IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 June 2022, accepted 25 June 2022, date of publication 29 June 2022, date of current version 8 July 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3187102

RESEARCH ARTICLE

RL_QOptimizer: A Reinforcement Learning Based Query Optimizer

MOHAMED RAMADAN[1], AYMAN EL-KILANY[1], HODA M. O. MOKHTAR[1,2], AND IBRAHIM SOBH[3]

[1]Information Systems Department, Faculty of Computers and Artificial Intelligence, Cairo University, Cairo 12613, Egypt
[2]Faculty of Computing and Information Sciences, Egypt University of Informatics, Cairo 11865, Egypt
[3]Valeo, Cairo 12577, Egypt

Corresponding author: Mohamed Ramadan (m.ramadan@fci-cu.edu.eg)

ABSTRACT With the current availability of massive datasets and scalability requirements, different systems are required to provide their users with the best performance possible in terms of speed. On the physical level, performance can be translated into queries' execution time in database management systems. Queries have to execute efficiently (i.e. in minimum time) to meet users' needs, which puts an excessive burden on the database management system (DBMS). In this paper, we mainly focus on enhancing the query optimizer, which is one of the main components in DBMS that is responsible for choosing the optimal query execution plan and consequently determines the query execution time. Inspired by recent research in reinforcement learning in different domains, this paper proposes A Deep Reinforcement Learning Based Query Optimizer (RL_QOptimizer), a new approach to find the best policy for join order in the query plan which depends solely on the reward system of reinforcement learning. The experimental results show that a notable advantage of the proposed approach against the existing query optimization model of PostgreSQL DBMS.

INDEX TERMS Join ordering problem, query execution plan and query optimization.

I. INTRODUCTION

In DBMS, a single query can be executed through different execution plans. The query optimizer attempts to choose the most efficient way to execute a given query from the space of execution plans. Most DBMSs use the cost-based model for query optimization where the optimizer estimates the cost of the execution plan and then selects the optimal plan that minimizes the cost among the set of candidate plans [1]. As the number of intermediate rows (results) is unknown at run time, the Optimizer uses pre-calculated statistics such as information about the distribution of data values and cardinality estimation to estimate the cost of a plan rather than calculating the real cost of querying the data during the query plan. One of the main challenges in query optimization and query plan generations is the selection of the order to perform the join operation between tables (i.e. relations). Even if the final results of the query could be the same regardless of the join order, the order in which the tables of a query are joined can have a dramatic effect on the query execution time. In addition, the number of possible join orders increases exponentially with the number of tables [2]. Hence, the query optimizer can't compute the different costs for all combinations to select the best join order during query execution. Consequently, most optimizers use heuristics such as considering the shape of the query tree [3] - to prune the search space. In this paper, we propose two versions of A Reinforcement Learning Based Query Optimizer (RL_QOptimizer) to identify the best execution plan based on the reward system of reinforcement learning. The first model uses Reinforcement Learning and the second one uses Deep Reinforcement Learning [4], [5].

The main contributions of this work are:

1) Proposing a new query optimizer model (RL_QOptimizer) for optimizing tables' join orders that is based on the Deep Reinforcement Learning technique. Deep Reinforcement Learning is used to find the optimal query execution plan.

The associate editor coordinating the review of this manuscript and approving it for publication was R. K. Tripathy.

2) Using a real environment to train the proposed model so that real feedback from the DBMS is obtained and consequently employed to discover the optimal execution plan.
3) Enhancing the overall execution plan time and proposing a new query optimizer that requires low or almost constant time to generate the execution plan for any query with any number of joins.

The rest of the paper is organized as follows: Section 2 presents background about the concepts used in this paper. Then, previous work related to the proposed models is discussed in Section 3. Section 4 details the proposed models and their different architectures. In section 5, the results of the performance evaluation of the proposed models are presented. Finally, Section 6 concludes the paper.

II. BACKGROUND

A. QUERY OPTIMIZATION

Query Optimization is the process of choosing a suitable execution strategy for processing a query [1]. A traditional query optimizer uses stored statistics and probabilities rules to estimate the cardinalities of the different tables and consequently finds the optimal query plan. These statistics include the number of records, the number of blocks, the number of distinct values in each column, and the selectivity of each attribute that represents the average number of records satisfying an equality condition [1]. The goal of the query optimizer is to generate a query execution plan that minimizes the overall query execution time. A traditional query optimizer depends on heuristic and cost-based optimization. For example, applying the SELECT and PROJECT operations before the JOIN operations or applying the most restrictive SELECT operations first before other SELECT operations are examples of heuristics rules that mostly guarantee less execution time when applied on the execution plans. In the cost-based optimization step, the optimizer estimates and compares the costs of query execution based on statistics and cardinality estimations using different execution strategies and algorithms. Then, it chooses the strategy with the lowest cost estimate [1]. The lowest cost estimate is usually found by performing the operations that initially reduce the size of intermediate results.

Example 1: Consider the following query on the Customer-Ordering database presented in Figure 1. The Customer-Ordering database has five entities which are "ORDER", "CUSTOMER", "PRODUCT", "CATEGORY", and "ADDRESS" with the cardinalities 1000000, 100000, 10000, 1000, and 1000 rows for each table respectively.

```
  SELECT C.NAME, C.ADDRESS, P.PRICE,
P.NAME FROM CUSTOMER AS C CROSS JOIN
ORDER AS O CROSS JOIN PRODUCT AS P
WHERE CUSTOMER.ID = O.CUSTOMER_ID
AND P.ID = O.PRODUCT_ID
AND C.PHONE_NUMBER = "0111"
```

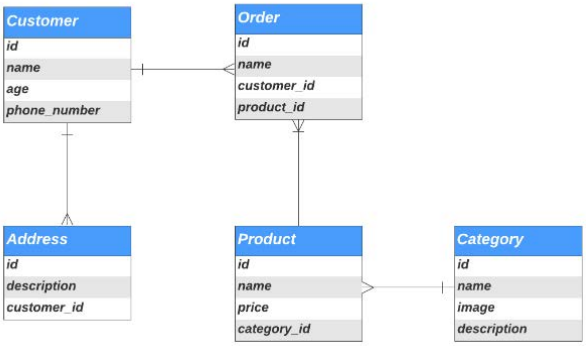

FIGURE 1. Customer-ordering database ERD.

Heuristic rules would recommend performing the query selection operator first in order to reduce query intermediate results [1] as shown in Figure 2. The following are the heuristic query optimization steps [1]:

1) Designing the initial tree of the query.
2) Moving the SELECT operation down the query tree.
3) Applying the more restrictive SELECT operation first.
4) Replacing CARTESIAN PRODUCT and SELECT with JOIN operation.
5) Moving PROJECT operations down the query tree.

The SELECT operation (σ c(R)) is used to select a subset of tuples from a relation that satisfies a condition specified in the selection. The selection operation is also known as horizontal partitioning because it partitions the table or relation horizontally, where 'c' is the selection condition, which is a Boolean condition. The PROJECT operation (π A(R)) is used to select certain attributes while discarding others. The Project operation is also known as vertical partitioning because it partitions the relation or table vertically, discarding other columns or attributes, where 'A' is the attribute list, which is the desired set of attributes from the attributes of relation(R), and finally, the JOIN operation (R1 $\bowtie$ R2) is used to join two tables R1 and R2 based on the join condition. The outcome of joining two or more relations is a set of all possible tuple combinations with the same common attribute.

B. JOIN ORDERING PROBLEM

A "join" operation is a relational operation that combines rows from two tables based on a related column. While join works with only two tables at a time, a query that joins N tables is executed through N-1 joins. The optimizer needs to take a critical decision regarding the selection of the optimal join order which greatly influences the execution time of a query. The process of choosing an efficient join order is difficult as the number of possible join combinations that the optimizer needs to explore and analyze increases exponentially with the number of tables [2]. In addition, the number of intermediate rows (results) is unknown at run time which forces the optimizer to use pre-calculated statistics and

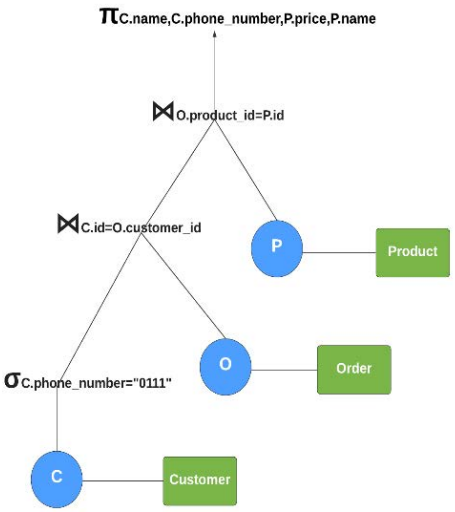

FIGURE 2. A query execution tree generated by traditional heuristic rules.

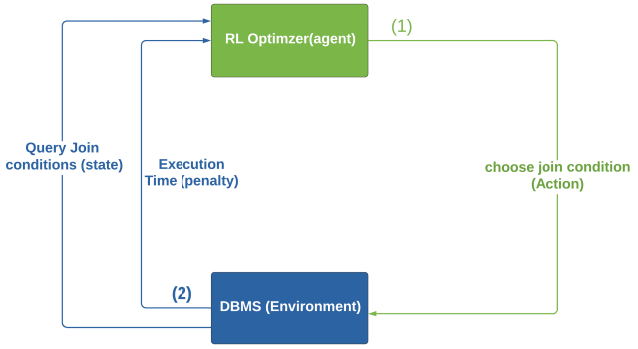

FIGURE 3. Simplified architecture of the RL mechanism.

cardinality estimation to estimate the cost rather than the real cost.

Example 2: Consider a query on the customer-ordering database shown in Figure 1, where we want to get the customers along with each product they have ordered. We can write this simple query in PostgreSQL like this:

```
SELECT * FROM CUSTOMER
CROSS JOIN ORDER CROSS JOIN PRODUCT
WHERE CUSTOMER.ID = ORDER.CUSTOMER_ID
AND
PRODUCT.ID = ORDER.PRODUCT_ID
```

If the optimizer chooses to join the relations "CUSTOMER" and "PRODUCT" tables first, it leads to a cross-product as there are no relationships between customer and product tables, which accordingly generates a very large set of intermediate results ($100000 \times 10000 = 10^9$ rows) and consequently results in a high execution time for the query but if the optimizer chooses to join "CUSTOMER" and "ORDER" tables first, it leads to 10^6 rows as a maximum number of intermediate results. The query optimizer's role is to select the best query join order that minimizes the query execution time. The better choice is affected by many factors including: database indexes, tables' cardinality, data distributions, etc. This problem is called a join ordering problem that has been studied by researchers for many years given the huge number of possible combinations where each candidate plan has a different effect on the query execution time. The problem has an exponential complexity while using a dynamic programming technique [6].

C. REINFORCEMENT LEARNING

Reinforcement learning is an important area of machine learning, where an agent learns to take an action to maximize the total rewards in an environment [7]. Reinforcement learning algorithms learn by performing actions and receiving rewards or penalties from the environment. The main goal of the agent is to maximize its cumulative rewards while interacting with the environment.

The main elements in reinforcement learning are agents, environments, states, actions, and a reward value [7].

Actions are the set of all possible actions that the agent can choose from. The environment takes the agent action and current state as input and returns a reward and the next state. The reward function defines the goal in a reinforcement learning problem, it maps the action state (or state-action pair) of the environment to the reward value (negative or positive). On each time step, the environment sends to the reinforcement learning agent a single value called the reward [7]. The policy is the strategy that an agent follows to determine the next action based on the current state. **Value function V(s)** is the expected long-term reward for an agent starting from state s under a specific policy. It measures how good to be in a given state. **Action Value or Q-Value function Q (s, a)** measures how good it is to take an action at a given state, it is the expected return or the overall reward for taking action a in a specific state s.

As shown in Figure 3 the proposed query optimizer (agent) interacts with the DBMS environment) by selecting one of the join ordering conditions (actions) and receiving a negative execution time (penalty). The reinforcement learning problems are closely related to optimal control problems, particularly stochastic optimal control problems that can be formulated as a Markov Decision Process (MDP) [7], [8]. A Markov process is a stochastic process in which the problem is a set of possible states and the future state depends only on the current state rather than the history (Markov Property). Markov Decision Process is described by the following tuples 1.

$$< S, A, P(s, a), R(s, a) > \quad (1)$$

where S stands for a set of states, A describes the set of actions the agent can take, P(s, a) describes a probability distribution to be in the new state by taking action a in state s and R(s, a) is the reward of taking action a in state s.

D. DEEP REINFORCEMENT LEARNING

Neural networks are function approximators that can be used in reinforcement learning when the state space or action space is very large [4]. Deep reinforcement learning is the result of applying reinforcement learning using deep neural networks. Deep neural networks are used as the agents that learn to map state-action pairs to rewards. Depending on the result, the neural network is encouraged or discouraged by the action on this input in the future.

Deep Reinforcement learning has been used widely in different domains. More specifically, it is used in those domains that a reward/penalty can be given for any action of the agent.

Google DeepMind Team developed many artificial intelligence models using deep reinforcement learning in different games like a model to play Atari games and improves itself [4] which used a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. In addition to developing AlphaGo which is a Chinese game that challenged artificial intelligence researchers for many years [7] by combining deep neural networks with reinforcement learning [9].

Deep reinforcement learning is used for other complex problems like autonomous driving [10], [11]. As mentioned in Sallab *et al.* [10], it is difficult to deal with autonomous driving as a supervised learning problem due to strong interactions with the environment. Finally, End-to-End Framework for Fast Learning Asynchronous Agents research [12] proposed a training framework that combines the benefits of imitation learning (IL) and deep RL for fast learning asynchronous agents through extending the Asynchronous Advantage Actor-Critic (A3C) algorithm.

III. RELATED WORK

Related work is categorized into two main directions: Traditional query optimization and Learning-based query optimization. Previous work in each of those directions is discussed in the following subsections.

A. TRADITIONAL QUERY OPTIMIZATION TECHNIQUES

Most of the query optimizers rely upon the dynamic programming approach of System R [13], [14] which was the first implementation of SQL and it pioneered several optimization techniques, including the utilization of dynamic programming for bottom-up join tree construction. They use the traditional cost model to determine the best plan for a given query by generating different strategies using cardinality estimations [1], [15]. These estimates rely upon statistics on the database and assumptions that may or may not be true. Invalid assumptions or inaccurate calculations for the cardinality estimation lead to poor execution plans [1], [15], [16].

Another type of query optimizer tries to use parametric query optimization [17]–[21] Where the traditional optimizers make assumptions about many parameters [20] whose values are unknown at compile-time, the time before the actual execution, the parametric techniques attempt to identify several execution plans where each plan is optimal for a subset of possible values of the run-time parameters [6]. The goal is to identify candidate plans at compile-time, each optimal for some region of the parameter space, and the optimal plan is selected once the actual parameter values become known at run time. This type has many drawbacks like the overhead of pruning all plans for the entire relational selectivity space for one query which is not a cost-effective approach [21]. This approach depends on assumptions that may not be true all the time like assuming that a plan is optimal for all values in a specific region [22].

Most DBMSs use histogram-based techniques as part of their cost model to summarize the data of tables to perform efficient selectivity estimations [23], [24]. A large number of algorithms have been proposed for constructing histograms over a single attribute and multiple attributes. A new algorithm to build a histogram is introduced in [25] to construct it by minimizing the aggregated error. As the algorithm needs huge construction time, generating an efficient execution plan for a given query from the space of possible execution plans is very expensive. In addition, estimating the join operation selectivity problem may lead to poor execution plans.

B. LEARNING-BASED QUERY OPTIMIZATION

Query Optimization using learning models has become one of the hot topics in the database research directions [26]–[35]. Some researchers have investigated the feasibility of applying machine learning techniques in query optimization to improve the query optimization process.

Some of the prior work used supervised learning to learn from old execution plans that were generated by the query optimizer for past queries to help in generating execution plans for new queries [1]. The authors in [26] proposed an execution plan recommendation system based on similarity identification between SQL queries. They used machine learning techniques to improve query similarity detection and hence were able to identify and associate similar queries having similar execution plans. This algorithm assumes that similar textual queries have similar execution plans, however, this is not always true in real-world where similar textual queries can have different optimal execution plans. In addition, the paper didn't use the query optimizers' feedback to enhance the query execution plan.

Other proposed machine learning-based query optimizer models focus on adjusting incorrect statistics and cardinality estimates of a query execution plan automatically by learning from query optimizer past mistakes. One of the first approaches that focused on adjusting this information is [27] which compares the optimizer's estimates with the actual cardinalities during run time and computes the errors. Then, the model is adjusted to perform better in future runs. Also, Adaptive Cardinality Estimation [32] proposes a cardinality estimation approach that is integrated with the use of machine learning techniques. The main contribution of this approach

is using query execution statistics of the previously executed queries to improve cardinality estimations. The proposed approaches have many issues e.g., they are designed for static queries. In addition, they focus on cardinality estimation so, they still require to use of the traditional cost model and heuristic rules that may lead to poor plans. Similar to the traditional optimizers, the proposed model planning time increases as the number of join conditions increases. Another approach was proposed by [28] which uses machine learning algorithms for cardinality estimation to learn selectivity that takes a bounded range on each column as input. This method focused on the selectivity estimation of several range clauses but did not consider Queries with joins. In addition, it focused on cardinality estimation, not the actual execution time which may affect dramatically the execution time.

The approaches presented in [29], [30] use a deep reinforcement learning technique in determining the execution plan. ReJOIN model [29] focuses on the Join order selection problem by applying deep reinforcement learning techniques. In this model, the agent learns to maximize the reward through continuous feedback with the help of an artificial neural network. ReJOIN used the traditional cost model based on cardinality estimation during the learning phase rather than the actual execution time which may lead to non-optimal plans. Learning State Representations for Query Optimization discussed in [30] used deep neural networks to learn state representations of queries in order to learn the optimal plans. More specifically, the paper introduced two approaches, the first approach transforms a query into a feature vector and trains a deep neural network to take such vectors as input and output the estimated cardinality. The second approach, a recursive approach, in which they train the model to predict the cardinality of a query consisting of a single new operation applied to a subquery to incrementally generate a representation of each subquery's intermediate results [30]. This paper explored the idea of training a deep reinforcement learning model to predict query cardinalities instead of relying entirely on basic statistics to estimate costs.

Neo (Neural Optimizer) presented in [31], uses a supervised learning model to guide a search algorithm through a large and complex space. Neo assumes the existence of a sample workload which consists of a set of queries that is considered a representative of the total workload. In addition, PostgreSQL optimizer is considered as the expert that is responsible for generating the best query plans. Given the sample workload and their best query plans generated by the expert, the learnt model tries to generalize a model that can infer the plan with the least execution time for a query. In later stages, Neo retrains the supervised learning model based on the feedback received while running the model on its environment. Towards a Hands-Free Query Optimizer through Deep Learning presented in [36], is another attempt that tries to identify potential complications for future research that uses deep reinforcement in Query optimization problems. Also, the authors referred to the possibility of using latency as a reward function in the research directions. The SkinnerDB system presented in [37] uses reinforcement learning for query optimization. The proposed model learns the optimal join order while running the query. The possible join orders are divided into slices where the possible join order is tested on each slice of the data until the best join order is obtained and considered for the remaining slices of data. Query performance is evaluated using regret bounds as a reward system that considers the difference between actual execution time and the time for an optimal join order. Fully Observed Optimizer (FOOP) presented in [38] uses a reinforcement learning model where the reward function is defined as the cost model of the traditional DBMS optimizer. Another model that utilizes reinforcement learning is presented in [33], which is the closest model to the proposed model in this paper. The model suggests a learning-based technique for join order based on the plans generated by the DBMS optimizer to bootstrap the reinforcement learning model before fine-tuning it using real-time execution time. BAO (the Bandit optimizer) presented in [39], is a learned component that sits on the top of an existing query optimizer in order to enhance query optimization rather than discarding the traditional query optimizer. Bao component learns to map the query to the best execution strategy for the query. Then, upon receiving a query, the query optimizer generates multiple plans according to different strategies where the learned model is expected to choose the best query plan given the possible strategies.

Another research direction explores the use of deep reinforcement learning to administrate a DBMS. The case for Automatic Database Administration investigated in [40] proposes a new model of index selection to decide on which attributes to create indexes on for a given workload based on deep reinforcement learning. UDO(the Universal Database Optimizer) [41] considers a variety of tuning choices, starting from picking transaction code variants over-index selections up to database system parameter tuning. UDO uses reinforcement learning to converge to near-optimal configurations.

All of the earlier models have utilized DBMS optimizer and its generated plans to train or at least bootstrap their learned models. Consequently, the purpose of this research is to develop reinforcement learning-based models that learn directly from real query performance of different join orders where the models are rewarded or penalized based on the actual execution time of different query plans. Furthermore, the proposed models explore the whole space of different query plans to learn the best join order for any given query.

IV. PROPOSED MODELS

In this paper two versions of A Reinforcement Learning Based Query Optimizer (RL_QOptimizer) are proposed to solve the join ordering problem during query optimization. Both approaches employ the Q-learning model [5], which is one of the most popular reinforcement learning algorithms. The first approach is a simple RL Q-Learning which uses a simple lookup table (Q-table) to calculate the maximum expected future reward for each action at each state.

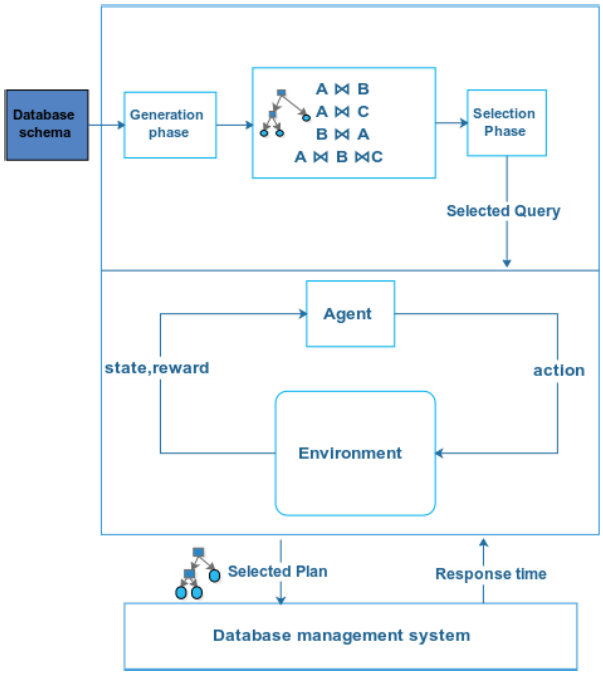

FIGURE 4. Model overview.

The second approach is a 'Deep' Q-Learning-based model, which is more suitable for large states and actions as it uses a neural network to approximate the Q-value function. Both models operate by applying a set of general steps as shown in Figure 4.

The system has two main phases that are applied for both models, the first phase is **the generation phase** and the second is **the selection phase**.

In **the generation phase**, the model either generates all possible join ordering queries that may happen in the database schema to learn or generate all the join ordering queries from a given database workload. Generating all possible join ordering queries allows the model to be trained from scratch on every possible scenario For example, if the database has JOINS between A, B, and C, the possible queries will be $A \bowtie B$, $A \bowtie C$, $B \bowtie C$, $A \bowtie B \bowtie C$. On the other hand, If a database workload is available, the join ordering queries in the workload will be used in the training process by the model. Following that, the system selects any one of join ordering queries in **the selection phase**. All possible execution plans of the selected query will be generated to **train the model**. For each possible execution plan, the agent interacts with the DBMS to get **the actual execution time** for this plan, which represents the reward in our models, multiplied by -1 to minimize the execution time. Both models are discussed in details in the following subsections.

A. JOIN ORDERING USING REINFORCEMENT LEARNING

The first model uses Q-table to store the expected reward for each action-state. The main function of the Q-table is to take a state and an action as input and produce the corresponding Q-value as shown in Figure 5. On the other hand, the agent performs a set of sequence actions to get the maximum total reward. The total reward is called the Q-value which can be calculated by performing an action in a specific state to get the immediate reward and add it to the highest Q-value possible from the next state using the following formula 2 [5], [42]:

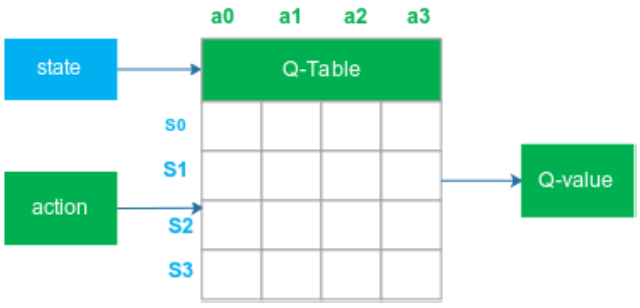

FIGURE 5. Q-learning.

$$Q(s, a) = r(s, a) + \gamma maxQ(s', a) \tag{2}$$

The first part r(s, a) is the immediate reward for the taken the action (a) given the state (s). The second part is the discount factor (γ) multiplied by the estimate of optimal future value max Q(s′, a) which is known as the discounted estimate of optimal future value.

The model consists of four components which are: the input of the model, the states, the set of possible actions and the reward function. The preceding equation shows how we compute the Q-value for an action (a) starting from a state (s). It is the sum of immediate rewards, and it takes greedy action from the next state(s′) (choose the action that has maximum Q value over other actions).

The input is typically represented as encoded query join conditions. The characteristics of join conditions are encoded in the form of a vector of size n, where n is the number of all possible join conditions in the database schema. Each cell of the vector can be a 0 or 1 where 1 means that this condition is included in this query. E.g. Input = [1, 1, 0, 0] means that this query includes first and second join conditions.

The main goal in the join ordering problem is to find the best possible join order for a given query. In the proposed models, **the states** represented by 0 or 1's vector where 1 refers to join conditions that will be applied to a query. In each state, the agent has a set of possible **actions** to select one from them to move from one state to another. These actions are all join conditions of the query represented by one's in-state vector. After selecting an action, the query builder adds this condition in its order and sets its value to zero in the state vector.

As no rules exist to correctly choose **the reward** function, the choice of the reward function is one of the most challenging tasks in any reinforcement model. In the proposed models, the goal is to optimize the total execution

time of queries, hence, the actual queries execution time multiplied by -1 is used as a reward. During experiments, PostgreSQL [43] DBMS is used to get the actual execution time of the query. Obviously, the lower the execution time, the higher the reward.

In the learning stage the system takes all possible join conditions for the given database and generates different possible queries to train on them. The model then builds a vectorized representation of the query that is later used as an input to the model. The agent selects one of the join ordering conditions in this vector which is represented by one's, then the environment gives a reward by interacting with PostgreSQL DBMS. This process is repeated until the terminal state. Finally, the function Q(s, a) in the Q-table using this equation is updated as follows 3 [5], [42]:

$$Q(s, a) = Q(s, a) + \alpha[r(s, a) + \gamma maxQ(s', a) - Q(s, a)] \tag{3}$$

The first part Q(s, a) is the current value in the state (s) if an action (a) is taken, and the second part is the learning rate (α) multiplied by the TD error. which is the difference between the TD target and the current Q(s, a). with the following three essential steps:

1) The agent begins in a state (s), acts (a), observe the next state s′, and reward r.
2) The agent chooses an action by referring to the Q-table with the greatest value for the next state(s′)
3) Update q-values.

Example 3: Consider the customer-ordering database shown in Figure 1 that has 4 join conditions [CUSTOMER ⋈ ADDRESS, CUSTOMER ⋈ ORDER, ORDER ⋈ PRODUCT, PRODUCT ⋈ CATEGORY] which is vectorized as [1,1,1,1]. In the training phase, the agent tries to explore all possible execution plans. First, the agent explores each join condition individually, then it trains on the vector [1, 0, 0, 0] and builds the first query with [CUSTOMER ⋈ ADDRESS]. The environment interacts with the DBMS to get the actual execution time of this query in addition to [ADDRESS ⋈ CUSTOMER] query execution time to update the Q-table. The model performs the same process for each join condition individually. Then, the model trains on all possible two join conditions with each other. For example, it will train on the vector [1, 1, 0, 0] for [CUSTOMER ⋈ ADDRESS] and [CUSTOMER ⋈ ORDER]. In this case, the model trains on the best join order out of the following two join orders. The first join order consists of [CUSTOMER ⋈ ADDRESS] and the better order from [CUSTOMER ⋈ ORDER] and [ORDER ⋈ CUSTOMER] where the second join order is [ADDRESS ⋈ CUSTOMER] with the better join order from [CUSTOMER ⋈ ORDER] and [ORDER ⋈ CUSTOMER]. Where, the best from [[CUSTOMER ⋈ ORDER] and [ORDER ⋈ CUSTOMER] joins is already discovered during the previous cycle of training. This process is repeated until the training process is terminated by training the whole [1, 1, 1, 1] vector.

During the actual running when the model is required to generate the best execution plan for a query that is vectorized as [1, 1, 1, 0], the agent moves to the corresponding row in Q-table to select the condition with maximum reward. If the agent selects the third join condition, which is coded as one in the vector, it will be replaced by zero and the new state will be [1, 1, 0, 0]. Then, the agent selects the best join condition given the chosen third join condition and this is also a one in the vector and will be replaced by a zero. This process is repeated recursively until the vector reaches the terminal state which is [0, 0, 0, 0] to retrieve the best join conditions order.

B. JOIN ORDERING USING DEEP REINFORCEMENT LEARNING

The join ordering using reinforcement learning model has many limitations that need to be solved before considering it as a practical solution. The main problem is related to the size of the database schema and the number of tables. A large database schema with many join conditions leads to gigantic state space that may reach up to millions of states. Consequently, the Q-table needs a large amount of memory to store. In addition, the exploration of the Q-table won't be efficient. Another limitation is related to the generalization as the Q-table model can't infer a Q value of a new state from the already trained states. Thus, join ordering using deep reinforcement learning model was introduced to address those limitations.

Join ordering using deep reinforcement learning model introduces a deep neural network to approximate the Q-value function. The state is given as input and the Q-value of all possible actions is generated as output as shown in Figure 6. Similar to any deep neural network, it uses coefficients to approximate the function that maps an input to the output. Accordingly, the algorithm learns the right coefficients by adjusting their values iteratively in the learning state. In the proposed model, weights of the deep neural network are updated during training instead of updating the Q-value directly in the Q-table.

The proposed model uses the Deep Q Network (DQN) which uses a neural network to approximate the Q-value function to tell the agent what action to take. This model was proposed in DeepMind's paper [4] to learn policies from high-dimensional sensory input using reinforcement learning. As stated in [42], RL is known to be unstable or even diverge when neural networks are used to represent the action-values. There are various factors that lead to this instability: The presence of correlations in the sequence of observations and the correlations between the action-values(Q) and the target values. In the proposed model we followed the following improvements on DeepMind's model presented in [4], [42] to tackle these issues:

1) *Experience Replay:* buffer replay was used to store the latest N experience tuples observed by an agent, including state, action, reward "response time", and next state which allows the network to reuse this data later by sampling from it randomly. During the training

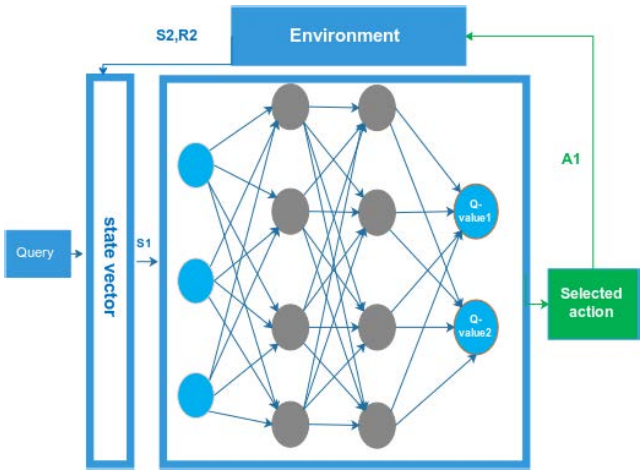

FIGURE 6. DQN model.

phase, the model uses random training samples from the replay data as input, which leads to more efficient use of previous experiences and helps to reduce database calls by using a buffer rather than calling the engine for the same queries again.

2) *Target Network:* the Bellman equation provides us with the value of Q(s, a) via Q(s′, a) as the following equation 4:

$$Q(s,a) = Q(s,a) + [r(s,a) + \gamma maxQ(s',a) - Q(s,a)] \tag{4}$$

In deep Q-learning, we need to minimize the mean squared error between the target Q value (TD target) and the current output which is called TD error and we need to estimate the TD target using the following equation 5 [5], [42]:

$$Q(s,a) = r(s,a) + \gamma maxQ(s',a) \tag{5}$$

When the parameters of our Neural Networks are changed to get Q(s, a) closer to the intended result, the value produced for Q(s′, a′) can be changed indirectly which can make our training very unstable. So, the target network was introduced to stabilize the learning process [42].

In the proposed model, a separate network with fixed parameters was used to estimate Q-targets. At every step, the parameters are copied from the DQN network to a separate target network to estimate the Q-targets. Similar to the first model, the deep-reinforcement learning model takes the database schema and possible join orders to explore all possible queries in the training process. Assume the same customer-ordering database presented in Figure 1 to train reinforcement learning model presented in the previous section. As presented in Example 3, it's required to train the model 4 join conditions database [CUSTOMER ⋈ ADDRESS, CUSTOMER ⋈ ORDER, ORDER ⋈ PRODUCT, PRODUCT ⋈ CATEGORY] which is encoded as vector [1,1,1,1]. In this model, the same training process is applied but the neural network is employed as an approximation function and is used instead of the Q-table to predict the reward. The query vector represents the input of the neural network and the actual query execution time is the target value. The weights of the neural network are modified to minimize the error between the predicted value and the target value. This process is repeated until the training process is terminated.

A four-layer feed-forward neural network with 30 neurons in each hidden layer is used, with the number of input layer neurons equal to the number of possible joins in the database and the number of output layer neurons also equal to the number of possible joins. The input is fed as vector of integers and the output is represented as a vector of integers. For example, when the model is required to find the best execution plan for a query of three join conditions in a database with five possible joins, the network is fed a vector with these conditions represented by ones and others represented by zeros, such as [1, 1, 0, 1, 0]. The neural network output is expected to be a vector with the reward value for each condition which guides the agent to choose which join condition to perform. The agent will apply the condition with the maximum reward. The selected join condition, which is coded as one in the input vector, will be replaced by zero in the input vector and the new input vector will be fed to the network in order to identify the next join condition. This process will be repeated until the terminal state vector [0, 0, 0, 0, 0] is reached. The discount factor is set to 0.9 and an Adam optimizer [44] is sued with a learning rate of 0.001.

V. EVALUATION

The objective of the evaluation is to assess the quality of the execution plans produced by the proposed models against the execution plans produced by PostgreSQL DBMS as a baseline in order to prove the proposed model effectiveness. Towards this goal, the performance of the proposed models are evaluated and compared with the results of PostgreSQL DBMS query optimizer. The models were evaluated on two databases, a real database that is used as a benchmark dataset for the join-ordering problem [45] and a synthetic database that is used in TPC-H benchmark with various sizes to test different scaling factors [46].

A. EXPERIMENTS SETUP

All experiments are conducted using a laptop running Ubuntu version 18.04.3 LTS with an 8-core Intel Core i7-8550U and 8 GB of RAM. Memory was set as available per operator (*work_mem*) to 512MB and the size of the buffer (*shared_buffer*) was set to 1 GB. Models were implemented using Python, Tensorflow [47] and Keras [48] library to implement the neural networks. For the DQN model, Adam query optimizer [44] and the RelU activation function were used. In addition, PostgreSQL (v10.13) was used and the join collapse limit parameter was set to 1 at run-time to force the planner to follow the join order. setting it to 1 prevents any reordering of explicit Joins. Thus, the explicit join order

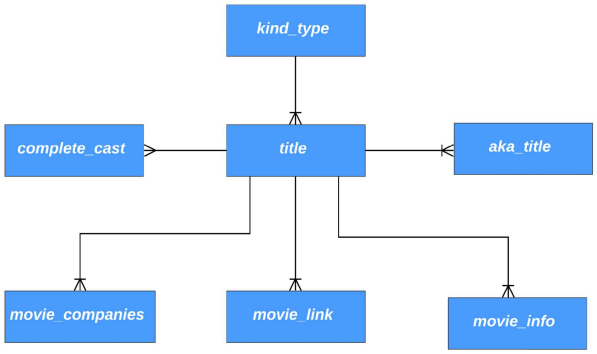

FIGURE 7. Simplified ER diagram of the IMDB database.

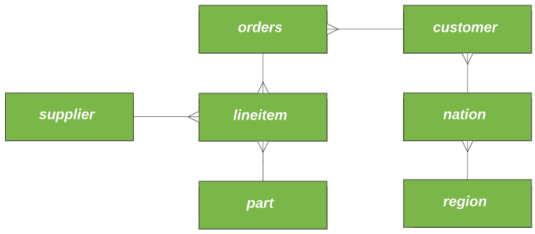

FIGURE 8. Simplified ER diagram of the TPC-H database.

specified in the query will be the actual order in which the relations are joined.

Databases:

1) IMDb (Internet Movie Database) [49] is an online real-world database containing a large amount of information about films, television programs, and home videos which is available for non-commercial use that is used as a benchmark for the join ordering problem in article [45]. During the training process, only the tables shown in Figure 7 was used where the average number of records per table is greater than 6 million records. According to figure 2 in the paper [45], a typical query graph has five relationships with the "title" table, which is the center table for the research workload. As a result, we chose all joins operations related to the "title" table to show us the query optimizer problem in join ordering, which are as follows: (movie_companies ⋈ title, title ⋈ kind_type, movie_info ⋈ title, aka_title ⋈ title, movie_link ⋈ title, title ⋈ complete_cast)
2) TPC-H Database [46]: Using the retail database existing in the TPCH benchmark which contains a large amount of information about customers, orders, lineitems, parts, part suppliers, suppliers, nations, and regions. TPC-H continues to be the most widely used benchmark for relational systems and most join queries operate on three tables or more [50]. lineitem and orders, which carry around 83 percent of the total data, are the most challenging and largest tables in this schema. As a result, all lineitem, orders and customer possible join operations are generated, totaling 7 tables, to demonstrate the join ordering challenge, which are as follows: (customer ⋈ nation, lineitem ⋈ part, lineitem ⋈ supplier, lineitem ⋈ orders, nation ⋈ region, orders ⋈ customers). Our TPCH experiments use a database scale factor of 1 which Consists of the base row size (equals 1GB raw data). The simplified ER diagram for entities without attributes is shown in Figure 8.

Because the model generates all potential execution plans for the generated query during the training process to learn a better plan than the traditional Postgresql model plan, some of the generated plans may be significantly worse than the PostgreSQL plan and take hours to complete. These plans are ineffective and inefficient execution plans from which the model shouldn't learn. To overcome this issue, we cut off and discard any query plan that takes execution time greater than a configurable maximum time which is set to 3 minutes in our experiments because using PostgreSQL's traditional model, all generated training queries in the experiments take less than 3 minutes.

In order to perform training and testing of the two reinforcement learning proposed models, all possible number of join ordering queries for each database schema were generated where each query consists of a different number of joins. Each query was encoded in vector format as shown in Table 1 in order to use it during training and testing phases where the vector encodes join conditions between different relations.

The proposed models performance are compared to PostgreSQL optimizer using their generated plans execution time. In addition, the planning time required to generate the plans by all models are collected during experiments. For example, the proposed models generated a plan for the query mentioned in Table 1 that was executed in 60 seconds while the traditional optimizer plan was executed in 100 seconds. The optimizer decided to start by joining [`MOVIE_INFO ⋈ TITLE`] which took a larger execution time and led to a very large intermediate results which was around 29 million records. On the other hand, the proposed models plan chose to start by joining [`MOVIE_COMPANIES ⋈ TITLE`] which led to less than 5 million intermediate results. In addition, the proposed models required around 3 milliseconds to generate their plans while the PostgreSQL optimizer required 25 milliseconds.

Another experiment was conducted to evaluate the model while using a database workload during training phase instead of training the model on generated queries. This experiment uses the Join Order Benchmark (JOB), a collection of queries used as a benchmark for the join ordering problem in article [45]. Each query in the benchmark joins between four and

TABLE 1. A query example and its encoded vector.

Query	SELECT * FROM MOVIE_COMPANIES CROSS JOIN TITLE CROSS JOIN KIND_TYPE CROSS JOIN MOVIE_INFO WHERE MOVIE_COMPANIES.MOVIE_ID = TITLE.ID AND TITLE.KIND_ID = KIND_TYPE.ID AND MOVIE_INFO.MOVIE_ID = TITLE.ID
Vector	[1, 1, 1, 0, 0, 0,]

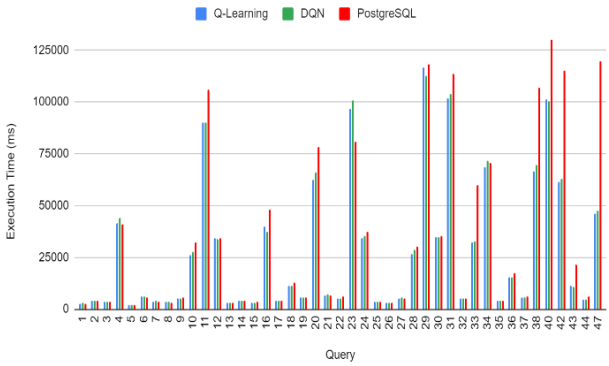

FIGURE 9. Comparison between the results of reinforcement learning models with PostgreSQL on IMDB database.

FIGURE 10. Comparison between the results of reinforcement learning models with PostgreSQL on IMDB using IQR.

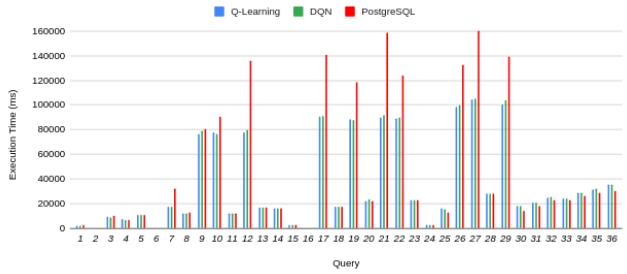

FIGURE 11. Comparison between the results of reinforcement learning models with PostgreSQL on TPCH database.

seventeen relations. Similar to the ReJOIN paper [29], the same 10 queries were utilized to test the suggested model. The last experiment was conducted to assess the generalization ability of the proposed models. The proposed models were trained on 80% of the queries where the remaining 20% were left for testing as unseen queries. The DQN model is compared against Q-Learning based model on the test set of unseen queries where the execution times are collected for each query plan that is generated by each model.

B. EXPERIMENTAL RESULTS

1) IMDb Database Results

As shown in Figure 9 where the X-axis represents the query IDs and Y-axis represents the query execution times in milliseconds, the proposed models outperform PostgreSQL optimizer in 30% of the queries and failed in 3% otherwise both provide the same execution plans. For one query, the optimizer failed to finish it with the maximum time of 3 minutes and the query execution was halted where the reinforcement learning-based models which finished before the maximum time. In Figure 10 the interquartile range (IQR) is used to measure the variability between different models using all queries performance on each model. The figure shows that the interquartile range (IQR), maximum execution time, mean execution time (defined by X mark in the graphs) of the PostgreSQL is larger than the reinforcement learning-based proposed models.

2) TPC-H Database Experiment

Applying the proposed models on the TPCH database shows that the proposed models outperform PostgreSQL in 27% of queries and failed in 8% with very small differences in execution time as shown in Figure 11. During the evaluation, the optimizer failed to finish one query within the maximum allowed time where the reinforcement learning-based models succeeded. As shown in Figure 12 interquartile range (IQR), maximum execution time, mean execution time (defined by X mark in the graphs) of the PostgreSQL is larger than the reinforcement learning-based proposed models.

Another important observation from Figure 9 and Figure 11 is that Q-Learning model results are very close to DQN model results in all queries.

3) Join Order Benchmark Queries Results

The proposed model discovered execution plans that outperform PostgreSQL in 70% of total queries in the benchmark queries and failed in 8% with small differences in execution time. Nevertheless, both provide the same execution plans. As shown in Figure 13 and Figure 14, the interquartile range (IQR) of PostgreSQL's mean execution time (shown by the X mark in graphs) is greater than that of the DQN model. Using the proposed DQN model, the

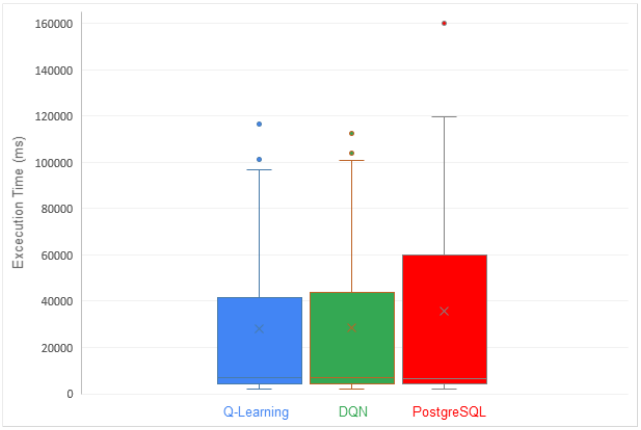

FIGURE 12. Comparison between the results of reinforcement learning models with PostgreSQL on TPCH database using IQR.

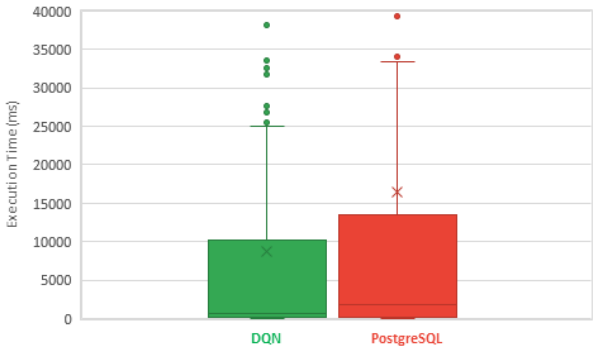

FIGURE 13. Comparison between the results of the DQN model with PostgreSQL on join order benchmark 113 queries using IQR without showing extreme outliers.

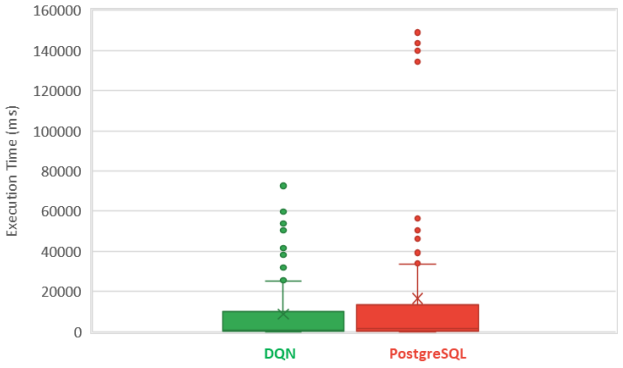

FIGURE 14. Comparison between the results of the DQN model with PostgreSQL on join order benchmark 113 queries using IQR with showing extreme outliers.

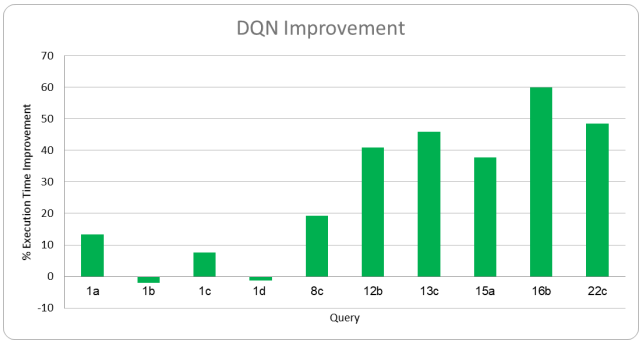

FIGURE 15. The percentage by which the DQN model outperformed PostgreSQL on the join order benchmark test queries.

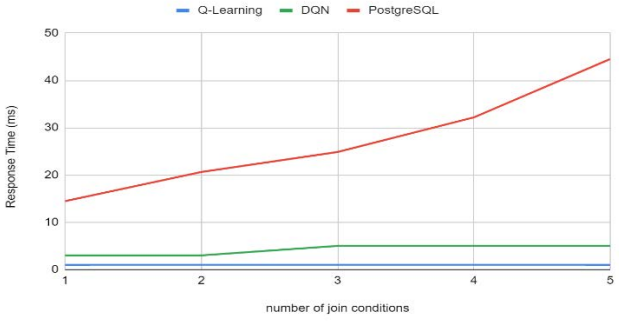

FIGURE 16. Comparison between the planning time of reinforcement learning models with PostgreSQL.

average execution time of queries in the benchmark is 8670 ms when the average execution time for PostgreSQL is 16479 ms. In addition, the DQN model outperforms PostgreSQL exceptionally in a subset of queries. For Example, Query "17b" in the benchmark queries requires 50000 ms while using the plan generated by the DQN model unlike the plan generated by PostgreSQL which requires 140000 ms. In comparison to the PostgreSQL optimizer, the DQN model generates query plans that are, on average, 27% less expensive for the 113 queries in the benchmark queries. In addition, The model was tested using the same ten queries that were used in ReJOIN [29] as shown in Figure 15 and demonstrated that the model provided join ordering plans that were 27% cheaper than those generated by the PostgreSQL optimizer Which is superior to the ReJOIN approach, which provides on average 20% improvement. In addition, for a query (16b), the plan generated by the DQN model is 60% less expensive than PostgreSQL generated plan. However, the ReJOIN generated plan is just 20% less expensive.

4) Planning Time
In Figure 16, the relationship between the planning time and the number of join conditions for both models is shown where the planning time increases as the number of join conditions increases in PostgreSQL DBMS while the planning time for Q-learning models is almost constant.
5) Generalization Assessment
Q-learning is built around the Q-table where it can predict the best actions only for the states that are used during the model training phase and doesn't generalize

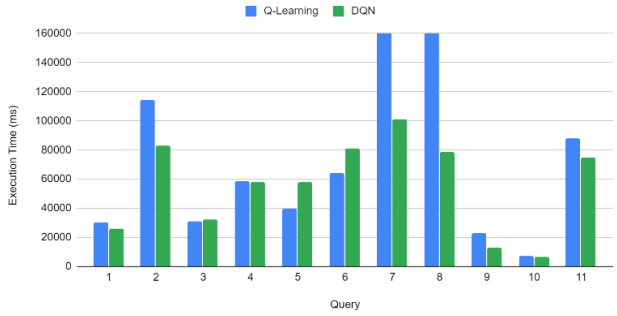

FIGURE 17. Comparison between the results of the DQN and the Q-learning models on IMDB database for new queries.

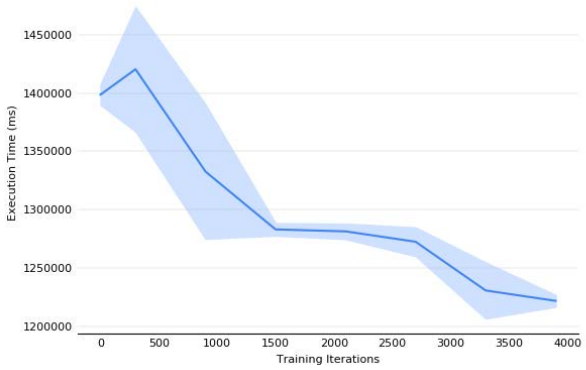

FIGURE 18. Training curve showing average penalty per episodes during the training process on IMDB database.

for queries that haven't been seen. In the proposed model, The Q-table has all possible join conditions in the database schema, and for states that have never been seen before, the Q-table will reward all actions equally. As a result, in the case of Q-table, the model will choose any action at random from the set of possible join conditions. On the other hand, DQN depends on a deep neural network that transforms the state's information and their best actions into neurons learned weights. Thus, the DQN model is expected to be able to take an action for states that were never seen before given their similarity with the states used previously during the training phase. To assess the DQN model generalization ability, 20% of the generated queries were chosen as test data to cover a variety of queries with varying numbers of joins operations, which included four queries with two joins operations, two with three joins, four with four joins, and one with five joins. The model was trained on the remaining queries on the IMDB Database. As shown in Figure 17, the DQN model outperforms the Q-learning model in 55% of queries, however, it loses in 18% of queries. Otherwise, the two models provide the same execution times. The experiment shows clearly the ability of the DQN model to generalize to queries states that were never seen before unlike the Q-learning model that depends on randomization to generate plans for unseen queries. Although Q-learning is better than DQN in Queries 5, 6, there's no guarantee to provide the same plans every time the same experiment is conducted as it's based on random choices.

Generally, the results show clearly the effectiveness of the proposed models against PostgreSQL DBMS optimizer where execution plans produced by the proposed models needed less execution time and produced smaller intermediate results than the execution plans produced by PostgreSQL DBMS.

In addition, the deep reinforcement learning based model had performed equally as the regular reinforcement learning model and was able to generalize for new and unseen queries during the training phase. Deep reinforcement learning based model provides a great flexibility to run the proposed models on real environments rather than the restrictions presented in the Q-Table of the reinforcement learning model.

6) Training Overhead

In supervised learning, tracking the model performance and adjusting it during training can be done using a validation set. On the other hand, tracking reinforcement-based model performance during training can be a challenging task [4]. Reinforcement-based model training is tracked using the average penalty applied during different training episodes. Figure 18 shows the evolution of the cost function during the training process of the DQN model. The figure shows how the total cost decreases during training on the IMDB Database. During training, it was found that 1000 iterations would require 30 minutes assuming that the actual response time of each query plan exists in the buffer replay. The network was able to learn from previous experience by using the buffer reply which was used to store the latest N experience tuples observed by an agent where each experience tuple includes state, action, reward (response time), and next state. Once the data is stored in the buffer reply, the network can utilize it when required through learning without having to interact with the database management system again. This feature has a significant impact on the training time. For example, storing 1000 states in buffer replay with their rewards (response times) prevents us from calling the database management system again to retrieve the response time for each of the 1000 state. More specifically, if the average response time for a query in each of the 1000 states in the buffer reply is 30 seconds, the total required time to retrieve their response times is 8 hours. Preserving the states data in the buffer reply would save 8 hours of training time in case the states response times were required twice during the training.

VI. CONCLUSION

The process of finding the optimal join order is a complex problem as the number of possible join combinations that the optimizer needs to explore increases exponentially with the number of tables that would be impossible for the optimizer to analyze costs for all possible combinations. So, most optimizers use heuristics rules to prune the search space which helps the optimizer to balance between the optimization time and the plan quality. Besides, the optimizer uses pre-calculated statistics to estimate the cost of a plan that may mislead the optimizer to choose an inefficient plan. The planning time for traditional models increases as the number of join conditions increases. In this paper, we presented A Reinforcement Learning Based Query Optimizer (RL_QOptimizer), a new approach to recommend a query execution plan focusing on join ordering problems using reinforcement learning-based models. As the query execution time is the most crucial requirement from any query engine, execution time was used as a reward for the proposed reinforcement learning models. The achieved results of the performance evaluation show that the RL_QOptimizer outperforms PostgreSQL in choosing the best join orders for many queries and show how the Learning-based models can save query planning time. Also, the deep reinforcement learning model achieves very close results to the Q-learning model with a key advantage that it is more suitable for large number of states and actions as it uses a neural network to approximate the query execution time. In addition, the experiments show the ability of the DQN model to generalize to queries states that were never seen before during training phase which is another key advantage for deep reinforcement learning model.

The current models have a limited scope in that they can only handle join-only queries with no extra predicates. The study focuses on Join Ordering only because it is one of the most difficult problems in query optimization [2], and the study aims to demonstrate the concept of using Reinforcement Learning in Query Optimizer to replace the cost model by focusing on execution time, which will pave the way for more research to create a comprehensive end-to-end Reinforcement Learning Optimizer.

REFERENCES

[1] R. Elmasri and N. Shamkant, *Fundamentals of Database Systems*. San Francisco, CA, USA: Benjamin-Cummings Publishing, 2001.
[2] B. Nevarez, *Inside the SQL Server Query Optimizer*, C. Massey, Ed. Salford, U.K.: Hight Performance SQL Server, Simple Talk, 2010.
[3] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Princ. Database Syst. (PODS)*, 1998, pp. 34–43.
[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with deep reinforcement learning," in *Neural Inf. Process. Syst. Workshop*, 2013, pp. 1–9.
[5] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, nos. 3–4, pp. 279–292, 1992.
[6] S. Vellevt, "Review of algorithms for join ordering problem in database query optimization," *Inf. Technol. Control*, vol. 1, pp. 1312–2622, Jan. 2009.
[7] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1998.
[8] R. Bellman, "A Markovian decision process," *Indiana Univ. Math. J.*, vol. 6, no. 4, pp. 679–684, Apr. 1957.
[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.
[10] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electron. Imag.*, vol. 29, no. 19, pp. 70–76, Jan. 2017.
[11] V. Talpaert, I. Sobh, B. Kiran, P. Mannion, S. Yogamani, A. El-Sallab, and P. Perez, "Exploring applications of deep reinforcement learning for real-world autonomous driving systems," in *Proc. 14th Int. Joint Conf. Comput. Vis., Imag. Comput. Graph. Theory Appl.*, 2019, pp. 564–572.
[12] S. Ibrahim and D. Nevin, "End-to-end framework for fast learning asynchronous agents," in *Proc. 32nd Conf. Neural Inf. Process. Syst., Imitation Learn. Challenges Robot. Workshop (NIPS)*, 2018. [Online]. Available: https://sites.google.com/view/nips18-ilr#h.p_6wGpM-tJnQIU
[13] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1979, pp. 23–34.
[14] P. M. G. Apers, A. R. Hevner, and S. B. Yao, "Optimization algorithms for distributed queries," *IEEE Trans. Softw. Eng.*, vol. SE-9, no. 1, pp. 57–68, Jan. 1983.
[15] N. Bruno and S. Chaudhuri, "Exploiting statistics on query expressions for optimization," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2002, pp. 263–274.
[16] S. Christodoulakis, "Estimating selectivities in data bases," Univ. Toronto, Toronto, ON, Canada, Tech. Rep. CSRG-136, 1982.
[17] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis, "Parametric query optimization," *VLDB J. Int. J. Very Large Data Bases*, vol. 6, no. 2, pp. 132–151, May 1997.
[18] R. L. Cole and G. Graefe, "Optimization of dynamic query evaluation plans," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1994, pp. 150–160.
[19] S. Ganguly, "Design and analysis of parametric query optimization algorithms," in *Proc. 24rd Int. Conf. Very Large Data Bases*. New York, NY, USA, Aug. 1998, pp. 228–238.
[20] A. Hulgeri and S. Sudarsh, "Parametric query optimization for linear and piecewise linear cost functions," in *Proc. 28th Int. Conf. Very Large Data Bases*, Aug. 2002, pp. 167–178.
[21] P. Bizarro, N. Bruno, and D. J. DeWitt, "Progressive parametric query optimization," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 4, pp. 582–594, Apr. 2009.
[22] N. Reddy and J. R. Haritsa, "Analyzing plan diagrams of database query optimizers," in *Proc. VLDB Endowment*, Aug. 2005, pp. 1228–1239.
[23] B. J. Oommen, "The efficiency of histogram-like techniques for database query optimization," *Comput. J.*, vol. 45, no. 5, pp. 494–510, May 2002.
[24] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1996, pp. 294–305.
[25] X. Lu and J. Guan, "A new approach to building histogram for selectivity estimation in query processing optimization," *Comput. Math. With Appl.*, vol. 57, no. 6, pp. 1037–1047, Mar. 2009.
[26] J. Zahir and A. E. Qadi, "A recommendation system for execution plans using machine learning," *Math. Comput. Appl.*, vol. 21, no. 23, p. 23, Jun. 2016.
[27] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, "LEO-DB2's learning optimizer," in *Proc. 27th Int. Conf. Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann, Sep. 2001, pp. 19–28.
[28] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte, "Cardinality estimation using neural networks," in *Proc. 25th Annu. Int. Conf. Comput. Sci. Softw. Eng.* Riverton, NJ, USA: IBM Corp., Nov. 2015, pp. 53–59.
[29] R. Marcus and O. Papaemmanouil, "Deep reinforcement learning for join order enumeration," in *Proc. 1st Int. Workshop Exploiting Artif. Intell. Techn. Data Manage.*, Jun. 2018, pp. 1–4.
[30] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi, "Learning state representations for query optimization with deep reinforcement learning," in *Proc. Workshop Data Manage. End End Mach. Learn.*, no. 4, Jun. 2018, pp. 1–4.
[31] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *Proc. VLDB Endowment*, vol. 12, no. 11, pp. 1705–1718, Jul. 2019.
[32] O. Ivanov and S. Bartunov, "Adaptive cardinality estimation," 2017, *arXiv:1711.08330*.

[33] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica, "Learning to optimize join queries with deep reinforcement learning," 2018, *arXiv:1808.03196*.
[34] K. Tzoumas, T. Sellis, and C. S. Jensen, "A reinforcement learning approach for adaptive query processing," Inst. Datalogi, Aalborg Universitet, Aalborg, Denmark, 1DB Tech. Rep. 22, 2008. [Online]. Available: https://vbn.aau.dk/en/publications/a-reinforcement-learning-approach-for-adaptive-query-processing
[35] R. B. Guo and K. Daudjee, "Research challenges in deep reinforcement learning-based join query optimization," in *Proc. 3rd Int. Workshop Exploiting Artif. Intell. Techn. Data Manage.*, Jun. 2020, pp. 1–6.
[36] R. Marcus and O. Papaemmanouil, "Towards a hands-free query optimizer through deep learning," in *Proc. 9th Biennial Conf. Innov. Data Syst. Res., (CIDR)*, 2019, pp. 1–8.
[37] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis, "SkinnerDB: Regret-bounded query evaluation via reinforcement learning," in *Proc. Int. Conf. Manage. Data*, Jun. 2019, pp. 1153–1170.
[38] J. Heitz and K. Stockinger, "Join query optimization with deep reinforcement learning algorithms," 2019, *arXiv:1911.11689*.
[39] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 1275–1288.
[40] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "The case for automatic database administration using deep reinforcement learning," 2018, *arXiv:1801.05643*.
[41] J. Wang, I. Trummer, and D. Basu, "UDO: Universal database optimization using reinforcement learning," *Proc. VLDB Endowment*, vol. 14, no. 13, pp. 3402–3414, Sep. 2021.
[42] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.
[43] (1996). *Postgresql*. [Online]. Available: https://www.postgresql.org/
[44] D. P. Kingma and J. B. Adam, "A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, San Diego, CA, USA, 2015, pp. 1–15.
[45] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 204–215, Nov. 2015.
[46] (1993). *TPC-H*. [Online]. Available: http://www.tpc.org/tpch/
[47] (2015). *Tensorflow*. [Online]. Available: https://www.tensorflow.org
[48] (2015). *Keras*. [Online]. Available: https://keras.io/
[49] (1990). *Imdb*. [Online]. Available: https://www.imdb.com/
[50] M. Dreseler, M. Boissier, T. Rabl, and M. Uflacker, "Quantifying TPC-H choke points and their optimizations," *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1206–1220, Apr. 2020.

MOHAMED RAMADAN received the B.Sc. degree from the Information Systems Department, Faculty of Computers and Artificial Intelligence, Cairo University, in 2016. He is currently a Teaching Assistant and a Researcher at the Faculty of Computers and Artificial Intelligence, Cairo University. He has six years of experience in the area of software development.

AYMAN EL-KILANY received the M.Sc. and Ph.D. degrees from the Information Systems Department, Faculty of Computers and Artificial Intelligence, Cairo University, in 2012 and 2018, respectively. He is currently an Assistant Professor and a Researcher at the Faculty of Computers and Artificial Intelligence, Cairo University.

HODA M. O. MOKHTAR received the B.Sc. (Hons.) and M.Sc. degrees from the Department of Computer Engineering, Faculty of Engineering, Cairo University, in 1997 and 2000, respectively, and the Ph.D. degree in computer science from the University of California at Santa Barbara, in 2005. She is currently the Dean of the Faculty of Computing and Information Sciences, Egypt University of Informatics. Before being the Dean, she was the Chair of the Information Systems Department, Faculty of Computers and Artificial Intelligence, Cairo University. In 2000, she was awarded a scholarship and the Dean's Fellowship from the Computer Science Department, UCSB. She taught multiple courses both for the undergraduate and graduate levels at the Faculty of Computers and Artificial Intelligence, Cairo University, where she has supervised a number of master's and Ph.D. theses at the Faculty of Computers and Artificial Intelligence. She has participated in several national committees, and was awarded multiple awards and certificates for her academic achievements. Her research interests include big data analytics, data warehousing, data mining, database systems, social network analysis, bioinformatics, and web services.

IBRAHIM SOBH received the B.Sc. and M.Sc. degrees in computer engineering from the Faculty of Engineering, Cairo University, and the Ph.D. degree in deep reinforcement learning for fast learning agents acting in 3D environments. Currently, he is a Senior Expert of AI at Valeo. He has more than 20 years of experience in the area of machine learning and software development. His M.Sc. thesis is in the field of machine learning applied on automatic documents summarization. He has participated in several related national and international mega projects, conferences and summits. He delivers training and lectures for academic and industrial entities. His publications including international journals and conference papers are mainly in the machine and deep learning fields. His research interests include computer vision, natural language processing, and speech processing.