An Efficient Query Optimization Framework Based on MCTS and LTR

Chaofan Zhao ZHAOCHAOFAN@NUDT.EDU.CN
Yize Sui SUIYIZE18@NUDT.EDU.CN
Ruochun Jin JINRC@NUDT.EDU.CN
Jing Ren[*] RENJING@NUDT.EDU.CN
College of Computer Science and Technology, National University of Defense Technology, Changsha, China

Editors: Vu Nguyen and Hsuan-Tien Lin

Abstract

Identifying the optimal query plan is always a fundamental task of query optimization in database management system (DBMS). However, traditional query optimization methods face significant challenges in continuously enhancing query performance due to complex query sentences, intricate data distributions and the exponentially growing search space of table joins. In this paper, we propose a formidable query optimization framework called MRQO (Integrating-MCTS-and-LTR-for-Query-Optimization). This framework utilizes the Monte Carlo Tree Search (MCTS) algorithm to find a comprehensive set of join orders for a query, and uses these join orders as hints to generate corresponding query plans. Additionally, it employs the Learning-to-Rank (LTR) approach to train a relative ranking model, achieving higher efficiency and accuracy in identifying the optimal query plan from all plans. Experimental results on PostgreSQL demonstrate that the proposed MRQO can achieve stable performance and match or even outperform both traditional query optimizers and advanced learned optimizers based on Deep Reinforcement Learning (DRL) in terms of query optimization efficiency.

Keywords: Query Optimization; MCTS; Learning-to-Rank.

1. Introduction

In relational databases, determining the optimal query plan is a critical component of query optimization, as it significantly affects the cost and efficiency of SQL query execution. A query plan typically includes the order of table joins, specific operations between joins, and index information, among others. Crucially, the order of table joins, owing to their commutative and associative properties, can be rearranged and combined in various ways, leading to significant differences in the execution performance of the same query with different join orders. Based on this principle, by adjusting the join order of tables, we can effectively change the execution path of queries and generate query plans with different efficiency. In this paper, we propose MRQO, a framework that utilizes the reliable join orders of queries as hints to generate sets of high-quality query plans and selects the optimal one for execution. This framework aims to optimize the overall query performance within the database by optimizing the join order. Therefore, this paper is mainly based on solving the following

[*] Corresponding Author

two problems to achieve the ultimate goal of query optimization. (1) How to find a reliable set of join orders to generate high-quality query plans? (2) How to reliably evaluate candidate query plans and select the optimal plan for execution?

In queries involving multiple table joins, the potential combinations of join orders grow exponentially as the number of tables increasing, making it complex and difficult to find the most cost-effective join order among numerous possibilities, which is an NP-hard problem (Yan et al., 2023). Thus, the critical challenge is to efficiently pinpoint the excellent join orders within a limited search space to enhance the performance of the query optimizer. We explore whether it is possible to search for all join orders for a given query, whether specific search strategies can be learned from previous results to reduce search time and efficiently determine the optimal join order. Therefore, this paper proposes a learning-based approach for join orders selection. We use the MCTS strategy to intelligently select the join orders to reduce the number of candidate join orders to be enumerated. And we train a value network to predict the most probable join orders based on the query's structure.

In MRQO, for determining the optimal query plan, we need to select from a reliable candidate plan set consisting of query plans generated based on join orders and the query plan produced directly by PostgreSQL itself. Due to the inherent difficulty of using machine learning models to predict the cost or latency of query plans, regression models require a large amount of training resources and may generate unnecessary errors. MRQO uses the learning-to-rank technology and uses Tree Convolutional Neural Network (TCNN) as the underlying model. During the prediction process, listwise relative ranking is performed on all candidate plans, until the plan with the lowest cost or latency is found. This transforms the regression problem that relies on accurate cost or latency estimation into a relative order prediction problem, ensuring more stable optimization performance of the evaluation model.

The key contributions of this paper can be summarized as follows:

1. We propose a query optimization framework, MRQO, which ingeniously integrates two strategies, optimizing the selection process of candidate plans on the one hand, and improving the selection accuracy of the optimal query plan on the other hand, achieving an improvement in the performance of the query optimizer.

2. We propose a join order recommendation algorithm based on the MCTS strategy to address the performance limitation caused by the difficulty in enumerating join orders. This method effectively avoids the uncertainty caused by random selection, while reducing the search space and ensuring high standards of plan quality.

3. We propose an LTR model to address the issue of optimal query plan selection caused by inaccurate cost estimation in regression models. The LTR model transforms the regression problem that relies on accurate cost estimation into a relative ranking problem, reducing errors and consumption in the query plan selection process and ensuring more stable optimization performance.

4. Experimental results robustly demonstrate the superiority of MRQO in query optimization. Compared with existing advanced learned optimizers, MRQO can achieve performance improvements in almost all benchmark tests.

2. Related Work

Over the past forty years of database technology development, numerous scholars have proposed various solutions to the query optimization problem. Firstly, there are traditional poptimization strategies that use pruning techniques based on cardinality and cost estimation to search for all possible join order solution spaces. For example, DPhyp (Moerkotte and Neumann, 2008) introduced an algorithm based on dynamic programming and top-down partition search for optimally ordering joins. GEQO (Chande and Sinha, 2011) employed heuristic search strategies based on genetic algorithm. LinDP (Neumann and Radke, 2018) introduced an adaptive optimization framework that can accurately solve most common join queries. MPDP (Mancini et al., 2022) discussed large-scale parallel processing techniques. These Traditional methods typically rely on static enumeration without providing feedback on the quality of query plans. They cannot learn from experience, so they often choose the same bad plans repeatedly.

To address the shortcomings of traditional solutions, many scholars have begun exploring new ways to improve query optimizers using machine learning techniques since 2018 (Li et al., 2021). At the current stage, a variety of learning-based query optimization methods have emerged. The core of these methods lies in using trained modules to replace key components of databases, thereby addressing the issue of query performance. Firstly, there are a series of individual learned components for join order selection. For instance, DQ (Krishnan et al., 2018) and ReJOIN (Marcus and Papaemmanouil, 2018) combine cost models with reinforcement learning to automatically derive efficient search strategies, with DQ being the first one to use value-based method and ReJOIN being the first one to utilize policy-based method. But in the encoding of join trees, both of them only consider the static information of tables and columns, without considering the structural information of the tree, which may lead to poor plans. RTOS (Yu et al., 2020) employs Tree-LSTM to capture structural information of join trees for join order selection, which further optimizes the strategies of DQ and ReJOIN. All three of them adopt random search strategy, which will bring limitations to optimization performance. JOGGER (Chen et al., 2022) reduces the parameter count in deep neural networks through a graph-based model, aiming to achieve higher optimization efficiency while consuming less optimization time and computational resources. AlphaJoin (Zhang, 2020) uses MCTS to determine join orders. It can simulate the process of joins in a tree structure and select the join order with the highest estimation performance. Although it can demonstrate good performance, the accuracy of the recommended join order in its value network only reaches 52%. This indicates that its performance is very unstable.

In addition to the DRL based join order optimization model, there are also some end-to-end learned optimizers. They can provide more selection functions besides join order selection, including join operator selection, index selection, query plan generation and selection. These optimizers have made significant progress in improving the quality of query plans. Taking Neo (Marcus et al., 2019) as an example, it is the first end-to-end learned optimizer. This innovative system does not rely on traditional database cost models and cardinality estimation models, but instead trains value networks to predict the execution time of query plans, greatly improving query efficiency. Another example is Bao (Marcus et al., 2020), which uses Context Multi Armed Bands (CMABs) for modeling based on the selection of physical operator hints, further optimizing the quality of query plans. This model only focuses on query hints and does not require rebuilding the entire optimizer, making the architecture simple and efficient. But both of Neo and Bao are based on regression models to predict plan execution latency. It is difficult to achieve accurate predictions when facing complex queries. Lero (Zhu et al., 2023) designs a LTR optimizer that does not predict the exact cost of query plans, but instead compares candidate plans pairwise to select the better one. COOOL (Xu et al., 2023) adopts a similar principle to Bao to recommend learned physical operator hints, but their difference is that COOOL does not rely on regression models to predict exact execution latency, while focuses on relative ranking between plans, which makes it more flexible and accurate in handling complex queries. However, when vectorizing query plan trees, COOOL introduces cost estimation of nodes based on the PostgreSQL cost model in node vectorization. This will introduce additional errors in the prediction model in COOOL.

Our method has been deepened and optimized based on previous research. By employing the MCTS strategy, we efficiently enumerated join orders, optimizing the random strategies used in methods such as DQ, ReJOIN, and RTOS, enabling us to explore the candidate solution space more accurately. Considering the uncertainty of the Upper Confidence Bounds applied to Trees (UCT) algorithm in Alphajoin and the spatial complexity of join orders in complex queries, we selected the top-k join orders searched by MCTS to eliminate the instability in Alphajoin. We utilized these chosen join orders as hints to generate high-quality candidate query plans, add a plan only generated by PostgreSQL itself. This completes the unique query plan generation process of MRQO. Subsequently, we adopted a LTR model instead of traditional learning regression models to achieve the selection of the optimal query plan. Unlike Lero and COOOL, MRQO does not consider all data-level features such as the cost and touched tables of each node during the query plan encoding process. This eliminates the impact of inaccurate cost and simplifies the information of tree node vectorization. And MRQO adopts a listwise approach for model training and prediction, reducing the number of iterations of each model. This not only accelerates the training and optimization of the LTR model, but also significantly improves the efficiency of selecting the optimal query plan.

3. Methods

3.1. The Framework of MRQO

Figure 1 demonstrates the framework of MRQO. It integrates a join order explorer and a query plan comparator with the DBMS. The join order explorer deploys a value network model based on MCTS strategy to list a set of optimal candidate join orders with the limitations of search steps. The query plan comparator integrates a learning-to-rank model to select the best query plan for execution in a more efficient way. The processing pipeline of SQL queries within MRQO is as follows: for an input query, the join order explorer is responsible for generating a set of diversified join order options that are potentially high-performing. Then, the join orders are processed by the DBMS to generate corresponding query plans based on join orders as hints. Moreover DBMS will generate a plan itself without using hint. After that, the plan comparator takes over the work. It will evaluate

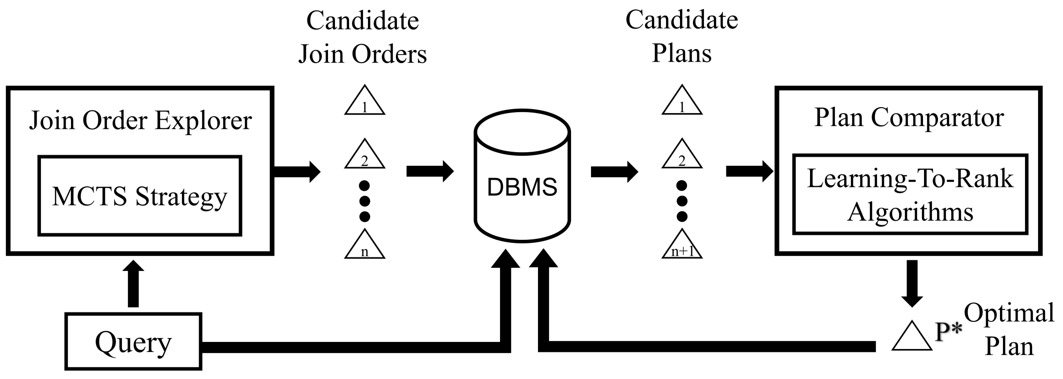


Figure 1: The framework of MRQO.

the relative ranking of a previously generated set of query plans and ultimately recommend the best plan to the DBMS execution engine to effectively obtain query results.

During the training phase, the execution engine assesses the performance of all query plans produced by candidate join orders. Then using this performance data as a training set to update and retrain the join order explorer and plan comparator. Notably, these training steps are separate from the real-time query plan selection process, thus they do not impact the efficiency of query execution. Such design ensures both the efficiency and adaptability of the system, making the optimization process to remain stable.

Algorithm 1 MRQO Framework

```
Input: A query $Q$.
Output: An optimal plan $p^*$.
 1: Candidate join order $J$ = MCTS($Q$, Join order explorer);
 2: Hint $H$ = Hints($J+\emptyset$);
 3: Candidate plan $P = \emptyset$;
 4: for each hint $h \in H$ do
 5:     $p$ = Optimizer($Q$, $h$);
 6:     Add $p$ to $P$;
 7: end for
 8: Rank = LTR($P$, Plan comparator);
 9: $p^*$ = Plan selection(Rank);
10: return $p^*$;
```

Overall algorithm. Algorithm 1 demonstrates the overall process of MRQO. For a given query, the framework first uses MCTS to find some good join orders, and stores these join orders as hints in the hint set. The hint set contains an empty subset (Line 1-2). For the input query, it generates a complete execution plan through the original query optimizer based on each hint (Line 3-6). Then it uses a LTR comparison model to select the optimalt query plan as the output (Line 7-9).

3.2. Join Order Explorer

In this subsection, we illustrate how MRQO utilizes the MCTS strategy to achieve exploration of join orders. This includes the process and algorithm of MCTS in the selection of join order, the structure of our designed join order value estimation network, and the details of the structure of join order encoding and query encoding.

3.2.1. MCTS for Join Order Selection

In the query plan tree, each leaf node corresponds to a specific join order. It is extremely costly to exhaust all possibilities and accurately estimate their performance. Therefore, we consider using the MCTS strategy to efficiently discover superior join orders.

The MCTS algorithm is a method for making optimal decisions within the field of artificial intelligence (Liu et al., 2018), combining the breadth of random simulation with the precision of tree search. This algorithm expands the search space outward through simulation until a complete tree structure is constructed, and then iteratively feeds back the final decision results to update each node in the tree.

In our search process, we employed the UCT (Gelly and Wang, 2006) algorithm to address the problem of choosing join order. The UCT algorithm balances exploration and exploitation, reducing the risk of falling into local optima, thus achieving significant optimization over random strategy. The detailed calculation formula for UCT is:

$$U(v_i, v) = \frac{Q(v_i)}{N(v_i)} + C\sqrt{\frac{\log N(v)}{N(v_i)}}. \tag{1}$$

In this formula, the symbol $U(v)$ represents the utility of a node. The magnitude of this utility directly influences which nodes are ultimately chosen as targets. Here, v_i denotes the currently considered node, while v represents its parent node. $Q(v_i)$ reveals the immediate reward value received by the current node. $N(v_i)$ and $N(v)$ record the number of times the current and parent nodes have been visited, respectively. C is the hyper-parameter used to control the sensitivity of exploration. The formula is divided into two parts: the first half embodies the concept of exploitation, which represents the possibility of selecting nodes. The latter half signifies exploration, implying that if a node has been visited too frequently, the algorithm will increase the likelihood of exploring other nodes. Thus, the UCT algorithm ingeniously balances the relationship between exploitation and exploration when choosing the next child node.

As the search process progresses, particularly when query Q involves an increasing number of tables, and with the increment in the number of search iterations, attempting to visit all possible nodes becomes impractical. To effectively control the size of the search space from the root node to the leaf nodes, we set a maximum simulation count for each node as $M_n = N_s \times S_t$. Here, N_s is the number of child nodes under the current node, and S_t is a coefficient factor used to control the size of the entire simulation space. Consequently, the total number of simulations for each query is limited to $\sum_1^J M_n$, where J represents the number of joins in each query. This design aims to ensure that the search process is targeted and efficient, preventing the wasteful use of resources.

Figure 2 illustrates the complete workflow of MCTS. The simulation process can be subdivided into the following four steps:

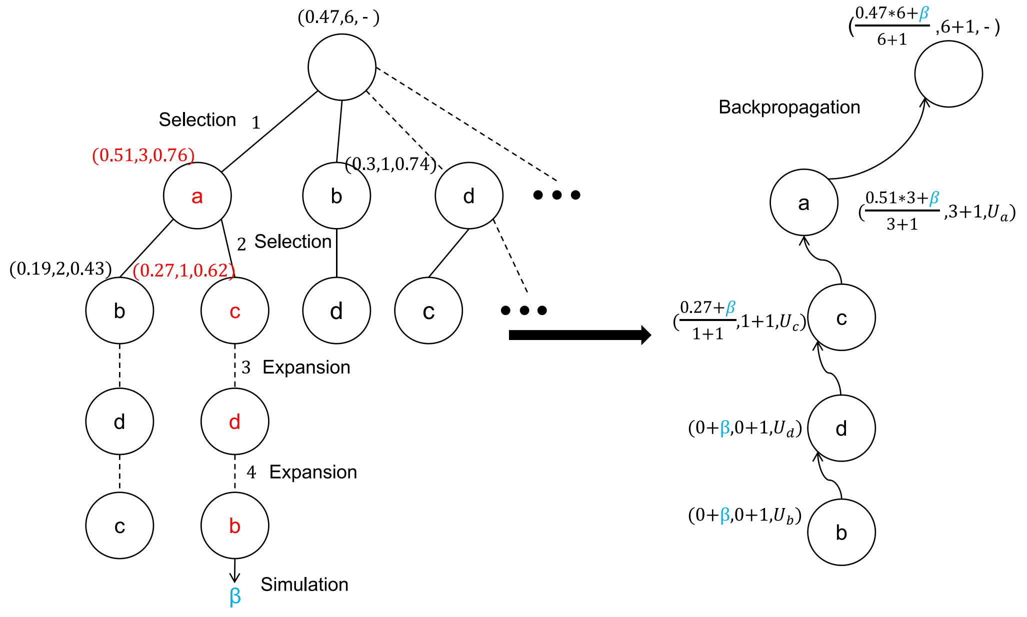


Figure 2: The workflow of MCTS. Within the tree structure, each node is defined by three parameter values $(Q(v), N(v), U(v))$: the immediate reward $Q(v)$, the visit count $N(v)$, and the node utility $U(v)$.

1. Selection: Employing the wisdom of the UCT algorithm, the search begins at the root node and recursively selects the most promising child nodes to explore. When the journey reaches an unexpanded leaf node, the search naturally transitions into the expansion phase. As shown in Figure 2, in the first step, MCTS selects node "a" because it has the maximum utility value of 0.76. In the second step, $U(c) > U(b)$, so it selects "c". Then, because "c" has node that have not been carried out, the program enters the expansion part.

2. Expansion: In this phase, the tree structure is enriched as the algorithm created new child nodes and added the best-performing one to the existing tree graph. That is, select nodes "b" and "d" for the expansion of the third step and the forth step in Figure 2, corresponding to the join order (a, c, b, d).

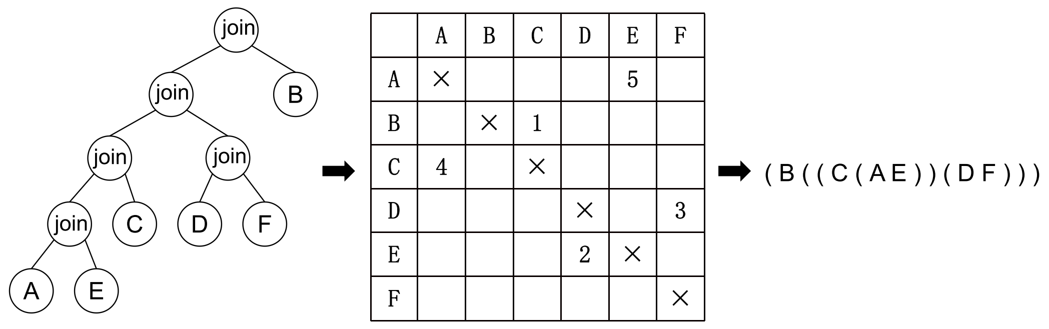
3. Simulation: Next, the algorithm carries out a series of simulations from the root to the leaves, continuing until all the tables involved in the query are joined.

4. Backpropagation: Finally, the results of the simulations are used to retroactively update the current decision sequence. This process propagates backward from the leaf nodes to the root node. In the unfolding of the backpropagation stage, the visit count for each node in the join order is incremented by one for each step. Concurrently, the simulated value feedback of this join order is β, derived from the estimate network we define, which assesses the value of generating the join order.

3.2.2. Estimate Network

To evaluate the performance of join order and provide feedback on the value of each node, we have constructed a deep neural network model based on the PyTorch framework. This model plays a crucial role in approximating the value function during the MCTS process. It consists of five hidden layers, which are interconnected through full connections, forming a formidable network for feature extraction and processing. Our neural network accepts inputs that include both query features and join order information, and outputs prediction scores β for join Orders, corresponding to the execution time of the input queries. We define β as between 0 and 1 based on the measurement of execution time, with lower β indicating longer execution time.

During the forward propagation process, the input data is sequentially passed through each hidden layer. At each layer, the ReLU activation function is applied to extract and refine the deep features of the input data. To prevent model overfitting, we employ Dropout technology after each hidden layer, randomly discarding some neurons to enhance the model's generalization capabilities. Finally, we use the LogSoftmax activation function to ensure that the output values fall within a reasonable range, making the model's predictions both accurate and reliable. We continuously train the estimate network using the executed query plan and the extracted complete join order in DBMS, which is independent of query optimization procession and therefore does not affect query efficiency.



	A	B	C	D	E	F
A	×				5	
B		×	1			
C	4		×			
D				×		3
E				2	×	
F						×

Figure 3: The encoding method of MCTS. The non-zero elements within the matrix unveil the specific join relationships between tables, and the magnitude of these values further delineates the progression of the join order.

3.2.3. Join Order Encoding and Query Encoding

To effectively encode the join order within queries, we have embraced the method of constructing a two-dimensional matrix. We begin by creating an $N \times N$ matrix, where N signifies the total count of table names in all SQL queries. As depicted in Figure 3, the join relationship in the plan tree can be expressed in the form of a matrix. A non-zero value represents a join between two tables, and a larger value indicates that the join is more advanced. This representational strategy offers not only feasibility for encoding but also convenience for decoding.

During the model training phase, to align with the algorithm's requirements, we flatten this two-dimensional matrix into a one-dimensional array. However, when it comes to presenting the outcomes, we revert it back to a format that can be comprehended by the database system. In Figure 3, the encoded join order could be manifested as (B((C(A E))(D F))). By employing this approach, our model achieves effective encoding and decoding of intricate query join orders, maintaining logical coherence while ensuring practicality and compatibility with the database system.

Our query encoding consists of two parts, which reveal the connection relationships between tables and specific operational predicates, respectively. The former adopts the same matrix as the join order encoding, ingeniously reflecting whether there is a relevant operation between two tables through the concise use of 0 and 1. The latter employs one-hot encoding to cleverly demonstrate which join predicates are utilized. This encoding method is both clear and efficient, facilitating the understanding and training of the estimate network.

3.3. Plan Comparator

In this subsection, we illustrate how MRQO utilizes the LTR model to select the optimal plan from the generated candidate plans. This includes the framework of the LTR model, the encoding method and contents of the plan, and the operational process of ranking of the plan.

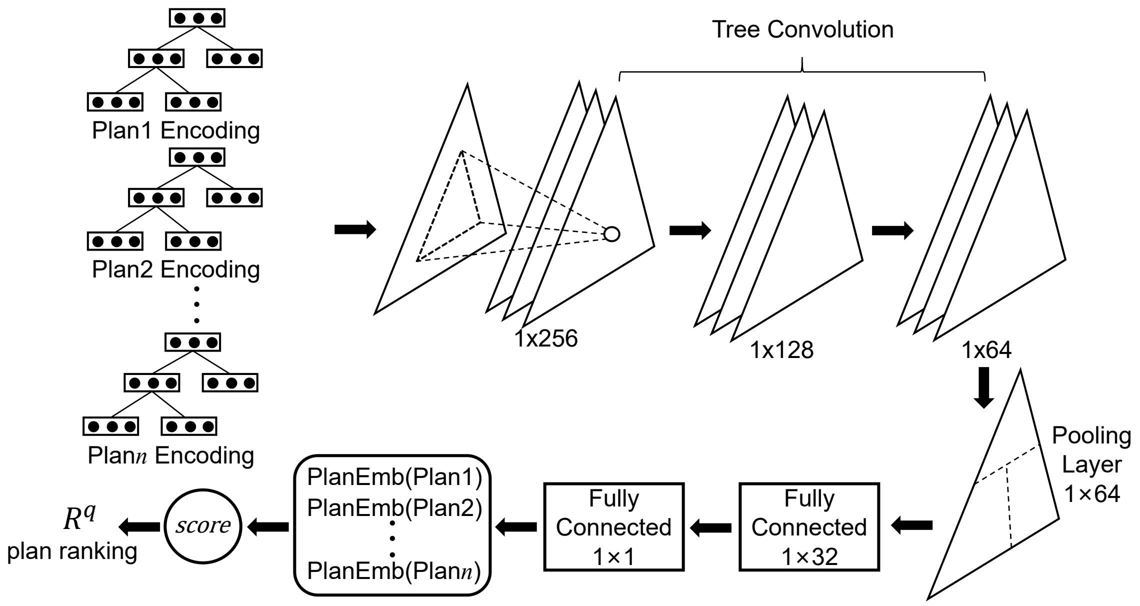


Figure 4: The structure of plan comparator.

3.3.1. The Structure of Plan Comparator

The comparator is essentially a LTR model. LTR is a supervised learning framework specifically designed for training tasks that require sample ranking, can be applied to most neural network models. In our research, we have adopted the listwise (Swezey et al., 2020) LTR approach and chosen the TCNN (Zhao and Xia, 2018; Marcus and Papaemmanouil, 2018; Marcus et al., 2019) as our underlying model. We use the latency relative ranking of the query plans as the label for training the LTR model. By transforming the regression problem of estimating the exact latency of query plans in the optimizer into a ranking problem of comparing the relative latency between query plans. This method may prove to be more robust than regression models for query plan latency that span a wide range of magnitudes. Figure 4 is the overall framework of our plan comparator, which calls plans into TCNN for listwise comparison after coding. Firstly, all vectorized query plan trees are convolved through a three-layers of tree convolution, and then the tree structure is flattened into a single vector using dynamic pooling. And the pooling vector is mapped out through two fully connected layers. Plan Comparator can output the ranking scores of each plan, and the estimated latency orders can be acquired by sorting the scores of all candidate plans.

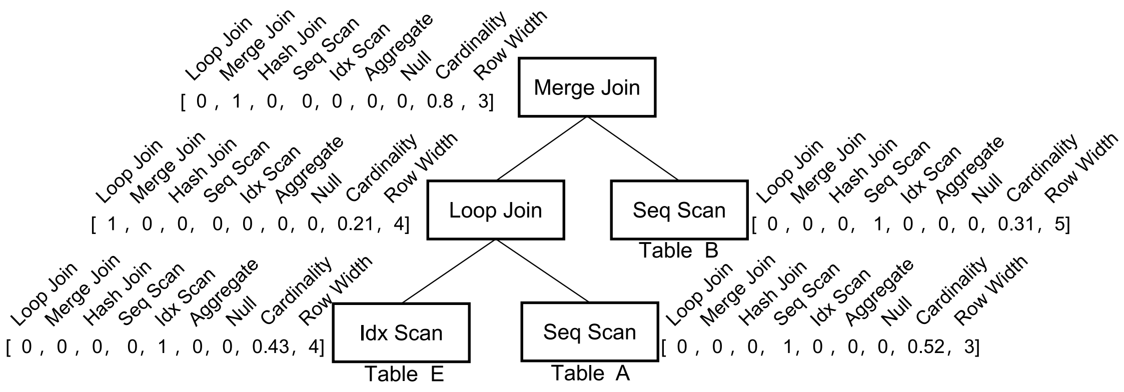


Figure 5: Vectorized query plan tree.

3.3.2. Plan Encoding

In response to the tree structure of query plans, MRQO encodes each node and then weaves all nodes into a complete tree graph according to the structure of the plan. As shown in Figure 5, it presents a example of our coding of the query plan tree nodes. In node encoding, we only include three features: one-hot encoding of operations on corresponding nodes, cardinality estimation, and output row width. To demonstrate the robustness of the LTR algorithm, we did not consider cost estimation in plan encoding, thereby eliminating the potential impact of inaccuracies in the database's cost model.

3.3.3. Comparing Operation

In MRQO, based on listwise approach, we do not need to predict the cost or execution time of the encoded query plan. Instead, we try to find useful information from the tree structure to achieve direct ranking comparison between plans. we map query plan P_i into one-dimensional vectors $PlanEmb(P_i)$. Comparator model calculates the ranking score of each plan:

$$s_i^q = CmpPlan(q, P_i). \tag{2}$$

Then obtain a ranking of all scores: $R^q = s_1^q > s_2^q > \cdots > s_n^q$. We use π represent the result of the target ranking: $R_\pi^q = s_{\pi 1}^q > s_{\pi 2}^q > \cdots > s_{\pi n}^q$. This indicates that for query $q \in Q$, $P_{\pi 1}$ has the lowest latency and $P_{\pi n}$ has the highest latency. We define the probability of implementing π based on the Plackett-Luce model (Xia et al., 2019) as:

$$Succ(R_{\pi}^{q};\theta) = \prod_{i=1}^{n} \frac{exp(s_{\pi i}^{q})}{\sum_{j=i}^{n} exp(s_{\pi j}^{q})}, \tag{3}$$

where θ is the parameter the model to trained.

The goal of Plan Comparator is to maximize the likelihood of correctly ranking candidate plans. For this goal, we define the loss function of the model as the negative log-likelihood function of the probability formula:

$$L(\theta) = -\sum_{q\in Q} \ln Succ(R_{\pi}^{q};\theta). \tag{4}$$

It can be seen that the more query data trained, the more convergent the model becomes.

4. Experiments

Our main objective in the experiment is to investigate the difference in execution latency of the optimal query plan determined by between MRQO and other methods, in order to verify whether MRQO can significantly improve query execution performance in DBMS.

4.1. Experimental Setup

4.1.1. Datasets and Workloads

Our experiments are conducted on two widely-used public datasets and their corresponding workloads. Initially, we use the IMDB dataset, constructed from real-world data, encompassing 21 tables that include information about movie directors, actors, production companies, and other related details. JOB is the corresponding workload of IMDB. It is a static workload, which contains 113 actual query templates. They contain different predicate processing and table join relationships. The number of relationship tables in each query ranges from 4 to 17. And these queries involve between 3 and 16 complex joins, with an average of 8 joins per query.

Another one is the STATS dataset and its workload. It is mainly used to evaluate the performance of query optimizers in planning generation and selection during end-to-end processes. Although STATS only contains 8 user contribution content tables, its data distribution is much more complex than IMDB. STATS contains 146 query templates with different join sizes and types. The number of relationship tables for each query ranges from 2 to 8, and the number of joins is between 2 and 18. We keep the join relationship of the template and randomly replace the predicates to generate 1000 new queries for each dataset as training data.

4.1.2. Environment

In our experimental setup, we choose PostgreSQL-13.1 as the database management system and install the pg_hint_plan module to integrate MRQO. Moreover, our neural network model is implemented using the PyTorch framework to ensure efficient and flexible computational performance.

4.1.3. Baselines

In our experiments, we compare MRQO with the original query optimizer of PostgreSQL-13.1 and the query optimizer deployed with Alphajoin and Lero. We conduct testing using two data benchmarks in the same environment and system to achieve performance comparison.

4.1.4. Metrics

To comprehensively evaluate the performance of different optimizers across various workloads, we focus on two key performance indicators: total latency and tail latency. Total latency serves as a measure of the optimizer's average performance, representing the cumulative latency time needed to complete all queries of an entire workload. Tail latency reflects the stability of the optimizer, indicating the time spent on the portion of queries with the highest latency within a set of workloads. These two metrics provide us with a detailed portrait of the optimizer's performance.

In comparison to PostgreSQL, MRQO requires additional planning time to generate more query plans. However, the creation of multiple query plans for a query incurs only a marginal increase in planning time since the relevant information about the query is already present in the cache. Moreover, planning time accounts for a minor fraction of the total latency (less than 1%) (Zhu et al., 2023), and thus does not impact the overall performance comparison between MRQO and other query optimizers. Therefore, in our experiments, we can just focus on the comparison of total latency and ignore the influence of planning time.

4.2. Experimental Results

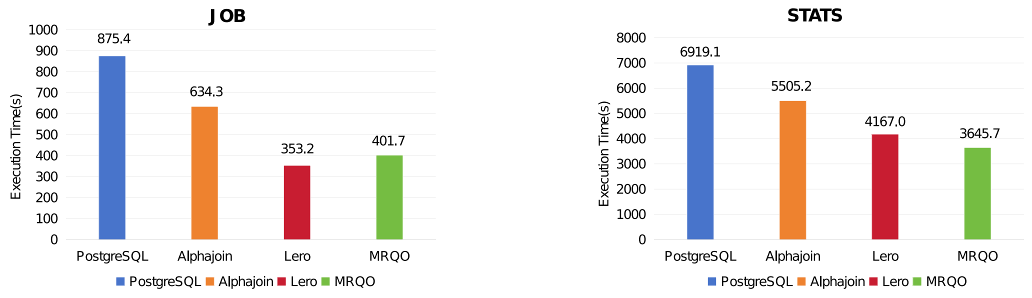
In our research, we will use a trained join order estimator to implement the MCTS strategy. The training data for this estimator comes from the results previously executed from DBMS.

4.2.1. performance of MRQO when working like Alphajoin

We validate the advantages of the MCTS algorithm in MRQO. We only use the optimal join order found by MCTS in MRQO to guide query optimization, which is similar to the optimization process of Alphajoin. The comparison results with Alphajoin are shown in Table 1. The MCTS strategy in MRQO can stably find better join orders on two benchmarks. This indicates that our estimate network can better predict the value of join orders.

Table 1: The total latency(s) of executing JOB and STATS test workloads after optimizing by Alphajoin's MCTS and MRQO's MCTS.

	Alphajoin	MRQO
JOB	634.3	548.1
STATS	5505.2	4773.3



(a) (b)

Figure 6: The total latency of executing JOB and STATS test workloads in PostgreSQL, Alphajoin, Lero and MRQO.

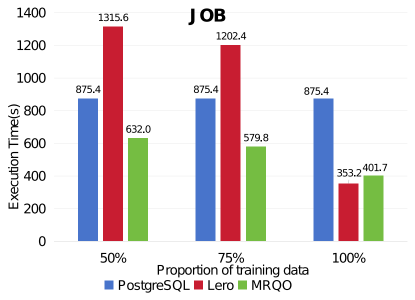
4.2.2. Performance of MRQO after sufficient training

In subsequent experiments, in order to obtain the best results during the search process, we decide to fix the coefficient factor S_t in MCTS to a value of 11 after conducting multiple verifications. And for each query, we extract the top-six join orders found based on MCTS. Then, we use six join orders as the hint set to generate candidate query plans for the corresponding query. And in candidates, we add a query plan that is generated by the original optimizer of PostgreSQL without using hint. Finally, we use the LTR model to predict candidate query plans and compile the best query plan for execution.

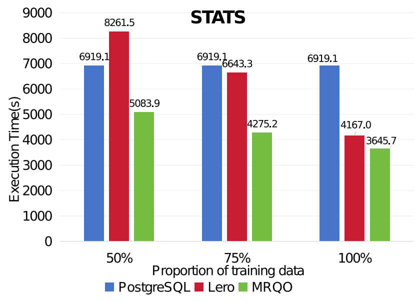
We compare the performance of different query optimizers with stable models. We fully train the plan selection model using all training data and execute template queries for two workloads in PostgreSQL, Alphajoin, Lero, and MRQO for testing. As shown in Figure 6(a), our experimental results reveal that when completing all JOB queries, MRQO improves time efficiency by 54.1% compared to the PostgreSQL optimizer and by 36.7% over AlphaJoin. In Figure 6(b), after executing the STATS workload, MRQO brings a 47.3% overall performance improvement compared to PostgreSQL and a 33.8% improvement compared to Alphajoin. MRQO and Lero show different optimization effects in two datasets. For the STATS dataset, although its data structure is more complex, MRQO can bring better optimization results than Lero. Due to fewer number of tables in the STATS, our MCTS strategy can achieve a more comprehensive search under the same constraints, thereby finding more better join orders. Therefore, when facing query workloads with fewer tables, MRQO can bring additional performance optimization.

4.2.3. Performance of MRQO after insufficient training

The results in Figure 6 indicate that both the pairwise algorithm in Lero and the listwise algorithm in MRQO can accurately select plans after sufficient training. So we consider training the plan selection models in Lero and MRQO on data with 50%, 75%, and 100% of two training workloads and then test them on the test sets. Figure 7 shows the test results of the experiment. If the test workloads include query features that are not present in the training workloads, the learned model will exhibit significant performance regression. Both Lero and MRQO in Figures 7(a) and (b) exhibit performance regression, with Lero even performing worse than PostgreSQL in some cases, while MRQO consistently outperforms

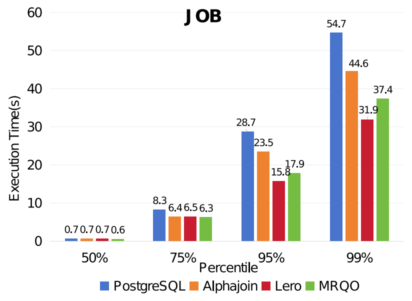


(a)

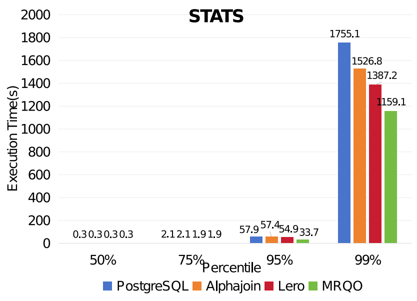


(b)

Figure 7: The total latency of executing JOB and STATS test workloads in PostgreSQL, Lero and MRQO after training Lero and MRQO on the 50%, 75% and 100% data of each training workloads.



(a)



(b)

Figure 8: The tail latency of 50%,75%,95%,99% percentile of executing JOB and STATS test workloads in PostgreSQL, Alphajoin, Lero and MRQO.

PostgreSQL. This is due to the candidate plan generation strategy of MRQO. Although the plan comparator cannot accurately rank the best plans, candidate plans generated with good join orders as hints are generally superior to the plan explorer in Lero. Because Lero's plan exploration strategy may generate both good and very bad plans.

4.2.4. Performance of MRQO in tail queries

As depicted in Figure 8(a) and (b), we select queries at the 50%, 75%, 95%, and 99% of execution time for comparison. It is evident that for simple queries before the 75% of execution time, the performance gap between all optimizing strategies is not significant. However, for the complex queries in the tail end. In IMDB, MRQO's execution time at the 99% was 37.4 seconds, while PostgreSQL's was 54.7 seconds, indicating a performance improvement of 31.6%. At the 95% of execution time, the performance increased by 37.6%. In STATS, due to the special complexity of the dataset, the performance improvement of MRQO on PostgreSQL is mainly reflected in the complex queries at the tail end. At the 99% of execution time, the performance increased by 34%. In the STATS dataset, the ability of MRQO to optimize tail queries is also superior to Lero.

Our experiments demonstrate that MRQO has the overall performance and stability of matching existing state-of-the-art learned query optimizers, as well as the advantages of using MCTS to search for join orders and generate candidate plans.

5. Discussions and Conclusions

In this paper, we present MRQO, a new strategy for optimizing the query performance of relational databases. This strategy employs the MCTS method to achieve a balance between exploration and exploitation in join order selection. Concurrently, we integrate a machine learning model based on learning-to-rank technology into the query optimization process, training it using a listwise approach. This model is specifically designed to select the most optimal query plan, aiming to avoid inaccuracies and unnecessary cost in cost estimation during the selection process. The candidate query plans proposed by MRQO are generated based on the selected join orders as hints, and the candidate query plans include one plan generated by the original query optimizer of PostgreSQL, which alleviates the unstable results caused by the MCTS strategy. After experimental verification, our method not only demonstrates higher efficiency but also yields significant results, offering a novel perspective for query optimization.

Acknowledgments

This work is partially supported by the National University of Defense Technology Foundation (No. ZK23-15).

References

Swati V. Chande and Madhavi Sinha. Genetic optimization for the join ordering problem of database queries. *2011 Annual IEEE India Conference*, pages 1–5, 2011.

Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Ruibo Zhou, and Kai Zheng. Efficient join order selection learning with graph-based representation. *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022.

Sylvain Gelly and Yizao Wang. Exploration exploitation in go: Uct for monte-carlo go. In *Neural Information Processing Systems*, 2006.

Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *ArXiv*, abs/1808.03196, 2018.

Guoliang Li, Xuanhe Zhou, and Lei Cao. Machine learning for databases. *Proceedings of the First International Conference on AI-ML Systems*, 2021.

Anji Liu, Jianshu Chen, Mingze Yu, Yu Zhai, Xuewen Zhou, and Ji Liu. Watch the unobserved: A simple approach to parallelizing monte carlo tree search. *arXiv: Learning*, 2018.

Riccardo Mancini, Venkatesh Srinivasan, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. Efficient massively parallel join optimization for large queries. *Proceedings of the 2022 International Conference on Management of Data*, 2022.

Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 2018.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12:1705–1718, 2019.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. *Proceedings of the 2021 International Conference on Management of Data*, 2020.

Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 539–552, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581026.

Thomas Neumann and Bernhard Radke. Adaptive optimization of very large join queries. *Proceedings of the 2018 International Conference on Management of Data*, 2018.

Robin M. E. Swezey, Aditya Grover, Bruno Charron, and Stefano Ermon. Pirank: Scalable learning to rank via differentiable sorting. In *Neural Information Processing Systems*, 2020.

Tian Xia, Shaodan Zhai, and Shaojun Wang. Plackett-luce model for learning-to-rank task. *ArXiv*, abs/1909.06722, 2019.

Xianghong Xu, Zhibing Zhao, Tieying Zhang, Rong Kang, Luming Sun, and Jianjun Chen. Coool: A learning-to-rank approach for sql hint recommendations. *ArXiv*, abs/2304.04407, 2023.

Zhengtong Yan, Valter Uotila, and Jiaheng Lu. Join order selection with deep reinforcement learning: Fundamentals, techniques, and challenges. *Proc. VLDB Endow.*, 16:3882–3885, 2023.

Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. Reinforcement learning with tree-lstm for join order selection. *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1297–1308, 2020.

Ji Zhang. Alphajoin: Join order selection à la alphago. In *PhD@VLDB*, 2020.

Zhibing Zhao and Lirong Xia. Composite marginal likelihood methods for random utility models. *ArXiv*, abs/1806.01426, 2018.

Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. Lero: A learning-to-rank query optimizer. *Proc. VLDB Endow.*, 16(6):1466–1479, feb 2023. ISSN 2150-8097.