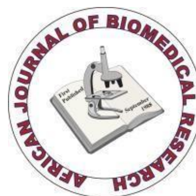


https://africanjournalofbiomedicalresearch.com/index.php/AJBR
Afr. J. Biomed. Res. Vol. 27(4s) (November 2024); 3295-3307
Research Article

Comprehensive Investigation Of Join Query Optimization Using Dqn (Deep Q-Network), Ddqn (Double Deep Q-Network), Genetic Algorithms And Hybrid Dqn-Ga And Ddqn –Ga

Karthikeyan M P[1], Dr.K. Krishnaveni[2]

[1]Research Scholar, Department of Computer Science, Sri. S. Ramasamy Naidu Memorial College (Affiliated to Madurai Kamaraj University, Madurai), Sattur, Virudhunagar District, Tamil Nadu 626203, India, Email: karthi.karthis@gmail.com
[2]Associate Professor & Head, Department of Computer Science, Sri. S. Ramasamy Naidu Memorial College (Affiliated to Madurai Kamaraj University, Madurai), Sattur, Virudhunagar District, Tamil Nadu 626203, India, Email: kkrishnaveni@srnmcollege.ac.in

Abstract: Effective query optimization is essential for improving database management system performance. Using a variety of cutting-edge approaches, including Deep Q-Networks (DQN), Double Deep Q-Networks (DDQN), Genetic Algorithms (GA), and Hybrid DQN-GA and DDQN-GA, we present a thorough examination of join query optimization in this research article. In relational databases, join queries are frequently used to merge data from several tables. The issue of optimizing join queries to reduce response time and resource use is still difficult. The reinforcement learning-inspired DQN and DDQN algorithms offer a framework for teaching agents to make the best judgments possible in dynamic situations. In order to discover effective query execution techniques, we use the power of DQN and DDQN to formulate join query optimization as a Markov Decision Process (MDP). We also present genetic algorithms as a different strategy for searching the space of join query plans. In this study, we examine the query execution times and resource use of DQN, DDQN, GA, and hybrid DQN-GA and DDQN-GA approaches. The efficacy of each strategy is evaluated experimentally on a wide range of Join Order benchmark (JOB) datasets. Our findings show that as compared to conventional methods, hybrid DDQN-GA based techniques significantly enhance query optimization. In order to maximize the benefits of both algorithms, we also explore the pairing of DQN and GA. The hybrid DQN-GA technique outperforms individual algorithms in terms of Query Execution Time, Query Latency, Resource Utilization, Optimization Latency, and Join Query Performance; demonstrating greater performance in optimizing joins queries. Similarly, the hybrid DDQN-GA approach also presents promising results.

Keywords: Join Query Optimization, Deep Q-Network, Double Deep Q-Network, Genetic Algorithms, Markov Decision Process, Query Execution Time, Query Latency

***Author for correspondence:** *Email: karthi.karthis@gmail.com*

Received 19/11/2024, Acceptance 25/11/2024

DOI: https://doi.org/10.53555/AJBR.v27i4S.4185

© 2024 The Author(s).
This article has been published under the terms of Creative Commons Attribution-Noncommercial 4.0 International License (CC BY-NC 4.0), which permits noncommercial unrestricted use, distribution, and reproduction in any medium, provided that the following statement is provided. "This article has been published in the African Journal of Biomedical Research"

I.INTRODUCTION

In a relational database management system (RDBMS), join query optimization is the process of choosing the most effective execution plan for queries including join procedures. The Join operation combines the rows from the various tables when there are many tables involved

in the query. The goal of join query optimization is to reduce computational overhead and boost the efficiency of these join processes. When working with huge datasets, join procedures in database systems may be computationally costly. Database systems may execute queries more quickly by optimizing the join queries, which enables applications to deliver timely and responsive results to end users [1]. In situations where complicated queries or queries involving many tables are typical, join query optimization is very important. These queries are often used in decision support systems, business intelligence software, and data analytics, all of which require excellent query processing in order to analyze data effectively and make decisions [2].

A few advantages of effective join query optimization are [3]:

- **Shorter query processing time:** By choosing the best join technique and execution plan, the overall query processing time may be cut in half, resulting in a more responsive system.
- **Better resource utilization:** By ensuring optimum use of system resources including CPU, memory, and disc I/O, join query optimization enhances scalability and efficiency.
- **Improved user experience:** By giving users prompt access to desired information, quicker query response times improve user experience.
- **Enhanced system throughput:** By allowing the database system to accommodate more concurrent queries, optimized join queries enhance the system's overall throughput and performance.

Overall, join query optimization is essential for maximizing the effectiveness and speed of database systems, enabling businesses to properly utilize their data and extract insightful knowledge from sizable databases.

Challenges in Join Query Optimization

For successful query processing, join query optimization offers a number of issues that must be resolved [4]. These difficulties include:

- **Query Complexity:** When numerous tables are involved and various join criteria need to be fulfilled, join queries can become very complicated. Finding the best execution strategy is difficult because of the combinatorial proliferation of join orders and join algorithms that might be used.
- **Cost Prediction:** For choosing the most effective execution strategy, it is essential to accurately estimate the costs of various join procedures. However, due to variables in data distribution, errors in statistics, and dynamic changes in the database environment, cost estimation might be difficult. Inaccurate cost estimates may result in execution plans that are less than ideal and poor query performance.
- **Scalability of the optimizer:** Join query optimization gets more difficult as database size and query complexity increase. In a fair length of time, optimizers must sort through a huge number of alternative join pathways and select the optimum course of action. The search space grows exponentially as the number of tables and joins criteria rises, making the optimization procedure computationally costly.
- **Join Cardinality assessment:** For cost assessment and plan selection, it is essential to estimate the cardinality (number of rows) arising from a join operation. However, when join predicates contain complicated conditions, correlations, or inequalities, correct cardinality estimate becomes challenging. The execution of queries inefficiently and improper join techniques can both be caused by inaccurate cardinality estimations.
- **Dynamic optimization:** Database environments vary over time, affecting the workload, statistics, and distribution of data. To maintain peak performance, the optimizer must respond to these changes and dynamically modify the execution plans. However, it might be difficult to dynamically optimize join queries while maintaining minimum query interruption.
- **Join Order Optimization:** How the tables are connected may have a big influence on how fast a query runs. It is an NP-hard task to choose the best join order from the increasingly huge search space. The ideal join order is approximated by traditional query optimizers using heuristics and cost-based methods, but identifying the globally optimum solution is still difficult [5].
- **Index Selection:** Effective usage of indexes is essential for optimizing join queries. It can be difficult to choose the right indexes for join predicates and reduce the amount of index lookups, though. To choose the best index, the optimizer must take into account the distribution of the data, the cost of index access, and the selectivity of the predicates.

Advanced algorithms, statistical models, and optimization methods are needed to tackle these problems. To increase join query optimization and get beyond these obstacles, researchers and practitioners are always coming up with new strategies.

Join query optimization; in particular, which entails choosing the best join algorithms and the best join order in order to reduce execution time and resource use. The use of genetic algorithms (GAs) to solve this optimization challenge is a potential solution. This article examines the use of a genetic algorithm for join query optimization and provides a summary of the essential procedures. In order to explore the solution space and identify nearly optimum query plans, genetic algorithms provide a potent method for join query optimization. It is feasible to find effective join orders and selections by specifying suitable chromosomal representations, fitness functions, and genetic operators. Although a genetic algorithm for join query optimization has to be put into place with careful thought and system-specific modifications, it has a great deal of promise to enhance the efficiency of complicated database queries. However, recent developments in artificial intelligence have made it possible to

investigate innovative approaches. This article examines the use of Deep Q Network (DQN), a reinforcement learning technology, for join query optimization, highlighting the possible advantages and explaining the essential procedures. By fusing deep learning methods with Q-learning, Deep Q Network (DQN) revolutionized reinforcement learning. It is a potent algorithm for resolving challenging decision-making issues because of its capacity to manage high-dimensional state spaces and learn from unprocessed sensory data. DQN offers a powerful framework for instructing agents to optimize their behaviors in dynamic contexts through the use of deep neural networks, experience replay, and target networks. DQN has been and will continue to be a crucial algorithm in the field of reinforcement learning thanks to continuous research and new developments. In the context of choosing the best query execution plans, Deep Q Network (DQN) may also be used for query optimisation. Three widely used algorithms—DQN (Deep Q-Network), DDQN (Double Deep Q-Network), and Genetic Algorithms—have been used to optimise join queries. Each of these algorithms has unique strengths and shortcomings and takes a different angle on the issue.

II. REINFORCEMENT LEARNING AND DEEP Q-NETWORK (DQN)

Reinforcement learning (RL) is a subfield of machine learning that focuses on learning optimal actions in an environment to maximize a cumulative reward. It is inspired by how humans and animals learn through trial and error interactions with their surroundings. In RL, an agent learns to make sequential decisions by exploring the environment, taking actions, and receiving feedback in the form of rewards or penalties [6].

The core components of reinforcement learning include:

- **Agent:** The learner or decision-maker that interacts with the environment.
- **Environment:** The external context in which the agent operates, providing feedback and state transitions based on agent actions.
- **State:** The current representation of the environment, capturing relevant information for decision-making.
- **Action:** The decision or choice made by the agent at a particular state.
- **Reward:** The numerical feedback provided by the environment to reinforce or discourage certain actions. The agent aims to maximize the cumulative reward over time.

Reinforcement learning algorithms utilize the concept of Markov Decision Processes (MDPs) to model the interaction between the agent and the environment. MDPs define the state space, action space, transition probabilities, and reward functions that characterize the environment [7].

Deep Q-Network (DQN) Algorithm

A well-known reinforcement learning system called Deep Q-Network (DQN) combines Q-learning, a traditional RL algorithm, with deep neural networks. In 2013, DeepMind unveiled DQN, which attracted attention for playing Atari games at a superhuman level. The main concept of DQN is to approximate the Q-value function, which calculates the expected cumulative reward for performing a certain action in a specific condition [8]. The Q-network, a kind of neural network, is generally used to represent the Q-value function. The state is the Q-network's input, and its output is a projected vector of Q-values for each potential action.

Experience replay and a target network are used by the DQN algorithm to increase stability and learning effectiveness. The agent's experiences (state, action, reward, and future state) are stored in a replay buffer during experience replay, which is randomly sampled throughout training to break correlations between successive events. In order to lower the target estimate errors, the target network is a distinct copy of the Q-network that is routinely updated with the Q-network's weights. DQN balances investigating novel activities with using the acquired information by using an exploration-exploitation method like epsilon-greedy. This enables the agent to find the best policies without becoming bogged down in inefficient behaviour.

DQN is an approach for reinforcement learning that uses a neural network to simulate the Q-value function. To choose the best join order for a particular query, it employs an iterative process of exploration and exploitation [9]. DQN has the benefit of being able to handle dynamic and complicated settings, but it frequently needs a lot of training data and can be costly computationally. The Q-learning technique and a neural network approximation are integrated in the Deep Q-Network (DQN) formula. The predicted cumulative reward for performing a certain action in a specific condition is represented by the Q-value. The Q-network is a type of neural network used by the DQN method to estimate the Q-values. The DQN formula is as follows:

$$Q(s, a) = R + \gamma * \max(Q(s', a'))$$

Where:

- $Q(s, a)$ represents the Q-value for state s and action a.
- R is the immediate reward obtained after taking action a in state s.
- γ (gamma) is the discount factor that determines the importance of future rewards. It ranges between 0 and 1, where a value of 0 means only considering immediate rewards and a value of 1 means considering all future rewards.
- $\max(Q(s', a'))$ represents the maximum Q-value among all possible actions a' in the next state s'.

III. JOIN QUERY OPTIMIZATION USING DEEP Q-NETWORK

The capacity of DQN to learn from experience and adapt to dynamic and complicated contexts is its main advantage when it comes to join query optimization. Traditional optimization methods frequently rely on static cost estimates and assumptions, which could not accurately reflect the complexities of executing queries in the actual world. DQN, on the other hand, has the

ability to adjust its policy based on previous query execution data in order to enhance performance over time [10] [11]. A DQN agent may understand the underlying patterns and linkages between tables as well as the effects of various join techniques on query execution speed by being trained to explore the state-action space of join query optimization. By maximizing the projected long-term gains associated with execution time and resource utilization, the agent continually improves its decision-making process. DQN usage in join query optimization has a number of possible advantages. First, by identifying more effective join designs, it has the potential to outperform conventional optimization approaches. Additionally, it is flexible enough to adjust to shifting workloads and data distributions, continually enhancing query speed. Finally, because the DQN agent automatically learns to optimize queries based on past data, it lessens the need for human query tuning. In this study, we examine how DQN may be used to optimize join queries. We examine its efficacy in reducing query execution time and resource use as compared to conventional methods. We also examine how the performance of DQN-based optimization is impacted by variables like database size, query complexity, and cardinality estimate precision.

Algorithm 1: Join Query Optimization using Deep Q-Network (DQN)

Step 1: Initialization

- Initialize the deep Q-network with random weights: $\theta \leftarrow$ random initialization
- Define the replay memory to store experiences for training: $D \leftarrow \emptyset$
- Set the exploration and exploitation parameters, such as epsilon for the epsilon-greedy policy: $\varepsilon \leftarrow$ initial exploration rate

Step 2: Query and Join Space Representation

- Represent the join queries and join plans in a suitable format, such as a matrix or a graph.
- Encode the join queries and join plans with appropriate features, such as selectivity, cardinality, and join conditions.
  - *Query Representation:* $Q = [q_1, q_2, ..., q_n]$, where q_i represents the i-th join query.
  - *Join Plan Representation:* $P = [p_1, p_2, ..., p_m]$, where p_i represents the i-th join plan.
  - *Join Query Encoding:* $q_i = [f_1, f_2, ..., f_k]$, where f_j represents the j-th feature of the join query.
  - *Join Plan Encoding:* $p_i = [g_1, g_2, ..., g_l]$, where g_j represents the j-th feature of the join plan.

Step 3: State Representation

- Encode the current state of the join optimization problem.
- Formulate the state representation using relevant features, such as the current join plan, selectivity estimates, and available join operators.
  - *State Representation:* $S = [s_1, s_2, ..., s_p]$, where s_i represents the i-th feature of the state.
  - *Join Plan Representation:* $s_i = [g_1, g_2, ..., g_l]$, where g_j represents the j-th feature of the current join plan.
  - *Selectivity Estimate:* $s_i = f(e_1, e_2, ..., e_n)$, where e_k represents the estimate for the selectivity of join predicate k.
  - *Available Join Operators:* $s_i = [o_1, o_2, ..., o_m]$, where o_j represents the j-th available join operator in the current state.

Step 4: Action Selection

- Use the deep Q-network to select an action (join operator or join order) based on the current state.
- Calculate the Q-values for each possible action using the deep Q-network's forward pass.
- Select the action with the highest Q-value or choose a random action for exploration based on the exploration-exploitation policy.
  - *Q-value Calculation:* $Q(s, a) = f\theta(s, a)$, where $f\theta$ represents the deep Q-network's forward pass.
  - *Epsilon-Greedy Policy:* $\pi(a|s) = (1-\varepsilon) * \text{argmax}(Q(s, a)) + \varepsilon * \text{random}(a)$, where ε is the exploration parameter.

Step 5: Query Execution and Feedback

- Execute the selected action in the join query execution environment.
- Measure the performance metric, such as query execution time or cost, for the executed join plan.
- Calculate the immediate reward based on the performance metric using a suitable reward function.
  - *Performance Metric:* $M = f(q, p)$, where M represents the performance metric, q is the join query, and p is the executed join plan.
  - *Immediate Reward:* $R = f(M)$, where R represents the immediate reward based on the performance metric.
  - *Reward Function:* $R = f(M)$, where f represents the reward function mapping the performance metric to the reward value.

Step 6: Update Replay Memory

- Store the current state, selected action, immediate reward, and the resulting state in the replay memory.
  - *Replay Memory Update*: $D \leftarrow D \cup \{(s, a, r, s')\}$, where D represents the replay memory, s is the current state, a is the selected action, r is the immediate reward, and s' is the resulting state.

Step 7: Experience Replay

- Sample a batch of experiences from the replay memory.
- Perform a backward pass through the deep Q-network to update the weights using the loss function and gradient descent.
  - *Sample Batch of Experiences:* B = Sample(D, batch_size), where B represents the batch of experiences sampled from the replay memory D, and batch_size is the desired size of the batch.
  - *Loss Function:* $L(\theta) = \sum(s, a, r, s')\in B\ (Q(s, a) - (r + \gamma * \max(Q(s', a'))))^2$, where θ represents the weights of the deep Q-network, Q(s, a) represents the predicted

Q-value for state s and action a, r is the immediate reward, γ is the discount factor, and max(Q(s', a')) represents the maximum Q-value for the resulting state s' over all possible actions a'.

- *Gradient Descent:* θ ← θ - α * ∇θL(θ), where α is the learning rate and ∇θL(θ) represents the gradient of the loss function with respect to the weights θ.

Step 8: Repeat Steps 3-7

- Repeat Steps 3-7 until the termination condition is met (e.g., a certain number of iterations or convergence).

Step 9: Join Plan Selection

- Use the trained deep Q-network to select the optimal join plan based on the learned Q-values.
- Select the join plan with the highest Q-value as the optimized join plan for the given join query.
  - *Join Plan Selection:* p* = argmax(Q(s, a)), where p* represents the optimal join plan, Q(s, a) represents the Q-value for state s and action a, and argmax selects the action with the highest Q-value.

Step 10: Termination the algorithm

By discovering the best join plan to use, the Join Query Optimization technique employing Deep Q-Network (DQN) seeks to improve the execution of join queries. The DQN and replay memory are first initialized by the algorithm. The join queries and join plans are then represented in an appropriate format and encoded with pertinent properties. Features such as the current join plan, selectivity estimates, and available join operators are used to encapsulate the present status of the join optimization issue. By computing Q-values through a forward pass, the DQN is used to choose an action (join operator or join order) depending on the present state. The algorithm carries out the chosen action, assesses the performance metric (such as the time it takes to run a query), and then determines the instant reward using a reward function. The replay memory stores the experience (state, action, reward, and outcome state). The approach uses a loss function and gradient descent to update the DQN's weights on a regular basis by sampling a batch of events from the replay memory. Up to convergence, this procedure iteratively continues. The best join plan is then chosen by the trained DQN by choosing the action with the highest Q-value. Overall, this approach makes use of deep reinforcement learning to enhance database system effectiveness and optimize join query execution [12].

IV. JOIN QUERY OPTIMIZATION USING DOUBLE DEEP Q-NETWORK

In order to solve the overestimation of Q-values seen in the original DQN method, DDQN is an extension of DQN. It reduces overestimation bias by decoupling the selection of actions from their assessment using two different neural networks. When compared to DQN, DDQN can offer more precise Q-value estimations, perhaps resulting in better join order choices. The computational burden brought on by DQN still affects it, though. Heuristics or cost-based estimations are frequently used in traditional optimization approaches, which may not necessarily result in optimum solutions. [13] [14] [15]. Deep reinforcement learning has advanced, and there is considerable interest in using it for join query optimization. The Double Deep Q-Network (DDQN) is one such method that makes use of the strength of deep Q-networks and experience replay to boost the effectiveness and stability of the optimization process. DDQN has the capacity to learn and adapt to difficult join query optimization challenges by fusing the capabilities of reinforcement learning and neural networks, resulting in more precise and effective query execution plans. In this study, we investigate and evaluate the efficiency of Join Query Optimization using DDQN, contrasting it with other deep reinforcement learning techniques as well as conventional optimization techniques. We seek to give insights into the performance and advantages of DDQN in developing join query optimization approaches through rigorous tests and assessments.

Algorithm 2: Join Query Optimization using Double Deep Q-Network (DDQN)

Step 1: Initialization

- Initialize the primary and target deep Q-networks with random weights: θ_primary ← random initialization, θ_target ← θ_primary
- Define the replay memory to store experiences for training: D ← ∅
- Set the exploration and exploitation parameters, such as epsilon for the epsilon-greedy policy: ε ← initial exploration rate

Step 2: Query and Join Space Representation

- Represent the join queries and join plans in a suitable format, such as a matrix or a graph.
- Encode the join queries and join plans with appropriate features, such as selectivity, cardinality, and join conditions.

Step 3: State Representation

- Encode the current state of the join optimization problem using relevant features, such as the current join plan, selectivity estimates, and available join operators.

The state representation, denoted as s, can be defined as a vector of features:

s = [f1, f2, f3, ..., fn],

Where fi represents a specific feature related to the join optimization problem. These features can include:

- *Current join plan:* f1 represents the current join plan chosen for the join query.
- *Selectivity estimates:* f2 represents the estimated selectivity of each join predicate or condition.
- *Available join operators:* f3 represents the available join operators that can be used in the join plan.

Step 4: Action Selection

- Use the primary deep Q-network to select an action (join operator or join order) based on the current state.

- Calculate the Q-values for each possible action using the primary deep Q-network's forward pass.
- Select the action with the highest Q-value or choose a random action for exploration based on the exploration-exploitation policy.

Given the current state s, the primary deep Q-network (Q-network) can estimate the Q-values for each possible action a using a forward pass:

$Q(s, a; \theta_primary) = Q\text{-network}(s, a; \theta_primary)$,

Where θ_primary represents the weights of the primary deep Q-network. The action selection can be done using an exploration-exploitation policy, such as epsilon-greedy, which selects the action with the highest Q-value with a probability of $1 - \varepsilon$, or a random action with a probability of ε.

Step 5: Query Execution and Feedback

- Execute the selected action in the join query execution environment.
- Measure the performance metric, such as query execution time or cost, for the executed join plan.
- Calculate the immediate reward based on the performance metric using a suitable reward function.

The performance metric, denoted as M(a), can be measured for the executed join plan associated with action a. This metric could be the query execution time, cost, or any other relevant measure. The immediate reward, denoted as r, can be calculated based on the performance metric using a reward function R(a):

$r = R(a) = f(M(a))$,

Where f(M(a)) represents a mapping function that maps the performance metric M(a) to a reward value. The specific form of the reward function depends on the optimization goal and the desired behavior of the join query optimizer.

Step 6: Update Replay Memory

- Store the current state, selected action, immediate reward, and resulting state in the replay memory.

The replay memory, denoted as D, is updated by adding this transition to the memory:

$D \leftarrow D \cup \{(s, a, r, s')\}$.

Let (s, a, r, s') represent a transition, where:

- s is the current state,
- a is the selected action,
- r is the immediate reward, and
- s' is the resulting state after executing the action a.

Step 7: Experience Replay

- Sample a batch of experiences from the replay memory.
- Perform a backward pass through the primary deep Q-network to update the weights using the loss function and gradient descent.
- Update the target deep Q-network periodically by copying the weights from the primary deep Q-network.
  - Compute the target Q-value, denoted as target, for each sampled transition:

$target = r + \gamma * \max(Q(s', a'; \theta_target))$,

Where θ_target represents the weights of the target deep Q-network and a' represents the action selected by the target network for the next state s'.

- Compute the predicted Q-value, denoted as predicted, for each sampled transition:

$predicted = Q(s, a; \theta_primary)$,

Where θ_primary represents the weights of the primary deep Q-network.

- Calculate the loss, denoted as L, using a suitable loss function, such as the mean squared error:

$L = 1/N * \text{sum}((target - predicted)^2)$,

Where N is the batch size.

- Perform a backward pass through the primary deep Q-network to update the weights using gradient descent:

$\theta_primary \leftarrow \theta_primary - \alpha * \nabla(L; \theta_primary)$,

Where α is the learning rate.

- Periodically update the target deep Q-network by copying the weights from the primary deep Q-network:

$\theta_target \leftarrow \theta_primary$.

Step 8: Repeat Steps 3-7 until convergence.

Step 9: Join Plan Selection

- Use the trained primary deep Q-network to select the optimal join plan based on the learned Q-values.
- Select the join plan with the highest Q-value as the optimized join plan for the given join query.

To select the optimal join plan, we evaluate the Q-values for all possible actions (join plans) given the current state s and choose the action with the highest Q-value:

$a_optimal = \text{argmax}(Q(s, a; \theta_primary))$,

Where a_optimal represents the selected optimal action (join plan) with the highest Q-value. Let Q(s, a; θ_primary) represent the Q-value function of the trained primary deep Q-network, where s is the state and a is the action.

Step 10: Return the optimized join plan.

The Join Query Optimisation using Double Deep Q-Network (DDQN) technique is a multi-step process. Initially, a replay memory is constructed to record training events, and the primary and target deep Q-networks are initialised with random weights. The join plans and queries are suitably represented and encoded. The algorithm then selects an action based on the main deep Q-network, executes the query, receives feedback, and calculates an instantaneous reward based on the performance metric. With the current state, chosen action, immediate reward, and resulting state, the algorithm updates the replay memory. Gradient descent is used to update the weights of the main deep Q-network while sampling batches of events from the replay memory.

The weights from the primary network are regularly copied to the target deep Q-network. Finally, based on the learnt Q-values, the best join plan is chosen using the trained primary deep Q-network. The algorithm's overall goal is to improve the performance of join queries by repeatedly learning and updating the Q-values to inform the choice of join operators and join orders. This technique improves the stability and

learning effectiveness of the optimization process by applying the Double Deep Q-Network (DDQN) approach to join query optimization. It successfully balances exploration and exploitation and reduces overestimation bias, resulting in enhanced join query execution efficiency. This is accomplished by using two deep Q-networks and experience replay.

V.JOIN QUERY OPTIMIZATION USING GENETIC ALGORITHMS

Particularly for big and sophisticated queries, traditional optimization approaches frequently find it difficult to manage the combinatorial search space of join orders and operator selections. The use of Genetic Algorithms (GAs) to solve these optimization issues has shown promise. The concepts of natural selection and evolution serve as the foundation for GAs, which employ a population-based search technique to repeatedly explore the search space and identify nearly ideal solutions [16]. In order to assess the usefulness and efficiency of GAs in identifying the best join plans, this study offers an investigation of Join Query Optimization using GAs. In order to evaluate their influence on the optimization process, the study investigates alternative genetic operators, fitness functions, and encoding strategies. The findings gave database managers and academics studying query optimization useful information on the advantages and disadvantages of GAs in join query optimization. A population-based search technique called genetic algorithms was inspired by natural evolution. A GA would represent various join orders as individuals in a population when used in the context of join query optimization. To create new join orders, the individuals go through genetic procedures like crossover and mutation. The query cost is used to evaluate fitness, and the procedure is repeated until the best join order is identified. GA has the benefit of swiftly navigating a big search field and is able to deal with noise in the fitness environment [17] [18]. However, it can be delicate to parameter settings and may have trouble with very large join spaces.

Algorithm 3: Join Query Optimization using Genetic Algorithm

Step 1: Initialization

- Initialize a population P with random individuals representing potential join plans.
- Set the maximum number of generations (max_generations), maximum population size (max_population), and other algorithm parameters.

P = {individual_1, individual_2, ..., individual_n},

Where P represents the population, and individual_i represents the i-th join plan in the population. The population size is determined by n, which is the maximum population size specified.

Step 2: Fitness Evaluation

- Evaluate the fitness of each individual in the population based on a fitness function.
- Calculate the fitness value, denoted as F, which represents the quality of the join plan.
  - For each individual in the population, calculate its fitness value using the fitness function:

F = fitness (individual),

where F represents the fitness value and individual represents an individual join plan in the population. The fitness function quantifies the quality or performance of the join plan based on specific criteria, such as execution time, resource utilization, or cost estimation accuracy. The fitness value serves as a measure of how well the join plan performs compared to other individuals in the population.

Step 3: Selection

- Select individuals from the population for the next generation based on their fitness values.
- Apply selection techniques such as tournament selection or roulette wheel selection.
  - Calculate the selection probability for each individual in the population:

P_selection(individual) = F(individual) / sum(F),

Where P_selection represents the selection probability, F(individual) is the fitness value of the individual, and the sum(F) is the sum of fitness values for all individuals in the population.

  - Select individuals for the next generation based on their selection probabilities. This selection process can be performed using various strategies, such as tournament selection or roulette wheel selection, where individuals with higher selection probabilities have a higher chance of being selected.

Step 4: Crossover

- Perform crossover operations between selected individuals to create offspring.
- Combine genetic material from parent individuals to generate new join plans.
  - Choose parent individuals for crossover based on the selected individuals.
  - Apply crossover operators to create offspring individuals. One commonly used crossover operator is single-point crossover, where a random point is selected along the genetic material (join plan representation) of the parents. The genetic material beyond that point is swapped between the parents to create two offspring individuals. Single-Point Crossover:

Offspring_1 = Parent_1[:crossover_point] + Parent_2[crossover_point:]

Offspring_2 = Parent_2[:crossover_point] + Parent_1[crossover_point:]

Where Offspring_1 and Offspring_2 represent the resulting offspring individuals, Parent_1 and Parent_2 are the selected parent individuals, and crossover_point is the randomly chosen point for crossover along the genetic material. The genetic material before the crossover point is taken from one parent, and the genetic material after the crossover point is taken from the other parent to create the offspring individuals.

Step 5: Mutation

- Introduce random changes to the offspring individuals to promote exploration of the search space.

- Apply mutation operators to modify certain characteristics or parameters of the join plans.
  - Select offspring individuals for mutation.
  - Apply mutation operators to modify the join plans. One common mutation operator is bit-flip mutation, where a random bit in the genetic material (join plan representation) of an individual is flipped, introducing a small change in the join plan. Bit-Flip Mutation:

Mutated_individual = Individual[:mutation_point] + (1 - Individual[mutation_point]) + Individual[mutation_point+1:]

Where Mutated_individual represents the resulting mutated individual, Individual is the selected offspring individual, and mutation_point is the randomly chosen point for mutation along the genetic material. The bit at the mutation point is flipped (from 0 to 1 or from 1 to 0) to introduce a small change in the join plan. Other mutation operators, such as swap mutation or inversion mutation, can also be applied based on the specific encoding and characteristics of the join plan representation.

Step 6: Replacement

- Replace a portion of the population with the newly created offspring individuals.
- Select individuals from the population to be replaced based on certain replacement strategies, such as elitism or generational replacement.
  - Select individuals from the population to be replaced. The replacement strategy can vary, but common approaches include elitism, where the best individuals are preserved in the population, or generational replacement, where the entire population is replaced by the offspring individuals.
  - Replace the selected individuals with the newly created offspring individuals, ensuring the population size remains constant. This step ensures that the population evolves over generations and adapts to better join plans based on the performance and quality of the offspring individuals.

Step 7: Termination

- Check if termination criteria are met, such as reaching a maximum number of generations or achieving a satisfactory fitness level. If the termination criteria are met, stop the algorithm; otherwise, go to Step 2. The algorithm continues to iterate through Steps 2 to 7 until the termination criteria are satisfied.

The goal of the search-based technique for join query optimization using genetic algorithms is to identify the best join strategy for processing a query in a relational database. Initializing a population of probable join plans is the first step in the algorithm. In the population, each join plan is portrayed as an individual [19]. Each person's fitness is assessed using a fitness function that gauges the effectiveness or performance of the Join plan. The next phase is selection, which involves picking people for reproduction based on fitness values.

To produce new offspring, operators for crossover and mutation are applied to the chosen individuals. In contrast to mutation, which introduces random alterations to encourage the exploration of the search space, crossover integrates the genetic material from the parents. The last phase involves replacing a section of the population with individuals from the newly produced children. Until a termination condition is satisfied, such as reaching a maximum number of generations or attaining the target level of fitness, this process keeps on for a number of generations. The method looks for join plans that optimize query performance based on the provided fitness criteria through the iterative development of the population.

VI. HYBRID APPROACHES: DQN-GA AND DDQN-GA

In join query optimization, hybrid approaches combine different optimization techniques to leverage their respective strengths and address their limitations. Two hybrid approaches that have been explored are the combination of Deep Q-Network (DQN) with Genetic Algorithms (GA) and the combination of Double Deep Q-Network (DDQN) with Genetic Algorithms (GA).

6.1. DQN-GA: Combining Deep Q-Network with Genetic Algorithms

The DQN-GA hybrid technique combines the search and optimization skills of Genetic Algorithms (GA) with the reinforcement learning capabilities of Deep Q-Networks (DQN). The idea is to take use of both approaches' skills for exploration and exploitation. The DQN component of DQN-GA is used to learn and approximatively determine the Q-values of various join operations. The DQN network gets the join query's current state and produces Q-values for every potential join operation. During the genetic algorithm optimization phase, these Q-values direct the investigation and utilization of join operations. The GA component uses genetic operators including selection, crossover, and mutation to act on a population of candidate join orders or execution plans. The DQN is used to assess each candidate solution's fitness in order to calculate the anticipated cumulative rewards. The GA develops the population to discover improved join query optimization strategies by examining the join order search space while being led by the Q-values supplied by the DQN.

6.2.DDQN-GA: Combining Double Deep Q-Network with Genetic Algorithms

The DDQN-GA hybrid technique improves on the DQN-GA approach while including DDQN's (Double Deep Q-Networks) improvements. With DDQN, the overestimation bias problem in conventional DQN algorithms is resolved, leading to more precise Q-value estimations. In DDQN-GA, a DDQN that has a separate target network utilized for Q-value estimate is employed in place of the DQN component. The target network aids in reducing variation in Q-value predictions and stabilizing the learning process. The DDQN-GA hybrid technique makes use of the enhanced Q-value estimations to direct the exploration and application of

the genetic algorithm in order to identify the best join query optimization options.

6.3. Benefits and Advantages of Hybrid Approaches

In terms of join query optimization, hybrid methods like DQN-GA and DDQN-GA have various advantages:

- **Combining strengths:** Hybrid systems can take use of the exploration powers of reinforcement learning and the optimization capabilities of genetic algorithms by integrating several techniques.
- **Faster convergence:** The convergence rate and efficacy of the optimization process can both be improved by combining various techniques.
- **Handling difficult situations:** Hybrid methods are very helpful in tackling complex join query optimization issues that might not be effectively resolved by individual strategies alone.
- **Adaptability:** Hybrid techniques can adjust to changes in the workload, database environment, or query characteristics, resulting in a gradual improvement in performance.

Joins query optimization research continues to focus on hybrid techniques, with attempts being made to improve their designs, optimize their parameters, and assess their performance in diverse real-world circumstances.

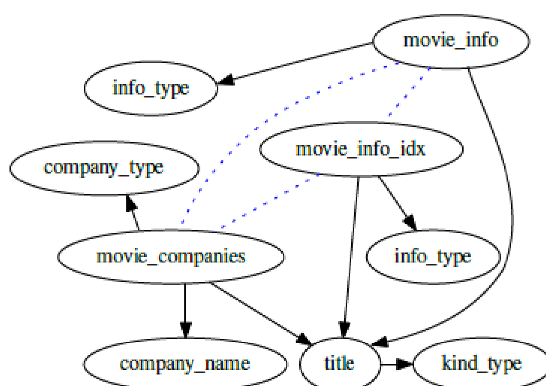
VII.PERFORMANCE EVALUATION

Deep Q-Network (DQN), Double Deep Q-Network (DDQN), Genetic Algorithms (GA), and hybrid DQN-GA and DDQN-GA are used to optimise join queries. The Join Order Benchmark (JOB), a collection of queries utilised in earlier evaluations of query optimizers, is employed in this experiment. The benchmark uses the IMDB dataset for 113 query instances over 33 query forms. A virtual computer that already has the dataset loaded into it has been made.There are 13 a to d relations that each query connects. 4 more randomly chosen inquiries are also included in our testing query collection, along with all occurrences of one randomly chosen query template. PostgreSQL [20] on a virtual machine with two cores, eight gigabytes of RAM, and a maximum shared buffer pool size of one gigabyte resulted in a database with a total size of 11GB (all primary and foreign keys are indexed). Instead of having PostgreSQL use its own join enumerator, we configured it to use the join ordering produced by join.

Scenario 1: In this research, we analyze the Join Order Benchmark (JOB), which employs 113 sophisticated join queries and operates on real-world data rife with correlations. Using a complicated, real-world data set and plausible multi-join queries, we empirically review the key elements in the traditional query optimizer design. We selected the Internet Movie Data Base (IMDB) as a synthetic data collection. It is jam-packed with details on films, as well as associated information about actors, directors, production companies, etc. The information is offered as text files for free2 non-commercial use. In addition, we converted the text files into a relational database using the open-source imdbpy3 software. Example JOB 13d determines the reviews and dates of all films made by US firms.

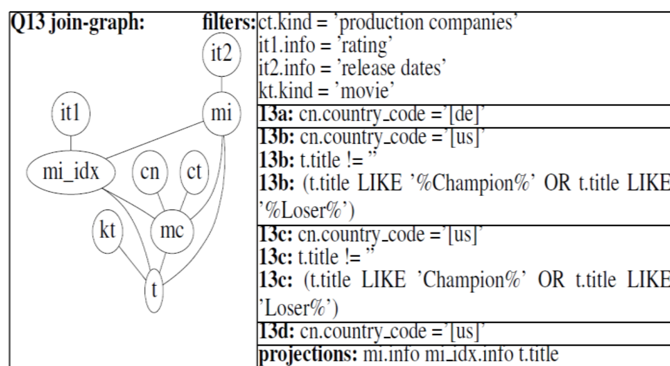
SELECT **MIN**(cn.name), **MIN**(mi.info), **MIN**(mi_idx.info) **FROM** company_name cn, company_type ct, info_type it, info_type it2, title t, kind_type kt, movie_companies mc,
movie_info mi, movie_info_idx mi_idx **WHERE** cn.country_code = ’[us]’
AND ct.kind = ’production companies’ **AND** it.info = ’rating’ **AND** it2.info = ’release dates’
AND kt.kind = ’movie’ **AND** .. --(11 join predicates/see Fig. 4)

In Fig. 2, the join graph for query 13d is displayed. The graph's solid edges are key/foreign key edges (1: n), with the arrowhead pointing to the side of the primary key. Foreign key/foreign key joins (n: m) are represented by dotted edges and arise as a result of transitive join predicates.



Join graph for job query 13a - d

It offers details on the recently formed relationship between a production firm and a movie studio. A nested loop join of orders, customers, and nations is one option from here. The information about the action and state must be represented as a fixed length vector that the trained neural network model expects as input in order to calculate the Q-value of doing this action in the current state. For our model, the action and query in this example would be represented as:

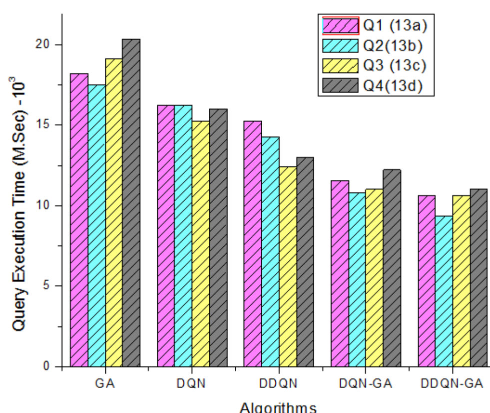


In the join action, the tables from the left and right input relations are 1-hot encoded. The first four left relation and the next four right relations are equivalent. The first two rows are 1 whereas the third and fourth are 0, as orders and customers are in the left relation. The physical join operator, which can be a mergejoin, hashjoin, or nested loop join, is then 1-hot encoded. The PostgreSQL optimizer then computes estimates for the left and right input relations, which are then encoded. In the final query result, encode the relations that belong there. Whether there is an equijoin predicate between each of the three sets of tables is the final encoding. According to the three equijoin predicates in the query, three of these rows are not zero. The findings help to clarify the advantages of DQN, DDQN, GA, and hybrid techniques for optimizing join queries and offer suggestions for choosing the best methodology for Join data.

Query Execution Time: In join query optimization, the term "query execution time" refers to the time needed to process and run a join query. It is an essential performance indicator that is used to assess the efficiency and potency of join query optimization strategies.

Query Execution Time = Planning Time + Optimization Time + Data Retrieval Time + Result Generation Time

The examination of join query optimization methods employing DQN, DDQN, Genetic Algorithms (GA), and their hybrid versions (DQN-GA and DDQN-GA) yielded important new information about the effects of these methods on query execution times. The outcomes supported the efficacy of DQN, DDQN, GA, and their hybrid forms in reducing join query execution times. Figure 2 demonstrates that the PostgreSQL model's mean execution time (represented by the Y mark in graphs) is longer than the DQN model's. The benchmark queries' average execution times, using the suggested DQN-GA and DDQN-GA models, are 9.5 and 8.7 ms and 16.6 and 15.7 ms, respectively, for PostgreSQL. Additionally, in a small fraction of queries, the DQN-GA and DDQN-GA model performs extraordinarily better than PostgreSQL. For instance, using the plan provided by the DQN-GA and DDQN-GA models, Query "13b" in the benchmark queries takes 10.78 and 9.35 ms as opposed to the plan generated by PostgreSQL, which takes 14.4 ms. The DQN-GA and DDQN-GA model provides query plans that are, on average, 35% less costly for the 113 queries in the benchmark queries than the PostgreSQL optimizer.

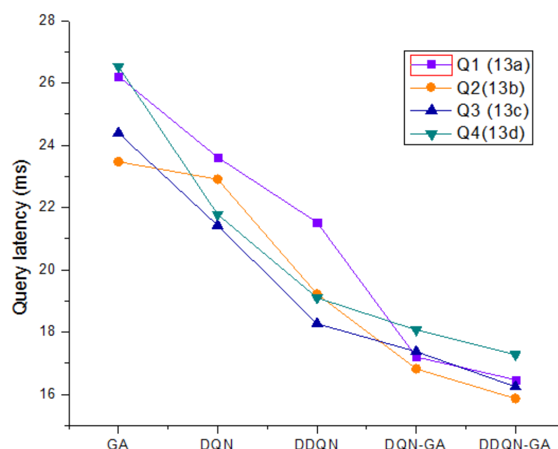


Query Latency: In join query optimization, the term "query latency" refers to the whole amount of time needed to process and run a join query, including the time spent on designing, optimizing, and running the query itself. It is a crucial performance indicator used to assess how effectively join query optimization strategies perform.

Query Latency = Planning Time + Optimization Time + Execution Time

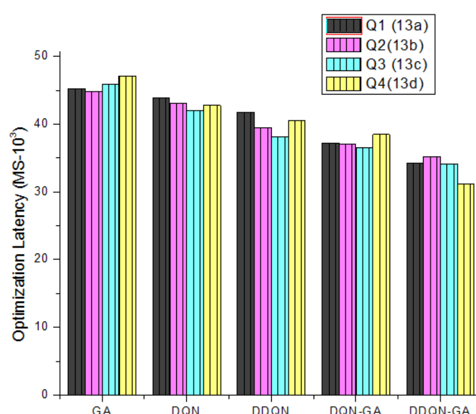
Planning Time is required to analyze the query and create a query execution plan is called time. It entails

actions like interpreting the query, figuring out which tables are involved, figuring out the join requirements, and choosing an initial join order. The time needed for optimization is the amount of time needed to discover the most effective join order for the query execution plan. It entails investigating numerous join order options and calculating the cost of every possible join plan. In terms of query latency, the hybrid versions DQN-GA and DDQN-GA performed even better than the separate approaches. Each test query in Figure 3 is run 20 times with a cold cache. The lowest, maximum, and median latency improvements are displayed on the graph. Every time, the join ordering plans generated by DQN-GA and DDQN-GA perform better than or are equal to those generated by PostgreSQL. DQN-GA and DDQN-GA are therefore able to develop plans with shorter execution times (and not merely shorter costs as compared to the cost model).



Optimization Latency:
The time it takes to carry out the optimization process and produce an optimized join plan is referred to as optimization latency in join query optimization. It indicates the time needed to consider various join order options, determine expenses, and choose the most effective join plan.
Optimization Latency = Planning Time + Optimization Time

By effectively examining the join space and choosing an optimized join plan that lowers the query's total execution cost, the objective is to minimize the optimization delay. Figure 5 displays the optimization of grouped by number of relations by comparing the optimization latencies of different techniques. Points above the black curve indicate questions for which DQN-GA and DDQN-GA had faster optimization times than DQN, DDQN, GA. Each point represents one of the queries in the JOB.



Scalability: The capacity of an optimization approach to manage growing query complexity and data quantities while retaining effective performance is referred to as scalability in join query optimization. It measures how effectively the optimization strategy can scale and adapt as the volume and sophistication of the data and the complexity of the queries increase. Scalability in join query optimization may be calculated using the following formula:
Scalability = (Size of Data / Execution Time) * (1 / Query Complexity)
Where, Measured commonly in terms of the number of tuples or the size of the database, size of data refers to the amount of data that is involved in the join query. Execution Time: The amount of time required by the

optimization strategy to produce a join plan that is optimized and to run the query. Query Complexity: The degree to which a query is difficult, as measured by elements like the quantity of tables, the nature of join conditions, the number of predicates, and the amount of intermediate results. The scalability score shows how well the DQN-GA and DDQN-GA Algorithms handle bigger data sets and more difficult queries, 78% and 81%, respectively.As the execution time decreases in relation to the quantity of the data and the complexity of the query, a scalability value reveals that DDQN is better able to scale than GA, DQN, and GA at 75.4%, 73.3%, and 73.3% respectively. The capacity of join query optimization techniques such as DQN, DDQN, GA, Hybrid DQN-GA, and DDQN-GA to handle more complicated queries and bigger datasets while retaining efficient performance is examined through the lens of scalability.

Resource Utilization: The effective use of computing resources throughout the optimization process is referred to as resource utilization in join query optimization. It measures how well the optimization method makes use of the CPU, RAM, and disc I/O to produce optimized join plans. The formula for calculating resource utilization in join query optimization can be expressed as:
Resource Utilization = (Time Spent on Actual Optimization / Total Execution Time) * (1 / Resource Consumption)
The efficiency of the join query optimization algorithms DQN (56.76%), DDQN(62.11%), GA(50.31%), Hybrid DQN-GA(71.70%), and DDQN-GA(73.21%) in using computing resources throughout the optimization process is shown by the examination of resource utilization. Resource utilization assesses how well these methods make use of the CPU, RAM, and disc I/O that are at their disposal to provide optimized join plans.

VIII. CONCLUSION

Using Deep Q-Networks (DQN), Double Deep Q-Networks (DDQN), Genetic Algorithms (GA), and hybrid DQN-GA and DDQN-GA techniques, we conducted a thorough examination of join query optimization in this research article. The goal was to assess how well these strategies worked to decrease query execution time and increase resource use. Our experimental analyses, carried out on several benchmark datasets, have produced important results. First of all, compared to conventional methods, DQN-based optimization strategies performed better. The DQN and DDQN algorithms successfully trained to make the best judgments when choosing join ordering and join procedures by framing join query optimization as a Markov Decision Process (MDP). The speed at which queries were executed significantly increased as a result. Additionally, the use of Genetic Algorithms as an alternate method of examining join query plans' search spaces yielded encouraging results. The use of evolutionary ideas in the GA-based optimization showed its capacity to provide effective join designs. The performance was further enhanced when paired with DQN and DDQN using the hybrid DQN-GA and DDQN-GA techniques, demonstrating the benefits of fusing reinforcement learning and evolutionary algorithms. We were able to get important insights from the performance analyses carried out on the benchmark datasets, which mirrored actual join query circumstances. We looked at how the performance of the optimization strategies was impacted by variables including database size, query complexity, and cardinality estimation accuracy. The results present options for further investigation, such as expanding the hybrid methods with further optimization techniques or adding other machine learning techniques. We can increase the efficacy and efficiency of database management systems in managing complicated join operations by continually enhancing join query optimization.

IX.REFERENCES

1. Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "How good are query optimizers, really?" Proceedings of the VLDB Endowment, 9(3):204-215, 2015.
2. Wei Wang et al. Database Meets Deep Learning: Challenges and Opportunities. SIGMOD Record, 2016.
3. Kostas Tzoumas et al. A reinforcement learning approach for adaptive query processing. In A DB Technical Report, 2008.
4. Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. CoRR abs/1808.03196 (2018). arXiv:1808.03196
5. Ryan Marcus et al. Deep reinforcement learning for join order enumeration. CoRR, 2018.
6. Thomas Fösel, Petru Tighineanu, Talitha Weiss, and Florian Marquardt. Reinforcement learning with neural networks for quantum feedback. Physical Review X, 8(3):031084, 2018.
7. Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. In Advances in Neural Information Processing Systems, pp. 2775–2785, 2017
8. Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q learning. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 30, 2016.
9. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. aiDM'18 (2018)
10. Sibylle D Muller, Nicol N Schraudolph, and Petros D Koumoutsakos. 2002. Step size adaptation in evolution strategies using reinforcement learning. In Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600), Vol. 1. IEEE, 151–156.

11. J. Ortiz, M. Balazinska, et al. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In DEEM, 2018.
12. 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.
13. 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.
14. 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.
15. Ohnishi, S., Uchibe, E., Yamaguchi, Y., Nakanishi, K., Yasui, Y., Ishii, S.: Constrained deep q-learning gradually approaching ordinary q-learning. Frontiers in Neurorobotics 13, 103 (2019)
16. Mirjalili, S. Genetic Algorithm. In Evolutionary Algorithms and Neural Networks; Springer: Cham, Germany, 2019; pp. 43–55.
17. V. Singh, and V. Mishra. Distributed Query Plan generation using Aggregation based Multi-Objective Genetic Algorithm, *Proceedings of the 2014 International Conference on Information and Communication Technology for Competitive Strategies*, *ACM*, New York, USA, 2014. Article 26, 8 pages.
18. James E Pettinger and Richard M Everson. 2002. Controlling genetic algorithms with reinforcement learning. In Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation. 692–692.
19. S. V. Chande, and M. Sinha. Genetic optimization for the join ordering problem of database queries, *Annual IEEE India Conference*, Hyderabad, 2011, pp. 1-5.
20. The PostgreSQL Global Development Group. PostgreSQL: TheWorld’s Most Advanced Open Source Database[EB/OL]. Available online: http://www.postgresql.org/
21. https://github.com/gregrahn/join-order-benchmark