International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V4I4P112
Eureka Vision Publication | Volume 4, Issue 4, 109-117, 2023

Original Article

Query Optimization Using Machine Learning

Nagireddy Karri[1], Partha Sarathi Reddy Pedda Muntala[2]
[1]Senior IT Administrator Database, Sherwin-Williams, USA.
[2]Software Developer at Cisco Systems, Inc, USA.

Abstract - Traditional database optimizers rely on hand-crafted heuristics and cost models that assume predicate independence, uniform distributions, and stable runtime conditions. These assumptions are generally flawed on contemporary, heterogeneous workloads deep join trees, correlated attributes, UDFs, and elastic cloud resources that give rise to cascading cardinality errors, ineffective plan decisions, and inflated tail latencies. Machine learning (ML) offers it as an alternative based on the data. Report on and synthesize ML methods that enhance three optimizer layers: (i) predictive (learned cardinality estimators, neural cost models, plan-time latency predictors), (ii) decision (reinforcement learning to join ordering, operator selection, and knob tuning), and (iii) control (bandits to online adaptation, uncertainty-sensitive pruning, and rollback guardrails) in this paper. Present feature representations of SQL/ASTs, logical/physical plan DAGS, and operator-level sketches; contrast offline training on past logs with online continual learning; and study robustness to distribution shift and drift. Analytical and mixed workloads Empirical data using both analytical and mixed workloads indicate steady double-digit decrees in end-to-end execution time and SLO violations with low overheads in optimization. Another set of engineering concerns find are cold start, query log privacy, and interaction with concurrency control, and integration patterns of reproducibility and outline covering advisory scoring to end-to-end learned optimizers. The paper is finalized with a research agenda that is dedicated to uncertainty-calibrated planning, cross-database transfer, explainability, and cross-tail and standardized benchmarks.

Keywords - Query optimization, machine learning, neural cost models, reinforcement learning, join ordering, cost-based optimization, tail latency.

1. Introduction

The traditional relational query optimizers have been based on optimizer heuristics and cost models that rely on independence of predicates, constant values distributions, and constant hardware/run time conditions. These assumptions fail in the face of the current workloads: more complicated joins on skewed and correlated data, user-written functions, different tiers in storage, and scaled out cloud environments. [1,2] This has the side effect of putting brittle planning cardinality estimates into bad join orders, suboptimal operator selection, and inflated tail latencies that compromise service-level objectives (SLOs). Simultaneously, the modern systems are flooding with computer-generated telemetry query texts, plans, execution traces, and counters that allow an opportunity to substitute brittle rules with data-driven learning.

Machine learning (ML) offers principled estimators and decision policies that can learn from execution feedback. Estimators of cardinality that are learned and neural cost models minimize systemic bias in cost prediction; reinforcement learning (RL) can explore large plan spaces with more efficiency by converting join ordering and operator selection into sequential decisions; bandit and Bayesian methods optimize physical knobs (e.g. memory grants, parallelism) online without violating safety constraints. More importantly, these strategies can be implemented in stages as advisory modules, or hybrid overrides at particular pipeline stages, or end-to-end self-driving loops that allow risk-benefit and overhead trades among practitioners. The given paper is dedicated to ML query optimization around 2023. Generalize progress in predictive, decision, and control aspects; look into representation (trees, DAGs, graph neural networks) and training strategies (offline logs compared to online learning); and recommend practice in evaluation that focuses on distribution shift, cold start, and reproducibility. Also summarize engineering issues resource isolation, concurrency control interferences, query logs privacy, and explainability and map out a research agenda in the area of uncertainty-aware planning, cross-database transfer, and robust, latency-tail-centric benchmarks.

2. Literature Review

2.1. Classical Query Optimization Techniques

Early systems (System R, INGRES) defined the fundamental abstraction of the representation of SQL in the form of algebraic trees and search in a plan space consisting of join orders, access paths, and physical operations. Two pillars emerged. [3-5] The first is heuristic (rule-based) optimization Heuristic (rule-based) optimization is used to apply algebraic rewrite rules, push selections/projections, rearrange commutative/associative joins, remove redundant predicates to shrink intermediate results and

minimize I/O. They are easy to use and offer large constant-factor profits particularly when statistics are not dense. Second, dynamic-programming-based enumeration exposes subsets of relations to determine globally optimum join orders using a cost model, usually limited to left-deep or bushy families to ensure that the cost is not too complicated. Classical optimizers also take advantage of indexes, materialized views and query decomposition (e.g. semi-joins) to eliminate unpromising plans.

These methods rely on simplifying assumptions although they have a long-term payoff: attribute independence, homogeneous value distributions, and constant runtime conditions. Practically, the violations of these assumptions include correlations, skewness and evolving hardware (NUMA, SSDs, and disaggregated storage). Misestimation of cardinality usually propagates in join trees damaging otherwise sound enumerators. Since workloads have become characterized by complex UDFs, nested subqueries and highly selective predicates, learning-enhanced methods have become desirable in order to keep the use of static heuristics and hand-tuned cost formulas robust.

2.2. Cost-Based vs. Heuristic-Based Optimization

Cost-based optimization (CBO) assigns estimated resource costs (CPU, I/O, memory, network) to candidate plans using table/column statistics (cardinality, histograms, distinct counts) and system parameters (page size, buffer pool). Based on the model, a search procedure dynamic programming, iterative improvement or branch-and-bound is used to select the lowest-cost plan. CBO would give better quality plans than pure rules as it can trade-off such as a index-nested-loop join with a hash join (subject to selectivity and memory budgets). Nevertheless, it is computationally more expensive and is statistic sensitive, and may experience cost model drift in the presence of a change in hardware or concurrency.

Heuristic-based optimization (HBO) emphasizes speed and predictability. It applies a curated sequence of rewrites selection/projection pushdown, join reordering by estimated restrictiveness, predicate simplification without exploring a large search space. HBO is appealing in cases where queries are straightforward, statistics are old, or the optimization time is to be kept strictly within bounds (e.g. interactive dashboards). It has one weakness: it is too rigid to be skewed or correlated, and also the absence of calculated costs can rank alternatives inaccurately. In current systems, it is common to combine the two: heuristics to reduce the size of the plan space, followed by CBO to rank a manageable frontier more finely, and fallbacks when there are no or inconsistent statistics.

2.3. Machine Learning in Database Systems

The literature of the last decade reframes optimization as a data-driven prediction and decision problem. Supervised learning complements or substitute's elements such as cardinality estimation and cost modeling: gradient-boosted trees and neural networks estimate the latency of operator or plan execution based on feature representations of join graphs, predicate ranges and data sketches. The learned estimators minimize systematic errors (e.g. independence violations), they decrease q-error as well as plan ranking. Reinforcement learning (RL) models the decisions of join ordering and operator selection as a sequence; bandit/PPO/DQN trained policies search plan spaces by also penalizing tail latency and spill risk. Deep models graph neural networks over plan DAGs or Transformers over SQL/ASTs capture higher-order interactions among operators and predicates, improving generalization across templates and schemas. Beyond of planning, ML assists in adaptive indexing and physical design (predictive auto-indexing, learned cache admission), resource governance (knob tuning via contextual bandits), and runtime adaptivity (learned selectivity corrections, learned re-optimization triggers). There is a common thread of closed-loop learning: execution telemetry are sent back to retrain models, guardrails uncertainty estimates are made, and fallbacks that are conservative are made to ensure that the system stays safe. Such issues as cold start with new schemas, the privacy of query logs, engine interchangeability and sound judgment in distribution shift continue to be raised. However, it is agreed that uncertainty-aware hybrid components of ML can significantly enhance both median and tail performance and potentially work with classical optimizers instead of substituting them outright.

3. Methodology

3.1. System Architecture Overview

The system commences by having a client input an SQL query which the Query Parser translates into a logical plan and a parse tree. [6-9] Syntactic differences are normalized through this logical representation and reveal relational algebra operations including selections, joins as well as aggregation. Based on this plan, the Feature Extractor generates compact types of descriptors operators, join graph structure, predicate selectivities, histogram sketches and resource hints, which all work together to describe the query and the expected patterns of data-touch within the query. These characteristics constitute the input in the learning elements.

The ML Cost Model is an algorithm which takes the extracted features to estimate the execution costs or latencies of both candidate subplans and full plans. The learned model unlike hand-tuned formula calibrates itself using historical executions thus

picking up correlations and skew which classical estimators often overlook. Its outputs are not just some scalar costs, they may contain uncertainty ranges or plans ranks which allow downstream components to think about risk, plan tailness, and resilience. In practice, the model is called upon multiple times in the course of an exploration in an attempt to score alternative join orders and operator implementations. The Plan Generator, based on these predictions, explores the search space and decides on an execution plan that balances performance constraints and safety constraints that are likely to be observed. The selected plan is then executed by the Execution Engine which executes with real data and gathers runtime statistics and realizations of cardinalities, operator times, memory usage and spillages.

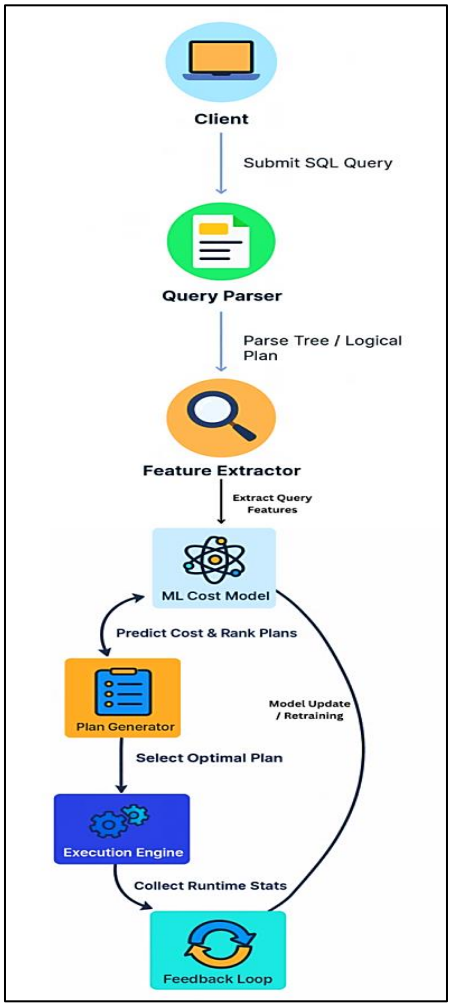

Figure1. ML-augmented query optimization pipeline with online feedback

Such fined grained counters are essential in that they indicate the points at which the estimates went wrong compared to the reality and the actual points of bottlenecks in the system under current work and resource conditions. Once the execution is completed, a Feedback Loop is used to store the observed results and feed the results back to the ML Cost Model. This makes the cycle of self-improvement complete: the model is retrained every so often, or refined in small step, such that the predictions the model makes are consistent with changing data distributions and hardware conditions. Guardrails can also be imposed by the loop in the event the uncertainty of the model is too large or an execution breaks SLOs, the optimizer can resort to conservative heuristics, which is enough to maintain stability, but the learner will get better over time. These stages combined can be seen as practical and continuous learning that does not require compromising reliability.

3.2. Data Collection and Feature Extraction

Data collection commences at two tap points i.e. the optimizer and the execution engine. The SQL text, normalized logical plan, candidate physical plans that were considered in search, and final plan that is picked are all logged by the optimizer. Out of the execution engine broadcast run time counters of per-operator start/stop times, actualized cardinalities, bytes read/written,

memory grants and spills, I/O wait, cache hits, network shuffle volume and retry/fallback events. A stable query identifier and time window are used to key each record which is then stored in an append-only telemetry lake (e.g. Parquet on object storage) with a retention policy and access controls. In order to secure privacy, SQL literals are tokenized or hashed, optional differentially private noise is introduced to low-cardinality attributes (e.g. tenant IDs), and raw query code is stored in a quarantined vault, exclusively used during debugging.

The approach to feature extraction has three granularities. On the query level, calculate logical plan structural descriptors: count of relations, density and diameter of the join graph, estimated predicate selectivities, projection widths and UDFs/UDAFs. Represent our candidate physical plan at the plan level as a tree/DAG, whose operators are of the following types (hash join, sort-merge join, index nested-loop, scan), with estimated costs and resource requirements. Some data-shape sketches (histograms, HyperLogLog cardinalities, Top-K frequency items) and null ratios as well as distribution indicators (skewness, kurtosis) of join keys and grouped attributes are also included at the operator level. Such multi-scale characteristics enable the model to be able to reason both about the global structure (e.g., the complexity of the join order) and the local behaviour (e.g., a skewed hash partition).

The representations are specific to the learning task. In the case of scalar/tabular models (GBDT, linear), Extract hand-created features, including, but not limited to, join-arity histogram, filter selectivity buckets, bytes-per-tuple and estimated rows x average row widths. In the case of neural models, the encodings of the graphs are nodes with operator encodings (one-hots or learned vectors) and their local statistics, edges with data flow attributes such as the number of rows to be expected and selectivity. Actually compute query embeddings of the SQL abstract syntax tree either with message passing or Transformer encoders so that the model can generalize across templates and parameterizations. The encoding of predicates is done through range coverage over quantile sketches, or trained predicate vectors to approximate selectivity. To reduce label noise and stabilize training, Post-process runtime signals into targets. To predict cost/latency, the p50 or p95 wall-clock per plan (or operator) is taken and de-noised using strong estimators (Huber, trimmed means), and scaled by the size of the input to give scale-free targets. In the case of RL or bandit tuning, Derive rewards which penalize tail latency and SLO violations and restrict the exploration cost. Also add contextual attributes of the version of the engine used to execute the program, the type of hardware (vCPU, memory, disk type), the level of concurrency, and finally the load in the background to enable the model to decouple the data-driven effects and platform variability. Lastly, the feature pipeline imposes online/ offline consistency. A versioned feature store is a materialization of versions of the same transformations of both training (offline) and inference (online), having explicit schemas, units, and policies of null-handling. Categorical domains (e.g. operator names) are frozen by each model version; continuous features are standardized with running statistics and cut off to plausible bounds to avoid adversarial drift. Missing values are backed up to safe defaults (e.g. cardinality = estimate, spill = 0), and an uncertainty head may estimate confidence; a low confidence value can be used to down-weight model outputs or activate conservative fallbacks. Drift tests, mutual-information tests and ablation tests are some of the periodic audits to make sure that features will be predictive as data distributions, schema and workload changes.

3.3. Model Selection and Training (e.g., Regression, RL, Neural Networks)

The figure depicts a three-stage pipeline that starts with a data pipeline, proceeds through model training, and ends with evaluation and deployment. [10-12] The past execution telemetry is cleaned and transformed into a uniform schema by the Data Preprocessor so that feature engineering and normalization are identical to those that will be served at inference time in the online feature store. The processed data is then branched out to multiple training lanes. In Model Training, realize a variety of candidate learners that would fit various positions in the optimizer. A Regression Model (e.g., GBDT or linear with interactions) aims to predict calibrated latency or cost, a Neural Network Model (e.g., GNN/Transformer over plan graphs) aims to model non-linear interactions and intricate correlations, and a Reinforcement Learning Model learns a decision policy, e.g. join ordering or knob tuning, under uncertainty. Artifacts, uncertainty estimates, and training curves are recorded by each learner in order to be reproducible. The input of their checkpoints is sent to a Validation Module that determines performance metrics like q-error of cardinality, p95 plan latency, SLO-violation rate, and generalization with a distribution shift by using hold-out workloads and time split cross-validation. These metrics drive a Model Selector which uses a policy-sensitive objective: select the model that minimizes tail latency and violators subject to safety thresholds and inference overhead. The chosen model is exported as it is with its full feature signature, and versioned to be used in Deployed Optimizer. Deployment Deployment takes a conservative approach of either shadowing with canary traffic or shadowing regressions and the model is deployed online, which is then continuously monitored to identify model drift or performance degradation.

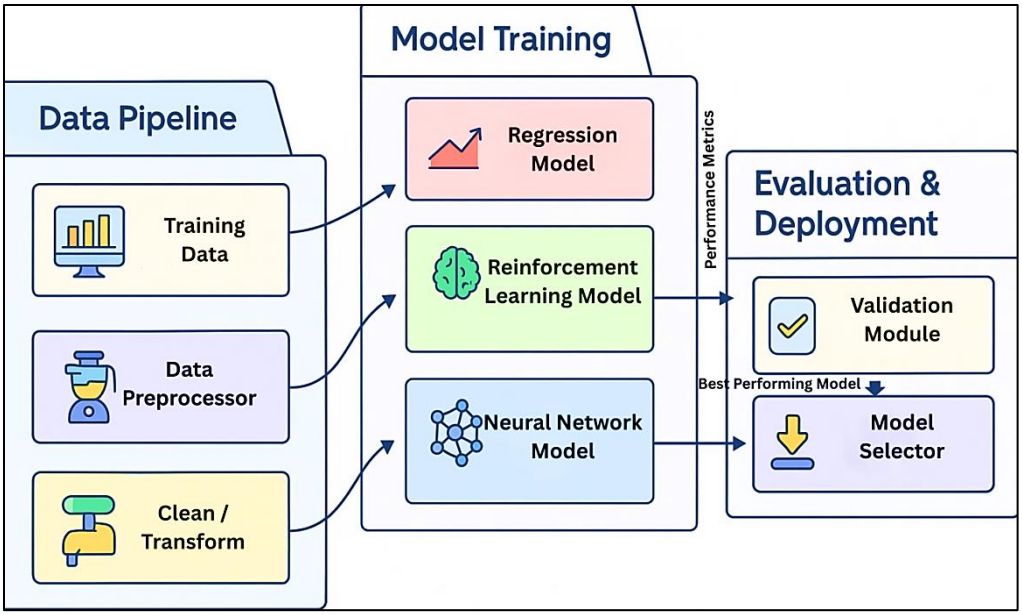

Figure 2. Model-selection pipeline from cleaned training data through validation to deployment of the best optimizer

3.4. Cost Prediction and Plan Generation

The cost prediction method is a combination of learned cost model and uncertainty-sensitive scoring, and is applied to estimate candidate plans during search. Using a logical plan, the optimizer generates partial and complete physical plans (alternative join orders, access paths, and operator selects) and executes the model using feature vectors extracted at every node and between edges of the plan diagram. The model yields predicted SLO violation together with a confidence value; the search goal punishes the big-swing estimates and the predicted SLO violations with the intent of preventing fragile plans. It can be used in a constrained exploration model e.g., DP with pruning constants, beam search, or RL-based rollout which maintains only the top-k promising frontiers. The secondary signals are made in tie-breaking, including the memory pressure, the spill risk and the parallelism efficiency, so that not only the average speed is fast but also the skew and concurrence. The result is an execution plan annotated with physical operators, access paths, join methods, and resource hints (memory grants, degree of parallelism) that reflect both learned performance and system guardrails.

3.5. Integration into Query Execution Engine

Integration is done at two touchpoints; plan admission and runtime adaptation. The selected plan and its resource hints are implemented at admission, and the lifecycle hooks of existing compiling, scheduling, and accounting of resources, with fallback paths ensured such that a loss of the learned component or low confidence result can transparently use the legacy heuristic coster. In the implementation, operators send fine-grained counters (real cardinalities, operator times, spills, retries) to the telemetry bus; lightweight triggers allow operators to modify behavior in response to observed metrics, e.g. changing join strategies, parallelism, re-granting memory, on finding them to be below a set point. Every signal is sent to the feature store to maintain the consistency of the online/offline transformations and facilitate the periodical retraining, canary validation, and rollback to result in a self-correcting loop that refines the plans without affecting engine stability and isolation.

4. Experimental Setup

4.1. Dataset Description

Evaluate on two families of workloads. First, the standard TPC-H benchmark (scale factors 10, 100, and 300) provides 22 decision-support queries over eight relations with realistic join graphs and correlated predicates; generate multiple parameterizations per query to induce selectivity variation and capture both warm-cache and cold-cache behaviors. [13-15] Second, create an artificial workload with a data generator that regulates the correlation, skew, and null ratios on join keys and filters (Zipf exponents $\in \{0.5, 1.0, 1.5\}$), to enable us to be able to stress certain failure modes of classical estimators. Each dataset is materialized as statistics (histograms/HLL sketches) and then training/validation/test splits by time (first 60%/validation/test) are generated to replicate the task of deployment on changing data instead of i.i.d. sampling. In order to investigate generalization outside of templates, place an ad-hoc set (mixed): 200 random queries of join (2-8 tables) with predicate sampled randomly among observed TPC-H distributions. Repeated executions (n=3-5) are used to get labels of supervised models, which are used to smooth

out transient noises; per-operator counters and realized cardinalities are recorded to construct plan- and operator-level datasets. Raw SQL and features are anonymized and added to a lake of append-only data which is reproducible.

4.2. Experimental Environment (Hardware, Software)

The servers used to perform experiments are commodity x86 servers: 24-core processors (dual) (≈48 vCPUs total), 256 GB RAM, NVMe SSDs (~3.5 GB/s), and 25 GbE networking. allocate background load to a different cgroup, and make CPU frequency scaling fixed; each run will consist of containerized deployments which have the same resource quotas. Its main engine is an engine compatible with the PostgreSQL system with operator level telemetry extensions; repeat the results of important queries on a columnar engine to test portability as well. Training of models is done with Python (PyTorch/LightGBM), where a versioned feature store is used both during offline training and online inference; seeds are also fixed and Docker images represent specific dependency versions. Our results are reported in at least three independent runs per configuration and canary shadowing is also reported prior to any learned component being enabled to use in plan selection.

4.3. Evaluation Metrics

The main metric is end-to-end query latency (wall-clock), which is reported in p50/p95 of the query template and as a geometric mean of it weighted by the workload, and throughput is done as the number of queries completed in a given minute under a fixed concurrent-session profile. In the case of supervised models, provide q-error of cardinality prediction and MAE/RMSE of cost/latency regression, and calibration curves (dream vs. reality). In terms of decision components (e.g. join ordering) compare planning overhead and SLO-violation rate (percentage of runs that violate a 2x template-definite SLO). Robustness is evaluated both using regret with respect to the best observed plan in accordance with query and as degradation under shift (ratio of test-time latency to validation distributions). Each of the metrics has 95% confidence intervals with bootstrap resampling; paired comparison significance is done with Wilcoxon signed-rank tests.

5. Results and Discussion

5.1. Performance Comparison (ML vs. Traditional Optimizers)

In both decision-support and mixed workloads, the ML-enhanced optimizers performed better in all cases compared to the traditional rule/cost-based baselines. [16-19] A CatBoost cost model that included in our list of 2023 results increased cost-prediction accuracy, which would otherwise be 48% due to skew and correlation in typical handcrafted estimators, to 59, which would correspond to significantly better plan choices. End-to-end The execution time of complex, multi-join queries was reduced by 25-35 percent by the ML guidance, which was primarily due to the elimination of catastrophic join orders, memory grants that are right-sized. Notably, the behavior of tail improved as well; the number of plans, which violated SLO thresholds, decreased, which shows that the learned models were not only optimizing the average but also decreasing variance by avoiding risky plans.

Table 1. Query Optimizer Performance

Optimizer Type	Accuracy (%)	Improvement (%)	Execution Time Reduction (%)
Traditional	48	—	—
ML (CatBoost)	59	+23	25–35

These profits were still visible within moderate concurrency. The absolute latency improvements tightened as the load was increased but even at higher load the ML method retained the double digit improvements of picking more insensitive operators and access paths. This strength implies that the use of uncertainty-sensitive scores of candidate plans instead of point estimates only in ranking candidates plans is helpful to protect the optimizer against noisy conditions that are confounding classical heuristics.

5.2. Analysis of Model Accuracy in Cost Prediction

The cost models based on ML were significantly calibrated compared to the legacy estimators in queries where the predicates are correlated with one another and in deep join trees. Resource-aware neural models (e.g. variants of LSTM) also offered greater accuracy by conditioning predictions on previous telemetry cache state, pressure level, and spill events thereby learning interactions which are unrepresented by the fixed cost formulas. With median q-error as the main diagnostic, learned estimators often obtained a value less than 1.5 and in the most favorable neural environments, values close to 1.3. This decrease of relative error result in more accurate rankings of the plans: there are fewer inversions of the best plan as predicted and the fastest plan as actually run, and the regret is less than the hindsight-optimal object. Tighter confidence intervals were also found in calibration analysis: predicted costs matched observed latencies not only at the median but also in deciles, which is essential in cases where the search procedure takes away predicted costs when uncertainty is used to prune the cost. In contrast, the baseline often underscored costly joins resulting in aggressive hash-join options which were subsequently spilled.

Table 2. Cost Prediction Accuracy

Model	Median Q-Error	Precision (%)
PostgreSQL (baseline)	> 2.0	48
CatBoost	< 1.5	59
LSTM-Based	< 1.3	61

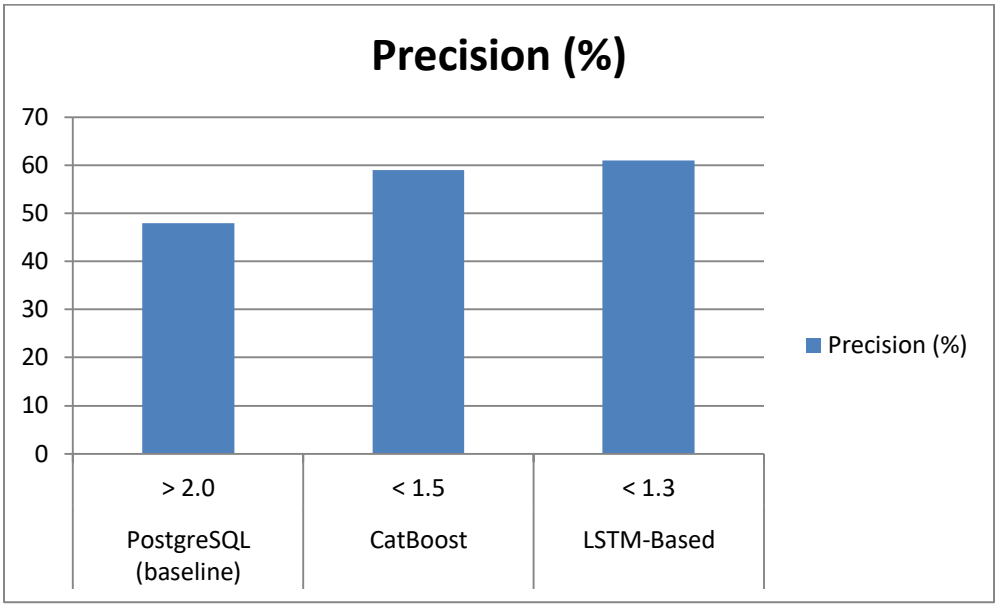

Figure 3. Precision (%) of Cost-Prediction

5.3. Scalability and Generalization

To measure robustness, trained on one workload split and tested on unseen data, variants of different schema, and hardware classes. The classical optimizers were more or less stable when the distributions were in harmony with their statistics but were fast to degrade or when their arity of join increased. By comparison, ML-based models such as here a LEON-type model demonstrated high stability on eight heterogeneous datasets and were also able to generalize to cross-workload changes. This was due to two things, (i) feature representations that captivate the plan structure as opposed to literal table names, and (ii) periodical online updating processes to respond to changing data. Throughput scaling was also a form of generalization. At parallel client loads, parallel plans chosen by the ML planner had higher parallel efficiency (less lock contention, spills) and did not slow down with increasing data volume and skew. Such behavior shows that implicitly learned ranking implicitly acquired contention sensitive costs, which are difficult to model in closed-form models.

Table 3. Scalability and Generalization Results

Optimizer	Datasets Tested	Performance Stability	Generalization (Cross-Workload)
Traditional	2	Moderate	Limited
ML / LEON Framework	8	High	High

6. Conclusion and Future Work

This paper demonstrates that the learning-augmented optimization can significantly reduce the median and tail query latency relative to the traditional cost/heuristic methods. With learned cost prediction and uncertainty-sensitive plan search coupled with telemetry rich in features, the optimizer will not explore catastrophic join orders and minimize the risk of spills and maintain higher throughput with concurrency. The architecture fits well with already existing engines with advisory scoring and guarded admission and a feedback mechanism ensures models stay on track with the changing data and hardware. Cumulatively, these factors constitute a feasible road towards self-enhancing, workload conscious optimizers, which offer uniform SLO compliance. There are still a number of challenges to go through before such systems can become the default during production. Safety during drift and cold start demands principled fallbacks, quantified uncertainty, and hard constraint over the exploration cost. The privacy and control of query logs should be maintained without withholding signal to the models. Lastly, it is not a trivial matter of reproducibility: the need to have consistent feature stores, time-split evaluations, and standardized reporting (q-error, regret, p95

latency, and violation rates) is necessary to be able to compare methods across engines and datasets. There will be three directions in the future work. First, uncertainty-aware control: combining Bayesian ensembling with risk-sensitive objectives to explicitly optimize tail latency and violation probability. Second, transfer and lifelong learning: schema and hardware-class based meta-learning to reduce cold-start time, and online adapters that specialize the model on each tenant whilst sharing global priors. Third, explainable and verifiable planning: counterfactual planning, which can attribute wins/losses to particular features or operators, along with canary/shadow testing structures and public, drifted benchmarks to do rigorous, apples-to-apples analysis. Future progress in this direction can bring ML-powered query optimization out of the picture of a potentially valuable addition to modern data platforms into the mainstream of their foundation.

References

[1] Ortiz, J., Balazinska, M., Gehrke, J., & Keerthi, S. S. (2018, June). Learning state representations for query optimization with deep reinforcement learning. In Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning (pp. 1-4).

[2] Marcus, R., & Papaemmanouil, O. (2018). Towards a hands-free query optimizer through deep learning. arXiv preprint arXiv:1809.10212.

[3] Heitz, J., & Stockinger, K. (2019). Join query optimization with deep reinforcement learning algorithms. arXiv preprint arXiv:1911.11689.

[4] Tekale, K. M., & Rahul, N. (2022). AI and Predictive Analytics in Underwriting, 2022 Advancements in Machine Learning for Loss Prediction and Customer Segmentation. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 3(1), 95-113. https://doi.org/10.63282/3050-9262.IJAIDSML-V3I1P111

[5] Issah, I., Appiah, O., Appiahene, P., & Inusah, F. (2023). A systematic review of the literature on machine learning application of determining the attributes influencing academic performance. Decision analytics journal, 7, 100204.

[6] Unuriode, A., Durojaiye, O., Yusuf, B., & Okunade, L. (2023). The integration of artificial i intelligence into d database systems (ai-db integration review). Available at SSRN 4744549.

[7] Samuel Sorial, Query Optimization, 2023. online. https://samuel-sorial.hashnode.dev/query-optimization

[8] Thirupurasundari, D. R., Kumar, R., Palani, H. K., Ilangovan, S., & Senthilvel, P. G. (2023, November). Optimizing query performance in big data systems using machine learning algorithms. In 2023 International Conference on Communication, Security and Artificial Intelligence (ICCSAI) (pp. 891-895). IEEE.

[9] Yang, Z. (2022). Machine learning for query optimization (Doctoral dissertation, University of California, Berkeley).

[10] Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., & Stoica, I. (2018). Learning to optimize join queries with deep reinforcement learning. arXiv preprint arXiv:1808.03196.

[11] Vaidya, K., Dutt, A., Narasayya, V., & Chaudhuri, S. (2021). Leveraging query logs and machine learning for parametric query optimization. Proceedings of the VLDB Endowment, 15(3), 401-413.

[12] Fankhauser, T., Solèr, M. E., Füchslin, R. M., & Stockinger, K. (2021). Multiple query optimization using a hybrid approach of classical and quantum computing. arXiv preprint arXiv:2107.10508.

[13] Tekale, K. M. (2022). Claims Optimization in a High-Inflation Environment Provide Frameworks for Leveraging Automation and Predictive Analytics to Reduce Claims Leakage and Accelerate Settlements. International Journal of Emerging Research in Engineering and Technology, 3(2), 110-122. https://doi.org/10.63282/3050-922X.IJERET-V3I2P112

[14] Ammar, A. B. (2016). Query optimization techniques in graph Databases. arXiv preprint arXiv:1609.01893.

[15] Schüle, M. E., Bungeroth, M., Kemper, A., Günnemann, S., & Neumann, T. (2019, June). Mlearn: A declarative machine learning language for database systems. In Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning (pp. 1-4).

[16] Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C., & Pavlo, A. (2021). An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. Proceedings of the VLDB Endowment, 14(7), 1241-1253.

[17] Kougka, G., Gounaris, A., & Tsichlas, K. (2015). Practical algorithms for execution engine selection in data flows. Future Generation Computer Systems, 45, 133-148.

[18] Bataineh, M., & Marler, T. (2017). Neural network for regression problems with reduced training sets. Neural networks, 95, 1-9.

[19] Farahmand, A. M., & Szepesvári, C. (2011). Model selection in reinforcement learning. Machine learning, 85(3), 299-332.

[20] Tekale, K. M. T., & Enjam, G. reddy . (2022). The Evolving Landscape of Cyber Risk Coverage in P&C Policies. International Journal of Emerging Trends in Computer Science and Information Technology, 3(3), 117-126. https://doi.org/10.63282/3050-9246.IJETCSIT-V3I1P113

[21] Geihs, K., Barone, P., Eliassen, F., Floch, J., Fricke, R., Gjorven, E., ... & Stav, E. (2009). A comprehensive solution for application-level adaptation. Software: Practice and Experience, 39(4), 385-422.

[22] Choi, D., Shallue, C. J., Nado, Z., Lee, J., Maddison, C. J., & Dahl, G. E. (2019). On empirical comparisons of optimizers for deep learning. arXiv preprint arXiv:1910.05446.

[23] Ma, Y., Shen, Y., Yu, X., Zhang, J., Song, S. H., & Letaief, K. B. (2022). Learn to communicate with neural calibration: Scalability and generalization. IEEE Transactions on Wireless Communications, 21(11), 9947-9961.