LSI: A Learned Secondary Index Structure

Andreas Kipf
MIT CSAIL
Cambridge, MA, USA
kipf@mit.edu

Dominik Horn
MIT CSAIL
Cambridge, MA, USA
horndo@mit.edu

Pascal Pfeil
MIT CSAIL
Cambridge, MA, USA
pfeil@mit.edu

Ryan Marcus
University of Pennsylvania, Intel Labs
Cambridge, MA, USA
ryanmarcus@mit.edu

Tim Kraska
MIT CSAIL
Cambridge, MA, USA
kraska@mit.edu

ABSTRACT

Learned index structures have been shown to achieve favorable lookup performance and space consumption compared to their traditional counterparts such as B-trees. However, most learned index studies have focused on the primary indexing setting, where the base data is sorted. In this work, we investigate whether learned indexes sustain their advantage in the secondary indexing setting. We introduce Learned Secondary Index (LSI), a first attempt to use learned indexes for indexing unsorted data. LSI works by building a learned index over a permutation vector, which allows binary search to performed on the unsorted base data using random access. We additionally augment LSI with a *fingerprint vector* to accelerate equality lookups. We show that LSI achieves comparable lookup performance to state-of-the-art secondary indexes while being up to 6× more space efficient.

ACM Reference Format:
Andreas Kipf, Dominik Horn, Pascal Pfeil, Ryan Marcus, and Tim Kraska. 2022. LSI: A Learned Secondary Index Structure. In *Exploiting Artificial Intelligence Techniques for Data (aiDM'22), June 17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3533702.3534912

1 INTRODUCTION

Unlike traditional index structures such as B-trees, learned indexes [14] build a model over the underlying data to predict the position of a lookup key in a sorted array. Learned indexes effectively compress the cumulative distribution function (CDF) of the data. When the underlying data has a learnable pattern, the resulting learned index can be both faster and smaller than its traditional counterpart. Learned index structures can be built in a variety of ways (e.g., top down [14, 19], bottom up [9, 13]), but all learned index structures provide (1) a mapping from keys to predicted positions, and (2) the maximum error that prediction can incur. Exact lookups can thus be performed via binary search on the underlying data, restricted to the area around the predicted position. While the initial proposal [14] considered neural networks as a building block, current proposals use simple functions which are fast to build and evaluate [23]. Extensive studies including the Search on Sorted Data benchmark [12, 18] and an independent analysis by Maltry and Dittrich [17] have shown that learned indexes are competitive with their traditional counterparts [11, 16] in at least one specific scenario: equality and range lookups of integer keys in a sorted, in-memory array.

Since their initial conception, learned indexes have been extended to support updates [7, 20, 26], strings [24], spatial data [22, 28], and disk-based systems [3, 6]. However, all of these proposals use learned indexes in a "clustered index" setting: where the underlying data is already sorted. In other words, existing proposals are primary index structures. The case when the underlying data is not sorted is also important (and arguably more common, as an instance of a table can only be sorted by a single key). Unfortunately, this case has received very little attention. As Ferragina and Vinciguerra point out [8], the most obvious way to use learned indexes in a secondary index scenario is to store sorted `(key, pointer)` pairs, very much like what is being stored in the leaf nodes of a B+-tree. These sorted pairs are the dominant size component of a traditional index, so the space overhead between a traditional and learned secondary index would be roughly similar. This raises the question whether the superior lookup efficiency and space savings of learned indexes prevail in the secondary indexing case.

In this paper, we introduce Learned Secondary Index (LSI)[1]. LSI is based on PLEX [25], which is a bottom-up learned index combining RadixSpline [13] and Hist-Tree [5]. We selected PLEX for its simplicity, since PLEX uses only one hyperparameter (the maximum prediction error). LSI addresses the following problem: given an unsorted, in-memory array of integer keys, find the smallest key that is greater than or equal to the lookup key (lower-bound lookup). Our key insight is that we do not need to explicitly store the key array (like in B+-tree leaf nodes) – instead, we store a *permutation vector* that stores a mapping between the key's position in a sorted order and the unsorted position of each key. Using PLEX's prediction, we index into this permutation vector and perform a binary search on the underlying (unsorted) data array using random access. For equality lookups, we provide an additional optimization: since the learned index is imperfect (and thus provides a range of values in the permutation vector instead of the exact value), multiple entries of the underlying unsorted data need to be checked.

[1] https://github.com/learnedsystems/LearnedSecondaryIndex

aiDM'22 , June 17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9377-5/22/06…$15.00
https://doi.org/10.1145/3533702.3534912

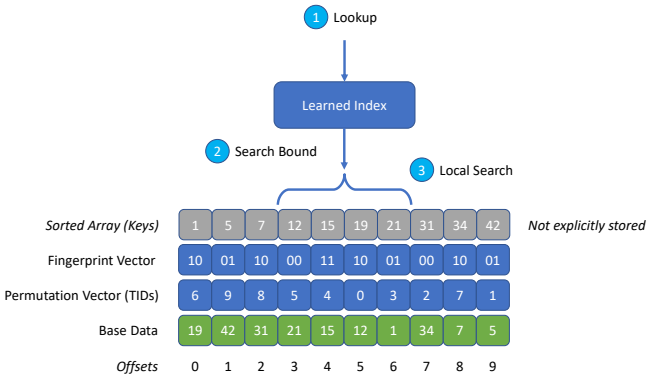

Figure 1: Learned Secondary Index and its lookup procedure. The local search uses the permutation vector to locate corresponding entries in the base data. The fingerprint vector prunes unnecessary accesses.

To reduce these "false positives," LSI additionally stores hash fingerprints (typically a few bits) for each entry in the permutation vector. These fingerprints introduce an interesting trade off: If we want to answer a search using fingerprint bits, we need to resort to a linear scan over the predicted (error-bounded) range, while we could use binary search otherwise. And, obviously, fingerprint bits only help with equality lookups (where the lookup key is part of the data) and do not accelerate lower-bound lookups in general.

We show that LSI can compete with established secondary indexes such as ART [16] in terms of lookup performance while consuming up to 6× less space.

Related Work. There is limited work on using learning for secondary indexing. HERMIT [27] learns a mapping between a primary and correlated secondary columns using linear functions. That way it can reuse a primary index for indexing secondary columns. Cortex [21] follows a similar approach but can also capture more complex correlations.

2 LEARNED SECONDARY INDEX

LSI consists of three parts: a permutation vector, a learned index, and a fingerprint vector, which we will describe in the following (see Figure 1).

Permutation Vector. The permutation vector is a compressed representation of the sorted order of the underlying unsorted base data d. Specifically, the permutation vector p is set so that every entry $p[i]$ contains the index into d corresponding to the ith smallest key. In other words, if one wished to access the 6th smallest key, one would access $d[p[6]]$. The permutation vector allows us to build a learned index over the sorted keys, and then map predictions from that learned index into the underlying unsorted base data. We bitpack these permutation vectors, allowing each entry to be stored using approximately $log_2 n$ bits, making the size of the permutation vector $O(n \log n)$. Additional techniques, like Lehmer codes [15], could be used to further compress this vector, at the cost of higher decompression time.

Learned Index. The learned (or approximate) index maps a lookup key to a bounded search range. One could think of the inner nodes of a B+-tree as such an approximate index. A lookup in a B+-tree's inner node structure will identify a leaf node, which depending on the fanout contains k entries. In other words, a lookup will bound the last-level search to k entries. In our implementation, we use the learned index PLEX [25], selected for its simplicity. PLEX builds a spline model over the cumulative distribution function (CDF) of the data and bounds the of the last-level search to a user-defined range (defined by the model's maximum error). Note that LSI does *not* store an explicit representation of the sorted key array. This is in contrast to a B+-tree, which stores actual keys in its leaf nodes. Instead, LSI uses the permutation vector to map PLEX's predictions into the underlying data. While this approach can save significant space, the downside is that LSI produces false positives, since PLEX only produces approximate ranges. We use the permutation vector to perform binary search within the approximate range. Note that the cost of this binary search is higher than for sorted data, as all memory accesses to the unsorted base data are likely out of cache.

Fingerprint Vector. For equality lookups, the approximate range returned by the learned index needs to be entirely searched. As a result, many non-relevant keys may be scanned. To mitigate this, we create a *fingerprint vector*, which stores fixed-sized hash fingerprints (e.g., 8 bits) for each key. We use the Murmur3 hash function to generate hash values and extract the first x bits as hash fingerprints. When performing an equality lookup, we first ensure that the fingerprint of the lookup key matches the fingerprint stored in the fingerprint vector – if it does, we then access the permutation vector to get the index into the underlying data, and then access the underlying data. However, if the fingerprints do not match, we skip accessing both the permutation vector and the underlying data, potentially saving cache misses.

2.1 Building LSI

Building LSI involves multiple steps. First, we create a sorted copy of the base data. Then we build a cumulative distribution function (CDF) on the sorted data, which maps each key to its position in the sorted array. Note that the data may contain duplicates, which will result in a "steeper" slope in the CDF that can in turn be more difficult to approximate. An alternative would be to remove duplicates upfront and maintain a rank structure to map from the duplicate-free representation to the base data. However, we found that to not be worthwhile, both in terms of space and lookup performance. Once we have created the CDF, we build an error-bounded learned index (PLEX) over it. Next, we create a bit-packed permutation vector that maps from the sorted data (over which we have built the index) to the unsorted base data. Finally, we build an array containing fingerprints. The sorted copy of the data is then discarded, as it is no longer needed. Overall, LSI has two parameters, the maximum error of its learned index and the number of fingerprint bits.

2.2 Lookup Procedure

Lookups proceed as follows (see Figure 1). First, we query the learned index model with the lookup key which returns a range that is guaranteed to contain the lookup key in a sorted version of the array. Next, we perform a local search within the search

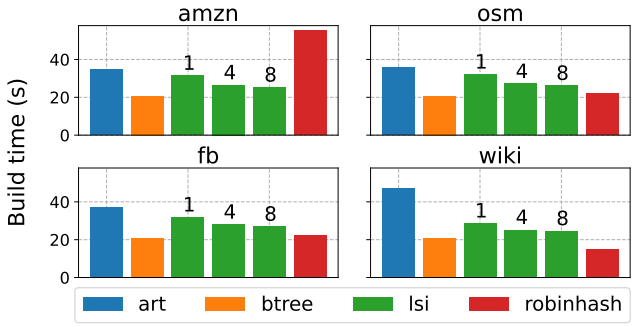

Figure 2: Build time in seconds. The text annotations denote the error bounds.

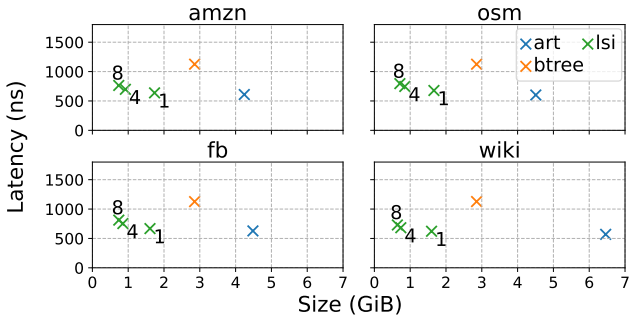

Figure 3: Lower-bound lookups using non-existing keys. The text annotations denote the error bounds.

range. If it is an equality lookup, we perform a linear search on the fingerprint vector and, if a fingerprint matches, perform a random access to the base data using the permutation vector. Once we have found a matching key in the base data, we keep scanning the fingerprint vector for matching fingerprints (as there may be duplicates in the base data). However, after having found a first qualifying key in the base data, we can stop scanning the fingerprint vector once we found a non-matching fingerprint. If the lookup is a lower-bound lookup, we cannot use the fingerprint vector and use binary search within the search range.

3 EVALUATION

We evaluate Learned Secondary Index (LSI) using a variation of the SOSD benchmark [12, 18] on a `c5.9xlarge` AWS machine with 36 vCPUs and 72 GiB of RAM. We perform single-threaded lower-bound and equality lookups on four real-world datasets from SOSD. To prevent out-of-order execution, we use a memory barrier in between individual lookups. Each dataset consists of 200 M 64-bit unsigned integer keys: `amzn` (book popularity data), `face` (randomly sampled Facebook user IDs), `osm` (cell IDs from Open Street Map), and `wiki` (timestamps of edits from Wikipedia). We generate random 8-byte payloads for each key. We compare LSI with several baselines: the STX B-Tree (BTree) [2], the Adaptive Radix Tree (ART) [16], and a robin-hood hash table (RobinHash) [1].

Build Times. While BTree and ART can index unsorted keys out of the box, LSI's learned index (PLEX) needs a sorted copy of the data to build a CDF and train its model on the CDF. However, inserting keys in random order into the BTree is about 8× slower than first sorting the keys and then using its bulk loading functionality. We hence bulk load the BTree, which also yields denser nodes. Likewise, ART achieves a speedup of almost 2× when inserting keys in sorted order, despite not having a separate bulk loading interface. Figure 2 shows the total build times, which includes the time for sorting the data (except for RobinHash where sorting does not have an effect). BTree achieves the lowest build times, followed by RobinHash and LSI with an error bound of 8. As one would expect, decreasing the error bound increases the build time since the learned index model requires more spline points to satisfy the error bound. ART does not offer bulk loading functionality and has the highest build times, except for `amzn` where RobinHash requires the most time to build.

Lower-Bound Lookups. We first study the performance of lower-bound lookups, i.e., returning the value of the first key that is not less than the lookup key. We remove a random subset of keys (10%) from the dataset and use them as lookup keys. For all datasets, except `wiki` which contains duplicates, lookups will be with keys that do not exist in the data. Hence, we cannot use fingerprint hashes and also cannot compare against hash tables. Figure 3 shows the results. LSI achieves the best trade off between size and lookup latency. It matches ART's lookup latency while consuming up to 6× less space. Note that both BTree and ART support updates while LSI does not. Also, compressing the leaf layer of BTree in a similar fashion (using bit-packed TIDs instead of 8-byte payloads) would yield space savings but would not necessarily improve performance. TIDs in ART cannot be compressed as easily as they are inlined into 8-byte child pointers.

Equality Lookups. We now look at the special case of equality lookups. In this experiment, we enable hash fingerprints (8 bits) in LSI and also compare against RobinHash. Figure 4 shows the results on the `amzn` dataset. RobinHash achieves a latency of around 440 ns per lookup which is faster than LSI's 660 ns but also consumes 4× the amount of space. Using hash fingerprint bits requires LSI to perform a linear scan through the search range. We now perform a micro experiment to study the trade off between using fingerprints (with linear search) and binary search. We train LSI with four different error (4, 16, 64, and 256) and six different fingerprint configurations (0, 1, 2, 4, 8, and 16 bits). If there are fingerprint bits, we use linear search and use binary search otherwise. As shown in Figure 5, the variants using 4 and 16 fingerprint bits are faster than binary search for certain error configurations.

Space Breakdown. Table 1 shows the space breakdown of LSI with errors of 4 (LSI4) and 8 (LSI8) compared to ART, BTree, and RobinHash for the `amzn` dataset. LSI spends 64% and 80% of its space on the permutation vector with an error bound of 4 and 8, respectively. This raises the questions whether we can compress the permutation vector.

Compressing the Permutation Vector. The information-theoretic lower bound for storing a permutation of n elements is $O(log_2(n!))$ bits. Compared to our current bit-packed representation which requires $O(n * log_2(n))$ bits, this saves at most 1.44 bits per key (see Figure 6). Using a Beneš network [4], we can get arbitrarily close

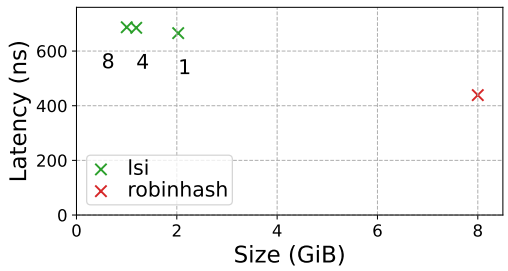

Figure 4: Equality lookups on the amzn dataset comparing LSI to RobinHash.

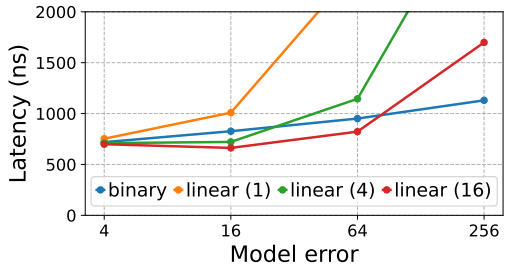

Figure 5: Binary search vs. linear search with varying fingerprint sizes (in brackets).

Table 1: Space breakdown of LSI for the amzn dataset.

Index	Overall Size (MiB)	Model (MiB)	Permutation (MiB)
LSI4	943	342	601
LSI8	754	153	601
ART	4,343	-	-
BTree	2,923	-	-
RobinHash	8,192	-	-

to the information-theoretic lower bound, however, we would sacrifice lookup time due to the compressed representation. An access to the compressed bitvector would cost $O(log_2(n))$ time instead of the $O(1)$ random access if we merely bitpack. Hence, in the best case, we can reduce the size of the permutation vector in Table 1 from 601 MiB to roughly 557 MiB.

Using a Different Approximate Index. The approximate index layer of LSI requires an index structure that given a lookup key returns an error-bounded range. Besides other error-bounded learned indexes such as PGM [9], one could also use a Recursive Model Index (RMI) [14] and remember the maximum model error. Another option is the recently proposed Compact Hist-Tree (CHT) [5]. CHT is a compact, read-only radix tree with a fixed fanout and also returns an error-bounded range. Figure 7 shows the results when replacing PLEX in LSI with CHT on the amzn dataset. In summary, LSI achieves a better space-performance trade off when using PLEX as learned index.

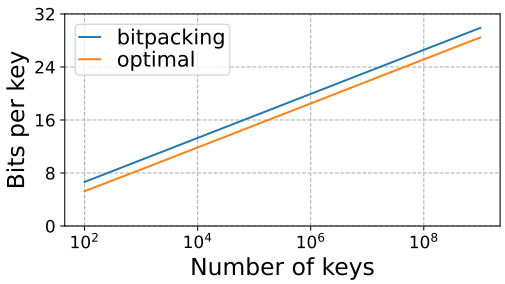

Figure 6: Size of the permutation vector. Information-theoretic lower bound vs. our bit-packed representation.

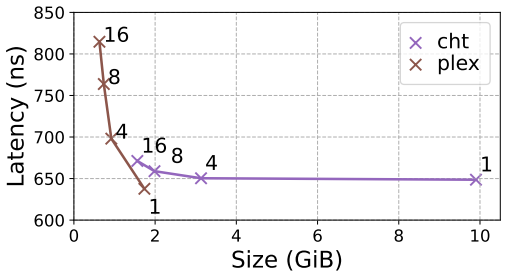

Figure 7: Using PLEX vs. CHT as models in LSI for lower-bound lookups. The text annotations denote the error bounds.

4 CONCLUSIONS

We have introduced LSI, a new learned data structure that can index unsorted data. LSI is a first step towards learned secondary indexing. We have shown that our approach can compete with state-of-the-art secondary indexes while being more space efficient. In future work, we plan to extend LSI into multiple directions. First, we want to explore indexing data blocks instead of individual tuples which will lower the overhead of the permutation vector for low and medium cardinality columns. Second, we plan to integrate model error correction techniques [10] to narrow the search range and hence reduce the number of false positives. Finally, we want to explore applications to disk-based systems.

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. Dominik Horn and Pascal Pfeil were supported by a fellowship within the IFI programme of the German Academic Exchange Service (DAAD).

REFERENCES

[1] RobinMap, https://github.com/Tessil/robin-map.

[2] STX B+ Tree, https://panthema.net/2007/stx-btree/.

[3] H. Abu-Libdeh, D. Altinbüken, A. Beutel, E. H. Chi, L. Doshi, T. Kraska, X. Li, A. Ly, and C. Olston. Learned indexes for a google-scale disk-based database. *CoRR*, abs/2012.12501, 2020.

[4] B. Cohen and D. Boneh. How to Store a Permutation Compactly, https://hackmd.io/@dabo/rkP8Pcf9t.

[5] 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.

[6] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. From WiscKey to Bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 155–171. USENIX Association, 2020.

[7] 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.

[8] P. Ferragina and G. Vinciguerra. Learned data structures. In *Recent Trends in Learning From Data*, volume 896 of *Studies in Computational Intelligence*. Springer, 2020.

[9] 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.

[10] A. Hadian and T. Heinis. Shift-Table: A low-latency learned index for range queries using model correction. In Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, and F. Guerra, editors, *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 253–264. OpenProceedings.org, 2021.

[11] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern cpus and gpus. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 339–350. ACM, 2010.

[12] 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.

[13] 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.

[14] 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.

[15] D. H. Lehmer. Teaching combinatorial tricks to a computer. In *Proc. Sympos. Appl. Math. Combinatorial Analysis*, volume 10, pages 179–193, 1960.

[16] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful indexing for main-memory databases. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

[17] M. Maltry and J. Dittrich. A critical analysis of recursive model indexes. *CoRR*, abs/2106.16166, 2021.

[18] 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.

[19] R. Marcus, E. Zhang, and T. Kraska. CDFShop: Exploring and optimizing learned index structures. 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 2789–2792. ACM, 2020.

[20] M. Mishra and R. Singhal. RUSLI: real-time updatable spline learned index. In R. Bordawekar, Y. Amsterdamer, O. Shmueli, and N. Tatbul, editors, *aiDM '21: Fourth Workshop in Exploiting AI Techniques for Data Management, Virtual Event, China, 25 June, 2021*, pages 1–8. ACM, 2021.

[21] V. Nathan, J. Ding, T. Kraska, and M. Alizadeh. Cortex: Harnessing correlations to boost query performance. *CoRR*, abs/2012.06683, 2020.

[22] V. Pandey, A. van Renen, A. Kipf, J. Ding, I. Sabek, and A. Kemper. The case for learned spatial indexes. *2nd International Workshop on Applied AI for Database Systems and Applications*, 2020.

[23] N. F. Setiawan, B. I. P. Rubinstein, and R. Borovica-Gajic. Function interpolation for learned index structures. In R. Borovica-Gajic, J. Qi, and W. Wang, editors, *Databases Theory and Applications - 31st Australasian Database Conference, ADC 2020, Melbourne, VIC, Australia, February 3-7, 2020, Proceedings*, volume 12008 of *Lecture Notes in Computer Science*, pages 68–80. Springer, 2020.

[24] B. Spector, A. Kipf, K. Vaidya, C. Wang, U. F. Minhas, and T. Kraska. Bounding the last mile: Efficient learned string indexing. *3rd International Workshop on Applied AI for Database Systems and Applications*, 2021.

[25] M. Stoian, A. Kipf, R. Marcus, and T. Kraska. PLEX: towards practical learned indexing. *3rd International Workshop on Applied AI for Database Systems and Applications*, 2021.

[26] J. Wu, Y. Zhang, S. Chen, Y. Chen, J. Wang, and C. Xing. Updatable learned index with precise positions. *Proc. VLDB Endow.*, 14(8):1276–1288, 2021.

[27] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing succinct secondary indexing mechanism by exploiting column correlations. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1223–1240. ACM, 2019.

[28] E. T. Zacharatou, A. Kipf, I. Sabek, V. Pandey, H. Doraiswamy, and V. Markl. The case for distance-bounded spatial approximations. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.