QueryFormer: A Tree Transformer Model for Query Plan Representation

Yue Zhao, Gao Cong, Jiachen Shi, Chunyan Miao
Nanyang Technological University
{zhao0342@e.,gaocong@,jiachen001@e.,ascymiao@}ntu.edu.sg

ABSTRACT

Machine learning has become a prominent method in many database optimization problems such as cost estimation, index selection and query optimization. Translating query execution plans into their vectorized representations is non-trivial. Recently, several query plan representation methods have been proposed. However, they have two limitations. First, they do not fully utilize readily available database statistics in the representation, which characterizes the data distribution. Second, they typically have difficulty in modeling long paths of information flow in a query plan, and capturing parent-children dependency between operators.

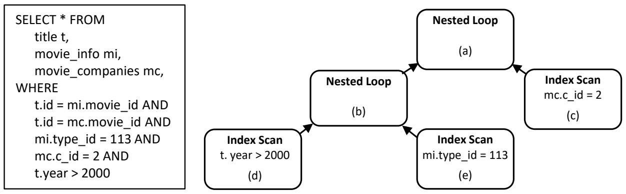
To tackle these limitations, we propose QueryFormer, a learning-based query plan representation model with a tree-structured Transformer architecture. In particular, we propose a novel scheme to integrate histograms obtained from database systems into query plan encoding. In addition, to effectively capture the information flow following the tree structure of a query plan, we develop a tree-structured model with the attention mechanism. We integrate QueryFormer into four machine learning models, each for a database optimization task, and experimental results show that QueryFormer is able to improve performance of these models significantly.

PVLDB Reference Format:
Yue Zhao, Gao Cong, Jiachen Shi, Chunyan Miao. QueryFormer: A Tree Transformer Model for Query Plan Representation. PVLDB, 16(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at github.com/zhaoyue-ntu/QueryFormer.

1 INTRODUCTION

A host of work [9, 16, 17, 30, 36, 37, 39] which leverages machine learning techniques for database optimizations depends on physical query plans. A physical query plan describes a sequence of operations, such as joins and scans, and the algorithms used for operators during query execution [3]. A physical query plan may contain up to hundreds of operations [39] and it can be modeled as a Directed Acyclic Graph (DAG) where each node describes an operation and each edge indicates the dependency of two nodes, i.e., all children nodes are executed before the parent node and the output of each children node is fed into the parent. A real-life example of a query and its query plan is shown in Figure 1.



```
SELECT * FROM
    title t,
    movie_info mi,
    movie_companies mc,
WHERE
    t.id = mi.movie_id AND
    t.id = mc.movie_id AND
    mi.type_id = 113 AND
    mc.c_id = 2 AND
    t.year > 2000
```

Figure 1: Example Query and Query Plan from JOB-Light.

Physical query plans have been used as the input to the machine learning models for database optimization tasks such as cardinality and cost estimation [30], index recommendation [9], query optimization [16, 17], view selection [37], and join order selection [18, 36]. Despite targeting on different tasks, the models proposed in these studies rely on the representations of query plans to learn the correlations between query plan properties and the targeted outputs. Therefore, representation learning for physical query plans, or encoding physical query plans is a cornerstone for the successful application of machine learning techniques to solve database tasks with physical plans as the input.

To extract useful features from physical query plans and encode them into vectors, a number of query plan representation methods have been proposed. A summary of these approaches is shown in Table 1 and we will review them and their limitations in Section 2.2. Overall, they have two limitations: (1) they do not fully utilize the statistics of database content in the representation, and (2) they have difficulty in modeling long paths of information flow and capturing parent-children dependency. We next illustrate the limitation using the Tree-LSTM model [31] as an example, which is used to represent physical plans in E2E-Cost [30] for cost estimation. First, E2E-Cost includes a small sample for each table in the encoding, similar to the encoding method [12]. However, sampling suffers from 0-tuple problem for some queries [22]. In contrast, we argue that the readily available per-table statistics, such as histograms, can provide useful knowledge about data distribution [2]. Second, E2E-Cost uses RNN model, which is generally difficult to train because of the recurrent steps [25]. The issue is more serious for physical plans with long paths. That is, information from leaf nodes may be lost in the recurrent processing steps before reaching top node [25].

Indeed, encoding a physical query plan is a non-trivial task, as it has to capture useful features from each individual node, and aggregate features from nodes in the physical plan. There are three main challenges in physical plan representation. (C1) Plan encoding should include the database statistics information which characterizes the underlying data distribution[2]. It has been shown that database statistics such as samples significantly improve the performance of machine learning models in database tasks like cost

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

and cardinality estimation [12, 30]. However, there is no consensus on what statistics is useful and there is no straightforward way to embed the statistics into machine learning models. Choosing useful statistics and encoding them effectively with query plans is a open problem. (C2) Plan encoding should capture the parent-children dependency in a physical plan. The behavior of a parent node is directly dependent on the behaviors of its children. For example, in the query plan of Figure 1, join node b works on tuples emitted from scan nodes d and e. The interesting features of output tuples from d and e are thus important when estimating the cost and output tuples of the join operation in b. (C3) It is difficult to model the long paths of information flow in a query plan. During query execution, there may be long-range dependencies between nodes. For example, in Figure 1, the cost of the root node a depend on all descendent nodes, e.g., node d, even though a and d are not directly connected. Hence, to study the cost of a node, information from all descendent nodes needs to be considered. The long paths in a query plan makes it difficult to design effective representation models.

To address the challenges, we propose to use self-attention techniques in Transformer [33] for representing query plans. The main characteristic of Transformer is that it gathers information from the entire sequence using self-attention mechanism, while RNN scans through input elements one by one. This makes Transformer more efficient and effective in sequence modeling. However, vanilla Transformer is designed for sequence data but not physical plan as a tree. To this end, we propose a novel tree-structured Transformer model to represent physical query plans (called QueryFormer), which uses self-attention mechanism to model the pair-wise dependencies between nodes in a plan. On top of transformer, QueryFormer models the structural information of query plans using two novel strategies: *height encoding* and *tree-bias attention*. These modifications enable effective capturing of parent-children dependencies (addressing C2), as well as the long paths of information flow (addressing C3). Lastly, we propose a novel histogram encoding scheme and integrate it into query plan representation, which improves model generalization to unseen query predicates (addressing C1).

QueryFormer can be seamlessly integrated to physical plans based machine learning models for database tasks by replacing the physical plan representation with the results of QueryFormer as the input. We apply QueryFormer to four different tasks, including cost estimation, cardinality estimation, index selection and steering query optimizer, and the experimental results show that QueryFormer can improve their performances significantly.

In summary, we make the following **contributions**:

- We abstract the task of physical plan representation, which is an essential and critical component of many machine learning for database methods, and we propose QueryFormer, a tree-structured Transformer model to learn the representation of physical query plans for a variety of machine learning tasks that take physical query plans as input. QueryFormer is featured with two novelties: (1) It effectively captures node dependencies and long paths of information flow of query plans. To the best of our knowledge, this is the first work that adapts attention networks for query plan representation. (2) We propose a new scheme to integrate histograms into physical plan encoding, which improves the generalization of the learned representation.
- We seamlessly integrate QueryFormer into four machine learning for database tasks, including cost estimation, cardinality estimation, index selection and query optimizer, by replacing their representation components with QueryFormer. Extensive experiments demonstrate that QueryFormer is able to significantly improve the effectiveness of these machine learning for database algorithms by replacing their representation components.

2 RELATED WORK

2.1 Physical Plans based Machine Learning

Physical query plans are used as input in a number of machine learning models that are designed for important database tasks, such as index selection [9], cost estimation [19, 30], optimizer [16, 17], etc. (1) For index selection, AIMeetsAI [9] proposes to train a classifier to compare the costs of a query in different index configurations. Specifically, it takes a pair of physical query plans as input, and predicts which plan is better. (2) For join order selection, ReJOIN [18] and RTOS [36] use reinforcement learning to determine the join order. They encode the state of intermediate physical plans as 'state vectors' or 'join forests', which are fed to neural networks to output a join action. (3) For cost estimation, E2E-Cost [30] and Plan-Cost [19] propose to train a regression model to predict the cost of a physical plan. (4) For view selection, AVGDL [38] is an automatic view generation approach. To estimate the benefit of a materialized view, it designs a wide-deep model to estimate the cost of physical plans with candidate view. (5) For query optimizer, NEO [17] proposes a learnable query optimizer which incrementally searches and builds the physical query plan. BAO [16] enhances an optimizer by learning to provide a set of hints or flags to the optimizer for each query. BAO compares physical plans generated by an optimizer under different hint sets using neural networks.

A common and fundamental component of the aforementioned approaches, despite their different targeted applications, is the physical query plan representation. We next review the query plan representation methods in these studies.

2.2 Query Plan Representation Methods

We divide the existing methods of representing physical query plans into four categories based on their representation techniques, as summarized in Table 1. We compare them based on whether they capture parent-children dependency, model long paths of information flow from leaf nodes to root, model database statistics, and have training difficulties, which refer to problems during model training, such as gradient vanishing and explosion [25].

Flattened method such as AVGDL [38] formats the physical plan into a flattened sequence, and combines the node features using the LSTM model. The output is treated as the plan's representation. This approach does not capture full structural information from a plan tree. Particularly, the parent-children dependency may not be captured because a parent node may have a child node far away in the flattened sequence. Additionally, LSTM suffers from forgetting problem for large query plan trees and is difficult to train [25].

Tree-RNN family of approaches improves upon flattened method by aggregating information hierarchically following the tree structure. RTOS [36] and E2E-Cost [30] use Tree-LSTM model [31] to aggregate node information bottom up from leaf nodes to root node,

Table 1: Summary of existing solutions to query plan representation.

Category	Paper	Task	Parent-Children Dependency	Long Path Information Flow	Database Statistics	Training Difficulty
Flattened	AVGDL [38]	View Selection	No	Yes	NA	Hard
Tree-RNN	RTOS [36]	Join Order Selection	Yes	Yes	NA	Hard
	E2E-Cost [30]	Cost, Cardinality	Yes	Yes	Sample	Hard
	Plan-Cost [19]	Cost Estimation	Yes	Yes	Estimated card, cost	Hard
Tree-CNN	NEO [17]	Optimization	Yes	No	Estimated card	Easy
	BAO [16]	Optimization	Yes	No	Estimated card, cost	Easy
	Prestroid [39]	Cost Estimation	Yes	No	NA	Easy
Feature Vectors	ReJOIN [18]	Join Order Selection	No	No	NA	Easy
	AIMeetsAI [9]	Index Selection	No	No	Estimated card, cost	Easy
	LQPP [5]	Cost Estimation	No	No	Estimated card, cost	Easy
Transformer	QueryFormer (Ours)	All	Yes	Yes	Sample, Histogram	Easy

and use the final output as the representation of a physical plan. Plan-Cost [19] designs two types of neural network modules for leaf node and intermediate node, respectively. These methods can model the parent-children dependency and the paths of information flow. However, due to the recursive nature, they suffer from the forgetting problem and are difficult to train for large query plans [25].

Tree-CNN [21] is a generalization to conventional CNN [14], which allows tree-structured inputs. NEO [17] and BAO [16] uses Tree-CNN with triangular-shape filters (parent, left-child, right-child) to slide over a query plan tree. Therefore, they can capture parent-children dependency of a physical plan tree. These methods do not have recursive steps and can process all nodes in parallel, and are thus easy to train. However, these methods have a small receptive field, which means each node can only 'see' features from near neighbors. Therefore, they cannot capture long paths of information flow from leaf nodes to root node. Prestroid [39] improves Tree-CNN by introducing a sub-tree sampling step. A large physical plan tree is decomposed into smaller sub-trees, so that a small receptive field would be less of a problem. However, this method potentially introduces a more serious challenge: important nodes may be dropped in the sampling process.

Unlike the aforementioned methods, Feature Vector approaches directly encode features from query plans using pre-defined rules, instead of using deep learning. In particular, AIMeetsAI [9] and LQPP [5] design a set of important features for each node operator in a physical plan, and collate a list of feature vectors as the physical plan representation. ReJOIN [18] represents the join state of a query plan as a 'state vector'. These methods are much easier to train as they do not involve information exchange modules among query plan nodes. However, they fail to capture parent-children dependencies and information flow from leave nodes to root node.

Database Statistics. As summarized in Table 1, most existing solutions encode some database statistics information that is related to data distribution. Many existing solutions encode the estimated cardinality and cost from database optimizers [2]. These estimates, despite not being accurate, could provide 'domain knowledge' as hints to the representation model [16, 27]. E2E-Cost maintains and encodes a sample for each table as a bitmap in query plan encoding. This sample bitmap helps describing the distribution of the underlying table. However, samples suffer from 0-tuple problem and are not useful for low selectivity queries [12]. On the other hand, histogram is a common per-table statistics readily available in database systems such as PostgreSQL [2] and MySQL [4], which provides a synopsis of data distribution. However, none of existing query plan encoding methods attempts to encode histogram. We fill the gap and propose a method to encode histogram in QueryFormer.

3 PROBLEM AND OVERVIEW

Physical plan based machine learning models have been developed to tackle various database tasks: cardinality estimation, cost estimation, index selection, view selection, join order selection, and query optimization. All the existing work first encodes and represents a physical plan into a vector representation using different techniques as summarized in Table 1. Then, this representation is fed to machine learning models used for different database tasks.

We note that a query plan representation serves as a fundamental block for solving these downstream database tasks. The success of a learning model for a database task depends heavily on whether the useful information in a query plan can be captured in its vector representation. Motivated by this, in this work we focus on learning a query plan representation for downstream database tasks.

Problem Statement. Given a physical query plan P, the task of representation learning aims to learn a vector representation of a query plan based on extracted useful features. The learned representation can be used as input to various machine learning models for downstream database tasks utilizing query plans.

Note that we aim to propose a general representation learning method for physical plans and the learned representations can be used for all the downstream machine learning tasks that take physical plan representation as input. Although previous work focuses on a particular database task only, their representation learning methods are actually general and can be used for other tasks. In fact, the important features of physical plans used across different tasks are the same. Hence, a good representation technique in one task will likely perform well on other tasks.

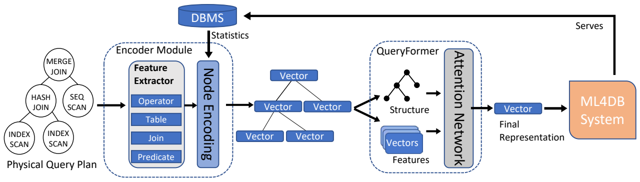


Figure 2: System Overview.

3.1 The Proposed Framework

Overview. To learn a good representation of a physical query plan, we aim to encode both individual node features and the tree structure. We present the workflow of our system in Figure 2. First, we encode important node features for each node of a query plan. We develop an *Encoder Module* that extracts features, and encodes each tree node together with database statistics into a fixed-size vector representation (Section 4). Next, to aggregate features from individual nodes, which follows the tree structure of a query plan, we develop the QueryFormer (Section 5), a tree-structured Transformer model. The output vector representation of QueryFormer can be fed to different machine learning models for various database tasks.

Encoder Module. It extracts useful features from each node of a query plan tree, and encode it in a fix-size vector. We propose to represent the categorical variables (such as operator, table, join and etc.,) by learned embeddings, which has concise dimensions and supports new variables (e.g., when database schema is updated). This is different from the existing methods [17, 30] that are based on one-hot encoding , and they cannot readily support new variables. In addition, we incorporate per-table statistics, including histogram and samples, into predicate encoding, which provides additional information of the data distribution of the tables and columns in the encoding. This addresses challenge C1 in Introduction. The encoder module is presented in Section 4.

QueryFormer Model. It is a tree-structured Transformer model designed to aggregate individual node representations following query plan's structural information. To address the two challenges C2 and C3 of query plan representation (in Introduction), QueryFormer is equipped with novel *Height Encoding* and *Tree-biased Attention* mechanisms, together with self-attention mechanism. Furthermore, QueryFormer aggregates physical plan information effectively without recursive processing, making it much easier to train. The details are explained in Section 5.

4 ENCODER MODULE

We proceed to present our Encoder Module, which aims to incorporate database statistics into the encoding (Challenge C1). We extract useful features from nodes of a physical query plan, and encode them as fixed-size vectors in the embedding space. The tree of vectors will be the input for the QueryFormer model.

4.1 Feature Extraction

A node of a query execution plan typically contains the operator, relation, and predicate and join as explained below.

(1) **Operator** describes the operation of a query plan node, such as Merge Join and Index Scan. It is a categorical variable with a finite domain (around 30, depending on the database system) [3].
(2) **Predicate** describes the filter conditions on relational table columns, such as $t.year > 2000$ from Figure 1. It can be encoded as a <column, operator, value> triplet [12, 30, 39]. Column and operator can be treated as categorical variables like operator, and the value of predicate is range-normalized to $[0, 1]$. This value normalization can generalize to string predicate with equality constraint, by hashing string to integers. However, we do not support wildcard predicate such as 'Like' because it is nontrivial to extract sufficient information such that the number of satisfied tuples can be inferred. One possible solution could be using Astrid [28], which focuses on 'Like' predicate. Nonetheless, it is not our focus and we leave it as future work.
(3) **Join** is the join condition present in join nodes. It is treated as a categorical variable as well, because there are finite possible joins given a database schema.
(4) **Table** refers to the relational table that the node operates on, and is treated as a categorical variable.

The aforementioned information can be used to describe the semantics of a node, and can be used to capture the features of physical plans for various database tasks. We illustrate the importance of the information using the task of cost estimation: such information can be modeled to learn to infer the execution cost of the node. For example, consider the scan node with predicate $t.year > 2000$ in the above example. The predicate condition defines the query region of the relational table t, and is thus essential in determining satisfied tuples. Similarly, the exact scan algorithm, i.e., index scan or sequential scan, affects the execution cost of the node as well, and should be included in the encoding.

We note that an alternative feature selection strategy used in some works such as BAO and AIMeetsAI [9, 16] is to directly use the estimated cost and cardinality from a database system. Specifically, if the physical query plan is obtained from a database system, each node in a query plan also contains estimates from the database optimizer. Our method can easily include the estimates as extra features. However, we choose not to include them because: the estimates may not be accurate for complex and difficult query plans [12, 30]; using these erroneous estimates as features might not be beneficial to our model for difficult query plans, and our model does not need them for easy query plans in the first place.

4.2 Node Encoding

Next, we present how to encode the extracted features from query plan nodes into vector representation. The goal is to capture useful features in the encoding while keeping the vector in a reasonable size. Note that a very large vector incurs unnecessary memory overheads, and limits the batch size during model training [39].

4.2.1 Learned Embedding for Categorical Variable. The features we extract for operator, table and join are categorical variables. Existing methods typically use 'one-hot' encoding, which has the following limitations: (1) Dimension of the one-hot vector can easily blow up for large database schema. For example, a database with hundreds of columns will require hundreds of bits for column encoding alone. (2) It is difficult to handle new categorical variables when database changes. For example, when new columns are added, the encoding scheme has to reset the required dimensions, and thus we have to retrain the whole machine learning system. To alleviate the problem, E2E-Cost [30] proposes reserving some extra bits of '0's at the end of the one-hot vector to accommodate new variables. However, reserving how many bits is an open problem: too few bits may not be sufficient, and too many bits will further blow up the encoding dimension. (3) Sparse representation of the one-hot vector does not reflect the semantic information of values, as all values are orthogonal to each other. For example, if we represent *operator* as one-hot, we may have '>' is 01000, '=' is 10000, and '>=' is 00010. It will be difficult to for a model to infer '>=' is simply the combination of the 2 operators of '>' and '='.

To address these limitations, we propose to use a fixed-size, dense embedding to represent each categorical variable. These embeddings will be learned automatically through back-propagation when a machine learning model is trained for a specific database task, such as cost estimation. As such, the important features that are relevant to the database task could be reflected in the embedding space [10]. As the learned embeddings have constant size and well capture the semantic information of variables, they can facilitate the training of machine learning models for downstream tasks. This setting also naturally supports inserting new variables. To insert a new category variable to the representation model, we simply add a new random-initialized embedding for the variable, without affecting other learned embeddings.

4.2.2 Predicate Encoding. A predicate feature is described as a <column, operator, value> triplet. To get the predicate representation E_p, we concatenate the representation of each element as it is done in previous work [12, 30], i.e., $E_p = [E_c || E_o || v_{norm}]$, where $\|$ represents concatenation, E_c and E_o are column and operator embedding, respectively, and v_{norm} is the normalised predicate value.

4.3 Integrating Database Statistics into Predicate Encoding

To be able to generalize to unseen predicates for database tasks, such as cardinality estimation, we need to model the data distribution of the columns in a predicate. We illustrate the challenge using the example query plan in Figure 1: suppose the model has seen predicates $t.year < 1990$ and $t.year < 2003$ in training data, and encounters predicates $t.year < 2000$ and $t.year < 2010$ in test time. The two test predicates correspond to model's generalization ability to interpolation and extrapolation. If the underlying distribution of *year* column is uniform, and the model is well-trained, the model is expected to 'guess' reasonably well for the two test predicates using linear interpolation and extrapolation. However, if the *year* column has a skewed distribution (which often happens in real-life datasets), the model cannot perform well on the two test cases.

A naive solution would be to collect a very large training data set that sufficiently covers all the possible query regions. Unfortunately, this is prohibitively expensive. To this end, our idea is to explicitly provide the model with additional information about the data distribution of columns. To achieve this, we integrate per-table statistics into the predicate encoding scheme.

4.3.1 Histogram. A histogram is a statistic that summarizes the data distribution in a table column [1]. Many database systems maintain a histogram for cardinality estimation. As histogram is useful for column predicate cardinality estimation, we propose to integrate the histogram to predicate encoding as follows. First, we obtain an equi-height histogram of each column (represented as a list of histogram boundaries), which can be taken from the stored statistics from database systems (e.g. *pg_stats* in PostgreSQL [2]). However, the number of bins for each column is not constant. To unify encoding, we need to re-group the histograms for all columns to the same N number of bins. We approximate the new histogram boundaries by linear-interpolation which assumes uniform distribution within each bin. For example, to re-group a 7-bin histogram to 10 bins, we approximate that the first new boundary is at 0.7 of the first original bin; the second new boundary is at 0.4 of the second original bin, and so on. Lastly, to encode a histogram, we propose to use a size-N vector where each element corresponds to a bin and is marked with the satisfiability of the predicate condition. For example, suppose we have a histogram with $N = 5$ bins and bounds as $(1987, 1999, 2003, 2004, 2006, 2010)$. Then for predicate $year < 2000$, the query region covers the first bin and a quarter of the second bin ($\frac{2000-1999}{2003-1999} = 0.25$). The histogram encoding is $E_h = [1, 0.25, 0, 0, 0]$. As such, we can encode any predicate present in a node using a size-N vector. Empirically, we found $N = 50$ works well for all tasks.

Incorporating histogram information into predicate encoding can improve both interpolation and extrapolation accuracy of the model. Recall the example at the beginning of this section. With the histogram encoding, the model can identify their differences if the number of tuples in $1990 < year \leq 2000$ is very different from those in $2000 < year \leq 2003$, because the histogram representation can reflect the difference.

The attribute-value-independence (AVI) assumption made in purely histogram-based methods often does not hold in practice and it will lead to inaccurate estimates [15]. Different from these methods, our method does not make the independence assumption between columns. We encode the query region for columns in the predicates, and use machine learning to model the correlation between columns. We also note that there exists a host of work on more advanced histogram construction [7, 11, 24, 32], which is orthogonal to our method. Our encoding can be applied to other histogram designs.

4.3.2 Sample. Following the previous work [12, 30], we maintain m sample tuples randomly drawn from each table. Next, for each predicate in the query node, we evaluate the satisfiability of each tuple, and represent the predicate as a m-bit bitmap as E_s. Specifically, if a tuple satisfies a predicate, the bit is set to 1, and 0 otherwise. We set $m = 1000$ by following [12, 30]. Sample bitmap encoding is helpful in predicate encoding, as the bitmap indicates the percentage of satisfied tuples.

4.4 Final Encoding Formulation

Using the methods discussed above, we encode individual components of a query plan node, including operator, join, table and predicates, as vectors which are denoted by E_o, E_j, E_t, and E_p, respectively. We denote the histogram and sample bitmap by E_h and E_s, respectively. Next, we feed each component to a specific linear layer, so that they are transformed to vectors with desired sizes. Lastly, we concatenate the individual parts together as one vector.

Specifically the embeddings are combined as

$$E = \Big\Vert Linear_i(E_i), \quad E_i \in \{E_o, E_j, E_t, E_p, E_h, E_s\} \tag{1}$$

where $\Vert$ represents concatenation, $Linear_i$ is the specific linear layer designed for each component and E is the final embedding for a node representation. For nodes without predicate or join conditions, such as 'Hash' node, we pad these field positions with zeros. As such, the encoder module encodes any operator type to a same size. After encoding each node in a physical plan, the tree of vectors is then taken as input for the QueryFormer model.

5 QUERYFORMER MODEL

To tackles the challenges in parent-children dependency (Challenge C2) and long paths of information flow (Challenge C3), we propose a representation model – QueryFormer. In this section, we first introduce briefly the Transformer architecture, as a main building block of QueryFormer. Next, we present the modifications to Transformer to incorporate the tree structure specific to physical plans. Last, we discuss the model training and the advantages of QueryFormer compared with existing methods.

5.1 Preliminary: Transformer Basics

Transformer consists of a composition of identical attention blocks. The input to an attention block is a list of vectors, e.g., a sequence of word embeddings, and the output is a list of vectors with the same dimension. The core operation is a self-attention module, which encodes each element in the list by incorporating information from other elements [33].

Self-attention Module. At high level, self-attention module allows the model to selectively focus on correlated parts of the input. Specifically, for an input $H \in \mathbb{R}^{n \times d}$ where d is the embedding dimension and n is the number of elements in the list, self-attention first projects it to 3 vectors, denoted by Q , K and V (queries, keys and values). They are obtained by multiplying H with the learned matrices $W_Q \in \mathbb{R}^{d \times d_Q}$, $W_K \in \mathbb{R}^{d \times d_K}$ and $W_V \in \mathbb{R}^{d \times d_V}$, respectively. Next, the Scaled Dot-Product Attention, defined as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V, \tag{2}$$

Attention essentially computes the 'compatibility scores' between queries Q and keys K, and use the normalized scores to multiply with values V. The denominator $\sqrt{d_k}$ is the scaling term which leads to more stable gradients during training. The result is the encoded representation of the input elements. If we break down the matrix form in Equation 2, the unnormalized attention score between 2 arbitrary tokens i and j before softmax function is:

$$A_{ij} = \frac{Q_i K_j^T}{\sqrt{d_k}} = \frac{(H_i W_Q)(H_j W_K)^T}{\sqrt{d_k}}, \tag{3}$$

where H_i, Q_i, K_i are the input vector, query and key for token i. As such, when encoding token i, Equation 2 essentially computes the attention scores between token i and all tokens in the list, and use the attention scores to compute a weighted sum of V as the encoded representation for token i.

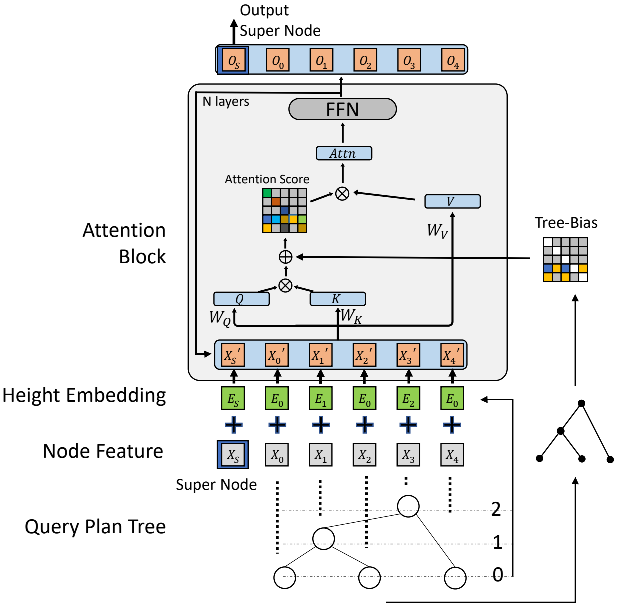


Figure 3: QueryFormer Architecture.

A powerful extension to self-attention module is the multi-head attention. In particular, input H is projected to h sets of different Q, K and V (called h heads), and performs self-attention h times. The output from each head is combined together as the final output. Therefore, self-attention is performed on h different 'representation subspaces'. The multi-head attention is widely used in Transformer variants for better performance [8, 34, 35]. We also use the multi-head extension in the implementation of our QueryFormer.

Positional Encoding: Unlike RNNs which processes the input sequence in order, the self-attention module in Equation 3 does not account for any ordering information of a given input sequence. To inject the ordering information, a positional encoding is added to each token of the input sequence [33]. The positional encodings have the same dimension d as that of an input token. Positional encodings can be either obtained from a pre-defined function [33], or learned embeddings [8].

5.2 QueryFormer

We proceed to present our proposed modifications on Transformer, to accept query plans (i.e., tree of vectors) as input. The goal is to incorporate the tree structure information of a physical query plan. Specifically, we aim to capture parent-children dependencies and long paths of information flow from leaf nodes to root node.

Figure 3 shows an overview of the model architecture of QueryFormer. On top of Transformer, we develop three new components: *height encoding*, *tree-bias attention* and *Super node*. For clarity, we do not show some components that are the same as the original Transformer, such as multi-head extension, softmax function, scaling factors, etc.

5.2.1 *Height Encoding.*

The height of a node is defined as the length of the longest downward path from the node to a leaf in the query plan, i.e., all leaf nodes have height 0, and the root node has the largest height, as shown at the bottom of Figure 3. Intuitively, the height information of a node is useful as it tells whether a node is close to a leaf node (which is often a scan operation), or it works on intermediate results. For example, a node of height 5 means it has at least 1 path that contains 4 operations from a leaf node.

Height information implicitly encodes the parent-children dependency and the paths of information flow as well, because a parent always has higher height than the nodes in its sub-trees. Inspired by the Positional Encoding method, we design a new method to incorporate the height information into the node embedding. As illustrated in Figure 3, the learned height embedding for each height E_h is added to the node features before they are fed into the attention block. This process can be formalized as follow:

$$x_i' = x_i + E_h, \tag{4}$$

where x_i is the node embedding, and E_h is the learned embedding for height h. Similar to how positional encoding allows Transformer to differentiate the order of tokens in a sequence, height encoding allows QueryFormer to distinguish the position information of nodes in a tree. This is because the height information is implanted to node embedding before projecting to Q, K and V in the attention block. As such, QueryFormer can take the relative positions between nodes into account in attention computation.

5.2.2 Tree-bias Attention. To model a query plan, we should allow information flow from bottom to top in order to capture the two types of dependencies, namely parent-child dependencies, and long paths of information flows, which are the two challenges we would like to attack. However, in standard self-attention module (in Equation 3), a node in the input can 'attend to' all other nodes. While this behavior makes sense in language models, it does not match the dependencies in query plan representation.

To this end, we introduce *Tree-bias Attention* as a modification to the original self-attention module. The goal is to use the tree structure to control the information flow in the self-attention module. Specifically, the idea is to mask out illegal information paths, and to account for the relative positions for legal information paths in self-attention module. For example, a node should not attend to nodes from other branches or its parent; It should probably attend more to its direct children than its far descendent nodes.

Tree-bias Attention works as follows. First, we compute the pairwise distance of all nodes in a query plan, i.e., d_{ij} denotes the distance from node i to node j in the query plan DAG, if node j is reachable from i. Otherwise, d_{ij} is marked as *unreachable*. Therefore, only parent nodes can reach to children nodes in a query plan, but not the other way around. Next, we associate each possible distance value with a learnable scalar b_d, i.e., a scalar for each distance. We bias the attention score between every pair of nodes using the learned scalar value based on their distance, by modifying Equation 3 as follows:

$$A_{ij}' = A_{ij} + b_d = \frac{(H_i W_Q)(H_j W_K)^T}{\sqrt{d_k}} + b_d, \tag{5}$$

where A_{ij}' is the new biased attention score and b_d is the learnable scalar bias for distance d from i to j. As shown in Equation 5, the semantic compatibility between nodes (the first term) and the tree-structural bias (the second term) are combined together and both contribute to the new attention score, as illustrated in Figure 3

This formulation allows QueryFormer to effectively control the attention score using the structural information of query plan. To illustrate, the distance for a parent node i to its child node j is always 1. The model can learn to use a large value for b_1 to ensure the new attention score A_{ij}' between node i and node j is always large. Similarly, the model can learn very small negative numbers to completely block attention between far away nodes.

The benefit of *Tree-bias Attention* is that it automatically determines the best way to model the information flow from bottom to top with just a few parameters (a scalar value for each distance). For example, the model may choose to use decreasing bias values for increasing distances, which means that a node always attends more to its near children, and attends less for far away descendent nodes. The model may also choose to set equal bias terms for all distances, which means a node can attend to all of its descendent nodes equally. *Tree-bias Attention* automatically learns the most suitable bias values. When QueryFormer is trained in a different database task, the model may learn to use a different set of bias values that suits the best for the task.

5.2.3 Super Node. The output from the attention blocks is a tree of feature vectors, each corresponding to a node in the query plan. It is challenging to obtain a representation of the whole query plan as off-the-shelf methods can hardly capture all important information. Firstly, Tree-CNN methods [16, 17] use the average or dynamic pooling method to aggregate features from nodes. These pooling operations down-sample information in a brute force manner which will cause information loss. For example, average pooling treats all nodes equally regardless of their data and execution cost. However, the contribution from each node to the final query characteristic is not necessary equal. For dynamic pooling, on the other extreme, it only uses the largest value at each dimension. This may result in some interesting features being pruned and thereby degrading the solution's quality. Secondly, Tree-RNN methods [19, 30, 37] take the root node as the representation of the whole query. However, this method cannot well capture the complicated dependencies between nodes. For example, consider the task of cost estimation, it is difficult for the root node representation to capture cost-related features for all descendent nodes, and the node itself.

To overcome the challenge above, we propose a strategy to collate features from all important nodes. Particularly, we add an artificial *Super* node at the input, as illustrated in Figure 3. *Super* node has a learnable node feature embedding and height embedding, which can be updated just like other tree nodes during training. *Super* node is set to be directly connected to all nodes in the query plan, so that it can 'attend' to all other nodes in the attention layers, and selectively focus on important nodes. The output vector for the *Super* node is treated as the whole tree's representation.

The benefit of using *Super* node is it can gather useful information from all nodes gracefully via the layers of attention calculation, and effectively produce an overall representation of a query plan. This idea is conceptually similar to the $[CLS]$ node in BERT [8], and the $[Graph]$ node in [35].

5.3 Model Training

5.3.1 Training Pipeline. QueryFormer model can be trained in an end to end fashion for a given machine learning for database task.

We next illustrate this with cost estimation as an example, and it follows the same logic for any other task. We first encode training data (query plan trees) using the encoder module and feed the tree of vectors to QueryFormer. Next, the vector representation of the *Super* node is used as the input to the estimation model, such as Multi-layer Perceptron (MLP) for cost estimation task [30]. The loss associated with the task back-propagates to learnable parameters in QueryFormer, and is used to update these parameters. We simply follow the loss functions used by the specific machine learning model for a database task. We further discuss the application of QueryFormer in different tasks in Section 6.

5.3.2 Batch Training. Batch Training with QueryFormer is easy because the model design is embarrassingly parallel, i.e., each mini-batch is simply a list of nodes and adjacency matrices, and thus query plans with different size and shape can be processed simultaneously. Existing methods, however, generally require special treatments on query plans. For example, E2E-Cost [30] performs the batch training by processing nodes level by level in topological order, so that different tree structures can appear in the same batch. This treatment requires additional steps to prepare and track tree levels before and during training. Similarly, Tree-CNN models such as BAO [16] perform batching by computing and tracking positions indexes of all nodes, so that the correct neighbors of nodes can be located in the convolutional layers.

5.3.3 Model Complexity. For a query plan with N nodes, QueryFormer's computational complexity for each layer is $O(d \cdot N^2)$, where d is the hidden dimension for node representation, as it computes pair-wise attention between nodes.

5.3.4 Learnable Parameters. The parameters of QueryFormer include the weights of embeddings in encoder module, height embedding, Super node embedding, tree-bias attention, and other parameters in a standard Transformer architecture.

5.3.5 Hyper-parameters . The hyper-parameters of QueryFormer include the sizes of learned embeddings in encoder module, number of sample points, number of histogram bins and other standard Transformer hyper-parameters, such as number of attention heads, attention layers, etc. For all tasks, we set embedding size of encoder modules to 32, 1000 sample points, 50 histogram bins. For the Transformer backbone, we set number of heads to 12, attention layers to 8, and attention dropout to 0.1. We train all the tasks using Adam optimizer with learning rate of 0.001 until convergence. The important hyper-parameters are sample points, histogram bins, learning rate and dropout. The performance is not sensitive to most other hyper-parameters if they are varied in a reasonable range.

5.4 Remarks

Compare to existing representation methods using Tree-CNN or Tree-LSTM, QueryFormer have several benefits: (1) QueryFormer can model the parent-child dependencies and long dependencies, whereas Tree-CNN and Tree-LSTM have limitations as discussed in Section 2.2. (2) QueryFormer is a more natural model to process query plan tree, as it can take query plans of any shape directly as input, whereas Tree-CNN methods [16, 17, 39] and Tree-LSTM methods [30, 36] assume strictly binary tree: each node must have zero or two children nodes. However, for query plans, nodes with only one child are very common (e.g., sorting, aggregating, etc.), and may have more than two children (e.g., multi-unions). Thus, they have to add a 'null node' for each single child branch, and split branches with more than two children [16, 17].

6 EXPERIMENTS

To evaluate the effectiveness of QueryFormer, we conduct experiments on four machine learning for database tasks: cost estimation, cardinality estimation, index recommendation, and query optimizer. We replace the query plan representation component of these solutions with the result of QueryFormer. For each task, we choose a state of the art baseline method, and adopt the same setting as the original method to ensure fairness. The selected baselines span different categories of existing solutions in terms of query plan representation. We run all experiments on a machine with Intel(R) i9-10900X and GeForce RTX 2080TI.

6.1 Cost Estimation

A fundamental challenge in database management systems is to predict the cost of a query. One major drawback of traditional methods is that they cannot learn from previous mistakes to improve their accuracy. To this end, machine learning based cost estimators have been proposed [30, 39].

6.1.1 Experimental Setting. For this task, we follow the setting of the recent work E2E-Cost [30]. Specifically, we only replace the representation methods of E2E-Cost [30] with QueryFormer, which takes the query plan as input. We follow the other parts of E2E-Cost [30], including a Multi-layer Perceptron (MLP) with Sigmoid activation function for normalized latency prediction. In addition, E2E-Cost [30] also proposes a multi-task learning framework, where a query plan representation can be used to predict both cost and cardinality simultaneously by connecting to two separate prediction models. We experiment QueryFormer on the same multi-task learning method as well. Specifically, we connect the output of QueryFormer to two separate and identical networks to predict cost and cardinality simultaneously. We compare the performance of QueryFormer with E2E-Cost on the multi-task setting.

Dataset We use real-life dataset IMDB [15] with 100,000 queries as training data, and use workload *Synthetic* and *JOB-light* [12] for evaluation. The IMDB dataset contains a snapshot of movie data and has 22 tables. It has skewed distributions within columns and correlations between tables, which make it notably more difficult than benchmark datasets like TPC-H and TPC-DS for cost estimation. The 100,000 queries are split to training and validation set with 9 : 1 ratio. The test set *Synthetic* workload reflects the so-called 'in-distribution' performance, as the queries are generated using the same script as the training data with a different seed. The test set *JOB-light* workload reflects 'out-of-distribution' performance , which are manually designed queries. We summarize the statistics of query plan sizes in Table 2, including the maximum and average number of nodes, and the depth of query plans. We also include datasets for subsequent tasks in the same table.

We execute the queries on PostgreSQL to obtain the query plans and latencies. Each query is executed four times. We ignore the first run to factor out the effect of cache states, and take the average time of the last three runs as the ground truth.

Table 2: Query Plan Sizes in Datasets.

Dataset	Max Nodes	Avg Nodes	Max Depth	Avg Depth
JOB-light	14	8.44	10	5.75
Synthetic	10	4.9	7	3.646
TPC-H	26	16.8	15	10.17
TPC-DS	143	44.4	20	15.2
JOB-extend	35	21.2	19	12.15

Evaluation Metrics Following [29], we report Pearson correlation coefficient of predicted costs and actual costs in logarithmic scale. This measures the goodness of fit between predicted costs and actual costs. We also report Q-Error following [30], which measures the error in ratio [20]: $Q(c) = max\left(\frac{actual(c)}{predicted(c)}, \frac{predicted(c)}{actual(c)}\right)$, where c is the cost of a query.

Algorithms QF is the proposed QueryFormer and is used to replace the query representation module of E2E-Cost. QF (no-hist) is a simplified version of QueryFormer without histogram encoding. The purpose of including QF (no-hist) is to show the effect of incorporating histogram in our method. QF (simple) encodes only the operator, cost and cardinality estimates, which are the feature sets used in AIMeetsAI and BAO as described in Section 4.1. We include QF (simple) to show the effectiveness of our feature selection strategy in Encoder Module. QF-Multi represents the result from multi-task training on both cost estimation and cardinality estimation, and is the counterpart of E2E-Multi [30]. For reference purpose, we also show the results of PostgreSQL and MSCN [12].

6.1.2 *Results.* The experimental results based on Q-Error and Pearson correlation are shown in Table 3.

Table 3: Cost Estimation Results.

Synthetic	Q-Error			Corr
	Mean	Median	90%	
PostgreSQL	12.94	3.78	16.48	0.84
MSCN	1.65	1.17	3.67	0.94
E2E-Cost	4.96	1.81	6.13	0.93
E2E-Multi	2.40	1.55	4.24	0.95
QF (no-hist)	1.61	1.09	2.16	0.98
QF (simple)	2.16	1.21	3.40	0.97
QF	**1.48**	1.08	**1.92**	0.992
QF-Multi	1.49	**1.07**	1.94	**0.994**
JOB-light	Q-Error			Corr
	Mean	Median	90%	
PostgreSQL	25.57	2.74	20.90	0.86
MSCN	25.94	3.43	25.53	0.84
E2E-Cost	45.37	3.39	21.80	0.86
E2E-Multi	21.53	4.84	28.21	0.88
QF (no-hist)	17.86	1.52	28.48	0.86
QF (simple)	15.12	2.47	18.40	0.88
QF	**10.43**	**1.50**	**15.46**	**0.91**
QF-Multi	11.41	1.74	17.77	0.90

Effectiveness of QueryFormer for cost estimation The only difference between QF and E2E-Cost is that QueryFormer is used to replace the query plan representation part of E2E-Cost. By comparing QF and E2E-Cost, we can observe the effectiveness of QueryFormer in improving the accuracy of E2E-Cost. The improvement of QueryFormer on E2E-Cost is significant on both datasets in terms of both Q-Error and Pearson Correlation. QueryFormer outperforms E2E-Cost by more than 230% in terms of mean Q-Error on the *Synthetic* workload, indicating it learns a better mapping from a query plan to cost value. QueryFormer also outperforms E2E-Cost significantly on out-of-distribution workload *JOB-light*, which indicates it is able to generalize and adapt to shifted workload. Similarly, we can also compare QF-Multi with its counterpart E2E-Multi, which uses multi-task Learning for both cost estimation and cardinality estimation. We observe that QueryFormer is still able to improve the performance of E2E-Multi when replacing its query plan representation component.

In addition, we include PostgreSQL and MSCN [12] in Table 3 for reference purpose. However, it is not our objective to compare with other baselines since our objective is only to see whether QueryFormer can improve E2E-Cost by replacing its query plan representation part.

Effectiveness of Model Components To see the effect of histogram, we compare QF and QF (no-hist). They have similar performance on Synthetic workload as shown in Table 3. This is because QF (no-hist) already can work well for in-distribution workload without histogram information. However, on *JOB-light* workload, adding histogram can increase the mean and tail Q-Error by almost 80%, while the median Q-Error remains similar. This shows that both models perform similarly for at least half of the queries in the workload, but divert in the other half of queries. This demonstrates that the benefit of histogram for the easy queries is less significant, and histogram mostly improves performance on hard queries. Similarly, we compare QF with QF (simple) to see the effect of our Encoder Module design. It can be seen that QF performs better in all metrics, showing the effectiveness of our Encoder Module.

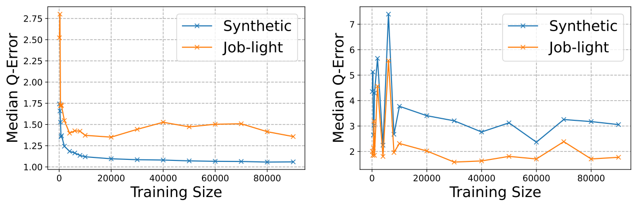
Sensitivity Study To study the sensitivity of model performance on training data size, we vary the training data size from 100 to 90,000 and plot the median Q-Error on the *Synthetic* and *Job-light* in Figure 4. We observe that the Q-error results of QueryFormer become reasonably good when the size of training data reaches around 6,000, and using more training data does not help much. However, the performance of E2E-Cost appears to be very unstable when the training data size is less than 10,000. We also observe that for each training size, QueryFormer consistently outperforms E2E-Cost by a large margin.

Efficiency The total pre-processing and training time for QueryFormer is 2,652.2s, and inference takes 0.010s. E2E-Cost as its counterpart takes 13,659s for training and on average 0.032s for inference. QueryFormer is faster because it does not have recurrent steps.

6.2 Cardinality Estimation

Cardinality estimation aims to estimate the number of output rows for a query. It is a critical component in database systems [3].

6.2.1 *Experimental Setting.* We follow the setting of E2E-Cost [30] for the task of cardinality estimation. We only replace the representation methods, similar to Section 6.1.1. We use the same datasets, and evaluation metrics as those for Cost Estimation task in Section



(a) QueryFormer **(b) E2E-Cost**

Figure 4: Median Q-Error against Training Sizes.

6.1.1. We experiment on both single-task and multi-task setting. We compare E2E-Cost [30] and QF to evaluate if the propose QueryFormer can improve E2E-Cost for cardinality estimation. Similarly, we compare E2E-Multi [30] and QF-Multi, which use multi-task training. We also report results of PostgreSQL [2], MSCN [12] for reference purpose.

6.2.2 Results. The purpose of this experiment is to evaluate whether the proposed QueryFormer (QF) is able to improve the cardinality estimation function of E2E-Cost [30] by replacing its the query representation module. The efficiency results are similar to that for cost estimation and are omitted.

Table 4: Cardinality Estimation Result.

Synthetic	Q-Error			Corr
	Mean	Median	90%	
PostgreSQL	25.44	2.10	13.56	0.94
MSCN	2.89	1.18	3.32	0.992
E2E-Cost	8.30	2.06	9.84	0.95
E2E-Multi	5.30	2.19	7.97	0.95
QF (no-hist)	3.09	1.25	3.87	0.986
QF (simple)	8.42	1.84	19.97	0.94
QF	2.72	**1.11**	3.34	0.996
QF-Multi	**2.64**	1.14	**3.22**	**0.997**
JOB-light	Q-Error			Corr
	Mean	Median	90%	
PostgreSQL	162.34	7.95	157.70	0.88
MSCN	57.9	3.82	78.4	0.81
E2E-Cost	56.85	2.43	39.29	0.89
E2E-Multi	31.63	3.18	30.61	0.89
QF (no-hist)	47.04	3.03	35.57	0.84
QF (simple)	**26.42**	5.34	54.94	0.89
QF	29.50	2.26	38.74	0.91
QF-Multi	31.34	**1.98**	**26.53**	**0.92**

Effectiveness of QueryFormer As shown in Table 4, QF improves E2E-Cost significantly in terms of both Q-Error and Pearson correlation for both workloads. Specifically, QueryFormer outperforms E2E-Cost more than 90% in terms of mean Q-Error for both workloads, indicating its ability to capture relevant features to the cardinality of a query plan. Similarly, QF-Multi performs better than its counterpart E2E-Multi, which is a multi-task learning setting for both cost and cardinality estimation. QF-Multi outperforms E2E-Multi by more than 60% in terms of median Q-Error, showing QueryFormer performs much better for most of the queries. We also include PostgreSQL and MSCN [12] in Table 4 for reference.

Effectiveness of Model Components We observe that the integration of histogram improves the performance of QueryFormer significantly on both workloads, by comparing QF with QF (no-hist) in Table 4. This is because histogram provides a detailed synopsis of the data distribution of tables, which is important to cardinality estimation. Next, we compare QF with QF (simple), as the ablation study for encoder module. We observe QF performs better than QF (simple) in most metrics. For example, the median Q-Error of QF outperforms QF (simple) by more than 65% on both datasets. This shows our encoder module can capture important features for cardinality estimation.

6.3 Index Recommendation

Index recommendation is an important task and it is challenging to determine the optimal set of indexes for a workload [13]. A key component in most index tuners is to compare the estimated costs of query plans of a query under different index configurations. Commercial database systems [6, 40] depend on optimizer's cost estimates to make the comparisons. The recent work AIMeetsAI [9] states that constructing an accurate cost estimator is challenging for both traditional and machine learning approaches. It proposes to train a classifier to compare two query plans of a given query.

6.3.1 Experimental Setting. AIMeetsAI [9] represents a query plan as a set of feature vectors. The feature vectors from two query plans are combined as a 'plan pair' by taking their differences. The combined vector is fed to a machine learning model for a ternary classification task with labels: *IMPROVE*, *NO-DIFF*, and *REGRESS* respectively. The threshold for *IMPROVE* and *REGRESS* is set at 20%. If the difference of the two plans is smaller, the label is *NO-DIFF*, where the difference of the two plans are considered to be negligible. AIMeetsAI [9] explores various machine learning models for the classification task, such as deep neural networks (DNN).

We follow the setting of AIMeetsAI, and only replace the query plan representation from feature vectors to QueryFormer model. We use DNN for classification for convenience, because QueryFormer requires loss signals to train its parameters, and DNN is the only option for an end-to-end trainable system. Specifically, we compute the difference of query plan representations as a plan pair, and feed it to 4-layer fully-connected networks with skip connections to predict the class labels, as AIMeetsAI describes. We use Cross Entropy loss to train the model.

Dataset We use benchmarks TPC-H [26] and TPC-DS [23], both with scale factor of 10. We do not use IMDB [15] because there are very few index candidates for each query in JOB workload, which makes index selection task trivial. We use the toolkit from [13] to generate the candidate index configurations for each query. We have on average 6.4 different plans for each query in TPC-H, and 71.9 different plans for each query in TPC-DS. We summarize the sizes of query plans in Table 2. It can be observed that TPC-DS is more complicated than TPC-H, with 150% more nodes and 50% more depths in query plans. We execute the query with the materialized index configurations to obtain the query plans and their latencies.

Following AIMeetsAI [9], we split the query plan pairs using 3 strategies, which are in ascending order of difficulty: (1) **Pair**: all plan pairs are divided into disjoint sets. (2) **Plan**: the model is trained only on a subset of query plans for each query, and test on plan pairs constructed with at least one unseen plan. This simulates the real-life index tuner setting. (3) **Query**: the model is trained on query plans from a subset of queries and test on the other subset. We notice that some queries are much easier than other queries in both TPC-H and TPC-DS, and it is unfair to put these queries in either training or test set when we perform splitting by Query. To this end, we split the queries randomly multiple times and report the average score.

Evaluation Metrics Following AIMeetsAI [9], we evaluate the performance in two aspects: the classification accuracy and the quality of the final index configuration selected. The classifier is to correctly predict if a query plan is better than the other, and is the key component proposed by AIMeetsAI. We measure classification with accuracy and average F1 score. We evaluate the index configuration quality by measuring the relative time of executing a query compared to execution without any materialized index.

Algorithms QF is the classifier with QueryFormer as query representation module, and the other parts the same as DNN of AIMeetsAI [9]. We also include QF (no-hist) and QF (simple) for ablation study. QF (simple), as a counter-part to QF, is to demonstrate the effectiveness of Encoder Module, while QF (no-hist) is to demonstrate the benefits of histogram. AIMeetsAI follows the original paper exactly. We also report PostgreSQL for reference and its classification accuracy is based on its own cost estimates.

6.3.2 *Results.*

We present the classification results of each model with different splitting strategies in Table 5.

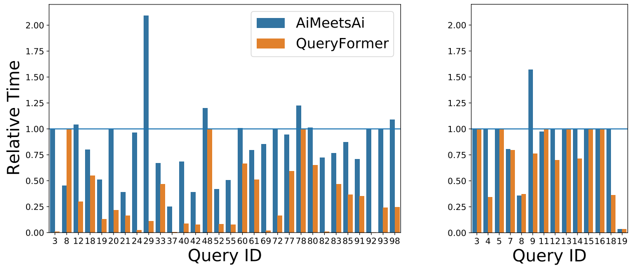
Table 5: Index Classification Accuracy.

TPC-H	by Pair		by Plan		by Query	
	Acc	F1	Acc	F1	Acc	F1
AIMeetsAI	0.95	0.948	0.77	0.76	0.67	0.61
PostgreSQL	0.59	0.59	0.59	0.59	0.59	0.59
QF (simple)	0.992	0.991	0.92	0.91	0.59	0.49
QF (no-hist)	**0.993**	0.992	0.92	**0.92**	0.72	0.70
QF (Ours)	**0.993**	**0.993**	**0.93**	**0.92**	**0.79**	**0.77**
TPC-DS	by Pair		by Plan		by Query	
	Acc	F1	Acc	F1	Acc	F1
AIMeetsAI	0.836	0.820	0.73	0.68	0.62	0.52
PostgreSQL	0.68	0.669	0.68	0.67	0.68	0.67
QF (simple)	0.934	0.929	0.87	0.87	0.67	0.60
QF (no-hist)	0.945	0.941	0.90	**0.91**	0.71	0.65
QF (Ours)	**0.953**	**0.950**	**0.91**	**0.91**	**0.77**	**0.71**

Effectiveness of QueryFormer in Plan Pair Classification QueryFormer is able to improve AIMeetsAI significantly for all splitting strategies on both datasets. For TPC-H, QueryFormer outperforms AIMeetsAI by more than 15% for both accuracy and F1 score in split by query, which is the most difficult splitting strategy. QueryFormer performs better in easier splitting as well, when compared with AIMeetsAI, which only differ with QueryFormer in the query plan representation component. For TPC-DS, QueryFormer outperforms AIMeetsAI by more than 20% in split by query for both accuracy and F1 score as well. This shows QueryFormer performs well on not only seen queries and query plans, but also on unseen queries. This could be because QueryFormer captures important features from all query plan nodes and can model the information flow between nodes. Therefore, QueryFormer can be generalized to unseen queries and query plans.

Effectiveness of Model Components QF (simple) ablates the encoder module, which performs only slightly worse than QF on easy splitting strategies (by pair and by plan) on both datasets. However, its accuracy and F1 score drops by 20% for TPC-H, and 10% for TPC-DS on spit by query. This shows that QueryFormer without our encoder module suffers from serious overfitting problem, and cannot predict correctly for unseen queries. On the other hand, QueryFormer with our encoder module is able to capture and encode useful features that are general across different queries. Next, we compare QF with QF (no-hist) to see the importance of histogram. Similarly, QF (no-hist) performs well compared to QF on easy splitting, but degrades on split by query. This shows the generalization power of QueryFormer drops without histogram and predicts less accurately for unseen queries.

Usefulness of QueryFormer in improving Index Recommendation Quality We compare the quality of index configuration selected by integrating the classifier to an index tuner, so that we can evaluate how the classification accuracy translates to index quality. Following AIMeetsAI, we adopt split by plan strategy which would resemble a real-life index tuning scenario, where model is trained on a subset of query plans. We apply the per-query level tuning, a common scenario for Database Administrator (DBA) to tune each query separately and to get the best index configuration for a query. This setting is the same as described in AIMeetsAI [9].



(a) TPC-DS

(b) TPC-H

Figure 5: Index Selection Results of AIMeetsAI and QueryFormer

We report the results in Figure 5, which depicts the relative execution time of each query with the chosen index configuration compared to execution without any index. For TPC-DS, we omit the queries where AIMeetsAI and QueryFormer has less than 20% difference for clarity, because the classification threshold is set at 20%. As shown Figure 5, QueryFormer performs better than AIMeetsAI and selects index configuration with the same or faster run time for most of the queries on both datasets. In total, QueryFormer chooses queries with 25% faster execution on average for TPC-DS, and 20% for TPC-H, compared to AIMeetsAI. An important requirement for index tuner is to not create indexes that cause *query performance regression* [9]. The number of queries with regression for AIMeetsAI is 3 in TPC-DS and 1 in TPC-H, but the number is 0 for QueryFormer on both datasets, as the relative time for QueryFormer is always smaller or equal to 1.0. The outstanding result of QueryFormer is unsurprising, as QueryFormer has higher classification accuracy when comparing index configurations. Indeed, it is shown that the high performance in classification translates to better index configurations.

Efficiency QueryFormer takes 695.1s for pre-processing and training, 0.027s for inference. AIMeetsAI takes 137.1s for pre-processing and training, 0.009s for inference. AIMeetsAI is faster due to its simpler encoding and model design.

6.4 Query Optimizer

The recent work BAO [16] proposes to build upon traditional query optimizer since it is more challenging to build machine learning based query optimizer from scratch. Specifically, it proposes to provide per-query 'hints' to the traditional query optimizer using reinforcement learning, so that the optimizer can produce a better query plan for each query. Such a task is called 'steering optimizer'.

This experiment is to evaluate whether QueryFormer can improve the task of steering optimizer if we replace the query plan representation module of BAO [16] with QueryFormer.

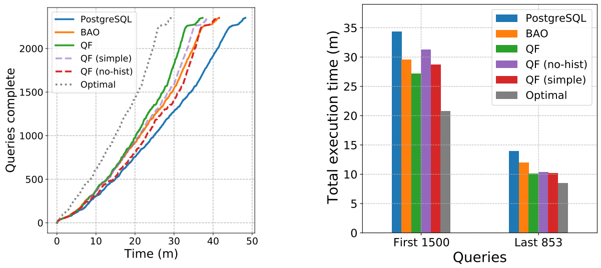
6.4.1 Experimental Setting. Given a sequence of queries to be executed by the database system. For each query, BAO requests a set of query plans from the database optimizer with respect to a set of hints, e.g., *disable_mergejoin*. BAO then evaluates the query plans based on a value network, and select a query plan. After that, the database system executes the query using selected query plan and BAO records the rewards in terms of the latency of query execution. Periodically, BAO retrains its value network using the record history. As such, BAO becomes more and more reliable over constant workload, and adapts to workload shift automatically.

In BAO's value network, it encodes the operator, cost and cardinality estimation of each node, and uses Tree-CNN model to learn a represent for a complete query plan. We replace the query representation module with QueryFormer, and evaluate whether QueryFormer can improve the performance of BAO. We follow the setting of BAO for all other parts of the system.

Dataset By following BAO [16], we use the IMDB dataset with JOB-extend workload, which includes 113 queries from original JOB, and additional 2,240 queries derived from the same query templates as JOB. The statistics for all query plans in JOB-extend are shown in Table 2. We follow the time-series split strategy for training and testing: queries from new templates are introduced incrementally. We evaluate the performance on the latency of next unseen query.

Algorithms We compare the execution time of query plans selected by the original BAO, and by BAO using QueryFormer for plan representation. We also report QF (no-hist) and QF (simple). Similar to previous task settings, QF (simple) uses BAO's encoding, and is to ablate our encoder module, while QF (no-hist) is to ablate histogram. We include the execution time of queries with PostgreSQL without any hint from BAO for reference. Last, we execute all candidate plans from BAO's hint sets for each query, and record the best possible execution time of the query. We denote the best possible solution as Optimal.

6.4.2 Results. We follow the settings of BAO. Specifically, we execute the 2,353 queries in the same order using different algorithms, and compare the execution time. The results are shown in Figure 6. Figure 6a depicts the number of queries completed versus elapsed time. The blue line shows the all queries are completed in about 48 minutes with PostgreSQL, whereas the dotted line shows the optimal total execution time is less than 30 minutes if the best hint set for each query is always chosen.



(a) Number of queries completed over elapsed time.

(b) Total time for executing queries.

Figure 6: Query execution time for different algorithms.

Effectiveness of QueryFormer in reducing running time Figure 6a shows that QueryFormer outperforms original BAO and finishes executing all queries with less time. This proves the effectiveness of QueryFormer in the setting of steering optimizer. Figure 6b shows the total execution time for the first 1500 queries and last 853 queries. QueryFormer performs similarly with BAO for the first 1500 queries. However, for the last 853 queries, QueryFormer significantly outperforms BAO by 18% (less running time). Surprisingly, this is comparable with the improvement of BAO over PostgreSQL, which is 16%!

Effectiveness of Model Components We compare QF with QF (simple) to study the effectiveness of our encoder module design. As shown in Figure 6a, Our encoding is more effective as QF finishes executing queries faster. Interestingly, QF (simple), which has the same encoder module as BAO, performs better than BAO. This suggests that the attention-based QueryFormer architecture can outperform Tree-CNN, even with BAO's node encodings. Last, we compare QF with QF (no-hist). It can be seen that QueryFormer degrades significantly without histogram, especially for the first 1500 queries if we look at Figure 6b. This is because without histogram, it is much harder to learn the information of predicates from limited amount of training data.

Efficiency Training time at each iteration for QueryFormer and BAO are 9.82s and 9.72s respectively. QueryFormer takes 0.011s for inference while BAO takes 0.021s. The difference is negligible for the use case.

7 CONCLUSION

In this paper, we consider the problem of query plan representation, which is a fundamental building block for machine learning for database algorithms. We propose QueryFormer, a tree-structured Transformer architecture which effectively capture the node dependencies and information flow in a query plan. We evaluate the effectiveness of QueryFormer by extensive experiments on 4 query plan based machine learning for database tasks: cost estimation, cardinality estimation, index selection and query optimizer. The results show that QueryFormer significantly improves the performance of existing methods by replacing their query plan representation.

REFERENCES

[1] [n.d.]. *Database SQL Tuning Guide, 11 Histograms.* https://docs.oracle.com/database/121/TGSQL/tgsql_histo.htm#TGSQL366

[2] 2021. *Documentation PostgreSQL 12 71.1. Row Estimation Examples.* https://www.postgresql.org/docs/12/row-estimation-examples.html

[3] 2021. *Documentation PostgreSQL 12, Explain.* https://www.postgresql.org/docs/12/sql-explain.html

[4] 2021. *MySQL 8.0 Reference Manual, EXPLAIN Statement.* https://dev.mysql.com/doc/refman/8.0/en/explain.html

[5] 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).*

[6] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. 2004. Automatic SQL Tuning in Oracle 10g. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004.*

[7] Amol Deshpande, Minos N. Garofalakis, and Rajeev Rastogi. 2001. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, 2001.*

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019.*

[9] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations *(SIGMOD '19).*

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Eng. Bull.* 40, 3 (2017).

[11] H. V. Jagadish, Hui Jin, Beng Chin Ooi, and Kian-Lee Tan. 2001. Global Optimization of Histograms. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, 2001.*

[12] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2018. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *CoRR* abs/1809.00677 (2018).

[13] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic Mirror in My Hand, Which is the Best in the Land? An Experimental Evaluation of Index Selection Algorithms. 13, 12 (2020).

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012.*

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

[16] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD '21: International Conference on Management of Data.*

[17] Ryan 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).

[18] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018.*

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

[20] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (2009).

[21] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*

[22] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *Proc. VLDB Endow.* 11, 9 (2018).

[23] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases, 2006.*

[24] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick Selectivity Learning with Mixture Models. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020.*

[25] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013 (JMLR Workshop and Conference Proceedings),* Vol. 28.

[26] Meikel Pöss and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.* 29, 4 (2000).

[27] Tom Seymoens, Femke Ongenae, An Jacobs, Stijn Verstichel, and Ann Ackaert. 2018. A Methodology to Involve Domain Experts and Machine Learning Techniques in the Design of Human-Centered Algorithms. In *Human Work Interaction Design. Designing Engaging Automation - 5th IFIP WG 13.6 Working Conference, 2018 (IFIP Advances in Information and Communication Technology),* Vol. 544.

[28] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning. *Proc. VLDB Endow.* 14, 4 (2020).

[29] 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.*

[30] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *PVLDB* 13, 3 (2019).

[31] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *CoRR* abs/1503.00075 (2015).

[32] Hien To, Kuorong Chiang, and Cyrus Shahabi. 2013. Entropy-based histograms for selectivity estimation. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13.*

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017.*

[34] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *CoRR* abs/1906.08237 (2019).

[35] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. 2021. Do Transformers Really Perform Bad for Graph Representation? *CoRR* abs/2106.05234 (2021).

[36] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).*

[37] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *36th IEEE International Conference on Data Engineering, ICDE 2020.*

[38] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *36th IEEE International Conference on Data Engineering, ICDE 2020, 2020.*

[39] 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.*

[40] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004.*

A HANDLING UPDATE

We demonstrate handling update on cost estimation. We consider two types of updates: database insert, and new workload. Database insert refers to a set of new tuples inserted into the database. In the case, the cost associated to a query plan changes, and thus the old model may not perform well for making predictions. Workload shift refers to a new set of queries with new characteristics. In this case, the old model may not perform well.

A.1 Database Insert

QueryFormer handles database update by either full re-training or fine-tuning on the old model. Fine-tuning refers to training a few epochs from the old model using the new data.

To simulate database updates, we split the IMDB database into 4 equal-size partitions in time order based on movie's production year, i.e., *production_year* column in the central fact table *title*. This partitioning defines a complete full join of the IMDB database. We treat the first partition as the initial database. We then incrementally insert all tuples from subsequent partitions to the fact table and all dimension tables. As such, the database is updated 3 times.

We compare the performance of (1) the stale model trained with the initial database, (2) fully retrained model after each update, (3) fine-tuned model (with only 4 epochs, which has very little training effort) from the stale model. Figure 7a shows the result. The stale model degrades in performance significantly after each update, as expected. Fine-tuned model performs only slightly worse than fully retrained model. Notice that the gap between Fine-Tune and Re-Train increases over time. This suggests that if the data in a database is changed drastically from the original version, a full re-training is recommended.

We implement the same update strategy for E2E-Cost as well with 20 epochs for fine-tuning, and present the results in Figure 7b. We can see the general trend is similar to QueryFormer, that stale model performs the worst, and re-trained model performs the best. QueryFormer consistently outperforms E2E-Cost in both fine-tune and re-train methods.

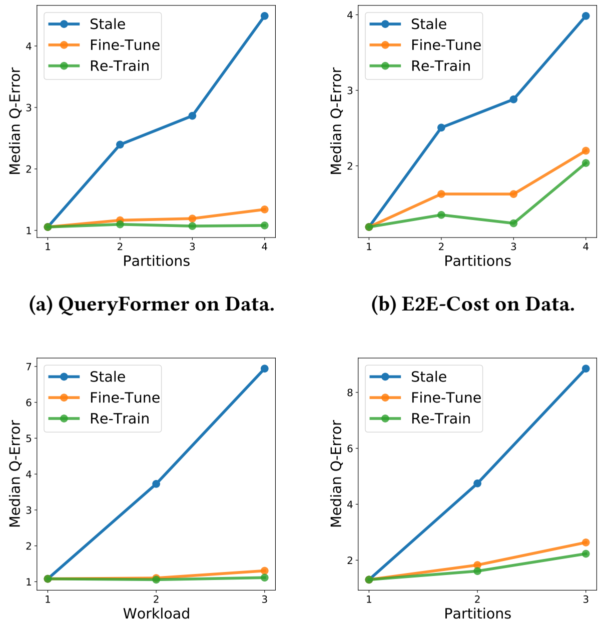
A.2 Workload Shift

To handle new workload, QueryFormer fine-tunes a few epochs from the old model using queries from the new workload.

To simulate the Workload shift scenario, we split the queries from the IMDB database based on the number of relations queried. Specifically, the first workload only queries on 1 relation, the second workload only queries on 2 relations, etc. We train QueryFormer on the first workload, and shift to subsequent workloads.

Similar to Update in Database, we compare the performance of (1) the stale model, (2) fully retrained model on the new workload, (3) fine-tuned model (with 20 epochs), as shown in Figure 7c. Stale model trained on Workload 1 performs poorly on unseen workloads. However, with a few epochs of fine-tuning, QueryFormer performs almost as well as a fully retrained model on the new workload.

We test the same update strategy on E2E-Cost with 25 epochs for fine-tuning, and present the results in Figure 7d. We observe similar trend in E2E-Cost as compared to QueryFormer. However, the gap between Fine-Tune and Re-Train is larger in E2E-Cost even with more epochs, indicating QueryFormer is more effective in fine-tuning for updates.



(a) QueryFormer on Data.

(b) E2E-Cost on Data.

(c) QueryFormer on Workload.

(d) E2E-Cost on Workload.

Figure 7: Updating QueryFormer and E2E-Cost with Database Insert and Workload Shift.