electronics

Article

Query Join Order Optimization Method Based on Dynamic Double Deep Q-Network

Lixia Ji [1,2], Runzhe Zhao [1,3], Yiping Dang [1], Junxiu Liu [4] and Han Zhang [1,*]

[1] School of Cyberspace Security, Zheng Zhou University, No. 100 Science Avenue, Zhengzhou 450001, China
[2] College of Computer Science, Si Chuan University, Chengdu 610041, China
[3] China Information Technology Designing Consulting Institute Co., Ltd., No. 1 Huzhu Road, Zhengzhou 450007, China
[4] Intelligent Systems Research Centre, School of Computing, Engineering & Intelligent Systems, Ulster University, Magee Campus, Londonderry BT48 7JL, UK
* Correspondence: zhang_han@zzu.edu.cn

Abstract: A join order directly affects database query performance and computational overhead. Deep reinforcement learning (DRL) can explore efficient query plans while not exhausting the search space. However, the deep Q network (DQN) suffers from the overestimation of action values in query optimization, which can lead to limited query performance. In addition, ε-greedy exploration is not efficient enough and does not enable deep exploration. Accordingly, in this paper, we propose a dynamic double DQN (DDQN) order selection method(DDOS) for join order optimization. First, the method models the join query as a Markov decision process (MDP), then solves the DRL model by integrating the network model DQN and DDQN weighting into the DRL model's estimation error problem in query joining, and finally improves the quality of developing query plans. And actions are selected using a dynamic progressive search strategy to improve the randomness and depth of exploration and accumulate a high information gain of exploration. The performance of the proposed method is compared with those of dynamic programming, heuristic algorithms, and DRL optimization methods based on the query set Join Order Benchmark (JOB). The experimental results show that the proposed method can effectively improve the query performance with a favorable generalization ability and robustness, and outperforms other baselines in multi-join query applications.

Keywords: query optimization; deep reinforcement learning; double DQN; search strategy; join plan

Citation: Ji, L.; Zhao, R.; Dang, Y.; Liu, J.; Zhang, H. Query Join Order Optimization Method Based on Dynamic Double Deep Q-Network. *Electronics* **2023**, *12*, 1504. https://doi.org/10.3390/electronics12061504

Academic Editor: Christos J. Bouras

Received: 2 February 2023
Revised: 13 March 2023
Accepted: 15 March 2023
Published: 22 March 2023

1. Introduction

The join order problem can be formulated as the problem of finding the best rearrangement in a multi-table join, and it is one of the most important problems in query optimization research [1]. Figure 1 shows an example of a three-table join. Firstly, we calculate the best access cost for each relationship and enumerate the costs of all two relationship joins. Next, we enumerate the query costs of two relationships based on the results of past lookup calculations and compare them with the other two relationship joins. Finally, we add a third relationship to get the best cost of the plan. For the same query, there are different join orders and, in turn, various join execution plans, which can have different execution efficiencies [2]. In join order problems, attention needs to be paid to controlling the extent of the search space to reduce the search overhead because the search space exponentially increases with the number of table relations up to N! (e.g., 2.4×10^{18} for N = 20). For connections over a certain number of relations, it is not even possible to give a reasonable [3] optimization order other than finding an efficient connection plan to control the execution overhead.

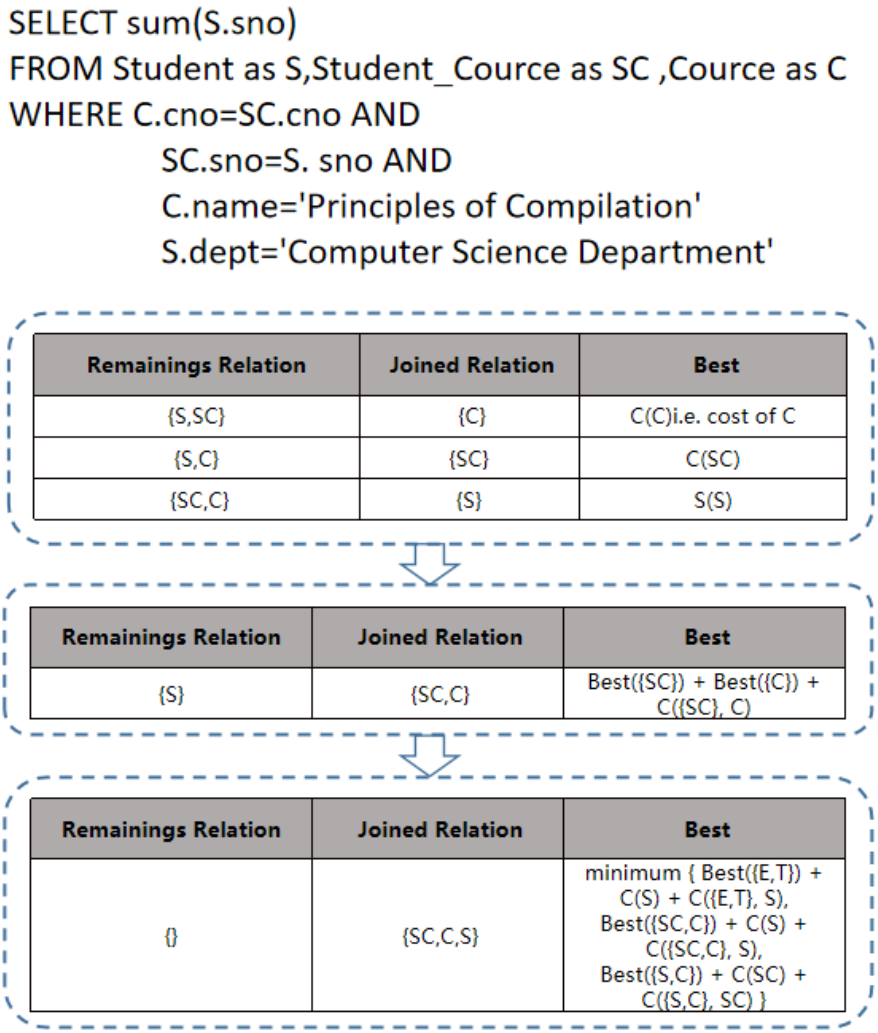

Figure 1. Order of joins regarding the three table relations. ($C(R_i)$ denotes the cost of coming to access the basic relation R_i, and $C(R_i,R_i+1)$ denotes the cost of R_i and R_i+1 joins).

The fast and accurate formulation of efficient query plans is the key to improving database system performance and query efficiency. Cost-based multi-table join query order selection in traditional databases is dominated by dynamic planning methods [4,5] and heuristics [6–8] based on the core idea of dynamic planning methods to reduce the size of the search space by merging redundant states. However, the search space is still close to the exhaustive global space, so it is suitable for queries that contain few relations. The heuristic join approach, which usually performs a heuristic search of a portion of the state space through a specific search strategy, refines the search space of potential query plans compared to traversing the search space. The latter approach not only reduces the time overhead spent on optimizing queries but also decreases the chance of finding the best query plan in the search space [9].

With the evolution of deep learning (DL), researchers have also attempted to apply DL models to query optimization work [10–12], but such methods require ample training to have a positive impact on the query performance and perform poorly on untrained datasets. To address the problem of optimizers having difficulty learning from their mistakes, deep Q network (DQN)-based deep reinforcement learning (DRL) methods have been applied to the sequential optimization of join queries. With the feedback from execution results, the join optimization is tuned to obtain good performance, and the training effect of achieving subjoins can be extended to large joins [13]. The DRL approach, despite being

able to demonstrate significant results in most cases from solutions that optimize the join order [9,10,13], cannot accurately approximate the Q function due to the limitations of the DQN model itself. Moreover, it encounters an overestimation problem in calculating action Q values, leading to large errors in the estimation. Consequently, this condition limits the model's accurate evaluation of the join plan and leads to the generation of poor plans. The model poorly performs in the training tail, requiring more resources to be spent in the training phase. Double DQN (DDQN) has two different sets of parameters to separate action selection and policy evaluation, thus solving the overestimation problem of Q values, but the presence of low estimates produces bias [14]. Therefore, we believe that combining DQN and DDQN using appropriate weights can overcome both problems and simultaneously address the overestimation of DQN and the underestimation of DDQN. In this study, we use weighted DQN and DDQN to train the DRL model, balance the overestimation and underestimation problems in DQN and DDQN, and improve the performance of queries by guiding the selection of the join order with query data as the driver. Then, the dynamic progressive search strategy was used instead of the greedy search strategy to solve the training exploration depth problem.

The main contributions of this study are as follows: (1) A dynamic DDQN join order optimization method is proposed. The method first models the join query as a Markov decision process (MDP), then solves the DRL model by integrating the network model DQN and DDQN weighting into the DRL model's estimation error problem in query joining, and finally improves the quality of developing query plans. (2) The method uses a dynamic progressive search strategy to select actions that can estimate information gain (IG) for network uncertainty data, which can accelerate the learning speed and achieve effective potential exploration. (3) Experiments in the Join Order Benchmark (JOB) show that, compared with other methods, the DDOS method produces a join query plan that not only reduces the optimization delay time but also improves the quality of the join query plan.

2. Related Work

2.1. Join Order Optimization

The join operation is a common type of query in relational databases, where the number of joined tables usually does not exceed 20. However, in practice, there may be dozens or even hundreds of joined queries, and existing methods generally rely on the size estimation of the intermediate results of the query [15]. This characteristic usually affects the optimization of the plan to some extent to reduce the latency time of developing a plan.

The MySQL database incorporates heuristic optimization in the cost-based optimizer [16] when implementing the query optimizer. The Oracle optimizer is also based on built-in rules for heuristic query optimization, which will remain throughout the query optimization phase, and the process requires considerable tuning and maintenance works [17]. In System R, the query optimization uses dynamic planning [4] by parsing the plan structure into a left-deep join tree form. However, the above approach is usually limited by the cost model, which can lead to missing plans that produce poor query quality under a nonlinear cost profile [18]. The above approaches do not provide feedback from the actual execution of query join plans in the past, which results in the optimizer not being able to obtain valid information from past experiences.

To address these issues, some researchers have applied different reinforcement learning (RL) algorithms to join order optimization [9,10,19,20]. These studies radically reduced the algorithm overhead and improved the query performance using feedback from past query executions. Ortiz et al. [21] proposed to incrementally learn the state representation of subqueries by training a model to predict the base estimate of the join query plan and improve the join order with RL. However, by selecting the query size as a cost function, query exploration has limitations that lead to suboptimal global query plans. Marcus et al. [10] proposed a neural optimizer (Neo) that iterative searches for the execution plan with the least relative overhead using a best-first search algorithm. However, it is difficult to

generalize the optimizer across different databases. Later, they proposed the join order enumerator ReJOIN [8] to learn the selection strategy of join actions using a policy gradient algorithm. First, all joins are completed, and then the entire join plan reward value is calculated based on the cost model. This strategy requires a large amount of data for training, and the join query plan is highly influenced by cost estimation. Trummer [19] et al. used an adaptive query processing strategy that does not maintain data statistics and does not use cost or base models. Krishnan et al. [13] used a DQN model to optimize the join order, which can achieve better performance than traditional methods using less data for training. However, the encoding used does not capture the structural information of the join tree. Guo et al. [20] proposed the introduction of display representation self-join encoding and beam search into join order optimization to improve model robustness and reduce the flaws of the model itself. However, such a model encounters overestimation, leading to the generation of poor query plans. Macdonald et al. [22] Proposed a novel framework, which can use the predicted execution time of various query rewrites to select alternatives based on each query to ensure effectiveness and efficiency.

2.2. DRL

As DL and RL methods continue to mature, in the past few years, several studies have focused on integrating the two methods to solve different problems [23–25]. Kipf et al. [26] Proposed a multi-set network model, which uses SQL to build neural networks and codes for table information, and finally, aggregate information to generate cardinality. Marcus [10] also proposed a learnable query optimizer, which also uses reinforcement learning to generate plans to be executed, and proposed a plan search method using the minimum heap, which has both the richness and efficiency of search. The DQN model proposed by Mnih et al. [27] is a pioneering work in DRL that learns the Q-matrix using a deep neural network. Two major mechanisms, i.e., experience replay (learning memory bank) [28] and target network, are introduced to alleviate the problem of up- and down-fluctuations in convergence when the neural network approximates the Q function. Because deep neural networks provide rich high-dimensional representations for end-to-end learning, the combination of the two enables RL (Reinforcement learning) to handle some action decision problems in high-dimensional state spaces. However, some researchers have argued that the integration of online RL with deep neural networks is unstable [23,29,30]. Hasselt et al. [23] proposed a DDQN algorithm to reduce overestimation. However, DDQN sometimes suffers from the underestimation of action values. To reduce training data while ensuring sampling efficiency during RL learning, Mahajan et al. [31] proposed a new framework combining symmetry detection procedures and symmetry implementation mechanisms to discover symmetries in the environment and apply them to the function approximation process to speed up the symmetry strategy learning process. In addition, the number of actions in a DQN with a simple discretization process exponentially grows with increasing degrees of freedom in a continuous domain of action, which leads to an action space that can then be too large and extremely difficult to converge. Therefore, Lillicrap et al. [32] proposed a model-free algorithm, DDPG (Deep Determinist Policy Gradient), based on DPG (Deterministic Policy Gradient) and AC (Actor-Critic) by applying the ideas of DQN to the continuous action domain; this algorithm can be applied to substantially complex problems and networks. However, compared to DDQN, a slow change in the respective network parameters of the actor and critic is needed to update them, and hence, substantial training is required for the algorithm to work with connected sequences.

DRL-based query optimizers [13,20,33] develop connected query plans entirely based on the output of the underlying neural network model. Therefore, the performance results are limited by the DQN model used. Although some works have made improvements to the model to enable good performance, DQNs are fundamentally unable to avoid the overestimation problem. Moreover, there are fluctuations in the training tail, which in turn affect the convergence results and the quality of the tail query plan. In summary, based on the current work on join order optimization and the progress of research on DRL

models [13,23,27,33], fusing DQN and DDQN models to optimize the join order in queries can further improve the performance of join queries.

Therefore, in this study, we use a combination of weighted DDQNs and dynamic search strategies to optimize the join order and effectively address the impact of the overestimation of DQN and the underestimation of DDQN on the query performance. Moreover, we use a progressive greedy strategy to de-train the connected actions. This method enables intelligence to learn the strategy precisely and improve the efficiency of mining actions in the training phase, which in turn is close to the optimal plan.

3. Methods

The main function of the query optimizer is to improve the execution efficiency of SQL statements. The optimization process is divided into two phases: logical query optimization and physical query optimization. The DDOS method proposed in this paper is mainly performed for the physical query execution plan generation phase of the optimizer, which outputs a predicted join plan for a given SQL query by guiding the selection of the join order. The framework of the method is shown in Figure 2. Although our methods look similar to DDPG, they are completely different approaches. The difference between our approach and DDPG is that DQN and DDQN have two sets of networks of their own, and there is no need to add a new neural network to configure the weights, while DDPG adds a Policy network and its Policy_target network to DQN to output a continuous value.

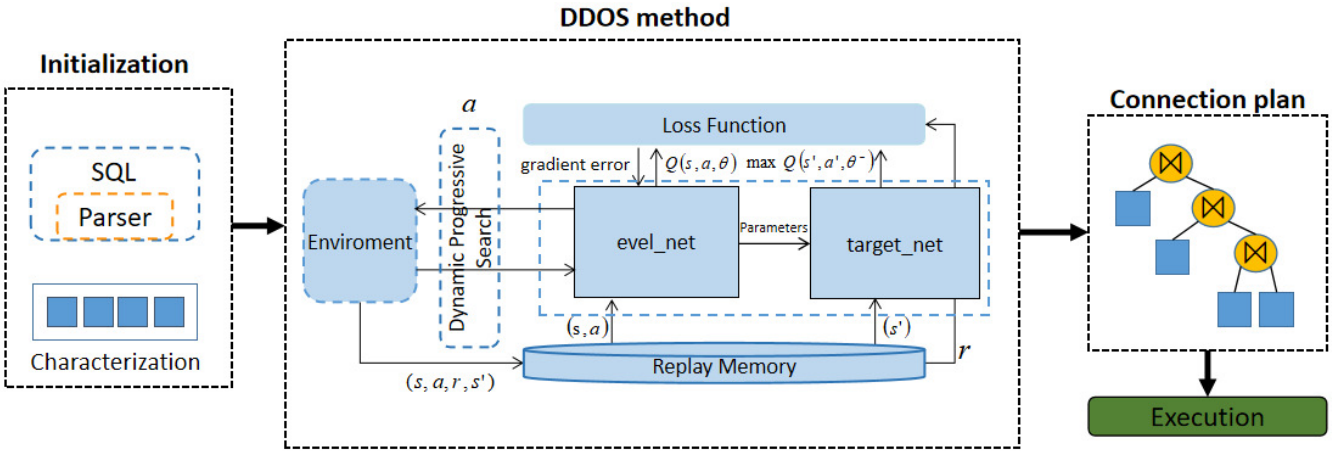

Figure 2. Workflow for developing a query plan. (The most important part is to adjust the network parameters through DDOS training to obtain the connection plan with the minimum cost and the shortest execution time. Where (s, a, r, s') represents the reward r and the next state s' obtained by taking action a in state s).

After the SQL statements are inputted, they are firstly vectorized to complete the initialization, a process that does not require the addition of additional information. Then, the join order problem is formally expressed as a series of definitions of the value-based RL, which is modeled as an MDP. Q-learning is used to solve the Markov problem of the join order. In the MDP model, the intelligence starts from the initial join state. By interacting with the environment, the evaluation network eval_net obtains the sample data of the reward value, which is used to update the evaluation network. After a fixed number of time steps, the parameters of the evaluation network are copied to the target network target_net. The target value network calculates the action value of the maximum Q value in the current state, selects the action of the join to act on the environment according to dynamic progressive exploration, and gets the new state. This process is repeated to produce the final join plan and complete the query execution plan by adding plan nodes before handing it over to the executor. When using the gradient descent function to update the evaluation network, a weighting approach is used to balance the DQN and DDQN

network parameter weights to effectively reduce estimation errors and improve neural network stability.

3.1. State and Action Characterization Representation

Expressing the join order as a problem that can be solved by DRL requires encoding the database and query information as inputs to the neural network. In this study, a vectorized representation is used for query encoding, In the design of the intelligent body in the selection of actions, the representation method of the tree structure is used, where each join query can be represented by a defined query tree. The table-to-table join is defined as joining to the join tree denoted by α and the obtained join tree state denoted by s. The leaf nodes are defined as the tables joining the join. A query is then a binary tree, and each table join makes the tree of the corresponding relationship inserted into a leaf node. The leaf nodes corresponding to r_i and r_j are removed from the action selection after they are inserted. When a new leaf is added, it is joined first with the subtree corresponding to (r_i, r_j). This method preserves the information about the subjoin structure between tables.

Each table to be joined is treated as a join candidate, and the full combination of all the candidate joins is used to join the table to be joined. When the total number of tables in a database is n, an n-dimensional vector action selection is used to express the action space. During the joining process, the vector value is changed for the sub-tree corresponding to the join. When the table is joined into the join, the action value in the sub-tree of the action value at the corresponding position is changed to $\frac{1}{h(n,i)^2}$. The value is 0 if the corresponding connected table is not joined in the subtree, and h is the height value at the time of subtree relationship i in the join.

As shown in Figure 3, in state s3, row 3 corresponds to the representation of the join (l $\bowtie$ o) $\bowtie$ c, where the o table corresponds to a value of 1/4 in row 3, the height of the subtree in which it is located is 2, and the n-no join corresponds to a value of 0. We compare the data characterization methods of existing learning optimizers (e.g., DQ [13]). Each column of the database is used as a single feature of the database. A state is represented as a binary one-hot vector, where each number represents a column of the database. This action behavior effectively preserves information about the hierarchy between tables, facilitates capturing information about the structure of the join tree, reduces the cost of the optimizer in cost estimation, and allows the combination to be extended to multi-table joins with a large number of joins after training data containing local joins. In this study, multiple 1-hot [24] is also incorporated for implementing self-join query encoding. The representation of the left-right relationship, physical operation, and intermediate relationship size regarding the join action is $f_c = A_L \bigoplus A_R \bigoplus ph \bigoplus c$. A_L and A_R denote the left-right relationship using n-dimensional vectors, ph is the physical join operation using one-hot encoding, and c is the base estimate of the join intermediate relationship.

3.2. DDOS

In this work, a new model network structure is constructed, and the exploration–utilization strategy in the training phase is improved. Both works are described below:

3.2.1. Model Construction

In the standard DQN network, the selection and evaluation of actions are based on the same parameters, which produces the overestimation of Q values and causes the DQN to overestimate action values. Haselt et al. proposed a dual DQN [22], where the estimation strategy is performed according to the online Q network and the selection of actions is performed using the target network for the estimation of Q values. This method can substantially alleviate the overfitting problem, but it also results in over- and underestimated action values, which affect the learned strategy and hence cause performance reduction. In the logic optimization stage, to reduce the negative impact of the overestimation problem of DQN and the underestimation problem of DDQN, this study applies the concept of a weighted dual estimator [13] in dual DQN to build a network that can accurately filter

and improve the performance of the current query plan by calculating the "action–state" pair. The neural network is trained to regress the Q value by calculating the "action–state" pair, which reduces the error of the target estimate and achieves an accurate estimate of the Q value by balancing the weights of the DQN and DDQN while retaining the DQN and DDQN. The vectorized query plan tree, through the input layer of the neural network and the state information of the two hidden layers, is fed to the fully connected linear layer and then the mapping vector to the prediction of the performance cost using the rectified linear unit [14] activation function and layer normalization, as shown in Figure 4.

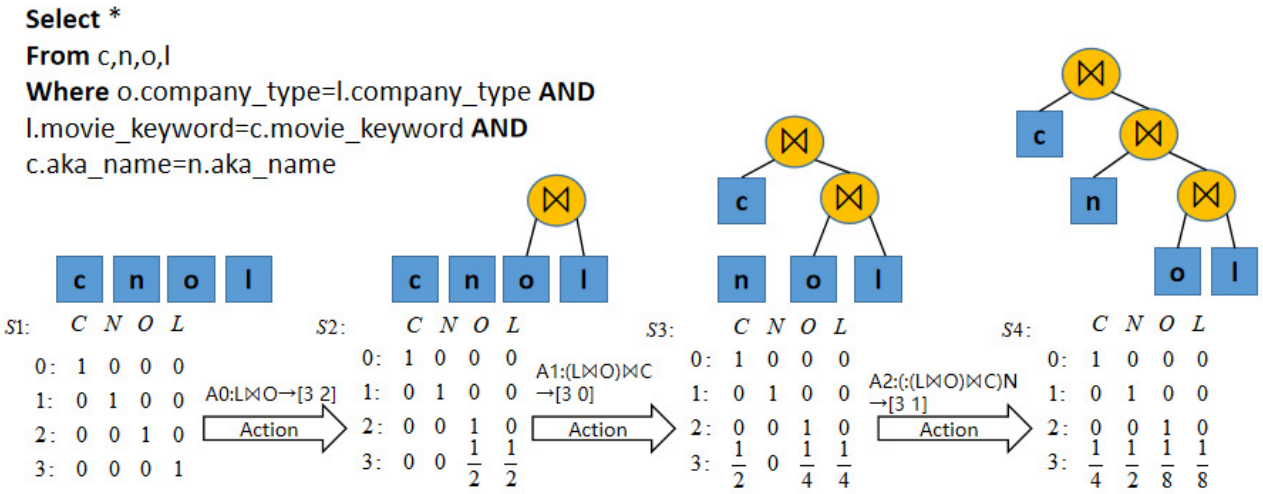

Figure 3. Action vector representation process.

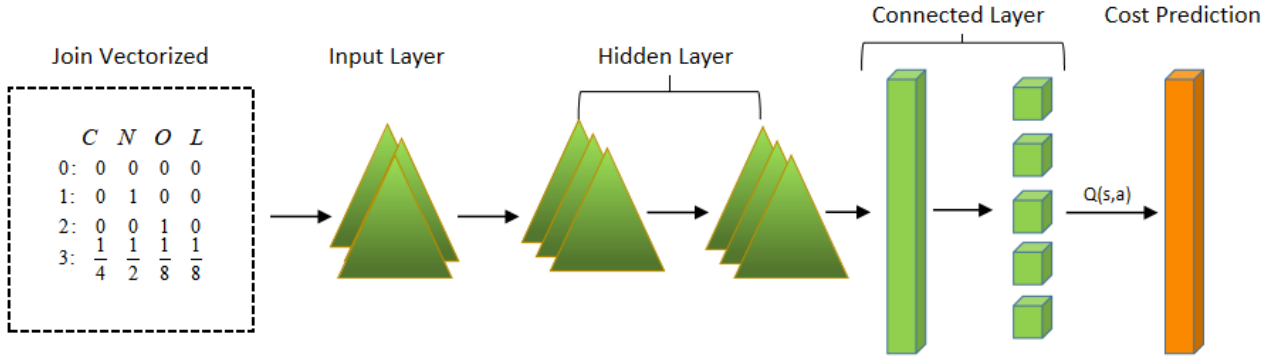

Figure 4. Architecture of the query cost prediction model (input to the network after obtaining the SQL vector, join on the query plan tree, and output of the long-term plan cost at a fully connected layer after network training).

In this study, the structure of the dual DQN network is used, including only the evaluation network and target network. The evaluation network aims at mining the actions corresponding to the maximum values, whereas the target network is responsible for the evaluation of the maximum action value. Initializing the vector encoded in the previous subsection to state s and inputting it into the DDOS network. The target value is calculated using a linear relationship between the compositions of $Q(s', a^*; \theta)$and $Q(s', a^*, \theta^-)$, which is calculated as follows:

$$y^{WDDQN} = r + \gamma\left[\beta Q\left(s', a^*; \theta\right) + (1-\beta) Q\left(s', a^*; \theta^-\right)\right] \quad (1)$$

where β is the weight; action selection (i.e., action value) is calculated by weighting the evaluation network and target network, where the range of β belongs to [0, 1]. When β takes the value of 0, the network is equal to the DDQN network, and when β takes the

value of 1, the algorithm completely ignores the DDQN evaluation and uses only the DQN network to select the action. The weight β is calculated as

$$\beta = \frac{|Q(s', a^*; \theta^-) - Q(s', a_L; \theta^-)|}{c + |Q(s', a^*; \theta^-) - Q(s', a_L; \theta^-)|} \tag{2}$$

The action a^* is denoted as the action with the maximum action value for the evaluation network, and a_L denotes the action with the minimum action value for the evaluation network, calculated as

$$a* = argmaxQ(s', a; \theta) \tag{3}$$

$$a_L = argminQ(s', a, \theta) \tag{4}$$

Therefore, $Q(s', a^*, \theta)$ denotes the maximum action value in the target network in state s', and $Q(s', a^L, \theta^-)$ denotes the minimum action value in the target network in state s'. c in Equation (2). is the hyperparameter used to calculate the weight β. When performing the calculation, the optimal value of c differs in problems with different characteristics, and the difference of hyperparameter c affects the network assignment weights. In the experiment, this study sets the size of the hyperparameter according to the join task characteristics and selects the value of c based on multiple experiments.

In solving the query optimization, the traditional bottom–up optimization idea is used to incorporate the long-term reward for performing the join action into the Q-value, and the correspondence between the action used in a certain state and the Q-value is obtained recursively. In learning the Q-value function, the loss function is defined as the difference between the estimated Q-value and actual Q-value, and the Q-value is updated after the action is selected using the $E_{dp}(a)$ dynamic progressive search strategy with the following equation:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)], a \in E_{dp}(a) \tag{5}$$

where s and a are defined as presented in Section 3.1, a is the next action to be executed after action a, and s is the state after state s executes action a. The traditional optimizer cost model is sometimes biased. However, its time cost is low, and it is still a good choice for training neural network models. The execution query action is obtained according to the cost model. The corresponding reward r normalizes the reward range and sets an upper limit on the reward range. Any reward value above the estimate is 0. γ denotes the learning rate, $max_{a'} Q(s', a')$ denotes the destination Q value, and $Q(s, a)$ denotes the estimated Q value. The state is calculated by the mean square error when updating the Q value, i.e., the mean square error $L(\theta)$:

$$L(\theta) = E\left[\left|\left|r + \gamma max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)\right|\right|\right] \tag{6}$$

3.2.2. Dynamic Progressive Search Strategy in Training

The ε-greedy strategy incorporates random action selection with a certain probability. According to the role of the parameter ε in the greedy strategy in RL, using a fixed value for each update will make the intelligence less adaptable, and in complex scenarios, it is difficult to achieve exploration far from the long distance with a constant size of ε. A low value of ε will reduce the exploration speed of the network, whereas a high value of ε will exponentially decrease with the increase of the distance from the initial state [21]. Hence, the exploration is far from reaching the deep exploration and increases the possibility of causing a local optimum. To avoid meaningless exploration, it is first necessary to evaluate the uncertainty as the new states of unexplored, uncertain, and unexplored actions generally have high IGs. During exploration, this part is used as the focus of the training period [34], and noise is added to increase the randomness of the execution at each moment of the exploration to achieve an efficient exploration environment. The dynamic progressive

search strategy is divided into three cases of action selection according to the value of ε. In the first case, the action with the maximum Q value is selected for updating. In the second case, an action is randomly selected in the current state. In the third case, the action selected by the softmax function is used.

$$E_{dp}(a) = \begin{cases} a = argmaxQ(s,a), \varepsilon > rand() \\ a = R(s), \varepsilon < rand() \\ a = argP(s,a), \varepsilon = 0 \end{cases} \tag{7}$$

where $argmaxQ(s,a)$ denotes the action with the maximum Q value under state s, $rand()$ is a random number in the range (0,1), $R(s)$ denotes a randomly selected action under state s, and $argP(s,a)$ denotes the action corresponding to $P(s,a)$. The process of setting the ε value from 1 to 0 enables the exploration to exploitation phases. Based on the number of training steps, the proportion of high and low rewards is used to dynamically adjust the parameter ε to achieve the dynamic switching between stages. This method leads to the design of an expression that can control the change in parameter ε as follows.

$$\varepsilon = 1 - u_1\left(\frac{i}{K}\right) - u_2\left(\frac{K_h}{K_t}\right) + u_3\left(\frac{K_l}{K_t}\right) \tag{8}$$

where K is the number of training steps, K_t denotes the number of training rounds, K_t denotes the number of episodes with the sum of reward values greater than R during training, K_l denotes the number of episodes with the sum of reward values less than R during training, and R is the bound used to distinguish the training effect. u_1, u_2, and u_3 denote the weights of each influencing factor. In the early stage of training, the value slowly decreases, but as the training progresses, the ε value decreases faster with the gradual increase of K_h in each round. After the ε value is reduced to 0, a short period of the softmax-based action strategy training is added, with the following equation:

$$P(s,a) = \frac{\exp(vQ(s,a))}{\sum_{a \in A} \exp(vQ(s,a))} \tag{9}$$

where v is a scalar and $P(s,a)$ is the probability of action a being selected under state s. As learning proceeds to this stage, each action has a different payoff value, and as learning progresses, the system prefers movements with high return values, improving the phenomenon that sub-optimal movements are also selected later in the training.

3.2.3. Training Algorithm

As the cost model can quickly give the cost to be estimated in a short period of time, the calculated cost is still referable, although there is a partial deviation from the actual delay in terms of computational accuracy. In addition, to alleviate the estimation bias brought by the cost model, in this study, the training of execution delay is added. Each query execution takes a long time to get actual feedback. Therefore, we only reset the network weights of the output layer after the cost training and combine the data collected from the previous cost model training as a guide using the execution delay for fine-tuning, after a small amount of data (the actual execution of the query with running time) are retrained. The specific training process is shown in Algorithm 1.

Algorithm 1: Weighted Double Q-learning.

Input: SQL query statements, initialize Θ, Θ^{-}
Output: Θ

initialize the evaluation network parameters, target network parameters, initialize replay memory D;
for *episode = 1 to T* **do**
  initialize the initial state s_0;
  for *t = 0 to T* **do**
    select actions based on dynamic progressive search strategies a_t;
    take action a_t, observe s' and r;
    store transition(s_t, a_t, r_t, s_{t+1}) in D;
    sampling of n batches of transfers for updating;
    $a^{*} = \arg\max Q(s', a, \Theta)$;
    $a_L^{*} = \arg\min Q(s', a, \Theta)$;
    calculate weights β;
    calculated target values:
    $y^{WDDQN} = r + \gamma[\beta Q(s', a, \Theta) + (1-\beta)Q(s', a, \Theta)]$;
    update Θ, SGD$(\Theta \leftarrow y_j^{WDDQN}, Q_j)$;
    copy parameter Θ to Θ^{-} for every C time step elapsed;
  end
end

4. Experimental Results and Analysis

This section will verify the superior performance of the DDOS method in query join order optimization through the following four experiments.

(1) First, the rewards obtained per query on average during the training phase are compared with methods, such as DQ, to verify the performance of DDOS during the training process and to explore the effect of hyperparameter c on the DDOS method.
(2) We compared the efficiency of DDOS with different optimization methods for generating plans with a different number of connection relations, i.e., the time required to generate query plans (connection latency time), and verified to what extent the model reduced the optimization latency while guaranteeing performance.
(3) The impact of the amount of training data on the DDOS formulation of connection plans is compared to explore the efficient performance of dynamic progressive search strategies.
(4) Finally, the execution efficiency of the model development query is analyzed, and the execution cost and cost distribution of the connection query plan are compared with the baseline method.

4.1. Experimental Setup and Baseline Model

In the dataset selection, no benchmarks, such as TPC-H, TPC-DS, or Star Schema Benchmark, which can evaluate the value in terms of query engines, are chosen, but they all use the same simplifying assumptions (uniformity, independence, and inclusion principle). To satisfy the data authenticity, the JOB [2] benchmark is chosen using the Internet Movie Database (IMDB), which contains a large amount of information about movies, release dates, actors, directors, production companies, and other related information. The data include a large number of connected table information, which puts 33 query structures into variants, with two to six variants in each structure, a total of 113 queries containing 21 tables, query joins from 4 to 17, and a data file size of 3.6 GB. Such a dataset has correlation and non-uniform data distribution.

The experiments are run under a server with a 4-core Intel Core i7-6700 CPU (3.4 Hz), running memory of 8G, and GPU set to OFF. In Apache Calcite, the NN of DDOS is deployed

through the DL4J library, and in PostgreSQL, the TensorFlow C API is used to call the NN. DDOS is responsible for performing the join order part of the query plan, and the remaining work (e.g., index access and fusion sorting joins) is performed in PostgreSQL. The parameter configuration is as follows: 20,000 training cycles, the learning rate of 2.5×10^{-4}, hidden layer of 256, 30 min of costly model training, 2 h of latency fine-tuning, and best performance with hyperparameter c of 10. The baseline models used in the experiments are as follows: (1) DQN-based: DQ [13], (2) DDQN, (3) Rejoin [9], (4) Postgres [5], (5) pruning classes: left-deep and zigzag tree pruning [35] (zigzag, ZZ), (6) exhaustive, and (7) QuickPick-1000 (QuickPick-1000 can quickly give a connection plan, and because it does not restrict the search space, it may give a poor plan).

4.2. Experiments and Results

4.2.1. Average Rewards for Different Parameters

First, this experiment compared the performance of DDOS with other DRL methods in terms of average reward values to validate the rationality of the connection scheme developed during training. The experiment chose to evaluate the average reward of hyperparameters at 1, 10, and 20 and compares it with the DDQN and DQ of existing DRLs. Figure 5 shows the average reward value of DDOS with different hyperparameter values c set versus DDQN and DQ with an increasing number of epochs compared with the reward value (the reward value is the total reward sum of the target query task at a time). There is a large performance difference among DDQN, DQ, and the average reward of DDOS with different hyperparameter c values. The average reward values are 5.10, 4.49, 5.76, 6.27, and 5.86. Although DDOS with different hyperparameter c values showed better performance, the average reward value of 6.27 obtained for c = 10 was better than the c values of other methods, which indicates that DDOS with c = 10 has the most effective performance. The reason is that the c value affects the weight of the network, which promotes the update of the network parameters and improves the accuracy of the neural network, which in turn enables a higher reward. In addition, DDOS converges and stabilizes in both training tails.

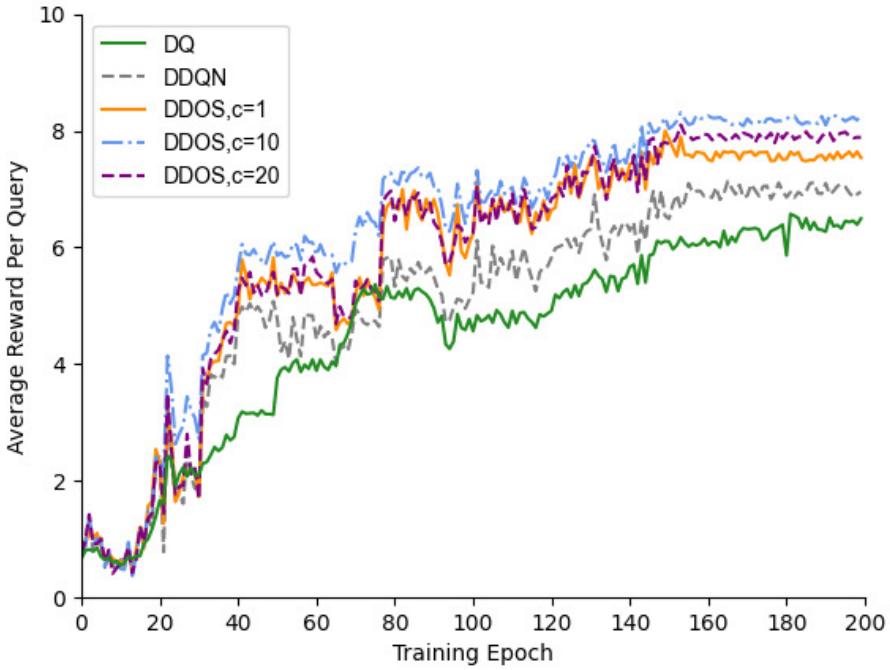

Figure 5. Comparison of the average reward value of DDQN and DDOS under different parameters (higher average reward value means a smaller and more reasonable query join cost).

4.2.2. Comparison of the Join Delay Time

In this experiment, we compared the latency time of this model with other baseline methods for JOB queries in Apache Calcite to verify the performance of DDOS in formulat-

ing execution plan efficiency for a different number of joins. Analysis of the 6 curves in Figure 6 shows that in the case of a small number of joins to the table, Postgres used the dynamic planning (left-deep) method with a shorter latency time, while the advantages of the DDOS algorithm were not obvious, because the network computation overhead increases the cost of queries planning and more pre-exploration, and the latency time was slightly higher than DQ when the number of joins was below 6. The search space corresponding to a smaller number of tables was also very small, and there is no significant difference in the execution efficiency of DDOS. However, as the number of joins increased, the plan search space grew exponentially and the computation time of dynamic planning as well as heuristic methods grew significantly. However, DDOS shows a clear optimization effect, and the more joins, the more significant the optimization effect. The average connection latency time of DDOS, when the number of joins is greater than or equal to 8, was better than all methods except QuickPick-1000. This is because QuickPick-1000 randomly tests 1000 possible plans after a given query and selects the lowest cost plan. This greatly reduces the search time, but the quality of the subsequent resulting plans is often difficult to guarantee. While DDOS learned based on the feedback from the execution, this ensured the quality of the generated plans. Throughout the experiment, the average DDOS connection latency was improved by 21.7% over DQ compared to the DRL method. Mainly because DDOS uses a weighted dual-Q network to train the model with higher accuracy, and it uses an asymptotic search strategy to train to a later stage to avoid many invalid explorations, thus reducing the training time of the tail.

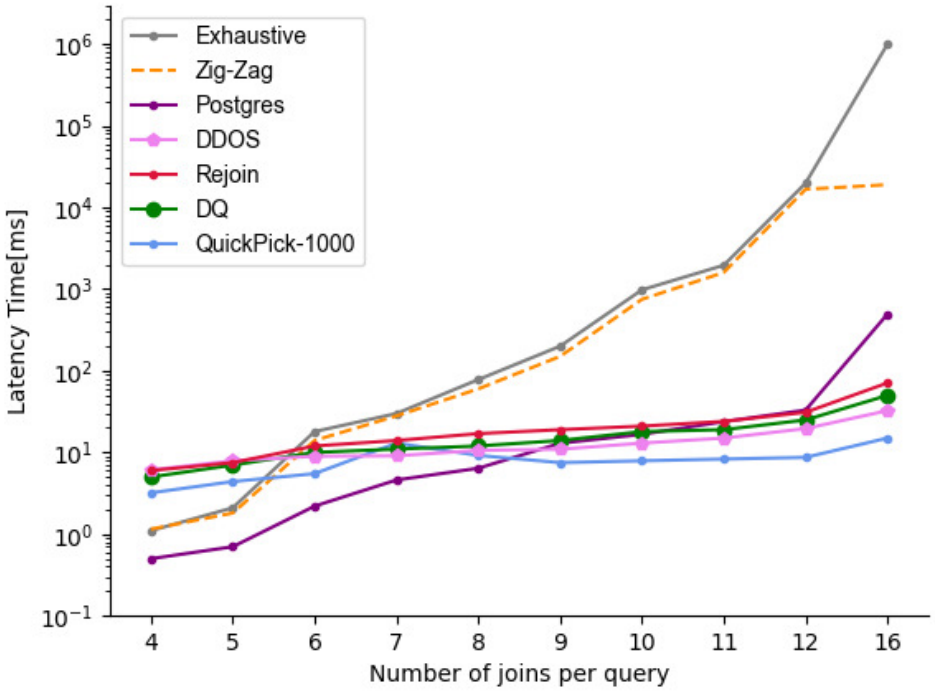

Figure 6. Comparison of the query plan delay times for different optimization methods.

4.2.3. Comparison of the Query Convergence Speed

To verify the effect of dynamic search strategy exploration on query convergence speed, the experiments used cross-validation methods to explore the relationship between query data and query plan cost. The average relative cost of DDOS, Rejoin, and DQ methods with a different number of query groups was used as a reference for the change of query cost. Setting the cost of the QuickPick-1000 method as a baseline, the number of training queries was roughly 25 when DDOS could reach the relative cost of the QuickPick-1000 plan. In addition, DDOS scaled to more join tasks by learning a combination of local structures. As the training queries increased, the local join coverage gradually expands and was able to

improve the performance during multi-join tasks. When the number of training queries reaches the range of 50–80 training can make the queries perform better. As can be seen from Figure 7, DDOS converged the fastest and stabilized at 70 groups, after which it tended to level off and the algorithm has high stability. Rejoin and DQ were also able to stabilize faster as the number of queries increased, and had a substantial increase in performance over QuickPick-1000, but there is still a gap compared to the relative cost of DDOS. DDOS was able to improve the connection performance better with fewer questions, which means that the method can be extended to more connection tasks by learning a combination of local structures. DDOS query plan cost is 13.48% less than DQ and 17.55% less than Rejoin. As the number of training groups increases, the local connection coverage gradually expands, and then it can improve the performance in multi-join tasks. In Addition, the dynamic asymptotic strategy exploration can control the jump of greedy parameters, which enhances the efficiency of exploration in the early stage of model training, while avoiding getting a local optimal plan as much as possible, and converging and stabilizing more rapidly in the late stage of training.

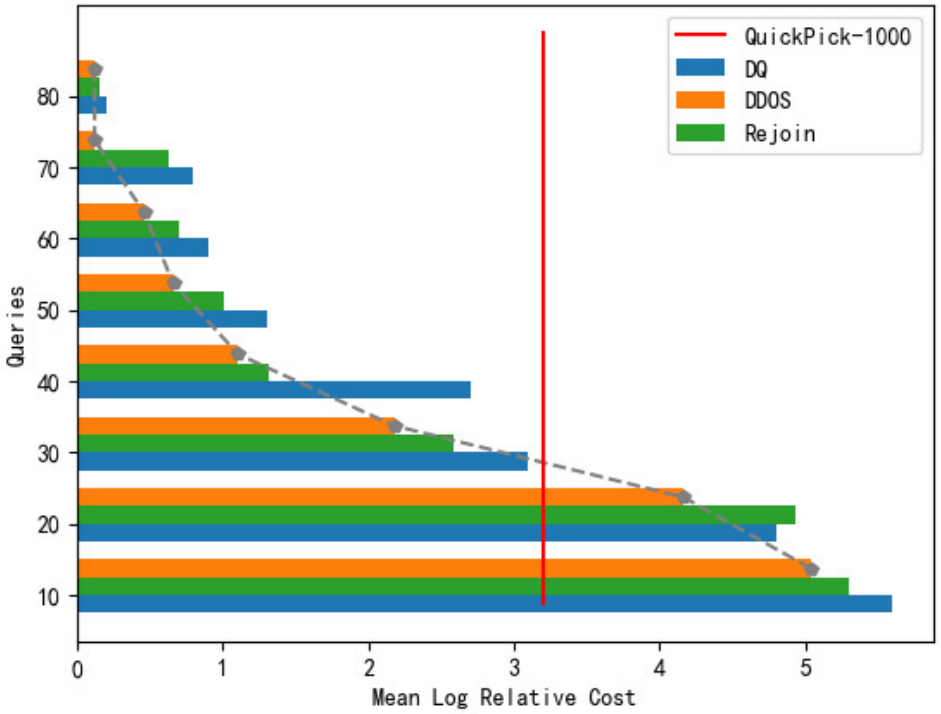

Figure 7. Comparison of the mean relative cost (in log scale) of DDOS, Rejoin and DQ.

4.2.4. Comparison of Relative Implementation Costs

In this part of the experiment, to verify that the DDOS method generates plans that can be executed in less time with higher execution efficiency. We divided the JOB queries into training and test sets, and after training, we counted the mean relative execution cost (MREC) of the DQ, DDQN, Rejoin, QuickPick-1000, and DDOS method generation plans in the test query set and obtained the distribution of the cost. To make the results more intuitive, the exhaustive method (which traverses all possible query plans and can produce near-optimal plans) is chosen as the baseline. The geometric mean (GM) of each method is counted, where is the cost estimate of the query, is the cost estimate of the query using the exhaustive method, and n is the number of query sets.

$$GM = \sqrt[n]{\left(\prod_{i=1}^{N} \frac{Cost(qry)}{Cost_{EX}(qry)}\right)} \tag{10}$$

The execution cost statistics are obtained after delayed collection and training. As shown in Table 1, the GMs of the execution cost of DDOS, DDQN, DQ, Rejoin, and QuickPick-1000 in the JOB query are 1.46, 1.82, 2.15, 2.58, and 2.99, respectively. The GM of DDOS is 19.7% lower than that of DDQN and 32% lower than that of DQ. In addition, the median and quantile values of its relative execution cost are lower than the other DRL methods. The results indicate that the lower execution cost of the join plan developed by DDOS indicates that the quality of the join plan is improved. The reason is that the tree-structured representation provides necessary and richer state information and uses a dynamic and progressive search strategy to reduce the possibility of missing the better plan.

Table 1. Relative execution cost statistics of DDOS, DDQN, DQ, Rejoin, and QuickPick-1000.

Mean Execution Cost / Method	GM	Median Value	Quartile	
			1st Quartile	3rd Quartile
QuickPick-1000	2.99	2.99	5.20	2.59
Rejoin	2.58	2.03	3.65	1.84
DQ	2.15	1.70	2.93	1.56
DDQN	1.82	1.49	2.24	1.35
DDOS	1.46	1.15	1.99	1.08

As shown in Figure 8, in some cases, DDQN and DQ could produce better-quality plans. However, the average relative cost maxima and minima of DQ vary greatly, with nearly 47% of the query execution costs being higher than DDOS, while the average and maxima of DDOS costs were lower than DQ. In addition, the number of outliers was significantly less for DDOS than for DDQN and DQ, which means that DDOS produced a more accurate action estimation and a more extensive action mining strategy produced a better query plan and a more stable query plan was obtained.

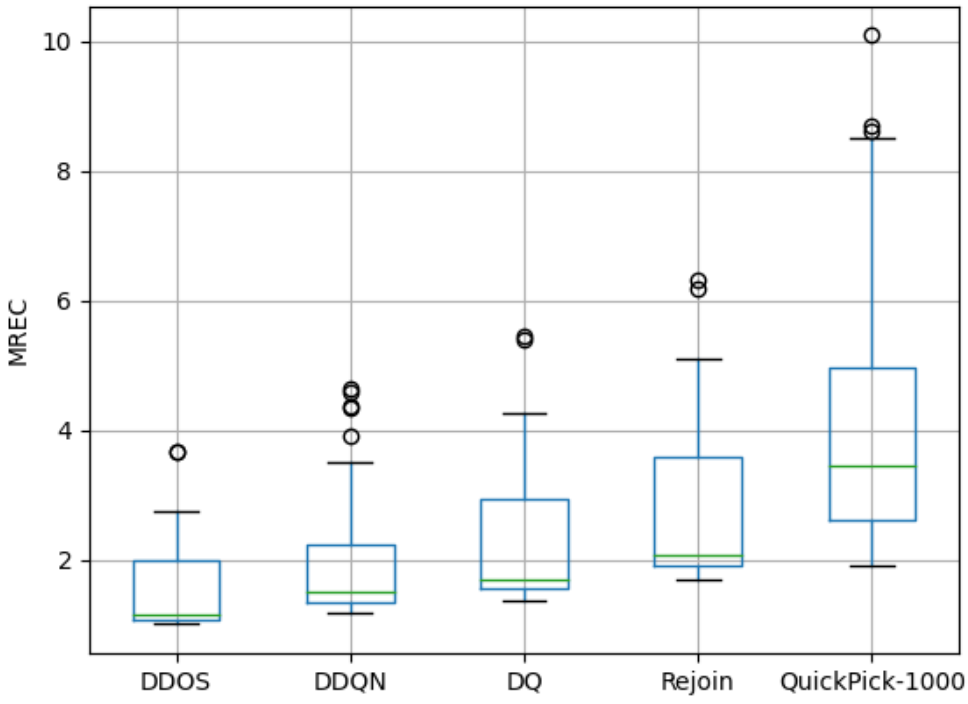

Figure 8. Relative execution cost of the DDOS, DDQN, DQ, Rejoin, and QuickPick-1000 queries.

4.3. Discussion

In the experiments, first, our approach was explored in terms of different hyperparameter performances and compared with two DRL-based approaches, which used the DQN and DDQN models to guide the connection order, respectively. It can be seen from the training process that our proposed DDOS obtained a higher reward value than the other two methods and was more stable at the tail end of the training (Figure 5). Then, we compared DDOS with six connection order methods, and in the delay time experimental results (Figure 6), although there was no obvious advantage in delay efficiency in the early stage when the number of joins was small, as the number of joins increases, the delay time of DDOS was lower than other methods, and the overall performance was better than all the methods except QuickPick-1000. However, QuickPick-1000 sacrificed execution performance to reduce optimization latency time. After that, we compared the query convergence speed, and using QuickPick-1000 as the baseline, DDOS with the dynamic progressive search strategy reached a performance close to that of QuickPick-1000 at a training group size of 25, which was significantly faster than that of DQ and Rejoin, and the relative cost was lower than that of DQ when the training group size reached 80 (Figure 7). In addition, in the query cost experiments, DDOS had a lower relative execution cost than the other four methods and effectively reduced the number of outliers (Figure 8). In summary, the proposed method in this paper produces good plans on the query set JOB by optimizing the join order, with good improvement in query latency and execution cost. In turn, it is demonstrated that the proposed method improves the accuracy compared with other advanced benchmark models, and also reduces the cost, and improves the training efficiency.

5. Conclusions

In this study, dynamic DDQN is used for join order optimization based on a network structure constructed by a weighted DDQN to optimize the join operation of the query optimizer. Combined with a dynamic progressive search strategy, it can evaluate a more accurate valuation of the join action and quickly learn a more efficient strategy for the query plan. The analysis of the experimental results shows that the method improves the overall performance of the DRL algorithm for optimizing delay time and execution cost compared to existing DRL methods, bringing further performance improvements for complex queries with multi-join in databases.

Based on the current work progress, there are still some future works to be studied: (1) The encoding of the current model, where the structure of subqueries is stored, is fixed and not flexible enough when extending the join, so more flexible encoding methods need to be explored. (2) There is an error estimation on the cost model, and although the error can be eliminated to some extent via delayed training, it still limits the connected query plan in achieving good performance. The cost model still has more room for improvement to provide a reliable and robust optimization basis for query estimation.

Author Contributions: Conceptualization, L.J. and R.Z.; methodology, H.Z. and R.Z.; software, L.J. and R.Z.; validation, L.J., R.Z. and H.Z.; formal analysis, R.Z.; data curation, R.Z. and Y.D.; writing—original draft preparation, R.Z. Y.D. and J.L.; writing—review and editing, L.J. and H.Z.; visualization, R.Z.; supervision, H.Z.; project administration, L.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Major Science and Technology Project in Henan Province grant number 201300210500 and in part by the Zhengzhou Major Science and Technology Innovation Project grant number 2020CXZX0053.

Data Availability Statement: Data will be made available on reasonable request.

Acknowledgments: This work was supported by the Major Science and Technology Project in Henan Province (grant No. 201300210500), and in part by the Zhengzhou Major Science and Technology Innovation Project under Grant No. 2020CXZX0053.

Conflicts of Interest: We declare that we have no financial and personal relationships with other people or organizations that can inappropriately influence our work, and there is no professional or other personal interest of any nature or kind in any product, service, and/or company that could be construed as influencing the position presented in, or the review of, the manuscript entitled.

References

1. Lan, H.; Bao, Z.; Peng, Y. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Sci. Eng.* **2021,** *6*, 86–101. [CrossRef]
2. Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P.; Kemper, A.; Neumann, T. How good are query optimizers, really? *Proc. VLDB Endow.* **2015**, *9*, 204–215. [CrossRef]
3. Dieu, N.; Dragusanu, A.; Fabret, F.; Llirbat, F.; Simon, E. 1000 Tables under the form. *Proc. VLDB Endow.* **2009**, *2*, 1450–1461. [CrossRef]
4. Snodgrass, R.T.; Currim, S.; Suh, Y.K. Have query optimizers hit the wall? *VLDB J.* **2022**, *1*, 1–20. [CrossRef]
5. The PostgreSQL Global Development Group. PostgreSQL: The World's Most Advanced Open Source Database[EB/OL]. Available online: http://www.postgresql.org/ (accessed on 16 October 2021).
6. Winslett, M.; Braganholo, V. Goetz Graefe speaks out on (not only) query optimization. *ACM SIGMOD Record.* **2020**, *49*, 30–36. [CrossRef]
7. Moerkotte, G.; Neumann, T. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. *Proc. VLDB Endow.* **2018**, *11*, 1069–1070.
8. Waas, F.; Pellenkoft, A. Join order selection (good enough is easy). In *Advances in Databases, Proceedings of the British National Conference on Databases*; Springer: Kunming, China, 2000; pp. 51–67.
9. Marcus, R.; Papaemmanouil, O. Deep reinforcement learning for join order enumeration. In Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, Houston, TX, USA, 10 June 2018; pp. 1–4.
10. Marcus, R.; Negi, P.; Mao, H.; Zhang, C.; Alizadeh, M.; Kraska, T.; Papaemmanouil, O.; Tatbul, N. Neo: A learned query optimizer. *arXiv* **2019**, arXiv:1904.03711.
11. Zhang, J. AlphaJoin: Join Order Selection à la AlphaGo, PhD@ VLDB 2020. Available online: https://ceur-ws.org/Vol-2652/paper05.pdf (accessed on 1 February 2023).
12. Park, Y.; Zhong, S.; Mozafari, B. QuickSel: Quick selectivity learning with mixture models. In Proceedings of the SIGMOD, Portland, OR, USA, 14–19 June 2020; pp. 1017–1033.
13. Krishnan, S.; Yang, Z.; Goldberg, K.; Hellerstein, J.; Stoica, I. Learning to optimize join queries with deep reinforcement learning. *arXiv* **2018**, arXiv:1808.03196.
14. Zhang, Z.; Pan, Z.; Kochenderfer, M.J. Weighted double Q-learning. In Proceedings of the IJCAI, Melbourne, Australia, 19–25 August 2017; pp. 3455–3461.
15. Färber, F.; Cha, S.K.; Primsch, J.; Bornhövd, C.; Sigg, S.; Lehner, W. SAP HANA database: Data management for modern business applications. *ACM Sigmod Rec.* **2012**, *40*, 45–51. [CrossRef]
16. Conrad, A. *Database Economic Cost Optimization for Cloud Computing*; Brown University: Providence, RI, USA, 2009.
17. Wu, H.; Cheng, C. Research on search optimization based on Oracle database. In *Journal of Physics: Conference Series*; IOP Publishing: Bristol, UK, 2020; Volume 1648. [CrossRef]
18. Gonçalves, F.A.C.A.; Guimarães, F.G.; Souza, M.J.F. Query join ordering optimization with evolutionary multi-agent systems. *Expert Syst. Appl.* **2014**, *41*, 6934–6944. [CrossRef]
19. Trummer, I.; Wang, J.; Wei, Z.; Maram, D.; Moseley, S.; Jo, S.; Antonakakis, J.; Rayabhari, A. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In Proceedings of the 2019 International Conference on Management of Data, Amsterdam, The Netherlands, 30 June–5 July 2019; pp. 1153–1170.
20. Guo, R.B.; Daudjee, K. Research challenges in deep reinforcement learning-based join query optimization. In Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, Portland, OR, USA, 14–20 June 2020; pp. 1–6.
21. Ortiz, J.; Balazinska, M.; Gehrke, J.; Keerthi, S.S. Learning state representations for query optimization with deep reinforcement learning. In Proceedings of the Second Workshop on Data Management for End-to-End Machine Learning, Houston, TX, USA, 15 June 2018; pp. 1–4.
22. Macdonald, C.; Tonellotto, N.; Ounis, I. Efficient & effective selective query rewriting with efficiency predictions. In Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Tokyo, Japan, 7–11 August 2017; pp. 495–504.
23. Van Hasselt, H.; Guez, A.; Silver, D. Deep reinforcement learning with double q-learning. In Proceedings of the AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; Volume 30.

24. Liu, J.; Gao, F.; Luo, X. Survey of deep reinforcement learning based on value function and policy gradient. *Chin. J. Comput.* **2019**, *42*, 1406–1438.
25. Tong, Z.; Ye, F.; Liu, B.; Cai, J.; Mei, J. DDQN-TS: A novel bi-objective intelligent scheduling algorithm in the cloud environment. *Neurocomputing* **2021**, *455*, 419–430. [CrossRef]
26. Kipf, A.; Kipf, T.; Radke, B.; Leis, V.; Boncz, P.; Kemper, A. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *arXiv* **2018**, arXiv:1809.00677.
27. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv* **2013**, arXiv:1312.5602.
28. Liu, Q.; Zhai, J.W.; Zhang, Z.Z.; Zhong, S.; Zhou, Q.; Zhang, P.; Xu, J. A survey on deep reinforcement learning. *Chin. J. Comput.* **2018**, *41*, 1–27.
29. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjel, A.K.; Ostrovski, G. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529–533. [CrossRef]
30. Schulman, J.; Levine, S.; Abbeel, P.; Jordan, M.I.; Moritz, P. Trust region policy optimization. In Proceedings of the International Conference on Machine Learning, PMLR, Lille, France, 6–11 July 2015; pp. 1889–1897.
31. Mahajan, A.; Tulabandhula, T. Symmetry learning for function approximation in reinforcement learning. *arXiv* **2017**, arXiv:1706.02999.
32. Lillicrap, T.P.; Hunt, J.J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; Wierstra, D. Continuous control with deep reinforcement learning. *arXiv* **2015**, arXiv:1509.02971.
33. Lee, K.M.; Kim, I.; Lee, K.C. DQN-based join order optimization by learning experiences of running queries on spark SQL. In Proceedings of the International Conference on Data Mining Workshops (ICDMW), Sorrento, Italy, 17–20 November 2020.
34. Li, J.; Xia, X.; Liu, X.; Wang, B.; Zhou, D.; An Y. Probabilistic group nearest neighbor query optimization based on classification using ELM. *Neurocomputing* **2018**, *277*, 21–28. [CrossRef]
35. Ziane, M.; Zaït, M.; Borla-Salamet, P. Parallel query processing with zigzag trees. *VLDB J.* **1993**, *2*, 277–301. [CrossRef]