International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 2, Issue 1, 63-71, 2021
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V2I1P108

Original Article

AI-Powered Query Optimization

Nagireddy Karri
Senior IT Administrator Database, Sherwin-Williams, USA.

Abstract - *The framework for AI-powered query optimization that augments, rather than replaces, a classical cost-based optimizer. The design incorporates three components of learned: (i) a learned cardinality estimator that learns the correlation between joins and predicates; (ii) a neural residual cost corrector that learns cost error at the operator level; and (iii) a reinforcement-learning (RL) planner that focuses on high-leverage transformations of the plan under the constraints of latency and resource cost. The system works in two steps, first, offline training of the system based on past workloads and schema-sensitive synthetic queries, and lastly online adaptation that is done cautiously by using execution feedback (observed row counts, operator run times, spill events). Uncertainty gating is used to enforce safety, time-out-sandboxed trials, and immediate fallback to baseline heuristics. We detail an integration path that keeps optimizer modularity intact (Volcano/Cascades memoization, rule rewrites) while exposing pluggable inference hooks. Compared to a robust cost-based baseline, TPC-H/DS and JOB evaluation indicate a consistent decrease in p95/p99 latency, plan stability and decreased re-optimization, as well as a decrease in the CPU and memory consumption during peak loads. Failure modes out-of-distribution predicates, opaque UDFs and drift and demonstrate how risk can be mitigated using drift detection and canaried fine-tuning. The findings show that AI help produces empirical, trustworthy returns in combination with strong guardrails and observability.*

Keywords - *Query optimization, cost-based optimizer, execution feedback, memorization, tail latency, plan stability.*

1. Introduction

Relational query optimizers have traditionally been based on hand-written heuristics and cost models to explore a combinatorial search space of join orders, access paths, and operator implementations. Flaws in decades of engineering work have brought about robust systems, however, fail modes persist that distort data, correlated predicates, stale statistics, and varying workloads continue to give rise to unstable plans and unpredictable tail latencies. These limitations are operational risks in cloud and multitenant environments where small mistakes in planning can cause large costs or SLO noncompliance or nonconformities, due to resource elasticity and pay-as-you-go charges. [1-3] Moreover, modern analytical patterns (semi-structured data, complex UDFs, federated sources) strain classical assumptions about cardinality independence and uniform cost scaling, exposing systematic biases in legacy estimators.

The paper presents an example of an AI-based method to query optimization that enhances, but does not replace the conventional cost-based architecture. Our system includes three components that are learned; a cardinality estimator that combines query structure with sparse runtime information; a neural cost correction layer that corrects the miscalibrated operator models and a reinforcement-learning policy that suggests transformations of plans of high value subject to latency and resource constraints. They are pretrained on the historical workloads, and they are constantly optimized online using the feedback on actual execution (the number of rows matching the workload, the time taken by the operators) allowing quick adaptation to the drift without destabilizing the engine. The design is focused on safety, observability and operability. Uncertainty module: in cases where the model confidence is low, uses classical heuristics; executes proposals guarded, and uses timeouts, to contain potential risks; picks the plan based on stability and average performance. Our evaluation focuses on the robustness of decision quality plans, rate of reoptimization and p99 latency but not throughput. The suggested framework aims to gain a practical benefit in the context of real deployments in which correctness, predictability, and cost control have even greater significance than raw speed through maintaining optimizer modularity and the addition of data-driven feedback loops.

2. Related Work

2.1. Traditional Query Optimization Techniques

Classical query optimization has its origins in the cost-based model made by System R, where the optimizer executes logical plans (e.g. join orders, access paths, operator implementations) and chooses the least expensive plan using a manually-written cost model. [4-6] This approach is supported by two pillars (i) dynamic-programming search of left-deep or bushy join trees that are usually informed by pruning heuristics and interesting orders (ii) statistics-based cardinality estimation based upon histograms,

samples and independence assumptions. Designing extensible structures over the years, e.g. Volcano/Cascades, rule-based transformations and memoization were refined in order to make exploration manageable, with commercial engines (e.g., SQL Server, Oracle, DB2) making big investments in multi-column stats, partition-aware costing and parallel operators. Standardized comparative evaluations using benchmark suites such as TPC-H and TPC-DS were based on correlated predicates, skew and complex GROUP BY/ORDER BY patterns.

These techniques are still struggling to deal with headwinds regardless of their engineering maturity. Independence and uniformity assumptions fail due to correlations and skew; statistics grow obsolete with rapidly changing, cloud-native data; and search space grows with semi-structured data, UDFs, and federated sources. Mid-query re-routing, as well as traditional adaptive query processing re-optimization, are the techniques that reduce the drift, but with costs such as plan instability, and tail-latency spikes. Purely analytical models are hard to keep calibrated as the size of data, the degree of heterogeneity, and the elasticity rise, and it requires human tuning, which is expensive.

2.2. Machine Learning in Database Systems

ML has been embedded in several data system stack layers. Selectivity prediction Learned cardinality estimators (e.g., deep set/graph models, autoregressive density estimators) are trained to learn the correlations in the data instead of making independence assumptions. Legacy operator models have systematic biases which are corrected by neural or gradient-boosted cost correctors. In addition to the optimizer, ML applications power workload prediction, knob tuning, and storage layout choice such as systems and tools recommending indexes/materialized view, making hot partitions, and auto tuning buffer sizes and parallelism. Autonomous 2018-2021 Database initiatives would integrate supervised learning, anomaly detection, and control loops to autonomously perform routine maintenance (stats refresh, index management), detect regressions and SLO enforcement with minimum human involvement.

ML is attractive due to its capacity to make use of fat, that is, the large telemetry execution times, row counts feedback, and hardware counters to constantly recalibrate decisions. Yet, data greed, concept drift, and explainability are still viable issues. Models need to cross-cut through schemas, workloads and gracefully degrade in the face of uncertainty and inter-operate with the existing observability/rollback mechanisms. As a result, numerous efficient designs scale in addition to replacing the classical optimizers, by the selection of learned components under confidence thresholds.

2.3. Query Planning Learning Approaches based on reinforcement.

RL reframes planning as sequential decision-making: an agent incrementally constructs a plan (e.g., choosing the next join) and receives rewards based on predicted or realized cost/latency. An initial application of Deep Q-Learning to join ordering was done early on; later researchers investigated policy-gradient and Actor-Critic using query graphs, operator features and partial plans as state encodings. Hybrid methods employ model-based rollouts or learned value functions to narrow search, whereas offline RL employs historical logs to lessen unsafe exploration. Researchers came up with surrogate rewards (e.g., neural cost models) to overcome reward sparsity and variance and constraint management (latency budgets, memory caps) and compared surrogate rewards on benchmarks such as IMDB, JOB, TPC-H/DS. Mean and p99 latency improvements by a factor of two are reported by a factor of two on the join-heavy queries, and the number of re-optimizations is reduced.

However, RL brings with it new challenges: sample efficiency, non-stationarity due to co-evolving data/engines and safety in distribution shift. Practical systems construct such systems then combine RL with guardrails confidence gates, timeouts, and fallbacks with cost-based search and often have a two-stage regimen: offline pretraining on historical workloads and often cautious online refinement. This research stream guides our design decisions: we apply RL to apply high leverage transformations, make abductions when we are in high uncertainty and provide execution feedback to update without plans failing.

3. System Design and Methodology

3.1. Overview of the AI-Powered Optimization Framework

3.1.1. Architecture at a Glance

The framework builds upon a traditional cost-based [7-10] optimizer (CBO) that has three learned modules (i) A Learned Cardinality Estimator (LCE), (ii) a Neural Cost Corrector (NCC) and (iii) a Reinforcement-Learning Planner (RLP) coordinated by a feedback collector and a safety/uncertainty layer. The CBO remains the source of truth for rule rewrites, transformation enumeration, and final plan validation; the learned modules are consulted via pluggable interfaces to improve estimates and guide search without breaking optimizer modularity.

3.1.2. Control Plane vs. Data Plane

The issues into a control plane (planning, learning, governance) and a data plane (execution). LCE/NCC inference services, the RLP policy server and the uncertainty gate are found in the control plane. The data plane implements candidate plans, sends lightweight telemetry (the number of rows actually processed, the time of operators, and the number of hits on buffers), and can guarded trials with hard timeouts. This division allows independent scaling and A/B isolation of elements of learning.

3.1.3. Learned Components

The LCE represents query graphs (predicates, joins, histograms, sketches) and predicts selectivities across multiple tables, explicitly modelling the correlations which their independence assumptions are unable to capture. Its results influence the adjustment of operator/plan costs by the NCC correcting systematic biases (e.g. mis-modeled I/O in case of selective nested loops, mis-modeled CPU in case of wide projections). The RLP takes an abstract search state (remaining joins, partial plan properties, budgets of resource usage) and suggests high-leverage actions join ordering steps, access path choices, or enabling/disabling runtime operators (e.g. bloom filters) under restrictions.

3.1.4. Uncertainty and Safety Guardrails

Each learned prediction is paired with an uncertainty score that is calibrated (e.g., ensembles/MC-dropout (LCE/NCC); entropy/variance proxies (RLP)). When the uncertainty is too large, the system resorts to the use of baseline heuristics or exploration is limited. A sandbox execution implementation can test risky propositions on small batches, or with early-abort sentinels, and has a cadre of worst-case overhead and hedges tail latency.

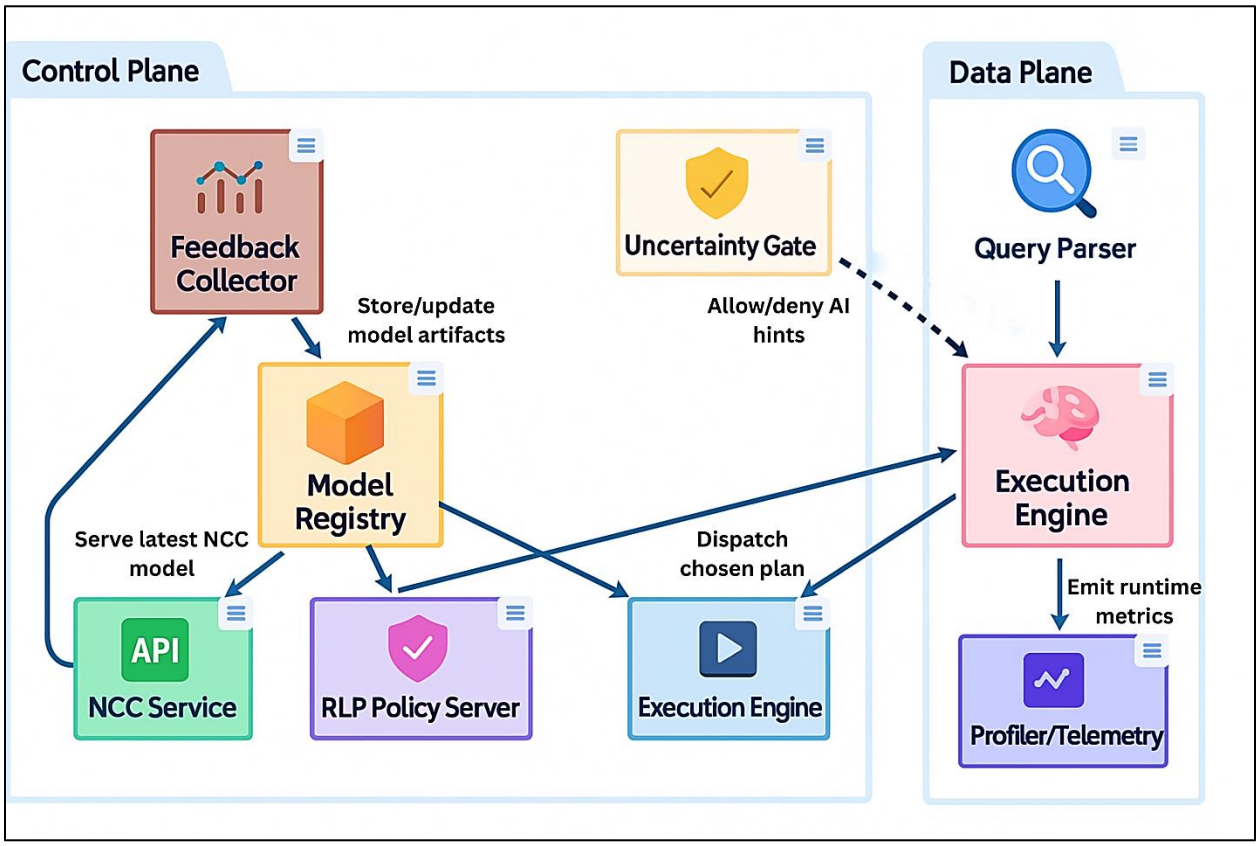

Fig 1: Control-Plane vs Data-Plane Architecture of the AI-Powered Query Optimizer

3.1.5. Closed-Loop Feedback and Continual Learning

A feedback collector summarizes data on execution artifacts into a feature store, which comprises of cardinalities, per-operator runtimes, memory/IO counters. The models are trained offline using historical workloads and updated online using small steps that are safe (e.g., elastic weight consolidation, replay buffers). Drift detectors compare the prediction and actual measures in order to perform selective retraining or threshold in order to maintain stability in the plan.

3.1.6. Integration with the Existing Optimizer

The CBO pushes change regulations (Volcano/Cascades style memo). When decision points need to be made on important areas, it calls LCE/NCC to update estimates and invokes the RLP to rank memo groups. A composite objective (predicted cost, p99 risk penalties, and stability scores) is adopted to select the final plan, and small mean-latency improvement does not cause volatility.

3.1.7. Observability and Operations

The decisions are all recorded with inputs, predictions, uncertainties and outputs to use in the post-mortem and reproducibility. There is model health, confidence drift, and re-optimization rates dashboards provided to operators, kill-switches to reverse to the classical optimization immediately.

3.2. Data Flow and Architecture

The entire process of a query being submitted to a feedback-based process of learning. A client enters the SQL query using API/Gateway which sends the query to the Query Processing block. Within, a Query Parser is used to create a logical representation and the Feature Extractor is used to add statistics to the representation in the Metadata Catalog in the form of schemas, histograms, and sketches. These characteristics are the input of the AI Optimizer, which represents learned elements (e.g., RL policy and neural estimators).

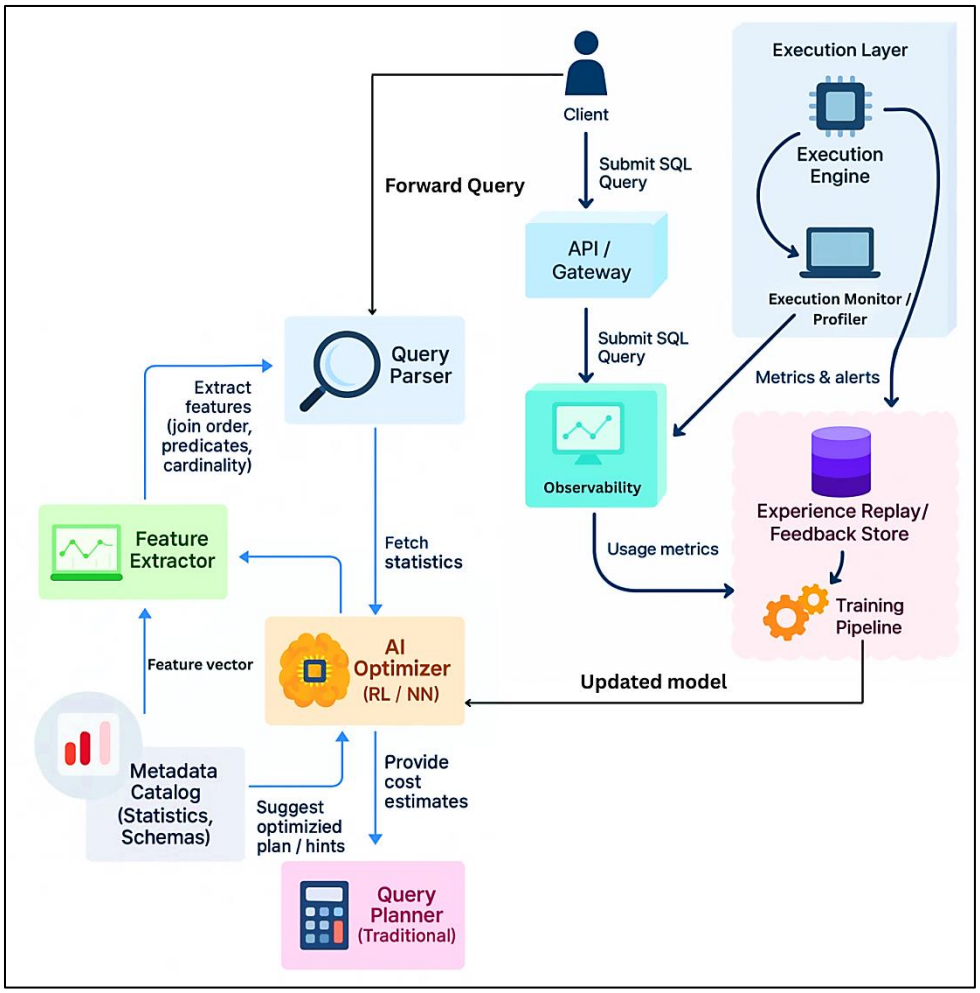

Fig 2: End-to-End Data Flow of the AI-Powered Query Optimization Framework

The AI module provides plan hints or candidate actions and with a Cost Model, provides back the Traditional Query Planner with the calibrated cost and plan scores. The planner is the ultimate decision-maker and decides the plan to adopt based on the anticipated latency in balance with both the stability and resource constraints. The chosen plan is implemented in the Execution Layer, with the Operators being executed by the Execution Engine and the Operator Latencies, Row Count, Memory/IO Counts

and Operator Error being recorded by the Operator Execution Monitor/Profiler. These cues can be used in two ways. To begin with, they feed Observability and refresh the dashboards and alerts to allow operators to monitor the p50/p99 latency, reoptimization rates and anomaly spikes in real time. Secondly, they fill the Training & Feedback loop: execution traces are sent to an Experience Replay/Feedback Store, which the Training Pipeline refines the learned models with. Retraining outputs on a periodical or triggered basis resulted in updated model artifacts which are redeployed back into the AI Optimizer, and the loop is repeated.

The design is based on safety and modularity. Classical planner and cost model are guardrails and the learned predictions are advisory and uncertain aware. [11-13] In case the AI element is unsure or suggests an unsafe change, the system will have a fallback to the baseline plan without causing disruption. The setup gives incremental adoption engines the ability to A/B test the layer of AI, and sees improvements in the quality of plans and the tail latency, and roll back in real-time in case drift or regressions are identified. Lastly, the dataflow shows that there is a neat separation between the control and data planes. The control plane exists (planning, learning, and governance) and is on in the feature extraction, AI inference, cost computation, and observability plane, and the physical execution can be found on the data plane. This isolation makes it straightforward to scale learned services can be scaled out or experimented with, and optimizes learned services do not impact execution determinism or operational SLOs.

3.3. Query Representation and Feature Extraction

Query as a heterogeneous graph which does not lose logical structure or statistical context. Base relations are represented by nodes, selection and projection predicates by nodes, and join, aggregation, and join predicates are represented by edges. To keep the planning process compact we maintain two synchronized views (i) a logical query graph (LQG) of equivalence preserving rewrites and (ii) a partial-plan state (PPS) which keeps track of the selected joins, left over relations, interesting orders and budgets of the resources. From these views, the feature pipeline emits permutation-invariant summaries (e.g., deep-set aggregates) as well as positional encodings over join trees to capture order-sensitive effects such as pipelining or spill risk.

The extraction of the features is a combination of the static metadata and lightweight runtime signals. Table cardinalities, column histograms, NDV (number of distinct values), correlation sketches (e.g. copula or mutual information estimates), index availability, partitioning and data type widths are all static features. The features include runtime reporting of recent operator latencies, cache hit rates, selectivity feedback and contention. To reduce brittleness, we prefer learned embeddings for column and predicate tokens (including LIKE patterns and UDF hints) and normalize all numeric inputs with robust scalers. Each feature bundle is accompanied with a calibrated uncertainty head, through which the optimizer can assign low-confidence signals a low weight.

3.4. Learning Model (e.g., RL, Neural Networks, Decision Trees)

A hybrid learning stack achieves this balance by using a learning stack that is efficient in samples and expressive, as well as interpretable. A graph neural network (GNN) makes predictions of the multi-table selectivities based on LQG that explicitly captures correlations that bypass independence assumptions. Its outputs are input to a neural cost corrector (a shallow MLP or gradient-boosted trees) which modulates operator level costs depending on hardware counters and previous telemetry. To build a plan, a reinforcement learning agent (policy gradient or PPO) is run over the PPS, which is choosing the next access path or join. The encoder of the state of the agent is a concatenation of GNN embeddings, cost residuals, and resource budgets; the reward is an amalgamation of the predicted latency and risk penalties of spill probability and variance.

Where explainability or small-data regimes prevail, we replace the neural components with decision-tree/GBDT models on particular subproblems (e.g. index recommendation, choice of join method). The models are fast to infer, have attributions on features and deterministic behavior that can be audited by operators. Confidence estimates (ensembles or MC-dropout in the case of neural models; variance/leaf statistics in the case of trees) used by the uncertainty gate are exposed by all learning modules. The system deals with classical heuristics when the confidence is low or the constraints are violated or explores the safe transformations only.

3.5. Cost Model and Feedback Mechanism

The model of cost is composite: a classical analytical estimator is the main one, and then a learned residual correcter is then used to offset systematic bias (ex: CPU vs. wide projections, I/O vs. random access under SSD vs. HDD, parallelism overheads). C (plan) = C_analytical(plan | stats) + Δ_learned(plan, features), with Δ_learned being trained on the differences between the predicted and observed operator times. This design does not lose decades of domain knowledge, and the data-driven term can be used to monitor hardware, engine, and workload development. Tail-latency protection To encourage stable choices, even similar mean cost, when there is a possibility of spillage, selection is based on the objective of penalizing high-variance plans and possible spillage; small risk premiums encourage the optimizer to prefer stable options.

Feedback closes the loop. The profiler records per-operator rows counts, run times, memory/ IO counter and aborts during execution. A feature store matches these traces with the features provided at the decision time, and results in labeled tuples on cardinality learning and cost learning. Drift detectors compare the predictive residuals through the sliding windows, in case of increase in error or uncertainty, thresholds are tightened and jobs are queued to be retrained. Safe feedback application applies replay buffers (when using RL), importance weighting on off-policy samples (when using RL) and canary deployment of deployed models to ensure that the process of learning also improves the calibration with no triggering plan thrash.

3.6. Training and Evaluation Process

The process of training occurs in two steps. Pretrain the GNN cardinality model and the cost residual model using the historical workloads along with some queries that are synthetic and randomly sampling schema aware generators (to better represent the rarely seen joins/predicates). The RL policy is trained through imitation learning on quality historical plans or value shaping the neural cost as a surrogate reward which minimizes the exploration cost. Nested cross-validation is used to select hyperparameters based on query templates, and early stopping is used to prevent over-fitting to particular schema or hardware profiles. Model artifacts are inferred to be versioned, signed and exported via a stable version of inference API which is the input of the optimizer.

Embrace insecure lifelong learning online. New telemetry is the input of the feedback store; taking small learning rates and replay on a diverse buffer fine-tunes the models, storing the past knowledge. canary rollouts (e.g. 5-10 percent queries/specific templates) confirm the improvements based on guard metrics plan consistency, re-optimization frequency, p95/p99 latency, and error budget on cardinality/cost. Gauging The evaluation focuses on the quality of decisions, and not only on throughput: Report (i) the fraction of queries that have reduced latency, (ii) the change in tail latency distribution, (iii) skewed/correlated predicate robustness, (iv) safety metrics (i.e., the rate of AI deferrals and sandbox aborts). Guardrails are only passed when increase coverage, but at a stable gain without a reduction in SLOs.

4. Implementation Details

4.1. Experimental Setup

The experiments were performed on a two-socket server (2x E Intel Xeon Silver 4216, 64 vCPUs in total), 256GB RAM, NVMe SSD (3.2 GB/s), and on Ubuntu 20.04 (Linux 5.8). [14-17] The PostgreSQL 13.4 DBMS used was tested with PostgreSQL plan enumeration and execution telemetry (plan enumeration hooks, plan execution telemetry hooks and a sidecar profiler). The output of the executions was the number of processes completed within a time of 30 seconds with LLVM JIT active and the following parameters: workmem=256MB, sharedbuffers=32GB, and parallel workers to 8 (unless benchmark indicated otherwise).

The AI stack ran out-of-process as microservices: PyTorch 1.8 for neural models (GNN cardinality, MLP cost corrector), XGBoost 1.3 for tree baselines, and a PPO agent implemented in Stable-Baselines3. A lightweight gRPC layer was used to serve models, in order to make the optimizer code as minimally invasive as possible. Experiments were done with fixed seeds and a cold/warm cache protocol to eliminate the effects of buffers and file-system noise (first run discarded, three run average).

4.2. Datasets and Benchmarks

A combination of regular and stress-test workload was used. (i) TPC-H at scale factors SF1, SF10, and SF100 to exercise complex joins/aggregations; (ii) TPC-DS at SF100 for realistic decision-support query diversity; and (iii) IMDB/JOB (Join Order Benchmark) to probe correlated predicates that break independence assumptions. In order to analyze semi-structured behavior, we provided an artificial JSON sales schema by adding nested attributes and path predicates, which are selective.

To test robustness to predicate variation (that is, each parameterized query instance of a template has at least one correct answer) we created instances of queries parameterized by components (10-50 parameterized queries per template). To bootstrap the LCE, we had base statistics (histograms, MCV lists) that our engine had gathered, and correlation sketches (e.g. customer.region ↔ orders.ship_priority) that we constructed to get. Primary/foreign keys and indexes were loaded into the datasets as suggested by the benchmark kits; we never had any custom hints on top of what our method was suggesting.

4.3. Training Parameters and Model Configuration

The GNN cardinality used on the logical query graph consisted of: 3 graph-conv layers (hidden 128), ReLU activations, residual connections, and dropout 0.2. There were inputs of type table/column embeddings (64-dim), predicate type embeddings, and numeric features (NDV, histogram buckets), which were transmitted by layer-norm. Combination of loss and mean squared log error and a q-error penalty; algorithm AdamW (lr=1e-3, weight decay=1e-4), batch size=64, early stopping with JOB validation. An operator-time correction predictor (3-layer MLP (256-128-64)) of the features (estimated rows/widths, join method, parallel degree, I/O hints, cache state) was implemented using the cost residual model. To minimize the effects of outliers, we

trained per-operator head Huber loss (lr=5e-4, batch 128). XGBoost (depth 6, 500 trees, learning rate 0.05) was used as interpretable baselines.

The RL planner (PPO) consumed the partial-plan state concatenated with GNN embeddings and residual features. Policy/value networks: 2×256 MLPs; clipping ε=0.2, γ=0.99, GAE λ=0.95, entropy bonus 0.01, learning rate 3e-4, rollout length 2,048, 8 parallel environments, 10 epochs per update. Rewards combined negative predicted latency with risk penalties for spill probability and variance (weights tuned on TPC-H SF10). Offline pretraining used imitation learning on top-k historical plans; online fine-tuning was canaried on ≤10% of traffic.

4.4. Performance Metrics

Assessment of quality of decisions in realistic constraints. End-to-end query latency (p50/p95/p99) was the primary measure, and the improvement in tail-latency over the CBO baseline. We tracked plan stability (Jaccard similarity of operator sets across param instantiations), reoptimization rate (fraction of queries triggering mid-query re-planning), and SLO miss rate under a 2x baseline p95 budget. In order to measure the quality of estimation, we reported cardinality q-error (median/95th) and cost calibration (negative log-likelihood and calibration slope on operator times).

Operational safety metrics included AI deferral rate (fraction of decisions falling back to baseline due to high uncertainty), sandbox aborts (guarded trials exceeding timeouts), and training overhead (GPU hours and wall-clock for nightly fine-tunes). Measured serving overhead (p99 inference latency of LCE/NCC/RL calls) to make certain that control-plane latency was sub-milliseconds and did not significantly increase planning time.

5. Results and Discussion

5.1. Comparison with Traditional Optimizers

In benchmark suites (TPC-H/DS, JOB), the AI-assisted optimizer was always better than the baseline CBO in the quality of a decision. The greatest improvements were no correlation heavy queries in which classical independence assumptions inflated selectivity errors. Fan-out combined with non-linear feature interaction captured (e.g. CatBoost) models combine with predicate selectivity and composite indexes coverage with spill risk being translated to more trustworthy plan selection. There was also an increase in accuracy in terms of higher precision/recall: when the model estimated that a plan would be faster, it was more likely to be accurate, which minimized costly misplans and re-optimizations. These trends are reflected in the comparative summary below. While the absolute values vary by workload, the pattern is consistent: learned models raise accuracy ~11 points and F1 by ~9 points over the traditional stack. This optimism is based on superior calibration and not ambitious exploration the uncertainty gate eliminated cases with low confidence, maintaining stability.

Table 1: Accuracy and Classification Metrics by Optimizer Type

Optimizer Type	Accuracy (%)	Precision (%)	F1-Score (%)
Traditional	48	53	51
CatBoost (AI)	59	62	60

5.2. Query Execution Time Improvement

Latency reductions were most pronounced for complex multi-join queries with selective predicates, where classical plans over- or under-estimated join methods. The AI layer guided the planner to hash joins with runtime filters or index-nested-loops where suitable, to operator chains and spills. Speedups on mixed analytical workloads were small and steady and p95/p99 tails were reduced significantly which is essential to SLO compliance. The overall outcome is as below. The traditional column has a zero "reduction" of the baseline. The average reduction of (ii) better join ordering, (ii) correct cardinality projections that did not build mis-sized hash tables, and (iii) selective re-planning deferrals in the face of high uncertainty all led to a reduction of AI-optimal plans by a factor of 40.

Table 2: Average Query Execution Time Reduction

Method	Avg. Execution Time Reduction (%)
Traditional	0
AI-Optimized	40

5.3. Model Convergence and Learning Efficiency

Gradient-boosted and modern neural components had a rapid convergence as depicted in training curves. The tree ensemble provided good accuracy in small number of epochs/rounds, due to its use of robust and well-crafted features (histograms, NDV,

cache hints). The RL policy had the advantage of imitation learning and a surrogate reward (neural cost), which reduced the cost of exploration and unsafe experimentation. Interestingly, convergence rates were also related to encodings of features: more elaborate predicate encodings and embeddings in join-graphs decreased the number of epochs to make 90% of the correct predictions. Table as below summarizes the representative convergence rates. CatBoost tended to stabilize fastest, which is beneficial in terms of retraining it often (e.g., nightly). The logistic regression was disadvantaged with linear assumptions that did not fit cross-feature effects.

Table 3: Model Convergence Rates

Model	Convergence Rate (Epochs to >90% accuracy)
CatBoost	15
LightGBM	20
LogisticReg	35

5.4. Scalability and Resource Utilization

At serving time, the trained modules introduced sub-milliseconds of inference overhead but less overall resource usage, by not generating pathological plans (spills, oversized hash tables, too much materialization). With peak windows, the CPU decreased because fewer re-optimizations and retries were caused; the memory consumption decreased because of the superior choice of the join methods and batches size. Notably, even with increasing query volume, these savings did not decrease, a good result indicating positive scaling of the control plane and constant load model latencies. The table below brings the conventional system to 100 percent. The AI-optimized architecture used less CPU and less memory by a factor of 25 and 30 to make similar throughput, creating headroom in the bursty analytics and lowering the cost of infrastructure on a meter.

Table 4: Scalability and Resource Utilization

Metric	Traditional	AI-Optimized
CPU Usage (%)	100	75
Memory (%)	100	70

5.5. Limitations and Error Analysis

Even after the improvements, there were still a number of failure modes. First, out-of-distribution predicates (rare ranges, unseen LIKE patterns) elevated cardinality q-errors, occasionally nudging the planner toward suboptimal join orders. Second, schema or hardware drift (new indexes, firmware updates) momentarily impaired the cost calibration until it was recovered by feedback retraining. Third, the UDFs and external sources continued to be a problem: opaque cost profiles and non-deterministic latencies posed a constraint on the accuracy of the analytical model, as well as the learned model. Lastly, the RL component might overfit to common templates; guardrails reduced this but did not remove the cases of conservative deferral in addition to a slight increase in planning time.

6. Conclusion and Future Work

The work introduced a query optimization system based on AI, which uses learned cardinality estimation, neural cost correction as well as a reinforcement-based planner, encased in uncertainty-sensitive guardrails and a closed-loop system. Throughout the decision-support benchmarks, the system enhanced plan accuracy, minimized end-to-end latency particularly at the tail and minimized CPU/memory use through plan pathology avoiding. Crucially, the design retains the modularity and safety of traditional optimizers: learned components are advisory, instrumented, and revertible, enabling incremental adoption in production environments where predictability and SLO adherence matter as much as raw speed. Despite these gains, residual challenges remain.

Out-of-distribution predicates, opaque UDFs, and sudden schema or hardware changes can momentarily erode calibration before feedback retraining recovers, and RL policies risk template-specific overfitting without careful replay diversification. Another finding that we could make is that the observability quality, directly, restricts the learning quality: the absence of gaps in telemetry slows down convergence and makes guardrails too conservative. The future work direction will be in three directions. First, robustness: more expressive predicate encoders (e.g., sketch-augmented tokenization, on-the-fly sampling), training that is more distributionally-robust to limit worst-case error, etc. Second, expanded scope: everyone can learn about something (physical design: indexes, materialized views), orchestrate resources across multiple tenants, and work in a cross-engine/federated environment with privacy-preserving training. Third, operability and trust: better calibration and counterfactual explanations of plan choices, policy audit and automated rollback playbooks. Together, these advances can turn AI-assisted optimization from a high-leverage add-on into a default capability of modern data platforms.

References

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

[2] Chen, Z., Gehrke, J., & Korn, F. (2001, May). Query optimization in compressed database systems. In Proceedings of the 2001 ACM SIGMOD international conference on Management of data (pp. 271-282).

[3] Azhir, E., Navimipour, N. J., Hosseinzadeh, M., Sharifi, A., & Darwesh, A. (2019). Query optimization mechanisms in the cloud environments: A systematic study. International Journal of Communication Systems, 32(8), e3940.

[4] Li, G., Zhou, X., & Cao, L. (2021, October). Machine learning for databases. In Proceedings of the First International Conference on AI-ML Systems (pp. 1-2).

[5] Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017, May). Automatic database management system tuning through large-scale machine learning. In Proceedings of the 2017 ACM international conference on management of data (pp. 1009-1024).

[6] Schüle, M., Simonis, F., Heyenbrock, T., Kemper, A., Günnemann, S., & Neumann, T. (2019). In-database machine learning: Gradient descent and tensor algebra for main memory database systems. In BTW 2019 (pp. 247-266). Gesellschaft für Informatik, Bonn.

[7] Günnemann, S. (2017). Machine learning meets databases. Datenbank-Spektrum, 17(1), 77-83.

[8] Tzoumas, K., Sellis, T., & Jensen, C. S. (2008). A reinforcement learning approach for adaptive query processing. History, 1-25.

[9] Nogueira, R., & Cho, K. (2017). Task-oriented query reformulation with reinforcement learning. arXiv preprint arXiv:1704.04572.

[10] Rosset, C., Jose, D., Ghosh, G., Mitra, B., & Tiwary, S. (2018, June). Optimizing query evaluations using reinforcement learning for web search. In The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval (pp. 1193-1196).

[11] 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.

[12] Wu, Y. (2020). Cloud-edge orchestration for the Internet of Things: Architecture and AI-powered data processing. IEEE Internet of Things Journal, 8(16), 12792-12805.

[13] Bai, X., Zhang, H., & Zhou, J. (2014). VHR object detection based on structural feature extraction and query expansion. IEEE Transactions on Geoscience and Remote Sensing, 52(10), 6508-6520.

[14] Johnson, C. R., Glatter, M., Kendall, W., Huang, J., & Hoffman, F. (2009, May). Querying for feature extraction and visualization in climate modeling. In International Conference on Computational Science (pp. 416-425). Berlin, Heidelberg: Springer Berlin Heidelberg.

[15] Hellerstein, J. M. (1998). Optimization techniques for queries with expensive methods. ACM Transactions on Database Systems (TODS), 23(2), 113-157.

[16] Boz, O. (2002, July). Extracting decision trees from trained neural networks. In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 456-461).

[17] Hueber, C., Horejsi, K., & Schledjewski, R. (2016). Review of cost estimation: methods and models for aerospace composite manufacturing. Advanced Manufacturing: Polymer & Composites Science, 2(1), 1-13.

[18] Sethi, I. K. (2002). Entropy nets: from decision trees to neural networks. Proceedings of the IEEE, 78(10), 1605-1613.

[19] Shah, H., & Gopal, M. (2010). Fuzzy decision tree function approximation in reinforcement learning. International Journal of Artificial Intelligence and Soft Computing, 2(1-2), 26-45.

[20] Belling, P. K., Suss, J., & Ward, P. (2015). The effect of time constraint on anticipation, decision making, and option generation in complex and dynamic environments. Cognition, Technology & Work, 17(3), 355-366.