# Reinforcement Learning with Tree-LSTM for Join Order Selection


Xiang Yu
Tsinghua University, China
x-yu17@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University, China
liguoliang@tsinghua.edu.cn

Chengliang Chai
Tsinghua University, China
chaicl15@mails.tsinghua.edu.cn

Nan Tang
QCRI, Qatar
ntang@hbku.edu.qa



***Abstract*—Join order selection (JOS) – the problem of finding the optimal join order for an SQL query – is a primary focus of database query optimizers. The problem is hard due to its large solution space. Exhaustively traversing the solution space is prohibitively expensive, which is often combined with heuristic pruning. Despite decades-long effort, traditional optimizers still suffer from low scalability or low accuracy when handling complicated SQL queries. Recent attempts using deep reinforcement learning (DRL), by encoding join trees with fixed-length hand-tuned feature vectors, have shed some light on JOS. However, using fixed-length feature vectors cannot capture the structural information of a join tree, which may produce poor join plans. Moreover, it may also cause retraining the neural network when handling schema changes (e.g., adding tables/columns) or multi-alias table names that are common in SQL queries.**

**In this paper, we present `RTOS`, a novel learned optimizer that uses *R*einforcement learning with *T*ree-structured long short-term memory (LSTM) for join *O*rder *S*election. `RTOS` improves existing DRL-based approaches in two main aspects: (1) it adopts graph neural networks to capture the structures of join trees; and (2) it well supports the modification of database schema and multi-alias table names. Extensive experiments on Join Order Benchmark (JOB) and TPC-H show that `RTOS` outperforms traditional optimizers and existing DRL-based learned optimizers. In particular, the plan `RTOS` generated for JOB is 101% on (estimated) cost and 67% on latency (i.e., execution time) on average, compared with dynamic programming that is known to produce the state-of-the-art results on join plans.**


## I. Introduction

Join order selection (JOS) is a key DBMS optimization problem, which has been extensively studied for decades [1], [26], [32], [36], [38]. Traditional methods typically search the solution space of all possible join orders with some pruning techniques, based on cardinality estimation and cost models. The dynamic programming (DP) based algorithms [10] often select the best plan but are very expensive. Heuristic methods, such as GEQO [3], QuickPick-1000 [38], and GOO [4], may compute plans more quickly, but often produce poor plans.

Recently, machine learning (ML) and deep learning (DL) based methods for learned optimizers [11], [15] have become popular in database community. In particular, deep reinforcement learning (DRL) based methods, such as ReJOIN [20] and DQ [16], have shown promising results – they can produce plans that are comparable with native query optimizers but can execute much faster after learning.

**Shortcomings of Existing DRL-based Methods.** Existing DRL-based methods for learned optimizers (e.g., DQ and ReJOIN) encode a join tree as a fixed-length vector, where the length of the vector is determined by the tables and columns in a database. This will cause two problems: (1) these vectors cannot capture the structural information of a join tree, which may lead to poor plans; and (2) the learned optimizers will fail when the database schema changes (e.g., adding columns/tables) or multi-alias table names, which requires a new input vector with a different length, and then retrains the neural network. Let us further illustrate through an example.

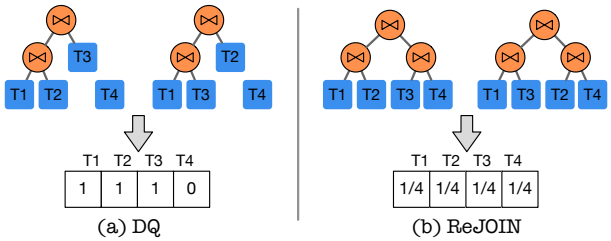



Fig. 1. Different joins trees with the same feature vector.

**Example 1:** Consider a database with 4 tables $T_1, T_2, T_3, T_4$.

DQ [16] uses one-hot encoding (1 means that a table is in the tree and 0 otherwise) to encode a join tree: $(T_1 \bowtie T_2) \bowtie T_3$ and $(T_1 \bowtie T_3) \bowtie T_2$ have the same feature vector $[1, 1, 1, 0]$, as shown in Figure 1(a).

ReJOIN [20] uses the depth of the table in the join to construct the feature vectors: $(T_1 \bowtie T_2) \bowtie (T_3 \bowtie T_4)$ and $(T_1 \bowtie T_3) \bowtie (T_2 \bowtie T_4)$ have the same feature vector $[\frac{1}{2}^d, \frac{1}{2}^d, \frac{1}{2}^d, \frac{1}{2}^d] = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]$, where the depth of each table is $d = 2$ as shown in Figure 1(b). □

**Key Observation.** Example 1 shows that, join trees with different join orders may be encoded into the same feature vector using existing learners; that is, they only consider the static information (such as tables and columns) of a join, without being able to capture the structural information of a join tree. Intuitively, a better learner should also understand different structural information of different join trees.

**Our Methodology.** Based on the above observation, we present `RTOS`, a novel learned optimizer using tree-structured long short-term memory (Tree-LSTM) [35]. `RTOS` trains a DRL model for JOS, which can automatically improve future JOS by learning from previously executed queries.

Tree-LSTM is one kind of graph neural networks (GNNs). GNNs on graph structure data have shown excellent performance [40] for various applications, such as social net-

works [7] and knowledge graphs [6]. Different from traditional LSTM structures that take serial data as input, Tree-LSTM directly reads a tree structure as input and outputs a *representation* of the tree. We use tree-based representation to tackle Shortcoming (1). Furthermore, we use the dynamic graph feature to support schema changes and multi-aliases, to overcome Shortcoming (2).

Our essential goal is to generate a plan with low latency (i.e., the execution time). However, estimating latency is much more expensive than cost (estimated from a cost model). We first use cost as feedback to train the model and then switch to latency as feedback for fine-tuning. Different from [21] that only uses latency as feedback, we treat these two feedback as two separate tasks but sharing one common representation, through *multi-task learning* [29]. By doing so, the model can learn cost and latency together, by setting the objective loss function as a weighted sum of these two problems.

One may concern that training a deep neural network is not cheap, because it requires a lot of training data and the feedback from interacting with the neural network, which is also reported in DQ [16]. However, training a DL model for a database is typically considered as an off-line operation, and once it is trained, it is very efficient to use it for prediction.

**Contributions and Roadmap.** We propose RTOS, using DRL with Tree-LSTM, to approach JOS. The main contributions and roadmap of this paper are as follows.

(1) We give an overview of RTOS. (Section II).

(2) We introduce the *representation* of an intermediate join plan (i.e., a join forest), which combines the representations of the given query and the join trees in the join forest, by leveraging Tree-LSTM. (Section III).

(3) We describe how to adapt Deep Q learning [23] to solve our DRL problem for JOS. (Section IV).

(4) We discuss how RTOS can handle database modifications such as adding columns/tables, as well as dealing with multi-aliases. (Section V).

(5) Extensive experiments on two popular benchmarks, Join Order Benchmark [17] and TPC-H, show that RTOS outperforms existing solutions by producing join plans with lower latency and cost. (Section VI).

## II. An Overview of RTOS

Our primary focus is on SPJ queries, simiar to DQ [16]. RTOS trains a learned optimizer using DRL with Tree-LSTM. Given an SQL query, it outputs a join plan. A DBMS will execute this join plan and send feedback, which is then used by RTOS as new training samples to improve itself.

### A. The Working Mechanism

*1) Deep Reinforcement Learning (DRL) in* RTOS*:* Reinforcement learning (RL) is a method that an *agent* (e.g., the optimizer) learns from the feedback through trial-and-error interactions with the *environment* (e.g., a DBMS), which typically incorporates deep learning (DL). For each *state* (e.g., an intermediate join plan), RTOS reads the plan and leverages

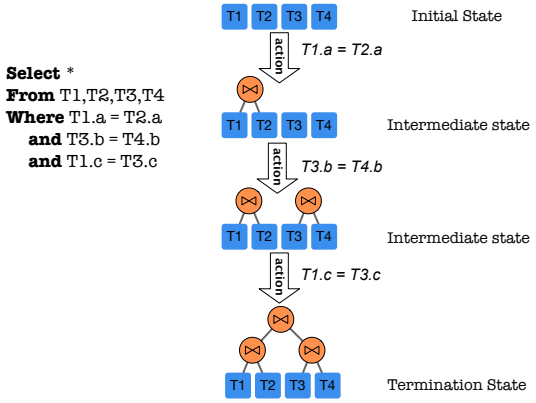



Fig. 2. RTOS selects one join as an action until all tables are joined.

Tree-LSTM [35] to compute an estimated long term *reward* (e.g., the feedback from a DBMS) of each *action* (e.g., which two tables should be joined to make the current state more complete). It then selects the expected optimal action with the largest reward (minimum cost). When all the tables are joined (i.e., a complete join plan is derived), the join plan will be sent to a DBMS to execute. RTOS will then take the feedback from the DBMS to train the Tree-LSTM and update the optimizer.

*2) Feedback of Latency and Cost:* There are two types of information from a DBMS that we use as feedback: latency and cost. *Latency* is the actual execution time of a join plan, which is the essential goal to optimize. However, it can only be obtained after executing SQL queries, which is expensive. *Cost* is an estimation of latency, which is typically given by cost models and oftentimes is inaccurate.

Besides using the speed advantage of cost to train a learner, we also use the latency feedback, with the following two steps.

**Cost training** first uses cost as a reward to train a RL model. After this phase is completed, our model can generate a good plan with cost as the indicator. Meanwhile, the neural network has an understanding of the database based on the cost.

**Latency tuning** further incorporates latency as feedback and uses it as the new optimization goal. Through the previous step of cost training, the neural network has already had an approximate understanding of the database. Based on this, we use the information of latency for continuously training. Hence, this step can be considered as fine-tuning.

*3) Incrementally Maintaining the Model:* Deep reinforcement learning has been widely used in dynamic environments, for which just the rewards of state-action pairs may change over time. For our problem, we record all the executed queries, and use the execution result (cost, latency) to update our model in the background. By doing so, our model can adapt to modifications in real-time. We have also discussed two cases "add a column" and "add a table" in Section V.

### B. The RTOS Framework

Informally speaking, a *state* in the join process is a *join forest*, which may consists of several *join trees*.

**Example 2:** Consider an SQL query in Figure 2. Its initial state is a join forest with four join trees $T_1, T_2, T_3$ and $T_4$. The 1st

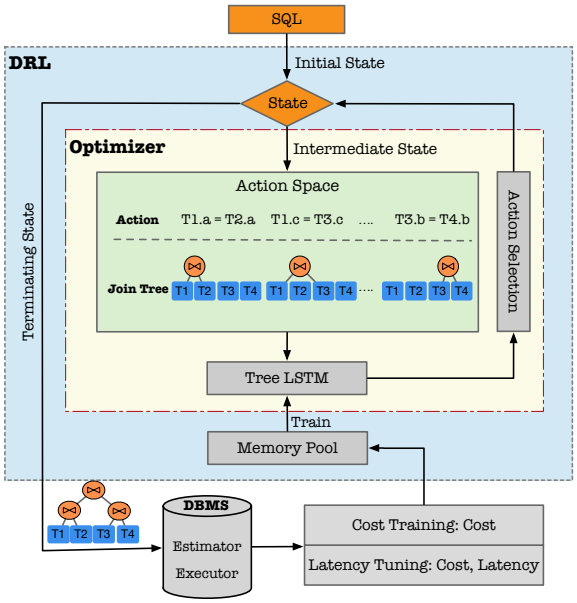



Fig. 3. The RTOS framework.

intermediate state is a join forest with three join trees: $(T_1 \bowtie T_2)$, $T_3$ and $T_4$. The 2nd intermediate state is a join forest with two join trees: $(T_1 \bowtie T_2)$, $(T_3 \bowtie T_4)$. The terminating state is a *complete* plan with only one join tree. □

Next we present its framework, as shown in Figure 3.

*1) DRL optimizer:* The learned optimizer.

**– State** maintains the current *state* information for the join process. **Terminating State** is one join tree that all tables are joined together, which will be converted to a query plan and passed to a DBMS for execution. **Intermediate state** is the *state* that holds partial join plans. An intermediate state will be passed to the **Optimizer** to pick the *action* that should be performed now, and **State** itself is updated according to the selected *action*.

**– Optimizer** corresponds to the *agent* in RL, which is the core part of the entire system. For a given *state*, each candidate join condition can be considered as an *action*. The **Action Space** contains all possible *actions* and corresponding join trees (e.g., $\{(T_1.a = T_2.a, \mathsf{joinTree}(T_1.a = T_2.a)), ...\}$). For join trees in the action space, we will use **Tree-LSTM** to represent them and get the corresponding estimated long term reward. **Action Selection** gets the result of each join tree from **Tree-LSTM** and selects the *action* corresponding to the optimal join tree:

$$action = \underset{action' \in \textbf{Action Space}}{\arg\min} \mathsf{TreeLSTM}(\mathsf{joinTree}(action'))$$

**– Memory Pool** records the status of the plans generated by RTOS and the feedback from the DBMS. DRL requires training data to train neural networks. Common practices use replay memory [23] to record the status and systems feedback. We use a memory pool here and sample training data from it to train the Tree-LSTM.

*2) DBMS:* RTOS generates a join plan for a given query and then passes it to a DBMS, e.g., PostgreSQL. We use two components from the DBMS, an **Estimator** and an **Executor**.

TABLE I
NOTATIONS.

| Notation | Description |
|---|---|
| $n$ | the number of tables in database |
| $R(c)$ | representation for column $c$ |
| $R(t)$ | representation for table $t$ |
| $\mathcal{T}$ | a join tree |
| $R(\mathcal{T})$ | representation for $\mathcal{T}$ |
| $\mathcal{F}$ | a join forest that consists of several join trees |
| $R(\mathcal{F})$ | representation for forest $\mathcal{F}$ |
| $R(q)$ | representation for SQL $q$ |
| $\theta$ | one join condition in query |
| $\Theta$ | all join conditions in query |
| $s$ | join state during the join process |
| $R(s)$ | representation for join state $s$ |
| $hs$ | the size of hidden layer in neural network |
| $A$ | an action in RL |
| $\mathbb{A}$ | action space of all actions |
| $M$ | attribute matrix for column $c$ in neural network |
| $F(c)$ | Feature vector of column c |

**Estimator** can give the cost of the plan using statistics to estimate the cost without executing the **Executor**.

Getting the latency of each join condition is difficult. In order to reduce the difficulty of implementation and make the system easier to migrate to other systems, we adopt a classical approach, which is also used in ReJoin [20] and DQ [16]. We set the feedback (reward) of each step of the intermediate state to 0, and the feedback of the terminating state to the cost (latency) of the entire plan.

## III. Representation of the State

The performance of ML/DL methods is heavily dependent on data representations (or features), which are typically represented by vectors. In our case, we need to learn representations of queries, columns, tables and join trees. To learn the representation $R(s)$ of a state $s$, we need to consider both the current status (i.e., a join forest $\mathcal{F}$) and its target status (i.e., an SQL query $q$). The representation $R(s)$ of the state $s$ is the concatenation of the representation $R(\mathcal{F})$ of the forest $\mathcal{F}$ and the representation $R(q)$ of the query $q$; that is, $R(s) = R(\mathcal{F}) \oplus R(q)$.

In what follows, we will first describe the representation $R(q)$ of query $q$ into a vector (Section III-A). We then discuss the representations of columns and tables of the given query into a vector (Section III-B). We introduce Tree-LSTM (Section III-C1), and discuss how to use Tree-LSTM to learn the representation of a join tree based on the representations of columns and tables (Section III-C2). We close this section by giving the representation $R(\mathcal{F})$ of a join forest $\mathcal{F}$ and the representation $R(s)$ of a state $s$ (Section III-C3).

We summarize the notations used in this paper in Table I.

### A. Representation of Queries

The representation $R(q)$ of a query $q$ is a vector that contains all the join information about the tables in $q$.

Let $n$ be the number of tables in a database, and each table has a unique identifier from 0 to $n-1$. Let $m$ be a $n * n$ matrix where each cell is 0 or 1. $m_{i,j}$ is 1 means that there is a join relation between the $i$-th table and $j$-th table and 0 otherwise. This matrix is then flattened into a vector $v$ by

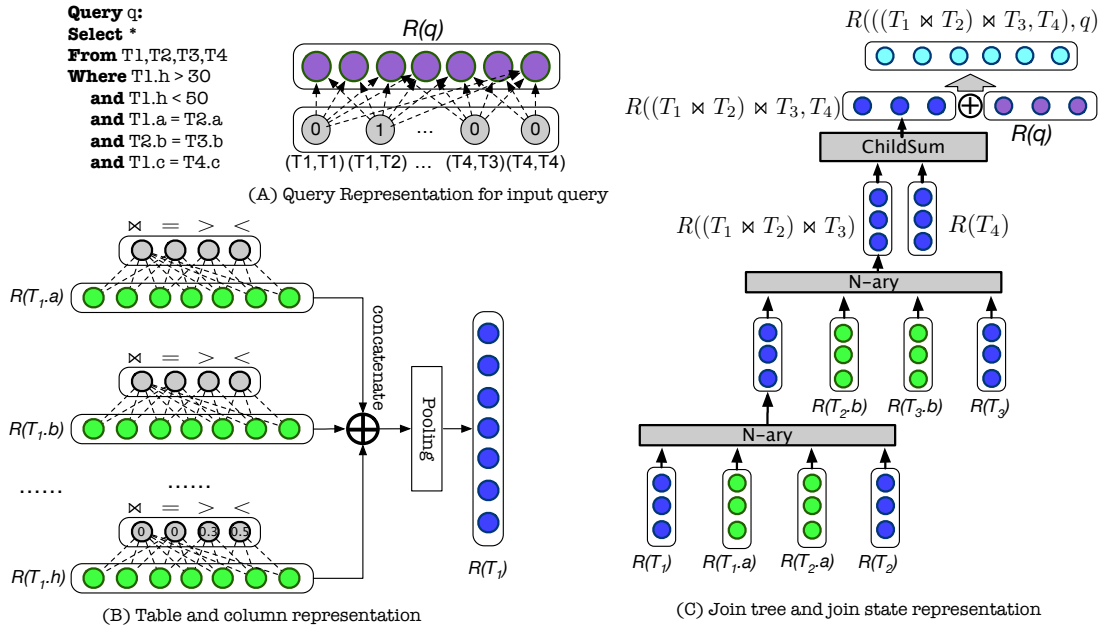



Fig. 4. Representations in RTOS.

putting all rows, from 0 to $n-1$, of the original matrix $m$ into the vector, i.e., $v_{i*n+j} = m_{i,j}$ and $|v| = n * n$.

Afterwards, we apply one Fully connect (FC) layer on the vector $v$ (see Figure 4(A)) to get the representation of $q$:

$$R(q) = \mathsf{FC}(v) = \sigma(vW + b)$$

where $\sigma$ is one activation function (e.g., tanh, Sigmoid, ReLU) in neural network; $W$ (matrix with shape $(n*n, hs)$) and $b$ (vector with shape $(1, hs)$) are parameters in FC layer which need to be learned using training samples; $hs$ is the size of hidden layers in neural network; and $R(q)$ is a vector with shape $(1, hs)$ that represents the query $q$.

One improved method from this matrix based representation for learning the query has good properties to handle schema changes and multi-aliases (see Section V-A for more details).

### *B. Representation of Columns and Tables*

Columns and tables are two key components in a query. Given a query $q$, we just need to use the predicates in $q$ to construct the representations of columns, without scanning the database. Next we first discuss the representation $R(c)$ of a column $c$, followed by using column representations to construct the representation $R(t)$ of a table $t$.

**Column Representation.** We consider two types of columns: numerical values and other values (e.g., string). Note that in a query, a column $c$ can be associated with two operations, `Join` and `Selection`, and both need to be considered. In particular, we encode `Selection` information into column representation $R(c)$ for column $c$, which can be viewed as pushing all `Selection` into leaves in join tree.

(i) For a column $c$ with numerical values, the `Selection` operation can be categorized into three cases: $=, >$ and$<$. Therefore, we encode $c$ as a feature vector of length 4: $F(c) = (c_{\bowtie}, c_{=}, c_{>}, c_{<})$, where $c_{\bowtie} = 1$ if the column $c$ exists in a join predicate and 0 otherwise. For the other three `Selection` operations, solely encoding the existence as the `Join` operation is not enough because the value in the predicate matters. For example, given a predicate $c > v$, we should consider the value $v$ as the feature. Since data in different columns has different scales, we normalize the value into $[0, 1]$ based on the the maximum ($c_{max}$) and minimum ($c_{min}$) value in the column. Next we illustrate the three cases.

- For predicate $c = v$, if $v < c_{min}$ or $v > c_{max}$, we set $c_{=} = -1$ because we cannot retrieve any data in that situation, otherwise $c_{=} = v_{nor} + 1 = \frac{v - c_{min}}{c_{max} - c_{min}} + 1$.
- For predicate $c > v$, if $v \geq c_{max}$ then $c_{>} = 1$, else if $v < c_{min}$ then $c_{>} = 0$, otherwise $c_{>} = v_{nor}$.
- For predicate $c < v$, if $v \leq c_{min}$ then $c_{<} = 1$, else if $v > c_{max}$ then $c_{<} = 0$, otherwise $c_{<} = 1 - v_{nor}$.

**Example 3:** Consider the query in Figure 4(A), $F(T_1.h) = (0, 0, 0.3, 0.5)$. Column $T_1.h$ has no join condition nor "=" predicate, hence $c_{=} = 0$ and $c_{\bowtie} = 0$. The maximum and minimum for $T_1.h$ are 0 and 100, so we set $c_{>} = 0.3$ for $T_1.h > 30$ and $c_{<} = 1 - 0.5 = 0.5$ for $T_1.h < 50$. □

We define a matrix $M(c)$ with shape $(4, hs)$ for each column $c$. $M(c)$ has learned parameters that contain the information of the column. The representation R(c) for numerical column $c$ with shape $(1, hs)$ is:

$$R(c) = F(c) * M(c)$$

For the column $c$ have predicates connected by "or", we handle these predicates separately, and apply max pooling to get one representation.

(ii) For a column $c$ with other values (e.g. string). It cannot be mapped to a value with interval meaning (">","<") by a simple method (e.g. hash), but requires a more complex encoding method (e.g. word2vec [22]). Here we only encode it using its selectivity information and get the column representation.

To align with the above feature vector, we also use a vector of size 4 to encode a column, i.e., $(c_{\bowtie}, c_{=}, c_{>}, c_{<})$.

$c_{\bowtie}$ is computed the same as above. $c_{>} = 0$ and $c_{<} = 0$ because interval information of a string column could not be represented by numerical value directly. For $c_{=}$, given a predicate $v$, we estimate the selectivity of $v$ and set $c_{=}$ to the estimated selectivity.

**Table Representation.** For a table $t$ with $k$ columns, we use its columns' representations to construct its table representation $R(t)$. More specifically, we concatenate $k$ column representations $(R(c_1), R(c_2), ..., R(c_k))$ into a matrix $M(t)$ with the shape $(k, hs)$, $M(t) = (R(c_1) \oplus \ldots \oplus R(c_k))$, and apply an average pooling layer (kernel shape $(k, 1)$) to get the table representation (see Figure 4(b)) of shape $(1, hs)$, as:

$$R(t) = \mathsf{MaxPool}(M(t))$$

### C. Representations of Join Tree, Join Forest and the State

Next, we will first describe how to use Tree-LSTM to learn the representation of a join tree, because a join tree is a graphical structure and it is difficult to directly obtain its representation by constructing a feature vector.

*1) **Tree-LSTM**:* Traditional LSTM [8] shows its power to acquire sequential data features even for a long sequence (e.g., in natural language processing). We have a sequence input $X = (x_1, x_2, x_3, \ldots)$. At each time step $i$, hidden state vector $h_i$ represents the current state at step $i$ and the memory cell vector $m_i$ preserves long-term information over $x_1, x_2, \ldots, x_i$ to make it handle the long sequence. LSTM uses an LSTMUnit over input $x_i$ and previous representation $(h_{i-1}, m_{i-1})$ at step $i-1$ to get $h_i$ and $m_i$, $h_i, m_i = \text{LSTMUnit}(x_i, h_{i-1}, m_{i-1})$.

However, traditional LSTM reads sequential data and cannot be directly applied to complex structures such as trees. In order to solve this limitation, Child-Sum Tree-LSTM and $N$-ary Tree-LSTM [35] have been proposed to run on a tree structured input. With the given tree, Tree-LSTM can automatically learn the structure information of the tree and give its representation.

**Child-Sum Tree-LSTM** does not consider the order of its children. For a given tree node $j$ with several child nodes $\alpha_{jk}$, it sums all the child nodes' representations, and then constructs its representations.

**$N$-ary Tree-LSTM** considers the order of its children. For a given tree node $j$, the representation of its $k$-th child node $\alpha_{jk}$ will be calculated separately. For the $k$-th child node, it will have its own weight matrix w.r.t. $k$. The order information will be caught by these position dependent weight matrices.

We combine these two Tree-LSTM according to the characteristics of join forest. The forest $\mathcal{F}$ is composed of several join trees $\{\mathcal{T}_1, \mathcal{T}_2, \ldots\}$. Without any human feature on tree structure, we take the join trees as input and use the model to automatically learn the representation $R(\mathcal{T})$ for join tree $\mathcal{T}$.

*2) **Representation for Join Tree**:* As shown in Figure 4(C), we construct a tree model for a join tree. The leaf node could be a table or a column. An internal node in a join tree corresponds to a join and is composed of 4 nodes $(\alpha_0, \beta_0, \beta_1, \alpha_1)$. $\alpha_0$ and $\alpha_1$ are two join trees (tables) need to be joined. $\beta_0$ and $\beta_1$ are nodes corresponding columns in this join. $\alpha_0, \beta_0, \beta_1, \alpha_1$ are position sensitive, we apply $N$-ary Tree-LSTM in the join tree.

**Function** `JoinTreeDFS(`*Node*`)`

**Input**: Table Representation, Column Representation, $N\text{-}aryUnit$
**Output**: State representation of Join Tree

**if** *Node is a leaf* **then**
  $h = R(Node)$
  $m = Zeros\_init()$
  Return $h, m$
**else**
  $h_{\alpha_0}, m_{\alpha_0} = \mathsf{JoinTreeDFS}(Node \to \alpha_0)$
  $h_{\alpha_1}, m_{\alpha_1} = \mathsf{JoinTreeDFS}(Node \to \alpha_1)$
  $h_{\beta_0}, m_{\beta_0} = \mathsf{JoinTreeDFS}(Node \to \beta_0)$
  $h_{\beta_1}, m_{\beta_1} = \mathsf{JoinTreeDFS}(Node \to \beta_1)$
  Return $N\text{-}aryUnit(h_{\alpha_0}, m_{\alpha_0}, h_{\alpha_1}, m_{\alpha_1}, h_{\beta_0}, h_{\beta_1})$

Fig. 5. Dynamically constructing the neural network of given join tree.

For a tree node $j$, we use $h_j$ to denote its representation, and $m_j$ for the memory cell.

- If node $j$ is a leaf representing a single table, $h_j = R(j)$, $m_j$ will be initialized as a zero vector.
- If node $j$ is a leaf representing a single column, $h_j = R(j)$, $m_j$ will be initialized as a zero vector.
- For a node $j$ representing a join, it has 4 child nodes $(\alpha_{j,0}, \beta_{j,0}, \beta_{j,1}, \alpha_{j,1})$. We apply $N$-aryUnit in $N$-ary Tree-LSTM on the representation of these four nodes to get $h_j$ and $m_j$.

The equations of unit in $N$-ary Tree-LSTM $N$-ary $\text{Unit}_{(W^*, U^*, b^*)}$ for a node $j$ are as following:

$$\begin{aligned}
i_j &= \sigma(W_0^i h_{\beta_{j,0}} + W_1^i h_{\beta_{j,1}} + U_0^i h_{\alpha_{j,0}} + U_1^i h_{\alpha_{j,1}} + b^i) \\
f_{j,0} &= \sigma(W_0^f h_{\beta_{j,0}} + W_1^f h_{\beta_{j,1}} + U_{0,0}^f h_{\alpha_{j,0}} + U_{0,1}^f h_{\alpha_{j,1}} + b^f) \\
f_{j,1} &= \sigma(W_0^f h_{\beta_{j,0}} + W_1^f h_{\beta_{j,1}} + U_{1,0}^f h_{\alpha_{j,0}} + U_{1,1}^f h_{\alpha_{j,1}} + b^f) \\
o_j &= \sigma(W_0^o h_{\beta_{j,0}} + W_1^o h_{\beta_{j,1}} + U_0^o h_{\alpha_{j,0}} + U_1^o h_{\alpha_{j,1}} + b^o) \\
u_j &= tanh(W_0^u h_{\beta_{j,0}} + W_1^u h_{\beta_{j,1}} + U_0^u h_{\alpha_{j,0}} + U_1^u h_{\alpha_{j,1}} + b^u) \\
m_j &= i_j \odot u_j + f_{j,0} \odot m_{\alpha_{j,0}} + f_{j,1} \odot m_{\alpha_{j,1}} \\
h_j &= o_j \odot tanh(m_j)
\end{aligned}$$

where $i, f, o, u$ are the intermediate neural network in the calculation, corresponding to each gate which is proved to help catch the long-term information (deep tree or long sequence) in the traditional LSTM. $W_p^*, U_{0,p}^*, U_{1,p}^*, b_p^*, * \in \{i, f, o, u\}, p \in \{0, 1\}$ are parameters corresponding to gates and node's positions in neural network that will be trained to minimize the given loss function.

We have defined $N$-aryUnit which can calculate the representation of a node. We can use the $N$-aryUnit to get the representation of this join tree. Benefit from the dynamic graph feature of PyTorch, we can construct the computation graph of the neural network after we obtain the tree. We use Depth First Search (DFS) to traverse the tree and generate the computation graph as shown in Function JoinTreeDFS(). The $h_j$ of the root node in each join tree can be viewed as the representation $R(\mathcal{T})$ of this join tree $\mathcal{T}$, $R(\mathcal{T}) = h_j$.

*3) **Representation for Join State**:* The join state $s$ is composed of forest $\mathcal{F} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots\}$ and query $q$.

For a query $q$ with $m$ tables to be joined, we have $m-l$ join trees after $l$ joins. For those join trees $\mathcal{T}$ that have not been joined, they are unordered and any two tables can be joined in the next time. To represent the combination of these join trees, we use a root node to represent the forest whose children are all these $m$-$l$ join trees $\mathcal{T}_i$. Join trees are position independent, the Child-Sum Tree-LSTM is used here to get the output of root $h^{root}$. The equations of unit in Child-Sum Tree-LSTM $CSUnit_{(U^*,b^*)}$ for root are as the following:

$$h = \sum_k h_{\mathcal{T}_k}$$
$$i = \sigma(U^i h + b^i)$$
$$f_k = \sigma(U^f h_k^{root} + b^f)$$
$$o = \sigma(U^o h + b^o)$$
$$u = tanh(U^u h + b^u)$$
$$m^{root} = i \odot u + \sum f_k \odot m_{\mathcal{T}_k}$$
$$h^{root} = o \odot tanh(m^{root})$$

where $i, f, o, u$ are the intermediate neural network in the calculation, corresponding to each gate in the traditional LSTM. The parameters $U^*, b^*, * \in \{i, f, o, u\}$ in equations are parameters in neural network that need to be trained. They are other parameters in $CSUnit$ different from $N$-aryUnit. The representation for root $h^{root}$ can be viewed as the representation of forest $R(\mathcal{F}) = h^{root}$. After we have the representation for forest $\mathcal{F}$ and query $q$, we concatenate these two representations to get a representation for join state $s$,

$$R(s) = R(\mathcal{F}) \oplus R(q)$$

$R(\mathcal{F})$ and $R(q)$ are vectors with shape $(1, hs)$ and $R(s)$ is a vector with shape $(1, hs * 2)$.

## IV. Deep Reinforcement Learning in RTOS

After mapping JOS into an RL problem and describing the presentation for join states, our next goal is to solve this problem and give a join plan with a small cost (latency), for which we use Deep Q Network (DQN) [23].

### A. DQN for Join Order Selection

**Q-learning** [39] is a value-based RL algorithm. Similiar to DP, Q-learning uses a Q-table to store the best value for all states. Given several possible actions, it enumerates all actions and then picks the best one. For JOS, the action space $\mathbb{A}(s) = \{(A_1, s_1), (A_2, s_2), \ldots\}$ corresponds to all possible join conditions $\theta_i$ that $A_i = \theta_i$ and $s_i$ denotes the new state that the current state $s$ transfers to after taking action $A_i$. We define Q-values $Q(s)$ from function as the expected smallest cost (latency) of the optimal join plan from a join state $s$. We can get the formula of $Q$ as:

$$Q(s) = \begin{cases} cost(latency), s \text{ is a termination state} \\ \min\limits_{(A',s')\in\mathbb{A}(s)} Q(s'), s \text{ is a non-terminating state} \end{cases}$$

Similar to dynamic programming, Q-learning uses a Q-table to compute Q($s$). Q-table uses a list to record all known state-value pair $(s, Q(s))$. For asking $Q(s)$ of $s$, it will return $Q(s)$ directly if $s$ is in Q-table, or recursively ask Q-table $Q(s')$ from its action space $\mathbb{A}(s)$ to compute $Q(s)$ according to the above equations otherwise.

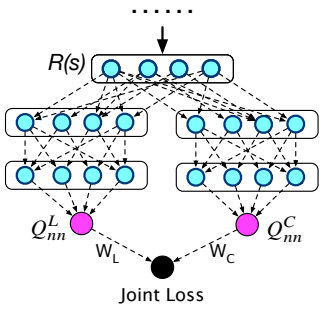


Fig. 6. Q-network of latency and cost share the same state representation.

The *policy* function $\pi(s)$ is used to return the selected optimal *action* for given state $s$. Once we get the Q-table, we can use it to guide the strategy function. The policy function $\pi(s)$ can be obtained as $\pi(s) = \arg\min_{(A',s')\in\mathbb{A}(s)} Q(s')$

However, as dynamic programming, Q-learning needs a large Q-table, which consumes large memory and costs too much time to enumerate all states.

**DQN** improves Q-learning by using a neural network (Q-network) to estimate the unseen state from the known state instead of using Q-table. We have obtained the join state representation $R(s)$ for state $s$ in Section III-C3. We use $Q_{nn}(R(s), \omega)$ to denote the Q-network, where $\omega$ is the parameters in Q-network.

One *episode* means a complete process that RTOS generates a join plan and get a feedback $v$. For each state $s$ in this episode, similar to Q-learning, $Q_{nn}(R(s), \omega)$ has a target value $V = \min(v, \min_{(A',s')\in\mathbb{A}(s)} Q_{nn}(R(s'), \omega))$ . The difference between target value of $Q_{nn}(R(s), \omega)$, $\delta(R(s)) = Q_{nn}(R(s), \omega) - V$ could indicate how accurate the Q-network can estimate the state $s$, which is closer to zero means a better estimation. We use the L2 loss to define the loss function for the Q-network which needs to be minimized to train the Q-network: $\mathsf{Loss}(R(s)) = (\delta(R(s)))^2$.

### B. Multi-task Learning for the Joint Loss of Cost and Latency

In RTOS, one issue that needs to be considered is the choice of cost and latency. We hope that RL can give the shortest execution time, but each step takes too long to get feedback. The cost model can quickly give an estimated cost but may be inaccurate. As mentioned in Section II-A, we divide the training process into two steps: **cost training** and **latency tuning**. When doing latency tuning, we do not directly move the target from cost to latency as used in previous work.

Instead, we consider them as two similar tasks and train neural network together, to benefit from the efficiency of cost and use latency for fine tuning, when large new training data is needed (e.g., modifications of database schema). This idea comes from multi-tasking learning [2], which shares the representation of the neural network and gets the output of multiple tasks at the same time. In this RTOS, we consider two Q-network $Q^L_{nn}, Q^C_{nn}$, which get the Q-value of latency $Q^L_{nn}$ and cost $Q^C_{nn}$, respectively.

During plan generation, $Q^L_{nn}$ is used to generate the join plan. After one episode (one plan generation process) is completed, we collect both latency $L$ and cost $C$. For one state $s$ in this episode, two Q-network will compute with the same representation for join state $s$. As shown in the Figure 6, we get the representation $R(s)$ of state $s$, then we calculate $Q^L_{nn}(s)$ and $Q^C_{nn}(s)$ separately with their own output layer. The losses of these two Q-network are:

$$\mathsf{Loss}_L(R(s)) = (\min(L, \min_{(A',s')} Q^L_{nn}(R(s'),\omega^L)) - Q^L_{nn}(R(s),\omega^L))^2$$
$$\mathsf{Loss}_C(R(s)) = (\min(C, \min_{(A',s')} Q^C_{nn}(R(s'),\omega^C)) - Q^C_{nn}(R(s),\omega^C))^2$$

We compute a weighted sum of the two loss functions to get the loss of the entire network.

$$\mathsf{JointLoss}(R(s)) = W_L \mathsf{Loss}_L(R(s)) + W_C \mathsf{Loss}_C(R(s))$$

We set the $W_L = 0, W_C = 1$ during **cost training**, which can be seen that we do not use the latency and set $W_L = 1, W_C = 1$ during **latency tuning**. Cost is used to train the network and latency is used to find the plan.

## V. Handling Modifications and Multi-Aliases

One major difference between learning methods and traditional methods is that learning methods need to be trained for a long time before being applied to real systems. Previously learned optimizers use a fix-length vector that is related to the number of tables and columns in database, which makes them hard to handle the following (frequent) cases:

- **Adding columns** to a table
- **Inserting tables** to a database
- **Multi-aliases:** a table may have multiple aliases in a query, such as "Select $*$ from $T$ as $t_1$, $T$ as $t_2$ ...", where $t_1$ and $t_2$ are aliases which can be treated as different tables with the same data in this SQL query.

Adding columns and inserting tables change the schema of database. Fix-length feature based neural networks need to change the size of network and retrain all over again. This process takes a long time and make the system fails during this time. What's more, multi-aliases can be viewed as an "inserting table" at any time that is really hard to handle. ReJoin [20] defines one more redundant position for each table and treats $t_1$ and $t_2$ as two totally different tables. The information of $t_1$ and $t_2$ cannot be shared during training and this will fail when we have $t_3$ for the reason that we cannot know how many aliases will occur in advance.

Below we will show how our model supports these operations using the dynamic feature and shared weight feature in `RTOS`. Without otherwise specified, `RTOS` uses the model below by default.

### A. Variable-length Query Encoding

In Section III-A we use a $n*n$ matrix to represent the query. When a table is inserted, the $n$ will be $n+1$ which makes the fully connect layer fail. We explore the representation of query encoding. For a given query $q$

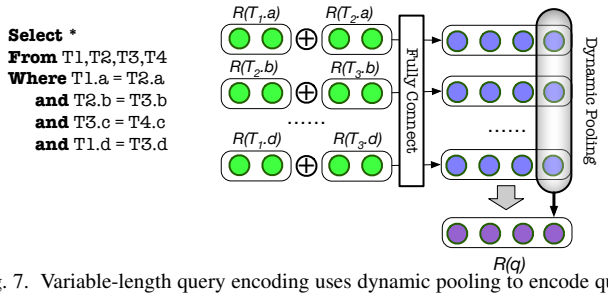



Fig. 7. Variable-length query encoding uses dynamic pooling to encode query with any number of join conditions in a query.

$$R(q) = \sigma(vW + b) = \sigma(\sum_{V_{i*n+j}=1} W_{i*n+j} + b)$$
$$= \sigma(\sum_{i\text{-th table join } j\text{-th table}} W_{i*n+j} + b)$$

We can know that FC layer uses a vector $W_{i*n+j}$ to represent each join condition. $R(q)$ is the sum of those vectors of which join is in the query. Join set $\Theta(q) = \{\theta_1, \theta_2, \ldots, \theta_k\}$ is the set of all $k$ join conditions $\theta_i$ in the query $q$. A join condition is composed of two columns, $\theta_i = (c_{i,0}, c_{i,1})$. We apply a fully connect (FC) layer on these two column's representation $R(c_{i,0}), R(c_{i,1})$ to construct the join condition representation $R(\theta_i) = \mathsf{FC}(R(c_{i,0}), R(c_{i,1}))$. Then we represent $R(q)$ by replacing $W$:

$$R(q) = \sigma(\sum_{\theta_i \in \Theta} \mathsf{FC}(R(c_{i,0}), R(c_{i,1})) + b)$$
$$= \sigma(R(\theta_1) + R(\theta_2) + \ldots + R(\theta_k) + b)$$

Our goal is to transfer these $k$ $(1, hs)$-vectors $R(\theta_i)$ (for $i \in [1, k]$) into one $(1, hs)$-vector while $k$ is a dynamic parameter according to different queries. Dynamic pooling [33] is proposed to solve this problem. Dynamic pooling is a pooling layer whose kernel size will be decided according to the expected output feature size. As shown in Figure 7, we use the dynamic average pooling over these $k$ vectors $R(\theta_i)$ and get one vector with shape $(1, hs)$ to represent $R(q)$.

$$R(q) = \sigma(\mathsf{DynamicPool}(R(\theta_1), R(\theta_2), \ldots, R(\theta_k)))$$

where $k$ is the number of joins in the query and independent of the number $n$ of tables in the database.

### B. Adding Columns and Inserting Tables

**Adding a column** will change the size of a table. When we want to add a column $c_{new}$ into table $t$. We first assigned attribute matrix $M(c_{new})$ for $c_{new}$ in neural network and then use $F(c_{new})$ to construct $R(c_{new})$. In Section III-B we have used the $k$ column representations $R(c_i)$ to construct table representation $R(t)$. Adding a column will change $k$ to $k+1$, so we switch simple pooling to dynamic pooling method to get the $R(t)$.

$$R(t) = \mathsf{DynamicPool}(R(c_1) \oplus R(c_2) \ldots \oplus R(c_k) \oplus R(c_{new}))$$

We learn the $M(c_{new})$ for queries related to $c_{new}$. For other queries have no $c_{new}$, the $R(c_{new})$ remains zero and keep $R(t)$ the same as before.

**Adding a table** $t_{new}$ will change the size of database from $n$ to $n+1$. We first assigned corresponding attribute matrix $M(t_{new}.c_i)$ for all columns $c_i$ in $t_{new}$ to construct $R(t_{new})$. Benefit from dynamic feature of Tree-LSTM, the neural network in Tree-LSTM are built by DFS when we get the join tree. Any node can be added to the join tree at any time. For those queries related to $t_{new}$, we directly add $R(t_{new})$ to the join tree and update the relevant parameters through Tree-LSTM.

**Weight initialization** is an important task to reduce the training time which is known as transfer learning [29]. We can initialize the parameter of the added column (table) using previously known similar column (table). Here we simply initialize the attribute matrix of added column who has join relations with previous column.

$$M(c_{new}) = M(c')$$

where $c'$ has a join relation with $c_{new}$ (e.g., a foreign key).

### C. Multi-aliases

Multi aliases can be viewed as inserting copy tables $(t_1, t_2)$ from the origin table $t$ when we execute the query. The copy tables share the same data as the origin one. What's more important is that when we learn the feedback using RL. The feedback to aliases $t_1$ and $t_2$ will both share parameters of $t$ in neural network.

Our idea is to use the shared information. $t_1, t_2$ will share the neural network parameters of $t$. For feature vector of column $c_i$ in $t_1$ and $t_2$, we use attribute matrix of $t$ construct their column representation.

$$R(t_1.c_i) = F(t_1.c_i) * M(t.c_i)$$
$$R(t_2.c_i) = F(t_2.c_i) * M(t.c_i)$$

We can construct table representation $R(t_1(t_2))$ with those $R(t_1(t_2).c_i)$. Like the operation "insert a table", We add these $R(t_1(t_2))$ and $R(t_1(t_2).c_i)$ to join tree and construct the neural network when needed. They feedback information of $t_1, t_2$ will work on parameters of $t$ due to the shared mechanism.

## VI. Experiments

The chief purpose of our experiments is to understand the performance of DRL-based methods and traditional methods for the problem of join order selection. We build `RTOS` on PostgreSQL.

**Datasets:** We conducted experiments on two datasets:

(1) *Join Order Benchmark (JOB) [17].* JOB is a real-world dataset based on IMDB to provide realistic workload. It has 113 queries from 33 templates. It has 3.6GB data (11GB when counting indexes) and 21 tables. The number of relations in each query ranges from 4 to 17.

(2) *TPC-H [30].* TPC-H is a standard industry database benchmark with 8 tables. We have generated 4GB data and 110 queries from 22 templates.

For each dataset, we split all queries into 10 folds and choose 9 as training set and the other 1 as test set.

**Algorithms.** For `RTOS`, by default, we used the final at Section V. `RTOS` is implemented on PostgreSQL and improves the join order part of a query plan and leaves the others (e.g., access methods such as hash join or index join) for PostgreSQL to decide. We compared with the following methods.

**- DQ** uses three one-dimensional vectors to represent the selectivity of columns, join state, and projections, respectively. It also uses the deep Q-network and the model is a two-layer fully connected network.

**- ReJoin** uses two $n*n$ matrices to represent the query and the join state, respectively, and a one-hot vector to represent the column existence in the query. It uses PPO [31], another DRL technique. The model is a two-layer fully connected network.

**- SkinnerDB [37]** is the latest method that chooses a good join order according to the RL policy. We compare with the join order finally generated by Skinner-C.

**- QP100 (1000)**: Given a query, it randomly tests 100 (1000) possible plans and chooses the plan with the lowest cost.

**- Dynamic-program (DP)**: Given a query, it enumerates all the join plans using dynamic programming, and chooses the plan with the lowest cost. We utilize the implementation in the PostgreSQL to test this method.

All methods conduct PK-FK joins. We tune the parameters of DQ and ReJoin for better performance. We set the hidden size as $hs = 128$, and take Adam [12] with learning rate of $3e-4$ as optimizer for the model. 2 hours are used on cost training and 10 hours are used on latency tuning.

**Training Sample Collection.** To explore the learning ability of the model from feedback (both good and bad samples), all DRL methods were randomly-initialized and learned from the feedback only. DQ pre-trains its model by collecting samples from dynamic programming and making model act as dynamic programming when the number of join is small, and does exploration when the number is large which needs complicated manual intervention. We can hardly know that the model just acts as dynamic programming (good samples only) or it can learn from feedback (good and bad samples). What's more, during latency tuning, only latency was used as feedback to improve the model which DP cannot provide. To be fair, all DRL methods here collected training samples by getting feedback only from the DBMS.

**Latency Collection.** We first got the latency $L_{DP}(q)$ and cost $C_{DP}(q)$ of DP's plan for each query $q$. To avoid wasting unnecessary time, we limited the execution time to 5 times of latency by DP's plan $5 * L_{DP}(q)$ and recorded the latency of plan. Single core was used when collecting latency.

**Planning Time.** DRL methods typically use polynomial time, e.g., $O(n^2)$ for ReJoin and $O(n^3)$ for DQ, depending on the calculation of the neural network as a constant. `RTOS` is also $O(n^3)$ but has a larger constant. The average time to obtain the representation of a join tree for queries in JOB is around 5ms. The planning time of the largest query (17 relations) in JOB is $213ms$ (`RTOS`), $97ms$ (DQ), and $63ms$ (ReJoin). For all 113 queries in JOB, the total planning time of `RTOS` is

TABLE II
MEAN RELEVANT COST TO DYNAMIC PROGRAMMING.

| MRC: benchmark / algorithm | JOB | TPC-H |
|---|---|---|
| RTOS | **1.01** | **1.00** |
| ReJoin | 1.75 | **1.00** |
| QP100 | 7.81 | 1.06 |
| QP1000 | 1.62 | **1.00** |
| DQ | 2.34 (1.31) | 1.01 |

$8s$, which is about 1% of the total latency $712s$ of DP's plan Hence, planning time will not be discussed in the following.

### A. Cost Training

We first evaluated the **cost training** which uses cost as feedback to guide RTOS. We choose DP as the baseline and reported the mean relative cost (MRC) of other methods as previous work, where MRC $= 1$ means the same cost as DP.

$$\text{MRC} = \frac{\sum_{q \in Q} \frac{cost(q)}{cost_{DP}(q)}}{|Q|}$$

From Table II, we can see that RTOS outperforms the other 4 methods and performs as good as DP. We did not report the cost result of Skinner-C here, because it had no cost model and only used latency to generate the join order. In TPC-H, all methods can get near optimal plan. After further investigating TPC-H queries, we found these queries are typically short (do not exceed 8 relations) which limit the search space. Hence, join order is not a major problem in TPC-H. QP1000 can even enumerate the search space and get the equal plan of DP. Even QP100 achieves MRC of 1.06. In JOB, the MRC gap between different methods is obvious for the reason that the queries in JOB are large (up to 17 relations), so it is harder to enumerate the search space. The MRC of RTOS (1.01) still outperforms the other two DRL methods (1.75 and 2.34) and QP1000 (1.62). The MRC of RTOS even exceeds the reported value 1.31 in DQ (pre-trained from good samples of dynamic programming). The MRC of RTOS 1.01 indicates the competitive performance of RTOS even on estimated cost.

Figure 8 depicts a training curve on JOB. We can see that all DRL methods performed bad at the beginning, and much better after enough training. We can see that RTOS went even beyond QP1000 after ~8000 episodes. The results show that RTOS not only benefits from the random exploration in DRL, but also learns the join feature of the database. The training curves of ReJoin and DQ vibrated up and down and cannot converge to a good one. One major reason is that these two methods project two different plans into the same representation. Neural network outputs the right value for one plan which makes the other fail. This conflict makes neural network difficult to converge to a real better plan for all queries.

### B. Latency Tuning

After **cost training**, we have got a neural network that has a understanding of data distribution and can generate a plan with low cost. From this trained model, we further used latency as feedback to fine-tune to get a plan with lower latency.

**GMRL.** We also chose DP as our baseline. Note that DRL-based methods generate better plans than DP after latency

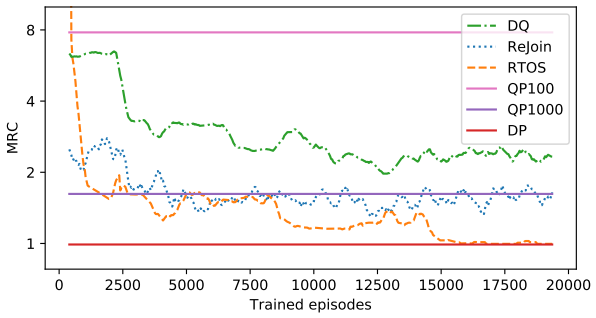



Fig. 8. The training curve on JOB. We train RTOS on 90% queries and the curve show the performance of other 10% queries. One episode means one time that DRL method generates a plan and gets feedback.

TABLE III
EXPONENTIAL MEAN LOG RELEVANT LATENCY (GMRL) TO DP

| GMRL: benchmark / algorithm | JOB | TPC-H |
|---|---|---|
| RTOS | **0.67** | **0.92** |
| ReJoin | 1.14 | 0.96 |
| QP100 | NA | 1.03 |
| QP1000 | 1.90 | 1.00 |
| Skinner-C | 0.89 | 1.03 |
| DQ | 1.23 | 0.99 |

tuning, which makes MRC unsuitable to capture the relative performance to DP. For example, given two queries, if a DRL method generates join plans with twice (2) and half (0.5) latency, compared with DP, the relative performance should be 1, but MRL will give $\frac{2+0.5}{2} = 1.25$. Hence, we use geometric average based Geometric Mean Relevant Latency (GMRL).

$$\text{GMRL} = (\Pi_{i=1}^{n} \frac{\text{Lantency}(q_i)}{\text{Lantency}_{DP}(q_i)})^{\frac{1}{n}}$$

GMRL for 2 and 0.5 will be $\sqrt{2 * 0.5} = 1$ which can indicate the performance improvement in ratio directly. We report GMRL as average performance ratio of DP.

Table III shows the GMRL of all methods. We can see that RTOS outperforms other methods on these two benchmarks. GMRL (0.67 on JOB and 0.92 on TPC-H) is lower than 1 means that we can get better plan than DP on latency. We did not give the results of QP100 on JOB because it gave bad plan for some queries which took long time to execute. For TPC-H, all DRL methods can get lower GMRL than DP which is different from the cost results in Table II. This result shows the inaccuracy of cost model. QP1000 performs the same as DP. For JOB, the GMRL of RTOS is smaller than all other methods. 0.67 Of RTOS means an improvement of 37% compared with DP. We can see that Skinner-C outperforms DQ, ReJoin and DP but is still worse than RTOS, because Skinner-C does not learn from previous queries and relies on online RL model to learn different join orders. Skinner-C measures a join order by executing it in a small time slice, which may not be accurate in the entire execution process.

We further grouped queries by their template id. Figure 10 shows the GMRL on different templates for TPC-H, we do not show the templates that DRL methods perform the same as DP to have a better view. We can see that for all templates RTOS can generate the plan no worse than DP (the horizontal

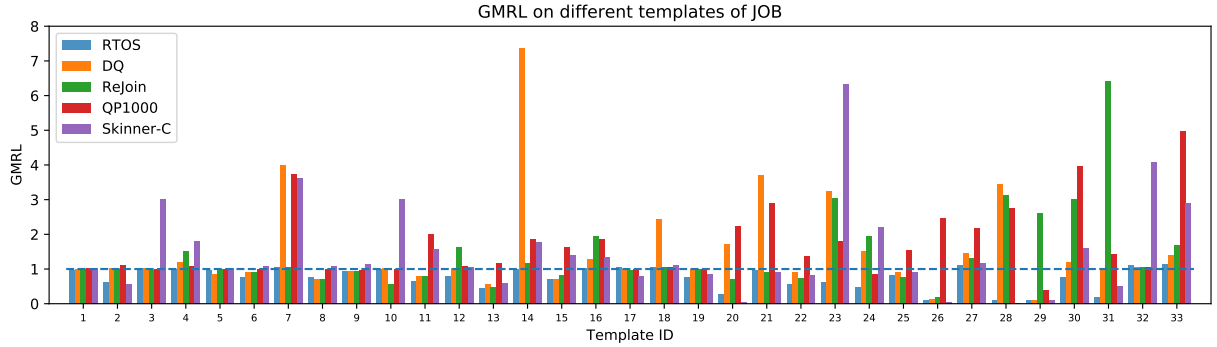



Fig. 9. GMRL on different templates in JOB, 1 is for DP and the lower the better.

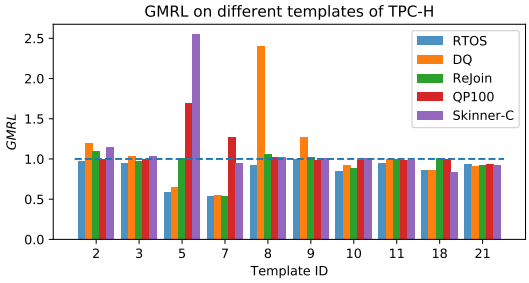



Fig. 10. GMRL on various TPC-H templates, 1 is for DP, the lower the better.

line at GMRL= 1) and better than other methods. Figure 9 shows the results for JOB. We can see that for all templates RTOS can almost generate the plan no worse than DP. One interesting point is that Skinner-C and RTOS are obviously better than DP on template T20,T26, T28, T29 and T31 which have *multi-aliases* in these queries, because RTOS can handle multi-aliases and Skinner-C on-the-fly learns the order. The traditional cost model failed on this case. For Rejoin and DQ, they can generate better plans than other methods on queries in some templates (e.g., T10 of Rejoin and T5 of DQ). However, they will yield bad plans on other queries which might suffer from weakness in representations.

### C. Recovery From Schema Changes

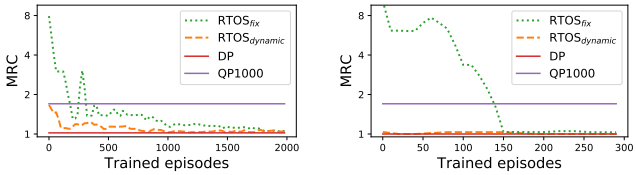
Schema changes make RTOS fail to generate plans for these added tables or columns at the beginning. We investigate the time (episode) of our methods taken for recovery from the fail state to perform similar as DP on cost.

We compared with two version of RTOS:

- $\texttt{RTOS}_{fix}$ uses the model described in Section III with fixed size representation. It builds a new neural network with new size of database and retrain the network.
- $\texttt{RTOS}_{dynamic}$ is the final version that uses the model described in Section V which tolerates the schema change and uses the known column to initialize the neural network of the added column.

*1) Insert Table:* We choose a table **char_name** in JOB and split all 113 queries into training set and update set according to whether the query contains this table or not. We first trained $\texttt{RTOS}_{dynamic}$ on training set (no table char_name) and then gave the queries in update set to $\texttt{RTOS}_{dynamic}$ and $\texttt{RTOS}_{fix}$ as if the table was newly added.

Figure 11(a) shows the training (recovery) curve of queries. $\texttt{RTOS}_{dynamic}$ quickly outperforms QP1000 at the beginning.




(a) Add a table (b) Add a column

Fig. 11. training episodes when database schema changes.

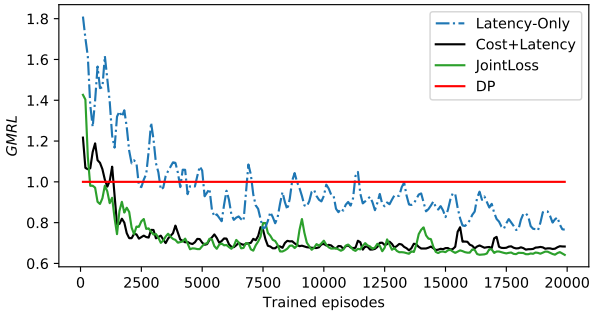



Fig. 12. The training curve when using latency as the feedback. The cost training process is not counted.

After 750 episodes the $\texttt{RTOS}_{dynamic}$ can generate competitive plan with DP because it contains the knowledge of other 20 tables in JOB. $\texttt{RTOS}_{fix}$ takes about 2000 episodes to get similar performance on update set which cost 2.5 times episodes. The new trained $\texttt{RTOS}_{fix}$ only contains the information from update set and totally trained from all queries in JOB will take 20000 episodes (about 2 hours) which is quite long.

*2) Add a column:* We choose a column **char_name.name** and then split all 113 queries into training set and update set.

From the curve in Figure 11(b) we can see that when the columns were added into the table, $\texttt{RTOS}_{dynamic}$ recovered very quickly even as no modification happens. The reason is that column (char_name.name) only occurs as a predicate. It does not have complex relationships with other tables or columns. Its selectivity is the most important information which is easier to be learned compared with join information. Join plan is not always sensitive to this column.

### D. Effectiveness of Cost

Feedback from latency (even seconds) costs more time than feedback from cost model (1ms). Using latency as feedback makes the training take a long time.

Here we explore the effectiveness of cost training. We use the following setting of RTOS to explore it:

TABLE IV
TIME OF DIFFERENT PART FOR TRAINING 20000 EPISODES ON LATENCY.

| | nueral network | get latency | get cost |
|---|---|---|---|
| time used(h) | 1.71 | 10.27 | 0.05 |

TABLE V
TIME USED TO GET COMPETITIVE PERFORMANCE OF DP.

| $S_i$ | | cost training | | latency tuning | | total time(h) |
|---|---|---|---|---|---|---|
| | | episode | time(h) | episode | time(h) | |
| 60 queries | Latency-Only | 0 | 0.0 | 5000 | 3.61 | 3.61 |
| | JointLoss | 10000 | 0.88 | 1500 | 1.27 | **2.15** |
| add a table | Latency-Only | 0 | 0.0 | 700 | 0.63 | 0.63 |
| | JointLoss | 700 | 0.06 | 450 | 0.45 | **0.51** |

- **Latency-Only**: RTOS will be trained from initial state and only use latency as feedback.
- **Cost+Latency**: Cost training based method. We first used cost as feedback (cost training with 10000 episodes with about 1 hour) and then changed the target of network from cost to latency (latency tuning).
- **JointLoss**: Cost training based method. When doing latency tuning, we received both latency and cost to train the neural network, as mentioned in Section IV-B.

Totally using latency as feedback will take a long time to train the network. We selected 60 queries in JOB with lowest latency (from 10ms to 2 seconds) as the query set to test the training process of above methods.

Figure 12 shows the training curve of all these methods when using latency as feedback. Cost training based methods (JointLoss and Cost+Latency) only used 1500 episodes to be competitive to Latency-Only that took about 5000 episodes. **Cost training** will reduce many episodes when switching feedback from cost to latency. Table IV shows the time consumed of different parts when using latency as feedback. Time used to get the latency after the plan executed is quite longer than getting the cost. **Getting latency** took $\frac{10.27}{10.27+1.71+0.05} = 85.4\%$ time of **latency tuning** for one episode. Table V shows that cost training will increase the number of episodes but reduce the total time. cost training based methods Only takes $\frac{2.15}{3.61} = 59.8\%$ time to train the RTOS to get competitive performance of DP when compared with Latency-only method.

JointLoss and Cost+Latency perform almost the same, which means we can keep the cost information and latency information in neural network without affecting the model performance. We switch to cost training and then back to latency tuning when adding a table. Table V shows that JointLoss uses less time (0.51 vs 0.63) when adding a table **char_name.name** on latency tuning.

### *E. Overall Training Time*

Figure 13 shows the GMRL in both cost training and latency tuning stage with the total training time increasing for JOB queries. The cost training stage first takes about 1.5 hours (15000 episodes) and then latency tuning takes 10 hours (8000 episodes). We can see that during the cost training stage, the GMRL of RTOS fluctuates around one because RTOS focuses on optimizing the cost in the stage but the cost model of PostgreSQL is not accurate enough. When we switch to the latency tuning stage, we can see that RTOS generates much better plan because it focuses on optimizing the latency and the GMRL finally converges to a value much smaller than one.

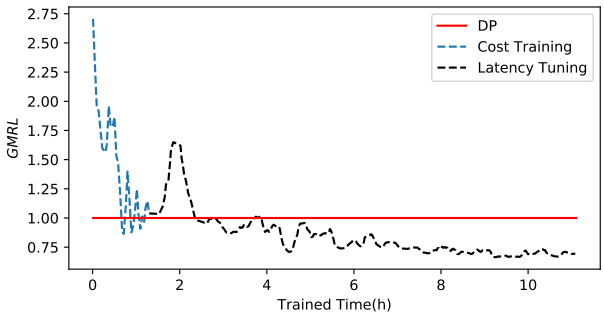


Fig. 13. GMRL in both cost training and latency tuning stage with the total training time increasing.

## VII. RELATED WORK

**Reinforcement learning for Join Order Selection.** The most similar works to ours are SkinnerDB [37], ReJoin [20] and DQ [16]. SkinnerDB [37] tries to use RL to select the join order during query processing. It executes some plans in a time slice by running the plan on a set of samples, and uses the performance to measure the plans. It relies on its customized memory execution engine to support the high-frequency switching of the join order and is not easy to integrate existing DBMSs. ReJoin [20] and DQ [16] use DRL with a preliminary neural network model to learn the join process from a given workload. RTOS is different from these two methods by effectively modeling structural information of join trees using Tree-LSTM and supporting database updates. The results show that RTOS outperforms ReJoin, DQ, and SkinnerDB. The work [27] tries to learn the entire plan generation process through DRL. The idea of DRL can help the optimizer learn from both cost and latency, and generate good join plan in a polynomial time complexity. RTOS focuses on join order selection which is more difficult to represent by a neural network, when compared with index selection or physical plan selection (usually encoded as one-hot vectors [16]). NEO [19] uses a value neural network to search the plan with low latency. NEO first encodes each node in a join tree into a feature vector and then uses Tree Convolution Network [24] to get the representation of join tree. Different from NEO, our RTOS can support database updates efficiently. Moreover, RTOS can not only estimate latency but the cost, while NEO focuses on estimating the latency. SageDB [15] presents a vision that machine learning (ML) can optimize database by modeling the data distribution, workload, and hardware.

**Cardinality Estimation.** Traditional cardinality estimation methods can be classified into three types: histogram-based [9], sketching-based [5], and sample-based methods [18]. Recently, the database community is investigating cardinality estimation techniques by leveraging deep neural network. [14] trained a multi-set convolutional network on queries and [28] proposed a vision of training representation for the

join tree with reinforcement learning. [34] proposed a tree-structured model to learn the representation of query plans.

RTOS is orthogonal to the cardinality estimation methods. Accurate selectivity input to RTOS can help do latency tuning and even good neural network-based methods can be directly ported to our model (e.g. column representation) in RTOS.

**Deep Learning for Databases.** Researchers have used DL on many database problems like entity matching [25], query optimizer [16] cardinality estimation [13]. This topics demonstrate the potential of DL. Due to the powerful representation ability of graph neural networks (GNNs) [40], GNNs achieve success in many topics [6], [7]. We apply Tree-LSTM, which is also a GNNs and proposed to represent the join tree.

## VIII. Conclusions and Future Work

We have proposed RTOS which uses Tree-LSTM to learn the tree-structure join plan. The results show that our method can generate good plans both on cost and latency on two benchmarks. It proves that our method can learn structure of join tree properly and catch the information of join operations. We also prove that the cost can pre-train the neural network and reduce the latency tuning time.

Naturally, there are many future works. (1) It is interesting to study transfer learning to adapt a model trained on a database to another database. (2) The latency might increase when the system in a high workload (e.g. multi-queries executing at the same time) for the factors like I/O bottleneck, memory size. We need to consider more information of workload if we want the system to learn the feedback from complex scenarios.

## References


[1] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *SIGMOD*, 2005.

[2] R. Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[3] S. V. Chande and M. Sinha. Genetic optimization for the join ordering problem of database queries. In *2011 Annual IEEE India Conference*, pages 1–5. IEEE, 2011.

[4] L. Fegaras. A new heuristic for optimizing large queries. In *International Conference on Database and Expert Systems Applications*, 1998.

[5] P. Flajolet, ric Fusy, O. Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.

[6] T. Hamaguchi, H. Oiwa, M. Shimbo, and Y. Matsumoto. Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach. *arXiv preprint arXiv:1706.05674*, 2017.

[7] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pages 1024–1034, 2017.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] Y. E. Ioannidis. The history of histograms (abridged). In *PVLDB*, pages 19–30, 2003.

[10] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *ACM SIGMOD Record*, 20(2):168–177, 1991.

[11] T. Kaftan, M. Balazinska, A. Cheung, and J. Gehrke. Cuttlefish: A lightweight primitive for adaptive query processing. *arXiv preprint arXiv:1802.09180*, 2018.

[12] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[13] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[14] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[15] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[16] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[17] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *VLDB*, 2015.

[18] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[19] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[20] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, page 3. ACM, 2018.

[21] R. Marcus and O. Papaemmanouil. Towards a hands-free query optimizer through deep learning. *arXiv preprint arXiv:1809.10212*, 2018.

[22] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv:1301.3781*, 2013.

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

[24] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[25] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *SIGMOD*, 2018.

[26] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, 1990.

[27] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. *arXiv preprint arXiv:1803.08604*, 2018.

[28] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *DEEM@SIGMOD*, pages 4:1–4:4, 2018.

[29] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[30] M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. *ACM Sigmod Record*, 29(4):64–71, 2000.

[31] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[32] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[33] R. Socher, E. H. Huang, J. Pennin, C. D. Manning, and A. Y. Ng. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Advances in neural information processing systems*, 2011.

[34] J. Sun and G. Li. An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560*, 2019.

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

[36] I. Trummer and C. Koch. Solving the join ordering problem via mixed integer linear programming. In *SIGMOD*, 2017.

[37] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *SIGMOD*, 2019.

[38] F. Waas and A. Pellenkoft. Join order selection (good enough is easy). In *British National Conference on Databases*, 2000.

[39] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[40] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.