite its performance advantage, there are two issues with the practicability of the learned range index as proposed by (Kraska et al., 2018): (1) there is no mechanism to handle writes, and (2) the performance of the learned range index heavily depends on both data and query distributions. Our work provides solutions to address these two issues.

Learning CDFs using neural networks (Magdon-Ismail & Atiya, 1999) is the first proposal to model a static CDF using a neural network. (Hadian & Heinis, 2019) approaches the task of modelling a changing CDF by selecting a small subset of evenly distributed reference keys in the key space. With each insertion, the true positions of the reference keys are updated. A key lookup can then interpolate between the two surrounding reference keys to find an offset by which to adjust the learned index's predicted position. One shortcoming of this approach is that if insertions are unevenly distributed, the reference keys themselves may become unevenly distributed in the key space, decreasing their effectiveness at being an accurate offset: if $k_1 < k_2$ are reference keys, and an overwhelming number of new keys are inserted just above k_1, then the estimated offset for the newly inserted keys would be quite inaccurate. Our learned index architecture bypasses this shortcoming.

Mixture-of-experts models (Miller & Uyar, 1997) originates the idea of segmenting a domain for learning using a mixture-of-experts architecture, which we adopt in our learned index architecture. (Shazeer et al., 2017) improves the scalablility of the mixture-of-experts architecture by employing sparsity.

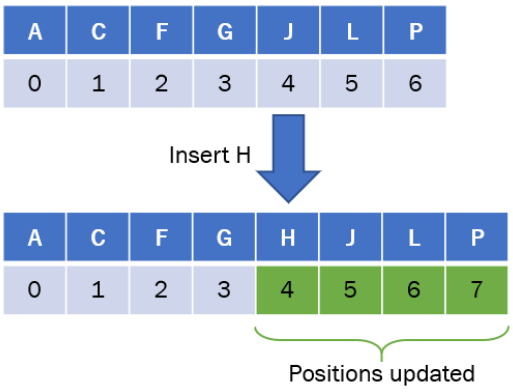

Figure 1. Key drift. In this example, a key 'H' is inserted into a sorted array, causing the position of every subsequent key to be incremented by one.

3. Problem statement

Our work tackles the following issues inherent to the original design of the learned index (Kraska et al., 2018).

3.1. Key drift

The performance of the original learned index relies upon a stringent constraint: it requires a completely static key distribution, not allowing for insertions and deletions from the key space. This is unrealistic for real-world applications, which demand consistent accuracy even if the underlying key space distribution changes frequently.

Insertions and deletions pose a unique challenge to learned indices because they portend the possibility of key drift. In this project, we concentrate specifically on insertions, and note that when a key k is inserted at a position, the position of every subsequent key greater than k must be incremented by one (Figure 1).

In other words, the learned index must approximate a key space CDF that is pinned at zero at the left end, but grows unbounded at the right end with each insertion. It would be impractical to retrain the entire learned index on the updated position distribution during each insertion of a key k, especially if the key space is already very large. Instead, we require a method of updating the learned index incrementally to "stretch" the learned key space distribution towards the right, accounting for the insertion of k. In other words, we need to make the learned index "forget" the previous position of the keys subsequent to k, and retrain those keys on their new positions.

3.2. Tunable space consumption to trade off accuracy

The learned index as proposed in (Kraska et al., 2018), which the authors term the recursive model index, relies upon a preconstructed "tree" of indices, relatively inaccurate neural networks at the upper layers, and completely accurate "last-mile" B-trees at the bottom layer (Figure 2). In essence, the upper layers form a hierarchical mixture-of-experts learner.

A few observations prompt us to reconsider the last-mile B-tree layer. First, a learned index of the architecture described above is essentially only an optimization over the array of keys contained in the B-tree nodes closer to the root; thus, in a key insertion workload, the lower-level B-tree nodes would expand with linear space complexity. Theoretically, its asymptotic space and time complexity in the number of keys it stores is not reduced from that of the B-tree. Practically, if the accuracy of the neural networks in the upper layer is good, then the B-tree would only be traversed minimally, making its space consumption less defensible.

Second, we believe that there is scope for simple key space distributions to be directly learned and predicted without the need for a last-mile B-tree layer. For example, if one expects the keys to fall into a linear distribution specified completely using two parameters, it would be overkill to represent the distribution using a tree of neural networks and B-trees. We believe that many key space distributions are simple, owing to which they can be more compactly approximated by a single neural network model with a constant number of parameters, as a corollary of the universal approximation theorem; in this way, we can guarantee that the index adheres to a constant space and time complexity. For simple distributions, we hope to find the B-tree to be redundant. We concede that any more complex key space distribution than can be represented by a handful of parameters is less likely to be learned accurately by a single neural network than several expert neural networks, but we nevertheless counter that over such distributions, the demand for accuracy is also distributed unevenly.

Third, we wish to design a mechanism that supports a trade-off between accuracy and space consumption. The purpose of the last-mile B-tree layers is to ensure that the index provides complete accuracy without the need to scan the underlying sorted data array. We imagine that for some applications, training an index would take up an unjustifiably large amount of space, for instance because the application throughput requirements are lax enough to allow scanning the underlying array to recover from an index having mispredicted the position of a key. B-trees, being rigidly linear in space complexity in the number of keys, would not permit any flexibility in tuning the needed level of accuracy. Thus, our goal is to obviate the last-mile B-tree layers due to their poor space consumption, the relative simplicity of most key space distributions, and the desire to trade off complete accuracy against space, which B-trees do not support.

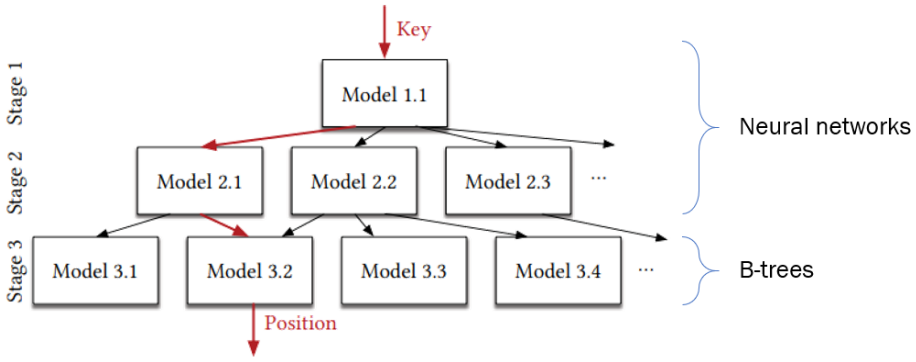

Figure 2. The structure of the recursive model index of (Kraska et al., 2018), including neural networks at the upper layers and B-trees at the bottom layer. Figure adapted from (Kraska et al., 2018).

3.3. Data access patterns

We confine our study to learned indices that have two access operations: insertions and lookups. We observe that in a typical database workload, lookups are interspersed with insertions, but different database applications have different patterns of insertions and lookups. In some cases, the number of lookups may outweigh the number of insertions (read-heavy workloads); in others, the opposite may be true (write-heavy workloads). The lookups could be unevenly distributed; they might be, for instance, uniformly random, random following a defined probability distribution, or dependent in some way upon the insertion order (e.g. only looking up the most recently inserted key or the keys at the extrema). Similarly, the insertions could be strictly sequential, or they could themselves be unsorted.

We hypothesize that the data access pattern poses an opportunity for optimization of the learned index. Consider a data access pattern that includes randomly ordered, infrequent insertions interspersed with more frequent reads that follow a tightly modal probability distribution. Intuitively, the index should be able to support sacrificing accuracy on the less frequently looked up keys in favor of improved accuracy on the more frequently looked up keys.

In addition, we observe that after the learned index provides a predicted position for a lookup or insertion key k, the lookup or insertion operation must always find the true position of k at which to subsequently update the underlying sorted array. For instance, in Figure 1, if the learned index predicted that the position of key 'J' is 4, but the key at position 4 is actually 'H', a forward scan would need to occur beginning from 'H' to find the true position for 'J'. Once the true position for 'J', 5, is found, then the learned index should be retrained to improve its accuracy on key 'J'.

4. Technical contributions

Taking into account our observations of:

1. key drift during insertion, and the difficulty of learned indices in "forgetting" previously trained positions
2. the space consumption of B-trees
3. arbitrary simple key space distributions
4. the desire to trade off accuracy for improved space utilization
5. variations in workloads, possibly interspersing lookups and insertions
6. the greater cost of inaccurate lookups for more frequently looked-up keys
7. the fact that during index operations, the true position of a key is always eventually known

our study proposes an improved architecture to the original learned index (Kraska et al., 2018), namely the adoption of a mixture-of-experts neural network, a feedback mechanism for retraining the learned index model on true positions of accessed keys, and a training cache.

The architecture for our learned index framework can be explained in the following steps (Figure 3):

1. The application queries the mixture-of-experts model for the predicted position of key k. The mixture-of-experts model returns the predicted position.
2. The application searches for the true position for key k in the underlying sorted array, starting at the predicted position returned by the mixture-of-experts model.

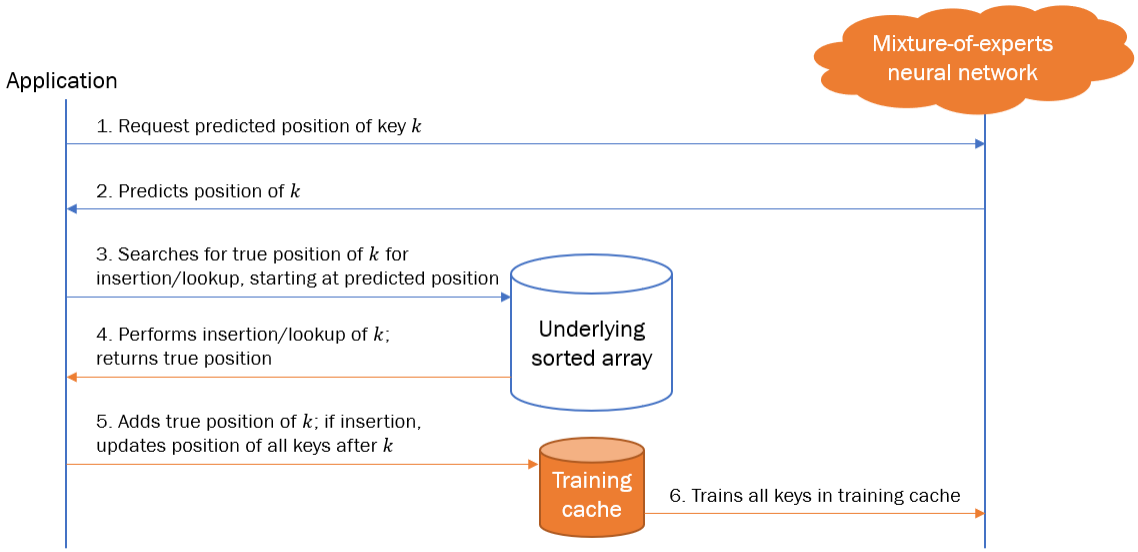

Figure 3. The architecture of the constant-space tunable-accuracy insertion-robust learned index. The additional components above the standard learned index are marked in orange. They are the designation of the mixture-of-experts neural network architecture, the feedback mechanism, and the training cache.

3. The insertion or lookup operation is performed in the underlying array: in the case of an insertion, the application inserts key k into the underlying sorted array, causing the positions of each key greater than k to be incremented by one. For a lookup operation, the value corresponding to the key is returned without any modification to the underlying sorted array. The true position of the key k is returned to the application.

4. Once the true position for the key k is obtained, the key-position mapping of k is appended into the training cache. In case the training cache is full, a random replacement policy is utilized to remove an existing entry from the cache before inserting the new mapping. Moreover, if an insertion operation has occurred, entries in the cache whose keys are greater than key k are updated. After all the entries in the cache are updated, they are used as training data to train the mixture-of-experts neural network.

Mixture-of-experts neural network We adopt the mixture-of-experts neural network architecture from the recursive model index (Kraska et al., 2018) as a means of modelling an arbitrary simple key space distribution. We envisage being able to tune the hyperparameters of the mixture-of-experts neural network, in particular the number of expert neural networks and the number of hidden nodes in each expert neural network, to allow the model's preference bias to match the expected key space distribution. This allows for an adjustable level of accuracy depending upon the application-specific cost of misprediction, permitting flexible accuracy without being encumbered by B-trees.

The mixture-of-experts is an ensemble of expert neural networks that are controlled by a gating function. The expert neural networks specialize at certain inference tasks, while the gating function acts like a discriminator network that decides the importance of the predictions made by each of the expert neural networks by assigning importance weights to each of the experts' predictions. The gating function endeavors to assign the highest importance weight to the expert model that would provide the most accurate prediction for the given task. The functionality of the expert neural network and the gating network within our mixture-of-experts model architecture can be explained as follows:

1. Each expert neural network generates an output y_i that can be represented as

$$y_i = f(W_i k + c_i) \tag{1}$$

where i denotes the i^{th} expert neural network in the mixture-of-experts model, f is the expert activation function, W_i is its weight parameters, and c_i is its bias. The output y_i is the prediction made by expert i for a given input key k.

2. Similarly, the gating network generates an output g_i that can be represented as

$$g_i = g(v_i k + b_i) \tag{2}$$

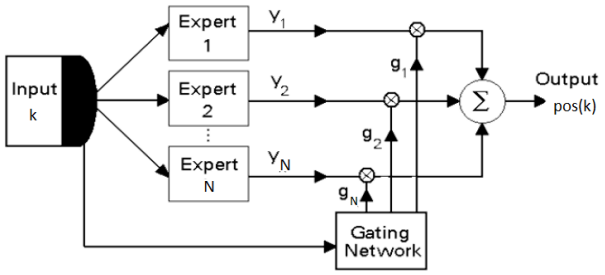

Figure 4. The architecture of the mixture-of-experts neural network. Figure adapted from (Melo et al., 2007)

where i denotes the i^{th} expert neural network. g is the gating activation function, v_i is its weight parameters, and b_i is its bias. The output g_i is the importance weight assigned by the gating network to the predictions made by expert i for a given input key k.

3. The final output $\text{pos}(k)$ is generated by the mixture-of-experts model by taking an inner product between the output of the expert neural network and the gating network. The output can be represented as

$$\text{pos}(k) = \sum_{i=1}^{N} g_i y_i \tag{3}$$

where N denotes the total number of experts. The output $\text{pos}(k)$ represents the predicted position for the given input key k.

Our choice of the mixture-of-experts neural network architecture is due to its preference bias for segmented functions. Because each expert is a fully connected layer, and there is only one connection between the initial gating layer and each expert, following (Battaglia et al., 2018), the relational inductive bias of this architecture may be considered to be a piecewise function, which is well suited for our application for an index over a monotonic key space CDF. Furthermore, a single-layer mixture-of-experts is adequate to model the simple data distributions that we focus on, striking a balance between model complexity, space consumption, and accuracy.

Feedback mechanism for retraining on true positions We propose to train the index on the known true position once it is found in the sorted array during index operations. This forms a feedback loop between the array and the index, which helps to mitigate key drift without having to retrain the subsequent keys after insertion. Note that we include true positions found during lookups as well: this way, the index is trained most often on frequently looked-up keys, increasing the accuracy for those keys.

The feedback mechanism allows the learned index to quickly improve its prediction based upon the true position of the key (Figure 5).

Training cache of recently accessed keys To further improve accuracy, our index maintains a cache of recently accessed keys and their true positions, and retrains the neural network on this cache. If a key k is inserted, the positions of all keys greater than k in the cache are updated before retraining.

In the function approximation nature of the index setting, our goal is to fit the distribution as closely as possible (because the true distribution is equivalent to the training data, overfitting is the desideratum). The cache supports this goal dually: it allows the neural network to more quickly memorize newly seen true positions without jeopardizing any other positions the model has already memorized, and it dilutes any positions memorized by the model that have become out of date due to key drift.

To best support random access patterns, we implement a random cache replacement policy. The size of the cache is also tunable, providing another dial by which to effect a tradeoff between accuracy and space consumption.

5. Empirical evaluation

We discuss the performance of our learned index architecture using synthetic experiment conditions.

5.1. Implementation

We implement the above learned index architecture in Python, using TensorFlow to implement the mixture-of-experts model, adapting (Orhan, 2016). Each expert is a feedforward neural network with 20 hidden nodes in a single layer and a ReLU activation function. During each index access (insertion or lookup), the training cache is updated and the model is trained by RMSprop optimization on the entire contents of the training cache for 10 epochs with a learning rate of 0.1 and a batch size equal to the training cache size, capped at 10.[1]

5.2. Training and evaluation

For various index hyperparameter values (Table 1), we measure the space consumption of the index (including the mixture-of-experts model and the cache) in response to various key space sizes for comparison against the B-tree, as well as the sensitivity of the learned index's accuracy to various training conditions (Table 2).

[1]The code for our learned index architecture experiments is at https://github.com/scottcwang/better_learned_index.

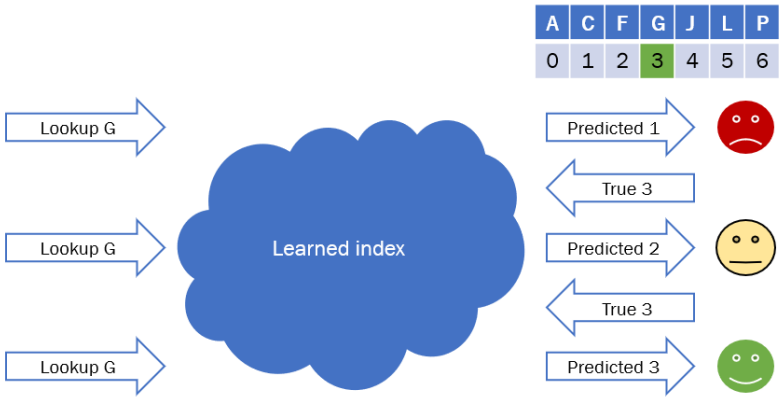

Figure 5. The idealized feedback mechanism. Each time the application requests a prediction of the key 'G' from the model, it reports the true position of 'G' back to the model.

Table 1. Index hyperparameter values

HYPERPARAMETER	VALUES. * below denotes value used to generate figures, unless specified in figure caption
Size of training cache	1, 3, 9, 27, 81*
Number of experts	1, 2, 4, 8*, 16

We train the index by generating a key space following some *key space distribution*. At each iteration, we either insert a (not-yet-inserted) key in the key space into the index following some *insertion pattern*; or, with some *lookup interspersion probability*, we randomly look up a key from the index following some *lookup pattern*. The iteration proceeds until every key is inserted. We vary each of these training conditions independently.

We measure the accuracy of an index operation by the absolute error between the position of a key as predicted by the index and the true position of the key in the underlying array. We measure the *mean absolute error during training* by taking the average absolute error of all insertions and lookups during the training process. After the training process completes, we measure the *mean absolute error during evaluation* by looking up every inserted key (without retraining the model) and taking the unweighted average of the absolute error of all such lookups. This measurement is done separately from training to highlight the disparate effect of uneven lookup patterns on the accuracy of the model.

Figure 6 shows the learned index after a sample run of our training process on a miniature key space.

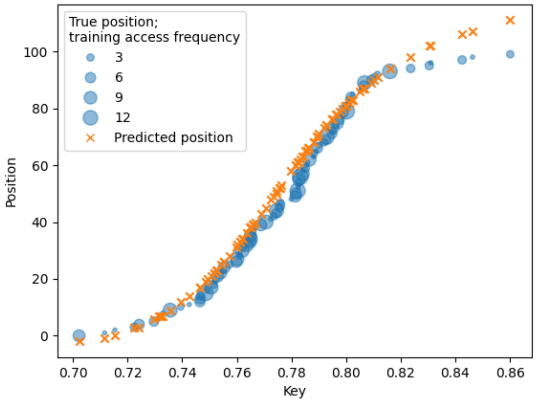

Figure 6. The learned index after a sample run of the training process on a miniature key space of size 100.

5.3. Space consumption, compared to B-trees

By varying the number of experts hyperparameter of the mixture-of-experts model, our learned index architecture permits a tradeoff between the desired space consumption and the accuracy (Figure 7). Such a tradeoff is not permitted by B-trees, whose space consumption is determined solely by the size of the key space.

The recursive model index of (Kraska et al., 2018) is incomparable to our index because it is exact and only permits an accuracy–space tradeoff by way of redistributing node sizes. Its structure is equivalent to a B-tree with a small node size at the top layers and bigger node sizes at the bottom layers.

5.4. Key space distribution

Increasing the cache size increases the number of training data points, that is, true key-position mappings, that can

Table 2. Training conditions

TRAINING CONDITION	VALUES. * below denotes value used to generate figures, unless specified in figure caption
Key space distribution	Uniform in $[0, 1)$ * Gaussian: $\mu \sim \text{Unif}(-1, 1), \sigma \sim \text{Unif}(0, 0.1)$ Mixture of ten Gaussians: $\forall 1 \leq i \leq 10 : \mu_i \sim \text{Unif}(-10, 10), \sigma_i \sim \text{Unif}(0, 0.1)$
Insertion pattern	Insertions in increasing order (sorted) Insertions in random order (shuffled) *
Lookup interspersion probability	No lookups ($1:0$ insertion-lookup ratio) 50% probability of lookup (approximately $1:1$ insertion-lookup ratio) * 66.67% probability of lookup (approximately $1:2$ insertion-lookup ratio) 75% probability of lookup (approximately $1:3$ insertion-lookup ratio) 80% probability of lookup (approximately $1:4$ insertion-lookup ratio)
Lookup pattern	"Loose" β distribution over key space ($a = b = 2$; skip if key not already inserted) "Tight" β distribution over key space ($a = b = 5$; skip if key not already inserted) Sequential (lookups follow insertion order; skip if all inserted keys are already looked up) Most recently inserted Random uniform (select one already inserted key) *

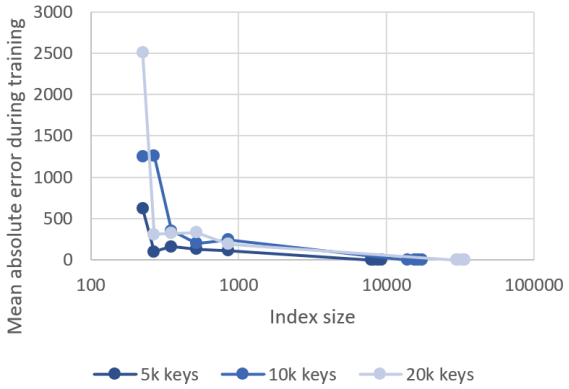

Figure 7. The tradeoff between the index size (logarithmic scale), which is a proxy for space consumption, and its accuracy. The leftmost points correspond to the learned index with 1, 2, 4, 8, and 16 experts. The rightmost points with zero error represent the sizes of the B-trees, of node sizes 128, 256, 512, and 1024, that would be required to store the key spaces.

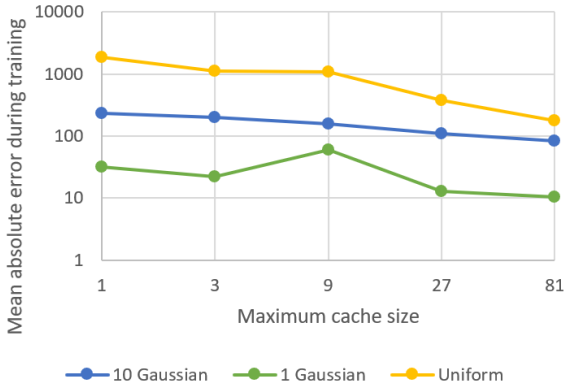

Figure 8. The effect of increasing the training cache size on accuracy for various key space distributions (both axes are on a logarithmic scale).

be used to train the model at each iteration, allowing the model to accurately interpolate to the variations in the data distribution. Hence, increasing the cache size improves the overall accuracy of our index model (Figure 8).

Similarly, increasing the number of experts used within the mixture-of-experts neural network augments the model's capacity to accurately learn relatively complex CDFs of various data distributions, hence having a similar effect on accuracy as increasing cache size (Figure 9).

We also notice that it is easier to learn a single Gaussian distribution compared to a more complex mixture of 10 Gaussian distributions or a uniform distribution. We hypothesize that this is because the CDF of a mixture of 10 Gaussian distributions appears as a piecewise function which requires a larger sample to learn completely in comparison to a Gaus-

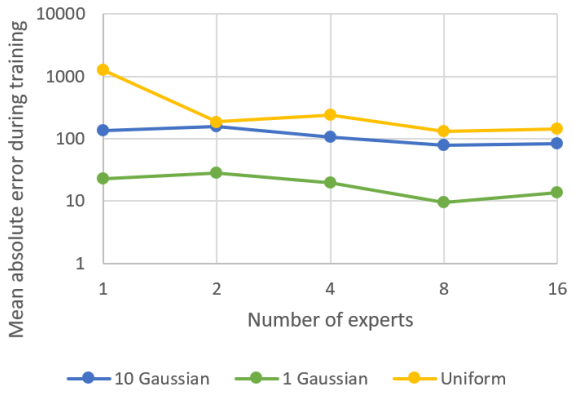

Figure 9. The effect of increasing the number of experts on accuracy for various key space distributions (both axes are on a logarithmic scale).

Figure 10. The effect of increasing the number of experts (logarithmic scale) on accuracy for shuffled and sorted insertions of Gaussian-distributed key spaces.

sian distribution. A larger sample is relatively difficult to provide in the setting of incremental insertions. The uniform distribution can be viewed as the limit of a mixture of a large number of Gaussian distributions.

5.5. Insertion pattern

Figure 10 shows that sorted insertions are more difficult to learn than shuffled insertions if there is only one expert, i.e. a simple feed-forward network in the absence of the mixture-of-experts architecture. As the number of experts increases, sorted insertions become relatively easier to learn the key space distribution from, justifying our use of the mixture-of-experts model.

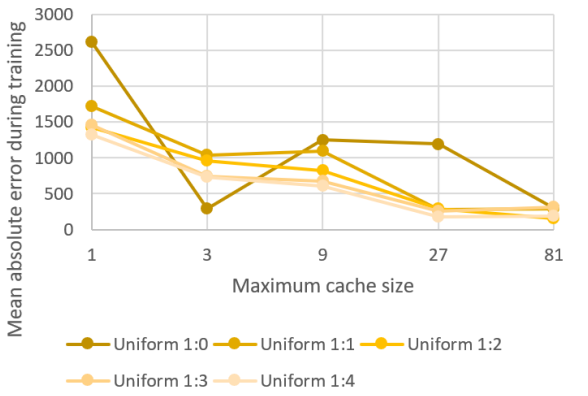

Figure 11. The effect of increasing the training cache size (logarithmic scale) on accuracy for various insertion-lookup interspersion ratios.

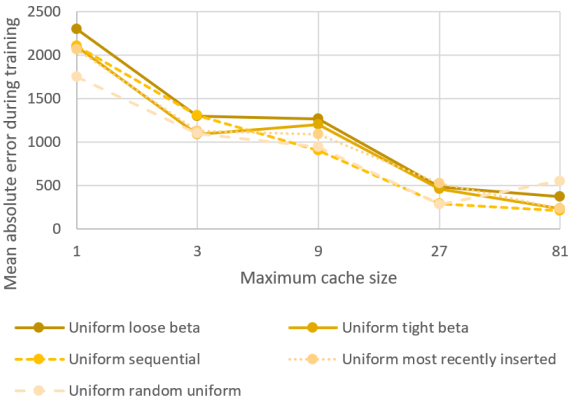

Figure 12. The effect of increasing the training cache size (logarithmic scale) on accuracy for various lookup patterns.

5.6. Lookup interspersion ratio and pattern

The model's performance improves markedly when the lookup interspersion ratio, i.e. the number of lookups of existing keys during training, is increased (in Figure 11, darker lines represent fewer lookups during training, and lighter lines represent more lookups). This observation reiterates the importance of having a feedback mechanism in our learned index architecture to assist in retraining the model on true positions that are found during lookup operations.

Furthermore, our learned index architecture achieves similar accuracy across all lookup patterns (Figure 12). This suggests that our learned index framework is able to adapt satisfactorily to variations in data access patterns.

In particular, our learned index architecture optimizes for tightly distributed lookups, especially at the smallest cache sizes. Figure 13 shows the reduction in accuracy of the index during evaluation, where all keys are uniformly accessed,

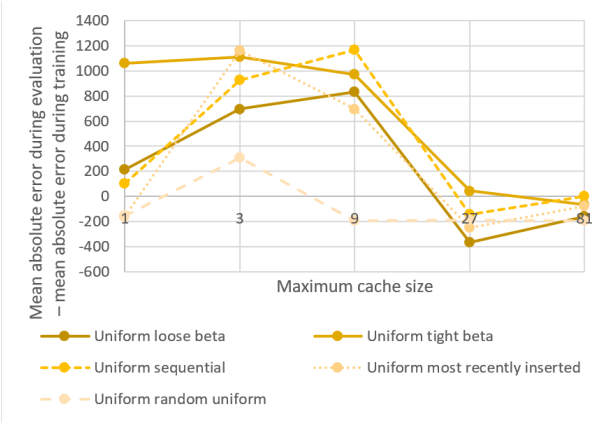

Figure 13. The effect of increasing the training cache size (logarithmic scale) on the *difference* between the accuracy of the index during evaluation (looking up all keys across the entire key space distribution) and the accuracy of the index during training, for various lookup patterns during training.

as compared to its accuracy for the keys accessed during training, which follows the captioned lookup pattern. At small training cache sizes, for the tight β distribution lookup pattern, and to a lesser extent the loose β distribution lookup pattern, the index displays a large disparity in accuracy between training and evaluation. This suggests that the index is optimizing strongly for the frequently looked up keys in the modal distributions. We attribute this behavior to the feedback mechanism of our learned index architecture, where the model is trained on the most frequently looked up keys, increasing the accuracy for those particular keys.

6. Conclusion

In this paper, we address three key challenges associated with designing an adaptive learned index framework: key drift, trade-off between accuracy and space consumption, and variations in data access patterns. To overcome these challenges we propose a novel learned index architecture that uses a mixture-of-experts model, trained on a training cache consisting of updated key-position mappings of recently accessed keys, along with a feedback mechanism that allows the learned index to optimize over various data access patterns. Our architecture allows the index to quickly learn unseen true positions without affecting other positions the model has learned, as well as accommodate changing data distributions and data access patterns by allowing the learned index to quickly improve its prediction for frequently accessed keys.

We evaluate the accuracy of our learned index framework on synthetically generated datasets that follow the uniform, Gaussian and mixture of Gaussian distributions, by analyzing various training conditions for index hyperparameter values. We demonstrate that our learned index framework is capable of adapting to varying data access patterns, provides high predictive accuracy with an increase in cache size and number of experts, and permits a tradeoff between space consumption and accuracy.

6.1. Directions for future research

Though our learned index framework was evaluated thoroughly using synthetic datasets, we attempted to evaluate our learned index framework on SOSD benchmarking framework (Kipf et al., 2019), which provides a variety of large, real-world datasets (200 million keys) and baseline implementations to compare the performance of traditional index structures against learned index structures. Unfortunately, we did not have the necessary system requirements needed to evaluate our learned index framework on these huge datasets.

The other major data access operation, not considered in our study, is that of deletion of keys from the underlying sorted array. Deletion patterns pose even more significant issues for empirical evaluation of learned index models because of the possibility of interspersion of deletions with insertions. Although neural network training can be reversed, e.g. (Maclaurin et al., 2015), little is understood about how to "remove" a training sample from a neural network.

We also intend on investigating the throughput of our learned index framework and comparing it with existing traditional index structures. Our framework's architecture lends itself well to adaptive batching (Crankshaw et al., 2017), wherein multiple requests from the user can be batched together depending on the workload and latency requirements, improving the throughput of our learned index framework during training and prediction serving.

The hypothesis space of the mixture-of-experts neural network architecture can contain arbitrary functions, which does not completely match the hypothesis space of CDFs, which are positive, strictly monotonic functions. At the moment, we know of no neural network architecture which can constrain the learned function to this hypothesis space; imposing such constraints may be effective at improving the rate at which the index learns.

Distribution of contributions

The team members contributed equally to the project.

References

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 613–627, 2017.

Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., and Kraska, T. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*, pp. 1189–1206, 2019.

Goldstein, J., Ramakrishnan, R., and Shaft, U. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*, pp. 370–379. IEEE, 1998.

Graefe, G. and Larson, P.-A. B-tree indexes and cpu caches. In *Proceedings 17th International Conference on Data Engineering*, pp. 349–358. IEEE, 2001.

Hadian, A. and Heinis, T. Considerations for handling updates in learned index structures. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pp. 1–4, 2019.

Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A., and Dubey, P. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 339–350, 2010.

Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., and Neumann, T. Sosd: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014*, 2019.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 489–504, 2018.

Leis, V., Kemper, A., and Neumann, T. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 38–49. IEEE, 2013.

Maclaurin, D., Duvenaud, D., and Adams, R. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pp. 2113–2122, 2015.

Magdon-Ismail, M. and Atiya, A. F. Neural networks for density estimation. In *Advances in Neural Information Processing Systems*, pp. 522–528, 1999.

Melo, B. d., Milioni, A. Z., Júnior, N., and Lucio, C. Daily and monthly sugar price forecasting using the mixture of local expert models. *Pesquisa Operacional*, 27(2): 235–246, 2007.

Miller, D. J. and Uyar, H. S. A mixture of experts classifier with learning based on both labelled and unlabelled data. In *Advances in neural information processing systems*, pp. 571–577, 1997.

Orhan, E. eminorhan/mixture-of-experts, Jul 2016. URL https://github.com/eminorhan/mixture-of-experts/blob/master/DenseMoE.py.

Rao, J. and Ross, K. A. Making b+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pp. 475–486, 2000.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017. URL http://arxiv.org/abs/1701.06538.

Tang, C., Wang, Y., Dong, Z., Hu, G., Wang, Z., Wang, M., and Chen, H. Xindex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 308–320, 2020.

Wu, Y., Yu, J., Tian, Y., Sidle, R., and Barber, R. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data*, pp. 1223–1240, 2019.

Zhang, H., Andersen, D. G., Pavlo, A., Kaminsky, M., Ma, L., and Shen, R. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*, pp. 1567–1581, 2016.