Robust Index Benefit Estimation via Hierarchical and Two-dimensional Feature Representation

Tao Li[†*] Feng Liang[‡*] Jinqi Quan[†], Zihang Yang[†], Teng Wang[†], Runhuai Huang[†], Xiping Hu[‡], Meng Li[§], Haipeng Dai[§]
[†]State Cloud, China Telecom, [‡]Shenzhen MSU-BIT University, [§]State Key Laboratory for Novel Software Technology, Nanjing University

***Abstract*—In recent years, machine learning-based index advisors have gained success as they can estimate the benefit of a given index without actually evaluating it via what-if optimizers. However, existing methods often fail to capture index-relevant features adequately, leading to limited accuracy and poor adaptability to changes in workload, schema, or data. To address these challenges, we propose EDDIE, a novel index benefit estimation approach based on *Hierarchical and Two-dimensional Encoding*. Our method encodes columns according to their positions in queries and indexes, consolidates index-related features into a compact representation, and leverages two-dimensional attention to model both query plan structure and index interactions. Finally, EDDIE can be seamlessly integrated with existing index advisors in real-world systems.[1] Extensive evaluations demonstrate that it significantly outperforms state-of-the-art estimators in both accuracy and robustness.**

***Index Terms*—Index Advisor, Machine Learning.**

I. Introduction

Indexes are critical for accelerating queries, but finding an optimal index configuration for a given workload is challenging [1]. Over the decades, many index advisors have been developed to automate or assist this task [2]–[5]. These index advisors typically generate candidate indexes from the database schema and workload, then painstakingly evaluate the efficiency of each configuration to determine the most optimal one. However, this method is time and compute-intensive owing to the huge configuration space, which involves numerous queries, indexable columns, and varying index sizes and column orders. Thus, it is crucial to efficiently estimate index benefits without physically materializing them.

To address this problem, what-if-based methods [6] [4] [7] let index advisors create hypothetical indexes, generate query plans, and estimate index benefits using optimizer cost models. However, they face three main limitations: (1) lack of support for databases without what-if capabilities (*e.g.*, MySQL); (2) long running time, *e.g.*, [8] reports that up to 90% of the running time of an index selection algorithm is taken by what-if optimizers; (3) high computational overhead on databases because of calling optimizers, which can interfere with production workloads [3]. To overcome these limitations, machine learning methods have been proposed to remove reliance on what-if calls. These models can be deployed independently from the database and parallelized on multiple GPUs, thus greatly reducing index tuning time.

As for machine learning methods, the key difficulty lies in building an accurate and robust model, especially in light of the following factors: (1) ***Capturing Rich Query-index Interacting Features:*** Estimating index benefits necessitates capturing a wide range of interacting features, including data columns, query predicates, data statistics, and index interactions [9]. (2) ***Sensitivity to Column Positions:*** The effectiveness of an index is highly sensitive to the position of columns in both the index and SQL operators. For example, ORDER BY in MySQL requires prefix alignment [10], making it essential to model positional constraints accurately. (3) ***Evolving Workloads, Schemas and Data:*** In practice, workloads, schemas, and data distributions on databases change over time [2], [11], [12]. Robust index benefit estimation under changing environments is thus of vital importance. Unsurprisingly, the state-of-the-art methods lack accuracy or robustness because they miss one or more key factors. For example, LIB [13] assumes a static workload and ignores initial indexes, ORDER BY sorting direction, and column positions in the query; DISTILL [14] handles only simple SQL templates, misses join conditions, and records only the highest column position in indexes. This reveals a clear gap between existing literature and the practical needs of database administrators.

Our Approach. To overcome the limitations of prior methods in handling complex index and query interactions as well as environmental changes, we propose EDDIE, an encoder-based estimator for index benefit prediction. EDDIE is designed to capture rich, position sensitive features while generalizing across diverse schemas and query patterns. At a high level, EDDIE takes as input the query plan under existing indexes, schema information, candidate index configuration, and column statistics, and outputs a benefit score for the candidate indexes. To achieve this, we introduce a novel representation technique called *Hierarchical and Two-dimensional Encoding (HTE)*, which embeds index and data features into the query plan and builds its representation from low-level elements to high-level logical structures, forming an encoding hierarchy. In this way, we can not only systematically capture

*These authors contributed equally to this work, ordered by surname.
Correspondence to: Tao Li (lit51@chinatelecom.cn), Xiping Hu (huxp@smbu.edu.cn), and Haipeng Dai (haipengdai@nju.edu.cn).

This work was supported in part by the National Natural Science Foundation of China (No. 62576213, No. U24B20153, and No. 62402212), Guangdong Province Key Projects in Artificial Intelligence (No. 2025ZDZX3047) and Jiangsu Provincial Natural Science Foundation (No. BK20241245).

[1]Source code: https://github.com/quanjnq/Eddie.

all influential factors from different levels of a query plan, but also enable low-level features to be reused by high-level ones. In addition, EDDIE employs a two-dimensional attention mechanism to model both structural dependencies in the query plan and interactions among candidate indexes. Finally, by using a position-based embedding scheme, EDDIE remains schema agnostic while accurately modeling column positions, enabling robust generalization in real-world scenarios.

Contributions. To summarize, our main contributions are:

- We propose and open-source EDDIE, an encoder-based model for index benefit estimation that effectively captures complex interactions between queries and indexes.
- We identify important features previously overlooked by existing methods, such as column positions, compound predicates, and interactions among query plan nodes, which are critical to accurate index benefit estimation.
- We propose HTE, a transferable feature representation technique with a novel two-dimensional attention mechanism to effectively capture the plan tree information and its positional correlation with indexes. HTE also enables transfer learning, and makes the trained model more robust to evolving table schema, workload and data distribution.
- We provide an open-source implementation, and conduct extensive experiments demonstrating that EDDIE reduces estimation error by over 70% compared to state-of-the-art baselines. Our method achieves superior accuracy with less training data via cross-dataset pre-training and can be seamlessly integrated with existing index advisors to deliver substantial end-to-end performance improvements.

Roadmap. The rest of the paper is organized as follows. Section II introduces preliminaries, defines the index benefit estimation problem, and presents a motivating example. Section III gives an overview of our approach, EDDIE, with its core component HTE detailed in Section IV. Section V describes model training and implementation. Section VI presents our experimental evaluation. Section VII reviews related work, and Section VIII concludes the paper.

II. BACKGROUND AND PROBLEM STATEMENT

A. Index Advisor and Index Benefit Estimation

As shown in Figure 1, the index advisors are generally composed of three components: index candidate generation, index selection, and index benefit estimation [15]. 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 [15]. The second component selects subsets of candidate indexes, forming what we call candidate configurations [15]. During the index generation and selection processes, the index benefit estimation component can be invoked from time to time to estimate the benefit of an index or an index configuration. After evaluating all candidate configurations or meeting the stop condition, the index advisor will output the optimal set of index configurations. Among these components, the index benefit estimation (IBE) component is critical, as it directly impacts both the quality of recommendations and tuning efficiency [13], [16]. Therefore, this paper focuses on enhancing IBE by building an accurate and robust model.

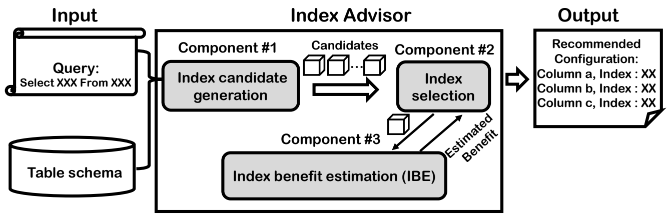

Fig. 1: The workflow of an index advisor.

B. Problem Statement

The index benefit estimation, which aims to predict the reduction in query execution cost brought by an index set, involves a continuous target and can therefore be formulated as a regression problem, with features from the query, data, and index as input and the estimated cost reduction as output.

Parameter Definition:

• *Database*. A database D comprises a set of tables T. Each table $t \in T$ has a schema A^t (*i.e.*, a set of columns) and statistics S^t (*e.g.*, row count, min/max, histograms). We represent the database as a tuple (A, S), where A is the column space and S collects the statistics for all tables in T.

• *Workload*. Workload W is a set of queries over a time interval. For each $q \in W$, T_q denotes its referenced tables.

• *Index*. An index I, built on table t to improve query performance, consists of an ordered set of columns from t. An index materialized before tuning is called an initial index; one generated by the advisor is a candidate index.

• *Initial index configuration*. An index configuration is a set of indexes. The initial configuration C^0 includes all initial indexes in D. For a query q, only a subset $C_q^0 \subseteq C^0$ is relevant.

• *Candidate index configuration*. A candidate configuration C' is a set of advisor-generated indexes for tables in D. The advisor may iteratively generate and select the optimal C'.

• *Index interaction*. Index interaction refers to the phenomenon where the benefit of one index for certain queries is influenced, positively or negatively, by the presence of other indexes [9]. As a result, the overall benefit of a candidate configuration must be evaluated jointly.

• *Physical query plan*. A physical query plan for q describes the steps to execute the query, typically as a tree where nodes are physical operators and edges represent intermediate results flowing upward. The plan can be denoted as $P_q = (N, E)$, where N and E are the sets of nodes and edges, respectively. In the rest of the paper, we mainly discuss physical query plans, and use query plan and physical query plan interchangeably if there is no ambiguity.

• *Execution cost*. Given a query q and an index configuration C, let $Cost(q, D, C)$ denote the execution cost of q's *optimal* query plan P_q^* on database D under C. This cost depends on various factors, including the query structure, index configuration, and table statistics.

• *Index benefit metric*. Index benefit $B(q, D, C^0, C')$ is commonly measured by the cost reduction when applying the indexes in C'. Inspired by [13], [16], we focus on the *relative* cost metric to characterize index benefit in this paper due to its robustness to small cost estimation errors.

Problem Formulation. Formally, for query q, the cost reduction when replacing C^0 by C' is

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

and the *relative* cost reduction is defined as the cost reduction ratio, *i.e.*,

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

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

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

In practice, index advisors generate candidate configurations that yield positive benefits, and their goal is to find the optimal configuration C^* that maximizes overall cost reduction. Since $Cost(q, D, C^0)$ in Equation 2 is fixed and known from query history or optimizer, the index benefit $B(\cdot)$ plays a key role in determining $R^W(D, C^0, C')$. Accurate estimation of index benefit is therefore essential. Then, our objective is to build a learned model to estimate index benefits.

Index Benefit Estimation (IBE) Problem Definition : *Given a database $D = (A, S)$, we have a dataset where each sample contains a query q, its optimal query plan P_q^*, an initial index configuration C^0, a candidate index configuration C', as well as the corresponding index benefit $B(q, D, C^0, C')$. All the queries in the dataset constitute the workload W. The objective of IBE 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., $\sum_{q \in W} (B(q, D, C^0, C') - B'(q, D, C^0, C'))^2$.*

C. Motivating Example

In this subsection, we will explain why existing methods are not sufficient to address the problem via a motivating example.

Motivating Representative Queries. To illustrate the underlying complexity overlooked by existing work, we present five representative queries over a simple database (Figure 2) with a single table *customer* containing 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*). Queries (a)–(d) share similar structures but differ in aspects such as column order, sort direction, and specific operators. As such, the estimated index benefit should vary based on whether a query can effectively leverage the candidate index. However, the state-of-the-art method LIB [13] produces identical estimates across these queries. The final query (e)

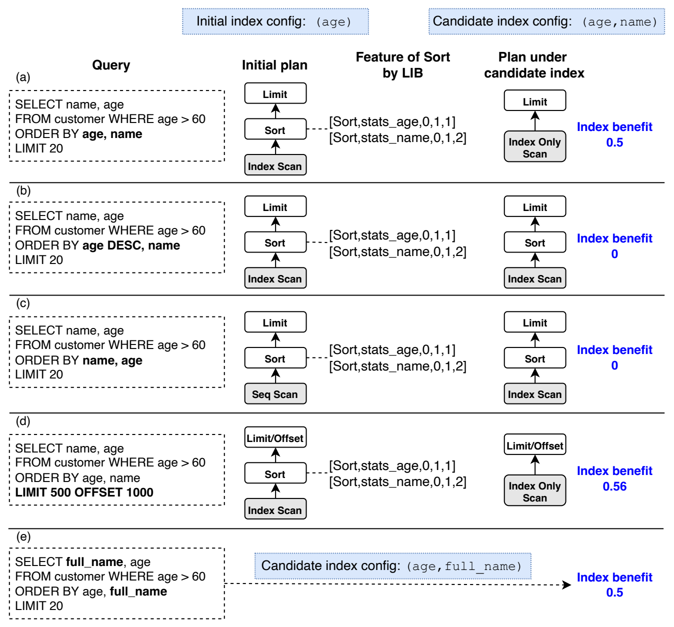

Fig. 2: Motivating example: (a) candidate index can serve filter, sort and projection together, bringing significant index benefit (0.5); (b) candidate index can serve filter only due to columns in ORDER BY sorted in different directions (DESC/ASC); (c) candidate index can serve filter only due to mismatched column order in ORDER BY and candidate index; (d) replacing LIMIT by LIMIT OFFSET can cause index benefit change from 0.5 to 0.56; (e) renaming a column should have no impact on index benefit.

is identical to (a) except that one column is renamed due to schema evolution. Ideally, the estimated benefit should remain consistent and be insensitive to column name changes.

• ***Base Query.*** For query (a), the candidate index (*name*, *age*) is a covering index [17], as it includes all columns required by the query output. Moreover, the index simultaneously supports both the WHERE and ORDER BY clauses: the WHERE condition references the leading column, and the index preserves the sort order needed by ORDER BY. As a result, the query can be significantly accelerated, yielding a substantial benefit (*e.g.*, 0.5). In the query plan, the initial *Index Scan* is replaced with an *Index Only Scan* when using the candidate index. For a specific node like *Sort*, LIB will generate two Index Optimizable (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). Specifically, the third and second last bits [*0,1*] are a one-hot encoding representing the multi-attribute index type, and the last bit is 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,

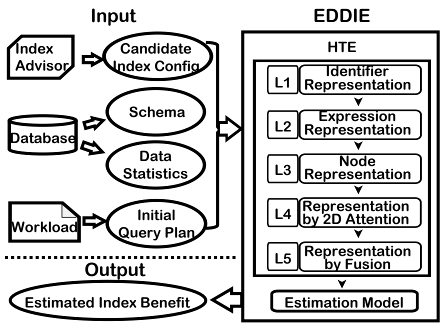

Fig. 3: Overview of EDDIE for index benefit estimation.

as well as the tree structure of the query plan. Moreover, LIB assumes empty initial indexes (i.e., initial indexes are not included in the input features), which generally does not hold in real-world databases.

• ***Distorted Sorting Order.*** For query (b), changing the sort direction on *age* prevents the candidate index from serving the ORDER BY clause. Initially, the WHERE clause can leverage the initial index, producing an *Index Scan* node in the plan. However, applying the candidate index does not alter the plan, resulting in zero index benefit. This highlights the importance of capturing all influential factors in IBE. Unfortunately, LIB cannot distinguish between queries (a) and (b), generating identical IO operations for both.

• ***Changed Column Positions.*** As for 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 a change when generating IO operations because it does not record the positions for columns in SQL operators.

• ***Plan Nodes and Structure.*** LIB only considers five types of IO operations (*i.e.*, join, sort, group, scan_range, scan_equal) and ignores the plan tree structure. However, other node types, like *Limit* and *Union*, can also contribute to the query execution cost and ultimately affect the benefit of a candidate index. We showcase this problem by query (d), which changes the LIMIT clause in query (a) to a LIMIT/OFFSET clause. Correspondingly, the index benefit increases from 0.5 to 0.56. Unfortunately, LIB is unable to capture this change.

• ***Evolved Table Schema.*** To simulate schema evolution, query (e) renames column *name* to *full_name*, while keeping semantics identical to query (a). The expected index benefit should thus remain unchanged. However, most query plan representation methods [18], including AVGDL [19] and QueryFormer [20], rely heavily on column names. As a result, queries (a) and (e) are encoded into different vectors, violating drift tolerance and causing inconsistent benefit estimation. Additionally, these methods lack index representations and therefore cannot be directly applied to the IBE problem.

III. 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 3, 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 $C' = \{I\}$. Then, the representation R^* is fed into the estimation model, 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. This hierarchical representation of the query plan is inspired by the top-down design of SQL grammar in ANTLR [21], where language structures are built progressively from coarse to fine levels. Specifically, HTE progressively constructs the final representation of a query plan through five levels as below.

- **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. There is a finite number of node types, such as *Join* and *Union*, for any database.
- **L4 – plan-dimension and index-dimension attentions**, which leverage the self-attention mechanism of Transformer [22] to capture the relations of different nodes from the perspectives of plan tree structure and the interaction of the candidate indexes in C' [9], respectively.
- **L5 – final representation**, produced by fusing the outputs of the two-dimensional attentions by a pooling mechanism.

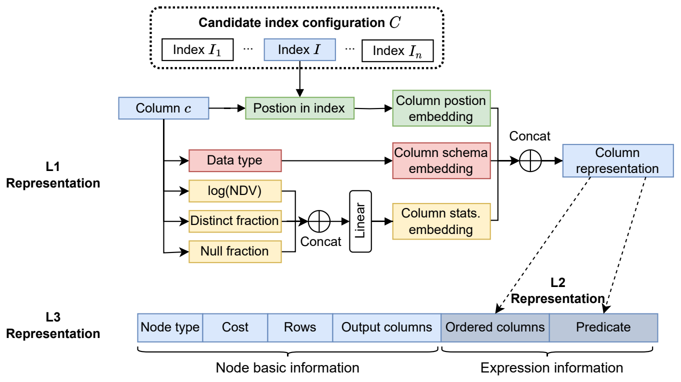

Fig. 4: Column, expression and representations for a candidate index.

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 captures the index benefit-related features as many as possible and encodes them at the suitable levels. Third, we respect the importance of column positions and encode the column position information regarding both the candidate index and the query plan node. Fourth, the position-based column identifiers also help overcome the drawback of name-based embedding methods, which are sensitive to schema changes. As a result, HTE is schema-agnostic and enables transferring knowledge, which has been learned from identical databases with different schemas/workload/data, to future new IBE tasks.

IV. 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 the corresponding entity by R_i.

A. L1: Identifier

In practice, index advisor tools, such as DTA [23] and AutoAdmin [24], 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 vanishes.

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 4, 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.

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

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 the 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 2, 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 statistical metrics, including the number of distinct values (NDV), fraction 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 column names. Meanwhile, column position information regarding the candidate index is preserved. In this way, two columns will have the same position embedding, as long as their positions in the index are the same. Also note that one column can yield multiple column representations, each corresponding to one index in the candidate index configuration.

B. L2: Expression

There exist dozens of SQL operators in the standard SQL specification. In expression-level encoding, we aim to describe the expressions appearing in these SQL operators with an emphasis on *Scan*, *Join*, *Aggregate*, and *Sort*, which can be optimized by indexes. 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 the operators are optimizable through indexes. In the following, we describe the encoding methods for these two types of expressions.

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.

In our approach, 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 the selectivity of a predicate.

However, most of the existing predicate encoding methods are only able to deal with atomic predicates [18]. In this paper, we further support compound predicates. Our insight is that compound predicates can be expressed in a 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 [20], 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 *aggregate predicate node*, with learnable feature embedding, and append it to the head of the sequence. Unlike other nodes, aggregate predicate node is connected with all the other nodes. Then, by applying the tree-structured Transformer on the node embedding sequence, the output vector of the aggregate predicate node, denoted by $E_2^{p_s} \in \mathbb{R}^{m \times d_p}$, will represent the entire compound predicate, that is, $R_2^p = E_2^{p_s}$.

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.

C. 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 4. Node encoding involves four basic types of information: node type, cost, output rows, and output columns, as detailed 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.
2) **Node cost**. It is the optimizer's estimated cost for the node. For certain 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**. A node's output columns contain useful information for recommending *covering indexes*. Specifically, 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. Thus, we are motivated to encode the output columns from the node, and leverage the L1 representation to describe those columns. However, existing query plan representation methods, like [20], [25] do not encode this information.

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 4. 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 one candidate index.

D. 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 x_s into the sequence fully connected with all other nodes, thus 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 5). In the first dimension, which we call *plan-dimension*, we leverage the tree-structured attention mechanism, similar to [20], 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 *index-dimension*, aims to capture the interaction among

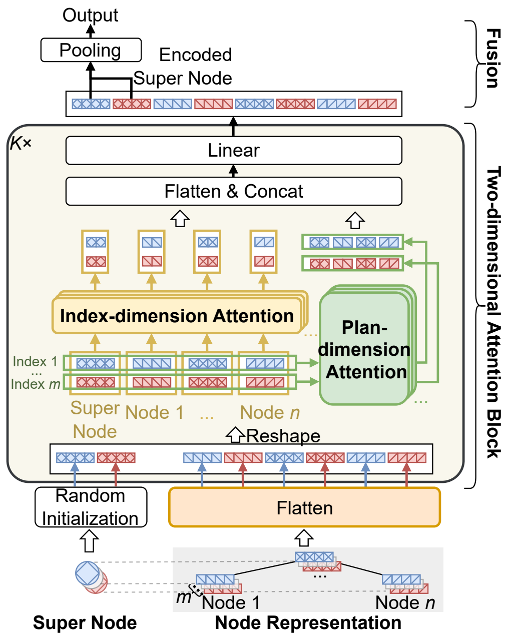

Fig. 5: Two-dimensional attention blocks followed by a fusion layer.

indexes [9]. 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 [22]. 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 $R_4 \in \mathbb{R}^{(n+1)\times m\times d_x}$.

E. L5: Fusion

Finally, we pick the super node's output from L4 representation, denoted by $R_4^{x_s} \in \mathbb{R}^{m\times d_x}$, 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 3, *i.e.*, $R^* = R_5$.

V. Training and Implementation

A. Estimation Model and Training

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:

$$\hat{B} = MLP(R^*), \tag{5}$$

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 database, 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 minimum 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.

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. Note that it is preferable that the databases in pre-training and fine-tuning phases should have identical database type, software version, and hardware profile in order to assure knowledge transfer quality. Minor software version differences (e.g., MySQL 8.0.29 and 8.0.30) may be acceptable, as long as the version change does not cause disparity in query execution logic. Also, as a cloud service provider who operates a large number of databases on the cloud, it is feasible for us to accumulate sufficient training data (with user permission) to pre-train a foundation model for each database instance type, and then adapt to downstream index tuning tasks. We leave the large-scale pre-training for future work; instead, we will demonstrate its effectiveness by certain benchmarks and realistic datasets in the evaluation section.

B. Integration with Index Advisors

During the index selection process, index advisor tools, 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 execution cost for the tuners to use. According to Equation 1, we can determine the execution cost under the candidate index configuration by the following formula:

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

Here, the original cost $Cost(q, D, C^0)$ can be obtained either by running the query or from historical query logs.

TABLE I: Summary of Datasets

Dataset	# Queries			# Cases		
	Templated	Synthetic	Total	w/o Initial	w/ Initial	Total
TPC-DS	390	0	390	4,910	22,118	27,028
TPC-DS+	390	3,000	3,390	8,378	22,664	31,042
TPC-H	1,120	0	1,120	2,400	2,240	4,640
TPC-H+	1,120	3,000	3,120	5,888	2,830	8,718
IMDB	240	0	240	6,240	23,900	30,140
IMDB+	240	3,000	3,240	9,858	24,516	34,374
Redbench	556	0	556	10,852	53,556	64,408

VI. Evaluation

A. Experimental Setup

Our experiments differ from existing work in three important ways. First, most of existing work experiments on benchmark workloads, like TPC-DS, TPC-H, and IMDB, which are based on fixed SQL templates and only cover specific application scenarios. But real-world workloads are dynamic, and SQLs are much more diversified. To evaluate an IBE method on more general scenarios, we further consider workloads extended from these templated ones by synthesizing random SQL or incorporating realistic cloud query characteristics. Second, we consider both empty and non-empty initial index scenarios. The latter is more general in the real world. Third, we simulate a variety of environmental changes, such as workload and schema drifts, as well as data updates, to test IBE methods under these realistic conditions.

1) Workloads:

- **Benchmark workloads**: First, we employ standard benchmarks, including TPC-DS, TPC-H with a scale factor (SF) of 10, and IMDB, whose queries are generated via predefined templates. Following prior practices [26], we exclude certain templates with disproportionately high execution costs, as they can dominate workload costs and skew index selection. Specifically, for TPC-DS, we use 78 out of 99 templates, generating 5 queries per template, resulting in 390 queries. For TPC-H, we use 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 workloads**: There are two ways to extend the above standard workloads. First, we synthesize some queries by combining different SQL components, such as JOIN, WHERE, GROUP BY, ORDER BY, and LIMIT, in a probabilistic and recursive manner. Then, the synthesized queries are mixed with the original ones to create the extended workloads. Second, the increasing adoption of cloud databases in recent years has motivated us to evaluate IBE methods under cloud environments. To this end, we employ an existing cloud-optimized workload named Redbench [27], which samples queries from IMDB based on the similarity to Redset, an AWS-released dataset containing real customer query metadata.

2) Method of generating index configurations: For each query q in the workload without any initial indexes, we use AutoAdmin [24] to generate the optimal configuration, denoted by $\mathcal{C}^*$. The maximum number of indexes in the candidate configuration is 5, where each index contains at most 2 attributes. Then, all the non-empty subsets of $\mathcal{C}^*$ is,

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

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

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

As an example, suppose $\mathcal{C}^* = \{(c1, c2), (c3)\}$, then $\mathbb{N}(\mathcal{C}^*) = \{\{(c1, c2)\}, \{(c3)\}, \{(c1, c2), (c3)\}\}$. 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 can create two practical scenarios for each workload:

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

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. Each query is executed 3 times without database cache warm-up, and the average execution time of the 3 runs is used for deriving index benefits. In this way, we can collect all the information needed for each training sample (see Section V-A1). The characteristics of the resulting datasets are summarized in Table I, detailing query and case counts across workloads.

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

- **PG**. As most existing index tuners rely on optimizer-based cost estimation, we compare Eddie against PostgreSQL (PG) 12.13's cost estimator. Specifically, we utilize HypoPG 1.3.1 [28] to create virtual indexes, invoke the optimizer to compute query costs before and after index creation, and calculate the estimated index benefit from these costs.
- **AMA-R**. AIMeetsAI [16] is a classifier to predict query performance regression with key features derived from the difference between two query plans. We replace its classification component with a regression one, which consists of a two-layer fully connected neural network and a Sigmoid output layer, so that the index benefit can be predicted. The altered model is called AMA-R. As input to the model, each pair of query plans is the one generated for an identical query before and after creating the candidate indexes.
- **LIB [13]**. A state-of-the-art attention-based model for index benefit prediction.
- **QF-I**. QueryFormer [20] is the state-of-the-art query representation method but lacks consideration of index configuration. To remedy it, we mix an index encoder from

TABLE II: Performance with and without Initial Index Configurations

Model	TPC-DS		TPC-DS+		TPC-H		TPC-H+		IMDB		IMDB+		Redbench	
	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE
Without Initial Index Configurations														
PG	3422.5/468.5	0.125	2003.7/335.2	0.092	1623.7/279.1	0.084	2236.6/379.7	0.074	7670.6/1071.6	0.238	8818.0/1551.5	0.341	6548.6/885.5	0.309
AMA-R	460.3/135.3	0.051	285.0/102.5	0.047	560.8/146.5	0.023	398.0/135.2	0.046	2670.4/408.5	0.142	1873.8/332.9	0.115	1591.7/271.8	0.131
LIB	345.5/133.7	0.065	499.3/169.9	0.073	134.3/30.3	0.013	200.8/201.5	0.058	2827.1/551.0	0.203	2680.1/466.2	0.161	1415.5/233.0	0.182
QF-I	208.6/95.1	0.035	274.1/142.0	0.056	148.8/22.9	0.010	214.1/167.3	0.044	1909.6/284.5	0.113	2065.3/346.0	0.126	1528.5/247.1	0.129
EDDIE	**162.2/44.9**	**0.028**	**97.7/86.6**	**0.039**	**70.8/15.9**	**0.009**	**90.6/126.8**	**0.033**	**1396.3/248.5**	**0.102**	**1154.2/247.0**	**0.096**	**1078.7/214.9**	**0.112**
With Initial Index Configurations														
PG	5063.7/749.7	0.226	3882.0/631.1	0.141	2197.1/263.5	0.075	2405.4/373.7	0.076	8896.8/1417.6	0.345	8894.8/1331.6	0.319	6493.1/761.0	0.282
AMA-R	426.3/88.1	0.041	319.3/90.0	0.043	255.9/57.2	0.014	223.9/109.7	0.035	2344.9/392.2	0.149	2205.2/356.3	0.134	1669.4/282.2	0.145
LIB	209.0/66.9	0.041	359.5/103.1	0.052	85.4/21.3	0.015	136.5/185.5	0.058	2087.7/369.7	0.156	2654.5/414.8	0.163	1319.1/229.5	0.149
QF-I	388.0/82.6	0.038	479.4/128.7	0.051	118.1/19.6	0.012	171.5/166.6	0.045	1968.8/304.4	0.134	2308.9/363.2	0.137	1505.4/276.3	0.145
EDDIE	**93.5/34.7**	**0.025**	**120.1/61.5**	**0.030**	**50.5/14.7**	**0.011**	**74.2/104.5**	**0.031**	**641.4/159.5**	**0.084**	**1159.0/228.8**	**0.087**	**857.7/188.9**	**0.099**

ChangeFormer [29] to make it capable of predicting index benefit. The resultant model is named QueryFormer-I (QF-I, in short). Specifically, the output of ChangeFormer's index encoder and QueryFormer's plan representation are concatenated into the final representation vector, which then serves as input to a two-layer fully connected neural network with a Sigmoid function, similar to AMA-R.

4) Model training: We have trained the EDDIE model with the Adam optimizer [30] 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.

5) Metrics: Q-error [13], [25], [31] and *mean absolute error (MAE)* [32] are two commonly used metrics to evaluate the accuracy of regression models in the AI4DB field. Essentially, Q-error characterizes the ratio between the predicated and the actual, whereas MAE excels due to sharing the same unit as the regression target. In our experiments, we measure both of them. Letting S be the test dataset, and letting $\hat{B}_s$ and B_s be the predicted and ground-truth index benefits, respectively, for each test sample $s \in S$, its Q-error $Qerror(s)$ is $max(\frac{\hat{B}_s+\epsilon}{B_s+\epsilon}, \frac{B_s+\epsilon}{\hat{B}_s+\epsilon})$, where ϵ is a very small constant for error correction, *i.e.*, 1e-4. The mean Q-error, i.e., $\frac{1}{|S|}\sum_{s\in S} Qerror(s)$, measures the average estimation performance over the test dataset, and the 95-th percentile Q-error reflects the estimation performance in some worst cases. Conventionally, MAE is defined as $\frac{1}{|S|}\sum_{s\in S} |\hat{B}_s - B_s|$.

6) Environment: Experiments have been 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 is PG 12.13. The training data collection and testing of EDDIE are conducted on the same database, except for the pre-training process, which is conducted on another database but with an identical version and configuration.

B. Prediction Accuracy

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

In the case of no initial index configurations, EDDIE consistently outperforms other baseline methods in terms of both Q-error and MAE under all workloads. For example, in TPC-DS, EDDIE achieves a mean Q-error of 44.9, which is 66.8%, 66.4% and 52.8% lower than AMA-R, LIB and QF-I, respectively. Regarding the 95-th percentile Q-error, EDDIE also demonstrates significant improvement, *e.g.*, EDDIE reduces the 95-th percentile Q-error by 53% over LIB in TPC-DS. Meanwhile, this phenomenon sustains in terms of MAE, although the scale of improvement slightly decreases due to its different definition from Q-error. EDDIE achieves such advantages mainly because it successfully encodes sufficient features and column position information. For example, in Query5 of TPC-DS workload [33], the CTE expression *csr*'s performance can be improved by introducing a configuration with two single-column indexes $\{$(*catalog_sales.cs_sold_date_sk*),(*catalog_returns.cr_returned_date_sk*)$\}$, which has a ground-truth benefit value of 0.23 in one sample of the dataset (a case of positive index interaction [9]). Meanwhile, either of the single-column indexes alone brings no benefit to the Query5, because the *Union* operator in the query hinders the subsequent *Join* operator from utilizing these indexes. Unfortunately, LIB misestimates a high benefit for these single-column indexes to be 0.011 and 0.406, respectively, since it neglects the *Union* operator and its impact. On the contrary, EDDIE obtains a much more accurate prediction (*i.e.*, 0.005 and 0.031), very close to the ground-truth benefit of 0. Besides, we have found that among the baselines, the accuracy of PG's optimizer is the worst under all workloads, which aligns with the findings in [15] [16] and implies that database optimizer's cost estimation is often untrustworthy, especially for complex SQL.

In the setting with initial index configurations, EDDIE continues to perform better. Furthermore, we have found EDDIE generally has lower estimation errors compared with the previous setting. In real-world scenarios where databases rarely run without initial indexes, EDDIE is more practical.

C. Robustness

To assess the *robustness* of EDDIE against environmental changes, we introduce three types of drifts (query variation, schema change, data volume shift) exclusively to the test set (with initial index configurations), while leaving the training

TABLE III: Performance Comparison under Drift Scenarios (with Initial Index Configurations)

Model	TPC-DS		TPC-DS+		TPC-H		TPC-H+		IMDB		IMDB+		Redbench	
	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE
Query Variations														
PG	4634.4/672.7	0.179	6513.9/946.4	0.194	5808.3/685.1	0.259	5667.0/578.8	0.179	9275.5/1756.0	0.363	9541.3/2099.7	0.351	7676.5/1051.8	0.314
AMA-R	2739.7/400.6	0.135	3024.2/446.2	0.132	5702.3/845.3	0.290	3283.0/392.0	0.208	4753.3/688.6	0.209	7361.5/1049.7	0.221	4201.4/546.8	0.199
LIB	1167.0/218.8	0.130	2158.4/351.9	0.135	2986.0/398.8	0.304	3081.6/805.3	0.253	3718.1/532.0	0.211	3075.2/445.7	0.110	3299.8/485.0	0.223
QF-I	3282.8/471.4	0.203	4751.5/605.0	0.159	6453.3/975.7	0.335	2287.9/735.0	0.267	5804.8/784.8	0.268	4993.6/1099.3	0.228	4114.5/547.9	0.220
EDDIE	**992.6/197.1**	**0.095**	**1697.1/296.5**	**0.098**	**1266.7/286.6**	**0.221**	**808.5/221.2**	**0.157**	**3538.2/501.2**	**0.156**	**1062.8/250.7**	**0.064**	**2577.8/391.9**	**0.153**
Schema Changes														
PG	5063.7/749.7	0.226	3882.0/631.1	0.141	2197.1/263.5	0.075	2405.4/373.7	0.076	8896.8/1417.6	0.345	8894.8/1331.6	0.319	6493.1/761.0	0.282
AMA-R	426.3/88.1	0.041	319.3/90.0	0.043	255.9/57.2	0.014	223.9/109.7	0.035	2344.9/392.2	0.149	2205.2/356.3	0.134	1669.4/282.2	0.145
LIB	209.0/66.9	0.041	359.5/103.1	0.052	85.4/21.3	0.015	136.5/185.5	0.058	2087.7/369.7	0.156	2654.5/414.8	0.163	1319.1/229.5	0.149
QF-I	1482.4/250.7	0.128	2083.8/323.6	0.126	2118.2/301.4	0.071	6241.5/752.1	0.331	4582.2/577.3	0.235	3258.7/450.9	0.279	2895.0/368.3	0.249
EDDIE	**93.5/34.7**	**0.025**	**120.1/61.5**	**0.030**	**50.5/14.7**	**0.011**	**74.2/104.5**	**0.031**	**641.4/159.5**	**0.084**	**1159.0/228.8**	**0.087**	**857.7/188.9**	**0.099**
Data Volume Shifts														
PG	6345.4/966.3	0.191	6079.5/883.1	0.175	3466.9/599.4	0.133	2421.6/366.8	0.083	N/A		N/A		N/A	
AMA-R	2576.4/401.9	0.123	2677.1/337.0	0.115	3781.4/472.6	0.092	341.7/200.3	0.057	N/A		N/A		N/A	
LIB	1921.5/304.7	0.117	1664.9/304.1	0.121	1682.8/281.3	0.060	1926.8/287.7	0.093	N/A		N/A		N/A	
QF-I	3863.1/568.8	0.189	4076.0/557.5	0.150	2904.9/443.2	0.086	5927.8/746.7	0.323	N/A		N/A		N/A	
EDDIE	**1883.5/273.1**	**0.094**	**1619.8/240.7**	**0.088**	**1450.5/227.1**	**0.057**	**193.2/163.4**	**0.050**	N/A		N/A		N/A	

TABLE IV: Performance under Index Perturbation (with Initial Index Configurations)

Model	TPC-DS		TPC-DS+		TPC-H		TPC-H+		IMDB		IMDB+		Redbench	
	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE
PG	4019.2/517.5	0.128	2188.6/345.6	0.095	1439.1/211.8	0.066	4304.6/547.5	0.092	8814.6/1774.6	0.360	8483.4/1258.4	0.263	6405.4/770.5	0.296
AMA-R	590.2/117.8	0.050	194.5/93.2	0.045	417.1/141.8	0.021	144.0/103.4	0.034	2353.0/414.6	0.140	2224.3/358.9	0.114	1401.7/247.4	0.133
LIB	356.3/135.8	0.065	4953.1/458.0	0.128	134.9/36.6	0.016	394.6/394.6	0.107	2709.2/477.8	0.206	3464.2/547.2	0.181	2403.4/233.8	0.193
QF-I	467.9/95.5	0.036	732.0/262.0	0.078	219.5/35.5	0.013	173.0/170.8	0.049	2004.2/294.6	0.115	2145.6/358.1	0.119	1317.5/230.1	0.129
EDDIE	**129.1/37.2**	**0.028**	**93.1/74.2**	**0.037**	**120.0/32.7**	**0.012**	**73.6/102.6**	**0.033**	**1144.8/233.5**	**0.110**	**1639.6/286.4**	**0.097**	**1013.5/183.5**	**0.114**

set unchanged. The purpose of this setup is to evaluate the trained model's resilience to real-world drifts, simulating workloads that evolve independently of the training data.

1) Query Variation: To examine EDDIE's performance under query variations, we modify the test set by appending a randomly generated predicate to each query. Specifically, for each query, we select an arbitrary column from the accessed table, pair it with a randomly chosen comparison operator (*e.g.*, $>$, $<$, $=$), and assign a value sampled from that column's data as the condition. This simulates query evolution in dynamic environments. Results in Table III show that EDDIE achieves the lowest mean/95-th percentile Q-error, as well as MAE, across all workloads. For instance, in TPC-H, EDDIE attains a mean Q-error of 286.6 and 95th of 1266.7, significantly outperforming AMA-R (845.3, 5702.3), LIB (398.8, 2986.0), and QF-I (975.7, 6453.3). These results demonstrate EDDIE's robustness in maintaining accurate under shifting query patterns, extending its effectiveness beyond static conditions. This advantage is primarily attributed to the position-aware column and tree-based predicate encodings.

2) Schema Change: To investigate the resilience to schema changes, we randomly alter 20% of the column names accessed by the 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 III, again EDDIE outperforms all baselines, achieving the lowest Q-error and MAE metrics across datasets. It is worth noting that compared with the results before the schema change (Table II), the performance of EDDIE and LIB is unaffected; oppositely, QF-I's Q-error and MAE degrade drastically (*e.g.*, mean Q-error rises by more than 3x from 82.6 to 250.7 in TPC-DS). This disparity occurs mainly because QF-I encodes columns by their names, rendering it sensitive to unseen names, whereas EDDIE leverages schema-agnostic encodings, contributing to its robustness in handling schema changes.

3) Data Volume Shift: To assess the impact of data volume shift, we adjust the scale factors of TPC-DS and TPC-H, shrinking the original 10GB databases to 5GB, and re-sample 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 and Redbench, experiments are not conducted on these datasets. As reported in Table III, EDDIE consistently outperforms baselines across TPC-DS, TPC-H and their extended versions. For instance, in TPC-H+, EDDIE achieves a mean Q-error of 163.4, reducing by 43.2% over the second best method, i.e., LIB. These results demonstrate that EDDIE adapts effectively to data volume changes. Such robustness property mainly stems from the fact that EDDIE considers data statistics and selectivity during its featurization process.

D. Changed Column Positions

A key strength of EDDIE lies in its column position-awareness, which we will validate through index perturbation experiments. Specifically, the 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

TABLE V: Ablation Study of EDDIE's Components (with Initial Index Configurations)

Model	TPC-DS		TPC-DS+		TPC-H		TPC-H+		IMDB		IMDB+		Redbench	
	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE	Q-error 95th/Mean	MAE
EDDIE	**93.5/34.7**	**0.025**	**120.1/61.5**	**0.030**	**50.5/14.7**	**0.011**	**74.2/104.5**	**0.031**	**641.4/159.5**	**0.084**	**1159.0/228.8**	**0.087**	**857.7/188.9**	**0.099**
w/o idx attn	93.6/37.2	0.026	128.5/63.7	0.032	76.3/17.9	0.012	94.9/115.2	0.034	900.2/184.2	0.091	1378.1/249.1	0.094	887.7/198.3	0.105
w/o histogram	96.0/34.8	0.026	125.3/66.6	0.032	53.8/15.3	0.012	81.5/113.1	0.034	705.4/163.3	0.085	1191.0/229.9	0.088	925.7/202.5	0.106
w/o statistics	101.5/37.8	0.025	146.4/62.3	0.034	54.8/15.5	0.013	86.8/111.9	0.033	656.8/163.1	0.085	1195.3/240.3	0.091	880.8/192.2	0.102
w/o predicate	97.8/37.3	0.025	126.9/63.7	0.031	59.7/15.8	0.013	95.1/131.3	0.037	817.7/185.2	0.089	1169.4/250.0	0.092	979.4/199.5	0.102
w/o output rows	123.1/46.4	0.026	121.4/61.9	0.031	58.0/15.2	0.012	87.3/124.9	0.036	695.3/169.7	0.086	1167.3/229.4	0.091	1027.1/211.6	0.105

for each index in the index configuration. For example, a two-column index (a, b) is changed to (b, a).

The experimental results are shown in Table IV. We can observe a clear trend that Q-error and MAE metrics are increasing, compared with the datasets without index perturbation (see Table II). Meanwhile, it is important to note that in TPC-DS+, which additionally involves synthetic queries, the baseline methods encounter a higher performance variation, in contrast with the vanilla TPC-DS workload. Specifically, the mean and 95th percentile Q-errors of LIB drastically degrade from (103.1, 359.5) to (458.0, 4953.1). Conversely, EDDIE's performance is improved. Its mean and 95th percentile Q-errors go in an opposite direction from (61.5, 120.1) to (74.2, 93.1). The trend is the same for the rest of the datasets containing synthetic queries (TPC-H+ and IMDB+). The main reason for this phenomenon 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. Additionally, we can observe a significant deterioration of LIB's performance because it neglects the importance of column position in SQL operators as exemplified in Section II-C. EDDIE's index-guided position encoding method properly correlates the orders of the column in the index and SQL operator, and consequently achieves a position-awareness effect. Unsurprisingly, EDDIE's estimation performance before and after index perturbation is more stable.

E. Ablation Study

To understand the effectiveness of EDDIE's two key components, i.e., the two-dimensional attention mechanism and novel featurization method, we have conducted the ablation study as follows by disabling the index-dimension attention and eliminating some input features.

1) Index-Dimension Attention: The original two-dimensional attention involves index and plan dimensions. To remove the index-dimension attention component from EDDIE, we replace its corresponding embedding in the model with zeros while keeping the remaining architecture intact. As shown in Table V (w/o idx attn), this removal results in a performance regression across all workloads. For instance, in the TPC-DS+ workload, the mean and 95th percentile Q-errors increase from 61.5 to 61.7, and from 120.1 to 128.5, respectively. Meanwhile, MAE increases from 0.030 to 0.032. Actually, this trend appears across all tested datasets, highlighting the positive utility of the index-dimension attention in improving the model's accuracy.

2) Complexity of Input Features: In order to better predict index benefit, EDDIE has judiciously mingled a rich set of relevant features, such as histogram, data statistics, predicate, and output rows (see Section IV). To verify the usefulness of these features and evaluate our approach when these features are sometimes not available in the real world (e.g., users do not run ANALYZE to collect histograms), we have conducted ablation studies by replacing the corresponding features with zeros. Experimental results have been reported in Table V, which shows that removing each of these features individually will cause performance degradation, even though in some cases the MAE metric is not impacted. Specifically, in TPC-DS, the MAE, mean and 95-th percentile Q-errors increase from (0.025, 34.7, 93.5) to (0.026, 34.8, 96.0) after removing the histogram feature. The same phenomenon happens in other workloads, indicating these features are beneficial to benefit estimation, and should be considered whenever available.

F. Pre-training for Enhanced Accuracy

HTE's key advantage is producing transferable features, enabling pre-training and fine-tuning for index benefit estimation. This enhances generalizability, resilience to schema/data shifts, and reduces training data cost. To demonstrate the effectiveness of this approach, we conduct an experiment where the model was pre-trained on a diverse set of unseen datasets and then fine-tuned on the target workload. The pre-training datasets reuse 18 of the 20 datasets from [34], excluding TPC-H and IMDB to avoid overlapping with the target workloads. For each dataset, 1000 queries are automatically generated using the complex mode workload generator from [34], and index configurations are generated as described in Section VI-A2. Then, the model is 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 VI-B. Then, we compare this fine-tuned model with baseline models trained from scratch on 50% and 100% of the training data.

The results shown in Table VI prove the advantages of HTE's transferable feature representation. For example, in TPC-DS, the model pre-trained and fine-tuned on 50% data outperforms the baseline trained on 100% data, achieving a mean Q-error of 39.1 and an MAE of 0.027, compared to the baseline's 44.9 and 0.028. Similar improvements can be observed in TPC-H, IMDB and Redbench workloads.

G. Integration with Index Advisor

We further investigate EDDIE integrated with existing index advisors. In this setup, we use AutoAdmin [24] as an advisor

TABLE VI: Performance of Pre-training and Fine-tuning

Dataset	Metric	From scratch (50% Data)	From scratch (100% Data)	Pre-trained (50% Data)
TPC-DS	Q-error 95th/Mean	275.8/73.3	162.2/44.9	**157.6/39.1**
	MAE	0.035	0.028	**0.027**
TPC-H	Q-error 95th/Mean	73.2/21.5	70.8/15.9	**57.4/14.0**
	MAE	0.011	0.009	**0.008**
IMDB	Q-error 95th/Mean	1621.5/303.9	1196.3/228.5	**1194.5/219.1**
	MAE	0.123	0.102	**0.098**
Redbench	Q-error 95th/Mean	1129.7/229.3	1078.7/214.9	**1064.7/207.9**
	MAE	0.119	0.112	**0.106**

example, substitute its estimation part with EDDIE, PG 12's what-if-based estimator and LIB, respectively, and compare the quality of their recommended optimal indexes. The experiment is conducted using the workloads of TPC-DS, TPC-H, IMDB, and Redbench, with the training and testing datasets split by query and 5-fold cross-validation applied (consistent with Section VI-B). For any test set, we divide its queries into groups each with 5 queries. This way, we are able to evaluate the efficacy of recommended indexes in the granularity of query group (*i.e.*, workload-level) rather than a single query. To measure the quality of recommended indexes, we focus on two metrics: a) *total cost saving*, which sums up the execution cost reduction for all the query groups after applying the recommended indexes; b) *average improvement ratio*, which is the percentage of cost savings averaged over the groups.

As shown in Figure 6, EDDIE consistently outperforms PG and LIB in terms of the two metrics across all four workloads. For example, in TPC-DS, our method achieves a total cost saving of 130s and an average workload improvement of 116.0%, outperforming both PG (66s, 5.4%) and LIB (69s, 8.7%). Besides, a much larger improvement is obtained under Redbench, which has cloud query characteristics. These results demonstrate that EDDIE not only offers more accurate index benefit estimation but also attains superior end-to-end performance in index tuning tasks. AutoAdmin searches the index space with a unique policy and may produce interim candidate indexes which EDDIE's model never encountered. But EDDIE is still able to achieve accurate prediction, showing its great potential in generalizability and robustness.

VII. Related Work

Index benefit estimation (IBE). Index optimization has been an active research area for decades due to its great value to query performance improvement. Recently, some surveys, *e.g.*, [2], [15], have dissected index advisors into several components and summarized important works related to each component. In this paper, we focus on one of the key components, *i.e.*, IBE, which can significantly affect the quality of index recommendation. Existing IBE methods can be mainly categorized into three types: *what-if-based*, *learning-based* and *hybrid*. The first type requires databases to have what-if capabilities [6], and has been adopted by several commercial database products, like SQL Server [24] [7] and DB2 [4]. However, this approach has been found to be time-consuming [8], inaccurate [16], and have high CPU overhead,

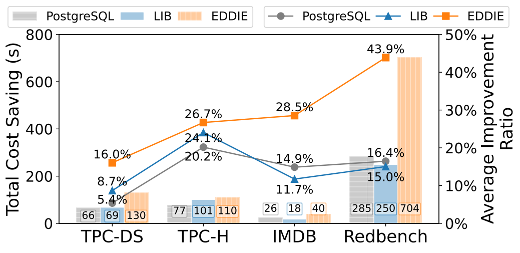

Fig. 6: E2E performance after integration with AutoAdmin

which can adversely affect user business on the database. The second type aims to learn index benefit estimators through ML models, thus completely avoiding what-if invocation. Typical work is LIB [13], which unfortunately suffers from low accuracy and poor generalization to changing workload and column order. Our approach also falls into this type, but addresses these important issues via an HTE approach. The third type is a mix of the above two types, *e.g.*, [12], [14], [16], [29], [35], [36], which still rely on what-if calls but utilize various ML techniques to overcome certain drawbacks of what-if operations. For example, DISTILL [14] proposes to filter out spurious what-if calls via a filtering model; AIMeetsAI [16] leverages a classification model to avoid performance regression due to what-if call's cost estimation errors; RIBE [29] develops a classification model called ChangeFormer to predict query plan structure change, thus deciding whether to launch what-if analysis or not; BALANCE [12] and SWIRL [36] leverages Reinforcement Learning to adapt to workload dynamics, but still relies on what-if calls to determine rewards. Despite these optimizations, the hybrid approaches still suffer from the limitations of what-if analysis, i.e., inaccuracy of cost estimation and inability to apply to databases without hypothetical indexes, like MySQL. What is more, they have limited adaptability to environmental changes. For example, SWIRL [36] can handle dynamic workloads, but is sensitive to schema changes because operators in a query plan are modeled based on text representations.

Learned cost estimation. Another lane of work related to ours is learning-based query cost estimation, such as [25], [31], [34], [37]–[39], in order to replace the traditional statistics-based query optimizer. However, the objective of these approaches is to predict the query execution cost, rather than index benefit, even though some works, e.g., E2E-Cost [25] and Bao [31], also propose Graph Neural Network (GNN)-based approaches to represent query plans. Because of this difference, they neither represent indexes in their features, nor consider the impact of candidate indexes on a query plan or the interactions of multiple candidate indexes. Consequently, they cannot be directly applied to the IBE problem.

VIII. Conclusion

We propose a hierarchical 2D encoding that yields transferable, position-aware feature representations for accurate index benefit estimation, achieving high accuracy and efficiency with less training data via pre-training and fine-tuning. Also, our method is robust under dynamic environments, and attains superior end-to-end index tuning performance.

AI-Generated Content Acknowledgement

AI was not used to generate this paper's content, including text, figures, images, and code. However, AI was used as a general-purpose writing aid to improve readability and correct grammar. All scientific contributions, including technical ideas, methodology implementation, experimental design, and paper writing, were independently developed by the authors. We assume full responsibility for the content of this work.

References

[1] G. Piatetsky-Shapiro, "The optimal selection of secondary indices is np-complete," SIGMOD Rec., vol. 13, no. 2, pp. 72–75, 1983.

[2] Y. Wu, X. Zhou, Y. Zhang, and G. Li, "Automatic database index tuning: A survey," IEEE Transactions on Knowledge and Data Engineering, vol. 36, pp. 7657–7676, 2024.

[3] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri, "Automatically indexing millions of databases in microsoft azure sql database," in Proceedings of the 2019 International Conference on Management of Data, 2019, pp. 666–679.

[4] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley, "Db2 advisor: An optimizer smart enough to recommend its own indexes," in Proceedings of 16th International Conference on Data Engineering (ICDE). IEEE, 2000, pp. 101–110.

[5] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, "Automatic sql tuning in oracle 10g," in Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, 2004, pp. 1098–1109.

[6] S. Chaudhuri and V. Narasayya, "Autoadmin "what-if" index analysis utility," ACM SIGMOD Record, vol. 27, no. 2, pp. 367–378, 1998.

[7] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database tuning advisor for microsoft sql server 2005," in Proceedings of the 2005 ACM SIGMOD international conference on Management of data, 2005, pp. 930–932.

[8] S. Papadomanolakis, D. Dash, and A. Ailamaki, "Efficient use of the query optimizer for automated physical design," in Proceedings of the 33rd international conference on very large data bases, 2007, pp. 1093–1104.

[9] K. Schnaitter, N. Polyzotis, and L. Getoor, "Index interactions in physical design tuning: modeling, analysis, and applications," Proceedings of the VLDB Endowment, vol. 2, no. 1, pp. 1234–1245, 2009.

[10] S. Botros and J. Tinley, High Performance MySQL, 4th ed. O'Reilly Media, Inc., 2021.

[11] W. Zhou, C. Lin, X. Zhou, G. Li, and T. Wang, "Trap: Tailored robustness assessment for index advisors via adversarial perturbation," in 2024 IEEE 40th International Conference on Data Engineering (ICDE). IEEE, 2024, pp. 42–55.

[12] Z. Wang, H. Liu, C. Lin, Z. Bao, G. Li, and T. Wang, "Leveraging dynamic and heterogeneous workload knowledge to boost the performance of index advisors," Proceedings of the VLDB Endowment, vol. 17, no. 7, pp. 1642–1654, 2024.

[13] J. Shi, G. Cong, and X.-L. Li, "Learned index benefits: Machine learning based index performance estimation," Proceedings of the VLDB Endowment, vol. 15, no. 13, pp. 3950–3962, 2022.

[14] T. Siddiqui, W. Wu, V. Narasayya, and S. Chaudhuri, "Distill: low-overhead data-driven techniques for filtering and costing indexes for scalable index tuning," Proceedings of the VLDB Endowment, vol. 15, no. 10, pp. 2019–2031, 2022.

[15] W. Zhou, C. Lin, X. Zhou, and G. Li, "Breaking it down: An in-depth study of index advisors," Proceedings of the VLDB Endowment, vol. 17, no. 10, pp. 2405–2418, 2024.

[16] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya, "Ai meets ai: Leveraging query executions to improve index recommendations," in Proceedings of the 2019 International Conference on Management of Data, 2019, p. 1241–1258.

[17] (2025) Index-only scans and covering indexes. [Online]. Available: https://www.postgresql.org/docs/current/indexes-index-only-scans.html

[18] Y. Zhao, Z. Li, and G. Cong, "A comparative study and component analysis of query plan representation techniques in ml4db studies," Proceedings of the VLDB Endowment, vol. 17, no. 4, pp. 823–835, 2023.

[19] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han, "Automatic view generation with deep learning and reinforcement learning," in 2020 IEEE 36th International Conference on Data Engineering (ICDE). IEEE, 2020, pp. 1501–1512.

[20] Y. Zhao, G. Cong, J. Shi, and C. Miao, "Queryformer: A tree transformer model for query plan representation," Proceedings of the VLDB Endowment, vol. 15, no. 8, pp. 1658–1670, 2022.

[21] T. Parr, The definitive ANTLR 4 reference, 2nd ed. The Pragmatic Bookshelf, 2013.

[22] A. Vaswani, "Attention is all you need," Advances in Neural Information Processing Systems, 2017.

[23] S. Chaudhuri and V. Narasayya, "Anytime algorithm of database tuning advisor for microsoft sql server," 2020.

[24] S. Chaudhuri and V. R. Narasayya, "An efficient, cost-driven index selection tool for microsoft sql server," in Proceedings of the VLDB Endowment, vol. 97. San Francisco, 1997, pp. 146–155.

[25] J. Sun and G. Li, "An end-to-end learning-based cost estimator," Proceedings of the VLDB Endowment, vol. 13, no. 3, pp. 307–319, 2019.

[26] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser, "Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms," Proceedings of the VLDB Endowment, vol. 13, no. 12, pp. 2382–2395, 2020.

[27] S. Krid, M. Stoian, and A. Kipf, "Redbench: A benchmark reflecting real workloads," arXiv preprint arXiv:2506.12488, 2025.

[28] Hypopg. [Online]. Available: https://github.com/HypoPG/hypopg

[29] T. Yu, Z. Zou, W. Sun, and Y. Yan, "Refactoring index tuning process with benefit estimation," Proceedings of the VLDB Endowment, vol. 17, no. 7, pp. 1528–1541, 2024.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in 3rd International Conference on Learning Representations, ICLR, 2015.

[31] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in Proceedings of the 2021 International Conference on Management of Data, 2021, pp. 1275–1288.

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

[33] R. O. Nambiar and M. Poess, "The making of tpc-ds." in VLDB, vol. 6, 2006, pp. 1049–1058.

[34] B. Hilprecht and C. Binnig, "Zero-shot cost models for out-of-the-box learned cost prediction," Proceedings of the VLDB Endowment, vol. 15, no. 11, pp. 2361–2374, 2022.

[35] X. Zhou, L. Liu, W. Li, L. Jin, S. Li, T. Wang, and J. Feng, "Autoindex: An incremental index management system for dynamic workloads," in 2022 IEEE 38th International Conference on Data Engineering (ICDE). IEEE, 2022, pp. 2196–2208.

[36] J. Kossmann, A. Kastius, and R. Schlosser, "Swirl: Selection of workload-aware indexes using reinforcement learning." in EDBT, vol. 2, 2022, pp. 155–2.

[37] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang, "Learned cardinality estimation: A design space exploration and a comparative evaluation," Proceedings of the VLDB Endowment, vol. 15, no. 1, pp. 85–97, 2021.

[38] P. Li, W. Wei, R. Zhu, B. Ding, J. Zhou, and H. Lu, "Alece: An attention-based learned cardinality estimator for spj queries on dynamic workloads (extended)," arXiv preprint arXiv:2310.05349, 2023.

[39] Z. Liang, X. Chen, Y. Xia, R. Ye, H. Chen, J. Xie, and K. Zheng, "Dace: A database-agnostic cost estimator," in 2024 IEEE 40th International Conference on Data Engineering (ICDE). IEEE, 2024, pp. 4925–4937.