IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, June 2026
Impact Factor: 8.2

SQL Query Optimization Using Artificial Intelligence: A Hybrid Optimization Approach for Relational Database Systems

Subham Shravan Chaurasiya
MCA Student
Vidyavardhini's College of Engineering and Technology, Mumbai, India
University of Mumbai CDOE, Mumbai, India

Abstract: *SQL query optimization is a key function of relational database systems because it directly affects execution time, resource usage, and system performance. Traditional cost-based optimizers select execution plans using statistics, estimated cardinalities, indexes, and cost formulas. However, skewed data, correlated predicates, outdated statistics, and complex joins can lead to poor plan selection. This paper studies how Artificial Intelligence can improve SQL query optimization through a hybrid approach. Instead of replacing the traditional optimizer, AI assists components such as cardinality estimation, cost prediction, index recommendation, join-order selection, and optimizer steering. The study refers to benchmark systems such as JOB, CEB, MSCN, Neo, and Bao, and evaluates optimization using metrics such as execution time, Q-error, rows processed, throughput, and tail latency. The paper concludes that AI is most practical as an assistant layer with monitoring, validation, and fallback mechanisms.*

Keywords: SQL query optimization, artificial intelligence, machine learning, relational databases, cardinality estimation, cost prediction, optimizer steering, index recommendation, execution plans, query latency

I. INTRODUCTION

SQL is a declarative language in which users specify the required result while the database system decides how to retrieve it. This decision is handled by the query optimizer, which selects access methods, join order, indexes, physical operators, and execution strategies. A single SQL query may have several possible execution plans, and choosing a poor plan can significantly increase execution time, memory usage, CPU cost, and I/O cost.

Most relational database systems, including MySQL, SQL Server, PostgreSQL, Oracle, and IBM Db2, use cost-based optimization. A cost-based optimizer estimates the number of rows produced by each operation, compares possible execution plans, and selects the plan with the lowest estimated cost. These decisions depend on table statistics, index metadata, histograms, estimated cardinalities, and cost formulas [1].

A major limitation of traditional optimizers is inaccurate cardinality estimation. If the optimizer underestimates or overestimates rows produced by filters, joins, grouping, or aggregation, it may choose an inefficient join order, index, or physical operator. This problem is common in workloads with skewed data, correlated predicates, stale statistics, missing indexes, and complex joins [1], [4].

Artificial Intelligence can reduce these problems by learning from historical query executions. Machine learning models can improve cardinality estimation, predict query runtime, recommend indexes, identify repeated slow-query patterns, and suggest optimizer hints or safe query rewrites. However, AI should not replace the database optimizer completely. A more practical approach is to use AI as an assistant layer that improves weak optimizer components while preserving correctness and reliability [2], [3], [4].

DOI: 10.48175/IJARSCT-37219

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, June 2026
Impact Factor: 8.2

This paper focuses on the following research question:

How can AI improve SQL query optimization across relational database systems while preserving the reliability of traditional cost-based optimizers?

II. LITERATURE REVIEW

SQL query optimization has traditionally been handled by cost-based optimizers. These optimizers generate candidate execution plans and select the one with the lowest estimated cost based on statistics, indexes, estimated row counts, CPU cost, I/O cost, and memory usage. This model forms the foundation of modern database optimizers [1].

A central challenge in optimization is cardinality estimation because the optimizer must estimate result sizes before executing the query. If these estimates are inaccurate, the optimizer may choose an inefficient join order or unsuitable physical access path. The Join Order Benchmark showed that cardinality-estimation errors are a major cause of poor query plans in realistic join workloads [1].

The Cardinality Estimation Benchmark extended this problem at a larger scale by introducing more than 15,000 queries across IMDb and StackExchange workloads. It showed that better estimation accuracy can improve plan quality, but also that learned models may become brittle when workloads change [4].

Machine learning has been applied mainly in three areas: learned cardinality estimation, learned cost prediction, and optimizer steering. MSCN demonstrated that deep learning can estimate correlated joins more effectively than traditional approaches in many scenarios [3]. Bao showed a practical hybrid approach by learning to guide an existing optimizer using query-specific hint selection instead of replacing the database optimizer [2].

Although systems such as Bao were implemented on a specific database engine, the general idea is relevant to relational databases such as MySQL, SQL Server, PostgreSQL, and Oracle because these systems provide plan inspection, tuning tools, optimizer hints, execution plans, or monitoring features [5], [6], [7], [8]. Overall, the literature supports the view that AI is most useful when applied to prediction-heavy optimizer components while the traditional optimizer continues to ensure SQL correctness and stable execution.

III. PROBLEM STATEMENT AND RESEARCH OBJECTIVES

Traditional SQL query optimizers are effective for many workloads, but they may produce inefficient execution plans when optimizer estimates do not accurately represent real data. Relational databases rely on statistics, index metadata, estimated row counts, and cost calculations to choose access paths, join order, and physical operators. However, skewed data, correlated columns, outdated statistics, missing indexes, and complex joins can cause inaccurate estimates and poor plan selection.

The main problem addressed in this paper is:

Traditional cost-based SQL optimizers can select inefficient execution plans because of inaccurate cardinality estimation, limited cost prediction, poor index selection, and changing workload patterns.

Artificial Intelligence can help reduce this problem by learning from previous query executions, runtime behavior, estimated rows, rows processed, selected indexes, execution plans, and slow-query patterns. However, AI models must be validated carefully because they can fail when workloads change or training data is insufficient.

Research Objectives

- To identify the limitations of traditional SQL query optimization, especially cardinality-estimation errors, poor index selection, and inefficient join ordering.
- To study how AI can improve query optimization through learned cardinality estimation, runtime prediction, index recommendation, and optimizer steering.
- To propose a practical hybrid AI-assisted framework for relational database systems.
- To define evaluation metrics such as execution time, Q-error, rows processed, throughput, P95/P99 latency, and fallback rate.
- To analyze the benefits, risks, and practical limitations of AI-assisted SQL query optimization.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, June 2026
Impact Factor: 8.2

Scope and Research Gap

This paper focuses on relational database workloads involving filtering, joins, aggregation, sorting, indexing, and analytical queries. It does not attempt to replace the database optimizer or modify SQL semantics. The research gap is the need for a practical hybrid model that uses execution-plan analysis, runtime feedback, optimizer hints, index recommendations, and workload monitoring to improve SQL performance while maintaining database reliability [1], [2], [4].

IV. PROPOSED AI-ASSISTED QUERY OPTIMIZATION FRAMEWORK

This paper proposes a hybrid AI-assisted query optimization framework for relational database systems. The goal is not to replace the traditional SQL optimizer, but to improve selected optimization decisions using learned predictions from previous query executions. The database optimizer remains responsible for parsing, rewriting, plan generation, correctness, and execution, while the AI layer assists in prediction-heavy tasks.

4.1 Framework Architecture

Table 1. Components of the Proposed AI-Assisted Query Optimization Framework

Component	Function
Query Feature Extractor	Extracts SQL features such as tables, joins, predicates, grouping, sorting, and filters
Plan Collector	Collects execution-plan details from tools such as EXPLAIN, actual execution plans, profiling tools, or monitoring utilities
Workload History Store	Stores query runtime, estimated rows, actual rows, rows processed, selected indexes, and physical operators
Learned Cardinality Estimator	Predicts row counts and detects inaccurate optimizer estimates
Learned Cost Predictor	Predicts query runtime and resource usage
Optimizer Steering Module	Recommends hints, index changes, join-order guidance, or safe query rewrites
Fallback Controller	Uses the default optimizer when AI confidence is low or no improvement is measured

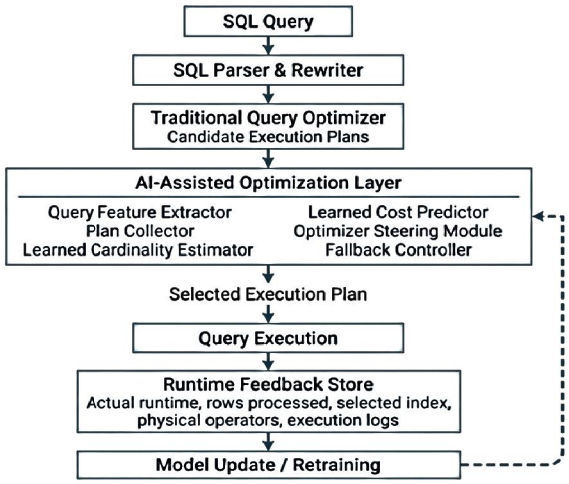

Figure 1. Proposed AI-Assisted Query Optimization Framework for Relational Database Systems

Figure 1 shows the proposed hybrid workflow. The traditional relational database optimizer remains responsible for parsing, rewriting, plan generation, and query execution. The AI-assisted layer analyzes plan features, predicts

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, June 2026
Impact Factor: 8.2

estimation risk, recommends improvements, and stores runtime feedback for future learning. The fallback controller ensures that the default optimizer is used whenever AI confidence is low or no measurable improvement is expected.

4.2 Working Process

When a SQL query is submitted, the database first optimizes it using its built-in optimizer. The AI-assisted layer then collects query and plan features and compares estimated behavior with actual runtime behavior.

The learned cardinality estimator predicts whether row-count estimates are likely to be inaccurate. The learned cost predictor estimates expected runtime and resource usage. If the AI layer finds a better alternative, the optimizer steering module may recommend an index, optimizer hint, join-order change, physical operator preference, or safe query rewrite.

After execution, actual runtime, rows processed, selected indexes, physical operators, and plan behavior are stored in the workload history store. This feedback allows the AI model to improve future predictions.

4.3 Practical SQL Example

Consider the following SQL query:

```
SELECT c.region, SUM(p.amount) AS total_amount
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN payments p ON o.order_id = p.order_id
WHERE c.region = 'West'
GROUP BY c.region;
```

If the optimizer underestimates the number of customers in the West region, it may choose an inefficient index, join order, or join method. The AI-assisted layer can learn from previous executions and recommend a better index, join strategy, physical operator preference, or optimizer hint.

4.4 Input Data and Safety

Data Type	Example
Query features	Joins, predicates, selected columns, grouping, sorting
Plan features	Access method, selected index, join order, estimated rows
Runtime data	Execution time, rows processed, CPU and I/O behavior
Monitoring data	Query logs, profiling data, wait statistics, slow-query records
Cardinality data	Estimated rows compared with actual rows
Workload history	Repeated slow queries and similar query patterns

A practical AI-assisted optimizer must include fallback logic. If the model has low confidence, insufficient training data, or high uncertainty, the system should use the default optimizer plan. All AI recommendations should be validated using execution-plan analysis and runtime testing before being applied in production [2], [4].

V. EXPERIMENTAL SETUP AND EVALUATION METRICS

To make the proposed framework practical and measurable, the study uses a controlled relational database environment with realistic SQL workloads. The experiment compares a default cost-based optimizer with an AI-assisted optimization layer that improves cardinality analysis, runtime prediction, index recommendation, and optimizer steering.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, June 2026
Impact Factor: 8.2

5.1 Experimental Environment

Item	Proposed Setup
Database system	MySQL, SQL Server, PostgreSQL, Oracle, or similar relational DBMS
Dataset	IMDb-style, StackExchange-style, or e-commerce relational dataset
Workload type	Queries with joins, filters, aggregation, sorting, and indexing
Baseline optimizer	Default database optimizer
AI-assisted method	Learned cardinality estimator, runtime predictor, index recommendation, and optimizer steering
Query analysis tools	Execution plans, actual execution plans, EXPLAIN, profiling, or SQL monitoring tools
Monitoring tools	Query logs, slow-query logs, wait statistics, performance views, or profiling tables
Hardware metrics	CPU usage, memory usage, disk I/O, rows processed

5.2 Evaluation Metrics

Metric	Meaning
Execution time	Actual query runtime in milliseconds
Optimization overhead	Time added by AI feature extraction and prediction
Q-error	Ratio between estimated and actual cardinality
Rows processed	Rows scanned, examined, or processed by the database
Index usage accuracy	Whether the selected index improves performance
Throughput	Queries completed per second
P95/P99 latency	95th and 99th percentile runtime
Slow-query reduction	Reduction in queries crossing the slow-query threshold
Fallback rate	Percentage of queries using the default optimizer

5.3 Practical Evaluation Method

For each SQL query, the default optimizer plan is recorded using the database's plan-inspection tool. Actual runtime behavior is collected using execution plans, profiling, monitoring tools, or query logs. AI-assisted recommendations such as index suggestions, optimizer hints, join-order guidance, or safe query rewrites are then applied and tested. The recommendation is accepted only if execution time, rows processed, or tail latency improves without changing the query output.

5.4 Expected Outcomeand Feasibility

The AI-assisted optimizer is expected to perform best on complex or repeated SQL queries involving joins, filters, aggregation, skewed data, and large tables. Simple primary-key lookups may show little improvement because traditional optimizers already handle them efficiently.

The proposed system can be implemented as an external advisory layer rather than as a modification to database internals. It can collect plans, runtime behavior, and workload history using database-specific monitoring tools. This makes the design practical for database administrators, developers, and performance engineers.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, June 2026
Impact Factor: 8.2

VI. RESULTS AND ANALYSIS

6.1 Published Quantitative Evidence

Published research provides strong evidence that AI can improve key optimizer decisions. The Cardinality Estimation Benchmark generated over 15,000 challenging queries across IMDb and StackExchange databases. In one example, the plan chosen using PostgreSQL's own cardinality estimates was almost 7× worse than the plan chosen using true cardinalities. CEB also reported that, in one equal-template split, almost 6,000 test queries showed a large gap between PostgreSQL estimates and learned models, translating into several hours faster total workload runtime [4].

Bao provides further evidence for optimizer steering. It evaluated IMDb, Stack, and Corp workloads with 5,000, 5,000, and 2,000 queries respectively. Bao showed that disabling loop joins improved JOB query 16b by 3×, but degraded query 24b by almost 50×, proving that fixed rules are risky and per-query steering is safer [2].

Bao also reported around 20% cost and performance improvement on a commercial database system. On 113 held-out JOB queries, Bao caused regressions on only 3 queries, all under 3 seconds, while 10 queries improved by more than 20 seconds. Maximum optimization time was 140 ms for PostgreSQL, 165 ms for the commercial database, and 230 ms for Bao [2].

6.2 Published Benchmark Results

Table 2. Published Benchmark Evidence Relevant to AI-Assisted Query Optimization

Source	Published Result	Practical Implication
CEB [4]	Over 15,000 queries across IMDb and StackExchange	Learned estimation can be tested at realistic scale
CEB [4]	PostgreSQL-estimate plan almost 7× worse than true-cardinality plan	Cardinality errors directly damage plan quality
CEB [4]	Almost 6,000 test queries showed several hours faster total runtime	Learned estimation can improve workload performance
Bao [2]	Query 16b improved by 3× when loop joins were disabled	AI steering can correct optimizer mistakes
Bao [2]	Query 24b became almost 50× slower when loop joins were disabled	Fixed heuristics are risky
Bao [2]	Around 20% cost and performance improvement on a commercial system	Hybrid optimization can help strong optimizers
Bao [2]	Only 3 regressions on 113 JOB queries; 10 queries improved over 20 seconds	Strong upside with limited downside
Bao [2]	Max optimization time: 230 ms for Bao	AI overhead is measurable and must be controlled

6.3 Interpretation for Relational Database Systems

These results should be interpreted as external evidence supporting the proposed hybrid design, not as direct results for every database system. The same ideas can be adapted to MySQL, SQL Server, PostgreSQL, Oracle, and similar relational databases by using execution-plan tools, runtime monitoring, optimizer hints, and profiling mechanisms.

The evidence supports a balanced conclusion: AI is most useful when it improves prediction-heavy tasks such as row estimation, cost prediction, index recommendation, and hint selection. However, it must be applied selectively to expensive or repeated queries, while retaining the default optimizer as a fallback path [2], [5], [6], [7], [8].

VII. BENEFITS, CHALLENGES, AND PRACTICAL LIMITATIONS

AI-assisted SQL query optimization offers important benefits. It can improve cardinality estimation by learning relationships between columns, predicates, and joins that traditional histograms may miss. It can also improve runtime

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, June 2026
Impact Factor: 8.2

prediction by learning from actual query execution history. This makes optimization more adaptive to workload behavior, hardware conditions, and repeated query patterns.

Another benefit is reduced manual tuning. Database administrators often spend time analyzing slow queries, updating statistics, rewriting queries, adding indexes, and applying optimizer hints. AI-assisted systems can reduce some of this effort by recommending better plans, indexes, or query rewrites.

However, AI-based optimization also has challenges. It requires high-quality training data, and models may perform poorly when workloads change. Model drift can occur as data grows, schemas evolve, and indexes are added. AI also adds optimization overhead, which may not be justified for short or simple queries.

Explainability is another limitation. Traditional optimizers provide execution plans that engineers can inspect, while AI decisions may be harder to explain. Deployment also requires monitoring, fallback logic, retraining, validation, and version-aware testing.

Although the proposed framework is generic, each relational database system has its own optimizer behavior, statistics model, plan format, hint syntax, indexing strategy, and monitoring tools. Therefore, AI recommendations should be tested using the target database's actual execution-plan tools and runtime measurements.

A practical rule is:

AI should be used only when it provides measurable performance improvement without sacrificing reliability, explainability, or SQL correctness.

VIII. FUTURE WORK

Future work can focus on developing self-learning database systems that continuously learn from query execution history and automatically improve cardinality estimation, cost prediction, indexing decisions, and plan selection. Another direction is adaptive workload-aware optimization, where AI is applied only to expensive or repeated queries where the expected benefit is greater than model overhead.

Large Language Models may also support SQL optimization by explaining execution plans, identifying inefficient SQL patterns, and suggesting indexes, hints, or query rewrites. However, these suggestions should always be validated using actual runtime measurements before production use.

Future systems should also handle workload drift across different database engines. As data volume, indexes, schemas, and query patterns change, the AI model should be retrained and compared against the default optimizer. Safe fallback mechanisms will remain necessary for production reliability.

IX. CONCLUSION

SQL query optimization is essential for improving relational database performance because it determines how efficiently queries are executed. Traditional cost-based optimizers are mature and reliable, but poor cardinality estimates, skewed data, missing indexes, complex joins, and changing workloads can still lead to inefficient plans.

This paper studied how Artificial Intelligence can support SQL query optimization through a hybrid approach. Rather than replacing the traditional optimizer, AI can assist it through learned cardinality estimation, runtime prediction, index recommendation, join-order analysis, and optimizer steering. Published research such as CEB and Bao shows that learned optimization techniques can improve plan quality, reduce workload latency, and limit severe regressions when applied carefully [2], [4].

The study concludes that AI-assisted SQL query optimization is most useful for complex and repeated queries involving joins, filters, aggregation, skewed data, and large tables. However, AI models must be monitored, validated, and supported by fallback mechanisms. The most practical direction is a hybrid system where AI improves selected optimization decisions while the traditional database optimizer continues to ensure correctness, stability, and reliable execution.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 10, June 2026
Impact Factor: 8.2

REFERENCES

[1] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How Good Are Query Optimizers, Really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.

[2] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making Learned Query Optimization Practical," *Proceedings of the 2021 International Conference on Management of Data*, pp. 1275–1288, 2021.

[3] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper, "Learned Cardinalities: Estimating Correlated Joins with Deep Learning," *CIDR*, 2019.

[4] P. Negi, R. Marcus, and A. Kipf, "Cardinality Estimation Benchmark," Learned Systems Group, MIT, 2021.

[5] Oracle, "MySQL Reference Manual: Optimizing Queries with EXPLAIN and Optimizer Hints," Oracle MySQL Documentation, accessed May 2026.

[6] Microsoft, "SQL Server Documentation: Execution Plans and Cardinality Estimation," Microsoft Learn, accessed May 2026.

[7] PostgreSQL Global Development Group, "PostgreSQL Documentation: Using EXPLAIN and Query Planning," PostgreSQL Documentation, accessed May 2026.

[8] Oracle, "Oracle Database SQL Tuning Guide: Execution Plans, Optimizer Statistics, and SQL Tuning," Oracle Documentation, accessed May 2026.

[9] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Making Learned Query Optimization Practical: A SIGMOD Record Perspective," *SIGMOD Record*, 2022.