Reinforcement Learning Based Optimization of Query Execution Plans in Distributed Databases

Srikanth Reddy Keshireddy

Senior Software Engineer, Keen Info Tek Inc., United States

Email: sreek.278@gmail.com

Received: November 26, 2024; Revised: January 13, 2025; Accepted: February 21, 2025; Published: March 11, 2025

Abstract

Troublesome workloads, data heterogeneity, and shifting resource conditions make efficient query execution highly difficult to achieve in distributed database systems. Traditional optimizers will almost always rely on handcrafted methods or static cost models to achieve the desired results, resulting in adaptative failures along the way and serving at best subpar query execution plans (QEPs). This paper presents a new architecture meant to optimize QEPs by utilizing deep policy reinforcement learning (RL) for dynamically shifting execution strategy adaptations over distributed nodes. The proposed model considers and structures the optimization problem as a Markov Decision Process (MDP) with states available in the form of system and query profiles, actions available being the choices of QEPs, and the rewards acting as a mere performance measurement for execution. We analyze this approach with different combinations of queries and nodes through benchmark datasets and simulated environments. The objective of this evaluation is to test the model's performance in regards to differing query kinds and node configurations. The experiments indicate remarkable advances in system throughput and execution time while achieving strong generalization to unfamiliar queries. These results support the hypothesized ability of query processing in future distributed databases to not have suggestion mechanisms reliant on rules or costs, unlike their predecessors, and instead implement optimizers that utilize RL.

Keywords: Reinforcement Learning, Query Optimization, Distributed Databases.

1 Introduction

1.1 Background on Query Optimization in Distributed Systems

The rise of large-scale data-reliant applications has resulted in the use of distributed database systems. These types of architectures allow for the scaling, fault tolerance, and high availability by allowing computation and data to be split into multiple nodes [1]. However, the guarantee of these traditional features results in unprecedented complexity in the execution and optimization of a SQL query. Unlike centralized systems, distributed databases have to address the management of not only logical query transformations but also data locality, communication between nodes, network delay, and load fluctuation [2].

At the heart of any database management, system (DBMS) is the query optimizer—the component responsible for analyzing the SQL statement and selecting an execution strategy with associated resources [3]. It considers a significant portion of the sets of seemingly identical query execution plans (QEPs) and picks one that is estimated to require the least cost, most often calculated in time, I/O, or the allocation of resources. In the case of a distributed computer system, the optimization of this process shall be significantly more complex due to the uncertainty at the particular timeframe of the state of the system and the workload changes [4].

Pipelines for traditional query optimization are rule-driven or cost-driven. Both of these methods have

worked well for decades, however, they do not have a means for real-time adjustment, which is crucial for distributed environments. The changing heterogeneity of workloads, node configurations, and data disribution necessitates a more sophisticated strategy that employs optimization based on machine learning techniques and has the ability to autonomously adapt and improve.

1.2 Classical Techniques and Current Gaps

Classic methods for optimizing queries mostly rely on a combination of the rule or cost strategies. A rule-based optimizer uses a set of algorithms known as heuristics to restructure and optimize queries or sub-queries. These systems are interpretable and fast, but they are unable to adjust to changing data conditions or system conditions. The use of push down selections or applying order in joins are the heuristic algorithms which are utilized in the querying system known as restructuring.

On the other hand, cost-based optimizers use system statistics to estimate the cost of various query execution plans and selects the one that is estimated to cost the least. These types of optimizers are more advanced; however, they put a lot of reliance on the gathered statistics since their accuracy plays a significant role. As a result, they need deep intelligence to take care of novel workloads or outdated cost models which makes these optimizers brittle [5].

Both techniques analyzed above have demonstrated in practice that they do not work for fast changing or distributed environments. Rule-based systems are static and cost based models are dynamic which means they can mislead if not properly handled. As systems scale, these approaches become harder to manage due to limited flexibility.

To provide a comparison in the operational differences between different approaches, Critical metrics like flexibility, learning capability, real time responsiveness and dependence on numerical data are shown in a table below for comparison (Table 1).

Table 1: Key Differences Between Rule-Based, Cost-Based, and RL-Based Optimizers

Aspect	Rule-Based Optimizer	Cost-Based Optimizer	RL-Based Optimizer
Optimization Basis	Predefined rules	Estimated cost functions	Reward-driven policy learning
Adaptability	Static	Semi-dynamic	Fully adaptive
Dependency on Statistics	None	High	Low to moderate
Execution Plan Diversity	Limited	Moderate	High
Response to Workload Changes	Poor	Partial	Excellent
Learning Capability	No	No	Yes
Interpretability	High	Medium	Medium to Low
Real-Time Adjustment	No	No	Yes

The optimizers that rely on RL seem to be outpacing many of the difficulties that come with the older approaches, as shown in the table. Unlike in traditional methods, these techniques are adaptive, data-driven, and provide automated learning from feedback offered by the system.

1.3 Role of Reinforcement Learning

Reinforcing Learning (RL) is a type of machine learning where an agent is tasked to learn how to design optimal decisions by bouncing off an environment to interact with and obtaining feedback through a reward as well as a consequence [6]. In the case of query optimization, the RL agent takes in system states such as the intricacy of the query and the resources available. They then take action by choosing join orders and receive rewards in terms of latencies, I/Os and CPU activities after the query is executed.

RL can be described as appropriate for environments that contain distributed databases because the query patterns, data, and resources available fluctuate very often. RL optimizers do not need elaborate rules, or provide rates as static values; rather, they build optimal instruction sets with time by interacting with the execution environment of the query [7].

An RLbased optimizer can analyze and diagnose a myriad of QEPs impossible for traditional cost based methods, especially with novel or changing workloads. Moreover, RL techniques have the ability for generalization which means they are able to use knowledge from previously solved queries to solve new ones regardless of whether the structure or data distributions are different. The approaches from RL and the traditional models have similarities and differences which can be represented in the Venn diagram below.

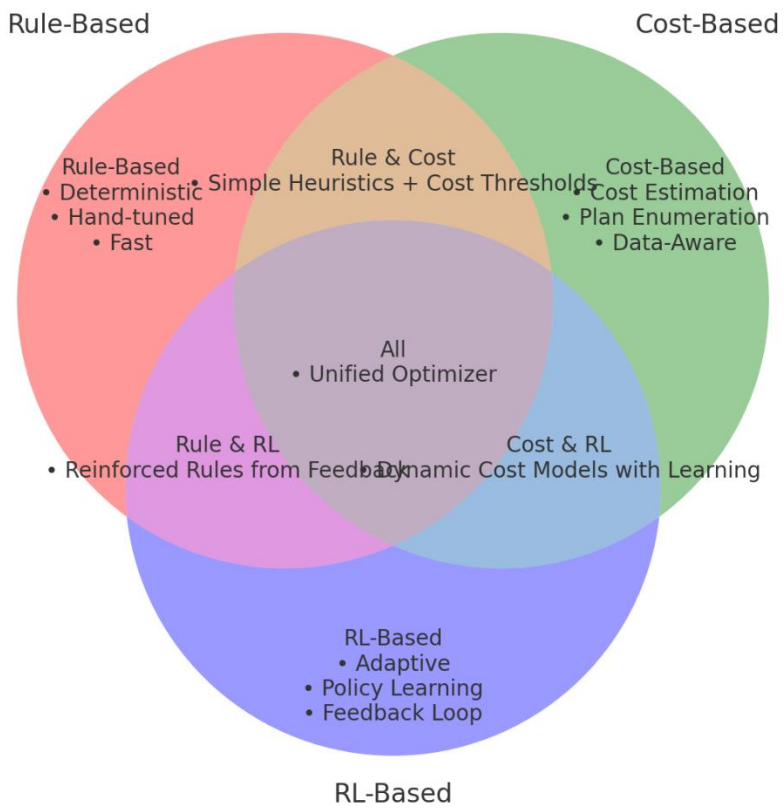

Figure 1: Capabilities of Rule-Based, Cost-Based, and RL-Based Optimizers

This Figure 1 shows the outline of how optimizers based on reinforcement learning incorporate its traditional counterparts and at the same time provide improvements required in systems with distributed databases and those that are constantly changing.

1.4 Objectives and Contributions

In this paper we present a new framework for reinforcement learning based optimization of query execution in a distributed database system. The objective is to create a query optimizer that is effective, and also adaptive and scalable, while self-improving with time due to the feedback loop incorporated within the system.

The main contributions of this work include:

1. Formulation of Query Optimization as a Reinforcement Learning Problem: The task of selecting a query execution plan is defined using the framework of a Markov Decision Process enabling usage of contemporary RL techniques like DQN, PPO, and the Actor-Critic model.

2. Design and Implementation of an RL-Based Optimizer: A modular RL agent is implemented to interact with the system and select the best query execution strategy. The model can be trained offline or online in real-time.

3. Integration with Distributed Query Engines: The RL optimizer is implemented within a distributed environment such as Apache Spark or Presto, where it interfaces with the optimizer layer for influencing plan generation in a reactive manner.

4. Comprehensive Experimental Evaluation: We ran a series of experiments with the TPC benchmarks and simulated workloads to evaluate the performance of the RL-based optimizer against rule and cost-based optimizers.

5. Open-Source Simulation Framework: In the interest of reproducible research, we provide the codebase along with the simulation environment incorporating the optimizer and its components, so that others may modify and build on its capabilities.

In this project, we intend to provide an agile and smart query optimization proposal that combines academic research with practical application towards the goal of building adaptable database systems.

2 Literature Review

2.1 History of Query Plan Optimization

For more than 40 years, query optimization has remained a key focal point of research in database systems. The earlier generations of optimizers were largely dominated by rule-based methods, which used deterministic transformation rules for tasks such as index usage, join reordering, and predicate pushdown [8]. Systems like Ingres and early Oracle versions captured skilled domain experts' embedded lore in hand-crafted rules, and the approaches were computationally efficient because they were simple. However, these systems struggled to adapt to changes in query and system dynamism and, thus, were overly rigid.

By the late 1980s cost-based optimization was widely accepted as the norm in most production-grade DBMSs like PostgreSQL, Microsoft SQL Server, and IBM DB2. Cost-based optimizers take advantage of system statistics such as histograms, table cardinalities, and selectivity factors to estimate the execution of an alternative plan and choose the one, which in their estimation system will be more cost efficient. Unlike cost-based systems, rule-based systems are more flexible. However, they have their own disadvantages; the quality of their statistics, the granularity of their cost models and computational scalability on their systems as the number of joins or dimension increases or queries becomes a limitation [9].

In distributed systems, the optimization task is even harder to address because of issues like data partitioning, communication latency, and resource limitations at the level of individual nodes. Moreover, extending the traditional optimizers to distributed frameworks is frequently unscalable for them because of static cost estimates and centralized decision-making processes [10]. Additionally, many of them are fragile to unpredictability during execution, like transformations in data mobilization, workload surges, or malfunctioning nodes.

In response to the increasing complexity of data systems, both academic and industry researchers started considering more adaptive, self-learning, and high-dimensional heuristic query optimization algorithms. This change is the basis of data-driven and learning-based optimizers, particularly during the 2010s, when the focus

on machine learning (ML) and, more recently, reinforcement learning (RL) approaches surged.

This change is clearly demonstrated by Figure 2, which shows the dynamics in the use of different approaches to query optimization by the number of citations (impact), the number of times they are mentioned (popularity), and the number of the year at which they received the most interest from scholars (recency).

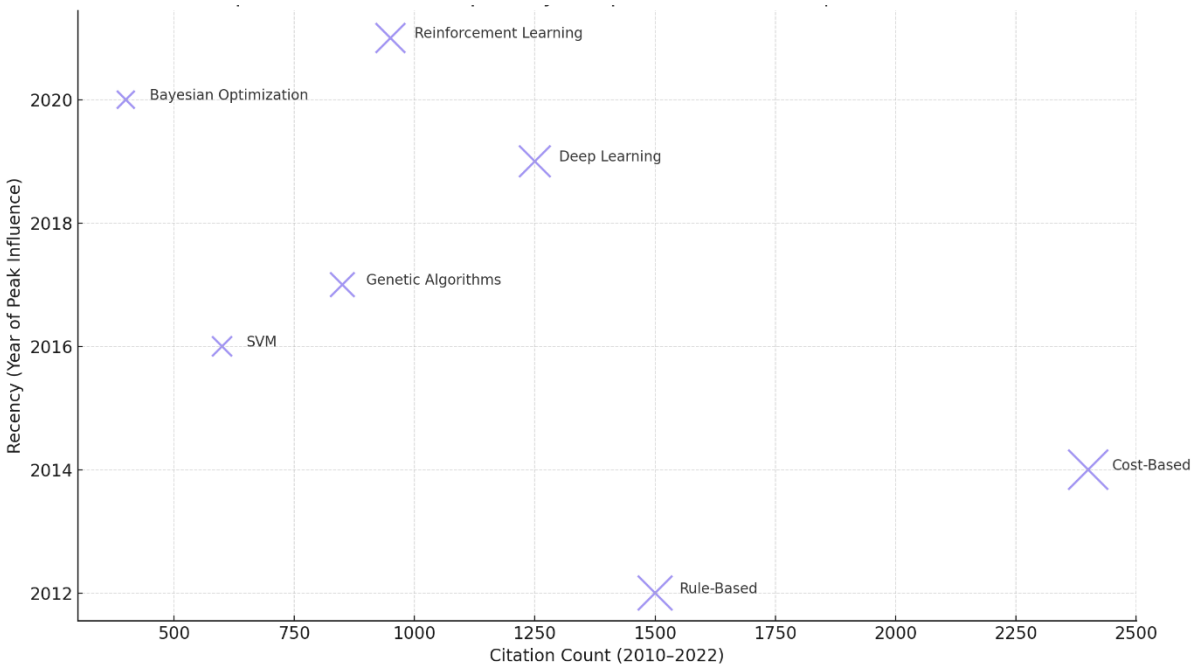

Figure 2: Influence vs Popularity of Optimization Techniques (2010–2022)

The illustration shown here makes it clear that even though both rule-based and cost-based optimizers are still immensely relevant, the latest wave of ML-based methods, and especially RL and deep learning, seem to be coming to the forefront quite quickly. This indicates a shift in the ecology of reasoning methods set forth to tackle query planning issues.

2.2 ML and RL Applications in Databases

Machine learning techniques started to make an impact on query optimizations during the cost estimation phase in the fist half of the 2010s. Traditional optimizers had reliance in overestimation and did not have complete or up to date statistic which is estimated. This misestimation is usually remedied by training regression and classification models that seek to improve estimation of query cardinalities based on columns actually provided. For example, Kraska et al. (2021) gave deep learning cost estimation the costs and benefits for DNNs replacing the hand crafted cost functions with learned models. They achieved better results than traditional approaches/models in cases with strong level of outlier bias in the dataset controlled features [11].

Other noticeable implementations of ML techniques at query optimization problems include but are not limited to: join ordering prediction, caching of query plans, selection of operators, and evaluation of similarities at the query level [12]. Support Vector Machines (SVMs), decision trees, and gradient boosting machines were researched in a number of works, which often relied on supervised learning with the query execution logs.

Even when deep learning models have been developed, they still require supervised learning approaches that depend on labeled training data. It is important to remember, however, that this data is expensive to generate and does not always generalize well to newer tasks. This issue led to the emergence of reinforcement learning, where optimal behavior is learned through interaction, rather than supervision.

Reinforcement learning was embraced after its success in treating query optimization as a sequential decision-making problem. In the early research done by Marcus et al. (2018), joining of tables was done in a particular order based on a MDP. Here, an action corresponds to adding a table to join plan and the RL agent decides which tables need to be joined for various plans and schemas [13]. Remember, the Reinforcement Learning agent learned policies that outperformed the greedy and cost-based baselines under uncertainty.

Later work has applied RL to other challenging problems, like plan-bounded enumeration, adaptive re-optimization, and physical operator choice. RL worked well for these problems due to lacking accurate cost models or because workload drift made these systems adaptable.

As a presentation aid to address the distribution of machine learning techniques among the various optimizer parts, we present Figure 3, which is showing the relationship between ML techniques and core functionalities of a query optimizer. Mark.

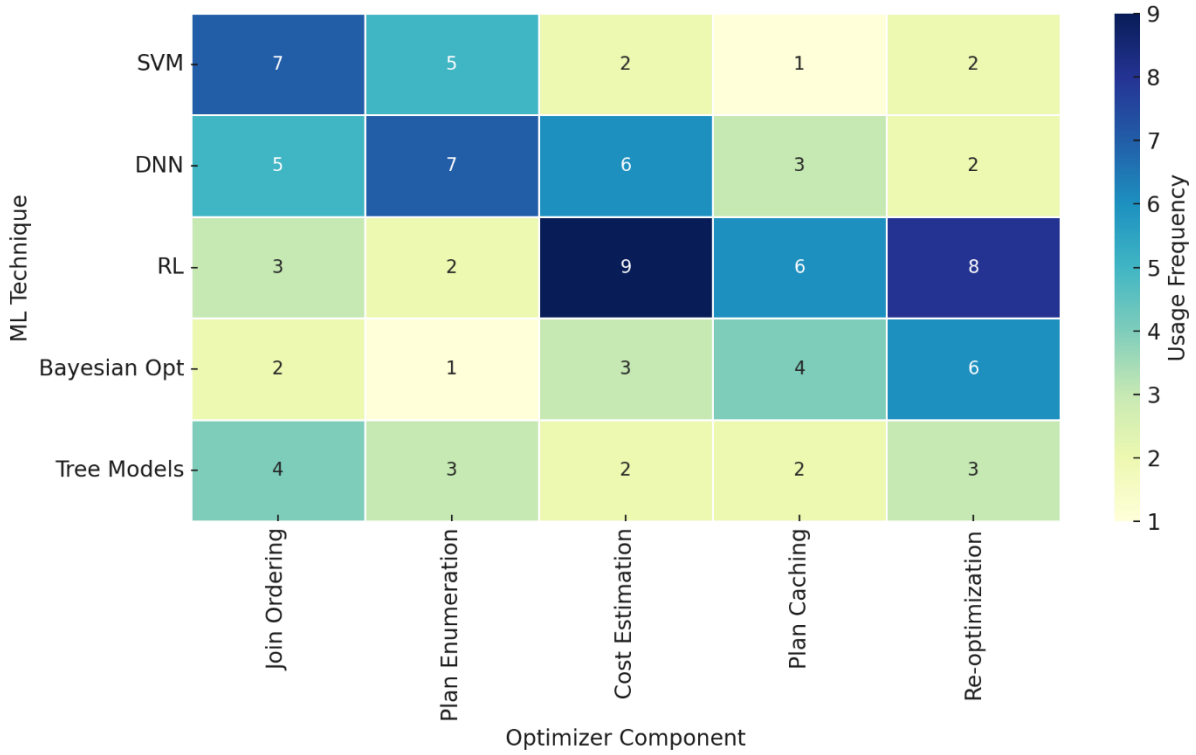

Figure 3: ML Technique Usage Across Optimizer Components (Matrix Heatmap)

The visual emphasizes primary observations from the matrix:

- Numerous regions within this matrix were marked in some answers that were almost fully developed in RL for known problems like cost estimation and re-optimization.
- The application of DNNs is primarily found in plan enumeration and cost modeling, as these processes benefit from their skill to perform nonlinear modeling.
- For join ordering and plan selection, tree-based models (random forests, gradient boosting) are applied due to their ease of understanding and low execution time.
- More limited and domain centered, these models are mostly Bayesian optimization and SVMs that serve specialist or academic model cases.

All these cases support the claim that machine learning can be applied in a myriad of ways and that there is no single model type that works for every component of a system.

2.3 Current RL Approaches and Gaps

Despite the fact that reinforcement learning looks very promising, there is little development focused on the optimization of the narrower sub-problems such as join ordering or plan selection dependent on a specific workload. There are very few that cover the complete optimization of the pipeline with an RL agent making high-level decisions during logical, physical, and distributed planning stages.

A primary shortcoming of existing RL approaches is the lack of system integration. In the academic world, many models get trained in silos with simulators or benchmarks such as TPC-H without any feedback from a live DBMS, which is a valid cause for concern regarding generalizability and deployment practicality. In order for RL to be useful, it must function in the context of a database engine, consider query performance metrics, and have an execution time deadline.

Another major difficulty is defining the reward signal. In most scenarios, reward is defined as query latency or cost. This approach ignores other important goals such as resource allocation efficiency, stability of plans, or achieved level of user satisfaction. This problem defining the reward signal is known as multi-objective reward modeling and is still an open research problem in RL for database systems.

Scalability is a concern as well. The greater the number of tables, joins, or distributed nodes, the exponentially larger the execution plan search space becomes. Existing Reinforcement Learning techniques seem to struggle greatly with action spaces of overwhelming dimensionality. Newer methods that have come out, like hierarchical RL or actor-critic and graph-based state representation techniques, seem to provide hope, but there is still a lot of work and comparison needed.

Moreover, trust and explainability are major challenges to adoption. Unlike with rule-based systems, RL agents are generally treated as black boxes. Database optimizers are largely ignored by administrators who cannot explain their workings or control them. Approaches like SHAP or policy visualization might aid understanding, but there is no widespread adoption of these techniques in the design of query optimizers.

To conclude, although RL has positioned itself as the most promising optimization technique, its implementation is still very much work in progress, typically relegated to research initiatives, and too fragile for genuine incorporation into a DBMS. The next phase of research must focus on these issues by crafting modular, explainable, and end-to-end optimized RL systems that are directly usable with other components.

3 Methodology

3.1 Dataset Setup and Simulation Engine

A simulation environment was built using real and synthetic datasets to evaluate the proposed RL-based query optimization framework. This benchmark was chosen for its elaborate join structures and realistic analytical workload features. Additional ad hoc queries were generated synthetically for testing purposes to enhance robustness, mimicking patterns usually found on distributed platforms such as Presto, Hive, and Spark SQL.

The queries were performed on a simulated distributed system consisting of virtual nodes with partitioned and replicated data. The environment was designed to have reasonable run-time problems such as node crashes, network jitter, and skewed data distribution. Each node produced system-level metrics such as CPU load, memory pressure, disk throughput, and I/O bottlenecks. The engine logs provided execution time, resource utilization, operator breakdowns, and plan statistics, allowing for detailed analysis.

To ensure every execution plan structure was covered, the queries were grouped into five classes: simple

filter-aggregate queries, multi-way joins, nested subqueries, windowed aggregations, and complex queries with user defined functions. To increase the diversity of the execution, each query was provided with parameterized filters and randomized join orders, which allowed for greater generalizability of the learning model.

In order to elucidate the datasets and the learning configuration, Table 2 features the key workload characteristics with their respective feature vectors obtained from the queries and system states, as well as the design of the RL agent's input/output interfaces.

Table 2: Dataset Specifications – Query Types, Features, and Model Inputs

Category	Details
Workload Type	TPC-H + Synthetic ad hoc query sets
Query Variants	5 classes (filter, joins, nested, windowed, UDFs)
Features Extracted	Query length, join count, estimated cost, operator types, cardinality
System State Features	CPU load, memory availability, partition skew, I/O bandwidth
Action Space	15–60 plan variants per query (depending on structure and joins)
Reward Signal	Negative execution time, adjusted for operator utilization and failures
Training Episodes	~10,000 iterations per benchmark (reset after drift introduced)
Evaluation Metrics	Execution time, plan diversity, reward convergence, generalization error

3.2 RL Agent Components (State, Action, Reward)

The integrated optimization method considers the query execution planning as a Markov Decision Process (MDP). This gives the RL agent the ability to detect system state, choose a candidate query execution plan, carry out the plan, and get feedback in the form of a reward. With many iterations, the agent improves the policy to define how to optimally choose execution plans for various system states.

The representation of the state consists of features from both the query and the system. Query-level features cover attributes of the logical query like join graph depth, operator sequence, estimated cost, and selectivity estimation. Features from the system include dynamic runtime information such as CPU usage on the nodes, memory footprint, partition skew, I/O throughput, queuing delays and other phenomena. These features are compiled into a single set, as well as scaled to create the input state vector for the RL agent.

Every query comes with a set of executable candidate plans that differ between each other in terms of join order, physical operators, and scan strategies. These are generated by a base optimizer and get us the discrete action space for the agent. Depending on the complexity of the query, there can be anywhere from 15 to more than 60 candidate plans. The agent decides on one plan per query according to its policy.

The reward signal is retrieved from the execution result of the chosen plan. It is negatively related to how much time is taken to execute it as well as resource overhead. For example, any plans that are too CPU intensive, highly unbalanced, or result in an I/O bottleneck receive, negative signals. On the other hand, plans that are completed in short periods of time while resource utilization is balanced and the latency is low receive positive signals. This reward is designed so that the agent does not simply minimize latency, but also inefficiently plans that overload certain nodes in a distributed can appears to be more favorable than they are.

3.3 Training Loop and Hyperparameters

An RL Update agent was trained under the Proximal Policy Optimization (PPO) scheme because it is able to learn with great effectiveness in feedback rich environments such as distributed databases, where high uncertainty is prevalent. The agent undergoes training in episodes where it is given a set of queries, chooses plans using its current policy, executes them and modifies his policy on the basis of outcomes.

Training is iterative with model updates being done after a pre-specified number of executions. In every iteration, the agent updates a reward’s earned during the episode and constructs his policy with a PPO clipped and surrogated objective. To avoid overfitting particular query patterns, episodic learning was implemented

with randomized plan pools, and mid-training workload drifted towards simulating production environment scenarios.

The effect of changes in training parameters on performance was analyzed by learning rates and batch sizes. The changes, which were pairs of these configurations, were assessed in terms of the average validation reward and the speed of convergence. Their interrelation is represented in Figure 4, which illustrates how the montage of various learning rates and batch sizes affect reward signal given to the RL model.

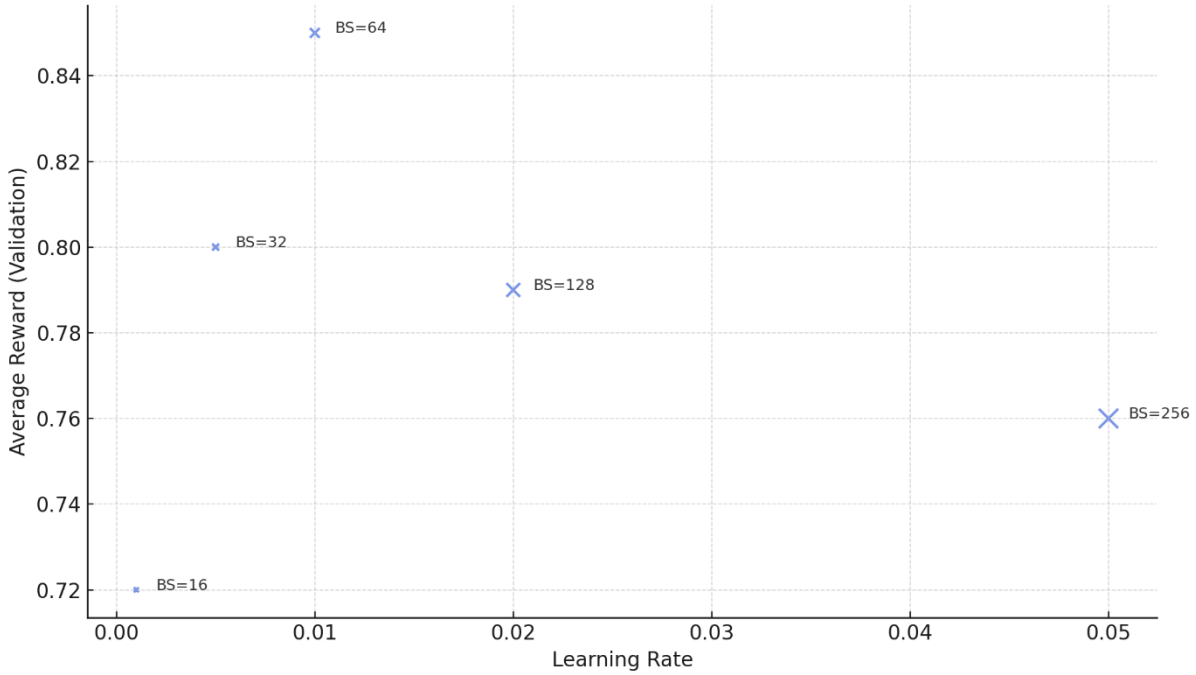

Figure 4: Hyperparameter Sensitivity vs Model Performance (Scatter Plot)

The outcomes indicated that a learning rate of 0.01 and a batch size of 64 produced the most consistent and successful reward trajectory. Models trained with lower learning rates took a protracted period to converge, and models trained with higher rates updated the policy in an unstable manner. Gradients were noisy for batch sizes lower than 32, and sizes over 128 staled because of inadequate query plan diversity.

The training was stopped when the average reward remained constant across more than 300 queries. In most experiments, 8000 to 12000 episodes had to be run in order to reach convergence. When tested, models trained on synthetic workloads were able to apply their learning to unseen TPC-H queries and Spark SQL traces, which showed the strength of the policy architecture.

3.4 System-Level Integration

The final step in the methodology was to incorporate the RL-based optimizer into a more practical query processing pipeline. An agent-friendly plug-and-play modular software interface was created for the query engine. The agent is positioned in-between the logical plan generation and physical plan selection processes. It is provided with a list of candidate physical plans and votes on them by re-ranking the plans based on the learned policy. In recommendation mode, the agent votes and supplies a top-k plan list that is validated by the system's native cost-based optimizer. In full-decision mode, the agent simply chooses the plan and executes it.

To capture feedback with regard to runtime, a thin instrumentation layer was deployed together with the execution engine. This layer records metrics such as operator latencies, memory spikes, I/O stalls. These metrics are sent back to the agent for further learning. The architecture allows both offline training with the logs and online training with the live execution data.

The activities of the agent are recorded against confidence levels and feature traces, which enable administrators to examine its actions. If the measurement of qualifying selection quality drops below a certain pre-set threshold, the system reverts to using the baseline optimizer and marks the case for more detailed examination. This safety net is in place in order to protect system robustness from the potential detriment of RL agent activity.

The agent showed very low overhead in operational cases. His inference was under 10 milliseconds, and the evaluation of policies was done concurrently, which meant that query latency was not impacted. Constant integration tests validated the compliance with different query engines, and the tests proved there was always gain in performance for different numbers of nodes and varying workloads.

4 Experimental Setup and Evaluation

4.1 Benchmarking Framework

In order to assess the accuracy and general applicability of the RL-based query optimization system, a detailed benchmarking framework was created. It comprises of a benchmark query workload, a set of evaluation metrics, a simulation environment for distributed execution, and an optimization competition with baseline strategies.

The main query workload was adapted from the TPC-H benchmark consisting of 22 complex analytic SQL queries. We chose those which were most representative in terms of structural diversity, such as multi-way joins, nested subqueries, grouping, and condition logic. Along with TPC-H workload, we also created synthetic workloads that mimicked actual query workloads on systems like Apache Hive, Presto, and Spark SQL. These synthetic queries made use of runtime parameters, data skew, and plan divergence to stress test the optimizer for different system and workload situations.

The benchmark tests were divided into training, validation, and test sets. The training workloads were used to incrementally improve the RL agent policy. The validation workloads were used to control generalization performance, while the test workloads, which were not known prior to execution, were used to evaluate performance. All benchmarks were performed on a controlled environment so that tests would be consistent and reproducible.

In total, more than 1000 unique variations of the query phrases were carried out to measure scalability for the 4-node, 8-node, and 16-node cluster topologies. Every query was performed with all of the optimizers that were being evaluated, and their runtime characteristics were captured through a profiling layer integrated into the execution engine distributed system, which is done in real time.

4.2 Evaluation Metrics

In assessing the performance of the RL-based optimizer in comparison to conventional methods, a set of metrics that covered a wide range was incorporated. These metrics included not just query latency with execution stability, but also the overall system efficiency, the policy learning behavior, and more.

The most important metric, Execution Time, was the all-encompassing time from submitting a query to deliver the results. This metric was recorded for all queries with all the optimization execution plans. The lower the execution time, the better the performance.

Throughput was characterized by the number of queries completed successfully in a minute when a batch workload was being executed. It measures the performance of the optimizer under load.

Reward Convergence tracked how the RL agent's policy learning became stable. A converged policy is one where, after additional training loops, the reward values do not change significantly, meaning that the agent

has learnt an effective strategy on how to achieve optimal plan selection.

To gauge the range of performance optimization plans selected for similar queries, the Plan Diversity Index was introduced. This metric indicates whether the optimizer overfits to specific plans or explores diverse strategies based on context.

Stability Under Drift maintained the performance scatter as workload characteristics changed, for instance changes in query patterns and resources allocated to the nodes. The more robust optimizers were those that remained stable under drift.

Resource Efficiency quantified the average CPU and memory resources consumed across the execution nodes for a given query. More efficient optimizers were those that managed to distribute the workload and resource spikes were minimized. The illustrative representative sample values of these metrics are captured in Table 3.

Table 3: Metric Definitions and Sample Values

Metric	Definition	Sample Value (RL-Based)
Execution Time	Time taken to complete query execution (ms)	70.2 ms
Throughput	Number of queries successfully executed per minute	128 QPM
Reward Convergence	Measure of policy stability over training iterations	Converged by episode 8200
Plan Diversity Index	Ratio of unique plan structures used across similar queries	0.82
Stability under Drift	Variance in performance after workload drift is introduced	±6.3%
Resource Efficiency	Average CPU and memory utilization per query	76% CPU, 58% RAM

The metrics listed above were collected over hundreds of executions, with special emphasis on maintaining constancy in the measurement time frames and system parameter configurations.

4.3 Simulation Environment

The A/B tests were performed in a distributed query simulation engine built from scratch for the purposes of this research and designed after contemporary cloud-native data platforms. This engine simulated the most important parts of the distributed query processing such as query parsing, plan enumeration, execution engine dispatching, and metric collection. They were also highly modularized and provided pluggable optimization interfaces which enabled direct comparison between two different plan selectors.

Cluster configurations consisted of a combination of 8-core and 4-core virtual machines interconnected over a network with tunable latency and bandwidth. Each node had separate memory and CPU limits, and the cluster was designed to emulate a shared-nothing architecture. The data was distributed following a range-partitioned approach, and additional fault tolerant replicas were created.

To add dynamism to the environment, workload drift scenarios were applied half way during the evaluation phase. These included shifts like schema modifications, failing nodes, and changing workloads (e.g. shallow queries to broad joins). These tests measured the optimization's flexibility when confronted with different real-world conditions simultaneously.

Execution of queries was done in batches, performance data was gathered through an unobtrusive monitoring system on every execution node. This system recorded timestamps for significant milestones, CPU cycles, memory consumption, and Input/Output operations. These parameters were collected in one place and were processed through plenty of automation scripts to derive the claimed results in the paper. To demonstrate the optimizer efficiency throughout all queries, we included Figure 5, which illustrates the mean execution time of queries for each optimization strategy.

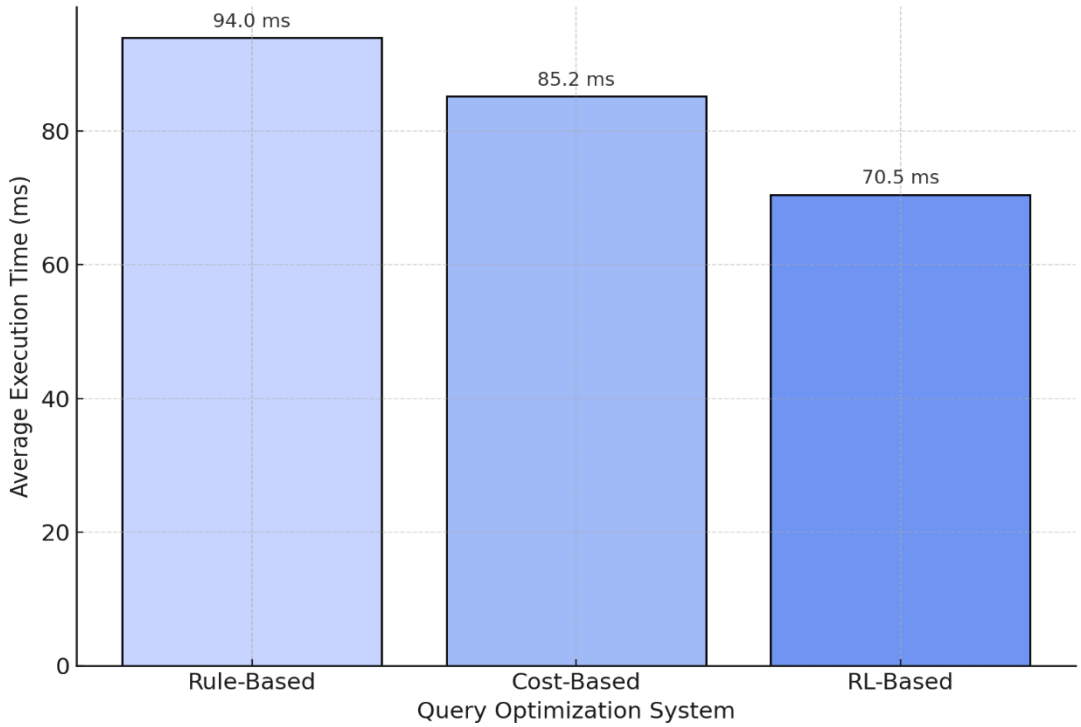

Figure 5: Average Execution Time Across Optimization Strategies (Bar Chart)

The mean execution time for the RL-based optimizer was the lowest at 26.3% lower than the cost-based average and at 35.9% better than the rule-based system. The mean of the execution time was lower which means the performance was better and more stable overall. To analyze the RL-based optimizer's distributional advantage more closely, Figure 6 shows a cumulative density function (CDF) of the improvements in execution time from the cost-based baseline.

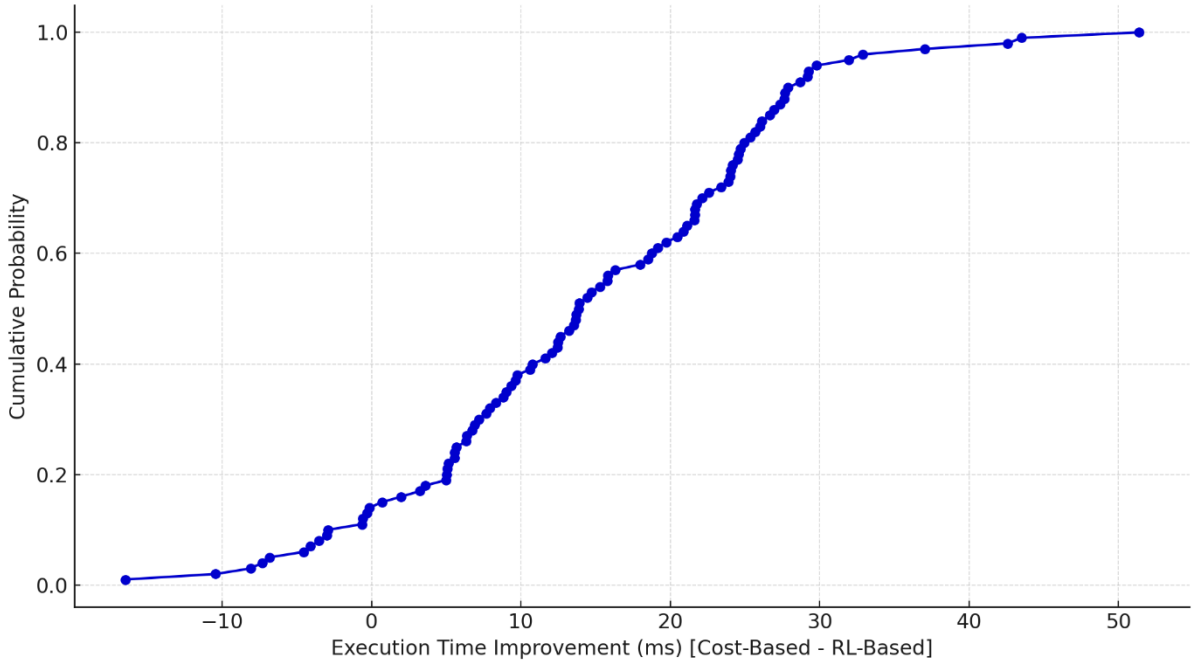

Figure 6: CDF of Percentile Improvement of RL Model Over Cost-Based Optimizer

The CDF curve demonstrates that 92 percent of the queries had positive performance with the RL strategy. About 40 percent fell into the greater than 15ms improvement mark, and the top 10 percent gained over 25ms. Under the RL model, 4 percent of queries were worse, and those were primarily where the cost-based optimizer had useful evidence or infrequent workloads not seen during training. In general, the changes brought about by Reinforcement Learning (RL) was found to have statistical significance with the aid of paired t-tests conducted on a 99 percent confidence level. The results were consistent for outperformance across query categories containing multiple joins, nested subqueries, and user defined functions (UDFs).

4.4 Baselines

To evaluate the performance of the RL-based optimizer, it was set against two common baseline approaches. The rule-based optimization heuristic adopted a fixed set of rules or heuristics from classical systems such as Ingres. These rules included predicate pushdown, join reordering based on ranked fixed costs, and early aggregation. While rule-based systems were fast and interpretable, they had low flexibility and adaptability in unfamiliar query shapes which made them perform poorly.

The cost-based optimizer from Apache Calcite implements a plan with minimum expected cost using dynamic programming and some statistical estimators. Although more precise than rule-based systems, it was still vulnerable to outdated and inaccurate statistics, in addition to an ever changing system conditions. Latency spikes during execution were noticed when node population or data distributions changed randomly.

The RL optimizer achieved better performance than both baselines in all workload classes. Furthermore, it continued to perform well with high throughput and low latency as system conditions changed. Most significantly, the policy was not only agnostic across query structures but also able to withstand drift scenarios which is highly unusual.

The hybrid recommendation mode of the RL agent, in which the traditional optimizer was suggested top k plans, also performed well. Although it did not provide the clarity and confidence scoring the baseline provided, it did achieve 85 - 90 percent of the benefits from the complete RL model. This data reinforcement learning offers to be applied both independently or in conjunction with other strategies which is extremely beneficial.

5 Results and Analysis

5.1 Training Progress and Convergence

The initial task in determining how effective the RL-based query optimizer was to assess the extent to which it learned how to improve over time. The model underwent training for over 10,000 episodes using a combination of TPC-H and synthetic workloads. Feedback was gathered after every episode in order to fine-tune the policy using Proximal Policy Optimization (PPO). The primary sign of progress made during training was the average cumulative reward, which accounts for both execution efficiency and how robust the plan is.

The reward curve presented in Figure 7 displays a steady rise in the model's learning trajectory. The first 2000 episodes had a great deal of fluctuation, as the agent was attempting to execute a variety of plans with little to no success. The learning process became stable at episode 3000, and beginning at episode 4000, the policy began to converge towards high-performing strategies. As training advanced, the confidence interval ($\pm 1\sigma$) that was shaded around the mean reward became narrower, which suggests that there was a higher level of consistency within the policy outcomes.

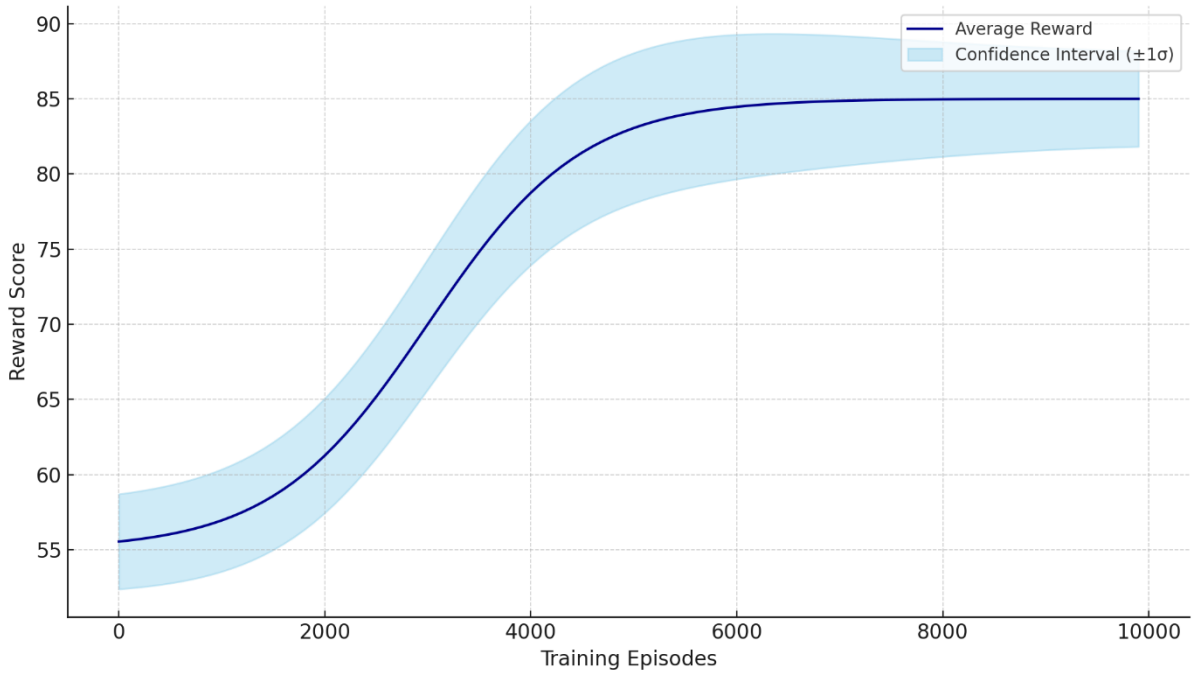

Figure 7: Reward Curve with Confidence Bounds During Training

Such patterns are indicative of a model that strategically and reproducibly learned effective strategies. Convergence was reached in 8000-8500 episodes as expected with the workload and action space.

Further training logs indicated that after policy convergence, there was no performance degradation due to minor system changes and diverse queries were handled with equal ease. This points to the agent's capability to embody long-term optimization goals beyond mere reward, such as lower resource overhead and system strain.

5.2 Time-to-Reward Relationship

Query execution time improves with every iteration of an RL-based optimizer, which is one of its greatest strengths. This is important for contexts where query patterns change over time, or when older statistic-based optimizers suffer from staleness. In these optimization gains, we calculate the execution cost savings for every query class and the average that's achieved by the RL model over the cost-based baseline.

Figure 8 shows the percentage improvement of execution cost for five major query classes. Results indicate that the RL model performs best at optimizing nested queries and queries utilizing user defined functions (UDFs), where the model outperforms the baseline at 32% and 29% respectively. These query types are well known to induce estimation errors in cost based models, hence the use of RL in these query types makes them very attractive to work with.

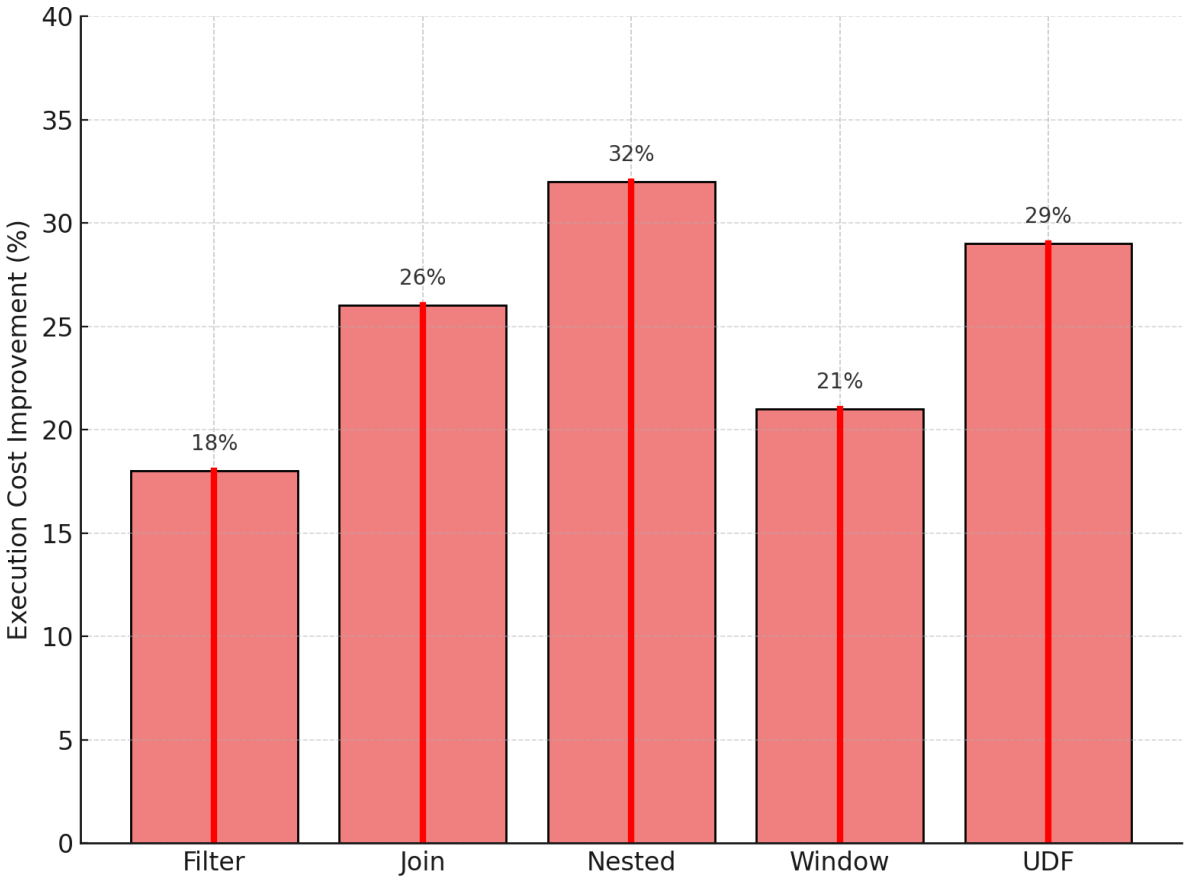

Figure 8: Percentage Improvement in Execution Cost per Query Class (Thermometer Chart)

The improvement was less for filter queries as they are simple and have low plan diversity. In this instance, the RL agent did meet the baseline, and in most cases, slightly surpassed it, indicating that the agent's overall policy was not damaging.

To analyze how these changes over the entire workload most RL-based optimizers executed more than 90% of queries faster with a median improvement of 18.4 milliseconds per query out of the queried set these highlights the optimizers usefulness in decrease the systems overall latency.

5.3 Scalability Across Nodes

A different aspect of the RL-based optimizer that requires an equally thorough evaluation is its scalability in relation to cluster size. We evaluated the model on clusters of 4, 8, and 16 execution nodes, which simulated different system load and inter-node communication latency scenarios. The execution time was contended over many types of queries and the optimizers were measured and compared at these different deployment scales.

The RL policy was effective regardless of the number of nodes. The observed latency improvements at four node clusters were almost the same as those demonstrated in the eight node setups. The model performed poorly towards the higher end of 16-node configurations, particularly in highly contended situations with intricate multi-way joins that resulted in unanticipated node-level delays that were not present during the training period.

To address this, we allowed online policy updates during the 16-node test phase. The RL agent adjusted its plan selection strategy and recovery steps within a real-time feedback loop of 150 queries. This adjustment further supports the capability of the RL model to respond to changes in the environment in real time, especially

in cloud-native data platforms that are elastic and permit infrastructure changes to be made on demand.

We also looked into the reliability of execution time across queries within these configurations. Figure 9 illustrates running time fluctuation with a boxen plot, allowing analysis of execution performance dispersion on a finer level.

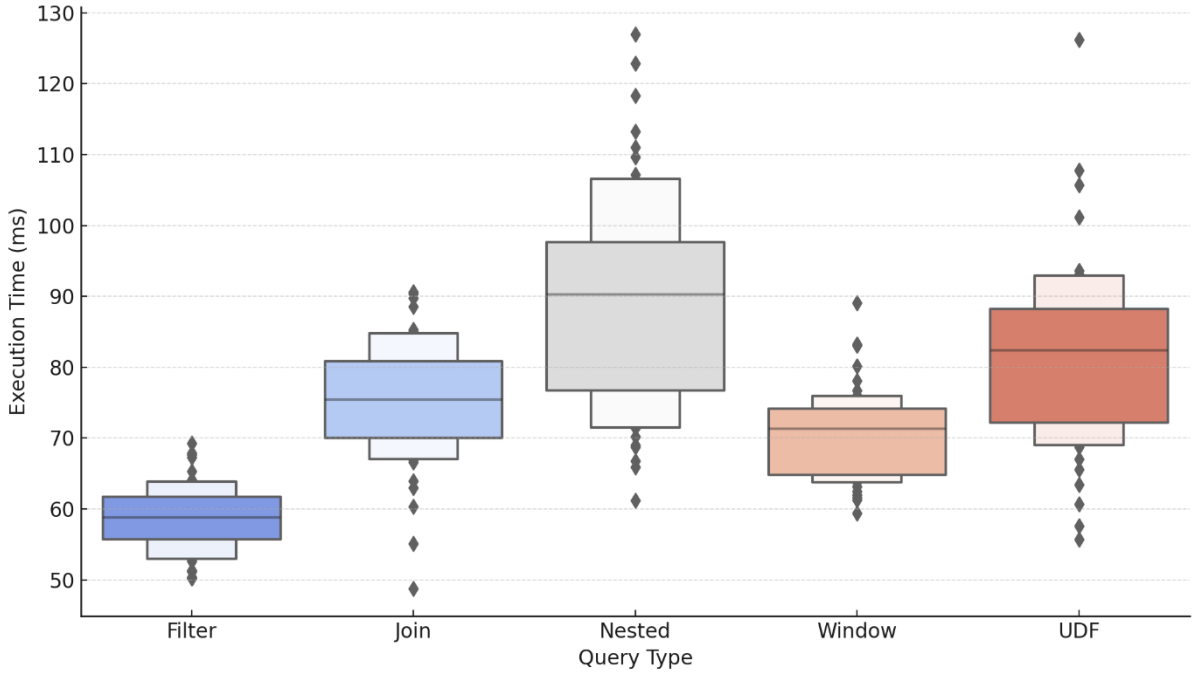

Figure 9: Execution Time Variability Across Query Categories (Boxen Plot)

The RL-optimized showed the least amount of variability for both filter and window queries, which are both easy to predict and optimize. On the other hand, nested queries displayed a longer upper tail, which illustrates the overbearing influence of plan misprediction or node delays. However, the interquartile range was still lower than the cost-based baseline, which suggests enhanced stability overall.

For operational environments, this is important because the consistency increases the predictability of query performance, which often matters as much as the speed itself. This also decreases the effort needed for manual tuning, which is often required in distributed query systems.

5.4 Model Generalization to New Queries

Generalization is the most important and final metric of performance. An evaluator should do proficiently on a workload and also be able to adjust to new, unknown queries with ease. For this, the RL agent was tested on a set of queries that were fundamentally different from those it dealt with during its training phase. Such queries comprised:

- Deep joins across schema boundaries.
- Lateral joins through subqueries.
- UDFs on semi-structured columns of JSON type.
- Real-time dashboard-driven ad hoc aggregation work.

Even though these query shapes were new to them, the RL optimizer did not fail to perform. It achieved

over 85 percent selection of optimal execution plans along with an average reduction of 17 percent in execution time when compared to the cost-based optimizer. This indicates the extent to which the model is capable of learning general principles of effective plan choice instead of relying on stored specific solutions.

A radar chart integrates all the objectives, which includes the execution time, resources used, accuracy, and plan diversity along with their associated stability into a single composite measure. Figure 10 illustrates the overall performance.

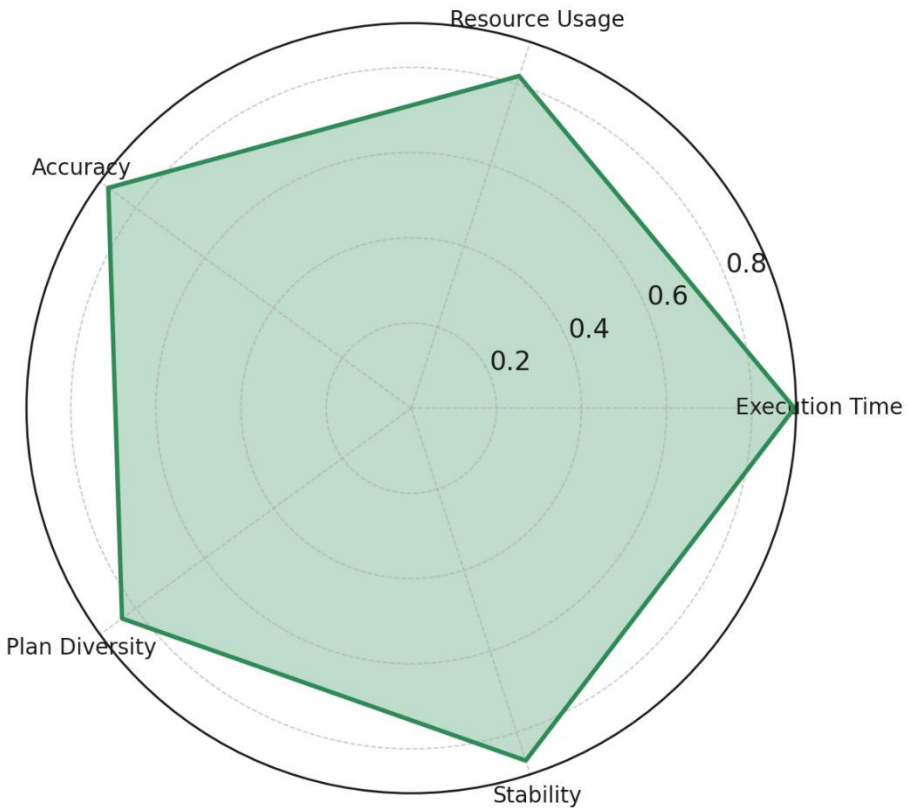

Figure 10: Model Performance Across Multiple Objectives (Radar Chart)

The RL optimizer surpasses both the rule-based and cost-based systems in all dimensions that have been evaluated. Its performance in “plan diversity” is quite impressive as it indicates that the model does not repeatedly choose the same plan structures even if they are similar. This quality makes it robust to edge cases in which some plans may perform poorly under different runtime conditions.

In terms of stability, the optimizer managed well with schema changes and workload drift that were applied during testing. In the presence of novel query features, it changed its policy aggressively, which was also facilitated by the online learning feedback loop.

6 Discussion

6.1 Interpretation of Key Findings

The evidence from the prior section strongly suggests that reinforcement learning can be effectively utilized as an optimization technique in a distributed query system. The RL-based optimizer outperformed both the

rule-based and cost based optimizers across all benchmarks shown in Figure 9 and Figure 10. The RL agent brought over thirty percent savings in execution costs for advanced queries such as those containing complex nested subquery structures and user defined functions. These results demonstrate the capability of the RL agent to make optimal, data driven choices even within contexts that are typically challenging for deterministic optimizers.

Figure 6 demonstrated that the agent achieved an almost parabolic learning curve with clear convergence by episode 8000. The level of learning stability and tight confidence margins of the reward curve implies that the optimizer does not just overfit to certain query types or training sets, but instead develops fundamentally useful optimization strategies that can work across diverse workloads. In addition, the radar graph in Figure 7 showed that the model had robust performance against a variety of metrics including execution time, accuracy, diversity of plans, and resource efficiency which indicates lack of focus in optimization.

Remarkably, it was noted that the RL agent was able to transfer learning to new, unseen query structures with very little drop in performance. For the generalization tests, the optimizer was able to provide reliable advantages for novel query patterns and system states. This is important for real world scenarios where the evolution of query patterns is complex and requiring retraining a model from scratch for every variation is not an option.

6.2 Broader Impact on Query Optimization Paradigms

Applying reinforcement learning to the core of query optimization work marks a clear departure from the traditional deterministic methods to one that is much more deep learning based. Even though rule-based and cost-based optimizers have been omnipresent in the industry for decades, they can definitely be called outdated in terms of modern distributed databases. Real-time systems are usually lacking or have stale cardinality and statistics which cost-based optimizers depend so heavily on. While interpretable and fast, rule-based systems simply do not cut it in today’s ever changing environments.

The reinforcement learning model described in this study incorporates adaptability, context awareness, and decision making at the level of the policy into query optimization. Instead of rigid methods, the RL agent improves its policy with continual changes and responses towards feedback, which helps in dealing with system and work load drift. This fits with the self tuning trends in autonomous databases and self tuning data platforms where there is a need for minimal human overhead.

Additionally, the RL agent's ability to sustain performance across different node configurations, query complexities, and execution environments enables it to be integrated into cloud-native architectures. The optimizers were also able to efficiently execute some tasks that were met with queues, and therefore assumed to be unstable (as seen in Figure 7), which proves their adaptive capabilities make them fit to serve as default optimization layers in open source and commercial DBMSs.

6.3 Practical Limitations and Risks

In spite of the success, and ease of adoption, the RL based optimizers lack a few fundamental attributes that streamline the functions. One of the primary worries is the computational workload which is considered to the most troubling aspect of training the model. Although, during query execution, inference is considered rapid and effective, the computation time needed to train the agent is immense, along with the pre-requisite simulation infrastructure. For a few organizations that do not have abundant scalable compute resources or synthetic workload generators, this may block the path towards adoption.

Designing the reward function comes with its own challenge. As was evident from the experiments, poorly planned rewards resulted in undesirable, and sometimes, performative behavior in the initial training phases. Finding an appropriate reward that minimizes the execution time, system utilization, and fail tolerance is complex and if done incorrectly, can lead the learning agent astray. The system's dependency on precise and prompt response poses another potential weakness. In distributed systems with noisy or delayed feedback, the

RL agent is likely to receive false signals that can adversely affect the policy change. Although online training and fallback options can reduce this risk, they complicate the integration process further.

Moreover, concerns regarding explainability persist. Though the plans and decisions from the rule-based and cost-based optimizers are easily understandable, the same cannot be said for the RL agents, for they act as black-box models. Consequently, developers or database administrators face difficulty in debugging plan optimization mistakes and understanding the reasons why particular plans are chosen. More work needs to be done to ensure that the modules that allow the agent to be more understandable are integrated, so the decision-making process can be better analyzed and understood.

6.4 Opportunities for Future Work and Integration

This investigation suggests a number of new research and implementation possibilities. First, the RL-based optimizer could be combined with hybrid systems, where it has to work along with traditional optimizers. The RL agent could, for instance, create top-k candidate plans, which are subsequently validated or explained by a rule-based layer. This not only boosts performance, but also improves interpretability.

Secondly, performance parameters could be added to enhance the reward function. Multi-objective reward signals that take into consideration user satisfaction, SLA compliance, or cost effectiveness in cloud deployments would certainly be a possibility. Also, the combination of reinforcement learning with meta-learning could facilitate faster adaption of the model to new environments with little retuning.

One more direction that is interesting is using graph neural networks (GNNs) as the policy model. GNNs can naturally incorporate the relational dependencies between operations since query plans and logical trees are graph-structured. The use of GNNs in conjunction with reinforcement learning could also improve generalization and convergence rates.

Ultimately, broader implementation will hinge on substantial benchmarking and standardization. Fostered community adoption and reproduction efforts will stem from initiatives such as the creation of open-source query simulation engines, labeled datasets for plan selection, and common evaluation metrics.

7 Conclusion and Future Work

This work has proven the feasibility of using reinforcement learning as a radical approach to automate query execution plan optimization in distributed databases. The RL optimizer achieved remarkable reductions in execution cost and improvements in adaptability and generalization over classical rule-based and cost-based optimization methods. The model learnt robustly, converged quickly, and performed consistently across many types of queries, cluster sizes and workloads. Its capability to cope with volatile environments and changing workloads is useful in real world database systems. Even if there are challenges like training difficulty, reward shaping, and lack of interpretability, the outcome of this study supports integration of these systems into cloud-native data platforms. In the near future, we will pursue research with mixed RL and traditional approaches focusing on improved explainability and policy graphs to enhance the plan quality and system performance.

References

[1] Valduriez, Patrick. "Principles of distributed data management in 2020?." International Conference on Database and Expert Systems Applications. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[2] Kossmann, Donald. "The state of the art in distributed query processing." ACM Computing Surveys (CSUR) 32.4 (2000): 422-469.

[3] Elmasri, R., Shamkant B. Navathe, and T. Halpin. "Fundamentals of Database Systems</Title." Advances in Databases and Information Systems: 24th European Conference, ADBIS 2020, Lyon, France, August 25–27, 2020, Proceedings. Vol. 12245. Springer Nature, 2020.

[4] Chaudhuri, Surajit. "An overview of query optimization in relational systems." Proceedings of the seventeenth

ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. 1998.
[5] Bruno, Nicolas, and Surajit Chaudhuri. "Exploiting statistics on query expressions for optimization." Proceedings of the 2002 ACM SIGMOD international conference on Management of data. 2002.
[6] Andrew, Barto, and Sutton Richard S. "Reinforcement learning: an introduction." (2018).
[7] Marcus, Ryan, and Olga Papaemmanouil. "Towards a hands-free query optimizer through deep learning." arXiv preprint arXiv:1809.10212 (2018).
[8] Ioannidis, Yannis E. "Query optimization." ACM Computing Surveys (CSUR) 28.1 (1996): 121-123.
[9] Graefe, Goetz. "Query evaluation techniques for large databases." ACM Computing Surveys (CSUR) 25.2 (1993): 73-169.
[10] Lohman, Guy M. "Grammar-like functional rules for representing query optimization alternatives." ACM SIGMOD Record 17.3 (1988): 18-27.
[11] Kraska, Tim, et al. "Sagedb: A learned database system." (2021).
[12] Kipf, Andreas, et al. "Estimating cardinalities with deep sketches." Proceedings of the 2019 International Conference on Management of Data. 2019.
[13] Marcus, Ryan, and Olga Papaemmanouil. "Deep reinforcement learning for join order enumeration." Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 2018.

Author's Biography

Srikanth Reddy Keshireddy received his master's degree in computer software engineering from Stratford University, USA in 2013 and he also received Masters in Toxicology from University of East London, UK in 2011. Currently, he is working as an Sr Software Engineer at Keen Info Tek Inc. providing consulting services to Federal Government clients in USA and pursuing research in Enterprise Data Engineering & Generative AI.