Rethinking Learned Cost Models: Why Start from Scratch?

JIANI YANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
SAI WU*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
DONGXIANG ZHANG, Zhejiang University, China
JIAN DAI, Alibaba Group, China
FEIFEI LI, Alibaba Group, China
GANG CHEN, Zhejiang University, China

Recent work has applied learning-based approaches to replace the conventional cost model, but these approaches are expensive to train and result in high inference overheads. Furthermore, due to a lack of explainability, models trained for one database may not be easily transferred to another, requiring a complete re-training process. In this paper, we propose a new approach to tuning the conventional formula-based cost model for DBMS. Our approach involves identifying important parameters within the cost model rules and using a fast-learning model to adjust them for each specific hardware and software configuration of the DBMS deployment. We dynamically partition the search space of hardware and software configurations to gradually refine the cost model estimation. To apply our cost model to a new DBMS instance, we start with a rough estimation and progressively refine it with finer granularity. Our experiments with different hardware and software configurations show that our approach enables the conventional cost model to be quickly transferred to any database instance, achieving comparable results to a fine-tuned learning-based model. Overall, our approach provides a practical solution to tuning the conventional cost model for DBMS, with significant benefits in terms of reduced cost and improved performance.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: Cost model, query optimization, explainability

ACM Reference Format:
Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch?. *Proc. ACM Manag. Data* 1, 4 (SIGMOD), Article 255 (December 2023), 27 pages. https://doi.org/10.1145/3626769

1 INTRODUCTION

Database researchers and developers have been working for a long time to summarize empirical formulas that can be used to estimate the cost of a database query. These formulas, collectively known as a formula-based cost model[31], reflect how a database system interacts with the operating system and hardware components such as memory, CPU, and disk. Despite the widespread use of formula-based cost models, they typically require database administrators (DBAs) to tune certain

*Sai Wu is the corresponding author.

Authors' addresses: Jiani Yang, jianiyang_cs@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China; Sai Wu, wusai@zju.edu.cn, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China; Dongxiang Zhang, zhangdongxiang@zju.edu.cn, Zhejiang University, China; Jian Dai, yiding.dj@alibaba-inc.com, Alibaba Group, China; Feifei Li, lifeifei@alibaba-inc.com, Alibaba Group, China; Gang Chen, cg@zju.edu.cn, Zhejiang University, China.

2836-6573/2023/12-ART255 $15.00
https://doi.org/10.1145/3626769

hyperparameters. For example, in PostgreSQL, users can set a ratio to normalize I/O and CPU costs in the formula-based cost model to reflect their relative importance, allowing the query optimizer to compare them. Even if manual tuning is required, formula-based cost models have proven to be effective in many cases and are by far widely used in many mainstream database systems today.

In recent years, deep learning models have gained a lot of attention in the field of cost estimation for database systems [11, 21, 23, 36, 43, 47]. These models leverage end-to-end neural networks to estimate the cost of a query. Given a large number of training samples, learning-based models can generalize the correlation between the query and the cost, and hence often produce more reliable cost estimation results than formula-based models. While they have shown promising results, learning-based models differ from formula-based models in that they are more difficult to interpret and use. This is because deep learning models treat cost estimation as an opaque box process, making it challenging for database administrators to understand how the cost values are produced. When the predicted cost values deviate significantly from the actual values, the learned end-to-end model needs to be retrained, which is costly.

General speaking, deploying learning-based models as the in-database cost models face three key challenges:

Generalization: Typically, learning-based models are trained on a specific database with specific hardware and software configurations. As a result, transferring these models to a new database or different hardware/software environment is not feasible, and a complete re-training process is required. Although building a general pre-trained model for all DBMSs and possible environments would be ideal, it is not feasible in the near future due to the lack of a large repository of structured datasets. Thus, current solutions must build models in an ad-hoc way, which can be time-consuming and resource-intensive.

Training Overhead: Learning-based models require a significant amount of training data, which is typically generated by executing queries on the underlying DBMS to collect their physical plans and corresponding processing costs. However, the size of the query space grows exponentially with the number of tables and columns, making it impractical to perform a thorough sampling. Additionally, the overhead of obtaining a single sample is equal to the processing latency of the query, which can be significant for complex queries. As a result, obtaining enough training data to train a learning-based model can be a time-consuming and expensive process.

Explainability: Furthermore, learning-based models are essentially opaque boxes, which perform end-to-end predictions without providing insights into how the cost estimate was generated. As a result, it is challenging for database administrators to conduct root cause analysis for unexpected processing costs or slow queries. In contrast, formula-based cost models can be more transparent and easier to interpret, allowing DBAs to tune specific hyperparameters and adjust the cost model accordingly. This transparency enables DBAs to understand the cost estimation process and identify any potential issues, leading to more effective performance tuning.

Therefore, the ideal approach would be to combine the advantages of formula-based models, such as their low cost, generalization, and explainability, with the precision of learning-based models. By doing so, we can effectively improve the cost model of existing DBMS. Our strategy is to identify the limitations of formula-based models that result in imprecise cost estimations and address them using light-weighted learning-based approaches. The hybrid model we aim to develop should be easy to train, capable of being transferred to different hardware and software environments, and provide explainable results. Such a hybrid model can improve the precision of cost estimation while

preserving the transparency and interpretability of the formula-based models, thus providing a more effective solution for DBAs to optimize query performance.

In this paper, we argue that the conventional formula-based model can still provide a comparable estimation result to the learning-based one, if its hyperparameters are properly tuned based on hardware, software and data distribution. To achieve this objective, we investigate the potential configurations that affect the hyperparameters of the formula-based cost model and train a tree-like model to recommend settings for all involved hyperparameters on a given combination of configurations. In order to differentiate between the two types of parameters, we employ the term **C-param** to denote configuration parameters and **R-param** to denote the hyperparameters of the formula-based cost model.

To model the correlations between those two types of parameters, we introduce a two-stage learning framework that accounts for different configurations during distinct training stages. In the offline stage, we identify possible static configurations that affect R-params to establish a base model for different hardware and software configurations, providing a warm-up. During the online stage, we enhance the base model by incorporating additional configuration parameters that capture online dynamics, based on query performance statistics. The base model is then fine-tuned gradually by splitting its leaf nodes via the activated C-params. The splitting process resembles that of a decision tree, but with customized splitting criteria to make the prediction of R-params as accurate as possible. Each leaf node represents a subspace of C-params that we use to generate predictions of R-params for various physical operators. When a new query is submitted to the DBMS, we search the tree for the involved subspace and replace the default values of R-params with our predicted values. This process is transparent to the query optimizer.

We make the following contributions in this paper:

- We propose the ParamTree model, which uses the formula-based model as a template and instantiates it with the predicted **R-params**. Experimental results demonstrate that our approach provides more accurate (or comparable) estimation results than retrained or fine-tuned deep learning models with fewer examples.
- ParamTree exhibits good adaptability, allowing for the reuse of a trained model for different hardware configurations, software configurations, and varying data distributions.
- Our model can be integrated non-intrusively with existing query optimizers and its results are explainable.

The remainder of the paper is organized as follows. Section 2 introduces the backgrounds and gives an overview of our approach. Section 3 and Section 4 discuss the offline training stage and online refinement stage of our approach, respectively. Section 5 evaluates the proposed approach and we briefly introduce some related work in Section 6. Section 7 concludes the paper.

2 SYSTEM OVERVIEW

In this section, we first explicitly define our problem and then give an overview of our approach.

2.1 Problem Definition

Major database systems such as PostgreSQL, MySQL, and Oracle use formula-based cost models to estimate computation overhead and disk I/O for specific operators. Table 1[1] summarizes the built-in formulas in PostgreSQL for estimating startup and runtime costs of several common physical operators, including `SeqScan`, `IndexOnlyScan`, `IndexScan`, `Sort`, and `HashJoin`. The startup cost refers to the overhead incurred before fetching the first tuple, while the runtime cost denotes the

[1]These cost formulas are summarized from the source code of PostgreSQL. All values except for the **R-params**, which can be customized by the user, can be calculated. Due to space limitations, we do not elaborate on all operators' cost formulas.

Operator	Cost Type	Cost Formula	
SeqScan	startup_cost	0	
	runtime_cost	cpu	$(r_t + r_o \times N_{qp}) \times tuples\_num$
		disk	$r_s \times table\_pages$
IndexOnlyScan	startup_cost	$\lceil \log_2 index\_tuples \rceil \times r_o + (index\_tree\_height + 1) \times 50 \times r_o$	
	runtime_cost	cpu	$indexSelectivity \times index\_tuples \times (r_t + r_i + r_o \times N_{cp})$
		disk	$\lceil indexSelectivity \times index\_pages \rceil \times r_r$
IndexScan	startup_cost	$\lceil \log_2 index\_tuples \rceil \times r_o + (index\_tree\_height + 1) \times 50 \times r_o$	
	runtime_cost	cpu	$indexSelectivity \times index\_tuples \times (r_i + r_o \times N_{cp} + r_t + r_o \times N_{qp})$
		disk	$\lceil indexSelectivity \times index\_pages \rceil \times r_r + max\_IO\_cost +$ $indexCorrelation^2 \times (min\_IO\_cost - max\_IO\_cost)$ $(max\_IO\_cost = r_r \times \Phi_{Mackert-lohman},$ $min\_IO\_cost = (\lceil indexSelectivity \times table\_pages \rceil - 1) \times r_s + r_r)$
Sort	startup_cost	cpu	$C_t^L + 2 \times r_o \times tuples\_num \times \log_2 tuples\_num$
		disk	**When input_bytes > work_mem:** $2 \times \lceil \frac{input\_bytes}{BLCKSZ} \rceil \times \max(1, \lceil log_{mergeorder} \frac{input\_bytes}{c_w} \rceil) \times (0.75 r_s + 0.25 r_r)$
	runtime_cost	cpu	$r_o \times tuples\_num$
HashJoin	startup_cost	cpu	$C_s^L + C_T^R + R_r \times (r_o \times N_{cp} + r_t)$
		disk	$\text{sgn}(nbatches - 1) \times (r_s \times P_r)$
	runtime_cost	cpu	$C_t^L - C_s^L + r_o \times N_{cp} \times R_l + N_{cp} \times r_o \times R_l \times R_r \times innerbucketsize \times 0.5$ $+R_o \times (r_t + r_o \times N_{qp})$
		disk	$\text{sgn}(nbatches - 1) \times (r_s \times (P_r + 2 \times P_l))$

Table 1. Main cost formulas and R-params for PostgreSQL. R-params are explained in table 2.

cost of manipulating the remaining tuples. These cost formulas typically represent a weighted aggregation of one or more elemental factors. For example, the disk I/O of `HashJoin` is estimated as an aggregation of P_r and P_l, which refer to the number of pages that the right and left relations occupy, respectively. In this cost formula, r_s is a weight parameter, referred to as **R-params** in this paper. Table 2 shows the five overall R-params for PostgreSQL, including r_t, which refers to the CPU cost of processing a tuple. All the formulas in Table 1 follow the same linear template:

$$cost_{op} = \sum f_i(op) \times r_i \tag{1}$$

where r_i refers to the corresponding R-params, $f_i(op)$ denotes the number of cost units for r_i, which is determined by the execution of the physical operator.

Optimizer Parameter	Symbol	Default	Description
cpu_tuple_cost	r_t	0.01	The CPU cost of processing a tuple.
cpu_operator_cost	r_o	0.0025	The CPU cost of processing an operator.
cpu_index_tuple_cost	r_i	0.005	The cost of processing an index entry.
seq_page_cost	r_s	1	The cost of sequentially accessing a disk page.
random_page_cost	r_r	4	The cost of randomly accessing a disk page.

Table 2. R-params for cost model

Optimizing the performance of the cost model requires considering the impact of external factors that may be ignored when using default R-params. We have conducted extensive analysis to identify the external factors that may affect R-params. For example, we observed that the I/O time varies significantly when executing the same query on HDD and SSD, which affects the adjustment of I/O-related R-params. Similarly, processing different types of data(integer, float) also results in varying execution times, affecting the adjustment of CPU-related R-params.

These factors are context-aware and referred to as **C-params**, including hardware resources, underlying OS and DBMS, data distribution, and query workload. To determine the optimal R-params for a specific operator in a query execution plan, these external factors must be taken into account. We can classify the configurations of hardware, OS, and database as static C-params

Category		C-params
Static C-params	hardware	IO read/write speed, CPU clock speed, Storage capacity etc.
	software	Windows/Linux/MAC, PostgreSQL/MySQL, Row/Column etc.
	DBMS configuration	*parameters require a server restart to take effect*: Shared_Buffers, Block_Size etc.
Dynamic C-params	Query-Related	Physical Operator, Structure of the query etc.
	Data-Related	Column Offset, Data Type, Index Correlation etc.
	DBMS configuration	*parameters can be modified without restarting the server*: Work_Mem, Temp_Buffers etc.

Table 3. Types of C-params.

since they remain unchanged during query processing, while parameters such as data type and column correlation are query-dependent and referred to as dynamic C-params. Table 3 provides an overview of these two categories of parameters.

The objective of our paper is to determine optimal choices of R-params in the built-in cost formulas of major databases, in order to achieve more accurate query cost estimation. To accomplish this goal, our research focuses on learning a mapping function from C-params to R-params, denoted as $G : C \xrightarrow{\text{op}} R$, to capture their dependencies for a given physical operator op. By using the learned function G, we can generate recommended choices of R-params for a specific hardware-software setup. In addition to the five operators presented in Table 1, we also implemented other operators such as `Nested Loop`, `Merge Join`, and `Aggregate`.

2.2 Basic Idea

Learning the mapping function $G : C \xrightarrow{\text{op}} R$ for query cost estimation is a major contribution of this paper. It presents a non-trivial regression task, for the generation of training samples can be expensive due to the high dimensionality of C-params and the existence of slow queries. Additionally, adapting the learned mapping function to the varying hardware and OS environments with dynamic query workloads is another major contribution we make. It requires an efficient and effective learning process.

Motivated by these, we propose a novel **two-stage training and refinement** framework. In the offline training process, we build an initial decision tree for the static C-params, as an initialized mapping function (i.e., model). The online refinement process then adaptively evolves the tree with the target query workload and environment according to dynamic C-params. This framework enables us to preserve and utilize the most of existing knowledge on one hand and evolves the model to maintain excellent performance on the other hand. In the following sections, we present the details.

2.3 Architecture

The workflow of the two-stage approach is illustrated in Figure 1 and we explain our design in the following.

2.3.1 Offline training Stage. The objective of the offline training phase is to generate an initial decision tree that captures the relationship between R-params and static C-params.

Although users' databases and workloads may vary, it is possible to adjust and roughly optimize R-params based on any database or workload. In this paper, because we use PostgreSQL as an example to demonstrate our approach, we conduct SQL queries defined in the Job-workload[2] on various instances of a cloud server platform equipped with different CPUs, memory sizes, SSDs, HDDs, operating systems, and other configurable hardware and software features. We leverage this historical data from diversified hardware and software setups to pre-train a decision tree for each operator, which we refer to as the **ParamTree**. This initial ParamTree for static C-params is

[2]https://github.com/gregrahn/join-order-benchmark

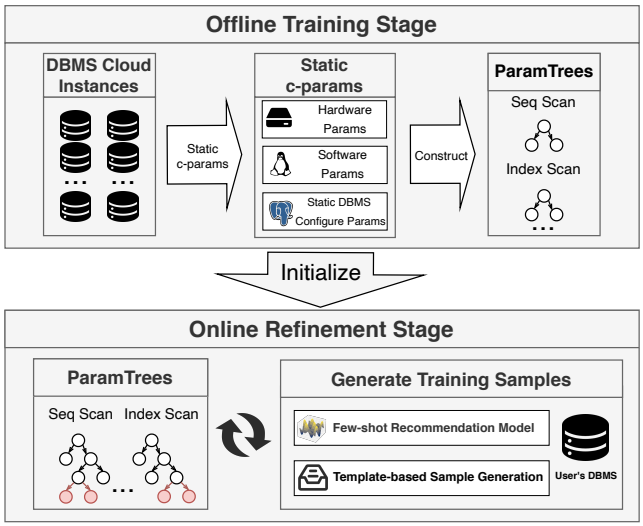


Fig. 1. Overview architecture.

built using the customized decision tree algorithm (Section 3.2), and will be expanded during the refinement process.

2.3.2 Online Refinement Stage. The offline training stage generates an initial ParamTree with static C-params, but this alone may not provide precise cost estimation due to the dynamic C-params that capture query-specific and data-specific factors. Therefore, we propose an online refinement process to adaptively tune the ParamTree.

There are two key differences between the two stages in terms of model training:

- First, the online refinement stage is workload-aware and optimizes the ParamTree for the online workload. If the workload changes, ParamTree's structure is updated correspondingly.
- Second and most importantly, there are hundreds of dynamic C-params (17 query-related, 18 data-related, and 97 DBMS configuration parameters), compared to less than 20 static C-params. This results in the curse of dimensionality during the refinement stage training. Therefore, a different strategy is needed for the online stage. Unlike the passive reception of training samples during the offline training stage, we actively generate samples based on needs in the refinement stage.

The online refinement works as follows: if the cost model using existing ParamTrees cannot provide a precise estimation for more than λ queries (error rate larger than ϵ), online tree expansion is invoked to split the leaf nodes. To cope with the curse of dimensionality, we use a few-shot recommendation model to rank all candidate C-params based on their effects on cost estimations. Then, we adopt a template-based sampling approach to retrieve enough samples to perform a tree split. This process repeats until the cost model generates estimation results for less than λ queries with error rates bounded by ϵ.

2.3.3 Cost Estimations with ParamTree. Finally, we present how to estimate the cost of a particular physical query plan P using our ParamTrees in Algorithms 1, which defines the recursive function $doPlanCostEstimation$. Firstly, we apply the learned ParamTrees, using the current values of C-params, to predict the values of R-params. Then, $fillCostFormula$ denotes computing cost by filling R-params and physical operator's statistics $n.statistics$ into cost formulas described in Table 1.

Algorithm 1: Estimate cost for a physical plan *doPlanCostEstimation*

Input: PhysicalPlan P, DB db, Environment env
Output: Estimated Startup_cost C_s, Estimated Total_cost C_t

Node n = P.root
if n.isLeaf() **then**
  ParamTree T = getParamTree(n.type)
  rParams rp = T.obtainRParams(db,env,$n.info$)
  C_s,C_t = fillCostFormula(rp,$n.statistics$,0,0,0,0)
else
  C_s^L, C_t^L = doPlanCostEstimation($P.left$,db,env)
  if $n.right$ not NULL **then**
    C_s^R, C_t^R = doPlanCostEstimation($P.right$,db,env)
  else
    C_s^R, C_t^R = 0,0
  end
  rParams rp = T.obtainRParams(db,env,$n.info$)
  C_s,C_t = fillCostFormula(rp,$n.statistics$,C_s^L,C_t^L,C_s^R,C_t^R)
end
return C_s,C_t

$fillCostFormula$ also takes the startup cost and total cost of subtrees (Eg.C_s^L and C_t^L denote startup cost and total cost of left subtree respectively, and C_s^R and C_t^R denote startup cost and total cost of right subtree respectively) as input to output the estimated startup and total cost of this plan. We take the total cost estimation as the final prediction of the physical plan.

In summary, we propose a new design principle for improving formula-based cost model, reducing the problem of query cost estimation to learning a mapping function $G : C \xrightarrow{\text{op}} R$. Compared to existing state-of-the-art approaches, which train an end-to-end model from scratch to obtain the mapping from vectorized query representation to estimated cost, our new learning scheme has several advantages:

Efficient: Instead of training from scratch, our model leverages the intelligence of domain experts in the cost model. As shown in our experiments, our method requires significantly fewer annotation samples, and is more efficient in both training and inference.

Explainable: Our method inherits the interpretability of the cost model, which allows DBAs to easily understand the elemental factors involved in cost estimation. Additionally, we use decision trees to learn the mapping function G, which is well-known for its good interpretability.

Transferable: Since our learning scheme is lightweight and requires fewer annotation samples, our approach, combined with online tuning, can be conveniently transferred to diversified hardware/software environments and dynamic workloads.

3 OFFLINE TRAINING STAGE

In offline training stage, we pre-train an initial ParamTree for static C-params. We collect benchmark data from various cloud instances and adopt the customized decision tree algorithm.

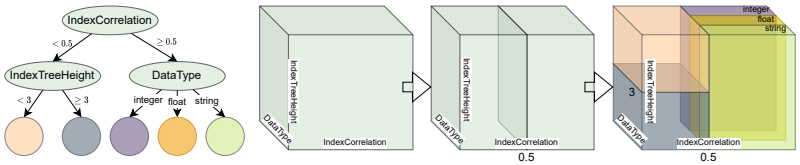


Fig. 2. Illustration of hypercube

3.1 Formal Definition of the Tree

As illustrated in Figure 2, the process of splitting a decision tree involves continuously dividing the space of C-params into hypercubes. Therefore, we present our tree construction from the perspective of spatial partitioning. We start by formalizing the decision tree construction process. Let $\mathbf{C} = c_1, ..., c_n$ denote all possible C-params. To simplify the problem, we normalize all C-params to the range $[0, 1]$. For a subset $\bar{C}$ of $\mathbf{C}$ ($\bar{C} \subseteq \mathbf{C}$), we define its hypercube H as $(\bar{C}, \mathbf{X})$, where $\mathbf{X}$ maintains the upper and lower bounds of C-params in $\bar{C}$. For hypercube H, the upper and lower bounds of each dimension are defined as follows:

$$[H[c_i].l, H[c_i].u] = \begin{cases} [0, 1], & \text{if } x \notin \bar{C} \\ [l_i, u_i], & \text{if } x \in \bar{C}, \exists [l_i, u_i] \in \mathbf{X} \end{cases} \tag{2}$$

We construct a decision tree for each physical operator op, with internal nodes and leaf nodes playing different roles. Each internal node v_i is represented as $v_i = [H_i, c_j, [l_1, u_1], ..., [l_m, u_m]]$. Here, H_i is a hypercube inherited from v_i's parent node, c_j is the next C-param used for tree split, and $[l_j, u_j]$ denotes the partitioning result. For the kth child node of v_i, if it is an internal node, its hypercube is generated as follows:

$$H_k.\bar{C} = H_i.\bar{C} \cup \{c_j\}, H_k.\mathbf{X} = H_i.\mathbf{X} \cup \{[l_k, u_k]\} \tag{3}$$

For root node, its hypercube is defined as $H = (\emptyset, \emptyset)$.

On the other hand, leaf node v_l is responsible for maintaining a pair $(H_l, \mathcal{F})$, where H_l is generated in the same way as the internal node and $\mathcal{F}$ represents a cost estimation formula for the corresponding physical operator op. Most $\mathcal{F}$ formulas follow the same linear expression as Equation 1, where r_i refers to the R-param, which is the main target of this paper. To estimate the R-params of $\mathcal{F}$ within the sub-space defined by H_l, we intentionally generate queries as samples to train a regression model.

3.2 Inital ParamTree Construction

In the offline stage, we start with an empty root node and recursively construct the internal nodes for the static C-params. We collect training samples from our cloud servers, where we establish data instances with diverse hardware and software environments to provide various services for customers. Before starting online services, we run workloads (e.g., Job-benchmark) to test their performance. The testing results are collected as our training dataset, denoted as $\mathcal{D} = \langle \mathbf{C}_i, \mathbf{f}_i(op), o_i \rangle$. Here, $\mathbf{C}_i$ represents the values of static C-params when performing the test, $\mathbf{f}_i(op)$ denotes the vectorized representations of the involved cost estimation functions (Table 1), and o_i denotes the real processing cost. It's important to note that o_i serves as our training label, and the instances in $\mathcal{D}$ are collected directly from historical query logs in our cloud database platform. The constructed subtree functions as a pre-trained model and can adapt to the target environment of any deployed database with ease.

Recall that our goal in this paper is to learn a mapping function from C-params to R-params, i.e., $G : C \xrightarrow{\text{op}} R$, to capture their dependency for a physical operator op. We will construct a decision tree for each operator op. Conventionally, decision tree construction algorithms such as C4.5 [30]

and CART [6] group data points with similar labels into the same leaf node for classification or regression tasks. For example, CART regression tree [6] uses metrics like mean squared error (MSE) and mean absolute error (MAE) to measure the purity of the data partitioning. However, these algorithms all need to know explicit R-params as labels in advance, which is not available in our case, so we cannot directly use these traditional decision tree algorithms to construct the mapping of $G : C \xrightarrow{\text{op}} R$. Samples in our dataset $\mathcal{D} = \langle \mathbf{C}_i, \mathbf{f}_i(op), o_i \rangle$ reflect the correlation as $C \rightarrow O$. R-params play like hidden parameters, working as $C \rightarrow R \rightarrow O$. So we need to find a method to establish the mapping function $C \rightarrow R$, without any explicit label for R-params, only samples reflecting $C \rightarrow O$.

To resolve this issue, we adopt splitting criteria proposed by [42] that relies on no explicit label information. In our case, we use C-params as splitting attributes and in every leaf node, we adopt least square method to compute the R-params through samples lying in the leaf node. Suppose a leaf node contains a dataset $\bar{\mathcal{D}}$ with n samples, we want to obtain R-params $\mathbf{r} = [r_1, r_2, \dots]$ that can minimize the query cost estimation error $\Psi_i\left(o_i, \mathbf{f}_i(op), \mathbf{r}\right) = (o_i - \mathbf{f}_i(op)^{\mathrm{T}}\mathbf{r})^2$, i.e., the optimized R-params can be obtained via

$$\hat{\mathbf{r}} = \underset{\mathbf{r}}{\operatorname{argmin}} \sum_{i=1}^{n} \Psi_i\left(o_i, \mathbf{f}_i(op), \mathbf{r}\right) \tag{4}$$

Initially, the decision tree consists of only one leaf node. However, varying C-params can result in distinct regression functions. Therefore, we require a method to determine which C-param will have a significant impact on the regression functions. By dividing the dataset based on this C-param, we can assign specific regression functions to each sub-dataset. Parameter instability test [42] helps group data with potentially similar R-params which achieves the goal without any explicit label for R-params. Next, we will explain the principle of the parameter instability test.

To obtain the optimal $\hat{\mathbf{r}}$ defined in Equation 4, the first order of Ψ should be 0:

$$\sum_{i=1}^{n} \frac{\partial \Psi\left(o_i, \mathbf{f}_i(op), \mathbf{r}\right)}{\partial \mathbf{r}} = \sum_{i=1}^{n} \psi\left(o_i, \mathbf{f}_i(op), \mathbf{r}\right) = 0 \tag{5}$$

If $\mathbf{r}$ does not change significantly, then ψ will vibrate near the origin point. Otherwise, it may indicate that $\mathbf{r}$ has a large deviation, in which case we are encouraged to split the node. We quantify the extent of vibration of ψ as

$$W(t, \mathbf{r}) = n^{-1/2} \sum_{i=1}^{\lfloor nt \rfloor} \psi_{\sigma(c^i)}\left(o_i, \mathbf{f}_i(op), \mathbf{r}\right), \quad t \in [0, 1] \tag{6}$$

Equation 6 represents the cumulative sum of the equation 5 sorted according to the value of C-param c, where $\sigma(c^i)$ is the ordering permutation of the observations of $\mathbf{c} = [c^1, c^2, \dots, c^n]$.

Then multiply covariance matrix of $\psi\left(o_i, \mathbf{f}_i(op), \mathbf{r}\right)$ to $W(t, \mathbf{r})$ to scale the result, which is

$$\hat{J} = n^{-1} \sum_{i=1}^{n} \psi\left(o_i, \mathbf{f}_i(op), \mathbf{r}\right) \psi\left(o_i, \mathbf{f}_i(op), \mathbf{r}\right)^{\top} \tag{7}$$

Then we obtain empirical fluctuation process $efp(t)$:

$$efp(t) = \hat{J}^{-1/2} W_n(t, \mathbf{r}) \tag{8}$$

We formulate the null hypothesis as R-params remain stable as c varies, while the alternative hypothesis posits that R-params are subject to change with alterations in c. Under the null hypothesis, Donsker's invariance principle[4] is satisfied. At the interval of $t \in [0, 1]$, $efp(t)$ converges to the k-dimensional Brownian bridge[29] W^0. Under the alternative, the value of $efp(t)$ will gradually approach the peak in the process.

For numeric and categorical C-params, we propose two instability tests respectively:

Numeric C-param We adopt the supLM test[3] for a numeric C-param c:

$$\sup_{t\in\Pi} LM(t) = \sup_{t\in\Pi} \frac{\|efp(t)\|_2^2}{t(1-t)}. \tag{9}$$

Cateogrical C-param To test categorical variables, the following chi-square test is used, where K is the number of categories, $\Delta_{I_k} efp(t)$ is the cumulative sum of $efp(t)$ on each category. Then the L2 norm of the weighted $\Delta_{I_k} efp(t)$ is subjected to a $dim(\mathbf{r}) \cdot (K-1)$ chi-square test.

$$\lambda_{\chi^2}(W) = \sum_{k=1}^{K} \frac{|I_k|^{-1}}{n} \left\|\Delta_{I_k} efp(t)\right\|_2^2 \tag{10}$$

The two instability tests described above aim to estimate the significance of C-params in determining the values of R-params. A higher instability test result suggests that the values of R-params remain stable even as the values of C-params change, while a lower result indicates that changes in C-params may cause the values of R-params to fluctuate significantly. To identify the next split attribute, we select the C-param with the lowest stability result.

We employ a multi-way split strategy, because previous research has suggested that this approach can generate smaller, more accurate, and easier-to-understand decision trees [10, 28]. We follow a recursive approach to divide intervals, using the following steps:

Step 1: Assuming the observations of c are sorted as $[v_1, v_2, \ldots, v_n]$ (For categorical variables, random order can be used). Traverse all of them to find a split point that minimizes the object function $\Psi(o, \mathbf{f}(op), \mathbf{r})$. Assuming the best-split point is v_4, then the value of c is divided into two intervals, $[v_1, v_2, v_3, v_4]$ and $[v_5, \ldots, v_n]$.

Step 2: For each interval, apply the parameter instability test to the C-param c. If the instability result is below the threshold α, it suggests that C-params have an influence on the values of R-params, and we continue splitting this range and repeat Step 1. If the instability result is above α, we consider this interval fixed and will no longer split it.

Step 3: Once no more splitting is performed on all intervals, we divide the range of values for c into several intervals, with each interval serving as a new node in the decision tree.

After expanding the decision tree and creating new leaf nodes, we train a linear regression model using the optimization objective function (Equation 4) to obtain R-params. To prevent severe overfitting, we adopt a termination strategy inspired by existing decision tree construction algorithms for static C-params. We terminate the tree construction if one of the following conditions is met:

- All p values obtained by C-params' parameter instability tests are greater than the threshold α.
- The number of samples at a leaf node falls below β.
- All samples at the leaf node share the same C-params values.

4 ONLINE REFINEMENT STAGE

When the initial ParamTree fails to provide a satisfactory cost estimation result, we resort to the online refinement process to gradually expand the tree. However, there are two significant differences between the static and dynamic C-params. Firstly, there are hundreds of dynamic C-params, making conventional decision tree algorithms too expensive. Secondly, dynamic C-params have a strong correlation with the query and data distributions. This means they are highly sensitive to the underlying data and target workload. Consequently, the refinement process operates as an ongoing process for each online DBMS.

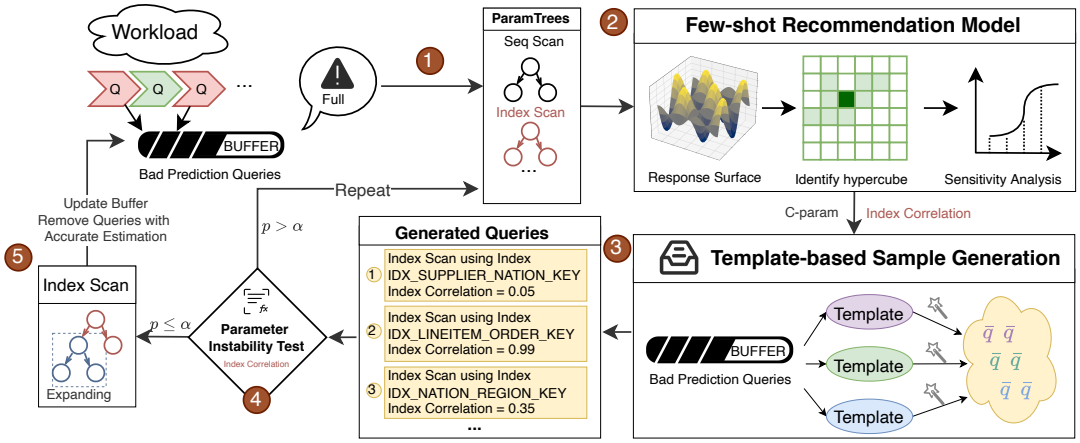


Fig. 3. Process of online tree expansion

4.1 Online Tree Expansion

The online tree expansion process consists of two major modules: the few-shot recommendation model (Section 4.2) and the template-based sampling approach (Section 4.3). As shown in Figure 3, our approach maintains a buffer for queries that receive imprecise estimated costs from the ParamTree. If the error rate of our cost estimation for query q_i is greater than a pre-defined threshold ϵ, we mark it as imprecise. We set an upper bound λ for the size of the buffer, and when it is full, the tree expansion process is triggered. The steps involved in the process are:

(1) We group queries in the buffer based on their operators and calculate the error rates for each operator. We use the R-params information of a leaf node for cost estimation of each operator. We take the average estimation error rate of each leaf node for operators in the buffer and select the leaf node with the highest error rate for expansion. In the following steps, we focus on the expansion of this individual leaf node.
(2) The few-shot recommendation model is employed to recommend the next most dominant C-param for the node split.
(3) The template-based sample generation approach is invoked to collect enough samples for expansion.
(4) We perform the parameter instability test (Section 3.2) to verify the necessity of node split. If the C-param does not have a significant impact on R-params, we go back to step 2.
(5) We split the node with the C-param based on generated samples and retrain our regression functions for R-params in the new leaf nodes. We re-evaluate the costs of queries in the buffer. For those that receive precise estimations, we remove them from the buffer.

In what follows, we will elaborate our few-shot recommendation model and template-based sample generation approach.

4.2 Few-shot Recommendation Model

In our paper, we include hundreds of candidate dynamic C-params, such as query and data related parameters, in order to model the hidden parameters of the cost model as extensively as possible. However, for a given query, only a few of these parameters will actually have a significant impact on query performance. Many database tuning researches [2, 7, 9, 14] also have demonstrated that some C-params are consistently more important than others. Thus, when expanding the decision

tree for a target workload, we need to select the next most dominant C-param. To achieve this, we build a model $Y = \widetilde{G}(\mathbf{C})$ to measure the correlations between C-params and R-params, which are the performance metrics. This enables us to identify the C-params that are most sensitive to R-params.

We generate samples and train the model in an offline way. We use PGBench[3] as our benchmark for generating samples to train the model. Suppose we want to split leaf node $v_l = (H_l, \mathcal{F})$ of existing ParamTree. If $Y = \widetilde{G}(\mathbf{C})$ has been learned, we can rank the importance of each C-params as follows:

$$c_{opt} = \underset{c \in \mathbf{C} - H_l.\bar{C}}{\operatorname{argmax}} \int_{H_l} \frac{\partial \widetilde{G}(\mathbf{C})}{\partial c} \tag{11}$$

Namely, we pick the C-param with the largest accumulative partial differential in hypercube H_l as our split dimension.

Unfortunately, training a function $\widetilde{G}(\mathbf{C})$ to measure correlations between C-params and R-params can be difficult as explicit samples revealing such correlations are not available. However, we assume that the formula-based cost model can provide a good estimation, if all R-params in Table 1 are set correctly. The perturbation of C-params can lead to imprecise estimations as the original settings of R-params may not be valid. Therefore, we can measure the importance of C-params by comparing the real processing cost to the estimated cost, and examining the resulting divergence.

To measure the importance of C-params, we simulate Y using $\bar{Y} = \frac{|t_{act} - t_{est}|}{t_{act}}$, where t_{act} and t_{est} are the costs of actual processing and estimation, respectively. Our goal is to learn a $\widetilde{G}(\mathbf{C})$ that estimates $\bar{Y}$. However, this is a very challenging problem because of the high dimensionality of the hypercube H_l, which makes it difficult to obtain enough samples.

To address the sparse space problem, we adopt a few-shot learning approach called Response Surface Method (RSM) [18]. In RSM, a response surface is constructed using a small set of sample points, and this surface is used to approximate the cost ratio within the hypercube. By doing so, we can reduce the number of samples required for estimating the correlations between C-params and $\bar{Y}$. In the following, we will describe how to construct a response surface and how to pick the dominant C-param in hypercube H_l in a more efficient way.

4.2.1 Surface Construction. RSM approximates a complex data distribution with a few simple functions (e.g., polynomial functions).

It was shown that RSM applies linear combinations of RBF (Radial Basis Functions) to produce good fits for both continuous and discrete response functions [25], and is suitable for multivariate high-order nonlinear problems [13]. Our RSM follows the same RBF strategy and simulates the $\bar{Y}$ as:

$$\widetilde{G}(\mathbf{C}) = \bar{Y} = \sum_{i=1}^{p} \beta_i g_i(\mathbf{C}) + \sum_{i=1}^{m} \lambda_i \phi\left(\left\|\mathbf{C} - \mathbf{C}^{(i)}\right\|\right) \tag{12}$$

where $g(\boldsymbol{C}) = \{g_1(\boldsymbol{C}), g_2(\boldsymbol{C}), ..., g_p(\boldsymbol{C})\}$ are low-order polynomial regression functions. For example, if we focus on two-orders of correlations, we have:

$$g(\boldsymbol{C}) = \{1, c_1, ..., c_n, c_1c_2, ..., c_{n-1}c_n, c_1^2, ..., c_n^2\} \tag{13}$$

$\beta = \{\beta_1, \beta_2, ..., \beta_p\}$ are regression coefficients of $g(\boldsymbol{C})$ ($p = |g(C)|$). m denotes the number of training samples. Samples refer to queries with different settings of C-params, and we submit them to the DBMS to collect real processing cost (The strategy of generating sample queries will be discussed in Secion 4.3). $\left\|\boldsymbol{C} - \boldsymbol{C}^{(i)}\right\|$ is the Euclidean norm, ϕ is the basis function, and λ_i is the

[3]https://www.PostgreSQL.org/docs/current/pgbench.html

coefficient of the ith basis function. In this paper, we apply Gaussian function as the Kernel function of RBF:

$$\phi(\left\|C - C^{(i)}\right\|) = exp(\left\|C - C^{(i)}\right\|^2 / \rho_i^2) \tag{14}$$

Equation 12 can be trained using the classic least square method. Initially, we apply the Latin Hypercube Sampling (LHS)[24] approach to generate m samples, which is a commonly used sampling method that can generate sample points with randomness and uniformity within a given parameter range. Every dimension will be divided into m equally probable intervals. The goal is to ensure that every interval within each dimension is covered by at least one sample, providing a representative and evenly distributed set of samples across the entire parameter space. Given the benchmark workload W, we ask the DBMS to process queries in W with different C-params from samples. The collected real processing costs and estimated ones are used as training samples. LHS allows us to achieve a reasonable approximation of $\bar{Y}$ with a few samples.

4.2.2 Sensitivity Analysis. Once the RSM model is established, we utilize the Sobol' method [34, 35] to estimate Equation 11 and rank the importance of C-params during the online refinement stage. The Sobol' method decomposes the total variance of $\bar{Y}$ to determine the contribution of each input variable to the change of $\bar{Y}$.

For a leaf node $v_l = (H_l, \mathcal{F})$ of ParamTree, we need to collect multiple sobol sequence samples, which should be multiples of $n + 2$ (n is the number of remaining C-params). The RSM can output estimated $\bar{Y}$ for these samples, which can be used to compute the sobol' index and rank the importance of C-params. By combining the RSM and Sobol' method, we can avoid the need to collect samples every time a global sensitivity analysis is performed. Instead, only a small amount of computation is required to obtain the results.

4.3 Template-Based Sample Generation

To generate the required random samples for both surface construction and online tree expansion, statistics for some running queries need to be collected from the DBMS. Specifically, after the processing of a query q, statistics for each operator of the query can be collected from the execution engine, resulting in a set of triples $\mathcal{D} = \langle \mathbf{C}_i, \mathbf{f}_i(op), o_i \rangle$, as mentioned in Section 3.2.

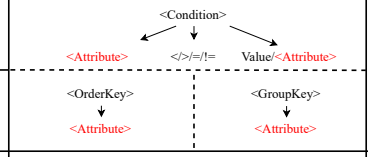
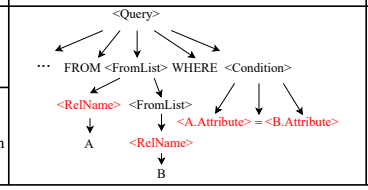
Running queries randomly may not always produce suitable samples for our training process. To address this issue, we propose a template-based approach for sample generation and provide two sampling APIs: $\mathcal{F}(S_k, op_i, H_l, c_j, m)$ and $\mathcal{F}(op_i, H_l, m)$.

The first API is used for splitting the ParamTree in the online stage, while the second one is a specialized version of the first API designed for training the C-param recommendation model (Section 4.2). We denote the query set where our ParamTree cannot provide accurate estimations as S_k. In our algorithm, we select the leaf node v_l with the highest error rate for expansion. Here, op_i is the operator of the ParamTree to which v_l belongs, and H_l is the hypercube that v_l maintains. We use c_j as the C-param to split the ParamTree, which also serves as our sampling criterion. m is the number of required samples. Using this API, we modify queries in S_k to generate new queries whose physical execution plan contains op_i and the samples inside the hypercube H_l.

The second API, $\mathcal{F}(op_i, H_l, m)$, retrieves m random samples from the hypercube H_l. This API can be represented as a series of invocations of the first API: $\mathcal{F}(S_k = pgbench, op_i, H_l, c_j, \frac{m}{|c|})$ for all valid c_j. The following discussion will focus on the implementation of the first API.

Our sampling process aims to enhance the ParamTree by minimizing estimation errors for queries that are similar to those in S_k. To achieve this, we adopt a twofold sampling strategy:

C-params related to Logical Plan

C-param	AST	Hidden Parameter	Logical Op
Name: DataType **Description:** The datatype of the attribute.		<Attribute>	Selection Join Sort Aggregation
Name: ColumnOffset **Description:** The offset of the column to the head of the corresponding tuple.			
Name: TuplesPerBlock **Description:** The average number of tuples in a block.		<RelName>	Scan
Name: LeftSinAvg/RightSinAvg **Description:** Left/Right operator's input tuples' average width.		<RelName> <Attribute>	Join
Name: InnerUnique **Description:** Whether there are no more than one inner row could match any given outer row.			

C-params related to Physical Plan

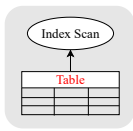
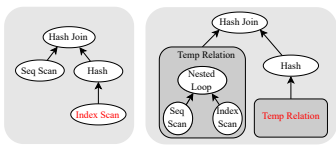
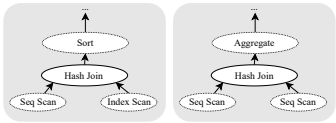
C-param	Physical Plan	Hidden Parameter	Physical Op
Name: IndexCorrelation **Description:** The correlation between the physical and logical storage of table.		Table/ Predicates	Index Scan
Name: IndexTreeHeight **Description:** The height of the index tree.			
Name: BatchesNum **Description:** The number of batches needed based on current *hash_mem*.		Inner Relation (#tuple/ #unique tuples)	Hash Join
Name: InnerBucketSize **Description:** The average number of tuples in a hash bucket.			
Name: Left/RightOperator **Description:** The type of left/right child operator.		Add Hint/ Select Template	Hash Join Merge Join Nested Loop ...
Name: ParentOperator **Description:** The type of parent operator.			

Fig. 4. C-params and their plan templates

Similarity Samples are generated based on the query set S_k. This enables the cost model to adjust its estimations by incorporating updated values of R-params, which in turn minimizes estimation errors for queries in S_k.

Versatility To ensure that the cost model is robust and can generate accurate estimations for unseen queries, samples are drawn uniformly from the value domain of c_j. Additionally, we aim to test as many variants of queries in S_k as possible.

4.3.1 Basic Idea. We have designed a modified stratified sampling strategy to generate samples that are representative of the distribution of C-param values. For a continuous C-param c_j, we use discretization to divide its domain into k equally sized buckets. If c_j is discrete, we uniformly hash its values into k buckets.

We maintain a histogram H for each C-param, which records the distribution of its values for historical queries. Specifically, $H(i)$ $(0 < i < k)$ returns the number of historical queries that have the c_j set to a value in the ith bucket. Therefore, H represents the popularity of each value of the C-param c_j.

Then, the probability $p(i)$ of retrieving a sample from bucket i is set as:

$$p(i) = \frac{max(\epsilon, H(i))}{\sum_{x=0}^{k-1} H(x)} \tag{15}$$

ϵ denotes the minimal probability assigned to each bucket. Let m be the required number of samples for training. We need to retrieve $mp(i)$ samples for bucket i, and within each bucket, the samples are generated uniformly at random.

4.3.2 Template-Based Approach. The two properties (Similarity and Versatility) require our sampling approach to generate samples based on S_k. However, this is a non-trivial task since, for specific queries and datasets, values of a C-param cannot be randomly selected. Some C-params are correlated with query structures. Therefore, we classify the dynamic C-params into three categories: C_0, C_1, and C_2.

The C-params in C_0 normally represent DBMS configurations, such as the size of buffer for temp tables. C-params from C_0 are irrelevant to query structures and datasets. To generate a new sample for $c_j \in C_0$, we simply select a random query from S_q that involves op_i during processing. We then set the C-param to a random value of bucket i with a probability of $p(i)$. The query is submitted for processing, and we collect the statistics for op_i as a triple $(op_i, \mathbf{f}(op_i), o)$.

On the other hand, C_1 and C_2 refer to C-params that are correlated with a query's logical plan and physical plan, respectively (e.g., column type, data type, and index tree height). For a C-param c_j from C_1 or C_2, we cannot arbitrarily set a value for c_j because its value is bound to the query structure. For example, Figure 4 shows some C-params that are late materialized during the logical plan generation (C_1 type) and physical plan generation (C_2 type) processes, respectively. For C-params such as DataType and ColumnOffset, their values are set when we add columns into query predicates. These values depend on the query structure (where clause) and targeted tables/columns.

Therefore, we can only obtain statistics about possible values of C-params in C_1 when a logical plan is available. To enable more value tests for those C-params, we can replace the target column with other columns from the same table. It should be noted that the available values of C-params are defined by the database schema.

On the other hand, for C-params in C_2 such as IndexCorrelation and BathesNum, their values can only be determined when physical plans are generated. The values of C-params in C_2 are materialized based on run-time configurations, such as whether indexes are available and whether the optimizer chooses to use them when generating the physical plan. Therefore, the values of these C-params cannot be determined until all physical operators are definite.

For C-params of C_1 and C_2 types, we build a template table that indicates the sub-trees of the involved ASTs (Abstract Syntax Trees) and physical plan trees. We also list the hidden parameters that can be used to change the values of the corresponding C-params. In this way, for a query q_j from S_q, we can apply a graph matching algorithm to find the same sub-tree in the AST and physical plan trees of queries. By adjusting the values of the hidden attributes, we can obtain a new query $\bar{q_j}$ that shares a similar query structure with q_j, but with a different configuration for the C-params.

We can simplify the sampling process by using an inverted index. For each operator op_i, there are multiple templates available for a C-param c_j, denoted as $t_0, t_1, ..., t_n$. After matching a template against a query q from S_q, we can obtain statistics on the possible values of c_j for that matching.

Algorithm 2: Greedy algorithm for sampling queries

Input: Operator op, C-param c, bucket probability p, Query set S_q, Template set T
Output: Generated queries Q

1 $Q = \{\}$; ▷ Store selected queries
2 **for** $B_i \leftarrow B_1$ to B_t **do**
3 Select satisfied queries and templates $S_i =< q, t >$ for c from query set S_q and template set T
4 **foreach** $(q, t) \in S_i$ **do**
5 $L = generateQueries(q, t, B_i)$;
6 Sort L by estimated processing cost o;
7 $Q = Q \cup$ select top $\lceil \frac{mp(i)}{\|S_i\|} \rceil$ queries from L
8 **end**
9 **end**
10 return Q

To build the inverted index, we use a composite key $[c_i, B_j]$, indicating the corresponding C-param and the involved bucket. The value of the key is a list of triples: $r = [t_i, q_j, o_{ij}]$, where template t_i is matched against query q_j, and the estimated processing cost of q_j with this matching is o_{ij}.

If we retrieve m samples for a C-param c_j, we are expected to pick $mp(i)$ samples for the ith bucket. Given our inverted index, we can retrieve a list of triples L, and now our target is to ensure diversity while minimizing costs. More formally, the sampling process will return a list L' with size $mp(i)$. L' shares the same set of triples with L, but allows a triple sampled multiple times. L' should satisfy:

$$\arg\min_{L'} \sum_{\forall r \in L'} r.o \tag{16}$$

and

$$\arg\max_{L'} \| \bigcup_{\forall r \in L'} r.t \| \tag{17}$$

where $\| * \|$ returns the number of unique elements in a set. Algorithm 2 illustrates our sampling process. The algorithm starts by creating an empty set Q to store selected queries. Next, it traverses all the buckets to add queries into Q(line 2-9). For every bucket B_i, it firstly selects a satisfied combination of queries from S_q and template t. Then, for each combination (q, t), generate a list of queries L with C-param values in B_i by modifying query q through template t(line 5). Sort queries in L according to their estimated cost o and then select the $\lceil \frac{mp(i)}{\|S_i\|} \rceil$ queries with the smallest cost(line 6-7), where $\|S\|$ denote the number of different (q, t) combinations. So, we get $\sum_{i=1}^{t} \lceil \frac{mp(i)}{\|S_i\|} \rceil \times \|S_i\| = m$ queries after running Algorithm 2.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate our model from four aspects: accuracy, generalization, and dynamic, training overhead.

5.1 Experimental Setup

Environment and Datasets. By default, our experiments are conducted on our in-house server, equipped with 4 Xeon CPUs, 32 GB dram, and 894 GB SSD. The default operating system and DBMS are Ubuntu 18.04.6 LTS and PostgreSQL 13.3, respectively. However, to show the generalization of

our model, we also test it on 20 cloud instances with different hardware and DBMS configurations. We adopt widely-used benchmark datasets such as IMDB, TPC-H and TPC-DS in our experiments.

Note that we disable the parallel execution to improve the stability of results.

Comparison Approaches. We denote our method as ParamTree and compare it with both formula-based and learning-based approaches:

(1) *PostgreSQL Cost Model (Scaled CM)*: This is PostgreSQL's default cost model. It provides estimated cost unit that can be scaled to the real processing time with a simple linear function.
(2) *PostgreSQL Tuned Cost Model (Tuned CM):* We follow the method proposed by [39] to adjust the R-params in Table 2. With the adjusted R-params and accurate cardinality estimation, we can improve the precision of default PostgreSQL cost model.
(3) We also compare with multiple recent learning-based models, including MSCN[17], E2E[36], QueryFormer[43], and TCNN[21, 47]. These models apply deep neural networks to encode queries or physical query plans and predict query execution time in an end-to-end manner. Training these models incurs high overheads, because we need to collect query execution time as the training samples for a large volume of query plans.
(4) *Zeroshot approach:* Zeroshot[11] builds a pre-trained model with samples from many well-known databases and shows good transferability to a new database. However, its leverage of pre-trained knowledge is not fair to other competitors. In addition, it requires thorough re-training for any new hardware and software environment. Therefore, we only compare with Zeroshot in the generalization experiment.
(5) Learned tuners aim at recommending knob values automatically, which include R-params in cost model. We also compare our work with state-of-the-art learned tuners, like OutterTune[2] and LlamaTune[15].

Among these approaches, MSCN, E2E, QueryFormer, and TCNN combine cardinality estimation with cost prediction to create an end-to-end model, while Zeroshot only focuses on cost estimation. For the latter ones, we adopt DeepDB [12] as the cardinality estimation model. DeepDB is a data-driven technique and is easy to train.

Evaluation Metrics. We use Q-error as our main evaluation metric. We also evaluate the training overhead of our approach. The Q-error is computed as:

$$\text{Q-error}(q) = \frac{1}{n}\sum_{i=1}^{n}\frac{\max\left(actual(q), predicted(q)\right)}{\min\left(actual(q), predicted(q)\right)}$$

$actual(q)$ is the actual execution time of query q, while $predicted(q)$ is the algorithm's estimated execution time of query q. A smaller Q-error value indicates better performance and the **optimality** occurs when Q-error equals 1.

5.2 Performance of Cost Estimation

In this set of experiments, we evaluate the overall performance of all approaches on IMDB. For all the learning-based comparison approaches, we construct a training set with 5,000 queries from IMDB synthetic workload, and estimate the query execution time for two workloads: job-light and scale. To demonstrate the superiority of ParamTree in terms of friendliness to training samples, we train ParamTree with only 2,000 samples. ParamTree+DeepDB implies that we adopt DeepDB for cardinality estimation, whereas ParamTree+Exact is the oracle scenario in which we assume the exact cardinality is available.

Prediction Accuracy. As shown in Table 4, ParamTree exhibits extremely accurate prediction performance when using exact cardinality, with Q-error below 1.11 in job-light and 1.15 in scale

IMDB(Job-light)	median	90th	95th	99th	mean
Tuned CM+DeepDB	2.55	6.69	9.65	16.16	3.72
Tuned CM+Exact	1.97	3.45	4.07	9.76	2.36
MSCN	3.40	20.71	37.91	75.77	9.36
E2E	2.27	10.53	17.75	87.54	7.16
QueryFormer	1.57	9.33	18.59	40.00	4.41
TCNN	1.90	13.37	26.25	61.24	6.65
ParamTree+DeepDB	1.58	5.07	7.35	14.37	2.59
ParamTree+Exact	**1.11**	**1.58**	**1.78**	**4.79**	**1.32**
IMDB(Scale)	median	90th	95th	99th	mean
Tuned CM+DeepDB	2.03	7.53	12.84	54.73	4.72
Tuned CM+Exact	1.75	3.65	3.73	6.68	2.14
MSCN	2.33	12.28	25.46	72.24	6.19
E2E	1.91	9.15	13.78	21.19	3.66
QueryFormer	1.25	5.08	9.37	44.36	3.26
TCNN	1.41	6.73	17.9	85.41	4.84
ParamTree+DeepDB	1.47	5.09	8.98	27.14	2.97
ParamTree+Exact	**1.15**	**1.90**	**2.67**	**5.31**	**1.41**

Table 4. Comparison of prediction accuracy

Error(ms)	SeqScan	IndexScan	IndexOnly	Sort
Tuned CM	139.76	690.15	1743.34	613.76
ParamTree	97.67	108.29	164.27	14.26
Error(ms)	MergeJoin	NestedLoop	HashJoin	Aggregate
Tuned CM	431.26	19.03	1022.09	1094.54
ParamTree	70.73	8.91	77.66	26.30

Table 5. Performance of different operators.

for 50% of queries. If DeepDB is adopted for cardinality estimation, ParamTree can still outperform its competitors. Traditional cost models have a relatively high median Q-error but perform well in predicting tail queries, making them stand out in terms of average performance. On the contrary, deep learning methods show poor performance when predicting tail queries. This is because traditional cost models rely on formula-based estimation, allowing them to generate robust predictions for a wide range of queries, less affected by shifts in the distribution of training and testing queries, while deep learning methods are highly sensitive to the change of query distribution. ParamTree inherits the merit of the traditional cost model so that it also shows good transferability for new data and queries.

Performance of individual operators. In Table 5, we give a more granular view of the performance of some key operators when predicting queries from job-light workload. Because many operators (e.g., Index Scan and Index Only Scan) may loop several times in a physical plan, we add up their costs in all loops. We use Median Absolute Error(MedianAE) to measure the accuracy. From the result, we can find that there are significant improvements in ParamTree compared to Tuned CM for the Aggregate and Sort operators.

Compare with leading learned tuners. We also compare our approach with learned tuners, and the results are shown in Table 6. In this experiment, we use default cardinality estimation results given by PostgreSQL. Our results show these learned tuners cannot provide a better result than Tuned CM and our approach. Learned tuners aim at increasing throughput or decreasing

IMDB(Job-light)	median	90th	95th	99th	mean
Sclaed CM	3.91	11.07	32.11	44.05	6.81
Tuned CM	2.07	9.42	11.07	21.65	3.83
OutterTune	10.08	12.22	12.77	**17.18**	8.06
LlamaTune(SMAC)	3.27	7.48	35.77	134.9	9.76
LlamaTune(DDPG)	3.28	7.76	43.23	152.41	10.94
ParamTree	**1.47**	**7.28**	**9.49**	35.23	**3.63**

Table 6. Performance on different learned tuners.

IMDB(Job-light)	median	90th	95th	99th	mean
MySQL	2.16	16.11	27.47	79.62	7.09
TCNN	1.97	7.92	10.07	29.58	3.74
QueryFormer	2.03	8.44	**9.78**	**15.8**	3.72
ParamTree	**1.81**	**7.67**	12.73	24.45	**3.66**

Table 7. Performance on MySQL.

latency, instead of improving cost model's prediction accuracy. So they pay more attention to parameters that control resources, instead of hyperparameters in cost model. Moreover, we find that even using default cardinality estimation, ParamTree still provides accurate predictions against learning-based cost models.

Performance on MySQL. ParamTree can be widely applied to any database that uses a formula-based cost model. We demonstrate the effectiveness of our method when applied to MySQL. As shown in Table 7, we use the default cardinality given by MySQL. ParamTree's predictive performance can still rival that of learning-based cost models.

5.3 Generalization Performance

Generality is a desirable feature for learning-based cost models. Here, we investigate two types of generalization performance and compare our ParamTree with PostgreSQL cost models as well as a pre-trained model called Zeroshot [11], which is specifically designed to support the transfer to unseen databases. The other methods[17, 21, 36, 43, 47] use embedding for table and column names which causes them to be unable to predict unseen databases.

Generalization among different machines. We select 20 cloud servers with *different hardware configurations*. For each server, we vary the DBMS configuration parameters to generate 100 database instances with *different combinations of static C-params*. For each database, we load IMDB and run the Job workload to obtain the execution results as the annotation labels. We randomly divide 20 cloud servers into training and testing sets with a split ratio of 0.8. ParamTree and Zeroshot are pre-trained with the training samples and evaluated on the test cloud servers.

For the PostgreSQL cost model, we generate two variants with different settings of R-params. One is to use the default values and map the cost unit to query execution time through a linear model. The other requests collecting several samples and calculating the appropriate R-params for that instance. We call the former Scaled Cost Model and the latter Tuned Cost Model.

Table 8 shows the evaluation results, which indicate that ParamTree outperforms all other methods on four test cloud database machines. Without collecting any samples to adjust R-params, it can even achieve better performance than Tuned CM. Zeroshot requires a large number of samples to adapt to a specific hardware environment and obtain a model and its performance is inferior when the hardware environment is changed.

Generalization among different databases. In this setting, we load multiple databases into the same DBMS and run the experiments within the same machine. As shown in Figure 5, the

ID	Methods	Q-error				
		median	90th	95th	99th	mean
#1	Scaled CM	2.76	21.30	41.10	239.56	13.02
	Tuned CM	2.69	5.96	12.41	21.85	3.59
	Zeroshot	5.69	19.00	25.51	35.62	8.99
	ParamTree	**2.26**	**5.36**	**10.69**	**18.64**	**3.09**
#2	Scaled CM	5.78	44.24	89.83	526.31	28.41
	Tuned CM	3.38	**7.22**	14.21	27.48	4.38
	Zeroshot	7.27	39.34	45.21	48.47	13.02
	ParamTree	**2.73**	8.27	**10.69**	**26.62**	**4.24**
#3	Scaled CM	2.25	20.43	38.54	251.29	13.26
	Tuned CM	2.46	6.40	13.69	25.14	3.85
	Zeroshot	7.80	28.20	32.93	46.68	11.79
	ParamTree	**1.78**	**5.04**	**9.92**	**18.54**	**2.90**
#4	Scaled CM	2.71	15.79	35.37	145.97	10.01
	Tuned CM	2.56	5.00	8.61	17.00	3.12
	Zeroshot	6.23	20.84	25.49	35.58	9.25
	ParamTree	**2.22**	**4.94**	**7.39**	**14.48**	**2.79**

Table 8. Prediction performance on four random unseen machines.

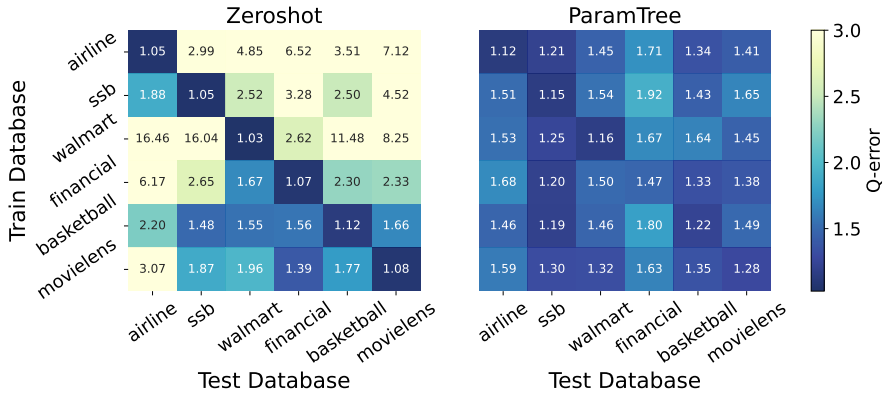


Fig. 5. Generalization performance of the model on different databases. The numbers on the chart represent the corresponding Q-error.

vertical axis represents the database used during model training, and the horizontal axis represents the database used for testing. We choose 6 publicly available databases to conduct this experiment and use the workload generated by [11] for these databases, which are Select-Project-Aggregate-Join (SPAJ) queries with conjunctive predicates on numeric and categorical columns. For Zeroshot, we use 5,000 samples from every database to train the model, and we only use 1,000 samples for ParamTree. For ParamTree, the Q-error of all models is below 1.92. Although Zeroshot can have very good predictive performance on the training dataset, the knowledge learned by the model is very limited, resulting in a sharp drop in performance when making predictions on other databases. ParamTree is built on top of the traditional cost model, and its abundant prior knowledge enables it to migrate well between different databases. Moreover, it only focuses on a small number of hyperparameters in the cost model, making it less prone to overfitting and failure to learn corresponding knowledge.

5.4 Performance on Dynamic Scenarios

In this experiment, we evaluate the performance of ParamTree when handling dynamic scenarios, including the update of query distribution, database size, and data skewness. In the online refinement stage, ParamTree can specifically improve queries with poor prediction results. Moreover, the rules of the cost model and dynamic C-params used for splitting can take data updates into account. However, it is difficult for the deep learning approaches to support the dynamic scenario because they collect samples on static databases, and cannot perceive the update of data.

Dynamic queries. Query distribution update is common in practice. In this experiment, we use the Initial ParamTree obtained from the Offline training stage as the initial model and then use IMDB's job-light as the query workload for online refinement. As shown in Figure 6, We record the average Q-error of job-light's query prediction results after each node expansion. It can be seen that the model only needs a very small amount of samples to achieve very good prediction performance. After generating 350 samples, ParamTree+Exact's mean Q-error can reach 1.26, which is better than the effect achieved by passive learning using 2000 samples in Section 5.2.

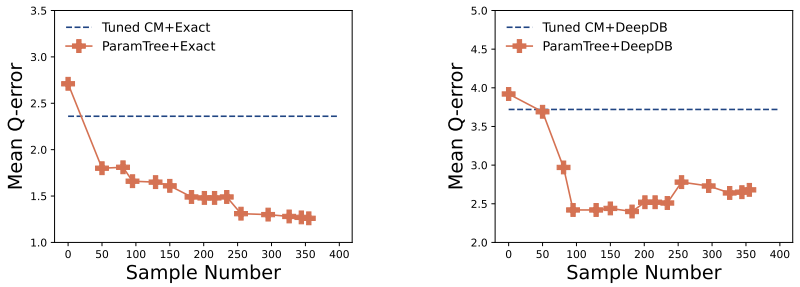
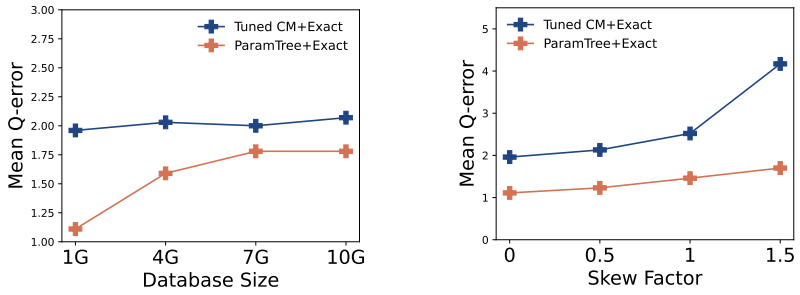


Fig. 6. Performance of ParamTree for every expansion in online refinement stage.

Generalization to Data Dynamics. We consider data dynamism from the size of database and the skewness of data distribution. We use the model trained on TPC-H (scale=1G, skew factor = 0) for this experiment. From Figure 7(a), it can be observed that as the size of the database increases, the predictive performance of the traditional cost model remains almost unchanged. The performance of ParamTree degrades when the database size increases from 1G to 7G and then slightly improves when the database size further expands to 10G. Nevertheless, it still outperforms the traditional cost model in all scenarios. Figure 7(b) shows that as data skewness increases, the performance of traditional cost model declines significantly. In contrast, ParamTree is less sensitive to data skewness. When the skew factor increases to 1.5, the traditional cost model's performance decreases by 112.8% compared to the original database, while ParamTree's performance only decreases by 53.2%. The Q-error of ParamTree is more than 2.5x smaller than Tuned CM.

5.5 Training and Inference Overhead

Training efficiency and friendliness to the number of required training samples are desirable advantages of ParamTree. As shown in Table 9, we compare the training time of methods that use physical execution plans as input. We can see that the training time of ParamTree is only about 1/5 of the fastest neural network method TCNN. We also compare the performance with varying numbers of training samples. In this experiment, we use TPC-H as the workload to generate numerous queries through different seeds. We chose a typical deep learning method with a small training overhead, TCNN, for comparison. For a fair comparison, both TCNN and ParamTree use



(a) Data Update (b) Data Skewness

Fig. 7. Performance of the model facing data dynamics.

Method	E2E	QueryFormer	TCNN	ParamTree
Time(s)	1364.41	5995.90	1248.67	272.04

Table 9. Comparison of model training time

exact cardinality for query execution time estimation. We can see from Figure 8 that ParamTree requires significantly fewer training samples to obtain satisfactory performance. To reach the same level of converged Q-error, TCNN requires 4, 000 samples to be fully trained, whereas ParamTree only consumes 1, 000 samples.

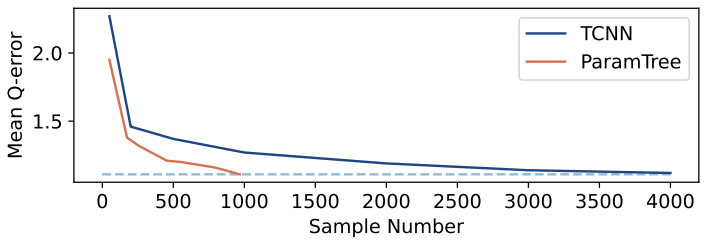


Fig. 8. Training process for TPC-H.

Since we adopt decision tree, the inference is in fact very fast. Embedding our method into a database system would not introduce significant additional costs to the existing cost estimation. It only requires a modest increase in search time for R-params within the decision tree. Furthermore, our trees have a height controlled within 10, so accessing fewer than 10 nodes is sufficient to obtain the required R-params. Additionally, the decision tree occupies minimal space and can be loaded entirely into memory.

5.6 Ablation Studies

5.6.1 Effect of Components. Firstly, we conduct experiments to demonstrate the benefits of designing three modules and reported the results in Table 10. Firstly, without conducting a parameter instability test, the decision tree would only have one root node and no branches. The results indicate that this leads to a significant decrease in accuracy. Secondly, we replace the few-shot recommendation model with a random model and the result shows a decline in effectiveness. Thirdly, to evaluate template-based sample generation module's effectiveness, we create a pool

Ablation (Median Q-error)	TPC-H	TPC-DS	JOB
ParamTree	1.08	1.18	1.18
w/o Parameter instability test	1.12	1.62	1.88
w/o Few-shot recommendation model	1.17	1.27	1.41
w/o Template-based sample generation	1.60	1.39	1.70

Table 10. Ablation study of ParamTree on diverse workloads.

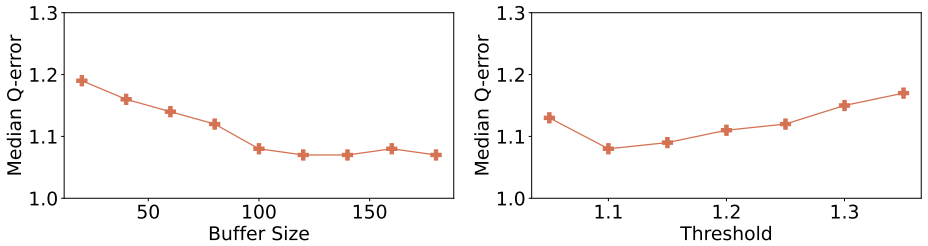


Fig. 9. Effect of hyperparameters.(TPC-H)

consisting of numerous random queries and selected queries from it during each epoch. Result shows taking random queries as training samples doesn't yield satisfactory results.

Many previous studies have utilized workload queries for both training and testing. However, this approach produces inferior results when the model is faced with queries that are significantly different from those in the training workload. For instance, TCNN is able to achieve a mean Q-error of 1.07 on TPC-H workload; however, its Q-error increases drastically to 4.79 when tested on randomly generated queries. To address the issue, we propose the template-based sampling approach, which achieves a good balance between randomness and locality. Taking workload queries as training samples, ParamTree achieves 1.11 on TPC-H, but also performs well on random queries, yielding a Q-error value of 1.63. Moreover, with template-based sampling, ParamTree can attain 1.25 on random queries.

5.6.2 Effect of hyperparameters. Finally, we explore the impact of the size of the buffered queries λ and the error threshold ϵ. These findings are presented in Figure 9. We can see that as buffer size λ increases, the performance is getting better. With a large buffer size, ParamTree is able to consider more queries in one expansion, which helps the model make good decisions. However, this kind of benefit will not continue to exist as the buffer size increases. After reaching a certain level, the performance will stabilize. The error threshold ϵ determines at what point the estimated errors of a query reach, conducting subsequent customization and improvement for that query. Result shows Q-error 1.1 is a good threshold for TPC-H. For different workloads, it should be set with different values.

6 RELATED WORK

Cost Estimation. Previous work [1, 19, 32] used machine learning algorithms to predict costs from two levels: operator-level and plan-level. The former has generalization ability, while the latter has higher accuracy. These works usually combine two types of models in order to achieve better results. With the development of deep learning, researchers turn their attention to using deep learning to estimate query costs[21, 23, 26, 36, 43, 47]. The focus of these methods is on how to capture the structural information of the physical execution plan tree well. [36, 43] also focus on how to improve the accuracy of cardinality estimation by utilizing the features of sampling.

[11] considers the generalization performance and transferability of cost estimation models, which can be extended to unseen databases, but does not consider the effect of different hardware and requires a large number of training samples. [39] aims at tuning hyperparameters in cost model, which is the most relative work with ours. However, its settings do not change according to different queries, operators, and database configurations, resulting in lower accuracy compared to ParamTree. Learned tuners[2, 15] also tune cost model's hyperparameters, but they aim at increasing throughput or decreasing latency, rather than improving cost model's accuracy. Some works [16, 33, 46] consider how to better estimate costs in the era of big data and cloud computing. [17, 37] encode queries and can accomplish multiple tasks including cost estimation, but did not consider different physical plans.[38] proposed a framework to gather training data for tasks like cost estimation which can serve as an effective tool for other methods.

Learned Database Components. As deep learning rapidly developed, researchers focus on using deep learning technology to replace some components of databases, eg. end-to-end optimizers[22, 40], cardinality and cost estimation[12, 20, 23, 36, 46], query rewrite[44, 45], join optimization[8, 41]. As models become more complex, there is a higher risk of overfitting on simple workloads. This can result in poor generalization performance, requiring us to be more vigilant. In our approach, we adopt a lightweight model instead of a deep learning model and generate diverse queries to enhance the generalization ability of the model.

Decision Tree Algorithms. Decision tree algorithms are popular due to their interpretability, simplicity, and ability to handle both categorical and numerical data. Classical algorithms like ID3 [27], c4.5 [30], CART [6], random forest [5] are designed to solve classification and regression problems. However, the problems we face in our work are not typical regression or classification problems. Hence, we customized a multi-way decision tree based on parameter instability test[42] and calculated the hyperparameters of the cost model from the leaf nodes.

7 CONCLUSION

The formula-based cost model shows good explainability and generalization when involved in cost estimations. However, its performance is outrun by recent learning-based approaches. In this paper, we identify key parameters within cost model formulas and design a fast-learning model. By partitioning the search space, we refine cost model estimation and achieve comparable performance to fine-tuned learning-based models. Extensive tests on commonly used datasets demonstrate that our method outperforms existing techniques. Moreover, our approach also shows a high transferability when dealing with dynamic scenarios. These advantages indicate its non-negligible prospects in practice.

ACKNOWLEDGMENTS

This work was supported by the Key Research Program of Zhejiang Province (Grant No. 2023C01037)

REFERENCES

[1] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 390–401. https://doi.org/10.1109/ICDE.2012.64

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1009–1024. https://doi.org/10.1145/3035918.3064029

[3] Donald W. K. Andrews. 1993. Tests for Parameter Instability and Structural Change With Unknown Change Point. *Econometrica* 61, 4 (1993), 821–856. http://www.jstor.org/stable/2951764

[4] Patrick Billingsley. 2013. *Convergence of probability measures*. John Wiley & Sons.

[5] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324

[6] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Wadsworth.

[7] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 646–659. https://doi.org/10.1145/3514221.3517882

[8] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-based Representation. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*, Aidong Zhang and Huzefa Rangwala (Eds.). ACM, 97–107. https://doi.org/10.1145/3534678.3539303

[9] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning Database Configuration Parameters with iTuned. *Proc. VLDB Endow.* 2, 1 (2009), 1246–1257. https://doi.org/10.14778/1687627.1687767

[10] Fernando Berzal Galiano, Juan C. Cubero, Nicolás Marín, and Daniel Sánchez. 2004. Building multi-way decision trees with numerical attributes. *Inf. Sci.* 165, 1-2 (2004), 73–90. https://doi.org/10.1016/j.ins.2003.09.018

[11] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374. https://www.vldb.org/pvldb/vol15/p2361-hilprecht.pdf

[12] 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. https://doi.org/10.14778/3384345.3384349

[13] Ruichen Jin, Wei Chen, and Timothy W. Simpson. 2000. Comparative Studies Of Metamodeling Techniques Under Multiple Modeling Criteria. *Structural and Multidisciplinary Optimization* 23 (2000), 1–13.

[14] Konstantinos Kanellis, Ramnatthan Alagappan, and Shivaram Venkataraman. 2020. Too Many Knobs to Tune? Towards Faster Database Tuning by Pre-selecting Important Knobs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*, Anirudh Badam and Vijay Chidambaram (Eds.). USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/kanellis

[15] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *Proc. VLDB Endow.* 15, 11 (2022), 2953–2965. https://www.vldb.org/pvldb/vol15/p2953-kanellis.pdf

[16] Johan Kok Zhi Kang, Gaurav, Sien Yi Tan, Feng Cheng, Shixuan Sun, and Bingsheng He. 2021. Efficient Deep Learning Pipelines for Accurate Cost Estimations Over Large Scale Query Workload. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1014–1022. https://doi.org/10.1145/3448016.3457546

[17] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p101-kipf-cidr19.pdf

[18] Thiagarajan Krishnamurthy. 2003. Response Surface Approximation with Augmented and Compactly Supported Radial Basis Functions. https://doi.org/10.2514/6.2003-1748

[19] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *Proc. VLDB Endow.* 5, 11 (2012), 1555–1566. https://doi.org/10.14778/2350229.2350269

[20] 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. https://doi.org/10.14778/3476249.3476254

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. *SIGMOD Rec.* 51, 1 (2022), 6–13. https://doi.org/10.1145/3542700.3542703

[22] 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. https://doi.org/10.14778/3342263.3342644

[23] 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. https://doi.org/10.14778/3342263.3342646

[24] M. D. McKay, R. J. Beckman, and W. J. Conover. 1979. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics* 21, 2 (1979), 239–245. http://www.jstor.org/stable/1268522

[25] Martin Meckesheimer, Russell R. Barton, Timothy Simpson, Frej Limayem, and Bernard Yannou. 2001. Metamodeling of Combined Discrete/Continuous Responses. *AIAA Journal* 39, 10 (2001), 1950–1959. https://doi.org/10.2514/2.1185 arXiv:https://doi.org/10.2514/2.1185

[26] Debjyoti Paul, Jie Cao, Feifei Li, and Vivek Srikumar. 2021. Database Workload Characterization with Query Plan Encoders. *Proc. VLDB Endow.* 15, 4 (2021), 923–935. https://doi.org/10.14778/3503585.3503600

[27] J. Ross Quinlan. 1986. Induction of Decision Trees. *Mach. Learn.* 1, 1 (1986), 81–106. https://doi.org/10.1023/A:1022643204877

[28] J. Ross Quinlan. 1996. Improved Use of Continuous Attributes in C4.5. *J. Artif. Intell. Res.* 4 (1996), 77–90. https://doi.org/10.1613/jair.279

[29] Daniel Revuz and Marc Yor. 2013. *Continuous martingales and Brownian motion*. Vol. 293. Springer Science & Business Media.

[30] Steven Salzberg. 1994. Book Review: C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993. *Mach. Learn.* 16, 3 (1994), 235–240. https://doi.org/10.1007/BF00993309

[31] 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 *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[32] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 99–113. https://doi.org/10.1145/3318464.3380584

[33] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 99–113. https://doi.org/10.1145/3318464.3380584

[34] Il'ya Meerovich Sobol'. 1990. On sensitivity estimation for nonlinear mathematical models. *Matematicheskoe modelirovanie* 2, 1 (1990), 112–118.

[35] I.M Sobol. 2001. Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. *Mathematics and Computers in Simulation* 55, 1 (2001), 271–280. https://doi.org/10.1016/S0378-4754(00)00270-6 The Second IMACS Seminar on Monte Carlo Methods.

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

[37] Xiu Tang, Sai Wu, Mingli Song, Shanshan Ying, Feifei Li, and Gang Chen. 2022. PreQR: Pre-training Representation for SQL Understanding. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 204–216. https://doi.org/10.1145/3514221.3517878

[38] Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Expand your Training Limits! Generating Training Data for ML-based Data Management. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1865–1878. https://doi.org/10.1145/3448016.3457286

[39] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899

[40] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 931–944. https://doi.org/10.1145/3514221.3517885

[41] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116

[42] Achim Zeileis, Torsten Hothorn, and Kurt Hornik. 2008. Model-based recursive partitioning. *Journal of Computational and Graphical Statistics* 17, 2 (2008), 492–514.

[43] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670. https://www.vldb.org/pvldb/vol15/p1658-zhao.pdf

[44] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2021. SIA: Optimizing Queries using Learned Predicates. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2169–2181. https://doi.org/10.1145/3448016.3457262

[45] 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. https://doi.org/10.14778/3485450.3485456
[46] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428. https://doi.org/10.14778/3397230.3397238
[47] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1466–1479. https://doi.org/10.14778/3583140.3583160

Received April 2023; revised July 2023; accepted August 2023