The Evolution of Tree-Based Indexing: From Static Structures to Adaptive Traversal

Nimisha A. Modi
Assistant Professor, Department of Computer Science, Veer Narmad South Gujarat University, Surat.
doi.org/10.64643/IJIRTV12I8-189997-459

***Abstract*—Tree-based index structures such as B-Trees and B+ Trees serve as the backbone of database indexing due to their robustness and predictable performance, yet their root-to-leaf traversal strategies remain largely static despite increasing workload skew and evolving hardware architectures. Recent advances in compression-aware indexing, persistent memory optimization, and learned index models have improved space efficiency and access latency, but they either focus on node-level optimizations or attempt to replace traditional structures entirely, often at the cost of system reliability and adaptability. This paper presents a comprehensive survey of these developments and identifies static traversal logic as an underexplored performance bottleneck in modern database systems. To address this limitation, this paper conceptually proposes an adaptive traversal model that augments a traditional B+ Tree with a lightweight, workload-aware learning component to predict likely leaf destinations. The model employs confidence-based routing with safe fallback to standard traversal. The study highlights how such an approach can reduce CPU overhead and tail latency, offering a promising direction for intelligent indexing.**

***Index Terms*—Adaptive Traversal, B -Tree, B+ Tree, Database Indexing, Learned Indexes**

I. INTRODUCTION

Indexing remains a fundamental component of database management systems (DBMS), enabling efficient query processing and reducing costly disk I/O operations. Among classical indexing structures, the B-Tree and its widely adopted variant, the B+ Tree, have dominated database systems for decades due to their balanced trade-off between search efficiency, update performance, and disk-oriented design. However, as data volumes continue to grow and hardware architectures evolve, an inherent limitation has become increasingly evident: traversal strategies in these structures remain largely static. Traditional root-to-leaf descent is inherently data-agnostic, offering consistent behavior but failing to adapt to workload skew, hardware-specific performance characteristics, or dynamic query access patterns.

Recent research efforts have attempted to modernize indexing mechanisms through compression-aware designs, flash-optimized node layouts, and adaptations for persistent memory technologies. In parallel, the emergence of learned index models has demonstrated the feasibility of leveraging machine learning to approximate data distributions and guide search operations. Despite these advancements, a critical gap persists. Much of the existing work either seeks to replace classical tree structures—often at the expense of reliability and robustness—or focuses on localized, node-level optimizations. As a result, the core traversal logic governing movement through index structures remains largely unexamined, leading to redundant pointer dereferences, excessive CPU overhead, and branch mispredictions, particularly in high-concurrency environments.

This paper addresses the limitations of static traversal by synthesizing recent research into a hybrid path-optimization perspective that combines traditional tree reliability with learning-based adaptivity. It presents a comparative analysis of B-Tree and B+ Tree architectures with respect to hardware sensitivity and reviews key advances in compression, reliability, and learned indexing. Based on these insights, the paper proposes a conceptual AI-driven traversal framework that uses lightweight learned approximations to selectively shortcut search paths while preserving the correctness and structural integrity of the B+ Tree.

II. FUNDAMENTALS OF TREE-BASED INDEXING

Database management systems primarily rely on balanced tree structures that provide predictable logarithmic performance, yet their design choices dictate their suitability for different hardware and workload environments.

A. B-Tree and B+ Tree Architectures

The B-Tree is a balanced M-way tree where both keys and data records are stored within internal and leaf nodes. This architecture allows for slightly faster point queries if the target key is located near the root, but it results in lower fan-out and inconsistent search paths. In contrast, the B+ Tree—the industry standard for disk-resident storage—relegates all data records to the leaf nodes, using internal nodes solely as navigational guides. This separation increases the fan-out, thereby reducing the total tree height. Furthermore, B+ Trees utilize linked leaf nodes to facilitate efficient linear scans, making them superior for range-based queries and clustering index implementations.

Despite these structural differences, both models share a static traversal behavior. Regardless of frequency or access patterns, every query must undergo a uniform root-to-leaf descent. This navigation logic ensures consistency but ignores the potential for path optimization in skewed or high-throughput workloads. Table I compares the basic characteristics of B Tree and B+ Tree structures.

Table I. Key Features of B-Tree and B+ Tree Indexes

Feature	B-Tree	B+ Tree
Key Placement	Internal & Leaf nodes	Leaf nodes only
Range Queries	Less efficient	Highly efficient
Fan-out	Lower	Higher
Space Utilization	Moderate	Efficient (Compression)
CPU Efficiency	Low (Branching)	Moderate
Workload Type	Point queries	Range / General purpose

B. Structural Limitations of Conventional Traversal

While structural optimizations like leaf compression [4][14] and reliability checkpoints [17] have improved node-level performance, the logic governing the movement between nodes remains a bottleneck. Three primary limitations persist:

Workload Insensitivity: Classical traversal logic treats *hot* (frequently accessed) and *cold* data identically. Popular keys incur repeated, redundant full traversals from the root, wasting valuable CPU cycles.

Architectural Bottlenecks: Static pointer-chasing increases cache misses and branch mispredictions. In modern multi-core systems, the overhead of navigating multiple internal levels can exceed the actual cost of data retrieval.

Static Path Latency: As trees grow in depth to accommodate massive datasets, the tail latency ($P99$) increases linearly with the number of levels, regardless of the predictability of the query distribution.

These constraints indicate that improving indexing performance requires rethinking traversal logic itself, rather than relying solely on further structural optimizations.

III. LITERATURE STUDY

Current research shows that index design now goes beyond simple structural changes, focusing on hardware awareness, data distribution, and space efficiency.

A. Structural and Compression- Advancements

Traditional B+ Trees have been augmented to handle massive datasets where the memory footprint is a primary bottleneck. Research by Sun et al. [14] and Gao et al. [4] provides a comparative analysis of B-Tree compression techniques, highlighting a critical trade-off: while the storage footprint is reduced, the CPU decompression cost can increase point-query latency, particularly in high-throughput environments. Furthermore, Amato et al. [2] explored the effectiveness of static indexes in memory-constrained environments, providing a benchmark for structural optimization without the overhead of machine learning.

B. Hardware-Aware and Reliability Enhancements

As storage media evolves from standard SSDs to Persistent Memory (PMem) and parallel computing architectures, indexing logic has become "hardware-sensitive." He et al. [5] and Che et al. [15] focus on the asymmetric read/write costs of PMem, proposing structures that minimize "write-amplification" to extend hardware lifespan. Additionally, Kim et al. [7] introduced cascade memory nodes to transform random writes into sequential ones for flash devices, while Yu et al. [17] addressed system reliability by proposing fault-tolerant enhancements to ensure consistency during high-concurrency operations.

C. Learned Indexes

The most significant shift in recent years is the move from data-agnostic structures to data-aware models. Kraska et al. [8] pioneered the "Learned Index," treating the index as a regression problem to predict key positions by approximating the data's Cumulative Distribution Function (CDF). To address the static nature of early models, ALEX [3] and LIPP [16] introduced updatable learned structures. However, Liu et al. [10] concluded that while these excel on stable distributions, they are highly sensitive to "model drift" and data volatility, often requiring a robust fallback.

Table II. Summary of major research areas and key contributions in database indexing.

Category / Primary Focus	Key Contribution	References
Structural / Compression (Space Efficiency)	Analyzed CPU–latency trade-offs in compressed B-Tree structures	Gao [4], Sun [14], Amato [2]
Hardware-Aware (PMem & Flash)	Optimized write paths and reduced tail latency for non-volatile memory systems	He [5], Che [15], Kim [7]
Learned Models (Data Awareness)	Applied CDF-based models to predict key positions in sorted data	Kraska [8], Ding [3], Wu [16]
Evaluation / Survey (Benchmarking)	Identified model drift as a primary failure point in learned index structures	Liu [10], Liu [11], Abbasi [1]
Reliability (Fault Tolerance)	Developed integrity and consistency checks for high-concurrency B+ Tree structures	Yu [17]

Table II summarizes the main research areas in database indexing based on their primary focus and technical contributions.

D. Identification of Research Gaps

Through a systematic review of the literature, three primary gaps have been identified:

Static Path Navigation: While structural optimizations (compression [4][14]) and hardware adaptations (PMem/SSD [5][7]) improve node performance, the logic used to navigate from the root to the leaf remains static and ignores workload skew.

Lack of Traversal Adaptivity: Most hybrid learned indexes focus on replacing tree nodes with models. There is a lack of research into adaptive path guidance—where the index maintains its classical structure for reliability but uses AI to shortcut the process dynamically.

Hardware-AI Disconnect: Learned models excel in memory but struggle with the physical constraints of storage hardware. There is a need for a framework that integrates AI-driven prediction with hardware-specific requirements like tail-latency control and fault tolerance.

IV. PROPOSED ADAPTIVE TRAVERSAL MODEL

Building upon the gaps identified in the literature, this section introduces a theoretical framework for an AI-driven adaptive traversal mechanism. The model is designed to transition from static, data-agnostic pointer chasing to a dynamic, workload-aware strategy.

A. System Design Objectives

The primary objective of the proposed model is to augment, rather than replace, the foundational B+ Tree structure. By maintaining the physical tree, the system ensures data integrity and structural reliability [17]. A critical design constraint is the minimization of computational overhead; the AI components must be sufficiently lightweight to ensure that model inference is significantly faster than the multiple memory accesses required by standard root-to-leaf traversal.

B. Architecture and Components

The architecture (Fig. 1) consists of four distinct layers that coordinate to optimize the search path.

Workload Analyzer: Continuously monitors incoming queries to detect frequency patterns, temporal locality, and distribution skew.

*Feature Abstrac*tion: Extracts high-level signals such as hot key identification and key-space density.
Learning Model: A lightweight predictor—such as a piecewise linear regression—trained to approximate the data distribution and predict the most likely leaf node for a given key.
Decision Layer: The adaptive traversal begins by checking if the model's confidence level C exceeds the reliability threshold τ. If $C > \tau$, the system bypasses the internal levels of the tree and jumps directly to a predicted leaf. Otherwise, it defaults to standard B+ Tree logic. To account for minor model inaccuracies, a local scan is performed within a defined error bound ε. If the key is not located within this range, or if the initial confidence was insufficient, the system initiates the fallback layer, performing a standard root-to-leaf descent to guarantee the query's success.

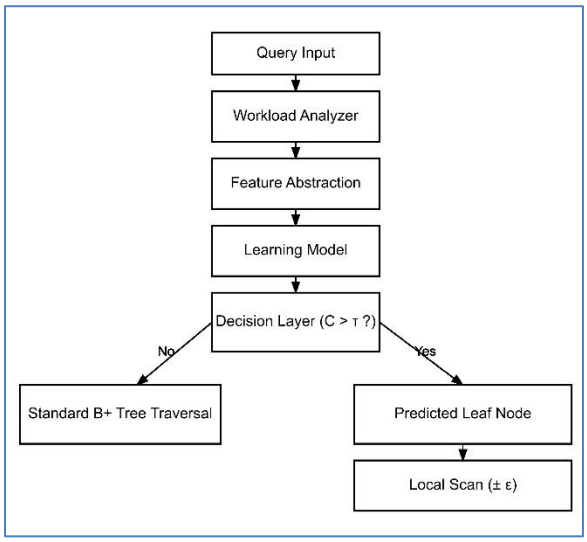

Fig. 1. Architecture of Adaptive Index Traversal

C. Optimization and Reliability Strategies
The model employs a multi-faceted strategy to balance performance with hardware and concurrency constraints:
Confidence-Based Routing: Draws on hybrid principles [3][8] to ensure shortcuts are only taken when the probability of success is high.
Tiered Storage Optimization: Implements hot/cold tiering where frequently accessed paths are kept in high-speed cache with minimal compression, while cold nodes are heavily compressed.
Hardware Alignment: Aligns write patterns with the specific needs of Flash SSDs and Persistent Memory to minimize tail latency [7][15].
Safety Protocols: Employs background reliability checks [17] to ensure AI shortcuts never compromise underlying data consistency.

D. Expected Benefits and Trade-offs
The proposed model is expected to significantly reduce median latency by eliminating redundant CPU cycles for hot data paths. By substituting unpredictable branch logic with deterministic mathematical approximations, the system can tighten the 99th percentile tail latency. However, a fundamental trade-off exists: the system must manage the memory and CPU overhead of the Workload Analyzer and model inference. The success of the model depends on ensuring these costs remain lower than the time saved by bypassing the traditional B+ Tree pointers.

V. RESEARCH CHALLENGES AND FUTURE DIRECTIONS

The transition from a theoretical AI-driven traversal model to a production-ready database engine involves several non-trivial challenges. These obstacles stem from the dynamic nature of real-world data and the strict performance requirements of modern storage architectures.

A. The Challenge of Model Drift and Retraining
The most significant challenge for any learned indexing strategy is model drift, which occurs when the underlying data distribution changes significantly enough to render the AI's approximations inaccurate [10]. In the proposed adaptive model, drift causes a drop in the confidence score C, forcing the system to revert to the standard B+ Tree traversal. While this ensures accuracy, frequent fallbacks eliminate the performance benefits of the AI shortcut. Future research must investigate online learning techniques that allow the model to incrementally retrain in the background without locking the index or consuming excessive CPU resources. Furthermore, by incorporating versioned optimistic locking, the model can ensure thread safety and consistency even during aggressive structural modifications.

B. The Cold-Start and Overhead Problem
A secondary challenge is the cold-start problem, where the Workload Analyzer has not yet gathered enough telemetry to generate a reliable model. During this phase, the system incurs the overhead of monitoring query patterns without reaping the

rewards of faster traversal. Balancing the cost of learning against the gain in speed is essential. Developers must determine the optimal granularity for feature abstraction to ensure that the Workload Analyzer does not itself become a bottleneck that increases, rather than decreases, total latency.

C. Hardware Generalization and Integration Complexity

Modern databases run on diverse hardware, ranging from cloud-native NVMe (Non-Volatile Memory Express) drives to PMem (Persistent Memory) arrays. Generalizing an AI-driven traversal model across these media is complex, as the cost of a cache miss or a pointer dereference varies significantly between devices [11]. Furthermore, integrating this logic into mature, legacy DBMS engines like PostgreSQL or MySQL presents significant engineering hurdles [13]. Future directions should focus on creating transparent middleware or pluggable indexing APIs that allow adaptive traversal to be toggled based on the specific hardware environment.

D. Future Directions: Autonomous Optimization

Looking forward, the integration of Reinforcement Learning (RL) offers a promising path for real-time path optimization [13]. Rather than using static thresholds τ, an RL agent could autonomously adjust traversal strategies based on shifting performance targets, such as prioritizing power efficiency during low-traffic periods or minimizing tail latency during peak bursts.

Additionally, the rise of distributed and cloud-native databases presents an opportunity to apply adaptive traversal at scale. The development of standardized benchmarking frameworks specifically designed for adaptive, hardware-aware traversal is necessary to provide an objective comparison between classical and AI-augmented systems [18].

VI. CONCLUSION

This paper has provided a comprehensive survey of the evolution of B-Tree and B+ Tree indexing, tracing the progression from classical structural optimizations to modern, hardware-aware, and AI-driven strategies. Through a systematic review of current literature, it is evident that while significant advancements have been made in index compression, reliability, and persistent memory adaptation, the fundamental logic of root-to-leaf traversal has remained largely static. This lack of adaptivity represents a critical performance bottleneck in high-concurrency environments where workload skew and hardware sensitivity dominate.

By integrating a lightweight workload analyzer with a confidence-based decision layer, the model offers a mechanism to shortcut traditional pointer-chasing paths without compromising the structural integrity or reliability of the underlying B+ Tree. The analysis suggests that such a hybrid approach can significantly tighten 99th percentile tail latency and improve CPU efficiency by substituting unpredictable branch logic with deterministic mathematical approximations.

While challenges regarding model drift, cold-start overhead, and hardware generalization remain, the shift toward autonomous, workload-aware indexing is inevitable. Future work must focus on the empirical validation of these theoretical shortcuts and the development of seamless integration strategies for existing database engines.

REFERENCES

[1] Abbasi, M., Bernardo, M. V., Váz, P., Silva, J., & Martins, P. (2024). Revisiting Database Indexing for Parallel and Accelerated Computing: A Comprehensive Study and Novel Approaches. Information, 15(8), Article 429. https://doi.org/10.3390/info15080429

[2] Amato, D., Giancarlo, R., & Lo Bosco, G. (2023). Learned sorted table search and static indexes in small-space data models. Data, 8(3), Article 56. https://doi.org/10.3390/data8030056

[3] Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D., & Kraska, T. (2020). ALEX: An updatable adaptive learned index. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (pp. 969–984). ACM. https://doi.org/10.1145/3318464.3389711

[4] Gao, C., Ballijepalli, S., & Wang, J. (2024). Revisiting database index compression: A comparative analysis of B-tree techniques. Proceedings of the ACM on Management of Data. https://doi.org/10.1145/3654972

[5] He, X., Yang, C., Zhang, R., Luo, H., Cao, Z., & Zhang, J. (2025). Optimizing both performance and tail latency for B+-tree on

persistent memory. Journal of Systems Architecture, Article 163, 103406. https://doi.org/10.1016/j.sysarc.2025.103406

[6] Karri, N., Jangam, S. K., & Muntala, P. S. R. (2023). AI-driven indexing strategies. International Journal of AI-Based Data and Cloud Management Systems, 4(2), 111–119. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V4I2P112

[7] Kim, B.-K., Kim, G.-W., & Lee, D.-H. (2020). A Novel B-Tree Index with Cascade Memory Nodes for Improving Sequential Write Performance on Flash Storage Devices. Applied Sciences, 10(3), Article 747. https://doi.org/10.3390/app10030747

[8] Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018). The case for learned index structures. In Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (pp. 489–504). ACM. https://doi.org/10.1145/3183713.3196909

[9] Li, G., Zhou, X., & Cao, L. (2021). Machine learning for database systems: Opportunities and challenges. ACM Computing Surveys, 55(1), Article 1. https://doi.org/10.1145/3486001.3486248

[10] Liu, Q., Han, S., Qi, Y., Peng, J., Li, J., Lin, L., & Chen, L. (2025). Why are learned indexes so effective but sometimes ineffective? Proceedings of the VLDB Endowment, 18(9), 2886–2898. https://doi.org/10.14778/3746405.3746415

[11] Liu, Q., Li, M., Zeng, Y., Shen, Y., & Chen, L. (2025). How good are multi-dimensional learned indexes? An experimental survey. VLDB Journal, 34(17). https://doi.org/10.1007/s00778-024-00893-6

[12] Modi, N., & Patel, J. (2025). Evaluating query processing architectures in relational and document databases. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 12(2), 2732–2736.

[13] Qiao, S.-J., Fan, H.-L., Han, N., Du, L., Peng, Y.-H., Tang, R.-M., & Qin, X. (2025). Learning database optimization techniques: The state-of-the-art and prospects. Frontiers of Computer Science, 19, 1912612. https://doi.org/10.1007/s11704-025-41116-7

[14] Sun, S., Gao, C., Ballijepalli, S., & Wang, J. (2026). An evaluation of B-tree compression techniques. VLDB Journal, 35(4). https://doi.org/10.1007/s00778-025-00950-8

[15] Che, W., Chen, Z., Hu, D., Sun, J., & Chen, H. (2024). ZBTree: A fast and scalable B++-Tree for persistent memory. IEEE Transactions on Knowledge and Data Engineering, 36(12), 9547–9563. https://doi.org/10.1109/TKDE.2024.3421232

[16] Wu, J., Zhang, Y., Chen, S., Wang, J., Chen, Y., & Xing, C. (2021). Updatable learned index with precise positions. Proceedings of the VLDB Endowment, 14(8), 1276–1288. https://doi.org/10.14778/3457390.3457393

[17] Yu, Y., Tang, J., Zhang, H., & Zhang, H. (2024). Research on reliability improvement and index structure enhancements for B+-tree. ACM Transactions on Database Systems. https://doi.org/10.1145/3712335.3712409

[18] Zhou, X., Hao, X., Yu, X., & Stonebraker, M. (2025). Tiered-indexing: Optimizing access methods for skew. VLDB Journal, 34(45). https://doi.org/10.1007/s00778-025-00928-6