RadixSpline: A Single-Pass Learned Index

Andreas Kipf[★] Ryan Marcus[★†] Alexander van Renen Mihail Stoian
Alfons Kemper Tim Kraska[★] Thomas Neumann
TUM MIT CSAIL[★] Intel Labs[†]
{renen, stoian, kemper, neumann}@in.tum.de {kipf, ryanmarcus, kraska}@mit.edu

ABSTRACT

Recent research has shown that learned models can outperform state-of-the-art index structures in size and lookup performance. While this is a very promising result, existing learned structures are often cumbersome to implement and are slow to build. In fact, most approaches that we are aware of require multiple training passes over the data.

We introduce RadixSpline (RS), a learned index that can be built in a single pass over the data and is competitive with state-of-the-art learned index models, like RMI, in size and lookup performance. We evaluate RS using the SOSD benchmark and show that it achieves competitive results on all datasets, despite the fact that it only has two parameters.

1 INTRODUCTION

In [13], Kraska et al. proposed learned index structures, a new type of index for sorted data which use learned models to predict the position of a lookup key. These learned index structures can be realized via supervised learning techniques, using the cumulative distribution function (CDF) of the underlying data for training. More recently, the SOSD benchmark [11] demonstrated that learned index structures (which can be viewed as CDF approximators) can compete favorably with state-of-the-art index structures [2, 4, 8, 9, 14, 32] in terms of size and lookup performance.

However, some learned index structures, such as RMIs [13], do not support inserts and cannot be constructed in a single pass over the data, which severely limits their applications. The recent learned index proposals ALEX [3] and PGM [5] add support for inserts. In this work, we argue that there are applications where indexes do not need to support individual updates and where it is sufficient to be able to build them efficiently. The most prominent example are LSM-trees [23].

In an LSM-tree, data is stored in several files, each of which is sorted by a key column. Each file generally stores additional metadata, such as a Bloom filter or an index. This metadata can be used at query time to either exclude a file from a lookup (via the Bloom filters or lower/upper bound) or to quickly locate the relevant tuples in a file (via the index). These files are organized into multiple levels, with higher levels containing exponentially more files than lower levels. New data is inserted into files in lower levels, which are periodically merged with files in higher levels. Unfamiliar readers may wish to see [15] for an in-depth survey of LSM-trees, but for this work one only needs to note that this merge process between two files is the perfect time to re-build a learned index. The merge produces data in sorted order, which can be passed through a *single-pass* training algorithm before it is written back to disk. Since the merge operation is expensive on its own and is usually done asynchronously, training such a one-pass learned index could only incur a negligible constant overhead. However, existing learned indexes do not allow for an efficient build.

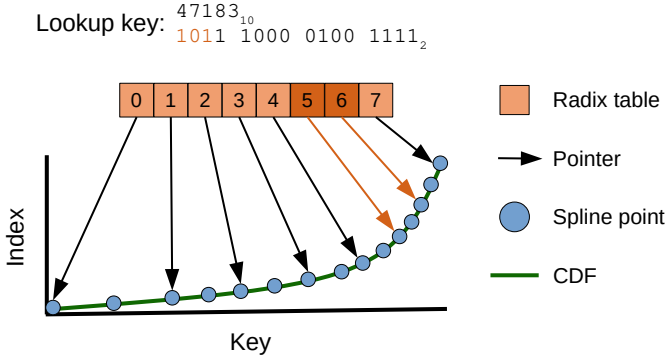

Figure 1: A radix spline index and example lookup process. The r (here, 3) most significant bits b of the lookup key are used as an index into the radix table. Then, a binary search is performed on the spline points between the bth pointer and the $b+1$th pointer.

Andreas Kipf, Ryan Marcus, and Alexander van Renen contributed equally.

aiDM'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8029-4/20/06…$15.00
https://doi.org/10.1145/3401071.3401659

In this work, we introduce RadixSpline (RS), a learned index that can be built in a single pass over sorted data. Notably, the proposals of FITing-Tree [6] and PGM [5] also support single-pass builds. However, for both of these indexes the amount of work per element is logarithmic in the number of levels (similar to inserts in a BTree). With RS, we propose the first single-pass learned index with a *constant* amount of work per new element.

Being an ordered index, RS supports both equality and range predicates (e.g., lower bound lookups). RS is built in two steps. First, a linear spline is fit to the CDF of the data that guarantees a certain error bound. This results in a set of spline points which can be significantly smaller than the underlying data. Second, we build a radix table (a flat radix structure) that serves as an approximate index into the spline points. Similar to the `Node256` in the Adaptive Radix Tree (ART) [14], we extract a certain radix prefix (e.g., the first 20 bits, neglecting common prefix bits shared by all keys) and use those as an offset into the radix table. Both steps can be performed in a single pass over the sorted data.

RS is not only efficient to build, but also competitive with state-of-the-art RMI models in size and lookup performance. Index size is an especially important factor for LSM-tree applications because indexes are kept in main memory (whereas each large sorted file is stored on disk). Furthermore, RS's implementation only consists of roughly one hundred lines of C++ code and does not have any external dependencies. Finally, RS only takes two hyper parameters (spline error and radix table size). Both have an intuitive and reliable impact on size and lookup latency. As a result, tuning RS is easier than tuning more complex learned indexes structures with many hyperparameters [20]. One caveat is that RS can be impacted by heavy skew, rendering the radix table largely ineffective. In such cases, one could fall back to a tree-structured radix table or handle outliers separately. However, we are yet to encounter such extreme skew in real-world data. In summary, we believe that RS is a practical learned index structure with potentially high impact in write-once/read-many settings such as LSM-trees.

More broadly, this work follows recent trends in integrating machine learning components into systems [7], especially database systems [10, 12, 16–19, 21, 24, 26–31].

2 RADIXSPLINE

A RadixSpline (RS) index is designed to map a *lookup key* to an *index* (the position of the key in the underlying data). Like an RMI [13], radix spline indexes require the underlying data to be sorted on the lookup key in a flat array.

An RS index consists of two components: a set of spline points and a radix table. The set of spline points is a subset of the keys, selected so that spline interpolation for any lookup key will result in a predicted lookup location within a preset error bound. For example, if the preset error bound is 32, then the location of any lookup key can be no more than 32 positions away from the location predicted by the RS index. The radix table helps to quickly locate the correct spline points for a given lookup key. Intuitively, the radix table limits the range of possible spline points to search over for every possible b-length prefix of a lookup key.

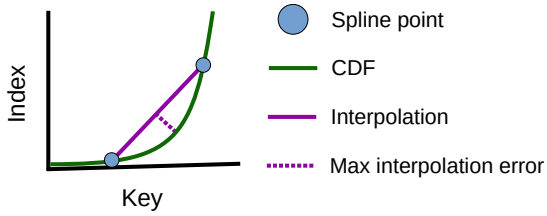

Figure 2: A single spline segment. We select enough spline points so that the maximum interpolation error (dashed line) stays within a specified bound.

At lookup time, the radix table is used to determine a range of spline points to examine. These spline points are searched until the two spline points surrounding the key are found. Then, linear interpolation is used to predict the location (index) of the lookup key in the underlying data. Because the spline interpolation is error-bounded, only a (small) range of the underlying data needs to be searched.

In contrast to other learned indexes [3, 13], RS can be built in a single pass over the sorted data. While the use of splines and a bottom-up approach has been explored before in FITing-Tree [6] and others [5, 25], in this work, we combine the ideas from [6] with radix trees, making it highly competitive with top-down built indexes [13].

2.1 Construction

RS indexes, like PGM indexes [5] or FITing-Trees [6], are built "bottom-up". First, we construct an error-bounded spline on top of the the underlying data. Then, the selected spline points themselves are indexed in a radix table.

Build Spline. As observed in [13], all index structures can be thought of as models that map lookup keys to positions. Let the dataset to index D be an indexed set of tuples, with $D_i = (k_i, p_i)$ where D_i represents the ith datapoint, k_i represents the key of the ith datapoint, and p_i represents the position (offset) of the ith datapoint. A radix spline index first builds a spline model S, such that $S(k_i) = p_i \pm e$, where e is a specified constant. In other words, the spline model S *always* predicts the correct location of the data within a constant error of e.

This error-bounded model is realized via spline interpolation (we use `GreedySplineCorridor` [22]). The parameters of the model S, $Knots(S)$, are a set of spline points, or

knots, which are a representative set of datapoints (see Figure 1). These data points are chosen such that, for any lookup key x, linearly interpolating between the two closest spline points in $Knots(S)$ will produce an estimate with error no larger than e (cf. Figure 2). Formally, to evaluate $S(x)$, letting $(k_{left}, p_{left}) \in Knots(S)$ be the knot with the greatest key such that $k_{left} \leq x$ and letting $(k_{right}, p_{right}) \in Knots(S)$ be the knot with the smallest key such that $k_{right} > x$, we compute

$$S(x) = p_{left} + (x - k_{left}) \times \frac{p_{right} - p_{left}}{k_{right} - k_{left}}.$$

For more details on the error-bounded spline algorithm, we refer the reader to [22].

Build Radix Table. Next, we build a radix table on top of the selected spline points to quickly find the two spline points surrounding the lookup key. The radix table is a flat `uint32_t` array that maps fixed-length key prefixes ("radix bits") to the first spline point with that prefix. The key prefixes are the offsets into the radix table while the spline points are represented as `uint32_t` values stored in the radix table (cf. pointers in Figure 1).

The radix table takes the number of radix bits r as a parameter. For example, for $r = 18$ we allocate an array with 2^{18} many entries (1 MiB in size). A larger r grows the size of the table exponentially (2^r) but may also increase its precision. That is, we may need to search a more narrow range of spline points to find the two spline points surrounding the lookup key. In Section 3, we show the impact of this parameter on size and lookup performance.

The build process itself is very straightforward and extremely fast: we first allocate an array of the appropriate size (2^r many entries), then we go through all spline points and whenever we encounter a new r-bit prefix b, we insert the offset of the spline point (a `uint32_t` value) into the slot at offset b in the radix table. Since the spline points are ordered, the radix table is filled in consecutive order from left to right. As an optimization, we eliminate common prefix bits shared by all keys when building the radix table.

Single Pass. Building the CDF, the spline, and the radix table can all be performed on-the-fly, in a single pass over the sorted datapoints. When encountering a new CDF point (i.e., when the key changes), we pass that point to the spline construction algorithm [22]. Filling the pre-allocated radix table within the same pass is also straightforward: whenever we encounter a new r-bit prefix in a selected spline point, we make a new entry to the table.

Lookups. Using the example in Figure 1, the lookup logic is as follows: We first extract an r-bit prefix b of the lookup key (101 in this case). Then, we use the extracted bits b to make an offset access into the radix table retrieving the two pointers stored at positions b and $b+1$ (here, positions 5 and 6). These pointers (marked in orange) define a narrowed search range on the spline points. Next, we search this range for the two spline points surrounding the lookup key using binary search. Subsequently, we perform a linear interpolation between these two spline points to obtain an estimated position p of the key. Finally, we perform a binary search within the error bounds ($p \pm e$) to find the first occurrence of the key.

3 EVALUATION

We evaluate RadixSpline (RS) using the SOSD benchmark [11] on a `c5.4xlarge` AWS machine. We use six 64-bit datasets, each of them containing 200 M key/value pairs (3.2 GiB) in size: `amzn` (book popularity data), `face` (Facebook user IDs), `logn` (synthetic lognormal distributed data), `osmc` (composite cell IDs from Open Street Map) and `wiki` (timestamps of Wikipedia edits). For details on these datasets, see [11].

Like [11], we build indexes on top of sorted arrays. An index takes in a key and produces a *search range* in the underlying data. This range must contain the lookup key if the lookup key exists, and must otherwise contain the first key not larger than the lookup key (lower bound search). Then, binary search is used to locate the exact key within the search range. Indexes are evaluated based on their end-to-end performance: the time to produce a search range plus the time to execute the binary search. We perform 10M lookups (1 thread) on a given dataset and report the average lookup latency. Lookup keys are uniformly chosen from the keys.

We compare RS against three traditional, non-learned approaches: ART [14], STX B+-tree (BTree) [1], and binary search (BS). For ART and BTree, we use a stride of 32 (meaning that every 32nd key is inserted into the index – this provides better space and performance compared to indexing each key). We also compare against the public implementation [20] of the recursive model index (RMI) [13], a learned approach that is built "top-down" (i.e., starting with a loose fit and then progressively learning finer-grained models) and internally uses a range of models (e.g., linear, cubic, or even BTrees). Hash-based methods are excluded because hash-based methods do not support lower bound searches. Since ART does not support duplicate keys, it does not have results for `wiki` (the only dataset with duplicates).

Build Times. (Figure 3, left). Due to its single-pass build process, RS is almost as efficient to build as ART or BTree and is significantly faster than RMI, which performs multiple training passes over the sorted datapoints.

Lookup Latency. (Figure 3, middle). Binary search (BS) takes around 850ns per lookup across all datasets. BTree improves upon BS by using the cache more efficiently, and requires only around 600ns. Like BS, it is largely independent

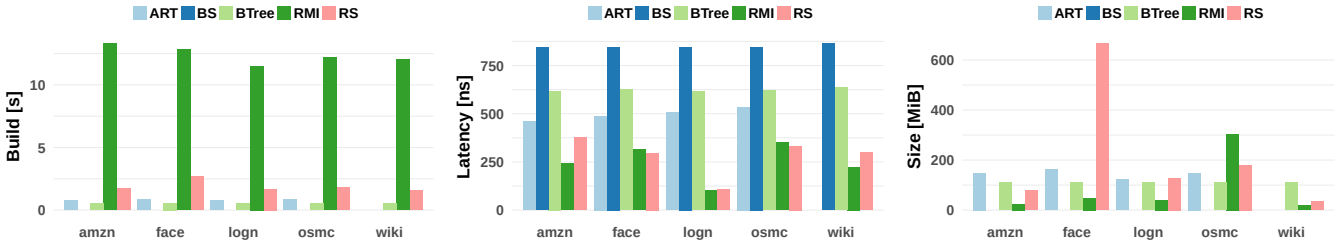

Figure 3: Build time (left), lookup latency (middle), and size (right) for lookup-optimized index configurations.

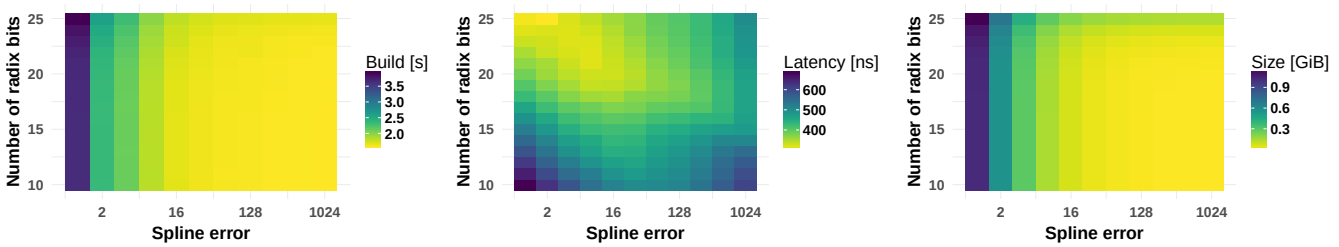

Figure 4: Build time (left), lookup latency (middle), and index size (right) for different RadixSpline configurations.

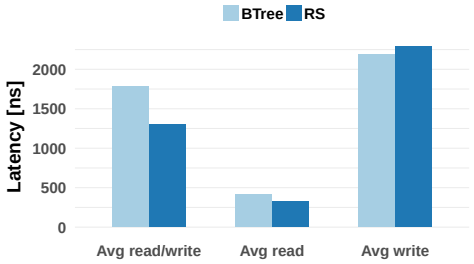

Figure 5: Average operator cost, LSM-tree.

of the data distribution. Both learned approaches, RMI and RS, are significantly faster than the traditional indexes but also more affected by the data distribution. Note that we have tuned both approaches for minimum lookup latency.

Index Size. (Figure 3, right). Except for BS and a few outliers, all indexes consume around 100 MiB which corresponds to 6.6% of the uncompressed key size (200 M 64-bit keys). For the `face` dataset, RS uncharacteristically requires more than 600 MiB (39%) to achieve its best performance. Next, we investigate different RS configurations for this dataset.

Configuration Space. The best RS configuration (Figure 3) for `face` uses a lot of memory (≈ 650 MiB). However, we can easily trade lookup performance for memory as shown in Figure 4: Both build time (left) and size (right) mostly depend on the spline error. Starting with an error of 16, RS can be built within 2s for this dataset and requires less than 200 MiB. For example, with a spline error of 16 (instead of 2) and 20 instead of 25 radix bits, RS trades performance (-11.5%) for a significant space reduction (-99.9%).

LSM Performance. To validate the applicability of RS to LSMs, we performed a preliminary experiment where we substitute the BTree index with a RadixSpline in RocksDB. We use the `osmc` dataset and executed 400 M operations, 50% reads and 50% writes (cf. Figure 5). When using RS, the average write time increased by ≈ 4%, but the average read time decreased by over 20%. The total execution time fell to 521 seconds from 712 seconds with a BTree. In addition, the RS variant used ≈ 45% less memory, potentially creating space for larger Bloom filters or increased caching. While obviously preliminary, this experiment indicates potential benefits of RS in LSMs.

4 CONCLUSIONS

We have described a new learned index, called RadixSpline, that can be built in a single pass over sorted data. Notably, RS only takes two hyper parameters and thus is rather easy to tune to a given dataset and memory budget. Our experiments with real-world data have shown that RS is competitive with a state-of-the-art learned index in size and lookup performance while being as efficient to build as traditional indexes. While our radix table has a constant size, it may become less useful as dataset size grows, or under large outliers.

In future work, we also plan to investigate how RS can be automatically tuned with minimal user-interaction, balancing memory footprint and performance. Such an auto-tuning could be informed by metrics extracted from the data. RS currently does not take advantage of any multi-threading, another potential direction for performance improvements.

Acknowledgments. This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, NSF IIS 1900933, DARPA Award 16-43-D3M-FP040, and the MIT Air Force Artificial Intelligence Innovation Accelerator (AIIA).

REFERENCES

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

[2] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 521–534, New York, NY, USA, 2018. Association for Computing Machinery.

[3] J. Ding, U. F. Minhas, H. Zhang, Y. Li, C. Wang, B. Chandramouli, J. Gehrke, D. Kossmann, and D. Lomet. ALEX: An Updatable Adaptive Learned Index. *arXiv:1905.08898 [cs]*, May 2019.

[4] P. Fent, M. Jungmair, A. Kipf, and T. Neumann. START âĂŤ Self-Tuning Adaptive Radix Tree. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pages 147–153, 2020.

[5] P. Ferragina and G. Vinciguerra. The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, Apr. 2020.

[6] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. FITing-Tree: A Data-aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1189–1206, New York, NY, USA, 2019. ACM.

[7] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, and T. Mattson. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 69–80, Philadelphia, PA, USA, June 2018. Association for Computing Machinery.

[8] G. Graefe. B-tree indexes, interpolation search, and skew. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware*, DaMoN '06, Chicago, Illinois, June 2006. Association for Computing Machinery.

[9] 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 *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, 2010.

[10] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[11] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. SOSD: A Benchmark for Learned Indexes. In *ML for Systems at NeurIPS*, MLForSystems @ NeurIPS '19, Dec. 2019.

[12] A. Kipf, D. Vorona, J. Müller, T. Kipf, B. Radke, V. Leis, P. Boncz, T. Neumann, and A. Kemper. Estimating Cardinalities with Deep Sketches. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1937–1940, Amsterdam, Netherlands, June 2019. Association for Computing Machinery.

[13] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. ACM.

[14] 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 '13, pages 38–49, USA, 2013. IEEE Computer Society.

[15] C. Luo and M. J. Carey. LSM-based storage techniques: A survey. *PVLDB*, 29(1):393–418, Jan. 2020.

[16] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Learning to Steer Query Optimizers. *arXiv:2004.03814 [cs]*, Apr. 2020.

[17] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[18] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM@SIGMOD '18, Houston, TX, 2018.

[19] R. Marcus and O. Papaemmanouil. Towards a Hands-Free Query Optimizer through Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[20] 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, Portland, OR, June 2020.

[21] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *Workshop on Self-Managing Databases*, SMDB @ ICDE '20, 2020.

[22] T. Neumann and S. Michel. Smooth interpolating histograms with error guarantees. In *Sharing Data, Information and Knowledge, 25th British National Conference on Databases*, BNCOD '08, pages 126–138, 2008.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[24] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *2nd Workshop on Data Managmeent for End-to-End Machine Learning*, DEEM '18, 2018.

[25] N. Setiawan, B. Rubinstein, and R. Borovica-Gajic. Function Interpolation for Learned Index Structures. In *Database Theory and Applications*, DTA '20, 2020.

[26] Shrainik Jain, Jiaqi Yan, Thiery Cruanes, and Bill Howe. Database-Agnostic Workload Management. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[27] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment*, 13(3):307–319, Nov. 2019.

[28] I. Trummer, S. Moseley, D. Maram, S. Jo, and J. Antonakakis. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *PVLDB*, 11(12):2074–2077, 2018.

[29] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. ACM.

[30] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '19, pages 1–8, Amsterdam, Netherlands, July 2019. Association for Computing Machinery.

[31] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment*, 13(3):279–292, Nov. 2019.

[32] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 323–336, Houston, TX, USA, May 2018. Association for Computing Machinery.