JoinGym: An Efficient Join Order Selection Environment

Junxiong Wang* **Kaiwen Wang*** **Yueying Li** **Nathan Kallus**
Immanuel Trummer **Wen Sun**

Cornell University
{jw2544, kw437, yl3469, kallus, it224, ws455}@cornell.edu

Abstract

Join order selection (JOS), the ordering of join operations to minimize query execution cost, is a core NP-hard combinatorial optimization problem in database query optimization. We present JoinGym, a lightweight and easy-to-use reinforcement learning (RL) environment that captures both left-deep and bushy variants of the JOS problem. Compared to prior works that execute queries online, JoinGym has much higher throughput and efficiently simulates the cost of joins offline by looking up the intermediate table's cardinality from a pre-computed dataset. We provide such a cardinality dataset for 3300 queries based on real IMDb workloads, which is the largest suite its kind and may be of independent interest. We extensively benchmark several RL algorithms and find that the best policies are competitive with or better than Postgres, a strong non-learning baseline. However, the learned policies can still catastrophically fail on a small fraction of queries which motivates future research using JoinGym to improve generalization and safety in long-tailed, partially observed, combinatorial optimization problems.

1 Introduction

Reinforcement learning (RL) has demonstrated impressive successes in video games (Bellemare et al., 2013), robotics simulators (Tassa et al., 2018), and real-world tasks such as inventory management (Madeka et al., 2022). In this work, we focus on the database query optimization task of ordering join operations to minimize query execution cost, a problem called join order selection (JOS) which is also known as join order optimization or access path selection. JOS is an NP-hard combinatorial optimization problem (Ibaraki & Kameda, 1984) and RL is a promising modern approach (Krishnan et al., 2018; Marcus et al., 2019; Yang et al., 2022). Unfortunately, there do not exist realistic and efficient simulators for JOS which makes research quite expensive and time-consuming; in particular, cost models, *i.e.*, estimators for query cost, can be inaccurate and take seconds per evaluation, while live query execution can take hours to days on large queries. To fill this gap, we provide JoinGym, the first lightweight and easy-to-use JOS simulator that can efficiently simulate query costs in real-world databases. Our goal is to make JOS accessible to the machine learning (ML) community and to accelerate methodological research in learning-based data systems.

The key advantages of JoinGym are the following. First, JoinGym can simulate *thousands* of queries per second on a commodity laptop and can be easily parallelized via multi-processing. In contrast, prior query optimization environments used cost models (Mao et al., 2019) or executed live queries (Lim et al., 2023), which can be much more expensive in time, compute and setup. Second, JoinGym also adheres to the Gymnasium API (Farama Foundation, 2023) and is as easy-to-use and setup as CartPole or Mountain Car. Third, we furnish JoinGym with a large suite of 3300 queries derived from real workloads in the Internet Movie Database (IMDb). Our query suite is 30× larger than the standard Join Order Benchmark (JOB) (Leis et al., 2015) used by prior works (Mao

*Equal contribution.

et al., 2019; Marcus et al., 2019). We note that JOB queries are also included in JoinGym but we recommend our larger query set as it is more diverse and representative.

The main idea behind JoinGym's efficiency is to simulate the cost of individual joins completely *offline* by replaying from a pre-computed dataset. To model the cost of a join, we use the cardinality of the intermediate table produced by the join. We made this design choice for two main reasons: (1) cardinalities can be pre-computed while runtime metrics cannot be; and (2) minimizing cardinality has arguably the largest impact on runtime metrics, *e.g.*, latency and resource consumption (Lohman, 2014; Leis et al., 2015; Neumann & Radke, 2018; Kipf et al., 2019). For (1), cardinalities are deterministic and agnostic to the hardware or database system, which means they can indeed be pre-computed for efficient offline replay. As part of JoinGym, we provide a cardinalities dataset for 3300 IMDb queries that took weeks of total CPU time. We note our novel cardinality dataset may be of independent interest, *e.g.*, for cardinality estimation research, which is beyond the scope of this paper. For (2), numerous works have observed that large join result cardinalities are often the main culprit of bad runtime metrics, which we detail in Appendix E. For example, Lohman (2014) observed that inaccurate cost models can account for $\leq 30\%$ degradation in runtime metrics, while large cardinalities can cause runtime metrics to blow up by *many orders of magnitude*. Thus, by reducing JOS to its core, we provide a lightweight simulator that is practically useful for RL research.

We now outline the paper. In Section 2, we rigorously describe JOS and 2×2 popular problem variants which are all included in JoinGym. Namely, JoinGym supports both left-deep and bushy plans as well as toggling on and off Cartesian products, which trade-off the search space size for a slight bias in optimality (Leis et al., 2015). Then, in Section 3, we model JOS as a Partially Observable Contextual Markov Decision Process (POCMDP) and describe the state, action and reward representations in JoinGym. Finally, in Section 4, we extensively benchmark standard RL algorithms on JoinGym and show that the best RL policies are competitive with or better than Postgres, a strong non-learning baseline. However, we still observe that RL algorithms in JOS face three key challenges: (1) long-tailed return distributions, (2) generalization in discrete combinatorial problems, (3) partial observability. Not typically captured by video game or robotic simulators, these challenges are understudied which motivates future research with JoinGym to develop better algorithms for systems applications. To summarize, our main contributions[1] are:

1. We provide a lightweight JOS simulator that is faster and cheaper than cost models or executing real queries. JoinGym supports left-deep and bushy plans, as well as toggling Cartesian products.

2. We release a cardinalities dataset of 3300 queries on IMDb, which is $30\times$ larger than the Join Order Benchmark. This dataset may be useful beyond JoinGym, *e.g.*, for cardinality estimation.

3. We extensively benchmark RL algorithms and find that the learned policies can attain lower cardinality than Postgres. However, they can still fail to generalize on 10% of queries, motivating future research with JoinGym to address safety in long-tailed combinatorial problems.

1.1 Related Works

There is a rich literature on applying ML and RL to database query optimization that can be divided into two types. First, numerous works show that learning-based approaches can be more effective than traditional non-learning approaches in query optimization (Yang, 2022; Marcus et al., 2019; Krishnan et al., 2018; Yang et al., 2022; Marcus et al., 2021; Gunasekaran et al., 2023; Trummer et al., 2021; Wang et al., 2023a;b) and general system optimization (Wang et al., 2021a;b; 2022). This first category aims to directly improve the state-of-the-art query optimizers. Second, there are "environment and benchmark" contributions that aim to provide a Gym-like interface for query optimization (Mao et al., 2019; Lim et al., 2023). This second category aims to make query optimization a useful test-bed for RL researchers, which provides unique challenges that are not captured by existing environments. Then, the new insights and algorithms can hopefully lead to real improvements in query optimizers

[1] Code is available at https://github.com/kaiwenw/JoinGym.

(first category). Our main contribution is a lightweight simulator for JOS and hence falls into the second category.

RL Environments for Query Optimization (Second Category). Park (Mao et al., 2019) and DB-Gym (Lim et al., 2023) are Gym-like interfaces for an RL agent to act as the query optimizer in a database management system (DBMS). In these environments, the reward signal can be derived from either the DBMS's cost model, which estimates runtime metrics, or the real physical runtime from an online execution of the join. Unfortunately, both require setting up a DBMS, *e.g.*, Postgres or Calcite, and may be slow and computationally expensive: cost models can take seconds per evaluation and can have estimation errors, while true physical runtime can take hours or days on a commercial database for large queries (*e.g.*, *q*29_44, *q*29_80 in our query suite). The key difference in JoinGym is how our reward signal is defined and computed: JoinGym's reward is derived from the true intermediate result cardinality which is a good proxy for true runtime (see Lohman (2014) and Appendix E) and has the advantage of being system-agnostic, deterministic, and hence pre-computable. Moreover, there is no estimation error from cost models as we use true cardinalities. Since we have pre-computed all the cardinalities already, JoinGym is very lightweight and efficient as it computes rewards by replaying cardinalities *offline*. JoinGym can simulate thousands of trajectories per second on a standard laptop, which is several orders of magnitude faster than prior environments.

RL for Query Optimization (First Category). DQ (Krishnan et al., 2018) applies Deep Q-learning with data collected by a cost model to learn a competitive policy that is faster to execute than the native optimizer. Neo (Marcus et al., 2019) first imitates an expert optimizer and then learns from real query executions with a tree-search algorithm. While DQ assumes an expert cost model and Neo assumes an expert optimizer, Balsa (Yang et al., 2022) bootstraps from a minimal cost model and learns on-policy with safe exploration, leading to faster convergence than Neo. Rather than replacing the query optimizer entirely, there have been works Marcus et al. (2021); Gunasekaran et al. (2023) that proposed a hybrid approach: use RL to tune the hyper-parameters of a native query optimizer. All these works rely on access to cost models or real query runtime as the reward signal, which as mentioned before can have estimation errors and are prohibitively slow and expensive. Thus, JoinGym provides an efficient simulator to enable rapid prototyping of algorithms whose insights can hopefully transfer over to real query optimizers.

2 Join Order Selection Background

A database consists of N_{tables} tables, $\texttt{DB} = \{T_1, T_2, \ldots, T_{N_{\text{tables}}}\}$, where each table T_i has a set of $N_{\text{cols}}(T_i)$ columns, $\text{Cols}(T_i) = \{C_{i,1}, C_{i,2}, \ldots, C_{i,N_{\text{cols}}(T_i)}\}$. A SQL query is described by a triple $q = (I, U, J)$. First, $I = \{i_1, \ldots, i_{|I|}\} \subset [N_{\text{tables}}]$ specifies the relevant tables for this query. Second, $U = \{u_i\}_{i \in I}$ specifies unary *filter* predicates such that for each $i \in I$, a filtered table $\widetilde{T}_i = u_i(T_i)$ is produced from keeping the rows of T_i that satisfy the predicate u_i. The fraction of rows that pass the filter is defined as the *selectivity*, $\text{Sel}_i = {|\widetilde{T}_i|}/{|T_i|}$. Third, $J = \{P_{i_1 i_2}\}_{i_1 \neq i_2 \in I}$ specifies binary *join predicates* that denote which columns should have matching values between two tables. Given two tables R and S and join predicates $P \subset [N_{\text{cols}}(R)] \times [N_{\text{cols}}(S)]$, define their binary join as

$$R \bowtie_P S = \{r \cup s \mid r \in R, s \in S, r_a = s_b \; \forall (a, b) \in P\}, \tag{1}$$

where $r \cup s$ means concatenating rows r and s, and $r_a = s_b$ stipulates that the a-th column of r matches the b-th column of s in value. Letting $\overline{P} = \{(b, a) : (a, b) \in P\}$, we restrict $P_{i_1 i_2} = \overline{P_{i_2 i_1}}$.

There are a combinatorial number of join orderings to compute q. For example, if $I = \{1, 2, 3, 4\}, U = \{u_1, u_2, u_3, u_4\}, J = \{P_{1,2}, P_{1,3}, \ldots\}$, two possible plans would be $\widetilde{T}_1 \bowtie_{P_{1,2} \cup P_{1,3} \cup P_{1,4}} (\widetilde{T}_2 \bowtie_{P_{2,3} \cup P_{2,4}} (\widetilde{T}_3 \bowtie_{P_{3,4}} \widetilde{T}_4))$ and $(\widetilde{T}_1 \bowtie_{P_{1,3}} \widetilde{T}_3) \bowtie_{P_{1,2} \cup P_{1,4} \cup P_{3,2} \cup P_{3,4}} (\widetilde{T}_2 \bowtie_{P_{2,4}} \widetilde{T}_4)$. The former involves the *intermediate results* (IRs) $\text{IR}_1 = \widetilde{T}_3 \bowtie_{P_{3,4}} \widetilde{T}_4$ and $\text{IR}_2 = \widetilde{T}_2 \bowtie_{P_{2,3} \cup P_{2,4}} \text{IR}_1$, while the latter involves $\text{IR}_1 = \widetilde{T}_1 \bowtie_{P_{1,3}} \widetilde{T}_3$ and $\text{IR}_2 = \widetilde{T}_2 \bowtie_{P_{2,4}} \widetilde{T}_4$. The IRs in each plan can have drastically different cardinalities, resulting in drastically different runtime metrics and resource consumption (Ramakrishnan

& Gehrke, 2003). The IR cardinality depends on the selectivity of base tables and the correlation of joined columns, which is not fully observed in general. To summarize, JOS is the problem of selecting the join order with the minimum cumulative IR cardinalities.

Left-Deep and Bushy Plans. Any join order is expressible as a binary tree where the leaves are the filtered base tables $\widetilde{T}_i$ and each internal node represents the IR from joining its two children. If no further restrictions are made on the binary tree structure, the join order is called *bushy*. To reduce the search space of join orders, one common restriction is to only allow *left-deep* trees, which corresponds to plans that iteratively join new tables with the IR of cumulative joins so far. In particular, joining two non-base-table IRs is allowed in bushy plans but disallowed in left-deep plans. Left-deep plans can often maintain reasonable fast query execution while reducing the search space (compared to bushy plans) by an exponential factor (Leis et al., 2015).

Toggling Cartesian Products. Given two tables R and S, the most expensive join is the Cartesian Product (CP), where no constraints are placed on the column values, *i.e.*, the CP between R and S is $R \bowtie_\emptyset S = \{r \cup s \mid r \in R, s \in S\}$, which always has cardinality equal to $|R||S|$. Disabling (*i.e.*, avoiding) CPs is a heuristic to rule out these expensive joins (Ramakrishnan & Gehrke, 2003) at a possible loss of optimality. In some rare queries, the optimal plan may indeed contain CPs since it may be beneficial to CP two small tables before joining a large table (Vance & Maier, 1996).

3 JoinGym: An RL Environment for Join Order Selection

We now formulate JOS as a Partially Observable Contextual Markov Decision Process (POCMDP) which is efficiently simulated by JoinGym. Abstractly, a POCMDP consists of context space $\mathcal{X}$, state space $\mathcal{S}$, finite action space $\mathcal{A}$, horizon H, transition kernel $P(s' \mid s, a)$, and contextual rewards $r(s, a; x)$, where $s, s' \in \mathcal{S}, a \in \mathcal{A}, x \in \mathcal{X}$. JOS can be viewed as playing a sequence of joins (actions) to maximize cumulative rewards (inverse IR cardinalities); the trajectory is generated as follows. First, a query q is sampled and encoded as context $x \in \mathcal{X}$, which is fixed for this trajectory. Then, for $h = 1, 2, \ldots$, the state s_h encodes the *partial join order* that has been executed so far and the action a_h specifies the next join to perform. For general bushy plans, a_h can be any pair of unjoined tables (*i.e.*, any edge in the join graph); while for left-deep plans, a_h is a single unjoined table (the next table to join with the left-deep tree). Next, the join specified by a_h results in an IR with cardinality c_h. We define the reward as $r_h \propto {}^{C^\star_{\text{plan_type}}}/_{H} - c_h$, where $C^\star_{\text{method}}$ is the minimum cumulative cardinality, for plan_type $\in \{\text{bushy}, \text{left-deep}\}$. For numerical stability, we clip each c_h by $100C^\star_{\text{plan_type}}$. The cumulative reward is non-positive with zero being optimal. This procedure iterates until all tables are joined. For bushy plans, the horizon is $H = |J| = |I| - 1$. For left-deep plans, the horizon is one larger since the a_1 stages the first table; since staging does not perform any joins, we set $r_1 = 0$ for left-deep plans. The POCMDP is summarized in Table 1 and example trajectories are in Fig. 4.

Remarks. First, the set of legal actions shrinks throughout the trajectory since completed joins cannot be re-selected. That is, if $\mathcal{A}_h$ represents the valid actions at time h, we have $\mathcal{A} = \mathcal{A}_1 \supset \mathcal{A}_2 \supset \cdots \supset \mathcal{A}_H$. We handle this by *action masking* (Huang & Ontañón, 2022), where we constrain the policy's actions and update rules to consider only legal actions at each step.[2] Second, the transition and rewards are deterministic, and the only stochasticity of the environment comes from sampling of the queries, *i.e.*, sampling of context. Third, the reward is contextual but the transition kernel is not. Our POCMDP formulation can be interpreted as a latent MDP (Kwon et al., 2021): each query is an MDP but we only see a partially observable context that cannot fully recover the query.

Partial Observability. As made precise in Section 3.1, the context x is a lossy encoding of the query q. That is, the information in x is not fully predictive of the IR cardinalities for query q, which creates partial observability in the contextual reward function. This is unavoidable since the contents

[2] Alternatively, we could penalize illegal actions with a very negative reward, but then the policy would need to learn to avoid such actions.

of the data tables are needed to fully determine the cardinality, but it is prohibitively expensive to use the entire table as the context. Hence, how to best compress tables into a feature vector is still an active area of research (Ortiz et al., 2018; Marcus et al., 2019; Yang, 2022). We describe our encoding scheme in Section 3.1. JoinGym can be easily updated to handle other encodings.

	Left-deep JoinGym	Bushy JoinGym
Context x	Query encoding described in Section 3.1.	
State s_h	Partial plan encoding described in Section 3.1.	
Action a_h	Table to join, from Discrete(N_{tables})	Edge to join, from Discrete($\binom{N_{\text{tables}}}{2}$)
Reward r_h	Negative step-wise regret: $r_h \propto C^\star_{\text{plan_type}}/H - c_h$ for plan_type $\in$ {left-deep, bushy}	
Transition P	Deterministic transition of dynamic state features, described in Section 3.1.	
Horizon H	$\lvert I \rvert$	$\lvert I \rvert - 1$

Table 1: POCMDP components for query $q = (I, U, J)$. The key difference between left-deep and bushy is their action.

3.1 Context and State Encoding

We encode each query q as a context $x = (v^{\text{Sel}}(q), v^{\text{goal}}(q))$ with two main components. First, the *selectivity encoding* is a vector $v^{\text{Sel}}(q) \in [0,1]^{N_{\text{tables}}}$ where the t-th entry is the selectivity of u_t if $t \in I$, and 0 otherwise. Second, the *query encoding* is a binary vector $v^{\text{goal}}(q) \in \{0,1\}^{N_{\text{cols}}}$ (where $N_{\text{cols}} = \sum_T N_{\text{cols}}(T)$) that represents which columns need to be joined for this query; the c-th entry is 1 if column c appeared in any join predicate, and 0 otherwise. Formally:

$$\forall t \in [N_{\text{tables}}], v^{\text{Sel}}(q)[t] := \text{Sel}_t \cdot \mathbb{I}\left[t \in I\right],$$
$$\forall c \in [N_{\text{cols}}], v^{\text{goal}}(q)[c] := \mathbb{I}\left[\exists (R,S,P) \in J, \exists p \in P : c = p[0] \vee c = p[1]\right],$$

where $\mathbb{I}[\cdot]$ is 1 if $\cdot$ is True and 0 otherwise.

While the context x encodes the query and stays constant through the trajectory, the state s_h encodes the partial join order and evolves through the trajectory. In particular, the state s_h at step $h \in [H]$ is the *partial plan vector* $v^{\text{pp}}_h \in \{-1, 0, 1, 2, \ldots\}^{N_{\text{cols}}}$ which represents the joins specified by prior actions $a_{1:h-1}$. The c-th entry of v^{pp}_h is (i) positive if column c has already been joined, (ii) -1 if the table of column c has been joined or selected but column c has not been joined yet, and (iii) 0 otherwise. We now explain each case in order.

Case (i) marks joined columns c, where the c-th entry is the index of its join-tree in the graph representing the join plan. In left-deep plans, there is only one join-tree so this value will always be 1. In bushy plans, there may be more than one tree in the graph, so the value is the tree index which can be larger than 1. Case (i) is important since the policy should know which columns have been joined to choose the next valid join.

Case (ii) marks unjoined columns belonging to joined or selected tables, and we use the special value -1. For example, in left-deep plans, we must be able to tell which table was selected by a_1 at $h = 2$, even though said table has only been "staged" but not joined. Beyond this special case, another important use-case of the -1 marking is that marked columns have potentially small IR; perhaps the rows of the table has been filtered from a prior join and so it is better to join with this table rather than an unjoined base table.

Case (iii) marks the remaining columns of unjoined tables with the special value 0. In sum, our partial plan vector v^{pp} contains enough information to recover the current IRs and is a sufficient statistic for deducing the future cardinalities. We note that v^{pp} does not contain the join graph itself, which would be enough to deduce how the current IRs were formed and was used in prior works (Marcus et al., 2019). This additional structure is not necessary for predicting future cardinalities and hence we omit it from the state vector. We provide example context and state vectors in Fig. 4(d).

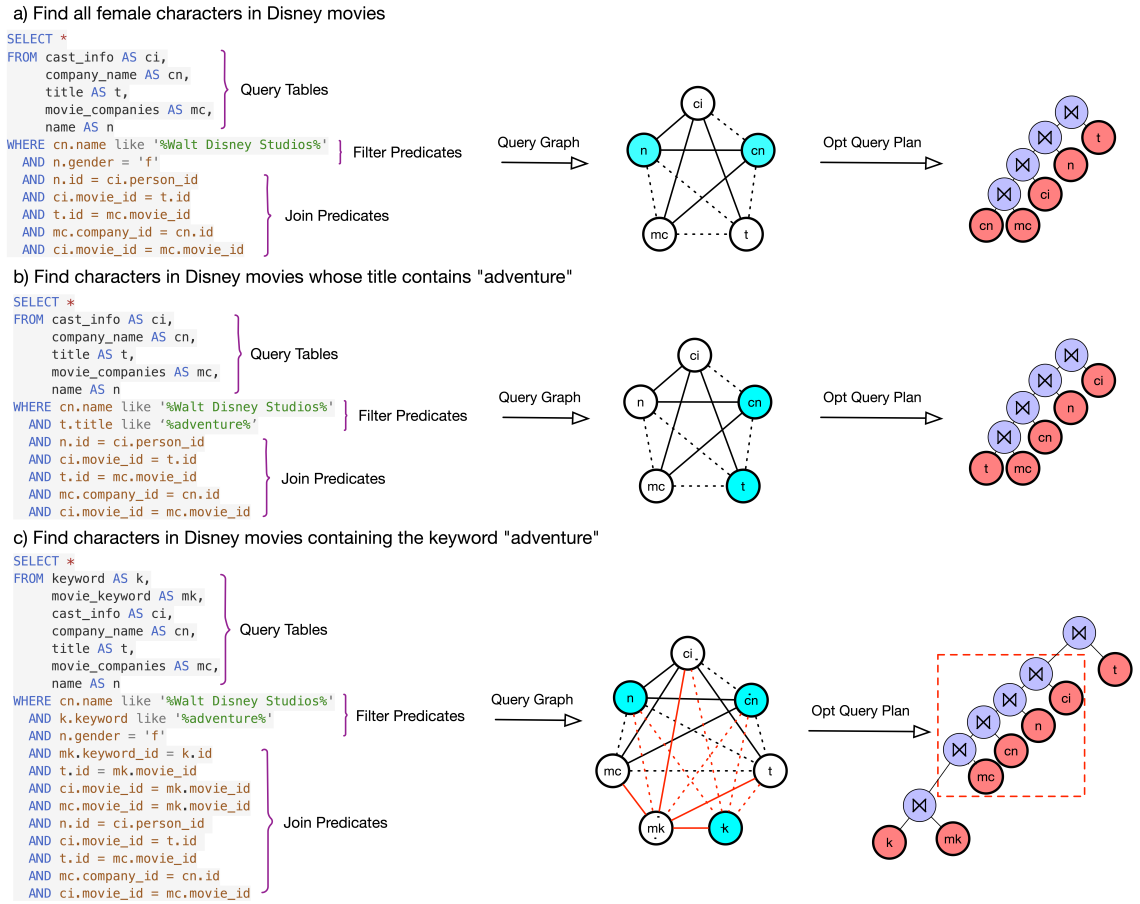

Figure 1: On the left are three SQL queries. The middle are their query graphs where each edge represents a join between two tables. Filtered tables are denoted by blue nodes and CPs are denoted by dashed lines. On the right, we show tree representations of the query plan, where each leaf is a table and each node is a join of its two children tables. Queries (a) and (b) are derived from the same template and so share an identical query graph but their optimal query plans are different due to different filters on the base tables. Query (c) is from a different template, but it contains (b) as a subgraph and their optimal query plans share a common sub-tree (highlighted in red).

3.2 The Query Suite and Cardinality Dataset

We furnish JoinGym with a suite of 3300 queries simulating real workloads on the Internet Movie Database (IMDb). Each query has a query_id of the form qN_M with $N \in [33], N \in [100]$ signifying that it is the M-th query from the N-th template, where the 33 templates are from the Join Order Benchmark (JOB; Leis et al., 2015). We selected the queries to be representative of user searches on IMDb; we simulated user searches by generating popular search terms with ChatGPT, manually inspecting them to be sensible, and ensuring they have non-empty search results. Please see Appendix G for more details and examples of our query selection process. The 113 JOB queries are also included in JoinGym, although we recommend our new query suite since it is 30× larger and more diverse than the JOB.

We then computed a dataset of all IR cardinalities for the query suite. That is, for each query, we computed the cardinalities of all possible intermediate tables from valid join orders using MonetDB. The exhaustive cardinality computation was a computationally heavy task that require online query execution and took weeks to complete on hundreds of CPUs. However, once this dataset is collected

(each join plan only needs to be executed once to observe the cardinality), JoinGym can use it to efficiently simulate costs of join plans entirely offline.

3.3 Query Templates and Generalization in Query Optimization

When interacting with a database for a real-world task (*e.g.*, searching a movie), users typically specify their searches with drop down menus and natural language rather than writing SQL. The drop down menus and natural language are copied into *query templates*, which can automatically construct a SQL query reflecting the user's interest. A query's template determines its final query graph and different query templates may often share common subgraphs. Using the query graph structure, deep RL models can *generalize* to improve future query execution planning. For example in Fig. 1, the optimal join plan for (b) is a sub-tree of the optimal plan for (c). However, while queries with the same template have a common query graph, optimal join orders can vary significantly due to different filter conditions that are applied, *e.g.*, Fig. 1 (a) & (b) are instances of the same template (and hence share the same graph) but have different optimal join orders. Thus, the key challenge in data-driven query optimization is to learn which correct query instances to mimic based on the context (*i.e.*, filter predicates, query graph).

3.4 JoinGym API

We now describe the JoinGym API, which adheres to Gymnasium (Farama Foundation, 2023). The left-deep and bushy variants are registered under the environment-ids '`joinopt_left-v0`' and '`joinopt_bushy-v0`'. JoinGym can be instantiated with `env = gym.make(env_id, disable_cp, query_ids)`, where `disable_cp` is a Boolean for Cartesian products (described in Section 2), and `query_ids` specifies the queries to load. JoinGym implements the POCMDP in Section 3 with two functions:

(i) `state, info = env.reset(options={query_id=x})`: reset the env to represent the query with id `x`, and observe the initial state. If no `query_id` is specified, then a random query is picked from the query set.

(ii) `next_state, reward, done, _, info = env.step(act)`: perform the join specified by `act`. `done` is `True` when all tables of the current query have been joined. There is no truncation.

`state` is the concatenation of the context x and s_h (defined in Section 3.1) and `next_state` is the concatenation of x and s_{h+1}. Also, `info['action_mask']` is a multi-hot encoding of the valid actions $\mathcal{A}_h$, which the algorithm should into account, *e.g.*, mask out Q-values, so only valid actions are considered.

4 Benchmark Results on JoinGym

Experiment Setup. Recall that our new dataset contains 100 queries for each of the 33 templates from the JOB (Leis et al., 2015). For each template, we randomly selected $60, 20, 20$ queries for training (1980 queries), validation (660 queries) and testing (660 queries) respectively. We benchmarked four different RL algorithms: (i) an off-policy Q-learning algorithm Deep Q-Network (DQN) (Mnih et al., 2015); (ii-iii) two off-policy actor-critic algorithms, Twin Delayed Deep Deterministic policy gradient (TD3) (Fujimoto et al., 2018) and Soft Actor-Critic (SAC) (Haarnoja et al., 2018); and (iv) an on-policy actor-critic algorithm Proximal Policy Optimization (PPO) (Schulman et al., 2017). For DQN, we conducted an ablation with the Double Q-learning (Van Hasselt et al., 2016). For (i-iii), we conducted ablations with standard replay buffer (RB) vs. prioritized experience replay (PER) (Schaul et al., 2015).

Train and Eval. All algorithms were trained for *one million steps* on the training queries. Define *cumulative cost multiple* (CCM) as the cumulative IR cardinality of the join plan divided by the smallest possible cumulative IR cardinality for this query. This can be interpreted as a *multiplicative*

regret and lower is better. For each algorithm, we sweeped over multiple learning rates and selected the best hyperparameter according to average CCM on the validation queries. For the best hyperparameter of each algorithm, we evaluate its CCM distribution over the test queries in Table 3. Our results are averaged over 10 seeds and we report additional CCM statistics (mean, $p95$ & $p99$) with standard errors in Appendix I. The numerical tables and Fig. 2 show that the CCM distributions are long-tailed, which can be challenging for RL algorithms.

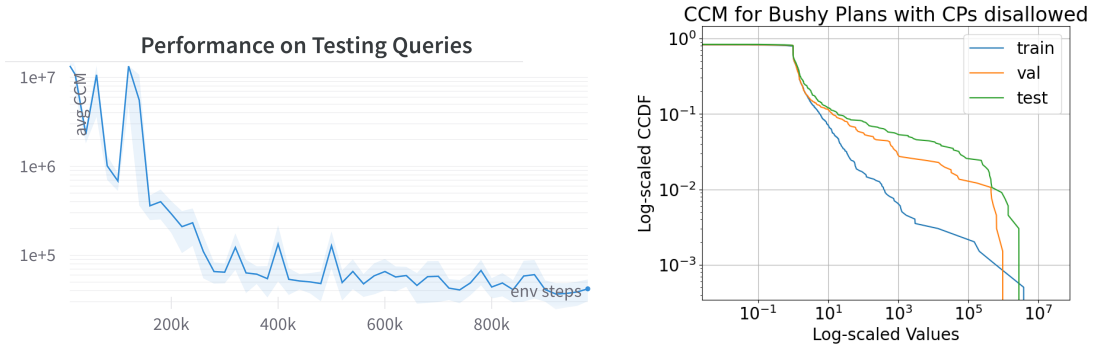

Figure 2: Left: learning curve of TD3 for 'left-deep, disable CP'; the y-axis is mean CCM, x-axis is no. update steps and shaded region is stderr over 10 seeds. Right: CCM distributions over train/val/test queries (viewed as complementary CDF) of the final policy for a seed.

Discussion. **1)** Algorithms uniformly perform much better in left-deep JoinGym than bushy JoinGym because the search space is exponentially smaller. For the same reason, algorithms uniformly perform better when CPs are disallowed. The hardest setting is bushy with CPs, where most algorithms diverge and those that converge have CCMs orders of magnitude worse than the other settings. **2)** Regarding generalization, the gap between the validation and test performance is only $2\times$ for $p90$ and it increases to $10\times$ for $p95$. For $p99$, both the validation and test performance are significantly worse than training. This exponential widening of the generalization gap is likely explained by the fact that the long tail is exactly where partial observability is more pronounced, causing catastrophic failure in planning of the policy. **3)** Off-policy actor critic methods (TD3 & SAC) achieve the best results for the mean and are also near-optimal for the quantiles. They are more sample efficient than PPO due to their sample reuse and more stable than DQN for the query optimization task. We also find that prioritized replay does not always improve performance.

Comparison with Postgres. We ran Postgres, a strong non-learning baseline, on the JoinGym query suite and show its CCM statistics in Table 2. As shown in Table 3 (and Tables 6 and 7), RL policies in the 'left-deep, disable CP' setup often yields lower cardinalities than Postgres. For example, PPO is consistently better for train, validation and test queries. This suggests that the learned policies are indeed proposing high-quality joins competitive with Postgres.

Data	Mean	p90	p95	p99
trn	1.7e+06	2.6e+03	4.9e+04	4.5e+06
val	5.8e+05	4.5e+03	1.0e+05	7.1e+06
tst	6.1e+04	5.0e+03	3.4e+04	2.3e+06

Table 2: CCM statistics of Postgres across train, validation and test queries of JoinGym.

5 Conclusion

We provide JoinGym, an extremely lightweight yet realistic RL environment for join order selection that is many orders of magnitude faster than calling cost models or real query execution. We accomplish this via offline replay of *true* intermediate cardinalities using of a novel cardinality dataset of 3300 queries, which is $30\times$ larger than the JOB and may be of independent interest. With JoinGym, we observed that standard RL algorithms can outperform Postgres in certain cases, but their generalization sharply declines for a small fraction of queries ($\sim 10\%$). Thus, we hope that

90% **Quantile**			DQN		DDQN		TD3		SAC		PPO
			RB	PER	RB	PER	RB	PER	RB	PER	
disable CP	bushy	trn	7.3	**4.4**	5.2	5.6	5.3	7.3	13	9.5	6
		val	16	**13**	14	15	15	18	18	**13**	23
		tst	46	**25**	30	30	26	40	55	33	42
	left	trn	5.5	5.6	7	6.5	6.9	**5.2**	11	8.6	**5.2**
		val	12	15	13	14	13	**9.5**	20	14	11
		tst	28	30	34	34	22	20	39	32	**19**
enable CP	bushy	trn	6.4e+04	2.4e+05	4.6e+04	3.2e+04	1.8e+02	42	7.7e+18	3e+14	**35**
		val	2e+05	1.1e+06	5.8e+04	6e+04	3.1e+02	1.4e+02	4e+18	2.8e+14	**1e+02**
		tst	1.6e+05	1.2e+05	2.1e+05	6.9e+04	2e+03	4.9e+02	2.2e+17	2.1e+17	**2.8e+02**
	left	trn	17	**6.3**	17	9.9	3.6e+02	17	7.7	6.8	9.9
		val	25	15	27	22	2e+02	24	13	**12**	27
		tst	64	36	66	59	1.7e+03	1e+02	**28**	30	92

Table 3: The 90% quantile CCM (lower is better) over train (trn), validation (val) and test (tst) queries of JoinGym. RB stands for "replay buffer"; PER stands for "prioritized experience replay". The best performing algorithm is highlighted in each row. We report confidence intervals in Appendix I.

JoinGym can be a useful environment for developing new RL algorithms for combinatorial query optimization problems, particularly those with long-tailed returns and partial observability.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant Nos. IIS-1846210, IIS-2154711 and CAREER 2339395.

References

Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 34–43, 1998.

Farama Foundation. Gymnasium, 2023. URL https://github.com/Farama-Foundation/Gymnasium.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pp. 1587–1596. PMLR, 2018.

Karthick Prasad Gunasekaran, Kajal Tiwari, and Rachana Acharya. Deep learning based auto tuning for database management system. *arXiv preprint arXiv:2304.12747*, 2023.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.

Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. Cardinality estimation in dbms: A comprehensive benchmark evaluation. *arXiv preprint arXiv:2109.05877*, 2021.

Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. In *The International FLAIRS Conference Proceedings*, volume 35, 2022.

Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, 1984.

Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*, pp. 1937–1940, 2019.

Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

Aviral Kumar, Rishabh Agarwal, Tengyu Ma, Aaron Courville, George Tucker, and Sergey Levine. Dr3: Value-based deep reinforcement learning requires explicit regularization, 2021.

Jeongyeol Kwon, Yonathan Efroni, Constantine Caramanis, and Shie Mannor. Rl for latent mdps: Regret guarantees and a lower bound. *Advances in Neural Information Processing Systems*, 34: 24523–24534, 2021.

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.

Wan Shen Lim, Matthew Butrovich, William Zhang, Andrew Crotty, Lin Ma, Peijing Xu, Johannes Gehrke, and Andrew Pavlo. Database gyms. In *CIDR 2023, Conference on Innovative Data Systems Research*, volume 14, pp. 1241–1253, 2023.

Guy Lohman. Is query optimization a "solved" problem. In *Proc. Workshop on Database Query Optimization*, volume 13, pp. 10. Oregon Graduate Center Comp. Sci. Tech. Rep, 2014.

Dhruv Madeka, Kari Torkkola, Carson Eisenach, Anna Luo, Dean P Foster, and Sham M Kakade. Deep inventory management. *arXiv preprint arXiv:2210.03137*, 2022.

Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. Park: An open platform for learning-augmented computer systems. *Advances in Neural Information Processing Systems*, 32, 2019.

Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pp. 1–4, 2018.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, jul 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342644. URL https://doi.org/10.14778/3342263.3342644.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pp. 1275–1288, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3452838. URL https://doi.org/10.1145/3448016.3452838.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Thomas Neumann and Bernhard Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 677–692, 2018.

Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, pp. 1–4, 2018.

Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pp. 23–34, 1979.

Ji Sun and Guoliang Li. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment*, 13(3), 2020.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *ACM Transactions on Database Systems (TODS)*, 46(3):1–45, 2021.

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.

Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. *ACM SIGMOD Record*, 25(2):35–46, 1996.

Junxiong Wang, Immanuel Trummer, and Debabrota Basu. Demonstrating udo: A unified approach for optimizing transaction code, physical design, and system parameters via reinforcement learning. In *Proceedings of the 2021 International Conference on Management of Data*, pp. 2794–2797, 2021a.

Junxiong Wang, Immanuel Trummer, and Debabrota Basu. Udo: universal database optimization using reinforcement learning. *Proceedings of the VLDB Endowment*, 14(13):3402–3414, 2021b.

Junxiong Wang, Debabrota Basu, and Immanuel Trummer. Procrastinated tree search: Black-box optimization with delayed, noisy, and multi-fidelity feedback. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 10381–10390, 2022.

Junxiong Wang, Mitchell Gray, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. Demonstrating adopt: Adaptively optimizing attribute orders for worst-case optimal joins via reinforcement learning. *Proc. VLDB Endow.*, 16(12):4094–4097, aug 2023a. ISSN 2150-8097. doi: 10.14778/3611540.3611629. URL https://doi.org/10.14778/3611540.3611629.

Junxiong Wang, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. Adopt: Adaptively optimizing attribute orders for worst-case optimal join algorithms via reinforcement learning. *Proceedings of the VLDB Endowment*, 16(11):2805–2817, 2023b.

Zongheng Yang. *Machine Learning for Query Optimization*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2022. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-194.html.

Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: one cardinality estimator for all tables. *Proceedings of the VLDB Endowment*, 14(1): 61–73, 2020.

Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. Balsa: Learning a query optimizer without expert demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*, pp. 931–944, 2022.

Appendices

A List of Abbreviations and Notations

Table 4: List of Abbreviations

JOS	Join Order Selection
DB	Database
IR	Intermediate result table
CP	Cartesian product join
JOB	Join Order Benchmark (Leis et al., 2015)
CCM	Cumulative Cost Multiple

Table 5: List of Notations

N_{tables}	Number of tables in the DB
$N_{\text{cols}}(T)$	Number of columns in table T
N_{cols}	Total number of columns amongst all tables in DB
$A \bowtie_P B$	Binary join operator, defined in Eq. (1)

B Statistics about JoinGym

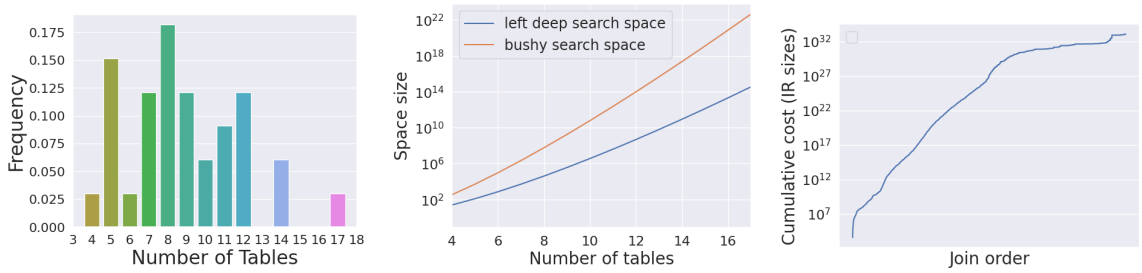

Figure 3: Left: distribution of the number of tables in JoinGym. Middle: size of search space in JoinGym. Right: cumulative cost (IRs) of different join orders for query q23_33.

Search space of Left-deep vs. Bushy plans. Recall that left-deep plans only allow for left-deep join trees, while bushy plans allow for arbitrary binary trees. We can compute the size of the search space for both types of plans, which is a simple exercise in combinatorics. With $|I|$ tables, there are $|I|!$ possible left-deep plans and $|I|!C_{|I|-1}$ bushy plans, where the k-th Catalan number C_k is the number of unlabeled binary trees with $k+1$ leafs. By Stirling's approximation, $n! \approx \Theta(\sqrt{n}(\frac{n}{e})^n)$ and $n!C_{n-1} \approx \Theta(n^{-1}(\frac{4n}{e})^n)$. The middle of Fig. 3 illustrates the exponential growth of two different search spaces (*i.e.*, left-deep search space and bushy search space) as the number of join tables increases. The bushy search space exhibits even faster growth compared to the left-deep plans. In our most challenging query template, search space exceeds 10^{15} for left-deep plans and surpasses 10^{23} for bushy plans. While left-deep plans usually suffice for fast query execution, bushy plans can sometimes yield join plans with smaller IR cardinalities and faster runtime (Leis et al., 2015).

Statistics of queries in JoinGym. The left of Fig. 3 shows the distribution of the number of join tables in JoinGym. More than half of the queries join at least nine tables. The right figure shows the sorted cumulative IR sizes of different join orders for the same representative query. The

left-most point is the optimal plan. The sharp jump in cardinalities from the optimal illustrates the key challenge of query optimization.

C Illustration of POCMDP and State Encodings

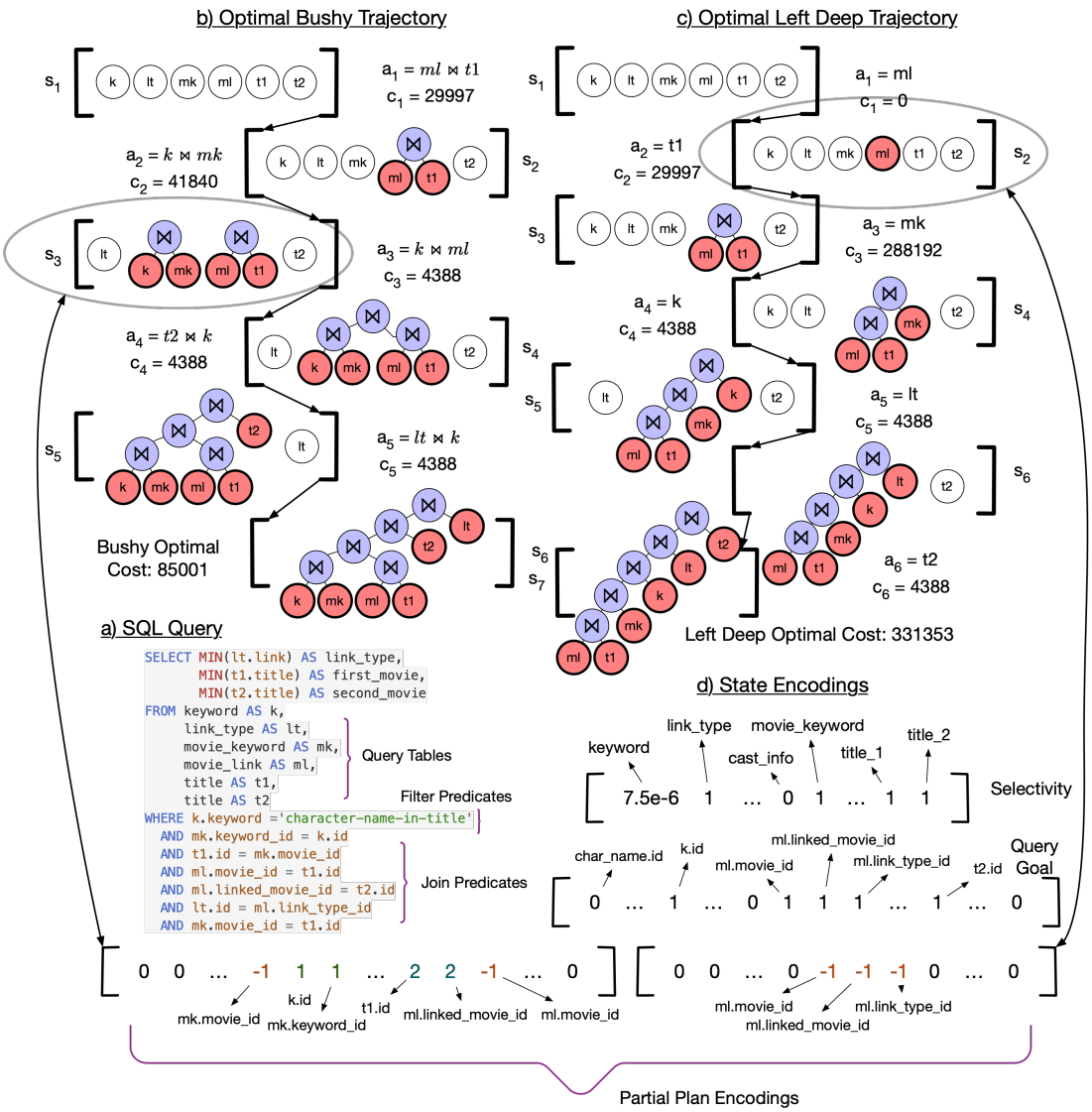

Figure 4: (a) query 110 from the JOB. (b) is its optimal bushy join plan and (c) is its optimal left-deep join plan. c_h denotes the cardinality of the IR incurred at time h. (d) shows the context (query encoding) for the query in (a) and also shows two state vectors (partial plan encoding) for left-deep and bushy states, as described in Section 3.1.

D Further Comparison to Related Works

Join Order Selection. Search engines and online transactions crucially hinge on the ability to execute queries efficiently and at massive scale (Selinger et al., 1979; Chaudhuri, 1998). There are several research areas in query optimization including selecting join order, selecting indices, materialized views. Among these, optimizing the join order is arguably the most important to get right since it has the largest impact on execution time (Leis et al., 2015; Marcus et al., 2021; Gunasekaran

et al., 2023). However, classical approaches that make strict data correlation assumptions to estimate the IR cardinality can often be quite sub-optimal (Lohman, 2014). Towards a more data-driven approach, Yang et al. (2022); Marcus et al. (2019); Krishnan et al. (2018) showed that RL is a promising tool for optimizing the join order, capable of learning policies with improved inference time and quality of join plans. These prior works are often benchmarked on the JOB, which consists of 33 query templates and 113 total queries. In JOB, queries from the same template have similar optimal query plans which does not require much generalization. In JoinGym, we provide a more diverse dataset of 3300 total queries, comprised of 100 queries per query template.

Cost Models and Cardinality Estimation. There are many ML-for-database works that aim to enhance cardinality estimation such as NeuroCard (Yang et al., 2020) and (Kipf et al., 2018; 2019; Sun & Li, 2020; Han et al., 2021). Instead of using estimated cardinalities, JoinGym uses the true cardinalities from our novel cardinality dataset. Hence, it does not make sense to use these cardinality estimation models in JoinGym since it would only introduce estimation error and slow-down simulation throughput. We note that our cardinality dataset, the largest of its kind, can be useful for research in cardinality estimation research and representation learning for table embeddings (Ortiz et al., 2018), which is beyond the scope of this paper.

State and context embeddings. We use the same selectivity encoding as Balsa (Yang et al., 2022) and Neo (Marcus et al., 2019). However, their query encoding is an adjacency matrix (at the table level) that preserves the tree structure, while we encode queries with a multi-hot vector (at the column level) marking which columns should be joined, similar to DP (Krishnan et al., 2018) and ReJoin (Marcus & Papaemmanouil, 2018). To the best of our knowledge, marking the tree's component index in the partial plan encoding is novel and allows us to handle bushy plans without keeping track of the whole tree; except DP, the aforementioned works only consider left-deep trees. Note that since the graph structure does not influence future IR sizes but column information does, our encoding is more compact than prior encoding schemes of Balsa and Neo. JoinGym is designed so that one can easily change the state and context encoding schemes without needing to collect any more data, which is the costly step of building JoinGym that we have already finished.

E Cardinality as a good proxy for query cost

In JoinGym, one design choice was to use the cumulative IR cardinality as cost which is a difference from prior works that optimized a DBMS's cost model or real runtime metrics. In this section, we answer the question: why is cumulative IR cardinality a good proxy for the cost of a query plan?

1. Lohman (2014) puts it eloquently: "The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities. In my experience, the cost model may introduce errors of at most 30% for a given cardinality, but the cardinality model can quite easily introduce errors of many orders of magnitude! Let's attack problems that really matter, those that account for optimizer disasters, and stop polishing the round ball." In short, minimizing the cardinality well is one of the most significant factors for finding the optimal join order.

2. Neumann & Radke (2018) adopts IR cardinalities as the key metric for benchmarking query optimization algorithms.

3. Numerous database papers focus on enhancing cardinality estimation with sketching/statistical methods (Kipf et al., 2019) and neural models (Kipf et al., 2018). Also, many database theory works (Atserias et al., 2013; Ngo et al., 2018) focus on designing new algorithms to minimize the size of the intermediate result.

4. Furthermore, the IR cardinality metric provides computational advantages: (i) IR cardinality does not depend on specific system configurations (*e.g.*, IO and CPU); (ii) IR cardinality is

deterministic so we can pre-compute it for all our queries; (iii) with our pre-computed dataset, users can simulate thousands of large joins per second.

In sum, cardinality is the most common and important metric for query optimization algorithms. At the same time, its system-independent and deterministic nature allows us to design a realistic environment that is lightweight, enabling ML & RL researchers from diverse communities to collaboratively tackle the core problem in query optimization.

F Implementation Details of JoinGym

We now describe the specific implementation details of JoinGym. This section is intended for advanced users who want to change how we encode state, actions or rewards. We appreciate any questions or feedback and welcome pull requests.

Our code registers two `gymnasium.Env` classes that implement bushy and left-deep join plans:

1. `JoinOptEnvBushy` (in file `join_optimization/envs/join_opt_env_bushy.py`),
2. `JoinOptEnvLeft` (in file `join_optimization/envs/join_opt_env_left.py`).

As mentioned in Table 1, the main difference between these two environments lies in their action space; a bushy plan's actions are pairs of tables, while a left-deep plan's actions are single tables. Since their state representations are nearly identical, both `JoinOptEnvBushy` and `JoinOptEnvLeft` subclass a base class called `JoinOptEnvBase` (in file `join_optimization/envs/join_opt_env_base.py`), which we describe first.

JoinOptEnvBase This base class handles most of the `__init__` initialization work of loading in the database schema, loading in the IR cardinality dataset, as well as constructing the selectivity encoding $v^{\text{Sel}}(q)$ and goal encodings $v^{\text{goal}}(q)$ (defined in Section 3.1) for all the queries q in our dataset. Recall that $v^{\text{Sel}}(q)$ and $v^{\text{goal}}(q)$ are static during the trajectory, so we can pre-compute them when initializing the environment.

`JoinOptEnvBase` also contains a helper function `log_cardinality_to_reward` that converts log IR cardinality at step h, *i.e.*, $\log c_h$, to this step's reward $r_h = \frac{1}{C_{\max}(q)}(C_{\min}(q) - \exp(\min\{\log c_h, \log C_{\max}(q)\}))$, where $C_{\max}(q) = 100 \cdot C^\star(q)$, $C^\star(q)$ is the optimal (minimum-possible) *cumulative* IR cardinality for query q, and $C_{\min}(q) = {}^{C^\star(q)}/_{\text{num tables to join in } q}$. To interpret this expression, note that $\exp(\min\{\log c_h, \log C_{\max}(q)\}) = \min\{c_h, C_{\max}(q)\}$. We perform the clipping since IR cardinalities can get large, especially with Cartesian products enabled; this is also why we perform clipping inside the exp and work in log-space. Next, we can interpret $\sum_h c_h - C_{\min}(q)$ as essentially the regret of this trajectory, as $C_{\min}(q) \cdot H = C^\star(q)$. Finally, the scaling by ${}^1/_{C_{\max}(q)}$ is for normalization. In essence, our reward is the per-step negative regret.

JoinOptEnvLeft and JoinOptEnvBushy Each class has three main jobs: 1) maintaining the left-deep join tree, 2) a function to compute the partial plan encoding, 3) a function for computing the valid action masks. As for (1), since left-deep and bushy trees have different structures, we maintain them in different ways, though they both use the `TreeNode` data structure to do so. For (2), the partial plan encoding can be computed by examining the `TreeNode` so far, and only retaining the useful information. Finally, since the action spaces are different, each class has different functions for the valid action mask (3). It is worth highlighting that each class has two functions for computing the valid action mask: `self.valid_action_mask()` is used when Cartesian products are allowed, and `self.valid_action_mask_with_heurstic()` is used otherwise.

G Mechanism for generating queries

We use the 33 predefined query templates of the Join Order Benchmark (JOB) (Leis et al., 2015) and introduce variations in unary predicates, *i.e.*, filter statements, to generate new queries. To randomly generate realistic unary predicates, we begin by conducting a manual examination of all columns within each table to select a subset of columns that are typically used in real user queries. The columns we identified were `aka_name(name)`, `aka_title(title)`, `char_name(name)`, `comp_cast_type(kind)`, `company_name(name, country_code)`, `company_type(kind)`, `info_type(info)`, `keyword(keyword)`, `kind_type(kind)`, `link_type(link)`, `movie_companies(note)`, `movie_info(info)`, `movie_info_idx(info)`, `name(name)`, `person_info(note)`, `role_type(role)`, `title(title, production_year)`. To simulate searches by real IMDb users, we compiled the top 100 most common values for each column using ChatGPT. If any column has less than 100 unique values, we do not need to use ChatGPT and simply used all the possible values.

For example, consider the SQL template `q1`, reproduced below.

```
SELECT MIN(mc.note) AS production_note,
       MIN(t.title) AS movie_title,
       MIN(t.production_year) AS movie_year
FROM company_type AS ct,
     info_type AS it,
     movie_companies AS mc,
     movie_info_idx AS mi_idx,
     title AS t
WHERE ct.id = mc.company_type_id
  AND t.id = mc.movie_id
  AND t.id = mi_idx.movie_id
  AND mc.movie_id = mi_idx.movie_id
  AND it.id = mi_idx.info_type_id
```

We consider unary predicates from those candidate columns `company_type(kind)`, `info_type(info)`, `movie_companies(note)`, `movie_info_idx(info)`, `title(title, production_year)`. For each candidate column, we flip a coin and decide to add a unary predicate with probability 50%. Suppose that the coin flips for each column were respectively $1, 0, 0, 0$, so we only choose the `company_type(kind)` column to create a unary predicate. Subsequently, we pick random number $n \sim \text{Unif}(\{1, 2, 3, 4, 5\})$ and take n random elements from the 'top-100 list' described above. Suppose that $n = 2$ and we randomly sampled 'production companies' and 'special effects companies' from the 'top-100 list' for the `company_type(kind)` column. This process leads us to the resulting query `q1_0`.

```
SELECT MIN(mc.note) AS production_note,
    MIN(t.title) AS movie_title,
    MIN(t.production_year) AS movie_year
FROM company_type AS ct,
    info_type AS it,
    movie_companies AS mc,
    movie_info_idx AS mi_idx,
    title AS t
WHERE ct.id = mc.company_type_id
    AND t.id = mc.movie_id
    AND t.id = mi_idx.movie_id
    AND mc.movie_id = mi_idx.movie_id
    AND it.id = mi_idx.info_type_id
    AND ct.kind in ('production companies',
        'special effects companies')
```

Note the key addition is the last filter statement, `AND ct.kind in ('production companies', 'special effects companies')`. We repeat this procedure 99 more times to produce `q1_0`, ..., `q1_99`. We repeat the above for the 32 remaining templates `q2`, ..., `q33`, which yields the $100 \times 33 = 3300$ random queries that make up our new dataset.

H Limitations

Multiple base tables in a query. Our current solution is to introduce duplicate tables and treat tables from the same basetables differently. Given query templates, our encoding has n positions for a basetable, where n is the maximum number of times this basetable appears among all query templates. We assume that query templates are fixed. We acknowledge that this solution may not be elegant and can be improved in future work.

Dynamic workload. In this benchmark, we assume the RL agent is trained and evaluated on the same database, *i.e.*, we assume the DB content is kept static as in prior works Yang et al. (2022). However, in real applications, the database may dynamically change over time. It is possible to add more queries and databases to JoinGym by simply running our data collection script to collect more cardinality data.

I Additional Results for Online RL

The following tables show the mean, $p90$, $p95$, $p99$ results with standard error confidence intervals computed over 10 seeds.

Mean			DQN		DDQN		TD3		SAC		PPO
			RB	PER	RB	PER	RB	PER	RB	PER	
disable CP	bushy	trn	8.9e+06 (8.8e+06)	2.7e+04 (2.6e+04)	3.6e+06 (3.6e+06)	4.1e+04 (3.7e+04)	**1.9e+04 (1.1e+04)**	1e+05 (5.7e+04)	8e+04 (3.2e+04)	4.9e+04 (1.3e+04)	1.8e+06 (1.7e+06)
		val	3.4e+04 (1e+04)	1.8e+04 (2.5e+03)	1.9e+04 (3.3e+03)	2.4e+04 (2.8e+03)	1.9e+04 (3.1e+03)	**1.7e+04 (3.7e+03)**	4.1e+04 (4.1e+03)	3e+04 (7.5e+03)	2.8e+04 (2.5e+03)
		tst	2.6e+05 (1.4e+05)	1.4e+05 (3.3e+04)	4.1e+05 (1.4e+05)	8.3e+04 (2.5e+04)	1.5e+05 (9.2e+04)	**3.4e+04 (6e+03)**	3.8e+04 (1e+04)	3.5e+04 (9.8e+03)	1.3e+05 (4.6e+04)
	left	trn	**4e+03 (7.7e+02)**	1.3e+04 (6.9e+03)	2.2e+04 (1.1e+04)	8.3e+03 (5.8e+03)	1.3e+06 (1.3e+06)	4.4e+03 (2.7e+02)	4e+06 (1.7e+06)	2.8e+05 (2.7e+05)	2.1e+04 (1e+04)
		val	1.5e+04 (1.5e+03)	1.2e+04 (1.3e+03)	1.4e+04 (2.6e+03)	**9.2e+03 (1.2e+03)**	1.6e+04 (3.7e+03)	1.3e+04 (1.4e+03)	2e+04 (2.4e+03)	1.2e+04 (1.4e+03)	1.1e+04 (1e+03)
		tst	2.9e+05 (2.3e+05)	1.7e+05 (8.8e+04)	1.1e+05 (6.2e+04)	4.5e+05 (2e+05)	1.9e+04 (3.7e+03)	**1.8e+04 (4e+03)**	4.7e+05 (1.9e+05)	2.5e+05 (1.1e+05)	4.6e+04 (2.6e+04)
enable CP	bushy	trn	2e+45 (2e+45)	1.5e+50 (1.5e+50)	3e+37 (2.9e+37)	5.8e+33 (5.7e+33)	7.8e+26 (7.8e+26)	1.1e+24 (1.1e+24)	1.7e+47 (1.7e+47)	2.1e+53 (2.1e+53)	**1.1e+15 (1.1e+15)**
		val	1.4e+30 (1.4e+30)	1.6e+29 (1.6e+29)	8.1e+25 (5.4e+25)	5.3e+21 (5.3e+21)	2.1e+05 (6.8e+04)	**1.8e+05 (4.2e+04)**	1.6e+42 (8.8e+41)	3.5e+42 (2.2e+42)	3.8e+05 (1.1e+05)
		tst	2.4e+46 (2.4e+46)	1.7e+45 (1.7e+45)	2.2e+41 (2.2e+41)	8.5e+30 (7.5e+30)	3.2e+25 (3.2e+25)	**3.6e+19 (3.6e+19)**	4e+49 (4e+49)	4.1e+51 (3.4e+51)	1.3e+28 (1.3e+28)
	left	trn	3.3e+24 (3.3e+24)	9.3e+08 (9.3e+08)	1.9e+08 (1.4e+08)	1.5e+14 (1.5e+14)	1.5e+14 (1.5e+14)	1.6e+05 (7.1e+04)	**1.3e+04 (7.3e+03)**	1.8e+04 (1.1e+04)	7.9e+10 (5.6e+10)
		val	2.4e+04 (5e+03)	1.8e+04 (2.9e+03)	2.6e+04 (3.1e+03)	2.2e+04 (2.1e+03)	1.6e+05 (5.7e+04)	2.3e+04 (4.1e+03)	**1.2e+04 (1.3e+03)**	**1.2e+04 (8e+02)**	4.8e+04 (1.2e+04)
		tst	7.7e+14 (5.5e+14)	1.9e+11 (1.5e+11)	8.1e+17 (7.5e+17)	1.4e+27 (1.4e+27)	2.1e+10 (2.1e+10)	1.7e+25 (1.7e+25)	**5.6e+05 (1.9e+05)**	1.1e+06 (7.5e+05)	5.2e+23 (5.2e+23)

90% **Quantile**			DQN		DDQN		TD3		SAC		PPO
			RB	PER	RB	PER	RB	PER	RB	PER	
disable CP	bushy	trn	7.3 (2.3)	**4.4 (0.11)**	5.2 (0.17)	5.6 (0.52)	5.3 (0.32)	7.3 (0.74)	13 (1.3)	9.5 (0.73)	6 (0.32)
		val	16 (2.7)	**13 (0.71)**	14 (0.85)	15 (0.75)	15 (1.5)	18 (1.9)	18 (1.8)	**13 (0.8)**	23 (2)
		tst	46 (15)	**25 (2.5)**	30 (5.4)	30 (2.9)	26 (3.6)	40 (7.7)	55 (11)	33 (3.1)	42 (4)
	left	trn	5.5 (0.51)	5.6 (0.37)	7 (1.1)	6.5 (1.4)	6.9 (0.98)	**5.2 (0.43)**	11 (0.99)	8.6 (0.69)	**5.2 (0.92)**
		val	12 (0.73)	15 (1.7)	13 (0.79)	14 (2)	13 (1.7)	**9.5 (0.41)**	20 (1.5)	14 (1.1)	11 (1.5)
		tst	28 (3.1)	30 (2.6)	34 (3.7)	34 (3.2)	22 (3.1)	20 (2)	39 (3.1)	32 (4)	**19 (4.6)**
enable CP	bushy	trn	6.4e+04 (5.8e+04)	2.4e+05 (2.4e+05)	4.6e+04 (4.5e+04)	3.2e+04 (2.9e+04)	1.8e+02 (1.5e+02)	42 (21)	7.7e+18 (7.6e+18)	3e+14 (3e+14)	**35 (5.5)**
		val	2e+05 (1.9e+05)	1.1e+06 (1.1e+06)	5.8e+04 (4.2e+04)	6e+04 (4.7e+04)	3.1e+02 (2.3e+02)	1.4e+02 (78)	4e+18 (3.9e+18)	2.8e+14 (2.8e+14)	**1e+02 (26)**
		tst	1.6e+05 (6.9e+04)	1.2e+05 (1.1e+05)	2.1e+05 (1.5e+05)	6.9e+04 (4.4e+04)	2e+03 (1.8e+03)	4.9e+02 (2.6e+02)	2.2e+17 (2.2e+17)	2.1e+17 (2.1e+17)	**2.8e+02 (43)**
	left	trn	17 (8.5)	**6.3 (0.38)**	17 (6.2)	9.9 (3.2)	3.6e+02 (2.9e+02)	17 (1.4)	7.7 (0.32)	6.8 (0.31)	9.9 (0.62)
		val	25 (8.9)	15 (2.2)	27 (7.7)	22 (4.2)	2e+02 (99)	24 (2.1)	13 (0.64)	**12 (0.57)**	27 (2.4)
		tst	64 (21)	36 (2.7)	66 (16)	59 (16)	1.7e+03 (1.1e+03)	1e+02 (8.7)	**28 (2.5)**	30 (3.3)	92 (14)

Table 6: The results of Table 3 with standard error computed over 10 seeds.

95% **Quantile**			DQN		DDQN		TD3		SAC		PPO
			RB	PER	RB	PER	RB	PER	RB	PER	
disable CP	bushy	trn	80 (67)	**9.4 (0.29)**	13 (0.68)	14 (1.9)	16 (1.3)	25 (3.7)	85 (14)	48 (7.9)	37 (3.5)
		val	2.6e+02 (1.1e+02)	2.2e+02 (39)	1.8e+02 (27)	2e+02 (19)	2.5e+02 (43)	**1.5e+02 (20)**	2.3e+02 (25)	1.8e+02 (23)	4.1e+02 (63)
		tst	1.5e+03 (4e+02)	1.1e+03 (2.5e+02)	2e+03 (6.5e+02)	1.3e+03 (3.1e+02)	**1e+03 (2.4e+02)**	1.3e+03 (2.5e+02)	2.4e+03 (3.5e+02)	1.2e+03 (2.9e+02)	3.7e+03 (1.3e+03)
	left	trn	19 (3.8)	**18 (2.7)**	28 (7.8)	24 (8.4)	32 (8)	19 (2.3)	47 (7.1)	34 (5.3)	38 (14)
		val	1.3e+02 (17)	1.6e+02 (21)	1.2e+02 (11)	1.5e+02 (39)	1.1e+02 (21)	**67 (7.8)**	1.5e+02 (30)	79 (8.6)	1.2e+02 (34)
		tst	7e+02 (1.8e+02)	6.2e+02 (79)	8.3e+02 (1.4e+02)	1.3e+03 (3.1e+02)	**5.2e+02 (72)**	5.4e+02 (1.1e+02)	1.7e+03 (2.6e+02)	1e+03 (1.8e+02)	5.5e+02 (1.3e+02)
enable CP	bushy	trn	3.7e+08 (3.4e+08)	3.8e+09 (3.8e+09)	2.8e+07 (2.7e+07)	2.9e+06 (1.9e+06)	1.2e+04 (1.2e+04)	4.2e+03 (3.8e+03)	4.7e+22 (3.2e+22)	7.6e+20 (7.3e+20)	**1.4e+03 (4.3e+02)**
		val	8.9e+07 (5.3e+07)	5.8e+11 (5.8e+11)	6.6e+08 (6.5e+08)	2.6e+06 (1.2e+06)	3e+04 (2.7e+04)	3.6e+04 (3.3e+04)	2.5e+24 (2.5e+24)	5.1e+21 (3.1e+21)	**7.1e+03 (1.9e+03)**
		tst	1.9e+09 (1.8e+09)	1.3e+08 (1.3e+08)	1.3e+08 (1.1e+08)	7.9e+06 (4.6e+06)	6.1e+04 (4.7e+04)	3e+04 (1.6e+04)	3.6e+22 (3.5e+22)	1.8e+24 (1.6e+24)	**2.1e+04 (4e+03)**
	left	trn	1.4e+02 (1.1e+02)	**22 (2.2)**	1.1e+02 (52)	54 (30)	5.9e+03 (4.3e+03)	83 (17)	28 (1.4)	23 (1.9)	1.6e+02 (30)
		val	2e+02 (56)	1.9e+02 (45)	3.9e+02 (1.5e+02)	2.5e+02 (68)	1.1e+04 (7.5e+03)	2.7e+02 (20)	**76 (7)**	82 (8.8)	5.9e+02 (66)
		tst	2.4e+03 (7.7e+02)	1.6e+03 (3.9e+02)	3.8e+03 (1.3e+03)	1.8e+03 (3.9e+02)	4.7e+04 (2.2e+04)	5.8e+03 (1.3e+03)	**1.1e+03 (2.4e+02)**	1.3e+03 (2.5e+02)	7.5e+03 (9.5e+02)

99% **Quantile**			DQN		DDQN		TD3		SAC		PPO
			RB	PER	RB	PER	RB	PER	RB	PER	
disable CP	bushy	trn	1.8e+04 (1.6e+04)	**55 (3.8)**	7e+02 (4.6e+02)	4.2e+02 (2.2e+02)	2.4e+03 (1.4e+03)	3e+03 (1.5e+03)	3.9e+04 (9e+03)	2.2e+04 (1e+04)	4e+04 (1e+04)
		val	5.1e+05 (2e+05)	3.1e+05 (7.1e+04)	2.7e+05 (7.2e+04)	3.3e+05 (9e+04)	2.7e+05 (7.3e+04)	**1.9e+05 (5.2e+04)**	4.3e+05 (5.2e+04)	4.7e+05 (6e+04)	5.7e+05 (6.5e+04)
		tst	**3.3e+05 (3.3e+04)**	7.5e+05 (4.1e+05)	6e+05 (1.6e+05)	5.4e+05 (1.3e+05)	4.5e+05 (9e+04)	4.6e+05 (3.3e+04)	6.2e+05 (1.1e+05)	4e+05 (8.6e+04)	6e+05 (5e+04)
	left	trn	2e+03 (9.1e+02)	**1.5e+03 (8.3e+02)**	7e+03 (3.6e+03)	3.3e+03 (2.2e+03)	9.6e+03 (4.1e+03)	2e+03 (7.6e+02)	9.9e+03 (3.7e+03)	4.6e+03 (2.3e+03)	2.8e+04 (1.6e+04)
		val	3e+05 (3.9e+04)	2.5e+05 (2.8e+04)	2.2e+05 (4.4e+04)	1.4e+05 (2.4e+04)	2e+05 (3.8e+04)	1.5e+05 (2.6e+04)	2.4e+05 (3.1e+04)	**1.2e+05 (2.1e+04)**	2.1e+05 (3.3e+04)
		tst	3.1e+05 (3.2e+04)	4.6e+05 (6.5e+04)	4.5e+05 (5.9e+04)	5.7e+05 (1.1e+05)	**3e+05 (3.2e+04)**	3.3e+05 (2.4e+04)	4.3e+05 (4.3e+04)	3.7e+05 (3.3e+04)	3.1e+05 (4e+04)
enable CP	bushy	trn	4.3e+26 (3.7e+26)	5.7e+23 (5.7e+23)	1.3e+17 (1e+17)	7.9e+14 (7.7e+14)	**4.2e+05 (1.6e+05)**	2.5e+06 (1.6e+06)	1.8e+37 (1.1e+37)	3.6e+42 (3.6e+42)	1.6e+06 (4.2e+05)
		val	1.2e+22 (1.2e+22)	5.7e+24 (5.7e+24)	2.8e+23 (2.1e+23)	1.7e+14 (9.8e+13)	**2e+06 (4.4e+05)**	3.1e+06 (1.8e+06)	3.7e+38 (2.9e+38)	4.2e+40 (2.8e+40)	5.6e+06 (1.5e+06)
		tst	2.5e+29 (2.5e+29)	5.7e+25 (5.7e+25)	1e+23 (1e+23)	6.5e+18 (6.4e+18)	**2.6e+06 (5.9e+05)**	4.1e+06 (1.6e+06)	9.2e+39 (9.2e+39)	1.3e+41 (1.3e+41)	7.6e+06 (2.3e+06)
	left	trn	7.1e+04 (5.5e+04)	3.5e+03 (1.2e+03)	3.7e+04 (1.9e+04)	1.9e+04 (1.3e+04)	1e+06 (5.1e+05)	7.6e+04 (2.3e+04)	5.1e+03 (1.8e+03)	**1.9e+03 (6.9e+02)**	4.3e+05 (1.1e+05)
		val	2.3e+05 (7.9e+04)	2.4e+05 (4.6e+04)	3.1e+05 (5.6e+04)	3.6e+05 (8.6e+04)	1.6e+06 (3.1e+05)	2.6e+05 (4.1e+04)	**1.3e+05 (4.5e+04)**	1.4e+05 (2.5e+04)	6.3e+05 (1.6e+05)
		tst	1.8e+06 (7.1e+05)	7.6e+05 (1.6e+05)	7.9e+05 (1e+05)	1.4e+06 (3.6e+05)	4.8e+06 (7.6e+05)	7.6e+06 (6.1e+06)	**3.8e+05 (4.1e+04)**	4.4e+05 (3.6e+04)	3.3e+06 (1.9e+06)

Table 7: The 95% and 99% quantile of CMMs with standard error computed over 10 seeds.

I.1 Additional Learning Curves

In this section, we plot the learning curves of the best performing algorithm in each of the four possible settings in $\{\text{disable CP}, \text{enable CP}\} \times \{\text{left}, \text{bushy}\}$. Performance is measured by mean CCM (from Table 3) and we plot the average performance over 10 runs (shaded region is stderr over said runs). For each setting, we also show the CCM distribution over the training, validation and testing query sets in a complementary CDF (CCDF) plot. The CCDF shows that these distributions are long-tailed.

I.1.1 Bushy plans with disable CPs heuristic

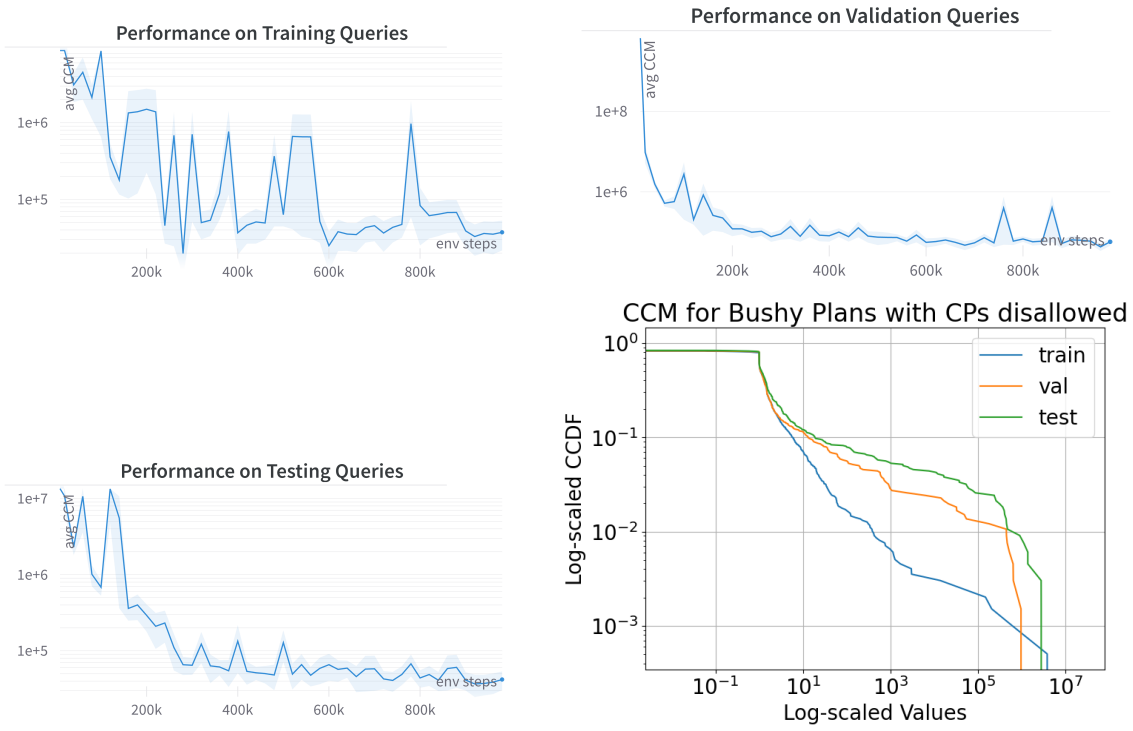

Figure 5: The best algorithm (in terms of CCM mean) for bushy plans with disable CP heuristic is TD3 with prioritized replay with policy learning rate 0.0003 and critic learning rate 0.0001. Shaded region is stderr over 10 runs.

I.1.2 Left-deep plans with disable CP heuristic

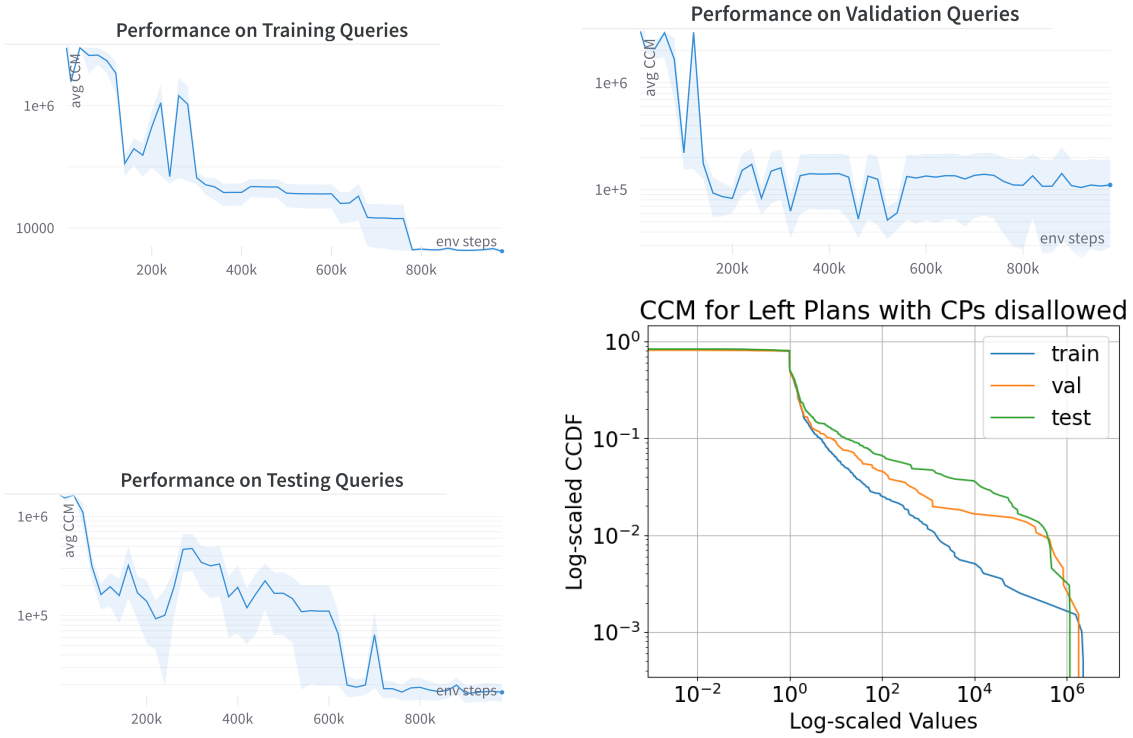

Figure 6: The best algorithm (in terms of CCM mean) for left-deep plans with disable CP heuristic is TD3 with prioritized replay with policy learning rate 0.0001 and critic learning rate 0.0003. Shaded region is stderr over 10 runs.

I.1.3 Bushy plans with CPs

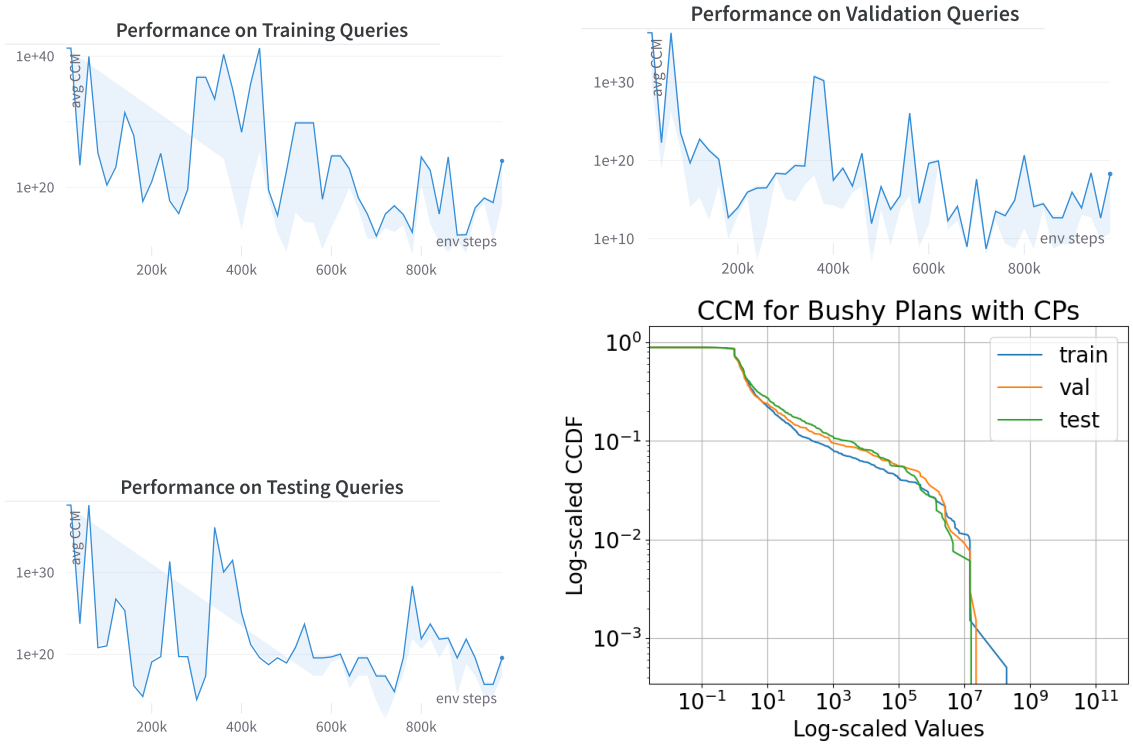

Figure 7: The best algorithm (in terms of CCM mean) for bushy plans with CPs enabled is TD3 with prioritized replay with policy learning rate 0.0003 and critic learning rate 0.0003. Shaded region is stderr over 10 runs.

I.1.4 Left-deep plans with CPs

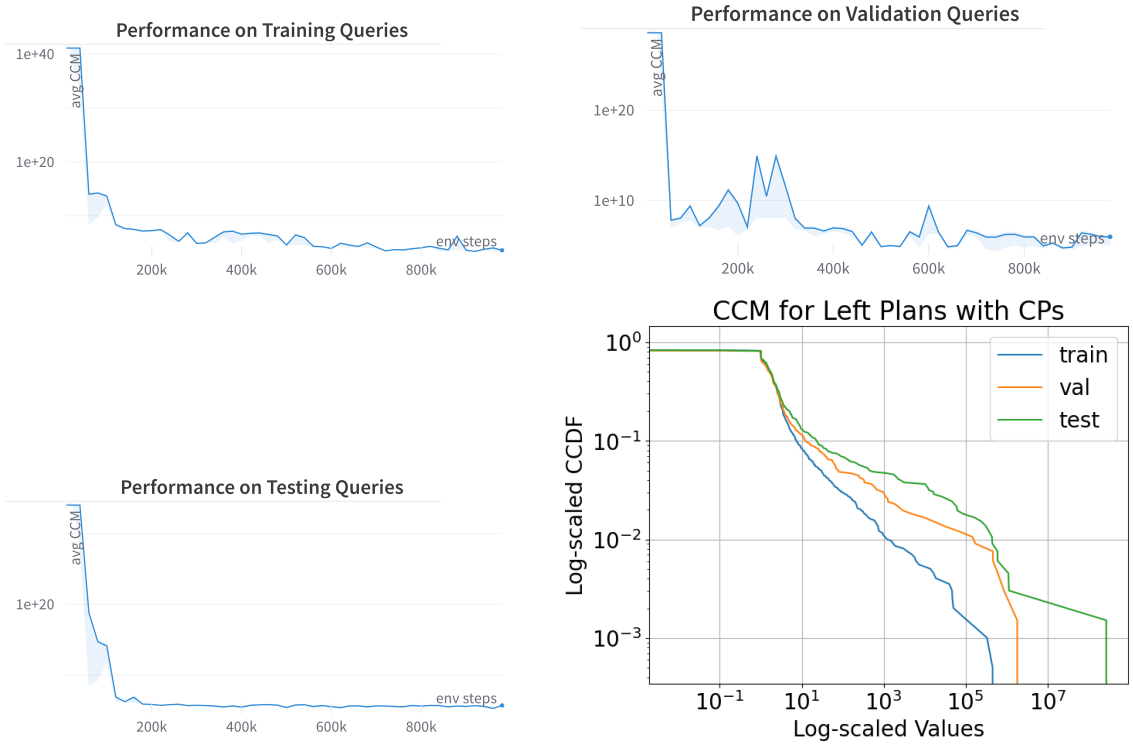

Figure 8: The best algorithm (in terms of CCM mean) for left-deep plans with CPs enabled is TD3 with policy learning rate 0.0001 and critic learning rate 0.0003. Shaded region is stderr over 10 runs.

I.1.5 Heuristic Method

We also plot results for a heuristic `ikkbzbushy` which builds join selectivity estimates based on the training data and uses dynamic programming (DP) to compute the best bushy plan according to the estimated IR cardinalities (based on the join selectivity estimates). Similar with RL agents which learns in training queries, we build selectivity estimation of join predicates using training queries. We follow the most classic approach (Ramakrishnan & Gehrke, 2003) for estimating selectivities of each join pair $R \bowtie_P S$ using $\text{Sel}_i(R \bowtie_P S) = \frac{|R \bowtie_P S|}{|R||S|}$ and take the average among all queries as the final selectivities. And in the validate and test, we estimate the IR size using $\text{Sel}_i(R \bowtie_P S)|R||S|$. `ikkbzbushy` can guarantee that the final plan has the smallest estimation cost.

So amongst all heuristics that use the same estimated IR cardinalities, this approach is the best possible heuristic since it uses the plan with the smallest estimated cost. However, since the selectivity estimation step is biased, the final performance is very poor, and worse than the RL-based approaches. It's worth mentioning that DP-based approaches can take hours on med-large size queries since the problem space grows exponentially. In contrast, RL methods are much faster to run.

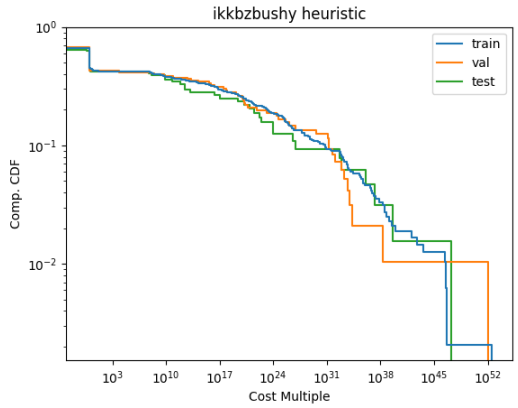

Figure 9: Distribution of cost multiples for the heuristic. The mean performance is train $7.8e+49$, val $1.1e+50$ and test $2.5e+45$. Notice that these are many orders of magnitude worse than the RL policy's performance.

J Benchmarking Offline RL

We also benchmark offline RL algorithms on optimizing left-deep plans with CPs disallowed. In this section, we focus on the 113 queries from the JOB (Leis et al., 2015) which is a smaller and easier setting than our main dataset of 3300 queries. We focus on the JOB for offline RL because Leis et al. (2015) provided behavior policy traces for all 113 queries, based on popular search heuristics (described below).

JOB Data			DQN		DDQN		BC		BCQ		CQL	
			Median	STD	Median	STD	Median	STD	Median	STD	Median	STD
disable CP	left	trn	**3.22**	1.9e6	1471	3.4e10	1.6e5	7.6e+11	3.1e3	2.7e13	79	4.1e12
		tst	**3.19**	1.9e6	1470	3.3e10	8.0e4	7.3e12	9908	2.6e13	76	3.9e12
		val	**3.21**	2.0e6	1.0e3	3.4e10	6.4e5	7.5e12	1.5e5	2.9e13	81	4.1e12

Table 8: CCM (lower is better) averaged over the training (trn), validation (val) or testing (tst) query sets.

Experimental Setup Our offline dataset is comprised of trajectories from the following behavior policies provided by the JOB (Leis et al., 2015): adaptive, dphyp, genetic, goo, goodp, goodp2, gooikkbz, ikkbz, ikkbzbushy, minsel, quickpick, simplification (Neumann & Radke, 2018). We highlight that these behavior policies are *search* heuristics, which operate given a cost model, *e.g.*, estimated IR cardinalities, to plan over. To generate behavior trajectories, we provided these heuristics access to the ground-truth IR cardinalities. Alternatively, one could take traces from existing DBMS such as Postgres. For each heuristic, we collected 1000 trajectories across different queries. We partition the dataset for training, evaluation and testing similarly as in our online experiments.

Discussions Table 8 summarizes our offline results. We find that DQN has the best performance in terms of median and the validation/testing results are even better than online. CQL also obtains reasonable performance, but all other methods seem to have relatively poor median performance even on the training set. It's worth noting that all methods seem to have a heavy tail performance distribution (over queries), as shown by the large standard deviations. In later sections of the appendix, we see this is the case for online RL as well. This heavy-tail distribution of returns motivates applying risk-sensitive RL methods to JoinGym for future work.

We also tested on some other offline algorithms, such as SAC, and it is hard to converge hence we didn't report the results. We observe that the TD error is increasing, although the Q value functions, actors and critics are learning. Making too many TD updates to the Q-function in offline deep RL is known to sometimes lead to performance degradation and unlearning, we can use regularization to address the issue (Kumar et al., 2021).

J.1 Hyperparameters for Offline RL Algorithms

We performed hyperparameter search with grid search and Bayesian optimization. The final parameters we used for evaluation is shown below in Tables 10-13.

J.1.1 Batch-Constrained Q-learning

Table 9: Hyperparameter of Batch-Constrained Q-learning algorithm (BCQ).

Hyperparameter	Value
Learning rate	6.25×10^{-5}
Optimizer	Adam ($\beta = (0.95, 0.999)$)
Batch size	32
Number of critics	6
Discount factor	0.99
Target network synchronization coefficiency	0.005
Action flexibility	0.3
Gamma	0.99

J.1.2 Behavior Cloning

Table 10: Hyperparameter of Behavior Cloning (BC).

Hyperparameter	Value
Learning rate	0.001
Optimizer	Adam ($\beta = (0.9, 0.999)$)
Batch size	100
Beta	0.5

J.1.3 Conservative Q-Learning

Table 11: Hyperparameter of Conservative Q-Learning (CQL).

Hyperparameter	Value
Actor learning rate	3×10^{-4}
Critic learning rate	3×10^{-4}
Learning rate for temperature parameter of SAC	1×10^{-4}
Learning rate for alpha	1×10^{-4}
Batch size	256
N-step TD calculation	1
Discount factor	0.99
Target network synchronization coefficiency	0.005
The number of Q functions for ensemble	2
Initial temperature value	1.0
Initial alpha value	1.0
Threshold value	10.0
Constant weight to scale conservative loss	5.0
The number of sampled actions to compute	10

J.1.4 DQN

Table 12: Hyperparameter of DQN.

Hyperparameter	Value
Learning rate	6.25e-4
Batch size	32
target_update_interval	8000

J.1.5 Double DQN

Table 13: Hyperparameter of DDQN.

Hyperparameter	Value
Learning rate	6.25e-4
Batch size	32
target_update_interval	8000