COOOL: A Learning-To-Rank Approach for SQL Hint Recommendations (Regular Papers)

Xianghong Xu, Zhibing Zhao, Tieying Zhang*, Rong Kang, Luming Sun and Jianjun Chen

ByteDance Inc.

Abstract

Query optimization is a pivotal part of every database management system (DBMS) since it determines the efficiency of query execution. Numerous works have introduced Machine Learning (ML) techniques to this field, but few of them are proven practical due to long training time, lack of interpretability, and integration cost. A recent study provides a practical method to optimize queries by recommending per-query hints but it suffers from two inherited problems. First, it follows the regression framework to predict the absolute latency of each query plan, which is very challenging because the latencies of query plans for a certain query may span multiple orders of magnitude. Second, it requires training a model for each dataset, which restricts the application of the trained models in practice.

In this paper, we propose COOOL to predict the Cost Orders of query plans to cOOperate with DBMS by Learning-To-Rank. Instead of estimating absolute costs, COOOL uses ranking-based approaches to compute relative ranking scores of query plans. We show that COOOL is theoretically valid to distinguish between good and bad query plans. We implement COOOL on PostgreSQL. Extensive experiments on join-order-benchmark and TPC-H data demonstrate that COOOL outperforms PostgreSQL and state-of-the-art methods on both single-dataset tasks and multiple-dataset tasks. Our experiments also shed some light on why COOOL outperforms regression approaches from the representation learning perspective, which may guide future research.

Keywords

SQL hint, query optimization, learning-to-rank, representation learning

1. Introduction

Query optimization is vital to the performance of every database management system (DBMS). A SQL query typically has many candidate plans that are equivalent in terms of final output but differ a lot in execution latency. The goal of query optimization is to select the candidate plan with the lowest latency for each query from a vast search space with sufficient accuracy.

Query optimization has been studied for decades [1] and is still an active research field [2]. Various ML-based research lines have been proposed: cost modeling, cardinality estimation, end-to-end query optimization, etc., among which the most practical approach is Bao [3]. Bao is a query optimization system leveraging tree convolutional neural networks (TCNN) [4] and Thompson sampling [5] to recommend SQL hints. These hints offer additional information for the underlying optimizer to generate plans, thereby improving query performance. Bao has made a remarkable improvement in the practicality of end-to-end query optimization, but it suffers from two problems inherited from previous models [6, 7, 8, 9, 10]. **(1) The regression paradigm may limit the potential of models for query plan selection.** Bao follows the regression paradigm where the model must predict the exact cost of each plan in order to select the plan with the minimum cost. While accurately estimating the cost is sometimes desirable, it is very challenging for existing models [11, 12, 13]. Moreover, the model may sacrifice the accuracy of fast query plans for the accuracy of slow query plans, which leads to a suboptimal query plan selection. **(2) Limited generalizability.** Bao respectively trains a model for each dataset and evaluates the model on the corresponding dataset. The datasets vary among DBMS instances and it is exceedingly costly to train and maintain individual models for every dataset. The generalizability of the model is desirable in real-world scenarios, therefore, it is expected to train a single model to improve query plans from multiple datasets.

To address these issues, we propose COOOL to estimate the Cost Orders of query plans to cOOperate with DBMS leveraging Learning-To-Rank (LTR) techniques. COOOL is designed on top of an existing DBMS, and we make similar assumptions as Bao in order to inherit its advantages for *practical* applicability: we assume that a finite set of hint sets are predefined and all hint sets result in semantically equivalent query plans. We leverage the

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Workshop on Applied AI for Database Systems and Applications (AIDB'23), August 28 - September 1, 2023, Vancouver, Canada

*Corresponding author.
xuxianghong@bytedance.com (X. Xu);
zhibing.zhao@bytedance.com (Z. Zhao);
tieying.zhang@bytedance.com (T. Zhang);
kangrong.cn@bytedance.com (R. Kang); sunluming@ruc.edu.cn (L. Sun); jianjun.chen@bytedance.com (J. Chen)
0000-0003-2447-4107 (X. Xu); 0000-0001-6473-8205 (Z. Zhao); 0009-0003-2250-5528 (T. Zhang); 0009-0005-8449-0223 (R. Kang); 0000-0003-0538-2960 (L. Sun); 0000-0002-3734-892X (J. Chen)

widely used TCNN [4, 3, 14] as the underlying model.

LTR is a supervised learning framework to train models for ranking tasks. There are primarily three categories of LTR approaches: pointwise, pairwise, and listwise. Most pointwise methods are the same as the regression paradigm. Pairwise methods concern the relative order between two items, and listwise methods focus on the order of items in the entire list. The three approaches can be applied to most existing models, e.g., neural networks, and the only difference lies in the loss function. To train an LTR model for query optimization, we use the orders of latencies as labels, which has the potential to be more robust than regression models against the large range of orders of magnitude in query plan latencies. By transforming the absolute cost estimation problem into relative cost order prediction, COOOL can utilize LTR techniques to train the TCNN so that the output tells which plan is the best. In this paper, we investigate the performance of pairwise (COOOL-pair) and listwise (COOOL-list) approaches on SQL hints recommendation for query optimization.

COOOL methods (LTR approaches) have advantages over BAO (regression approach) for the following reasons. (1) Accurately estimating the cost of every query plan is not necessary since the optimizer only needs to select one of them to execute. A higher accuracy for a regression-based model does not always lead to the optimal plan selection because improving the accuracy on the slow query plans does not directly help. On the contrary, predicting the cost orders of query plans with high accuracy is more likely to help find the optimal one, which is exactly the objective of LTR. (2) Predicting the exact cost of a plan is extremely difficult. The plan execution latency ranges from several milliseconds to thousands of seconds. Minor structure differences in semantically equivalent plans may lead to large latency differences. The regression paradigm aims to minimize L_2 error, but the squared error formula is sensitive to anomalous large or small latencies, making model performance unstable [14]. Even though normalization can alleviate this, it may distort the latency distribution. By contrast, LTR approaches focus on the cost orders rather than the exact costs, which can alleviate the impact of outlier-prone data distributions.

We compare COOOL and BAO on both single-dataset tasks and multiple-dataset tasks and provide elaborate analysis on how to maintain a unified model to improve query plans from multiple datasets. Our experiments show that COOOL can consistently improve query plans in various settings, achieving as large as 6.09× total query execution speedup over PostgreSQL on single-dataset tasks and 6.73× speedup on multiple-dataset tasks. Moreover, we investigate the regression framework and ranking strategies from the perspective of representation learning [15]. We show that the model trained by the regression approach has a dimensional collapse [16] in the plan embedding space. Whereas COOOL does not have a dimensional collapse using the same embedding method. The dimensional collapse will hurt the performance of ML methods in building a unified model because the collapsed dimensions may be different for different datasets. This can result in features learned from one dataset becoming noise when applied to another dataset. Furthermore, Bao requires more effort to implement because it is fully integrated into PostgreSQL, while COOOL is built on top of PostgreSQL.

To summarize, we make the following contributions:

- We propose COOOL, a learned model that predicts **C**ost **O**rders of the query plans to c**OO**perate with DBMS by **L**TR techniques, to recommend better SQL hints for query optimization. To our best knowledge, it is the first end-to-end query optimization method that maintains a unified model to optimize queries from multiple datasets.
- We theoretically show that COOOL can distinguish between good and bad query plans when optimizing the loss functions, and verify that COOOL is superior to regression approaches from the representation learning perspective.
- Comprehensive experiments on join-order-benchmark and TPC-H show that COOOL can outperform PostgreSQL and state-of-the-art methods on multiple dimensions of evaluation criteria on both single-dataset and multiple-dataset tasks.

2. Preliminaries

2.1. Task Definition and Formalization

Let Q denote the set of queries and $\mathscr{H} = \{HS_1, HS_2, \dots, HS_n\}$ be the set of n hint sets. Each hint set $HS_i \in \mathscr{H}$ contains only boolean flag SQL hints (e.g., enable hash join, disable index scan). For any query $q \in Q$ and $i \in \{1, 2, \dots, n\}$, the traditional optimizer Opt can generate the corresponding plan tree t_i^q with the hint set HS_i

$$t_i^q = Opt(q, HS_i). \tag{1}$$

Let $T^q = \{t_1^q, t_2^q, \dots, t_n^q\}$ denote the set of candidate plans of query q. The query plans in T^q are semantically equivalent, but may have different execution latencies. A model M is a function that takes the candidate plan tree as input and produces a score for the plan tree.

$$s_i^q = M(q, t_i^q; \vec{\theta}), \tag{2}$$

where $\vec{\theta}$ is the parameter of the model to be trained. The query execution engine then selects the plan with the highest predicted ranking score in the scenario of LTR.

$$\hat{HS}^q = HS_{\arg\max_i s_i^q}, \tag{3}$$

where $\hat{HS}^q$ is the hint set with the maximum score, corresponding to the minimum estimated cost.

2.2. Learning-To-Rank (LTR)

LTR is a machine learning approach aiming to automatically rank items based on their relevance or importance [17]. It is a natural approach for the SQL hint recommendation task, which selects the optimal query plan for execution, as shown in Equation (3).

In the context of LTR, $\mathscr{H}$ is the set of items, and the goal is to recommend the best item from $\mathscr{H}$ for each $q \in Q$. More specifically, LTR is to define a loss function on s_i^q's so that the underlying model M can be properly trained to predict the orders of query plans. Throughout this paper, "$t_{i_1}^q \succ t_{i_2}^q$" means the plan $t_{i_1}^q$ is superior to (has a lower latency than) $t_{i_2}^q$. Given a query $q \in Q$, let $\sigma_q = t_{i_1}^q \succ t_{i_2}^q \succ \ldots \succ t_{i_n}^q$ denote the total order of query plans w.r.t. their latencies, where $t_{i_1}^q$ has the lowest latency, $t_{i_2}^q$ has the second lowest latency, and $t_{i_n}^q$ has the highest latency.

Before delving into our approaches, we first introduce the Plackett-Luce model (PL) [18, 19], which serves as the basis for our ranking methods. PL is a pioneering work in LTR and stands as one of the most popular models for discrete choices, which was later used as a listwise loss function in information retrieval [17] and softmax function in classification tasks [20]. In the context of SQL hint recommendations, we provide a definition of PL as follows. Given any $q \in Q$, the probability of $\sigma_q = t_{i_1}^q \succ t_{i_2}^q \succ \ldots \succ t_{i_n}^q$ is

$$\Pr_{\mathrm{PL}}(t_{i_1}^q \succ t_{i_2}^q \succ \ldots \succ t_{i_n}^q; \vec{\theta}) = \prod_{j=1}^{n} \frac{\exp(s_{i_j}^q)}{\sum_{m=j}^{n} \exp(s_{i_m}^q)}, \tag{4}$$

where s_i^q's are functions of the parameter $\vec{\theta}$ defined in Equation (2). The rankings of multiple queries are assumed to be independent. Therefore, the probability of multiple rankings is simply the product of the probability of each individual ranking. The marginal probability of any pairwise comparison $t_i^q \succ t_j^q$ is

$$\Pr_{\mathrm{PL}}(t_i^q \succ t_j^q; \vec{\theta}) = \frac{\exp(s_i^q)}{\exp(s_i^q) + \exp(s_j^q)}. \tag{5}$$

2.2.1. Listwise Loss Function

Given the training data $\{\sigma_q | \forall q \in Q\}$, the listwise loss function is simply the negative log-likelihood function:

$$\mathscr{L}_{\mathrm{list}}(\vec{\theta}) = -\sum_{q \in Q} \ln \Pr_{\mathrm{PL}}(\sigma_q; \vec{\theta}), \tag{6}$$

where $\Pr_{\mathrm{PL}}(\sigma_q; \vec{\theta})$ is defined in Equation (4). This listwise loss function also coincides with the listMLE loss by [21]. [21] proved that this loss function is consistent, which means as the size of the dataset goes to infinity, the learned ranking converges to the optimal one.

2.2.2. Pairwise Loss Function

To apply a pairwise loss function, the full rankings σ_q for all $q \in Q$ need to be converted to pairwise comparisons. This process is called rank-breaking [22]. A rank-breaking method defines how the full rankings should be converted to pairwise comparisons. Basic breakings include full breaking, adjacent breaking, and others [22]. Full breaking means extracting all pairwise comparisons from a ranking, and adjacent breaking means extracting only adjacent pairwise comparisons. For example, given a ranking $t_1 \succ t_2 \succ t_3$, full breaking converts it to ($t_1 \succ t_2$, $t_1 \succ t_3$, $t_2 \succ t_3$) while adjacent breaking converts it to ($t_1 \succ t_2$, $t_2 \succ t_3$). Though adjacent breaking is simple and plausible, [22] proved that adjacent breaking can lead to inconsistent parameter estimation, which means that even if the model is trained using an infinite amount of data, it may not make unbiased predictions. On the other hand, full breaking is proven consistent [22, 23]. Other breakings are more complicated and beyond the scope of this paper, so we employ the full breaking strategy.

Let $P = \{\pi_1, \pi_2, \ldots, \pi_m\}$ be the dataset of pairwise comparisons, where $\forall j \in \{1, 2, \ldots, m\}$, π_j has the form of $t_{i_1}^{q_j} \succ t_{i_2}^{q_j}$. Here q^j denotes the corresponding query for π_j. For different j values, q_j may refer to the same query since different pairwise comparisons can be extracted from the same query. The objective function is

$$\mathscr{L}_{\mathrm{pair}}(\vec{\theta}) = -\sum_{j=1}^{m} \ln \Pr_{\mathrm{PL}}(\pi_j; \vec{\theta}), \tag{7}$$

where $\Pr_{\mathrm{PL}}(p_j; \vec{\theta})$ is defined in Equation (5). The model parameter can be estimated by maximizing $L_{\mathrm{pair}}(\vec{\theta})$.

This is equivalent to maximizing the composite marginal log-likelihood based on Equation (5). [24] and [23] proved that $\vec{\theta}$ can be efficiently estimated under a consistent rank breaking method. And full breaking, which extracts all pairwise comparisons from the full rankings, is one of the consistent breaking methods.

3. COOOL Architeture

The data flow pipeline of COOOL is shown in Figure 1, where model-related modules are shown in green, and existing DBMS modules are displayed in blue. Given a query (SQL), we use the hint sets in $\mathscr{H}$ to generate the corresponding query plans $t_1, t_2, \ldots, t_n$ (with possible duplicates). At the training stage, we execute plans and

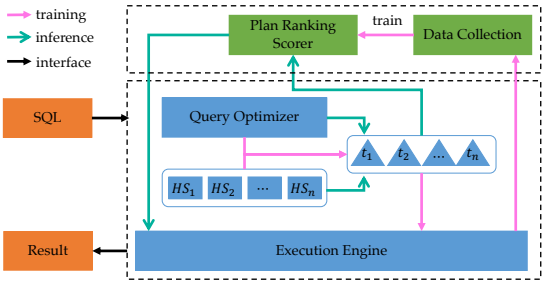


Figure 1: A brief view of the COOOL pipeline.

collect their performance as training data. Then we utilize the pairwise or listwise approach to train the ranking scorer. At the inference stage, when a user submits a query, the traditional optimizer will generate n query plans by utilizing the corresponding hint sets. Next, the scorer will compute the relative ranking score of each plan and recommends the optimal one for the execution engine to obtain results. For the models trained by different methods, they are exactly the same at the inference stage.

Data Collection Our approach is trained in a standard supervised learning paradigm. For the given queries, we generate n query plans that correspond to the hint sets $\mathscr{H}$ for each query by the underlying traditional optimizer. Then we sent them to the execution engine and record the observed execution performance of each query plan. The collected data is used to train the neural ranking scorer, and the training stage is separate from the DBMS.

Cost Order Estimation To execute the optimal plan with minimum estimated latency, we need to recommend its corresponding hint set to the DBMS. The scorer takes a plan tree as input and outputs the ranking score of the plan, the estimated latency orders can be acquired by sorting the scores of all candidate plans. Specifically, we first transform the nodes of the input plan tree into vectors, then feed the vector tree into a plan embedding model constructed by a TCNN. Finally, we feed the plan embedding into a multilayer perceptron (MLP) to compute the estimated score of the plan. At the training stage, we use the collected plan and latency data to train the model. So it can estimate relative orders of plans by latency and cooperate with DBMS to improve query plans at the inference stage.

Assumptions and Comparisons We assume that applying each hint set to the given query will generate semantically equivalent plans. Besides, the hints are applied to the entire query rather than the partial plan. Though allowing fine-grained hints (e.g., allows nested loop joins between specific tables, others not.) are available, it will bring an exponential candidate plan search space, which significantly increases training and inference overhead. We make the same assumption as Bao for practicality.

Bao requires more effort to implement because it is fully integrated into PostgreSQL, while we build on top of PostgreSQL, which makes it easy for COOOL to migrate to other DBMSs. Moreover, we take a step forward to maintain a unified model to optimize queries from different datasets, which has not been investigated in the previous end-to-end query optimization studies.

4. COOOL for Hint Recommendations

The architecture of COOOL is shown in Figure 2, with model-related modules depicted in green, existing DBMS modules in blue, and training stages in red. We utilize the underlying DBMS optimizer to generate candidate plans using the hint sets in $\mathscr{H}$. Then we vectorize plan tree nodes and binarize the plan, facilitating TCNN application to acquire the plan embedding $\vec{p}$. Subsequently, we employ an MLP to compute the score of each plan, supporting either pairwise or listwise training loops.

4.1. Cost Order Estimation

Each hint set corresponds to a query plan tree, so recommending the optimal hint set for given queries is to select the plan tree with the maximum ranking score, as shown in Equation (3). Similar to [6, 3, 14], we use a TCNN to obtain plan embeddings and leverage an MLP to compute the ranking scores.

Plan Tree Vectorization We can use the EXPLAIN command provided by the underlying optimizer to obtain the plan tree text, as shown in Equation (1). First, we need to transform each node in the plan tree into a vector. The same as in [3], we use a data/schema agnostic encoding scheme, which solely contains a one-hot encoding of operator types, and cardinality and cost provided by the underlying traditional optimizer. Then, we transform the original tree into a binary tree to facilitate tree convolution operations.

One-hot Node Operation Type Encoding. We summarize the types of all operations (nested loop, hash join, merge join, seq scan, index scan, index only scan, and bitmap index scan) in the plan trees and number these seven operations. Then we create a vector for each node with the number of types of bits, and we set the bit of the type corresponding to each node to the high bit. For

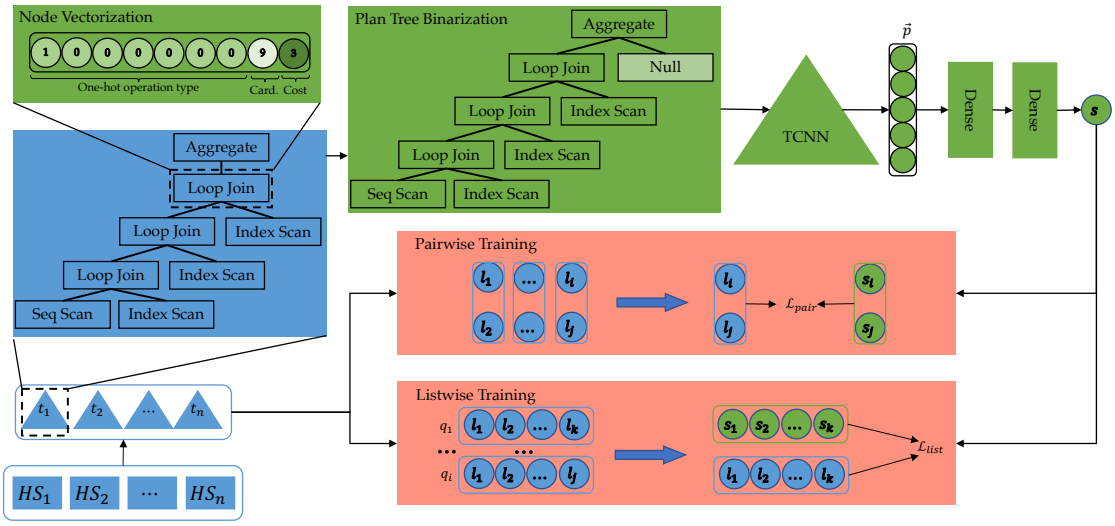


Figure 2: Cost order estimation for tree convolutional neural network using pairwise and listwise LTR techniques.

instance, $E_o(v)$ is the one-hot encoding of node v and $E_o(v)[i] = 1$ indicates node v is the i-th operation type and the rest elements in $E_o(v)$ are 0. Though one-hot node type encoding is simple, it is capable to extract structural information in plan trees.

Tree Nodes Vectorization. Apart from the operation information, each node can contain the cost and cardinality. It is applicable to acquire cost and cardinality from multiple traditional optimizers and learned models, but we only use two values obtained from the underlying traditional optimizer for simplicity. Therefore, the node encoding is the concatenation of operation type encoding, cardinality, and cost, i.e., $E(v) = \text{Concat}(E_o(v), \text{Cost}(v), \text{Card}(v))$, where $\text{Cost}(v)$ and $\text{Card}(v)$ are the cost and cardinality estimated by the traditional optimizer, respectively. We apply the node encoding method to the nodes in the plan tree to obtain a vectorized plan tree.

Tree Structure Binarization. Some nodes in a plan tree may have only one child, e.g., nodes for aggregation and sorting operations. To facilitate tree convolution operations, we transform the non-binary trees into binary trees by adding a pseudo-child node Null to each node with only one child node, and the costs and cardinalities of pseudo-child nodes are 0. Then the original plan tree t can be transformed into vectorized tree p.

4.1.1. TCNN Plan Embedding

TCNN was proposed in [4] to treat tree structure data in programming language processing. TCNN was first introduced in plan representation in [6], and it was well-established in [3, 14]. We briefly introduce how to represent a plan using TCNN in this section. More technical details can be found in [4, 6].

During the execution of the original plan tree in execution engines, the computation of one node relies on the results of its child nodes. Based on this fact, the plan embedding method should reflect the recursive properties to obtain a proper inductive bias [25]. To be consistent with plan execution, the model is naturally required to simultaneously capture the features of a node and its child nodes. Specifically, let $l(v)$ and $r(v)$ denote the left and right child nodes of node v, respectively. The statistic cost/cardinality information in vector $E(v)$ is closely related to $E(l(v))$ and $E(r(v))$. Tree convolution can naturally address this requirement.

Tree convolution is similar to image convolution, it has binary tree structure filters to capture local features. We take a tree convolution filer as an example, there are three weight vectors in the filter, i.e., w, w_l, w_r. Applying tree convolution to the current node $E(v)$ can acquire the new representation:

$$E(v)' = \sigma(E(v) \odot w + E(l(v)) \odot w_l + E(r(v)) \odot w_r), \quad (8)$$

where σ is a non-linear activation function, $\odot$ is a dot product operation. The new representation of node v contains its child nodes' information. By this means, the model is able to capture high-level features of a long chain of plan execution for representing one node. The output of tree convolution operations is a tree with the same structure as the input, and we employ a dynamic pooling method to aggregate the latent representations of all nodes to represent the query plan. To sum up, we can

obtain plan embedding by $\vec{p} = \text{TCNN}(p)$, where $\vec{p} \in \mathbb{R}^h$ is the vector of plan representation and h is the size of plan tree embedding space.

4.1.2. Ranking Score Computation

Finally, we can leverage plan embedding to compute the relative ranking score. We use a simple MLP to take the plan embedding vector as input and output a scalar as the ranking score s, i.e., $s = \text{MLP}(\vec{p})$. Stacking fully connected layers and non-linear activation functions in the MLP can enhance the representation ability, and this practice has been widely adopted in query optimizations [6, 14]. We add a hidden layer and an activation function for simplicity. The cost orders can be obtained by sorting the ranking scores of all candidate plans.

4.2. Learning-To-Rank Training Loop

The training loop consists of three parts: data collection and deduplication, label mapping and pairwise data extraction, and model training and evaluation.

Data Collection and Deduplication Let Q_{train} denote the set of training queries. We generate n plans for each query $q \in Q_{\text{train}}$ by the traditional optimizer, as shown in Equation (1). Then we sent the plans to the execution engine, and record each data point $(\text{query} = q, \text{plan} = t, \text{latency} = l)$. There are duplicate query plans because, for a given query, different hints may result in the same query plan. We remove the duplicate query plans for pairwise and listwise training loops.

Label Mapping and Pairwise Data Extraction Because a lower latency indicates a better plan, we use the reciprocal of the latency of each query plan as the label to reverse the orders of query plans. Any other mapping function that reverses the orders works equivalently because only the orders matter.

After the label mapping, we get a list of query plans ordered by the reciprocals of their latencies for each query. For each list of n^q query plans ($n^q \leq n$ after deduplication), we extract all $\binom{n^q}{2}$ pairwise comparisons to get the pairwise data $P = \{\pi_1, \pi_2, \dots, \pi_m\}$, where for any $j \in \{1, 2, \dots, m\}$, $\pi_j = t_{i_1}^{q_j} \succ t_{i_2}^{q_j}$.

Model Training and Evaluation At the training stage, the model parameter $\vec{\theta}$ is updated by optimizing the loss function. Specifically, for the listwise approach, the model parameter $\vec{\theta}$ is computed by minimizing the listwise loss defined in Equation (6). For the pairwise approach, $\vec{\theta}$ is computed by minimizing the pairwise loss defined in Equation (7).

During the inference stage, we use the learned model to compute the score for each candidate plan and sort the scores to obtain the corresponding orders, then we select the estimated best plan for each query to execute.

4.3. Theoretical Basis and Comprehension

4.3.1. Theoretical Analysis

In this section, we briefly analyze how COOOL learns from the order of latencies of different query plans. To show that, we consider the query plans $\{t_1, t_2, \dots, t_n\}$ with the corresponding latencies $\{l_1, l_2, \dots, l_n\}$ for a given query $q \in Q$. Our goal is to select the best plan using the model. At the training stage, they have different initial scores $\{s_1, s_2, \dots, s_n\}$. Without loss of generality, we assume $l_1 > l_2 > \dots > l_n$. We focus on the difference between the model outputs of adjacent query plans, we denote $\delta_i = s_{i+1} - s_i, \forall i \in \{1, 2, \dots, n-1\}$. For convenience, we define $\delta_0 = 0$. Then $\forall i = \{1, 2, \dots, n\}$, we have $s_i = s_1 + \sum_{j=0}^{i-1} \delta_j$. The loss function of the listwise approach can be written as:

$$
\begin{aligned}
\mathscr{L}_{list} &= -\ln \prod_{j=1}^{n} \frac{\exp(s_{n-j+1})}{\sum_{m=1}^{n-j+1} \exp(s_m)} \\
&= -\sum_{j=1}^{n} \ln \frac{\exp(s_1 + \sum_{k=0}^{n-j} \delta_k)}{\sum_{m=1}^{n-j+1} \exp(s_1 + \sum_{k=0}^{m-1} \delta_k)} \\
&= -\sum_{j=1}^{n} \ln \frac{\exp(\sum_{k=0}^{n-j} \delta_k)}{\sum_{m=1}^{n-j+1} \exp(\sum_{k=0}^{m-1} \delta_k)} \\
&= -\sum_{j=1}^{n} \left(\sum_{k=0}^{n-j} \delta_k - \ln(\sum_{m=1}^{n-j+1} \exp(\sum_{k=0}^{m-1} \delta_k)) \right).
\end{aligned}
$$

In this equation, the first equality is obtained by substituting Equation (4) in Equation (6), and the second equality is obtained by substituting s_i with $s_1 + \sum_{j=0}^{i-1} \delta_i$. The third equality is obtained by dividing both the numerator and denominator by $\exp(s_1)$ since $\exp(s_1) > 0$. And the last equality is due to the property of the ln() function. Now we compute the partial derivative of $\mathscr{L}_{list}$ with respect to δ_i for any $i \in \{1, 2, \dots, n-1\}$:

$$
\frac{\partial \mathscr{L}_{list}}{\partial \delta_i} = -\sum_{j=1}^{n-i} \left(1 - \frac{\sum_{m=i+1}^{n-j+1} \exp(\sum_{k=0}^{m-1} \delta_k)}{\sum_{m=1}^{n-j+1} \exp(\sum_{k=0}^{m-1} \delta_k)} \right) < 0. \tag{9}
$$

The inequality holds because when $1 \leq i \leq n-j$, we have $\sum_{m=i+1}^{n-j+1} \exp(\sum_{k=0}^{m-1} \delta_k) < \sum_{m=1}^{n-j+1} \exp(\sum_{k=0}^{m-1} \delta_k)$. This means $\forall i \in \{1, 2, \dots, n\}$, an increase in δ_i leads to a decrease in the loss function $\mathscr{L}_{list}$. δ_i tends to go up while the loss function $\mathscr{L}_{list}$ is minimized during the training process, which is desired. We omit the theoretical analysis of the pairwise approach, as it follows a similar logic to the listwise method.

To summarize, by minimizing $\mathscr{L}_{list}$ or $\mathscr{L}_{pair}$, the differences between the ranking scores of different query plans tend to increase, which means that our approaches are able to distinguish the best plans from others.

4.3.2. Intuitive Explanation

To intuitively demonstrate the effectiveness of our approaches, we present an illustrative example to explain how the regression and ranking strategies work based on Section 4.3.1.

Given the assumption that $l_1 > l_2 > ... > l_n$, where plan tree t_n has the lowest latency represents the optimal plan. For the regression paradigm, the model endeavors to learn to minimize the gap between each of the model output score s_i and the corresponding latency l_i. The pairwise ranking strategy enables the model to learn to discriminate the preferable plan by assigning it a higher score, whereas the listwise ranking strategy allows the model to learn to provide the order score of each plan in the given plan list. Essentially, the regression and ranking strategies train the model to output ideal scores of plan trees under their respective training paradigms, whereby the optimal plan selection operation is accomplished through the procedure of sorting the model output scores. On the one hand, the three strategies have different objectives and train the model in different ways, while the goal of query optimization is to simply select the optimal single plan, which is the approach of the ranking strategies. On the other hand, for the scenario of unseen query plans, it may be difficult to predict the exact latency of the plans, yet predicting the orders of plans may alleviate this issue.

In summary, the proposed COOOL methods can effectively distinguish between better and worse plans to achieve excellent optimization performance.

5. Experiments

5.1. Experimental Setup

Datasets and Workloads To promote reproducibility, we use two widely-used open-source datasets (IMDB, TPC-H) and their corresponding workloads (JOB, TPC-H). To maintain a fair comparison, we do not alter the original queries.

- **Join Order Benchmark (JOB)**. JOB [26] contains 113 analytical queries, which are designed to stress test query optimizers over the Internet Movie Data Base (IMDB) collected from the real world. These queries from 33 templates involve complex joins (ranging from 3 to 16 joins, averaging 8 joins per query).
- **TPC-H**. TPC-H [27] is a standard analytical benchmark that data and queries are generated from uniform distributions. We use a scale factor of 10. There are 22 templates in TPC-H, we omit templates #2 and #19 because some nodes in their plan trees have over two child nodes, which makes tree convolution operation unable to handle. For each template, we generate 10 queries by the official TPC-H query generation program[1].

Baseline Methods. We compare our proposed model with traditional and state-of-the-art (SOTA) methods as follows.

- PostgreSQL: We use the optimizer of PostgreSQL (version 12.5) itself with default settings.
- Bao: We substantially optimize the Bao source code[2] as follows. First, we use all 48 hint sets in Bao paper, rather than the 5 hint sets in the open-sourced code. Second, because we use the standard benchmarks without modifying queries, we train Bao on all sufficiently explored execution experiences of the training set.

Model Implementation We use a three-layer TCNN and the number of channels are respectively set as {256, 128, 64}, the plan embedding size h is 64, and the hidden size of MLP is set as 32. The activation function is Leaky ReLU [28], the optimizer is Adam [29] with an initial learning rate of 0.001, the batch size is 128, we apply an early stopping mechanism in 10 epochs on the training loss, we save the model that performed best on the validation set and report the results on the test set. The validation set is 10% of the training set, except for TPC-H "repeat" settings, where we use 20% of the training set. We implement our models in Pytorch 1.12.0.

Device and PostgreSQL Configurations We use a virtual machine with 16 GB of RAM, an 8-core Xeon(R) Platinum 8260 2.4GHz CPU, and an NVIDIA V100 GPU. For excellent PostgreSQL performance, we set PostgreSQL configurations suggested by PGTune[3] in this paper: 4 GB of shared buffers, 12 GB of effective cache size, etc.

Scenario Descriptions To comprehensively evaluate the performance of models, we examine three scenarios: **single instance**, **workload transfer**, and **maintaining a model** (i.e., a unified model). The first scenario is common in ML for query optimization, which refers to

[1] https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp
[2] https://github.com/learnedsystems/baoforpostgresql
[3] https://pgtune.leopard.in.ua/#/

learning a model for each dataset. The others are not commonly studied but are crucial for ML model deployment in the DBMS. Workload transfer indicates the scenario where a model is trained on one workload and evaulated on another. The last scenario implies training a model to improve query plans from different datasets.

Definition of Latency and Speedup We repeat each experiment 10 times and exclude the best and the worst test results except for the unified model scenario, then we report the average performance of the rest runs. There are two aspects of performance we consider: one is latency, and the other is speedup. The total execution **latency** is defined as the sum of per-query execution latency. The **speedup** represents the acceleration ratio of executing queries with the ML model compared to using PostgreSQL's default optimizer under the same PostgreSQL configurations. A speedup greater than 1 indicates that the model performs better than the default PostgreSQL optimizer.

Model Performance Evaluation Criteria According to the evaluation details in the previous works [3, 14], there are primarily two types of evaluation settings in the scenario of single instance optimization: 1) randomly dividing the standard benchmark into train and test sets. 2) Using the standard benchmark as the test set while augmenting queries by randomly replacing predicates, ensuring that the training data covers all templates present in the test set. These two evaluation aspects may not provide a comprehensive assessment, so we adapt them as follows while using the standard benchmark without adding extra queries.

First, we consider two situations that frequently arise in practical settings, which we refer to as **adhoc** and **repeat**. The former refers to a common real-world situation where the test templates have not been covered in the model's training set, we use queries from seven templates in the JOB dataset and four templates in the TPC-H dataset respectively as test sets for this setting. The latter represents the queries in the test data are "similar" to the queries in the training data, but not the same. Here "similar" means the queries are from the same templates as in the training data. In practice, we take one or multiple queries from each template as the test set and keep the remaining queries in the training set. For the JOB dataset, we take one query from each template and for the TPC-H dataset, we take two queries from each template as test data. For queries from the same template, we take their average latency to represent the corresponding template's latency.

Then, to substantially evaluate the general and tail latency query performance, we develop two methods to select the test sets. For both "adhoc" and "repeat" settings, we have two options for selecting test sets from the dataset: we can either randomly select templates/-queries, denoted by **rand**, or choose the slowest templates/queries, referred to as **slow**.

Therefore, we can establish four evaluation criteria for a single dataset in a specific scenario.

Research Questions (RQs) We conduct extensive experiments to primarily answer the following RQs:

- *RQ1: can COOOL achieve the best performance compared to the baseline methods in terms of total query execution latency and improvement on slow queries?*
- *RQ2: is it possible to directly transfer a schema agnostic model to another dataset?*
- *RQ3: can the proposed methods improve query plans from different datasets compared with PostgreSQL by maintaining a unified model?*
- *RQ4: can the experiments provide some insight on why COOOL methods are better than the regression-based approach?*

5.2. Single Instance Experiments (RQ1)

In this section, we focus on the performance in the *single instance* scenario, and the overall results are summarized in Table 1. The individual query performance in "repeat" settings is shown in Figure 3, where we depict queries with an execution latency greater than 1s on PostgreSQL to facilitate observation, and "Optimal" represents the lowest latency under the given $\mathscr{H}$.

Observations In Table 1, there are four settings for each of the two workloads and we observe that:

- The listwise approach (COOOL-list) achieves the best performance in most settings, and beats Bao by large margins in almost all settings except for "repeat-rand" on JOB, where COOOL is slightly slower than Bao.
- The pairwise approach (COOOL-pair) also outperforms Bao in almost all settings except for "adhoc-rand" on TPC-H. It achieves the best performance on three settings ("repeat-rand" on JOB, "adhoc-slow" and "repeat-slow" on TPC-H). In most settings, the performances of COOOL-pair and COOOL-list are similar.
- Bao does not have the best performance under any of these settings. For "adhoc-slow" on JOB, Bao even has a total execution latency *regression*, which means the running speed is lower than the PostgreSQL optimizer itself.

Combined with Figure 3, we have the following observations:

Table 1
Total query execution latency speedups on single datasets over PostgreSQL. The best performance on each workload is in boldface.

	JOB				TPC-H			
	adhoc-rand	adhoc-slow	repeat-rand	repeat-slow	adhoc-rand	adhoc-slow	repeat-rand	repeat-slow
Bao	1.07	0.91	3.02	1.37	5.36	1.17	5.28	4.73
COOOL-list	**1.35**	**1.46**	3.01	**1.57**	**6.09**	3.63	**5.33**	5.55
COOOL-pair	1.30	1.36	**3.47**	1.56	3.86	**3.86**	5.28	**5.56**

Table 2
Number of regressions for Bao and COOOL compared with PostgreSQL (single instance)

Setting	Bao	COOOL-list	COOOL-pair
JOB repeat-rand	24	17	**13**
JOB repeat-slow	17	11	**8**
TPC-H repeat-rand	3	**1**	**1**
TPC-H repeat-slow	**1**	**1**	**1**

- Bao is significantly worse than COOOL on some queries under "slow" settings (20c in "repeat-slow" on JOB and template 9 in "repeat-slow" on TPC-H.). "Slow" settings are obviously more challenging for ML models than "rand" settings and COOOL has clear advantages over Bao.
- Both COOOL and Bao are close to optimal on "repeat-rand" scenarios for JOB and TPC-H data.
- It happens that Bao or COOOL are not as good as PostgreSQL on a small number of queries, which is normal for ML models. Nevertheless, performance regressions occur less frequently on COOOL models than Bao models in most settings, and COOOL-pair has the lowest number of query regressions for these settings. See Table 2 for details.

To summarize, we observe that COOOL methods have advantages over Bao in speeding up total query execution, alleviating individual query performance regression, and optimizing slow queries. Besides, although COOOL-list is better than COOOL-pair in total query execution latency speedups, it is not as good in avoiding individual query regression.

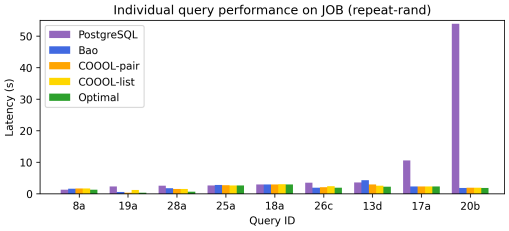
5.3. Workload Transfer Investigation (RQ2)

It is widely acknowledged that an instance-optimized model may exhibit poor performance on another workload because it does not learn the patterns from the unseen data. Despite most ML models being schema specific, hindering model transferability, there is limited research on schema agnostic models. In this section, our goal is to provide an intuitive perspective on directly transferring a learned model to another workload. More concretely, we train a model on a *source* workload and then test its performance on another workload, namely *target* workload (*source*→*target*).

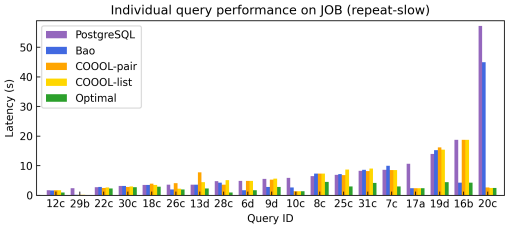
We conduct experiments on training on JOB data and evaluating the model on TPC-H data (JOB→TPC-H), as well as training a model on TPC-H data and evaluating its performance on JOB data (TPC-H→JOB). In these settings, we also consider the aforementioned "adhoc" and "repeat" scenarios under "rand" and "slow" settings. To make an intuitive comparison, we train the model on the source workload's training set and show the performance on the target workload's test set.

The overall query execution speedups are shown in Table 4, and we have the following observations.

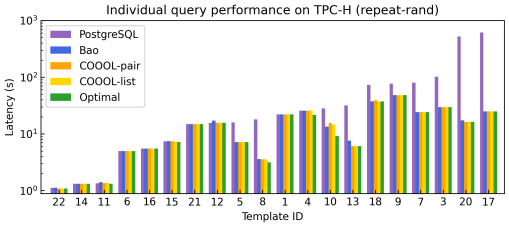
- The difference of the performances of models is unstable compared with the corresponding instance-optimized models, and most settings have a performance decline. Our experiments show that directly applying a model trained on one workload to another workload cannot obtain good performances even if the model is schema agnostic.
- Compared with instance-optimized models, most models learned on TPC-H perform worse on JOB. By contrast, there is a performance improvement on "TPC-H adhoc" settings especially on "TPC-H adhoc-slow". We may conclude that the data in JOB may benefit TPC-H and especially improve the performance of slow queries. We will provide a deeper analysis in Section 5.5.2.
- We show the plan tree statistics for nodes and depth of the two workloads in Table 3. On the one hand, it indicates that JOB is more complicated than TPC-H, so directly transferring a model learned on JOB helps optimize queries from TPC-H. On the other hand, it suggests that data distributions in different datasets affect the performance of ML models in workload transfer, because the experiment results on "TPCH repeat" settings demonstrate that a model learned on the templates from JOB cannot achieve the same performance as that learned on the template from



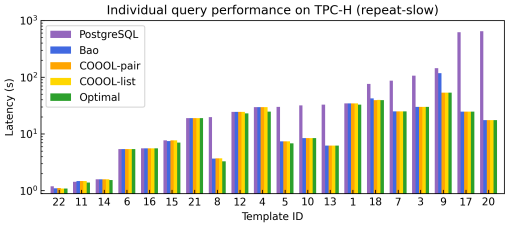
(a) JOB repeat-rand



(b) JOB repeat-slow



(c) TPC-H repeat-rand



(d) TPC-H repeat-slow

Figure 3: Individual query performance of the models in the single instance scenario.

TPC-H.

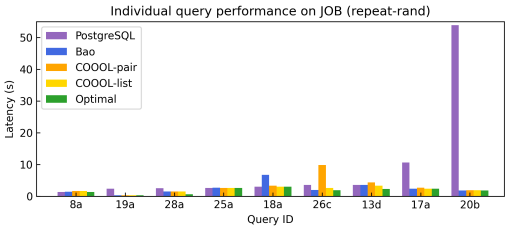
Based on the observations, introducing JOB data has a better performance than the model trained on TPC-H "adhoc" settings. We can conclude that for a given workload, using the training data from the same workload may not obtain satisfactory performance. Therefore, how to utilize data from other workloads is an emerging problem for query optimization models.

5.4. Unified Model Performance (RQ3)

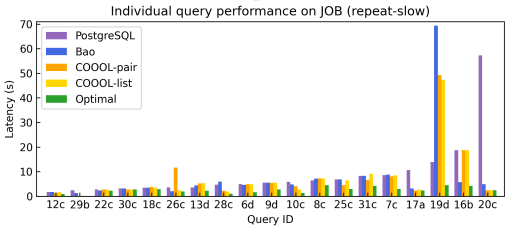
In this section, our objective is to explore whether it is possible to train a model on multiple datasets to improve

Table 3
Overall plan tree statistics of the two workloads.

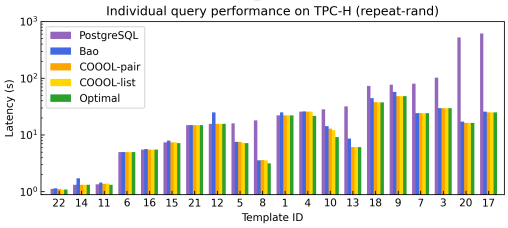
Work-load	Max Nodes	Avg. Nodes	Max Depth	Avg. Depth
JOB	72	23.6	36	12.0
TPC-H	35	14.3	20	9.6



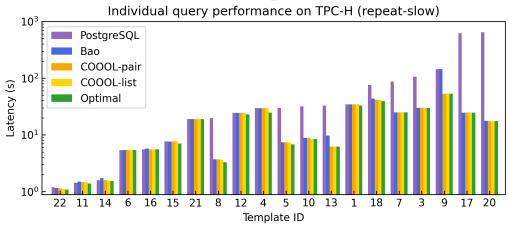
(a) JOB repeat-rand



(b) JOB repeat-slow



(c) TPC-H repeat-rand



(d) TPC-H repeat-slow

Figure 4: Individual query performance of the unified model.

query plans in comparison with PostgreSQL. Similar to the previous section, we consider the four scenarios for each dataset. For each scenario, the training data of JOB and TPC-H were combined as the new training set and the model is respectively evaluated on JOB and TPC-H test sets. The results are summarized in Table 5. We

Table 4
Total query execution latency speedups on the target workload over PostgreSQL of direct transfer, where ↑ indicates an increase in performance compared to the corresponding instance-optimized model.

	TPC-H → JOB				JOB → TPC-H			
	adhoc-rand	adhoc-slow	repeat-rand	repeat-slow	adhoc-rand	adhoc-slow	repeat-rand	repeat-slow
Bao	**1.07**	**1.19**↑	**1.07**	0.69	**6.35**↑	1.56↑	**1.64**	1.42
COOOL-list	0.97	0.93	0.92	0.82	0.85	**4.70**↑	**1.64**	1.70
COOOL-pair	0.96	1.18	0.89	**0.86**	5.90↑	4.60↑	1.48	**1.82**

Table 5
Total query execution latency speedups of a unified model (trained on both JOB and TPC-H data) over PostgreSQL. The best performance on each workload is in boldface. "↑" indicates that the performance of the unified model is better than that trained on the corresponding single dataset.

	JOB				TPC-H			
	adhoc-rand	adhoc-slow	repeat-rand	repeat-slow	adhoc-rand	adhoc-slow	repeat-rand	repeat-slow
Bao	0.80	0.92	2.79	1.21	5.77↑	1.93	4.83	4.31
COOOL-list	1.00	1.21	**3.24**↑	**1.26**	6.59↑	2.91	**5.37**	**5.53**
COOOL-pair	**1.06**	**1.71**↑	2.90	1.21	**6.73**↑	**3.92**↑	5.34↑	5.51

Table 6
Number of regressions for Bao and COOOL compared with PostgreSQL (unified model)

Setting	Bao	COOOL-list	COOOL-pair
JOB repeat-rand	23	**12**	17
JOB repeat-slow	20	**9**	11
TPC-H repeat-rand	8	**1**	**1**
TPC-H repeat-slow	4	**1**	**1**

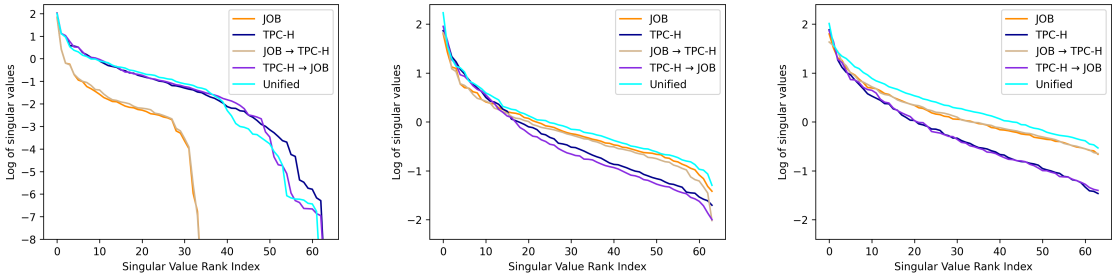
observe that:

- COOOL-pair performs the best in all "adhoc" settings and COOOL-list performs the best in all "repeat" settings. They have similar overall performances, and outperform Bao in almost all scenarios.
- COOOL-pair has the most performance boost when the model is trained using both JOB and TPC-H datasets (unified model) compared with Table 1, especially on TPC-H where the unified model beats the single-instance model in three out of four scenarios.
- Does a different training dataset help? We observe that JOB data help improve the model performance on TPC-H test data, especially under "adhoc" scenarios. While TPC-H training data do not help improve the model performance on JOB test data under most scenarios. We believe this is because JOB queries are more complicated (more nodes and larger depth than TPC-H queries as shown in Table 3).

The individual query performance in "repeat" settings of the unified models are shown in Figure 4, where we also depict queries with an execution latency greater than 1s on PostgreSQL. Combined with the observation in Figure 4, we can get the following conclusions:

- The performances of both COOOL approaches are close to optimal. This echoes our observation that JOB training data help improve the model performances on TPC-H.
- On the other hand, the model performances on JOB test data are hurt by the TPC-H training data. For example, both Bao and COOOL methods have poor performances on query 19d in Figure 4b.
- In terms of the number of regressions in execution time, we list the results in Table 6. Both COOOL-pair and COOOL-list perform better than Bao, and COOOL-list has the lowest number of query regressions for these settings.

Summary on Model Performances Different data distributions in different datasets is a challenge for a unified query optimization model, but COOOL models are able to alleviate this issue to learn a unified model better than the SOTA regression approach to speed up total query execution, alleviate individual query regression, and optimize slow queries. Our experiments have shown the overwhelming advantages of our proposed models. For the single instance scenarios, COOOL-list achieves the best performance in total query execution speedups while COOOL-pair is the best to reduce the number of individual query regressions. When a unified model is trained using multiple datasets, COOOL-list performs the best in "repeat" settings, while COOOL-pair is the best in "adhoc" settings and exhibits no total query execution performance regression. COOOL-pair is the best



(a) Bao (b) COOOL-pair (c) COOOL-list

Figure 5: Singular value spectrum of the plan embedding space for Bao and COOOL in "adhoc-slow" setting of the two workloads in the scenarios of the single instance, workload transfer, and unified model.

Table 7
Comparison of training time required for convergence in the "adhoc-slow" setting

	JOB	TPC-H	Unified
Bao	119.5s	34.5s	265.9s
COOOL-list	167.0s	159.8s	317.1s
COOOL-pair	493.6s	222.8s	894.6s

approach among the three models in terms of **performance** and **stability**.

5.5. Model Comparison and Analysis (RQ4)

In COOOL methods, we use exactly the same plan representation model as Bao to confirm that the improvements are brought by our LTR methods. We analyze the "adhoc-slow" setting of the two workloads in this section.

5.5.1. Model efficiency.

We compare the space and time efficiency of our approaches and the SOTA method.

- Space complexity. The two COOOL models use a TCNN with the same structure and hidden sizes as Bao, the number of parameters for all of them is 132,353, i.e., 529,412 bytes. The storage consumption of the model is about 0.5MB, which is efficient enough in practice.
- The number of training samples. Let N denote the number of training queries. And there are n candidate hint sets in $\mathscr{H}$ for each query. The number of training samples of Bao is Nn. Let m_i denote the number of unique query plans for each query given $\mathscr{H}$, where $1 \leq i \leq N, m_i \leq n$. The number of samples is $\Theta(\sum_{i=1}^{N} \frac{m_i(m_i-1)}{2}) = O(Nn^2)$ for COOOL-pair. For COOOL-list, there are N samples.
- Training time consumption. To intuitively demonstrate the time efficiency of the three methods, we summarize the average training time required for convergence in the "adhoc-slow" settings, as shown in Table 7. COOOL models necessitate more training time for convergence than Bao. Besides, COOOL-pair requires much more time for convergence than COOOL-list due to much more training samples.
- Conclusion. Notwithstanding COOOL-pair and COOOL-list require longer convergence time than Bao, they have the same **inference efficiency** and **the number parameters** as Bao. Therefore, we can conclude that COOOL exhibits comparable practicality to Bao.

5.5.2. A representation learning perspective for plan tree embeddings.

We have conducted extensive experiments and obtained promising results. But it is not clear how the ranking strategies affect the model training. This section aims to provide insight on why the two COOOL models outperform Bao, especially for the unified model training.

Let an h-dimensional vector $\mathbf{z_i}$ denote the embedding for query plan i. We compute the covariance matrix $C \in \mathbb{R}^{h \times h}$ for all plan embeddings obtained by the model. Formally,

$$C = \frac{1}{M} \sum_{i=1}^{M} (\mathbf{z_i} - \overline{\mathbf{z}})(\mathbf{z_i} - \overline{\mathbf{z}})^\top,$$

where M is the number of plans, $\overline{\mathbf{z}} = \frac{1}{M} \sum_{i=1}^{M} \mathbf{z_i}$. Then, we apply singular value decomposition on C s.t. $C = USV^\top$, where $S = diag(\sigma^k)$. Then we can obtain the singular value spectrum in sorted order and logarithmic scale ($\lg(\sigma^k)$). We depict the singular value spectrum of the plan embedding space of the three models learned in

different scenarios in "adhoc-slow" setting, as shown in Figure 5. We have the following observations.

- In Figure 5a, we observe a significant drop in singular values on a logarithmic scale ($\lg(\sigma^k)$) in all tasks. The curve approaches zero (less than 1e-7) in the spectrum, indicating that a dimensional collapse [16] occurs in Bao's embedding space in each of Bao's experimental scenarios. The model learned on JOB even collapsed in half of the plan embedding space. Bao's plan embedding vectors only spans a lower-dimensional subspace, which harm the representation ability of the model and therefore hurt the performance. Moreover, the models in the single instance scenario on different datasets result in a different number of collapsed dimensions. Based on the workload transfer scenario curves, we observe that the plan embedding space of the target workload spans the same number of dimensions as the source workload. So we can conclude that the embedding space is dependent on the training data.
- In Figures 5b and 5c, singular values of the COOOL approaches do not drop abruptly and greater than 1e-7, meaning that there is no collapse in their plan embedding space. The plan embedding methods of Bao and COOOL are the same, and the comparison reflects that the embedding spaces learned from the regression framework and ranking strategies are significantly different.
- The spectrum of Bao's unified model is closer to the spectrum of the learned representation from TPC-H data than that learned from JOB data, whereas the spectra of COOOL-pair and COOOL-list are closer to the spectrum of the representation from JOB data than that learned from TPC-H data. In fact, the number of training samples of TPC-H is greater than that of JOB, but JOB is more complicated than TPC-H (see Table 3). When learning from multiple datasets, Bao's performance is affected more by the datasets with more samples. By contrast, the two COOOL methods are able to learn from sophisticated query plans.
- Singular values are related to the latent information in the covariance matrix. We can get three conclusions from this perspective. First, the overall spectrums of Bao in different settings are lower than COOOL in terms of absolute singular value, which indicates that the representation ability of Bao is worse than COOOL. Second, the curve of Bao's unified model is not the highest in most dimensions, while that of the two COOOL models are just the opposite. It may reveal that the ranking strategies can help models learn latent patterns from two workloads but the regression framework cannot. Third, the range and the trend of the curves of COOOL-pair and COOOL-list are different, which may explain the different performances of the two models in different scenarios.
- The dimensional collapse issue may not greatly hurt single instance optimization since the embedding space is learned from one workload and the latent collapsed dimensions are fixed when the model converges. Because the collapsed dimensions may be different in different datasets, there may be orthogonal situations, which may cause the subspace representation learned in one dataset to be noise in another, resulting in unstable performance. Therefore, it may limit the scalability of machine learning query optimization models.

To sum up, dimensional collapse in plan tree embedding space is a challenge for machine learning query optimization models in maintaining a unified model to learn from different workloads. The two variants of COOOL have the exact same model as Bao but consistently outperform Bao in almost all scenarios and settings. We provide some insight on why COOOL methods outperform Bao from the query plan representation perspective. It may guide researchers to further improve the performance of ML models for query optimization.

6. Related work

6.1. Machine Learning for Query Optimization

Traditional cost-based query optimizers aim to select the candidate plan with the minimum estimated cost, where cost represents the execution latency or other user-defined resource consumption metrics. Various techniques have been proposed in cost-based optimizers [30, 31, 32], such as sketches, histograms, probability models, etc. Besides, they work with different assumptions (e.g., attribute value independence [33], uniformity [34], data independence [35]) and when these cannot be met, the techniques usually fall back to an educated guess [36]. Traditional optimizers have been studied for decades [1], focusing on manually-crafted and heuristic methods to predict costs, estimate cardinality, and generate plans [37]. However, the effect of heuristics depends on data distribution, cardinality estimation, and cost modeling. Therefore, query optimization has been proven difficult to solve [38]. Consequently, cost-based optimizers may generate plans with poor performance.

Therefore, the database community has attempted to apply machine learning techniques to solve these issues. For example, some efforts introduced reinforcement learning and Monte Carlo Tree Search (MCTS) to

optimize the join order selection task [39, 40, 41, 42, 43], and some studies use neural networks to accomplish cost modeling and cardinality estimation [7, 8, 9, 10]. Refer to surveys [44, 45] for more ML for query optimization details. These works depict that the elaborated models can improve the performance of parts of those components in query optimization. However, they may not improve the performance of optimizers since none of them demonstrate that the performance improvement of a single component can actually result in a better query plan [46].

In recent years, some studies have attempted to build end-to-end ML models for query optimization [6, 3, 14]. They all use a TCNN to predict the cost/latency of query plans and leverage a deep reinforcement learning framework to train the model. Though they have made improvements compared with the previous works of replacing some components of the optimizer with ML models, an optimizer should be evaluated in multiple dimensions [2], and some of the most concerned aspects are *practicality*, *explainability*, *performance*, *generalizability*, whether it is *data/schema agnostic*, etc. Neo [6] learns row vector embeddings from tables, so it needs to maintain row vector embeddings for different data. Balsa [14] can not treat advanced SQL features (e.g., sub-queries) due to its dependency, which limits its generalizability. Furthermore, Neo and Balsa are less practical than Bao [3]. Bao introduced a novel approach to end-to-end query optimization that recommends per-query SQL hints over an existing optimizer. SQL hints can limit the search space of existing optimizers and each set of hints corresponds to a plan, which makes it practical in real scenarios.

6.2. Learning-To-Rank

The objective of LTR is to develop ML models that can automatically rank items according to their relevance or significance (such as relevance score or query latency). LTR is an active research topic [47] and has many applications in information retrieval [17], meta-search engines [48], recommendation systems [49], preference learning [50], etc. A search engine or a recommendation system usually needs to rank a large and variable number of items (webpages, movies, for example). The SQL hint recommendation problem is more like the preference learning problem, where a relatively small and fixed number of items (SQL hints in our case), are to be ranked for each user (SQL query in our case). Many studies have been conducted in the field of LTR (see the survey [51, 52, 17] for a comprehensive review). We briefly discuss Plackett-Luce model [18, 19] in this section. Plackett-Luce model, which was later used as a listwise loss function in information retrieval [17] and softmax function in classification tasks [20], is one of the most popular models in preference learning, which fits the task of SQL hint recommendation well. It is a listwise model but can be learned efficiently with pairwise methods [24, 23], by breaking the full rankings into pairwise comparisons and minimizing a pairwise loss function. Similar to [24] and [23], we maximize the marginal pairwise likelihood for its simplicity.

7. Conclusion

In this paper, we propose COOOL that predicts cost orders of query plans to cooperate with DBMS by LTR techniques to recommend SQL hints for query optimization. COOOL includes both pairwise and listwise methods, which can improve overall query execution speed, especially on slow queries, and alleviate individual query regression. Moreover, COOOL makes a step forward to maintaining a unified end-to-end query optimization ML model on two datasets. We shed some light on why COOOL approaches outperform Bao from the representation learning perspective, and the elaborated analysis may provide preliminary for large-scale pre-trained query optimization models. While COOOL methods are promising, they cannot estimate the cost of the recommended query plan, or quantitatively compare the costs of two query plans.

For future work, we plan to investigate the evaluation metrics for ranking candidate plans that differ by multiple orders of magnitude in execution latency for query optimization, which facilitates the introduction of state-of-the-art LTR techniques. LTR is still an active research field in recent years, so there are considerable future works to be explored in query optimization. Besides, developing a large-scale pre-trained query optimization model to improve query plans from multiple datasets and quantitatively comparing the costs of different query plans accurately are also challenging future directions.

References

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access path selection in a relational database management system, in: Proceedings of the 1979 ACM SIGMOD international conference on Management of data, 1979, pp. 23–34.

[2] D. Tsesmelis, A. Simitsis, Database optimizers in the era of learning, in: 2022 IEEE 38th International Conference on Data Engineering (ICDE), IEEE, 2022, pp. 3213–3216.

[3] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, T. Kraska, Bao: Making learned query optimization practical, ACM SIGMOD Record 51 (2022) 6–13.

[4] L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin, Convolutional neural networks over tree structures for

programming language processing, in: Thirtieth AAAI conference on artificial intelligence, 2016.

[5] W. R. Thompson, On the likelihood that one unknown probability exceeds another in view of the evidence of two samples, Biometrika 25 (1933) 285–294.

[6] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, N. Tatbul, Neo: A learned query optimizer, Proc. VLDB Endow. 12 (2019) 1705–1718. URL: http://www.vldb.org/pvldb/vol12/p1705-marcus.pdf. doi:10.14778/3342263.3342644.

[7] H. Liu, M. Xu, Z. Yu, V. Corvinelli, C. Zuzarte, Cardinality estimation using neural networks, in: Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering, 2015, pp. 53–59.

[8] J. Sun, G. Li, An end-to-end learning-based cost estimator, Proc. VLDB Endow. 13 (2019) 307–319. URL: http://www.vldb.org/pvldb/vol13/p307-sun.pdf. doi:10.14778/3368289.3368296.

[9] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, I. Stoica, Deep unsupervised cardinality estimation, arXiv preprint arXiv:1905.04278 (2019).

[10] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, I. Stoica, Neurocard: one cardinality estimator for all tables, Proceedings of the VLDB Endowment 14 (2020) 61–73.

[11] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, S. B. Zdonik, Learning-based query performance modeling and prediction, in: 2012 IEEE 28th International Conference on Data Engineering, IEEE, 2012, pp. 390–401.

[12] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, J. F. Naughton, Predicting query execution time: Are optimizer cost models really unusable?, in: 2013 IEEE 29th International Conference on Data Engineering (ICDE), IEEE, 2013, pp. 1081–1092.

[13] F. Ventura, Z. Kaoudi, J. A. Quiané-Ruiz, V. Markl, Expand your training limits! generating training data for ml-based data management, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 1865–1878.

[14] Z. Yang, W. Chiang, S. Luan, G. Mittal, M. Luo, I. Stoica, Balsa: Learning a query optimizer without expert demonstrations, in: Z. Ives, A. Bonifati, A. E. Abbadi (Eds.), SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022, ACM, 2022, pp. 931–944. URL: https://doi.org/10.1145/3514221.3517885. doi:10.1145/3514221.3517885.

[15] D. Zhang, J. Yin, X. Zhu, C. Zhang, Network representation learning: A survey, IEEE transactions on Big Data 6 (2018) 3–28.

[16] T. Hua, W. Wang, Z. Xue, S. Ren, Y. Wang, H. Zhao, On feature decorrelation in self-supervised learning, in: 2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021, IEEE, 2021, pp. 9578–9588. URL: https://doi.org/10.1109/ICCV48922.2021.00946. doi:10.1109/ICCV48922.2021.00946.

[17] T.-Y. Liu, Learning to rank for information retrieval, Foundations and Trends in Information Retrieval 3 (2009) 225–331.

[18] R. L. Plackett, The analysis of permutations, Journal of the Royal Statistical Society Series C: Applied Statistics 24 (1975) 193–202.

[19] R. D. Luce, Individual choice behavior, 1959.

[20] B. Gao, L. Pavel, On the properties of the softmax function with application in game theory and reinforcement learning, arXiv preprint arXiv:1704.00805 (2017).

[21] F. Xia, T.-Y. Liu, J. Wang, W. Zhang, H. Li, Listwise approach to learning to rank: theory and algorithm, in: Proceedings of the 25th international conference on Machine learning, 2008, pp. 1192–1199.

[22] H. Azari Soufiani, W. Chen, D. C. Parkes, L. Xia, Generalized method-of-moments for rank aggregation, Advances in Neural Information Processing Systems 26 (2013).

[23] Z. Zhao, L. Xia, Composite marginal likelihood methods for random utility models, in: International Conference on Machine Learning, PMLR, 2018, pp. 5922–5931.

[24] A. Khetan, S. Oh, Data-driven rank breaking for efficient rank aggregation, in: International Conference on Machine Learning, PMLR, 2016, pp. 89–98.

[25] T. M. Mitchell, The need for biases in learning generalizations, Citeseer, 1980.

[26] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, T. Neumann, How good are query optimizers, really?, Proceedings of the VLDB Endowment 9 (2015) 204–215.

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

[28] B. Xu, N. Wang, T. Chen, M. Li, Empirical evaluation of rectified activations in convolutional network, arXiv preprint arXiv:1505.00853 (2015).

[29] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015.

[30] Y. E. Ioannidis, V. Poosala, Balancing histogram optimality and practicality for query result size estimation, Acm Sigmod Record 24 (1995) 233–244.

[31] G. Cormode, M. Garofalakis, P. J. Haas, C. Jer-

maine, et al., Synopses for massive data: Samples, histograms, wavelets, sketches, Foundations and Trends® in Databases 4 (2011) 1–294.
[32] G. Cormode, M. Garofalakis, Histograms and wavelets on probabilistic data, IEEE Transactions on Knowledge and Data Engineering 22 (2010) 1142–1157.
[33] S. Christodoulakis, Implications of certain assumptions in database performance evauation, ACM Transactions on Database Systems (TODS) 9 (1984) 163–186.
[34] Y. E. Ioannidis, S. Christodoulakis, Optimal histograms for limiting worst-case error propagation in the size of join results, ACM Transactions on Database Systems (TODS) 18 (1993) 709–748.
[35] A. Deshpande, M. Garofalakis, R. Rastogi, Independence is good: Dependency-based histogram synopses for high-dimensional data, ACM SIGMOD Record 30 (2001) 199–210.
[36] V. Poosala, Y. E. Ioannidis, Selectivity estimation without the attribute value independence assumption, in: VLDB, volume 97, 1997, pp. 486–495.
[37] Y. E. Ioannidis, Query optimization, ACM Computing Surveys (CSUR) 28 (1996) 121–123.
[38] G. Lohman, Is query optimization a "solved" problem, in: Proc. Workshop on Database Query Optimization, volume 13, Oregon Graduate Center Comp. Sci. Tech. Rep, 2014, p. 10.
[39] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, I. Stoica, Learning to optimize join queries with deep reinforcement learning, arXiv preprint arXiv:1808.03196 (2018).
[40] R. Marcus, O. Papaemmanouil, Deep reinforcement learning for join order enumeration, in: Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, 2018, pp. 1–4.
[41] I. Trummer, J. Wang, Z. Wei, D. Maram, S. Moseley, S. Jo, J. Antonakakis, A. Rayabhari, Skinnerdb: Regret-bounded query evaluation via reinforcement learning, ACM Transactions on Database Systems (TODS) 46 (2021) 1–45.
[42] J. Zhang, Alphajoin: Join order selection à la alphago, in: PVLDB-PhD, 2020.
[43] X. Zhou, G. Li, C. Chai, J. Feng, A learned query rewrite system using monte carlo tree search, Proceedings of the VLDB Endowment 15 (2021) 46–58.
[44] G. Li, X. Zhou, L. Cao, Ai meets database: Ai4db and db4ai, in: Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 2859–2866.
[45] X. Zhou, C. Chai, G. Li, J. Sun, Database meets artificial intelligence: A survey, IEEE Transactions on Knowledge and Data Engineering 34 (2020) 1096–1116.
[46] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, T. Neumann, Query optimization through the looking glass, and what we found running the join order benchmark, The VLDB Journal 27 (2018) 643–668.
[47] H.-T. Yu, R. Piryani, A. Jatowt, R. Inagaki, H. Joho, K.-S. Kim, An in-depth study on adversarial learning-to-rank, Information Retrieval Journal 26 (2023) 1.
[48] C. Dwork, R. Kumar, M. Naor, D. Sivakumar, Rank aggregation methods for the web, in: Proceedings of the 10th international conference on World Wide Web, 2001, pp. 613–622.
[49] A. Karatzoglou, L. Baltrunas, Y. Shi, Learning to rank for recommender systems, in: Proceedings of the 7th ACM Conference on Recommender Systems, 2013, pp. 493–494.
[50] Z. Zhao, A. Liu, L. Xia, Learning mixtures of random utility models with features from incomplete preferences, arXiv preprint arXiv:2006.03869 (2022).
[51] A. Rahangdale, S. Raut, Machine learning methods for ranking, International Journal of Software Engineering and Knowledge Engineering 29 (2019) 729–761.
[52] J. Guo, Y. Fan, L. Pang, L. Yang, Q. Ai, H. Zamani, C. Wu, W. B. Croft, X. Cheng, A deep look into neural ranking models for information retrieval, Information Processing & Management 57 (2020) 102067.

A. Online Resources

The source code of our paper is available at https://github.com/xianghongxu/COOOL.