Considerations for Handling Updates in Learned Index Structures

Ali Hadian
Imperial College London
hadian@imperial.ac.uk

Thomas Heinis
Imperial College London
t.heinis@imperial.ac.uk

ABSTRACT

Machine learned models have recently been suggested as a rival for index structures such as B-trees and hash tables. An optimized learned index potentially has a significantly smaller memory footprint compared to its algorithmic counterparts, which alleviates the relatively high computational complexity of ML models. One unexplored aspect of learned index structures, however, is handling updates to the data and hence the model. In this paper we therefore discuss updates to the data and their implications for the model. Moreover, we suggest a method for eliminating the drift – the error of learned index models caused by the updates to the index– so that the learned model can maintain its performance under higher update rates.

KEYWORDS

Data management, machine learning, indexing

1 INTRODUCTION

ML-based range index models such as the RMI model [6, 7] learn the underlying CDF of the key distribution. Given a sorted array of keys, the model θ maps the input key k to its estimated position in the sorted array:

$$\text{pos}_\theta(k) = F_\theta(\text{k}) \times N \tag{1}$$

Here, $F_\theta(k)$ is the CDF of keys estimated by model θ, and N is the number of data points in the sorted set of keys, and $p_\theta(k)$ is the estimated position. In a range query, $p_\theta(k)$ indicates the first key in the key set that is equal or higher than the look-up key. Learned index models can be very efficient if their estimated position of the keys are very close to the actual positions. For read-only queries, these indexes are shown to rival classical indexes such as B-trees [6, 7].

Extensive research has been recently done on adopting machine learning techniques for indexing problems. This includes ML-based inverted indexes [9], learned bloom filters [8, 9], and linear learned indexes [2, 3]. Machine learning has been also been applied to other problems in database systems [4, 5].

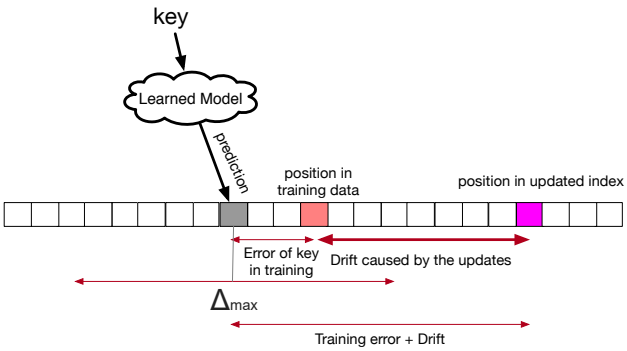


Figure 1: A learned index can perform binary search within a Δ_{min}-sized error window. If the index is updated, the prediction error can exceed Δ_{min}

1.1 Handling Updates

Inserting and deleting keys from the index can drastically degrade performance. Each insertion or deletion slightly modifies the empirical CDF of the keys. If the empirical CDF of the modified index drifts significantly from the base (initial) CDF, it will drastically degrade the accuracy and hence the performance of the learned index.

A drift also violates the guaranteed error window of a learned index, which is essential for enabling binary search to find the actual position of the query key. Figure 1 illustrates this issue. The distance between the predicted position and the actual position of the given key is the training error of the index. If the index is read-only, one can measure the maximum training error Δ_{max} for all the keys of the index. This allows to do a binary search in a window of size Δ_{max} [7]. However, if the sorted index is updated, the insertions can increase the distance between the predicted and actual positions, such that the actual key is no longer guaranteed to be within Δ_{max} distance from the estimated position.

aiDM'19, July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6802-5/19/07…$15.00
https://doi.org/10.1145/3329859.3329874

In the following, we discuss alternative approaches to retraining the learned index that can be used for alleviating the drift in the key distribution. Then, we suggest a low-cost method for preventing the loss of accuracy for a learned index in face of heavy update workloads.

2 MODELLING THE UPDATE DISTRIBUTION

A learned index determines the position of a key by modelling the CDF of the key distribution. By considering the CDF of keys in the update workloads, we can extend this approach to predict the position of keys in an updated index. Suppose that $F_b(k)$ is the empirical CDF of the base key distribution (for the training data) of size N_b. After the updates, say N_i insertions and N_d deletions, the index contains $N = N_b + N_i - N_d$ keys. The position of a key k in the updated index is:

$$pos(k) = NF(k) = N_b F_b(k) + N_i F_i(k) - N_d F_d(k) \quad (2)$$

F(k) is the empirical CDF of the updated index and $F_i(k)$ and $F_d(k)$ are the empirical CDFs of insertions of deletions, respectively. If we know the empirical distributions $F_b(k)$, $F_i(k)$, $F_d(k)$ one can estimate the position of the key in the updated array of sorted keys.

Modelling deletions. $F_d(k)$ is the CDF of the delete distribution, which indicates which portion of the the deleted keys were located before k. Equation 2 assumes that the deleted keys are removed from the sorted array. Nonetheless, even if we delete the keys from the index, Equation 2 does not violate the monotonically increasing property of the F(K), because if a key k is deleted, it means that k should had also been observed earlier in the base or insert workloads. However, when modelling the empirical distributions with continuous functions, say $F_{b,\theta}$ (learned model),$F_{i,\theta}$, and $F_{d,\theta}$; we should use models for $F_{b,\theta}, F_{i,\theta}, F_{d,\theta}$ in a way that $F_\theta(k)$ is monotonically increasing.

A more efficient alternative is to assume that the deleted keys remain in the array but are flagged as deleted, as suggested by Kraska et al. [7]. In this way, the position can be determined by $F_b(k), F_i(k)$:

$$pos(k) = NF(k) = N_b F_b(k) + N_i F_i(k) \quad (3)$$

which yields a statistically valid CDF for $F(k)$ for arbitrary CDF models of F_b, F_i.

In the rest of this paper, we assume that the deleted keys are only flagged and are not removed from the index. Therefore, we use Equation 3 for position estimation.

2.1 Measuring the Drift

The average lookup error in a learned index is the difference between the estimated position for a key k and its actual position. The error depends on two factors:

(1) **Training error**: A training error occurs when the learned model $F_{b,\theta}(k)$ does not exactly match the empirical distribution of the training data $F_b(k)$. Therefore, the training error for each key is $N|F_{b,\theta}(k) - F_b(k)|$. If model θ can 'memorize' the positions of the training keys, the training error will be zero.
(2) **Concept drift due to updates**: This happens when distribution of the updates $F_i(k)$ does not follow the training set distribution, therefore the empirical CDF of keys after the update $F(k)$ is different than the CDF at training time $F_b(k)$. We define it as $\Delta_{F_b} = |F_b(k) - F(k)|$

It is trivial that the error of the model for each key, namely Δ_θ, is bounded by the training error and the drift:

$$\begin{aligned}\Delta_\theta(k) &= |\text{pos}_\theta(k) - \text{pos}(k)| = N|F_{b,\theta}(k) - F(k)| \\ &\leq N \underbrace{|F_{b,\theta}(k) - F_b(k)|}_{\text{Training error}} + N \underbrace{|F_b(k) - F(k)|}_{\text{Concept drift} = \Delta_{F_b}} \end{aligned} \quad (4)$$

For the sake of simplicity, we assume that the training error is zero, so the trained model computes the exact empirical distribution of the base data. $F_{b,\theta}(k)$ is thus equal to $F_b(k)$ and we use them interchangeably.

If the model is oblivious to the updates, the estimated position will be $N \cdot F_b(k)$. The actual position of k is $N \cdot F(k)$ where $F(k)$ is the empirical distribution of current keys. Therefore, the *lookup error caused by the inserts* is $\Delta(k) = N(F(k) - F_b(k))$. Recall that $NF(k) = N_b F_b(k) + N_i F_i(k)$, hence the drift for point k is:

$$\begin{aligned}\Delta_{F_b}(k) &= |\underbrace{N_b F_b(k) + N_i F_i(k)}_{NF(k)} - N F_b(k)| \\ &= N_i |F_i(k) - F_b(k)| \end{aligned} \quad (5)$$

This means that when the insertions follow a different distribution than that of the base data, **the error of the model grows linearly with the number of inserted items**.

3 DRIFT CORRECTION

Is it efficient to model the updates? A key question is whether it is worth, from a performance point of view, to learn the update distributions for correcting the estimated position? Or shall we instead solely rely on the learned index and use linear/binary local search for correcting its error? Learning the update distributions incurs computational overhead, and it takes more CPU cycles to compute and apply the corrections to the prediction of the base $F_b(k)$ model. However, we argue that if the adaptation mechanism is lightweight yet effective, it will add only a few cycles, but the improved precision will save more CPU cycles because fewer items need to be scanned. Note that the superiority of learned indexes is due to them fitting the index/model in the cache

and thus drastically reducing main memory look-ups, which saves more clock cycles than doing computations. The same principle applies to the update models: if the update model is still compact enough such that the base and update models fit in cache, the performance gains can be maintained. Such an approach potentially saves more time compared to re-training the learned index or switching to a traditional indexing algorithm for update-heavy workloads.

3.1 Update-adaptive Models

Some index models can inherently handle updates. For example, the IFB-tree is a modified version of B-tree which uses linear interpolation if the accuracy is guaranteed to be in a pre-defined error tolerance $\Delta_{\max}$. However, since the IFB-tree still maintains the underlying B-tree, the updates are handled in $O(n \log n)$ time [3]. Also, the A-tree is a learned index based on the RMI model by Kraska et al. [7], but adopts linear models instead of deep models. Therefore, re-training each model can be done in shorter time.

On the downside, models that can easily adapt to the upcoming distributions are either simple linear models or use classical index structures alongside, limiting the potential performance gain achievable by a learned index. In other words, complex non-linear models such as deep neural networks that can learn non-linear CDFs in a compact representation, are not an easy match for incremental learning (i.e., adapting to new training sets without re-training the CDF model).

3.2 Drift estimation with reference points

In this section, we suggest a method for correcting the drift caused by the updates. Our method does not require any explicit modelling of the update distribution. Instead, we assume that the drift for a few points in the key set, called the 'reference points', is known. This allows us to estimates the drift of a point in the key distribution by considering the drift of its closest reference points.

Consider a given query k. The predicted position of k in the training data (before applying the updates) and the position in the updated index are $N_b F_b(k)$ and $NF_b(k)$, respectively. We are interested to know the drift of k, which is $\Delta_{F_b}(k) = N(F_b(k) - F(k))$, but the value of $F(k)$ is unknown. Now suppose that there exists a key from the the training data, say r, for which we know the exact location of r in both the trained and the updated indexes. If r is close enough to k, we can assume that the amount of drift for k is also similar to the drift of r, i.e. $\Delta_{F_b}(k) \simeq \Delta_{F_b}(r)$, hence we can leverage r as a reference point to correct the predicted position of k by subtracting the drift from the prediction.

A more precise correction can be applied if we know *two* reference points on both sides of the query point. In this case, we can use *interpolation* to estimate the drift at k using the left-side ($r_L \leq k$) and right-side ($r_R \geq k$) reference points. If we know the drifts of r^L, r^R, namely $\Delta_{F_b}(r^L)$, $\Delta_{F_b}(r^R)$, we can estimate the drift for k by interpolating based on the distance of k's position to in the original training distribution. The position of two keys in the training data (original index) is $N_b(F_b(k_2) - F_b(k_1))$, hence the interpolated drift is:

$$\hat{\Delta}_{F_b}(k) = \Delta_{F_b}(r^L) + \frac{\Delta_{F_b}(r^R) - \Delta_{F_b}(r^L)}{F_b(r^R) - F_b(r^L)}(F_b(k) - F_b(r^L)) \quad (6)$$

The estimated position of k after eliminating the drift is:

$$\hat{\text{pos}}(k) = NF_b(k) - \hat{\Delta}_{F_b}(k) \quad (7)$$

Drift correction is done merely to maintain the performance of an updated learned index. Since running a learned index model is computationally expensive, it is not efficient to run the model against the reference point keys, so we should run Equation 6 without explicitly computing $F_b(r^L)$, $F_b(r^R)$. Moreover, computing the drift should be implemented with a small set of lightweight operations that can be executed in a few CPU cycles. Therefore, we use the "positions" of the reference points in the both indexes instead of using their values.

Given the base index of size N_b, we can think of the index as a set of segments of size 2^M. For a given query point k, we can estimate its position in the base index, which is $\text{pos}_b(k) = N_b F_b(k)$. The segment that contains k in the base index is:

$$\text{SEG}(k) = [\text{pos}_b(k)/2^M] \quad (8)$$

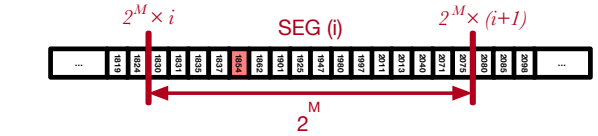
We take the first key in each segment as a reference point, thus $\text{pos}_b(r^L) = 2^M \text{SEG}(k)$ and $\text{pos}_b(r^R) = 2^M(\text{SEG}(k) + 1)$.

The positions of each of the reference points in the new index, say $\text{pos}(r)$, is its position in the base index $\text{pos}_b(r)$, plus the number of keys that are inserted before the r's segment. Therefore, if we can efficiently keep track of "*the number of keys inserted before each segment*", shown as $I@\text{SEG}(r)$, then we can compute the drift at r. This is shown in figure 2. The correct position of r in the updated index is $pos_b(r) + I@\text{SEG}(r)$ and its estimated position using F_b is $NF_b(r) = N\frac{pos_b(r)}{N_b}$. Therefore, the drift of F_b for r is:

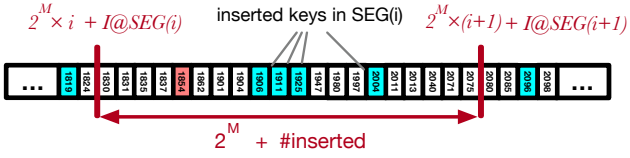
$$\Delta_{F_b}(r) = \text{pos}_b(r) \times \left(\frac{N}{N_b} - 1\right) - I@\text{SEG}(r) \quad (9)$$

Note on implementation. the advantage of choosing a segment size that is power of 2 is that the segment number and positions of the reference points can be computed by bit operations, e.g., using bit-masking to compute $\text{pos}_b(r)$ and $\text{SEG}(r)$ from $\text{pos}_b(k)$. Therefore, correcting the drift takes only a few clock cycles and does not add much overhead.

Updating per-segment insert counts. When a new key k is inserted to the index, we need to update the count of its segment in the base index $\text{SEG}(k)$. More importantly, drift



(a) Base index



(b) Updated index

Figure 2: Computing the drift using I@SEG counts

correction in lookup time requires computing $I@\text{SEG}(k)$, i.e., the total count of insertions in all segments before SEG(k). This is a prefix-sum problem which can be efficiently done with $\log\lfloor N_b \rfloor - M$ operations using a Fenwick tree [1]. However, if the number of segments ($N_b/2^M$) is small (e.g., < 64), it is more efficient to keep the insertions of each segment in an array and count the partial sum for each query using a vectorized (SIMD) implementation.

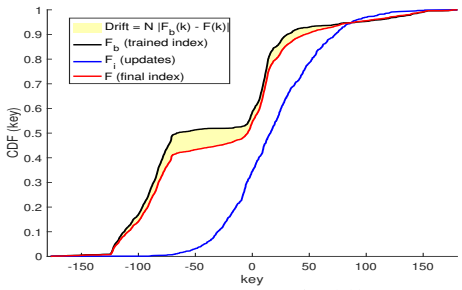
Even though counting the partial sums is very lightweight, the number of segments should be small so that the index and the segments' insert counts fit in cache altogether. Recall that the advantage of a learned index is that its learned model is compact enough to fit in cache (in contrast with classical indexes), so we can only store a few reference points. The straightforward solution is to use a very large M, e.g., splitting the entire dataset into less than 64 segments.

Figure 3 illustrates how only 64 segments can correct an index with 10M nodes. The base index data are sampled without replacement from the longitudes of over 1.5 billion locations extracted from the OpenStreetMap database. After inserting 1M extra nodes (10% of the original data), the resulting CDF deviates from the base data (Figure 3a). The mismatch between the base and update distributions yields large drifts of up to 1M nodes in position estimation. As shown in Figure 3b, the drift can be reduced by over 4 orders of magnitude using only 64 segments. Using more segments allows more correction.

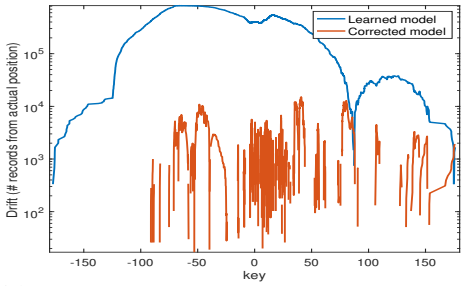
4 CONCLUSION

In this paper, we outlined different approaches for handling updates in a learned index structure. We demonstrated that when a learned index faces an update distribution that is different from its learned model, it starts to drift linearly with the number of inserted items. We suggest a method for alleviating this problem by correcting the error (drift) caused by the update framework.

We are waiting for the release of the Learning Index Framework (LIF) [7] – the only learned index library that generates



(a) CDFs of the base index ($F_b(k)$), the updates ($F_i(k)$), the final index ($F(k)$); and the drift



(b) Effect of correction on the prediction error

Figure 3: Drift correction in a learned index

efficient C++ code for a deep learned index structure, which execute models in the order of 30 nano-seconds – so that we can test our theoretical considerations in practice.

ACKNOWLEDGEMENT

We acknowledge the support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS, Grant Reference EP/L016796/1).

REFERENCES

[1] Peter M Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and Experience* 24, 3 (1994), 327–336.
[2] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2018. A-tree: A bounded approximate index structure. *arXiv preprint arXiv:1801.10207* (2018).
[3] Ali Hadian and Thomas Heinis. 2019. Interpolation-friendly B-trees: Bridging the Gap Between Algorithmic and Learned Indexes. In *EDBT*.
[4] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. In *ICML*.
[5] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *arXiv preprint arXiv:1809.00677* (2018).
[6] 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'19*.
[7] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. 489–504.
[8] Michael Mitzenmacher. 2018. Optimizing Learned Bloom Filters by Sandwiching. In *NIPS*.
[9] Harrie Oosterhuis, J. Shane Culpepper, and Maarten de Rijke. 2018. The Potential of Learned Index Structures for Index Compression. In *ADCS*.