Tuning Hierarchical Learned Indexes on Disk and Beyond

Supawit Chockchowwat
University of Illinois at Urbana-Champaign
USA
supawit2@illinois.edu

CCS Concepts

• **Information systems** → **Point lookups**; Unidimensional range search; **Record and block layout**; • **Mathematics of computing** → Regression analysis; Density estimation.

Keywords

index, learned index, data access methods, point lookup, key-value database, external memory, storage, storage-aware, latency, bandwidth, automatic tuning, optimization

ACM Reference Format:
Supawit Chockchowwat. 2022. Tuning Hierarchical Learned Indexes on Disk and Beyond. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3514221.3520255

1 Problem and Motivation

Entry retrieval—a process to retrieve rows whose field(s) associates with the given key(s)—is one of the core operations in databases. Classical indexes such as B-tree [2, 4] and skip list recursively partition the key space into a hierarchical structure. Such a structure retrieves an entry by traversing the path of partitions that enclose the given key, effectively reducing uncertainty of key's position as the traversal proceeds.

Recently, index designers have developed interests in *learned indexes*, a concept introduced by [13]. Using patterns in key-position pairs, a learned model can provide a significantly higher information gain about the key's position than a pessimistic partitioning index can [8]. Many works have successfully outperformed classical indexes by multiple factors in latency and memory usage. Overall, they incorporate various combinations of models, partitioning, error corrections (a.k.a. last mile search), mutability, and tunable parameters [6, 9, 10, 12, 16] among many other works.

These progresses focus on in-memory settings that ensure a fast random access; nevertheless, numerous database deployments rely on peripheral storage such as disks, SSDs, remote storages, and object stores that break the "fast random access" assumption. Is this assumption necessary to show a learned index supremacy? How can we redesign learned indexes to exploit key-position patterns on disk and beyond? Specifically, we study learned indexes with respect to the *external memory model*, widely adopted in on-disk performance analyses. *External memory* refers to the farthest storage which is the slowest and/or most expensive. In this model, the external memory usage dominates the total cost; as a result, an efficient index must minimize accesses to such a memory.

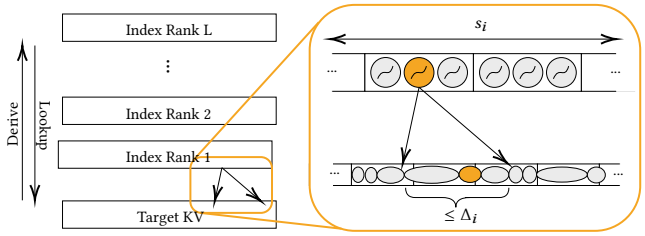

Figure 1: Overview of AirIndex as a key-value hierarchical learned index of rank L where index rank i derives index rank $i+1$ and conversely looks up index rank $i-1$ (rank 0 refers to the target key-value data). The close-up diagram layouts key-value pairs (circles and ellipses) stored in pages (rectangles) on the external memory. It also denotes the size s_i and maximum error Δ_i of any index rank i.

Problems arise when a system stores its learned index in an external memory. First, because existing learned indexes produce the key's position in the sorted key list, the system would make an extra round trip to retrieve the actual entry. Comparison-based error corrections (e.g. binary search like [12] or exponential search [3]) would induce many sequential random accesses, even if the error is small. To workaround these problems, one might take a naïve approach of fetching the entirety of the learned indexes into memory before accepting any query; however, the system would incur a huge cost, especially to the initial query of each data collection touched by the workload.

2 Approach

Guided by the external memory model, we design our learned index prototype, AirIndex, to minimize accesses to the external memory. Our observation identifies that an external memory algorithm should be aware of unique storage properties. To cope with the lack of a fast random access, AirIndex solidifies learned index's functionality and encourages independence between decisions within a lookup. Furthermore, In learning optimal index structures, its tuner assesses storage's capabilities and optimizes fine-grained configuration parameters appropriately.

Toward Key-Value Index In lieu of a key model $g : K \to [n]$ for key set K where $|K| = n$, AirIndex's model directly predicts the key's byte offset, $f : K \to [B]$ for B bytes of data. Although it is possible to amend an existing key index with a fixed-size pointer, the resulting index would require an extra round trip to follow the pointer indirection to read the value. On the other hand, key-value

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3520255

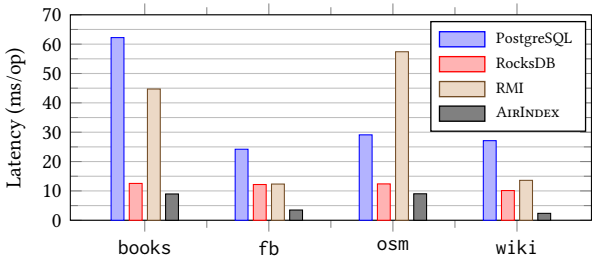

Figure 2: Average Lookup latency (over 70k queries) across SOSD datasets of different systems on Azure NFS.

index saves one round trip per each key-value lookup. As a matter of fact, AirIndex cuts away half of its round trips by this decision.

Recall that most learned indexes [6, 9, 10, 12, 13, 16] internally model piecewise functions that can be decomposed into multiple submodels partitioned by their separator keys. We can then consider an index model as a key-value collection where each key describes the effective key interval while its value is a submodel. Naturally, a key-value index model is now equivalent to yet another key-value collection, derived from the origin key-value collection and potentially deriving further. After L derivations of indexes on top of indexes, we obtain a rank-L hierarchical learned index as illustrated in Fig. 1. Furthermore, each submodel can have different sizes, opening for opportunities to tailor a diverse collection of submodels accordingly to the key-position pairs.

Moreover, AirIndex is designed to exploit the efficiency of sequential search in accessing data in external memory. That is, AirIndex writes key-value pairs compactly in fixed-size pages. Given a byte offset, its storage module fetches the corresponding page and prefetches next pages in parallel. Combining with accurate position predictions, sequential search on pages with prefetching is much more attractive than comparison-based searches.

Storage-aware Auto-tuning Assuming all data are in the external memory, each lookup retrieves the entire topmost rank-L index (i.e. root), predict-correct-retrieves rank-$(L-1)$ submodel, predict-correct-retrieves rank-$(L-2)$ submodel, and so on, until reaching the target key-value pair. Hence, our optimization formulation is a constrained problem where each rank-i model has the error at most Δ_i bytes and total size s_i bytes. AirIndex needs to balance between each (Δ_i, s_i) pair. If it tightens up Δ_i, the model needs to refine and enlarges its size s_i which increases its key-position complexity and creates a burden for the upper index model. For a fixed L, the tuner additionally imposes a upper size constraint $s_i \leq O(\sqrt[L+1]{n})$ which consequently guarantees $\Delta_{i+1} \leq O(\sqrt[L+1]{n})$, matching the classical index compression ratio.

The final objective function boils down to $\mathcal{L}(\Delta_1, \Delta_2, \ldots, \Delta_L, s_L)$ a function monotonic in all arguments. It implies that the tuner should minimize the errors Δ_i of intermediate indexes $i \in [L-1]$ with the earlier size constraints, while it should allocate s_L and Δ_L equally. To tune the only remaining parameter L, AirIndex's tuner profiles the storage for its round trip time (RTT) and bandwidth in bytes per unit time. It then sketches the index structure for each L (skipping index writing) and evaluates the latency using the storage's profile and $\{(\Delta_i, s_i)\}_{i=1}^{L}$ to select the best L. In general, a remote external memory such as NFS favors a shallow-but-wide hierarchy (smaller L, larger Δ_i and s_i) and vice versa.

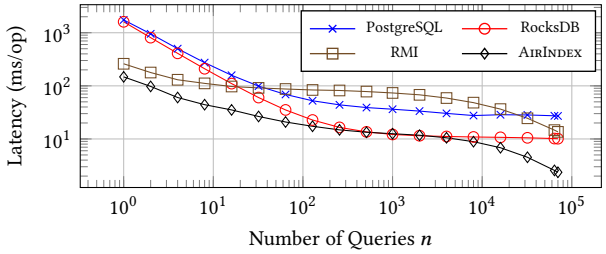

Figure 3: Average latency over first n queries of different systems on `wiki` dataset. The plot shows $n \in \{1, 2, 4, \ldots, 512, 1k, 2k, 4k, \ldots, 64k, 70k\}$. Both axes are in the logarithmic scale.

3 Preliminary Results

Our preliminary experiment validates our tuning methodology in comparison with PostgreSQL [15], RocksDB [7], and RMI (described in [13] Section 3) on the Search On Sorted Data (SOSD) benchmark [11, 14]. Each system persists all data on an Azure NFS server as the external memory. To access its data on the storage, RMI instead operates on virtual memory addresses via `mmap`. Even though SOSD is a key dataset, AirIndex supports index lookup by replacing each i-th key k_i with (k_i, i) key-value pair. Its internal model is based on piecewise linear functions as a proof of concept. Fig. 2 demonstrate that AirIndex is consistently the fastest in all 4 datasets, both in terms of 1.21×–23.09× faster initial latency (cold latency) and 1.37×–11.50× faster converged latency (warm latency after 70k queries).

Fig. 3 displays a representative trend of average latency as a function of "hotness" (number of queries). In general, average latency decreases rapidly from the beginning where no data is in memory. After 200 queries, the latency trends start to plateau, indicating that the systems have cached their index pages closer to the root. After 10k queries, RMI and AirIndex accelerate even more by exploiting lower caches in the memory hierarchy through mmap. In summary, AirIndex not only is optimized for initial retrieval but also converges and searches more quickly in its warm state.

4 Related Works

To the best of our knowledge, two closely related works have successfully shown benefits of learned indexes on disk. BOURBON [5] applies learned indexes to accelerate search within files in levels of log-structured merge (LSM) trees. This is orthogonal from our work; our framework can tune the parameters in each index to learn each file in BOURBON, accordingly to the underlying storage. On the other hand, instead of predicting locations of written records, learned indexes in [1] are responsible for distributing data into blocks as evenly as possible before writing to disk. Nonetheless, with a unit conversion between blocks and bytes, we believe that our framework can optimize index structure for block data storage with coarser unit of reads like BigTable as well.

5 Acknowledgement

We would like to offer special thanks to Yongjoo Park, Wenjie Hu, and Yuzhou Mao as well as supports from UIUC.

References

[1] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H. Chi, Lyric Doshi, Tim Kraska, Xiaozhou, Li, Andy Ly, and Christopher Olston. 2020. Learned Indexes for a Google-scale Disk-based Database. arXiv:2012.12501 [cs.DB]

[2] R. Bayer and E. M. McCreight. 1972. Organization and maintenance of large ordered indexes. *Acta Informatica* 1 (1972), 173–189. Issue 3. https://doi.org/10.1007/BF00288683

[3] Jon Louis Bentley and Andrew Chi-Chih Yao. 1976. An almost optimal algorithm for unbounded searching. *Inform. Process. Lett.* 5, 3 (1976), 82–87. https://doi.org/10.1016/0020-0190(76)90071-5

[4] Douglas Comer. 1979. UBIQUITOUS B-TREE. *Comput Surv* 11 (1979). Issue 2.

[5] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 155–171. https://www.usenix.org/conference/osdi20/presentation/dai

[6] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 969–984. https://doi.org/10.1145/3318464.3389711

[7] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (Oct. 2021), 32 pages. https://doi.org/10.1145/3483840

[8] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 3123–3132. https://proceedings.mlr.press/v119/ferragina20a.html

[9] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proc. VLDB Endow.* 13, 8 (April 2020), 1162–1175. https://doi.org/10.14778/3389133.3389135

[10] 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* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1189–1206. https://doi.org/10.1145/3299869.3319860

[11] 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).

[12] 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* (Portland, Oregon) *(aiDM '20)*. Association for Computing Machinery, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3401071.3401659

[13] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. *Proceedings of the ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/3183713.3196909

[14] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking Learned Indexes. *Proc. VLDB Endow.* 14, 1 (2020), 1–13.

[15] PostgreSQL. [n. d.]. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org. [Online; accessed November-12-2021].

[16] 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* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 308–320. https://doi.org/10.1145/3332466.3374547