Automatically Finding Optimal Index Structure

Extended Abstracts

Supawit Chockchowwat, Wenjie Liu, Yongjoo Park
{supawit2,wenjie3,yongjoo}@illinois.edu
CreateLab @UIUC, USA

ABSTRACT

Existing *learned indexes* (e.g., RMI, ALEX, PGM) optimize the *internal regressor* of each node, *not the overall structure* such as index height, the size of each layer, etc. In this paper, we share our recent findings that **we can achieve significantly faster lookup speed by optimizing the structure as well as internal regressors**. Specifically, our approach (called AutoIndex) expresses *the end-to-end lookup time as a novel objective function*, and searches for optimal design decisions using a purpose-built optimizer. In our experiments with state-of-the-art methods, AutoIndex achieves 3.3×–7.7× faster lookup for the data stored on local SSD, and 1.4×–3.0× faster lookup for the data on Azure Cloud Storage.

1 INTRODUCTION

Indexes enable fast lookup and are employed by many data systems for fast search and analytics: search engines [19, 40], key-value stores [5, 25, 37], RDBMS [4, 29], etc. Conventional indexes (e.g., B-tress [6, 39, 44], red-black trees [9, 11, 20], skip lists [46]) offer $O(\log n)$ lookup speed for n data items. Learned indexes [15, 17, 18, 22, 24, 41, 43] can offer even lower latencies based on more compact representations [16]. To achieve high performance, indexes are typically optimized specifically for target storage media (e.g., SSD [27], NVMe [42], memory [12], L1 cache [38]). For example, a B-tree node fits in a disk page (e.g., 4KB). RMI [24] has three layers with the fast random-access assumption.

Problem. *The performance of these indexes is suboptimal* when they operate in an environment different from what they are designed for. We have observed this suboptimality (thus, missed opportunity) as we develop a cloud-optimized document store [13]. Also, see § 2 for a concrete example with B-tree indexes.

Opportunity & Challenge. We observe that we can greatly improve lookup speed by carefully choosing structural parameters such as the number of layers, layer sizes, the types and accuracy of internal regressors, etc. Our intuition is as follows. As depicted in Fig. 1, we should prefer shallow and wide indexes if a storage device has very long latency (round-trip time), because we want to reduce the number of I/O operations. In contrast, if the latency is very small in comparison to bandwidth, tall indexes should be preferred. However, finding an optimal design is non-trivial in practice because there are exponentially many candidates (§ 3.2).

 4th International Workshop on Applied AI for Database Systems and Applications (AIDB '22). September 5th, 2022, Sydney, Australia.

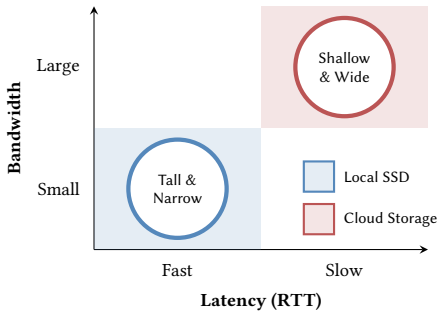


Figure 1: Expected optimal structures by I/O characteristics. Our approach, AutoIndex, can automatically find optimal index structures as well as internal regressors.

Our Approach. We propose a new index builder (called AutoIndex) that *can learn the optimal structure in a principled manner by navigating in a high-dimensional design space.* Specifically, we formulate a novel optimization problem that expresses an end-to-end lookup time in terms of core design parameters (e.g., the number of layers, the size of each node, regressors, etc.) as well as I/O performance. By solving the optimization problem using our highly parallel search method, we can build a low-latency index for large-scale data.

Note that our approach is *orthogonal to learned indexes* [15, 17, 18, 22, 24, 41, 43], which propose the use of regressors for branching functions (instead of exact pointers). In contrast, *this work focuses on optimizing the overall index structure.* This work is in line with the recent advances at the intersection of data systems and machine learning [7, 8, 23, 26, 28, 31–36, 45].

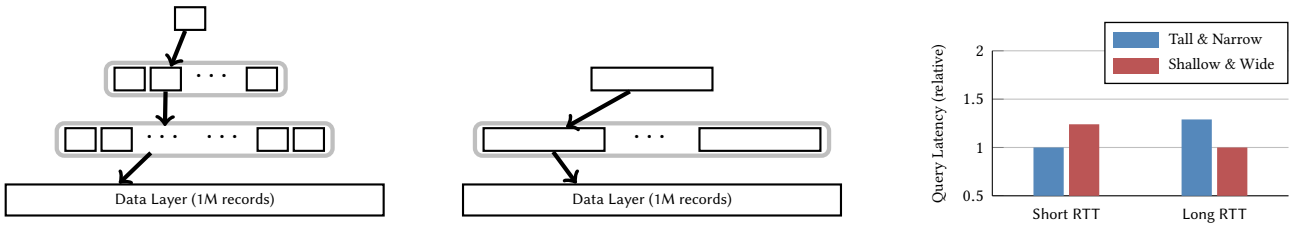
2 MOTIVATION & SCOPE

2.1 Motivation

Our intuition is the following: the structure of an index is *crucial* for its performance because depending on the system environment it operates on, the optimal index structure can *vastly differ*. For example, optimal index structures can be different if we store data on network-attached devices instead of local SSD, *even if we index the same dataset.* We illustrate this with a concrete example.

Concrete Example. Suppose two different B-tree structures: Tall and Wide. (Note: our model formulation in § 3 supports more than B-trees.) Tall consists of 4 KB nodes, where each node has 200 fanout. Wide consists of 100 KB nodes, where each node has 5,000 fanout. Both Tall and Wide index the same dataset with one million distinct keys. The dataset is stored in 4 KB pages.

To index the dataset, Tall needs three layers (because the third layer can hold up to $200^3 = 8M$ pointers). Likewise, Wide needs two layers (because the second layer can hold up to $5000^2 = 25M$



(a) Tall & Narrow (w/ 200 fanout) (b) Shallow & Wide (w/ 5,000 fanout) (c) Query Latency Comparison

Figure 2: A motivating example for optimizing the entire index structure. Two candidate structures in (a) and (b). (a) has a lower branching factor (200); thus, it is quicker to retrieve each node while the index is higher. (b) has a greater branching factor (5,000); thus, its height is lower while retrieving each node takes more time. Figure (c) shows that depending on system environments (i.e., Short RTT vs. Fast RTT), different structures can show better performance (note: the latency of "High and Narrow" is used as a unit latency). See the text for the simulation setup.

pointers). Note that while Wide is shallower than Tall, fetching each node of Wide takes longer because its page size is 25× larger. Figs. 2a and 2b depict these structures.

Interestingly, neither of these two indexes (i.e., Tall and Wide) is superior to the other; that is, there is no single dominant index structure that offers faster lookup speed in all different system environments. To illustrate this, we suppose two storage devices, ShortRTT and LongRTT. ShortRTT operates with 5 ms latency (RTT) and 100 MB/sec bandwidth. LongRTT operates with 100 ms latency (RTT) and 100 MB/sec bandwidth. That is, their bandwidth is the same, but their latencies are different. (Note that our formulation in § 3 handles more general cases.)

Now, we show that (i) Tall offers higher performance than Wide if we store data on ShortRTT, and that (ii) Wide offers higher performance than Tall if we store data on LongRTT. For this, we adopt a widely used formula: (data transfer time) = (latency) + (data size) / (bandwidth). For instance, fetching a single page of the dataset stored on ShortRTT takes 5 ms + 4 KB / (100 KB/ms) = 5.04 ms.

Environment—ShortRTT**:** Wide **is 24% slower than** Tall**.**

- Tall needs 3 nodes and 1 data page = 3 × (5ms + 4KB / (100KB/ms)) + (5ms + 4KB / (100KB/ms)) = 20.16 ms
- Wide needs 2 nodes and 1 data page = 2 × (5ms + 500KB / (100KB/ms)) + (5ms + 4KB / (100KB/ms)) = 25.04 ms

Environment—LongRTT**:** Tall **is 29% slower than** Wide**.**

- Tall needs 3 nodes and 1 data page = 3 × (100ms + 4KB / (100KB/ms)) + (100ms + 4KB / (100KB/ms)) = 400.16 ms
- Wide needs 2 nodes and 1 data page = 2 × (100ms + 500KB / (100KB/ms)) + (100ms + 4KB / (100KB/ms)) = 310.04 ms

Fig. 2c summarizes this relative performance strength. For each environment, the figure reports the *relative* difference in end-to-end lookup time. It proves that different index structures offer higher lookup performance, depending on the storage device.

Need for Efficient Algorithm. While we use a relatively simple example to illustrate the core intuition, finding an optimal index structure is non-trivial to solve by hand, because there are exponentially many configurations we need to consider. To efficiently find an optimal index, we develop an intelligent learning technique that considers the entire index building as an optimization problem.

2.2 Scope of This Work

Read-only Index. We focus on building read-only indexes on a sorted key-value dataset. There are two reasons. First, this is a novel attempt at optimizing the *entire* index by casting it into a statistical optimization problem. Thus, we aim to ensure at least we can build best-in-class read-only indexes. Second, our long term plan is to integrate our index with write-oriented data systems based on LSM trees [30]. Note that segments of an LSM tree are not updatable; they are only merged occasionally to compose bigger (next level) segments. For these compaction operations, it suffices to build read-only indexes as part of compaction.

Point Lookup. We focus our evaluation on point lookup—fetching a value associated with a search key. There are two reasons. First, by optimizing point lookup, we also optimize range search because they involve essentially the same internal operations—traversing nodes from top to bottom. Second, range search can easily be supported by extending point lookup. To retrieve values for a key range `(begin, end)`, we can find the offset for `begin` using point lookup, then continue fetching data until we see `end`.

Persistent Index. We focus on the index we persist on storage (e.g., SSD, NVMe, flash drive), not the index maintained in main memory. This is because **we aim to enable faster lookup against large-scale data** kept in persistent databases (e.g., RocksDB, PostgreSQL, SQLite, etc). For this reason, our cost analysis (for fetching data) must incorporate storage-specific data transfer speed.

3 INDEX AS STATISTICAL OPTIMIZATION

We build an index by solving a statistical optimization problem. Specifically, our algorithm builds the fastest index for the data stored on a certain storage system, where the storage system is abstracted by the *data transfer function* $T(o)$. $T(o)$ is the time it takes to retrieve the data o (serialized on a storage device; see § 3.3 for its usage). In the rest of this section, we formulate an optimization problem and discuss how to solve it.

3.1 Index Class

An *index class* is a class of indexes we can express. The more general an index class is, the larger the set of indexes that we can represent; however, finding an optimal index can be more computationally

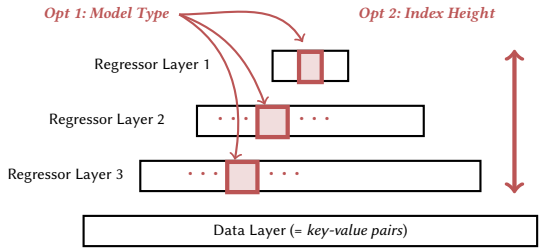


Figure 3: Index structure and optimization parameters.

expensive. **Our index class generalizes hierarchical indexes** such as B-trees [10, 14] and learned indexes [24], as described below.

An *index* consists of layers (see Fig. 3). There are two types of layers: *regressor layers* and *the data layer*. The data layer contains records sorted by keys. Each regressor layer consists of regressor(s). A regressor outputs a range `(start, end)` which is the data range (in byte offsets) we need to fetch in the next layer. If the next layer is a regressor layer, we fetch regressor(s); if the next layer is a data layer, we fetch key-value pairs (including the search key).

Querying. First, we fetch the entire root layer (i.e., Regressor Layer 1) containing multiple regressors. We use an appropriate regressor (for the given search key) to obtain an offset range for the next layer. This step continues until we reach the data layer. Note that we fetch the entire range at a time; that is, if there are L regressor layers, we fetch $L+1$ times until we obtain a desired key-value.

3.2 Parameter Space

A *parameter tuple* Θ captures the design considerations for an index. The goal of our statistical optimization is to find the optimal Θ ($= \Theta^*$) that minimizes the end-to-end lookup time ℓ. Θ consists of the following parameters:

(1) Number L of regressor layers ($R_1, \ldots, R_L$). (Note: For convenience, we use R_{L+1} for the data layer.)
(2) Regressor Type ($\boldsymbol{C} = C_1, \ldots, C_L$): A regressor layer R_l consists of regressor(s) of the same type (e.g., linear regression, step functions), which we denote by C_l. We search for the optimal regressor type for each regressor layer.
(3) Precision ($\boldsymbol{\lambda} = \lambda_1, \ldots, \lambda_L$) of regressor layers: A regressor outputs a range. While we ensure the range contains an appropriate regressor (in the next index layer) or a desired key-value pair (in the data layer), we must tune the sizes of those ranges, which we achieve via $\boldsymbol{\lambda}$.

In determining $\boldsymbol{\lambda}$, there is a natural trade-off. That is, if we build an accurate regressor (with a fine λ_i), its range output tends to be small, but the size of the regressor itself (and the regressor layer) must be big (e.g., with more coefficients for higher accuracy).

Using $\Theta = (L, \boldsymbol{C}, \boldsymbol{\lambda})$, we express the end-to-end lookup latency in terms of Θ as described in the next section.

3.3 Objective Function for Latency

We express a (tight) upper-bound on the end-to-end lookup time ℓ in terms of Θ and the data transfer function, $T(o)$. Specifically, ℓ is expressed in terms of actual regressor layers ($R_1, \ldots, R_L$) created as specified by Θ; here, the size of R_l depends on R_{l+1} as well as Θ (because R_l is to obtain the data fetch range for R_{l+1}).

Objective Function. Let $m(\cdot)$ be a process for creating a regressor. The created regressor's output range size is expressed by ε.

$$R_l := m(C_l, \lambda_l; R_{l+1})$$
$$\text{where } R_l \text{ fetches } \varepsilon(C_l, \lambda_l; R_{l+1}) \text{ bytes for } R_{l+1}$$

For a lookup, we first fetch the entire root layer (R_1); then, we subsequently fetch every range output by the latest obtained regressor until the data layer. Thus, ℓ can be expressed as:

$$\ell(L, \boldsymbol{C}, \boldsymbol{\lambda}) = T(m(C_1, \lambda_1; R_2)) + T(\varepsilon(C_1, \lambda_l; R_2)) + \sum_{l=2}^{L} T(\varepsilon(C_l, \lambda_l; R_{l+1})) \quad (1)$$

The goal is to find $\Theta^* = (L^*, \boldsymbol{C}^*, \boldsymbol{\lambda}^*)$ at which the above expression is minimized. We present our parameter search algorithm in § 3.4.

Discussion. Our objective function is an *upper bound* because some data may be cached in memory; in these cases, no data transfer is needed. We empirically observed that our approach also delivers strong performance in partially cached scenarios.

3.4 Index Build via Parameter Search

We provide a high-level sketch of our parameter search algorithm. We omit theoretical justifications and parallelization techniques. Our optimization is in the form of branch-and-bound algorithms—we *branch* out to construct the *best* R_l given (i) already constructed $R_{l+1}, \ldots, R_{L+1}$ and (ii) *optimal upper layers* (i.e., $R_1, \ldots, R_{l-1}$); then, we *bound* by pruning candidates. Here, the *optimal upper layers* are obtained via a recursive call of our algorithm, and the *best* R_l is evaluated using Eq. (1). There are three features that make our approach logically sound and efficient:

(1) Principled search for L^*: In designing our algorithm, one of the greatest technical challenges is finding the optimal number of layers. Our algorithm does this by asking "does stacking another layer reduce the total lookup time?" based on Eq. (1). This approach makes our algorithm logically sound.
(2) Effective enumeration of candidate Θ: Our algorithm is not amenable to gradient-descent-style algorithms for two reasons. First, the parameters for Θ include discrete choices (i.e., integer L and regressor type $\boldsymbol{C}$). Second, the regressor creation process $m(\cdot)$ is not always differentiable. Nevertheless, our search algorithm can find almost optimal Θ by effectively navigating in the discrete search space.
(3) Highly parallel: While our algorithm examines many different Θ, our total search & build process is highly efficient (comparable to existing *learned indexes* building) because our algorithm is designed to run in an *embarrassingly parallel* manner.

4 EXPERIMENT

We share our latest experiment results. While we continue making further enhancements in our implementation, our current results are strong, with the following key points:

(1) AutoIndex's learning algorithm exhibits expected behavior of finding optimal index structure for given system environments (§ 4.2).

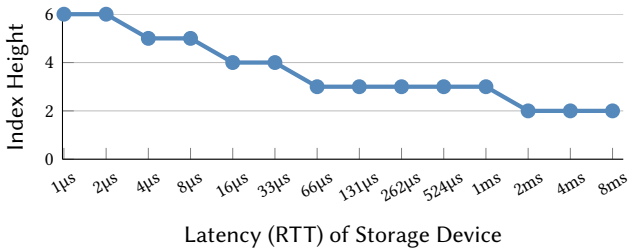


Figure 4: AutoIndex can adapt. For greater RTTs, AutoIndex properly creates shallower & wider indexes.

(2) AutoIndex's lookup speed outperforms state-of-the-art methods on different storage devices (§ 4.3).

We present each point after explaining our experiment setup.

4.1 Setup

We conduct experiments on Microsoft Azure. Our experiments use a large-scale benchmark dataset.

System. For VMs, we use `Standard_D8s_v3` instances (16 vCPUs, 64 GB memory). For Small RTT, we use OS disks (Premium SSD LRS, 256 GB, P20-2300 IOPS, 150 MBps). For Large RTT, we use Azure Cloud Storage (StorageV2) [2] connected via NFS v3 [3].

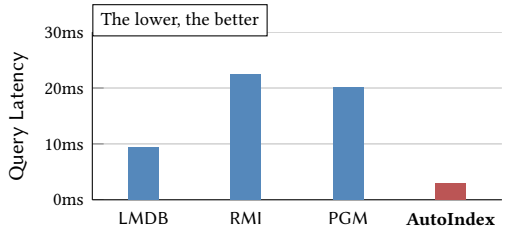
Compared Methods. We compare AutoIndex against several state-of-the-art index libraries:

(1) RMI: RMI [24] is one of the most commonly studied learned index implementations. RMI comes with an accuracy knob—our results compare RMI with the highest accuracy.
(2) PGM: PGM-Index [17] is one of the latest learned indexes. It offers provable worst-case bounds.
(3) LMDB: Lightning Memory-Mapped Database Manager (LMDB) is one of the fastest open-source B-tree libraries. A benchmark reports that LMDB is orders-of-magnitude faster than commonly used indexing systems such as LevelDB and SQLite3 [1].

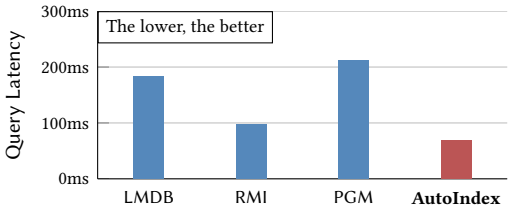
Dataset & Queries. We use the Facebook dataset included in "SOSD: A Benchmark for Learned Indexes" [21]. This dataset has 200 million keys. For querying, we randomly pick a key from the dataset and measure the end-to-end time for retrieving the value associated with the key (for each method we compare). To reduce variance, we report the average over twenty iterations.

4.2 AutoIndex Adapts

First, we test if AutoIndex produces different (and properly optimized) index structures according to given system environments. While there are many internal parameters defining the structure of an index, we specifically examine the number of layers (or equivalently, index height) because its results are easier to understand than other parameters (e.g., size of regressor layers, types of regressors). Expectation: As illustrated in § 2, a shallow index should be preferred if the latency (RTT) is large. Actual: To see if our learning algorithm also exhibits such outcomes, we run AutoIndex for different latency values while the bandwidth is fixed at 134 MB/sec. Fig. 4 shows the results. When the latency is extremely small (i.e., 1 microsecond), the algorithm tells us that a six-layered index delivers the fastest lookup speed. The number of layers gradually decreases



(a) Query Latency under Small RTT (Local Disk)



(b) Query Latency under Large RTT (Azure Storage)

Figure 5: AutoIndex (ours) delivers significantly faster query latency compared to other state-of-the-art methods. This is because AutoIndex optimizes the entire index according to target system environments.

(from 6 to 2) as we increase the latency, which is an expected result. We compare the actual lookup speed below.

4.3 AutoIndex is Fastest

Second, we show that AutoIndex outperforms the other indexes. Our comparison is performed using both local SSDs (small RTT) and Azure Cloud Storage (large RTT). § 4.1 describes the details about the methods we compare and also the dataset we use. Fig. 5 summarizes the results. Fig. 5a compares the lookup time for Small RTT (local disks). AutoIndex is the fastest: it is 3.3×–7.7× faster than other methods. Fig. 5b compares the lookup time for Large RTT (Azure Storage). Again, AutoIndex is the fastest: it is 1.4×–3.0× faster than other methods. This is because AutoIndex can optimize an entire index structure for target system environments.

5 ONGOING WORK

We are enhancing AutoIndex in several orthogonal directions. First, we are conducting additional theoretical analyses to formally study the optimality of our parameter search process. Second, we are extending our optimization problem to incorporate possible variance in I/O performance. Third, we are improving our builder to stream-process very large data.

ACKNOWLEDGMENTS

We thank the creators/developers of RMI [24], PGM-Index [17], and LMDB [1], who open-sourced their software, which helped us greatly in conducting accurate experiments. This work is supported in part by Microsoft Azure.

REFERENCES

[1] 2012. Database Microbenchmarks. http://www.lmdb.tech/bench/microbench/

[2] Accessed: 2022-05-29. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs.

[3] Accessed: 2022-05-29. Mount Blob storage by using the Network File System (NFS). https://docs.microsoft.com/en-us/azure/storage/blobs/network-file-system-protocol-support-how-to

[4] Accessed: 2022-05-29. MySQL database service. https://www.mysql.com/.

[5] Accessed: 2022-05-29. RocksDB: A persistent key-value store. https://rocksdb.org/.

[6] Marcos K Aguilera, Wojciech Golab, and Mehul A Shah. 2008. A practical scalable distributed b-tree. *Proceedings of the VLDB Endowment* 1, 1 (2008), 598–609.

[7] Michael R Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael J Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. 2013. Brainwash: A Data System for Feature Engineering.. In *Cidr*.

[8] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. Saqe: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.

[9] Rudolf Bayer. 1972. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica* 1, 4 (1972), 290–306.

[10] R Bayer and EM McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1, 3 (1972), 173–189.

[11] Juan Besa and Yadran Eterovic. 2013. A concurrent red–black tree. *J. Parallel and Distrib. Comput.* 73, 4 (2013), 434–449.

[12] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. Hot: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*. 521–534.

[13] Supawit Chockchowwat, Chaitanya Sood, and Yongjoo Park. 2021. Airphant: Cloud-oriented Document Indexing. *arXiv preprint arXiv:2112.13323* (2021).

[14] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[15] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[16] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2021. On the performance of learned data structures. *Theoretical Computer Science* 871 (2021), 107–120. https://doi.org/10.1016/j.tcs.2021.04.015

[17] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.

[18] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*. 1189–1206.

[19] Clinton Gormley and Zachary Tong. 2015. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.".

[20] Leo J Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*. IEEE, 8–21.

[21] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *NeurIPS Workshop on Machine Learning for Systems* (2019).

[22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–5.

[23] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. In *CIDR*.

[24] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[25] Leveldb. Accessed: 2022-05-29. LevelDB: A fast key-value storage library. https://github.com/google/leveldb.

[26] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[27] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. 2010. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1195–1206.

[28] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: an end-to-end economic method for elastic database fragmentation, replication, and provisioning. In *Proceedings of the 2018 International Conference on Management of Data*. 1253–1267.

[29] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.

[30] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[31] Yongjoo Park. 2017. Active Database Learning.. In *CIDR*.

[32] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. 2016. Visualization-aware sampling for very large databases. In *2016 ieee 32nd international conference on data engineering (icde)*. IEEE, 755–766.

[33] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*. 1461–1476.

[34] Yongjoo Park, Jingyi Qing, Xiaoyang Shen, and Barzan Mozafari. 2019. BlinkML: Efficient maximum likelihood estimation with probabilistic guarantees. In *Proceedings of the 2019 International Conference on Management of Data*. 1135–1152.

[35] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. 2017. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 587–602.

[36] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. Quicksel: Quick selectivity learning with mixture models. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1017–1033.

[37] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[38] Jun Rao and Kenneth A Ross. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 475–486.

[39] Benjamin Sowell, Wojciech Golab, and Mehul A Shah. 2012. Minuet: A Scalable Distributed Multiversion B-Tree. *Proceedings of the VLDB Endowment* (2012).

[40] Splunk. Accessed: 2022-05-29. Splunk: The Data-to-EverythingTM Platform. https://www.splunk.com/

[41] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.

[42] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. 2020. PA-Tree: Polled-Mode Asynchronous B+ Tree for NVMe. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 553–564.

[43] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. 2020. SIndex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 17–24.

[44] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. 2010. Efficient B-tree based indexing for cloud data processing. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1207–1218.

[45] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proceedings of the VLDB Endowment* (2019).

[46] Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. 2019. S3: a scalable in-memory skip-list index for key-value store. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2183–2194.