2025 7th International Conference on Robotics, Intelligent Control and Artificial Intelligence (RICAI)

Research on Cypher Expansion Order Optimization Based on Deep Reinforcement Learning

Siqi Liu[1,2], Chuan Hu[1,2], Zhihong Shen[1,*]
Computer Network Infomation Center, Chinese Academy of Sciences[1], Beijing, China
University of Chinese Academy of Sciences[2], Beijing, China
{sqliu, huchuan, bluejoe}@cnic.cn

2025 7th International Conference on Robotics, Intelligent Control and Artificial Intelligence (RICAI) | 979-8-3315-6934-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/RICAI68060.2025.11384884

***Abstract*—With the widespread adoption of graph databases, Cypher has emerged as a predominant query language. Query execution time is highly sensitive to the path expansion order, where traditional rule-based and cost-based optimizers often fail to find optimal plans for complex patterns. Although machine learning has shown promise for SQL join order optimization, these methods cannot be directly applicable to Cypher. To address this gap, we introduce a novel approach that leverages Deep Reinforcement Learning (DRL) to optimize the Cypher expansion order. We formulate this task as a sequential decision problem and train a Deep Q-Network (DQN) agent to select an efficient expansion sequence. A key aspect of our work is the modeling of Cypher query features into vector representations, which are then used to train the model. We evaluated our method on a real-world graph dataset, and the experimental results demonstrate that our approach significantly reduces query execution times, validating the effectiveness and efficiency of the proposed solution.**

***Keywords*—Graph Database ; Reinforcement Learning ; Query Optimization**

I. Introduction

In recent years, the complex relationships within data have become increasingly crucial across various domains, including social networks, financial risk control, knowledge graphs, and bioinformatics. Consequently, graph databases have become pivotal for their natural ability to model, store, and query these highly interconnected datasets. As a declarative query language for property graphs, Cypher has emerged as the de facto standard, enabling users to succinctly express complex traversal queries [1]. However, this expressive power introduces a significant performance challenge: the execution time of a Cypher query is heavily dependent on the path expansion order chosen in its execution plan. An optimized expansion order can lead to performance improvements of several orders of magnitude compared to a poor one. For a query involving N expansions, the number of possible execution orders grows factorially, creating a vast combinatorial search space that makes finding the optimal plan an NP-hard problem [2].Figure 1 shows how a single Cypher query can be represented as a query graph, which in turn can be translated into numerous possible expansion orders. Traditional query optimizers, which rely on hand-crafted rules and cost models, often struggle with the inherent complexity and variability of graph queries. Their performance is highly dependent on accurate cardinality estimation [3]—a task that is notoriously difficult in graph databases due to complex topological structures, deep correlations, and the prevalence of cyclic patterns. Although machine learning techniques, particularly reinforcement learning (RL), have achieved notable success in optimizing join orders for SQL queries [4], these methods are not directly transferable to the graph database domain. The fundamental differences in data models and operational paradigms necessitate the development of specialized methods for Cypher query optimization.

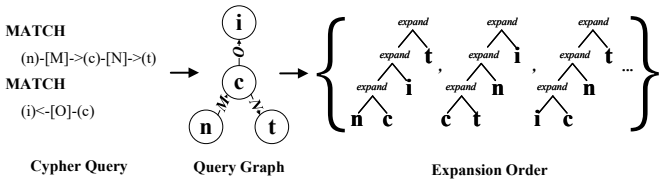

Figure 1. Example of Cypher query Transformation.

To address this gap, we introduce a Deep Reinforcement Learning (DRL) approach for optimizing the path expansion order in Cypher queries. Our method formulates query optimization as a sequential decision task where an agent interacts directly with the database environment to learn optimal expansion strategies. The agent selects expansion actions at each step based on the current state, progressively constructing complete execution plans. Our approach centers on a novel state representation scheme tailored for graph queries, which combines a Graph Attention Network (GAT) [5] to capture the global topology of the entire query and a Tree-Long Short-Term Memory (Tree-LSTM) [6] network to dynamically encode the hierarchical structure of partial execution plans.

Our main contributions are as follows:

- We design a unified feature representation scheme for the fundamental elements in Cypher queries—nodes and relationships—enabling a deep semantic awareness of the graph structural elements.
- By combining a GAT-based graph structure embedding with a Tree-LSTM-based plan tree embedding, we construct a state representation method that captures both node and edge features, providing a comprehensive basis for the DRL agent to perceive query semantics and execution context.
- We build an end-to-end framework for query optimization

This work was supported by the Informatization Plan of Chinese Academy of Sciences (Grant No. CAS-WX2022GC-02).

and rewriting, encompassing the entire process from original query parsing and DRL-based decision-making to the final query rewriting, thereby automating the pipeline from query understanding to execution plan generation.

II. Related Work

Query optimization, a core engine of database systems, has consistently been a research hotspot in both academia and industry. Traditional query optimization methods, typified by the dynamic programming algorithm pioneered in System R [7] and related heuristic approaches, primarily rely on cost estimation to search for an optimal execution plan within a vast plan space. While these methods have achieved great success in relational databases, their performance is highly dependent on the accuracy of cardinality estimation, often leading traditional optimizers to select sub-optimal execution plans. To address these challenges, researchers have begun to explore the application of machine learning to build a new generation of learning-based query optimizers.

In relational databases, modeling the Join Order Selection problem as a sequential decision-making task and employing Reinforcement Learning (RL) for end-to-end optimization has become a fruitful research direction. Marcus and Papaemmanouil introduced ReJOIN [8], using a simple encoding scheme for join trees and employing proximal policy optimization (PPO) to select joins iteratively. Later work built on this foundation with significant enhancements. For instance, JOGGER [9] adopts a graph-based model, streamlining parameter counts in deep neural networks through graph convolutional networks and custom tree-based attention modules, which enables JOGGER to achieve state-of-the-art performance with reduced optimization time and computational resources.

Beyond individual learning components designed for join order selection, several more comprehensive learning-based optimizers have been developed. These systems incorporate functionalities such as join operator selection, index selection, plan generation, and plan search into their architectural design. Bao [10] employs a Tree-CNN to estimate query execution time and utilizes Contextual Multi-Armed Bandits (CMABs) to model hint sets, rather than reconstructing the entire optimizer using a learning model. LOGER [11] aims to simultaneously generate high-quality join orders and physical operators. It leverages a Graph Transformer to encode relationships between tables and predicates and adopts beam search to explore the plan space. COOOL [12] follows a similar principle as Bao but incorporates a learning-to-rank mechanism to compare and select query plans.Although these methods show considerable advances, their design remains bound to the relational model. Their feature engineering typically depends on a fixed vector space comprising all tables and columns in the database, making them difficult to apply directly to graph databases with complex and varied topological relationships.

Compared to SQL, graph query optimization presents unique challenges and has developed its own research trajectory. Early optimization techniques were primarily drawn from distributed systems and graph theory, including methods such as data partitioning, query decomposition, and incremental processing [13]. These approaches provided a foundational basis for handling large-scale graph data but primarily focus on physical-level optimizations or specific query types. Recently, inspired by the success of learning-based optimizers in relational databases, such methods have begun to gain traction in graph query optimization. For example, Eschauziere et al. [14] proposed a learning-based optimizer that iteratively constructs optimized join plans for SPARQL queries on RDF graphs. The subsequent ReJOOSp [15] framework extends this approach by accommodating various join types between triple patterns and supporting complex SPARQL queries incorporating operations beyond basic joins. Wang et al. developed April [16], an automated graph data management system based on reinforcement learning, which encompasses optimization across storage structure selection, index selection, and query processing. Collectively, these studies have advanced the field of database query optimization and offer valuable insights for future research endeavors.

III. Methods

In this section, we describe the overall framework of the proposed approach. We first provide an overview of the framework, followed by detailed specifications of each component.

A. Framework

As shown in Figure 2, our proposed optimization framework follows the classic agent-environment interaction model of reinforcement learning.The system receives a raw Cypher query and parses it into graph-structured data. At the beginning of each episode, the environment is reset to an initial state. During each decision step, the agent selects an action from a set of legal expansion operations based on its evaluation of the current state. The environment executes the action, updates its internal state (i.e., the partial execution plan), computes a new state representation, and feeds it back to the agent. Ultimately, when all edges are processed, the system outputs a complete expansion sequence and calculates the final reward. The transition tuples (state, action, reward, next state) generated during the interaction are stored in an experience replay buffer. The agent progressively optimizes its decision-making policy by continuously updating the parameters of its internal neural network through batch sampling from this buffer.

B. Element Representation

In graph databases, the basic constituents of queries are nodes and relationships. To effectively capture the schema information and semantic details of the query, we design specialized feature representation methods. We create multiple learnable embedding tables for atomic components of the graph: node labels $E_L \in \mathbb{R}^{(N_L+1)\times d_L}$,relationship types $E_R \in \mathbb{R}^{(N_R+1)\times d_R}$, node properties $E_P \in \mathbb{R}^{N_P \times d_P}$, filter operators $E_O \in \mathbb{R}^{N_O \times d_O}$, and expansion directions $E_D \in \mathbb{R}^{2\times d_D}$. Here, N and d represent the number of elements and the embedding dimension for each category, respectively.

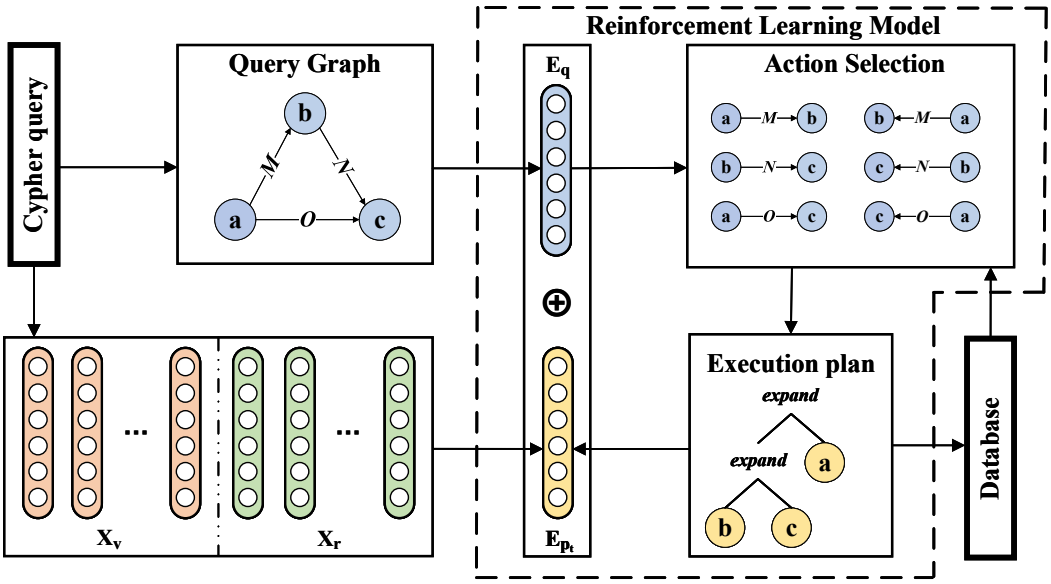

Figure 2. Framework Overview.

- **Node Feature Representation:** For a node v in the query graph, its feature vector $\mathbf{X}_v$ is determined by its label information and filter conditions. Let the label index of node v be $idx(l_v)$, and the set of attached filter conditions be $\mathcal{F}_v = \{f_1, f_2, \ldots, f_k\}$, where each filter f_i consists of a property index $idx(p_i)$ and an operator index $idx(o_i)$. We first compute the aggregated predicate representation $e_{pred}(v)$ by averaging the sum of property and operator embeddings for all filters:

$$\mathbf{e}_{pred}(v) = \frac{1}{|\mathcal{F}_v|} \sum_{f_i \in \mathcal{F}_v} \left(E_P[idx(p_i)] + E_O[idx(o_i)]\right) \tag{1}$$

  The final node feature vector is obtained by projecting the concatenation of its label embedding and the aggregated predicate embedding through a two-layer multi-layer perceptron (MLP):

$$\mathbf{X}_v = \text{MLP}_{\text{node}}\left([E_L\left[\text{idx}(l_v)\right] \oplus \mathbf{e}_{pred}(v)]\right) \tag{2}$$

- **Relationship Feature Representation:** For a relationship r, its feature vector $\mathbf{X}_r$ is determined by its type index $idx(t_r)$ and the selected direction d at the current expansion step. Its feature vector is projected by an MLP after concatenating the corresponding embeddings:

$$\mathbf{X}_r = \text{MLP}_{\text{edge}}\left([E_R\left[idx(t_r)\right] \oplus E_D[d]]\right) \tag{3}$$

C. State Representation

The agent's state S_t comprises a global query embedding e_q and a local plan embedding e_{p_t}, generated by distinct neural network modules to capture static global information and dynamic local information required for decision-making.

- **Global Query Embedding(e_q):** To capture the structural information of the entire query, we parse the Cypher query into a query graph $G_q = (V_q, E_q)$. Each node $v \in V_q$ is initialized with its feature $\mathbf{X}_v$ generated via the method in Section B, and each edge $e \in E_q$ is initialized with feature $\mathbf{X}_r$. We employ a graph attention network (GAT) to encode G_q. Through its message-passing and self-attention mechanisms, the GAT can learn the importance of different nodes and their neighborhoods. The update rule for the representation of node i at layer l is as follows:

$$\mathbf{h}_i^{(l)} = \sigma\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{ij}^{(l)} W^{(l)} \mathbf{h}_j^{(l-1)}\right) \tag{4}$$

  where α_{ij} is the attention-based weight coefficient. After two layers of GAT propagation, we apply global average pooling to the final node representations to obtain a fixed-dimensional query graph embedding vector e_q. This vector remains constant throughout the decision-making process for a single query, serving as static context.
- **Local Plan Embedding(e_{p_t}):** To dynamically encode the progressively generated execution plan, we introduce a Tree-Long Short-Term Memory (Tree-LSTM) network. We recursively construct a left-deep, ternary tree from a sequence of t expansion operations. Each internal node represents an expansion operation, with three children corresponding to: (1) the partial plan subtree from the previous step, (2) the relationship used in the current expansion, and (3) the newly added node in the current expansion.We use an N-ary Tree-LSTM cell to encode the tree bottom-up. For any node j, its hidden state h_j and cell state c_j are computed as:

$$\mathbf{i}_j = \sigma(W_i \mathbf{x}_j + \sum_{k \in C(j)} U_{ik} \mathbf{h}_k) \tag{5}$$

$$\mathbf{f}_{jk} = \sigma(W_f \mathbf{x}_j + U_{jk} \mathbf{h}_k), \quad \forall k \in C(j) \tag{6}$$

$$\mathbf{o}_j = \sigma(W_o \mathbf{x}_j + \sum_{k \in C(j)} U_{ok} \mathbf{h}_k) \tag{7}$$

$$\mathbf{u}_j = \tanh(W_u \mathbf{x}_j + \sum_{k \in C(j)} U_{uk} \mathbf{h}_k) \tag{8}$$

$$\mathbf{c}_j = \mathbf{i}_j \odot \mathbf{u}_j + \sum_{k \in C(j)} \mathbf{f}_{jk} \odot \mathbf{c}_k \tag{9}$$

$$\mathbf{h}_j = \mathbf{o}_j \odot \tanh(\mathbf{c}_j) \tag{10}$$

where $\mathbf{x}_j$ is the input feature of node j (generated as per Section B, depending on whether it is a node, relationship, or internal expansion node), and $C(j)$ is the set of its children. The hidden state of the root node $\mathbf{h}_{\text{root}}$ aggregates the structure and order information of the entire partial execution plan, forming the dynamic plan representation vector e_{p_t}.

D. Reinforcement Learning Model

We formulate the expansion order selection task as a Markov Decision Process (MDP), with the core elements $M = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$ specified in this context as:

- **State Space** $\mathcal{S}$: As described in Section C, the state is $S_t = e_q \oplus e_{p_t}$.
- **Action Space** $\mathcal{A}$: An action $a \in \mathcal{A}$ is an expansion operation (r, d), where r is a relationship in the query and $d \in \{0, 1\}$ indicates the direction. The set of legal actions $\mathcal{A}_t$ at state S_t is constrained by graph traversal logic, allowing only expansions from a node already in the partial plan to a new node via an unused relationship.
- **Reward Function** R: A sparse reward is used, where $R_{final} = \log(T_{\text{original}}/T_{\text{rewritten}})$ is given only at the terminal state, and intermediate rewards are 0.

We employ a Deep Q-Network (DQN) [17]to approximate the optimal action-value function $Q^*(s, a)$. Both the policy network Q_θ and target network $Q_{\theta'}$ utilize a four-layer fully connected network with Layer Normalization at the input to stabilize training and prevent issues arising from varying input vector scales. The network takes the concatenated state vector S_t as input and outputs Q-value predictions for all possible actions. During action selection, we use an Invalid Action Masking mechanism, setting the Q-values of illegal actions to negative infinity, ensuring the agent only chooses from $\mathcal{A}_t$. The algorithm is based on the Double DQN [18] framework to mitigate the overestimation of Q-values common in standard DQN. The target Q-value is computed as follows:

$$a^* = \arg \max_{a' \in \mathcal{A}_{t+1}} Q_\theta(S_{t+1}, a') \tag{11}$$

$$Y_t = R_{t+1} + \gamma Q_{\theta'}(S_{t+1}, a^*) \tag{12}$$

The network parameters θ are updated by minimizing the Huber loss (Smooth L1 Loss) between the predicted Q-value $Q_\theta(S_t, A_t)$ and the target Q-value Y_t.

E. Query Rewriting

To apply the optimal action sequence learned during reinforcement learning to actual database queries, we design and implement an automated query rewriting module. This module systematically converts the agent's output expansion sequence into a Cypher query that enforces the execution order in Neo4j. The expansion sequence is decomposed into a series of single-step expansion operations, leveraging nested CALL subqueries to receive variables matched in the previous step, execute a new expansion, and pass newly matched variables to the next block. This approach decomposes a complex MATCH pattern into a strictly ordered chain of subqueries, enforcing the optimized plan at the database level. Algorithm 1 outlines the query rewriting process.

Algorithm 1: Query Rewriting Algorithm

Input : Original parsed query object P, Expansion sequence $S_E = [(r_1, d_1), \ldots, (r_N, d_N)]$
Output: Rewritten Cypher query string

Initialize cypher_blocks = [], visited_nodes = {}
Determine the starting node v_{start} from $S_E[0]$
block = GEN_MATCH(v_{start})
cypher_blocks.append(block)
visited_nodes.add(v_{start})
for (r_i, d_i) *in* S_E **do**
    Determine the known node v_{known} and the new node v_{new} for the current expansion
    block = GEN_EXPAND($v_{\text{known}}, v_{\text{new}}, r_i, d_i$))
    cypher_blocks.append(block)
    visited_nodes.add(v_{new})
end
return JOIN(cypher_blocks) + GEN_RETURN(P)

IV. Experiment Analysis

This section presents a comprehensive empirical evaluation of the proposed DRL-based Cypher query optimization framework. The objective is to quantify and validate the performance advantage of our method over native optimizers in mainstream graph databases on a standard graph query benchmark.

A. Experimental Setup

*a) **Dataset**:* Our evaluation utilizes the Graph Join Order Benchmark (G-JOB) dataset, a specialized benchmark for graph databases derived from real-world IMDB data. It comprises 113 distinct Cypher queries generated from 33 templates, showcasing diverse graph patterns with complexities (number of relationships) ranging from 2 to 13. For our experiments, these original 113 G-JOB queries constitute our test set, used to rigorously assess our model's performance on standard workloads. To train our DRL model, we generated a separate and extensive training set by programmatically instantiating parameters from the G-JOB templates. This methodology ensures a varied learning environment and a clear separation of parameter values between the training and test queries.

*b) **Baseline**:* To objectively evaluate the performance of our method, we selected the native query optimizer of Neo4j database (version 4.2) as the primary baseline. We recorded and compared actual execution times between queries rewritten by our framework (Rewritten Query) and original queries executed directly on Neo4j.

*c) **Evaluation Metrics**:* We employed two standard metrics to evaluate query optimizer performance:

- **Average Query Latency (AQL):** Defined as the total time required to execute a single Cypher query, measured in milliseconds (ms), averaged over all queries. This is the most direct and crucial indicator of query execution efficiency.
- **Average Performance Gain Ratio (APGR):** Quantifies the average speedup achieved by our method relative to the baseline. For each query q_j in the test set, its individual performance gain ratio is calculated as $\text{PGR}_j = T_{\text{original},j}/T_{\text{rewritten},j}$. The overall Average Performance Gain Ratio is then computed as the arithmetic mean across all M queries in the test set:

$$\text{APGR} = \frac{1}{M}\sum_{j=1}^{M}\frac{T_{\text{original},j}}{T_{\text{rewritten},j}} \tag{13}$$

where $T_{\text{original},j}$ and $T_{\text{rewritten},j}$ represent the latency of executing the original and rewritten query q_j, respectively. An average ratio greater than 1 indicates overall performance improvement by the proposed method.

*d) **Implementation Details**:* Our DRL model is implemented using the PyTorch and trained on a server equipped with an NVIDIA Tesla T4 GPU (16GB). Training consisted of 5000 episodes. Key DQN hyperparameters included: Adam optimizer with learning rate 1×10^{-4}, the discount factor γ is 0.99, the experience replay buffer capacity of $10,000$, and the training batch size of 64. All queries were executed on Neo4j Community Edition 4.2 instances. Database cache was cleared programmatically before each query execution to ensure fair and reproducible results.

B. Experimental Results and Analysis

To illustrate the learning process and convergence of our DRL agent, Figure 3 presents the training loss curve over 5000 episodes. We observe a consistent decrease in loss, indicating that the agent effectively learns to approximate the optimal Q-values and stabilize its policy. The curve's gradual flattening suggests that the model has largely converged, achieving stable learning performance towards the end of the training process. This convergence demonstrates the robustness of our DRL architecture and training methodology.

We evaluated our trained DRL model (Our Method) against the Neo4j native optimizer (Neo4j Native) on the G-JOB test set. The experimental results clearly demonstrate that our method achieves significant performance improvements across the majority of test queries.

As shown in Table 1, our method reduces the average query latency from 13675.88 ms to 9627.45 ms, yielding an APGR of 3.51×.This result empirically demonstrates that the proposed DRL framework can effectively learn and generate superior query execution plans than the database's built-in optimizer, thereby substantially enhancing Cypher query execution efficiency.

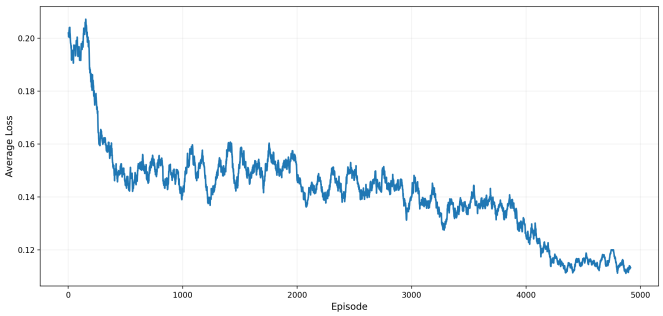

Figure 3. Model Training Loss Curve.

Table 1. Average Performance Comparison on G-JOB Test Set

Method	AQL (ms)	APGR
Neo4j Native	13675.88	1.00x
Our Method	9627.45	3.51x

Figure 4 illustrate the performance advantage of our method becomes more pronounced with increasing query complexity. For queries involving a higher number of relations, the reduction in AQL is particularly significant, aligning with the overall trend of improved optimization efficacy for complex patterns.

To further illustrate the per-query performance improvements, Figure 5 presents the performance gain ratio achieved for each query template in the G-JOB test set. We observe that our method consistently outperforms the native optimizer across most templates, with varying degrees of acceleration.Specifically, the performance gain ratio exceeds 1.0x (the horizontal line at APGR= 1) for the vast majority of query templates, indicating that our DRL-based approach effectively shortens query execution times. For several templates, the improvements are notably significant, underscoring the DRL agent's strong capability in learning optimal expansion orders for complex graph queries.

While a few templates exhibits limited optimization gains, highlighting specific improvement opportunities rather than fundamental constraints. These cases typically occur when the native Neo4j optimizer generates already-efficient execution plans, or where the specific patterns within these templates may benefit from more extensive or targeted exploration during the DRL training phase. Future work will address these edge cases through adaptive training strategies and enriched structural representations to enhance model generalization.

V. Conclusion

In this paper, we introduced a novel DRL framework for optimizing the Cypher expansion order in graph databases. We formulate this complex query optimization task as a sequential decision problem, where a DQN agent learns to select

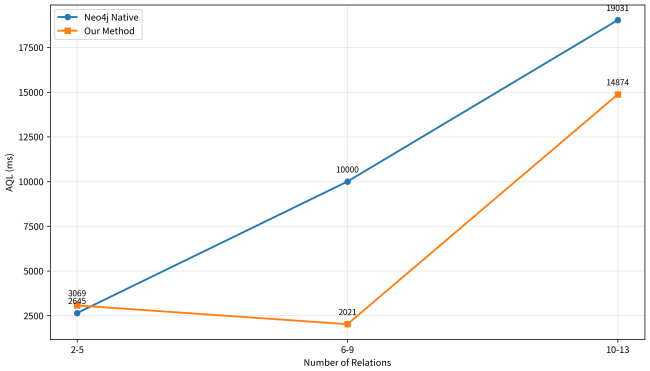

Figure 4. Comparing AQL by Number of Relations.

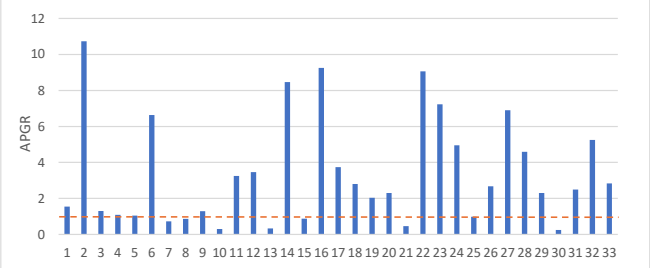

Figure 5. Comparing APGR by G-JOB Templates.

optimal expansion sequences through environment interaction. We effectively combine a GAT for capturing global query topology and a Tree-LSTM network for encoding the dynamic progression of the execution plan. We conducted extensive experiments on the G-JOB dataset, comparing our DRL-based approach with the native Neo4j optimizer. The results demonstrate that our method significantly reduces query execution times across a wide range of queries. This validates the effectiveness and efficiency of our proposed DRL framework in generating superior query execution plans, particularly for complex graph patterns where traditional optimizers often underperform.

REFERENCES

[1] M. Besta, R. Gerstenberger, E. Peter, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler, "Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries," ACM Comput. Surv., vol. 56, no. 2, art. 31, 2023.

[2] A. Mikhaylov, N. S. Mazyavkina, M. Salnikov, I. Trofimov, F. Qiang, and E. Burnaev, "Learned Query Optimizers: Evaluation and Improvement," IEEE Access, vol. 10, pp. 75205–75218, 2022.

[3] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui, "Cardinality estimation in DBMS: a comprehensive benchmark evaluation," Proc. VLDB Endow., vol. 15, no. 4, pp. 752–765, 2021.

[4] Z. Yan, V. Uotila, and J. Lu, "Join Order Selection with Deep Reinforcement Learning: Fundamentals, Techniques, and Challenges," Proc. VLDB Endow., vol. 16, no. 12, pp. 3882–3885, 2023.

[5] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," arXiv Preprint arXiv:1710.10903, 2017.

[6] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," arXiv Preprint arXiv:1503.00075, 2015.

[7] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: relational approach to database management," ACM Trans. Database Syst., vol. 1, no. 2, pp. 97–137, 1976.

[8] R. Marcus and O. Papaemmanouil, "Deep Reinforcement Learning for Join Order Enumeration," in Proc. First Int. Workshop on Exploiting Artificial Intelligence Techniques for Data Management, Houston, TX, USA, 2018, art. 3.

[9] J. Chen, G. Ye, Y. Zhao, S. Liu, L. Deng, X. Chen, R. Zhou, and K. Zheng, "Efficient Join Order Selection Learning with Graph-based Representation," in Proc. 28th ACM SIGKDD Conf. Knowledge Discovery and Data Mining, Washington DC, USA, 2022, pp. 97–107.

[10] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making Learned Query Optimization Practical," in Proc. 2021 Int. Conf. Management of Data, Virtual Event, China, 2021, pp. 1275–1288.

[11] T. Chen, J. Gao, H. Chen, and Y. Tu, "LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans," Proc. VLDB Endow., vol. 16, no. 7, pp. 1777–1789, 2023.

[12] X. Xu, Z. Zhao, T. Zhang, R. Kang, L. Sun, and J. Chen, "Coool: A learning-to-rank approach for sql hint recommendations," arXiv Preprint arXiv:2304.04407, 2023.

[13] A. B. Ammar, "Query optimization techniques in graph Databases," arXiv Preprint arXiv:1609.01893, 2016.

[14] R. Eschauzier, R. Taelman, M. Morren, and R. Verborgh, "Reinforcement learning-based SPARQL join ordering optimizer," in Proc. Eur. Semantic Web Conf., Cham, Switzerland, 2023, pp. 43–47.

[15] B. Warnke, K. Martens, T. Winker, S. Groppe, J. Groppe, P. Adhiyaman, S. Srinivasan, and S. Krishnakumar, "ReJOOSp: Reinforcement Learning for Join Order Optimization in SPARQL," Big Data and Cognitive Computing, vol. 8, no. 7, p. 71, 2024.

[16] H. Wang, Z. Qi, L. Zheng, Y. Feng, J. Ouyang, H. Zhang, X. Zhang, Z. Shen, and S. Liu, "April: An Automatic Graph Data Management System Based on Reinforcement Learning," in Proc. 29th ACM Int. Conf. Information & Knowledge Management, Virtual Event, Ireland, 2020, pp. 3465–3468.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," arXiv Preprint arXiv:1312.5602, 2013.

[18] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in Proc. AAAI Conf. Artificial Intelligence, vol. 30, no. 1, 2016.