Eddie: Accurate and Robust Index Benefit Estimation Through Hierarchical and Two-dimensional Feature Representation

Tao Li*†
Database Group
State Cloud, China Telecom
lit51@chinatelecom.cn

Feng Liang*‡
AI Research Institute
Shenzhen MSU-BIT University
fliang@smbu.edu.cn

Jinqi Quan
Database Group
State Cloud, China Telecom
quanjq1@chinatelecom.cn

Zihang Yang
Database Group
State Cloud, China Telecom
yangzh34@chinatelecom.cn

Teng Wang
Database Group
State Cloud, China Telecom
wangt_5@chinatelecom.cn

Runhuai Huang
Intelligent Edge Department
State Cloud, China Telecom
huangrh@chinatelecom.cn

Xiping Hu†‡
AI Research Institute
Shenzhen MSU-BIT University
huxp@smbu.edu.cn

ABSTRACT

Index advisors have become indispensable tools for database administrators to optimize index configuration, especially in the face of complex workloads and databases. To find the optimal index configuration for a given workload, index advisors need to evaluate a large number of candidate indexes, and thus necessitate an accurate and efficient method to estimate the benefits of candidate indexes. Recently, machine learning (ML) techniques have been leveraged for index benefit estimation, showing significant advantages in estimation efficiency and removing the dependence on "what-if" calls. However, existing ML-based approaches encounter several drawbacks, such as unsatisfactory accuracy and vulnerability to environmental changes (like workload/schema drift and data update). In this paper, we first pinpoint the reasons for these drawbacks, and then propose a novel approach, called Eddie, to address them. Eddie introduces a new feature extraction and representation technique, called *hierarchical and two-dimensional encoding (HTE)*. It hierarchically consolidates all the index benefit-relevant features into a concise representation and leverages a two-dimensional attention mechanism to characterize both the tree-structure query plan and index interaction. Furthermore, HTE can produce *transferable* feature representation and thus enables pre-training and fine-tuning techniques applied for index benefit estimation. Experimental evaluation shows that Eddie can significantly outperform state-of-the-art index benefit estimators in terms of both accuracy and robustness to environmental changes. Eddie can be easily integrated into existing index tuners and demonstrates notable performance improvement from an end-to-end perspective.

PVLDB Reference Format:
Tao Li, Feng Liang, Jinqi Quan, Zihang Yang, Teng Wang, Runhuai Huang, and Xiping Hu. Eddie: Accurate and Robust Index Benefit Estimation Through Hierarchical and Two-dimensional Feature Representation. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

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

1 INTRODUCTION

Indexes are important data structures on databases that, if created appropriately, can greatly improve query performance. However, given a database and a query workload, finding the optimal *index configuration* (i.e., a set of indexes) is challenging due to the hardness of the problem [23]. Over the past decades, both the academia and industry have been actively building and polishing various types of index advisors to automate or aid this task [9, 10, 31, 38]. At a high level, index advisors first generate valid indexes (called *candidate indexes*) based on the given workload and database schema, and then search for the optimal index configuration over the index space. During the search, index configurations are iteratively enumerated and evaluated in terms of their benefits to the workload, called *index benefits*, which are typically measured by the reduction of the workload's execution cost once the index configuration replaces the original one on the database. Considering the huge search space resulted from the large number of queries and indexable columns, as well as the combinatorial choices of different index sizes and column orders, it is paramount to devise an efficient and accurate

*These authors contributed equally to this work, ordered alphabetically by surname.
†Corresponding authors.
‡Also affiliated with Guangdong-Hong Kong-Macao Joint Laboratory for Emotional Intelligence and Pervasive Computing, Shenzhen MSU-BIT University, China
This work was partially supported by Innovation Team Project of Guangdong Province of China (No. 2024KCXTD017).

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

method to *estimate index benefits* without having to physically materialize the candidate indexes.

So far, most of existing index benefit estimation (IBE) methods rely on *what-if* calls [6], which enable index advisors to create hypothetical indexes for candidate index configuration, obtain optimal query plans under the index configurations, and measure index benefits based on the estimated query execution cost. However, these methods suffer from three drawbacks: 1) they are inapplicable to databases without what-if analysis capability, like MySQL, which actually has a huge user base. 2) they are costly, as what-if calls can take considerable amount of tuning time (up to 90% [20]). This thus has attracted some endeavors to reduce the number of what-if calls by machine learning (ML) models [28], cost cache [16], and approximation techniques [40], but the overhead of what-if calls is still non-negligible. 3) what-if calls run on user databases, can consume database resources, and adversely disrupt user businesses [10]. To address these drawbacks, ML techniques have been introduced for IBE so as to eliminate the dependence on what-if calls. Another advantage of this approach is that the estimation models can be deployed separately from databases and parallelized on multiple GPUs, resulting in significant savings of index optimization time.

However, it is challenging to build an IBE model that is both accurate and generalizable to unseen scenarios. Our insight is that such an IBE model should satisfy the following three requirements. **(R1) Comprehensiveness**: There exist a lot of factors that have impact on index benefit, such as column order in the indexes and SQL operators, predicate expression in the query, interaction of the candidate indexes [24], statistics of data accessed by the query, etc. All these factors should be featurized and encoded into a comprehensive representation in order to learn an accurate index benefit estimator. **(R2) Position-awareness**: Column positions play a key role in determining whether an index can serve a query. For example, in MySQL, the ORDER BY clause can exploit B-tree indexes to sort data only if the columns in the clause have the same order as the index and form a leftmost prefix of the index [3]. Hence, both the column positions in indexes and in SQL operators should be considered. **(R3) Drift-tolerance**: As applications on the database evolve, workload, table schema, and data distribution are likely to drift over time [38]. For example, queries are generally not fixed during exploratory analysis, new columns can be added, and data will be updated. Clearly, the IBE model should be schema-agnostic and resilient to those changes in order to make accurate prediction for unseen data. Unfortunately, some of these requirements have not been considered by existing work, resulting in unsatisfactory performance in accuracy and generalization. For example, LIB [25] assumes static workload and neglects factors such as the initial indexes on databases, sorting direction in ORDER BY, and column position in SQL operators (violation of R1 and R2); DISTILL [28] only focuses on simple workloads pertaining to specific template SQL, cannot capture join conditions, and only records the highest column position across the indexes in the index configuration (violation of R1, R2, and R3). A concrete example will illustrate these problems in Section 2.3.

To meet the above requirements simultaneously, we propose an encoder-based index benefit estimator, called Eddie. It accepts four inputs (i.e., query plan under initial indexes, table schema, candidate index configuration, and column data statistics) and predicts a benefit value for the given candidate index configuration. The main idea of Eddie is to learn a good representation [2] that can capture useful index benefit-related features (satisfying R1) and is *transferable* across diverse databases and workloads (satisfying R2&R3) [13]. To this end, we propose a new feature extraction and representation technique, called *hierarchical and two-dimensional encoding (HTE)*. First, features related to candidate index configuration and column data statistics are extracted and embedded into the query plan to reflect their potential impacts on the query plan. Then, HTE progressively builds the representation of the entire query plan from low-level elements (e.g., identifier) to high-level ones (e.g., expression), forming an encoding hierarchy. In this way, we can not only systematically capture all influential factors from different levels of a query plan, but also enable the reuse of low-level features in high-level ones. In addition, we introduce a *two-dimensional attention mechanism* that can capture the inter-node correlation in a plan tree [42] (i.e., plan-dimension) and the cross-index interaction [24] on the same plan node (i.e., index-dimension). Furthermore, HTE leverages a novel position-based embedding method, rather than relying on column names, to encode the columns appearing in indexes and plan nodes, thus achieving both position-awareness and schema-agnostic.

In the experiments, we have evaluated our approach and compared with three existing ones (i.e., traditional what-if based estimator, the state-of-the-art LIB [25], and an augmented version of QueryFormer [42]) based on a variety of open workloads (TPC-DS, TPC-H and IMDB) and synthesized ones. Results have shown that Eddie outperforms the existing approaches in terms of both estimation accuracy and robustness to various changes in workload, table schema and data statistics. Specifically, Eddie can reduce the estimation error by more than 50% compared to LIB. By leveraging the models pre-trained on other datasets, Eddie is able to reach higher estimation accuracy even with a half of the training data. In addition, Eddie can be easily integrated into existing index advising tools, like AutoAdmin [8], and demonstrate significant performance improvement from an end-to-end perspective.

To summarize, our main contribution is a novel ML-based method to predict index benefit which establishes the new state-of-the-art performance. Such method is also promising due to 1) robustness to various types of environmental changes, 2) adaption to new index tuning tasks by requiring few training data, 3) easy integration with existing tools.

2 BACKGROUND AND PROBLEM STATEMENT

2.1 Index Advisor and Index Benefit Estimation

Index advisors are generally composed of three components: *index candidate generation*, *index selection* and *index benefit estimation* [44]. The first component extracts indexable columns based on query syntax and table schema, and then combines them to generate *candidate indexes* according to different strategies, like random column permutation and prefix-based expansion [44]. The second component selects subsets of the candidate indexes to form index configurations, which we call *candidate index configurations*, or candidate configurations for short. Again, there exist various selection methods to explore the candidate index space, such as

heuristic searching and learned selection strategies [44]. During the index generation and selection processes, the third component can be invoked from time to time to estimate the benefit of an index or an index configuration. After evaluating all the candidate configurations or the stop condition has been met, the index advisor will output the optimal set of index configurations, which yield the greatest benefit for the entire workload.

In the above workflow of index advisor, index benefit estimation plays a key role. Previous work [12, 25] has already shown that efficiency and accuracy of index benefit estimation are crucial to an index advisor. So far, there are mainly two categories of approaches to estimate the index benefit: *what-if calls* and *learned models*. The former relies on the database optimizer's cost model and its associated hypothetical index capabilities. However, such approach is prone to cause large estimation error, takes majority of execution time, consumes a lot of database's CPU resources, and is inapplicable to databases without hypothetical index support, like MySQL. To address these issues, the latter approach leverages machine learning models to predict index benefits, and can run more efficiently than what-if calls and be generalized to different database types. Our approach in this paper also falls into this category.

2.2 Problem Statement

We define index benefit estimation as a regression problem.

Database. A database D contains a set of tables, denoted by T. Each table $t \in T$ is characterized by a set of columns/attributes A^t (i.e., table schema) and a number of data rows whose distribution is described by the statistics S^t. Typical statistics include number of rows, max, min and histogram for each column, etc. Equivalently, we can think of database D as a tuple (A, S), where A is the entire column space and S is the data statistics for all the tables in T, respectively.

Workload. A workload W is a set of queries that have been executed on database D within a certain period of time. For a query $q \in W$, we use T_q to denote the set of tables referenced by the query.

Index. Indexes are auxiliary data structures on tables that can be used to improve query performance. Each index I is defined on a table t^I and contains an ordered set columns from the table. An index that has been physically materialized before the index tuning process is called an *initial index*, whereas an index generated by the index advisor is called a *candidate index*.

Initial index configuration. An *index configuration* is a group of indexes, either from a same table or multiple tables. The initial index configuration C^0 contains all the initial indexes on the tables in database D. However, for a specific query q, only a subset of C^0 is relevant to it, which is denoted as C^0_q.

Candidate index configuration. A candidate index configuration C' is a set of candidate indexes generated by the index advisor for the tables in database D. The index advisor may iteratively generate a series of candidate index configurations and select the optimal one.

Query Plan. A query plan for query q represents the execution plan to run the query. We denote it by $P_q = (N, E)$, where N and E represent the node set and edge set, respectively. Essentially, a query plan is organized as a tree structure, where each node is an operator and an edge denotes the intermediate results flowing from child to parent.

Execution Cost. Given a query q and an index configuration C, let $Cost(q, D, C)$ be the execution cost of the query when C is the index configuration on database D. In fact, the execution cost is dependent on many factors, such as the query, index configuration and the table data statistics.

Index Benefit. For a single query q, the index benefit $B(q, D, C^0, C')$ is defined as the *cost reduction ratio* when the initial index configuration C^0 is replaced by a candidate index configuration C' on database D. Formally, the cost reduction is

$$R(q, D, C^0, C') = Cost(q, D, C^0) - Cost(q, D, C'),$$

and the cost reduction ratio is

$$B(q, D, C^0, C') = \frac{R(q, D, C^0, C')}{Cost(q, D, C^0)}. \tag{1}$$

Thus, the overall cost reduction over the entire workload W is:

$$\begin{aligned} R^W(D, C^0, C') &= \sum_{q \in W} R(q, D, C^0, C') \\ &= \sum_{q \in W} Cost(q, D, C^0) \cdot B(q, D, C^0, C'). \end{aligned} \tag{2}$$

Note that in practice, index advisors are smart enough to create candidate index configurations that bring positive benefits during the index selection process. The objective of the index advisor is to find the optimal candidate index configuration C^* that can maximize the overall cost reduction. Since $Cost(q, D, C^0)$ in Equation 1 is fixed, which can be obtained from query history or database optimizer, index benefit $B(\cdot)$ becomes crucial to $R^W(D, C^0, C')$. As a result, it is important to estimate the index benefit accurately and efficiently. Inspired by [12, 25], we focus on *relative* cost metric to characterize index benefit in this paper due to its robustness to small cost estimation errors. Then, our objective is to build a learned model to estimate index benefit. To be specific, we define the *Index Benefit Estimation (IBE)* problem as follows.

Problem Statement (Index Benefit Estimation). *Given a database $D = (A, S)$, we have a dataset where each sample contains a query q, an initial index configuration C^0, a candidate index configuration C', as well as the correponding index benefit $B(q, D, C^0, C')$. All the queries in the dataset constitute the workload W. The objective of Index Benefit Estimation problem is to learn a regression model $\mathcal{M}$ from the dataset that can minimizes the overall squared error between the actual index benefit $B(q, D, C^0, C')$ and the estimated one $B'(q, D, C^0, C')$, i.e., $\sum_{q \in W} (B(q, D, C^0, C') - B'(q, D, C^0, C'))^2$. Furthermore, it is desirable that $\mathcal{M}$ can be generalized to unseen prediction scenarios where query workload, table schema and data statistics will diverge from the ones in the training phase.*

Let us denote the new query workload and the new database in the prediction phase by W' and D', respectively. Note that the IBE problem has dual objectives of accuracy and generalizability. Subsequently, we will explain why existing approaches are not sufficient to address the problem through a motivating example.

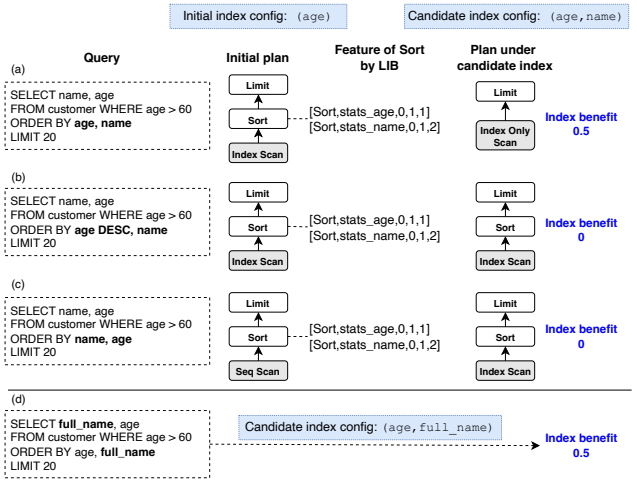

Figure 1: Motivating example: (a) candidate index can serve filter, sort and projection together, bringing significant index benefit (0.5); (b) candidate index can serve filter only due to columns in ORDER BY sorted in different directions (DESC/ASC); (c) candidate index can serve filter only due to mismatched column order in ORDER BY and candidate index; (d) renaming a column should have no impact on index benefit.

2.3 Motivating Example

In this subsection, we will explain the importance of comprehensiveness, position-awareness and drift-tolerance mentioned in the Introduction and point out the drawbacks of existing work through a concrete example. Suppose there is a table named *customer* with three columns: *c_id*, *name*, and *age*. Initially there is an index on the table, i.e., (*age*), and the candidate index configuration generated by the index advisor contains only one index, i.e., (*age,name*). There are four queries, (a), (b), (c), and (d) as shown in Figure 1, which share similar structure but with small variations in the column order or name.

For query (a), the candidate index (*name,age*) becomes a *covering index* as it includes all the columns that the query needs to return as results. Meanwhile, the index can serve ORDER BY and WHERE conditions simultaneously, as the WHERE condition only references the leading columns in the index and index data are sorted in the same column order as the ORDER BY clause. Overall, the index can significantly accelerate the query, bringing large index benefit (e.g., 0.5). We can see from the figure that *Index Scan* in the initial plan will change to *Index Only Scan* under the candidate index configuration. To extract the index benefit-related features, LIB [25] proposed a concept called the Index Optimizable (IO) operation. For a specific node like *Sort*, LIB will generate two IO operations (shown in the middle of the figure), [*Sort,stats_of_age,0,1,1*] and [*Sort,stats_of_name,0,1,2*], each corresponding to a column in the candidate index. Note that an IO operation consists of three parts: operation information (e.g., node type), index column statistics (e.g., number of rows, ratio of distinct values), and index configuration information (e.g., index type and column order in the index).

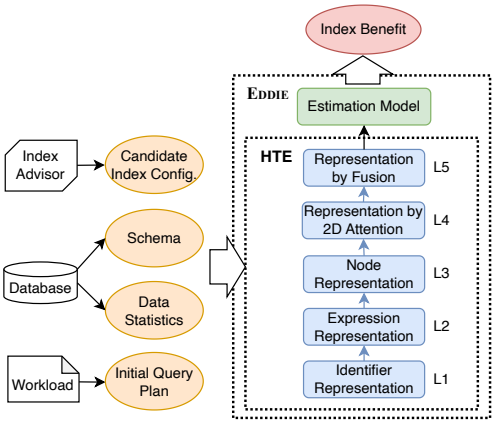

Figure 2: Overview of Eddie for index benefit estimation.

Specifically, the third and second last bits [*0,1*] are a one-hot encoding representing the multi-attribute index type, and the last bit indicates the column position in the index. Unfortunately, several important features are neglected in LIB, such as the sorting direction (ASC/DESC), column positions in the *Sort* node, as well as the tree structure of the query plan. Moreover, LIB assumes empty initial index, which generally does not hold in real world databases.

In the case of query (b), we adjust the sorting direction for *age*, causing the candidate index unable to serve the ORDER BY clause. Initially, the conditional lookup (i.e., the WHERE clause) can utilize the initial index, generating a *Index Scan* node in the query plan. However, after applying the candidate index, the query plan does not change, resulting in a zero index benefit. It implies that a thorough capture of all the influential factors (i.e., the comprehensiveness requirement) is crucial to IBE. Unfortunately, LIB is unable to discern the difference between query (a) and (b), and the generated IO operations in this case is the same as in query (a). A similar issue occurs in query (c). When we permute the order of *age* and *name* in the ORDER BY clause, the candidate index cannot be used for sorting these columns. The index benefit consequently remains zero in this case. Again, LIB failed to recognize such change when generating IO operations, because it only records the positions for columns in the candidate index (violating the position-awareness requirement).

Lastly, we mimic the schema change by renaming column *name* to *full_name* in query (d). In this case, the index benefit is supposed to be the same as query (a). However, most of existing query plan representation approaches [43], such as AVGDL [41] and QueryFormer [42], are dependent on column names. As a result, query (a) and (d) will be encoded into different vector representations (violating the drift-tolerance requirement), resulting in discrepancy of index benefit. Moreover, they lack the representation of indexes and thus can not directly apply for the IBE problem.

3 OVERVIEW

Eddie is an ML-based index benefit estimator invoked by index advisors to predict the benefit of a candidate index configuration for a specific query. As shown in Figure 2, Eddie consists of two main

components: *hierarchical and two-dimensional encoding (HTE)* and *estimation model*. Given a triplet $\langle D, C', q\rangle$, where D is the target database, C' is a candidate index configuration to evaluate, and q is a query on the database, HTE is responsible for extracting features from the following four inputs and outputs a condensed representation R: 1) **Initial query plan**. The physical query plan P_q for q under the initial indexes. Typically, it can be obtained by running the EXPLAIN statement on database D. 2) **Table schema**. Schema information from database D, such as column data type, column-table relationship, etc. 3) **Data statistics**. Data statistics of the columns referenced by query q. 4) **Candidate index configuration**. It contains a set of candidate indexes $\{I\}$. Then, the final representation is fed into the estimation model, which is essentially a multi-layer perceptron (MLP). Finally, the estimation model will generate an index benefit value between 0 and 1.

The main objective of HTE is to encode the entire query plan P_q into a vector representation in a hierarchical manner. Meanwhile, the table schema, data statistics, and candidate index configuration are treated as auxiliary information to augment the representation of the query plan during the encoding process. In this way, the impacts of the information on index benefit can be expressed. For example, to encode a column appearing in P_q, both the column's position in the candidate index and its corresponding data statistics are utilized to generate the column's representation. In fact, the hierarchical approach of building the query plan representation is inspired by the top-down approach of designing a SQL grammar with ANTLR [21], that is, language structures are constructed level by level from the coarsest to the finest. Specifically, HTE progressively constructs the final representation of a query plan through five levels, which are listed below from the bottom to the top.

- **L1 – identifier representation**, which focuses on encoding column identifiers appearing in P_q. For each column, we generate an embedding based on its position in candidate indexes, rather than based on column name, and additionally combine column schema and data statistics information into the column's representation.
- **L2 – expression representation**, which encodes SQL expressions appearing in the query, e.g., predicates. For SQL expressions where column order matters, such as the ordered column list in Sort node, we intentionally keep such order when encoding these expressions.
- **L3 – node representation**, which encodes typical features for single nodes in the query plan, including the node type, execution cost, output columns, and SQL expression in the node.
- **L4 – plan-dimension and index-dimension attentions**, which leverage the self-attention mechanism of Transformer [32] to capture the relations of different nodes from the perspectives of plan tree structure and the interaction of the candidate indexes in C' [24], respectively.
- **L5 – final representation**, which are produced by fusing the outputs of the previous two-dimensional attentions by a pooling mechanism.

The design of HTE has several advantages. First, it enables the reuse of representations of low-level elements to encode the high-level ones, and decomposes the complex work of constructing a representation for multiple inputs into a sequence of simple and manageable steps. Second, it achieves comprehensiveness by recognizing the index benefit-related features as much as possible and encoding them at the suitable levels. Third, we respect the importance of column positions and successfully encode the column position information regarding both the candidate index and the query plan node. Fourth, the position-based column identifier also helps overcome the drawback of name-based embedding methods, which are sensitive to schema changes. As a result, HTE is schema-agnostic and enables the application of transfer knowledge learned from other databases and workloads to new IBE tasks.

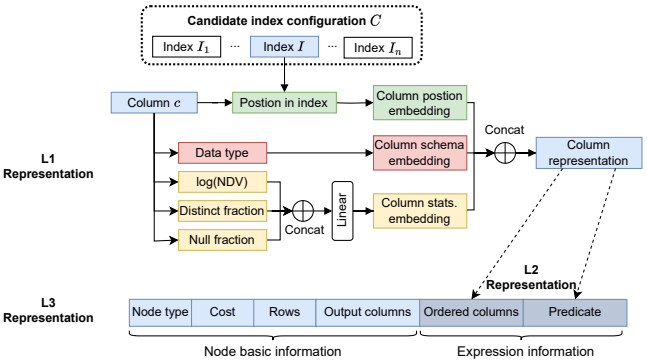

Figure 3: Column representation and node representation with respect to a candidate index

4 HIERARCHICAL AND TWO-DIMENSIONAL ENCODING

In this section, we introduce the detailed encoding method for each of the representation levels in HTE. For the i-th level, we denote the representation of corresponding entity by R_i.

4.1 L1: Identifier

In practice, index advisor tools, such as DTA [7] and AutoAdmin [8], support a number of constraints that users can set for their index tuning tasks, such as index storage budget, maximum number of recommended indexes, maximum size of the multi-column indexes, etc. Let m be the maximum number of indexes within each candidate index configuration, and k be the maximum number of columns permitted in a candidate index. In the identifier level, we focus on encoding the columns referenced by a query with respect to each of the m indexes in the candidate index configuration. Properly encoding columns is crucial, because a candidate index will cause the initial query plan to change, e.g., turn *Seq Scan* to *Index Scan*, only if some columns in the initial query plan and a candidate index overlap and appear in matching order. Otherwise, the benefit of the candidate index to the query becomes zero.

For a column c, HTE produces a column embedding with size d_c for each candidate index $I \in C'$. Overall, an index configuration with a maximum of m candidate indexes result in the column's representation $R_1^c \in \mathbb{R}^{m \times d_c}$. As shown in the upper diagram of Figure 3, a column's embedding is basically the concatenation of three parts: *position-based column embedding*, *column schema embedding* and *column data statistics embedding*.

The first part is a learned embedding for the column based on its position in index I. Let $Pos(c, I)$ be the c's positional relation with index I.

$$Pos(c, I) = \begin{cases} P_I^c & \text{if } c \in I, \\ \text{MIS\_COL} & \text{if } c \notin I \wedge t^c == t^I, \\ \text{MIS\_TBL} & \text{if } c \notin I \wedge t^c != t^I, \end{cases} \quad (3)$$

where P_I^c is the position of column c in index I, t^c is the column's table, and t^I is the index's table. MIS_COL and MIS_TBL are two special tokens that denote whether column c and I belong to a same table and two separate tables, respectively. Finally, the position-based column embedding is $Embed(Pos(c, I))$, where $Embed(\cdot)$ is a learnable embedding function. For example, column *age* is referenced by query (a) in Figure 1, and its embedding is $Embed(2)$ corresponding to index (*name,age*), where 2 is *age*'s position in the index.

In addition to column positions, column schema information and data statistics are also useful to IBE. For example, column data distribution will affect the selectivity of a predicate and further lead query optimizers to choose different data scan methods. To this end, the second part of the column embedding involves the encoding of column data type; the third part concatenates the embeddings of multiple data statistics, including the number of distinct values (NDV), faction of distinct values, and fraction of NULLs.

Overall, this identifier-level encoding brings two advantages: schema-agnostic and position-awareness. None of the embeddings in the column representation depend on the column name. Meanwhile, column position information regarding the candidate index is preserved. In this way, two columns with different names but having the same position in the index correspond to the same embedding. As a result, our method can be more generalizable because it facilitates transferring knowledge learned from other training data with different schemas. Also note that one column can yield multiple column representations, each corresponding to an index in the candidate index configuration.

4.2 L2: Expression

There exist dozens of logical SQL operators in standard SQL specification. However, only some of them can be optimized by indexes, e.g., *Scan*, *Join*, *Aggregate* and *Sort*. In expression-level encoding, we aim to describe the expressions appearing in these SQL operators. The expressions in *Scan* and *Join* are basically predicates that evaluate to be true or false, while the expressions in *Aggregate* and *Sort* are position-sensitive columns. That is to say, the position of a column in *Aggregate* and *Sort* largely determines whether an index can optimize the operators. In the following, we describe the encoding methods for these two types of expressions.

4.2.1 Predicate. Predicates can be either *atomic* or *compound*. An atomic predicate, like $age > 10$, typically compares a column with a value or another column, and is often expressed in the form of a triplet, e.g., $\langle column, comparison\ operator, value \rangle$. A compound predicate combines one or multiple atomic predicates by logical operators, such as AND/OR/NOT.

An atomic predicate is encoded by concatenating the three elements of the corresponding triplet, followed by the selectivity of the predicate. The resultant embedding of the atomic predicate p is $E_2^p \in \mathbb{R}^{m \times d_p}$, where d_p is the predicate embedding size. Selectivity is utilized because it plays an important role in determining whether a conditional table scan will utilize an index. More importantly, selectivity is between 0 and 1, and thus a more stable input compared with the comparison *value* of the predicate. A wide range of database engines have mature cost-based optimizers to compute selectivity of a predicate.

However, most of the existing predicate encoding methods are only able to deal with atomic predicates [43]. In this paper, we further support compound predicates. Our insight is that compound predicates can be expressed in tree structure where each node is either an atomic predicate or a logical operator. Thus, we are able to leverage a tree-based attention mechanism, similar to the approach in [42], to aggregate the information of the entire tree. For compound predicate p, its predicate tree is first flattened into a sequence of tree node embeddings, where the previous method is reused to encode atomic predicates and one-hot encoding is adopted to deal with logical operators. Additionally, we introduce an auxiliary node p_s, called *super token*, with learnable feature embedding, and append it to the head of the sequence. Unlike other nodes, super token is connected with all the other nodes. Then, by applying the tree-structured Transformer on the node embedding sequence, the output vector of the super token, denoted by $E_2^{p_s} \in \mathbb{R}^{m \times d_p}$, will represent the entire compound predicate, that is, $R_2^p = E_2^{p_s}$.

4.2.2 Ordered Column List. *Aggregate* and *Sort* operators may contain an ordered list of columns. For example, a company's total revenue can be grouped by a sequence of dimensions, like region and product type; employees can be sorted first by the age and then by the salary. The representation of an ordered list of columns l, denoted as R_2^l, consists of a list of column representations $\{R_1^c\}$ and a consistency value, where column $c \in l$. Consistency is a boolean value that indicates whether the sorting directions (ASC/DESC) of overlapping columns between ordering keys and indexed columns are exactly aligned. For *Aggregate* operators where the sorting direction is not applicable, the consistency value is always set to 1. To maintain the position information of the columns, the column representations in R_2^l are concatenated in the same order as in the list l. In this way, HTE can consolidate the column position information from both candidate indexes and *Aggregate*/*Sort* operators together. On one hand, the column position regarding candidate indexes is captured by R_1^c. On the other hand, column positions within SQL operators are preserved in R_2^l.

4.3 L3: Plan Node

Now we start to describe how HTE encodes the nodes in a query plan. A plan node is composed of a group of basic node information along with the expression representation, as shown in the lower part of Figure 3. There are four types of basic information involved in node encoding, i.e., node type, cost, output rows, and output columns, which are explained below.

(1) **Node type**. It specifies the node's physical operator type, such as *Indexed Scan* and *Merge Join*. The number of node types for a specific database is finite, and thus node type can be expressed by a categorical variable.

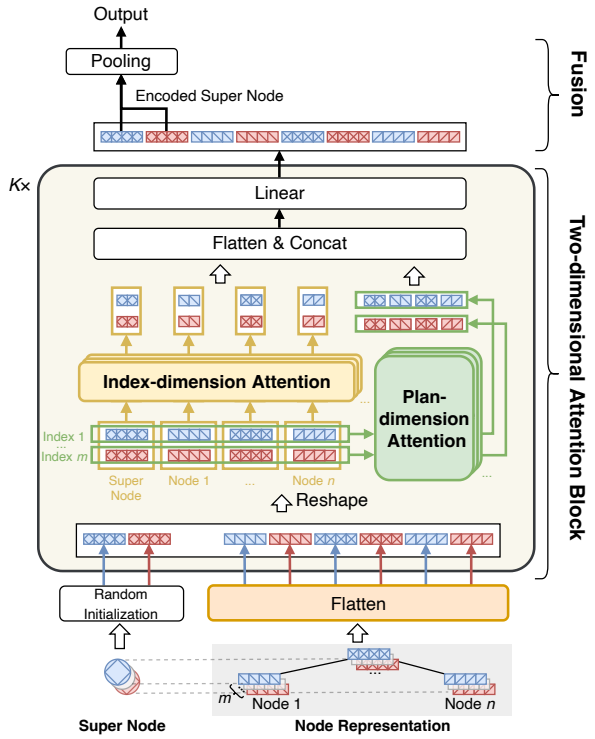

Figure 4: Two-dimensional attention blocks followed by a fusion layer.

(2) **Node cost**. It is the optimizer's estimated cost for the node. For some databases like PostgreSQL, the default cost value in the `EXPLAIN` output additionally includes the cost of all its child nodes. In this case, we can exclude the children to determine a single parent node's cost. The start-up cost, however, is not closely related to index recommendation and thus ignored.
(3) **Output rows**. It denotes the estimated number of rows output by the node.
(4) **Output columns**. If a node's output contains some columns that do not appear in the search condition, it is possible to create a *covering index* with both those columns and the search key to speed up the query. In order to recognize such case, it is crucial to encode the columns output by the node. We leverage the L1 representation to describe those columns.

Regarding the expressions in the plan node, we extract the ordered column list and predicates, and then adopt the L2 representation method to encode them. The resultant representations are concatenated together to produce R_3 as shown in Figure 3. For a plan node x, we have its representation: $R_3 \in \mathbb{R}^{m \times d_x}$, where d_x is the size of the embedding for node x with respect to a candidate index.

4.4 L4: Two-dimensional Attention

Given a query plan $P_q = (N, E)$, where N and E represent the node set and edge set, respectively, we can now obtain a sequence of node representations $\{R_3^x\}$ for the query plan based on the previous L3 method. Recall that $R_3^x \in \mathbb{R}^{m \times d_x}$, then the node representation sequence is of size $n \times m \times d_x$, where $n = |N|$ is the size of the node set. In order to gather information from the entire query plan tree, we add an assistive super node into the sequence fully connected with all other nodes, increasing the sequence size to $(n+1) \times m \times d_x$.

To create a good representation for such a complex sequence, we innovatively propose a two-dimensional attention mechanism (see Figure 4). In the first dimension, which we call the *plan-dimension*, we leverage tree-structured attention mechanism in QueryFormer [42] to encode $n+1$ nodes based on the query plan tree, yielding a plan-dimension representation of size $(n+1) \times m \times (d_x/2)$. The second dimension, which we call the *index-dimension*, aims to capture the interaction among indexes [24]. Again, we leverage the attention mechanism to encode the representations for m candidate indexes. The result is an index-dimension representation of size $(n+1) \times m \times (d_x/2)$. Through concatenation of the two results from the attention mechanism, the output restores the size of $(n+1) \times m \times d_x$, and is forwarded to a linear layer. Overall, the two-dimensional attention block is an IBE-specialized variant of the well-known Transformer block [32]. Moreover, HTE stacks K of such blocks, where K is a tunable parameter depending on the computation power. The output of the K blocks is the L4 representation, denoted by $R_4 \in \mathbb{R}^{(n+1) \times m \times d_x}$.

4.5 L5: Fusion

Finally, we pick the super node's output from L4 representation, denoted by $R_4^s \in \mathbb{R}^{m \times d_x}$, and apply a pooling layer on it to fuse the node's information, that is,

$$R_5 = AvgPool(R_4^{x_s}), \tag{4}$$

where $AvgPool(\cdot)$ is the average pooling function over the node's m embeddings (each with size d_x) and $R_5 \in \mathbb{R}^{d_x}$ is the final condensed representation for all the input in Figure 2, i.e., $R = R_5$.

5 TRAINING AND IMPLEMENTATION

5.1 Estimation Model and and Training

5.1.1 Training. Now that HTE helps to generate a useful representation for all the inputs, Eddie predicts index benefit through an estimation model which is a multi-layer perceptron (MLP). The predicted index benefit is:

$$\hat{B} = MLP(R_5), \tag{5}$$

where $MLP(\cdot)$ is a multi-layer perceptron with the residual connection. Let B be the corresponding ground-truth index benefit. The training loss is the mean squared error (MSE) between the predicted and ground-truth index benefits:

$$L = MSE(\hat{B}, B). \tag{6}$$

To train the model for Eddie, we need to collect training data in the form of ⟨query text, initial query plan, query-referenced table schema, query-referenced column statistics, candidate index, index benefit value⟩. To collect enough data for training, there are multiple feasible and safe solutions in practice, e.g., replicate an extra databasem, modify indexes and replay the application SQL to obtain the index benefit by comparing the execution time before and after the index modification. For analytic only queries, it is also

viable to execute them on the read-only replica of databases, thus posing minumum impact on user business. Instead of physically creating those indexes and running the SQL, we can also leverage hypothetical indexes and obtain estimated execution cost by running `EXPLAIN` statements. The latter is more scalable, although not as accurate as the former approaches.

5.1.2 Pre-train and Fine-tune. A notable merit of our approach is that it enables transfer learning, as HTE creates transferable feature representations. Given a pre-trained Eddie model $\mathcal{M}'$, we can fine-tune it based on training data collected from the target database, yielding a fine-tuned model $\mathcal{M}$. Typically, fine-tuning requires significantly less training data, thus saving considerable cost for data collection. As a cloud service provider who operates a large number of databases on the cloud, it is feasible for us to accumulate a large amount of data with sufficient diversity to pre-train a foundation model, and then easily adapt to downstream index tuning tasks. We leave the large-scale pre-training for future work. only demonstrate its effectiveness by benchmark datasets in the evaluation section.

5.2 Integration with Index Tunner

During the index selection process, index tuners, such as DTA and AutoAdmin, rely on the estimated cost of the query execution plan under the candidate index configuration. Since Eddie is only able to provide an index benefit, which is the relative performance improvement, we have to convert the benefit value back to a predicted cost for the tuners to use. As defined in Equation 1, the index benefit $B(q, D, C^0, C')$ is the cost reduction ratio between the initial index configurations C^0 and the candidate index configurations C'. We can determine the execution cost under the candidate index configuration by the following formula:

$$Cost(q, D, C') = Cost(q, D, C^0) \times (1 - B(q, D, C^0, C')). \quad (7)$$

Here, the original cost $Cost(q, D, C')$ can be obtained by running the query on the database or from historical query log.

6 EVALUATION

6.1 Experimental Setup

Our experiments differ from existing work in three important ways. First, most of existing work experiments on benchmark workloads, like TPC-DS and TPC-H, which are based on fixed SQL templates and only cover specific application scenarios. But real-world workloads are dynamic and SQL are much more diversified. To evaluate an IBE method on more general scenarios, we extend the original benchmark workloads by generating new SQL in a probabilistic approach. Second, we consider both empty and non-empty initial index scenarios. The latter is more general in real world. Third, we simulate environmental changes, such as workload and schema drift, as well as data updates. These changes are common in real-world applications, where both the queries being executed and the underlying data can evolve over time, causing variation on the estimation quality.

6.1.1 Workloads. We construct two sets of workloads:

- **Benchmark workload**: We employed standard benchmark datasets, including TPC-DS, TPC-H with a scale factor (SF) of 10, and IMDB, whose queries are generated via predefined templates. Following prior practices [15], we excluded templates with disproportionately high execution costs, as they would dominate workload costs and skew index selection. Specifically, for TPC-DS, we used 78 out of 99 templates, generating 5 queries per template, resulting in 390 queries. For TPC-H, we used 14 out of 21 templates, with 80 queries per template, yielding 1,120 queries. Regarding IMDB, 80 out of 113 templates are used and 3 queries are generated per template, yielding 240 queries in total.
- **Extended workload**: The extended versions of the above workloads, denoted by TPC-DS+, TPC-H+ and IMDB+, respectively. These extended workloads consist of both the original SQL queries and newly generated random SQL queries. The generation process involves combining different SQL components, such as JOIN, WHERE, GROUP BY, ORDER BY, and LIMIT, in a probabilistic and recursive manner (code is open sourced). This approach enriches the original benchmark queries, allowing us to more thoroughly evaluate the index benefit estimation methods.

Table 1: Summary of Datasets

Dataset	# Queries			# Cases		
	Templated	Synthetic	Total	w/o Initial	w/ Initial	Total
TPC-DS	390	0	390	4910	22118	27028
TPC-DS+	390	3000	3390	8378	22664	31042
TPC-H	1120	0	1120	2400	2240	4640
TPC-H+	1120	3000	3120	5888	2830	8718
IMDB	240	0	240	6240	23900	30140
IMDB+	240	3000	3240	9858	24516	34374

6.1.2 Method of generating index configurations. For each query q in the workload without any initial indexes, we use AutoAdmin [8] to generate the optimal configuration, denoted by C^*. The maximum number of indexes in the candidate configuration is 5, where each index contains at most 2 attributes. Then, we find all the non-empty subsets of C^*, that is,

$$\mathbb{N}(C^*) = \{C | C \subseteq C^* \wedge C \neq \emptyset\}.$$

Let k be the number of indexes in the optimal configuration, then $|\mathbb{N}(C^*)| = 2^k - 1$. We also introduced a *superset* function for an index configuration C, defined as:

$$\mathbb{S}(C) = \{C' | C' \subseteq C^* \wedge C \subset C'\},$$

As an example, suppose $C^* = \{(c1, c2), (c3)\}$, then $\mathbb{N}(C^*) = \{\{(c1, c2)\}, \{(c3)\}, \{(c1, c2), (c3)\}\}$. Given an index configuration $C = \{(c1, c2)\}$, its superset $\mathbb{S}(C) = \{\{(c1, c2), (c3)\}\}$. By enumerating the non-empty subsets and their supersets, we are able to spawn a great number of index configurations for each query.

Besides, by combining empty/non-empty initial configurations, we are able to create two meaningful scenarios for each workload:

- **w/o Initial**: Initial configuration is empty, and the non-empty subsets $\mathbb{N}(C^*)$ serve as the candidate configurations.
- **w/ Initial**: The non-empty subsets $\mathbb{N}(C^*)$ serve as the initial configurations, while the union of all the initial configuration's supersets, i.e., $\bigcup_{C \subseteq \mathbb{N}(C^*)} \mathbb{S}(C)$, constitutes the candidate configurations.

Table 2: Performance with and without Initial Index Configurations

Model	TPC-DS		TPC-DS+		TPC-H		TPC-H+		IMDB		IMDB+	
	Mean	95th	Mean	95th	Mean	95th	Mean	95th	Mean	95th	Mean	95th
Without Initial Index Configurations												
PostgreSQL	466.72	3399.46	333.32	1921.47	265.79	1567.43	187.41	408.12	953.08	7199.56	703.39	6155.24
LIB	97.85	383.69	169.13	443.26	32.87	107.21	160.79	193.65	503.97	2482.43	435.79	2394.13
QueryFormer-I	88.81	255.11	132.68	274.68	29.37	168.49	162.39	237.84	287.49	1936.70	326.25	2065.60
Eddie	**45.84**	**118.65**	**78.83**	**98.99**	**17.82**	**86.56**	**117.02**	**69.77**	**242.12**	**1216.18**	**260.87**	**1353.50**
With Initial Index Configurations												
PostgreSQL	705.73	5029.87	629.26	3890.31	349.45	2503.55	282.71	1982.32	756.04	6569.13	1299.53	8675.51
LIB	66.55	194.06	99.89	313.40	22.28	71.14	155.29	186.86	321.24	1828.99	373.59	2293.92
QueryFormer-I	78.12	313.43	130.09	471.37	20.98	129.08	161.59	163.43	288.50	1838.43	365.99	2439.44
Eddie	**31.28**	**76.37**	**56.73**	**96.75**	**14.92**	**67.44**	**109.62**	**90.06**	**153.13**	**547.03**	**206.94**	**949.00**

For each query in the workload, we materialize the initial configurations on the tables, and run the query by EXPLAIN ANALYZE command before and after applying the candidate configuration. The results of EXPLAIN ANALYZE not only show the query plan, but also the real execution time. To avoid the impact of cache miss, we execute the query 4 times. The first run intends to warm up the buffer pool, while the average execution time of the last three runs are used to calculte index benefits. In this way, we can all the information needed for each training sample (see Section 5.1.1). The characteristics of resulting dataset are summarized in Table 1, detailing query and case counts across workloads.

6.1.3 Baselines. We compared Eddie against three baselines to evaluate its performance in index benefit estimation:

- **PostgreSQL**. As most existing index tuners rely on optimizer-based cost estimation, we compare Eddie against PostgreSQL 12.13's cost estimator. Specifically, we utilized HypoPG 1.3.1 [1] to create virtual indexes, invoked the optimizer to compute query costs before and after index creation, and calculated the estimated index benefit from these differences.
- **LIB [25]**. A state-of-the-art attention-based model for index benefit prediction.
- **QueryFormer-I**. QueryFormer [42] is the state-of-the-art query representation method with considering index configuration. To enhance it, we mix an index encoder from ChangeFormer [40] to make it capable of predicting index benefit. The resultant model is named QueryFormer-I. Specifically, the output of ChangeFormer's index encoder and QueryFormer's plan representation are concatenated into the final representation vector, and then serve as input to a two-layer fully connected neural network with Sigmoid function.

6.1.4 Model training. We have trained the Eddie model with the Adam optimizer [14] using the decaying learning rate. Under our hardware condition, we set K to 4 and the number of heads in plan-/index-dimension attention to 4. The learning rate starts at 1e-4 and decays by 0.7 every 20 steps. We conduct five-fold cross validation by using four folds as the training set and one fold as the test set. With a batch size of 16, each training runs for 100 epochs.

6.1.5 Metrics. We use Q-error [29] to evaluate the accuracy of various methods in index benefit estimation. Q-error measures the difference between the predicted and ground-truth index benefits by ratio. For a sample in the test dataset, let b_{pred} and b_{act} be the predicted and ground-truth index benefits, respectively. The Q-error of the a test sample is $max(\frac{b_{pred}+\epsilon}{b_{act}+\epsilon}, \frac{b_{act}+\epsilon}{b_{pred}+\epsilon})$, where ϵ is a very small constant for error correction, i.e., $1e-4$. The mean Q-error measures the average estimation performance on the overall workload, and the 95-th percentile Q-error implies the estimation performance in some worst cases.

6.1.6 Environment. Experiments were conducted on a Linux server equipped with an Intel 13th Gen i7-13700K CPU, an NVIDIA GeForce RTX 4080 GPU, and 64 GB of RAM. The DBMS used was PostgreSQL 12.13.

6.2 Prediction Accuracy

First, we examine the accuracy of the predicted index benefits by Eddie and the baseline methods in the setting with or without initial index configurations. The results are shown in Table 2.

In the setting without initial index configurations, Eddie outperforms all baseline methods under all workloads. For example, in TPC-DS, Eddie achieves a mean Q-Error of 45.84, which is 53.2% and 48.4% lower than LIB and QueryFormer-I, respectively. Looking at the 95-th percentile Q-error, Eddie also demonstrates significant improvement over the baseline methods. Such advantages are mainly due to the comprehensiveness objective that Eddie aims to achieve by encoding more sufficient features to tolerate index benefit estimation noise for the outlying cases. The accuracy of PostgreSQL's optimizer is the worst under all workloads, which aligns with the findings in [44] and implies that database optimizer's cost estimation is often untrustworthy.

In the setting with initial index configurations, Eddie demonstrates the superiority over other methods in all the workloads again. By comparing the each method's performance in both settings, an important observation is that though the Q-errors of the baseline methods is not stable between the two settings, Eddie consistently generates significant lower Q-errors in the setting with initial index configurations. In real world scenarios where databases rarely run without initial indexes, Eddie has paramount practical value for index recommendation.

Table 3: Performance Comparison under Drift Scenarios

Model	TPC-DS		TPC-DS+		TPC-H		TPC-H+		IMDB		IMDB+	
	Mean	95th	Mean	95th	Mean	95th	Mean	95th	Mean	95th	Mean	95th
Query Variations												
PostgreSQL	669.46	4915.04	920.89	6624.34	787.67	7604.96	653.26	6855.49	2012.57	8889.16	1664.16	9158.08
LIB	203.74	1089.05	361.19	2158.61	30.04	85.72	248.58	263.61	505.33	3580.07	544.70	3740.16
QueryFormer-I	390.87	2608.08	551.75	4227.41	24.37	171.09	287.82	537.06	650.40	4605.47	650.40	4605.47
Eddie	**192.72**	**912.36**	**272.48**	**1649.15**	**15.34**	**50.10**	**211.39**	**131.64**	**493.76**	**3543.77**	**473.97**	**3119.38**
Schema Changes												
PostgreSQL	705.73	5029.87	629.26	3890.31	349.45	2503.55	282.71	1982.32	756.04	6569.13	1299.53	8675.51
LIB	66.55	194.06	99.89	313.40	22.28	71.14	155.29	186.86	321.24	1828.99	373.59	2293.92
QueryFormer-I	221.91	1280.31	600.78	4073.96	36.52	136.48	201.81	163.84	474.83	3600.09	426.73	2779.24
Eddie	**31.28**	**76.37**	**56.73**	**96.75**	**14.92**	**67.44**	**109.62**	**90.06**	**153.13**	**547.03**	**206.94**	**949.00**
Data Volume Shifts												
PostgreSQL	874.56	5528.55	843.81	6226.73	496.33	2503.63	290.83	2346.11	N/A		N/A	
LIB	428.98	3472.33	484.68	3548.34	109.65	834.28	262.95	1547.30	N/A		N/A	
QueryFormer-I	390.87	2608.08	544.70	3910.15	177.32	1491.96	268.04	1193.22	N/A		N/A	
Eddie	**273.72**	**1847.36**	**287.68**	**2164.34**	**100.75**	**214.13**	**157.48**	**153.76**	N/A		N/A	

6.3 Drift-Tolerance

To assess the robustness of Eddie against environmental changes, we introduced three types of drifts (the query variation, schema change, data volume shift) exclusively to the test set, leaving the training set unchanged. This setup evaluates the trained model's resilience to real-world drift scenarios, simulating conditions where workloads evolve independently of the training data.

6.3.1 Query Variation. To examine Eddie's performance under query variations, we modified the test set by appending a randomly generated predicate to each query. Specifically, for each query, we selected a random column from the accessed table, paired it with a randomly chosen comparison operator (e.g., <, >, =), and assigned a value sampled from that column's data as the condition. This simulates query evolution in dynamic environments. Results in Table 3 show that Eddie consistently achieves the lowest mean and 95-th percentile Q-error across all datasets. For instance, on TPC-H, Eddie records a mean Q-error of 15.34 and 95th of 50.10, significantly outperforming QueryFormer-I (24.37, 171.09) and LIB (30.04, 85.72). This superior robustness demonstrates Eddie's ability to maintain accurate index benefit estimation under shifting query patterns, extending its effectiveness beyond static conditions.

6.3.2 Schema Change. To investigate resilience to schema changes, we randomly altered 20% of the column names accessed by queries in the test set. For example, a query like `SELECT a FROM A` was transformed to `SELECT a0 FROM A` by replacing column a with a0, mimicking real-world schema updates. As shown in Table 3, Eddie outperforms all baselines, achieving the lowest Q-error metrics across datasets. It is worth noting that compared with the results before the change (Table 2), Eddie and LIB have not changed, while QueryFormer-I's Q-error degraded significantly (e.g., from 88.81 and 255.11 to 221.91 and 1280.31 on TPC-DS). This disparity arises because QueryFormer-I encodes column by column names, rendering it sensitive to unseen names, whereas Eddie relies on schema-agnostic encodings. These results highlight Eddie's advantage in handling schema changes effectively.

6.3.3 Data Volume Shift. To assess the impact of data volume shifts, we adjusted the scale factors of TPC-DS and TPC-H, reducing the original 10 GB databases to 5 GB, and resampled all test set queries on these smaller datasets to create new test sets. This simulates real-world scenarios where data size fluctuates. Due to the absence of a scale function in IMDB, experiments were not conducted on this dataset. As reported in Table 3, Eddie consistently outperformed baselines across TPC-DS and TPC-H. For instance, on TPC-H+, Eddie achieved the mean and 95-th percentile Q-errors of 157.48 and 153.76, significantly lower as compared to LIB's 262.95 and 1547.30, and QueryFormer-I's 268.04 and 1193.22. These results demonstrate that Eddie adapts effectively to data volume shifts, sustaining superior estimation accuracy despite significant reductions in data size. Such property is mainly because Eddie considers data statistics and selectivity during its featurization process.

6.4 Position-Awareness

A key strength of Eddie lies in its column position-awareness, which we validate through index perturbation experiments. In this experiment, we add index perturbations to a set of datasets without initial index configuration to evaluate the ability of each method to perceive the order of the columns' positions in the index. The specific method of index perturbation is to perturb the index configuration of each sample (query, index configuration) in the dataset, and then add the new sample (query, perturbed index configuration) to the dataset. When perturbing an index configuration, we reverse the column order for each index in the index configuration, for example, an index with a two-column attribute (a,b) is perturbed to get the index (b,a).

The results are shown in Table 4. Compared with the dataset without index perturbation (refer to Table 2), the Q-errors of all method in TPC-DS, TPC-H, IMDB does not change much. However, it is important to note that in TPC-DS+, a dataset containing synthetic queries, the mean and 95th percentile Q-errors of LIB significantly degrade from (169.13, 443.26) to (445.73, 4832.06), while those of Eddie improve from (78.83, 98.99) to (73.70, 90.03). The case is the same in the other two datasets containing synthetic queries,

Table 4: Performance under Index Perturbation

Model	TPC-DS		TPC-DS+		TPC-H		TPC-H+		IMDB		IMDB+	
	Mean	95th	Mean	95th	Mean	95th	Mean	95th	Mean	95th	Mean	95th
PostgreSQL	514.92	4030.00	343.22	2158.96	213.84	742.84	378.47	2416.02	1101.21	7095.64	638.12	5502.59
LIB	75.82	317.92	445.73	4832.06	40.92	162.59	380.36	2992.72	490.51	2817.18	622.23	3898.80
QueryFormer-I	109.94	705.41	272.94	1038.08	32.17	196.18	147.98	163.07	301.05	2082.65	385.48	2385.63
Eddie	**32.64**	**104.80**	**73.70**	**90.03**	**32.15**	**143.66**	**107.86**	**93.16**	**235.08**	**1240.20**	**295.21**	**1907.15**

Table 5: Ablation Study of Eddie's Components

Model	TPC-DS		TPC-DS+		TPC-H		TPC-H+		IMDB		IMDB+	
	Mean	95th	Mean	95th	Mean	95th	Mean	95th	Mean	95th	Mean	95th
Eddie	**31.28**	**76.37**	**56.73**	**96.75**	**14.92**	**67.44**	**109.62**	**90.06**	**153.13**	**547.03**	**206.94**	**949.00**
Eddie w/o index attn	39.77	105.30	59.41	109.41	16.75	82.39	125.20	98.28	173.86	769.90	240.45	1311.22
Eddie w/o histgram feature	37.79	81.36	57.14	102.85	17.03	72.10	119.25	97.52	184.20	856.14	224.46	1092.93

TPC-H+ and IMDB+. The main reason for this experimental result is that there are fewer optimization opportunities for indexes in TPC-DS, TPC-H, and IMDB, which cannot reflect whether different methods are robust to index perturbation, while in TPC-DS+, TPC-H+, and IMDB+, there are more optimization cases related to index position. Meanwhile, we observe a significant deterioration of LIB performance because it neglects the importance of column position in SQL operators as exemplified in Section 2.3. Eddie's index-guided position encoding method cleverly associates the order of the column in the index and SQL operator, and achieves a position-awareness effect. Unsurprisingly, Eddie's Q-error performance before and after index perturbation is stable.

6.5 Ablation Study

6.5.1 Index-Dimension Attention. To evaluate the effectiveness of the two-dimensional attention mechanism, we remove the index-dimension attention component from the model by replacing the corresponding embedding with zeros while keeping the remaining architecture intact. As shown in Table 5, the removal of the index-dimension attention results in a noticeable degradation in performance across all datasets. For instance, on the TPC-DS workload, the 95th percentile Q-error increases from 76.37 to 105.30, and the mean Q-error increases from 31.28 to 39.77. This trend is consistent across all tested datasets, highlighting the utility of the index-dimension attention in achieving the model's accuracy.

6.5.2 Histogram Feature. In our featurization approach, histogram is used to calculate the selectivity value during the encoding of predicates. To evaluate the effectiveness of the histogram, we tested Eddie's performance after removing the histogram feature by replacing selectivity with 0. As shown in Table 5, in TPC-DS, the mean and 95-th percentile Q-errors before removing the histogram feature are (31.28, 76.37) and increase to (37.79, 81.36) after removal, showing a slight performance degradation. The same result persists in other datasets, indicating that the histogram feature is useful for improving the prediction accuracy of the model.

Table 6: Performance of Pre-training and Fine-tuning

Model	TPC-DS		TPC-H		IMDB	
	Mean	95th	Mean	95th	Mean	95th
From scratch (50% Data)	50.70	132.55	26.38	92.13	300.27	1652.83
From scratch (100% Data)	45.84	118.65	17.82	86.56	242.12	1216.18
Pre-trained (50% Data)	**44.82**	**105.35**	**14.02**	**79.78**	**222.01**	**1127.22**

6.6 Pre-training for Enhanced Accuracy

One of the key advantages of HTE is its ability to produce transferable feature representations, which enables the effective application of pre-training and fine-tuning techniques for index benefit estimation. This approach allows the model to generalize across different datasets and workloads, making it resilient to schema changes and data distribution shifts, a critical feature for dynamic and evolving database environments.

To demonstrate the effectiveness of this approach, we conducted an experiment where the model was pre-trained on a diverse set of datasets and then fine-tuned on the target workload. The pre-training datasets comprised 18 of the 20 datasets from [13], excluding TPC-H and IMDB to avoid overlap with the target workloads. For each dataset, 1000 queries were automatically generated using the complex mode workload generator from [13], and index configurations were generated as described in Section 6.1.2. The model was fine-tuned on only 50% of the training data from each target workload (TPC-DS, TPC-H, and IMDB), with performance evaluated on the test set as is conducted in Section 6.2. We compared this fine-tuned model with baseline models trained from scratch on 50% and 100% of the training data.

The results shown in Table 6 highlight the advantages of HTE's transferable feature representation. For example, on the TPC-DS workload, the pre-trained model fine-tuned on 50% data outperformed the baseline trained on 100% data, achieving the 95-th percentile Q-error of 105.35 and the mean of 44.82, compared to the baseline's 118.65 and 45.84. Similar improvements are observed on the TPC-H and IMDB workloads. These results highlight that the

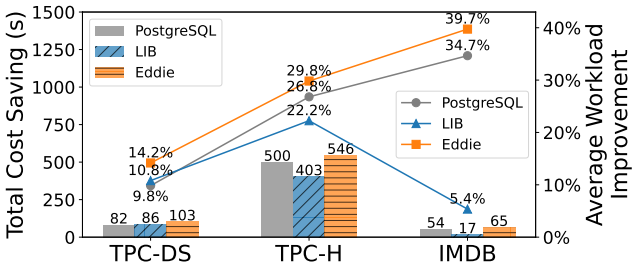

Figure 5: Integration with the Index Tuner

the knowledge learned from diverse datasets can enhance data efficiency and prediction accuracy, even when working with smaller amount of target-specific data.

6.7 Integration with the Index Tuner

We further explore the integration of Eddie with the index tuner, as described in Section 5.2. In this setup, we use AutoAdmin [8] as the index tuner, replace the cost estimation component with Eddie, and compare its performance to both PostgreSQL 12's built-in cost estimation (using "what-if" calls) and LIB.

The experiment was conducted using the original workloads from TPC-DS, TPC-H, and IMDB, with the training and testing datasets split by query and 5-fold cross-validation applied, consistent with Section 6.2. For the training set, candidate index configurations were generated using two strategies. First, we employed the optimal index configuration method described in Section 6.1.2. Second, we randomly combined remaining candidate indexes to create additional configurations. The ratio of optimally generated to randomly combined configurations was set to 1:2. During evaluation, the test set was divided into multiple workloads, each containing 5 queries. The end-to-end index tuner was then run on these workloads to evaluate the performance.

The results shown in Figure 5 clearly prove the superiority of Eddie. Eddie consistently outperforms PostgreSQL and LIB in both total cost savings and average workload improvement across all three workloads. For example, on the TPC-DS workload, our method achieves a total cost saving of 103 s and an average workload improvement of 14.2%, outperforming both PostgreSQL (82 s, 9.8%) and LIB (86 s, 10.8%). These results demonstrate that Eddie not only offers more accurate index benefit estimation but also achieves superior end-to-end performance in index recommendation tasks. Such performance gain is mainly due to Eddie's more accurate estimation method which helps the tuner to identify useful candidate indexes.

7 RELATED WORK

Query cost estimation: Over the past decades, there has been intensive work [5, 8, 11, 39] on the query optimizer relying on what-if calls. Earlier methods [15, 26, 35] typically develop heuristic cost models to estimate the query plan cost. Such heuristic cost models can generate inaccurate estimates due to modeling or statistical biases [17, 36], such as the well-known cardinality estimation bias based on static statistics [17, 18, 36]. To address the problem of these biases, recent methods usually apply learned models for cardinality estimation [30], query cost estimation [12, 34], or both [29]. However, these methods usually require frequent invoking of what-if calls, which is time-consuming. Recently, some work has focused on reducing the index optimization overhead by reducing the workload of what-if calls, e.g., by limiting the number of index configurations to reduce the number of calls [33, 37] or by filtering out components in the queries [4] or even queries themselves [27] if they are not expected to benefit from index tuning. Despite all these efforts, these methods cannot be applied to some popular database systems that do not provide what-if calls [22], such as MySQL, which is widely deployed in cloud services. Our method adopts the learned cost model but does not rely on what-if calls. Its deep learning approach can capture the knowledge of query-and-index runtime performance and avoid what-if calls.

Query plan representation: Some studies focus on learning query plan representations potentially applicable to multiple tasks, including index benefit estimation. They typically use graph-based [13] or tree-structured [19, 40, 42] neural networks to grasp the node and structural information of the query plan. Specifically, our method adopts a mechanism similar to QueryFormer [42] for capturing node dependencies. QueryFormer learns query plan representation by incorporating the tree structure information of a query plan in the Transformer model to capture node dependencies and address the issues of extended information flow paths in the query plan. The query plan representation can be combined with other index representation techniques, such as RIBE [40], to estimate index benefits. However, the separate encoding of query plan and index representations fails to highlight the meaningful influence of the index on certain query plan operators, such as Sort and Scan. Unlike QueryFormer and other methods that encode a query plan without indexes, our method integrates the index information into the query plan operators that the indexes will probably affect, and jointly considers the indexes and query plan as a whole for representation learning. In effect, our method is able to achieve higher estimation accuracy.

Query-index cost estimation: In contrast to the approach that purely estimates query cost, some work [25, 28, 45] estimates the cost for query and index pairs. Specifically, LIB [25] embeds the index information into the query plan node encoding, and proposes an abstraction of index-optimizable operations. However, LIB ignores several crucial factors, e.g., initial indexes, plan tree structure, column order in Sort resulting in poor estimation accuracy. Our method is inspired by LIB, but outperforms it in various aspects.

8 CONCLUSION AND FUTURE WORK

ML-based index advising encounters various accuracy and robustness issues, especially in scenarios with drifts in the query, schema, and data volume. In this paper, we develop a novel hierarchical and two-dimensional encoding method to generate comprehensive index-query-aligned representations for accurate index benefit estimation. Our method is demonstrated to be robust under diverse workload drift scenarios, has superior end-to-end performance, and has lower requirement on training data by leveraging pre-training and fine-tuning paradigm. In the future, we plan to collect a large volume of data to build a solid pre-trained model so that online index tuning can be eased and prevail.

REFERENCES

[1] [n.d.]. *HypoPG*. Retrieved Feb 20, 2025 from https://github.com/HypoPG/hypopg

[2] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* 35, 8 (2013), 1798–1828.

[3] Silvia Botros and Jeremy Tinley. 2021. *High Performance MySQL* (4th ed.). O'Reilly Media, Inc.

[4] Matteo Brucato, Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2024. Wred: Workload Reduction for Scalable Index Tuning. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[5] Nicolas Bruno and Surajit Chaudhuri. 2007. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 227–238.

[6] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-if" index analysis utility. *ACM SIGMOD Record* 27, 2 (1998), 367–378.

[7] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime algorithm of database tuning advisor for microsoft sql server.

[8] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *Proceedings of the VLDB Endowment*, Vol. 97. San Francisco, 146–155.

[9] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. 2004. Automatic SQL tuning in Oracle 10g. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 1098–1109.

[10] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically indexing millions of databases in microsoft azure sql database. In *Proceedings of the 2019 International Conference on Management of Data*. 666–679.

[11] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: a scalable, portable, and interactive index advisor for large workloads. *Proceedings of the VLDB Endowment* 4, 6 (2011), 362–372.

[12] 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. In *Proceedings of the 2019 International Conference on Management of Data*. 1241–1258.

[13] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2361–2374.

[14] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR*.

[15] 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. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2382–2395.

[16] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An index advisor using deep reinforcement learning. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 2105–2108.

[17] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[18] Feng Liang, Francis C.M. Lau, Heming Cui, Yupeng Li, Bing Lin, Chengming Li, and Xiping Hu. 2024. RelJoin: Relative-cost-based selection of distributed join methods for query plan optimization. *Information Sciences* 658 (2024), 120022.

[19] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1733–1746.

[20] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. 2007. Efficient use of the query optimizer for automated physical design. In *Proceedings of the 33rd international conference on very large data bases*. 1093–1104.

[21] Terence Parr. 2013. *The definitive ANTLR 4 reference* (2nd ed.). The Pragmatic Bookshelf.

[22] Gan Peng, Peng Cai, Kaikai Ye, Kai Li, Jinlong Cai, Yufeng Shen, Han Su, and Weiyuan Xu. 2024. Online Index Recommendation for Slow Queries. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 5294–5306.

[23] Gregory Piatetsky-Shapiro. 1983. The Optimal Selection of Secondary Indices is NP-Complete. *SIGMOD Rec.* 13, 2 (1983), 72–75.

[24] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. 2009. Index interactions in physical design tuning: modeling, analysis, and applications. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1234–1245.

[25] Jiachen Shi, Gao Cong, and Xiao-Li Li. 2022. Learned index benefits: Machine learning based index performance estimation. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3950–3962.

[26] 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 ACM SIGMOD International Conference on Management of Data*. 99–113.

[27] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently compressing large and complex workloads for scalable index tuning. In *Proceedings of the 2022 International Conference on Management of Data*. 660–673.

[28] Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2022. DISTILL: low-overhead data-driven techniques for filtering and costing indexes for scalable index tuning. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2019–2031.

[29] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment* 13, 3 (2019), 307–319.

[30] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: A design space exploration and a comparative evaluation. *Proceedings of the VLDB Endowment* 15, 1 (2021), 85–97.

[31] Gary Valentin, Michael Zuliani, Daniel C Zilio, Guy Lohman, and Alan Skelley. 2000. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of 16th International Conference on Data Engineering (ICDE)*. IEEE, 101–110.

[32] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[33] Xiaoying Wang, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2024. Wii: Dynamic Budget Reallocation In Index Tuning. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[34] Wentao Wu. 2025. Hybrid Cost Modeling for Reducing Query Performance Regression in Index Tuning. *IEEE Transactions on Knowledge and Data Engineering* 37, 1 (2025), 379–391.

[35] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. 2013. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment* 6, 10 (2013), 925–936.

[36] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 1081–1092.

[37] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A Bernstein. 2022. Budget-aware index tuning with reinforcement learning. In *Proceedings of the 2022 International Conference on Management of Data*. 1528–1541.

[38] Yang Wu, Xuanhe Zhou, Yong Zhang, and Guoliang Li. 2024. Automatic Database Index Tuning: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 36 (2024), 7657–7676.

[39] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch? *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[40] Tao Yu, Zhaonian Zou, Weihua Sun, and Yu Yan. 2024. Refactoring Index Tuning Process with Benefit Estimation. *Proceedings of the VLDB Endowment* 17, 7 (2024), 1528–1541.

[41] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic view generation with deep learning and reinforcement learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1501–1512.

[42] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. Queryformer: A tree transformer model for query plan representation. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1658–1670.

[43] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2023. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *Proceedings of the VLDB Endowment* 17, 4 (2023), 823–835.

[44] Wei Zhou, Chen Lin, Xuanhe Zhou, and Guoliang Li. 2024. Breaking It Down: An In-Depth Study of Index Advisors. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2405–2418.

[45] Xuanhe Zhou, Luyang Liu, Wenbo Li, Lianyuan Jin, Shifu Li, Tianqing Wang, and Jianhua Feng. 2022. Autoindex: An incremental index management system for dynamic workloads. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2196–2208.