# Efficient Join Order Selection Learning with Graph-based Representation

Jin Chen*
University of Electronic Science and Technology of China
chenjin@std.uestc.edu.cn

Guanyu Ye*
University of Electronic Science and Technology of China
ygy@std.uestc.edu.cn

Yan Zhao
Aalborg University
yanz@cs.aau.dk

Shuncheng Liu
University of Electronic Science and Technology of China
liushuncheng@std.uestc.edu.cn

Liwei Deng
University of Electronic Science and Technology of China
deng_liwei@std.uestc.edu.cn

Xu Chen
University of Electronic Science and Technology of China
xuchen@std.uestc.edu.cn

Rui Zhou
Huawei Technologies Co., Ltd.
zhourui24@huawei.com

Kai Zheng†
University of Electronic Science and Technology of China
zhengkai@uestc.edu.cn

## ABSTRACT

Join order selection plays an important role in DBMS query optimizers. The problem aims to find the optimal join order with the minimum cost, and usually becomes an NP-hard problem due to the exponentially increasing search space. Recent advanced studies attempt to use deep reinforcement learning (DRL) to generate better join plans than the ones provided by conventional query optimizers. However, DRL-based methods require time-consuming training, which is not suitable for online applications that need frequent periodic re-training. In this paper, we propose a novel framework, namely efficient Join Order selection learninG with Graph-basEd Representation (JOGGER). We firstly construct a *schema graph* based on the primary-foreign key relationships, from which table representations are well learned to capture the correlations between tables. The second component is the state representation, where a graph convolutional network is utilized to encode the query graph and a tailored-tree-based attention module is designed to encode the join plan. To speedup the convergence of DRL training process, we exploit the idea of curriculum learning, in which queries are incrementally added into the training set according to the level of difficulties. We conduct extensive experiments on JOB and TPC-H datasets, which demonstrate the effectiveness and efficiency of the proposed solutions.

*Equal contribution.
†Corresponding author

## CCS CONCEPTS

• **Information systems → Data management systems**; **Database management system engines**.

## KEYWORDS

Database; Join Order; Graph Representation

**ACM Reference Format:**
Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-based Representation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), August 14–18, 2022, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3534678.3539303

## 1 INTRODUCTION

Join order selection, which aims to find the optimal join order, plays a critical role in DBMS query optimizers. Figure 1 shows an example of join order selection for an SQL query from a database with four tables, i.e., $T_1, T_2, T_3, T_4$. There are many possible sequential join orders to execute the query, e.g., $(T_1 \bowtie T_2) \bowtie (T_3 \bowtie T_4)$, $((T_1 \bowtie T_2) \bowtie T_3) \bowtie T_4$, $((T_1 \bowtie T_3) \bowtie T_2) \bowtie T_4$, etc., from which join order selection aims to find one with the lowest expected cost. With the increase of the participating tables, the search space grows exponentially, rendering the search for the optimal join order intractable eventually. From the theoretical aspect, the join order selection problem can essentially be reduced from NPC problems, so practical solutions often resort to sub-optimal results such as dynamic programming [7], greedy search [4], and heuristic search [3, 4]. These methods either search for the sub-optimal plans or prune the search space depending on the estimated cardinality or cost. In spite of the widespread adoption, they are prone to falling into local optimum, resulting in unsatisfactory query plans with poor performance.

Deep reinforcement learning (DRL) has attained growing interest in the join order selection problem [11, 17, 32, 33], as it has shown superior performance over traditional methods and even the native DBMS join plans. The DRL-based methods search for better

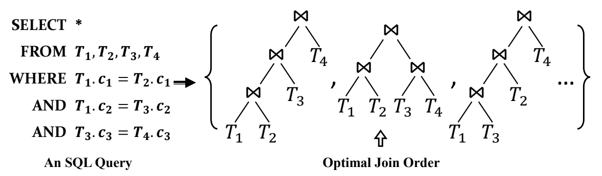

**Figure 1: Example of Join Order Selection for An SQL Query**

query plans relying on previous tried episodes and the long-term benefit rather than the immediate reward of the current sub-plan. By incorporating deep learning, DRL allows agents to make decisions without manual engineering of the state space. However, one major barrier that prevents the DRL-based methods from being practically adopted is the inefficiency of the model learning, due to the following two reasons.

- **"Deep"**. To enhance the expressiveness of state representations, deep neural networks are introduced to encode the numerous join plans and queries. However, the utilization of deep neural networks introduces a large number of parameters for complex computations, which is especially much more than the number of training queries in public benchmark dataset. Such large-scale networks are also prone to overfitting during the training process, which would lead to poor performance on test data.
- **"Reinforcement Learning (RL)"**. During the learning process of DRL, the experience replay mechanism is employed to update model parameters, where uniform sampling is adopted to sample training data. A common strategy, prioritized experience replay [25], is utilized for selecting more challenging samples. Despite the fact that the prioritized experience replay mitigates the issue caused by breaking the links of consecutive samples, it is still hard to maintain a stable training process. The reason lies in that deep neural networks are very sensitive to parameter changes. The training queries and their sub-join plans are dislocated in the experience memory with varying rewards, so the model parameters can change significantly during the training process. It has a negative impact on the convergence process, which often leads to poor join order selections.

To tackle the inefficiency issue of DRL-based methods, we propose a novel framework, efficient Join Order selection LearninG with Graph-basEd Representation (JOGGER). An intuitive idea to address the inefficiency is to reduce the parameters of the deep neural networks without harming the expressiveness of state representations, which is a key component of DRL. To this end, our framework contains two graph-based components to learn the informative representations with fewer parameters. The first component is table representation learning, where the correlations between tables would be well captured. We construct a *schema graph* based on the primary-foreign key relationships from the native database and then utilize the DeepWalk [22] algorithm to learn table representations. Compared with the simple concatenation of column embeddings in previous studies, these learned table representations capture the correlations between tables and include more semantic information of joins. The second component for state representations utilizes a graph convolutional network to encode queries and contains a tailored-tree-based attention module to represent the join plans. Both components have relatively small-scale networks with fewer parameters, of which the training is highly efficient. In addition, with the aid of the two components, we vectorize the state embedding with rich information of correlations between tables.

In order to further speed up the convergence of DRL, we integrate curriculum learning into the training process, where the training data gradually shifts from easy to more difficult samples. The difficulty level of the curricula is determined by the number of participating tables in the query, since a larger number of tables indicates a larger search space.

Our main contributions are summarized as follows:

- We construct a schema graph to extract the correlations between tables from the native primary-foreign key information and learn the expressive representations of tables, where the global joinable information is encoded and strong connections between tables are captured.
- We design a tailored-tree-based attention method to encode the sequential join structures for (sub) join plans, i.e., forests with binary trees. Compared with Tree-LSTM [32], our method has fewer parameters and further mitigates the overfitting problem.
- We adopt a curriculum learning strategy to link the number of participating tables in the query to the difficulty level of learning these samples. Samples are fed to the model at an ascending level of difficulties to achieve a more stable learning process.
- We conduct extensive experiments using the public JOB and TPC-H datasets, which demonstrate the effectiveness and efficiency of the proposed methods.

## 2 PRELIMINARIES

### 2.1 Preliminary Concepts

Definition 1 (Join Plan). *A join plan (i.e., join order), $p^q$, is a binary tree, where each inner node corresponds to a join predicate of the given query $q$, and each leaf node represents a candidate joinable table. Each query plan $p^q$ comes with a certain cost, denoted by $c(p^q)$, computed by a cost function or a cost model.*

In the rest of the paper, we will use the terms *join plan*, and *join order* interchangeably, whenever the context is clear. We use $P^q$ to denote the set all possible join plans for $q$.

Definition 2 (Join Order Selection). *Given an SQL query $q$, the join order selection problem is to find a query plan $p^q_{opt}$ that achieves the lowest cost, i.e., $\forall p^q_i \in P^q(c(p^q_i) \geq c(p^q_{opt}))$.*

### 2.2 DRL-based Join Order Selection

The deep reinforcement learning (DRL) has been exploited to the join order selection, usually consisting of the following components: - *Agent*: the DBMS optimizer that aims to find the optimal join plan from previous trials, interacting with the environment by taking actions to change states and being rewarded. - *Environment*: the DBMS, e.g., PostgreSQL, which provides the reward after agent taking a certain action. - *State*: join plans with partial tables or all tables, where the immediate states refer to join plans with partial tables in a query and the terminal state is the join plan with all participating tables. - *Action*: joins referring to which two tables can be joined, where the action space changes for different queries and states. - *Reward*: the cost from query planner of a DBMS. The agent (optimizer) attempts to maximize the cumulative reward until the

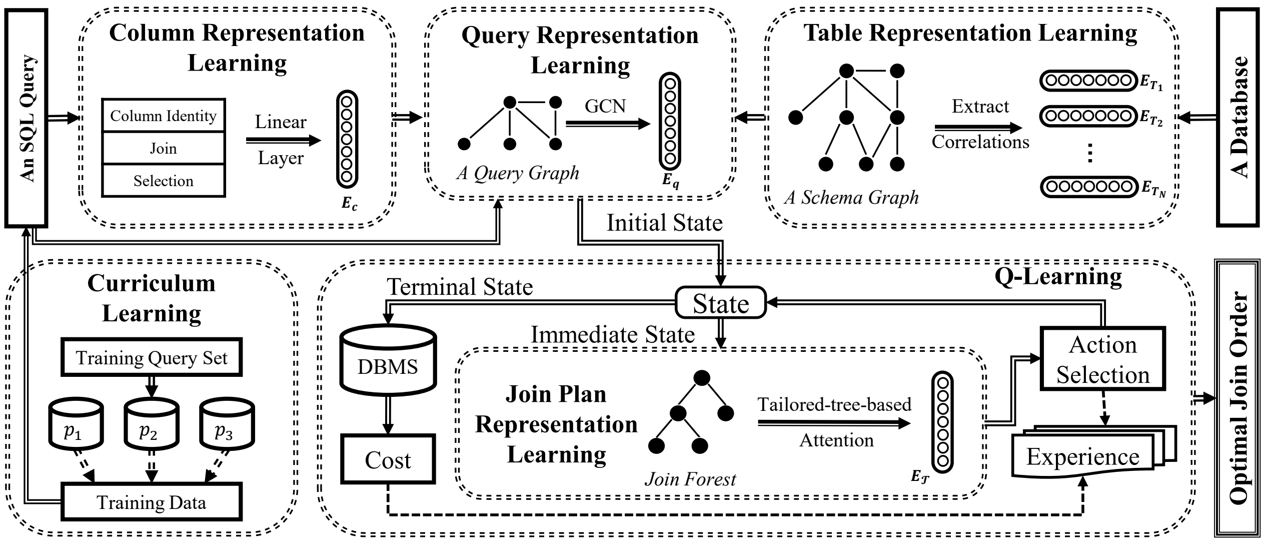

**Figure 2: Framework Overview**

end of the episode by learning from previous trials. In this work, we use the cost from the query planner of a DBMS to denote the reward, which is an estimate of the actual execution time for a join plan and can be efficiently acquired from the DBMS.

It is worth to mention that another important concern related to our framework design is the learning strategy selection in DRL. Two main strategies, value-based (e.g., Q-learning) and policy-based (e.g., policy gradient) methods, have been exploited into the join order selection problem [17, 32]. In particular, Q-learning saves a Q-table, where each state is recorded with its maximum Q-value, while the policy-based method attempts to learn a policy to maximize the expected cumulative reward. Considering that the value-based strategy can scale better in the discrete action space which is the case in our problem, we adopt deep Q-learning strategy here.

## 3 FRAMEWORK AND METHODOLOGY

We propose a framework, namely efficient Join Order selection learninG with Graph-basEd Representation (JOGGER), to find the optimal join plan. We first give an overview of the framework and then provide specifics on each component in the framework.

### 3.1 Framework

The JOGGER framework is comprised of four components: column representation learning, table representation learning, state representation learning, and Q-learning, as shown in Figure 2.

(1) ***Column Representation Learning***. The column representations are incorporated with the representations of column identity and the vectorized predicates (i.e., join and selection).
(2) ***Table Representation Learning***. Considering that it is helpful to estimate the cost (i.e., the reward in DRL-based join order selection) accurately by capturing the correlations between tables, we construct a schema graph based on primary-foreign key relationships to capture the correlations between tables, from which table representations are learned.
(3) ***State Representation Learning***. The state representation is modeled by the concatenation of the query representation and the current join plan representation, where the query representation is learned by a Graph Convolutional Network based on a query graph and the current join plan representation is learned by a tailored-tree-based attention model based on a join forest.
(4) ***Q-learning***. We follow previous strategies [11, 32] that maintain a pool to save the experience and learn the join order selection under a huge search space, where a curriculum-learning-based optimization strategy is proposed to accelerate the training process by ranking the training queries from easy to hard.

### 3.2 Column representation Learning

The key to DRL-based join order selection is the value estimation of the intermediate/terminal state, where a regression model is learned to update model parameters. The representation of a state generally includes the query and join sub-plan representations, which rely on column and table representations. We first introduce the column representation in detail in this section.

Given a query $q$, there are two types of predicates related to the columns: join and selection. The join predicate decides which two columns of the joining tables are connected. The selection predicate decides the selectivity, i.e., how many tuples in a table are selected after the predicates, and then participates in the join. Thus, we consider both join and selection into the column representation.

We concatenate the representations of join and selection predicates as well as the embedding of the column identity (ID) as the final column representation. Specifically, for the representation of the selection predicate, we take the selectivity into account, which represents the percentage of rows having the same value as the indexed column. Since the selectivity is a continuous value, a discretization strategy is adopted here to generate a feature vector. The selectivity is discretized into three partitions with the same width. In this way, each column has a vector $\boldsymbol{z}_c \in \mathbb{R}^4$ to represent the join and selection predicates. The embedding of column identity, $\boldsymbol{e}_c$, is obtained from the embedding matrix $\boldsymbol{M}_E \in \mathbb{R}^{M \times D}$, where $M$ denotes the number of all columns in the database and $D$ denotes the number of dimensions. The whole column representation $\boldsymbol{E}_c$ is

**Algorithm 1:** CTRL

```
Input: A database
Output: Table representations
1 Build a schema graph $\mathcal{S} = (V^T, E^T)$ based on the
    primary-foreign key relationships;
2 repeat
3     Uniformly select a node $T_i$ from $\mathcal{S}$;
4     Perform Random Walks on $T_i$ and collect the sequence $\boldsymbol{L}$;
5 until reach the number of trails;
6 Perform Skip-gram algorithm and use an average pooling
    layer to get the representation $\boldsymbol{E}_T$ of each table node;
```

calculated by two linear layers:

$$\boldsymbol{E}_c = (\boldsymbol{z}_c \cdot \boldsymbol{W}^{L_0} \oplus \boldsymbol{e}_c) \cdot \boldsymbol{W}^{L_1},$$

where $\boldsymbol{W}^{L_0} \in \mathbb{R}^{4 \times D}$ denotes the linear layer to translate the predicate information into a continuous vector, $\boldsymbol{W}^{L_1} \in \mathbb{R}^{2D \times D}$ is another linear layer to integrate the column identity, and $\oplus$ denotes the concatenation of two vectors. Here, the embedding matrix $\boldsymbol{M}_E$, and the linear layers $\boldsymbol{W}^{L_0}$ and $\boldsymbol{W}^{L_1}$ are model parameters to learn.

## 3.3 Table Representation Learning

Although the existing models (e.g., DQ [11] and RTOS [32]) model the representations of tables as those of their inclusive columns, they fail to capture correlations between tables, which imply important semantic information for joins. Inspired by Fauce [15], which captures correlations between tables through exploring the possible links, we design a Correlation-based Table Representation Learning (CTRL) method to extract the correlations between tables with two important steps:

**Schema Graph Construction.** Intuitively, a database schema, which describes both the organization of data and the relationships between tables, contains the primary-foreign key relationships between tables. Thus, we construct a schema graph to model the correlations between tables. Specifically, a schema graph $\mathcal{S} = (V^T, E^T)$ is an undirected graph, where $V^T = \{T_1, T_2, ..., T_N\}$ denotes a set of nodes consisting of $N$ tables, and $E^T$ denotes a set of edges connecting table nodes that have foreign key relationships.

**Table Representation Learning based on Schema Graph.** After constructing the schema graph, we are able to learn the table representations by the DeepWalk [22] algorithm, which utilizes the graph structure to learn the representations of nodes without any labels. Specifically, the DeepWalk algorithm includes the random walk and sequence update procedures. First, DeepWalk randomly samples a node $T_i$ as the start of the walk from the schema graph and randomly chooses a neighbour to move on. The random walk is done until the maximum length of the walk is reached. After many trials, we obtain multiple sequences with table nodes as elements. Second, the node representations are learned according to the sampled random walks. The language model, Skip-Gram [18], is utilized here to learn the node representations since the walks have the same structure as the texts. Based on all of the previous observed nodes in the random walk, Skip-Gram aims to maximize the likelihood of observing the next nodes to learn a preliminary representation $\boldsymbol{E}_T^0$ of each table.

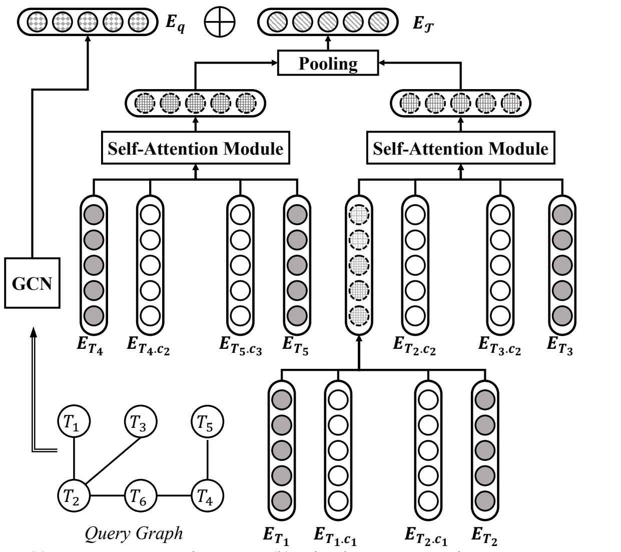
$E_q$
$\oplus$
$E_{\mathcal{T}}$
Pooling
Self-Attention Module
Self-Attention Module
GCN
$E_{T_4}$ $E_{T_4.c_2}$ $E_{T_5.c_3}$ $E_{T_5}$
$E_{T_2.c_2}$ $E_{T_3.c_2}$ $E_{T_3}$
$T_1$ $T_3$ $T_5$ $T_2$ $T_6$ $T_4$
Query Graph
$E_{T_1}$ $E_{T_1.c_1}$ $E_{T_2.c_1}$ $E_{T_2}$
(a) Query Representation (b) Join Plan Representation

**Figure 3: State Representation Learning**

For a table $T$ with $n$ columns, the final table representation is computed with an *average pooling* layer by considering $\boldsymbol{E}_T^0$ and column representations of $T$ as follows:

$$\boldsymbol{E}_T = \text{AvgPool}([\boldsymbol{E}_T^0, \boldsymbol{E}_{T.c_1}, \boldsymbol{E}_{T.c_2}, ..., \boldsymbol{E}_{T.c_n}]),$$

where $\boldsymbol{E}_{T.c_i}$ denotes the representation of the $i$-th column in table $T$, which is calculated in Section 3.2. The table representations are summarized in the matrix $\boldsymbol{M}_T \in \mathbb{R}^{N \times D}$, where $D$ denotes the number of dimension, and $N$ is the number of tables in the database. The $i$-th column in $\boldsymbol{M}_T$ denotes the representation of the table $T_i$. The overview of the whole CTRL process is detailed in Algorithm 1. More description can be attached in the Appendix.

## 3.4 State Representation Learning

The value estimation of the state in DRL affects the learning performance, where a function, usually a multi-layer perception, is utilized to map the state representation to the value. Thus, an informative representation of the state may improve the estimation accuracy. In join order selection problems, the state often includes the target query $q$ and the current join forest $\mathcal{T}$ containing a set of current join plans. The final representation of the state is modeled by the concatenation of two components, i.e., $\boldsymbol{E}_q \oplus \boldsymbol{E}_{\mathcal{T}}$. Figure 3 shows the details for state representation learning, which includes query representation learning and join plan representation learning.

*3.4.1 Query Representation Learning.* We encode the participating tables and their links into the query representation. To do so, we first construct a query graph.

Definition 3 (Query Graph). *Given a query $q$, the query graph is denoted as an undirected graph $\mathcal{J} = (V^q, E^q)$, where $V^q$ is a set of nodes denoting the participating tables $\{T_1, T_2, ..., T_N\}$ in the query and $E^q$ is a set of edges connecting joinable tables.*

We represent the query graph $\mathcal{J}$ with a symmetry adjacent matrix $\boldsymbol{M}_A \in \mathbb{R}^{N \times N}$, where $N$ denotes the number of the tables in the database. If the tables $T_i$ and $T_j$ are joined in the given query, the element of the $i$-th ($j$-th) row and $j$-th ($i$-th) column in the matrix is denoted with 1, otherwise, it is 0. Previous studies [11, 32] flatten the adjacent matrix to represent the query. However, the symmetry adjacent matrix just captures the link information but ignores the node information. To this end, we apply a more expressive graph-based network [10] to capture the structure, as shown in Figure 3(a). In particular, we apply two-layer graph convolutional networks, the outputs of which are as follows:

$$\begin{aligned} H^{(2)} &= f(H^{(1)}, M_A) \\ H^{(1)} &= f(H^{(0)}, M_A) = f(M_T, M_A), \end{aligned} \tag{1}$$

where the function $f(\cdot)$ denotes a layer-wise propagation rule: $f(H^{(l)}, M_A) = \sigma(M_A H^{(l)} W^{(l)} + b^{(l)})$. Next, $W^{(l)} \in \mathbb{R}^{D\times D}$ is the weight matrix for the $l$-th layer, and $b^{(l)} \in \mathbb{R}^D$ is the bias. The weight matrix and the bias are the model parameters to learn. $\sigma(\cdot)$ is the non-linear activate function (e.g., sigmoid, ReLU, and tanh). Once we get the output $H^{(2)}$ after the graph convolutional networks, the representation of the query $E_q \in \mathbb{R}^D$ is obtained through the *average pooling* over $H^{(2)}$, shown in Equation (2).

$$E_q = \text{AvgPool}(H^{(2)}) \tag{2}$$

To sum up, we adopt the graph convolutional networks to vectorize the target query according to the query graph, which embeds both table and link information of queries.

### 3.4.2 Join Plan Representation Learning.

Another important component of the current state is the join plan after selecting the latest action. The join plan during the generation follows a forest structure, called join forest, indicating sequential information of the join order. We first define the join forest in the following.

Definition 4 (Join Forest). *Given a join plan and a set of selected joins, a join forest is denoted as an undirected acyclic graph $\mathcal{T}$ containing one or more join trees, in which the leaf nodes are the tables. The non-leaf nodes denote the join connections, whose left and right children represent the detailed join conditions. Each tree denotes a sub-join plan for the given join plan and the whole forest denotes the join plan.*

The join trees are the basic components of the forest, and sequential information from the bottom to the top implies the join order. To capture the sequential information under tree structures, several tree-based LSTM models [12, 27, 35] have been proposed. However, these models cannot perform efficiently due to the huge number of parameters and complex calculations. Thus, we propose to design a light-weight model for learning join plan representations. Inspired by the success of RTOS [32], a Tree-LSTM model, for encoding immediate join plans, we propose a tailored-tree-based attention model with fewer parameters, where tailored trees are used to denote immediate join plans by their nodes and reflect the join conditions by their columns. We further design an attention mechanism on top of the tailored trees to effectively and efficiently encode the immediate join plans.

Figure 3(b) shows an example of learning representations for join plans. We firstly obtain the representation of each join tree $t$ in the join forest and then encode the join forest by the *average pooling* operator as follows:

$$E_{\mathcal{T}} = \text{AvgPool}([o_{t_1}; o_{t_2}; \ldots]), \tag{3}$$

where $o_{t_i}$ denotes the representation of the $i$-th join tree $t_i$ in the join forest $\mathcal{T}$. We use the join forest representation $E_{\mathcal{T}}$ to denote the join plan representation.

To encode each join tree, we design a Join Tree Representation (JTR) algorithm with a join tree $t$ as input, as detailed in Algorithm 2. Benefiting from the tree structures, we encode the join tree recursively. There are two types of nodes in the join tree. (1) If the root node of the current tree $t$ is a leaf, which corresponds to table $T$, we represent the node with the table representation, i.e., $o_t \leftarrow E_T$ (lines 1–2). (2) If the node is not a leaf, which actually indicates a join, we represent the node with the join information, including the left and right join conditions (lines 4–7). The tailored trees and a self-attention mechanism are integrated to represent the non-leaf nodes (lines 8–9).

Specifically, the input of the self-attention module contains the representations of two nodes, $E_{n_L}$ and $E_{n_R}$, and the joined columns, $E_{c_L}$ and $E_{c_R}$, where $n_L$ and $n_R$ denote the left and right tables, respectively, and $c_L$ and $c_R$ denote the joined columns, respectively. The input is the formulated as $E = [E_{n_L}; E_{c_L}; E_{c_R}; E_{n_R}]$ with a shape of $(4, D)$. The self-attention mechanism has three parameters to learn, i.e., the key matrix $W^k \in \mathbb{R}^{D\times D}$, the value matrix $W^v \in \mathbb{R}^{D\times D}$, and the query matrix $W^q \in \mathbb{R}^{D\times D}$. The output of the self-attention module is formulated as:

$$o_t = \text{softmax}\left(\frac{EW^k(EW^q)^\top}{\sqrt{D}}\right)(EW^v) \tag{4}$$

This attention module indeed computes the importance weight of joined tables and joined columns, and then combines the inputs and the importance weight as the output.

**Algorithm 2:** JTR

```
Input: A join tree $t$, Attention model parameters
       $W = \{W^k, W^q, W^v\}$, Dimension of representations
       $D$, Table representations $\{E_T\}$, Column
       representations $\{E_c\}$
Output: Join tree representation $o_t$
1  if the root node $n$ of $t$ is a leaf then
2      $o_t \leftarrow E_T$; // $T$ denotes the corresponding table
3  else
4      $E_{n_L} \leftarrow$ JTR($t_{n_L}, W, D, \{E_T\}, \{E_c\}$);
       // $t_{n_L}$ denotes a tree with the root node $n_L$
5      $E_{n_R} \leftarrow$ JTR($t_{n_R}, W, D, \{E_T\}, \{E_c\}$);
6      $E_{c_L} \leftarrow$ Left column representation;
7      $E_{c_R} \leftarrow$ Right column representation;
8      $E \leftarrow [E_{n_L}; E_{c_L}; E_{c_R}; E_{n_R}]$;
9      $o_t \leftarrow \text{softmax}(\frac{EW^k(EW^q)^\top}{\sqrt{D}})(EW^v)$;
10 end
```

After traversing all the join trees in the join forest, we eventually get the representation of the join forest for the current state.

By concatenating the query representation and the join plan representation, the whole representation for the current state is calculated as $E_s = E_q \oplus E_{\mathcal{T}}$.

## 3.5 Q-learning

We adopt the widely-used value-based strategy, DQN, to learn the join order selection optimizer. Given the current state with its representation $\boldsymbol{E}_s$, the optimizer selects the action depending on the parameterized Q network, $Q_\theta(\boldsymbol{E}_s, a)$, shorted as $Q_\theta(s, a)$, where $\theta$ denotes the model parameters for the Q network. In this work, the parameter $\theta$ denotes the weight of the full connected layer with the shape of $(D, |\mathcal{A}_s|)$, where $D$ denotes the dimension of representations, and $\mathcal{A}_s = \{a_1, a_2, \dots\}$ denotes the action space. For each episode, each $Q_\theta(s, a)$ has a estimated value $y(s, a)$ which is calculated as:

$$y(s, a) = r(a) + \max_{a'} Q_{\theta'}(s', a'),$$

where $s'$ refers to the next state of the selected action $a$ under the current state $s$, and $r(a)$ denotes the reward of taking action $a$. Since the cost can be obtained until the complete join plan is generated, we get $r(a) = c(p^q)$ if the terminal state is obtained after taking action $a$, otherwise $r(a) = 0$. Next, $Q_{\theta'}$ denotes another network (target network) with the same structure with $Q_\theta$ but with different parameters. In practice, the two networks are initialized with the same structure. The backpropagation is performed on $Q_\theta$, whose parameters are copied to the target network $Q_{\theta'}$ periodically. The object function is to minimize the gap between the predicted Q value and the estimated Q value, defined with a regression loss:

$$\mathcal{L}(Q) = \sum_{(s,a)\in\mathcal{B}} \|y(s, a) - Q(s, a)\|_2^2, \tag{5}$$

where $\mathcal{B}$ denotes the sampled data from the experience memory pool. Each tuple in the experience memory pool records the current state $s$, the selected action $a$, and the next state $s'$ with the reward (i.e., cost). To increase the probability of reaching the global optimal, a $\epsilon$-greedy strategy is adopted to collect more diverse episodes. Especially, the optimizer randomly selects an action with the probability of $\epsilon$ and otherwise selects the action with the maximum Q value. In this way, the reinforcement learning process and the deep learning framework are integrated to learn the optimizer.

**Curriculum-Learning-based Optimization.** The training process of DRL fluctuates frequently. The reason lies in that uniformly sampling from the experience pool breaks the correlations between the consecutive data, i.e., the episode with incremental join tables. Meanwhile, deep networks are sensitive to the changes of parameters [24], which has a negative effect on the convergence of the training process and harms the search for finding the optimal join plan. To mitigate this phenomenon, we adopt curriculum learning, with the aim of obtaining a relatively stable learning process.

Curriculum learning refers to a learning strategy that learns from easier data to more difficult data [2, 5, 8, 20, 24]. This learning process imitates the learning behavior in human education, which is organized as a curriculum. The core of curriculum learning is the curriculum setting. An appropriate curriculum setting can effectively accelerate the training convergence and improve the robustness of the training process.

In the join order selection problem, the number of participating tables in a query can be taken as the learning difficulty. The setting is intuitive, i.e., a larger number of tables in a query implies more possible join orders hence a larger search space, which increases the difficulty of finding the optimal join order. Therefore, we propose a Curriculum-Learning-based Optimization (CLO) strategy, which sets the curriculum based on the number of participating tables in a query and gradually increases the difficulty of training data. The overview of the whole CLO process is detailed in Algorithm 3.

**Algorithm 3:** CLO

```
   Data: Training Query Set $\{q_1, q_2, ..., q_n\}$
   Input: Number of curriculum $k$, updating interval $I$
   Output: The optimal model $M^*$
1  Sort the training set in ascending order based on the
     number of participating tables ;
2  Split the sorted data into $k$ partitions $\{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_k\}$ ;
3  Initialize the training set $\mathcal{D} = \emptyset, i = 0$;
4  Initialize DQN model $M$;
5  for t=1,2,...,T do
6      if t is divided by I and $i < k$ then
7          $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{P}_i$ and $i \leftarrow i + 1$;
8      end
9      train($M$,$\mathcal{D}$) ;
10 end
```

## 4 EXPERIMENTS

### 4.1 Experimental Setup

*4.1.1 Dataset.* Experiments are conducted on two publicly available datasets, Join Order Benchmark (JOB) and TPC-H, to validate the effectiveness and efficiency of the proposed methods.

**Join Order Benchmark (JOB)** [14]: JOB is a real dataset from Internet Movie Data Base (IMDB) datasets, which reflects the relationship between movies, actors, etc. It has 113 queries generated from 33 templates, and each query involves a number of tables ranging from 4 to 17. JOB has 21 tables and 108 columns in total. The constructed schema graph for JOB has 21 nodes and 22 edges. **TPC-H** [23]: TPC-H is a standard database benchmark for the industrial test, where data is derived from the decision support applications for an ad-hoc querying workload. It has 8 tables and 61 columns. We finally get the version with 110 queries from 22 templates. The schema graph for the TPC-H dataset contains 8 nodes and 8 edges. For each dataset, we randomly select 90% of the queries for training and the rest for testing.

*4.1.2 Baseline.* We compare JOGGER with both the traditional methods and DRL-based methods.

**DQ** [11]: DQ encodes the query graph, join types, and the left and right side of the join to represent the state with one-hot vectors, where predicate information is also considered. DQ adopts DQN to learn the join order. **RTOS** [32]: RTOS constructs the feature vectors of join trees by leveraging the Tree-LSTM to capture the dynamic sequential information of the join trees. RTOS also utilizes the Deep Q-Network to train its agent. **QP100 (QP1000)**: QP100 and QP1000 randomly generate 100 and 1000 join plans respectively, from which the one with the lowest cost is selected.**Dynamic Programming (DP)**: For each query, it chooses the plan with the lowest cost by the dynamic programming algorithm. We use PostgreSQL to achieve this method by setting the parameter 'geqo_threshold' larger than the maximum number of joining tables.

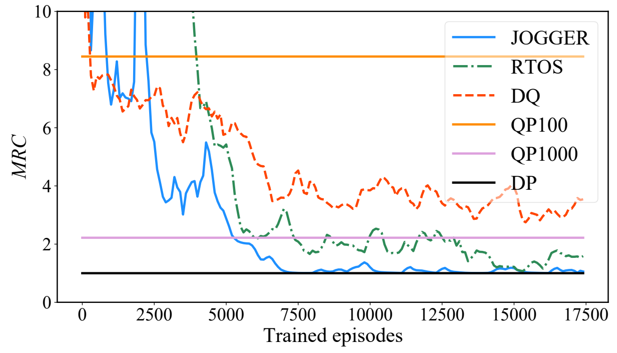

**Figure 4: Training curve on JOB. The *MRC* is evaluated on the testing queries.**

DQ and RTOS both utilize the feedback of latency from the DBMS to train the model. For fair comparisons, we train all of the DRL-based methods depending on the cost to validate the efficiency of the proposed method.

*4.1.3 Metrics.* The metric, Mean Relative Cost (*MRC*), is utilized to evaluate the methods following the previous study [32]. We take the DP method as the baseline to calculate the relative cost. A method with $MRC = 1$ has the same cost as DP, and a smaller value of *MRC* reflects a better performance.

## 4.2 Experimental Results

*4.2.1 Overall Performance.* We compare JOGGER with the baseline methods and report the performance in terms of *MRC*.

**Table 1: MRC TO DYNAMIC PROGRAMMING**

| Algorithm | *MRC* on JOB | *MRC* on TPC-H |
|---|---|---|
| JOGGER | **1.0038** | **1.0000** |
| RTOS | 1.0698 | **1.0000** |
| DQ | 2.7492 | 1.0217 |
| QP100 | 8.4484 | 1.1473 |
| QP1000 | 2.2188 | **1.0000** |

Table 1 reveals that JOGGER shows superior performances in terms of *MRC* on both JOB and TPC-H datasets. On the TPC-H dataset, whose search space is limited to 8 joins, all of the approaches can find relatively optimal join plans. QP1000 achieves better performance than QP100, because QP1000 can explore more different join orders when faced with a larger search space. However, the JOB dataset is more difficult to learn, and the disparities between the approaches become significant. JOGGER and RTOS outperform DQ, with improvements of 173% and 157% on the JOB dataset, respectively. The reason lies in that capturing the tree structures of join plans enhances the expressiveness of the state and thus improves the estimation accuracy. Furthermore, the proposed JOGGER outperforms RTOS with a relative 6.6% improvement on the JOB dataset. JOGGER achieves better performances with fewer model parameters, demonstrating the effectiveness of JOGGER.

Figure 4 depicts the training curve for the JOB dataset, which has a larger search space and makes finding the best join order more difficult. JOGGER decreases rapidly at the same time of converging to the lowest *MRC*. JOGGER achieves a lower *MRC* than QP1000 after 5,000 episodes, whilst RTOS just surpasses QP100. Moreover, JOGGER achieves nearly the same performance as DP with 7,500 episodes, whereas RTOS has just gone beyond QP1000. Compared with JOGGER and RTOS, DQ has not only the worst performance but also frequent fluctuations. The reason lies in that DQ may generate identical embeddings for different plans, ignoring the depth information, and lead to confusion for the optimizer. Although RTOS solves the problem, it requires a large number of parameters and takes a long time to reach a better result. Owing to the greater expressiveness and curriculum learning, JOGGER shows superior performance throughout the training process.

*4.2.2 Ablation Study.* In this section, we first verify the effectiveness of curriculum learning and then validate the effects of the different representation learning methodologies for tables. We gradually remove the important components of the JOGGER. Curriculum-Learning-based Optimization (CLO) is removed first, followed by Correlation-based Table Representation Learning (CTRL).

**Table 2: MRC TO DYNAMIC PROGRAMMING**

| Settings | *MRC* on JOB | *MRC* on TPC-H | Notation |
|---|---|---|---|
| JOGGER | **1.0038** | **1.0000** | JOGGER |
| w/o CLO | 1.0069 | **1.0000** | JOGGER-C |
| w/o CTRL&CLO | 1.1376 | **1.0000** | JOGGER-2C |

**Effect of CLO.** As shown in Table 2, JOGGER achieves the lowest *MRC* on the JOB dataset and performs slightly better than JOGGER-C. Curriculum learning also shows superior performance on the convergence speed, as indicated in Figure 10, where JOGGER converges within around 7,500 episodes and maintains a relatively stable performance after 7,500 episodes. JOGGER-C achieves a similar result until 10,000 episodes. This result demonstrates the advantage of accelerating the training process with curriculum learning. It can be seen that JOGGER has a relatively turbulent performance near the 2,000 and 4,000 episodes. This can be explained by the fact that we incrementally add the curricula $\mathcal{P}_2$ in the episode of 2,000 and $\mathcal{P}_3$ in the episode of 4,000. The optimizer encounters the unfamiliar states and makes certain corrections to its previous action choices. But the optimizer quickly adjusts and converges to a better value. By integrating CLO, JOGGER not only converges faster but also achieves a lower *MRC*.

**Effect of CTRL.** Table 2 illustrates the *MRC* of the three methods on the JOB and TPC-H datasets. JOGGER-C and JOGGER-2C only vary in the table representation approach. Both JOGGER-2C and JOGGER-C perform well on the simpler TPC-H dataset, with an *MRC* of 1.00. The simple representation strategy can work well in a small search space. JOGGER-2C, however, fails to capture the complicated information as the action space expands, resulting in an *MRC* of 1.1376 on the JOB dataset. The proposed JOGGER-C achieves a relatively 12.98% improvement compared with JOGGER-2C, illustrating the effectiveness of capturing the correlations between tables. The reason lies in that there are frequent correlations between tables, rather than independence. Capturing these relationships helps to improve estimation accuracy by increasing the expressiveness of representations.

*4.2.3 Efficiency.* To validate the learning efficiency of the proposed JOGGER, we further conduct experiments to measure the running time of the different approaches. We also take the state-of-the-art method, RTOS, into consideration. Each method is run for 17,500 episodes, with the time it takes to attain the lowest *MRC* recorded as well. Experiments are tested on the JOB dataset, which is more challenging to learn. The results are summarized in Table 3.

**Table 3: TIME-CONSUMPTION IN SECONDS**

| Algorithm | Total running time | #Episode of optimal value | Time of optimal value |
|---|---|---|---|
| JOGGER | **28,084.92** | **11,200** | **16,644.89** |
| JOGGER-C | 31,285.19 | 13,500 | 24,053.99 |
| RTOS | 49,008.07 | 15,300 | 42,551.12 |

From Table 3, it can be concluded that JOGGER shows superior performance in running time and convergence speed. The distinction between JOGGER and JOGGER-C is whether or not curriculum learning methodologies are used. Compared to JOGGER-C, JOGGER reduces the training time and reaches the optimal value faster. Absolutely, at the start of the training, queries with a smaller number of tables are trained, which take fewer iterations to get the representations, so that these queries require less time of training. Specifically, the total training time of JOGGER is 28,084.92 seconds, achieving a relative improvement of 13.39% over JOGGER-C. It reaches the optimal value in 11,200 episodes within 16,644.89 seconds, earlier than JOGGER-C. This further validates that curriculum learning can bring faster and more stable training to the model.

Another observation is that JOGGER shows superior performances in learning efficiency as evident by the result that even JOGGER-C runs faster than RTOS. The running time for running 17,500 episodes of JOGGER-C is 31,285 seconds, which is significantly smaller than that of RTOS. On the average time for 100 episodes, JOGGER-C even reduces 36.2% time compared to RTOS. The reason lies in that JOGGER-C has a substantially smaller number of model parameters than RTOS. By applying the two different types of Tree-LSTM to learn the representation of join plans, RTOS includes 22 variables for updating, whereas there are only three weight matrices in the tailored-tree-based attention module. Our proposed JOGGER-C has a much smaller number of parameters and thus requires less training time. We further summarize the parameters of baselines i.e., 2.2787M for DQ, 2.5156M for RTOS, and 1.1997M for JOGGER. These statistics correspond with the number of the models instead of the actual memory cost. In addition, JOGGER-C is more efficient than RTOS, as evidenced by the fact that JOGGER-C achieves the optimal value before RTOS.

*4.2.4 Latency Evaluation.* The ultimate goal of the join order selection is to find the optimal join order for execution and therefore we also care about the performance of latency. According to a recent study [32], we employ the Geometric Mean Relevant Latency (*GMRL*) to evaluate the latency performances of various models. Similar to *MRC*, a lower *GMRL* suggests a better performance over the latency. We evaluate the performance on the testing dataset and report the results of JOB and TPC-H with different templates. We compare JOGGER with the DRL-based method, RTOS, and traditional method, QP1000, where results are reported in Table 4.

From Table 4, we can clearly observe that JOGGER achieves the lowest *GMRL* on both JOB and TPC-H, followed by RTOS and QP1000. The *GMRL* of JOGGER is less than 1 in both two datasets, which implies that JOGGER develops better join plans than DP, i.e., PostgreSQL. Especially in the JOB dataset with a larger search space, JOGGER still shows superior performances, achieving a relatively 28.8% lower *GMRL* than RTOS and even a relatively 76.3% lower *GMRL* than QP1000.

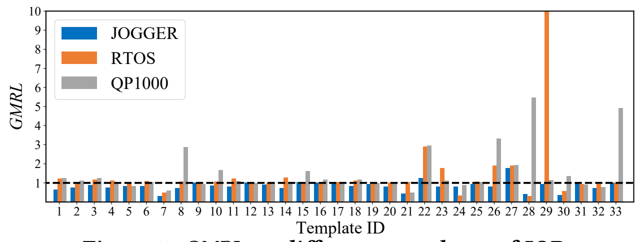

**Figure 5: *GMRL* on different templates of JOB**

**Table 4: GMRL TO DYNAMIC PROGRAMMING**

| Algorithm | *GMRL* on JOB | *GMRL* on TPC-H |
|---|---|---|
| JOGGER | **0.7993** | **0.9701** |
| RTOS | 1.1228 | 1.0038 |
| QP1000 | 3.3683 | 1.0823 |

Figure 5 further shows the performances in terms of *GMRL* for each template on the JOB datasets. Both JOGGER and RTOS have lower *GMRL* than QP on almost all templates of the JOB dataset. But RTOS performs dramatically bad on T29, which has the greatest number of tables, while JOGGER shows superior performance on this template. It once again provides evidence that our model can improve the effectiveness and efficiency of the DRL-based join order selection problem.

**Summary of Our Empirical Study:**

- JOGGER achieves the highest *MRC*, which outperforms the best among the baseline methods by 6.6% in the JOB dataset.
- Correlation-based Table Representation Learning (CTRL) improves 12.98% of *MRC* on the JOB dataset, demonstrating the effectiveness of capturing table correlations.
- Curriculum-Learning-based Optimization (CLO) strategy significantly reduces the running time, achieving a relative 13.34% improvement compared with that without curriculum learning, and reaches the optimum faster.
- JOGGER shows superior performances in terms of running time, reducing by 74.50% w.r.t. the state-of-the-art method.

## 5 RELATED WORK

### 5.1 RL for Join Order Selection

Reinforcement learning (RL), an essential part of machine learning, has been researched for databases over the last five years [34]. Reinforcement learning models do not require extensive high-quality training data. Especially for the join order selection, the community prefers to apply reinforcement learning in both online [28–30] and offline [6, 11, 13, 17, 32, 33] approaches to optimize join orders. DQ [11] uses deep Q learning as the whole learning framework where a basic neural network is employed. Rejoin [17] applies the value-based strategy, PPO [26], and embeds the height of the join trees into the feature vectors. AlphaJoin [33] utilizes the Monte-Carlo search tree as the basic architecture for reinforcement learning. These works utilize the fixed-size vectors to represent the join plans, disregarding the sequential information of the join orders. RTOS [32] further proposes a Tree-LSTM based optimizer to allow dynamically changing updates. However, none of the prior studies take into account learning efficiency, which is crucial for applying learning strategies.

## 5.2 Representations of Join Plans

The representations of join plans and queries are important, depending on which the cardinality and the value function are estimated. Neo [16] first utilizes a Tree Convolutional Network [19] to encode the tree structures of join plans. RTOS further adopts a Tree-LSTM [27] to capture the sequential information, where the non-leaf nodes in the join trees are encoded by the output of the Tree-LSTM network. This improves the expressiveness of the representations and the fits in dynamic changes of the join plans. However, LSTM requires a great number of parameters to train, which could lead to an overfitting problem for a small number of queries. Thus, we apply the lightweight attention mudole to encode the join plans. Despite of the existing models for tree structures in NLP tasks [1, 21, 31], it is still challenging to directly apply the attention mechanism to the join tree structures.

## 6 CONCLUSION

This paper focuses on the learning efficiency of deep reinforcement learning for the join order selection. To improve the efficiency in deep networks, we design the graph-based encoder to enhance the expressiveness of the representations for joins and queries while reducing the number of parameters in networks. As for the reinforcement learning process, we integrate a curriculum-learning-based optimization, where queries with a greater number of participating tables are later added to the training set. This strategy shows superior performance in accelerating the training process.

## ACKNOWLEDGMENTS

This work is partially supported by NSFC (No. 61972069, 61836007 and 61832017), and Shenzhen Municipal Science and Technology R&D Funding Basic Research Program (JCYJ20210324133607021).

## REFERENCES

[1] Mahtab Ahmed, Muhammad Rifayat Samee, and Robert E Mercer. 2019. Improving Tree-LSTM with Tree Attention. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*. IEEE Computer Society, 247–254.
[2] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the Annual International Conference on Machine Learning*. 41–48.
[3] Swati V Chande and Madhavi Sinha. 2011. Genetic optimization for the join ordering problem of database queries. In *India International Conference*. 1–5.
[4] Leonidas Fegaras. 1998. A new heuristic for optimizing large queries. In *International Conference on Database and Expert Systems Applications*. 726–735.
[5] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. 2017. Reverse curriculum generation for reinforcement learning. In *Conference on Robot Learning*. 482–495.
[6] Jonas Heitz and Kurt Stockinger. 2019. Join query optimization with deep reinforcement learning algorithms. *arXiv preprint arXiv:1911.11689* (2019).
[7] Yannis E Ioannidis and Younkyung Cha Kang. 1991. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the SIGMOD International Vonference on Management of Data*. 168–177.
[8] Lu Jiang, Deyu Meng, Shoou-I Yu, Zhenzhong Lan, Shiguang Shan, and Alexander Hauptmann. 2014. Self-paced learning with diversity. *Advances in Neural Information Processing Systems* 27 (2014), 2078–2086.
[9] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*.
[10] Thomas Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. *ArXiv* abs/1609.02907 (2017).
[11] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018).
[12] Phong Le and Willem H. Zuidema. 2015. Compositional Distributional Semantics with Long Short Term Memory. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics*. 10–19.
[13] Kyeong-Min Lee, InA Kim, and Kyu-Chul Lee. 2020. DQN-based Join Order Optimization by Learning Experiences of Running Queries on Spark SQL. In *International Conference on Data Mining Workshops*. 740–742.
[14] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the Very Large Data Base Endowment* 9, 3 (2015), 204–215.
[15] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proceedings of the Very Large Data Base Endowment* 14, 11 (2021), 1950–1963.
[16] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proceedings of the Very Large Data Base Endowment* 12, 11 (2019), 1705–1718.
[17] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.
[18] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *International Conference on Learning Representations*.
[19] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.
[20] Sanmit Narvekar, Jivko Sinapov, and Peter Stone. 2017. Autonomous Task Sequencing for Customized Curriculum Design in Reinforcement Learning.. In *International Joint Conference on Artificial Intelligence*. 2536–2542.
[21] Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, and Richard Socher. 2020. Tree-Structured Attention with Hierarchical Accumulation. In *International Conference on Learning Representations*.
[22] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *International Conference on Knowledge Discovery and Data Mining*. 701–710.
[23] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *Sigmod Record* 29, 4 (2000), 64–71.
[24] Zhipeng Ren, Daoyi Dong, Huaxiong Li, and Chunlin Chen. 2018. Self-paced prioritized curriculum learning with coverage penalty in deep reinforcement learning. *Transactions on Neural Networks and Learning Systems* 29, 6 (2018), 2216–2226.
[25] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).
[26] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).
[27] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*. 1556–1566.
[28] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *International Conference on Management of Data*. 1153–1170.
[29] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *Transactions on Database Systems* 46, 3 (2021), 1–45.
[30] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. 2008. A reinforcement learning approach for adaptive query processing. *History* (2008).
[31] Yaushian Wang, Hung-Yi Lee, and Yun-Nung Chen. 2019. Tree Transformer: Integrating Tree Structures into Self-Attention. In *Conferenceon Empirical Methods in Natural Language Processing-International Joint Conference on Natural Language Processing*. 1061–1070.
[32] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *International Conference on Data Engineering*. 1297–1308.
[33] Ji Zhang. 2020. AlphaJoin: Join Order Selection à la AlphaGo.. In *Very Large Data Base*.
[34] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database meets artificial intelligence: A survey. *Transactions on Knowledge and Data Engineering* (2020).
[35] Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. 2015. Long short-term memory over recursive structures. In *International Conference on Machine Learning*. 1604–1612.

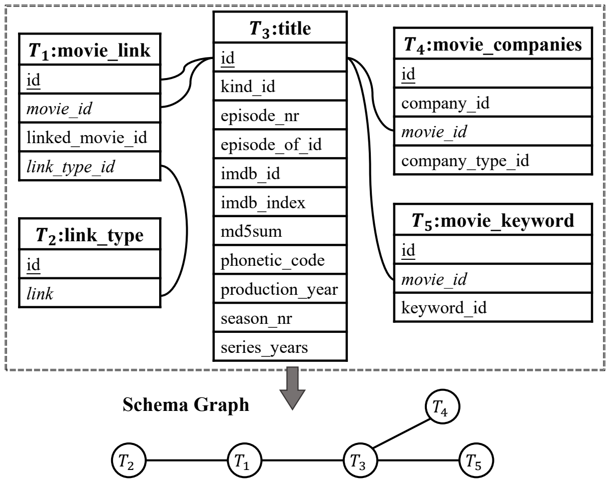

**Figure 6: Schema Graph Construction**

## A EXAMPLES

In order to better understand some concepts mentioned in this work, we provide several examples.

### A.1 MDP for Join Order

Reinforcement Learning (RL) is an important branch in machine learning concerned with how an agent can pick its actions in a dynamic environment to transit to new states in such a way that optimizes the sum of cumulative reward. The core of RL is to build a Markov Decision Process (MDP), where the effects of an action taken in a state depend only on that state and instead of the prior history. Deep learning techniques are then applied in RL to memorize the numerous states due to its strong expressiveness, called Deep RL (DRL). Deep neural networks also have the ability to estimate the unknown states depending on the previously states.

The agent (optimizer) interacts with the environment (the DBMS) in the following way. The environment records the current state, i.e., the immediate join plan, and generates a set of valid actions on which two tables can be joined under the current state. The optimizer selects an action from the action set depending on the value estimation, where deep neural networks are applied to estimate the value for actions and states. Then the selected action is rewarded by the DBMS, which changes to the next state with the selected join. This process is a MDP and stops until a complete join plan is constructed, where all the tables are joined. The whole process from the initial state to the terminal state is called an episode, after which a new episode will repeat until the model converges. Figure 7 shows an episode of the DRL-based join order selection procedure.

### A.2 Schema Graph

Example 1. *Figure 6 shows a database with five tables, denoting five nodes to be connected. Table* $T_3$ *is connected to table* $T_4$ *by the column id of* $T_3$ *and movie id of* $T_4$*. Thus, the table nodes* $T_3$ *and* $T_4$ *are connected. By enumerating all the primary-foreign keys, we obtain a schema graph.*

## B DETAILS OF ALGORITHM

### B.1 Correlation-based Table Representation Learning (CTRL)

First of all, we give an overview of the whole CTRL process in Algorithm 1. Given a database with $N$ tables as input, CTRL first builds a schema graph $\mathcal{S}$ based on the primary-foreign keys of tables (line 1), which represents the semantic information of the given database to some extent. Then the DeepWalk [22] algorithm is adopted, which iteratively performs *Random Walks* on a selected table node $T_i \in V^T$ and collects the sequence $L$ until reaching the number of trails (lines 2–5). After that, it performs *Skip-gram* algorithm and uses an average pooling layer to get the representation $\boldsymbol{E}_T^0$ of each table node (line 6). We can see that CTRL focuses on the global location information, i.e., the neighbours, and thus captures the correlations between tables by randomly walking on the schema graph. The learned table representations can either be frozen or be further updated in the later training steps. Next, we elaborate on the details of the schema graph construction and table representation learning in Algorithm 1.

### B.2 Curriculum-Learning-based Optimization

The overview of the whole CTRL process is detailed in Algorithm 3 Given a set of training queries $\{q_1, q_2, ..., q_n\}$ as input, CLO first sorts the queries in ascending order based on the number of participating tables (line 1) and splits the training query set into $k$ partitions (line 2). Thus, we obtain the curriculum for training, which rises in difficulty as the number of tables increases. After that, CLO incrementally adds the partitions into the training set at regular intervals (lines 6-8) until all of the training queries have been put into training. DQN is then trained based on the current training dataset (line 9). By incorporating curriculum learning to DQN, the training set shifts from easier to more difficult queries, which is beneficial to the model convergence.

### B.3 Column Embeddings

As shown in Figure 8, we concatenate the representations of join and selection predicates as well as the embedding of the column identity (ID) as the final column representation.

Example 2. *For the predicate "title.production_year BETWEEN 1980 AND 1984", the estimated selectivity is 0.5. The value of 0.5 can be allocated into the second partition and thus the vector for the column t "title.production_year" is* $[0, 1, 0]$*, which will be sent to the following linear layers.*

## C EXPERIMENT DETAILS

### C.1 Dataset Statistics

**Table 5: DATASET INFORMATION**

| DataSet | Size | #Tables | #Columns | #Templates | #Queries |
|---|---|---|---|---|---|
| JOB | 3.6G | 21 | 108 | 33 | 113 |
| TPC-H | 4.0G | 8 | 61 | 22 | 110 |

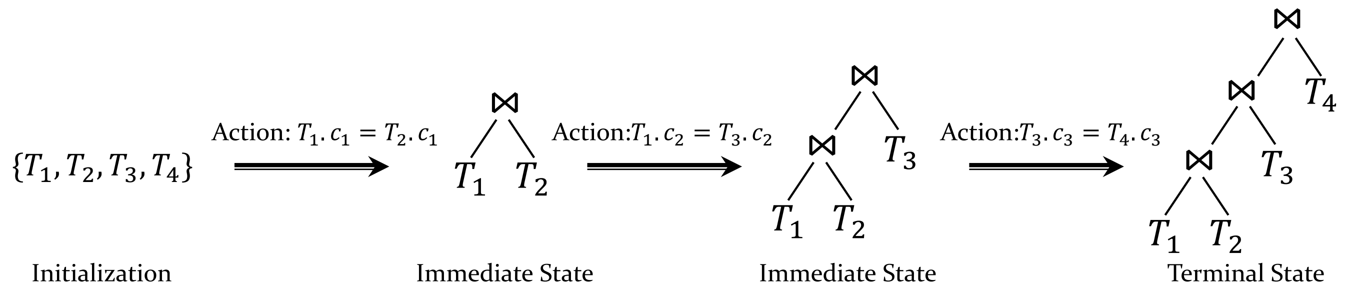

**Figure 7: An Episode of DRL-based Join Order Selection**

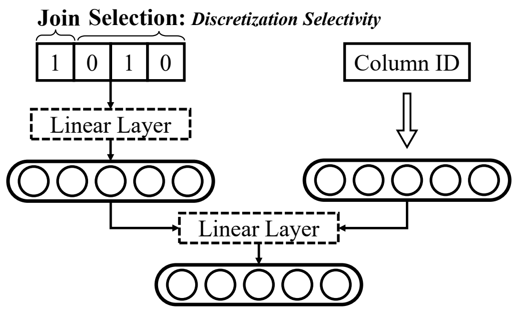

**Figure 8: Column Representation Learning**

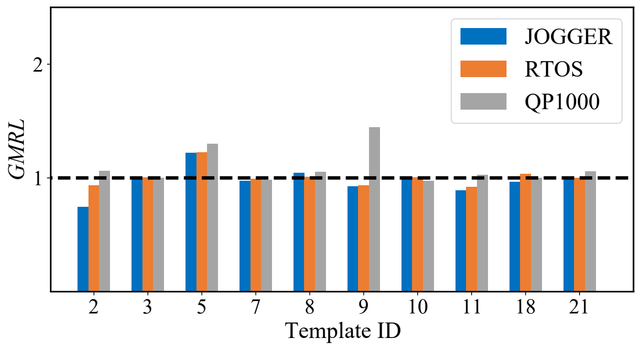

**Figure 9:** ***GMRL*** **on different templates of TPC-H**

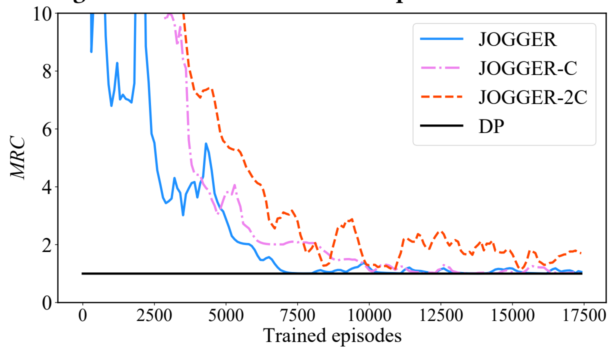

**Figure 10: Training curve on JOB for ablation study.**

## C.2 Implementation Details

All of the experiments are run on an Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, and NVidia GeForce RTX 3080 GPU. We tune the parameters of DRL-based methods, DQ, RTOS, and JOGGER. The dimension of the embeddings is set to 128 and the batch size is set to 256. The optimizer Adam [9] with a weight decay of 0.005 is adopted here to update the model parameters. The learning rate is set as 0.003 by default.

To get table representations, we utilize the public implementation of DeepWalk[1]. The dimension of embeddings is set to 128, the number of walks to 10, and the maximum length of the walks to 40. These table representations are input into the models without updating during the training process.

For curriculum learning, we divide the original training set into three curricula: $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$, with no duplicate data in each curriculum. The updating interval for the training data is set to 2,000. Specifically, The first 2,000 episodes are trained with $\mathcal{P}_1$, where the queries have fewer participating tables. The later 2,000 episodes are trained with $\mathcal{P}_1 \cup \mathcal{P}_2$. After 4,000 episodes, the curriculum $\mathcal{P}_3$ is added for training, and the model is trained on the entire training set until the end.

## C.3 Latency Performance on TPC-H

Figure 9 shows the performance on the TPC-H dataset. On the TPC-H dataset, due to the smaller number of tables in the queries, the discrepancy gap between models on each template is not large. However, we observe that the DRL-based methods JOGGER and RTOS have better performance than the traditional method QP.

## C.4 Ablation Study

A training curve on JOB is shown in Figure 10, where we can see that both JOGGER-C and JOGGER-2C have a decreasing tendency. Both approaches go below 2 after 8,300 episodes. But the difference occurs after 8700 episodes. JOGGER-C reaches the minimal value of 1.0069 after 11,200 episodes while JOGGER-2C is in a fluctuating state. This result implies that introducing the primary-foreign key relationships benefits better convergence to learn the join order.

[1]https://github.com/phanein/deepwalk