A Learned Query Rewrite System
using *Monte Carlo Tree Search*

Xuanhe Zhou
Department of Computer Science, Tsinghua University
zhouxuan19@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

Jianhua Feng
Department of Computer Science, Tsinghua University
fengjh@tsinghua.edu.cn

ABSTRACT

Query rewrite transforms a SQL query into an equivalent one but with higher performance. However, SQL rewrite is an NP-hard problem, and existing approaches adopt heuristics to rewrite the queries. These heuristics have two main limitations. First, the order of applying different rewrite rules significantly affects the query performance. However, the search space of all possible rewrite orders grows exponentially with the number of query operators and rules and it is rather hard to find the optimal rewrite order. Existing methods apply a pre-defined order to rewrite queries and will fall in a local optimum. Second, different rewrite rules have different benefits for different queries. Existing methods work on single plans but cannot effectively estimate the benefits of rewriting a query. To address these challenges, we propose a *policy tree* based query rewrite framework, where the root is the input query and each node is a rewritten query from its parent. We aim to explore the tree nodes in the *policy tree* to find the optimal rewrite query. We propose to use *Monte Carlo Tree Search* to explore the policy tree, which navigates the policy tree to efficiently get the optimal node. Moreover, we propose a learning-based model to estimate the expected performance improvement of each rewritten query, which guides the tree search more accurately. We also propose a parallel algorithm that can explore the tree search in parallel in order to improve the performance. Experimental results showed that our method significantly outperformed existing approaches.

1 INTRODUCTION

The performance of a slow SQL query (e.g., redundant operators) can be improved by orders of magnitude if the SQL query is properly rewritten by query rewrite [17, 46]. Query rewrite is a fundamental problem in query optimization [9, 32, 46], which aims to transform a SQL query into an equivalent one but with higher performance. Specifically, query rewrite transforms a SQL query in logic level

(e.g., removing redundant operators, pulling up subqueries) such that (1) the rewritten query is equivalent to the original one and (2) the execution time is reduced. For example, in Figure 1, if the origin query q is rewritten into an equivalent query q_2 by the order of (1) removing aggregate function in o_5, (2) pulling up the subquery in o_4, and (3) removing aggregate function in o_2, we achieve over 600x speedup than PostgreSQL that rewrites q in a top-down manner.

Query rewrite is an NP-hard problem [9, 32], and existing methods rewrite SQL queries by matching queries with a predefined rule order (e.g., attempt to pull up the subquery before pushing down predicates). However, the limitation is that they only use a default order (e.g., top-down, arbitrary), which may fall in a local optimum. For example, in Figure 1, PostgreSQL rewrites query q in a top-down manner, i.e., (o_1, o_3) (we omit operators that do not match any rules). This order achieves limited improvement, because it first creates a temporary table for subquery o_3, and then the operators in the subquery (e.g., o_4, o_5) cannot be rewritten. Instead, if we first rewrite o_4 and o_5, and then rewrite o_3, the operators in o_3 can be rewritten and the execution time is reduced. A straightforward method samples some orders and selects the best one. However, as there are a huge number of possible orders, it is hard to select the optimal one through sampling within limited rewrite time.

Therefore, existing methods suffer from several challenges. First, the search space of possible rewrite orders is exponential to the number of applicable rules, so *how to represent such a large amount of orders* (**C1**) is a major challenge. Second, given a large search space, *how to find the optimal order efficiently* (**C2**) is another challenge. Third, to select a good rewrite order, an intuitive idea is to estimate the cost reduction of a rewrite (or a sequence of multiple rewrites) and prune a rewrite if the reduced cost by the rewrite is small. The third challenge is *how to estimate the cost reduction of a rewrite* (**C3**).

To address these challenges, we propose a query rewrite system, `LearnedRewrite`, which takes as input a SQL query and a set of rewrite rules, finds the optimal rewrite order and outputs an optimized rewritten query. `LearnedRewrite` first models the possible orders as a policy tree, where the root is the input query. Each non-root node is a rewritten query obtained by applying rewrite rules on its parent, and a path from the root to a tree node corresponds to a rewrite order (addressing **C1**). The advantage of the policy tree is that different paths (rewrite orders) are likely to share common ancestors (rewrite operations), so we avoid applying repetitive operations on these orders. Second, we propose *Monte Carlo Tree Search (MCTS)* to explore the policy tree to find the optimal node

---

[1] Guoliang Li is the corresponding author.

Table 1: Example Rewrite Rules.

	rule	description	example
r_1	RemoveAggregate	Remove redundant aggregates	**e.g.,** *"select max(distinct a) from t;"* → *"select max(a) from t;"*
r_2	TemporaryTable	Create a temporary table for subquery not accessing the outer query	**e.g.,** *"select * from t1 where a1 < any(select a2 from t2 where a2>10);"* → *"with t as (select a2 a from t2 where a2>10) select t1.* from t1,t where a1<a;"*
r_3	Subquery2Join	Pull up the subquery as a join if it is correlated with the outer query	**e.g.,** *"select t1.* from t1 where a1 in (select a2 from t2 where a2<2);"* → *"select t1.* from t1 semi join t2 on a1=a2 and a2<2;"*
r_4	SplitSubquery	Divide the query into subqueries if there are AND/OR in predicates	**e.g.,** *"select * from t where (c1='f' and c2>5) or c2>8;"* → *"select * from t where (c1='f' and c2>5) union all select * from t where c2>8;"*
r_5	NormalizePredicate	Transform predicate with common expressions	**e.g.,** *"select * from t where (c2>18 or c1='f') and (c2>18 or c2>15);"* → *"select * from t where (c1='f' and c2>15) or c2>18;"*
r_6	SimplifyPredicate	Replace "in" with "="	**e.g.,** *"··· t1.c in (10,20,30);"* → *"··· t1.c=10 or t1.c=20 or t1.c=30;"*
r_7	OuterJoin2InnerJoin	Replace outer join with inner join	**e.g.,** *"select * from t1 left join t2 on t1.a=t2.a where t2.a is not null;"* → *"select * from t1,t2 where t1.a=t2.a and t2.a is not null;"*
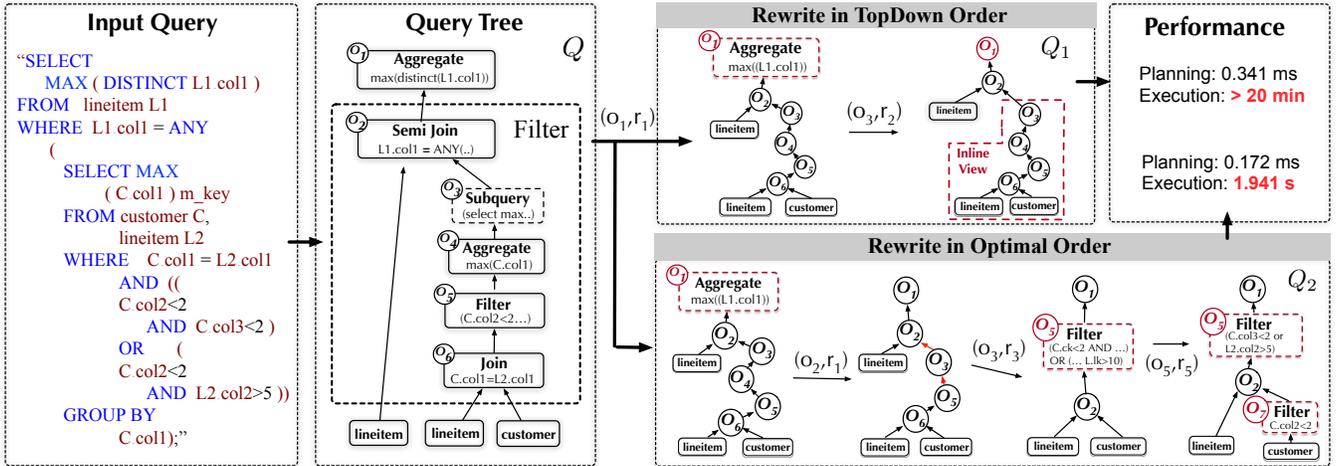
Figure 1: Query Rewrite Example: q_1 **is obtained by rewriting the query tree from top down, i.e.,** (o_1, o_3) **and other operators cannot be rewritten; and** q_2 **is obtained by the optimal rewrite order** (o_1, o_4, o_3, o_5) **that** `LearnedRewrite` **found.**

(i.e., rewritten query with maximum cost reduction) iteratively. In each iteration, MCTS selects the most *beneficial* node (which has high possibility to lead to the optimal node) and expands children of the node (applying rewrite operations). We propose *node utility* to define the beneficial nodes by considering both cost reduction and access frequency (i.e., the number of selected times), which guides the search to find the optimal order (addressing **C2**). We then propose a deep estimation model to estimate the cost reduction of each node accurately (addressing **C3**), by considering query operators, applicable rewrites and columns features. To improve the efficiency when the policy tree is large, we propose a multi-node selection algorithm (addressing **C2**). We make the following contributions.

(1) We propose a tree-based framework to judiciously select an optimal rewrite order, for a SQL query and a set of rewrite rules.

(2) We build a policy tree to represent rewrite orders. We use a MCTS algorithm to search on the policy tree, so as to find the optimal order efficiently and effectively.

(3) We propose an effective model to estimate the cost reduction of a rewritten order.

(4) We design a parallel MCTS algorithm that selects the nodes in parallel to improve the search efficiency.

(5) Experimental results showed that our method significantly outperformed existing approaches.

2 PRELIMINARIES

In this section, we first introduce rewrite rules and rewrite benefits (Section 2.1), and then formalize the problem of *query rewrite* (Section 2.2). Finally, we discuss the related work (Section 2.3).

2.1 Query Rewrite Rules

Query Tree. Given a SQL query q, we model it in the form of a logical query tree, where each tree node is a query operator, e.g., scan, filter, aggregate, join, subquery, union, intersect. For example, in Figure 1, the query tree of query q contains 6 operators and 3 tables. Note that if there is a subquery in q, we add a subquery node as a virtual operator in the query tree.

We interchangeably use a SQL query and its corresponding query tree if there is no ambiguity.

Query Rewrite Rules. Given a query q, a query rewrite rule performs equivalent transformation on the query, e.g., removing a useless operator or exchanging two operators.

DEFINITION 1 (REWRITE RULE). *A rewrite rule r is a triple $r = (o, c, \alpha)$, where o is an operator, c is a condition and α is a rewrite action. Given a query q, the rule first matches a query operator o, and if c is true on o or the subtree rooted at o, we can apply the action α to query q and get $q^{(o,r)}$, where q and $q^{(o,r)}$ are equivalent.*

Table 2: Notations.

Notation	Description
q, q_i	origin query, equivalent rewritten query
$R = \{r_1, \cdots, r_{|R|}\}$	a set of rewrite rules
$r = (o, c, \alpha)$	a rewrite rule (operator, condition, action)
$O = \{o_1, \cdots, o_{|O|}\}$	a set of query operators
T	the *policy tree* of rewrite orders
v_i	a node in the *policy tree*
$C^{\uparrow}(v_i)$	node cost reduction
$C^{\downarrow}(v_i)$	subsequent cost reduction
$\mathcal{F}(v_i)$	access frequency

As a rewrite rule r can be applied to multiple operators, we use $q^{(o,r)}$ to denote rewriting q with rule r on operator o. If there is no ambiguity, we abbreviate $q^{(o,r)}$ as q^r.

EXAMPLE 1. *In Figure 1, for the* filter *operator o_5, since its predicate is "((c.ck<2 and c.cn<2) or (c.ck<2 and l2.lk>10))" and satisfies the condition "there are common expressions in the predicate" of rule r_5 (as shown in Table 1), we can apply the rule to o_5 and the predicate of o_5 will change into "c.ck<2 or (c.cn<2 and l2.lk>10)".*

Rewrite Benefit of Applying A Rewrite Rule. Given a query q and a rule r, let $q^{(o,r)}$ denote the rewritten query by applying r on q. Let $\text{Cost}(q)$ and $\text{Cost}(q^{(o,r)})$ denote the cost of executing q and $q^{(o,r)}$ respectively. The benefit of applying a rewrite rule r to a query q is $\Delta\text{Cost}(q^{(o,r)}, q) = \text{Cost}(q) - \text{Cost}(q^{(o,r)})$. We get the cost of a query (e.g., $\text{Cost}(q)$ and $\text{Cost}(q^{(o,r)})$) from cost estimator.

EXAMPLE 2. *As shown in Figure 1, we can apply the rule r_1 (reduce redundant aggregates) to the two aggregate operators o_1 and o_4. However, we gain different benefits, i.e., $\Delta\text{Cost}(q^{(o_1,r_1)}, q) = 0.0001$ and $\Delta\text{Cost}(q^{(o_2,r_1)}, q) = 0.11$, because the max function in o_1 only sorts one row, while the aggregate operator o_4 under o_2 computes on the joined table.*

The Rewrite Order of Applying Multiple Rewrite Rules. Given a query q and two rules r_1 and r_2, let q^{r_1,r_2} and q^{r_2,r_1} denote two different rewrite orders of the query q. $\text{Cost}(q^{r_1,r_2})$ and $\text{Cost}(q^{r_2,r_1})$ may be largely different, as different rewrite rules may affect each other. For example, applying rule r_1 may make r_2 inapplicable (see below). Thus it is important to select a good rewrite order.

EXAMPLE 3. *In Figure 1, (1) for the top rewrite order, we first rewrite o_3 with the rule r_2 (create a temporary table (inline view) that can be repeatedly referenced by the query, and cannot rewrite the aggregate operator o_4, i.e., $q^{(o_3,r_2)}$, because the subquery is taken as a temporary table and the operators in it cannot be rewritten. (2) For the bottom rewrite order, we consider from a different perspective: we first reduce the aggregate o_4 under o_2 with the rule r_1. And then o_3 does not contain aggregates and is rewritten with the rule r_3 (replace the correlated subquery with a join), i.e., $q^{(o_2,r_1),(o_3,r_3)}$. In (o_3, r_3), because tables L1 and L2 are the same tables that are joined with table C, they can be replaced with table L and only join with C once, which achieves much more cost reduction than the top order.*

2.2 Query Rewrite

Next we formally define the query rewrite problem. Given a query q and a set $R = (r_1, \cdots, r_{|R|})$ of rules, we optimize the query by applying the rewrite rules to the *policy tree* in order to get the best optimized query, defined as below.

DEFINITION 2 (QUERY REWRITE). *Given a query q and a set R of rules, query rewrite is to select a sequence of rewrite operations (o_i, r_j), where o_i denotes the i-th operator of q and r_j is a rewrite rule for o_i, and apply the rewrite operations in this order to obtain a rewritten query $q*$ such that (1) $q*$ is equivalent to q and (2) the cost of executing $q*$ is minimized among all rewritten queries.*

EXAMPLE 4. *In Figure 1, the first order rewrites query q in a top-down manner, i.e., $q^{(o_1,r_1),(o_3,r_2)}$, where only two rules can be applied and it achieves limited cost reduction. Instead, the second order gains the optimal cost reduction with the rewritten query as $q^{(o_1,r_1),(o_2,r_1),(o_3,r_3),(o_5,r_5)}$, where the costly subquery o_3 and the join in o_3 are removed and the cost is significantly reduced.*

2.3 Related Work

Query Rewrite. There are two baselines for *query rewrite*. (1) Human-involved methods. They rely on DBAs to select a subset of rewrite rules and apply the rules to queries, which can achieve relatively high performance. However, it cannot generalize to a large number of queries, because DBAs may take hours or even days to analyze and rewrite a query. Although there are some assistant tools (e.g., ARE-SQL [8], SOAR [1]) that automatically recommend applicable rules based on statistics like table cardinality and disk size, they cannot explore rewrite orders and still cannot automatically rewrite without DBAs. Moreover, it usually finds sub-optimal queries, because DBAs mainly rewrite a query based on past experience (e.g., first optimizing subqueries) and cannot explore more efficient rewriting strategies for different query structures. (2) Heuristic *query rewrite* (e.g., Calcite [6], PostgreSQL [3]). They traverse operators in the logical query plan in a top-down manner. For each operator, it matches rules based on the operator type and reconstructs the query plan with matched rules. Heuristic methods are more efficient but have two main limitations. First, they apply the rewrite rules in a fixed order, and may miss a better rewrite order. Second, they do not estimate rewrite benefits and thus some rewrites may be useless or even slow down the query. For example, adding a semi-join inside a join operator may bring in rewrite overhead and cannot reduce cost when the join column of the driven table is unique. To avoid this problem, heuristic methods only select a small subset of the rules with high possibility to improve the query performance, but they may miss many useful rules. Hence, it calls for an automatic query rewrite system that can efficiently find near-optimal rewrite order from all rewrite rules, rewrite the query by the selected order, and optimize the query.

Learning Models for Databases. Recently there are many works that utilize machine learning techniques to address database problems [5, 10, 12, 18–23, 25, 41, 44–48]. For optimizers, most works focus on physical optimization modules like cardinality/cost estimator [14, 28, 34–36, 40, 42], plan enumerator [15, 27, 30, 39, 43], and plan hinter [26, 31], but ignore the logical rewrite stage. For cost estimation, existing methods [28, 34] estimate the cost of a given plan, but cannot effectively estimate the cost reduction of rewriting the plan. Given a logical plan, there could be a large number of possible rewritten plans, and existing methods need to sample some rewritten queries, estimate the costs of their plans, and select
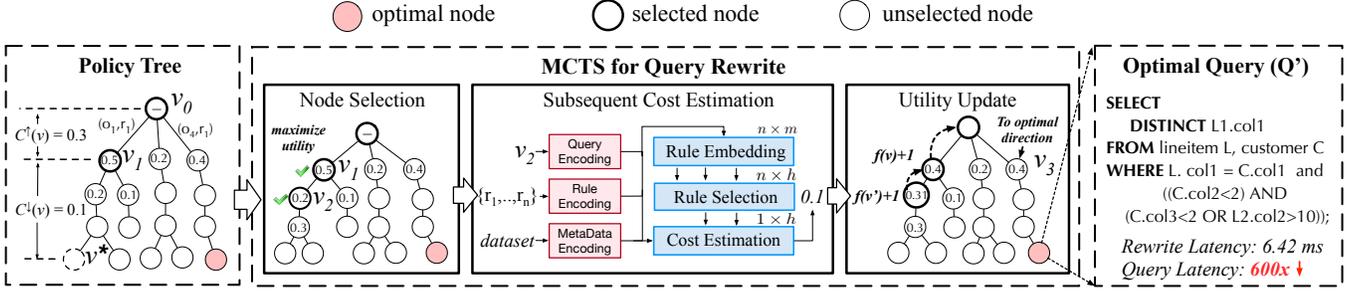
Figure 2: The Workflow of Monte Carlo Tree Search Based Query Rewrite.

the minimum one as the cost reduction of the query. Obviously this method is rather expensive. *Instead,* LearnedRewrite *directly computes the promising rewrite rule combinations and estimates the cost reduction without sampling, and thus* LearnedRewrite *gains higher accuracy and takes less time to infer the best rewritten query.* Note that LearnedRewrite has been applied in openGauss [20] that provides slow query optimization.

Reinforcement Learning. RL methods include model-based RL (e.g., MCTS) and model-free RL (e.g., Q-learning, DDPG) [2, 37, 38]. Model-free RL needs to fine-tune and update the policy model during online inference, which usually cannot support instant queries. Model-based RL pre-trains the estimation model and directly uses the model for online inference. For our problem, query rewrite requires low rewrite overhead (e.g., milliseconds) and high rewrite benefits. Thus we select a light-weight model-based RL (MCTS) that judiciously explores the policy tree based on the utility function.

3 TREE SEARCH FOR QUERY REWRITE

We first present the basic idea of using *policy tree* search to automatically rewrite SQL queries (Section 3.1), and then introduce the *Monte Carlo Tree Search (MCTS)* based method to explore promising rewrite orders by searching the policy tree (Section 3.2).

3.1 Overview of Policy Tree Search

Traditional rewrite methods apply rewrite rules in a default order (e.g., traversing operators in a top-down order and applying the appropriate rules for each operator) and may fall in a local optimum. However, enumerating all possible rewrite orders is not practical, because there are a huge number of possible rewrite orders, especially for SQL queries with dozens of operators. For example, given a query, suppose there are 50 applicable rewrites (pairs of operator and applicable rules), there can be over 50! "potential" rewrite orders. Thus it is vital to judiciously select the best order.

Policy Tree. We build a *policy tree* to represent all the possible rewrite orders, where the root denotes the original query and each node is a rewritten query. A child node is rewritten from its parent by applying a rewrite rule to a query operator. For example, in Figure 2, the edge from the root to v_1 denotes rewriting the origin query with rewrite operation (o_1, r_1).

DEFINITION 3 (POLICY TREE). *Given a query q and a set of rewrite rules, we build a policy tree T, where the root node denotes the origin query q, any non-root node denotes a rewritten query (that transforms the query of its parent by applying a rewrite operation), and a leaf denotes a query that cannot be rewritten by any rewrite rules.*

Each node v corresponds to a query, and we use $\text{Cost}(v)$ to denote the cost of executing the query. The node with the smallest cost in the policy tree is the *optimal rewrite node.*

DEFINITION 4 (OPTIMAL NODE). *The node on the policy tree with the smallest cost is the optimal node.*

Obviously, we want to select the *optimal node* which is equivalent to the root (the given query) but with the smallest execution cost. To find the optimal node, we want to judiciously expand the node with high possibilities leading to the optimal node. To find such nodes to expand, we define the *node utility.*

Node Utility. A node has large potential if it is on the *optimal path* from the root to the optimal node. However, it is rather hard to know whether the node is on the optimal path. To address this issue, given a node, we compute its *benefit*, and the larger the benefit is, the high possibility the node is on the optimal path. We compute the benefit by considering two factors.

(1) *Previous cost reduction* $C^{\uparrow}(v_i)$ of node v_i. It captures the reduced cost between executing the origin query v_0 and executing the rewritten query v_i, i.e., $C^{\uparrow}(v_i) = \text{Cost}(v_0) - \text{Cost}(v_i)$, where the cost can be estimated by the database optimizer or some statistical functions [11]. Intuitively, the higher $C^{\uparrow}(v_i)$ is, the larger the benefit of this node is.

(2) *Subsequent cost reduction* $C^{\downarrow}(v_i)$ of v_i. The query of v_i may be further rewritten and achieves much higher cost reduction. For any descendant v_i^d of v_i, we have

$$\text{Cost}(v_0) - \text{Cost}(v_i^d) = \text{Cost}(v_i) - \text{Cost}(v_i^d) + \text{Cost}(v_0) - \text{Cost}(v_i)$$
$$= \text{Cost}(v_i) - \text{Cost}(v_i^d) + C^{\uparrow}(v_i).$$

Obviously, we want to select the descendant v_i^d with the largest $\text{Cost}(v_i) - \text{Cost}(v_i^d)$ to rewrite the query. Suppose v^* is the descendant of v_i with the largest $\text{Cost}(v_i) - \text{Cost}(v_i^d)$, and we call

$$C^{\downarrow}(v_i) = \text{Cost}(v_i) - \text{Cost}(v_i^*)$$

the largest subsequent cost reduction of v_i, which is abbreviated as *subsequent cost reduction* if there is no ambiguity. For example, in Figure 1, the rewrite operation (o_4, r_1) only eliminates a redundant aggregate (low $C^{\uparrow}(v'')$), but can allow pulling up the subquery and gain much cost reduction (high $C^{\downarrow}(v_3)$). Hence, for any node v_i, its subsequent rewrites (i.e., descendant nodes of v_i) are likely to further reduce the cost. It is naturally to compute $C^{\downarrow}(v_i)$ using a descendant node with the largest cost reduction, i.e.,

$$C^{\downarrow}(v_i) = \max\{C^{\uparrow}(v) + C^{\downarrow}(v) - C^{\uparrow}(v_i) | v \in d(v_i)\}.$$

where $d(v_i)$ denotes the descendant set of v_i.

However, it is hard to obtain the value of $C^{\downarrow}(v_i)$, because we cannot actually rewrite all the descendant nodes and derive the optimal one [4], so we propose learned techniques to estimate the subsequent cost reduction (see Section 4).

Algorithm 1: MCTS Based Policy Tree Search

Input: q: a query tree; R: a set of rewrite rules
1 Initiate a root node v_0 with query q;
2 **while** *within computational budget* **do**
3   v = NodeSelection(v_0);
4     $C^{\downarrow}(v)$ = SubsequentCostEstimation(v);
5   UtilityUpdate($v, C^{\downarrow}(v)$);
6 **return** *the node with the largest previous cost reduction*;

Function NodeSelection(v_0)

Input: v_i: a tree node of the *policy tree*
1 Select node v with the largest utility under v_i;
2 **if** *v is not selected* **then**
3   **foreach** *$o \in$ a query operator of v* **do**
4   **foreach** *$r \in$ an applicable rule of o* **do**
5   r = an applicable rule of o';
6   v' = the node that rewrites v with (o, r);
7           $C^{\uparrow}(v')$ = estimated cost reduction of v';
8           $C^{\downarrow}(v')$= 0; $\mathcal{F}(v')$ = 1;
9   Add v' as a child of v;
10 **else** NodeSelection(v); ;
11 **return** v;

Function UtilityUpdate($v, C^{\downarrow}(v)$)

Input: v: a tree node; $C^{\downarrow}(v)$: subsequent cost reduction
1 $\mathcal{F}(v) = \mathcal{F}(v) + 1$;
2 **foreach** *v' is an ancestor of v* **do**
3     $C^{\downarrow}(v') = \max(C^{\uparrow}(v) + C^{\downarrow}(v) - C^{\uparrow}(v'), C^{\downarrow}(v'))$;
4     $\mathcal{F}(v') = \mathcal{F}(v') + 1$;
5   Update the utility of v';

DEFINITION 5 (NODE BENEFIT). *Given a policy tree, we define the node benefit of a node v_i as the sum of the previous cost reduction $C^{\uparrow}(v_i)$ and subsequent cost reduction $C^{\downarrow}(v_i)$,*

$$\mathcal{B}(v_i) = C^{\uparrow}(v_i) + C^{\downarrow}(v_i)$$

Obviously, we want to access the node with large benefit. However, the estimated benefit may not be accurate, and if we always access such nodes, we may miss the real optimal node. To address this issue, we also consider the access frequency of a node and balance the benefit and frequency in order to avoid falling into a local optimum or wrong directions.

(3) *Access frequency* $\mathcal{F}(v_i)$. It represents the number of visits of the node v_i when selecting new child nodes. Besides rewrite benefits, we tend to select the node that is rarely accessed in order to try more possible rewrites and avoid falling in a local optimum. For example, in Figure 2, after selecting v_2, the access frequencies of v_2 and v_3 are increased by one and the updated utilities of v_2 and v_1 get smaller than v_3. Thus, v_3 can be selected in next iteration, whose subtree contains the optimal node.

Next we define the node utility by combining node benefit and access frequency as blow.

DEFINITION 6 (NODE UTILITY). *Given a policy tree, we define the node utility as the upper confidence bound (UCB) of the probability*

that v_i is on the path from the root to the optimal node, by considering node benefit ($C^{\uparrow}(v_i) + C^{\downarrow}(v_i)$), and access frequency $\mathcal{F}(v_i)$,

$$\mathcal{U}(v_i) = \left(C^{\uparrow}(v_i) + C^{\downarrow}(v_i) \right) + \gamma \sqrt{\frac{ln(\mathcal{F}(v_0))}{\mathcal{F}(v_i)}}$$

where $\mathcal{F}(v_0) = \sum_{i \geq 1} \mathcal{F}(v_i)$ is the number of total accesses, γ is the exploration parameter that adjusts the amount of explorations of uncovered rewrite orders.

EXAMPLE 5. *In Figure 2, given $\gamma = 0.1$, v_1 has highest node benefit (i.e., $C^{\uparrow}(v_1)$=0.3 and $C^{\downarrow}(v_1)$=0.1) and the utility value equals 0.5. At this iteration, v_1 has the highest utility and we further select the subsequent nodes of v_1 since v_1 has been selected.*

3.2 MCTS based Policy Tree Search

To efficiently get the optimal node in the policy tree, we propose a *Monte Carlo Tree Search* (MCTS) based search strategy that judiciously explores the nodes to obtain the optimal node. MCTS [24] is a well-known tree search algorithm, which balances exploitation (high benefit) and exploration (low frequency) when searching the *policy tree* and can acquire more information to the "optimal" node.

MCTS-based Framework. Figure 2 shows the system architecture. Given a SQL query q, we build a *policy tree* with the root node v_0 as q and initialize $C^{\uparrow}(v_0) = 0$, $C^{\downarrow}(v_0) = 0$, and $\mathcal{F}(v_0) = 0$. Then we iteratively explore/exploit promising rewrite rules to expand the *policy tree* in three steps.

1. Node Selection and Expansion. As the node with the maximum utility has the largest possibility of leading to the optimal node, we select the node v with the maximum utility to explore. (*i*) If the selected node has not been expanded (not rewritten by any rules), we enumerate the rewrite operation ($o \in v, r \in R$), where o is an operator in v and r is an applicable rule on o. For each operation, we use it to rewrite v and generate a new child $v^{(o,r)}$ of v. (*ii*) If the node has been expanded, we select a node with the maximum utility from the descendants of this node.

2. Subsequent Cost Estimation: For the selected node v, we estimate its *subsequent cost reduction*, which corresponds to the maximal cost reduction from v to its descendant node. Intuitively, according to the traditional MCTS algorithm [7], we can randomly explore $\mathcal{K}$ descendants of v and take the maximum cost reduction as *subsequent cost reduction*. However, since v may have a large number of descendants and it is expensive to estimate the cost, it is hard to obtain the highest cost reduction through sampling. Therefore, we propose a deep estimation model to compute relatively accurate $C^{\downarrow}(v)$ by considering both the query operators, rewrite rules, and column features (see Section 4).

3. Utility Update. As the subsequent cost reduction of v is updated from 0 to $C^{\downarrow}(v)$, some ancestors may have smaller benefit than v, and we need to update their subsequent cost reductions based on that of v, i.e., redirecting to a descendant node with higher cost reduction. Thus, for each ancestor v' of v,

(*i*) If v has larger benefit than v' (i.e., $C^{\uparrow}(v) + C^{\downarrow}(v) > C^{\uparrow}(v') + C^{\downarrow}(v')$), then the subsequent cost estimation of v' is not correctly estimated, and we update $C^{\downarrow}(v')$, i.e.,

$$C^{\downarrow}(v') = (C^{\uparrow}(v) + C^{\downarrow}(v)) - C^{\uparrow}(v')$$

(ii) We increase the frequency of v' by 1 (i.e., $\mathcal{F}(v') = \mathcal{F}(v') + 1$).
(iii) We update the utility of v' according to Definition 6.

Termination. We repeat above steps until arriving the maximum iteration number or meeting the performance expectation, i.e., the estimated cost reduction $C^{\downarrow}(v_0)$ is approaching $\min(\text{Cost}(v_0) - \text{Cost}(v_i))$, where v_i is a tree node.

Algorithm. Algorithm 1 shows the pseudo code. It first initiates the *policy tree*, with a root node denoting the origin query (line 1). And then it iteratively explores promising rewrite orders by (1) selecting high utility node and expanding its child nodes on the *policy tree* (line 3); (2) estimating the subsequent cost reduction of the selected node (line 4); (3) updating the *policy tree* based on the estimation results (line 5). Finally, it outputs the query with the minimum cost in the *policy tree* (line 6).

EXAMPLE 6. *For a SQL query q, Figure 2 shows a running example of finding the optimal rewrite orders. First, after initiating the policy tree, we compute the maximum utility node v_1, which has the highest cost reduction among all the expended nodes. As v_1 has been selected, we further select v_2 from the subtree rooted at v_1. As v_2 has not been selected, next we estimate the subsequent cost reduction of v_2, i.e., $C^{\downarrow}(v_2) = 0.1$. And then, we update the policy tree from two aspects: (1) For v_2, we update the subsequent cost reduction with 0.1, and increase the access frequency by one; (2) For the ancestor v_1, because the node benefit of v_2 is lower than that of v_1 (i.e., 0.2+0.1<0.3+0.1), we only increase the access frequency of v_1 by one. Finally, we incrementally update the utility values for all the existing tree nodes.*

4 DEEP REWRITE ESTIMATION

Given a tree node (a rewritten query) in the *policy tree*, it is vital to estimate its subsequent cost reduction to compute the utility and find optimal rewrite orders (i.e., high cost reduction). However, there are three main challenges. (C1) There are multiple factors (e.g., rewrite rules, query operators, data distributions) that can affect the cost reduction and it is challenging to characterize these factors from different feature spaces. (C2) The rewrite rules are correlated and the cost reduction of applying multiple rules is not the sum of the cost reduction of each rule, because different rules may conflict (e.g., applying a rule may make other rules inapplicable). It is challenging to learn the rule correlations and detect the optimal rule order. (C3) Deep learning models require a large volume of labeled training data, where a label data includes a query and its optimal rewritten query which has the largest cost reduction. However, it is expensive to obtain the optimal rewritten queries. To address these issues, we propose a multi-ahead attention model. We first present the framework in Section 4.1, and then discuss the models in Section 4.2 and present the training step in Section 4.3.

4.1 Model Framework

Overview. Figure 3 shows the architecture of our framework.

(1) Rewrite Feature Encoding. To characterize the factors from different feature space (C1), we extract effective features (i.e., rewrite rules, query operators, metadata) that affect the rewrite benefit, encodes them into the same feature space, and outputs embedding vectors/matrix for these features (see Section 4.2.1).

(2) Rewrite Rule Embedding. To capture the rule correlations (C2), we use a multi-ahead attention model to learn the rule correlations between different rules (e.g., rule conflicts) and produces an embedding vector for each rule that captures rules that may work together with this rule. It takes the embeddings (output of Step 1) as input and outputs the rule embedding vectors (see Section 4.2.2).

(3) Rewrite Rule Selection. From all the rule embedding vectors for each rule, we select the optimal rule embedding vector with the highest benefit and outputs this optimal rule vector. To this end, it first normalizes the rule embedding vector (output of Step 2) using the *sigmoid* function, and then selects the optimal one using the fully connection neural network (see Section 4.2.3).

(4) Rewrite Cost Estimation. With the selected optimal rule vector (output of Step 3), it estimates the cost reduction of the rules. To estimate the actual cost reduction, it first utilizes another attention layer to learn the relations between the embedding rules (output of Step 3) and the related data and operator features (output of Step 2) and conducts non-linear transformation to convert the embedding vector into the cost reduction (see Section 4.2.4).

Training. As it is costly to get the labeled data (expensive to get the optimal rewrite orders), we first cluster a large number of queries by the operator features, label a query in each cluster (get the optimal rewritten query and the cost), take the cost reduction of the selected query as the ground truth of other queries in that cluster, and train the queries with the approximate ground truth (see Section 4.3).

4.2 Rewrite Cost Estimation Models
4.2.1 Rewrite Feature Encoding

There are three main factors that affect rewrite costs (Figure 3-①).

Rewrite Rule Features $M_{n \times m}^{R}$. The reduced cost of a rewrite rule mainly depends on (1) the operators that can be rewritten by the rule and (2) the rewrite cost of applying the rule to rewrite the operator (e.g., reorder two operators). Hence, we build a rule matrix M^R with n rows (rules) and m columns (operators) to denote the rewrite features, where $M^R[i, j] = 0$ if rule r_i is not applicable to operator o_j; otherwise $M^R[i, j]$ is the cost reduction of applying r_i to operator o_j, which can be estimated by the optimizer. For example, in Figure 3, rule r_1 can rewrite o_1 and o_4, and its encoding rewrite rule vector is $[0.01, 0, 0, 0.01, 0, 0]$.

Query Column Features $M_{k \times m}^{Q}$. Query column features (e.g., index and distinct values) also affect the actual cost reduction. We use a matrix M^Q with m rows (operators) and k columns (database columns) to encode the column features, where $M^Q[i, j]$ denotes whether operator o_i contains column c_j: 1 yes; 0 otherwise.

Metadata Features $M_{2 \times k}^{D}$. Metadata affects the rewrite benefit. For example, inner join works better on indexed columns; and outer join gains lower cost when the right join column has few distinct values (e.g., the *gender* attribute). Hence, we define the metadata matrix M^D with 2 rows and k columns from two aspects: (1) *Index Utilization.* $M^D[0, i] = 1$ if the column i has an index, and $M^D[0, i] = 0$ otherwise; (2) *Distinct Value* indicates the data distribution. $M^D[1, i]$ is the distinct value ratio of this column.

To capture both the query and metadata feature, we integrate M^Q and M^D together and generate a $M_{m \times 2k}^{Q+D}$ vector, where the first k columns capture the index information and the second k columns capture the distinct value information. Specifically, $M^{Q+D}[i, j] = M^Q[j, i]^T \circ M^D[1, j]$ for $j < k$; and $M^{Q+D}[i, j] = M^Q[j - k, i]^T \circ M^D[1, j - k]$ for $j > k$, where $\circ$ is a dot product operation.
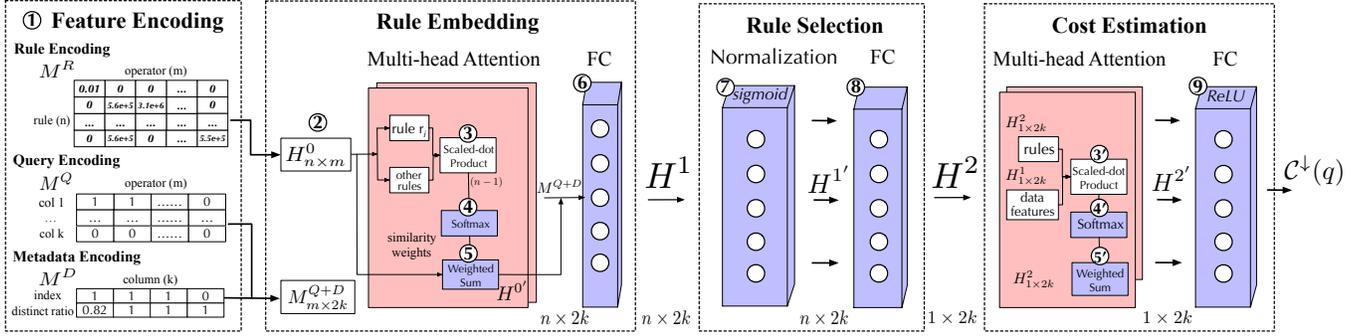
Figure 3: Learned query rewrite model.

4.2.2 Rewrite Benefit Embedding

With the three rewrite features, $[M^Q, M^R, M^D]$, the Rule Embedding Module learns rewrite benefits (e.g., conflicts with other rules) from those factors and outputs rule embedding vectors.

Step 1 - Learn the rewrite correlations from the rule matrix. M^R encodes the cost reduction of each rule. However, M^R cannot reflect the correlations between different rules, which are vital to obtain the overall cost reduction. To this end, we utilize an attention based mechanism to encode input rules. (1) We take M^R as the input matrix of the multi-head attention layer, i.e., $H^0 = M^R$ (Figure 3-②); (2) For any row $H^0[i, *]$, we compute the similarity with all the other rows, i.e., $\alpha_{i,j} = \frac{H^0[i,*]^T \cdot H^0[j,*]}{\sqrt{m}}$, where $j \neq i$ and m is the size of the rows (equals to the operator number). Note that we divide by the square root of the dimension ($\sqrt{m}$) to keep the products in a certain scale. We use $\alpha_{i,j}$ as the attention weight between the two rules, indicating the similarity of their cost vectors. And the higher the attention weight gets, the higher possibility that r_j can affect the rewrite benefit of r_i (Figure 3-③); (3) With attention weights between two rules, we compute the rule combinations for each rule r_i, such that the rules in the combination cause few rewrite conflicts (e.g., few same operators) and high cost reduction; (4) We use a *softmax* function to normalize the attention weights ($\alpha_{i,j}$), i.e., $\alpha'_{i,j} = \frac{exp(\alpha_{i,j})}{\sum_{i,j} exp(\alpha_{i,j})}$, which represents the degree that the rule r_j can affect the rule r_i in cost reduction (Figure 3-④); (5) For any row in $H^0[i, *]$, we divide all the other rules by the normalized attention weights $\alpha'_{i,j}$ (denoting the side-effects of other rules) and sum them with the rule vector $H^0[i, *]$, i.e., $H^{0'}[i, *] = H^0[i, *] + \sum_{j \neq i} \alpha_{i,j} H^0[j, *]$, where $H^{0'}$ are the embedding vectors that represent the rewrite benefits of rule r_i by considering both the rewrite features of r_i and the other rules (Figure 3-⑤).

Step 2 - Embed the overall rewrite benefits based on the data features. We input both the embedding vector $H^{0'}$ and matrix M^{Q+D} into a fully connected layer (FC), where each neural unit denotes a rule combination. To learn the actual cost reduction of any rule combination, we input the multiplication of the embedding matrix $H^{0'}$ and M^{Q+D} (section 4.2.1) and learn the optimal combination for each rule, i.e., M^D, i.e, $H^1 = W^1 H^{0'} M^D + b$, where W^1 is the network weights learned from the training data, b is the standard deviation from the cost space. Any row $H^1[i, *]$ embeds the maximum cost reduction that can be reduced by rule r_i and the rules that can be co-used with r_i (Figure 3-⑥).

4.2.3 Best Rule Selection

Since H^1 represents the n cost embeddings of the rules, next we further embed H^1 and select the optimal rule order as a $1 \times 2k$ dimension vector with two steps.

Step 1 - Normalize the n combinations. We first add a *sigmoid* activation function to normalize the rows in H^1, i.e., $H^{1'} = \frac{1}{1+e^{-H^0}}$, so as to limit the values of the input embedding matrix within the same value range (Figure 3-⑦).

Step 2 - Learn the maximal cost reduction from the n embeddings. With the normalized embedding matrix $H^{1'}$, we utilize a fully connected layer to further combine the n rows of $H^{1'}$ (applying more rules together), learn the cost reduction, and output the best combination ($1 \times 2k$ dimension vector), which denotes the rules with the maximum cost reduction H^2 (Figure 3-⑧) .

4.2.4 Rewrite Cost Estimation

We learn the overall cost reduction based on H^2, i.e., the n embedding vectors of optimal rules. Since the values in H^2 are normalized, we first compute the data factors to the actual cost with another multi-head attention layer, whose output vector is denoted as $H^{2'}$. And then, as the overall cost reduction is not simply the sum of all the values in $H^{2'}$, we conduct nonlinear transformation on $H^{2'}$ to learn the actual cost reduction, i.e., $\hat{C}^{\downarrow}(q) = H^3 = ReLU(W^3 H^{2'})$, where $\hat{C}^{\downarrow}(q)$ is the estimated subsequent cost reduction of q, and W_3 denotes the network weights. We apply the *ReLU* function (i.e., $f(x) = max(0, x)$) to learn the nonlinear mapping from $H^{2'}$ to $\hat{C}^{\downarrow}(q)$ based on the network weights W_3 (Figure 3-⑨).

4.3 Training Mechanism

We discuss the training mechanism for the estimation model. The optimal rewrite order of a SQL query is hard to obtain, because even for a query with a few operators, there can be a large number of possible rewrite orders. Hence, to train the learning model with limited labeled data, we propose an effective training mechanism.

Training Data Generation. The training data is a set of $\langle q, R, D, C^{\downarrow}(q) \rangle$, where q is a query, R is the set of rewrite rules, D denotes the meta data features, and $C^{\downarrow}(q)$ is the highest subsequent cost reduction of q. We take the first three terms as the input features and take the last term as the label. For query q, we generate slow queries (e.g., over one second) as training data, which have large potential to be rewritten. To this end, we adopt a two-step method. First, to generate some representative queries, we randomly assemble {Table, Join, Predicate, Aggregate Operation, Column} based

on query structures (e.g., correlated subquery, uncorrelated subquery, tree structured predicates). Second, we utilize SQLSmith, a random query generator, to synthesize queries via the SQL syntax tree. For generating the highest cost reduction of q, as it is time-consuming to collect the labels (e.g., months to run all rule combinations for 1K queries), we first cluster all queries (e.g., DBSCAN) based on their cost vectors (e.g., six types of query operators and corresponding summed costs); then, for each cluster we sample 5% queries to enumerate their optimal rewrite costs, and compute the average cost reduction as the label for queries in the cluster.

Training. To efficiently train the model, for each sample, we compute the loss function based on both the sample query (accurate evaluation) and queries in the same cluster (robust evaluation) and update the network weights based on the loss function values.

Loss Function. Loss function is vital for rewrite cost estimation, which measures the accuracy between the estimated cost reduction and real cost reduction. However, there are many noises in the training data (i.e., we label the queries based on a small part of the queries), which will slow down model convergence and affect the final accuracy. Hence, to improve both estimation accuracy and training efficiency, we use both the query q in a training sample and other queries Q^* in the same cluster of q to compute the estimation loss. (1) To evaluate the estimation accuracy, we calculate the loss on query q using the Mean Squared Error (MSE), i.e., $L_0 = (F(q) - C^{\downarrow}(q))^2$, where $F(*)$ denotes our estimation model and $C^{\downarrow}(q)$ is the labeled cost reduction of q; (2) To smooth the estimation results across queries within the same cluster, we calculate the loss on other queries of the cluster of q as a Laplacian regularization term: $L_{reg} = \sum_{i,j} \mu_{i,j} ||F(q_i) - F(q_j)||$, where $|F(q_i) - F(q_j)|$ denotes the L1 distance, in order to minimize the sum of absolute differences. L_{reg} represents that the queries within the same cluster have similar operator costs and should have similar subsequent cost reduction. Finally, we denote the overall loss function as: $L_{total} = L_0 + \gamma L_{reg}$, where γ is a weight factor to tradeoff the importance between the accuracy and estimation robustness.

5 PARALLEL QUERY REWRITE

The *policy tree* can be very large (e.g., with 10 rewrite operations and over 30,000 tree nodes) and the policy tree search algorithm may require hundreds of iterations to find the optimal rewrite orders, which is not tolerable for queries that require to be answered within milliseconds. To reduce the rewrite overhead, we need to search the tree in parallel, i.e., selecting multiple nodes at each iteration. For example, in Figure 4, if we select one node to expand each time, it requires three iterations to obtain the "optimal" branch (Figure 4 (a)). Instead we can find the optimal tree node by simultaneously selecting three nodes (Figure 4 (c)).

Intuitively, we want to select the top-τ nodes with the highest utilities. However, the utility of one node may affect the utility of other nodes based on Definition 6. Given two nodes v_i, v_j, if they have no ancestor-descendant relationships, their node utilities will not be affected by each other, because we only update the ancestors and itself of any selected node; otherwise supposing v_i is an ancestor of v_j, if v_j is selected, v_i's utility will be affected as the access frequency will be increased by 1 and the subsequent cost reduction may be updated. To address this issue, we want to select
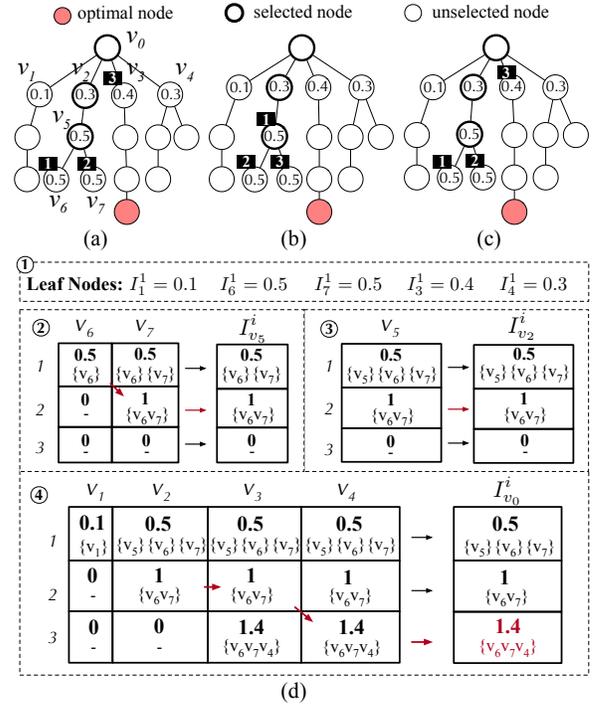
Figure 4: Comparison of Node Selection Strategies. (a) Select single nodes for three times. (b) Select top-3 nodes with maximum total utility. (c) Select three nodes with maximum total utility and without ancestor-descendant relations. (d) Select three nodes using dynamic programming.

top-τ nodes with the highest overall utility such that the selected nodes have no ancestor-descendant relationships.

Multi-Node Selection Problem. Given a policy tree T and a given number τ, we aim to select a subset $V^* \subset V$ of nodes in *policy tree*, which satisfies (1) the node number is τ; (2) any two nodes $v_i, v_j \in V^*$ do not have ancestor-descendant relations; (3) the sum of utilities of nodes in V^* is maximized. For example, in Figure 4, although v_5, v_7 have high utilities, they cannot be selected together because they have ancestor-descendant relationships.

DEFINITION 7 (MULTIPLE NODE SELECTION). *Given a policy tree T and an integer τ, we select the subset $V^* \subset V$ with τ nodes such that (1) any two nodes $v_i, v_j \in V^*$ have no ancestor-descendant relationship, and (2) V^* has the highest utility, i.e.,* $\arg\max_{V^* \subset V} \sum_{v \in V^*} \mathcal{U}(v)$.

Node Selection Algorithm. A naive method first selects the node with the largest utility, and then greedily selects a node from the remainder nodes which has the largest utility and has no ancestor-descendant relationships with selected nodes. However, this method may not select the optimal nodes. To address this issue, we propose a dynamic programming algorithm to select the optimal τ nodes without ancestor-descendant relationships in a bottom-up way.

We first discuss how to select τ nodes under node v which have the largest utility without ancestor-descendant relationships. Let S_v^i denote selecting i nodes with the largest utility under node v without ancestor-descendant relationships, and $I_v^i = \sum_{u \in S_v^i} \mathcal{U}(u)$ denote the corresponding utility.

Computing S_v^i and I_v^i when v is a leaf node. If v is a leaf, $S_v^1 = \{v\}$ and $I_v^1 = \mathcal{U}(v)$, and $S_v^i = \phi$ and $I_v^i = 0$ for $1 < i \leq \tau$.

Computing S_v^i and I_v^i based on v's children. If v is not a leaf node, let $c_1, c_2, \cdots, c_x$ denote the set of children of v in the policy tree. Suppose $S_{c_z}^i$ and $I_{c_z}^i$ of child c_z have been computed for $1 \leq z \leq x$ and $1 \leq i \leq \tau$. Next we discuss how to use them to compute S_v^i and I_v^i. Obviously, v has ancestor-descendant relationships with any node under v, and thus v can only appear in S_v^1 and will not appear in $S_v^{i>1}$. We first consider the case of not selecting v. Let $\mathcal{I}$ denote a matrix with τ row and x columns, where $\mathcal{I}[i, j]$ is the maximal utility of selecting i nodes with the largest utility from nodes under $c_1, \cdots, c_j$. Next we discuss how to compute $\mathcal{I}$.

First, we compute the first row. Any node under c_y and c_z for $1 \leq y \neq z \leq \tau$ have no ancestor-descendant relationships. Thus $\mathcal{I}[1, 1] = I_{c_1}^1, \mathcal{I}[1, 2] = \max(I_{c_1}^1, I_{c_2}^1), \cdots, \mathcal{I}[1, x] = \max_{1 \leq j \leq x} I_{c_j}^1$.

Next, we compute the i-th row based on the first $i - 1$ rows. $\mathcal{I}[i, 1] = I_{c_1}^i$. We compute $\mathcal{I}[i, j]$ by considering the following cases.

(1) We do not select any node under c_j. In other words, we select i nodes from the first $j - 1$ children, i.e., $\mathcal{I}[i, j] = \mathcal{I}[i, j - 1]$.

(2) We select i nodes under c_j. Thus we have $\mathcal{I}[i, j] = I_{c_j}^i$.

(3) We select z nodes under c_j for $1 \leq z < i$. In other words, we select $i - z$ nodes from the first $j - 1$ children, i.e., $\mathcal{I}[i, j] = \mathcal{I}[i - z, j - 1] + I_{c_j}^z$. Thus we have the following recursive function.

$$\mathcal{I}[i, j] = \max \begin{cases} \mathcal{I}[i, j - 1] & \text{0 node under } c_j \\ I_{c_j}^i & i \text{ nodes under } c_j \\ \mathcal{I}[i - z, j - 1] + I_{c_j}^z & z \in [1, i - 1] \text{ nodes under } c_j \end{cases}$$

We use the dynamic programming algorithm to compute $\mathcal{I}$. Then we can get $I_v^i = \mathcal{I}[i, x]$, and obtain S_v^i by selecting the nodes to maximize $\mathcal{I}[i, x]$. Next, if we also select v, we only need to update I_v^1 and S_v^1. If $\mathcal{U}(v) > I_v^1$, we update $I_v^1 = \mathcal{U}(v)$ and $S_v^1 = \{v\}$. Finally, we output $S_{v_0}^\tau$ as the selected nodes, such that achieving highest overall utility without ancestor-descendant relations.

Algorithm Complexity. First, we compute each entry in the matrix $\mathcal{I}$ for each node v, and the number of entries is $O(\tau|v|)$, where v is the number of children of v. Second, for any entry $\mathcal{I}[i, j]$, we compare at most i entries and the time complexity is $O(\tau)$. Hence, the time complexity for node v is $O(\tau^2|v|)$ and the overall time complexity for the tree is $O(\tau^2 \sum_v |v|) = O(\tau^2|T|)$, where $|T|$ is the number of nodes in the policy tree.

EXAMPLE 7. *Figure 4 shows the matrix to select three nodes from the policy tree to achieve the highest overall utility without ancestor-descendant nodes. We first compute the entries in row $\mathcal{I}(1, *)$, where any $\mathcal{I}(1, j)$ ($j \in [1, 7]$) denotes the highest utility to select one node from the first j nodes. For example, $\mathcal{I}(1, 1) = \mathcal{U}(v_1) = 0.1$ and $\mathcal{I}(1, 2) = \max\{\mathcal{I}(1, 1), \mathcal{U}(v_2)\} = 0.3$. Next, for each entry $\mathcal{I}(2, j)$ in the second row, we compare (1) selecting two nodes without using the node v_j and (2) selecting two nodes using the node v_j. Considering the entry $\mathcal{I}(2, 6)$, we can either select v_3, v_5 without using v_6 ($\mathcal{I}(2, 5) = 0.9$) or select v_3 and v_6 ($\mathcal{I}(1, 3) + \mathcal{U}(v_6) = 0.9$). We cannot combine v_5 and v_6 for higher utility, as they have parent-child relationship (v_5 relies on the update of v_6). So $\mathcal{I}(2, 6) = 0.9$ for $\{v_3, v_5\}$ and $\{v_3, v_6\}$.*

6 EXPERIMENTS

We evaluated our techniques and demonstrated the experimental results from several aspects. (1) We compared the performance (e.g, execution cost, query latency, rewrite latency) of `LearnedRewrite`

Table 3: Parameters of Rewrite Estimation Network.

Step	Layer	Dimension
1	Attention + Full-Connection	82×20
2	Norm (Sigmod) + Full-Connection	$82 \times$ column_num
3	Attention + ReLU	$1 \times$ column_num
4	Output	1

with three typical rewrite orders (top down, bottom up, arbitrary); (2) We separately evaluated the efficiency of components in `LearnedRewrite` (i.e., policy tree search, rewrite estimation, and parallel rewrite algorithms); (4) we evaluated the adaptability of `LearnedRewrite` under different rules and queries.

6.1 Experiment Setting

We utilized rewrite rules in Calcite [6], an advanced query engine that independently encapsulates rewrite rules and supports user-defined rewrite orders. And we executed the rewritten queries in popular databases (e.g., PostgreSQL). The server was a machine with 16GB RAM, 256GB disk, 4.00GHz CPU. We used Pytorch to implement the neural networks (estimation module only) and the hyper-parameters were shown in Table 3. We trained the neural networks on a Titan RTX 2080Ti GPU with 11GB frame buffer. Our system was open-sourced and publicly available on Github (https://github.com/zhouxh19/LearnedRewrite).

Datasets. To verify the efficiency of `LearnedRewrite` on different scenarios, we conducted experiments on three types of datasets. (1) TPC-H is an OLAP benchmark. It contains 62 columns and 10,673 synthetic queries whose latency is over 1s. We separately tested `LearnedRewrite` with different data sizes, i.e., TPC-H 1x (~4.7G) and TPC-H 50x (~50.0G). (2) JOB is an OLAP benchmark [16]. It uses a real-world dataset IMDB which contains 134 columns, 1.1G data, and 15,750 synthesis queries whose latency is over 1s. (3) XuetangX is a real-world OLTP benchmark for online education. We took 14 tables with 204 columns, 11.5G data, and 22,000 real queries whose latency was over 1s. For each dataset, we randomly generated complex queries with tools like SQLSmith [33], and split into training/validation/test sets by 8:1:1 (9-fold cross-validation).

Hyper-parameters. Table 3 shows the network parameters. We separately trained three networks for the three datasets. The learning rate was 0.001 and epoch number was 2.

Rewrite Metrics. We evaluated `LearnedRewrite` using four metrics. (1) `Execution Cost`: we first used the execution cost provided by the optimizer to evaluate the quality of a rewritten query, which indicated the performance under the same physical optimization strategy; (2) `Rewrite Latency`: the time of rewriting a query; (3) `Query Latency`: the actual time of answering a rewrite query; (4) `Overall Query Latency`: the overall runtime of rewriting and answering a query. High rewrite latency may slow down the overall latency, especially for relatively simple queries in OLTP workloads. We executed each query for three times and took the average latency to reduce runtime noises. For each metric, we verified the performance on median, 90th, and 95th, because (1) the average metric was easily affected by extreme values, so we used median to report the overall performance; (2) 90th/95th percentiles showed the performance in the worst cases.

(a) Execution Cost   (b) Query Latency
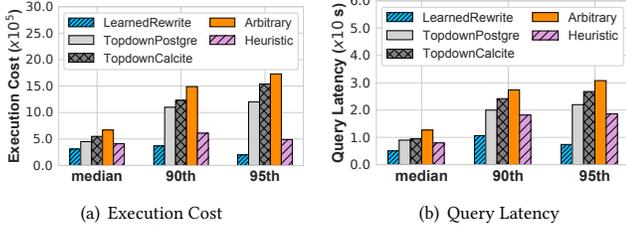
Figure 5: Performance Comparison on 1G TPC-H Queries.

(a) Execution Cost   (b) Query Latency
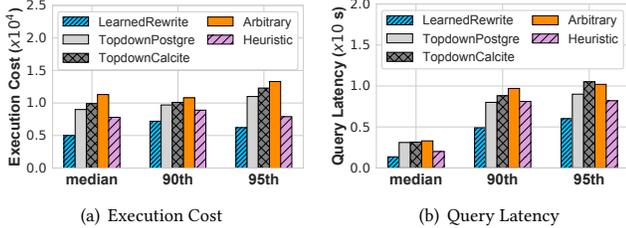
Figure 6: Performance Comparison on JOB Queries.

(a) Execution Cost   (b) Query Latency
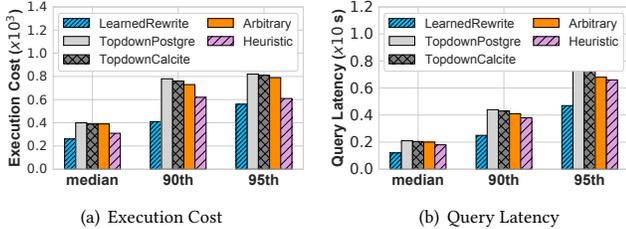
Figure 7: Performance Comparison on XuetangX Queries.

Baseline Methods. We compared the performance of LearnedRewrite with three typical rewrite orders.

(1) TopdownPostgre and TopdownCalcite rewrote from the root node in a top-down order, which are separately the default rewrite orders of PostgreSQL and Calcite.

(2) Heuristic matched any rewrite rules with the query tree iteratively until no more rules can be applied [32].

(3) Arbitrary rewrote any operators in the tree until all the operators cannot be rewritten, which was supported in Calcite [6].

To obtain rewrite latency, we approximated the rewrite latency of TopdownPostgre by the "planning time" statistics in the execution plans, and timed the code running time for the other methods.

6.2 Performance Comparison

We compared LearnedRewrite with four baselines (topdown in PostgreSQL, topdown in Calcite, arbitrary in Calcite, and heuristic in Calcite) on the standard datasets. We separately demonstrated their performance on the test sets in Figures 5-7.

Execution Cost Reduction. Figures 5-7(a) showed the cost of rewritten queries. LearnedRewrite outperformed the other rewrite strategies on all the datasets. For example, the overall latency (i.e., the rewrite latency and query latency) of LearnedRewrite was over 46.9% less than TopdownPostgre, 52.3% less than TopdownCalcite, 80.9% less than Arbitrary, and 19.9% less than Heuristic on the TPC-H dataset. The reasons were two-fold. Firstly, LearnedRewrite explored more beneficial rewrite orders with much lower execution cost than the default top-down order in PostgreSQL and Calcite. For example, with an outer join, TopdownPostgre cannot first push predicates close to the input table, but LearnedRewrite rewrote by first converting outer-join into inner-join and then pushing down the predicate. Second, the

Table 4: Average Rewrite/Query Latency on 50G TPC-H.

Method	Rewrite Latency	Query Latency
Arbitrary	3.3 - 10.1 ms	553.2 s
TopdownPostgre	**0.3 - 3.9 ms**	427.5 s
TopdownCalcite	1.5 - 18.9 ms	431.1 s
Heuristic	5.8 - 24.2 ms	331.7 s
LearnedRewrite	6.1 - 69.8 ms	**224.5 s**

estimation model can predict the potential cost reduction and guided LearnedRewrite to efficiently select orders. For queries with low previous cost reduction and high subsequent cost reduction, LearnedRewrite found the optimal orders by first sampling rarely selected orders (low previous cost reduction) and then finding the optimal rewrite orders (high subsequent cost reduction). Instead, the baselines only found sub-optimal rewrite orders, because Heuristic only selected orders with high previous cost reduction, and Arbitrary randomly sampled rewrite orders and had high possibility to find sub-optimal ones within limited steps.

Query Latency Reduction. Figures 5-7(b) showed the latency of rewritten queries. We had two observations. First, LearnedRewrite outperformed existing methods, because LearnedRewrite can find high-quality rewrite orders based on the policy tree search method and significantly reduced the query latency than other methods. For example, LearnedRewrite achieved over 23.7% latency reduction than the baselines in average. Second, LearnedRewrite worked better on TPC-H than JOB, because TPC-H queries contained many subqueries that can be removed by *query rewrite*.

Rewrite Latency. We also evaluated rewrite latency of each method and Table 4 showed the results. Topdown rewrite order took the least rewrite latency, but the overall query latency was worse than Heuristic and LearnedRewrite; and LearnedRewrite took the highest rewrite latency, but achieved the lowest overall latency. The reasons were four-fold. First, LearnedRewrite took a bit more time (e.g., average 20ms on TPC-H) to try out different rewrite orders by the exploration-and-exploitation policy, and found rewrite orders with much higher cost reduction and improved the slow queries by much more time. Instead, other methods only adopted limited rewrite orders, i.e., by default (topdown in PostgreSQL), greedily (Heuristic), or randomly (Arbitrary), and caused sub-optimal rewritten queries. Second, LearnedRewrite needed to estimate the subsequent cost reduction of each selected rewrite order, which took a bit time (e.g., average 0.5 ms for each query) but enhanced efficiency by identifying optimal orders at early iterations. Note that LearnedRewrite took around 40 minutes (on 10,673 queries) to train the estimation model, but can generalize to any queries on the same dataset. Third, TopdownPostgre and TopdownCalcite took the smallest rewrite latency, because it only traversed the plan tree once. Fourth, compared with TopdownPostgre, Heuristic and Arbitrary took higher rewrite latency, because they repeatedly matched applicable rules and conducted more redundant rewrites.

Summary. LearnedRewrite outperformed existing methods and achieved lower latency on the three datasets, i.e., gaining 37.5% latency reduction on TPC-H, 30% on JOB, and 29.3% on XuetangX.

6.3 Evaluation on Our Techniques
6.3.1 Evaluation on the exploration parameter γ

Since LearnedRewrite was based on the tree search algorithm, the exploration parameter γ balanced between selecting high cost reduction orders and selecting rarely accessed rewrite orders (with
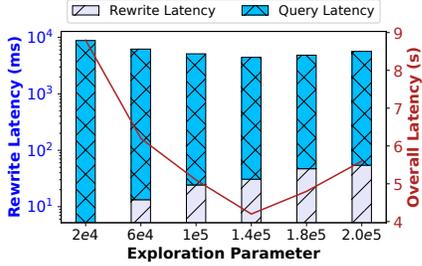
Figure 8: The Impact of Exploration Parameters (TPC-H).

(a) Estimation Accuracy (b) Overall Latency
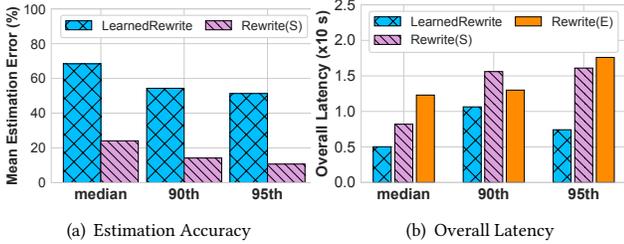
Figure 9: Comparison of Estimation Methods (TPC-H). `Rewrite(S)`: sampling. `Rewrite(E)`: enumerating all orders.

(a) Rewrite Latency (b) Overall Latency

Figure 10: Comparison of Parallel Algorithms (TPC-H). `DP` denotes DP-based parallel rewrite; `TopK` denotes greedy parallel rewrite; and `Single` denotes selecting single node.

(a) Rewrite Latency (b) Overall Latency

Figure 11: Comparison of Tree Search Algorithms (TPC-H).

potential high subsequent cost reduction), which was vital to the query performance. Hence, we evaluated the impact of different exploration parameters to the rewrite performance. Figure 8 showed the results. And we made the following observations.

First, exploration parameter γ affected the query latency and rewrite latency. A larger γ expected to try many more rewrite orders, leading to high rewrite latency; while a larger γ leaded to lower query latency, because it found a better rewrite order. Thus we required to select an appropriate γ to balance query latency and rewrite latency to achieve the lowest overall latency. For example, when the exploration parameter γ was smaller than 1.4×10^5, the query latency went down by increasing γ, because the policy tree of a query was generally large (e.g., over 320,000 branches for a 10-operator query); while it could select more rarely selected orders in case of avoiding falling in local optimum and got lower overall latency. However, when γ was bigger than 1.4×10^5, the rewrite latency become higher, because too large exploration parameter took more iterations to exploit the orders with best previous cost reduction, and query latency would be further reduced (cannot find better orders). Thus we needed to select an appropriate γ, i.e., $\gamma = 1.4 \times 10^5$. Second, for slow queries in Figure 8, query latency reduction was higher than the ratio of rewrite latency, and it would be more beneficial to take more time to explore different rewrite orders (e.g., we selected $\gamma = 1.4 \times 10^5$ for slow queries). Moreover, for fast queries running quickly, we could select a relatively small τ (e.g., around 2×10^4) and reduced the rewriting overhead.

6.3.2 Evaluation on the tree search algorithms

We compared with two traditional tree search strategies, i.e., best-first algorithm (`BestFirst`) and depth-first algorithm (`DepthFirst`) [13]. We used our attention-based estimation model to estimate the cost reduction of the selected nodes. As shown in Figure 11, `MCTS` outperformed `BestFirst` and `DepthFirst` in latency reduction, because `MCTS` can balance between selecting nodes with high cost reduction and nodes with low access frequency, which helped to find global optimal queries. Instead, `BestFirst` always selected the nodes with highest cost reduction and leaded to
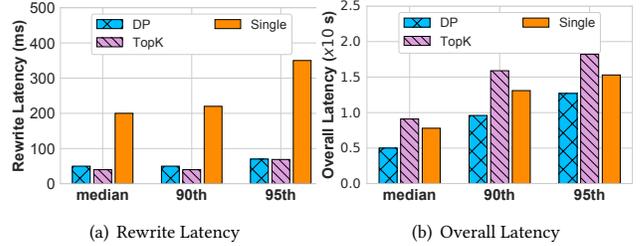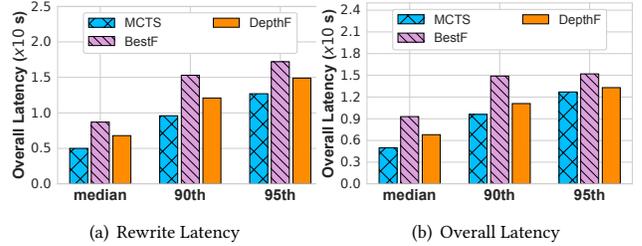
local optimum. `DepthFirst` cannot efficiently explore untouched branches on the policy tree and caused sub-optimum, but via pruning nodes, `DepthFirst` performed better than `BestFirst` within limited iterations.

6.3.3 Evaluation on the estimation model

We compared the estimation accuracy and efficiency of three estimation strategies: (1) enumerate all the rewrite orders and output the highest cost reduction (denoted by `Rewrite(E)`); (2) sample $\mathcal{K}$ rewrite orders, and output the highest cost reduction estimated by a deep neural network [29] (denoted by `Rewrite(S)`); (3) estimate the highest subsequent cost reduction using our estimation model (`LearnedRewrite`). Figure 9 showed the results. And we made the following observations.

First, `LearnedRewrite` gained relatively high estimation accuracy, i.e., over 29% higher than `Rewrite(S)`, because, for `Rewrite(S)`, it was hard to sample rewrite orders to find the highest cost reduction order when the policy tree was large, e.g., sampling 50 orders from 320,000 candidates, and caused great errors even if the neural network can effectively estimate the cost of sampled queries. Instead, `LearnedRewrite` directly predicted the rewrite rule combination with the highest cost reduction by features like query operators and rule conflicts, which were captured by the attention layer and embedded as an embedding vector. Note that `LearnedRewrite` worked much better for queries with high cost reduction (e.g., the queries 9c and 11a in TPC-H), because these queries usually had large rewrite potential and `LearnedRewrite` can simulate more rewrite combinations with multi-head attention than random sampling. Second, `LearnedRewrite` achieved lowest overall latency than `Rewrite(E)` and `Rewrite(S)`. Although `Rewrite(E)` enumerated all the rewrite orders and could derive the optimal rewritten query, it took much longer rewrite time (e.g., over 2000ms) and the overall latency was worse than `LearnedRewrite`. `LearnedRewrite` selected the orders based on the estimation model, which can give relatively accurate results within 10ms. Moreover, `Rewrite(S)` also had higher rewrite latency than `LearnedRewrite` because it needed to sample many rewritten queries and separately estimate their costs, which was more time-consuming than model inference in `LearnedRewrite`.
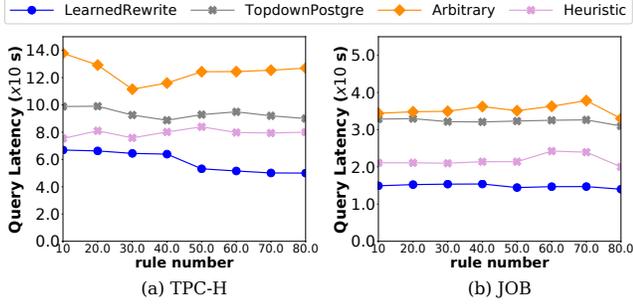
Figure 12: Adaptability on Different Number of Rules.

Figure 13: Adaptability on Different Number of Operators.

6.3.4 Evaluation on the parallel algorithm

Finally we evaluated our parallel rewrite algorithm, which was vital to enhance query rewrite with numerous candidate orders (tree nodes). We compared the efficiency of three methods on `LearnedRewrite`. (1) dynamic programming based node selection method (DP); (2) top-τ based node selection method (TopK); (3) single node selection (Single). The results were shown in Figure 4. And we made two observations. First, parallel algorithms can significantly reduce the rewrite overhead (i.e., the searching time). This was because Single selected one rewrite order (a tree node) at each iteration and could evaluate different orders in parallel. Second, DP gained lowest query latency than TopK and Single. On one hand, TopK greedily selected top-τ nodes, which might come from the same rewrite order, cannot be updated by the selection frequency, and caused suboptimal solution. Instead, DP selected maximum overall utility nodes without ancestor-descendant relations (i.e., the selected nodes were on different rewrite orders) and could find optimal orders. Second, Single took much more time in tree search and caused its overall query latency was worse than DP.

6.4 Evaluation on Adaptability

6.4.1 Varying Number of Rewrite Rules

We first tested the adaptability of `LearnedRewrite` by changing the number of rules. We first ranked the rules by their usage frequency, and varied the number of rules from 10 to 80.

TPC-H Dataset. We made two observations. First, `LearnedRewrite` outperformed the other methods and achieved the lowest query latency with different rule numbers. The main reasons were two-fold. (1) `LearnedRewrite` characterized the rule features as the input of the estimation model. And the well-trained model can adapt to different rule combinations (i.e., padding the positions if corresponding rules are absent). While other methods cannot estimate the rule benefits and selected many useless rewrite operations. (2) `LearnedRewrite` can adaptively learn the benefits of rewrite operations of the new rules, i.e., estimating their utility and exploring the subsequent order if the utility was relatively high. While, lack of the estimation model, other methods cannot rewrite based on the rewrite benefits. Second, the query latency of `LearnedRewrite` decreased by increasing the rule numbers, because it utilized the tree search algorithm to efficiently explore rewrite orders even if the policy tree was large. While other methods used limited rewrite order and cannot select from many potential rewrite orders.

JOB Dataset. We got similar results. `LearnedRewrite` achieved lowest query latency, around 54.9% less than `TopdownPostgre`, 61.1% less than `Arbitrary`, and 38.6% less than `Heuristic`. This
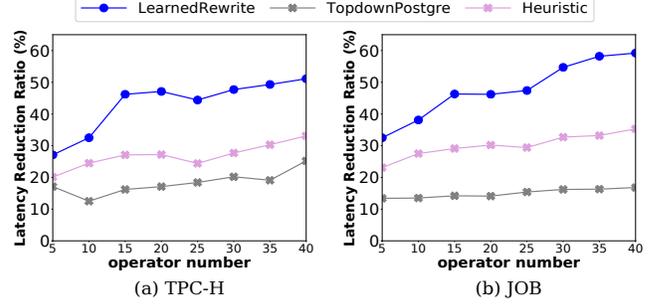
was because the baselines cannot efficiently explore the new rewrite rules and sampled sub-optimal solutions when the rewrite space got large. Instead, `LearnedRewrite` can handle it by exploring the policy tree by the estimation results. However, different from TPC-H, the latency change on JOB was relatively stable, because the logical problems in JOB were much fewer than TPC-H, whose queries had many subqueries, correlations, and complex predicates. Thus, with a small part of rules, we can optimize the JOB queries well.

6.4.2 Varying Number of Query Operators

Next we tested the rewrite performance under different operator numbers. Figure 13 showed the results and we made two observations. First, the methods had similar performance when the operator number was small (e.g., less than 5), as the order space was small and there was little opportunity to reduce the latency. Second, `LearnedRewrite` worked better when the operator number increased, as more operators caused larger rewrite order space and the baselines were hard to gain high performance. Besides, more operators led to more correlations between different query segments (e.g., an outer query referenced by a subquery), and the baselines mismatched many rewrite rules without properly arranging the rewrite orders. Instead, `LearnedRewrite` solved the problem from two aspects. First, it had an efficient tree search algorithm that tried different orders to rewrite the operators and selected the highest cost reduction one as the rewrite result. Second, the estimation model adapted to variable operator numbers by encoding the operator features into the matrix of rules and metadata features, and gave relatively accurate benefit estimation based on different operators.

7 CONCLUSION

We have proposed a learning-based query rewrite system. We first modeled the rewrite orders as a policy tree, and then used *Monte Carlo tree search* to explore the policy tree to judiciously find the optimal rewrite query. We proposed a deep rewrite estimation model that effectively predicted the rewrite benefit of a rewrite order. We proposed a parallel query rewrite algorithm to select multiple nodes at each iteration. Experimental results showed that our method significantly outperformed state-of-the-arts.

REFERENCES

[1] https://github.com/xiaomi/soar/.

[2] https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#citations-below.

[3] https://www.postgresql.org/.

[4] R. Ahmed, A. W. Lee, A. Witkowski, and et al. Cost-based query transformation in oracle. In *VLDB*, pages 1026–1036, 2006.

[5] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2012.

[6] E. Begoli, J. Camacho-Rodríguez, J. Hyde, and et al. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, pages 221–230, 2018.

[7] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games*, 4(1):1–43, 2012.

[8] A. H. M. de Araújo and et al. On using an online, automatic and non-intrusive approach for rewriting SQL queries. *J. Inf. Data Manag.*, 2014.

[9] B. Finance and G. Gardarin. A rule-based query rewriter in an extensible DBMS. In *ICDE*, pages 248–256. IEEE Computer Society, 1991.

[10] Z. Gharibshah, X. Zhu, A. Hainline, and M. Conway. Deep learning for user interest and response prediction in online display advertising. *Data Science and Engineering*, 5(1):12–26, 2020.

[11] F. Giroire. Order statistics and estimating cardinalities of massive data sets. *Discret. Appl. Math.*, 157(2):406–427, 2009.

[12] Y. Han, G. Li, H. Yuan, and J. Sun. An autonomous materialized view management system with deep reinforcement learning. In *ICDE*, 2021.

[13] B. Huang. Pruning game tree by rollouts. In B. Bonet and S. Koenig, editors, *AAAI*, pages 1165–1173, 2015.

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

[15] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[16] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[17] F. Li. Cloud native database systems at alibaba: Opportunities and challenges. *VLDB*, 12(12):2263–2272, 2019.

[18] G. Li, X. Zhou, and L. Cao. AI meets database: AI4DB and DB4AI. In *SIGMOD*, pages 2859–2866, 2021.

[19] G. Li, X. Zhou, and L. Cao. Machine learning for databases. *Proc. VLDB Endow.*, 14(12):3190–3193, 2021.

[20] G. Li, X. Zhou, S. Ji, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. opengauss: An autonomous database system. *VLDB*, 2021.

[21] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *PVLDB*, 12(12):2118–2130, 2019.

[22] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[23] M. Li, H. Wang, and J. Li. Mining conditional functional dependency rules on big data. *Big Data Mining and Analytics*, 03(01):68, 2020.

[24] A. Liu, J. Chen, M. Yu, and et al. Watch the unobserved: A simple approach to parallelizing monte carlo tree search. In *ICLR*, 2020.

[25] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Learning to steer query optimizers. *CoRR*, abs/2004.03814, 2020.

[26] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In *SIGMOD*, pages 1275–1288, 2021.

[27] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In R. Bordawekar and O. Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018.

[28] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *arXiv preprint arXiv:1902.00132*, 2019.

[29] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *CoRR*, abs/1902.00132, 2019.

[30] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[31] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, and A. Jindal. Steering query optimizers: A practical take on big data workloads. In *SIGMOD*, pages 2557–2569, 2021.

[32] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In M. Stonebraker, editor, *SIGMOD*, pages 39–48, 1992.

[33] A. Seltenreich. Sqlsmith, 2020. https://github.com/anse1/sqlsmith.

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

[35] J. Sun, G. Li, and N. Tang. Learned cardinality estimation for similarity queries. In *SIGMOD*, pages 1745–1757, 2021.

[36] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang. Learned cardinality estimation: A design space exploration and a comparative evaluation. *VLDB*, 2021.

[37] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[38] S. Tian, S. Mo, L. Wang, and Z. Peng. Deep reinforcement learning-based approach to tackle topic-aware influence maximization. *Data Science and Engineering*, 5(1):1–11, 2020.

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

[40] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.

[41] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüş, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.

[42] Z. Yang, E. Liang, A. Kamsetty, C. Wu, and et al. Deep unsupervised cardinality estimation. *VLDB*, 13(3):279–292, 2019.

[43] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1297–1308. IEEE, 2020.

[44] H. Yuan, G. Li, L. Feng, and et al. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, pages 1501–1512, 2020.

[45] J. Zhang, Y. Liu, K. Zhou, and G. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, pages 415–432, 2019.

[46] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. In *TKDE*, 2020.

[47] X. Zhou, L. Jin, S. Ji, X. Zhao, X. Yu, S. Li, T. Wang, K. Li, and L. Liu. Dbmind: A self-driving platform in opengauss. *VLDB*, 2021.

[48] X. Zhou, J. Sun, G. Li, and J. Feng. Query performance prediction for concurrent queries using graph embedding. *Proc. VLDB Endow.*, 13(9):1416–1428, 2020.