American Journal of Engineering Research (AJER)
e-ISSN: 2320-0847 p-ISSN: 2320-0936
Volume-15, Issue-5, pp-44-51
www.ajer.org
Research Paper Open Access

Cost-Asymmetric Learned Query Ordering for Exact Offline Range Query Processing

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

ABSTRACT : *Mo's algorithm is an offline method for static range-query processing. Classical block and Hilbert orderings reduce geometric endpoint movement, but they do not model that adding or removing elements at different interval endpoints may have unequal costs. This paper proposes Cost-Asymmetric Learned Mo, an exact range-query framework that fits a small workload-calibrated transition-cost model and uses it only to choose the query order. Correctness is preserved because every query is still answered by exact kernel-specific add and remove operations after the maintained interval reaches the requested range. The model distinguishes add-left, add-right, remove-left, and remove-right operations, then constructs orders with learned-greedy scheduling and learned-2opt local refinement. The experimental artifact contains 4,504 verified execution rows spanning full-baseline profiles, repeated timing, ablations, mismatch tests, small-instance optimality checks, adaptive-policy checks, and 5k-20k scale slices. In the expanded n=q=2400 profile, learned-2opt processing-only speedups over Hilbert were 1.2183x, 1.2196x, 1.3653x, and 1.3051x for distinct-count, weighted-frequency, artificial-asymmetric, and noncommutative kernels. In repeated paired timing at n=q=1800, learned-2opt median processing speedups were 1.1189x to 1.2411x by kernel, with bootstrap confidence intervals excluding 1.0. Fidelity, optimality, adaptive-policy, and scale-slice checks show that the learned objective is a useful operation-price proxy, not a perfect wall-clock predictor. Construction overhead dominates total runtime in one-off Python runs, so the method is promising when ordering cost can be amortized, not as a universal replacement for Hilbert ordering.*

KEYWORDS: *Mo's algorithm, offline range queries, learned cost model, Hilbert ordering, algorithm engineering, local search.*

Date of Submission: 03-05-2026 Date of acceptance: 13-05-2026

I. INTRODUCTION

Offline range-query processing is useful when all queries are known before execution. Mo's algorithm reorders static array queries so that consecutive intervals are close, reducing the number of endpoint updates needed to maintain an exact answer state [1,2]. Classical range-searching and geometric data-structure work provide the broader background for such interval and multidimensional query problems [6,7].

Most Mo orderings minimize movement rather than true operation cost. Standard block Mo sorts by the block of the left endpoint and then by the right endpoint. Alternating block Mo reverses the right-endpoint direction in adjacent blocks. Hilbert ordering maps (l,r) to a space-filling-curve key, exploiting locality properties studied for multidimensional indexing [3-5]. These orderings are strong defaults, but they do not distinguish add-left from add-right or removal from addition.

The contribution is an exact algorithm-engineering framework for offline range-query ordering: it calibrates four endpoint-operation prices and uses them to choose a query permutation, while all query answers remain exact. The novelty is not Mo's algorithm itself, Hilbert ordering, a learned index, or learned query answering. The evidence is deliberately narrow: learned ordering reduces processing-only runtime in the tested benchmarks but learned-2opt is not a universal replacement for Hilbert because construction time dominates one-off total runtime.

Table I: Claim boundary for the proposed framework.

Category	Boundary
Proved	Exactness under any query permutation because learned ordering changes only visitation order.
Observed	Processing-only improvements in tested Python benchmarks, especially under asymmetric endpoint costs or amortized ordering.
Not claimed	Universal end-to-end superiority, production-scale superiority, or replacement of Hilbert for one-off latency-sensitive batches.

This boundary is central to the paper. Exactness is proved for any query permutation. Processing-only speedups are experimental observations. Universal end-to-end superiority is not claimed.

Literature Review and Closest Prior Work

Offline range queries combine data-structure design with workload-aware execution. Range searching is a classical area [6,7], while Mo's algorithm is widely used as a practical offline square-root decomposition strategy [1,2]. Space-filling curves are a standard way to preserve multidimensional locality after linearization; Hilbert curves have been studied for clustering and indexing behavior [3-5].

The paper follows the empirical algorithmics tradition, where practical algorithm claims require measurement methodology, implementation detail, and reproducible artifacts [10,11]. Database query optimization uses cost models to choose among exact execution alternatives [8,9], and later studies show that optimizer quality depends on estimate fidelity and robustness [19,20,27]. Learned optimizers, learned indexes, and workload-aware layouts show that models can exploit regularity in data and workloads [12-14,21-26], but this paper uses a conservative design: the model predicts transition cost and never replaces the exact maintained data structure. Query ordering also resembles route optimization, where local improvement methods such as 2-opt and Lin-Kernighan search are established practical heuristics [15-17]. Cache and external-memory effects can separate a transition-count objective from realized runtime [18].

Table II: Closest prior-work areas and difference from this paper.

Area	Refs.	Optimizes	Exact?	Order?	Asym. ops?	Difference
Mo / offline range queries	[1,2]	Endpoint movement	Yes	Yes	No	Does not price four endpoint operations
Hilbert / space-filling curves	[3-5]	Geometric locality	Yes	Yes	No	Strong baseline but cost-symmetric
Cost-based query optimization	[8,9,19, 20,27]	Execution plans	Usually	No	No	Chooses database plans, not Mo permutations
Learned indexes / layouts	[12-14,23-26]	Data access/layout	Usually	No	No	Learns structures or layout, not exact query order
Route local search	[15-17]	Tour length/cost	N/A	N/A	Can	No exact range-query semantics

II. MATERIALS AND METHODS

Problem Formulation and Cost Model

Let A be a static array and let $Q = \{q1, \ldots, qm\}$ be an offline query set, where $qi = [li, ri]$ and $0 <= li <= ri < n$. A Mo runner maintains a current interval $[L, R]$ and a kernel state $S(L, R)$ that exactly represents the answer for that interval.

For a transition to $qj = [lj, rj]$, define $Delta\_addL = max(0, L - lj)$, $Delta\_addR = max(0, rj - R)$, $Delta\_remL = max(0, lj - L)$, and $Delta\_remR = max(0, R - rj)$. The learned objective is $C(qi -> qj) = b + c\_addL * Delta\_addL + c\_addR * Delta\_addR + c\_remL * Delta\_remL + c\_remR * Delta\_remR$.

The learned model is a ridge-fitted linear transition-cost model. It uses the four operation deltas as features and predicts transition/order cost only. The scheduler fits operation prices from transition samples and timing/nominal calibration signals; it does not use a neural network, does not learn approximate query answers, and does not replace the exact range-query kernel.

After ordering, every query is answered exactly by the kernel's add/remove operations. Every query appears exactly once, and an answer is emitted only after exact endpoint updates have reached the requested interval.

Exact Kernels and Workloads

Four exact kernels are used throughout the main experiments. The artificial-asymmetric kernel is controlled by design; it is included to stress the cost-asymmetry hypothesis, not as a production workload.

Table III: Exact range-query kernels used in the benchmark.

Kernel	Exact query task	Nominal weights
Distinct count	Number of distinct values in [l, r]	(1.0, 1.0, 1.0, 1.0)
Weighted frequency	sum_v w_v f_v^2 over the interval	(2.8, 1.0, 2.0, 1.2)
Artificial asymmetric	Range sum with deliberately asymmetric endpoint work	(8.0, 1.0, 4.0, 2.0)
Noncommutative hash	Rolling sequence hash with direction-sensitive updates	(4.5, 1.4, 3.5, 2.2)

The workload suite includes uniform, hot-zone, sliding-window, short-clustered, mixed-adversarial, and trace-locality interval batches. Trace-locality is trace-inspired synthetic data that simulates repeated local analytics windows around drifting session centers with occasional global jumps. It is not a real production trace.

Ordering Algorithms, Hyperparameters, and Timing

The benchmark includes input order, random order, standard block Mo, alternating block Mo, Hilbert order, hand-weighted greedy, learned-greedy, and learned-2opt. Standard and alternating block Mo use block size $floor(sqrt(n))$ with a minimum of one. Hilbert order uses the smallest number of Hilbert bits such that $2\text{^}bits >= n$.

Table IV: Method and artifact constants needed for reproduction.

Item	Value
Candidate pool	k=64 by default; ablation tests 16 and 128. Pool uses l, r, Hilbert neighbors and deterministic random fill.
Calibration/regression	25% measured timing ratio + 75% nominal operation ratio; ridge lambda=1e-6; operation coefficients clipped nonnegative; bias not clipped.
Transition samples	3200 main; 1200 small scaling; 3600 large scaling; 5000 n=q=5000; 2400 repeated; 1800 mismatch.
Local search	Budget=max(800, min(12000,4q)); relocation, segment relocation length 2..6, short 2-opt window <=20, adjacent swap.
Timing/statistics	Two warm-ups and 20 timed repetitions for repeated timing; paired bootstrap with 2000 samples; Wilcoxon signed-rank with Holm correction.
Commands	Smoke: python3 python/run_experiments.py --profile smoke --out data/results/smoke_results.csv; tables: ./scripts_regenerate_tables_from_csv.sh; correctness: python3 python/verify_correctness.py.

The experimental artifact contains 4,504 verified execution rows with zero failed correctness checks: 736 full-baseline main/scaling/larger rows, 1,920 repeated-timing rows, 1,200 ablation rows, 96 mismatch rows, 192 targeted 5k-20k scale-slice rows, and 360 small-instance optimality rows. The reported tables use dependency-free Python on the environment recorded in the artifact metadata: Python 3.14.4, macOS 15.5 arm64. Apple clang 17.0.0 was available, but no C++ timing results are claimed.

Code and Data Availability

The public code repository is available at https://github.com/svanidzen-gtu/earned-mo-ajer-artifact/. It contains the source code, README run commands, and citation metadata needed to regenerate experiment outputs locally. The repository is code-only and does not include archived CSVs, generated tables or figures, manuscript files, or checksum artifacts. Fresh timing values may vary across machines and runtime environments.

III. RESULTS AND DISCUSSION

Processing speedup is defined as Hilbert processing time divided by the candidate orderer's processing time. Total speedup also includes ordering construction time. Unless explicitly labeled total, every speedup below is processing-only.

The main n=q=2400 profile shows that learned-2opt reduces processing time relative to Hilbert for all four kernels, with larger gains for the more asymmetric kernels.

Table V: Expanded n=q=2400 learned-2opt processing speedup over Hilbert.

Kernel	Runs	Processing speedup	Weighted-cost ratio
Distinct count	15	1.2183x	0.8462
Weighted frequency	15	1.2196x	0.7672
Artificial asymmetric	15	1.3653x	0.6901
Noncommutative hash	15	1.3051x	0.7564

Repeated paired timing strengthens the processing-only result, but it also makes the construction-overhead limitation visible. The confidence intervals below exclude 1.0; total speedups are all below 1 because learned-2opt ordering takes roughly 2.1 seconds in these Python runs.

Table VI: Repeated paired timing at n=q=1800, learned-2opt versus Hilbert.

Kernel	Processing speedup	95% bootstrap CI	Holm p	Total speedup
Distinct count	1.1928x	[1.1548, 1.2383]	<1e-6	0.0116x
Weighted frequency	1.1189x	[1.0936, 1.1394]	<1e-6	0.0170x
Artificial asymmetric	1.2411x	[1.2265, 1.2706]	<1e-6	0.0356x
Noncommutative hash	1.2065x	[1.1768, 1.2568]	<1e-6	0.0235x

Hilbert is the primary baseline, but the stronger comparison is against the best non-learned orderer among standard block Mo, alternating block Mo, Hilbert, and hand-weighted greedy. In the n=q=2400 profile, learned-2opt remains processing-faster than this strongest non-learned baseline, although one-off total runtime remains worse.

Table VII: Learned-2opt versus strongest non-learned baseline at n=q=2400.

Kernel	Best non-learned	Proc. vs best	Total vs best	Cost ratio	L-2opt order ms
Distinct count	hand-weighted	1.2100x	0.1347x	0.8555	3395.9
Weighted frequency	hand-weighted	1.2169x	0.1366x	0.8075	3563.3
Artificial asymmetric	hand-weighted	1.2282x	0.1482x	0.8684	3547.4
Noncommutative hash	hand-weighted	1.2345x	0.1420x	0.7970	3539.8

The learned objective is useful but imperfect. The artifact reports median Spearman correlations of at least 0.9286 in the n=q=2400 profile and at least 0.9744 in repeated timing. High rank correlation supports orderer selection among the tested alternatives; it does not prove wall-clock optimality because cache effects, Python interpreter overhead, and finite local-search budgets are not fully modeled.

Table VIII: Targeted n=q=20000 Python scale slice, learned-2opt over two workloads.

Workload	Kernel	Proc. speedup	Total speedup	Peak RSS KB	Interpretation
hot-zone	Distinct count	1.0737x	0.0741x	59968	processing win
hot-zone	Weighted frequency	1.0879x	0.1145x	66784	processing win
hot-zone	Artificial asymmetric	0.9333x	0.2148x	68192	processing loss
hot-zone	Noncommutative hash	1.0763x	0.1542x	68192	processing win
trace-locality	Distinct count	1.0670x	0.0628x	68192	processing win
trace-locality	Weighted frequency	0.9116x	0.0896x	68192	processing loss
trace-locality	Artificial asymmetric	0.9878x	0.1874x	68192	near tie
trace-locality	Noncommutative hash	1.0850x	0.1333x	68192	processing win

The n=q=20000 scale slice shows that scaling does not simply amplify the learned-2opt advantage. Five of the eight workload-kernel rows remain processing-faster than Hilbert, but weighted frequency on trace-locality and artificial-asymmetric on hot-zone lose in processing time, and every total-speedup value remains below 1.0 because Python ordering construction dominates. These results indicate that the learned objective is an operation-price proxy rather than a complete wall-clock model; at larger sizes, interpreter overhead, cache behavior, memory locality, and finite candidate/local-search budgets can offset lower predicted transition cost. The appropriate deployment interpretation is therefore selective: validate on a sample, use adaptive policy or Hilbert fallback under mismatch, and rely on learned-2opt only when order construction can be amortized.

Ablations, mismatch, optimality, and adaptive-policy checks are kept in the artifact rather than repeated as full main-paper tables. In summary, learned-greedy and manual greedy are close at n=q=1200, learned-2opt gains mainly come from local refinement, and equal-cost controls show that some distinct-count improvement comes from movement/local-search quality rather than cost asymmetry. Small q=8,10,12 dynamic-programming checks show learned-2opt reaches the calibrated-objective optimum in the median q=12 cases. Mismatch transfer is mixed: learned-2opt remains processing-faster in the tested transfers, while learned-greedy can fail under workload shift.

Construction overhead is the key deployment issue. Representative break-even reuse counts are 41.25 for learned-greedy and 143.80 for learned-2opt in artificial-asymmetric hot-zone; 67.70 and 324.70 in weighted-frequency sliding-window; and no break-even for learned-greedy versus 1148.05 reuses for learned-2opt in the distinct-count hot-zone control. The adaptive policy matches the oracle in all 12 one-off cases and 10 of 12 cases at 200 reuses, so it is useful guidance but not yet a production scheduler.

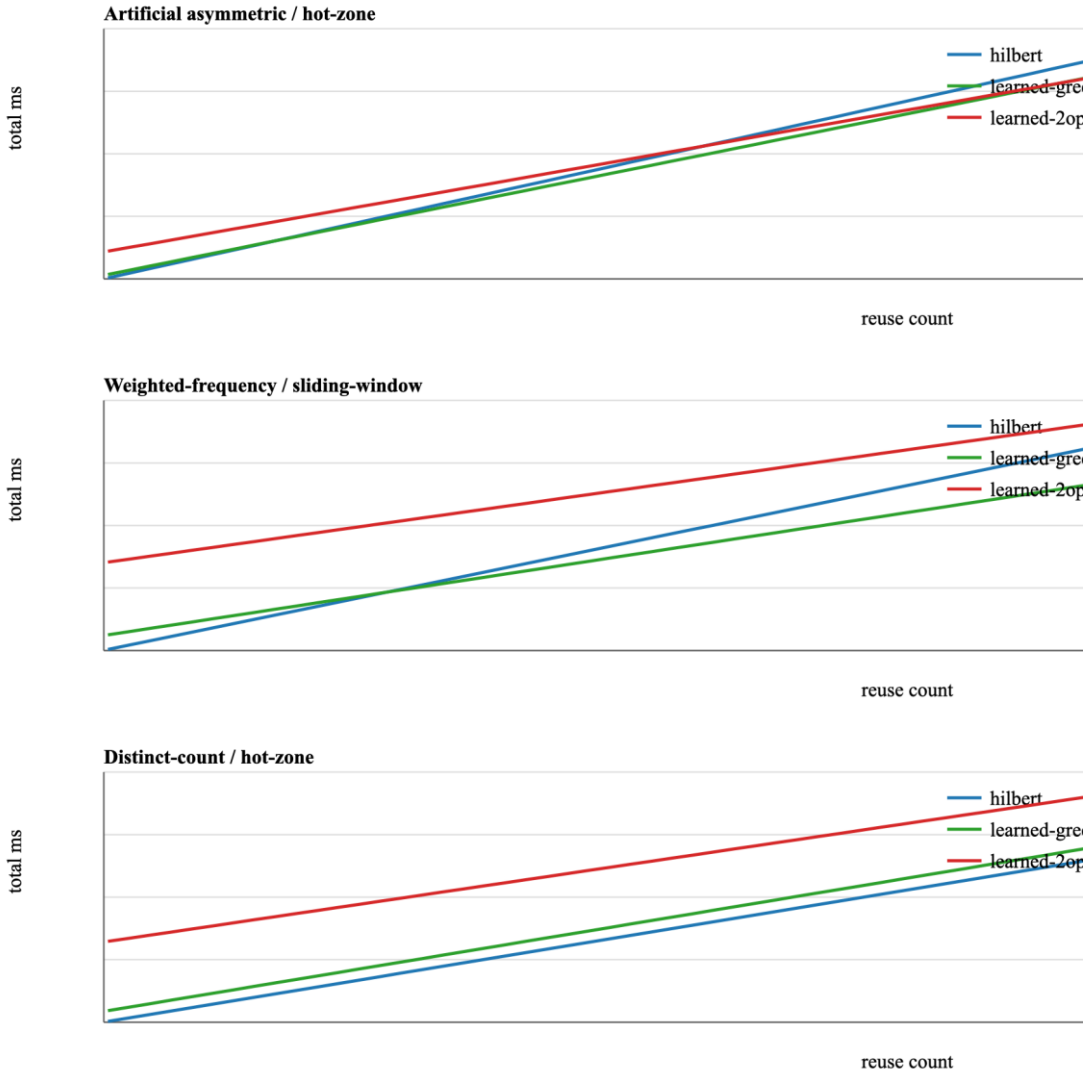

Figure 1: Total runtime as a function of reusing the same query order.

Table IX: Practical deployment guidance.

Scenario	Recommended orderer	Reason
One-off batch or cheap kernel	Hilbert or alternating Mo	Low construction overhead matters more than predicted traversal savings.
Expensive asymmetric kernel with reusable query set	learned-2opt	Processing savings can be amortized across repeated executions.
Moderate asymmetry or limited reuse	learned-greedy or adaptive policy	Lower construction cost can be a better compromise than local search.
Uncertain workload or model mismatch	adaptive policy or Hilbert fallback	Pilot estimates reduce the risk of choosing a costly learned order.
Large n where objective/runtime fidelity weakens	validate on a sample before learned-2opt	The 20k slice includes non-winning processing cases.

In plain terms, learned ordering must pay back its construction cost. That payback can happen when the same query permutation is reused, when endpoint updates are expensive, or when the application can amortize ordering across many batches. Without such reuse, Hilbert or alternating Mo is usually safer.

Limitations and Threats to Validity

The evidence is single-machine and Python-bound. Repeated timing improves statistical reliability, but it does not replace a lower-overhead C++ study at larger n and q. The targeted Python scale slice reaches n=q=20000, but it is not a full-baseline repeated study, and it shows that learned-2opt processing gains are not uniform at larger sizes. A complete lower-overhead learned-2opt/repeated-timing implementation was not completed in the available artifact.

The workloads are synthetic or trace-inspired; no production trace is claimed. The artificial asymmetric kernel is controlled by design. The calibrated objective blends nominal operation weights with short, measured timing ratios, so the learned coefficients are stable operation-price estimates rather than pure wall-clock predictors. Construction overhead is the main limitation and learned-2opt is not appropriate for one-off latency-sensitive settings where order construction counts immediately.

IV. CONCLUSION

In the tested benchmarks, learned ordering reduced processing-only runtime relative to Hilbert in the n=q=2400 expanded profile and the n=q=1800 repeated-timing profile. Repeated paired timing showed learned-2opt median processing speedups from 1.1189x to 1.2411x by kernel, and the expanded n=q=2400 profile showed processing-only speedups from 1.2183x to 1.3653x. These are experimental observations; exactness under any query permutation is the proved property.

The evidence does not establish learned-2opt as a universal replacement for Hilbert ordering. Construction overhead dominates total runtime in one-off Python runs, and the 20k scale slice contains non-winning processing cases. Cost-asymmetric ordering is therefore best framed as an exact, amortizable ordering technique: promising for large or repeated batches with visible endpoint-cost asymmetry, but not a blanket end-to-end improvement.

V. ACKNOWLEDGEMENT AND DISCLOSURE STATEMENTS

Conflict of Interest: The author declares no known conflict of interest. Funding: No external funding was used for this study. Data and Code Availability: The code and data artifact is prepared for public archival deposit as described in the Code and Data Availability subsection; insert the public identifier here before submission https://github.com/svanidzen-gtu/earned-mo-ajer-artifact/. Human/Animal Subjects: The work uses generated benchmark data and does not involve human participants, animal subjects, or personal data. Originality: The manuscript is original work and, to the author's knowledge, is not under review elsewhere.

REFERENCES

[1]. A. Laaksonen, Competitive Programmer's Handbook, 2018.
[2]. CP-Algorithms Contributors, "Mo's algorithm, sqrt decomposition and offline query processing," 2024.
[3]. I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," in Proc. VLDB, 1994, pp. 500-509.
[4]. B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the Hilbert space-filling curve," IEEE Trans. Knowl. Data Eng., vol. 13, no. 1, pp. 124-141, 2001.
[5]. J. K. Lawder and P. J. H. King, "Using space-filling curves for multi-dimensional indexing," in Proc. BNCOD, 2001, pp. 20-35.
[6]. J. L. Bentley, "Multidimensional binary search trees used for associative searching," Commun. ACM, vol. 18, no. 9, pp. 509-517, 1975.
[7]. M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, Computational Geometry: Algorithms and Applications. Berlin, Germany: Springer, 2008.
[8]. 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. SIGMOD, 1979, pp. 23-34.
[9]. Y. E. Ioannidis, "Query optimization," ACM Comput. Surv., vol. 28, no. 1, pp. 121-123, 1996.
[10]. C. C. McGeoch, A Guide to Experimental Algorithmics. Cambridge, U.K.: Cambridge University Press, 2012.
[11]. P. Sanders, "Algorithm engineering: An attempt at a definition," in Efficient Algorithms. Berlin, Germany: Springer, 2009, pp. 321-340.
[12]. T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in Proc. SIGMOD, 2018, pp. 489-504.
[13]. J. Ding et al., "ALEX: An updatable adaptive learned index," in Proc. SIGMOD, 2020, pp. 969-984.
[14]. A. Kipf et al., "RadixSpline: A single-pass learned index," in Proc. aiDM, 2020.
[15]. S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," Oper. Res., vol. 21, no. 2, pp. 498-516, 1973.
[16]. K. Helsgaun, "An effective implementation of the Lin-Kernighan traveling salesman heuristic," Eur. J. Oper. Res., vol. 126, no. 1, pp. 106-130, 2000.
[17]. D. L. Applegate, R. E. Bixby, V. Chvatal, and W. Cook, The Traveling Salesman Problem: A Computational Study. Princeton, NJ, USA: Princeton University Press, 2006.
[18]. A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," Commun. ACM, vol. 31, no. 9, pp. 1116-1127, 1988.
[19]. V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" Proc. VLDB Endow., vol. 9, no. 3, pp. 204-215, 2015.

[20]. V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in Proc. SIGMOD, 2004, pp. 659-670.
[21]. R. Marcus et al., "Neo: A learned query optimizer," Proc. VLDB Endow., vol. 12, no. 11, pp. 1705-1718, 2019.
[22]. R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in Proc. SIGMOD, 2021, pp. 1275-1288.
[23]. T. Kraska et al., "SageDB: A learned database system," in Proc. CIDR, 2019.
[24]. A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "FITing-tree: A data-aware index structure," in Proc. SIGMOD, 2019, pp. 1189-1206.
[25]. P. Ferragina and G. Vinciguerra, "The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds," Proc. VLDB Endow., vol. 13, no. 8, pp. 1162-1175, 2020.
[26]. Z. Yang et al., "Qd-tree: Learning data layouts for big data analytics," in Proc. SIGMOD, 2020, pp. 193-208.
[27]. R. Heinrich, M. Luthra, J. Wehrstein, H. Kornmayer, and C. Binnig, "How good are learned cost models, really? Insights from query optimization tasks," Proc. ACM Manag. Data, vol. 3, no. 3, pp. 1-27, 2025, doi: 10.1145/3725309.