PostLearn: Towards A Learned Index For PostgreSQL

Abrar Fuad
abrar.fuad@mail.mcgill.ca
McGill University

Shubham Vashisth
shubham.vashisth@mail.mcgill.ca
McGill University

Oana Balmau
oana.balmau@mcgill.ca
McGill University

Bettina Kemme
bettina.kemme@mcgill.ca
McGill University

Abstract

Recent advances in machine learning based data structures, such as learned indexes, show that models can be trained to approximate the cumulative distribution of keys, thereby predicting key positions more efficiently than traditional indexes. While research prototypes have shown significant promise, so far there has been no comprehensive integration of learned indexes into a relational database system. In this paper, we explore the feasibility of learned indexes for relational systems by introducing `PostLearn`, an integration of the ALEX+ learned index as a native PostgreSQL index access method. We detail the design and the implementation challenges of embedding a model-based index within PostgreSQL's existing infrastructure and explore its potential performance benefits. While `PostLearn` can achieve up to 1.5x speedups for point lookups and small range scans compared to the built-in B$^+$-Tree under selected workloads, end-to-end performance benefits are considerably lower than when tested in isolation.

CCS Concepts: • **Information systems → Database indexing**; *Database query processing*; *Database management system engines*; • **Computing methodologies** → Machine learning approaches.

Keywords: Learned Indexes, PostgreSQL, Indexing, Database Systems

ACM Reference Format:
Abrar Fuad, Shubham Vashisth, Oana Balmau, and Bettina Kemme. 2026. PostLearn: Towards A Learned Index For PostgreSQL. In *6th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3805687.3806257

This work is licensed under a Creative Commons Attribution 4.0 International License.
CHEOPS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2604-0/26/04
https://doi.org/10.1145/3805687.3806257

1 INTRODUCTION

Recent research has demonstrated that learned indexes can approximate key distributions using machine learning models to reduce search latency and memory footprint [8] compared to the traditional B-tree/B$^+$-Tree index approaches. In principle, learned indexes use a hierarchy of models (often linear regressions) to map a key directly to an estimated position in a sorted array, progressively refining predictions across stages of models. Despite strong performance results over the B$^+$-Tree when tested in isolation [2, 24], to the best of our knowledge, no relational database system yet offers a learned index, and thus their potential and limitations within relational database systems remains largely unexplored. As such, the goal of our research is to provide a full integration of a learned index into a relational database system to better understand the performance implications for end-to-end workloads and the challenges when embedding such indexes into a production system. In this preliminary work, we address the following research questions:

Q1) Execution Pipeline Performance. *How does a learned index perform when embedded within a complex relational execution pipeline?* We investigate the performance impact as the index moves from a standalone micro-benchmark to a production system.

Q2) Multi-Process Access. PostgreSQL employs a multi-process architecture where client requests are handled by different backend processes. *How can a learned index be accessed by these different backends?* In PostgreSQL, relational tables and traditional indexes are stored in files on persistent storage and organized as fixed size pages that are only partially cached in a page-based shared buffer pool managed by the buffer manager. In contrast, most learned indexes are designed as main-memory data structures that ignore fixed page boundaries. Prior work [9] demonstrates that forcing learned indexes into page-based layouts introduces performance bottlenecks due to fragmentation or model maintenance overheads. Consequently, we maintain the default page-free design, assuming that modern machines have sufficient memory to hold learned indexes. As such, we explore how such a structure can be shared safely and efficiently across multiple PostgreSQL backends while operating outside the traditional page-based buffer framework.

Q3) Index Selection. *How can a learned index be transparently used for a query?* That is, how can the execution engine treat it like any native index, using it to perform an index scan when available?

In this paper, we assume a read-only workload: once an index is created on an existing table, there are no further updates on the table. Ultimately, however, the index needs to support concurrent update transactions offering the appropriate atomicity and durability properties. Therefore, we decided to use ALEX+ [24] for our integration, an updatable learned index that is also concurrent, so that we can eventually integrate transactional support. Our contributions are as follows:

- **Native PostgreSQL Index Access Method:** `PostLearn` is a native integration of the ALEX+ learned index into PostgreSQL, implemented as an extension to the existing index framework.
- **Comparative Analysis:** We provide an empirical evaluation of `PostLearn` in a relational setting, comparing its performance against PostgreSQL's built-in B$^+$-Tree index.
- **Design Space Analysis:** We identify and discuss the remaining challenges to make the integration production-ready. This serves as a roadmap for future research, focusing on the multi-process concurrency control, transaction support, and persistence.

2 BACKGROUND AND MOTIVATION

Overview of ALEX+: ALEX+ [24], is a multi-threaded extension of ALEX [2]. It uses a hierarchical structure composed of *internal nodes* and *data nodes*, as shown in Figure 1. Internal nodes facilitate top-down traversal by using linear regression models to "compute" the array index of the appropriate child pointer. At the leaf level, data nodes store the keys and payloads, and a local model to predict key positions within the leaf. To support efficient updates, ALEX+ uses *gapped arrays* inside data nodes. Unlike a B$^+$-Tree leaf that fills contiguously and leaves free space only at the end of the node, a gapped array strategically distributes empty "gaps" across the data node based on the underlying data distribution. This enables model based insertion, where new keys are placed at their model predicted positions minimizing expensive shifts or retraining.

A bitmap is maintained in each data node to track occupied positions and skip gaps. The size of the data nodes is dynamic and depends on the data distribution. Additionally, ALEX+ supports duplicate keys, and thus, can be applied to any attribute across a relation. An initial ALEX+ index over an existing dataset is created via a "bulkload" function and then the index can be further updated. For point lookups, internal models calculate child pointers down to a data node, where a local model predicts the key's position, followed by exponential search to correct any mispredictions. Range scans begin with a lookup for the starting key, followed by a linear traversal via sibling pointers.

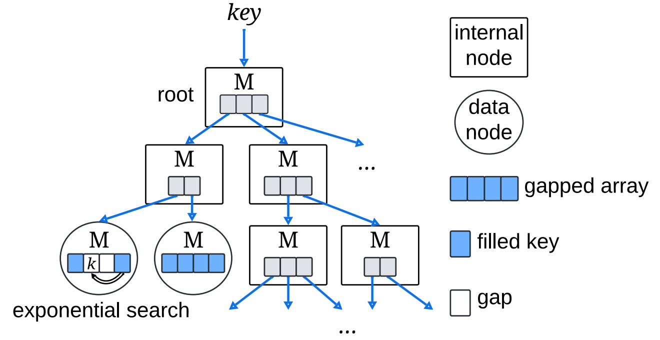

Figure 1. Overview of the ALEX+ Design

ALEX+ vs B$^+$-Tree in isolation: In Figure 2 we evaluate the strengths of ALEX+ over a traditional B$^+$-Tree in an index-only configuration using a read-only workload from the GRE benchmark on the *YCSB* dataset, confirming previous results [23, 24]. ALEX+ achieves up to 2.8× and 2.3× higher throughput for point lookups and short range scans, respectively, due to its model based navigation. The speedup over B$^+$-Tree diminishes with increasing scan length, as the bitmap driven gap traversal in data nodes introduces increasing overhead. One of our key questions with `PostLearn` is understanding how much of this advantage can be preserved end-to-end, when ALEX+ runs inside PostgreSQL.

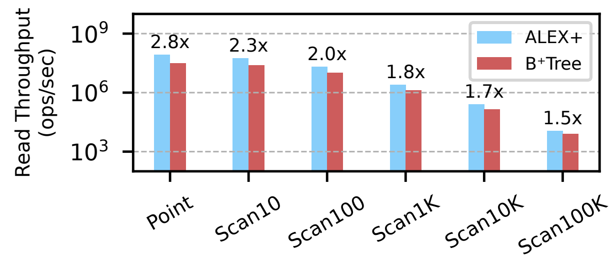

Figure 2. ALEX+ vs B$^+$-Tree in a read-only (point queries and scans), in-memory workload, outside of PostgreSQL.

Indexes and Shared Memory in PostgreSQL: PostgreSQL offers various index implementations (e.g. B$^+$-Tree , GiST, or GIN [18, 21]). Indexes do not contain the actual data records, but *Tuple Identifiers* (TID) that map to the location of the record inside the relational file. Indexes need to implement the *Index Access Method* (AM) interface (API) that allows the core database engine to interact with the index. Query optimization evaluates different execution paths, including the choice of index, estimating their cost via the AM API. At runtime, the Executor drives the scan by calling AM-specific functions (e.g., `amgettuple`). Finally, the index returns the

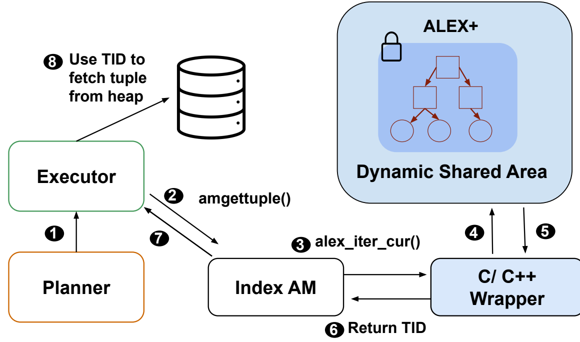

Figure 3. Index Scan via `PostLearn`: Bridging the PostgreSQL Executor to the ALEX+ backend via C/C++ Wrapper and Shared Memory.

TID, which the executor uses to fetch the corresponding record from the relation [17].

In addition to the page-based shared buffer pool that caches the relational and traditional index data, the backends also have access to a Dynamic Shared Area (DSA) [15]. DSA provides a heap-like allocator that allows multiple memory segments to be mapped as a single logical workspace. This ensures that shared memory pointers remain valid across different backend address spaces and that allocations can expand dynamically.

3 POSTLEARN DESIGN & INTEGRATION

`PostLearn` is implemented as a native *Index Access Method* (AM) [16] and packaged as a PostgreSQL extension in the `src/contrib` directory. Figure 3 shows the integration of `PostLearn` into the PostgreSQL workflow.

Implementation of the Index AM Interface: To achieve native integration, `PostLearn` implements the PostgreSQL Index AM API. Index construction is performed via the *ambuild* callback [16], which invokes ALEX+'s bulk loading function. Just like in the B^+-Tree, entries in the `PostLearn` data nodes store the TID as their payload. For query execution, `PostLearn` leverages PostgreSQL's "one by one" tuple delivery mechanism [16] by mapping *amgettuple* calls to ALEX+'s internal iterators. During a a point lookup (`WHERE x = a`) or a range query (`WHERE x BETWEEN y AND z`), the scan state including bounds and active iterators is maintained. Upon the first *amgettuple* invocation, the index performs a model-based traversal to locate the initial key-TID pair at leaf level. Subsequent calls then utilize leaf-level hopping, where the ALEX+ iterator is advanced to retrieve next-in-sequence TIDs until the search space is exhausted. This architectural alignment allows `PostLearn` to support complex scans while avoiding the re-traversal of the index for every tuple. As a native Index AM, `PostLearn` is integrated into the system catalogs upon creation via SQL command *CREATE INDEX*, ensuring that the index is discoverable by all backends.

C++/C Bridging: To resolve the language mismatch between PostgreSQL (C) and the ALEX+ library (C++), we developed a C/C++ wrapper with an `extern "C"` interface. This layer encapsulates the ALEX+ templates, abstracting away C++ specific constructs such as template instantiations which are incompatible with the PostgreSQL C source code. This bridge allows PostgreSQL to invoke learned index operations as if they were native C functions.

Storing the Index in DSA: PostgreSQL follows a multi process model in which each client operates within a private address space, whereas ALEX+ was originally designed for multi threaded access within a single process. If now a single backend creates the index as is, other backends would not be able to access it. To ensure global visibility, we create the index within the Dynamic Shared Area (DSA) introduced in Section 2. It can be accessed by all backends and is well suited for the adaptive main-memory structure of ALEX+. By replacing private heap allocations in ALEX+ with `dsa_allocate` for model nodes' child pointer arrays, and data nodes' bitmaps and key-TID slots, we ensure that the entire `PostLearn` resides in shared memory and is accessible to all backends.

While `PostLearn` is registered in the PostgreSQL system catalogs upon creation, the catalogs only store metadata and not the runtime memory addresses of the index. Since the index resides in volatile shared memory rather than on disk, we maintain a global shared state in DSA shared memory to track these structures across backend processes. This state maintains a mapping of `(RelationOID, AttributeNumber)` to the corresponding DSA handle (address) of the ALEX+ instance. By ensuring that the ALEX+ object and all its internal members are themselves allocated via the same DSA approach, we enable the co-existence of multiple `PostLearn` indexes on different attributes of a relation, each independently accessible.

Concurrent Access: We do not rely on the actual ALEX+ concurrency mechanism as it only considers threads within a single process. So far, we implement a simplistic concurrency control mechanism that will work correctly if there is first a single writer that creates the index, and afterwards only readers access the index. For that, `PostLearn` uses PostgreSQL's native *LWLocks* (Lightweight Locks) [14] on each inner/data node. It allows multiple readers to access the index structure concurrently. *LWLocks* are managed within PostgreSQL's shared memory by default, being visible to all backends and allowing PostgreSQL to automatically release held locks and prevent system-wide deadlocks if a backend process crashes unexpectedly. This initial implementation will be our starting point when we implement update and transactional support.

4 EVALUATION

Our experiments aim to answer the fundamental question: *Can the throughput advantages of ALEX+ over the B^+-Tree*

be preserved within PostgreSQL, or will the overhead of the executor's pipeline and buffer management negate the inherent benefits of the learned model?

Hardware and Software Environment. Experiments were conducted on a dedicated laboratory virtual machine equipped with a 2.60GHz 4-core Intel Xeon E5-2690 processor, 128 GB of main memory and 500 GB of local storage. The system runs Ubuntu 22.04 (Linux 5.15) on x86_64. Experiments use PostgreSQL 18 with `PostLearn` implemented as a native index access method, compiled using GCC/G++ 11.4.0 with C++14 support.

PostgreSQL Configuration. PostgreSQL was configured with `shared_buffers=16GB`. Our evaluation is restricted to an in-memory setting where all data is either in the PostgreSQL buffer or in DSA to eliminate disk I/O effects during measurements.

Experiment Design. We focus on synthetic read-only workloads designed to evaluate lookup and scan performance. We construct a synthetic relation R consisting of a primary key *id* and two further padding attributes. The *id* attribute serves as the indexed column and ranges uniformly from 1 to 100 million. Our evaluation compares `PostLearn` against the standard B$^+$-Tree, the primary general-purpose access method for indexing sorted scalar data and the baseline used in most learned index studies [2, 24, 25]

4.1 Lookups & Scans

We first consider a single-client configuration, where we execute stored procedures that either perform 1 Million randomized point lookups over the indexed *id* attribute or evaluate 100K random range scans of varying selectivity to examine execution time with the number of tuples returned. These procedures execute entirely on the server side, eliminating client–server network latency.

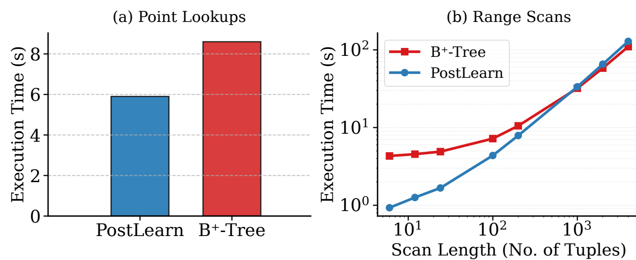

Figure 4. PostLearn vs. B$^+$-Tree on read-only workload. (a) 1M random point lookups; (b) Range scans of increasing length.

Figure 4 shows the average execution time for the (a) 1M point-queries and (b) individual scans of different lengths. There is a significant difference compared to our isolated micro-benchmarks (Section 2); the 2.8× performance advantage of `PostLearn` (underlying ALEX+) for point queries is reduced to 1.46× inside the PostgreSQL engine. This slowdown occurs because of the executor overhead and tuple visibility checks (PostgreSQL is a multi-version system [19]). For range scans (Figure 4(b)), `PostLearn` maintains a strong lead for short ranges, but the B$^+$-Tree becomes faster as scan lengths exceed 1000 tuples: while `PostLearn`'s model based lookup accelerates the initial search to the starting key, the subsequent sequential scan is hampered by the Gapped Array layout. Unlike the B$^+$-Tree 's packed leaf pages, `PostLearn` must skip over empty slots and check bitmaps for valid data. The isolated ALEX+ in Section 2 also shows a slowdown, but it is more pronounced in `PostLearn`.

Nevertheless, `PostLearn` still outperforms B$^+$-Tree for a wide range of queries, and as such, remains a promising database index.

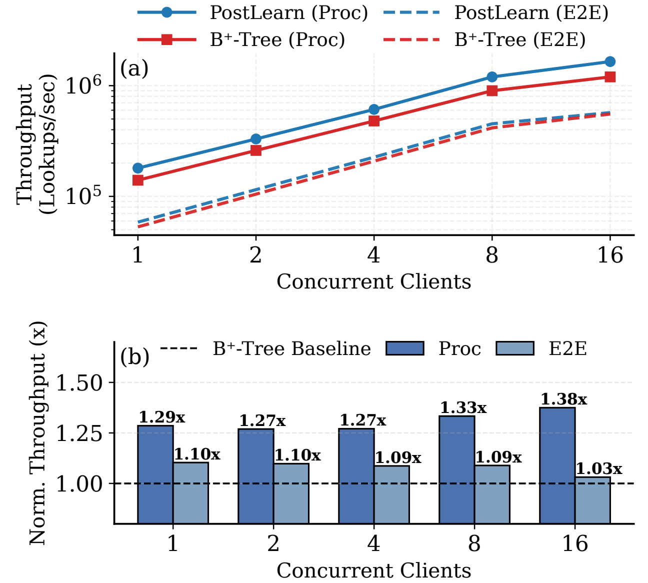

Figure 5. (a) Multi client scalability comparing PostLearn and B$^+$-Tree under server-side procedural (PROC) and end-to-end (E2E) workloads. (b) Normalized Throughput with varying client counts.

4.2 Multi-Client Scalability

We evaluate scalability using 1 to 16 clients. We utilize two workloads: a **Server-Side Procedure** (PROC) that executes 1M randomized lookups using the indexed attribute *id*, and an **End to End** (E2E) workload where 10 random IDs are queried and the resulting tuples are returned to the client. E2E captures the overhead of the PostgreSQL frontend-backend protocol. We use `pgbench` [13] for this experiment and run each workload for 30 seconds.

As shown in Figure 5(a), there is an order of magnitude gap (30x) in throughput between PROC and E2E workloads. As the procedural workload runs entirely within the PostgreSQL backend, it avoids client–server communication and

result materialization overheads. Figure 5(b) further highlights the performance differences between B$^+$-Tree and PostLearn. While PostLearn provides a consistent 1.27× to 1.38× speedup in the procedural environment, this margin compresses to approximately 3-10% in the E2E scenario. This offers a critical insight: when the total execution time is dominated by fixed architectural bottlenecks such as the executor and network latency, the algorithmic efficiency of a learned index is effectively amortized across the total query cost.

5 FUTURE DIRECTIONS

Concurrent Update Transactions. While PostLearn currently targets read only workloads, supporting mixed read-write transactional workloads remains an important next step. This involves several considerations. PostgreSQL employs Multi-Version Concurrency Control (MVCC) [19], creating a new record version on each update. Reads access committed versions without locking, while conflicting writes trigger transaction aborts. Aborted versions remain invisible in the system. PostgreSQL's B$^+$-Tree index tracks all versions and marks obsolete entries, so aborts do not require undoing index changes, and its concurrency control prevents inconsistencies during node updates or splits.

Thus, although ALEX+ lacks transactional awareness, its concurrency mechanism could potentially be adapted for PostLearn. However, ALEX+ supports only thread-level concurrency within a single process, whereas PostgreSQL uses a multi-process architecture, requiring new mechanisms for cross-process synchronization.

Persistence and Storage Integration. We deliberately avoided a page-based design for PostLearn to leverage the main-memory characteristics of existing implementations. While PostgreSQL's DSA API enables this approach, it prevents the use of PostgreSQL's built-in transactional persistence mechanisms based on dirty page tracking, checkpointing, and Write-Ahead Logging (WAL) [20]. As an initial step, PostLearn could be persisted by writing the index to a file at creation time and reconstructing it if the in-memory structure is discarded. However, this supports only read-only indexing. To support updates and crash recovery, a promising direction is file-based checkpointing coordinated with transactional logging to reconstruct a consistent index state.

Cost Modelling. An effective cost model is necessary for PostgreSQL's query planner to choose among multiple index types. Since learned indexes have different performance characteristics than traditional indexes [24], future work should develop adaptive cost models that consider the learned index type and workload patterns. This becomes particularly important when multiple indexes exist on the same attribute. In our current evaluation, we manually tuned the cost parameters to ensure the planner selected PostLearn for index scans, allowing us to isolate its performance. Future work will replace this static configuration with a dynamic cost estimation function integrated into PostgreSQL's optimizer.

6 RELATED WORK

Learned Indexes. Early learned indexes have evolved rapidly from targeting static datasets and read-only workloads [4, 7, 8], to support for inserts and deletes [2, 3, 25], concurrency control [24], and durability [10, 11].

Prior work has integrated learned indexes into Log-Structured Merge (LSM) stores; Bourbon [1] improved LevelDB lookups by 1.78×, while DobLIX [5] boosted RocksDB throughput by 2.21×. However, these systems target lightweight key-value stores designed around simple Get/Put APIs and optimized for sequential disk I/O. These KV stores also have a very different architecture compared to relational database systems, both in main-memory and on disk. In contrast, relational database systems such as PostgreSQL employ a more complex execution pipeline, including query parsing and optimization followed by the executor's tuple by tuple delivery, while performing MVCC visibility checks and interacting with the shared buffer manager. These layers introduce substantial per-tuple processing overhead absent in typical KV-store workloads. KV stores also typically do not support transactions. As far as we are aware of, PostLearn is the first learned index deployed to operate within the full architectural constraints of a production RDBMS.

Machine Learning in Relational Databases. Recent work explores augmenting traditional relational database components with machine learning techniques. In query optimization, Neo [12] relies on deep neural networks to generate query execution plans on top of existing optimizers (such as the one from PostgreSQL) while applying reinforcement learning [22] to learn from incoming queries. On the other hand, Lero [26] reformulates plan selection as a pairwise ranking problem, training classifiers to compare and discriminate between plan candidates to refine the output of existing optimizers. Complementary research on learned cardinality estimation [6] also shows that neural models can outperform classical histogram-based techniques by capturing complex data correlations, thereby improving optimization decisions.

7 CONCLUSION

In this paper, we introduced PostLearn, a native learned index access method integrated directly into PostgreSQL. Our performance evaluation shows that while learned indexes maintain a performance lead over B$^+$-Tree , the relative advantage is attenuated by the database executor's overhead. Still, in specific scenarios (e.g., heavy on point lookups) learned indexes can be a compelling alternative to the B$^+$-Tree .

References

[1] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 155–171.

[2] 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*. ACM, 969–984. doi:10.1145/3318464.3389711

[3] 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. doi:10.14778/3389133.3389135

[4] 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 (SIGMOD)*. ACM, 1189–1206. doi:10.1145/3299869.3319860

[5] Alireza Heidari, Amirhossein Ahmadi, and Wei Zhang. 2025. DobLIX: A Dual-Objective Learned Index for Log-Structured Merge Trees. *Proceedings of the VLDB Endowment* 18, 11 (2025), 3965–3978. doi:10.14778/3749646.3749667

[6] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-Depth Study. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. ACM, 1214–1227.

[7] 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 aiDM 2020*. ACM. doi:10.1145/3401071.3401659

[8] 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 (SIGMOD)*. ACM, 489–504. doi:10.1145/3183713.3196909

[9] Hai Lan, Zhifeng Bao, J. Shane Culpepper, and Renata Borovica-Gajic. 2023. Updatable Learned Indexes Meet Disk-Resident DBMS: From Evaluations to Design Choices. *Proceedings of the ACM on Management of Data* 1, 2 (2023).

[10] Hai Lan, Zhifeng Bao, J. Shane Culpepper, Renata Borovica-Gajic, and Yu Dong. 2024. A Fully On-Disk Updatable Learned Index. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. IEEE, 4856–4869.

[11] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *arXiv preprint arXiv:2105.00683* (2021).

[12] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[13] PostgreSQL Global Development Group. 2025. *pgbench — Run a Benchmark Test on PostgreSQL*. https://www.postgresql.org/docs/16/pgbench.html

[14] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Concurrency Control*. https://www.postgresql.org/docs/current/mvcc.html

[15] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Dynamic Shared Memory*. https://doxygen.postgresql.org/dsa_8h.html

[16] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Index Access Method*. https://www.postgresql.org/docs/current/index-functions.html.

[17] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Index Access Method Interface Definition*. https://www.postgresql.org/docs/current/indexam.html

[18] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Index Types*. https://www.postgresql.org/docs/current/indexes-types.html

[19] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Multiversion Concurrency Control (MVCC)*. https://www.postgresql.org/docs/current/mvcc.html

[20] PostgreSQL Global Development Group. 2025. *PostgreSQL Documentation: Write-Ahead Logging (WAL)*. https://www.postgresql.org/docs/current/wal-intro.html.

[21] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*.

[22] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT Press.

[23] Chaichon Wongkham et al. 2022. *GRE: A Benchmark Suite for Learned Indexes*. https://github.com/gre4index/GRE

[24] Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. 2022. Are Updatable Learned Indexes Ready? *Proceedings of the VLDB Endowment* 15, 11 (2022), 3004–3017. doi:10.14778/3551793.3551848

[25] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288. doi:10.14778/3457390.3457393

[26] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *arXiv preprint arXiv:2302.06873* (2023).