American Journal of Engineering Research (AJER)
e-ISSN: 2320-0847 p-ISSN: 2320-0936
Volume-15, Issue-5, pp-52-59
www.ajer.org
Research Paper Open Access

Operator-Aware Layout Selection for Exact Segment Trees Under Nonuniform Merge Costs

Nikoloz Svanidze
[1]PhD student, Faculty of Informatics and Control Systems, Georgian Technical University

ABSTRACT : *Exact segment trees usually split intervals at midpoints, a design that is simple and robust but implicitly treats merge operations as nearly uniform. This assumption can be weak for exact summaries such as sparse matrix products, compressed function composition, and frequency-map aggregation, where the cost of combining two summaries depends on their contents. This paper studies Operator-Aware Layout Selection for exact segment trees under nonuniform merge costs. The proposed Operator-Aware segment-tree approach changes only the tree layout and split points; it does not approximate answers, alter the associative operator, or relax left-to-right execution for noncommutative monoids. Candidate splits are selected from midpoint, quantile, workload-boundary, heat-median, complexity-change, and random-probe sources, then scored using estimated merge cost, routing cost, balance penalties, runtime calibration, and validation-based portfolio selection. A primary full-builder evaluation covers $n \leq 288$, and a selected medium-scale full-builder check covers $n = 512$ for sparse matrix and function composition workloads. Larger $n = 512$ and $n = 1024$ experiments use a lower-overhead fast-layout path, so they support mechanism and scale-direction evidence rather than production-scale learned_portfolio claims. Across the studied synthetic and trace-inspired workloads, operator-aware layouts often reduce latency over balanced recursive trees for stable content-dependent merges. However, iterative trees often win on cheap or cache-friendly cases, Fenwick remains the recommended sum baseline, learned_portfolio can require large amortization, and workload shift can erase or reverse gains.*
KEYWORDS: *segment tree; range queries; monoids; learned layout selection; workload-aware optimization; nonuniform merge cost*

Date of Submission: 05-05-2026 Date of acceptance: 16-05-2026

I. INTRODUCTION

Segment trees decompose an array interval into stored subinterval summaries and combine them to answer exact range queries. The classic balanced layout is attractive because it has predictable height, simple construction, and good worst-case behavior. For cheap operators, these properties are often more important than any learned split choice. For content-dependent operators, however, the midpoint split can expose expensive summary pairs even when the query workload is highly nonuniform.

This paper studies exact segment-tree layout selection under workload and operator-cost signals. The contribution is not a replacement for all segment-tree implementations. It is a layout-selection method for associative range aggregation when merge costs are nonuniform, the workload is reasonably stable, and enough queries are issued to amortize layout construction.

The paper gives an exactness proof sketch and reports controlled empirical evidence. It does not claim broad real-world superiority, nor does it claim production-scale performance for the full learned_portfolio builder. The most important boundary is summarized in Table I.

Table I: Claim boundary.

Category	Statement
Proved	Exactness for associative monoids, including noncommutative monoids when query execution preserves left-to-right order.
Observed	Latency reductions on selected synthetic and trace-inspired stable workloads with

Category	Statement
	content-dependent merge costs.
Not claimed	Broad real-world superiority, production-scale learned_portfolio performance, superiority for cheap/uniform operators, or robustness under arbitrary workload shift.
Deployment implication	Use OAST selectively; prefer Fenwick for sum, iterative for cheap/cache-friendly operators, and balanced/iterative fallback under shift or small batches.

Literature Review and positioning

Classical range-query data structures include segment trees, Fenwick trees, sparse tables, and geometric range-searching structures [1]-[6]. Fenwick trees are especially strong for prefix-invertible operations such as sum [4], which is why this paper treats sum as a control rather than as the natural target for OAST. Optimal binary search trees, optimal alphabetic trees, and related ordered tree construction algorithms optimize search depth or code length under ordered access probabilities [7], [8], [24]-[26]. OAST is adjacent to this tradition but optimizes exact interval-cover aggregation cost, not only comparison depth.

Learned indexes and learned data layouts use models to predict key positions, organize multidimensional data, or improve block locality [9]-[11], [19]-[22], [32]. Recent work has strengthened this area with concurrent and dynamic learned indexes, index-benefit prediction, comprehensive learned-index evaluation, learned space-filling curves, and multidimensional learned-index surveys [33]-[38]. Adaptive indexing, query-optimizer physical design, and access-method design similarly use workload signals to choose storage or access paths [12]-[14], [27], [31]. Cache-aware and cache-oblivious structures show that layout alone can strongly affect constants and locality [15]-[18], [28]-[30]. These lines motivate workload-aware physical choices, but they do not directly solve the exact associative range-aggregation setting considered here.

Why OAST is not a learned index

OAST does not learn key rank, skip tuples, approximate answers, or change the array order. It preserves exact associative aggregation and chooses binary split points for stored interval summaries. For noncommutative monoids, query execution must combine covered intervals in left-to-right order. The optimization target is therefore content-dependent merge exposure in an exact interval-cover tree, rather than search-depth minimization, tuple placement, or block pruning.

II. MATERIALS AND METHODS

Algebraic setting

Let $A[1..n]$ be an array over a monoid (S, combine, e), where combine is associative and e is the identity element. For an interval [l,r], the exact summary is A[l] combine A[l+1] combine ... combine A[r]. A layout is a binary tree whose leaves, in order, are the array elements and whose internal nodes store exact summaries of contiguous intervals. A balance parameter alpha constrains each split so $alpha \leq left\_size / parent\_size \leq 1 - alpha$.

The objective is to minimize expected query time under a training workload distribution, while retaining exactness. Practical scoring also includes balance penalties, routing estimates, runtime calibration, optional robustness probes, and validation selection. The reported experiments use static arrays and batched query workloads. Update time is secondary instrumentation only; the experiments do not optimize update-heavy workloads.

OAST construction and validation

Candidate split sources include the midpoint, quantiles, workload boundaries, heat-weighted medians, complexity-change points, and seeded random probes. Each legal split is scored by predicted local merge cost, future routing cost, a balance penalty, and optional memory, update, or robustness terms. The validation portfolio builds several candidate layouts and selects using validation queries and shifted probes where applicable. Test queries are never used for model fitting, validation, or layout selection.

Runtime prediction model

The learned component is a lightweight log-linear regression cost model, not a neural network and not a learned index. It is trained by online stochastic gradient descent on 1600 sampled summary pairs for 24 epochs, learning rate 0.035, batch size 1, and no regularization. Features are log-transformed: $log1p$(left summary size), $log1p$(right summary size), $log1p$(size product), $log1p$(estimated operator cost), a bias term, and a left-larger indicator. The model predicts merge/combine cost for split scoring only. Its prediction is blended with analytical estimated cost, and it is nonessential to correctness because exact answers follow from exact stored summaries and associative left-to-right combination.

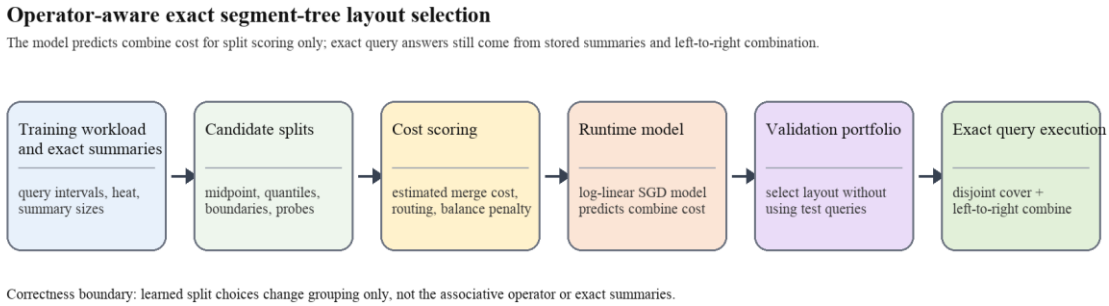

Figure 1: OAST construction and exact-query execution workflow.

Correctness and complexity

Every node stores the exact summary of its interval. A range query decomposes [l,r] into disjoint stored intervals and combines their exact summaries in array order. Associativity makes the grouping irrelevant, and left-to-right execution preserves correctness for associative but noncommutative operators. Thus learned split choices change only grouping and routing, not semantics.

For static construction, memory is $O(n)$ nodes and build time is dominated by summary construction plus split scoring and optional calibration. The balance constraint bounds height by $O(log\ n\ /\ log(1/(1-alpha)))\ in$ the worst case. Query time depends on tree height, the number of covered intervals, and the actual content-dependent merge cost. Full learned_portfolio construction is substantially more expensive than balanced or iterative construction, so amortization is a deployment constraint.

III. RESULTS AND DISCUSSION

The empirical evidence is intentionally separated into tiers. Full OAST-family results support statements about learned_runtime and learned_portfolio only at the scales where those builders were actually run. Fast-layout results are useful for mechanism and scale-direction evidence, but they are not evidence of production-scale learned_portfolio performance.

Table II: Evidence tiers and supported claims.

Experiment	Scale and repeats	Builder family	Supported claim
Primary small-scale evaluation	sizes 72, 96, 192, 288; seeds 2; repeats 1 after warm-up	portfolio yes; runtime yes; fast path no	Supports claims about full learned_runtime and learned_portfolio at small-scale sizes only.
Medium-scale full-builder check	sizes 512; seeds 3; repeats 3 after warm-up	portfolio yes; runtime yes; fast path no	Checks selected n=512 full-builder behavior; it is not a production-scale portfolio claim.
Strengthened five-seed key study	sizes 192; seeds 5; repeats 5 after warm-up	portfolio no; runtime no; fast path yes	Supports mechanism plausibility and paired comparisons for a lower-overhead approximation.
Larger-size fast-layout study	sizes 512, 1024; seeds 3; repeats 3 after warm-up	portfolio no; runtime no; fast path yes	Supports scale-direction evidence only; it does not prove full learned_portfolio production performance.
Robustness and workload-shift study	sizes 192; seeds 3; repeats 3 after warm-up	portfolio no; runtime no; fast path yes	Supports failure-mode and fallback-selection evidence under distribution shift.
Oracle comparison	sizes 72; seeds 2; repeats 1 after warm-up	portfolio yes; runtime yes; fast path no	Checks whether learned heuristics move toward the dynamic-program optimum at small n.

Primary small-scale full-builder results

The primary full-builder evaluation uses $n \leq 288$ and two seed families. These runs include learned_runtime and learned_portfolio, so they support only small-scale full-builder claims. Paired statistics from this small sample are exploratory, not definitive population-level inference.

Table III: Selected primary full-builder results (average microseconds per query).

Scenario	Balanced	Iterative	Best OAST-family	Interpretation
sum; uniform; uniform; n=192	4.30 +/- 0.07	3.69 +/- 0.23	learned: 4.05 +/- 0.08	B/OAST 1.06; B/Iter 1.17; best overall fenwick.
sparse matrix; uniform; uniform; n=192	258.09 +/- 21.19	260.14 +/- 27.57	learned_runtime: 158.81 +/- 7.36	B/OAST 1.63; B/Iter 0.99; best overall learned_runtime.
sparse matrix; hot expensive; hot; n=192	1142.19 +/- 69.00	1061.69 +/- 24.10	learned_runtime: 970.67 +/- 37.73	B/OAST 1.18; B/Iter 1.08; best overall learned_runtime.
sparse matrix; sparse to dense; expensive crossing; n=192	671.07 +/- 87.36	571.47 +/- 90.91	learned_portfolio: 549.39 +/- 13.42	B/OAST 1.22; B/Iter 1.17; best overall learned_portfolio.
function comp.; hot expensive; hot; n=192	87.50 +/- 3.43	79.79 +/- 3.38	learned_portfolio: 75.53 +/- 2.33	B/OAST 1.16; B/Iter 1.10; best overall learned_portfolio.

Scenario	Balanced	Iterative	Best OAST-family	Interpretation
frequency map; hot expensive; hot; n=192	5.68 +/- 0.21	4.97 +/- 0.22	learned_runtime: 5.13 +/- 0.24	B/OAST 1.11; B/Iter 1.14; best overall iterative.

The table shows the main pattern. Learned-family layouts improve over balanced recursive trees on several content-dependent workloads, including sparse matrix product and hot function composition. The sum control is different: Fenwick is the best overall structure, so OAST should not be recommended for ordinary prefix-invertible sum queries. Iterative trees also win in several cheap or cache-friendly cases.

Medium-scale full-builder check

A selected n = 512 check was completed for two representative content-dependent workloads. This is the strongest additional evidence for the full builders beyond $n \leq 288$, but it remains selected and small. It should be read as a feasibility check, not a broad scale claim.

Table IV: Medium-scale full-builder check at n = 512.

Scenario	Seeds	Baselines	Best full builder	Build and amortization
function_composition/hot_expensive/hot/n=512	3	balanced 122.75 +/- 6.94; iterative 112.13 +/- 8.75	learned_runtime: 110.65 +/- 1.79	build 1850.69 ms; break-even 152951 vs balanced and 1249477 vs iterative; Iter/OAST 1.01.
sparse_matrix/hot_expensive/hot/n=512	3	balanced 1821.38 +/- 86.63; iterative 1796.96 +/- 96.69	learned_runtime: 1514.48 +/- 64.63	build 21426.16 ms; break-even 69815 vs balanced and 75850 vs iterative; Iter/OAST 1.19.

Both n = 512 checks selected learned_runtime as the best full builder. Sparse matrix product improved by about 1.20x over balanced and 1.19x over iterative, but required about 21.4 seconds of build time, giving break-even counts around 69,815 queries versus balanced and 75,850 versus iterative. Function composition improved over balanced but only slightly over iterative, with a much larger break-even versus iterative. Query-latency improvement alone is therefore insufficient when build cost is counted.

Large-size fast-layout and workload-shift boundary

The larger n = 512 and n = 1024 runs use a lower-overhead fast-layout path. They suggest that operator-aware layout choices can remain useful at larger sizes, but they do not validate full learned_portfolio performance at those sizes. Shift experiments use the same boundary: they show failure modes and possible recovery, not a solved deployment policy.

Table V: Scale-direction and workload-shift summary.

Case	Tier	Selected layout	Baselines	Interpretation
frequency_map/block_structured/trace/n=1024	large fast-layout	oast_fast_portfolio: 6.55 +/- 0.32	balanced 7.62 +/- 0.22; iterative 6.75 +/- 0.16	B/best 1.16; Iter/best 1.03; scale-direction evidence only.
function_composition/hot_expensive/hot/n=1024	large fast-layout	oast_fast_portfolio: 127.37 +/- 10.99	balanced 140.42 +/- 4.66; iterative 123.33 +/- 4.60	B/best 1.10; Iter/best 0.97; scale-direction evidence only.

Case	Tier	Selected layout	Baselines	Interpretation
sparse_matrix/hot_expensive/hot/n=1024	large fast-layout	oast_fast: 2063.17 +/- 27.36	balanced 2281.31 +/- 40.47; iterative 2242.92 +/- 36.73	B/best 1.11; Iter/best 1.09; scale-direction evidence only.
sparse_matrix hot->hot_far	severe shift	operator_greedy_fast: 66.39	balanced 68.26; iterative 35.50	winner iterative; portfolio recovered no.
sparse_matrix hot->uniform	moderate shift	oast_fast_portfolio: 338.70	balanced 395.04; iterative 271.32	winner iterative; portfolio recovered yes.
sparse_matrix uniform->hot	moderate shift	oast_fast_portfolio: 1099.30	balanced 1204.20; iterative 1112.32	winner oast_fast_portfolio; portfolio recovered yes.
function_composition short->long	severe shift	oast_no_future: 45.89	balanced 47.25; iterative 36.77	winner iterative; portfolio recovered no.

Workload shift is central. OAST can fail when training and test query distributions diverge, and a balanced or iterative fallback may be the right choice. The artifact also includes a simple fallback/relayout policy study, but it should be treated as a preliminary mitigation rather than as proven robustness under arbitrary shifts.

Practical deployment guidance

Table VI: When to use OAST.

Case	Recommended structure/layout	Reason	Evidence source
Sum / invertible prefix operation	Fenwick or domain-specific prefix structure	Prefix structures dominate cheap invertible aggregation and remain the recommended sum baseline.	sum control and Fenwick baseline
Cheap or cache-friendly operator	iterative segment tree	Lower constants and compact array layout usually dominate learned split choices.	primary small-scale evaluation
Stable workload + expensive content-dependent merge + enough queries to amortize build	learned_runtime or operator-aware layout	Use when validation shows query savings and build cost is acceptable.	primary small-scale and medium-scale full-builder checks
Large stable batch where validation cost is acceptable	learned_portfolio	Use only when construction/validation cost can be amortized and test queries remain held out.	amortization tables
Small batch or frequent updates	balanced or iterative	Avoid build cost and stale layouts; update-heavy workloads are not optimized here.	static-focused update scope
Workload shift likely	balanced/iterative fallback or robust validation/relayout	Shift can erase or reverse learned-layout gains.	robustness and workload-shift study

Case	Recommended structure/layout	Reason	Evidence source
Noncommutative associative operator	OAST is valid only with left-to-right query execution	Exactness relies on associativity plus order preservation, not commutativity.	correctness section

The most defensible use case is a static or rarely updated array, stable query distribution, expensive content-dependent combine operation, and enough queries to amortize learned construction. The safest non-use cases are cheap uniform operators, small query batches, frequent updates, or settings where a specialized structure such as Fenwick is available.

Brute-force correctness checks passed for all reported configurations in the generated correctness CSVs. The checker reports 24 correctness files and zero failures, including the primary result CSV, medium-scale full-builder CSV, fast-layout CSVs, alpha/ablation runs, and workload-shift runs.

Limitations and threats to validity

The main threat is empirical scale. The full OAST-family study is still centered on small sizes, with a selected n = 512 full-builder check. Larger results use a fast-layout path and should not be treated as production-scale evidence for learned_portfolio. Most workloads are synthetic or trace-inspired rather than public production traces. Timing and paired statistical tests are exploratory where seed counts are small. Python implementation constants may favor or penalize baselines. Finally, robustness under arbitrary workload shift is not established.

IV. CONCLUSION

OAST shows that exact segment-tree layouts can be chosen using workload and operator-cost signals without changing associative semantics. The proof obligation is clean: exactness follows from exact node summaries, disjoint interval covers, associativity, and left-to-right order for noncommutative monoids. Across the studied synthetic and trace-inspired workloads, operator-aware layouts often reduced latency for expensive, content-dependent operators.

The conclusion is deliberately narrow. OAST is promising under stable workloads with expensive nonuniform merges, but it is not a general replacement for balanced or iterative segment trees. Fenwick remains the recommended baseline for sum, iterative trees often win when constants and cache behavior dominate, learned_portfolio can require large amortization, and workload shift can erase or reverse gains. Larger-scale, lower-overhead, and more realistic experiments are needed before claiming broad practical superiority.

ACKNOWLEDGEMENT AND DISCLOSURE STATEMENTS

Conflict of Interest: The author declares no known conflict of interest. Funding: No external funding is reported for this work. Human/Animal Subjects: This work uses no human or animal subjects. Originality: The manuscript is presented as original work and should not be submitted elsewhere simultaneously. Data and Code Availability: https://github.com/svanidzen-gtu/oast-ajer-artifact

REFERENCES

[1]. J. L. Bentley, ``Solutions to Klee's rectangle problems,'' Technical Report, Carnegie Mellon University, 1977.
[2]. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
[3]. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, Computational Geometry: Algorithms and Applications, 3rd ed. Berlin, Germany: Springer, 2008.
[4]. P. M. Fenwick, ``A new data structure for cumulative frequency tables,'' Software: Practice and Experience, vol. 24, no. 3, pp. 327--336, 1994, doi: 10.1002/spe.4380240306.
[5]. A. C. Yao, ``Space-time tradeoff for answering range queries,'' in Proc. 14th ACM Symposium on Theory of Computing, 1982, pp. 128--136.
[6]. B. Chazelle, ``Lower bounds for orthogonal range searching: I. The reporting case,'' Journal of the ACM, vol. 37, no. 2, pp. 200--212, 1990.
[7]. D. E. Knuth, ``Optimum binary search trees,'' Acta Informatica, vol. 1, pp. 14--25, 1971.

[8]. T. C. Hu and A. C. Tucker, ``Optimal computer search trees and variable-length alphabetical codes," SIAM Journal on Applied Mathematics, vol. 21, no. 4, pp. 514--532, 1971.
[9]. T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, ``The case for learned index structures," in Proc. ACM SIGMOD, 2018, pp. 489--504, doi: 10.1145/3183713.3196909.
[10]. J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. Lomet, and T. Kraska, ``ALEX: An updatable adaptive learned index," in Proc. ACM SIGMOD, 2020, pp. 969--984, doi: 10.1145/3318464.3389711.
[11]. P. Ferragina and G. Vinciguerra, ``The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds," Proc. VLDB Endowment, vol. 13, no. 8, pp. 1162--1175, 2020, doi: 10.14778/3389133.3389135.
[12]. S. Idreos, M. L. Kersten, and S. Manegold, ``Database cracking," in Proc. CIDR, 2007, pp. 68--78.
[13]. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, ``Access path selection in a relational database management system," in Proc. ACM SIGMOD, 1979, pp. 23--34.
[14]. G. Graefe, ``Query evaluation techniques for large databases," ACM Computing Surveys, vol. 25, no. 2, pp. 73--169, 1993.
[15]. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, ``Cache-oblivious algorithms," in Proc. 40th IEEE FOCS, 1999, pp. 285--297.
[16]. M. A. Bender, E. D. Demaine, and M. Farach-Colton, ``Cache-oblivious B-trees," SIAM Journal on Computing, vol. 35, no. 2, pp. 341--358, 2005, doi: 10.1137/S0097539701389956.
[17]. J. Rao and K. A. Ross, ``Cache conscious indexing for decision-support in main memory," in Proc. VLDB, 1999, pp. 78--89.
[18]. V. Leis, A. Kemper, and T. Neumann, ``The adaptive radix tree: ARTful indexing for main-memory databases," in Proc. IEEE ICDE, 2013, pp. 38--49.
[19]. V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, ``Learning multi-dimensional indexes," in Proc. ACM SIGMOD, 2020, pp. 985--1000, doi: 10.1145/3318464.3380579.
[20]. J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, ``Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," Proc. VLDB Endowment, vol. 14, no. 2, pp. 74--86, 2020, doi: 10.14778/3425879.3425880.
[21]. Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-A. Larson, D. Kossmann, and R. Acharya, ``Qd-tree: Learning data layouts for big data analytics," in Proc. ACM SIGMOD, 2020, pp. 193--208, doi: 10.1145/3318464.3389770.
[22]. R. Marcus, M. Stoian, A. Kipf, S. Misra, A. van Renen, A. Kemper, T. Neumann, and T. Kraska, ``Benchmarking learned indexes,'' Proc. VLDB Endowment, vol. 14, no. 1, pp. 1--13, 2020, doi: 10.14778/3421424.3421425.
[23]. J. L. Bentley and J. B. Saxe, ``Decomposable searching problems I: static-to-dynamic transformation," Journal of Algorithms, vol. 1, no. 4, pp. 301--358, 1980.
[24]. J. A. G. Garsia and M. L. Wachs, ``A new algorithm for minimum cost binary trees," SIAM Journal on Computing, vol. 6, no. 4, pp. 622--642, 1977.
[25]. E. N. Gilbert and E. F. Moore, ``Variable-length binary encodings," Bell System Technical Journal, vol. 38, no. 4, pp. 933--967, 1959.
[26]. K. Mehlhorn, ``Nearly optimal binary search trees," Acta Informatica, vol. 5, pp. 287--295, 1975.
[27]. J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah, ``Adaptive query processing: technology in evolution," IEEE Data Engineering Bulletin, vol. 23, no. 2, pp. 7--18, 2000.
[28].] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, ``Weaving relations for cache performance," in Proc. VLDB, 2001, pp. 169--180.
[29]. J. S. Vitter, ``External memory algorithms and data structures: dealing with massive data," ACM Computing Surveys, vol. 33, no. 2, pp. 209--271, 2001.
[30]. R. Bayer and E. McCreight, ``Organization and maintenance of large ordered indexes," Acta Informatica, vol. 1, pp. 173--189, 1972.
[31]. M. Athanassoulis and S. Idreos, ``Designing access methods: the RUM conjecture," in Proc. EDBT, 2016, pp. 461--466.
[32]. A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, ``SOSD: A benchmark for learned indexes," in Proc. NeurIPS Workshop on Machine Learning for Systems, 2019, arXiv:1911.13014.
[33]. P. Li, Y. Hua, J. Jia, and P. Zuo, ``FINEdex: A fine-grained learned index scheme for scalable and concurrent memory systems," Proc. VLDB Endowment, vol. 15, no. 2, pp. 321--334, 2021, doi: 10.14778/3489496.3489512.
[34]. Z. Zhang, P.-Q. Jin, X.-L. Wang, Y.-Q. Lv, S.-H. Wan, and X.-K. Xie, ``COLIN: A cache-conscious dynamic learned index with high read/write performance," Journal of Computer Science and Technology, vol. 36, pp. 721--740, 2021, doi: 10.1007/s11390-021-1348-2.
[35]. J. Shi, G. Cong, and X.-L. Li, ``Learned index benefits: Machine learning based index performance estimation," Proc. VLDB Endowment, vol. 15, no. 13, pp. 3950--3962, 2022, doi: 10.14778/3565838.3565848.
[36]. Z. Sun, X. Zhou, and G. Li, ``Learned index: A comprehensive experimental evaluation," Proc. VLDB Endowment, vol. 16, no. 8, pp. 1992--2004, 2023, doi: 10.14778/3594512.3594528.
[37]. J. Gao, X. Cao, X. Yao, G. Zhang, and W. Wang, ``LMSFC: A novel multidimensional index based on learned monotonic space filling curves," Proc. VLDB Endowment, vol. 16, no. 10, pp. 2605--2617, 2023, doi: 10.14778/3603581.3603598.
[38]. Q. Liu, M. Li, Y. Zeng, Y. Shen, and L. Chen, ``How good are multi-dimensional learned indexes? An experimental survey," The VLDB Journal, vol. 34, article 17, 2025, doi: 10.1007/s00778-024-00893-6.