Towards Practical Learned Indexing (Extended Abstracts)

Mihail Stoian
TUM
mihail.stoian@tum.de

Andreas Kipf
MIT CSAIL
kipf@mit.edu

Ryan Marcus
MIT CSAIL, Intel Labs
ryanmarcus@mit.edu

Tim Kraska
MIT CSAIL
kraska@mit.edu

ABSTRACT

Latest research proposes to replace existing index structures with learned models. However, current learned indexes tend to have many hyperparameters, often do not provide any error guarantees, and are expensive to build. We introduce Practical Learned Index (PLEX). PLEX only has a single hyperparameter ϵ (maximum prediction error) and offers a better trade-off between build and lookup time than state-of-the-art approaches. Similar to RadixSpline, PLEX consists of a spline and a (multi-level) radix layer. It first builds a spline satisfying the given ϵ and then performs an ad-hoc analysis of the distribution of spline points to quickly tune the radix layer.

AIDB Workshop Reference Format:
Mihail Stoian, Andreas Kipf, Ryan Marcus, Tim Kraska. Towards Practical Learned Indexing. *AIDB* 2021.

1. INTRODUCTION

We introduce Practical Learned Index (PLEX). Compared to existing learned indexes, PLEX only has a single hyperparameter ϵ (maximum prediction error) and is hence easy to use. PLEX builds upon RadixSpline [7] and CHT [2], and uses a spline layer to ensure an ϵ error bound and a radix layer that is inspired by CHT.

RadixSpline. RadixSpline (RS) [7] consists of a linear spline model that approximates the CDF of the data within an error bound and a radix table that indexes the computed spline points. The main feature of RS is that it can be built with a constant amount of work per new element, which allows a single-pass build phase. However, when the radix table is not able to properly index the spline keys (i.e., in the presence of outliers), RS can have poor performance.

PLEX addresses two problems of RS: (i) the radix layer is hard to parametrize and (ii) the radix layer can be affected by outliers. Despite the fact that PLEX needs to perform an additional pass over the spline points array, PLEX manages to retain the high build performance of RS.

3rd International Workshop on Applied AI for Database Systems and Applications (AIDB'21), August 20, 2021, Copenhagen, Denmark.

Hist-Tree. Hist-Tree (HT) [2] approximates the data distributed as a histogram, instead of using functions, with a fast radix tree-based traversal method to find the right histogram bucket. As such, HT has similarities with a traditional radix tree, as its indexing approach also uses the binary representation of the keys, and with a learned index as it tries to approximate the data distribution rather using the traditional key-comparison present in B-tree traversals. HT recursively splits the data into bins until a certain threshold for the number of elements is reached. Apart from allowing updates, Hist-Tree also comes in a *compact* version (CHT), optimized for read-only workloads. CHT is essentially a lookup table which stores the nodes of the initial HT and their prefix sums. CHT can either be built from a HT, by running a depth-first pre-order tree traversal, or, as we have implemented it in this work, directly from the data itself. The latter is more advantageous for our case, as keys can be processed in chunks. While CHT did not have an auto-tuner, we now introduce one as part of this work (auto-tuning CHT is part of our auto-tuning process).

Other Related Work. The first learned index, which paved the way for the development of new alternatives of index structures, is the recursive model index (RMI) [8]. RMIs use learned models, arranged in a hierarchical structure, which are trained via supervised learning techniques on the cumulative distribution function (CDF) of the underlying data. Since then, a large suite of learned index structures have been proposed. Few of them support writes: PGM-index, a multi-level structure, where each level represents an error-bounded piecewise linear regression [5] or ALEX [3], which combines insights from RMI with proven storage and indexing techniques. With regard to multi-dimensional data, there exist several other recent approaches, including Flood [12] and Tsunami [4].

2. PLEX

PLEX is a combination of CHT [2] and RS [7]. We build a spline on the underlying data, and then index the spline points in a CHT. The resulting index structure features fast build times, error-bounded lookups, and is easy to use as it has only one hyperparameter, the maximum error ϵ.

RS employs a radix table for indexing its spline points. However, when the keys of the dataset cannot be easily indexed by a radix table, i.e., the longest common prefixes of their binary representation are large, RS can have a decrease in performance. PLEX addresses this issue by replacing the radix table with a radix tree, represented by CHT. Like RS,

PLEX is built "bottom-up", by first constructing an error-bounded spline, which is then indexed in CHT.

Build Spline. The core part of PLEX is the linear spline model S, which approximates the position of each key k within ϵ positions, $\epsilon > 0$. Formally, if p^* is the position of k in the CDF and $\tilde{p}$ is the approximated position computed by the spline, then $|\tilde{p} - p^*| \leq \epsilon$. In other words, the spline model *always* predicts the correct location of the data within a maximum error of ϵ.

The error-bounded spline model is defined as a set of connected linear spline points, which are picked *from* the set of CDF points. An optimal spline, i.e., the one with the fewest number of spline points, can be computed via dynamic programming in $O(N^2)$, where N is the CDF size. Since this approach does not scale for large datasets, we use instead a greedy algorithm, which does not guarantee optimality anymore, but can be implemented in $O(N)$ [13].

A lookup consists of first determining in which spline segment σ the key k is located, i.e., between which spline nodes the key lies in, and then performing a linear interpolation in the respective segment. For more details on the error-bounded spline algorithm, we refer the reader to [13].

Build CHT. Once the spline points have been selected, they can be indexed in CHT, using our new implementation, which iteratively builds each level of the tree by analyzing chunks of keys. This is different from the proposed bulk-loading in [2], as our method *directly* builds CHT instead of first building a sparse tree.

CHT has two hyperparameters: the number of radix bits r of each tree node, i.e., the fanout of the tree equals 2^r, and the error δ within the index approximates the positions of the keys. Formally, if q^* is the position of a spline point and $\tilde{q}$ is its estimated position computed by CHT, then $q^* \in \{\tilde{q}, \ldots, \tilde{q} + \delta - 1\}$. This is a generalization of the radix table of RS, since setting $\delta = \infty$ leads to a CHT with a single node (a radix table). Notably, a radix table does not have a global bounded error (it must instead use the position of the *next* prefix to obtain an upper bound on $\tilde{q}$). This requires us to develop separate cost models for each of them (cf. Section 3).

Lookup. A lookup for key k starts by searching the position of the spline segment σ. This routine is done in two steps: First, a lookup in CHT returns an approximated position $\tilde{q}$. Next, a binary search is performed in the range $\{\tilde{q}, \ldots, \tilde{q}+\delta-1\}$ to find the exact position of σ. Subsequently, we perform a linear interpolation between the two spline points of σ to obtain an estimated CDF position $\tilde{p}$ of the key. Finally, we perform a binary search within the error bounds $\tilde{p} \pm \epsilon$ to find the first occurrence of k. We may use SIMD for the linear search

3. AUTO TUNING

One main drawback of (learned) index structures is the choice of hyperparameters, as they have to be manually tuned in order to find the best lookup time under certain constraints (e.g., space).

We introduce cost models that approximate the lookup times and accurately compute the space consumption for both radix table and CHT without building the actual data structures.

Radix Table. A radix table with parameter r splits the input data into 2^r buckets based on the first r most significant bits (prefix) of the keys. In RS, the input data for the radix table are the keys of the spline points. Therefore, a lookup for key k consists of first finding out in which radix bucket b_k the key is located, and then performing a local search on the spline nodes within the bucket to find the exact spline segment. If the local search is implemented as a binary search, then the number of steps equals $\lceil log_2(|b_k|) \rceil$, where $|b_k|$ is the number of spline points within bucket b_k.

Assume that only positive lookups are performed, i.e., the lookup key lies within the stored data D. Then the average lookup time λ_r can be estimated as

$$\lambda_r = \frac{1}{|D|} \sum_{k \in D} \lceil log_2(|b_k|) \rceil. \quad (1)$$

This cost model has the advantage that it can be computed for all r while building the spline model, without storing the actual radix tables. It also allows us to *detect* the outlier problem of RS: Consider, for example, $r = 1$. The optimal λ_1 is achieved when the radix table splits the set of spline points in two equal-sized buckets, i.e., $\lambda_1 = \lceil log_2(\frac{|S|}{2}) \rceil = \lceil log_2(|S|) \rceil - 1$. Note that we increased the number of bits and the cost decreased by $\lambda_0 - \lambda_1 = 1$ unit ($r = 0$ corresponds to a simple binary search, i.e., $\lambda_0 = \lceil log_2(|S|) \rceil$). This is not always the case: when all spline keys have the same value for the most-significant bit, then the radix table is not able to split the initial bucket and we have $\lambda_1 = \lambda_0$. In general, $\lambda_r - \lambda_{r+1} \in [0, 1]$ tells us whether it is worth increasing the number of radix bits by one. Since we must update the model for each radix table of parameter r as each spline node is computed, the time complexity is $O(r^+|S|)$, where r^+ is the maximum r allowed.

Finally, the memory consumption of a radix table with parameter r is $O(2^r)$, as we only need to store a value per bucket, namely the position of the first spline point with that prefix.

CHT. The same reasoning can be applied for CHT. The aim is to consider a large set of CHTs with different configurations (r, δ) and estimate their average lookup time and memory consumption.

A lookup for key k consists of first finding out in which node v_k of CHT the key is located, i.e., the node which contains the bin corresponding to k with size $\leq \delta$, and then performing a local search on the spline nodes within that bin. If the local search is implemented as a binary search, then the number of search steps equals $depth(v_k) + \lceil log_2(\delta) \rceil$. As the term $\lceil log_2(\delta) \rceil$ is the same for all lookup keys, we only have to compute the average depth of the leaf nodes.

In the cost model for radix table (Eq. 1), the cost of each bucket is weighted by the number of data keys falling into that bucket. This is more difficult for CHT, as we would have to examine the original data multiple times, namely, as many times as the number of levels, to collect such statistics. Instead, we only calculate the average tree depth given lookups from the set of spline keys. Note that this is a simplification that does not guarantee that we can still accurately model the average lookup time for the data itself. This was not a problem for the radix table, where there is only one radix level to consider, and this one level can be covered when building the spline. Thus, the average lookup time can be estimated as

$$\lambda_{(r,\delta)} = \lceil log_2(\delta) \rceil + \frac{1}{|S|} \sum_{k \in S} depth(v_k). \quad (2)$$

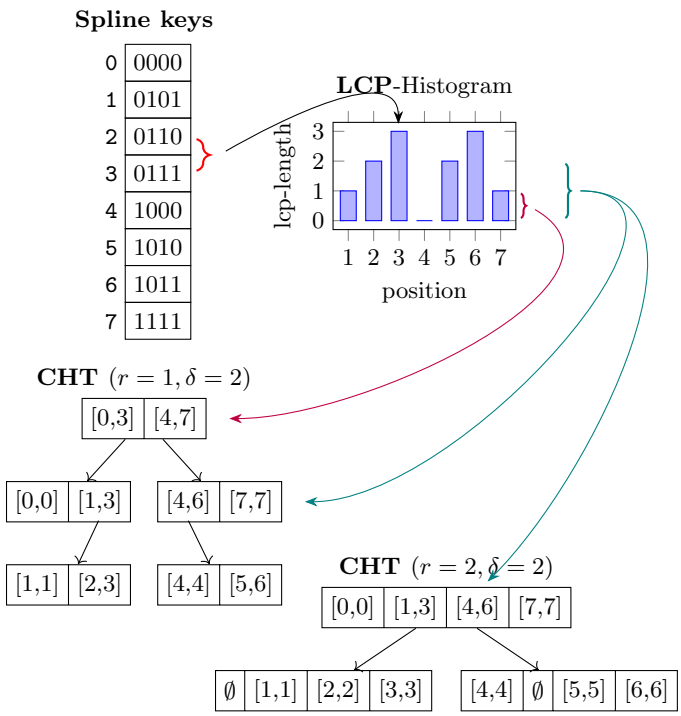

Figure 1: The lcp-histogram and depiction of strategy for computing the average tree depths of a large set of trees with different configurations (r, δ). Lcp-length $p = 1$ is used by the CHT with $r = 1$, but not also by the one with $r = 2$, while lcp-length $p = 2$ is used by both CHTs.

To compute the average tree depths for different pairs (r, δ), we make use of the longest common prefix (lcp) of any two adjacent spline keys. To this end, we build the *lcp-histogram* whose values are defined as $lcp_i := lcp(keys_i, keys_{i-1}), \forall i \in [|S| - 1]$ (cf. Fig. 1).

The idea is to traverse each lcp-length p of the histogram and filter out the positions whose values are smaller than p. The *surviving* positions form contiguous sequences representing exactly the intervals covered by bins in particular CHTs. If the length of a contiguous sequence exceeds the value of δ for a particular CHT, i.e., the corresponding bin of that interval is not a final one, then that bin *increases* the search path by one for each key k passing through it.

To illustrate this method, consider again Fig. 1: At lcp-length $p = 1$, the surviving positions, after we filter out the positions whose values are less than 1, form the contiguous sequences $1, 2, 3$ and $5, 6, 7$. Since *lcp* also considers the previous key, the sequences actually define the intervals $[0, 3]$ and $[4, 7]$ from the set of spline keys, which can be seen in the first node of CHT$(r = 1, \delta = 2)$. Filtering out further in the same way at $p = 2$, we again get two contiguous sequences, $2, 3$ and $5, 6$, representing the intervals $[1, 3]$ and $[4, 6]$, respectively. These define the only two bins in both CHT$(r = 1, \delta = 2)$ and CHT$(r = 2, \delta = 2)$ that have outgoing pointers, as the length of both intervals exceeds $\delta = 2$.

In the same figure we see that lcp-length $p = 1$ is *used* by the CHT with $r = 1$, i.e., the intervals are present in the bins of that particular CHT, but not also by the one with $r = 2$, while $p = 2$ is *used* by both. Formally, a lcp-length p is *used* by a particular CHT(r, δ) if $p \equiv_r 0$.

Algorithm 1 Computation of average lookup times for multiple pairs (r, δ).

Input: upper-bounds $r^+, \delta^+ \in \mathbb{N}$ on r and δ, respectively
Output: $\lambda_{(r,\delta)}, \forall 1 \leq r \leq r^+, 1 \leq \delta \leq \delta^+$

```
1:  intervals ← {[1, |S| − 1]}
2:  for all lcp-length p do
3:    for all I in intervals do
4:      for all maximal subinterval J ⊆ I,
        s.t. lcp_j ≥ p, ∀j ∈ J do
5:        for all r in 1, ..., r+ s.t. p ≡_r 0 do
6:          depth_r(min(|J| − 1, δ+)) += |J|
7:        end for
8:      end for
9:    end for
10:   intervals ← newly found subintervals
11: end for
12: for all r in 1, ..., r+ do
13:   for all δ in δ+ − 1, ..., 1 do
14:     depth_r(δ) += depth_r(δ + 1)
15:   end for
16:   for all δ in 1, ..., δ+ do
17:     λ_(r,δ) ← ⌈log2(δ)⌉ + depth_r(δ)/|S|, acc. Eq. 2
18:   end for
19: end for
```

The method for computing $\lambda_{(r,\delta)}$ for different pairs (r, δ) is given in Alg. 1. The algorithm receives as input the upper-bounds r^+ and δ^+ on r and δ, respectively, and outputs average lookup times for all possible pairs. In Line 1, we initialize the list of intervals with the entire range of keys, i.e., $[1, |S|-1]$. Then we iterate over all lcp-lengths p and split the current intervals at positions with values $< p$ (Line 4). Each new subinterval J is consumed by updating $depth_r$ for all r that *use* the current lcp-length p, i.e., that satisfy $p \equiv_r 0$. In the end, $depth_r(\delta)$ will store exactly the term $\sum_{k \in S} depth(v_k)$ from Eq. 2.

As mentioned earlier, if the length of the interval exceeds the value of δ for a particular CHT, i.e., the corresponding bin of that interval is not a final one, then that bin increases the search path by one for each key k that passes through it. The maximum δ for which this occurs is $|J|-1$. However, it would be expensive to update all $\delta \leq |J| - 1$. Therefore, we employ suffix sums and only update $depth_r(|J| - 1)$ in Line 6, while completing the update step for the others in Line 14. The algorithm finishes by computing $\lambda_{(r,\delta)}$ according to Eq. 2. Since we have at most 2^p intervals for each lcp-length p, the algorithm takes $O(r^+(|S| + d^+))$ time.

As mentioned in [2], CHT can be stored as a flat array (there is no need for pointers), by representing each node as 2^r cells. Thus, the memory equals the number of nodes times 2^r. The number of nodes can be computed exactly using the above strategy, since each *surviving* interval represents exactly a node. For instance, CHT$(r = 1, \delta = 2)$ from our example has four nodes apart from the root node, represented by the aforementioned intervals $[0, 3]$, $[4, 7]$, $[1, 3]$, and $[4, 6]$, respectively.

PLEX. The auto-tuning of PLEX relies on the optimization of both the radix table and CHT. After λ_r and $\lambda_{(r,\delta)}$ have been computed, we can choose the best average lookup time

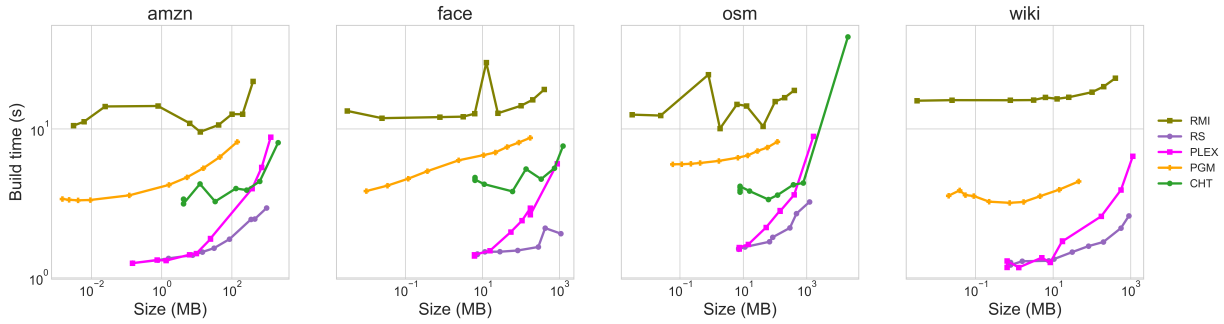

Figure 2: Size versus build time on the four real-world datasets from SOSD.

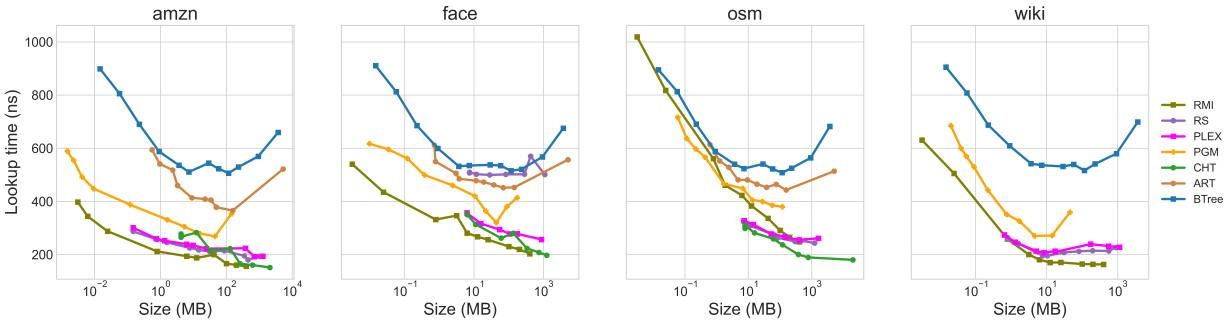

Figure 3: Size versus lookup time on the four real-world datasets from SOSD.

under the constraint that the space consumption does not exceed the space requirement of the spline model, i.e., in the worst case the final index structure is twice the size of the spline model. This way, PLEX requires only a single hyperparameter ϵ.

4. EVALUATION

Setup. Experiments are conducted on a m5zn.metal AWS machine with 192 GiB of RAM and 48 vCPUs (4.5 GHz turbo).

Datasets. We evaluate PLEX using the SOSD benchmark [6, 10]. We use four 64-bit datasets, each of them containing 200 M key/value pairs (3.2 GiB): amzn (book popularity data), face (Facebook user IDs), osm (composite cell IDs from Open Street Map) and wiki (timestamps of Wikipedia edits). See [10] for details on these datasets.

Build Time. We compare indexes regarding build time, as this highlights how expensive the learning process for each dataset is. We therefore consider the *build time / size* trade-off offered by RMI, CHT, PGM, RS, and PLEX, respectively, for each dataset, as shown in Fig. 2. RS achieves the lowest build times, due to its single-pass build phase. Note that the build time of PLEX already includes the auto-tuning time, unlike RS, CHT, or RMI [11], which were tuned offline via an expensive grid search. Our current implementation of CHT does not support key duplicates, which is the case for the `wiki` dataset. PLEX does not have this problem, as the spline keys are unique.

Probe Time. We further compare with classical index structures, namely ART [9] and BTree [1]. In the following, we consider the *performance / size* trade-off offered by each index structure for each dataset, as shown in Fig. 3. A first observation is that PLEX successfully solves the outlier problem of RS on the `face` dataset, while preserving the performance of RS on the other datasets. As pointed out in [10], the `osm` dataset is difficult to learn, as evidenced by the behavior of RMI on this dataset. However, that is exactly where the radix approach of RS and PLEX is beneficial. An interesting fact to observe for both `amzn` and `face` datasets is the constant offset of $\approx$ 25ns between PLEX and RMI. It should also be noted that for almost all datasets, PGM, ART, and BTree eventually lose performance as the index size is increased, because the cost of navigating the index structure no longer makes up for the time required to binary search the underlying data.

Auto Tuning. We have empirically verified that our auto-tuning strategy succeeds in finding optimal configurations for PLEX, which have been found by grid-search on the hyperparameter triple (ϵ, r, δ), with $\epsilon, \delta \in \{2^1, \ldots, 2^{10}\}$ and $r \in \{1, \ldots, 10\}$. The auto-tuner correctly decides to fall back on the radix table as its subindex on all datasets except the `face` dataset (where it successfully detects the outlier problem and decides to use CHT). In some cases, it even outperforms the manually-tuned PLEX because it explored more options for r and δ.

5. CONCLUSIONS

We have introduced PLEX, an auto-tuned learned index which offers a better trade-off between build and lookup time than state-of-the-art approaches. PLEX is easy to use, as its only hyperparameter is the maximum prediction error. In future work, we plan to extend PLEX to support efficient updates by using a Fenwick tree on each tree node.

Acknowledgments. This research is supported by Google, Intel, and Microsoft as part of DSAIL at MIT, and NSF IIS 1900933. This research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

6. REFERENCES

[1] R. Bayer and E. M. Mccreight. Organization and maintenance of large ordered indexes. *Acta Inf.*, 1(3):173–189, Sept. 1972.

[2] A. Crotty. Hist-Tree: Those who ignore it are doomed to learn. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[3] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska. ALEX: an updatable adaptive learned index. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 969–984. ACM, 2020.

[4] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proc. VLDB Endow.*, 14(2):74–86, 2020.

[5] P. Ferragina and G. Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, 2020.

[6] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[7] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. RadixSpline: a single-pass learned index. In R. Bordawekar, O. Shmueli, N. Tatbul, and T. K. Ho, editors, *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, pages 5:1–5:5. ACM, 2020.

[8] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 489–504. ACM, 2018.

[9] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, page 38–49, USA, 2013. IEEE Computer Society.

[10] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.

[11] R. Marcus, E. Zhang, and T. Kraska. CDFShop: Exploring and optimizing learned index structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2789–2792, New York, NY, USA, 2020. Association for Computing Machinery.

[12] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 985–1000. ACM, 2020.

[13] T. Neumann and S. Michel. Smooth interpolating histograms with error guarantees. In W. A. Gray, K. G. Jeffery, and J. Shao, editors, *Sharing Data, Information and Knowledge, 25th British National Conference on Databases, BNCOD 25*, volume 5071 of *Lecture Notes in Computer Science*, pages 126–138, Cardiff, UK, July 2008. Springer.