International Journal of Computer Techniques–IJCT Volume 13 Issue 1, Jan-Feb - 2026

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

RESEARCH ARTICLE OPEN ACCESS

AI-Assisted Query Optimization in Relational Databases: A Comparative and Experimental Review

Mohamed Chetouani[1]

Nanjing University of Information Science and Technology

Email: medchet2015@gmail.com

****

Abstract:

This paper presents a comparative and experimental review of AI-assisted query optimization techniques in relational databases. The study examines traditional optimization methods, explores AI-based approaches including machine learning and reinforcement learning, and evaluates their effectiveness through experimental results.

Keywords **— Query Optimization, Relational Databases, Artificial Intelligence, Machine Learning, Reinforcement Learning, Genetic Algorithms**

****

I. INTRODUCTION

Query optimization is a fundamental aspect of relational database management systems, ensuring that data retrieval is carried out efficiently while minimizing resource usage. Traditional query optimization methods rely on predefined rules, cost-based analyses, and heuristic strategies to select execution plans that reduce response time. Although these approaches have proven effective in many cases, they often face challenges when dealing with dynamic workloads, complex queries, or large-scale datasets.

The rapid growth of data and the increasing complexity of applications have sparked interest in applying artificial intelligence (AI) to enhance query optimization. AI-based techniques, including machine learning, reinforcement learning, and evolutionary algorithms, can learn from past query executions, adapt to changing conditions, and explore optimization strategies beyond conventional heuristics. By incorporating AI into query optimization, database systems have the potential to improve performance, lower latency, and manage diverse workloads more effectively.

This paper examines the role of AI in query optimization, providing a comparative review of traditional and AI-assisted methods. It explores different AI techniques, evaluates their applicability to relational databases, and assesses the performance gains they can offer. In addition, experimental results from recent studies are analyzed to determine the practical effectiveness of these approaches. The objective is to provide a comprehensive understanding of how AI can advance query optimization and to highlight the challenges, limitations, and future directions in this evolving field.

II. FUNDAMENTALS OF QUERY OPTIMIZATION

A. Definition and Purpose of Query Optimization

Query optimization is a fundamental process in relational databases that aims to enhance the efficiency of query execution. It involves

selecting the most effective execution plan from a range of possible strategies to retrieve or manipulate data [5]. The primary role of a query optimizer is to translate a high-level declarative query, such as SQL, into an execution plan that minimizes resource usage and execution time [4].

Query optimization is particularly crucial in relational databases because poorly optimized queries can severely degrade performance, especially as data volume and complexity grow [2]. Efficient optimization directly improves system performance by reducing execution time, lowering CPU and memory usage, and minimizing input/output operations. This not only leads to cost savings but also enhances throughput and responsiveness for users.

The optimization process generally involves analyzing different execution strategies, estimating their costs, and selecting the most efficient plan [7]. For example, the optimizer may decide between a sequential table scan and an index scan depending on table size and query selectivity. It may similarly choose among join algorithms such as nested loop join, hash join, or sort-merge join, based on data distribution and available resources [18].

By systematically evaluating potential execution plans and their associated costs, query optimization ensures that database systems maintain high performance even under heavy workloads and complex query conditions [5]. Thus, effective optimization serves as a cornerstone for the performance and scalability of relational databases.

B. Query Processing Steps

Query processing in relational databases refers to the sequence of operations that transforms a high-level SQL query into an optimized execution plan. This process ensures that queries are executed efficiently, minimizing response time, resource consumption, and overall cost [4] The main steps in query processing include parsing, translation, logical plan generation, physical plan generation, cost estimation, plan selection, and execution.

Parsing. The first step in query processing is parsing. During this phase, the SQL query is converted into a structured internal representation, typically a parse tree, which captures the syntactic structure of the query. The system also performs syntax checking and validates semantic correctness, ensuring that referenced tables and columns exist. This step is essential, as it provides the foundation for all subsequent optimization stages [4].

Translation to Relational Algebra. After parsing, the query is translated into a relational algebra expression. This logical representation separates query operations from their physical implementation, enabling algebraic transformations that maintain equivalence while potentially reducing intermediate results or improving join order [4], [10].

Logical Plan Generation. The logical plan represents the sequence of operations required to produce the query result. It is typically represented as a logical operator tree, including operations such as selection, projection, and joins. In this phase, the optimizer may apply transformations such as pushing selections closer to the data source or reordering joins to reduce intermediate costs [4], [10]. As shown in Fig. 1, the logical plan tree illustrates how selection and join operations are structured in the query execution process.

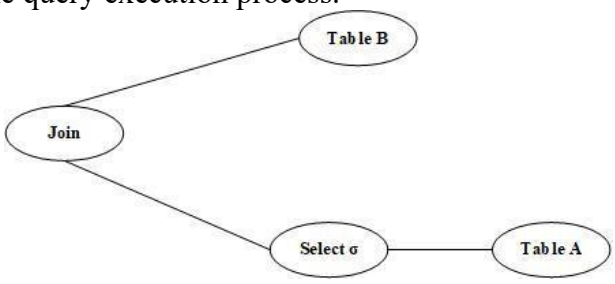

Fig. 1 Logical plan tree illustrating selection and join operations.

Physical Plan Generation. Once a logical plan is constructed, the database generates a physical plan by selecting specific algorithms to implement each operation. Examples include nested-loop joins, hash joins, and sort-merge joins. This phase also considers factors such as index availability, sorting methods, and system resources to identify the most efficient execution strategy [10], [9]. As shown in Fig. 2, the physical plan tree illustrates the chosen execution strategy with specific algorithms.

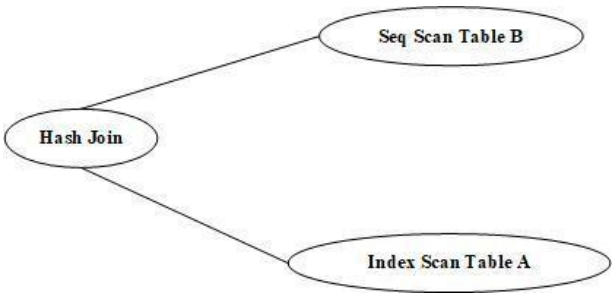

Fig. 2 Physical plan tree showing the chosen execution strategy with specific algorithms.

Cost Estimation and Plan Selection. For each candidate physical plan, the optimizer estimates associated costs, including CPU usage, I/O operations, and memory consumption. The plan with the lowest estimated cost is chosen for execution. Cost estimation can be performed incrementally or guided by heuristics to reduce the search space. Accurate cost estimation and careful plan selection are critical for improving query performance [10], [9].

Query Execution. The final step involves executing the selected physical plan. The query engine performs each operation in the plan, potentially employing caching, buffering, or pipelining techniques to enhance performance. Efficient execution ensures correct results while minimizing resource usage and response time [9].

By understanding these steps, database designers and administrators gain insight into how relational systems optimize queries and why traditional query optimizers are essential for performance. These steps also form a foundation for exploring advanced or AI-assisted optimization techniques in subsequent sections.

C. Cost-Based Optimization

1) Definition and Purpose:

Cost-Based Optimization (CBO) is a technique used in relational database management systems (RDBMSs) to select the most efficient query execution plan by estimating the "cost" of alternative plans. The cost typically includes factors such as CPU usage, disk I/O, and memory consumption, which are critical resources for query execution [4], [10], [9].

In CBO, the database optimizer examines multiple logically equivalent ways of executing a query, often represented as operator trees. For each plan, it estimates the required resources to determine the overall cost. This process allows the optimizer to choose the plan that minimizes execution time and system resource usage, which is especially important for complex queries involving multiple joins, projections, and selections [4], [10] .

CBO is widely adopted in traditional RDBMSs, such as PostgreSQL, Oracle, and SQL Server, because it ensures efficient query execution and can adapt to variations in data distribution, index availability, and system architecture [9], [1]. Without cost-based optimization, query execution would rely on heuristic rules alone, often resulting in suboptimal plans and poor performance.

2) Components of Cost Estimation:

Cost-Based Optimization (CBO) depends on accurately estimating the costs of different query execution plans. The total cost of a plan combines several factors, including CPU usage, I/O operations, memory consumption, and, in distributed or parallel systems, communication costs. The optimizer selects the plan with the lowest total cost.

- **CPU Cost:**

CPU cost represents the time required to process tuples for each operator in a query plan. It can be calculated as:

$$\text{CPU Cost} = \sum (\text{Tuples}_i \times \text{Cost per Tuple}_i)$$

This calculation includes operations such as selection, projection, joins, and aggregations. Accurate estimation of CPU cost is particularly important for complex queries involving multiple joins [4], [7] .

- **I/O Cost:**

I/O cost accounts for reading and writing data to and from disk, which is often the most expensive operation in a database system. It can be estimated by summing the number of blocks read and written across all operators:

$$\text{I/O Cost} = \sum (\text{Read Blocks}_i + \text{Write Blocks})$$

I/O cost depends on factors such as physical data organization, indexing, and buffer utilization. Applying early selection and projection can reduce intermediate results, thereby lowering I/O cost [4], [10] .

- **Memory Cost:**

Memory cost considers the buffer or memory used during query execution. For operators like hash joins or sort-merge joins, it can be expressed as:

$$\text{Memory Cost} = \text{Memory per Operator} \times \text{Number of Operators}$$

Accurately estimating memory usage helps prevent excessive disk spilling and additional I/O overhead [7].

- **Communication Cost (for Parallel/Distributed Systems)**

In distributed or parallel systems, communication cost arises when data is transferred between processors. It can be modeled as:

$$t_{\text{send}} = t_{\text{receive}} = \alpha + B \cdot \beta$$

where:

- α = constant startup overhead for sending a message
- B = size of the message in bytes
- β = per-byte transmission cost

Partitioning strategies, such as hash-based joins, aim to minimize communication by sending only relevant tuples to processors [7].

- **Total Cost**

The total cost of a query execution plan is the sum of CPU, I/O, memory, and communication costs:

$$\text{Total Cost} = \text{CPU} + \text{I/O} + \text{Memory} + \text{Communication}$$

This metric enables the optimizer to compare alternative plans and select the most efficient one. Accurate cost estimation relies on statistical summaries of the data, including table sizes, distinct values, and histograms [4], [7], [10] .

3) Cardinality Estimation:

Cardinality estimation is a critical step in query optimization, as it involves predicting the number of tuples produced by relational operations such as selection, projection, and join. Accurate cardinality estimates enable query optimizers to choose efficient execution plans and minimize resource usage [10].

- **Selection**

For a selection operation, where a predicate is applied to a relation , the estimated number of resulting tuples is given by:

$$|\sigma_P(R)| = |R| \cdot f_P$$

where | | denotes the number of tuples in relation , and is the selectivity factor of predicate , representing the fraction of tuples that satisfy the predicate [10].

- **Projection**

For projection operations, denoted ( ) , which retain only a subset of attributes from relation , the estimated cardinality depends on the number of distinct tuples in the projected attributes:

| ( )| = number of distinct tuples in [ ]

This estimation often relies on histograms or statistics of attribute distributions maintained within the database [10].

- **Join**

For join operations, where two relations and are combined based on equality of attributes . = . , the cardinality can be estimated as:

$$|R \bowtie S| = \frac{|R| \cdot |S|}{\max(\ ,\ ),\ (\ ,\ ))}$$

Here, (,) and (,) represent the number of distinct values in attributes and of and , respectively [10].

- **Multiple Predicates**

When multiple predicates $P_1, P_2, \ldots, P_n$ are applied under the assumption of independence, the cardinality estimation generalizes to:

$$|\sigma_{P_1 \wedge P_2 \wedge \ldots \wedge P_n}(R)| = |R| \cdot f_{P_1} \cdot f_{P_2} \cdots f_{P_n}$$

In practice, this estimation may be inaccurate when predicates are correlated, which remains a well-known challenge in query optimization [10].

Cardinality estimation forms the backbone of cost-based optimization in relational databases. By providing approximate sizes of intermediate results, it guides the optimizer in determining the most efficient order of operations, thereby significantly improving query performance [10].

4) Selectivity

Selectivity is defined as the fraction of rows in a relation that satisfy a given predicate. It measures how restrictive a selection condition is on a dataset [4], [10], [19] .

Formally, for a relation and predicate , selectivity is expressed as:

$$= \frac{|\ (\ )|}{|\ |}$$

where | | denotes the total number of tuples in , and | ( )| represents the number of tuples that satisfy .

Selectivity plays a critical role in estimating the sizes of intermediate results during query execution. Smaller selectivity values, corresponding to more restrictive predicates, reduce the number of tuples passed to subsequent operations. This directly affects CPU and I/O costs [4], [10] .

Example:

If a selection on a relation containing 1000 tuples produces 100 tuples, the selectivity is:

$$= \frac{100}{1000} = 0.1$$

This smaller intermediate result reduces the cost of downstream operations, such as joins, because fewer tuples need to be processed [19] .

5) Cost Function Equation

The cost function is central to cost-based query optimization, offering a quantitative framework for comparing alternative execution plans. It aggregates resource consumption into a single metric, enabling the optimizer to identify the most efficient plan.

At its simplest, the total cost of a query plan can be expressed as:

$$\text{Total Cost} = \text{CPU\_Cost} + \text{I/O\_Cost} + \text{Memory\_Cost}$$

This abstraction highlights the main categories of resource usage, but in practice, each component must be estimated carefully with system-specific parameters. For example, CPU cost is typically measured by the number of tuples processed per operator, I/O cost corresponds to the number of disk or page accesses, and memory cost reflects buffer usage during execution [7].

More advanced optimizers extend this model to incorporate additional factors, such as communication cost in distributed or parallel databases, or buffer utilization to account for cache hit ratios, index scan locality, and physical data layout [4]. These extensions provide more realistic predictions of system performance.

Importantly, cost functions are not uniform across all database management systems. Some systems assign greater weight to I/O cost due to disk bottlenecks, while others emphasize CPU or memory usage depending on hardware characteristics and workload profiles [10]. This system-dependent weighting ensures that the optimizer adapts cost models to reflect actual resource constraints.

In summary, the cost function equation establishes the foundation for evaluating execution strategies. While the simplified formula captures the essential components, effective optimization requires extending it with system- and workload-specific parameters to

ensure accurate and robust performance predictions.

6) Plan Selection

Plan selection is a critical phase of query optimization, where the optimizer evaluates multiple candidate execution plans and selects the one expected to run most efficiently. Each plan is assigned a cost estimate that considers CPU usage, I/O operations, memory consumption, and, in distributed systems, network latency. The optimizer compares these alternatives and chooses the plan with the lowest predicted cost.

A foundational example of plan selection is the System R optimizer for Select–Project–Join (SPJ) queries [4]. Its search space consists of operator trees that represent logically equivalent join sequences, enabled by the associative and commutative properties of joins. Scans may use sequential or index access, while joins can be executed using nested-loop or sort-merge algorithms. The optimizer applies a cost model that incorporates intermediate result sizes, predicate selectivity, and operator costs, enabling it to evaluate plans in a bottom-up manner.

To manage the large search space, System R introduced dynamic programming and interesting orders. Dynamic programming ensures only the best plans for subqueries are retained when constructing larger plans, reducing the search from factorial to polynomial complexity. Interesting orders preserve beneficial output properties (such as sorted tuples), preventing premature pruning that might eliminate globally optimal plans [4]. Modern optimizers generalize these ideas by considering physical properties of plans, which enables aggressive pruning without sacrificing performance.

Beyond classical systems, contemporary optimizers rely on heuristics and extended cost models to further reduce complexity [9]. For example, heuristics may restrict the number of join permutations, prioritize selective indexes, or treat short and long queries differently. PostgreSQL, for instance, uses histograms to estimate predicate selectivity and chooses between full table scans, index scans, or index-only scans based on statistics and query characteristics. It also applies rule-based transformations, such as subquery elimination or join reordering, to reduce intermediate result sizes and improve overall efficiency.

In distributed SQL databases, adaptive optimization extends plan selection by incorporating runtime feedback [24] . Systems such as CockroachDB and Google Spanner monitor cardinality estimates, workload distribution, and node performance during execution. If runtime observations deviate from optimizer predictions, plans may be dynamically re-optimized or query fragments redistributed across nodes to balance load. These adaptive strategies help maintain efficiency in the presence of skewed data distributions, heterogeneous hardware, or changing workloads.

In summary, effective plan selection balances accurate cost estimation, heuristic pruning, and, when possible, adaptive feedback. This balance enables modern database systems to avoid exhaustive plan enumeration while still delivering efficient query execution, minimizing both resource consumption and response time.

D. Query Optimization: Heuristics, Indexing, Joins, and Plan Selection

Query optimization is a fundamental component of modern relational database management systems (RDBMS). Its goal is to execute SQL queries efficiently while minimizing resource usage. The optimization process combines rule-based heuristics, indexing strategies, join algorithms, and plan evaluation techniques to select the most efficient execution plan from multiple alternatives.

1) Rule-Based Optimization

Rule-based optimization (RBO) applies predefined heuristics to transform SQL queries

into more efficient forms without relying on detailed cost estimates. Common rules include predicate pushdown, which moves selection conditions closer to the data source, join reordering to reduce intermediate result sizes, and simplifying expressions where applicable [4].

For example, consider a query joining `Orders` and `Customers` with a selection condition on `Orders.OrderDate`. RBO may apply the date filter before performing the join, reducing the number of tuples involved in the join operation. Similarly, when multiple joins exist, heuristics may prioritize smaller tables or more selective predicates first to minimize intermediate result sizes.

Advantages:

- Simple to implement.
- Predictable transformations based on fixed rules.

Limitations:

- Ignores actual data distribution and index availability.
- May produce suboptimal plans for large or skewed datasets [4].

1) Indexing Techniques

Indexes are essential for improving query execution performance, especially for selection, join, and aggregation operations. Common indexing structures include:

- **B-Tree Indexes:** Efficient for both range and point queries; maintain data in a balanced tree structure.
- **Hash Indexes:** Optimized for exact-match point queries; not suitable for range scans.
- **Composite Indexes:** Index multiple columns together to accelerate multi-attribute searches.

Indexes enable the optimizer to locate relevant tuples quickly, reduce full table scans, and facilitate joins. For example, filtering `Customers` by `City` and `State` can use a composite index on (`City, State`). Similarly, indexed join columns reduce the cost of nested-loop or merge joins [9].

Example Usage:

- Point query: `SELECT * FROM Customers WHERE CustomerID = 102;` → Hash index.
- Range query: `SELECT * FROM Orders WHERE OrderDate BETWEEN '2024-01-01' AND '2024-01-31';` → B-Tree index.
- Join optimization: Using an index on `Orders.CustomerID` when joining with `Customers.CustomerID`.

2) Join Algorithms

Joins are typically the most resource-intensive operations. The optimizer selects among several join algorithms depending on data sizes, indexing, and memory availability:

- **Nested-Loop Join (NLJ):** Iterates over each tuple of one table and searches for matches in the other. Efficient for small tables or when an index exists on the join column.
- **Merge Join (Sort-Merge Join):** Requires both inputs to be sorted on the join key; merges tuples efficiently in a single pass. Suitable for large, pre-sorted datasets.
- **Hash Join:** Builds a hash table for one input and probes it with tuples from the other. Highly efficient for equi-joins on large, unsorted tables [4].

Cost Considerations:

- NLJ: $O(m \times n)$ for tables of size m and n, reduced to $O(m \log n)$ with an index.
- Merge join: $O(m \log m + n \log n)$ if inputs are not pre-sorted.
- Hash join: $O(m + n)$ with sufficient memory.

Example Scenario: A query joining a small `Departments` table with a large `Employees` table on `DeptID` may favor a nested-loop join with index probing, whereas joining two large unsorted tables may favor a hash join.

3) Query Plan Evaluation and Selection

After generating candidate plans via rule-based transformations and indexing heuristics, the optimizer evaluates them to select the plan with the lowest estimated cost. Key factors include:

- Estimated I/O and CPU costs based on table statistics and available indexes.
- Data distribution, including skewed data or selective predicates.
- Interesting orders, where partially sorted outputs can benefit subsequent operations such as `GROUP BY` or `ORDER BY` [4].

Example: Consider two plans for joining `Orders` and `Customers`:

- Plan A: Nested-loop join using a customer index with selection pushed down; estimated cost = 1500 units.
- Plan B: Hash join without indexes; estimated cost = 3000 units.

The optimizer chooses Plan A, minimizing I/O and CPU usage while leveraging indexes and selective predicates. Advanced systems may employ dynamic programming to exhaustively explore join orders [10].

E. Limitations of Traditional Optimization

Traditional query optimization methods, including both rule-based and cost-based approaches, depend heavily on the accuracy of catalog statistics such as histograms, selectivity factors, and cardinality estimates. When these statistics become outdated or inaccurate, the optimizer may generate suboptimal execution plans, for instance, by choosing inefficient join orders or inappropriate index usage. This issue becomes particularly significant in dynamic or high-throughput environments such as streaming and OLTP systems, where data updates occur frequently and statistics cannot be refreshed continuously [4], [10] . Consequently, even well-designed cost models can misrepresent the actual data characteristics, leading to degraded query performance.

As query complexity and dataset size increase, the limitations of traditional optimizers become even more evident. Queries involving multiple joins, nested subqueries, or correlated predicates dramatically expand the search space of possible execution plans, often growing exponentially. To manage this, optimizers employ heuristics or pruning strategies that reduce computation time but risk overlooking globally optimal plans. Moreover, in distributed or partitioned database environments, traditional cost models fail to accurately capture network latency, data skew, and parallelism overheads, which are critical to determining true query cost [9], [24]. These challenges collectively limit the scalability and reliability of conventional optimization techniques.

Another major constraint lies in the static nature of traditional optimization, which assumes stable workloads and data distributions. In practice, workloads evolve, and data characteristics shift over time, causing static query plans to perform poorly under new conditions. Although adaptive query optimization techniques attempt to address this by adjusting plans during execution, they still depend on the same cost models and precomputed statistics that underlie conventional optimizers [1], [24]. To overcome these challenges, researchers have increasingly explored AI-assisted optimization, leveraging machine learning and deep learning to learn cost patterns directly from observed query workloads. These AI-based approaches can adapt dynamically to data and workload changes, learn from execution feedback, and generalize better to unseen queries than traditional rule or cost-based systems [17], [22], [14].

III. AI Techniques in Query Optimization

A. Overview and Motivation

Traditional query optimization techniques, including both rule-based and cost-based approaches, face increasing limitations in modern database environments. Their dependence on predefined heuristics and manually designed cost models restricts

adaptability, especially when data distributions shift or when queries contain complex interdependencies among predicates and attributes. Furthermore, the effectiveness of these optimizers depends heavily on the accuracy of statistical metadata such as histograms and selectivity estimates, which often become outdated or incomplete in dynamic systems [24] . Consequently, traditional optimizers may misestimate cardinalities, select inefficient join orders, or fail to utilize indexes effectively, resulting in significant performance degradation.

To address these challenges, a new paradigm of AI-driven query optimization has emerged. This approach incorporates learning-based models into the core of the optimization process. Instead of relying solely on handcrafted formulas, AI-based systems learn directly from historical query workloads, execution feedback, and runtime statistics to build predictive models that generalize across different contexts [17], [22]. These models can estimate query selectivity, execution cost, or even the optimal join order based on data-driven features, allowing the optimizer to adapt continuously as data and workloads evolve. Deep learning techniques have shown strong potential in capturing complex relationships between attributes and predicates that traditional models are unable to represent effectively [22].

AI-assisted optimization can enhance or replace specific stages of the optimization pipeline. Supervised learning methods are applied to predict cardinalities and query costs from past executions. Reinforcement learning approaches enable optimizers to improve planning strategies through iterative trial and feedback. Evolutionary and genetic algorithms perform population-based searches to efficiently explore large plan spaces and identify high-quality solutions [24], [22]. Collectively, these approaches represent a shift toward self-learning and self-adaptive query optimizers that surpass traditional, statistics-dependent systems in handling complex or rapidly changing database environments [17].

B. Machine Learning-Based Optimization

1) Cost and Selectivity Estimation via ML

Machine learning has increasingly been applied to query optimization, particularly for cardinality and cost estimation, which are crucial for selecting efficient execution plans. Traditional empirical estimators often fail to capture correlations between multiple columns, leading to inaccurate predictions. Early ML-based approaches, such as MSCNN, used convolutional neural networks to model multi-set representations of query joins, achieving improved cardinality estimation compared with histogram-based methods. However, MSCNN is limited to predicting cardinality and does not directly estimate query costs.

The End-to-End framework extends this concept by employing a tree-structured model capable of predicting both cost and cardinality simultaneously. It leverages advanced feature extraction from queries and physical operations, including numeric and string attributes. Embedding techniques, such as hash-bitmap or rule-based embeddings, are used to handle sparse string values, improving generalization to previously unseen queries. Multitask learning further enhances the model's ability to manage complex predicates and multi-join queries [22].

In contrast, Neo addresses query optimization from a plan-centric perspective. It utilizes a deep neural network called a value network to approximate the best achievable latency for partial execution plans. Neo incorporates tree convolution layers to process execution plan trees, capturing parent-child relationships and identifying patterns indicative of high- or low-quality plans. The predicted costs guide a best-first search to construct complete query plans, effectively combining supervised learning with search-based plan selection [14].

Experimental results demonstrate the benefits of these ML-based techniques. MSCNN improves cardinality estimation compared with traditional methods, End-to-End significantly reduces both cost and cardinality errors across numeric and string-heavy workloads, and Neo

generates query plans that are often faster than those produced by commercial and open-source optimizers, while generalizing well to new queries and datasets. Together, these works illustrate a clear progression from single-task cardinality estimation to fully integrated, cost-aware, plan-guided machine learning for query optimization.

C. Reinforcement Learning (RL) for Query Optimization

Reinforcement Learning (RL) provides a dynamic framework for viewing query optimization as a sequential decision-making process rather than a static cost estimation task. Traditional optimizers depend on fixed heuristics and manually designed cost models, which often perform poorly when workloads or data distributions change. In contrast, RL models the optimizer as an agent that interacts with the database environment, learns from experience, and continuously improves its policy to minimize query execution cost.

1) Core Idea

The foundation of RL-based optimization derives from Bellman's Principle of Optimality, which underlies dynamic programming and Markov Decision Processes (MDPs) [11]. In this context, query optimization can be formulated as an MDP defined by the tuple $\langle S, A, P(s,a), R(s,a), s_0 \rangle$, where:

- S represents the set of query states, such as partially constructed execution plans.
- A represents the available actions, including choosing the next join or physical operator.
- $P(s,a)$ defines the transition to a new state after taking an action.
- $R(s,a)$ defines the reward, typically the negative of the estimated plan cost.

The objective is to learn a policy that maps states to actions to maximize the expected cumulative reward. In practical terms, this corresponds to minimizing total query execution cost. The RL agent must balance short-term actions, such as choosing an immediate join, with their long-term effects on overall performance.

1) RL Applications

RL has been explored across multiple aspects of query optimization:

- **Join Order Optimization**

Traditional dynamic programming explores all possible join orders, which grows factorially with the number of relations. RL reformulates this as learning a Q-function over query graphs, where each join represents an action that transitions the state to a new partial plan [11]. Unlike greedy heuristics that focus only on immediate costs, RL methods capture long-term dependencies and identify globally better join orders through experience.

- **Plan Selection and Re-optimization**

Adaptive database systems such as CockroachDB and Google Spanner employ feedback-based re-optimization when actual performance diverges from estimated costs [24] . This feedback loop reflects an RL-like learning process in which the system refines its internal model of selectivity and cost through repeated interactions.

- **Index Tuning and Caching Decisions**

RL agents can also manage physical design tasks such as index and cache selection. Treating database configurations as states and tuning operations as actions allows the agent to autonomously learn efficient configurations that minimize resource usage and query latency.

2) Algorithms and Architectures

Various RL algorithms have been adopted for query optimization. Early studies applied Q-learning and Policy Gradient methods, while recent research employs Deep Q-Networks (DQN) and Actor-Critic architectures capable of handling large and continuous state spaces.

A notable example, [14], introduces a plan-structured neural network that mirrors the hierarchical organization of query plans. Each relational operator corresponds to a neural unit, and these units form a tree that encodes the operator dependencies within the plan.

Information flows upward from child operators to parents, and shared weights across identical operator types enable generalization across different queries. This design allows the model to learn how individual operator behaviors influence overall latency, which aligns well with RL training where plan performance serves as the reward signal.

Integrating such architectures with RL enables end-to-end learning, where the agent observes the query structure, predicts performance outcomes, and refines its policy based on observed rewards. However, challenges remain due to delayed feedback, high-dimensional state spaces, and expensive environment interactions. Hybrid approaches, which combine analytical cost models with learned RL policies, offer a practical balance between stability and adaptability, ensuring robust optimization under diverse workloads.

D. Evolutionary and Genetic Algorithms

1) Principle

Evolutionary and genetic algorithms model query optimization as a population-based search problem, where each individual represents a complete query execution plan, including join ordering, operator selection, and access paths. This perspective emerged from early research recognizing that exhaustive enumeration of query plans becomes infeasible as query complexity increases, particularly for multi-join queries and large optimization search spaces [10].

In this approach, the optimizer maintains a population of candidate plans and iteratively improves them using genetic operators. Selection favors plans with lower estimated execution cost according to a fitness function. Crossover combines subplans or join subtrees from high-quality individuals to generate new candidate plans, while mutation introduces random structural changes to preserve population diversity and avoid premature convergence. By operating on a population rather than a single plan, genetic algorithms can explore a wider portion of the plan space and are less constrained by local optimality assumptions.

Unlike traditional cost-based optimizers that rely heavily on heuristic pruning and deterministic rules, evolutionary algorithms enable exploration of large, non-linear, and discontinuous optimization landscapes. This makes them particularly suitable for complex queries, multi-objective optimization scenarios, and environments where cost models are noisy or incomplete. Empirical studies have shown that population-based optimization can identify high-quality execution plans even under inaccurate cost estimates. However, this benefit comes at the cost of higher optimization overhead and longer convergence times compared to classical optimizers [1].

2) Advantages

Evolutionary and genetic algorithms are particularly effective in large, non-linear, and highly complex optimization spaces where traditional query optimization techniques struggle. Query optimization inherently suffers from combinatorial explosion due to the vast number of possible join orders, operator choices, and access paths. By maintaining a population of candidate execution plans, genetic algorithms can explore diverse regions of the plan space without requiring exhaustive enumeration. This makes them well suited for complex analytical queries and multi-objective optimization scenarios, such as jointly optimizing execution time, resource utilization, and robustness under uncertainty [10].

Another significant advantage is their ability to escape local minima more effectively than greedy heuristics or classical cost-based optimizers. Traditional optimizers often make early commitments to locally optimal decisions based on imperfect or incomplete cost estimates, which can result in globally suboptimal execution plans. In contrast, genetic operators such as mutation and crossover introduce controlled randomness and structural recombination, enabling the search process to move beyond locally optimal regions and

discover higher-quality plans. Empirical studies have shown that this capability is especially valuable when cost models are inaccurate or data distributions are skewed, as evolutionary approaches are less sensitive to individual estimation errors and can still converge toward efficient solutions [1].

3) Limitations

Despite their advantages, evolutionary and genetic algorithms have notable limitations. One major drawback is their high computational cost and extended convergence time. These methods maintain and evolve a population of query plans over multiple generations, and each plan must be evaluated using either the cost model or actual execution metrics. For large queries or complex workloads, this evaluation can become computationally expensive and time-consuming [10].

Another limitation is their dependence on the accuracy of the fitness function, which estimates the quality or cost of a query plan. If the fitness function does not accurately reflect actual execution performance, the algorithm may converge to suboptimal plans, since selection, crossover, and mutation operators rely on these evaluations to guide the search [1].

These challenges underscore why evolutionary approaches are often combined with heuristics or machine learning techniques. Such hybrid strategies can reduce evaluation overhead, improve fitness estimation, and enhance convergence toward high-quality query execution plans.

E. Comparative Evaluation and Future Directions

1) Strengths and Weaknesses

When evaluating AI-driven query optimization strategies, several key observations emerge. Machine learning (ML) models, such as plan-structured neural networks, excel at capturing complex operator-level interactions and can generalize across queries within the same workload, providing high accuracy in latency prediction [14], [17]. Reinforcement learning (RL) approaches offer a principled framework for sequential decision-making in tasks such as join ordering and operator selection. By considering long-term rewards and adapting dynamically to execution feedback, RL methods can improve plan quality over time [11], [24]. Evolutionary and genetic algorithms are particularly effective at exploring large, non-linear, and multi-objective search spaces, with the ability to escape local minima more reliably than greedy or traditional cost-based approaches [1], [10].

Hybrid systems that integrate AI techniques with classical cost-based optimizers demonstrate promising results. These systems leverage the predictive power of ML models or the adaptive learning of RL agents while retaining the reliability, explainability, and safety mechanisms inherent to traditional database management systems [22]. Such combinations often achieve a balance between performance gains and operational robustness.

2) Practical Considerations

Despite their potential, AI-driven optimizers face several practical challenges. Training overhead can be substantial, particularly for deep neural networks or RL agents that require extensive query execution experience. Ensuring generalization to unseen queries is essential to prevent overfitting to the training workload. Explainability remains a concern, as ML and RL models often produce query plans without intuitive human-readable reasoning, which can complicate debugging, tuning, and adoption in production environments [24] .

Integrating AI-driven optimizers into existing DBMS platforms, such as PostgreSQL or Microsoft SQL Server, also demands careful planning. Techniques like experience bootstrapping from traditional cost-based optimizers, query hints, or hybrid evaluation pipelines can facilitate gradual integration while maintaining consistent query performance [22]. These strategies allow AI-based systems to enhance optimization incrementally without disrupting established database operations.

3) Future Research Directions

Future research in AI-driven query optimization is focused on improving adaptability, generalization, and interpretability. Meta-learning approaches may enable models to quickly adapt to new workloads with minimal retraining, while continual learning techniques can maintain performance as database schemas and query patterns evolve. Federated optimization provides opportunities for collaborative learning across multiple organizations without compromising data privacy.

Cross-system transfer learning is another promising avenue, allowing a single model to operate across different DBMSs, reducing training overhead and improving generalization. Additionally, research on explainable AI for query optimization is expected to grow, aiming to produce interpretable predictions that database administrators can understand, trust, and act upon [17], [22], [24].

IV. Comparative Analysis

A. Optimization Paradigm

Traditional query optimizers primarily rely on cost-based and heuristic-driven techniques. These systems estimate the cost of alternative execution plans using statistical summaries, including table cardinalities, histograms, and join selectivities, and then apply rule-based or dynamic programming strategies to select the plan with the lowest estimated cost [10], [1], [24]. While effective for many workloads, these methods often struggle when underlying assumptions—such as uniformity or independence of data—are violated, particularly in queries involving multiple correlated columns or complex predicates. Additionally, maintaining accurate cost models typically requires careful tuning by database administrators, which can be challenging in dynamic or distributed environments [24], [22].

In contrast, AI-assisted query optimization adopts a data-driven, learning-based paradigm. These systems employ machine learning models, including neural networks and reinforcement learning agents, to learn representations of queries, execution plans, and associated costs directly from historical execution data [14], [10], [22]. For instance, Neo uses row vector embeddings to encode correlations between columns and tables, enabling it to predict cardinalities and select optimal join orders even for previously unseen predicates [14]. End-to-end deep learning frameworks further extend this capability by jointly modeling cost and cardinality using tree-structured neural networks, capturing the structure of subplans and complex predicates in a unified representation [22], [14].

By leveraging semantic relationships within both the data and plan structures, AI-assisted optimizers can generalize to new queries, adapt to evolving workloads, and achieve performance improvements beyond the reach of traditional cost-based methods. These advantages, however, require sufficient training data, significant computational resources, and carefully designed features or embeddings to ensure robust model performance [15], [22].

B. Decision Making and Search Space Exploration

Traditional query optimizers explore the space of possible execution plans using dynamic programming, exhaustive or pruned search, and greedy heuristics [10], [1], [24]. For instance, in join order selection, classical methods enumerate candidate subplans, estimate their costs, and retain only the most promising options based on intermediate calculations. Greedy heuristics simplify this process further by selecting the locally optimal choice at each step, such as joining the tables with the smallest estimated cardinalities first. While these techniques provide strong theoretical guarantees, they are limited by the combinatorial explosion of potential plans as the number of tables and join conditions grows. They also struggle to account for correlations across multiple attributes or tables, which can lead to suboptimal decisions for complex queries [22], [14].

AI-assisted optimizers, by contrast, adopt data-driven and adaptive strategies to navigate the plan space more efficiently [14], [22], [10]. Neural network–based models transform query plans into fixed-size embeddings, enabling rapid cost prediction and effective pruning of unlikely candidates. Reinforcement learning approaches frame query optimization as a Markov Decision Process (MDP), where an agent sequentially selects joins, indexes, or subplans to maximize a cumulative reward, typically defined as minimizing overall execution time [11]. Evolutionary and genetic algorithms represent candidate plans as populations, applying selection, crossover, and mutation iteratively to explore the search space and discover high-quality plans [1].

Compared with traditional methods, AI-driven approaches can generalize across diverse queries, capture intricate correlations, and often identify efficient plans more quickly in large or non-linear search spaces. However, these benefits come at the cost of requiring sufficient training data, careful feature or embedding design, and potentially significant computational resources to train or simulate models before deployment [15], [22].

C. Adaptability and Learning

Traditional query optimizers are largely static, relying on pre-defined cost models, heuristics, and historical statistics [10], [1], [24]. Some systems include limited adaptive features, such as runtime re-optimization or feedback loops (e.g., Eddies or adaptive operators in distributed databases), but their capacity to improve over time is constrained by the assumptions encoded in the optimizer and the accuracy of the underlying statistics [24] . For instance, when column distributions deviate from uniformity or independence assumptions, the optimizer may consistently choose suboptimal plans [22], [14].

AI-assisted optimization, in contrast, introduces self-improving capabilities through learning-based approaches. Neural network models can learn from past query executions to predict costs and select efficient plans even for unseen queries. Reinforcement learning–based optimizers refine their decision policies continuously using execution feedback, allowing them to generalize across workloads and evolving query patterns. Furthermore, AI-driven approaches can incorporate online learning, transfer learning, and meta-learning, enabling models to adapt to changing data distributions, schema modifications, or entirely new databases while maintaining high performance [15], [22].

This adaptability enables AI-assisted optimizers to capture correlations across attributes, handle complex query structures, and improve over time—capabilities that are difficult or impossible for traditional cost-based systems. However, these benefits come with trade-offs, including the need for training overhead, ongoing model maintenance, and careful monitoring, making careful integration into production environments essential ([22], [14].

D. Computational Overhead

Traditional query optimizers typically have low computational overhead. Their cost estimation and plan selection processes are deterministic, relying on analytical formulas, heuristics, and dynamic programming to efficiently explore feasible plans [10], [1], [24]. Because these systems do not require model training, their runtime cost is minimal, making them suitable for real-time query planning even on large datasets.

AI-assisted optimizers, however, introduce additional computational costs associated with training and inference. Models must first be trained on historical query workloads to learn cost, cardinality, or plan representations. This training can be resource-intensive and time-consuming, particularly for deep neural networks or reinforcement learning frameworks that explore large search spaces [15], [22]. Once trained, though, inference is generally fast and can be amortized across multiple queries. For example, Neo's row vector embeddings and tree-structured models allow quick prediction of

execution costs for new queries, enabling the system to select better plans without modifying the DBMS at runtime [14], [15].

In summary, while AI-assisted optimizers require an upfront computational investment, the resulting performance gains and adaptability can outweigh these costs in high-throughput or repetitive query environments. Nonetheless, careful attention to training frequency, model size, and inference latency is essential to ensure practical integration into production database systems (Sun and Li 2019; Rigden, n.d.).

E. Explainability and Debugging

Traditional query optimizers are highly interpretable. Their decisions are based on explicit cost models, heuristics, and rules, which allows database administrators (DBAs) to trace the reasoning behind plan choices, understand bottlenecks, and debug performance issues [10], [1], [24]. The transparency of these methods makes it straightforward to predict how changes in table statistics or query structures affect execution plans.

AI-assisted optimizers, on the other hand, often have lower interpretability. Deep learning models generate plans based on learned embeddings, neural network predictions, or reinforcement learning policies, which can be difficult to explain in human-understandable terms [14], [22], [15]. For instance, Neo's row vector embeddings capture complex correlations between attributes, but understanding why a specific join or scan order was selected may not be immediately evident.

To mitigate this, hybrid approaches are emerging that combine AI-driven predictions with cost-based reasoning or feature importance analysis. These methods provide some level of transparency while still benefiting from the predictive power of AI models [22], [24]. In practice, the trade-off between explainability and performance is a key consideration when deploying AI-assisted optimizers in production environments.

V. Experimental Evaluation

A. Evaluation Objectives

The experimental evaluation in this chapter aims to examine how effectively AI-assisted query optimization techniques address the key weaknesses of traditional rule-based and cost-based optimizers discussed in previous sections. Rather than introducing new experimental setups, this section synthesizes and analyzes results reported in prior studies to identify where learning-based approaches provide measurable improvements and where their limitations persist.

A primary evaluation objective is **cost estimation accuracy**, since inaccurate cost models are a central cause of suboptimal plan selection in traditional optimizers. Classical approaches rely on handcrafted formulas and static assumptions that often fail to reflect actual execution behavior, particularly in the presence of correlated predicates or complex execution plans. AI-assisted techniques attempt to learn cost functions directly from execution feedback, making cost prediction accuracy a critical metric for assessing their effectiveness [15], [22].

Closely related is **cardinality estimation error**, which strongly influences join ordering and operator selection. Traditional histogram-based estimators struggle with multi-column correlations and skewed data distributions, frequently leading to significant underestimation or overestimation. Prior studies evaluate AI-based estimators by comparing predicted cardinalities with true result sizes, assessing whether learned models can better capture correlations and reduce estimation errors in complex queries [15], [22].

Beyond estimation accuracy, **query execution time** serves as the most practical end-to-end performance metric. Even when estimation quality improves, the ultimate objective of query optimization is to reduce execution latency and overall resource consumption. Experimental evaluations therefore compare the execution times of plans generated by AI-assisted optimizers with those produced by native database optimizers under

identical workloads. Reported speedups or slowdowns provide concrete evidence of whether learned optimization decisions translate into tangible system-level performance gains [14], [24].

Another important objective is evaluating **plan quality relative to native optimizers**, including mature commercial and open-source systems. Traditional optimizers benefit from decades of engineering effort and domain expertise, making direct comparison essential for establishing the credibility of AI-assisted approaches. Experimental studies often examine whether learning-based optimizers can match or outperform native systems across diverse workloads, query templates, and data distributions, particularly in scenarios involving complex joins or highly correlated attributes [14], [10].

Taken together, these evaluation objectives directly reflect the limitations of traditional optimization identified earlier, including dependence on accurate statistics, weak handling of complex queries, and limited adaptability to dynamic workloads. By jointly examining cost accuracy, cardinality error, execution performance, and plan quality, the experimental analysis provides a comprehensive assessment of whether AI-assisted optimization constitutes a robust and meaningful alternative to conventional techniques [1], [24].

B. Experimental Results and Key Findings

Experimental studies from prior work demonstrate clear performance differences between AI-assisted and traditional query optimization approaches, highlighting improvements in execution efficiency, adaptability, and robustness. This section summarizes key findings from Neo [14], end-to-end learning frameworks [22], and adaptive query optimization systems [24] .

1) Overall Performance

Neo consistently outperforms native optimizers across multiple DBMS platforms. On the JOB workload, Neo achieved query execution times approximately 40% faster than PostgreSQL's native optimizer after 100 training iterations [14]. When evaluated on commercial systems such as MS SQL Server and Oracle, Neo generated plans up to 10% faster than the native optimizers, despite being bootstrapped solely with PostgreSQL plans [14]. For TPC-H workloads, performance gains were slightly lower, likely because commercial systems are heavily tuned for this benchmark [14]. These results indicate that AI-assisted optimizers can surpass both open-source and commercial systems, particularly for queries with complex joins or highly correlated predicates [14].

2) Training Time and Convergence

Neo's learning curves indicate rapid initial improvements, with significant performance gains observed after only nine training iterations on PostgreSQL [14]. Full convergence to performance comparable with commercial systems requires additional iterations, reflecting the more sophisticated planning algorithms used in those optimizers [14]. Wall-clock analysis shows that Neo reaches parity with PostgreSQL within approximately two hours and approaches the performance of all tested optimizers within half a day [14]. Using initial demonstration data from PostgreSQL was crucial to reduce training time and avoid poor performance from randomly sampled plans, which can cause order-of-magnitude execution delays [14].

3) Predicate Representation and Cardinality Estimation

End-to-end learning methods highlight the importance of effective feature representation for optimizer performance [22]. Models incorporating string embeddings combined with rule-based pretraining consistently outperform simpler encodings, especially on complex multi-join queries [22]. Tree-pooling structures further improve semantic representation of compound predicates [22]. These AI-assisted techniques reduce both cardinality and cost estimation errors compared to traditional

estimators, yielding more concentrated and predictable error distributions [22].

4) Adaptability and Robustness

Experiments using Ext-JOB queries demonstrate Neo's ability to generalize to previously unseen query structures [14]. While initial performance on novel queries may be lower, the optimizer quickly adapts within a few training episodes, illustrating the benefits of self-improving learning-based systems [14]. Similarly, adaptive query optimizers in distributed systems such as CockroachDB and Google Spanner exhibit resilience under varying workloads and failure conditions [24] . Adaptive mechanisms achieve up to a 42% reduction in query latency and a 30% improvement in throughput under dynamic conditions, while significantly reducing failed transactions and improving recovery times [24] .

5) Sensitivity to Feature Quality and Search Time

Neo demonstrates selective reliance on input features [14]. For example, when cardinality estimates are unreliable, such as in queries with more than three joins, Neo prioritizes patterns learned from prior executions rather than inaccurate estimates [14]. Analysis of search time indicates that queries with more joins require longer optimization, but modest cutoffs (approximately 250 milliseconds) are sufficient for most scenarios [14].

6) Per-Query Optimization Flexibility

AI-assisted optimizers like Neo support customizable optimization objectives, such as prioritizing total workload execution time versus per-query improvement [14]. Experiments show that this flexibility can reduce overall workload time while avoiding regressions for individual queries, providing an advantage over traditional deterministic optimizers [14].

7) Summary of Key Findings

- AI-assisted optimizers (Neo [14], End-to-End [22]) consistently outperform traditional and commercial optimizers on complex queries.
- Effective feature representation, such as row vector embeddings [14] and string embeddings with tree-pooling [22], is critical for accuracy and generalization.
- Adaptive and learning-based approaches handle dynamic workloads and unseen queries more effectively than static optimizers [14], [24].
- Training and search overheads are manageable, with convergence to competitive performance achievable within reasonable time frames [14].
- These methods enable optimization goals to be tailored at the workload or per-query level [14].

Overall, experimental evidence confirms that AI-assisted and adaptive optimization techniques offer measurable improvements in performance, adaptability, and robustness, supporting the observations made in Sections 5 and 6.1 [14], [22], [24].

C. Cost and Cardinality Estimation Accuracy

Accurate cost and cardinality estimation is fundamental for effective query optimization. Traditional histogram-based estimators rely on precomputed statistics and independence assumptions, which often fail to capture correlations across columns or tables, particularly in complex queries [14]. These inaccuracies can lead to suboptimal join ordering, poor operator selection, and increased query execution time.

1) ML-Based Estimators

Modern machine learning–based approaches, including MSCN, End-to-End learning models, and Neo, leverage learned representations of query structure and data distributions to improve estimation accuracy [22], [14], [15]. By capturing inter-column correlations, complex predicate interactions, and join patterns, these models overcome limitations inherent to traditional histogram-based methods.

2) Evaluation Metrics

The performance of cost and cardinality estimators is typically assessed using the following metrics:

- **Mean Error** – the average deviation from the true cardinality or cost.
- **q-error** – the ratio of estimated to actual values, which emphasizes large discrepancies.
- **Max Error** – the worst-case deviation across all evaluated queries.

3) Key Findings from Literature

- Neo [14] demonstrates substantially lower mean and max errors than histogram-based estimators, particularly for queries involving multiple joins.
- End-to-End models [22] outperform traditional estimators at high-percentile q-errors (90–99th percentile), showing their ability to reduce extreme estimation errors.
- MSCN and Neo [15], [14] achieve more concentrated error distributions, indicating robust predictions even for unseen or complex queries.
- Overall, ML-based estimators can reduce cardinality and cost estimation errors by up to 2–3 × compared to traditional methods [22], [14].

4) Conclusion

Machine learning–based estimators significantly enhance the reliability of query optimization by capturing complex data patterns that traditional approaches cannot. These improvements directly translate into more accurate execution plan selection, reduced query latency, and more predictable performance [22], [14], [15].

D. Query Execution Performance

Query execution performance reflects the ultimate goal of query optimization: generating plans that execute efficiently on the target database system. Prior work has compared actual runtimes of query plans produced by native and AI-assisted optimizers [14], [24].

1) Native Optimizers

Traditional systems, such as PostgreSQL, Oracle, and MS SQL Server, rely on static cost models and heuristic-driven plan selection. While generally robust, these systems can exhibit suboptimal execution under complex queries, skewed data distributions, or dynamic workloads [24] .

2) AI-Assisted Optimizers

Learning-based approaches, including Neo, leverage neural networks trained on prior query executions to generate efficient execution plans.

- **Benchmark Performance:** On the JOB workload, Neo reduced median execution time by up to 40% compared with PostgreSQL and, in some cases, matched or exceeded commercial optimizer performance on MS SQL Server and Oracle [14].
- **Robustness to Unseen Queries:** Neo maintained strong relative performance on extended workloads, demonstrating effective generalization to previously unseen query structures [14].

3) Performance Insights

- AI-assisted optimizers achieve substantial median speedups and reduce worst-case query runtimes.
- Adaptive query mechanisms in distributed systems, such as CockroachDB and Google Spanner, improve performance under dynamic workloads, achieving up to 42% lower latency and 30% higher throughput compared to static plans [24] .
- The combination of learned cost models and adaptive execution allows AI-based optimizers to compete with mature commercial systems without requiring modifications to the underlying database engine [14].

4) Summary

AI-assisted optimizers consistently provide runtime improvements, reduce latency spikes, and adapt to changing workloads,

demonstrating a tangible advantage over traditional static optimization approaches [14], [24].

E. Generalization and Robustness

Generalization and robustness are critical for AI-assisted query optimizers, as real-world workloads often include unseen queries, novel predicates, and shifting data correlations [15], [22].

1) Handling Unseen Queries

Neo has demonstrated the ability to generalize to entirely new queries not present during training. By leveraging row-level embeddings (R-Vectors), Neo can select efficient execution plans for Ext-JOB queries, often outperforming or matching native optimizers with minimal additional training [14].

End-to-End deep learning models similarly exhibit strong generalization to queries with complex compound predicates, accurately predicting cardinalities and costs for multi-join queries outside their training workload [22].

2) Adapting to Data Shifts

AI-assisted optimizers can adjust to correlations across tables and columns that traditional histogram-based estimators often miss. For instance:

- Neo selectively trusts cardinality estimates depending on query complexity, ignoring unreliable inputs when needed [14].
- End-to-End models capture semantic patterns in numeric and string predicates, enhancing robustness to changes in workload distributions [22].

3) Key Takeaways

- AI-based optimizers maintain high plan quality across diverse and evolving workloads.
- Techniques such as feature embeddings and end-to-end learning enable generalization beyond training data, offering a significant advantage over static, traditional optimizers.

F. Training and Overhead Analysis

AI-assisted query optimizers incur additional computational overhead due to model training and inference, but these costs are often offset by improved query execution performance over repeated workloads [14], [15].

1) Training Time

Models such as Neo require multiple training iterations to reach competitive performance. For example:

- Neo reached parity with PostgreSQL in as few as 9 training episodes for simpler workloads.
- Achieving performance comparable to commercial optimizers like MS SQL Server or Oracle required additional iterations [14].
- Wall-clock training time depends on dataset size and DBMS, but typically less than half a day is sufficient to reach robust performance on common workloads [14].

2) Inference Time and Runtime Overhead

Once trained, inference time is minimal relative to overall query runtime. AI-assisted optimizers can generate execution plans efficiently, making runtime overhead practically negligible for repeated or high-throughput queries [14], [15].

The initial training cost is effectively amortized in scenarios where queries are repeated or workloads are substantial, providing consistent performance improvements over traditional static optimizers.

3) Practical Feasibility in DBMSs

Modern relational and distributed systems can integrate AI-assisted optimizers without disrupting standard operations, as training can be performed offline and inference applied at query planning time [14].

Hybrid approaches that combine AI predictions with traditional cost models further balance performance, reliability, and interpretability, enabling practical deployment in production DBMSs [14].

4) Key Takeaways

- Training introduces an initial overhead but provides long-term performance benefits, particularly for complex or repeated workloads.
- When properly amortized, AI-assisted optimization improves query performance without imposing significant operational costs.

G. Adaptive and Online Optimization Results

Adaptive and online query optimization techniques leverage runtime feedback and dynamic plan adjustments to maintain efficient query execution under changing workloads [24], [4].

1) Runtime Feedback and Re-Optimization

Systems such as CockroachDB and Google Spanner collect real-time statistics during query execution and adjust plans when observed row counts or operator costs deviate from estimates [24] .

- Re-optimization can occur at operator-level granularity, enabling the system to correct suboptimal plans mid-execution without restarting queries.

2) Performance Stability under Workload Shifts

Adaptive strategies reduce latency spikes caused by data skew, fluctuating query rates, or node throttling.

- Experiments report up to 42% reduction in query latency under fluctuating read-heavy workloads, and 30% throughput improvement for mixed OLTP/OLAP scenarios [24] .
- During node failures or unexpected system load, adaptive optimizers reduce retry and failure rates by 60–80%, maintaining consistent performance [24] .

3) Distributed Systems Perspective

Adaptive optimization is particularly valuable in distributed and partitioned environments, where static cost models fail to account for network latency, resource contention, or skewed data distributions [11].

Techniques such as plan fragment redistribution, dynamic load balancing, and cost recalibration enable distributed DBMSs to remain resilient and maintain query performance across heterogeneous nodes [24] .

4) Key Takeaways

- Adaptive and online optimizers enhance robustness and stability in dynamic workloads.
- They complement AI-assisted optimization by continuously refining plan quality based on real-time execution metrics, particularly in distributed or high-throughput systems.

H. Summary of Findings

The experimental evidence from prior studies [14], [22], [15] highlights the comparative strengths and limitations of AI-assisted and traditional query optimization approaches.

1) Advantages of AI-Assisted Optimization

- **Cost and cardinality estimation:** Machine learning models such as MSCN, End-to-End, and Neo significantly reduce estimation errors compared to histogram-based methods, particularly for complex queries with correlated columns or multiple joins [22], [14], [15].
- **Query execution performance:** Neo consistently outperforms PostgreSQL and achieves performance comparable to commercial optimizers, with median execution times often improved by 30–40% [14].
- **Generalization and robustness:** AI-assisted optimizers handle unseen queries and novel predicates effectively. Techniques such as row embeddings in Neo and End-to-End models enable rapid adaptation to previously unseen workloads [15], [22].
- **Adaptive and online optimization:** Systems that incorporate runtime feedback,

such as CockroachDB and Google Spanner, maintain stable performance under dynamic workloads, reducing latency spikes and improving throughput [24], [4].

2) Scenarios Where Traditional Optimization Remains Competitive

- For simple or static workloads, rule-based or cost-based optimizers often provide sufficient accuracy with minimal computational overhead.
- Deterministic execution plans are advantageous when explainability, predictability, or minimal runtime overhead is required.
- AI-assisted methods may incur significant training time or require careful feature engineering before consistently outperforming traditional systems [14], [15].

3) Transition Toward Hybrid Optimizers

- Combining AI-based estimators with traditional cost models can balance accuracy, efficiency, and interpretability.
- Hybrid approaches enable incremental adoption of AI techniques in production DBMS environments without compromising reliability.

4) Key Takeaways

AI-assisted optimization provides clear advantages in complex, dynamic, and distributed query workloads, while traditional optimization methods remain relevant for simpler, stable scenarios. The convergence of AI and adaptive optimization paves the way for practical, production-ready hybrid systems.

I. Discussion and Implications

The experimental evaluation across multiple studies highlights several key implications for AI-assisted query optimization in both practical deployment and research contexts.

1) Real-World Deployment

- AI-assisted optimizers, such as Neo and End-to-End models, demonstrate significant improvements in query runtime and estimation accuracy, particularly for complex queries and workloads with highly correlated tables and predicates [15], [22].
- Training overhead and system-specific dependencies must be carefully managed; one-time model training is generally justified when amortized over repeated queries or long-running workloads [14], [15].
- Integration into production DBMSs can benefit from hybrid strategies that combine AI-guided decisions with traditional cost-based heuristics, balancing robustness, reliability, and interpretability [24], [4].

2) DBMS Design Considerations

- Support for online learning and adaptive re-optimization allows optimizers to respond dynamically to shifting workloads and data distributions, enhancing resilience in distributed or high-throughput systems [24] .
- AI models must be paired with effective feature representations, such as row embeddings or predicate encodings, to generalize across query patterns and unseen workloads [15], [22].
- Systems should provide telemetry and execution feedback mechanisms to facilitate model training, validation, and continuous improvement.

3) Research Directions

- Exploration of transfer learning, meta-learning, and domain adaptation to reduce training cost and improve generalization across heterogeneous workloads.
- Development of explainable AI-assisted optimization, enhancing transparency, user trust, and debuggability of complex query plans [14].
- Evaluation on diverse benchmarks and real-world datasets to address limitations of standard workloads (e.g., TPC-H, JOB) and reduce benchmark bias [15], [22].
- Investigation of customizable optimization goals, such as per-query latency targets, SLAs, or workload prioritization, to align

AI-assisted planning with practical business objectives [14].

VI. Conclusion

AI-assisted query optimizers have demonstrated clear advantages over traditional cost-based and rule-based systems. By leveraging machine learning and deep learning models, these optimizers can significantly reduce cardinality and cost estimation errors, resulting in improved query execution performance across diverse workloads, including unseen queries and complex predicate structures. Models such as Neo and End-to-End illustrate the ability to generalize beyond the training data, dynamically adapting to changes in data distributions and query patterns—a capability that remains a key limitation of conventional systems.

Despite these advancements, several challenges persist. Deep learning models often function as black boxes, making interpretability and debugging difficult. The integration of adaptive and online learning mechanisms introduces runtime overhead, which may affect overall system efficiency. Furthermore, performance can vary depending on the underlying DBMS, hardware, and benchmark choice, raising concerns about reproducibility and general applicability.

Future research should focus on explainable and hybrid optimization frameworks that combine the reliability of traditional methods with the adaptability of AI-assisted approaches. Emphasis on robust evaluation, realistic workloads, and production-ready deployment will be crucial for translating experimental gains into operational database systems, ensuring that AI-assisted optimization can be safely and effectively integrated into real-world environments.

VII. References

[1] Almeida, Fernando, Pedro Silva, and Fernando Araújo. 2019. "Performance Analysis and Optimization Techniques for Oracle Relational Databases." Cybernetics and Information Technologies 19 (2): 117–32. https://doi.org/10.2478/cait-2019-0019.

[2] Ammar, Ali Ben. 2016. "Query Optimization Techniques in Graph Databases." International Journal of Database Management Systems 8 (4): 01–14. https://doi.org/10.5121/ijdms.2016.8401.

[3] Azhir, Elham, Nima Jafari Navimipour, Mehdi Hosseinzadeh, Arash Sharifi, and Aso Darwesh. 2019. "Query Optimization Mechanisms in the Cloud Environments: A Systematic Study." International Journal of Communication Systems 32 (8). https://doi.org/10.1002/dac.3940.

[4] Chaudhuri, Surajit. 1998. "An Overview of Query Optimization in Relational Systems." Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (Seattle Washington USA), May, 34–43. https://doi.org/10.1145/275487.275492.

[5] A. Aljanaby, E. Abuelrub, and M. Odeh, "A Survey of Distributed Query Optimization."

[6] Dantuluri, Venkata Narasimha Raju. 2025. "AI-Powered Query Optimization in Multitenant Database Systems." Journal of Computer Science and Technology Studies 7 (4): 802–13. https://doi.org/10.32996/jcsts.2025.7.4.93.

[7] Ganguly, Sumit, Akshay Goel, and Avi Silberschatz. 1996. "Efficient and Accurate Cost Models for Parallel Query Optimization (Extended Abstract)." Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (Montreal Quebec Canada), June, 172–81. https://doi.org/10.1145/237661.237707.

[8] Heitz, Jonas, and Kurt Stockinger. 2019. Join Query Optimization with Deep Reinforcement Learning Algorithms. arXiv. https://doi.org/10.48550/arXiv.1911.11689.

[9] Inersjö, Elizabeth. 2021. Comparing Database Optimisation Techniques in PostgreSQL : Indexes, Query Writing and the Query Optimiser. https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-306703.

[10] Jarke, Matthias, and Jurgen Koch. 1984. "Query Optimization in Database Systems." ACM Computing Surveys 16 (2): 111–52. https://doi.org/10.1145/356924.356928.

[11] Krishnan, Sanjay, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2019. Learning to Optimize Join Queries With Deep Reinforcement Learning. arXiv. https://doi.org/10.48550/arXiv.1808.03196.

[12] Lehmann, Claude, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. arXiv. https://doi.org/10.48550/arXiv.2309.01551.

[13] Li, Ziming, Youhuan Li, Yuyu Luo, Guoliang Li, and Chuxu Zhang. 2025. Graph Neural Networks for Databases: A Survey. arXiv. https://doi.org/10.48550/arXiv.2502.12908.

[14] Marcus, Ryan, Parimarjan Negi, Hongzi Mao, et al. 2019. "Neo: A Learned Query Optimizer." Proceedings of the VLDB Endowment 12 (11): 1705–18. https://doi.org/10.14778/3342263.3342644.

[15] Marcus, Ryan, and Olga Papaemmanouil. 2018. Towards a Hands-Free Query Optimizer Through Deep Learning. arXiv. https://doi.org/10.48550/arXiv.1809.10212.

[16] Mikhaylov, Artem, Nina S. Mazyavkina, Mikhail Salnikov, Ilya Trofimov, Fu Qiang, and Evgeny Burnaev. 2022. "Learned Query

Optimizers: Evaluation and Improvement." IEEE Access 10: 75205–18. https://doi.org/10.1109/ACCESS.2022.3190376.

[17] Ortiz, Jennifer, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2019. An Empirical Analysis of Deep Learning for Cardinality Estimation. arXiv. https://doi.org/10.48550/arXiv.1905.06425.

[18] Padia, Shyam, Sushant Khulge, Akhilesh Gupta, and Parth Khadilikar. 2015. Query Optimization Strategies in Distributed Databases. 6.

[19] P.Karthikeyan, M., K. Krishnaveni, and Dac-Nhuong Le. 2024. "Analysis of Multi-Join Query Optimization Using ACO and Q-Learning." International Journal of Computing and Digital Systems 16 (1): 1523–33. https://doi.org/10.12785/ijcds/1601113.

[20] Ramadan, Mohamed, Ayman El-Kilany, Hoda M. O. Mokhtar, and Ibrahim Sobh. 2022. "RL_QOptimizer: A Reinforcement Learning Based Query Optimizer." IEEE Access 10: 70502–15. https://doi.org/10.1109/ACCESS.2022.3187102.

[21] Schmidt, Michael, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. 2011. "FedBench: A Benchmark Suite for Federated Semantic Data Query Processing." In The Semantic Web – ISWC 2011, edited by Lora Aroyo, Chris Welty, Harith Alani, et al. Springer. https://doi.org/10.1007/978-3-642-25073-6_37.

[22] Sun, Ji, and Guoliang Li. 2019. An End-to-End Learning-Based Cost Estimator. arXiv. https://doi.org/10.48550/arXiv.1906.02560.

[23] Zhou, Xuanhe, Chengliang Chai, Guoliang Li, and Ji Sun. n.d. Database Meets AI: A Survey.

[24] D. J. Rigden, "Adaptive Query Optimization in Distributed SQL Databases," International Journal of Technology Management, vol. 2, no. 3.

[25] R. Marcus and O. Papaemmanouil, "Plan-Structured Deep Neural Network Models for Query Performance Prediction," Proceedings of the VLDB Endowment, vol. 12, no. 11, pp. 1733–1746, Jul. 2019, arXiv:1902.00132 [cs]. [Online]. Available: http://arxiv.org/abs/1902. 00132