Vol. 07, No. SI*, pp. 46 – 54 (2026)
ISSN: 2708-0757

JOURNAL OF APPLIED SCIENCE AND TECHNOLOGY TRENDS

www.jastt.org

Reinforcement Learning for Join Order Optimization in PostgreSQL: Query Rewriting and Evaluation on JOB and TPC-H Benchmarks

Mohammed R. Omar*, Nawzat S. Ahmed

[1]*Department of Information Technology Management, Duhok Polytechnic University, Duhok, Iraq,*
mohammed.omar@dpu.edu.krd, nawzat.ahmed@dpu.edu.krd

**Correspondence: mohammed.omar@dpu.edu.krd*

Abstract

Join order optimization is a critical combinatorial problem in query processing. This paper applies reinforcement learning (RL) techniques to the join order optimization task in PostgreSQL by implementing and evaluating three separate RL-based optimizers: Proximal Policy Optimization (PPO), Deep Q-Network (DQN), and Advantage Actor-Critic (A2C). Each method is trained on the Join Order Benchmark (JOB) and evaluated on both JOB and TPC-H workloads. Performance is compared against PostgreSQL's default planner without join reordering (PG-Old) and the built-in PostgreSQL optimizer with reordering enabled (PG-Join). Results show that RL-based methods significantly reduce execution times compared to PG-Old and often perform on par with or better than PG-Join, especially for complex multi-join queries. Among the tested methods, PPO achieves the most consistent improvements, with up to 4.47× average speedup over PG-Old on JOB and measurable gains on TPC-H. These findings demonstrate the potential of reinforcement learning as a practical and adaptive approach to join order optimization in relational databases.

Keywords: *Reinforcement Learning, Query Optimization, Join Order, PostgreSQL, TPC-H, Join Order Benchmark (JOB)*
Received: October 04th, 2025 /Revised: March 24th, 2026 / Accepted: April 16th, 2026 / Online: April 19th, 2026

I. INTRODUCTION

Efficient query processing is central to relational databases, yet join order optimization—choosing the best sequence to join tables—remains NP-hard [1]. Early optimizers like System R applied dynamic programming over left-deep trees [2], balancing plan quality with complexity. Modern systems (e.g., PostgreSQL) still rely on cost models and cardinality estimates [1], [3], which can be highly inaccurate [4]. Small misestimates often cascade into poor plans, and while self-learning ideas (e.g., IBM's LEO) exist, they are rarely deployed [5]. As argued by [6], more adaptive, data-driven approaches are needed.

Recent work explores machine learning, particularly reinforcement learning (RL), to optimize queries using execution feedback instead of static models [6], [7], [8]. This work focuses on join order optimization in PostgreSQL by framing join enumeration as a sequential decision-making task, where an agent observes a partial join plan and selects the next relation to join. Three RL algorithms—PPO, DQN, and A2C—are evaluated against PostgreSQL baselines on the JOB and TPC-H benchmarks.

The contributions of this study are:

a) an RL environment for join order optimization with a query rewriting module;

b) implementation and comparison of DQN, PPO, and A2C agents;

c) extensive experiments on JOB (90 train, 23 test) and TPC-H (test-only), measuring real execution times;

d) comparison with prior learned optimizers (e.g., DQ, ReJOIN, RTOS), showing PPO achieves strong latency reductions relative to PostgreSQL baselines;

e) analysis of why PPO outperforms DQN/A2C and its generalization to unseen TPC-H queries.

The remainder of this paper is structured as follows. Section 2 reviews related work in join order optimization, covering both classical techniques and recent learning-based methods such as ReJoin, DQ, SkinnerDB, and RTOS. Section 3 introduces the methodology, including query rewriting, PostgreSQL integration, state/action/reward design, and the implementation of PPO, DQN, and A2C agents. Section 4 outlines the training

doi: 10.38094/jastt7031087

procedure used to convert SQL queries into JOIN-style syntax and enforce agent-generated plans. Section 5 presents our experimental study, detailing the JOB and TPC-H benchmarks, setup, and runtime comparisons of RL agents against PostgreSQL's PG-Old and PG-Join. Section 6 discusses the findings, analyzing algorithm performance, generalization, and trade-offs relative to prior approaches. Finally, Section 7 concludes the paper, summarizing contributions and highlighting future directions such as improved state representations, adaptive training, and broader integration of RL into query optimization pipelines.

II. RELATED WORK

Decades of research in query optimization have produced many approaches for join order selection. Classical dynamic programming algorithms for join order (e.g., System R's algorithm) guarantee optimal plans within a restricted search space (usually left-deep trees), but have exponential complexity in the number of joins. To cope with complex queries, commercial optimizers often resort to heuristics or randomized algorithms (e.g., PostgreSQL's genetic optimizer GEQO) to reduce planning time, at the risk of suboptimal plans. The hardness of join order optimization was formally established by [9], who proved the problem NP-hard and even NP-complete under certain cost formulations [10].

The idea of learning a query optimizer has gained traction in recent years. Reinforcement learning has been applied to query optimization problems including join ordering. [11], was one of the first RL-based join optimizers. ReJoin trained a policy network to construct join trees given a query, using a policy gradient method (REINFORCE) with query cost estimates as the reward. While it eventually learned plans competitive with a traditional optimizer, it required a very large number of training episodes and careful reward shaping due to the sparse feedback (reward only obtained after a full plan is formed). [12] proposed DQ, a Deep Q-Learning approach for join order selection. DQ's agent was first trained using the optimizer's cost model to simulate rewards, and was later fine-tuned with actual execution runtime feedback to compensate for cost model inaccuracies [13]. They reported that the learned policy could achieve better performance than a cost-based optimizer with far fewer training queries than exhaustive search, highlighting the benefit of learning from experience.

Beyond pure learning approaches, there are hybrids like SkinnerDB [14]. SkinnerDB doesn't train a model across queries; instead, it uses reinforcement learning concepts within a single query's execution. It treats the problem as a multi-armed bandit: during query execution, it interleaves different partial join orders and progressively focuses on the faster paths, effectively re-optimizing the query on the fly. This avoids relying on any cost model or training phase, but requires a specialized execution engine and only addresses intra-query adaptivity (no cross-query learning). In a different vein, [15] proposed RTOS (Reinforcement Learning with Tree-LSTM for Join Order Selection), which uses graph neural network representations of join states to better generalize across schemas. They demonstrated improved performance on JOB and TPC-H by capturing the structural information of join trees with a tree-LSTM encoder, in contrast to earlier methods that used fixed-size feature vectors.

More broadly, learned query optimization has also been explored via supervised learning and other ML techniques. For instance, [16] work on OtterTune and [11] BaO focus on tuning database configuration and improving the cost model using ML, rather than directly selecting join orders. This work focuses specifically on using model-free RL for join order optimization, building on the above RL approaches. Compared to prior RL methods, the study provides an integrated system evaluation where the RL agent is implemented inside a real DBMS(PostgreSQL) and evaluate on full benchmark query sets (JOB and TPC-H), whereas prior works often used simulators or limited query subsets. Multiple RL algorithms (PPO, DQN, and A2C) are compared within the same framework to assess which is most effective for this task, providing practical insight beyond any single algorithm's performance.

III. METHODOLOGY

The proposed approach combines query rewriting, a custom reinforcement learning environment, and integration with PostgreSQL. All queries are first converted to explicit JOIN-style syntax. From these, 90 JOB queries are used to train three independent RL agents (PPO, DQN, A2C) within the JoinOrderEnv, which interacts with PostgreSQL to obtain cost and runtime feedback. After training, each agent is evaluated on 23 unseen JOB queries and 22 TPC-H queries, where the predicted join orders are executed in PostgreSQL to measure actual runtime. The performance of RL-generated plans is then compared against PostgreSQL's baselines (PG-Old and PG-Join).

Figure 1 illustrates the workflow of the proposed system. First, all queries are rewritten into explicit JOIN-style syntax. From these, 90 JOB queries are used for training each RL agent (PPO, DQN, A2C) within the JoinOrderEnv, interacting with PostgreSQL for cost and runtime feedback. The remaining 23 JOB queries and 22 TPC-H queries are used for testing. Finally, execution times of RL-generated join plans are compared against PostgreSQL baselines (PG-Old and PG-Join).

A. Query Rewriting and PostgreSQL Integration

Query rewriting is a key step in our system. PostgreSQL's optimizer can only reorder joins if queries use explicit JOIN syntax or certain planner settings. However, many JOB queries use legacy implicit joins (tables listed in FROM with conditions in WHERE). To address this limitation, a preprocessing module was implemented to rewrite implicit joins into explicit JOIN ... ON ... form, making join order visible and controllable by the RL environment. This ensures join order is visible and controllable by the RL environment. It also allows fair baseline comparisons:

- PG-Old = old-style implicit joins + no reordering.
- PG-Join = explicit JOIN-style + PostgreSQL optimizer.
- RL = explicit JOIN-style + reinforcement learning agent.

Integration with PostgreSQL is done via Python (psycopg2). During training, the environment builds join plans step by step

based on the agent's actions. At each step, a partial plan is sent with EXPLAIN to obtain PostgreSQL's cost estimate, which provides reward feedback. Only after a full join order is built do we run EXPLAIN ANALYZE to capture the true execution time for evaluation. Each query forms an episode, where the agent starts from an empty plan and continues until all joins are placed.

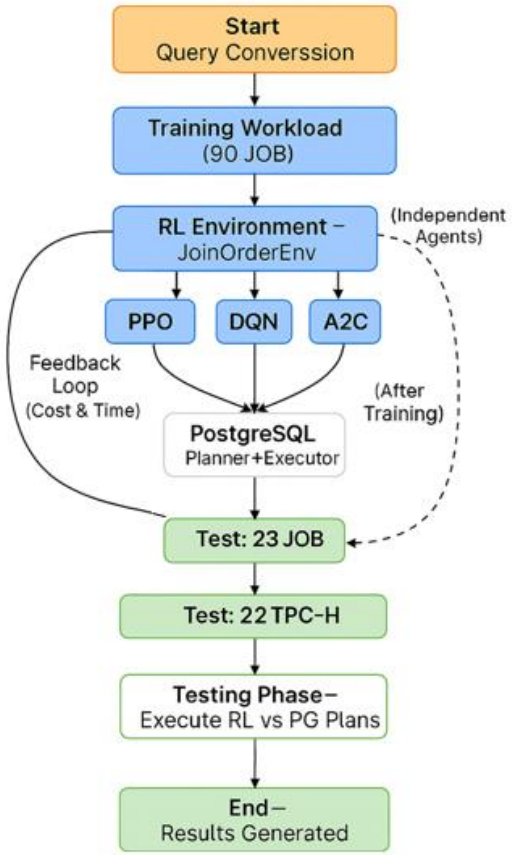

Fig. 1. Workflow of the RL-based join order optimization system.

B. State Representation and Action Space

Join order optimization is formulated as a reinforcement learning task by defining the state, action, and reward within query planning.

- State: A bitmask of length N (tables in the query) indicates which tables have been joined (1) and which remain (0). This is augmented with join predicates (an N×N adjacency matrix) and selection predicates to reflect data reduction opportunities. As joins are executed, the bitmask and predicate information are updated.
- Action: At each step, the agent selects the next table to join with the current partial plan. This left-deep restriction simplifies the action space to (N–k) choices when k tables have already been joined.
- Reward: After each join, PostgreSQL's estimated cost is retrieved via EXPLAIN and negated (–cost) so that lower costs yield higher rewards. At the final step, the negative actual execution time from EXPLAIN ANALYZE can be included. This reward shaping provides dense feedback and guides the agent toward minimizing query execution time.

C. Reinforcement Learning Algorithms

Three reinforcement learning algorithms were implemented and evaluated in the PostgreSQL-integrated environment: Deep Q-Network (DQN), Advantage Actor-Critic (A2C), and Proximal Policy Optimization (PPO). Each was trained and tested independently to allow a fair comparison of different RL paradigms—DQN as value-based, A2C as hybrid actor-critic, and PPO as policy-based—highlighting their strengths and weaknesses in join order optimization.

- DQN (Deep Q-Network): A value-based method that learns an action-value function for expected cumulative reward. In this work, DQN used a feed-forward network with two hidden layers (128 units each) and ReLU activations, trained with ε-greedy exploration, experience replay, and temporal-difference learning [17], [18]. The proposed approach extends prior work [12] by evaluating both cost estimates and actual runtimes.
- A2C (Advantage Actor-Critic): A synchronous actor-critic algorithm [19] where the actor outputs join action probabilities and the critic estimates state values [20]. Both share a base network, with the advantage function guiding policy updates. A2C was selected for its simplicity and stable training in policy-gradient tasks.
- PPO (Proximal Policy Optimization): An advanced actor-critic method that improves stability using a clipped surrogate objective [3], [21]. Implemented with Stable-Baselines3, PPO collects join-order episodes, computes advantages, and updates policies with multiple epochs per batch. It showed faster convergence and better generalization than DQN and A2C, consistent with prior systems like ReJoin [11].

All three algorithms were trained and tested individually in our PostgreSQL-integrated RL environment. Each model used flattened binary join bitmasks and statistical query features as input. The models were trained on 90 queries from the JOB dataset and evaluated on 23 remaining JOB queries and 22 TPC-H queries to test generalization.

IV. Training Procedure

key steps in the proposed approach were converting all SQL queries from old-style joins (comma-separated tables with conditions in the WHERE clause) to explicit JOIN ... ON ... syntax. These queries were rewritten makes the join graph explicit and allows PostgreSQL to follow the agent's specified join order using parentheses. For each test query, the agent's chosen join order is translated into a JOIN-style query and executed with EXPLAIN ANALYZE to measure runtime. The rewriting ensures consistent plan generation without altering PostgreSQL's native join algorithms, and was sufficient for our experiments without requiring source code modifications.

V. EXPERIMENTS

This section presents the experimental results of our RL-based join optimizers on the JOB and TPC-H benchmarks. The database and benchmark setup are first described, then discuss the performance on JOB, followed by generalization results on TPC-H.

A. Setup and Benchmarks

All experiments were conducted on a Windows 11 x64 personal computer running PostgreSQL 17 on an Intel Core i5-1135G7 processor (2.40 GHz, 11th Generation) with 8 GB of RAM. PostgreSQL was installed locally using default configuration parameters, with only minor adjustments to ensure fair and reproducible join-ordering comparisons. No GPU acceleration was employed; all reinforcement learning training and evaluation were performed on CPU using Python within the Visual Studio Code environment and standard scientific computing libraries.

Two widely used benchmark workloads were employed to evaluate performance. The Join Order Benchmark (JOB), derived from the IMDb database, was fully imported into PostgreSQL and occupies approximately 16 GB of disk space. The JOB schema (Figure 2) comprises 21 relational tables—including title, cast_info, movie_info, and company_name—connected through numerous primary–foreign key relationships, resulting in a dense and highly interconnected join graph. JOB queries typically involve between 3 and 16 joins, making the benchmark particularly challenging and well suited for evaluating join order optimization techniques under complex multi-join conditions.

The second workload, TPC-H, was generated at Scale Factor 1 using the standard dbgen tool and loaded into PostgreSQL, producing a database of approximately 4.1 GB. Its schema (Figure 3) consists of 8 core tables—such as customer, orders, lineitem, and nation—organized in a snowflake structure with well-defined relational dependencies. In contrast to JOB, TPC-H queries generally contain fewer joins (typically fewer than eight) but incorporate more analytical operations, including aggregations, complex selection predicates, grouping, and nested subqueries. This complementary workload provides a meaningful test of generalization, enabling evaluation of whether models trained on the complex IMDb-derived JOB queries can transfer effectively to a different schema and query distribution.

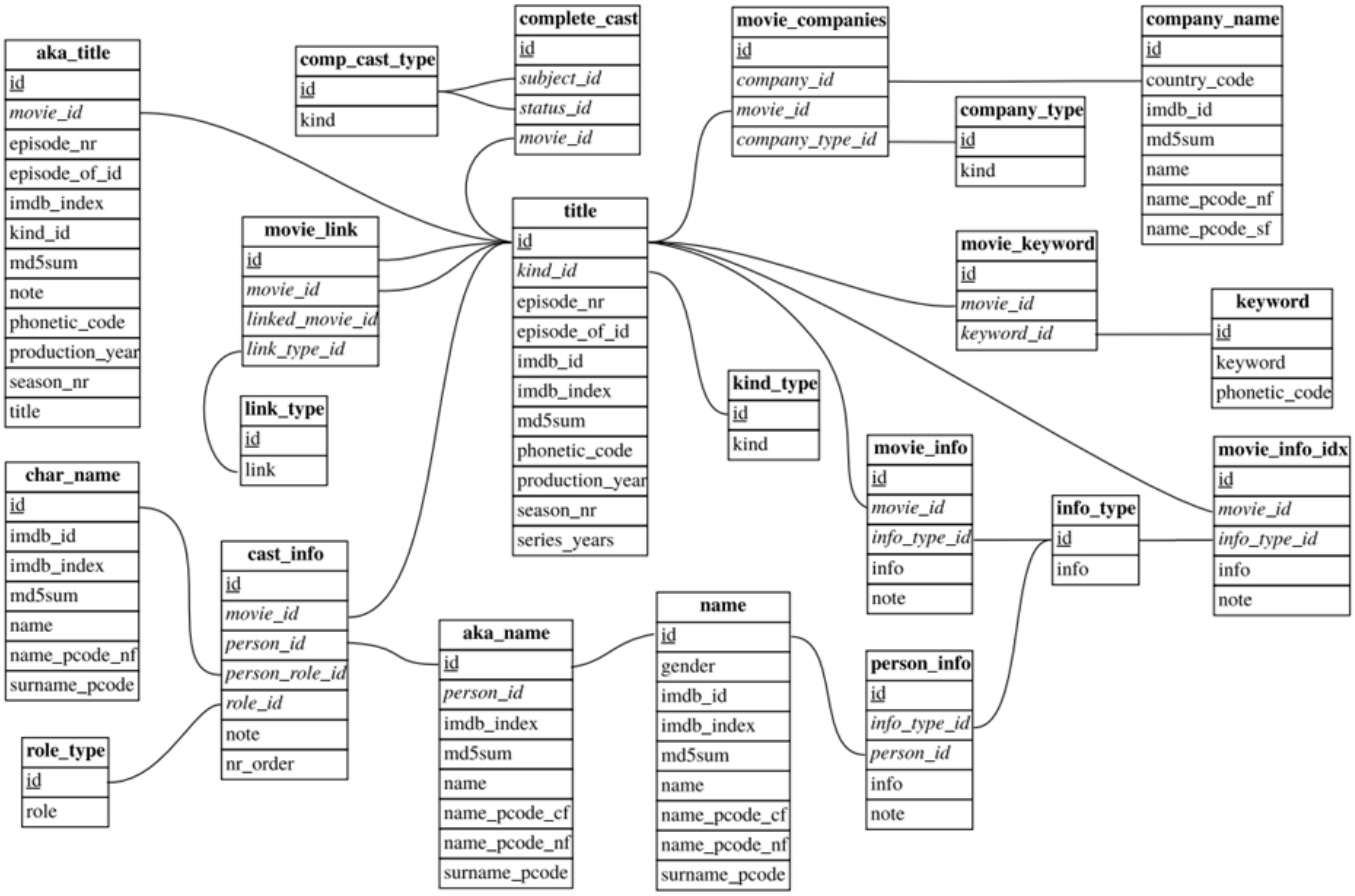

Fig. 2. Schema diagram of the JOB dataset showing key relationships between the 21 tables used in the benchmark [22]

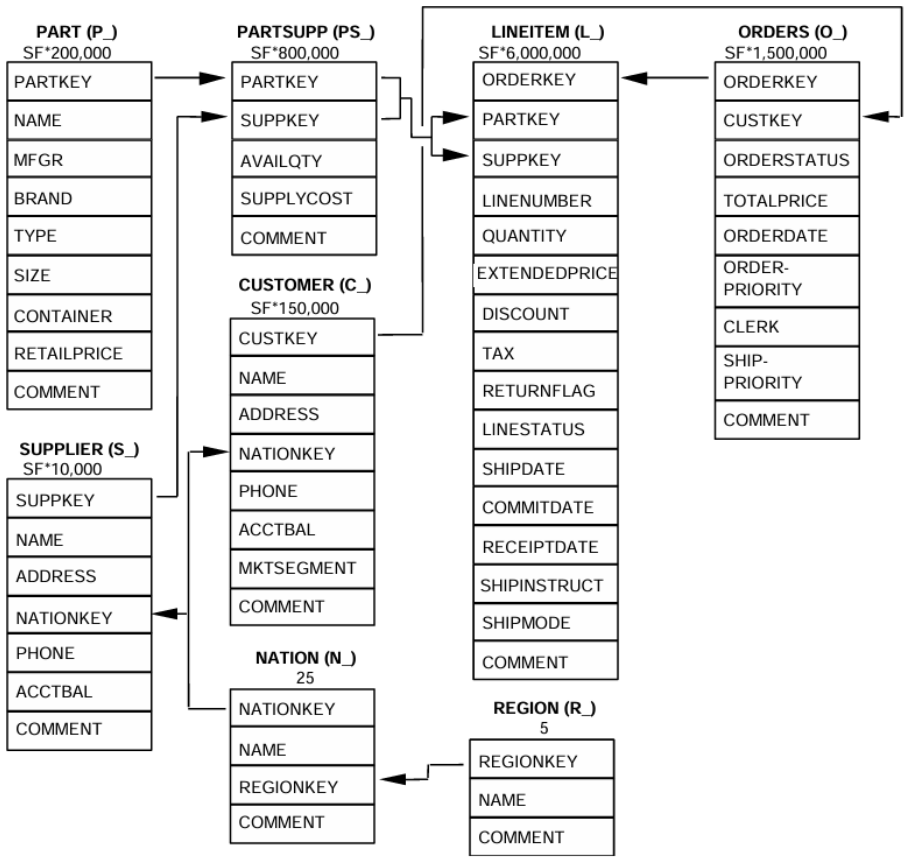

Fig. 3. Schema of the TPC-H benchmark [23].

B. Implementation and Training Details

All reinforcement learning agents were implemented in Python and integrated with PostgreSQL using the psycopg2 interface. During training, PostgreSQL cost estimates were obtained via EXPLAIN, while final performance evaluation used EXPLAIN ANALYZE to measure actual execution time. Training and evaluation were conducted on the same CPU-only machine described above, without GPU acceleration.

Network architectures:

- DQN: Multilayer perceptron (MLP) with two hidden layers of 128 units and ReLU activation functions.
- A2C: Shared actor–critic backbone with two hidden layers of 128 units, ReLU activations, followed by separate actor and critic output heads.
- PPO: Stable-Baselines3 MLP policy with two hidden layers of 128 units and ReLU activations.

Key hyperparameters:

- DQN: learning rate = 5×10^{-4}, discount factor $\gamma = 0.99$, batch size = 64, replay buffer size = 50,000, target network update every 1,000 steps, and ε-greedy exploration with ε decaying from 1.0 to 0.05 over training.
- A2C: learning rate = 7×10^{-4}, discount factor $\gamma = 0.99$, entropy coefficient = 0.01, rollout length = 16 steps.
- PPO: learning rate = 3×10^{-4}, discount factor $\gamma = 0.99$, clip range = 0.2, batch size = 64, and 10 policy-update epochs per batch.

Training time and stopping criteria - Each algorithm was trained on 90 JOB queries and evaluated on 23 unseen JOB queries and 22 TPC-H queries. Training ran for 5,000 episodes per algorithm on CPU-only hardware, taking several hours. Training stopped upon reaching the episode limit, with convergence monitored using moving-average rewards.

C. Results and Analysis

Raw execution times are reported for each query on the JOB and TPC-H benchmarks. Table I presents results where each row corresponds to a query and each column shows execution time (seconds) under a specific planner or RL method. PG-Old denotes PostgreSQL without join reordering, PG-Join refers to the built-in optimizer with reordering enabled, and the remaining columns correspond to the RL-based planners (PPO, DQN, A2C). The evaluation compares the proposed methods against these two PostgreSQL baselines, while prior learned optimizers such as ReJOIN, DQ, and SkinnerDB are discussed for context but were not re-implemented.

TABLE I. EXECUTION TIMES ON JOB (23 TEST QUERIES) AND TPC-H (22 TEST QUERIES) BENCHMARKS (TIMES IN SECONDS)

Query	Join Order Benchmark (Job)						TPC-H Benchmark					
	PG-Old	*Conv. Time (s)*	*PG-Join*	*PPO*	*DQN*	*A2C*	*PG-Old*	*Conv. Time (s)*	*PG-Join*	*PPO*	*DQN*	*A2C*
Q1	2.62	0.0011	0.97	0.87	0.87	1.06	0.94	0.0137	0.94	0.94	0.94	0.94
Q2	0.72	0.003	0.51	0.44	0.47	0.51	4.32	0.003	2.17	1.86	2.01	2.08
Q3	1.08	0.0011	0.75	0.72	0.79	0.75	1.85	0.004	1.2	1.09	1.14	1.2
Q4	4.34	0.0004	1.7	1.52	1.67	1.71	2.41	0.0002	1.6	1.51	1.6	1.6
Q5	1.51	0.001	0.8	0.73	0.77	0.85	10.46	0.003	3.84	3.45	4.04	4.06
Q6	3.92	0.001	1.29	1.16	1.2	1.25	0.21	0.0016	0.21	0.21	0.21	0.21
Q7	2.46	0.0004	1.09	1.03	1.09	1.13	10.79	0.002	7.76	7.79	7.98	8.28
Q8	5.98	0.001	2.11	2	2.14	2.2	9.94	0.003	4.91	4.62	4.97	5.2
Q9	3.56	0.003	1.3	1.18	1.25	1.32	13.28	0.001	3.47	2.99	3.37	3.66
Q10	7.52	0.0011	3.01	2.78	2.92	3.08	3.67	0.0011	2.53	2.42	2.53	2.53
Q11	11.62	0.003	2.48	2.4	2.53	2.61	1.94	0.003	1.31	1.25	1.31	1.34
Q12	6.38	0.001	3.73	3.73	3.89	3.73	2.1	0.0002	1.02	0.99	1.02	1.02
Q13	8.87	0.007	3.87	3.75	3.92	4.15	1.44	0.0008	0.78	0.78	0.78	0.78
Q14	4.91	0.001	2.01	1.87	1.78	2.04	2.76	0.0004	1.42	1.42	1.42	1.42
Q15	5.08	0.0004	2.17	2.17	2.12	2.25	1.1	0.0003	0.75	0.75	0.78	0.82
Q16	9.15	0.005	3.04	2.9	3.06	3.15	0.85	0.003	0.43	0.43	0.43	0.43
Q17	7.21	0.0009	2.54	2.46	2.54	2.68	2.57	0.0031	1.29	1.29	1.29	1.29
Q18	12.04	0.002	4.03	3.85	3.95	4.27	11.32	0.002	3.96	3.96	4.16	4.35
Q19	25.47	0.003	5.28	5.02	5.28	5.8	1.52	0.003	1.27	1.27	1.27	1.27
Q20	8.1	0.001	2.62	2.62	2.75	2.89	2.89	0.001	2.08	2.08	2.08	2.08
Q21	15.66	0.0009	5.61	5.61	5.89	6.31	12.56	0.0005	3.74	3.87	4.01	4.08
Q22	10.48	0.0006	4.18	4.01	4.18	4.53	0.98	0.003	0.66	0.66	0.66	0.66
Q23	64.09	0.0007	6.8	6.8	7.14	7.61						

Note: PG-Old = PostgreSQL without join reordering; PG-Join = PostgreSQL with join reordering; PPO, DQN, A2C = trained reinforcement learning agents evaluated independently. TPC-H results are reported for 22 queries (Q1–Q22); Q23 is not included in the TPC-H workload.

Text The tables reveal the following trends:

- JOB Benchmark: PostgreSQL's join reordering (PG-Join) significantly outperforms the fixed join order (PG-Old), demonstrating the importance of join planning. For instance, Q23 shows a substantial reduction in execution time (64.09 s with PG-Old vs. 6.8 s with PG-Join), illustrating how poor join orders can lead to catastrophic slowdowns. The RL methods (PPO, DQN, A2C) consistently avoid these worst cases, with PPO often matching or slightly outperforming PG-Join (e.g., Q9 and Q14). DQN and A2C generally produce competitive results but are occasionally slower by approximately 5–10%.
- Overall on JOB: All RL agents substantially outperform PG-Old and typically achieve performance close to PG-Join. PPO demonstrates the most robust behaviors, rarely producing inferior plans compared to the PostgreSQL optimizer.
- TPC-H Benchmark: Performance differences are smaller because many TPC-H queries are less join-intensive and PG-Join already produces strong plans. PPO occasionally improves performance (e.g., Q9: 2.99 s vs. 3.47 s, ~14% faster), while in other cases it matches or slightly underperforms PG-Join (e.g., Q21). DQN and A2C show similar trends. Nevertheless, all RL methods avoid the severe slowdowns observed with PG-Old, indicating reasonable generalization despite being trained on a different workload.

To provide a fine-grained evaluation of performance, we present the execution time for each individual query across the JOB and TPC-H benchmarks. This analysis enables a detailed comparison between the default PostgreSQL optimizer (PG-Old), the rewritten JOIN-based queries (PG-Join), and the reinforcement learning approaches (PPO, DQN, and A2C), highlighting the effectiveness of query rewriting and learning-based optimization under varying query complexities.

The per-query execution time results on the JOB benchmark are shown in Figure 4. PG-Old exhibits large performance spikes on several queries (e.g., Q19, Q21, Q23), whereas PG-Join and all reinforcement learning methods achieve significantly lower execution times. PPO frequently matches or outperforms PG-Join, while DQN and A2C demonstrate comparable performance, with A2C occasionally being slightly slower. The logarithmic scale highlights the substantial performance gap between PG-Old and the optimized methods, particularly on more complex queries.

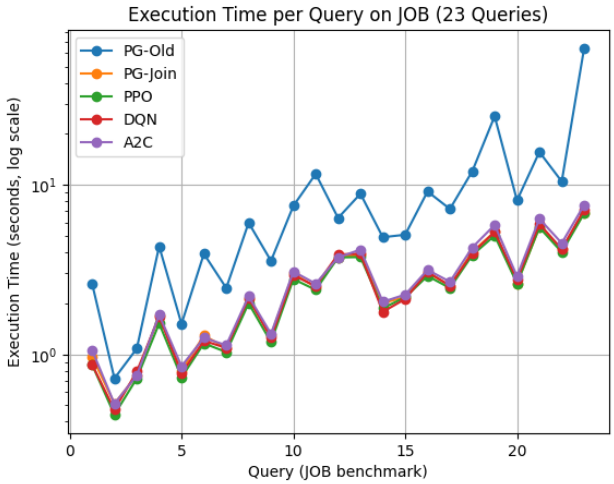

Fig. 4. Execution time per query on the JOB benchmark (log scale)

The execution time behavior on the TPC-H benchmark is shown in Figure 5. Compared to JOB, performance differences are less pronounced, although PG-Old remains slower on several queries. PG-Join and PPO show closely aligned performance, with PPO achieving marginal improvements in some cases, while DQN and A2C remain competitive but occasionally slightly slower. For queries with minimal join complexity (e.g., Q1 and Q6), all methods produce nearly identical runtimes. Overall, reinforcement learning approaches achieve near-optimal performance, as supported by the geometric mean speedups in Table II.

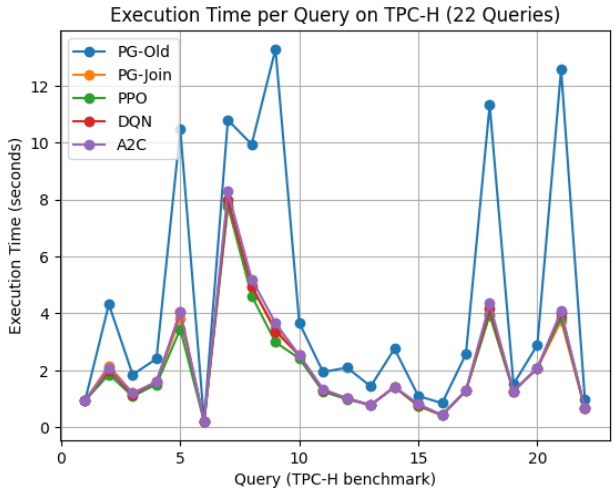

Fig. 5. Execution time per query on the TPC-H benchmark (linear scale)

TABLE II. AVERAGE SPEEDUP OF RL METHODS OVER POSTGRESQL BASELINES (GEOMETRIC MEAN OF RUNTIME RATIOS)

RL Method	JOB vs PG-Old	JOB vs PG-Join	TPC-H vs PG-Old	TPC-H vs PG-Join
PPO	4.47×	1.10×	1.75×	1.08×
DQN	4.18×	1.03×	1.62×	1.01×
A2C	3.90×	0.96×	1.53×	0.95×

From Table II, PPO achieves the best overall performance among the evaluated reinforcement learning methods. On the JOB benchmark, PPO obtains an average speed up of about 4.47× over PG-Old and around 10% improvement over PG-Join, indicating that the learned policy can occasionally outperform PostgreSQL's optimizer. DQN also shows strong results, achieving approximately 4.18× speedup over PG-Old and performance close to PG-Join. In contrast, A2C achieves about 3.90× improvement over PG-Old, but remains slightly slower than PG-Join.

A similar pattern appears on the TPC-H benchmark. PPO again provides the best results, achieving roughly 1.75× speedup over PG-Old and about 8% improvement over PG-Join, demonstrating good generalization. DQN achieves around 1.62× improvement over PG-Old and performs nearly equal to PG-Join, while A2C provides about 1.53× improvement over PG-Old but remains slightly below PG-Join.

Overall, PPO demonstrates the most consistent performance across both workloads, while DQN and A2C also provide clear improvements over the PG-Old baseline. In addition to runtime measurements, PostgreSQL EXPLAIN cost estimates were recorded during training, enabling comparison between optimizer cost predictions and actual execution latency.

D. Comparison with Previous Approaches

Our findings can be contextualized with prior learning-based optimizers, as summarized in Table III. ReJoin [11] approached the performance of a dynamic programming optimizer on JOB but required extensive training and faced stability issues [24]. In contrast, the PPO implementation in this study not only matched but slightly exceeded PostgreSQL's optimizer on JOB, showing that modern RL with execution feedback can surpass cost-based planners. DQ [12] achieved strong speedups and roughly matched a DBMS optimizer after fine-tuning; The DQN results indicate shows similar results, while PPO provides greater stability and higher performance, consistent with RL literature on policy-gradient methods. SkinnerDB [14] focused on avoiding worst-case outliers by testing multiple join orders during execution, but incurred runtime overhead. The PPO agent in this study avoids regressions proactively through training, achieving robustness and average performance gains with negligible overhead. Thus, unlike prior systems aimed mainly at stability, this work demonstrates both robustness and consistent improvements beyond PostgreSQL's optimizer.

TABLE III. COMPARISON OF OUR RL-BASED JOIN ORDER OPTIMIZATION WITH PRIOR LEARNING-BASED APPROACHES

Approach	Key Idea / Methodology	Reported Performance	Limitations	Comparison with This Work
ReJoin [11], [24]	Learned policy to approximate dynamic programming optimizer	Approached DP optimizer performance on JOB	Required extensive training, stability issues	Our PPO slightly exceeds PostgreSQL optimizer on JOB and is more stable
DQ [12]	Deep Q-learning for join ordering	Significant speedups over baseline; roughly matched DBMS optimizer after fine-tuning	Needed fine-tuning to match optimizer	Our DQN similarly matches optimizer, but PPO surpasses it, confirming policy-gradient superiority
SkinnerDB [14]	Robustness via on-the-fly join order testing to avoid worst-case plans	Prevented worst-case regressions; sometimes outperformed PostgreSQL	Incurs runtime overhead for testing multiple plans; not designed to beat optimizer's best case	Our PPO achieves robustness + average performance gains with negligible runtime overhead (pre-trained)
This Work (PPO, DQN, A2C)	RL with actual execution feedback and policy-gradient (PPO)	PPO: ~4.5× over PG-Old, ~10% faster than PG-Join (JOB); strong generalization	A2C slightly underperforms PG-Join on TPC-H	Demonstrates both robustness and consistent average gains, surpassing prior learning-based optimizers

VI. DISCUSSION

The experiments (Table III) demonstrate both the potential and limitations of reinforcement learning (RL) for join order optimization in PostgreSQL. Among the algorithms tested, Proximal Policy Optimization (PPO) consistently outperformed Deep Q-Network (DQN) and Advantage Actor-Critic (A2C), confirming its stability and effectiveness for complex query planning. DQN learned useful policies but was hindered by overestimation bias, while A2C achieved moderate improvements yet often converged to local optima. Notably, PPO trained on JOB generalized reasonably well to the distinct TPC-H workload, suggesting it captured transferable heuristics. Using PostgreSQL's cost model accelerated training but risked bias, so we validated with execution times to ensure real performance alignment. Overall, RL-generated plans frequently surpassed PostgreSQL's default, particularly on complex queries, though occasional regressions highlight the need for hybrid safeguards. These findings indicate that RL—especially PPO—can deliver robust, workload-aware optimization beyond traditional cost-based planners.

Portability of the proposed approach across database systems is an important consideration. While the reinforcement learning framework and query rewriting strategy are largely database-independent, certain components of the implementation are tied to PostgreSQL. In particular, the environment relies on PostgreSQL's EXPLAIN and EXPLAIN ANALYZE commands to obtain cost estimates and execution runtimes during training and evaluation. Other DBMS platforms such as MySQL, SQL Server, or Oracle provide similar plan-inspection interfaces, meaning that the reinforcement learning environment could be adapted with minimal modification. The query rewriting module that converts implicit joins to explicit JOIN ... ON syntax is also fully portable, since this SQL syntax is standardized across relational systems. Therefore, the overall learning framework is conceptually transferable to other relational DBMSs, although the integration layer used to extract cost estimates and execution statistics would need minor adjustments depending on the specific database engine.

VII. CONCLUSIONS

This study demonstrates the feasibility and effectiveness of reinforcement learning (RL) for join order optimization in PostgreSQL. By training and evaluating three RL algorithms—PPO, DQN, and A2C—on the JOB benchmark, we showed that RL-based planners can often outperform PostgreSQL's native optimizer and significantly improve execution times over static join orders. Among them, PPO delivered the most consistent and robust performance, while DQN and A2C also showed promising results with some variance. Notably, PPO generalized reasonably well to the TPC-H benchmark, indicating the potential of RL models to transfer learned strategies across different workloads. This work highlights that reinforcement learning can serve as a valuable complement to traditional cost-based optimizers, especially in complex queries or uncertain environments. Future directions include improving state representations, optimizing training efficiency, and extending RL to broader aspects of query planning within adaptive DBMS architectures.

REFERENCES

[1] J. Heitz and K. Stockinger, "Join Query Optimization with Deep Reinforcement Learning Algorithms," Nov. 2019, [Online]. Available: http://arxiv.org/abs/1911.11689

[2] P. Griffiths, S. M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," 1979.

[3] K.-M. Lee, I. Kim, and K.-C. Lee, "DQN-based Join Order Optimization by Learning Experiences of Running Queries on Spark SQL," in 2020 International Conference on Data Mining Workshops (ICDMW), IEEE, Nov. 2020, pp. 740–742. doi: 10.1109/ICDMW51313.2020.00107.

[4] Z. Yang, "Machine Learning for Query Optimization," EECS Department, University of California, Berkeley, 2022. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-194.html

[5] U. Gupta, "Harnessing Learning for Database Optimization: A Paradigm Shift," International Journal of Engineering Research & Technology (IJERT), vol. 12, no. 07, Jul. 2023, doi: 10.17577/IJERTV12IS070104.

[6] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?," Proc. VLDB Endow., vol. 9, no. 3, pp. 204–215, Nov. 2015, doi: 10.14778/2850583.2850594.

[7] K. Wang, J. Wang, Y. Li, N. Kallus, I. Trummer, and W. Sun, "JoinGym: An Efficient Query Optimization Environment for Reinforcement Learning," Jul. 2023, [Online]. Available: http://arxiv.org/abs/2307.11704

[8] A. A. Abdullah, N. S. Mohammed, M. Khanzadi, S. M. Asaad, Z. Kh. Abdul, and H. S. Maghdid, "In-depth Analysis on Machine Learning Approaches," ARO-THE SCIENTIFIC JOURNAL OF KOYA UNIVERSITY, vol. 13, no. 1, pp. 190–202, May 2025, doi: 10.14500/aro.12038.

[9] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing N -relational joins," ACM Transactions on Database Systems, vol. 9, no. 3, pp. 482–502, Sep. 1984, doi: 10.1145/1270.1498.

[10] K. Wang, J. Wang, Y. Li, N. Kallus, I. Trummer, and W. Sun, "JoinGym: An Efficient Query Optimization Environment for Reinforcement Learning," Jul. 2023, [Online]. Available: http://arxiv.org/abs/2307.11704

[11] R. Marcus and O. Papaemmanouil, "Deep reinforcement learning for join order enumeration," in Proceedings of the 1st International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM 2018, Association for Computing Machinery, Inc, Jun. 2018. doi: 10.1145/3211954.3211957.

[12] Sanjay Krishnan, Zongheng Yang., Ken Goldberg., Joseph Hellerstein., and Ion Stoica, "Learning to Optimize Join Queries With Deep Reinforcement Learning," in IEEE International Conference on Program Comprehension, IEEE Computer Society, 2022, pp. 36–47. doi: 10.1145/nnnnnnn.nnnnnnn.

[13] A. A. Nafea, M. S. Ibrahim, A. A. Mukhlif, M. M. AL-Ani, and N. Omar, "An Ensemble Model for Detection of Adverse Drug Reactions," ARO-THE SCIENTIFIC JOURNAL OF KOYA UNIVERSITY, vol. 12, no. 1, pp. 41–47, Feb. 2024, doi: 10.14500/aro.11403.

[14] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis, "SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning," Jan. 2019, doi: 10.1145/3299869.3300088.

[15] L. Shao et al., "Design and implementation of real-time robot operating system based on freertos," J. Phys. Conf. Ser., vol. 1449, no. 1, p. 012115, Jan. 2020, doi: 10.1088/1742-6596/1449/1/012115.

[16] B. Zhang et al., "A demonstration of the ottertune automatic database management system tuning service," Proceedings of the VLDB Endowment, vol. 11, no. 12, pp. 1910–1913, Aug. 2018, doi: 10.14778/3229863.3236222.

[17] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning," Dec. 2013, [Online]. Available: http://arxiv.org/abs/1312.5602

[18] A. Brim, "Deep Reinforcement Learning Pairs Trading with a Double Deep Q-Network," in 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), IEEE, Jan. 2020, pp. 0222–0227. doi: 10.1109/CCWC47524.2020.9031159.

[19] V. Mnih et al., "Asynchronous Methods for Deep Reinforcement Learning," Feb. 2016, [Online]. Available: http://arxiv.org/abs/1602.01783

[20] 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, doi: 10.1109/ACCESS.2022.3190376.

[21] Y. Gu, Y. Cheng, C. L. P. Chen, and X. Wang, "Proximal Policy Optimization With Policy Feedback," IEEE Trans. Syst. Man Cybern. Syst., vol. 52, no. 7, pp. 4600–4610, Jul. 2022, doi: 10.1109/TSMC.2021.3098451.

[22] V. Leis et al., "Query optimization through the looking glass, and what we found running the Join Order Benchmark," The VLDB Journal, vol. 27, no. 5, pp. 643–668, Oct. 2018, doi: 10.1007/s00778-017-0480-7.

[23] "TPC Benchmark TM H Standard Specification Revision 2.17.1 TPC BENCHMARK TM H," 1993.

[24] R. Marcus and O. Papaemmanouil, "Towards a Hands-Free Query Optimizer through Deep Learning," Dec. 2018, [Online]. Available: http://arxiv.org/abs/1809.10212