# Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection

Xiang Yu
Department of Computer Science, Tsinghua University
x-yu17@mails.tsinghua.edu.cn

Chengliang Chai
Department of Computer Science, Tsinghua University
ccl@tsinghua.edu.cn

Guoliang Li
Department of Computer Science, Tsinghua University
liguoliang@tsinghua.edu.cn

Jiabin Liu
Department of Computer Science, Tsinghua University
liujb19@mails.tsinghua.edu.cn

## ABSTRACT

Traditional cost-based optimizers are efficient and stable to generate optimal plans for simple SQL queries, but they may not generate high-quality plans for complicated queries. Thus learning-based optimizers have been proposed recently that can learn high-quality plans based on past experiences. However, learning-based optimizers cannot work well for dynamic workloads that have different distributions with training examples.

In this paper, we propose a hybrid optimizer that adopts the advantages and avoids the shortcomings of these two types of optimizers, which first generates high-quality candidate plans from each type of optimizers and then selects the best plan from the candidates. There are two challenges. (1) How to generate high-quality candidates? We propose a hint-based candidate generation method that leverages the learning-based method to generate highly beneficial hints and then uses a cost-based method to supplement the hints to generate complete plans as candidates. (2) How to evaluate different candidate plans and select the best one? We propose an uncertainty-based optimal plan selection model, which predicts the execution time and the uncertainty for each plan. The uncertainty reflects the confidence of the execution time prediction. We select the plan using the uncertainty model. Experiment results on real datasets showed that our method outperformed the state-of-the-art baselines, and reduced the total latency by 25% and the tail latency by 65% compared to PostgreSQL.

**PVLDB Reference Format:**
Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. Cost-based or Learning-based? A Hybrid Query Optimizer for Query Plan Selection. PVLDB, 15(13): 3924 - 3936, 2022.
doi:10.14778/3565838.3565846

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/yxfish13/HyperQO.

Chengliang Chai and Guoliang Li are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 15, No. 13 ISSN 2150-8097.
doi:10.14778/3565838.3565846

## 1 INTRODUCTION

Query optimization is a fundamental problem that has been widely studied for many years in DBMS [2, 34, 37, 43]. Besides traditional cost-based optimizers, learning-based optimizers have been proposed recently. These two types of optimizers have their limitations and advantages. To alleviate their limitations, we propose a hybrid optimizer that takes the essence and discards the dregs.

**Cost-based Optimizer.** The advantage of cost-based optimizers is efficient and stable to generate a good plan based on a cost estimation model. However, they may select sub-optimal plans for complicated queries with multiple joins, because the cost may not well capture the plan quality (especially due to accumulate cost errors for multiple joins).

**Learning-based Optimizer.** Learning-based optimizers have been proposed recently [21, 29–31, 42, 44, 48–50, 53], which utilize machine learning techniques to learn high-quality plans from past experiences. For example, DQ [21], ReJoin [30] and RTOS [50] use reinforcement learning to select a good join order. Neo [31] uses Tree-CNN to achieve an end-to-end learning-based optimizer. The advantage of learning-based optimizers is that they can optimize complicated queries by learning from training examples. They work well for static workloads (i.e., the testing workloads have similar distributions with training workloads), but they cannot support dynamic workloads well where the testing workloads are out of distributions with the training workloads.

**Adaptive Hybrid Optimizer.** To address the limitations of the two types of optimizers, it is challenging to design a hybrid optimizer that adopts their advantages and avoids their shortcomings. An intuitive solution is to obtain an optimized plan for each type of optimizers and compare them to get a better one. However, this straightforward method still cannot address the adaptability problem of learning-based optimizers (i.e., cannot get good plans for dynamic workloads). To address this problem, we propose to leverage the *hint* functionality provided by the DBMS optimizer to adaptively generate plans. First, we use the learning-based optimizers to generate a *leading* prefix of a plan as a hint, based on which we use the hint functionality provided by optimizers to generate a complete plan as a candidate. Finally, we compare the candidates with those generated by the cost-based optimizers and select the better one. Note that the hints usually have better quality than complete plans, because learned optimizers are usually error-prone for complicated queries (multiple joins) when training data is not enough or not similar to query workloads. Thus, we combine the two types of optimizers to get an efficient, stable, adaptive optimizer.

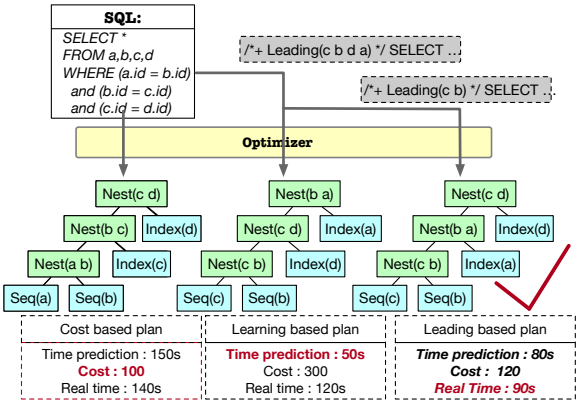



**Figure 1: Cost-based vs Learning-based vs Hybrid Optimizers.**

**Motivating Example.** As shown in Figure 1, given a query $Q$, the cost-based optimizer generates a plan (with a join order *abcd*) with the smallest cost (100), but the real execution time is not optimal because of the cost estimation error. The learning-based optimizer selects the plan (with a join order *cbda*) with the best estimated performance (50s). However, this plan is also not good, because this query is not captured in training examples. The plans generated with multiple joins are error-prone if the query is out-of-distribution. Fortunately, we observe that those well-performed plans share similar prefixes of join orders. Prefixes are very important in common left-deep plans. Hence, it is possible to generate plans by recommending good prefixes. For example, we use the learned model to find a good prefix *cb* rather than a complete join order and feed it to the cost-based optimizer to supplement the subsequent plan, and the plan *cbda* is obtained. *cb* is a good prefix for a learned model, *cbda*(90s) and *cbad*(120s) are better plans than *abdc*(140s) because of the inaccurate estimation of join *ab* by the cost model. The predicted execution time of *cbad* (80s) is smaller than the cost-based optimizer's plan *abcd* (150s), and its cost (120) is also smaller than the complete join order *cbda* (300) produced by the learning-based optimizer. The execution time of the plan *cbad* is the smallest (90s). Hence, we propose to discover good candidate plans by utilizing leadings (prefixes of plans) as *hints* and select the best plan from different candidates.

**Challenges.** There are two main challenges in designing a hybrid optimizer. The first is how to generate high-quality candidate plans using hints. We propose a method by computing the benefits of hints. Considering the large space of possible hints, we model them as a tree and use the Monte-Carlo Tree Search (MCTS) to efficiently explore good hints. For each hint on the tree, we compute a *benefit* score, and the hints with high benefit scores are potentially expected to generate good candidate plans. The second is how to evaluate different candidate plans and select the best one. We propose an uncertainty-based optimal plan selection model, which predicts the execution time of each candidate plan as well as an uncertainty that measures the confidence of this plan, and selects the plan by considering the both factors.

**Contributions.** We summarize our contributions as below.

(1) We propose an adaptive hybrid optimizer that combines the cost-based and learning-based optimizer through the hint functionality.

(2) We design a hint-based candidate plans generation approach, which uses the MCTS to accelerate the generation process, while still producing high-quality plans.

(3) We design an uncertainty-based optimal plan selection approach, which considers the uncertainty of the learning-based plans to avoid wrong selection of plans due to inaccurate estimates of the learned method (e.g. out-of-distribution queries). This method can select the optimal plan from the candidates produced by the learning-based and cost-based optimizer.

(4) Experiment results on real datasets showed that our system outperformed the state-of-the-art baselines, reduced the total latency by 25% and the tail latency by 65% compared to PostgreSQL.

## 2 THE FRAMEWORK OF HYBRIDQO

In this section, we first introduce some preliminaries and then propose the overall framework of our system HybridQO.

### 2.1 Preliminaries

**Leading Hint** is a partial prefix plan (a.k.a. leading) of a complete plan, which is a type of hint that is supported in many database systems (e.g., PostgreSQL, MySQL, Oracle, etc.). Formally, given an SQL query $Q$, we use $p_Q^l$ to denote a leading hint with length $l$. When $l = |Q|$ (the query length), it represents a complete join order. Given a query, DBAs can specify a leading hint (e.g., a sequence of joined tables for this query) and the optimizer will extend the leading hint to generate a complete plan.

**Example 1:** Consider the query Q as shown in Figure 1, which contains 4 tables $\{a,b,c,d\}$ with the join relation $a \bowtie b \bowtie c \bowtie d$. If a DBA wants to specify that tables $c$ and $b$ are joined first. She can provide a leading hint of length 2 to the optimizer, *i.e.*, $p_Q^2 = (c, b)$. In PostgreSQL, by providing "/*+ leading(c b) */ SELECT ...", the optimizer will generate an execution plan with $c$ and $b$ joining first, where "leading(c b)" is the hint syntax in PostgreSQL. Other database systems also support this functionality. □

*Remark.* In this paper, given the leading hint (sequence $p_Q^l$ of tables), we just consider the left-deep join, which is a structure frequently used in existing database optimizer (e.g., MySQL, PostgreSQL).

**Leading Hint Tree.** Given a query $Q$, the number of possible leading hints can be very large, which can be naturally organized as a tree. To discover appropriate hints efficiently, we build the leading hint tree, denoted by $T_Q$ to accelerate the leading hint search. In this part, we describe the learning tree structure, and introduce how to adaptively construct it and search on it later in Section 3.1.

A leading hint tree $T_Q$ is a tree of height $|Q|$, where each node of $T_Q$ is a table in $Q$. $T_Q$ begins with a root node $r$ without any hint applied to the query, and the children of the root are all tables in $Q$. For each node $v$, its children are all the tables that can be joined with $v$, except the ones that have been joined with it, *i.e.*, the ancestors of $v$. Hence, each node $v$ corresponds to a leading hint, that is, the path from $r$ to $v$ with length $|Q|$. Each leaf node corresponds to a complete join order. It is expensive to construct a full leading hint tree, we will discuss it later in Section 3.1.

**Example 2:** As shown in Figure 2, given the query $Q$ with the join schema $a \bowtie b \bowtie c \bowtie d$, a leading hint tree of height 4 can be constructed. Since there are 4 tables, the second level contains the

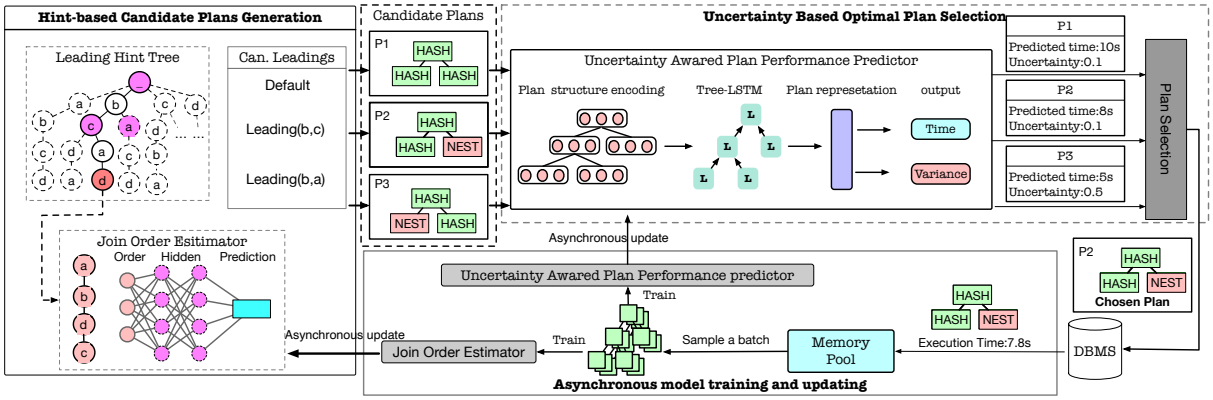



**Figure 2: System Framework of** `HybridQO`**.**

nodes (tables) $a$, $b$, $c$ and $d$. Since $b$ can be joined with $a$ and $c$, $a$ and $c$ are the children of $b$. For the purple node $c$, it corresponds to a leading hint $(b, c)$. For the red leaf node $d$, it corresponds to the leading hint $(b, c, a, d)$, which is also a complete join order. □

## 2.2 `HybridQO` Framework

*2.2.1 Overview of* `HybridQO`*.* Our basic idea is to combine the learning-based optimizer and the cost-based optimizer to select a good plan. To this end, we first select some good leading hints from the leading hint tree. As it is expensive to fully traverse the leading hint tree, we propose a learning-based leading hint selection method *i.e.*, MCTS, to select leading hints. Then, we use the hint functionality in DBMS to generate complete plans as candidate plans. Next, we leverage a learning-based uncertainty model to select the best plan from candidate plans. Finally, we run the selected plan to answer the query and use this plan (and its running time) as a training example to update the two learning models. As shown in Figure 2, the entire framework consists of three key modules.

[*1. Candidate plans generation.*] As we know, a well-performed plan depends on a good join order, and we observe that good join orders usually share similar prefixes. Thus we build a leading hint tree for a given query. Then we first select leading hints from the leading hint tree by MCTS which is a typical tree search algorithm. We will discuss how to use MCTS to select leading hints later. For each selected leading hint, *i.e.*, the left-deep join orders of the prefixes, we leverage the cost-based optimizer in DBMS to produce a complete query plan. For each hint, we generate a candidate plan.

[*2. Optimal plan selection.*] Secondly, we leverage a learning-based uncertainty model to select the best plan among these candidate plans, because the estimated cost may not be accurate enough due to the bias of statistics information. *Especially, the model not only predicts the query performance, but also computes the uncertainty of the prediction.* The uncertainty measures how confident is the model for the prediction. Therefore, intuitively, we should select the optimal plan with high performance as well as low uncertainty.

[*3. Incremental model training and updating.*] After the plan is selected, we execute the query with this plan, and the result (with execution time) will be used as a training example to improve the model incrementally. Then the updated model can be utilized to select the candidate plans and the optimal one.

**Challenges.** There are two main challenges *w.r.t.* our `HybridQO` framework. (1) Given the large search space of possible join orders (leading hints), how to efficiently perform on the leading hint tree to discover potentially well-performed plans. (2) Given the candidate plans, how to design a model that can simultaneously predict the accurate performance as well as the uncertainty.

*2.2.2 Hint-based Candidate Plans Generation.* Given a query $Q$, we summarize how to generate well-performed candidate plans using $T_Q$. Recap that on the tree, each leaf node corresponds to a complete join order, but it is prohibitively expensive to enumerate all possible orders and estimate their performance. Hence, MCTS is applied to discover good orders. To evaluate the performance of a join order(leaf nodes), as shown in Figure 2, we utilize an RNN-based join order estimator to predict the performance. The training data of the model is derived from the extracted complete join order of a plan that has been executed in DBMS.

With the help of $T_Q$ and the join order estimator, we can utilize the learned-based estimator to obtain some good complete orders. Afterwards, a straightforward way is to leverage the cost-based optimizer to supplement the physical operators, producing some complete query plans and then selecting the best one. However, there are two limitations. (1) The learned-based estimator is based on a join order rather than a complete plan, so the estimated performance can just be used to filter out these bad orders, but not a very precise estimation. (2) These join orders obtained from the learned-based estimator are error-prone, because, for a complete order, an error in the middle node will propagate the error. Fortunately, we find that those good join orders usually share similar prefixes (leadings), and thus to address these limitations, we select $H$ good leading hints from the tree, based on which $H$ candidate plans are generated. Obviously, there is a trade-off to select a good value of $H$ – a larger $H$ provides more plan options, provides high possibilities to get the optimal plan, but also takes longer plan generation time; while a smaller $H$ takes shorter plan generate time but has low possibilities to get the optimal plan. We will discuss how to select $H$ later.

**Algorithm 1**: HybridQO Framework

**Input**: A query $Q$ to be executed.
**Output**: An optimal query plan.

1 Candidate join orders $\mathcal{O}$ = MCTS($Q$, $T_Q$, JoinOrderEstimator);
2 Candidate hints $\mathcal{H}$ = CandHints($\mathcal{O}$), $H = |\mathcal{H}|$;
3 $\mathcal{P} = \emptyset$; // set of candidate plans.
4 **for** *each hint* $h \in \mathcal{H}$ **do**
5  $p$ = CostOptimizer($h$);
6  Add $p$ to $\mathcal{P}$;
7 **for** *each* $p \in \mathcal{P}$ **do**
8  $T_p, U_p$ = PlanEsModel($p$);
9  // estimate the performance and uncertainty of a plan.
10 $p^*$ = OptimalPlanSel($\{(T_p, U_p) | p \in \mathcal{P}\}$);
11 $T_{p^*}$ = Execute($p^*$);
12 Update PlanEsModel by ($p^*$, $T_{p^*}$);
13 Update JoinOrderEstimator by (ExtractOrder($p^*$), $T_{p^*}$);
14 **return** $p^*$;

As the leading hints are shorter than the complete order, the second limitation can be alleviated. Then we use the cost-based optimizer to supplement these hints, so as to generate relatively well-performed candidate plans. Thus the first limitation is also be addressed when we further conduct more precise performance estimation on these candidate plans, which will be introduced next.

**Advantage.** Based on the aforementioned modules, we elaborate the key advantage of HybridQO. For an incoming query, if its distribution has been captured by the learning-based model from previous examples, the join order estimator and optimal plan selection model can work together to generate the optimal plan confidently (with low uncertainty). However, in practice, it is common that the query is out-of-distribution. Thus, although the learning-based model may output some high performance plans, the predictions are likely to be inaccurate (with high uncertainties). In this case, we tend to drop these plans and use the ones that are generated by the cost-based optimizer, which relies on the statistics information rather than the training examples. In a nutshell, our proposed framework can well handle the dynamic online scenario, which can *generate well-performed query plans whatever the newly coming queries are out-of-distribution or not.*

**Overall Algorithm.** Algorithm 1 shows the overall process of HybridQO. Given a query $Q$, HybridQO first leverages the MCTS and join order estimation model to discover some good orders on the leading hint tree (Line 1), and then select $H$ hints from these orders (Line 2). For each hint, we use the cost-based optimizer to generate a complete query plan as candidates (Line 5-6). Then given each candidate plan, we use the plan estimation model to estimate the performance as well as the uncertainty (Line 8), based on which we select the optimal plan (Line 10). Next, we execute the query using the plan (Line 11). Afterwards, we use the execution result to incrementally update the above two models. Note that the steps are conducted separately with the online plan selection and execution, which does not influence the query efficiency.

# 3 CANDIDATE PLANS GENERATION

In this section, we aim to leverage the cost-based optimizer to generate $H$ well-performed candidate plans from $H$ hints, and thus these hints should be also well-performed.

## 3.1 MCTS in Candidate Plans Generation

In this subsection, we illustrate how to conduct MCTS to efficiently search good join orders and then generate good hints, so as to generate well-performed candidate plans. Since the search space of all possible orders is large, we use MCTS to search on $T_Q$, The motivation of using MCTS is that it follows an *exploration-exploitation* strategy, where we will focus more on the directions that have led to join orders with high estimated performance (*i.e.*, *exploitation*), and will also pay attention to the directions that are rarely picked (*i.e.*, *exploration*). To achieve this, we first define the node utility $\mathcal{U}(v)$, which takes both exploration and exploitation into account, so as to guide the tree search.

**Node utility** $\mathcal{U}(v)$ is computed based on the following two factors, considering the exploitation and exploration respectively.
[*1. Node benefit* $\mathcal{B}(v)$]: each node $v$ has a benefit score $\mathcal{B}(v)$ which indicates the expected performance of a complete join order passing through $v$. Thus, the higher $\mathcal{B}(v)$, the more likely we are to select the hint corresponding $v$ as a candidate. For a leaf node, $\mathcal{B}(v)$ is directly computed by the normalized estimated performance of the join order corresponding to $v$. We will introduce how to estimate the performance using the join order estimator in Section 3.2. Then for a non-leaf node $v$, suppose that $L(v)$ denotes the set of leaf nodes that are children of $v$ and have been estimated by the join order estimator. Then $\mathcal{B}(v) = \frac{1}{|L(v)|}\sum_{v' \in L(v)} \mathcal{B}(v')$. Note that if a leaf node is visited and estimated multiple times, we will also consider these estimation results in the above benefit computation.
[*2. Access frequency* $\mathcal{F}(v)$ ]: we use $\mathcal{F}(v)$ to denote the number of times that the node $v$ has been visited during the tree search all the way to the leaf nodes. Focusing on the node with high benefit is likely to result in the local optimum, so we also attempt to visit the nodes with low $\mathcal{F}(v)$.

Considering the factors above, following the commonly-used upper confidence bound (UCB) [3] based solution, we define $\mathcal{U}(v)$ as follows by combining the node benefit and access frequency.

$$\mathcal{U}(v) = \mathcal{B}(v) + \gamma\sqrt{\frac{\ln(\mathcal{F}(v_f))}{\mathcal{F}(v)}} \tag{1}$$

where $v_f$ denotes the father node of $v$ and $\gamma$ is the hyper-parameter to achieve the trade-off between exploration and exploitation. Consequently, we will iteratively pick the node with the highest utility to expand the search on $T_Q$. Intuitively, we can see that the UCB-based solution well handle the exploitation-exploration trade-off, where it not only focuses more on the nodes with large benefit (*i.e.*, $\mathcal{B}(v)$), but also the ones with low access frequency (*i.e.*, $\sqrt{\frac{\ln(\mathcal{F}(v_f))}{\mathcal{F}(v)}}$). Next, more concretely, we use an example to show how to conduct the MCTS based on the definition of node utility.

**Monte-Carlo tree search.** Figure 3 shows the entire MCTS workflow in HybridQO. Note that although we can build the entire $T_Q$ at the beginning, it is not necessary because the size of $T_Q$ is large,

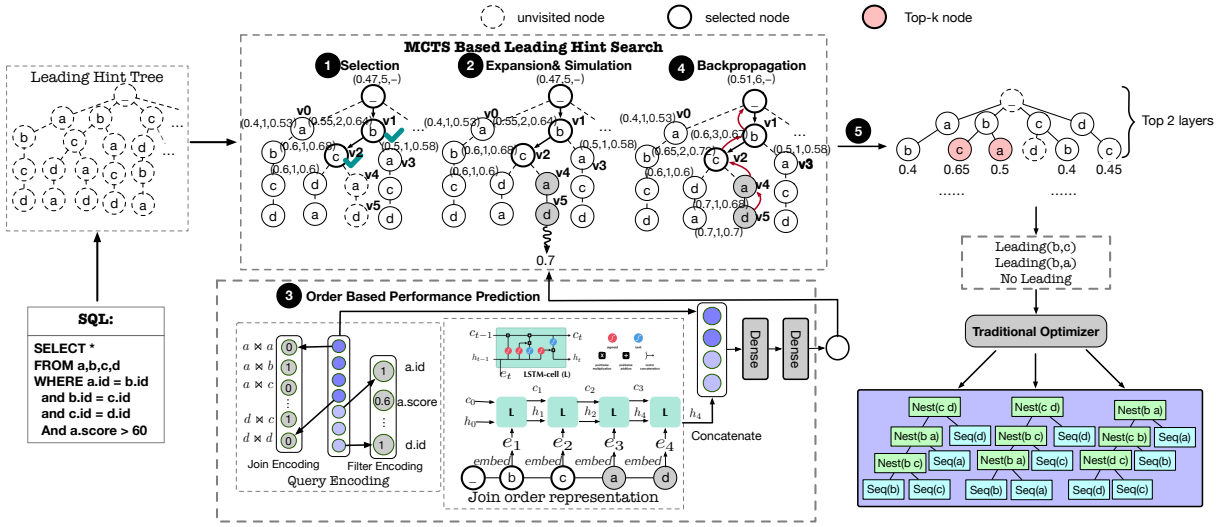



**Figure 3: Candidate Plan Generation Framework with MCTS and Learning-based Join Order Prediction.**

and we can continuously expand the tree during the search. For each node $v$, we show a triple $(\mathcal{B}(v), \mathcal{F}(v), \mathcal{U}(v))$ on the example. Initially, we initialize the triple of the root node as (0,0,0), and then MCTS produces only one path from the root node to the leaf node in a single search, and repeats it several times. The workflow of MCTS consists of the following 5 steps.

[**Step 1.** *Node selection*] This step iteratively selects the child node with the largest utility from the root, which repeats until a node has a child that has not been expanded. As shown in Figure 3, the search starts from the root. Next, $v_1$ is the child of the root with the largest utility (0.64), and $v_1$ is selected. Then, since $\mathcal{U}(v_2) > \mathcal{U}(v_3)$, $v_2$ is selected. Afterwards, since $v_2$ has a node $a$ that has not expanded, the node selection process stops and comes to the next step.

[**Step 2.** *Tree expansion*] For the child nodes of $v$ that have not been expanded yet, we randomly select a child to expand. Since this is a new node, its children must not have been expanded, and thus we continue to expand iteratively until a leaf node. For example, $v_4$ is a not expanded child node of $v_2$, and we create $v_4$. Then we create $v_5$, which is a leaf node corresponding to a join order $(b, c, a, d)$.

[**Step 3.** *Join order estimation*] It is used to predict the execution time of the plan, *i.e.*, corresponding to a join order, where $v$ is the leaf node. For example,we estimate that $f_\theta^j(v_5) = 90s$, and $\mathcal{B}(v_5) = 0.7$. This benefit value will be utilized to update the utilities of its ancestors. We will discuss the join order estimator in Section 3.2.

[**Step 4.** *Utility update*] Based on the benefits of leaf nodes, we will update the triples of non-leaf nodes, and thus further MCTS search can be improved. For example, currently we are at the leaf node $v_5$ corresponding to the path $[v_1, v_2, v_4, v_5]$. Hence, the access frequency of $v_2$ plus one, *i.e.*, $\mathcal{F}(v_2) = 2$. At the same time, we update $\mathcal{B}(v_2) = \frac{0.6+0.7}{2} = 0.65$, and its utility is updated to $\mathcal{U}(v_2) = 0.72$. Iteratively, we can update the triple of $v_1$ to $(0.6, 3, 0.67)$.

[**Step 5.** *Termination*] The above 4 steps iterate and many join orders as well as the benefits of hints are estimated. With the number of iterations increasing, the benefit estimation will be more accurate. However, it is not practical to visit all nodes of $T_Q$ because of the large size. Hence, we limit the search number by a budget $b$. Once the budget is used up, the search terminates.

**Top-$H$ candidate hints.** After the MCTS process terminates, we select the top-$H$ nodes with the largest benefit scores (*i.e.*, $\mathcal{B}$) as the candidate hints. For example, as shown in Figure 3, suppose that $H = 3$, and we select hints $(b, c)$, $(b, a)$ and $(d, c)$ as candidates with leading length is 2, because hopefully they can lead to good join orders that can generate well-performed query plans.

## 3.2 Performance Prediction for Join Orders

In this section, we build a join order estimator to predict the query performance. In general, it feeds the features of the query and the join order into an RNN, and outputs the estimated performance.

**Query encoding** aims to encode the query $Q$, which considers the features of join predicates ($E_Q^J$) and filter predicates ($E_Q^F$) of $Q$. More concretely, $E_Q^J$ is a $n \times n$ matrix, where $n$ is the number of tables. Each element $E_Q^J[i, j]$ is either 1 (the $i$-th and the $j$-th table are joined in $Q$) or 0 (the $i$-th and the $j$-th table are not joined). $E_Q^J$ is a vector of length $m$, where $m$ is the number of columns in the database. $E_Q^J[k]$ denotes the selectivity of the filter predicate in $Q$ for the $k$-th column, with a default value of 1. Consequently, we represent the feature of $Q$, denoted by $E_Q$, by concatenating $E_Q^J$ (we will flatten this 2-D matrix) and $E_Q^F$.

**Example 3:** As shown the query $Q$ in Figure 3, since $a$ joins with $b$ but does not join with $c$, $E_Q^J[1, 2] = 1$ and $E_Q^J[1, 3] = 0$. Since $Q$ has the predicate `a.score>60` with a selectivity 0.6, so $E_Q^F[1] = 0.6$. □

**Join order representation.** As discussed in Section 2.1, each join order corresponds to a sequence of $n = |Q|$ tables, denoted by $[t_1, t_2, ..., t_n]$. Naturally, we can use Long short-term memory (LSTM) to capture the sequence characteristics of each join order. More concretely, we first embed each table as $e(t_i)$, which will be

learned automatically from training data. Then we apply LSTM to encode the join order, which consists of several steps. At $i$-th step, the hidden state vector $h_i$ represents current state and the memory cell $m_i$ preserves the information over $(e(t_1), e(t_2), \ldots, e(t_i))$. LSTM uses an LSTMUnit to get the state $h_i$ and the memory $m_i$ based on the input $e(t_i)$ and the previous state representation $(h_{i-1}, m_{i-1})$, *i.e.*, $(h_i, m_i) = \texttt{LSTMUnit}(e(t_i), h_{i-1}, m_{i-1})$, and the join order representation $E_O = h_n$.

**Performance prediction.** Finally, the join order estimator will output the estimated performance of an order corresponding to a leaf node $v$. To be specific, $\mathcal{B}(v)$ =FC (Concat($E_Q$, $E_O$)), which is computed by concatenating the query encoding and join order representation, followed by a fully connected layer.

# 4 OPTIMAL PLAN SELECTION

In this section, we leverage the uncertainty-aware plan performance predictor to generate the estimated execution time $t_i$ and the corresponding uncertainty $u_i$ for each candidate plan $p_i$. Then we select the optimal plan from these candidates according to $t_i$ and $u_i$.

## 4.1 Feature extraction and encoding

Given a tree structure execution plan $p$ of the query $Q$, we first encode each node (operator) of the plan. Then all the nodes are encoded into a tree according to the structure of the plan. The node encoding mainly consists of three parts: operator encoding, table Encoding, and cost model encoding.

**Operator encoding** $E_o(v)$ is a one-hot vector that encodes the physical operation of a node $v$ in the plan. This physical operator may be a join operator (e.g., nest loop join, hash join or merge join), a scan operator (e.g., Seq Scan, Index Scan, Index Only Scan, Bitmap Index Scan), or an aggregation operator. To be specific, $E_o(v)[i] = 1$ indicates that node $v$ is associated with the $i$-th physical operator and other elements in $E_o(v)$ are 0. For example, in Fig. 4, for node *Nest(a,b)* means that table $a$ and table $b$ are joined by nest loop join (1-th join operator), so $E_o(v)[1] = 1$.

**Table encoding.** We use $E_t(v)$ to denote the table(s) involved in the node $v$. To this end, given a table $a$, we first use a one-hot vector $e(a)$ of length $n$ to encode it. For a binary operator (e.g. join operator), suppose that it involves 2 tables $a$,$b$. We concatenate $e(a)$,$e(b)$ to form a vector of length $2n$ to represent $E_t(v)$. For a unary operator (e.g., a scan opeartor), suppose that it involves a single table $c$, so we concatenate a **0**-vector $e(0)$ to form $E_t(v)$ of length $2n$. For example, in Fig. 4, the node *seq(a)* denotes a Sequential Scan operator on table $a$, *i.e.*, $E_t(v) = \texttt{Concat}(e(a), e(0))$.

**Cost model encoding.** The cost and cardinality of the node $v$ in the plan are denoted by a vector $E_c(v) = (\texttt{Cost}(v), \texttt{Card}(v))$ of length 2. The cost/cardinality is estimated by the optimizer based on statistics, which can accelerate the training process.

**Node encoding.** We concatenate the operator encoding, table encoding, cost model encoding of node $v$ in the plan $p$, and get the node encoding $E_N(v) = \texttt{Concat}(E_o(v), E_t(v), E_c(v))$. Then we can use the encodings of all nodes as the features of the tree. In the next subsection, based on these features, we will show how to use Tree-LSTM to compute the plan representation.

## 4.2 Tree-LSTM-based Plan Representation

Tree-LSTM [40] is a dynamically constructed neural network that captures the structural features of the target data. In this section, we will first introduce the motivation for using Tree-LSTM, and then describe how to represent the execution plan using Tree-LSTM.

The execution plan $p$ is a tree structure. During the execution of the plan, the computation of each node $v$ depends on the results of its child nodes, which is a recursive process. Since the physical operator of the execution plan involves no more than 2 tables. The physical operators corresponding to nodes $v$ in plan $p$ consist of a triple, i.e., the left child $l(v)$, the right child $r(v)$ and node $v$ itself, *i.e.*, $(l(v), r(v), v)$. For leaf node operations (e.g. scan) a triple can be made by setting the left and right child to 0. If we use the traditional LSTM algorithm, we are required to flatten the nodes in the plan into a sequence directly. We will lose the structure information of the plan. In contrast, Tree-LSTM can be computed directly on the tree-structure plan, and the structural information of each node is naturally extracted. We use a Tree-LSTM Unit[40] to give a plan representation vector $R_p$ for a plan. For each node $v$ in the tree, it takes as input the information of the left child node $l(v)$, the right child node $r(v)$, and $v$ itself to obtain a neural network representation of the subtree corresponding to $v$. We set $h_v, m_v$ as the representation of state and memory for the node $v$ respectively.

Then, we perform a depth-first search (DFS) on the tree to access each node and compute the representation of the tree. For each node $v$, we get $(h(l(v)), m(l(v))), (h(r(v)), m(r(v)))$ first. Then input these representations of children with $E_N(v)$ into Tree-LSTM Unit to get the neural network output of $v$, $(h(v), m(v)) =$ *Tree-LSTMUnit*$(h(l(v)), m(l(v)), h(r(v)), m(r(v)), E_N(v))$.

The state of the root node is represented as the plan representation $R_p = h(root)$. As shown in Fig. 4, we perform a DFS on the tree. The leaf node corresponding to the operator $Seq(b)$ is firstly computed by Tree-LSTM Unit. Finally, the root node of the plan, *i.e.*, $Nest(c\ d)$ will be computed by Tree-LSTM Unit. $h(root)$ is the final representation of the plan $p$.

## 4.3 Multi-head Performance Estimator

Query encoding $E_Q$ and the representation $R_p$ are the input of the output layer denoted by $x = (E_Q, R_p)$, based on which, HybridQO not only predicts the execution time of query $Q$ under plan $p$, but also gives the confidence for this prediction, *i.e.*, the uncertainty [15].

The uncertainty output by the neural network is composed of *Epistemic Uncertainty* and *Aleatoric Uncertainty* [15, 18]. (1) *Epistemic Uncertainty* $U_E$ is caused by the fact that the parameters of the neural network model are not good enough, such as lacking of the knowledge of the out-of-distribution data. (2) *Aleatoric Uncertainty* $U_A$ is caused by the noise in the training data. For example, the execution time of the same plan varies because of the cache, or the same selectivity feature brings different cardinality estimates, etc. In general, the uncertainty of the model is a combination of two uncertainties, $U = (U_E, U_A)$.

Inspired by previous works [35], we propose *multi head performance estimator* (MHPE), which outputs the execution time of the plan along with $U_E$ and $U_A$, respectively.

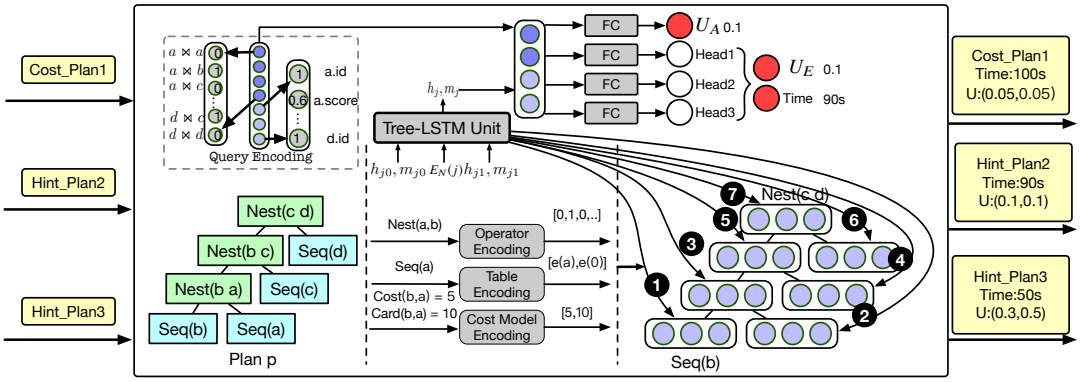



**Figure 4: Optimal Plan Selection with An Uncertainty Model.**

**Epistemic Uncertainty.** For performance prediction, the output layer (i.e., fully connected layer) is used to predict the execution time $T(x)$ of the plan, and we call such an output layer as a head $H$, *i.e.*, $T(x) = H(x)$. MHPE includes $k$ heads, which are randomly initialized and independently select the training data. Each head $H_i$ gives its prediction $t_i(x) = H_i(x)$ for the plan. For an out-of-distribution data instance, different $H_i$ will give inaccurate estimates and therefore generate high variance of the estimates. The final prediction time is the mean of the different heads' predictions, *i.e.*, $T(x) = \bar{t} = \frac{\sum_i H_i(x)}{k}$, and the uncertainty is the variance of the predictions given the $k$ heads, *i.e.*, $U_E = \frac{\sum_i (H_i(x)-\bar{t})^2}{k}$.

**Aleatoric Uncertainty.** $U_A(x)$ captures the noise (variance) of the input data $x$ and the true label $y$ of MHPE. We compute $U_A(x)$ using a separate model with input x. Similar to a head, given an output layer (e.g., a fully connected layer) for computing aleatoric uncertainty,$U_A(x) = \texttt{FC}(x)$. The value of $U_A(x)$ is inferred from previous $N$ training data. According to the previous work [33], $y_i$ is the label (running time) of the data $(x_i)$, then the loss function is

$$loss(\theta) = \frac{1}{N}\sum_{i=1}^{N} \frac{log\, U_A(x_i)}{2} + \frac{(y_i - T(x_i))^2}{2U_A(x_i)} \tag{2}$$

where $\theta$ is the parameter of MHPE. We optimize $\theta$ to minimize the loss. In real scenarios, mini-batch gradient descent is used, i.e., one batch is sampled at a time for training, instead of using all.

## 4.4 Uncertainty-based Optimal Plan Selection

MHPE outputs the execution time and uncertainty $(T(p), U(p)) = f_\theta(E_Q, R_p)$ for each plan $p$. We need to choose the best plan to give to the DBMS for the final execution.

Given $H$ learning-based plan $p_l^i, i \in [1, H]$ generated by hints and a cost-based plan $p_c$, we need to determine if a condition such as the workload shift has occurred, making the learning-based plan unreliable. We first filter out the learning-based plans with high uncertainty to obtain the remaining $H'$ accurately estimated plans, each of which is denoted by $p_l'^i$. Then we selected the optimal plan $p_l'^o = \arg\min_{p_l'^i} T(p_l'^i)$ from them. Finally, we compare the chosen $p_l'^o$ with the cost-based plan $p_c$ and choose the plan with the lowest estimated execution time to the executor for execution. When all candidate plans have too high uncertainty, we consider $Q$ as a new query and choose cost-based plan $p_c$ by default.

A natural important problem is how high the uncertainty value $U(p)$ is high, so as to determine whether the estimation of the plan $p$ is inaccurate or not. Hence, to measure the uncertainty explicitly, we introduce `Q-error` to build a mapping between the two measurements using previous training examples, and then use the mapping to keep (drop) low (high) uncertainty queries. `Q-error` is to represent the error between the estimated execution time $T(p)$ and the true execution time $Acc(p)$ of a model for a plan $p$, *i.e.*$\texttt{Q-error}(p) = \frac{max(T(q),Acc(p))}{min(T(q),Acc(p))} - 1$. Obviously, the higher the `Q-error`, the higher the uncertainty is, and thus we want to filter out plans with large `Q-error`. Since `Q-error` has the explicit meaning, *e.g.*$\texttt{Q-error}(p) > 1$ means that the real execution time can be over twice longer than the estimated time. We can use `Q-error` as a bridge to filter out uncertain queries.

To be specific, given an uncertainty $U(p)$, we want to predict the `Q-error` of $p$, but $p$ has not been executed. We consider aleatoric uncertainty and epistemic uncertainty, respectively. We first take aleatoric uncertainty as an example.

We resort to several plans (say 10 queries) that have been executed and have similar aleatoric uncertainty with $p$. We get their `Q-errors` and compute the median as the `Q-error` of $p$, denoted by $\texttt{Q-error}_{median}(p)$. Finally, we filter the plans with $\texttt{Q-error}_{median}(p)$ greater than a threshold, say 1, to avoid choosing a plan which is predicted inaccurately. Then, we use the epistemic uncertainty to filter the plans following the above process.

## 4.5 Model training and updating

As shown in Figure 2, `HybridQO` collects all training data with a memory pool $M$. For a query $q$, after a plan $p$ is selected, the DBMS executes the query $q$ according to the plan $p$, and the execution time of the plan $p$ is $t(p)$. The triple $D_i = (q, p, t(p))$ constitutes a training instance, and all $D_i$ are collected in $M$.

For every 10 executed queries, `HybridQO` will perform a training process. For each training process, `HybridQO` samples $bs$ triples from $M$ to form a batch $B$. For each training sample in $B$, the features are extracted according to the requirements of MHPE and the join order estimator in MCTS, respectively, and then the two neural networks are trained. The training process is asynchronous and the

**Table 1: Datasets**

| Workload | Database | Data Size(GB) | Scenario | Relations |
|---|---|---|---|---|
| JOB | **IMDB** | 3.6 | Static | 4-17 |
| JOB-EXT | **IMDB** | 3.6 | Static | 3-11 |
| JOB-D | **IMDB** | 3.6 | Dynamic | 4-17 |
| Stack | **Stack** | 100 | Static | 4-12 |

**Table 2: The original and current scenario settings for different methods.**

| | Plans/query | Pretrain | Incremental learning |
|---|---|---|---|
| HybridQO | 1 | False | Yes |
| PostgreSQL | 1 | False | No |
| AlphaJoin | **Many → 1** | **True → False** | **No → Yes** |
| RTOS | **Many → 1** | **True → False** | **No → Yes** |
| BAO | 1 | False | Yes |

model is updated directly after the training is completed to avoid blocking the online workload.

# 5 EXPERIMENT

In this section, we conduct experiments mainly answering the following questions: (1) Can HybridQO reduce the total latency of the query execution on both static and dynamic workloads without pretraining? (Section 5.2) (2) Do the short leading hints help HybridQO? (Section 5.3.1) (3) Whether HybridQO can adaptively select the good plans for the static and dynamic workload? (Section 5.3.2)

## 5.1 Experimental Setup

**Environment.** We built HybridQO in PostgreSQL 12.4, using the pg_hint_plan [1] plugin to provide the functionality of the leading hint. The server is a machine with 64 GB of RAM, a 4.00 GHz i9 CPU, and an NVIDIA 1080ti GPU. The neural networks are implemented using Pytorch.

**Datasets and Workloads.** As shown in table 1, our experiment contains 4 workloads on 2 databases. **IMDB** is a real-world database and is short for Internet Movie DataBase. It contains a plethora of information about movies as well as related facts about actors, directors, production companies and is 3.6GB. **Stack** [29] is the database of 170 different StackExchange websites and is 100GB. Each workload consists of 20,000 randomly generated queries which will be sequentially fed into the database. The query is generated by using the query in the original dataset (*e.g.*, 113 queries in the JOB) as a template. The four workloads are generated as follows.

- JOB [23]: It's a static workload. When generating 20,000 queries of this workload, all 113 queries in original JOB are used as templates. Each time a template is randomly selected, its join relationship is preserved, and the predicates in it are randomly replaced to generate a new random query. The replacement of the predicate is to randomly extract the cell values on the corresponding column from the database for replacement, to ensure that the generated query can find the corresponding values. The number of relations in each query ranges from 4 to 17.
- JOB-EXT [31]: It's a static workload. Similar to JOB, we generated the queries based on the join graph in the original JOB-EXT workload and generated predicates from randomly sampled data. The number of relations in each query ranges from 3 to 11.
- JOB-D: It's a dynamic workload generated from the original JOB [23]. When generating 20,000 queries of dynamic workload, we need to capture the dynamic changes of the queries. Thus we generate queries with both new templates and old templates. We dynamically add templates. First, we add one template and use this template to generate 200 queries. Then, we add the second template, and use the first two templates to generate 200 queries. Iteratively, we can add all 113 templates and generate 20K queries. The number of relations in each query ranges from 4 to 17.
- Stack [29]: It's a static workload. This workload contains queries generated from 25 different templates based on the **Stack** database. We extend the workload to 20000 queries. The number of relations in each query ranges from 4 to 12.

**Baselines.** The baselines are shown as below:

- PostgreSQL: We use the optimizer of PostgreSQL (version 12.4) itself with default settings.
- AlphaJoin [53]: AlphaJoin is a learned join order selection method. Similar to HybridQO, it uses MCTS to perform a search of join order. The difference is that it recommends the complete join order and directly uses the optimal join order estimated by the neural network.
- RTOS [50]: RTOS is a DQN-based method that also uses the Tree-LSTM to encode the query plan, and then directly predicts a complete join order that is expected to be optimal.
- BAO [29]: BAO uses the physical operator hint to guide the traditional optimizer generating plan which is similar to HybridQO. We use its open-source code. We use the recommended parameters, e.g. retrain the neural network every 100 queries with window size set to 2000.

**Experimental scenario.** The table 2 describes the changes to the settings for running and training of each algorithm. (1) The training workload is not prepared in advance to pretrain the neural network, so it is no longer necessary to consider the training workload and the future workload to be identically distributed. (2) The optimizer generates only one plan for execution. The learning-based optimizer can no longer generate multiple plans for the same query for exploration. (3) Once a plan has been executed, we add the plan and its execution time to the experience for incremental learning. This scenario is close to traditional database users' habits.

**Hyper-parameter setting.** For AlphaJoin and HybridQO, every 10 queries are executed, the neural network is trained using 128 training data sampled from the previous experience.

For HybridQO, we set the length of the leading hint as 2. The number of candidate hints is set to 5. The time budget for MCTS is 20ms. We filter out the plans with estimated Q-error larger than 1 considering the uncertainty.

**Timeout and latency normalization.** To avoid the impact of bad plans, we set the timeout of queries. Specifically, we set the timeout

**Table 3: Total latency (h) of executing 20k queries.**

| | JOB | JOB-EXT | JOB-D | Stack |
|---|---|---|---|---|
| HybridQO | **6.40** | **7.35** | **6.70** | **6.68** |
| PostgreSQL | 8.54 | 9.33 | 8.76 | 7.49 |
| AlphaJoin | 12.17 | 15.04 | 12.28 | 12.60 |
| RTOS | 8.05 | 10.92 | 11.05 | 9.93 |
| BAO | 7.51 | 8.37 | 8.17 | 7.24 |

to 3 times of the time predicted by the learning-based optimizer. We also set a global maximum timeout to 120s if 3 times of predicted time is too large. To facilitate the training of the learning model, we normalize the execution time of [0s,120s] to the interval [0,1].

## 5.2 The performance of HybridQO

In this section, we analyze the performance of different optimizers on each workload by evaluating the total latency and tail latency.

**Total latency.** An important consideration is the total latency of each workload, which indicates the average performance of the optimizer. Table 3 records the total latency after finishing all 20,000 queries. HybridQO outperforms PostgreSQL, AlphaJoin, BAO and RTOS on different workloads. HybridQO spends a total of 6.40 hours on JOB, which is less than PostgreSQL (8.54 hours). Compared to PostgreSQL, HybridQO saves $\frac{8.54-6.4}{8.54} = 25.1\%$ latency. On the hardest workload **Stack**, HybridQO also saves $\frac{7.49-6.6}{7.49} = 11.9\%$ latency compared to PostgreSQL.

Figure 5 shows the total latency curves of HybridQO, PostgreSQL, BAO, RTOS and AlphaJoin when finishing the same number of queries on different workloads. We can find that HybridQO can achieve performance close to PostgreSQL at the very beginning (*e.g.*first 1000 queries) on all 4 workloads. The reason is that HybridQO is able to integrate well with traditional cost-based plans and avoid selecting inaccurately estimated plans.

After executing about 1500 queries on JOB, HybridQO begins to spend less time than PostgreSQL, indicating that HybridQO starts to generate better plans than PostgreSQL. On JOB-EXT, this number is about 2500 queries. However, after executing about 7000 queries on JOB-D, HybridQO starts to spend less time to finish the same number of queries, indicating that dynamic scenarios are more difficult to give good execution plans for the learning-based optimizer. Note that AlphaJoin, which also uses MCTS, does not converge to a good model on the workload of 20,000 random queries. It indicates that our test scenario (no pretrain, one plan for one query) will bring challenges to the convergence efficiency of a learned optimizer that generates a complete join order. When completing 20,000 queries, RTOS performs better than PostgreSQL on JOB, but worse than PostgreSQL on JOB-D, indicating that dynamic workloads are more challenging for these optimizer that requires pre-training. BAO is better than PostgreSQL, RTOS and AlphaJoin on JOB, JOB-D and JOB-EXT, but worse than HybridQO, indicating that HybridQO can get better plans and choose fewer bad plans with recommended join order hints and uncertainty-based plan selection.

**Tail latency.** The tail performance can reflect the stability of an optimizer. We choose 50%,75%,99%,99.5% as the reported percentiles. Figure 6 shows the performance of HybridQO, PostgreSQL, AlphaJoin, RTOS and BAO at different percentile with 20,000 queries executed on different workloads. We find that HybridQO outperforms other methods on JOB, JOB-EXT,JOB-D and Stack at most percentiles. BAO is the best learning-based optimizer in the baselines. On JOB, HybridQO's 99.5% latency is 18.72 seconds, saving $\frac{54.29-18.72}{54.29} = 65.52\%$ latency compared to PostgreSQL's 54.29 seconds and saving $\frac{27.82-18.72}{27.82} = 32.71\%$ latency compared to BAO's 27.82 seconds. Thus, HybridQO significantly reduces the tail latency. As for the 50% latency on JOB, HybridQO's latency is 0.216 seconds, saving $\frac{0.289-0.216}{0.289} = 25.25\%$ latency compared to PostgreSQL's 0.289 seconds and saving $\frac{0.296-0.216}{0.296} = 27.02\%$ latency compared to BAO's 0.296 seconds. On Stack, since the database is very large (100GB), HybridQO outperforms PostgreSQL on 50% and 75% percentiles. The latency of HybridQO on 99% and 99.5% percentiles are similar to those of PostgreSQL. HybridQO is better than BAO on 99% and 99.5% latency, because HybridQO chooses good plans and the uncertainty-based plan selection can filter out some bad plans.

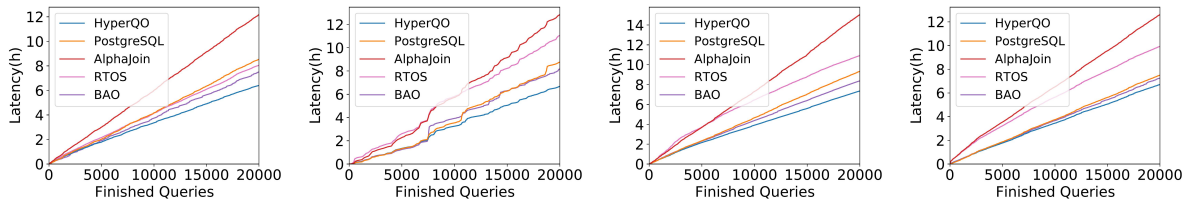
## 5.3 Evaluation on HybridQO

In this section, we observe the effect of different parts of HybridQO.

*5.3.1 Leading hints v.s. complete join order.* We analyze whether recommending a leading hint of length 2 is more beneficial for a learning-based optimizer to produce good plans in a short time compared to recommending a complete join order. We first analyze the quality of the search space for leading hints of different lengths . For a given length $k$, we randomly generate leading hints of length $k$ on JOB workload. Comparing the execution time of these leading hint-based plans with the default plans generated by PostgreSQL, we obtain the relative performance distribution for leading hints of different lengths. Figure 7(a) shows the relative performance distribution of leading hints with lengths of 2,3,7,11. We find that for randomly generated leading hints, there is a 23% probability that leading hints of length 2 can generate plans that are not worse than PostgreSQL. In contrast, there is only a 6% probability that a leading hint of length 11 will produce a plan that is not worse than PostgreSQL. This shows that short leading hints are less likely to generate wrong plans and have a better search space.

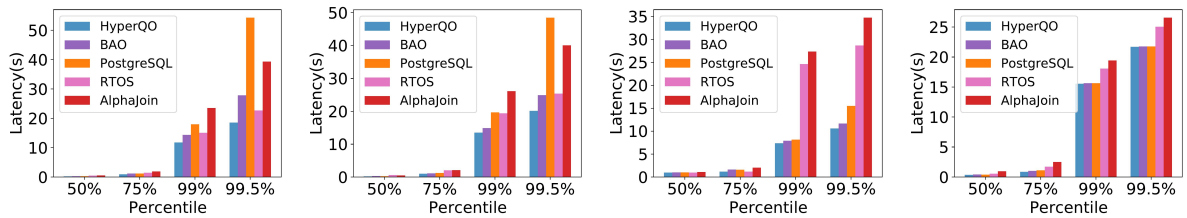
Then, on the static workload JOB and dynamic workload JOB-D, we test the impact of different leading lengths (2,3,7,all), and the results show that HybridQO can perform better than PostgreSQL on all lengths. We can see from Figure 7(b) and Figure 7(c) that the shorter the length, the better performance, because the hint with a short length indicates that our MCTS can search in a small search space, which is likely to discover good hints with only a small number of training examples. For example, on JOB (JOB-D), length 2 spends 6.40 hours (6.68 hours), which is better than that of length 7 on both workload types (6.90 hours and 7.54 hours respectively).

*5.3.2 Adaptivity of Plan Selection.* We examine whether HybridQO can adaptively select cost-based and learning-based plans for dynamic workloads and static workloads. For every 1000 queries, we define the chosen rate $R_c$ as the rate of queries that choose the leading hint-based plan finally to 1000 queries(*i.e.*line "Chosen" in Figure 8). For those queries, for which HybridQO chooses leading hint-based plan finally, we define the win rate $R_w$ as the rate of



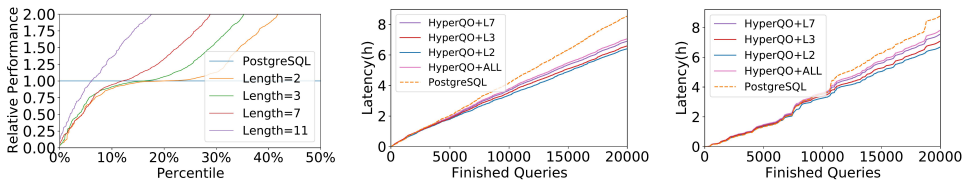

(a) JOB (b) JOB-D (c) JOB-EXT (d) Stack

**Figure 5: Total latency (hour) on each dataset.**



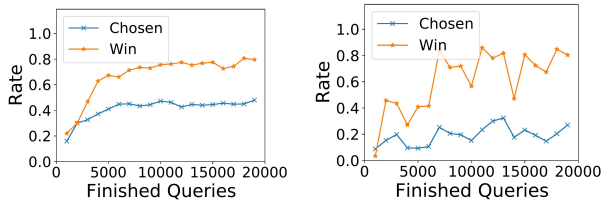

(a) JOB (b) JOB-D (c) JOB-EXT (d) Stack

**Figure 6: The latency of 50%, 75%, 99%, 99.5% percentile of executing 20K queries.**




(a) Relative performance distribution when sorting the plans for each leading length. (b) Total latency(h) on JOB when HybridQO using different leading lengths. (c) Total latency(h) on JOB-D when HybridQO using different leading lengths.

**Figure 7: The impact of leading hint length.**




(a) JOB (b) JOB-D

**Figure 8: The chosen rate and win rate of leading hint-based plan on JOB and JOB-D.**

queries whose leading hint-based plans outperform the cost-based plans to them all.(*i.e.*line "Win" in Figure 8).

As shown in Figure 8, we find that $R_c$ and $R_w$ are increasing on both JOB and JOB-D, which indicates that our model becomes more and more accurate for estimating the execution time of the plans and helps the plan selection in HybridQO. When comparing JOB-D and JOB, we find that the $R_c$ and $R_w$ of JOB-D are both lower, indicating that dynamic workload is more difficult to learn compared with the static workload. Note that the dynamic workload continuously introduces new templates, which makes the learning-based optimizer fail to estimate the execution time accurately, and thus there is a sudden drop in $R_w$. We find that $R_c$ decreases at the same time, which indicates that HybridQO is able to sense out-of-distribution queries and adaptively select cost-based plans.

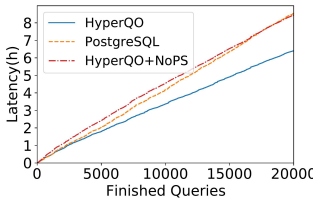
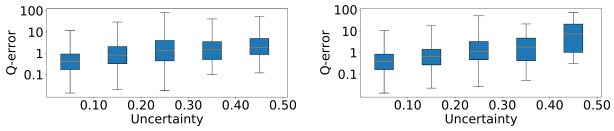
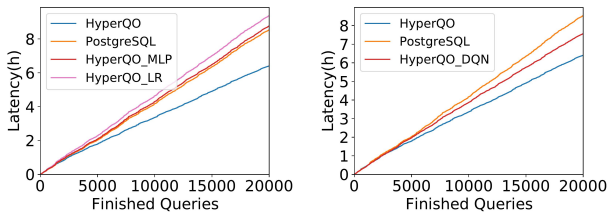



**Figure 9: The impact of total latency(h) on JOB**




(a) Aleatoric Uncertainty (b) Epistemic Uncertainty

**Figure 10: Q-error of queries with different uncertainties.**




(a) Use multi-layer perceptron and linear regression to replace the MPHE (b) Use DQN to replace the MCTS

**Figure 11: Total latency (hour) of executing 20K queries of JOB when using multi-layer perceptron, linear regression and DQN to replace the HybridQO's model.**

**Does the plan selection help HybridQO?** We verify whether the uncertainty-based plan selection in HybridQO can help HybridQO avoid selecting plans with incorrect estimated execution time. HybridQO +NoPS directly selects the plan with the lowest estimated execution time. Figure 9 compares the performance of HybridQO +NoPS, HybridQO and PostgreSQL. We can find that HybridQO +NoPS is worse than PostgreSQL for a long period, and surpasses PostgreSQL at 17000 queries. This shows that the uncertainty-based plan selection can help HybridQO filter out some plans with incorrect estimated execution time to reduce the total latency.

We find that the method of HybridQO +NoPS is similar to AlphaJoin. A major difference is that AlphaJoin recommends a complete join order while HybridQO +NoPS recommends a leading hint of length 2. However, AlphaJoin cannot perform better than PostgreSQL at 20,000 queries which indicates that short leading hints make it easier for HybridQO +NoPS to generate good plans.

**Does uncertainty respond to estimation error?** We analyze whether uncertainty value can really reflect the estimation error. Figure 10 shows the relationship between uncertainty and estimation error on the JOB workload. Q-error is used to represent the estimation error. Dividing the uncertainty intervals with a size of 0.1, we report the Q-error distribution for each uncertainty interval. As shown in Figure 10(a), we find that the median of Q-error($q$) distribution increases as $U_A(q)$ increases. For more than 75% plans $p$ with $U_A(p) < 0.1$, Q-error($p$) does not exceed 1. We can obtain similar conclusions for epistemic uncertainty. This shows that epistemic uncertainty and aleatoric uncertainty can capture the estimation errors and help HybridQO filter out bad plans in advance.

*5.3.3 Model selection .* Here, we verify the effectiveness of choosing Tree-LSTM and MCTS as our models.

**Compared MCTS with DQN.** The MCTS is utilized to discover good hints. For MCTS, we add a DQN-based method as an alternative. As shown in Figure 5, we have shown that comparing with RTOS (a DQN-based method), our method has better performance because the DQN-based method aims at finding a complete join order that is expected to be optimal, which is error-prone when there exist a large number of possible join orders. In addition, we also extract the prefixes of the well-performed complete join orders (output by the DQN-based method) as the hints, so as to generate candidate plans in our HybridQO. The experimental results are shown in Figure 11(b). The DQN-based method perform worse than MCTS because it is hard to generate good complete orders accurately, and thus the hints extracted from them are also not good.

**Table 4: The planning time(s) and total latency(s).**

| PostgreSQL | | HybridQO | | | |
|---|---|---|---|---|---|
| PG | Total | MCTS | PG | MHPE | Total |
| 350.81 | 30765.26 | 441.50 | 384.21 | 356.43 | 23056.73 |

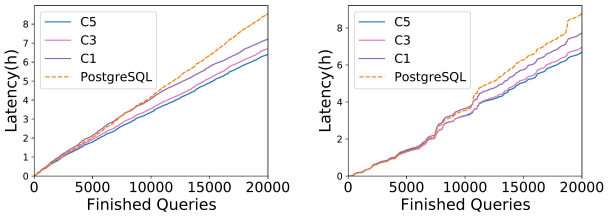
**Compared Tree-LSTM with linear regression and multi-layer perceptron.** Given these candidate plans generated based on hints, the Tree-LSTM is utilized to select an optimal one among these candidates. We test linear regression (LR) and multi-layer perceptron (MLP) as alternatives. We can see from Figure 11(a) that MLP outperforms LR, because MLP uses the neural network to predict the performance, which has a more powerful learning ability than LR. Our Tree-LSTM model achieves a better performance than MLP because it can capture the tree structure of the query plan, leading to a more accurate predication.

*5.3.4 Planning time.* In this section, we discuss the impact of the extra plan search time brought by HybridQO on the total latency. Table 4 shows the total running time of each part in HybridQO and PostgreSQL in finishing 20,000 queries on JOB.

Unlike PostgreSQL, which requires only one plan generation process, HybridQO's planning time consists of 3 parts.

- **MCTS**. The candidate leading hints is generated by MCTS. For each query, we define a maximum search time of 20ms. The total search time is 441.50s.
- **PG**. PostgreSQL generates the corresponding plan based on the candidate leading hints. The plans are generated in parallel, and the total wall time is 384.21s.
- **MHPE**. The execution time and uncertainty of each plan are estimated by MHPE. The plans are combined into a batch feeding into the neural network for computation. The total neural network time is 356.43s.

Compared with 350.81s PostgreSQL, HybridQO uses 441.5 + 384.21 + 356.43 = 1182.14s on planning, which is 3.3 times longer than the planning time of PostgreSQL. However, compared to the total latency, the planning time of HybridQO only accounts for 3.84% of the total latency of PostgreSQL. The planning time of HybridQO accounts for a small proportion of the total latency for many OLAP scenarios (such as real workloads JOB and Stack), far less than the improvement by HybridQO. MCTS and MPHE can also be further




(a) JOB (b) JOB-D

**Figure 12: Total Latenc (hour) of HybridQO when it chooses different number of candidate leading hint.**

optimized to reduce latency by using highly optimized C++ code. For those short running queries (*e.g.*, some OLTP scenarios), which do not have much room for improvement. A straightforward solution is that for such workloads, we just use the cost-based optimizer in the database for query optimization rather than HybridQO.

*5.3.5 Effect of the number of candidate leading hints .* As shown in Figure 12, in terms of the number of candidate hints, the number 3 performs better than that of 1 (5.9% in JOB, 8.9% in JOB-D). The number 5 is better than that of 3, but the improvement is not large (3.5% in JOB, 2.9% in JOB-D). Obviously, more candidates are likely to improve the recall of good plans, thus improving the query performance. However, as the number of candidates ($\geq 3$) continuously increases, the improvement becomes smaller. As discussed in Section 5.3.4, more number of candidate hints require more parallel computing resources for PostgreSQL to generate the plan, so 3-5 is a suitable range of number of candidate hints.

## 6 RELATED WORK

**Learning based plan generation** Learning-based plan generation methods have shown excellent performance in recent years. They can be classified into two categories based on whether they support dynamic workloads. One kind of learning-based methods need to collect training data and are trained in advance to do well on similar workload in the future. DQ [21] and ReJoin [30] first use DRL with a preliminary neural network model to learn the join process on a given workload. RTOS [50] and Neo [31] further improve the previous work by using graph neural networks(Tree-LSTM, Tree-CNN) to effectively model the structural information of plan trees. DQ, ReJoin, RTOS and NEO all require users to collect training data in advance and only give predictions on similar workloads, which poses a great limitation for practical applications. Another kind of learning-based methods can collect training data directly online and quickly give good plans [29, 42]. Skinner-DB [42] relies on a specific in-memory database that supports efficient switching plans. It uses RL to select the join order during query processing. The most similar work to ours is BAO [29], which uses the physical operator hint to guide the traditional optimizer generating plan. It also uses MAB combined with neural networks to select the final plan. Our approach uses a leading hint for the join order. The search space of join order is larger than the search space of the physical operator, making it more difficult to generate good hints. Finally, HybridQO uses an uncertainty-aware performance prediction method to avoid selecting bad plans.

**Learning-based cardinality estimation** Learning-based cardinality estimation methods can be classified into two categories: data-based and query-based [39]. Previous work [39] provides a specific comparison of data-based and query-based cardinality estimators in terms of performance and accuracy. The data-based cardinality estimator [14, 44, 48, 49] can learn the correlation between data directly from the data. The data-based approach has a high accuracy of prediction. However, due to the huge learning space, it leads to slow training, and the efficiency of model inference is not high. The query-based approach [19, 28, 32, 38] requires the user to provide the execution plan corresponding to the future workload and to give the execution time as training data. These kinds of methods are efficient in terms of inference. But the accuracy of these methods is limited by the quality of training queries, making it difficult to handle OOD queries.

**Uncertainty learning** Uncertainty is a very important research topic in machine learning [15, 33]. Uncertainty not only focuses on the results of the learned model but also on the model's confidence in the results. In general, there are a number of methods can be used to measure uncertainty [10, 12, 16, 22, 36]. Uncertainty is important in scenarios where safety is a concern, such as manipulating the car to avoid collisions [17], uncertainty regions in image detection [18], reinforcement learning for action selection for EE problems [8, 11, 35, 41, 47], etc.

**Learning model and database.** Recently, many works have utilized ML methods to optimize the database, like query generation [54], query rewrite [56], knob tuning [25], index construction [9, 20], view management [13, 51], and database systems [24, 26] (see [55] for a survey). Also, there exist some works that utilize database techniques to improve the efficiency [6, 45, 46, 52] and effectiveness [4, 7, 27] of the ML models (see [5] for a survey)

## 7 CONCLUSION

In this paper, we have proposed an adaptive hybrid optimizer that combined the cost-based and learning-based optimizer through the hint functionality provided by the DBMS optimizer, which can produce well-performed query plan in both static and dynamic workload scenarios. We proposed to utilize hints to achieve the hybrid optimizer. To overcome the large space of possible hints, we designed a benefit-based hint generation approach, which used the MCTS to accelerate the process of generating good hints. We also designed an uncertainty-based plan selection approach, which considered the uncertainty of the learning-based plans to cope with the out-of-distribution queries. Experimental results showed that our system outperformed state-of-the-arts significantly.

In this paper, we focus on optimizing OLAP workloads and leaving optimizing OLTP workloads as a future work.

## ACKNOWLEDGMENTS

This work is supported by NSF of China (62232009, 61925205, 62102215, 62072261), BNRist, Huawei, TAL education, China National Post- doctoral Program for Innovative Talents (BX2021155), China Post- doctoral Science Foundation (2021M691784), Shuimu Tsinghua Scholar.

# REFERENCES


[1] 2022. pg_hint_plan. https://pghintplan.osdn.jp/pg_hint_plan.html.

[2] Brian Babcock and Surajit Chaudhuri. 2005. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *SIGMOD*. ACM, 119–130.

[3] Cameron Browne and Edward Jack Powley et al. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intell. AI Games* 4, 1 (2012), 1–43.

[4] Chengliang Chai, Jiabin Liu, Nan Tang, Guoliang Li, and Yuyu Luo. 2022. Selective Data Acquisition in the Wild for Model Charging. *Proc. VLDB Endow.* 15, 7 (2022), 1466–1478.

[5] Chengliang Chai, Jiayi Wang, Yuyu Luo, Zeping Niu, and Guoliang Li. 2022. Data management for machine learning: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2022).

[6] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2017. Towards Linear Algebra over Normalized Data. *Proceedings of the VLDB Endowment* 10, 11 (2017).

[7] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David R. Karger. 2020. ARDA: Automatic Relational Data Augmentation for Machine Learning. *Proc. VLDB Endow.* 13, 9 (2020), 1373–1387.

[8] Felipe Leno da Silva, Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. 2020. Uncertainty-Aware Action Advising for Deep Reinforcement Learning Agents. In *AAAI*. AAAI Press, 5792–5799.

[9] Haowen Dong, Chengliang Chai, Yuyu Luo, Jiabin Liu, Jianhua Feng, and Chaoqun Zhan. 2022. RW-Tree: A Learned Workload-aware Framework for R-tree Construction. In *ICDE 2022*. IEEE, 2073–2085.

[10] Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In *ICML*, Vol. 48. JMLR.org, 1050–1059.

[11] Zhabiz Gharibshah, Xingquan Zhu, Arthur Hainline, and Michael Conway. 2020. Deep Learning for User Interest and Response Prediction in Online Display Advertising. *Data Science and Engineering* 5, 1 (2020), 12–26.

[12] Biraja Ghoshal and Allan Tucker. 2021. Hyperspherical Weight Uncertainty in Neural Networks. In *IDA 2021 (Lecture Notes in Computer Science)*, Vol. 12695. Springer, 3–11.

[13] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. 2021. An Autonomous Materialized View Management System with Deep Reinforcement Learning. In *ICDE*. 2159–2164. https://doi.org/10.1109/ICDE51399.2021.00217

[14] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.

[15] Eyke Hüllermeier and Willem Waegeman. 2021. Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. *Machine Learning* 110, 3 (2021), 457–506.

[16] Yaniv Ovadia et al. Jasper Snoek. 2019. Can you trust your model's uncertainty? Evaluating predictive uncertainty under dataset shift. In *NeurIPS*. 13969–13980.

[17] Gregory Kahn, Adam Villaflor, Vitchyr Pong, Pieter Abbeel, and Sergey Levine. 2017. Uncertainty-Aware Reinforcement Learning for Collision Avoidance. *CoRR* abs/1702.01182 (2017).

[18] Alex Kendall and Yarin Gal. 2017. What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?. In *NIPS*. 5574–5584.

[19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org.

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

[21] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR* abs/1808.03196 (2018).

[22] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles. In *NIPS*. 6402–6413.

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[24] Guoliang Li, Xuanhe Zhou, and Sihao Li. 2019. XuanYuan: An AI-Native Database. *IEEE Data Eng. Bull.* 42, 2 (2019), 70–81.

[25] 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.

[26] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: An Autonomous Database System. *Proc. VLDB Endow.* 14, 12 (2021), 3028–3041.

[27] Jiabin Liu, Chengliang Chai, Yuyu Luo, Yin Lou, Jianhua Feng, and Nan Tang. 2022. Feature Augmentation with Reinforcement Learning. In *ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 3360–3372.

[28] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proc. VLDB Endow.* 14, 11 (2021), 1950–1963.

[29] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. ACM, 1275–1288.

[30] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM@SIGMOD 2018*. ACM, 3:1–3:4.

[31] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[32] Ryan C. Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746.

[33] David A Nix and Andreas S Weigend. 1994. Estimating the mean and variance of the target probability distribution. In *Proceedings of 1994 ieee international conference on neural networks (ICNN'94)*, Vol. 1. IEEE, 55–60.

[34] Kiyoshi Ono and Guy M. Lohman. 1990. Measuring the Complexity of Join Enumeration in Query Optimization. In *VLDB*. Morgan Kaufmann, 314–325.

[35] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. Deep Exploration via Bootstrapped DQN. In *NIPS*. 4026–4034.

[36] Carlos Riquelme, George Tucker, and Jasper Snoek. 2018. Deep bayesian bandits showdown: An empirical comparison of bayesian deep networks for thompson sampling. *arXiv preprint arXiv:1802.09127* (2018).

[37] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. ACM, 23–34.

[38] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.

[39] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.

[40] 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).

[41] Shan Tian, Songsong Mo, Liwei Wang, and Zhiyong Peng. 2020. Deep Reinforcement Learning-Based Approach to Tackle Topic-Aware Influence Maximization. *Data Science and Engineering* 5, 1 (2020), 1–11.

[42] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *ACM Trans. Database Syst.* 46, 3 (2021), 9:1–9:45.

[43] Florian Waas and Arjan Pellenkoft. 2000. Join Order Selection - Good Enough Is Easy. In *BNCOD*, Vol. 1832. Springer, 51–67.

[44] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84.

[45] Jiayi Wang, Chengliang Chai, Nan Tang, Jiabin Liu, and Guoliang Li. 2022. Coresets over Multiple Tables for Feature-rich and Data-efficient Machine Learning. *Proc. VLDB Endow.* 16, 1 (2022), 64–76.

[46] Ruofan Wu, Feng Zhang, Jiawei Guan, Zhen Zheng, Xiaoyong Du, and Xipeng Shen. 2022. DREW: Efficient Winograd CNN Inference with Deep Reuse. In *Proceedings of the ACM Web Conference 2022*. 1807–1816.

[47] Yue Wu, Shuangfei Zhai, Nitish Srivastava, Joshua M. Susskind, Jian Zhang, Ruslan Salakhutdinov, and Hanlin Goh. 2021. Uncertainty Weighted Actor-Critic for Offline Reinforcement Learning. In *ICML*, Vol. 139. PMLR, 11319–11328.

[48] Zongheng Yang and Eric Liang et al. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292.

[49] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.

[50] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*. IEEE, 1297–1308.

[51] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *ICDE*. 1501–1512. https://doi.org/10.1109/ICDE48307.2020.00133

[52] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Xiaoyong Du. 2022. POCLib: A high-performance framework for enabling near orthogonal processing on compression. *IEEE TPDS* 33, 2 (2022), 459–475.

[53] Ji Zhang. 2020. AlphaJoin: Join Order Selection à la AlphaGo. In *VLDB (CEUR Workshop Proceedings)*, Vol. 2652. CEUR-WS.org.

[54] Lixi Zhang, Chengliang Chai, Xuanhe Zhou, and Guoliang Li. 2022. LearnedSQLGen: Constraint-aware SQL Generation using Reinforcement Learning. In *SIGMOD Conference 2022*. ACM, 945–958.

[55] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2022. Database Meets Artificial Intelligence: A Survey. *IEEE Trans. Knowl. Data Eng.* 34, 3 (2022), 1096–1116.

[56] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58.