What Drives Learned Optimizer Performance? A Systematic Evaluation

Kostas Mparmparousis
k.mparmparousis@athenarc.gr
Archimedes, Athena Research Center
Athens, Greece

Christos Tsapelas
ctsapelas@athenarc.gr
Department of Informatics and Telecommunications, National and Kapodistrian University of Athens
Archimedes, Athena Research Center
Athens, Greece

Georgia Koutrika
georgia@athenarc.gr
Archimedes, Athena Research Center
Athens, Greece

Abstract

Traditional query optimization, though highly refined, faces limitations in today's data-intensive landscape. Learned Query Optimizers (LQOs) aim to overcome these challenges and transform query execution. As research accelerates, a clear analytical framework is essential to understand the design choices that drive LQO performance and unlock their full potential. We present a structured evaluation framework for Learned Query Optimizers (LQOs), grounded in five core dimensions: performance, robustness, learning procedure, decision-making, and generalization. Using this framework, we compare the classic query optimizer with five state-of-the-art LQO systems. Our findings highlight the potential of LQOs to outperform traditional optimizers in challenging scenarios, while also revealing critical dependencies on training strategies, model design, and workload characteristics.

Keywords

learned query optimizers, evaluation

1 Introduction

Learned query optimization is an active area in database research, aiming to overcome limitations of traditional optimizers, such as assumptions of attribute independence, uniform data distributions, and heuristic-based planning [33]. With increasingly dynamic workloads, Learned Query Optimizers (LQOs) leverage machine learning to infer execution plans from data and past executions, offering improved adaptability, reduced manual tuning, and enhanced performance [3]. While several LQOs have shown promising results in specific settings [6, 9, 37, 38, 68, 72, 84], their broader applicability remains uncertain. This stems primarily from the absence of a unified rigorous analytical framework, which would help shed light on the factors driving LQO performance. Current system evaluations exhibit two major limitations:

(1) Inconsistent evaluation settings. Table 1 summarizes current practices in three dimensions: benchmarks, training/testing split strategies, and experimental controls (query order, training epochs, and experiment repetitions). These practices lack consistency, making cross-system comparisons difficult. We argue that objective evaluation should follow four core principles.

– *A broad set of benchmarks* is essential to capture the full range of LQO capabilities, yet prior work often relies on a narrow pool.

– *Well-defined training and test splits* are essential for reliable evaluation, but existing studies adopt inconsistent split strategies.

– *Thorough experimental controls* are critical. Query order can influence learning outcomes, and multiple repetitions are needed to account for execution variability [32]. Yet, these aspects are scarcely considered when examining LQOs, leading to large variances in evaluation settings. For query order specifically, prior evaluations either shuffled queries randomly or followed the input order.

– *A consistent hardware and database environment* is critical, as system setup significantly impacts performance. However, as shown in Tables 5 and 6, existing evaluations exhibit substantial diversity in hardware and configuration.

(2) Limited insights into optimizer behavior. Most evaluations focus narrowly on performance metrics, overlooking deeper LQO behavioral aspects such as learning dynamics, decision-making processes, and stability under varying conditions.

Recent studies have laid valuable groundwork by attempting to address specific aspects of the evaluation challenge. Lehmann et al. [32] exposed inconsistencies in evaluation practices and proposed more systematic train/test splits. Mikhaylov et al. [45] investigated architectural robustness by refining NEO's [38] components and comparing them to its original design. Zhang et al. [80] analyzed BAO [37] and BALSA [72] to determine which optimization components (e.g., join ordering vs. operator selection) influence performance and generalization. While insightful, these efforts remain narrow in scope, each targeting isolated questions rather than offering a comprehensive evaluation framework.

Our work introduces a comprehensive evaluation framework with two goals: enabling objective, cross-system comparison grounded in established evaluation principles, and systematically examining LQO behavior across dimensions largely overlooked by prior evaluations. To this end, we address five research questions, and Table 2 contrasts our study with previous work.

(E1) End-to-End Performance & Value Model Fidelity: How do state-of-the-art LQOs perform under a unified evaluation across diverse workloads, and to what extent does value model accuracy correlate with execution latency?

(E2) Sensitivity & Execution Stability: How sensitive are LQOs to training query order, and how consistently do they produce stable, high-quality plans across repeated executions?

(E3) Learning Trajectory & Convergence: How do training policies, based on epochs, loss thresholds, or query exposure, affect LQO learning speed, stability, and final performance?

(E4) Internal Decisions & Plan Representation: What patterns do LQOs exhibit in core decisions like access path and operator selection? How well do their embeddings reflect plan quality?

(E5) Generalization to Novel Conditions: How well do LQOs generalize beyond their training distribution when faced with unseen queries, schema changes, or significant data shifts?

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.49

Table 1: Evaluation Setting Across Optimizers

Optimizer	Benchmarks	Train/Test Split	Query Order	Epochs	Reps
NEO	JOB, TPC-H, Corp	Random 80/20	Random shuffle	100	50
BAO	JOB, STATS, Corp	Time series (train Q1..Qt, test Qt+1)	Random shuffle	100	8
LOGER	JOB, TPC-DS, STATS	Template-based (e.g., JOB: 80/33)	Follows input ordering	200	N/A
FASTgres	JOB, STATS, TPC-H	Stratified 80/20 per context	Random shuffle	N/A	N/A
LERO	JOB, STATS, TPC-H, TPC-DS	Time series	Follows input ordering	N/A	N/A

Epochs: Training Epochs; Reps: Experiment Repetitions.

Table 2: Analysis of Our Exper. Contributions and Prior Work

Experiment	Contribution and Prior Work Analysis
1. Performance & Value Model	**Latency:** Inconsistently Addressed by all LQOs. **Accuracy:** Not systematically evaluated.
2. Sensitivity & Exec. Stability	**Splits & Order:** Partially addressed by [32]; ordering sensitivity unexamined. **Plan Stability:** Not systematically evaluated.
3. Learning Trajectory	**Epoch-Based:** Addressed by NEO, LOGER. **Loss-Based:** Addressed by BAO. **Query-Exposure:** Addressed by LERO and [32].
4. Internal Decisions	**Access Paths:** Not systematically evaluated. **Operator Comp.:** Not systematically evaluated. **Operator Bias:** Not systematically evaluated. **Emb. Quality:** Not systematically evaluated.
5. Generalization	**Unseen Queries:** Addressed incons. by most, [80]. **Schema Changes:** Addressed by BAO, LOGER. **Data Shifts:** Addressed by BAO, FASTgres, [32].

Briefly, this paper makes three primary contributions: *a)* we deconstruct LQOs into their core architectural components and establish a unified anatomy, which we leverage to compare their diverse designs, *b)* we introduce a novel evaluation framework for LQOs, consisting of five experiments designed to assess their capabilities and internal behaviors in realistic scenarios, *c)* we apply this framework to a collection of prominent LQOs and conduct extensive analysis. Our findings highlight the notable performance gains from LQOs, but showcase high sensitivity in their training scheme, while PostgreSQL remains highly competitive. This work addresses key gaps in LQOs by enabling structured comparisons, uncovering internal behaviors, and informing the development of more robust designs, including adaptive LQO architectures and more interpretable, resilient optimization strategies.

2 Related Work

2.1 Learned Components

Recently, substantial progress has been made to integrate ML approaches into query optimization. *Learned Cardinality Estimators (LCEs)* [14, 23, 29, 34, 36, 41, 50, 52, 53, 59, 59, 65, 69, 69, 73, 74, 79, 86] provide estimates to the optimizer either relying on a training workload or capturing underlying data distributions. *Join Order Search Methods* [27, 39, 39, 77, 78] generate a join order for the query execution. Finally, *Learned Cost Models (LCMs)* [8, 22, 25, 26, 35, 39, 58, 71, 81] estimate the cost of execution strategies by capturing complex data distributions and inter-dependencies with workloads. Zhu et al. [85] serves as a comprehensive curator of the learned component landscape, introducing and designating representative approaches from each of the previously defined categories.

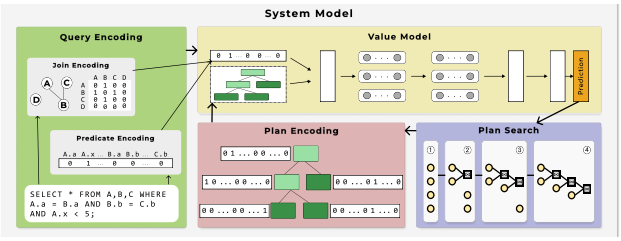


Figure 1: Template architecture of an LQO.

Benchmarks and Evaluations. An extension of the Join Order Benchmark [33] addresses size and diversity requirements for LCE training sets [51] (see Sec. 4.2), while a benchmark of 20 real-world datasets was introduced for LCE generalization [10].

Prior work has proposed a range of evaluation methodologies for LCEs [17, 18, 28, 60, 66] and LCMs [19, 20]. For LCEs, studies evaluate: **(a)** estimation accuracy, runtime, and memory overhead [18], **(b)** generalization under static and dynamic data distributions [66], **(c)** practical applicability, including inference latency, model size, training and update efficiency [17], **(d)** sensitivity to database and training configurations, such as schema characteristics, correlations, skew, and join depth [60], and **(e)** the impact of dataset properties on estimation quality and downstream plan performance [28]. For LCMs, Heinrich et al. [19, 20] evaluate how LCMs affect three central query optimization tasks, i.e., ordering joins, access paths and physical operators selection.

2.2 Learned Query Optimizers

At a high level, LQOs and LCMs share the same goal: reducing query execution time by modeling data distributions and workload characteristics, however, they differ fundamentally in approach. LCMs focus on accurate cost prediction, whereas LQOs aim to learn effective optimization policies -often through reinforcement learning- typically incorporating a value network that estimates performance at the sub-plan level [19].

To that end, LQOs can be distinguished to *a) end-to-end LQOs* [2, 5, 7, 9, 30, 38, 62, 72], which aim to replace the database optimizer by predicting execution plans from scratch, and *b) learned assistants* [1, 6, 13, 24, 37, 42, 49, 68, 75, 83], which aim to "steer" the existing optimizer to predict better execution plans by providing *query hints*, like `SET enable_nestloop TO off`. Similarly, various efforts have been made for evaluation and benchmarking of LQOs [17, 32, 45, 60, 63, 64]. In Sections 4.1 and 4.2, we discuss their characteristics and limitations, and finally the set of benchmarks we chose for our evaluation suite.

3 The Anatomy of LQOs

We now analyze LQOs architectural designs through five core axes: *System Model*, *Value Model*, *Query Encoding*, *Plan Encoding*, and *Plan Search*. Fig. 1 presents a template LQO architecture.

The **System Model** implements the optimization process, determining join order, access paths, and physical operators for a given query. Its outputs range from complete plans to optimizer hints, depending on whether the system is end-to-end or assistant-based A key distinction is whether the system employs learning from demonstration [21, 40], a training phase that accelerates learning by observing expert and if so, which it learns to emulate (e.g., the DBMS optimizer or a simulator).

Table 3: Comprehensive Architecture Comparison of Learned Query Optimizers

Dimension	Attribute	NEO (2019)	BAO (2020)	LOGER (2023)	BALSA (2022)	FASTgres (2023)	BASE (2023)	LERO (2023)
System Model	Type	End-to-End	Learned Assistant	End-to-End	End-to-End	Learned Assistant	End-to-End	Learned Assistant
	Output	QEP	Hint Set	QEP	QEP	Hint Set	QEP	QEP
	Learning from Demonstration	✓	✓	✗	✓	✓	✓	✓
	Expert Source	Classic Optimizer	Classic Optimizer	N/A	Simulator*	Classic Optimizer	DBMS Cost Model	DBMS Cost Model
Value Model	Prediction Task	Regression	Regression	Regression	Regression	Classification	Regression	Learning-to-Rank
	Prediction Target	Execution Latency	Execution Latency	Execution Latency	Cost + Latency	Optimal Hint Set	Execution Latency	Relative Order
	Model Architecture	Tree-CNN	Tree-CNN	Tree-LSTM	Tree-CNN	Gradient Boosting	Deep Policy Net	Tree-CNN
	Model Count	1	1	1	2	No. of Contexts	1	1
Query Encoding	Join Encoding	Join Adjac. Matrix	-	Graph Transformer	Join Adjac. Matrix	-	Join Adjac. Matrix	-
	Predicate Encoding	R-Vectors	-	1-Hot of Predicate Operator.	Table→Selectivity Vector	1-Hot of Pred. Op. + Value Norm.	Column→Selectivity Vector	-
	Uses Column Stats	✗	-	✓	✗	✗	-	-
	Dedicated NN Encoder	✓	-	✓	-	-	-	-
	Separate Training	✓	-	✗	-	-	-	-
Plan Encoding	Encodes Op. Types	✓	✓	✗	✓	✗	✓	✓
	Encodes DB Schema	✓	✗	✓	✓	✗	✓	✓
Plan Search	Search Objective	QEP Construction	Hint Selection	Operator Pruning	QEP Construction	Hint Selection	QEP Construction	Hint Selection
	Search Algorithm	Best-First (Heap)	N/A	e-Beam Search	Beam Search	Decision Tree	Stochastic Policy	N/A

* BALSA's simulator uses a logical-only cost model with classic cardinality estimator.
✓ = Yes; ✗ = No; - = Not Applicable or Not Used.

The **Value Model** is the core predictive component of query optimization, evaluating candidate strategies. It typically performs one of three tasks: *regression* (predicting strategy cost), *classification* (selecting among discrete strategies), or *learning to rank* (ordering strategies by expected performance), which align with its prediction target. Value Models vary in architecture and may comprise a single model or multiple models per system.

The **Query Encoder** is an optional component that transforms raw SQL queries into structured representations. It is characterized by its *join* and *predicate* encoding techniques, and key design choices include whether *column-level statistics* are incorporated and whether a dedicated neural network is used –trained either jointly with the Value Model or in a separate pre-training phase.

The **Plan Encoder** translates query execution plans (QEPs) by encoding each node into a fixed-length feature vector. Key design choices include whether join and scan operators are encoded per node and whether the encoding incorporates the database schema.

The **Plan Search** defines how a system traverses the query plan space. To this end, systems may employ a variety of search algorithms depending on their design and objectives.

Existing taxonomies–most notably the LCM-focused framework of [20] later applied in [19] –classify learned cost models but do not capture the architectural dimensions needed to analyze full learned query optimizers. Consequently, they cannot address key questions derived from Table 3 such as (**a**) differences in effectiveness across LQO system types (Section 5.2.2), (**b**) the comparative suitability of learning paradigms (e.g., Learning-to-Rank [70] vs. Regression in Section 5.2.1), or (**c**) the impact of query encodings on learning behavior (e.g., LOGER and NEO in Section 5). We extend the analysis of [32] by detailing (**a**) how LQOs achieve competitive performance, (**b**) how they encode joins, predicates, and database metadata, and (**c**) how they explore the plan search space.

3.1 Dissecting the Anatomy

The following LQOs, presented in chronological order, represent diverse architectures and learning strategies. Table 3 summarizes their key design dimensions.

NEO [38] is an end-to-end neural optimizer that directly outputs QEPs, demonstrating that holistic plan prediction can outperform methods relying on intermediate estimates. NEO learns from the classic optimizer's execution history and employs a single Tree-CNN *Value Model* for latency regression over sub-plans and candidate QEPs. Its *Query Encoding* combines a join adjacency matrix with R-vectors, dense table embeddings inspired by word2vec [46], trained separately. The *Plan Encoder* represents operator types and schema information via one-hot encodings, while *Plan Search* uses best-first search to construct QEPs in bottom-up fashion.

BAO [37] is a learned assistant that augments traditional optimizers by framing hint selection as a reinforcement learning problem. It initializes from classic optimizer executions and refines its policy via Thompson sampling [61]. Its *Value Model* employs a single Tree-CNN to predict the latency of hinted QEPs. Its *Plan Encoding* is schema-agnostic, representing only operator types via one-hot vectors augmented with node-level estimates. It greedily selects the hint set with the lowest predicted latency.

BALSA [72] is an end-to-end optimizer that generates full QEPs via a two-stage learning process that avoids dependence on classic optimizer demonstrations. It first trains in a simulated environment using a logical cost model with a traditional cardinality estimator, then fine-tunes on real executions. Its *Value Model* mirrors this design, employing two Tree-CNNs to regress sub-plans and full QEPs on both cost and latency. The *Query Encoder* reuses NEO's join encoding and represents predicates through a *Table→Selectivity* vector, while *Plan Encoding* follows NEO's approach and *Plan Search* uses beam search to construct QEPs in bottom-up fashion.

BASE [9] is an end-to-end optimizer that generates QEPs without relying on optimizer demonstrations. Similar to BALSA, it follows a two-phase approach –cost-based exploration using the DBMS cost model, followed by lightweight calibration. Its *System Model* learns directly from the cost model, while the *Value Model* uses a single deep policy network for latency regression. The *Query Encoder* combines a join adjacency matrix with a *Column→Selectivity* vector, *Plan Encoding* reuses NEO's design, and *Plan Search* employs policy-based sampling for bottom-up plan construction.

FASTgres [68] is a hint-based learned assistant that partitions queries by structural and statistical features and assigns each a dedicated classifier. Its *System Model* learns from classic optimizer executions, while the *Value Model* employs gradient-boosted trees per query context. The *Query Encoder* represents predicate operators via normalized one-hot vectors. Owing to its context-based design, FASTgres omits explicit *Plan Encoding*, and *Plan Search* directly selects the optimal hint through its decision-tree models.

Table 4: Benchmark dimensions

Dim.	Attribute	TPC-H	TPC-DS	JOB	STATS	JOB-COMPLEX	CEB	SQLStorm
Schema	Data Model	3NF	Snowflake	Star	Star	Star	Star	Star
	# Tables	8	24	21	8	21	21	21
	Min # Columns	3	3	2	4	2	2	2
	Max # Columns	16	34	13	21	13	13	13
Data	Type	Synthetic	Hybrid	Real	Real	Real	Real	Real
	DB Size	1GB–100TB	1GB–100TB	3.6GB	5.78GB	3.6GB	3.6GB	3.6GB
	Correlations	Avoids	Avoids	Contains	Contains	Contains	Contains	Contains
Scaling	**Fact Tables**							
	Linear Scaling	N/A	✓	N/A	N/A	N/A	N/A	N/A
	Sub-linear Scaling	N/A	✗	N/A	N/A	N/A	N/A	N/A
	Dimension Tables							
	Linear Scaling	N/A	✗	N/A	N/A	N/A	N/A	N/A
	Sub-linear Scaling	N/A	✓	N/A	N/A	N/A	N/A	N/A
Queries	# Query Templates	22	99	33	70	30	21	7
	Variants per Templ.	≥ 1	≥ 1	2–6	1–4	1	>100	>1000
	# Queries	SF dep.	SF dep.	113	6,200	30	13,646	11,714
	Avg. Joins per Query	~3	5	8	~4	10	8	6
	Join Range	0–6	1–10	3–16	2–8	5–14	5–15	1-13
	Types	1, 2, 4	1, 2, 3, 4	1, 2	1, 2, 4	1, 2	1, 2	1, 2, 3, 4
	Updates	✗	✗	N/A	✓	N/A	N/A	N/A

* Query Types: 1 = Ad-Hoc, 2 = Reporting, 3 = OLAP, 4 = Data Mining.
✓ = Yes; ✗ = No; N/A = Not Applicable; SF dep. = Scale Factor dependent.

LOGER [6] adopts a hybrid learning strategy that prunes the search space while delegating final plan generation to the native optimizer. Despite involving the classic optimizer, we classify LOGER as end-to-end, as it outputs a QEP via bottom-up construction. Its *Value Model* employs a single Tree-LSTM for latency regression over sub-plans and candidate QEPs. The *Query Encoder* uses a Graph Transformer with column-level statistics, whose representations are reused by the *Plan Encoder*. *Plan Search* applies e-beam search to prune operators during plan construction.

LERO [84] formulates query optimization as a learning-to-rank problem, emphasizing plan ordering over cost prediction while collaborating with the native optimizer. Its *System Model* generates candidate QEPs by injecting alternative cardinality estimates into the classic planner and ranks them using the DBMS cost model. The *Value Model* employs a Tree-CNN as a pairwise classifier to predict relative plan orderings. LERO omits a *Query Encoder*, represents operators and schema via one-hot encodings augmented with node-level statistics in the *Plan Encoder*, and greedily selects the best plan from the candidate set without an explicit *Plan Search* algorithm.

4 Experimental Setup and Design

4.1 Benchmark Dimensions

Benchmark design plays a pivotal role in evaluating LQOs, as it directly influences the validity and depth of performance comparisons. Below, we outline the dimensions presented in Table 4, and we discuss each dimension's influence on LQO evaluation.

Schema. Schema design strongly shapes query-optimization complexity and thus LQO performance. Common models include **(a)** 3NF, which reduces redundancy [43], **(b)** star schemas, which simplify and speed up queries [11], and **(c)** snowflake schemas, which support richer structures with lower storage cost [15].

Data & Scaling. Dataset characteristics strongly influence the behavior of (learned) query optimizers. Benchmarks vary in using real versus synthetic data, and accordingly in whether they capture realistic correlations and multi-modal distributions. Real-world datasets pose substantial challenges to optimizers [33], while synthetic ones approximate such distributions using scalable generation techniques tailored to table types (e.g., Fact vs. Dimension).

Queries. Benchmarks differ in the statistical, structural, and semantic properties of their queries. Key factors include **(a)** the number of query templates, determining structural diversity, **(b)** the number of query variants per template, affecting LQO generalization, **(c)** join depth, often correlated with optimization complexity, and **(d)** the analytical task type [55], including ad-hoc, reporting, iterative OLAP, and data mining queries.

4.2 Evaluation Benchmarks

For our evaluation framework we select a set of benchmarks that have been widely adopted in both academia and industry. **TPC** benchmarks assess query performance on large-scale workloads, with **TPC-H** and **TPC-DS** being the most widely used. **TPC-H** [54] focuses on decision support via 22 templates that test join ordering, predicate evaluation, aggregation, and sorting over large datasets. **TPC-DS** [48] extends TPC-H with a more complex schema and data skews, including 99 parameterized template in four workload classes: reporting, ad hoc, iterative OLAP, and data mining.

The **Join Order Benchmark** (JOB) [33] evaluates database optimizers' join ordering capabilities using real-world IMDB data with authentic correlations, skew, and irregularities. It includes 113 Select-Project-Join (SPJ) queries designed to test an optimizer's ability to generate efficient join trees and minimize execution costs.

JOB-Complex [67] and **CEB** [51] are handcrafted extensions of the JOB workload with distinct design goals. **JOB-Complex** emphasizes real-world complexity by introducing **(a)** joins on string or non–PK-FK columns, **(b)** string-based predicates, and **(c)** intra-table column comparisons. **CEB** expands JOB with 21 new templates and hundreds of queries per template, targeting improved learning via **(a)** large training sets, **(b)** diverse execution scenarios, and **(c)** edge cases. **SQLStorm** [56] focuses on scale and diversity through LLM-generated queries derived from seven prompt templates, covering a wide range of complexities, including outer joins, recursive CTEs, semantic corner cases, and string-processing workloads.

STATS [17] evaluates the performance of cardinality estimation (CE) techniques using real-world Stack Exchange data with complex correlations and skewed distributions. Its workload includes 6.2K diverse multi-table join queries varying in structural and statistical complexity. Model training begins on a baseline snapshot, with incremental data injections simulating real-world shifts.

Structuring the Evaluation Suite. Although existing benchmarks were not designed specifically for LQOs, they remain standard baselines in prior evaluations (cf. Table 1). Our evaluation suite builds upon these benchmarks to probe distinct optimizer behaviors, with JOB as the core workload due to its real-world complexity and challenge to traditional heuristics. JOB-Complex, CEB, and SQLStorm complement JOB in distinct ways: JOB-Complex introduces additional real-world complexity, CEB targets deep-learning models through rich query sets and execution scenarios, and SQLStorm provides broad SQL construct coverage via a diverse workload. Conversely, the TPC benchmarks utilize rigid schemas, predictable distributions, and decision-support queries, while STATS is very fitting for evaluating covariate shift. Table 4 summarizes our evaluation benchmarks using the previous dimensions.

4.3 LQOs Integration within the Suite

This subsection examines the LQOs' code structure showing how they interact with the DBMS in practice and to highlight

Table 5: Hardware Setups Across Different Papers

System	CPU	RAM	Storage	GPU
NEO	Intel Xeon E5-2640 v4	32GB	SSD	Not specified
BAO	Intel Xeon Gold 6230	15GB (VM) + 256GB (host)	Not specified	Tesla T4
LOGER	2× Intel Xeon Gold	256GB	Not specified	NVIDIA RTX 3090
FASTgres	Intel Xeon Gold 6216 (12 cores)	92GB	1.8 TiB	Not specified
LERO	Intel Xeon Platinum 8163 (96 cores)	512GB	1TB SSD	NVIDIA RTX 2080 Ti
Our Setup	QEMU Virtual CPU v2.5	110GB	2.0 TiB SSD	NVIDIA RTX A6000

Table 6: PostgreSQL Configs across different papers.

PostgreSQL Config Param.	Default	BAO	BALSA	LOGER	LERO	Ours
Join Order						
geqo_threshold	12	-	-	1,024	-	-
geqo	on	-	off	off	off	off
Working Memory						
work_mem	4 MB	-	4 GB	-	-	4 GB
shared_buffers	128 MB	4 GB	32 GB	64 GB	4 GB	32 GB
temp_buffers	8 MB	-	32 GB	-	-	32 GB
effective_cache_size	4 GB	-	-	-	-	64 GB
Parallelization						
max_parallel_workers	8	-	-	1	0	8
max_parallel_workers_per_gather	8	-	-	1	0	8
max_worker_processes	2	-	8	-	-	8
Scan Types						
enable_bitmapscan	on	-	off	-	-	on
enable_tidscan	on	-	off	-	-	on

the extra engineering required to operate within our evaluation framework.

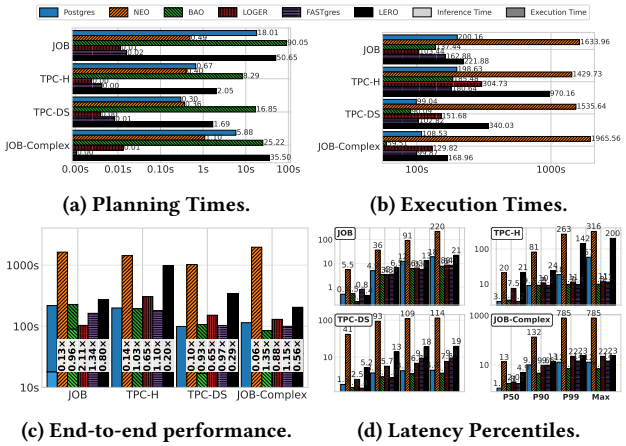
Tested LQOs fall into two groups based on their degree of interaction. NEO, FASTgres, and LOGER operate mostly standalone, requiring minimal runtime communication with PostgreSQL: **(a)** queries are passed directly to the LQO, **(b)** an optimization strategy[1]is derived, **(c)** translated into hints and executed by PostgreSQL using `pg_hint_plan`. In contrast, BAO and LERO integrate directly with PostgreSQL through planner hooks: **(a)** queries are submitted to PostgreSQL, **(b)** intercepted before optimization, **(c)** LERO injects alternative cardinality estimates and BAO evaluates available hint sets, and **(d)** the selected plan is executed by PostgreSQL.

Code additions. In this passage we summarize the shared functionality implemented across LQO repositories. Specifically, we **(a)** ensured model-checkpoints throughout training, used later for evaluation **(b)** added hooks within LQOs models to analyze plan embeddings and performance predictions, **(c)** enabled training with different workload orders. Finally, LERO's evaluation required disabled parallel workers, consistent with its original configuration (Table 6), thus its results reflect this setting. Parallelism was retained for all other LQOs to avoid penalizing systems that support it.

5 Evaluation Results

We evaluate each LQO across five experimental dimensions: **(a)** end-to-end performance and value-model fidelity, **(b)** sensitivity to query ordering and execution-time stability, **(c)** learning dynamics and convergence properties, **(d)** plan representation and its effect on operator selection, and **(e)** generalization capability.

[1]NEO predicts join order and operators via plan search, while LOGER and FASTgres infer join-order restrictions and optimal hint sets, respectively.



(a) Planning Times. (b) Execution Times.

(c) End-to-end performance. (d) Latency Percentiles.

Figure 2: Planning and exec. times across benchmarks and LQOs.

Table 5 lists the hardware specifications used in each LQO publication, including our own hardware. Similarly, Table 6 includes the configurations used in each LQO's paper, along with the set we utilize (denoted as "Ours") for PostgreSQL v12.5. Our configuration set was first introduced in [32]. PostgreSQL was selected as the sole DBMS compatible with our tested LQOs.

5.1 End-to-End Perform. & Value Model Fidelity

We evaluate all optimizers in a single, unified setup that mirrors their original evaluation to ensure a fair comparison. For each query workload, we compare learned optimizers to PostgreSQL by examining two aspects: *(a)* planning and execution latency and *(b)* prediction accuracy. We conduct these experiments on four benchmarks: JOB, TPC-DS, TPC-H, and JOB-Complex. For each benchmark, each LQO is trained for the number of epochs expected to achieve performance comparable to PostgreSQL. During evaluation, each optimizer executes the full workload three times to account for variability in query execution times.

5.1.1 End-to-End Performance. We analyze each LQO's end-to-end performance, focusing on the trade-off between planning overhead and plan quality (i.e., query execution latency). All speedups/slowdowns are reported relative to PostgreSQL for each workload.

Planning Times. We isolate planning overhead by analyzing Fig. 2a and highlighting the systems with notable cost. LERO and BAO show notably high overhead, up to 5× that of PostgreSQL. Postgres shows varying planning times, 18.01s for JOB, but only 0.14s and 0.61s for TPC-H and TPC-DS, due to JOB's significantly larger plan search space. The remaining LQOs require a small overhead: NEO takes 0.49s for JOB, while LOGER and FASTgres are lightweight.

Execution Latency. We evaluate plan quality by analyzing cumulative execution latency (Fig. 2b). NEO demonstrates latencies roughly 7x slower than the baseline across the board. BAO, FASTgres, and LOGER show varying performance across workloads. On JOB, all three achieve speedups (1.6×–2.0×) against Postgres, but this does not generalize. BAO and FASTgres barely match baseline latency on TPC workloads, while LOGER performs worse, with a 0.65× slowdown. LERO exemplifies this trend, roughly matching the baseline on JOB but incurring severe slowdowns (0.2×–0.33×) on both TPC benchmarks. Notably, on

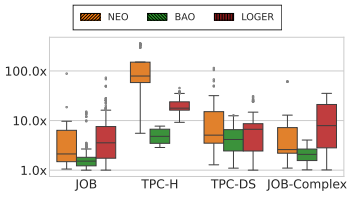
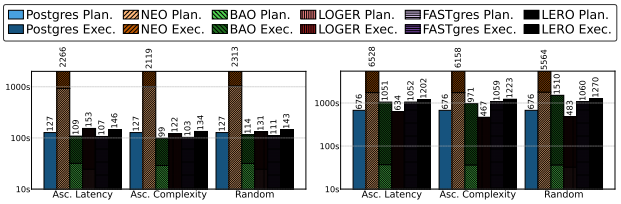


Figure 3: Prediction Q-Error distr. for LQOs across workloads.



(a) Join Order Benchmark **(b) Cardinality Est. Benchmark**

Figure 4: Performance by Workload Order (Log Scale, (R2.D7,R1.D8) aggregated by train/test splits).

JOB-Complex, BAO achieves a 1.83× speedup, while LOGER underperforms with a 0.83× slowdown. This behavior is examined further in Section 5.4.

Overall speedup. By combining planning and execution costs (Fig. 2c), we evaluate end-to-end LQO benefits relative to the classic optimizer and find that no system consistently outperforms the baseline. NEO and LERO exhibit major slowdowns on JOB and TPC-H, where BAO's high planning overhead yields only a marginal 1.03× speedup on TPC-H, FASTgres remains consistently reliable, peaking at 1.34× on JOB and LOGER achieves a 2.11× speedup on JOB but incurs a 0.65× slowdown on TPC-H.

Latency Percentiles. Fig. 2d summarizes execution-time distributions across benchmarks using median and tail latencies. NEO is consistently an order of magnitude slower than the baseline. At median latency (P50), BAO and FASTgres match or outperform the baseline, while LERO and LOGER exhibit substantial slowdowns on TPC workloads, consistent with prior results. At the tail (P90–Max), BAO, FASTgres, and LOGER achieve the largest gains, explaining the speedups observed in earlier analyses.

Verdict. Planning overhead is affected by LQO design choices. LOGER minimizes the search space by limiting operator selection, while FASTgres leverages its precomputed contexts. Conversely, BAO and LERO show elevated planning times, as during runtime BAO evaluates all hint combinations and LERO injects cardinality estimates before actually selecting a candidate.

LERO's TPC slowdown highlights the workload-dependence of LQO performance and the limited headroom for LQO improvements on certain benchmarks. Here, LERO's cardinalities degrade plan quality relative to the baseline, whose estimates are accurate due to the synthetic data. In contrast, the higher complexity and real-world correlations of JOB and JOB-Complex offered substantially greater improvement potential for LQOs, which led to notable speedups.

5.1.2 Value Model Accuracy Assessment. We address whether a highly accurate Value Model is key to LQO success by analyzing prediction Q-Error [47] (Fig. 3) for BAO, LOGER, and NEO. LERO and FASTgres are excluded, as their learning-to-rank and classification objectives do not produce plan latency predictions at runtime, making Q-Error analysis inapplicable.

BAO's Value Model shows reasonably accurate latency predictions, with a median Q-Error typically between 2×–5×. NEO's Value Model fails to converge effectively, especially on TPC benchmarks, often predicting near-zero latency for slow plans, leading to severe performance drops noted in Section 5.1.1.

LOGER's strong performance on JOB, despite higher Q-Error, shows that predictive accuracy is neither necessary nor sufficient for robust end-to-end optimization, because effectiveness depends on how the value model is used. BAO ranks complete plans and thus requires accurate latency estimates, whereas LOGER employs its model to prune inferior operators while delegating final plan selection to PostgreSQL. Accordingly, LOGER need only capture relative plan quality, not exact latencies, which it achieves through a well-structured embedding space, as demonstrated in Section 5.4.4.

Verdict. While a flawed Value Model guarantees failure, low prediction error alone does not ensure strong performance. LOGER's success on JOB, despite higher Q-Error than BAO, shows that effectiveness depends on the model's role. For tasks like pruning poor choices, relative quality matters more than precise latency estimates, leading to the conclusion that prediction accuracy does not always lead to speedups.

Key Takeaways from Experiment 1:

- **Benchmarks Influence LQO Performance**: No LQO performs best across all benchmarks; effectiveness varies significantly with workload structure and data characteristics.
- **LQOs Excel Where Classic Optimizers Struggle:** LQOs offer the greatest benefits on workloads that challenge classic optimizer assumptions, otherwise PostgreSQL remains competitive.
- **Prediction Accuracy Requires "Smart" Usage:** A system with less precise estimates may outperform a more precise system, depending on the model objective.

5.2 Sensitivity & Execution Stability

This experiment aims to *(a)* assess LQO performance under different train/test split philosophies, *(b)* investigate the impact of the order in which training queries are presented and *(c)* measure how consistently LQOs produce plans of the same quality across multiple executions of the same query. We evaluate LQOs by systematically varying train/test splits and training orders using i) JOB and ii) a subset of CEB introduced in [16] that maximizes the diversity of query structures in CEB's workload. We take a step further from [32] on the splitting strategies, by introducing the impact of query order during training, and how sensitive each LQO is to different presentation sequences.

5.2.1 Performance with Different Train/Test Split Strategies and Workload Orderings. Fig. 4 reveals performance trends of LQOs across workload orderings. In both figures (Figs. 4a and 4b), we firstly observe that PostgreSQL optimizer is consistently efficient across configurations. FASTgres also stands out for its robustness, as it was consistently one of the better performing LQOs, whilst also exhibiting only an 8 second variation between the best and worst performing workload orders in both occasions. This resilience can be attributed to its context-based classification approach based on dedicated decision trees per query structure, which remain largely stable across the different workload orderings.

Regression-based LQOs on the other hand demonstrate an entirely different pattern of behavior. More specifically, LOGER performs best with ascending complexity order, leveraging a simpler set of queries first to build foundational knowledge. In contrast, ordering by latency introduces a weaker learning signal overall, where queries with significant structural diversity all result in latencies within the same order of magnitude. This trend also appears for both BAO and, to a lesser extent, NEO, suggesting that sensitivity to query order is a general trait for the regression-based systems. Finally, for LERO, the aforementioned pattern reappears on Fig. 4a, where the ascending complexity ordering outperformed its two counterparts, yet this did not generalize to Fig. 4b.

Verdict. Workload ordering significantly impacts regression-based LQOs, with ascending complexity leading to the most stable learning, while latency-based ordering degrades performance. In contrast, FASTgres remains robust across orderings due to its context-based classifiers, highlighting that sensitivity to training order is value model-dependent.

5.2.2 Workload Robustness. This experiment assesses LQO stability by measuring consistency when executing the same query multiple times under identical conditions.

Table 7: Top 4 Most Variable Queries for NEO

Optimizer	Query	Training Regime	Range (s)*
NEO	7c	Leave One Out, Asc. Latency	104.38
NEO	20c	Leave One Out, Asc. Latency	44.43
NEO	14a	Leave One Out, Random Order	35.07
NEO	8c	Random Split, Random Order	9.86

*Range: The abs. diff. between fastest and slowest query execution.

Table 8: Top 4 Most Variable Queries for Learned Assistants

Optimizer	Query	Training Regime	Range (s)
BAO	30a	Leave One Out, Random Order.	5.34
LOGER	8c	Random Split, Asc. Latency	2.41
BAO	25c	Random Split, Random Order	2.27
BAO	25c	Random Split, Asc. Latency	2.13

*Range: The abs. diff. between fastest and slowest query execution.

NEO exhibits the highest execution-time variability (up to 104s - Table 7), an order of magnitude above all other systems. In contrast, learned assistants show minimal variance (up to ~5.3 s - Table 8).

Verdict. This difference stems from plan stability: for BAO, FASTgres, LERO, and LOGER, optimization decisions (e.g., hints and operator pruning) remain consistent across runs, yielding stable latencies, whereas NEO's Value Model fails to reliably predict the impact of join ordering and operator choices during plan search, producing different QEPs and high variability. Overall, hint-based approaches are more robust than end-to-end optimization.

Key Takeaways from Experiment 2

• **Regression Models Benefit from Ascending-Complexity Training**: Training with ascending complexity fosters stable, effective learning by capturing foundational patterns early, while latency-based ordering introduces noise and yields weaker results.

• **Steering Assistants Provide More Consistent Optimization**: Systems that provide optimization hints tend to reuse their directives consistently across consecutive query executions. NEO demonstrates unstable behavior, producing divergent query execution plans as a consequence of an unreliable Value Model.

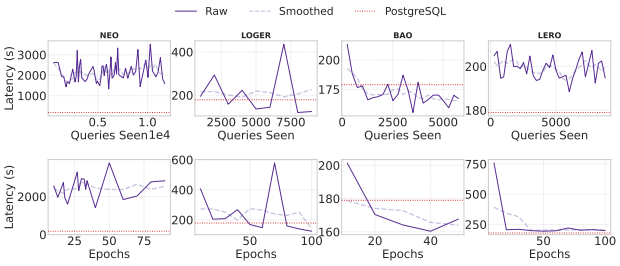


Figure 5: Latency vs Epochs and Q. Seen across selected optimizers.

5.3 Learning Trajectory & Convergence

This experiment examines how training progress metrics and stopping criteria affect convergence speed and final performance across LQO architectures. We evaluate each LQO on JOB using the Random Split + Ascending Complexity setup from Section 5.2. Models saved under each training policy are evaluated on a held-out test set, with each query executed three times and average latency reported. We consider the following three training policies:

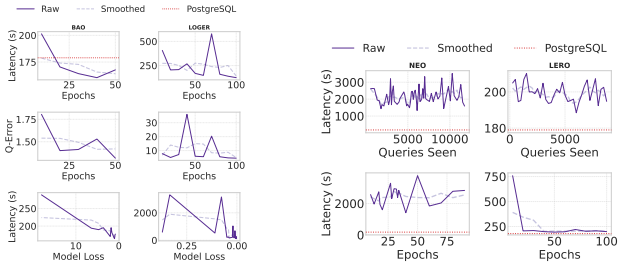
- **Epoch-Based:** Progress is measured in epochs, where each epoch corresponds to a full pass over the entire training workload, with checkpoints saved every 10 epochs.
- **Loss-Based:** Progress is tracked via validation loss, with checkpoints saved when the loss improves by a fixed threshold (0.05).
- **Queries-Seen:** Progress is measured by the number of processed training queries, with checkpoints saved after fixed query-count intervals.

Again, LERO and FASTgres are excluded from Q-Error analysis due to their learning-to-rank and classification objectives, resp.

5.3.1 Learning Trajectory Analysis. Our findings highlight how each LQO's architecture dictates system behavior throughout training. For almost all LQOs, the *Queries Seen* (Fig. 5 - top) policy ultimately matched the *Epoch-based* (Fig. 5 - bottom) policy in final performance, however its trajectory was considerably noisier.

NEO's training loss (Fig. 6b - bottom left) at no time dipped below the 0.939 mark, pointing to a persistent failure to learn effective policies. This clearly explains the observed latency spikes and instability across experiments. Yet BAO (Fig. 6a) exhibits stable learning behavior, enabling analysis across training policies. Under the *Epoch-based* policy, it initially improves latency from 201.42s to 160.38s before regressing slightly. Notably, the lowest latency (epoch 40) coincides with a Q-Error peak, while the lowest (epoch 50) aligns with the latency regression, supporting the idea that lower prediction error does not always yield better plans (Section 5.1.2). The *Loss-Based* policy however, delivers consistent improvements, with latency decreasing steadily as loss improves.

LOGER (Fig. 6a) exhibits similar spikes under both *Epoch-* and *Query Seen* policies, suggesting that fixed training is insufficient and convergence should guide termination. Notably, Q-Error mirrors latency trends (e.g., epochs 40 vs. 60), likely due to its structured embedding space, explored further in Section 5.4.4. Finally, LERO's *Queries Seen* signal is noisier than the *Epoch-based*



(a) BAO, LOGER (b) NEO, LERO

Figure 6: Learning trajectories across optimizers.

policy (Fig. 6b - bottom vs top right), though neither surpasses the classic optimizer.

Verdict. Training procedures and stopping criteria are highly depend on the underlying LQO architecture. Metrics such as Q-Error are not fully reliable indicators of end-to-end performance: LOGER's behavior is aligned with metric behavior, but BAO did not produce better plans despite lower prediction errors. Among the three policies, the loss-based one provided the most stable and consistent convergence for regression-based LQOs, while epoch- and Queries-Seen-based ones introduced noise. This stems, in part, from the fact that the number of queries or epochs required for convergence is unknown *a priori*.

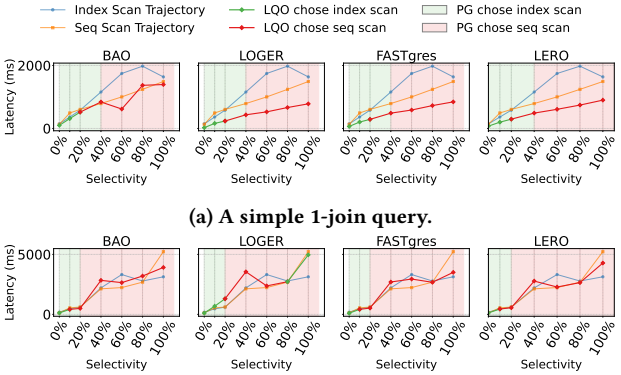
Key Takeaways from Experiment 3

- **Training Progress Depends on LQO Architecture.** Different architectures result in completely different learning dynamics.
- **Align Training Policy with Learning Objective.** Epoch and query-seen policies fail to capture meaningful progress. Systems should adopt policies grounded in the LQO's objective and design.
- **Performance Stability Requires Convergence**: Converged training produces consistent latencies and plans; failure to converge leads to spikes and unstable optimization outcomes.
- **Training Metrics Are Poor Proxies for Execution Latency**: Observed gaps between training progress and execution latency indicate that performance depends on how the value model is used for optimization.

5.4 Operator Selection & Plan Representation

We leverage the access path and physical operator analysis for Learned Cost Models by [20] to investigate how LQOs optimize a query by systematically analyzing their internal decisions. We: *a)* evaluate access-path selection under varying predicate selectivities and measure the regret the suboptimal decisions incur, *b)* test their ability to pick appropriate join operators (e.g. hash vs. nested-loop), as query complexity grows, *c)* compute prediction accuracy, prediction error magnitude and error frequency per join type, and *d)* study how distances in learned plan embeddings relate to performance gaps, especially between LQOs plans and PostgreSQL.

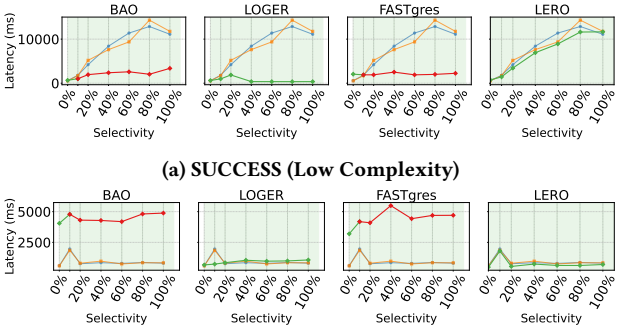
5.4.1 Access Path Selection. We evaluate the accuracy of LQOs in selecting scan operators (sequential vs. index scan) and their sensitivity to predicate selectivity across three training datasets: JOB, JOB-Synthetic [29], and JOB-Complex [67]. Separate models are trained for each optimizer on all three datasets. The inclusion of JOB-Complex, which contains string-based predicates known to exacerbate cardinality estimation errors compared to integer



(a) A simple 1-join query.

(b) A 3-join query.

Figure 7: Optimizer scan decisions on the synthetic benchmark. Background indicates classic optimizer choice per selectivity level; Alternating line between green and red shows LQO choice.



(a) SUCCESS (Low Complexity)

(b) FAILURE (Medium Complexity)

Figure 8: `BAO`'s and `FASTgres`' aggressive decision-making leads to mixed results, depending on query complexity.

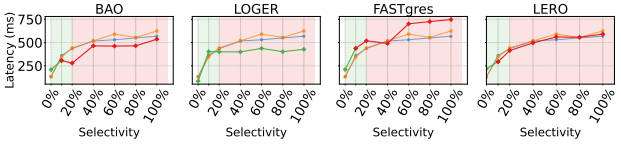


Figure 9: High Complexity (16 Joins)

attributes [31, 33, 57], enables a more challenging evaluation setting. For this experiment, we create an index on the column `title.production_year` and select test queries with a range predicate (`title.production_year < ?`)), evaluated at six selectivity levels: 10%, 20%, 40%, 60%, 80%, and 100%.

Each query is executed three times with the classic optimizer under different scan settings: **(i)** default, **(ii)** forced sequential scan, and **(iii)** forced index scan. We record runtimes to evaluate whether the optimizer correctly identifies the selectivity threshold at which index scans outperform sequential scans. Each query is also executed once with each LQO without restrictions, enabling direct comparison of switch-point detection across selectivity levels.

JOB-Synthetic. On single-join queries (Fig. 7a), the optimal crossover occurs at 20% selectivity. PostgreSQL switches later (40%), while all LQOs correctly identify the optimal point. For three-join queries (Fig. 7b), index scans regain efficiency at 100%

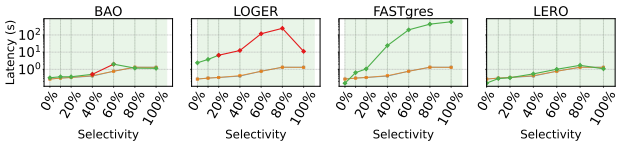
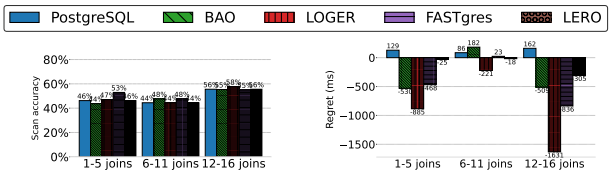


Figure 10: Severe regressions for JOB-Complex (log scale)



(a) Scan selection decision acc. **(b) Mean Absolute Regret.**

Figure 11: Optimizer performance on access path selection, grouped by query join complexity (Join Order Benchmark)

.

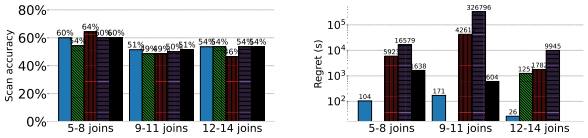
selectivity, contradicting the common heuristic favoring sequential scans at higher selectivities. LOGER uniquely captures this non-monotonic behavior, correctly switching back to index scans.

JOB. NEO is excluded due to severe regressions caused by poor join ordering. As shown in Fig. 8, BAO and FASTgres aggressively favor sequential scans at higher selectivities, yielding strong gains for simpler queries (Fig. 8a) but causing significant slowdowns in more complex cases (Fig. 8b), where PostgreSQL's conservative strategy is near-optimal. Fig. 9 highlights LOGER's robustness, where it consistently selects index scans in a sixteen-join query, outperforming both PostgreSQL and other LQOs, which persist with sequential scans and miss the optimal plan.

JOB-Complex. Fig. 10 encapsulates a frequent LQO pattern on the JOB-Complex based workload. BAO and LERO, who almost always follow PostgreSQL's example of selecting the Index Scan, both register consistently moderate performances, with latencies that are within the same order of magnitude as the baseline. LOGER and FASTgres on the other hand are often susceptible to severe regressions, in part due to suboptimal decisions like resorting to sequential scans on this query set.

Aggregated Scan Analysis. Figs. 11 and 12 summarize scan-selection accuracy across workloads and query complexities. Overall accuracy remains modest (45–60%), indicating that JOB substantial room for improvement in terms of access-path selection, however, LQOs only marginally outperform PostgreSQL, thus this potential ultimately remains untapped. In JOB (Figs. 11a and 12a), hint-based optimizers show both large gains and costly errors, while LOGER delivers the most consistent improvements. JOB-Complex (Fig. 12b), accumulated mispredictions lead to extreme regressions for LOGER and FASTgres, suggesting difficulty in modeling string-based predicate cardinalities. BAO and LERO perform more reliably, likely due to reusing PostgreSQL's cardinality estimates.

Verdict. Accurate access-path selection remains challenging, with LQOs achieving only modest gains over PostgreSQL. The systems that encode cardinality estimates in plan node level, are more robust than the others. This is shown especially in JOB-Complex, which aims to filter mostly on string predicates and is hard to model in the LQOs, the learned models struggle to generalize.



(a) Scan selection decision acc. **(b) Mean Abs. Regret (log scale).**

Figure 12: Access path selection (JOB-Complex).

5.4.2 Physical Operator Composition. We study join operator selection as query complexity increases by grouping JOB and JOB-Complex queries by join count, executing them with both classic and learned optimizers, extracting the distribution of selected physical join operators, and measuring relative speedup over the classic optimizer. This analysis captures how operator choices evolve with complexity and their impact on performance.

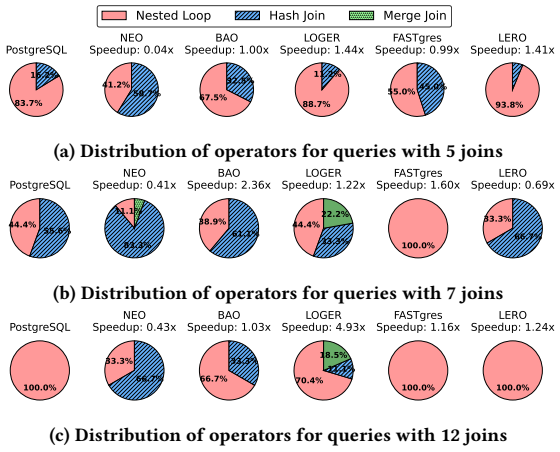
Figs. 13 and 14 showcase the following trends. Firstly, the classic optimizer demonstrates a dynamic, context-aware behavior that sets a high-performance baseline for LQOs to surpass, especially in Fig. 14. Moreover, as was observed in previous experiments, NEO's Value Model never converged, leading to consistently making poor join operator decisions. While its operator distributions may appear balanced, analysis reveals frequent use of suboptimal choices, like Merge Joins, leading to severe slowdowns (0.04×–0.43× speedups).

JOB. Instances of significant speedups recorded across Fig. 13 underline the potential headroom JOB offers to LQOs in selecting more sophisticated join distributions than the baseline. LOGER emerges as the top performer, showcasing a strong ability to learn superior strategies. In Fig. 13c, while PostgreSQL relies solely on NL Joins, LOGER identifies a more efficient operator mix, achieving a 4.93× speedup. BAO and LERO show a blend of outcomes: (a) successful learning (e.g., BAO's 2.36× speedup in Fig. 13b), (b) average performance matching PostgreSQL (e.g., Fig. 13c), and (c) clear regressions (e.g., LERO in Fig. 13b, or BAO's 0.35× speedup on 13-join queries). This inconsistency suggests that while both models can exploit optimization opportunities, they struggle to generalize as join complexity and search space grow.

Consistent with its context-based classification approach, FASTgres often commits to a single join operator per workload context, regardless of changing conditions. For instance, in Figs. 13b and 13c, it exclusively selects Nested Loop Joins, unlike LOGER, which adapts with a diverse operator mix and achieves remarkable performance. This rigidity, when applied uniformly, can lead to suboptimal decisions across varied query contexts.

JOB-Complex. While some LQO patterns carry over from JOB's analysis, i.e. NEO's severe slowdowns, LOGER's capacity for detecting efficient join operators (Fig. 14a) and LERO's moderate performance, Fig. 14 gives us two extra insights into LQO behavior. More specifically, BAO exclusively selected the Hash Join operator in this setting, which paid dividends and yielded a significant speedup, while FASTgres and LOGER struggled to find good operator balances (e.g., Fig. 14b), due to the increased challenge JOB-Complex poses with its non PK-FK and string column joins.

Verdict. Join operator selection is very architecture-dependent, being increasingly difficult as query complexity grows. While some LQOs can provide significant speedups, these gains do not generalize as join depth and structural diversity increase. In the

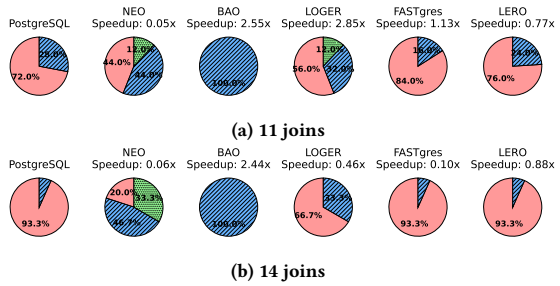


(a) Distribution of operators for queries with 5 joins

(b) Distribution of operators for queries with 7 joins

(c) Distribution of operators for queries with 12 joins

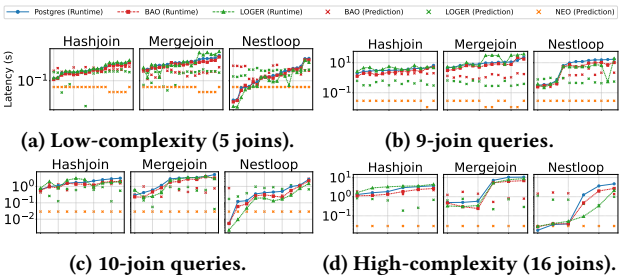
Figure 13: Join operator distribution by query complexity; title shows speedup vs. PostgreSQL (Join Order Benchmark)



(a) 11 joins

(b) 14 joins

Figure 14: Join operator distribution (JOB-Complex).



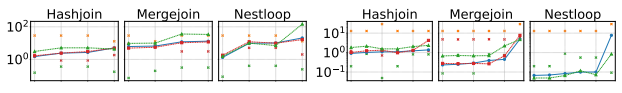
(a) Low-complexity (5 joins).

(b) 9-join queries.

(c) 10-join queries.

(d) High-complexity (16 joins).

Figure 15: Comparison of latency predictions across varying query join complexities (Join Order Benchmark).



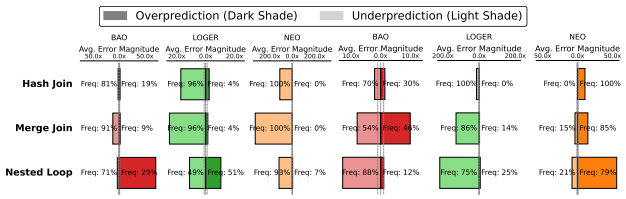
(a) 6 joins

(b) 8 joins

Figure 16: Join operator latency predictions (JOB-Complex).

most extreme case of string-based and non–PK-FK joins, operator selection further degrades.

5.4.3 Join Operator Prediction Accuracy. We assess LQO prediction accuracy under controlled settings by grouping JOB and JOB-Complex queries by join count and executing each group with enforced join operators (nested-loop, hash join, merge join). Predicted latencies are compared against measured runtimes to compute relative error. The inclusion of JOB-Complex enables evaluation under non–PK-FK and string-based join conditions.



(a) JOB

(b) JOB-Complex

Figure 17: Prediction error magnitude (bar length) and frequency (inline text) by optimizer and join type.

LERO and FASTgres are excluded due to their learning-to-rank and classification objectives respectively, as during runtime they do not make plan latency predictions. As shown in both Figs. 15 and 16, NEO's Value Model again consistently misestimates latency across all join operators, predicting non-accurate runtimes and offering no meaningful guidance for operator trade-offs.

JOB. For low-complexity queries (Fig. 15a), both BAO and LOGER closely track actual latencies, indicating effective modeling. However, at mid-level complexity (9–10 joins; Figs. 15b and 15c), differences emerge. BAO's accuracy appears tied to PostgreSQL's cardinality estimates—performing well on Hash Joins (which are typically robust to CE errors [33]), but underestimating costs for Merge Joins and, to a lesser extent, Nested Loops. Those underestimations ultimately reflect its reliance on the classic optimizer's CE quality, especially in more complex scenarios.

Unlike BAO, LOGER does not reuse PostgreSQL's cardinality estimates. Instead, it learns patterns over tables, columns, and predicates via a Graph Transformer capturing global query structure. This design often underestimates Hash and Merge Joins but yields more balanced predictions for Nested Loop Joins, which are common in the workload (cf.cFig. 15), indicating a deeper data-driven understanding. For the most complex queries (Fig. 15d), predictive accuracy drops sharply across all LQOs: BAO and LOGER, which previously handled Nested Loops well, now significantly overestimate their costs. In BAO, this stems from increasingly unreliable PostgreSQL cardinality estimates, particularly for NL Joins, while LOGER continues to underestimate Hash and Merge Join costs.

JOB-Complex. Throughout Fig. 16 the following patterns are noticeable. LOGER appears to struggle in most cases, regardless of join operator and complexity, while BAO's predictions are accurate initially but later degrade with the increasing complexity, which again is in-line with the classic optimizer's diminishing CE accuracy.

Aggregated analysis of the join decisions. Fig. 17 aggregates the prediction error magnitude and frequency for each optimizer and confirms the operator-specific tendencies observed previously. NEO's predictions are orders of magnitude off the actual costs, while BAO, despite struggling immensely with modeling NL Joins, is able to accurately predict HJ and (to a certain extent) MJ costs, which also explains its tendency for leveraging Hash Joins in Fig. 14.

Finally, LOGER consistently underestimates the non-NL join operators, to even an extreme degree (100x) for the Merge Join Op. in Fig. 17b. Yet, its LOGER's behavior for the NL operator in both instances that completes the picture. The extremely

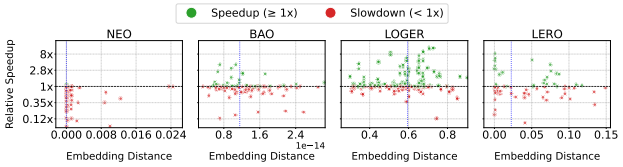
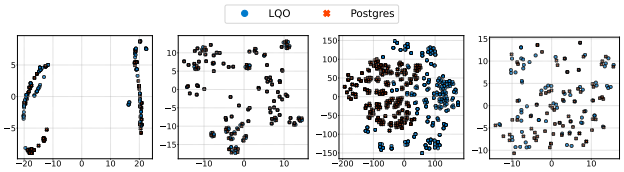


Figure 18: Cosine embedding distance (x-axis) vs. execution time difference (y-axis) for PostgreSQL plans against each LQO



(a) NEO (b) BAO (c) LOGER (d) LERO

Figure 19: t-SNE of plan emb.: blue = LQO, orange = PostgreSQL.

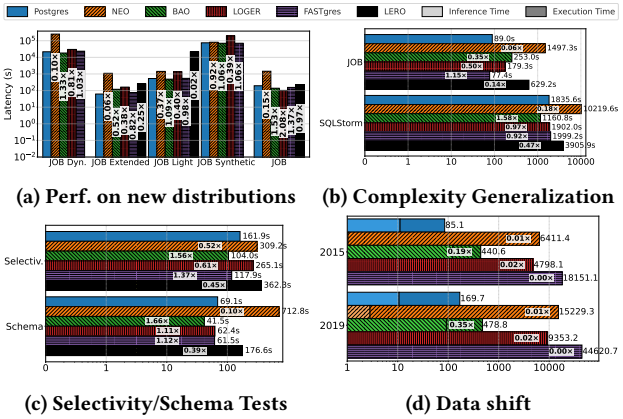
balanced NL join predictions in Fig. 17a, which happens to be the workload's most frequent operator (see Fig. 13), highlight LOGER's overall potential by showcasing a fundamental understanding of the dynamics between the dataset and the workload. This also suggests that this could generalize to the remaining join operators for LOGER if the training process was extended. Conversely, LOGER's NL join predictions degrade immensely in Fig. 17b, signaling that the LQO was unable to cope with the non PK-FK and string column joins prominent in JOB-Complex's workload. This also explains the severe underestimation of the MJ operator in Fig. 17b, as well as LOGER's performance dropoff for JOB-Complex throught Section 5.1.

Verdict. Join operator prediction accuracy is strongly operator and workload-dependent and degrades with query complexity. Systems that rely on cardinality-driven signals inherit their strengths and weaknesses across operators, while relative-cost models generalize only to dominant operator patterns. In the extreme case of string and non PK-FK joins, prediction accuracy declines, making the LQOs predictions inaccurate.

5.4.4 Embedding Similarity and Plan Quality. We examine the relationship between plan embedding similarity and execution performance. For each JOB query and LQO, we compare the LQO-generated plan against the classic optimizer's plan executed within the LQO, retaining embeddings for both. Embedding divergence is then correlated with performance differences using cosine similarity and t-SNE [4] visualizations to assess whether the embedding space reflects plan quality.

Fig. 18 plots cosine distance between PostgreSQL and LQO embeddings against actual speedup. Ideally, similar-performing plans (speedup ≈ 1) should have low distances, with larger deviations scaling linearly. Likewise, Fig. 19 shows t-SNE projections, where ideal embeddings cluster plans by structural quality, not origin, grouping good and poor plans separately.

LOGER's Value Model most closely matches ideal behavior, showing the strongest (though imperfect) linear correlation between embedding distance and performance (Figs. 18 and 19c). Its t-SNE projection reveals structured clustering, with high-speedup plans forming a distinct cluster and slower plans overlapping PostgreSQL baselines. In contrast, NEO fails to learn meaningful embedding–cost relationships: its embeddings show minimal runtime correlation, map dissimilar plans to similar embeddings,



(a) Perf. on new distributions (b) Complexity Generalization

(c) Selectivity/Schema Tests (d) Data shift

Figure 20: Robustness experiments. Model performance under unseen queries, schema shifts, and data distribution changes.

and collapse in t-SNE, which arises from query encoding dominating plan encoding, causing the Value Model to prioritize query features over plan structure.

Both BAO and LERO, despite sharing a similar Value Model architecture with NEO, avoid failures by omitting query encoding and relying solely on their plan embeddings. As shown in Figs. 18 and 19, said LQOs capture plan performance partially but imprecisely, landing between LOGER's structured success and NEO's breakdown, with noticeable noise in its embedding space.

These results suggest that LOGER's integration of a Tree-LSTM with a Graph Transformer learns structured embeddings more effectively that reflect relative plan quality, outperforming the Tree-CNN architecture used by other regression-based optimizers.

Verdict. An LQO's internal representation quality is key to its effectiveness. Structured embedding spaces that preserve relative plan quality, enable speedups, while poor ones lead to performance degradation.

Key Takeaways from Experiment 4

- **Hint-based LQOs Learn Aggressive Strategies:** They often achieve large speedups by identifying cost crossover points missed by traditional optimizers. However, their aggressive behavior can lead to severe regressions when misapplied.
- **Access-Path Selection Remains Challenging**: LQOs get only modest improvements over PostgreSQL, while being vulnerable to workload characteristics. Estimates directly from the system, are a strong signal for reliable access path selection.
- **Context-Sensitive Join-Operator Prediction Accuracy**: As queries grow more complex, accuracy declines. Models tied to cardinality signals reflect estimator biases, whereas ranking approaches generalize to dominant operator patterns in training.
- **Representation Quality Drives LQO Effectiveness:** Structured internal plan representations preserve relative plan quality, where in other cases lead to performance degradation.

5.5 Generalization to Novel Conditions

We evaluate LQO generalization and adaptivity along three axes: *(a)* generalization to unseen queries, *(b)* adaptability to schema changes, and *(c)* robustness to covariate shifts. Planning times are omitted unless explicitly analyzed (Fig. 20d). LERO is excluded from large-query workloads (JOB-Dynamic, JOB-Synthetic, Stack) due to prohibitive planning overhead.

5.5.1 Generalization to Unseen Queries. We assess generalization under *distributional*, *complexity*, and *selectivity* shifts.

Distributional Shifts. We extend the "workload distribution" definition first introduced in [82], to evaluate LQO performance under fair distribution shift by characterizing each workload via aggregated query features (e.g., selected tables, join patterns, predicate types) and quantifying dissimilarity using Jensen–Shannon (JS) divergence [44]. LQOs are trained on JOB and evaluated on its variants, following completely different distributions (Light [29], Extended [38], Dynamic [76], and Synthetic [12]).

LOGER's performance (Fig. 20a) degrades with increasing divergence, performing best on JOB-Dynamic (0.72× speedup, JS=0.44) and worst on JOB-Light (JS=0.82) and JOB-Extended (JS=0.77). FASTgres remains close to the baseline, while BAO achieves modest gains on simpler or familiar workloads (1.09× JOB-Light, 1.06× JOB-Synthetic, 1.19× JOB-Dynamic) but degrades sharply on JOB-Extended. NEO exhibits severe regressions and instability, while LERO shows strong overfitting.

Complexity Shifts. We evaluate join complexity based on the number of join within the queries (training on <10 joins, testing on ≥10) and structural complexity using SQLStorm, excluding queries with CTEs and aggregations for compatibility, (ii) classify the remaining queries using the methodology of [56], and (iii) train on the *medium*-complexity subset and test on the corresponding *high*-complexity subset.

Join-based complexity appears more challenging, which correlates with the larger train–test join distribution distances[2](JOB = 0.49 vs. SQLStorm = 0.29). Evidently, systems trained on SQLStorm benefited from substantial join overlap, enabling effective transfer, whereas JOB's limited overlap led to degradations. On this latter test, BAO and LOGER under-perform, while FASTgres achieves a speedup of 1.36×.

Selectivity Shifts. In Fig. 20c we evaluate LQOs trained on low-selectivity JOB-Extended queries (0–20%) and testing on high selectivity (60–100%). BAO achieves the strongest gains (1.56×) due to reliance on PostgreSQL cardinality estimates and FASTgres improves (1.37×), while the rest fail to adapt effectively.

Verdict. Depending on the distribution shift increase, LQO performance degrades, as systems struggle to generalize their knowledge. Fastgres is the sole exception, as its context-based design can reconstruct decision trees for new query contexts.

The complexity shift test boiled down to train–test join distribution distance, which indicates that join distribution is the most influential aspect of workload generalization. Here, LOGER and BAO struggle to an uncharacteristic degree, as LOGER's GT fails to transfer its knowledge to the larger join graph, while BAO's learned hints are now obsolete.

Systems whose encoding included PostgreSQL cardinality estimations gained an advantage in selectivity generalization.

5.5.2 Schema and Data Distribution Shift Adaptation. **Schema Changes.** LQOs are trained on JOB excluding the `char_name` table and tested on queries joining this unseen table (Fig. 20c). BAO and FASTgres adapt moderately, benefiting from schema-agnostic features, context-based inference, and graph-based representations, respectively. In contrast, NEO and LERO, relying on one-hot schema encodings, suffer severe slowdowns.

Temporal Data Shifts. In Fig. 20d we showcase models trained on early snapshots of STATS (2008–2011) are tested on later ones (2008–2015, 2008–2019). FASTgres and LOGER incur extreme slowdowns (0.02×–0.04×) due to outdated contexts and ineffective pruning. BAO degrades less (0.43×–0.45×) by leveraging live PostgreSQL statistics, while NEO and LERO again exhibit extreme latency.

Verdict. Systems relying on one-hot schema encodings (e.g., NEO, LERO) fail under schema changes, whereas schema-agnostic designs (BAO, FASTgres, LOGER) show greater resilience.

FASTgres and LOGER misfire severely under covariate shifts, while BAO is more resilient due to live statistics but still degrades.

Key Takeaways from Experiment 5

- **Query Generalization Is Architecture-Dependent:** No LQO generalizes across all dimensions, however, adaptive context-based systems tend to be most stable.
- **Join Distributions are Critical:** Declines in join overlap between train-test sets lead to severe performance degradations, more than any other query generalization aspect.
- **Schema-Dependent Encodings Limit Adaptation:** One-hot encoding methods struggled immensely with schema shifts, whereas schema-agnostic designs are more robust.
- **Learned Policies Struggle under Covariate Shift:** Hint-based optimizers are highly sensitive to data distribution changes, however this can be managed with integration of live statistics.
- **Database-Level Generalization Still Unsolved:** LQOs do not transfer reliably across databases; generalizing over workloads, schemas, and data distributions is still unmet.

6 Conclusion and Future Work

We present a unified evaluation framework for learned query optimizers spanning across five dimensions: performance, robustness, learning trajectories, decision-making, and generalization, evaluated on eight benchmarks. Our analysis shows no universal winner, as LQOs can deliver strong gains on complex workloads, but PostgreSQL remains competitive especially when plan overhead is measured, opening up avenues towards hybrid optimizers. Because predictive accuracy alone is insufficient, search-guiding/pruning systems need to associate with good embedding quality. We encourage all future systems to include training order/complexity and internal plan representations during evaluation. Generalization under schema or distribution shift remains a key open challenge.

Acknowledgments

his work has been partially supported by DataGEMS, funded by the European Union's Horizon Europe Research and Innovation programme, under grant agreement No 101188416. This work has been partially supported by project MIS 5154714 of the National Recovery and Resilience Plan Greece 2.0 funded by the European Union under the NextGenerationEU Program.

7 Artifacts

All artifacts related to this work (code, datasets, experimental scripts) are available at https://github.com/athenarc/Learned-Optimizers-Benchmarking-Suite, with detailed instructions for installation, reproduction of experiments, and data usage.

[2]The join distribution of a workload is determined by i.) tables joined, ii.) their join order, iii.) join conditions and finally iv.) the number of rows each join corresponds to.

References

[1] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3515–3527. doi:10.14778/3611540.3611544

[2] Henriette Behr, Volker Markl, and Zoi Kaoudi. 2023. Learn what really matters: A learning-to-rank approach for ml-based query optimization. In *BTW 2023*. Gesellschaft für Informatik eV, 535–554.

[3] Laurent Bindschaedler, Andreas Kipf, Tim Kraska, Ryan Marcus, and Umar Farooq Minhas. 2021. Towards a benchmark for learned systems. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 127–133.

[4] T. Tony Cai and Rong Ma. 2022. Theoretical foundations of t-SNE for visualizing high-dimensional clustered data. *J. Mach. Learn. Res.* 23, 1, Article 301 (Jan. 2022), 54 pages.

[5] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2022. FiGO: Fine-Grained Query Optimization in Video Analytics. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 559–572. doi:10.1145/3514221.3517857

[6] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans. *Proc. VLDB Endow.* 16, 7 (March 2023), 1777–1789. doi:10.14778/3587136.3587150

[7] Tianyi Chen, Jun Gao, Yaofeng Tu, and Mo Xu. 2024. GLO: Towards Generalized Learned Query Optimization. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4843–4855.

[8] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. LEON: A New Framework for ML-Aided Query Optimization. *Proc. VLDB Endow.* 16, 9 (May 2023), 2261–2273. doi:10.14778/3598581.3598597

[9] Xu Chen, Zhen Wang, Shuncheng Liu, Yaliang Li, Kai Zeng, Bolin Ding, Jingren Zhou, Han Su, and Kai Zheng. 2023. BASE: Bridging the Gap between Cost and Latency for Query Optimization. *Proc. VLDB Endow.* 16, 8 (April 2023), 1958–1966. doi:10.14778/3594512.3594525

[10] Yannis Chronis, Yawen Wang, Yu Gan, Sami Abu-El-Haija, Chelsea Lin, Carsten Binnig, and Fatma Özcan. 2024. CardBench: A Benchmark for Learned Cardinality Estimation in Relational Databases. arXiv:2408.16170 [cs.DB] https://arxiv.org/abs/2408.16170

[11] CloudQuery. [n. d.]. Explainer: 3NF vs Star Schema. https://www.cloudquery.io/blog/explainer-3nf_vs_star-schema. Accessed: 2025-08-20.

[12] Zhicheng Cui, Wenlin Chen, and Yixin Chen. 2016. Multi-scale convolutional neural networks for time series classification. *arXiv preprint arXiv:1603.06995* (2016).

[13] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Parametric Query Optimization. *Proc. ACM Manag. Data* 1, 1, Article 109 (May 2023), 25 pages. doi:10.1145/3588963

[14] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endow.* 12, 9 (May 2019), 1044–1057. doi:10.14778/3329772.3329780

[15] GeeksforGeeks. [n. d.]. Difference between Star Schema and Snowflake Schema. https://www.geeksforgeeks.org/dbms/difference-between-star-schema-and-snowflake-schema/. Accessed: 2025-08-20.

[16] Zhuohan Gu, Jiayi Yao, Kuntai Du, and Junchen Jiang. 2024. LLMSteer: Improving Long-Context LLM Inference by Steering Attention on Reused Contexts. *arXiv preprint arXiv:2411.13009* (2024).

[17] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality estimation in DBMS: a comprehensive benchmark evaluation. *Proc. VLDB Endow.* 15, 4 (dec 2021), 752–765. doi:10.14778/3503585.3503586

[18] Hazar Harmouch and Felix Naumann. 2017. Cardinality estimation: an experimental survey. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 499–512. doi:10.1145/3186728.3164145

[19] Roman Heinrich, Xiao Li, Manisha Luthra, and Zoi Kaoudi. 2025. Learned Cost Models for Query Optimization: From Batch to Streaming Systems. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 5482–5487. doi:10.14778/3750601.3750699

[20] Roman Heinrich, Manisha Luthra, Johannes Wehrstein, Harald Kornmayer, and Carsten Binnig. 2025. How Good are Learned Cost Models, Really? Insights from Query Optimization Tasks. *Proc. ACM Manag. Data* 3, 3, Article 172 (June 2025), 27 pages. doi:10.1145/3725309

[21] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. 2018. Deep q-learning from demonstrations. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[22] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *Proc. VLDB Endow.* 15, 11 (July 2022), 2361–2374. doi:10.14778/3551793.3551799

[23] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *Proc. VLDB Endow.* 13, 7 (March 2020), 992–1005. doi:10.14778/3384345.3384349

[24] Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. 2024. LLMOPT: Learning to Define and Solve General Optimization Problems from Scratch. *arXiv preprint arXiv:2410.13213* (2024).

[25] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. 2024. RobOpt: A Tool for Robust Workload Optimization Based on Uncertainty-Aware Machine Learning. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 468–471. doi:10.1145/3626246.3654755

[26] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. 2024. Roq: robust query optimization based on a risk-aware learned cost model. *arXiv preprint arXiv:2401.15210* (2024).

[27] Antonios Karvelas, Ioannis Foufoulas, Alkis Simitsis, and Yannis E. Ioannidis. 2023. Toulouse: Learning Join Order Optimization Policies for Rule-based Data Engines. In *EDBT/ICDT Workshops*. https://api.semanticscholar.org/CorpusID:258559110

[28] 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* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1214–1227. doi:10.1145/3514221.3526154

[29] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[30] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[31] Suyong Kwon, Kyuseok Shim, and Woohwan Jung. 2025. Cardinality Estimation of LIKE Predicate Queries using Deep Learning. *Proc. ACM Manag. Data* 3, 1, Article 20 (Feb. 2025), 26 pages. doi:10.1145/3709670

[32] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proc. VLDB Endow.* 17, 7 (May 2024), 1565–1577. doi:10.14778/3654621.3654625

[33] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. doi:10.14778/2850583.2850594

[34] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 197–210. doi:10.14778/3626292.3626302

[35] Zibo Liang, Xu Chen, Yuyang Xia, Runfan Ye, Haitian Chen, Jiandong Xie, and Kai Zheng. 2024. DACE: A Database-Agnostic Cost Estimator. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 4925–4937. doi:10.1109/ICDE60146.2024.00374

[36] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proc. VLDB Endow.* 14, 11 (jul 2021), 1950–1963. doi:10.14778/3476249.3476254

[37] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *SIGMOD Rec.* 51, 1 (June 2022), 6–13. doi:10.1145/3542700.3542703

[38] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. doi:10.14778/3342263.3342644

[39] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Houston, TX, USA) *(aiDM'18)*. Association for Computing Machinery, New York, NY, USA, Article 3, 4 pages. doi:10.1145/3211954.3211957

[40] Ryan Marcus and Olga Papaemmanouil. 2018. Towards a hands-free query optimizer through deep learning. *arXiv preprint arXiv:1809.10212* (2018).

[41] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1733–1746. doi:10.14778/3342263.3342646

[42] Volker Markl, Guy M Lohman, and Vijayshankar Raman. 2003. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal* 42, 1 (2003), 98–106.

[43] Matillion. [n. d.]. 3NF vs Star Schema: Choosing the Right Approach. https://www.matillion.com/blog/3nf-vs-star-schema. Accessed: 2025-08-20.

[44] M.L. Menéndez, J.A. Pardo, L. Pardo, and M.C. Pardo. 1997. The Jensen-Shannon divergence. *Journal of the Franklin Institute* 334, 2 (1997), 307–318. doi:10.1016/S0016-0032(96)00063-4

[45] Artem Mikhaylov, Nina S Mazyavkina, Mikhail Salnikov, Ilya Trofimov, Fu Qiang, and Evgeny Burnaev. 2022. Learned query optimizers: Evaluation and improvement. *IEEE Access* 10 (2022), 75205–75218.

[46] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[47] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 982–993. doi:10.14778/1687627.1687738

[48] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1049–1058.

[49] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2557–2569. doi:10.1145/3448016.3457568

[50] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-loss: learning cardinality estimates that matter. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2019–2032. doi:10.14778/3476249.3476259

[51] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2019–2032. doi:10.14778/3476249.3476259

[52] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2020. Cost-guided cardinality estimation: Focus where it matters. *Proceedings - 2020 IEEE 36th International Conference on Data Engineering Workshops, ICDEW 2020*, 154–157. doi:10.1109/ICDEW49219.2020.00034

[53] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1520–1533. doi:10.14778/3583140.3583164

[54] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *SIGMOD Rec.* 29, 4 (Dec. 2000), 64–71. doi:10.1145/369275.369291

[55] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why you should run TPC-DS: a workload analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1138–1149.

[56] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQL-Storm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11 (July 2025), 4144–4157. doi:10.14778/3749646.3749683

[57] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: accurate selectivity estimation for string predicates using deep learning. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 471–484. doi:10.14778/3436905.3436907

[58] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-Based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (nov 2019), 307–319. doi:10.14778/3368289.3368296

[59] Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1745–1757. doi:10.1145/3448016.3452790

[60] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: a design space exploration and a comparative evaluation. *Proc. VLDB Endow.* 15, 1 (Sept. 2021), 85–97. doi:10.14778/3485450.3485459

[61] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3/4 (1933), 285–294.

[62] Christos Tsapelas and Georgia Koutrika. 2023. QPSeeker: An Efficient Neural Planner combining both data and queries through Variational Inference. *Advances in Database Technology - EDBT* 27, 307–319. Issue 2. doi:10.48786/edbt.2024.27

[63] Dimitris Tsesmelis and Alkis Simitsis. 2022. Database Optimizers in the Era of Learning. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 3213–3216. doi:10.1109/ICDE53745.2022.00301

[64] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (aug 2024), 3694–3706. doi:10.14778/3681954.3682031

[65] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: a normalizing flow based cardinality estimator. *Proc. VLDB Endow.* 15, 1 (Sept. 2021), 72–84. doi:10.14778/3485450.3485458

[66] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are we ready for learned cardinality estimation? *Proc. VLDB Endow.* 14, 9 (May 2021), 1640–1654. doi:10.14778/3461535.3461552

[67] Johannes Wehrstein, Timo Eckmann, Roman Heinrich, and Carsten Binnig. 2025. JOB-Complex: A Challenging Benchmark for Traditional & Learned Query Optimization. *arXiv preprint arXiv:2507.07471* (2025).

[68] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proc. VLDB Endow.* 16, 11 (July 2023), 3310–3322. doi:10.14778/3611479.3611528

[69] Peizhi Wu and Gao Cong. 2021. A Unified Deep Model of Learning from both Data and Queries for Cardinality Estimation. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2009–2022. doi:10.1145/3448016.3452830

[70] Xianghong Xu, Zhibing Zhao, Tieying Zhang, Rong Kang, Luming Sun, and Jianjun Chen. 2023. Coool: A learning-to-rank approach for sql hint recommendations. *arXiv preprint arXiv:2304.04407* (2023).

[71] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch? *Proc. ACM Manag. Data* 1, 4, Article 255 (Dec. 2023), 27 pages. doi:10.1145/3626769

[72] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 931–944. doi:10.1145/3514221.3517885

[73] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *Proc. VLDB Endow.* 14, 1 (Sept. 2020), 61–73. doi:10.14778/3421424.3421432

[74] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.* 13, 3 (Nov. 2019), 279–292. doi:10.14778/3368289.3368294

[75] Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2024. Low Rank Approximation for Learned Query Optimization. In *Proceedings of the Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management* (Santiago, AA, Chile) *(aiDM '24)*. Association for Computing Machinery, New York, NY, USA, Article 4, 5 pages. doi:10.1145/3663742.3663974

[76] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-Based or Learning-Based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (Sept. 2022), 3924–3936. doi:10.14778/3565838.3565846

[77] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-LSTM for join order selection. *Proceedings - International Conference on Data Engineering* 2020-April, 1297–1308. doi:10.1109/ICDE48307.2020.00116

[78] Ji Zhang, ‡ Supervised, Ke Zhou, Sebastian Schelter, and Hua Zhong. 2020. AlphaJoin: Join Order SelectionàSelection'Selectionà la AlphaGo. https://github.com/HustAIsGroup/AlphaJoin

[79] Jintao Zhang, Chao Zhang, Guoliang Li, and Chengliang Chai. 2023. AutoCE: An Accurate and Efficient Model Advisor for Learned Cardinality Estimation. *Proceedings - International Conference on Data Engineering* 2023-April, 2621–2633. doi:10.1109/ICDE55515.2023.00201

[80] Yunjia Zhang, Yannis Chronis, Jignesh M. Patel, and Theodoros Rekatsinas. 2023. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. *Proc. VLDB Endow.* 16, 11 (July 2023), 2962–2975. doi:10.14778/3611479.3611501

[81] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a tree transformer model for query plan representation. *Proc. VLDB Endow.* 15, 8 (April 2022), 1658–1670. doi:10.14778/3529337.3529349

[82] Zhanhao Zhao, Haotian Gao, Naili Xing, Lingze Zeng, Meihui Zhang, Gang Chen, Manuel Rigger, and Beng Chin Ooi. 2025. NeurBench: Benchmarking Learned Database Components with Data and Workload Drift Modeling. *arXiv preprint arXiv:2503.13822* (2025).

[83] Kai Zhong, Luming Sun, Tao Ji, Cuiping Li, and Hong Chen. 2024. FOSS: A Self-Learned Doctor for Query Optimizer. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 4329–4342.

[84] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1466–1479. doi:10.14778/3583140.3583160

[85] Rong Zhu, Lianggui Weng, Bolin Ding, and Jingren Zhou. 2024. Learned Query Optimizer: What is New and What is Next. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 561–569. doi:10.1145/3626246.3654692

[86] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: fast, lightweight and accurate method for cardinality estimation. *Proc. VLDB Endow.* 14, 9 (May 2021), 1489–1502. doi:10.14778/3461535.3461539