LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans

Tianyi Chen
Key Laboratory of High Confidence Software Technologies, CS, Peking University, China
tianyichen@stu.pku.edu.cn

Jun Gao*
Key Laboratory of High Confidence Software Technologies, CS, Peking University, China
gaojun@pku.edu.cn

Hedui Chen
ZTE Corporation, China
chen.hedui@zte.com.cn

Yaofeng Tu*
ZTE Corporation, China
tu.yaofeng@zte.com.cn

ABSTRACT

Query optimization based on deep reinforcement learning (DRL) has become a hot research topic recently. Despite the achieved promising progress, DRL optimizers still face great challenges of robustly producing efficient plans, due to the vast search space for both join order and operator selection and the highly varying execution latency taken as the feedback signal. In this paper, we propose LOGER, a **l**earned **o**ptimizer towards **g**enerating **e**fficient and **r**obust plans, aiming at producing both efficient join orders and operators. LOGER first utilizes Graph Transformer to capture relationships between tables and predicates. Then, the search space is reorganized, in which LOGER learns to restrict specific operators instead of directly selecting one for each join, while utilizing DBMS built-in optimizer to select physical operators under the restrictions. Such a strategy exploits expert knowledge to improve the robustness of plan generation while offering sufficient plan search flexibility. Furthermore, LOGER introduces ϵ-beam search, which keeps multiple search paths that preserve promising plans while performing guided exploration. Finally, LOGER introduces a loss function with reward weighting to further enhance performance robustness by reducing the fluctuation caused by poor operators, and log transformation to compress the range of rewards. We conduct experiments on Join Order Benchmark (JOB), TPC-DS and Stack Overflow, and demonstrate that LOGER can achieve a performance better than existing learned query optimizers, with a 2.07x speedup on JOB compared with PostgreSQL.

PVLDB Reference Format:
Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans. PVLDB, 16(7): 1777 - 1789, 2023.
doi:10.14778/3587136.3587150

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/TianyiChen0316/LOGER.

*Corresponding authors

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587150

1 INTRODUCTION

Query optimization has long been critical in the database field for its difficulty and importance in query execution performance. To find an efficient execution plan for each query, query optimizers have to search in an extremely vast search space, in which we know that finding optimal join order is an NP-hard problem [7]. In addition, the performance is affected by the selection of physical operators, leading the problem to be more complicated. Hence at present, almost all relational database management systems apply heuristic methods and various strategies to balance between query optimization complexity and execution latency. With great effort in development, these traditional query optimization methods achieve stable performance under different circumstances.

However, traditional optimizers still face challenges as data distribution and queries can be highly complicated in real applications, and it's impossible to evaluate all possible plans. Thus, these optimizers rely on simple strategies and assumptions to make decisions and often produce fine but sub-optimal plans. In addition, these optimizers typically require years or decades to develop. When changes like running on new hardware are taken into consideration, traditional optimizers are difficult to adapt to different environments.

To overcome the limitations of traditional query optimizers, DRL-based optimizers have come into view in recent years [12, 16, 17, 28, 29]. With DRL, plan generation is naturally converted to a sequence decision problem in the plan search space by considering each join as an action. The DBMS serves as the environment of DRL, replying reward feedback, *i.e.* execution latency, when a complete plan is generated. These DRL query optimizers then learn from the rewards to generate efficient plans on current data distribution. The success of these learned optimizers demonstrates the feasibility of DRL query optimization, showing that after sufficient training, learned optimizers can significantly outperform traditional methods under specific workloads and environments, on both open-source DBMSs like PostgreSQL and commercial DBMSs.

Although learned optimizers achieve promising competitive performance, these methods are still in their early stage. DRL optimizers face the same vast search space as traditional methods. To reach high and robust performance, proper strategies are required to obtain promising plans. In addition, DRL optimizers are trained with execution latency, which varies largely with both join order and operators, leading stable training of robust query optimization to further be difficult. Thus, the method design should consider the factors impacting the robustness of plan generation. We summarize the requirements for a successful learned optimizer as follows.

First, expressive query representation is a pre-requirement for learned optimizers. Various previous methods [16, 17, 28] represent each query as a join graph, in which each node represents a table and each edge is a join predicate. To utilize the graph structure, the model should exploit sufficient information in node representations, and exchange them along edges to extract relationships between tables and predicates. However, these previous methods exploit the join graph by simply processing the adjacency matrix as a flattened vector with a network like MLP, which is inefficient to extract information through graph structure. A more expressive model for query representation is needed for better performance.

Second, it is crucial to robustly search for efficient plans from an extremely large search space. A number of existing learned query optimizers [17, 29] reduce the difficulty of finding efficient plans by handling searching for only join orders, leaving the choice of physical operators to built-in optimizer of DBMS, which provides fewer chances to achieve higher performance. Others [16, 28] attempt to generate plans with both join order and physical operators to reach potentially better performance, but the search difficulty is significantly higher as the search space is further enlarged. To robustly obtain efficient plans from such a space that contains both join order and operators, not only a search method that produces promising plans is required, but it's also worth discussing how to leverage traditional optimizers to reduce search difficulty.

Third, a strategy is required to deal with scattered and highly fluctuating rewards, which have a severe impact on the robustness of learned optimizers. An inefficient plan can be tens or even thousands of times more costly than a better one, leading the latency to be sparse within an extremely wide range. Meanwhile, the interaction between join order and physical operators makes the latency vary drastically. Plans with identical join order but different join operators can have distinct performances, resulting in a large variance of reward feedback in reinforcement learning, and further increasing difficulty for robust plan generation. To solve the problem, a simple timeout mechanism is proposed to limit feedback on poor plans to a predefined value [28]. However, more flexible methods are necessary for robust performance.

In order to provide a practical idea for overcoming the aforementioned problems, we propose LOGER, a **l**earned **o**ptimizer towards **g**enerating **e**fficient and **r**obust plans, aiming at producing both efficient join orders and operators.

To improve representation expressiveness, we apply Graph Transformer (GT) [5], a graph neural network (GNN) model, to capture information from tables and predicates in the join graph. With GT, LOGER can not only sufficiently exchange information between adjacent table nodes, but also integrate structural information of the graph into each node, thus reflecting relationships between tables and predicates in table representations.

In order to utilize expert knowledge to improve the robustness of plan generation, we propose *Restricted Operator Search Space (ROSS)*, which combines the advantages of both learned and traditional optimizers to select operators along with join order selection. ROSS uses a *restricted operator* to forbid the use of a specific physical operator instead of directly selecting an operator for each join, and invokes DBMS built-in optimizer to complete operator selection. The key insight of ROSS is to utilize DBMS optimizer to reduce plan search difficulty of learned optimizer, while providing adequate flexibility to avoid inefficient operator selection. To further improve search efficiency, we propose a new search method called *ϵ-beam search*, which takes steps in multiple paths simultaneously and selects the predicted best candidate paths with a wide vision. The proposed method adopts the idea of ϵ-greedy [26] algorithm to introduce exploration mechanism, which is further guided by value prediction to find the most potential plans, and adaptively balances between exploration and exploitation.

Finally, LOGER utilizes different strategies to lower the impact of fluctuating rewards on robustness during training. We propose *reward weighting* to stabilize reward values and make LOGER focus on learning efficient join order. By taking the weighted value of *operator-relevant* latency and *operator-irrelevant* latency as the reward, the fluctuation caused by previously selected poor operators is alleviated. In addition, we use *log transformation* to compress the scattered reward values of disastrous plans and let LOGER pay more attention to better plans by making their reward values more distinguishable from poor ones.

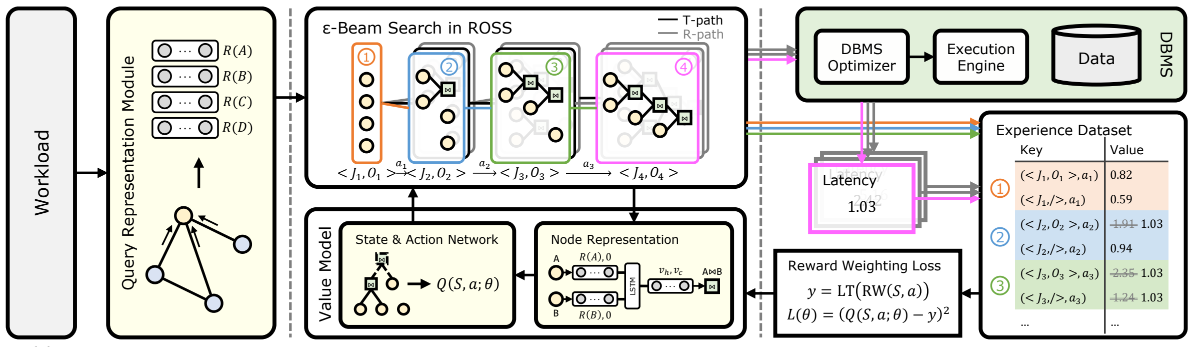
We tested LOGER's performance on Join Order Benchmark (JOB) [13], TPC-DS [19], and Stack Overflow [15] with PostgreSQL. Experiments show that on JOB testing workload, our method can outperform traditional optimizer after about 2 hours of training, and achieve a 2.07x speedup in the end. We further tested our method on a commercial DBMS and obtained similar results. Compared with prior state-of-the-art learned query optimizers, LOGER performs best and also has a low inference time.

2 METHOD FRAMEWORK

The framework of LOGER is shown in Figure 1. Similar to existing methods [12, 16, 28, 29], LOGER focuses on select-project-join (SPJ) statements without sub-queries and uses a value-based DRL framework, in which the qualities of intermediate plans (subplans) are evaluated to guide plan generation during search procedure. The latency of each generated plan then serves as feedback from the environment, utilized as the reward to train LOGER. In this section, we give a brief introduction to the value-based DRL method in LOGER and list three main procedures, while discussing similarities and differences between existing methods and LOGER.

Value-based DRL for query optimization. In reinforcement learning, an agent starts from an initial state, searches in the state-action space by taking actions according to the policy, and receives reward value from the environment, which is then used to improve performance by training. In value-based DRL methods, the policy is determined by a learned value network $V(S)$ or $Q(S,a)$ that estimates the largest cumulative future reward of states or state-action pairs. The agent evaluates all action candidates and decides between selecting the action with the largest predicted reward and exploring others, until reaching a terminal state.

DRL-based query optimization adopts the idea of DRL by converting the query optimization problem to the decision of join sequence for each query. An incomplete sequence, *i.e.* a subplan, is then regarded as a state, and the behavior of selecting the next join is an action. Similar to previous methods like Neo [16] and RTOS [29], we



(a) Query representation (b) Plan search (c) Model training

Figure 1: Framework overview for LOGER. The boxes in yellow contain learnable parameters, and different border and line colors represent data correlations of distinct state-action pairs.

consider the sequences as forests of join trees, each tree representing a single table or the result of joining multiple tables. Trees need to be combined to produce new trees until only one tree is left in the forest, indicating all tables are joined. Starting from a state where none of the tables in the query is joined, LOGER selects actions by estimating the lowest reachable latency of all action candidates. We note each subplan S as a pair (J, O) below, where J is a set of join trees indicating the join order and O is a set containing information of selected join operators. Taking a query with four tables A, B, C and D as an example, the initial state is $S_1 = (\{A, B, C, D\}, \emptyset)$, and state $S_2 = (\{A \bowtie B, C, D\}, \{A \bowtie_{NN} B\})$ is reached after the action of joining A and B with $\bowtie_{NN}$, where $\bowtie_{NN}$ is a restricted operator that will be explained later in detail. This process comes to an end when a complete join tree $((A \bowtie B) \bowtie C) \bowtie D$ is obtained, as all tables are joined together. After this process, the latency is obtained from the DBMS engine and used for model training.

Query representation. To capture information in the query for evaluating subplans, LOGER converts it into vectorized representations, which are used as the input of the value model, as is shown in Figure 1(a). In this step, the representation of each table is obtained by *query representation module*. While previous methods use simple networks like MLP which is inefficient in utilizing graph structure, LOGER exploits it by using GT to effectively extract information and capture relationships between tables and predicates.

Plan search. After query representation, LOGER sets up the initial subplans and generates plans step-by-step via DRL search procedure, shown in Figure 1(b). LOGER uses a *value model* to predict latency values for state-action pairs at each step with table representations and selects actions according to the predictions. To fairly evaluate plans of both slow and fast queries, the value model predicts relative execution latency. Given a query, its relative latency l is calculated by the following formula, where $T_{\text{exe}}(S)$ is the execution latency of the plan S generated by LOGER, and $T_{\text{exe}}(S_b)$ is for the plan S_b generated by DBMS optimizer.

$$l = \frac{T_{\text{exe}}(S)}{T_{\text{exe}}(S_b)}$$

Different from existing work, LOGER reorganizes the search space using restricted operators to utilize expert knowledge of traditional optimizer and applies ϵ*-beam search* to obtain potentially better plans for training with balanced exploration and exploitation. In Figure 1, the subplans ① - ④ are sequentially generated along with other subplans shown behind them, selected by ϵ-beam search with either exploration or exploitation. The branch nodes in each subplan represent selections of restricted operators.

Model training. After plan generation, the generated plans are executed to obtain latency values, which are then recorded in *experience dataset*. Experience dataset contains a hash lookup table T that records the reached lowest latency for each state-action pair $((J, O), a)$. As incomplete subplans like ① - ③ in Figure 1 cannot be evaluated alone, their table values $T((J, O), a)$ are updated with the latency of the corresponding final plan like ④. The loss function is then used to update the parameters of value model and query representation module. Different from previous work, we propose *reward weighting* to reduce the impact of previous poor operators and make LOGER pay less attention to disastrous plans by *log transformation*. To perform reward weighting, another table value $T((J, /), a)$ is also recorded, which will be explained later.

In the following sections, we describe each procedure in detail. We list the frequently used symbols in Table 1.

Table 1: Frequently used notations in the paper.

Notation	Description
S	A subplan that is represented as a pair (J, O).
J	A set of join trees in a subplan.
O	A set about operator selections in a subplan.
a	An action of joining tables with a specified operator.
$T_{\text{exe}}(S)$	The execution latency of a complete plan S.
$C(S, a)$	The *reachable* lowest latency of S after action a.
$T(S, a)$	The recorded *reached* lowest latency of pair (S, a).
ϵ	The exploration probability of ϵ-beam search.

3 QUERY REPRESENTATION

In this section, we first introduce the construction of join graphs, and then demonstrate how LOGER extracts information and obtains table representations through *query representation module*.

3.1 Join Graph Construction

As is shown in Figure 2(a) and 2(b), a select-project-join query can be naturally transformed into a join graph, in which each node represents a table and each edge is a join predicate. LOGER integrates other information into node attributes, including knowledge of the corresponding tables, columns, and predicates involved in the query, supported by statistical information collected by DBMS. The node attributes are further utilized by query representation module to generate node representations.

LOGER uses a table-level learned embedding vector R_t for each table to integrate the knowledge of the table into node representation. For example, the learned embedding vector of table A is denoted as $R_t(A)$. The values of R_t are trained to capture information for different tables in the database.

LOGER uses a column-level statistic vector R_s of size 13 for each column to utilize statistical information of columns, containing (1) the value type of the column, described as a 3-bit one-hot vector indicating integer, float, and non-numeric, respectively; (2) the proportion of unique values, described as 3 bits indicating interval $[0, 0.001)$, $[0.001, 0.999]$ and $(0.999, 1]$; (3) the prediction of whether unique value count will increase according to collected statistics,[1] described as 2 bits indicating true or false; (4) the proportion of null values, described as 3 bits indicating interval $[0, 0.001)$, $[0.001, 0.999]$ and $(0.999, 1]$; and (5) whether the column has indexes, described as 2 bits.

LOGER uses a column-level predicate vector R_p for each column involved in the query to represent information of predicates. Each R_p is a vector of size 3, denoted as $[r_{\bowtie}, r_s, r_{1-s}]$. We take R_p of column $A.a1$ as an example below.

$$R_p(A.a1) = [r_{\bowtie}(A.a1), r_s(A.a1), r_{1-s}(A.a1)]$$

$r_{\bowtie}$ indicates whether a corresponding join predicate exists. For example, since the join predicate $A.a1 = B.b1$ presents in the query, $r_{\bowtie}(A.a1)$ is set to 1. If not, $r_{\bowtie}$ is set to 0.

r_s represents the selectivity of the single-table predicate on the column. When such a predicate does not exist, r_s is set to 0. Otherwise, r_s is set to the negative logarithm of the selectivity, with the purpose of distinguishing differences. For example, the selectivities of two predicates are 10^{-4} and 10^{-5} respectively, which are close in value, yet the results of two predicates applying to the table have a 10x difference. By employing logarithms, these small values can be distinctly identified by the model. Assuming that the selectivity of predicate $A.x1$ >= 100 is 0.1, the value of $r_s(A.x1)$ is calculated as follows.

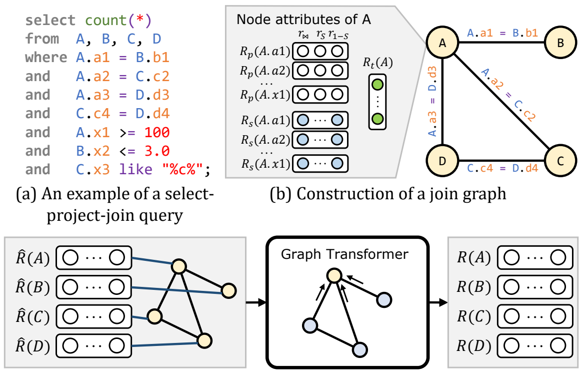
$$r_s(A.x1) = -\ln \mathrm{Sel}[A.x1 >= 100] = -\ln 0.1 \approx 2.30$$

r_{1-s} represents the selectivity of the inverse predicate on the column, whose purpose is to lower the sensitivity issue of negative logarithm in handling large values. Taking the above example, the value of $r_{1-s}(A.x1)$ is calculated by the selectivity of $A.x1 < 100$.

$$r_{1-s}(A.x1) = -\ln \mathrm{Sel}[A.x1 < 100] = -\ln 0.9 \approx 0.11$$

LOGER utilizes the approximation of the DBMS cardinality estimator to estimate the selectivity of each predicate, as exact selectivities are not practical to obtain. When there are multiple single-table predicates on a column, LOGER uses only the predicate with the smallest selectivity and discards others to avoid underestimation caused by distribution assumptions. For the calculation of logarithms above, all input values are restricted to a lower bound 10^{-9}.

[1]In PostgreSQL, the prediction can be obtained through pg_stats.



```
select count(*)
from  A, B, C, D
where A.a1 = B.b1
and   A.a2 = C.c2
and   A.a3 = D.d3
and   C.c4 = D.d4
and   A.x1 >= 100
and   B.x2 <= 3.0
and   C.x3 like "%c%";
```

(a) An example of a select-project-join query

(b) Construction of a join graph

(c) Node representations generated by Graph Transformer

Figure 2: Steps of query representation.

3.2 Query Representation Module

Query representation module produces vectorized representation for nodes in the join graph, which are provided for plan generation.

Initial representation. After table-level embedding vectors R_t, column-level predicate vectors R_p and statistic vectors R_s are obtained, query representation module combines them to get initial node representations $\hat{R}$. A distinct matrix M_c for each table is learned to combine R_s and $R_p = [r_{\bowtie}, r_s, r_{1-s}]$ together into column representation R_c, following the equations below, where W_1, W_2 and b are learned matrices and vectors. $\hat{M}_c(A.a1)$ is an intermediate result split into vectors $M_{\bowtie}$, M_s and M_{1-s} with the same size.

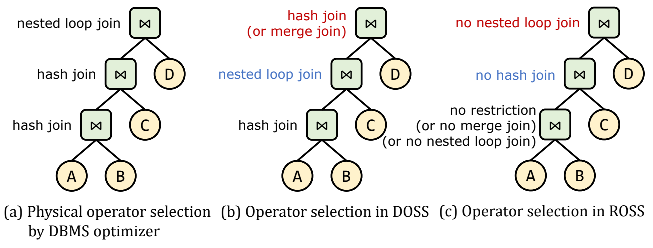
$$\begin{aligned}\hat{M}_c(A.a1) &= \mathrm{ReLU}\left(R_s(A.a1) W_1 M_c(A)\right) W_2 + b \\ &= [M_{\bowtie} \,||\, M_s \,||\, M_{1-s}] \\ R_c(A.a1) &= [r_{\bowtie} M_{\bowtie} \,||\, r_s M_s \,||\, r_{1-s} M_{1-s}]\end{aligned}$$

Query representation module forms each table's column-level representation by applying pooling to column representations. We use max-pooling because a predicate with a smaller selectivity, which has a higher r_s, should be concerned more than larger ones. Considering the example in Figure 2(a), column-level representation $R_c(A)$ is formed by aggregating $R_c(A.a1)$, $R_c(A.a2)$, $R_c(A.a3)$ and $R_c(A.x1)$ together. The initial node representation $\hat{R}(A)$ is then calculated as follows, where W_3 is a learned matrix.

$$R_c(A) = \mathrm{MaxPooling}\left\{R_c(A.a1), R_c(A.a2), R_c(A.a3), R_c(A.x1)\right\}$$

$$\hat{R}(A) = \left([R_c(A) \,||\, R_t(A)]\right) W_3$$

Final representation. To exchange information in table node representations along graph structure, query representation module applies Graph Transformer [5] to the join graph, as is demonstrated in Figure 2(c). With the attention mechanism of Transformer [25] and Laplacian positional encodings, GT not only efficiently captures relationships between table representations that include both table and predicate information, but also integrates global structural information of join graph into table representations.



(a) Physical operator selection by DBMS optimizer (b) Operator selection in DOSS (c) Operator selection in ROSS

Figure 3: An example of operator selection in DOSS and ROSS. Choices inferior to the DBMS optimizer are highlighted in red, and better choices are highlighted in blue.

Taking the join graph along with initial node representations $\hat{R}(\cdot)$ as input, the final representations $R(\cdot)$ of all tables in the query are calculated by a GT layer, and are then utilized by the value model for latency prediction during plan search.

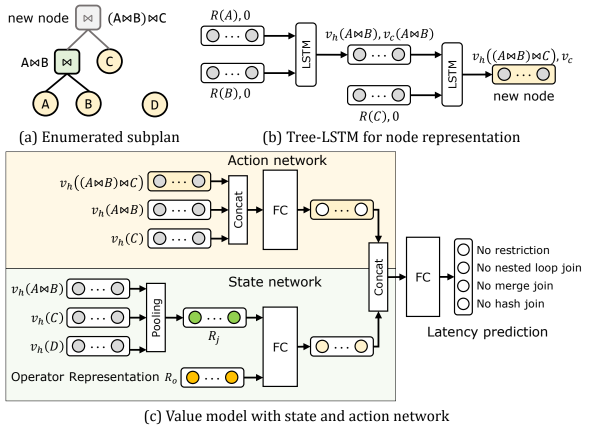
4 PLAN SEARCH

In this section, we introduce the idea of *ROSS*, explain the approach to evaluate state-action pair candidates with the value model, and describe how LOGER performs efficient search with adaptive exploration and exploitation by *ϵ-beam search*.

4.1 Restricted Operator Search Space

LOGER utilizes knowledge of DBMS optimizer to improve the robustness of plan generation. As is adopted by various previous works [12, 16, 28], a plan search space contains both join order and physical operators. In such a space, physical operators totally depend on the learned optimizer to select, which has the potential to obtain efficient plans along with high risk to generate poor ones. Therefore, we reorganize it and propose *Restricted Operator Search Space (ROSS)*, in which the learned optimizer selects join order and *restricted operator* in each step, and physical operators are determined by both restricted operators and choices of DBMS optimizer to reduce the risk of poor plans. In LOGER, ROSS allows four types of restricted operators to forbid specific physical operator selections: no restriction ($\bowtie$), no nested loop join ($\bowtie_{NN}$), no merge join ($\bowtie_{NM}$) and no hash join ($\bowtie_{NH}$). Correspondingly, the search space in which physical operators are directly selected is called *Direct Operator Search Space (DOSS)*.

Figure 3 shows an example of operator selection in both search spaces, from which we can find that restricted operators reduce the difficulty of efficient plan search when the DBMS optimizer selects optimal physical operators. The example in Figure 3(b) shows that in DOSS, the physical operator has to be exactly specified to obtain equal or higher performance than the DBMS optimizer. The difficulty of selecting the best one out of 3 choices may lead to failure and therefore poor performance. In contrast, LOGER only needs to select among 3 choices out of 4 in ROSS when the choice of DBMS optimizer is correct. In addition, ROSS keeps the potential to select better operators than the DBMS optimizer by disabling specific physical operators. Figure 3(c) shows that it's equally efficient to select among three choices at the first step, which is apparently easier to learn. In the second step, a $\bowtie_{NH}$ operator that forbids DBMS optimizer to use hash join can be selected for a better plan.



(a) Enumerated subplan (b) Tree-LSTM for node representation

(c) Value model with state and action network

Figure 4: The process of state-action evaluation.

4.2 Value Model for State-Action Evaluation

The value model evaluates state-action pair candidates (S, a) of subplans S by predicting the reachable lowest latency, *i.e.* the lowest latency among all plans that can be reached from (S, a), denoted as $C(S, a)$ below. The information of a subplan includes join order and operators, which together determine the performance. Thus LOGER respectively obtains the representation of join order and operators to provide input for value model.

Candidate generation. The state-action pair candidates (S, a) of each state $S = (J, O)$ are generated by enumerating all possible joins. We adopt the strategy of System R [22] in this process, which avoids Cartesian products when there exist two join trees in J that can perform a conditional join. Figure 4(a) shows an example of a candidate, in which the join of two trees $A \bowtie B$ and C is enumerated. These candidates are then evaluated through the following procedures.

Join order representation. Inspired by RTOS [29], LOGER utilizes Tree-LSTM [24] to vectorize join order J for each plan $S = (J, O)$. As a variant of LSTM [6], Tree-LSTM is adapted to tree-structured data to capture bottom-up sequential information. For each join tree in J, we use the final representations $R(\cdot)$ yielded by query representation module as the hidden vectors v_h of leaf nodes, and cell vectors v_c of the leaf nodes are set to 0. We apply N-ary Tree-LSTM to calculate hidden and cell vectors for branch nodes in the tree. In the example shown in Figure 4(b), the hidden vector and cell vector of node $(A \bowtie B) \bowtie C$ are calculated as follows.

$$\begin{aligned}&[v_h((A \bowtie B) \bowtie C), v_c((A \bowtie B) \bowtie C)]\\ &= \text{TreeLSTM}([v_h(A \bowtie B), v_c(A \bowtie B)], [R(C), 0])\end{aligned}$$

Operator representation. The operator representation R_o is a combination of operator representations on all tables concerned in the query. Each table has a list of four learned embedding vectors $M_o = [o_{\bowtie}, o_{\bowtie_{NN}}, o_{\bowtie_{NM}}, o_{\bowtie_{NH}}]$, representing operator $\bowtie$, $\bowtie_{NN}$, $\bowtie_{NM}$ and $\bowtie_{NH}$ respectively. When a table is joined, its operator representation is selected from M_o according to the used restricted operator. For example, $o_{\bowtie_{NN}}(A)$ is taken as operator representation $R_o(A)$ for table A when it is joined with B by operator $\bowtie_{NN}$. For tables whose restricted operators are not determined, their R_o are set to 0. The operator representation $R_o(O)$ of a subplan $S = (J, O)$ is the mean R_o of all tables in the query (denoted as a set $\mathcal{T}$).

$$R_o(O) = \frac{1}{|\mathcal{T}|} \sum_{\tau \in \mathcal{T}} R_o(\tau)$$

State-action pair evaluation. The value model predicts latency values for each state S with all its enumerated joins. As is shown in Figure 4(c), the value model includes two sub-networks called action network and state network. The state network handles information about subplan S, and the action network takes the information of enumerated join as input. The outputs of two sub-networks are combined to produce predictions of $C(S, a)$ for all four actions a that share the same join but with different restricted operators.

The input of state network includes hidden vectors of all root nodes in join tree set J, along with the operator representation $R_o(O)$. Through mean-pooling, the root node representations are aggregated into join representation vector R_j. For the above example, R_j is obtained by the following equation.

$$R_j(J) = \frac{1}{|J|} \sum_{\tau \in J} v_h(\tau) = \frac{1}{3} \left(v_h(A \bowtie B) + v_h(C) + v_h(D) \right)$$

For the input of action network, LOGER calculates the hidden vector of the produced new node specified by the enumerated join. Taking the case shown in Figure 4(c) as an example, when $A \bowtie B$ and C are joined together, the hidden vector of $(A \bowtie B) \bowtie C$ is calculated. Then the new hidden vector is inputted along with v_h of the two joined nodes into action network. Value model combines the output of state network and action network by a fully connected (FC) layer to produce a vector of size 4, each element representing $C(S, a)$ for an action a with a specific restricted operator.

4.3 ϵ-Beam Search with Adaptive Exploration

Starting from an initial subplan in which none of the tables is joined, LOGER generates plans step-by-step in ROSS with predictions of the value model. To select promising subplans in the vast search space, we propose *ϵ-beam search*, a variant of beam search with adaptive exploration. Classic algorithms including ϵ-greedy [26], UCB [2], Boltzmann exploration [9] and Thompson sampling [20] perform linear search, which preserves only one search path, denoted as a list of sequentially chosen state-action pairs $[(S_1, a_1), (S_2, a_2), \ldots]$. In contrast, beam search simultaneously searches on multiple paths to seek globally better results. In addition, to introduce exploration, ϵ-beam search performs random selection following the idea of ϵ-greedy algorithm, which is further guided by value model to select potential actions. Besides, we apply an adaptive method to balance between exploration and exploitation.

Combination of beam search and ϵ-greedy. ϵ-beam search preserves K search paths on each step, where K is a constant. The search paths are categorized into explo$\underline{\text{r}}$ation paths (R-paths) and exploi$\underline{\text{t}}$ation paths (T-paths). R-paths refer to the search paths in which at least one state is obtained by random pick, and T-paths indicate that states in the search path are always obtained by selecting the best ones according to predictions of value model. Before each step, ϵ-beam search determines the number of preserved R-paths and T-paths in the next step. At least one R-path and one T-path

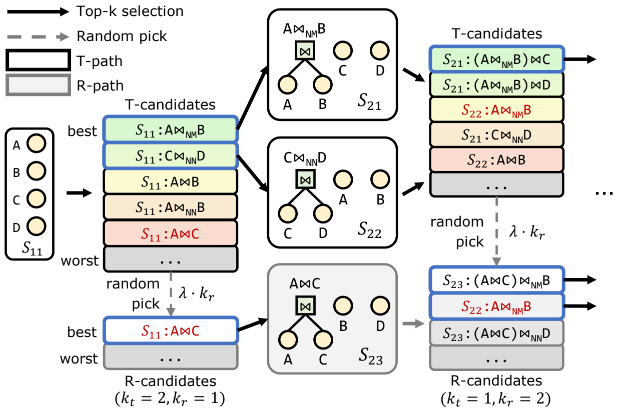


Figure 5: An example of ϵ-beam search. Solid lines represent top-k selection procedure, and dashed lines represent random picking from T-candidates to R-candidates.

are kept, and each of the rest $K - 2$ paths becomes an R-path with exploration probability ϵ or otherwise a T-path. In other words, denoting the number of R-paths as k_r and T-paths as k_t, $k_r - 1$ follows binomial distribution $B(K - 2, \epsilon)$.

Figure 5 presents an example of ϵ-beam search. When k_r and k_t are determined, ϵ-beam search categorizes all enumerated state-action pair candidates from current paths into R-candidates and T-candidates. T-candidates are the pairs from current T-paths, and the top k_t pairs are selected as the next T-paths. In the example, S_{11} produces the next T-paths by selecting the best 2 candidates. R-candidates are from two sources, including randomly picked $\lambda \times k_r$ of the remaining T-candidates and all candidates from current R-paths, where λ is a constant factor. S_{21} and S_{22} produce T-candidates in the second step, and after deciding one T-path by top-1 selection, 4 T-candidates are randomly put into R-candidates along with the pairs produced by S_{23}. The subsequent R-paths are then decided by top-k_r selection of R-candidates.

This example shows the advantage of ϵ-beam search over both ϵ-greedy and the original beam search. Top-k_t selection in the second step compares candidates of different subplans S_{21} and S_{22}, providing a wide vision of the search space. In contrast, ϵ-greedy only compares locally among candidates of one state. Moreover, ϵ-beam search selects the exploration paths by choosing the predicted best R-candidates instead of the random decision in ϵ-greedy, which ensures that LOGER explores promising plans in the extremely large space. The original beam search is equivalent to ϵ-beam search when the number of R-paths $k_r = 0$, which will always select identical paths after convergence.

Adaptive exploration with probability adjustment. To balance between exploration and exploitation, exploration probability ϵ should be initially large to sufficiently explore the search space and gradually decrease during training. In addition, different queries require distinct exploration probabilities. When a query is poorly optimized with large relative latency, it requires more exploration until reaching a fine performance. Increasing explorations of queries with a long execution time also help improve the throughput of the workload. LOGER utilizes a decay period N_{half} for ϵ. Starting from an initial value ϵ_i, the distance between ϵ and a constant final value ϵ_f is half decreased after generating plans for N_{half} queries. Meanwhile, $\epsilon(q_j)$ of each query q_j in workload Q is increased according to a value $\alpha(q_j)$ that varies with performance. The formulas are as follows, where n is the number of iterated queries during training.

$$\epsilon_b = \epsilon_i + (\epsilon_f - \epsilon_i)(1 - 0.5^{n/N_{\text{half}}})$$
$$\epsilon(q_j) = \epsilon_b + (1 - \epsilon_b)(1 - e^{-\lambda_p \alpha(q_j)})$$
$$\alpha(q_j) = \gamma_p \alpha_{\text{rel}}(q_j) + (1 - \gamma_p)\alpha_{\text{abs}}(q_j)$$

The value of α is composed of two parts α_{rel} and α_{abs}. α_{rel} shows whether the plan generated by LOGER is relatively poor compared with the DBMS optimizer, obtained as follows, where l_j is the relative latency of q_j and $\bar{l}$ is the geometric mean of l_j.

$$\alpha_{\text{rel}}(q_j) = \frac{\hat{\alpha}_{\text{rel}}(q_j)}{\sum_{k=1}^{|Q|} \hat{\alpha}_{\text{rel}}(q_k)}, \ \hat{\alpha}_{\text{rel}}(q_j) = \max\{l_j - \bar{l}, 0\}$$

$\alpha_{\text{abs}}(q_j)$ shows whether q_j is one of the queries that require the longest time to execute, with the plan generated by LOGER. It is set to $1/\lceil \gamma_a |Q| \rceil$ if q_j is one of the slowest queries within a ratio γ_a, or otherwise 0.

We set $K = 4$, $\lambda = 2$, $\epsilon_i = 0.8$, $\epsilon_f = 0.2$, $N_{\text{half}} = 200$, $\lambda_p = 0.5$, $\gamma_a = 0.125$ and $\gamma_p = 0.6$ in our experiments. During testing, exploration is disabled to stabilize performance by setting $k_r = 0$.

5 MODEL TRAINING

In this section, we first introduce *experience dataset*, the data structure that stores rewards from execution, and then describe the loss function of LOGER, which applies *reward weighting* and *log transformation* to reduce variation caused by poor operators and handle rewards of disastrous plans respectively.

5.1 Experience Dataset

LOGER trains the value model to predict the *reachable* lowest latency $C(S, a)$ for each state-action pair (S, a). Since obtaining real $C(S, a)$ is a costly process that requires exhausting the search space, LOGER stores the *reached* lowest latency $T(S, a)$ as the approximation of $C(S, a)$ into *experience dataset*. When a latency value l is obtained from the DBMS engine, $T(S, a)$ of all state-action pairs in the corresponding search path are updated, and the stored values are used to train the agent after each iteration. Denoting S as (J, O), when an old value of $T((J, O), a)$ does not exist, the value is initialized with l. Otherwise, $T((J, O), a)$ is updated only when l is smaller than the original value, as is described in formula (1).

$$T((J, O), a) := \min\{T((J, O), a), l\} \tag{1}$$

To perform reward weighting in the loss function, another table value $T((J, /), a)$ is also updated through formula (1), where operator information O is replaced with a placeholder. $T((J, /), a)$ represents the minimum latency of all recorded pairs $((J, O'), a)$ that has the same join order as $S = (J, O)$, named as *operator-irrelevant* latency. The value of table entry $((J, O), a)$ is called *operator-relevant* latency correspondingly.

The data in the experience dataset is not directly sampled for training. Following the settings of Deep Q-learning [18], LOGER keeps a replay memory, which is a queue with a limited capacity that stores state-action pairs (S, a) discovered during search. During training, mini-batches of state-action pairs are sampled from the queue, with their ground truth values obtained from T. We set the capacity to 4000 during training. When the capacity is exceeded, the head items are discarded to make room for new items.

DRL methods typically face a cold-start problem due to insufficient knowledge for selecting actions of high quality initially, which has to be alleviated by effective strategies [4]. LOGER introduces initial knowledge by taking the DBMS optimizer as an expert model to generate plans for each query in training workload, and stores corresponding state-action pairs into experience dataset at the beginning of training. Those experiences are then sampled from the queue to train LOGER with expert knowledge.

5.2 Reward Weighting Loss with Log Transformation

Since the execution latency is greatly influenced by both join order and operators, directly learning to predict reached lowest latency $T(S, a)$ can be extremely difficult, leading to issues on robust plan generation. We list two major difficulties in detail below.

First, inefficient plans can result in disastrous latency values, which cause great interference in model training. A value of relative latency within the interval $(0, 1)$ shows that the plan is better than the DBMS optimizer's plan. However, an improper plan can have a large relative latency within $[1, +\infty)$. Such plans lead the latency values to be scattered in a wide range, significantly increasing the difficulty of value prediction.

Second, the interaction between join order and physical operators makes the latency vary drastically. The previously selected poor operators in a subplan can thus influence the learning of subsequent action selections. Consider an example in which two tables A and B are joined together with an efficient operator in subplan S_1 and a poor one in S_2. Although similar subplans S_1 and S_2 are followed by the same subsequent action a that joins $A \bowtie B$ and C, (S_2, a) finally receives a latency significantly larger than (S_1, a). This increases the difficulty in evaluating action a for value model, and easily causes underestimations of efficient actions.

To deal with the aforementioned difficulties of value prediction, we propose a loss function with reward weighting and log transformation, through which the relative latency is transformed to a value with a much smaller variance. Let θ denote all learnable parameters of LOGER. For the output $Q(S, a; \theta)$ of the value model, LOGER uses the following loss function $L(\theta)$, which will be explained below.

$$y = \text{LT}\left(\text{RW}\left(S, a\right)\right)$$
$$\text{RW}((J, O), a) = wT((J, O), a) + (1 - w)T((J, /), a) \tag{2}$$
$$\text{LT}(x) = \begin{cases} \ln(2 + \ln x) & x \geq 1 \\ \ln(1 + x) & 0 \leq x < 1 \end{cases} \tag{3}$$
$$L(\theta) = (Q(S, a; \theta) - y)^2$$

Reward weighting. To reduce the impact of poor operators in state $S = (J, O)$ on the evaluation of subsequent action a, we combine *operator-relevant* latency $T((J, O), a)$ and *operator-irrelevant* latency $T((J, /), a)$ by weighting with a factor w as is shown in formula (2). The weighted value is taken as the ground truth value instead of directly using $T((J, O), a)$. Reward weighting stabilizes the evaluation and prevents underestimation of action a by decreasing the influence of previous operators, with a stable value $T((J,/),a)$ that is irrelevant to operator information O. Through weighting, even if improper operators in S lead to poor performance with a high $T((J,O),a)$, the evaluation of a correct action a will not be largely affected, as the impact is reduced by taking $T((J,/),a)$ into consideration.

Reward weighting also encourages LOGER to search for better join orders in preference to operators, with the drawback of selecting sub-optimal plans. We set $w = 0.1$ to balance the trade-off. We will further demonstrate its effectiveness through experiments.

Log transformation. To reduce the impact of disastrous plans, the log transformation function $\mathrm{LT}(\cdot)$ in formula (3) is applied to the weighted reward values. The key insight of $\mathrm{LT}(\cdot)$ is to compress the scattered poor reward values for reducing model sensitivity to them and make LOGER focus more on distinguishing better plans. For a plan better than the baseline, $\mathrm{LT}(\cdot)$ maps the value to the interval $(0, 0.69)$. Meanwhile, it significantly compresses the range of values for poorer plans. For example, the values of poor plans scattered in a long range $(1, 100]$ are mapped to a much smaller interval $(0.69, 1.89]$.

6 EXPERIMENTS

In this section, we describe the settings of our experiments in detail, demonstrate the results of both LOGER and other competitors, and show the effectiveness of components through ablation study.

6.1 Experiment Setup

Datasets and workloads. We conduct experiments on three different datasets: Join Order Benchmark [13] on IMDB, TPC-DS [19] and Stack Overflow [15].

Join Order Benchmark (JOB) is a query workload established on IMDB, a real-world dataset. Different from synthetic datasets, IMDB is highly skewed and contains a large number of correlations, bringing great challenges to query optimization. We load IMDB dataset into the database and add all foreign keys. JOB consists of 113 queries from 33 different templates, each query including from 4 to 17 relations. To investigate LOGER's performance on all templates of JOB, the testing workload in the experiment contains 33 queries from 1a to 33a, with the remaining 80 queries as the training workload. Besides, following the split strategy of RTOS [29] and Balsa [28], we perform tests on Balsa's random split workload (noted as *JOB-RS* below), which samples 19 queries as the testing workload, to show more comprehensive results.

TPC-DS is a standard benchmark with data and query generator. We use a scale factor of 4 to generate data. Since most query templates of TPC-DS are not SPJ templates, which are supported by neither LOGER nor most competitors, we use only 20 simple templates out of 99 for experiments. We choose 5 templates as the testing workload and the rest as the training workload, ensuring that all concerned tables in testing workload appear in training workload.[2] 3 queries are generated for each template in both training and testing workload.

[2]The templates used as testing workload are 18, 27, 52, 82 and 98, and the templates in training workload are 3, 7, 12, 20, 26, 37, 42, 43, 50, 55, 62, 84, 91, 96 and 99.

Stack Overflow (noted as *Stack* below) is a large real-world dataset with 16 templates, each containing at least 100 queries with the same join graph but different predicates. We discard templates 9 and 10 in our experiments since they are not for SPJ queries. Templates 1, 3, 5, 12, 14 and 16 are used as testing workload, with the rest as training workload. We randomly pick 5 queries for each template in testing workload, and 10 queries for each in training workload.

Database and environment settings. We do experiments on PostgreSQL 13.5 with 64GB shared buffers. We disable GEQO and apply the *pg_hint_plan* plugin to enable hints for PostgreSQL's query optimizer to implement ROSS. We also use an anonymous commercial DBMS (noted as CommDB below) with its default settings to test the generalization capability of LOGER. We implement LOGER with Python 3.8 and Pytorch 1.8.0, and run experiments on a Ubuntu 20.04 server with two Intel(R) Xeon(R) Gold 2.30GHz CPUs, 256GB memory and an NVIDIA RTX 3090 GPU.

Training configuration. We train LOGER on training workload for 200 epochs in each experiment. In an epoch, we shuffle the queries in training workload and iteratively generate K plans for each query with ϵ-beam search. After that, a mini-batch of size 128 is randomly sampled from experience dataset to train the network. During training, we use Adam optimizer with an initial learning rate 3×10^{-4}, which is gradually decreased to 3×10^{-5} between the 50th and the 100th epoch, and 3×10^{-6} until the 200th epoch.

To improve training speed, we use a random beam size from 1 to 4 during the first 10 epochs. As the agent does not have sufficient knowledge initially, this strategy enables LOGER to perform quick exploration. After 10 epochs, the beam size is set to $K = 4$. Meanwhile, we cache the latency of plans to avoid repetitive execution. We also set a timeout limit on plan execution. When the execution latency of a plan exceeds the limit, the execution is terminated and LOGER uses an estimation instead of real latency $T_{\text{exe}}(\cdot)$. Denoting $\mathrm{cost}(\cdot)$ as the DBMS I/O cost model, the estimation is obtained through the following approximation formula.

$$T_{\text{exe}}(S) \approx \frac{\mathrm{cost}(S)}{\mathrm{cost}(S_b)} T_{\text{exe}}(S_b)$$

We use 4x the longest execution latency of training workload queries as the timeout limit in each experiment. Besides, we limit the plan tree structure as left-deep in experiments of JOB, JOB-RS and TPC-DS, which significantly reduces the search space while preserving the most promising plans. Left-deep restriction is fairly efficient as a left-deep plan can exploit existing indexes and is suitable for the execution pipeline. We do not use left-deep restriction on Stack as it requires bushy plans for some queries, but we disallow bushy plans in the first 4 epochs to stabilize the performance.

Metrics. We apply two major metrics to evaluate the performance.

Workload relative latency (WRL) shows the total execution latency on the entire workload. In query workload Q, letting S_{bi} and S_i represent the plan generated by DBMS optimizer and LOGER for query q_i respectively, WRL is calculated as follows. We can see that 1/WRL represents the speedup of workload latency, which is used as a metric in a number of previous methods [15, 28].

$$\mathrm{WRL}(Q) = \frac{\sum_{q_i \in Q} T_{\text{exe}}(S_i)}{\sum_{q_i \in Q} T_{\text{exe}}(S_{bi})}$$

Geometric mean relative latency (GMRL) [29] is the geometric mean of relative latency within the workload, calculated as follows.

$$\mathrm{GMRL}(Q) = \sqrt[|Q|]{\prod_{q_i \in Q} \frac{T_{\mathrm{exe}}(S_i)}{T_{\mathrm{exe}}(S_{bi})}}$$

From the formulas above, it can be seen that lower WRL and GMRL indicate higher optimization performance. WRL and GMRL evaluate the performance from distinct aspects. WRL demonstrates overall execution latency, which focuses more on slow queries. GMRL focuses on query-wise relative latency speedup and fairly evaluates both fast and slow queries, but is not able to measure total speedup on the workload.

Apart from these two major metrics, we show other details including model training time and the average inference time during testing for more comprehensive analyses. The metrics on both training and testing workload are evaluated every 4 epochs.

Comparison. We compare LOGER with the following prior state-of-the-art learned query optimizers, along with a simple baseline denoted as PG-NoMerge below.

PG-NoMerge improves the performance of PostgreSQL by disabling the merge join operator on the entire workload. We do not use disabling nested loop join or hash join as a baseline, since we find that it leads to a slowdown on JOB with no apparent advantage on other workloads compared with PG-NoMerge, which produces stabler performance.

RTOS [29] is a learned query optimizer that uses Tree-LSTM to vectorize subplans. RTOS focuses on join order selection only, leaving physical operator selection to the DBMS optimizer.

Bao [15] is a learned query optimizer that relies on the DBMS optimizer to generate plans. Different from most previous methods that select plans from the plan search space, Bao selects from a number of arms, each arm representing a set of global parameters that disables specific operators. Bao leverages the DBMS optimizer to generate a plan for each parameter set and selects the predicted best one. We configure Bao to select from 5 arms in our experiments.

Balsa [28] is a learned query optimizer that completely avoids the participation of the DBMS optimizer by bootstrapping from a simulator. Balsa further introduces a variant of beam search and applies a timeout mechanism to handle disastrous plans. Balsa provides two modes that require training 1 agent and 8 agents respectively. Since the time cost of training 1 agent is similar to that of LOGER, we perform experiments on the former mode.

Experimental design. We demonstrate LOGER's performance on both PostgreSQL and CommDB, analyze its per-query speedup and tail performance, and make a comparison with previous methods in terms of WRL, GMRL, training time and average inference time. An analysis for generalizing to the updated schema by incremental training is also shown in the experiments.

Furthermore, we verify the effects of components and design choices including (1) ϵ-beam search algorithm, (2) ROSS, (3) GT layer and pooling method in query representation module, (4) reward weighting and log transformation, (5) expert knowledge in experience dataset and (6) left-deep restriction, by ablation study. Through this process, we analyze the effects on a number of issues including cold-start ability, training stability and final performance.

6.2 Overall Performance and Comparison

Performance overview. Figure 6 shows the performance on different workloads on PostgreSQL, from which we can see that LOGER quickly outperforms PostgreSQL's optimizer and finally reaches a high performance on both JOB, TPC-DS and Stack. The solid lines in the figure indicate testing workload performances, and transparent lines are for training workload performances. LOGER outperforms PostgreSQL after 12 epochs (about 2 hours) and finally reaches **0.502**(train)/**0.482**(test) in WRL, which indicates a **1.990x**/**2.076x** total speedup, and **0.426**/**0.664** in GMRL on JOB. The shades in Figure 6 show relative latency from 25% to 75% of queries in testing workload. After sufficient training, about 75% of queries reach a relative latency lower than 1.1, and 25% of queries are lower than 0.4. We can also see that LOGER performs well on TPC-DS and Stack. An interesting phenomenon is that WRL on Stack testing workload is much higher than on training workload, which is caused by PostgreSQL's weakness on template 16 with an average time of 2.5 secs. In contrast, LOGER can reduce the average to 0.3 sec, significantly improving the workload performance. Besides, we can see that LOGER has a similar WRL but a higher GMRL on JOB-RS than on JOB. The time required to outperform the DBMS optimizer is also much longer, indicating that JOB-RS is a more challenging workload.

To verify whether LOGER can generalize to different database systems, we test it on JOB workloads on CommDB as is shown in Figure 7, from which we can find that LOGER also quickly reaches a better performance than CommDB's optimizer, and achieves **0.417**(train)/**0.474**(test) in WRL and **0.526**/**0.508** in GMRL after 200 epochs. The results show that LOGER can cooperate with different DBMS optimizers by ROSS to achieve better performance.

Workload latency analysis. A robust learned query optimizer should not only produce efficient plans for most queries but also ensure a good tail performance, *i.e.* good performance for the poorest optimized queries. LOGER has a total execution latency speedup of 52.96 seconds on JOB training workload, and 13.25 seconds on testing workload. Figure 9 shows the execution latency distribution on JOB. About 55% of queries achieve a speedup of at least 50ms on training workload, and 42% achieve a speedup of 50ms on testing workload. Although a few queries with relatively poor performance do exist, the largest slowdown is not disastrous. While 12% of test workload queries reach a speedup higher than 1000ms, the poorest is slowed down by only less than 500ms. From the results, we can see that LOGER not only produces efficient plans for most queries but also has a robust performance on the poorest optimized ones.

Performance on a changed schema. LOGER is designed to be capable of responding to schema changes through incremental training. For each table in the database, LOGER holds a set of learned parameters, including the learned embedding vector R_t, the learned matrix M_c and the learned operator representation M_o. When a new table is created, LOGER randomly initializes these parameters for the new table and fine-tunes the network on the previously trained checkpoint. We first train LOGER on JOB workload that excludes all queries containing table *char_name* for 100 epochs, and then train on the entire workload for another 100 epochs. The red and green curves in Figure 8 show the performance of LOGER on

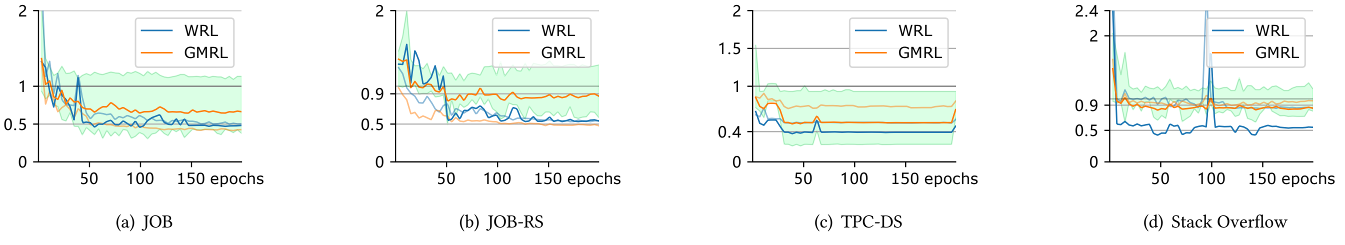


Figure 6: Performance of LOGER on PostgreSQL. Solid lines and transparent lines show the performances on testing workloads and training workloads respectively. The shades show relative latency from 25% to 75% of queries in testing workload.

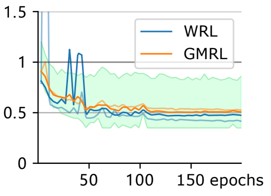


Figure 7: Performance of LOGER on CommDB.

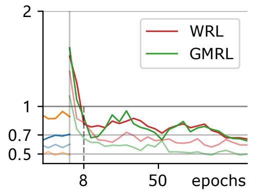


Figure 8: Performance responding to schema update.

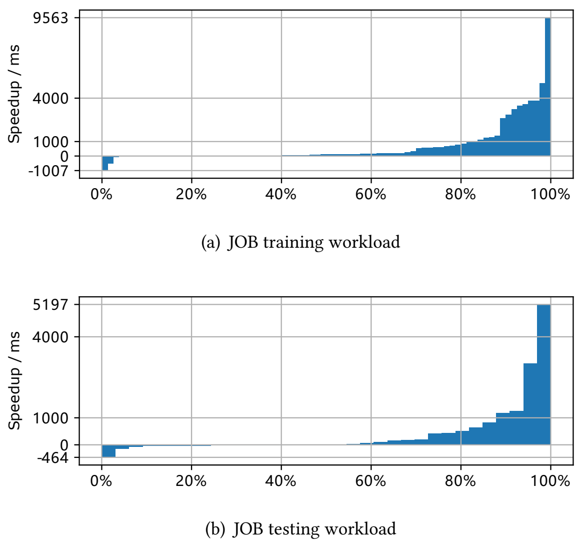


Figure 9: Execution latency speedup on JOB workload, compared with plans produced by PostgreSQL's optimizer.

all queries of *char_name* when responding to the schema update. Initially, LOGER has no information about the new table and has to exploit the information of other tables to select plans for the new queries. After 8 epochs of incremental training, LOGER performs better than PostgreSQL on the new queries, and finally reaches convergence on both training and testing workload.

Comparison with other methods. We compare LOGER with PG-NoMerge, RTOS [29], Bao [15] and Balsa [28] on each workload. Table 2 shows WRL, GMRL, average inference time T_{inf} and training time T_{train} of the methods, from which we can see that LOGER has highly competitive performance on all workloads with reasonable training and inference time. The results of PG-NoMerge show that globally disabling merge join operator performs well on Stack, but generally has very limited performance. As RTOS makes no decision on physical operators, LOGER can produce better results than it. The performance of Bao is also limited since it selects from only a small set of global settings. Balsa outperforms RTOS and Bao on JOB-RS and TPC-DS, but it fails to train on Stack. Only less than 20% of its training process is completed in 60 hours. This issue also occurs on RTOS. We further find that both RTOS and Balsa generate disastrous plans on Stack initially and are unable to outperform PostgreSQL afterwards.

6.3 Ablation Study

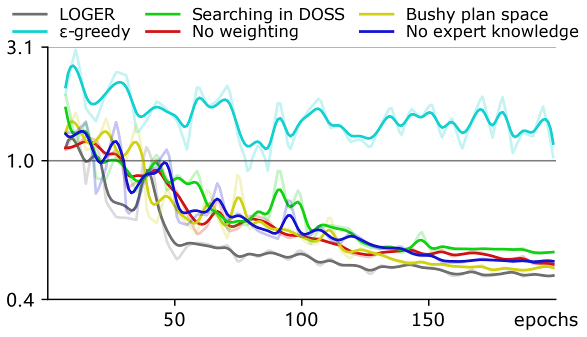
Through the following experiments, we analyze the design choices of LOGER shown in Table 3. We individually use alternatives to replace each component and test LOGER on JOB workload. Apart from WRL and GMRL of the last epoch, we use σ_{last}, the standard deviation of WRL in the last 50 epochs, to show training stability towards convergence. From the results, we can conclude that all replacements degrade performance to different extents.

Effect of ϵ-beam search. Since ϵ-beam search adopts the idea of the ϵ-greedy algorithm, we analyze both the advantage of introducing the idea of ϵ-greedy and the advantage over ϵ-greedy. The results show that ϵ-beam search is significantly better than both ϵ-greedy and original beam search without exploration, on both performance and training stability. ϵ-greedy fails to reach a performance better than PostgreSQL's optimizer. The original beam search is much more stable on training workload with lower performance, since it makes no exploration and selects almost the same plans in the last 50 epochs. However, this leads to low and unstable performance on testing workload because the model cannot learn sufficient information for generalization.

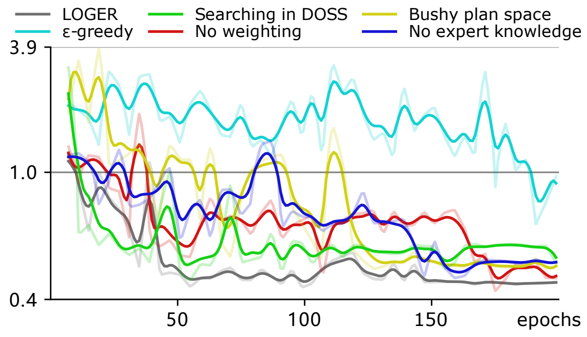
Effect of ROSS. ROSS balances between complete dependence on the DBMS optimizer and learning to select physical operators. We experiment on both searching in DOSS and selecting only join order while depending on the DBMS optimizer to select operators. LOGER with ROSS achieves the best performance on testing workload, and both alternatives are inferior to ROSS. Meanwhile, it can be seen from σ_{last} that searching is much less stable in DOSS than in

Table 2: Results of LOGER and other methods on different training and testing workloads and metrics. JOR, TDS and STK are the abbreviations of JOB-RS, TPC-DS and Stack respectively. TLE means the competitor cannot finish training. T_{inf} represents the average inference time, and T_{train} represents the training time.

		Ours (LOGER)				PG-NoMerge				RTOS				Bao				Balsa			
		JOB	JOR	TDS	STK	JOB	JOR	TDS	STK	JOB	JOR	TDS	STK	JOB	JOR	TDS	STK	JOB	JOR	TDS	STK
Training workload	WRL	**0.50**	**0.54**	0.56	0.85	1.00	1.00	1.01	**0.64**	0.62	0.83	**0.53**	TLE	0.81	0.68	0.86	1.01	0.56	0.55	1.34	TLE
	GMRL	**0.43**	**0.48**	**0.80**	0.97	0.97	0.97	1.01	0.90	0.65	0.65	0.89	TLE	0.96	0.96	0.96	**0.83**	0.85	0.86	1.22	TLE
Testing workload	WRL	**0.48**	**0.54**	**0.47**	**0.55**	1.00	1.00	0.99	0.83	0.80	0.88	0.73	TLE	0.58	0.70	0.79	0.75	1.02	0.59	0.49	TLE
	GMRL	**0.66**	**0.87**	**0.69**	0.85	0.97	1.00	1.00	0.98	0.84	0.94	0.97	TLE	1.00	0.87	0.86	**0.68**	1.22	1.09	0.95	TLE
T_{inf}/ms		**59.6**	**53.0**	37.0	**49.2**	N/A	N/A	N/A	N/A	72.7	64.0	**14.8**	TLE	92.5	73.3	18.6	158	188	160	44.9	TLE
T_{train}/h		23.2	25.0	5.6	15.6	N/A	N/A	N/A	N/A	7.2	7.0	**0.9**	TLE	**6.2**	**6.0**	1.9	**7.2**	22.3	17.9	9.3	TLE



(a) Training



(b) Testing

Figure 10: WRL of LOGER and a number of alternatives that have a major impact on cold-start ability and training stability. Transparent lines show the original values, and solid lines show smoothed values to better visualize the results.

Table 3: The results of the ablation study. σ_{last} refers to the standard deviation of WRL in the last 50 epochs.

Experiment	Training workload			Testing workload		
	WRL	GMRL	σ_{last}	WRL	GMRL	σ_{last}
Original LOGER	**0.502**	**0.426**	0.051	**0.482**	0.664	0.022
ϵ-greedy	1.039	0.940	0.154	0.980	1.061	0.284
No exploration	0.830	0.510	0.006	0.778	0.717	0.164
Searching in DOSS	0.606	0.450	0.026	0.572	0.703	0.113
No join operators	0.630	0.573	**0.003**	0.576	0.700	**0.019**
No GT layer	0.551	0.497	0.040	0.626	0.720	0.027
No weighting	0.548	0.503	0.064	0.500	0.649	0.336
Simple log function	0.578	0.543	0.054	0.571	0.719	0.149
Mean-pooling	0.579	0.477	0.028	0.563	0.657	0.107
Sum-pooling	0.542	0.451	0.023	0.656	0.646	0.087
No expert knowledge	0.565	0.470	0.030	0.576	**0.619**	0.127
Bushy plan space	0.536	0.541	0.038	0.557	0.694	0.046

ROSS. The results are not surprising as restricted operators decrease the difficulty of operator selection while keeping the potential to explore efficient plans. The stability of selecting only join order is even better than ROSS since the search space is much smaller, but the operators are completely decided by the DBMS optimizer, resulting in fewer chances to achieve better performance.

Effect of GT layer for query representation. The function of GT is to capture relationships between tables and predicates along with structural information of the join graph into table representations. We discard GT and directly use the initial representations $\hat{R}$ of tables as the final representations R in the experiment. It can be seen from Table 3 that without GT, the query representation module is unable to capture sufficient information, leading to worse generalization on testing workload.

Effect of reward weighting and log transformation. For the experiment of reward weighting, we set the weight factor w to 1, indicating that only operator-relevant latency is used. Such a modification results in a worse WRL but a slightly better GMRL, along with a disastrous σ_{last} indicating extremely unstable performance, as reward weighting makes a trade-off between the robustness and the efficiency of plans. We also test the effect of the log transformation function LT in formula (3) by replacing it with a simple $\log(x+1)$ function. The log transformation helps stabilize the performance, while the simple log function cannot handle disastrous reward values, incurring a low and unstable performance.

Effect of other alternatives on model performance. Query representation module aggregates column-level information by max-pooling. We test its reasonableness by replacing it with mean-pooling and sum-pooling respectively. It can be seen on testing workload that max-pooling achieves a much better WRL and the stablest performance but a slightly worse GMRL than the alternatives. Since columns are equally handled, mean- and sum-pooling may capture more complete information, but they lack the ability to focus on columns with small selectivities which have the most influence on the outputs of table scans.

To test the effectiveness of introducing expert knowledge, we disable the initialization of experience dataset with expert plans. The original LOGER has a higher GMRL but much better WRL and σ_{last} than the alternative. Although the expert knowledge might lead to sub-optimal plans for some queries, it enables LOGER to avoid poor plans initially, significantly improving training stability.

We also run an experiment on JOB without left-deep restriction, and find that the performance on testing workload decreases. Searching in bushy plan space leads to a higher potential to find better plans, but it also significantly enlarges the space and makes searching for efficient plans much harder, and finally reaches a lower performance on JOB.

An overall analysis of cold-start ability and training stability. Figure 10 shows the variation of WRL with epochs using a number of different alternatives that have a major impact on cold-start ability and training stability. Since the original values fluctuate severely, we smooth the curves for better visualization. ϵ-beam search makes the greatest improvement on these issues. It can be seen that on both training and testing workload, replacing it with ϵ-greedy causes a cold-start issue and a low performance with drastic fluctuation. Both removing reward weighting and removing expert knowledge make LOGER much harder to train from scratch, showing that these methods help search for efficient plans and are crucial for fast convergence. Allowing bushy plans raises a similar issue on JOB workload, making LOGER spend much more time to outperform PostgreSQL's optimizer. Besides, the training stability of using DOSS is lower than that of using ROSS, and LOGER finally reaches an inferior performance with DOSS.

7 RELATED WORK

Learned query optimizers. In recent years, various learned query optimizers have been proposed. ReJOIN [17] provides an idea of using reinforcement learning to select join orders. ReJOIN makes no consideration of physical operator selection, relying on the DBMS optimizer's operator selection algorithm. DQ [12] further presents an idea to predict the estimated cost of each subplan by a value model and fine-tune the model by using real execution latency. Despite the limited performance, the insights of these methods have made great contributions to later proposed learned query optimizers. To solve the ambiguity problem caused by flat vector subplan representations applied in ReJOIN and DQ, RTOS [29] utilizes tree-structured representations, which are vectorized by Tree-LSTM [24]. Similar to ReJOIN, RTOS only focuses on join order selection, which causes limitations to the performance. Neo [16] demonstrates the possibility to replace the traditional optimizer with a learned optimizer for SPJ queries. Neo initializes the model by training with plans generated by an expert optimizer and fine-tunes it to obtain a query optimizer that selects both join order and physical operators. Balsa [28] uses a simple simulator to replace the expert optimizer in Neo and applies various other strategies to completely avoid dependence on a traditional optimizer.

Our method proposes an idea to utilize knowledge of a traditional non-learning optimizer by restricted operator search space, which shares similar insight with Bao [15]. Established on top of a traditional optimizer, Bao optimizes the plan by globally enabling or disabling specific physical operators. While Bao achieves stable performance, small search space limits its performance as it relies on the traditional optimizer to generate plans. In contrast, our method specifies join order along with physical operator restriction on each join, and thus the search space is much larger with a higher potential to contain efficient plans.

Deep learning for database. With the great success of deep learning, database researchers seek to leverage advances of deep learning into different components of DBMS and achieve promising results. Previous works include but are not limited to learned index based on a recursive model [11], GNN-based entity resolution [3], cardinality estimation based on multi-task learning [23], RL-based database tuning [14], RL-based query scheduler [21], etc. The success of these methods shows the great potential of adapting deep learning methods to fields of database.

Graph neural networks. Graph neural network (GNN) is a type of network that learns node representations by embedding features into graph-structured data, which supports various downstream tasks [27]. With Graph Convolutional Network [10] as a pioneering method, a number of variants including Graph Transformer [5] have emerged in recent years. GT adopts the insight of Transformer [25] into GNN to capture relationships between adjacent nodes, and integrates structural information into each node representation through positional encoding, achieving competitive performance on various datasets [1, 8]. In LOGER, we use a join graph to represent each query and apply GT on it to produce table representations.

8 CONCLUSION

We propose LOGER, a learned query optimizer towards generating efficient and robust plans. We improve representation expressiveness by applying GT to the join graph of each query. We introduce ROSS to reduce difficulties in searching for efficient plans by leveraging the knowledge of DBMS optimizer, and propose ϵ-beam search to find potentially better plans while adaptively balancing exploration and exploitation. Robustness is further improved by a loss function with reward weighting and log transformation. Experiments show that our method achieves highly competitive performance on both PostgreSQL and a commercial DBMS and outperforms previous state-of-the-art learned optimizers, and that all the proposed components have a positive effect on performance.

We plan to extend LOGER in the following aspects. We will further study resource-aware or hardware-aware plan generation to adapt LOGER to different environments. It's also necessary to improve the method to support more complicated predicates like subqueries. Furthermore, we are seeking a way to employ more sophisticated DRL techniques to further improve robustness and performance.

ACKNOWLEDGMENTS

This work was partially supported by NSFC under Grant No.61832001 and 62272008, and ZTE-PKU Joint Program.

REFERENCES

[1] Emmanuel Abbe. 2017. Community detection and stochastic block models: recent developments. The Journal of Machine Learning Research 18, 1 (2017), 6446–6531.

[2] Peter Auer. 2002. Using confidence bounds for exploitation-exploration trade-offs. Journal of Machine Learning Research 3, Nov (2002), 397–422.

[3] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1335–1349. https://doi.org/10.1145/3318464.3389742

[4] Nan Ding and Radu Soricut. 2017. Cold-start reinforcement learning with softmax policy gradient. Advances in Neural Information Processing Systems 30 (2017).

[5] Vijay Prakash Dwivedi and Xavier Bresson. 2020. A generalization of transformer networks to graphs. arXiv preprint arXiv:2012.09699 (2020).

[6] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. Neural computation 9, 8 (1997), 1735–1780.

[7] Toshihide Ibaraki and Tiko Kameda. 1984. On the Optimal Nesting Order for Computing N-Relational Joins. ACM Trans. Database Syst. 9, 3 (1984), 482–502.

[8] John J Irwin, Teague Sterling, Michael M Mysinger, Erin S Bolstad, and Ryan G Coleman. 2012. ZINC: a free tool to discover chemistry for biology. Journal of chemical information and modeling 52, 7 (2012), 1757–1768.

[9] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. Journal of artificial intelligence research 4 (1996), 237–285.

[10] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In ICLR 2017.

[11] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In SIGMOD 2018. 489–504.

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

[13] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? Proceedings of the VLDB Endowment 9, 3 (2015), 204–215.

[14] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. Proc. VLDB Endow. 12, 12 (2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[15] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making learned query optimization practical. ACM SIGMOD Record 51, 1 (2022), 6–13.

[16] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. arXiv preprint arXiv:1904.03711 (2019).

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

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013).

[19] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. Tpc-ds, taking decision support benchmarking to the next level. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data. 582–587.

[20] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. 2018. A tutorial on thompson sampling. Foundations and Trends® in Machine Learning 11, 1 (2018), 1–96.

[21] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In SIGMOD 2022. ACM, 1228–1242. https://doi.org/10.1145/3514221.3526158

[22] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1989. Access path selection in a relational database management system. In Readings in Artificial Intelligence and Databases. Elsevier, 511–522.

[23] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. Proc. VLDB Endow. 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296

[24] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. arXiv preprint arXiv:1503.00075 (2015).

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In NIPS 2017. 5998–6008.

[26] Christopher John Cornish Hellaby Watkins. 1989. Learning from delayed rewards. (1989).

[27] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. IEEE Trans. Neural Networks Learn. Syst. 32, 1 (2021), 4–24.

[28] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. arXiv preprint arXiv:2201.01441 (2022).

[29] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In 2020 IEEE 36th International Conference on Data Engineering (ICDE). IEEE, 1297–1308.