Refactoring Index Tuning Process with Benefit Estimation

Tao Yu
Harbin Institute of Technology
21B903056@stu.hit.edu.cn

Weihua Sun
Harbin Institute of Technology
1190200228@stu.hit.edu.cn

Zhaonian Zou
Harbin Institute of Technology
znzou@hit.edu.cn

Yu Yan
Harbin Institute of Technology
yuyan@hit.edu.cn

ABSTRACT

Index tuning is a challenging task aiming to improve query performance by selecting the most effective indexes for a database and a workload. Existing automatic index tuning methods typically rely on "what-if tools" to evaluate the benefit of an index configuration, which is costly and sometimes inaccurate. In this paper, we propose RIBE, a novel method that effectively eliminates redundant queries from the workload and harnesses statistical information of query plans to enable fast and accurate estimation of the benefit of an index configuration. With RIBE, a considerable portion of what-if calls can be skipped, thereby reducing index tuning time and increasing estimation accuracy. At the heart of RIBE is a deep learning model based on attention mechanism that predicts the impact of indexes on queries. A practical advantage of RIBE is that it achieves both improved accuracy of benefit estimation and time savings without making any changes to DBMS implementation and index configuration enumeration algorithms. Our evaluation shows that RIBE can achieve competitive tuning results and 1–2 orders of magnitude faster performance compared with the tuning method based on the full workload, and RIBE also attains higher tuning quality and comparable efficiency against the tuning methods based on the state-of-the-art workload compression methods.

1 INTRODUCTION

Index tuning is the task of selecting the most effective indexes to speed up query processing while minimizing storage overhead [39], which is known to be NP-hard [3]. Unsuitable indexes may lead to increased query execution time and reduced performance of database systems [11]. Due to the vast amount of potential index

configurations (sets of indexes) and the complexity of query workloads, enormous time and resources are required to find the optimal index configuration for a given workload [8].

Existing automatic index tuning methods typically list all possible indexes that can be built on the workload and continuously enumerate index configurations using pre-defined strategies [2, 9] or reinforcement learning [16, 25]. For each enumerated index configuration, virtual indexes are generated using the what-if tools [4], and the estimated costs of the queries in the workload given the virtual indexes is computed by the query optimizer to determine the next index configuration to be enumerated. The calls to the what-if APIs take a majority of index tuning time [11, 27].

Traditionally, the studies on index tuning focus on improving the effectiveness of index configuration enumeration [2, 5–7, 9, 22–24, 26, 31]. Most studies treat the what-if tool as a block box and overlook its overhead and accuracy. Recently, the researches start to take the overheads and the accuracy of what-if calls into account.

One approach is reducing what-if calls by compressing the workload to keep only the "essential" queries. The optimal index configuration for the compressed workload is expected to be as effective as the optimal one for the full workload. GSUM [10] formalizes the representativeness and the coverage of the compressed workload, aiming to preserve the distribution of characteristics of the original workload while including both common queries and outliers. ISUM [28] formulates the benefit of queries for index tuning and greedily selects the queries with the maximum benefit to form the compressed workload. Although these algorithms can significantly reduce the number of queries, they prefer to queries that can be substantially optimized with indexes, so useful information in the discarded queries is inevitably lost. As a result, these methods often lead to sub-optimal tuning results especially when an adequate budget is given on index tuning.

An alternative approach is reducing the number of potential indexes as it determines the number of index configurations. DISTILL [29] employs heuristic rules and machine learning models to filter out indexes with low benefits. In addition, an individual cost estimation model is trained online for each group of similar queries in the workload. The what-if calls can be partially replaced by applying these models. However, maintaining numerous individual cost estimation models is complex and error-prone, but it is challenging to learn a universal model with a good generalization ability to unseen queries. Moreover, DISTILL is not compatible with workload compression as eliminating duplicate queries makes it less effective to learn the cost estimators in DISTILL.

Another approach is incorporating the overheads of what-if calls into index configuration enumeration. Wu et al. [36] propose

budget-aware index tuning that enumerates index configurations by Monte Carlo Tree Search within a given number of what-if calls.

In this paper, we propose RIBE, a new method to refactor index tuning process with benefit estimation. This method is designed to handle both redundant queries in the workload and frequent costly what-if calls. Our key observation is that, for a considerable portion of queries, creating indexes may not change the structure of a query plan but just alter a few operators, e.g. replacing a sequential scan with an index scan. The original and the altered plans therefore share the same intermediate results on every pair of corresponding operators. In this situation, it is unnecessary to compute the estimated cost of the altered plan using the what-if tool and the query optimizer. Instead, the estimated cost can be computed very fast and accurately based on the actual statistics (cardinalities, costs, and so on) of the operators in the original plan (section 4). By exploiting the actual statistics, the estimated cost is more accurate than that evaluated by the optimizer as the optimizer relies on estimated statistics which are sometimes very inaccurate [21, 33–35].

In RIBE, the operator-level actual statistics of the query plans are stored in a set of representations called workload matrices. The workload matrices support not only the aforementioned evaluation of estimated costs but also workload compression. The queries in the workload are first clustered according to their operator-level actual statistics in the workload matrices. Then, the workload is compressed based on the cluster centers. Unlike the existing compression methods, our compression method is based on the operational characteristics of query plans, not biased towards queries that can be significantly optimized with indexes.

In addition, RIBE is easy to deploy as it needs not to modify the implementation of the query optimizer and can work with any index configuration enumeration algorithms [2, 5–7, 9, 22–24, 26, 31, 36].

The paper makes the following technical contributions:

(1) The traditional index tuning process is refactored by introducing workload matrices (section 3), operator-level workload representations. The workload matrices support both workload redundancy elimination and what-if call reduction with a clustering-based workload compression method and a fast and accurate method for evaluating the estimated costs of query plans (section 4).

(2) A deep learning model called ChangeFormer is learned to predict if an index configuration will have the structure of a query plan changed (section 5). The prediction decides to compute the estimated cost of a query plan by the query optimizer or our fast and accurate method based on workload matrices. ChangeFormer applies the self-attention mechanism to represent query plans, join schemas and index configurations which is superior to the state-of-the-art query representation schemes [18, 19, 38].

(3) An extensive evaluation is performed on RIBE. RIBE can achieve competitive tuning results and 1–2 orders of magnitude faster performance compared to the tuning method based on the full workload. RIBE also leads to higher tuning quality and comparable efficiency against the tuning methods based on the state-of-the-art workload compression methods (section 6).

2 BACKGROUND

The component in a DBMS for index tuning is called *index advisor*. The typical process of index tuning is illustrated in Figure 1.
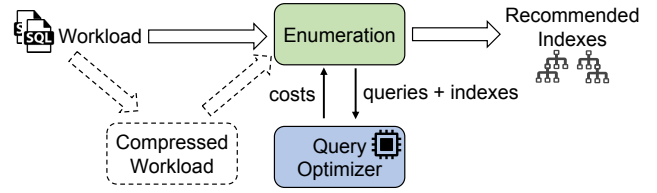

Figure 1: Traditional process of index tuning.

(1) A query workload W is collected during a certain period under the current index configuration $\mathcal{I}_0$. An index configuration refers to a set of indexes on a database. For each query $q \in W$, the SQL statement and the execution plan of q are acquired. According to the size of W and the configuration of index tuning, workload compression can be applied to reduce the number of queries in W, while keeping the original information in W as much as possible.

(2) A set of candidate indexes are enumerated according to the database schema and the tables and the attributes involved in the queries in W. Each candidate index I is related to a base table T that is involved in a query $q \in W$, and the index key of I is made up of an attribute or a list of attributes in T that occur in q.

(3) Given a budget on the number of indexes or the storage space occupied by the indexes to be created, the enumerator continually produces index configurations within the budget constraint.

(4) For each new index configuration $\mathcal{I}$ produced by the enumerator, we estimate the cost of sequentially executing all queries in W given that the indexes in $\mathcal{I}_0$ are replaced with the indexes in $\mathcal{I}$ by calling a "what-if" API of the query optimizer that does not really build the indexes in $\mathcal{I}$. The enumerator's behavior can be affected by the estimated cost returned by the optimizer. The enumeration stops once the termination condition of index tuning is met, e.g., the maximum enumeration time is reached.

Problem Formulation. Let D be a database. For a query q on D and a set $\mathcal{I}$ of indexes created on D, the cost of evaluating q with the support of $\mathcal{I}$ is denoted by $c(q, \mathcal{I})$. Let $\mathcal{I}_0$ be the current set of indexes created on D. The *benefit* of changing $\mathcal{I}_0$ to $\mathcal{I}$ with respect to q is $B(q, \mathcal{I}_0, \mathcal{I}) = c(q, \mathcal{I}_0) - c(q, \mathcal{I})$. Let $W = \{q_1, q_2, \ldots, q_n\}$ be a workload on D, where each query $q_i \in W$ is weighted by $w_i \in \mathbb{R}$ according to the frequency and the importance of q_i. Given $\mathcal{I}_0$, W and the maximum number k of indexes that can be created on D, the index tuning problem is to find an index configuration $\mathcal{I}$ that has $|\mathcal{I}| \leq k$ and maximizes the *total benefit*

$$B(W, \mathcal{I}_0, \mathcal{I}) = \sum_{i=1}^{n} w_i \cdot B(q_i, \mathcal{I}_0, \mathcal{I}). \tag{1}$$

Maximizing $B(W, \mathcal{I}_0, \mathcal{I})$ is equivalent to minimizing the *total cost*

$$c(W, \mathcal{I}) = \sum_{i=1}^{n} w_i \cdot c(q_i, \mathcal{I}). \tag{2}$$

Given an arbitrary deterministic algorithm A that enumerates an index configuration for an input workload, the *workload compression* problem aims to find a subset $W_m \subseteq W$ with $|W_m| \leq m$ such that the index configuration $A(W_m)$ returned by A based on W_m achieves the maximum total benefit $B(W, \mathcal{I}_0, A(W_m))$ on W. If A is a randomized algorithm, W_m should maximize the expected total benefit $E[B(W, \mathcal{I}_0, A(W_m))]$.

Node	Type	Attributes	Selectivity	Cost	Benefit
1	Agg.	{B.val}	1	20	0
2	HashJoin	{B.cid, C.id}	0.02	15000	0
3	HashJoin	{A.id, B.aid}	0.46	300	0
4	Hash	{C.id}	1	0.1	0
5	Hash	{A.id}	1	0.1	0
6	SeqScan	Ø	1	900	0
7	SeqScan	Ø	1	890	0
8	SeqScan	{B.Val}	0.85	2000	300

(a) Tables and SQL statement (b) No index (c) Index(B.val) (d) Index(B.val, B.cid) (e) Statistics and Estimated Benefit
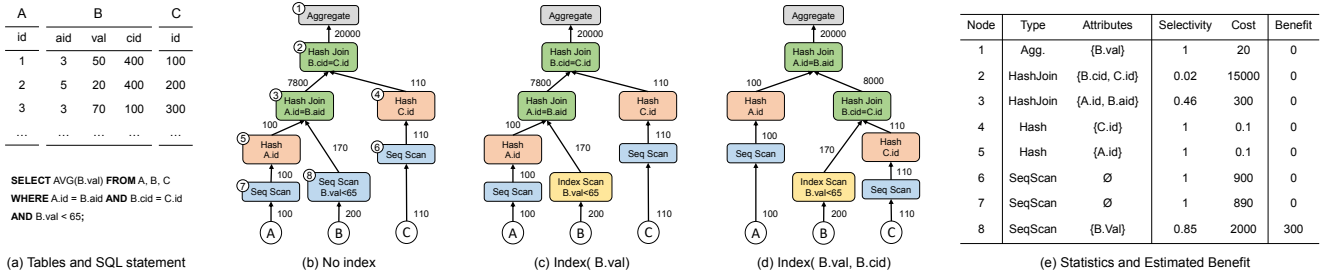
Figure 2: An example of how indexes affect query plans. (a) A database of size 1GB and an SQL query. Unrelated attributes are omitted. (b) The plan of the query when no index has been created. The number beside each arrow indicates the cardinality of transmitted tuples. (c) The plan of the query after creating an index on (B.val). (d) The plan of the query after creating a composite index on (B.val, B.cid). (e) The actual statistics of all plan nodes in (b) and the estimated benefits of creating an index on (B.val) to all plan nodes.

3 OVERVIEW OF OUR SOLUTION

Design Goal. In the traditional index tuning process depicted in Figure 1, the index advisor executes a lot of calls to the what-if API of the query optimizer. Suppose the index configuration enumerator generates totally m index configurations $\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_m$. Let $R(W, \mathcal{I}_i)$ be the set of queries in W whose execution may be affected by one or more indexes in $\mathcal{I}_i$. For each query $q \in R(W, \mathcal{I}_i)$, the index advisor invokes a what-if call to estimate the cost $c(q, \mathcal{I}_i)$. Therefore, the total number of what-if calls is $\sum_{i=1}^{m} |R(W, \mathcal{I}_i)|$. As tested in [36], the time spent on what-if calls accounts for approximately 80% of the total execution time of index tuning. We aims to decrease the number of what-if calls in two ways while improving accuracy. One way is to invoke what-if calls only for a fraction of queries rather than all queries in $R(W, \mathcal{I}_i)$. The other way is to decrease the number of queries in W by workload compression.

Fundamental Idea. The fundamental idea of our index tuning method RIBE is as follows: Note that there are two types of changes in query plans after creating indexes. (1) The indexes do not change any intermediate results generated by the query plan, i.e., the data transmitted between query plan nodes, but only alter some data access paths to improve read performance or make some operations access data in a sorted manner. (2) The indexes alter the structure of the query plan tree, e.g., changing the join order of tables, thereby resulting in different intermediate results.

Consider the example in Figure 2. Figure 2(a) gives a database and a query. When no index has been created, the plan of the query is depicted in Figure 2(b). If an index is created on attribute B.val, the plan is changed to the one shown in Figure 2(c), whose structure is the same as the old plan in Figure 2(b), but the SeqScan operator on table B is replaced by IndexScan. The index only affects node 8 that is responsible for data retrieval. Therefore, the intermediate results generated by the new plan remain the same as the old plan, and only the execution time of the nodes in the old plan is changed. In this case, it is unnecessary to make a what-if call to the query optimizer. Instead, as will be presented in subsection 4.2, a more precise and efficient estimation of the benefit led to be the index can be computed based on the statistics of the database and the sizes of the intermediate results produced by the old plan. This method helps avoid unnecessary what-if calls.
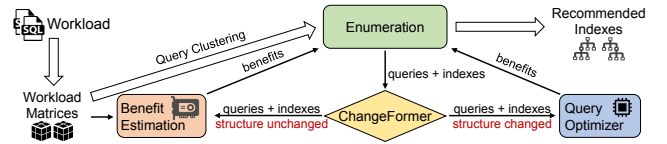

Figure 3: New process of index tuning.

If a composite index is created on the attribute list (B.val, B.cid), the plan in Figure 2(b) is changed to the one shown in Figure 2(d), in which the join order is changed. Therefore, we have to estimate the cost of the new plan via a what-if call.

New Architecture of Index Advisor. Based on the above idea, we devise a novel index advisor which works as illustrated in Figure 3. The new index tuning process is different from the traditional one depicted in Figure 1 in the following aspects:

(1) **Workload Matrices.** We design an internal representation of workload that represents the execution plans of all queries in a workload using a compact data structure called "*workload matrix*". As depicted in Figure 3, after collecting a workload W, we represent W by several workload matrices. The workload matrices play three important roles in the new index tuning advisor.

First, given a query $q \in W$ and an index configuration $\mathcal{I}$ enumerated by the index enumerator, if the plan of q will not be changed by replacing the current indexes in $\mathcal{I}_0$ with the indexes in $\mathcal{I}$, the advisor can estimate the benefit $B(q, \mathcal{I}_0, \mathcal{I})$, i.e., the improvement in the cost of q, based on the workload matrices. As will be evaluated in subsection 6.4, this method is not only faster but also more accurate than invoking a what-if call. In addition, this method can be run in parallel with the query optimizer to estimate benefits simultaneously, further increasing efficiency.

Second, the index advisor can utilize the workload matrices to efficiently cluster queries with similar impacts on index tuning and remove unnecessary queries, thereby reducing the size of W.

Third, unlike what-if calls that are unable to be executed in parallel, computations on a workload matrix can be easily parallelized. Thus, the total benefit $B(W', \mathcal{I}_0, \mathcal{I})$ with respect to a subset $W' \subseteq W$ can be estimated based on the workload matrices in parallel rather than being estimated by the query optimizer sequentially.

(2) **ChangeFormer.** The introduction of workload matrices brings up a key problem: should the benefit $B(q, \mathcal{I}_0, \mathcal{I})$ be estimated based on the workload matrices or by the query optimizer? The decision mainly depends on if the execution plan of q will be significantly changed by replacing $\mathcal{I}_0$ with $\mathcal{I}$. To make correct choices, we design a deep learning model called ChangeFormer based on the tree-structured transformer [38]. Given the workload W and a set of enumerated index configurations, we use ChangeFormer to predict in parallel if the plan structure of each query in W will be changed by each given index configuration. ChangeFormer is very efficient. Due to the parallel processing power of GPUs, Change-Former can make predictions for thousands of pairs of query and index configuration within a few milliseconds.

4 WORKLOAD MATRICES

In this section, we introduce *workload matrix*, a new representation of workload, and propose a new benefit estimation method based on workload matrices.

4.1 Foundations of Benefit Estimation

Let P_q be the execution plan of a query q under the current index configuration $\mathcal{I}_0$. The structure of P_q is a tree of plan nodes. Each node is an operator in P_q. For a node N, let t_N be the operator type of N, I_N be the input of N, and O_N be the output of N. Let $c(t_N, I_N)$ denote the cost of executing N on the input I_N. Therefore, the cost of P_q is formulated by

$$c(P_q) = \sum_{N:N \text{ is a node in } P_q} c(t_N, I_N). \qquad (3)$$

The formula of $c(t_N, I_N)$ is determined by t_N and is formulated based on two types of variables. One type of variables is the cost parameters that estimate the costs of individual physical operations such as disk page fetches, tuple processing and index entry processing, which are specified by the DBMS by default and can be modified by users. The other type of variables are the numbers of physical operations in each type performed during the execution of N, which are determined by the statistics of I_N and O_N, e.g., the numbers of pages and tuples. For example, PostgreSQL formulates the cost of a SeqScan operator without filter conditions as seq_page_cost × relation_pages + cpu_tuple_cost × output_tuples, where seq_page_cost is the estimated cost of a disk page fetch, cpu_tuple_cost is the estimated cost of processing a tuple, relation_pages is the number of pages in the input, and output_tuples is the number of output tuples.

Replacing the current index configuration $\mathcal{I}_0$ with a new index configuration $\mathcal{I}$ may change the current plan P_q to a new plan P'_q with $c(P'_q) < c(P_q)$. If P_q and P'_q have different plan structures, it is inevitable to use the query optimizer to estimate the benefit $B(q, \mathcal{I}_0, \mathcal{I}) = c(P_q) - c(P'_q)$. However, if P_q and P'_q have the same plan structure, but some corresponding nodes are of different operator types, such as the plans shown in Figure 2(b) and Figure 2(c), $B(q, \mathcal{I}_0, \mathcal{I})$ can be estimated in a more efficient way. Let $\mathcal{N}$ be the set of nodes in P_q whose operator types are changed in P'_q. For $N \in \mathcal{N}$, let t'_N be the operator type of N in P'_q. We have

$$B(q, \mathcal{I}_0, \mathcal{I}) = c(P_q) - c(P'_q) = \sum_{N \in \mathcal{N}} c(t_N, I_N) - c(t'_N, I_N). \qquad (4)$$

As P_q and P'_q have the same plan structure, every common node N in P_q and P'_q has the same input I_N and the same output O_N. Since the costs $c(t_N, I_N)$ of all nodes $N \in \mathcal{N}$ have already been known, estimating $B(q, \mathcal{I}_0, \mathcal{I})$ reduces to estimating $c(t'_N, I_N)$ for all $N \in \mathcal{N}$. Directly using the query optimizer to estimate $c(t'_N, I_N)$ attains two main disadvantages:

(1) When estimating $c(t'_N, I_N)$, the optimizer must fetch from the catalog or estimate the statistics required by the formula of $c(t'_N, I_N)$. However, the statistics estimators in a DBMS such as the cardinality estimator are really inaccurate in some situations. For example, the estimated cardinality of a query can be over 10^4 times of its actual cardinality [33]. Moreover, when the plan tree is large, the errors in the estimated statistics accumulate gradually and can lead to inaccurate estimates that significantly deviate from their actual values. In fact, in the scenario of index tuning, every query $q \in W$ must have been executed under $\mathcal{I}_0$. Therefore, the plan P_q of q under $\mathcal{I}_0$ and the actual statistics of all nodes in P_q can be kept with W and reused when estimating $c(t'_N, I_N)$.

(2) In the typical design and implementation of query optimizers, what-if calls are handled sequentially. Therefore, the optimizer can only estimate the costs $c(t'_N, I_N)$ of all nodes N in P'_q sequentially rather than in parallel.

In addition, a lot of index configurations are enumerated during index tuning. Even if an index configuration is unlikely to change the structure of the current plan P_q, the optimizer still has to re-estimate the cardinalities of intermediate results to generate alternative plans of q and select the plan P'_q with the minimum estimated cost. If P'_q has the same plan structure as P_q, the entire work performed by the optimizer is wasted because the benefit $B(q, \mathcal{I}_0, \mathcal{I}) = c(P_q) - c(P'_q)$ can be easily estimated by Eq. (4).

Consequently, our new index advisor calls for new methods to undertake the following tasks:

(1) Determining if the plan structure of P_q can be changed by a new index configuration $\mathcal{I}$ without using the optimizer.

(2) Estimating the benefit $B(q, \mathcal{I}_0, \mathcal{I}) = c(P_q) - c(P'_q)$ without using the optimizer when P_q and P'_q have the same plan structure.

(3) Parallelizing the execution of the two tasks above.

4.2 Workload-Matrix-based Benefit Estimation

Let P_q and P'_q be the plans of a query q under the current index configuration $\mathcal{I}_0$ and a new index configuration $\mathcal{I}$, respectively. Suppose P_q and P'_q have the same plan structure. In this subsection, we propose a novel method to fast estimate the benefit $B(q, \mathcal{I}_0, \mathcal{I}) = c(P_q) - c(P'_q)$ without using the query optimizer.

Straightforward Method. Recall the formula of $B(q, \mathcal{I}_0, \mathcal{I})$ given in Eq. (4). In the scenario of index tuning, P_q must have been executed under $\mathcal{I}_0$. Therefore, for each node $N \in \mathcal{N}$ in Eq. (4), the actual statistics of the input I_N and the output O_N of N have already been known and can be reused to estimate $c(t'_N, I_N)$ because N has the same input I_N and the same output O_N in P'_q. This straightforward approach ensures the accuracy of $c(t'_N, I_N)$ by incorporating the actual statistics instead of their estimated values. However, this method has two intrinsic drawbacks:

(1) This method has a very high space overhead because it must store all actual statistics for all nodes $N \in \mathcal{N}$.

(2) This method is faced with a compatibility issue. It must ensure that the formula of $c(t'_N, I_N)$ is the same as the one used by the optimizer for the operator type t'_N. However, such formulas may vary across various database engines or hardware configurations. **Approximate Method.** To overcome the drawbacks of the straightforward method, our new index advisor adopts a fast approximate method to compute an estimate of $c(t'_N, I_N)$, which does not rely on all actual statistics and is independent of the cost formulas specified by the DBMS. Basically, if the execution of N can be accelerated by some enumerated indexes, the improvement $c(t_N, I_N) - c(t'_N, I_N)$ can be estimated by $\alpha \cdot c(t_N, I_N)$, where $0 \le \alpha \le 1$ is a multiplicative factor. According to the previous work [29] and our empirical evaluation, α is linear to the *selectivity* of N. Let N be an n-ary operator on n input relations or intermediate results $R_1, R_2, \ldots, R_n$. The selectivity of N is $\theta(N) = |O_N|/\prod_{i=1}^{n} |R_i|$. If R_i is a base table, $|R_i|$ can be fetched from the catalog. If R_i is an intermediate result output by a child node N_i of N in P_q, we have $|R_i| = |O_{N_i}|$, which can been stored for reuse later.

Siddiqui et al. [29] simply formulate the multiplicative factor α as $1 - \theta(N)$. However, this formulation is independent of the operator type of N and the extent to which N is supported by I, as well as the hardware and the DBMS configuration that affect the performance of query execution. Instead, we consider the impacts of these practical factors on α and formulate α as

$$\alpha = r_N \cdot (1 - \theta(N, I)) + b_N, \tag{5}$$

where $r_N \in \mathbb{R}$ and $b_N \in \mathbb{R}$ depend on the operator type of N, the hardware and the DBMS configuration. The coefficients r_N and b_N can be set using the method described in subsection 5.6 based on the performance information of historical queries. The term $\theta(N, I)$ generalizes the selectivity $\theta(N)$ by considering the impacts of I on $\theta(N)$ when the index keys in I only cover a portion of attributes accessed by N. In this case, the execution of N can be regarded to be composed of two phases. In the first phase, a set of intermediate results, denoted as O_N^I, is retrieved based on the indexes such that $O_N \subseteq O_N^I$. In the second phase, the tuples in O_N^I are filtered to obtain the output O_N of N. Indexes are only useful for the first phase. Hence, $\theta(N)$ is generalized to $\theta(N, I)$ as

$$\theta(N, I) = \frac{|O_N^I|}{\prod_{i=1}^{n} |R_i|} = \theta(N) \cdot \frac{|O_N^I|}{|O_N|}. \tag{6}$$

When all the attributes filtered by N are covered by the index keys in I, we have $\theta(N, I) = \theta(N)$. The term $|O_N^I|$ is estimated by sampling in our implementation. Therefore, Eq. (5) is independent of the cost formulas specified by the DBMS, thereby addressing the compatibility issue of the straightforward method.

To accurately and efficiently compute $\theta(N, I)$, we store $|O_N|$, the number of tuples actually returned by N, for all nodes N in P_q after P_q is actually executed under the current index configuration I_0. Besides, we store $c(t_N, I_N)$, the actual cost of N in P_q, obtained after P_q is actually executed under I_0. Therefore, our method only stores a fraction of actual statistics, thereby overcoming the high space overhead of the straightforward method.

Our experimental evaluation in subsection 6.4 verifies that the accuracy of our approximate method (even for $r_N = 1$ and $b_N = 0$, that is, $\alpha = 1 - \theta(N, I)$) is comparable to that of the query optimizer when P_q and P'_q have the same plan structure.

Workload Matrices. To support parallel benefit estimation for a lot of plan nodes, we store the actual statistics required by the approximate benefit estimation method in matrices called *workload matrices*. For all plan nodes N in all queries in the input workload W, the actual cost $c(t_N, I_N)$ is stored in the workload matrix $\mathbf{C}$, and the selectivity $\theta(N)$ of N is pre-computed and stored in the workload matrix $\mathbf{S}$. Both $\mathbf{C}$ and $\mathbf{S}$ are 3-dimensional sparse matrices. Each plan node N is uniquely identified by 3 features: the ID of the query, the operator type of N and the attributes filtered by N. These 3 features are uniquely mapped to indexes on 3 dimensions of the workload matrices. For example, in the query plan shown in Figure 2(b), node 7 is a sequential scan, so its type is SeqScan. Since this table scan has no filter, the attribute set filtered by this operator is $\emptyset$, and it cannot be accelerated by any indexes. Thus, the elements of the workload matrices $\mathbf{C}$ and $\mathbf{S}$ for node 7 are $\mathbf{C}[q, \mathsf{SeqScan}, \emptyset] = 890$ and $\mathbf{S}[q, \mathsf{SeqScan}, \emptyset] = 1$, respectively. Node 3 is a join on the condition A.id = B.id, and its type is HashJoin. The attributes filtered by this node are (A.id, B.id). Thus, the elements of $\mathbf{C}$ and $\mathbf{S}$ for node 3 are $\mathbf{C}[q, \mathsf{HashJoin}, (\mathsf{A.id}, \mathsf{B.id})] = 890$ and $\mathbf{S}[q, \mathsf{HashJoin}, (\mathsf{A.id}, \mathsf{B.id})] = 0.46$, respectively. Figure 2(e) lists the actual statistics related with all plan nodes in Figure 2(b).

The workload matrices are constructed simultaneously as the queries in W are executed under the current index configuration I_0. The matrices are of bounded size. The size of the first dimension (query ID) is $|W|$. The size of the second dimension (operator type) is at most the total number of operator types (typically less than 32). The size of the third dimension (filtered attributes) does not exceed the number of all possible indexes. Due to the bounded size and the simplicity of the workload matrices, constructing the matrices is 3 orders of magnitude faster than the entire index tuning process. Updating the matrices can also be done very efficiently. When a query is added to W, a new slice corresponding to the query is appended to each matrix; when a query is deleted from W, the slice corresponding to the query is removed from each matrix. In addition, the construction process can be easily parallelized.

Parallel Benefit Estimation based on Workload Matrices. For an index configuration I enumerated by the advisor, let $U \subseteq W$ be the subset of queries whose plan structures are not altered by replacing the current indexes in I_0 with the indexes in I. The benefit $B(U, I_0, I)$ of replacing I_0 with I with respect to U can estimated based on the workload matrices $\mathbf{C}$ and $\mathbf{S}$ in parallel.

Let $\mathbf{x}$ be the indexes in the first dimension of the workload matrices corresponding to all nodes in the plans P_q of all queries $q \in U$, and let $\mathbf{z}$ be the indexes in the third dimension of the workload matrices corresponding to the attribute lists that can be accelerated by I. Based on $\mathbf{x}$ and $\mathbf{z}$, we can identify the nodes whose filter attributes are fully or partially covered by I. For each of these nodes N, we estimate the cardinality of O_N^I, calculate $\theta(N, I)$ by Eq. (6) and set the element of the matrix $\mathbf{S}$ corresponding to N to $\theta(N, I)$. Then, the benefits $B(q, I_0, I)$ for all queries $q \in U$ can be approximated in parallel by the following equation:

$$\mathbf{B} = \mathbf{C}[\mathbf{x}, :, \mathbf{z}] \circ ((\mathbf{1} - \mathbf{S}[\mathbf{x}, :, \mathbf{z}]) \circ \mathbf{r}[\mathbf{x}, :, \mathbf{z}] + \mathbf{b}[\mathbf{x}, :, \mathbf{z}]), \tag{7}$$

where $\circ$ is the Hardamard product, i.e., the element-wise product, $\mathbf{1}$ is the matrix of 1's with the same shape as $\mathbf{S}[\mathbf{x}, :, \mathbf{z}]$, and $\mathbf{r}$ and $\mathbf{b}$ are the matrices of the coefficients r_N and b_N for all plan nodes N of all queries in W. Finally, the total benefit $B(U, I_0, I)$ can be

computed as the weighted sum of all elements of **B** because

$$\sum_i w_i \cdot \mathbf{B}_i = \sum_{q \in U} w_q \cdot \left(\sum_{N:N \text{ is a node in } P_q} c(t_N, I_N) - c(t'_N, I_N)) \right)$$
$$= B(U, \mathcal{I}_0, \mathcal{I}).$$

For all queries $q \in W \setminus U$, the benefit $B(q, \mathcal{I}_0, \mathcal{I})$ has to be estimated by the query optimizer via what-if calls.

4.3 Workload-Matrix-based Query Clustering

The workload matrices contain underlying information about the workload W, which can not only be used for index benefit estimation but also in many other tasks. In many practical applications, queries are formulated based on templates, and thus, many queries in W are similar or even redundant. By clustering similar queries in W and compressing them, the size of W can be substantially reduced, and a significant decrease in index tuning time can be achieved. Specifically, for each query $q \in W$, we use the cost matrix $\mathbf{C}[q, :, :]$ as the feature of q. Then, all queries in W are clustered according to their features. For each cluster C, only the clustroid q of C is selected into the compressed workload for index tuning, and the weight of q is computed as:

$$w(q) = \frac{\sum_{q_i \in C} w_i \cdot c(q_i, \mathcal{I}_0)}{c(W, \mathcal{I}_0)}, \tag{8}$$

where w_i is the weight of the query q_i given in W. It is worth noting that queries q from the same template may have significantly different cost distributions in $\mathbf{C}[q, :, :]$ due to various parameter settings. As a result, a number of representative queries from the same template can be retained in different clusters.

5 DETECTION OF CHANGES IN PLAN STRUCTURES

Given the current execution plan P_q of a query $q \in W$ and an index configuration $\mathcal{I}$, the structure of the plan P_q may be changed by some indexes in $\mathcal{I}$. Due to the complicated relationships between indexes and query plans, it is impossible to pre-define a set of rules to determine if an index will cause a change in the structure of a plan. In RIBE, we do not use the "heavyweight" query optimizer to detect such changes because the optimizer has to enumerate a large number of alternative query plans. Instead, we formulate the problem of predicting if the structure of P_q will be changed by some indexes in $\mathcal{I}$ as a binary classification problem in machine learning and build a "lightweight" classification model called ChangeFormer to make predictions. In this section, we introduce the design of ChangeFormer.

5.1 Challenges

A classification model to predict whether the structure of P_q will be changed under the given index configuration $\mathcal{I}$ requires the information about P_q, $\mathcal{I}$ and the database schema as input. Meanwhile, the model needs to have a reasonable structure to ensure that it can be trained to acquire useful knowledge. Here, we assume that all indexes in $\mathcal{I}$ are relevant to q because irrelevant indexes cannot affect the plan of q, and therefore, can be removed from $\mathcal{I}$.

On the contrary, keeping irrelevant indexes in $\mathcal{I}$ may decrease the accuracy of the classification model.

In recent years, there have been many studies in the field of AI4DB [18–20, 38] that focus on encoding query plan information. These studies utilize a vector as the representation of the query plan P_q and have developed various approaches according to the characteristics of the tree structures of query plans. These methods are capable of capturing parent-child information of nodes and preventing information loss caused by long paths in P_q. However, there is still room for improvement in these methods, and they need to be integrated into our model in a reasonable manner. Overall, our model design faces the following challenges:

Challenge 1. If q is a join query, the plan P_q only specifies one way to join the tables involved in q. In fact, there are many alternative ways to join these tables. Creating indexes may change the join order. Following the terminology in [37], a possible way to join tables is called a *join schema*. The existing query plan encoding methods only represent the join schema specified by P_q. However, to build an accurate classification model, it is not enough to only give the join schema specified in P_q as input because all join schemas except this one are unknown to the model.

Challenge 2. The complicated relationships between the operators in P_q, the indexes in $\mathcal{I}$ and the join schemas determine the structure of P_q. The join schemas characterize all potential join orders for q, and various indexes in $\mathcal{I}$ selectively influence the costs of the nodes in P_q. They jointly determine the structure of P_q. On the one hand, the same operator with different costs may cause different plans to use different join orders. On the other hand, even if the join order is fixed, changes in the positions of non-join nodes in the plan can still occur, affecting the structure of P_q. Therefore, the design of the model architecture should take into account the complicated interactions between the operators in P_q, the indexes in $\mathcal{I}$ and the join schemas.

Challenge 3. A binary classifier can produce both false positives and false negatives. A false positive refers to that the structure of P_q actually cannot be changed by $\mathcal{I}$ but is predicted to be changed. It implies that the cost improvement of q will be estimated using the optimizer instead of using our method proposed in subsection 4.2, which will increase the time of index tuning. A false negative refers to that the structure of P_q will actually be changed by $\mathcal{I}$ but is predicted to be unchanged. It implies that the cost improvement of q will be estimated using our method proposed in subsection 4.2 instead of using the optimizer, which may be inaccurate and will degrade the quality of index tuning. Therefore, we prefer to decrease the false negative rate of the classifier.

5.2 Model Architecture

To solve the prediction problem, we design a tree-structured classification model called ChangeFormer based on the Transformer framework [32, 38]. Figure 4 depicts the architecture of ChangeFormer. The input of ChangeFormer includes three parts:

(1) The plan tree P_q. The plan tree P_q represents the execution process of q under the current index configuration $\mathcal{I}_0$. P_q is composed of nodes, where each node contains partial information about the execution process such as the actual statistics of the node.
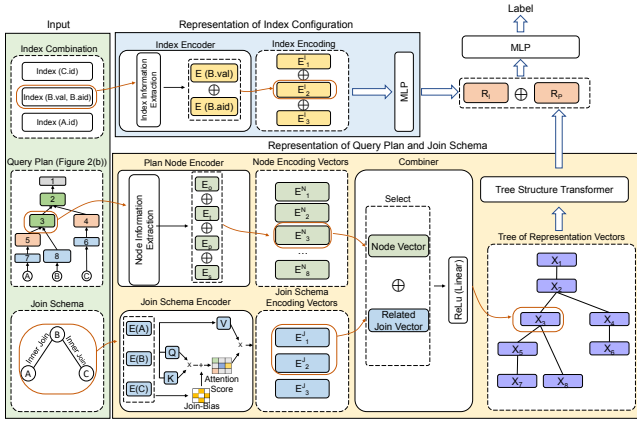
Figure 4: The architecture of ChangeFormer.

(2) The index configuration $\mathcal{I}$. Each index in $\mathcal{I}$ is represented by the table on which the index is built and the index key.

(3) All join schemas of the joinable tables in q. A join schema is represented as a graph, where the vertices represent the tables, and each edge connects two tables directly joined in P_q.

We adopt the following process to encode and integrate these three parts of input and pass them to ChangeFormer:

Step 1: The plan node encoder encodes each node N_i in P_q as a vector E_i^N. The index encoder encodes each index $I_j \in \mathcal{I}$ as a vector E_j^I. The join schema encoder encodes each join schema J_k as a vector E_k^J. These encoding procedures will be described in details in subsections 5.3–5.5.

Step 2: These encoding vectors are combined. There are various ways to combine them. We do not simply concatenate them because it overlooks the tree structure of P_q and the relationships among plan nodes, indexes and join schemas. Instead, our approach first augment the representation of each join node in P_q with its related join schemas. Specifically, for each join node N_i in P_q, we identify the join schemas J_k that are related to N_i (see §5.4) and compute the augmented representation vector X_i of N_i by applying a single-layer perceptron to the vector obtained by concatenating the plan node vector E_i^N with the related join schema vectors E_k^J, that is,

$$X_i = \mathrm{ReLu}\left(\mathrm{Linear}\left(E_i^N \oplus \left(\bigoplus_{J_k} E_k^J\right)\right)\right), \qquad (9)$$

where $\oplus$ is the concatenation operation of vectors, and ReLu and Linear form a single-layer perceptron. To ensure that the augmented representation vectors of all nodes in P_q have the same length, for each non-join node N_i in P_q, its encoding vector E_i^N is padded with 0's to form the augmented representation X_i of N_i. Now, we obtain a tree of augmented representation vectors X_i of all plan nodes N_i in P_q, which has the same tree structure as P_q. Then, the tree is given as input to the tree-structured transformer, and the transformer returns a vector R_P representing P_q and the join schemas.

Step 3: We concatenate the encoding vectors E_j^I of all indexes $I_j \in \mathcal{I}$ and input them into a multi-layer perceptron (MLP) to

obtain the representation vector R_I of the index configuration $\mathcal{I}$, as detailed in subsection 5.5.

Step 4: Our decoder, which is also an MLP, takes the concatenated vector of R_I and R_P as input and returns a label in $\{0, 1\}$. Label 1 indicates that the plan structure of P_q will be changed by some indexes in $\mathcal{I}$, and label 0 implies that the plan structure of P_q will not be changed.

In Steps 1 and 2, all join schemas are encoded, thus addressing Challenge 1 presented in subsection 5.1. In Steps 2 and 4, the interlinks among plan nodes, join schemas and indexes are properly represented, thereby addressing Challenge 2. Challenge 3 will be handled by designing the loss function of ChangeFormer formulated in subsection 5.6.

5.3 Plan Node Encoding

Features. The plan node encoder encodes each node in P_q as a fixed-length vector. A variety of schemes [1, 27, 30, 38] have been designed to encode plan nodes. Similar to these encoding schemes, we encode a plan node N based on the following features to satisfy the requirements of index tuning.

(1) The type of the operator N, which is represented as a number in $\{1, 2, \ldots, n_o\}$, where n_o is the number of distinct operators that constitute query plans.

(2) If N is a scan operator, e.g. SeqScan, the table scanned by N is an essential feature of N, which is represented as a number in $\{1, 2, \ldots, n_t\}$, where n_t is the number of tables in the database. Other types of plan nodes that do not directly process tables, but intermediate results, do not have this feature. Their table-related information is not explicitly encoded but is implicitly captured by the tree-structured transformer that understands the relationships between the non-scan nodes and the scan nodes in P_q. In particular, for join nodes, their table-related information has a more significant impact on P_q, so we encode such information separately as will be described in subsection 5.4.

(3) The statistics of the input and the output of N. The statistics commonly used by the existing plan node encoding schemes include the estimated cost of N, the estimated number of tuples returned by N, and histograms and samples of the input of N, where histograms and samples are used to make more accurate estimation on the cost and the cardinality of the output.

Notably, index tuning is different from the problems that are faced by the existing plan node encoding methods such as cardinality estimation [13, 41]. During index tuning, the queries in the input workload are really executed under the current index configuration $\mathcal{I}_0$, so the actual cost of N and the number of tuples returned by N can be known. Therefore, histograms and samples of N are useless to our plan node encoding, and we only require the actual cost and the actual output cardinality of N which are represented as two real numbers in $[0, 1]$ normalized within the ranges of costs and cardinalities of all plan nodes in P_q.

(4) If N has a predicate such as the condition of a selection operator, the predicate is also an essential feature of N. In the existing plan node encoding methods, a simple predicate "attr op val" is usually represented as a triple (attr, op, val), where attr is an attribute, op is an operation in $\{<, <=, =, >=, >, <>\}$, and val is a constant. Normally, attr and op are represented as categorical

values, and `val` is range-normalized to a real number in $[0, 1]$. For the existing methods, `val` is used to estimate the statistics of N. However, as the actual statistics of N are known in our work, it is unnecessary to keep `val`, that is, our method just represents this simple predicate as (`attr`, `op`). Obviously, our method has another advantage. The existing plan node encoding methods cannot handle string predicates like "`city LIKE 'San %'`" because the pattern `'San %'` is not an exact value and cannot be mapped to a real number. Our method just represents it as (`city`, `LIKE`) and avoids encoding the pattern `'San %'`.

A compound predicate is composed of several simple predicates connected by logical operators `AND`, `OR` or `NOT`. Encoding compound predicates is a complicated issue which is often over-simplified by the existing methods. For example, in the state-of-the-art query plan encoding method QueryFormer [38] that inspires our work, only the simple predicates in a compound predicate are encoded, whereas the logical operations are totally ignored. However, the logical operations are very important for the semantics of the compound predicate because different logical expressions of the simple predicates may result in totally different query plans. To address this issue, we represent a compound predicate as an expression tree, where leaf nodes are simple predicates, and non-leaf nodes are logical operators. Then, the expression tree is traversed in pre-order to form a sequence of simple predicates and logical operators. Here the logical operators are encoded as categorical variables.

Embedding. Finally, the atomic features of a plan node N (the operator type of N, the table scanned by N, the actual cost of N, the actual output cardinality of N, the attribute in a simple predicate, the operation in a simple predicate, and logical operations) are embedded as fixed-size vectors in the embedding space. Any feature that is not applicable to N is encoded as a vector of zeros. These embedding vectors are then concatenated to form the encoding vector of N. To cope with variable-length compound predicates, we require a compound predicate to be encoded as a sequence of L fixed-size embedding vectors (unused embedding vectors are set to zero vectors).

5.4 Join Schema Encoding

The join schema of the query q represents how the tables involved in q can be joined, which is represented as a graph. The type of an edge indicates the type of the join such as inner join or (left, right or full) outer join. A left (right) outer join is represented as a directed edge. Figure 5 depicts a join schema. All possible join orders can be derived from the join schema. Particularly, any spanning tree of the join schema contains an unordered set of joins that must be done in a query plan of q. After specifying an execution order on these joins, we obtain a specific join order.

Join schemas are essential for increasing prediction accuracy. Although our join schema encoding method is inspired by Query-Former, they are quite different in their basic thought. To represent the join-related information of a query plan, QueryFormer only encodes the join order specified in this plan. Notably, this join order corresponds to a spanning tree of the join schema. The process of encoding the join order by QueryFormer is as follows: First, the joins (edges) in the join order (spanning tree) are mapped to categorical values independently. Then, the embedding vectors of these
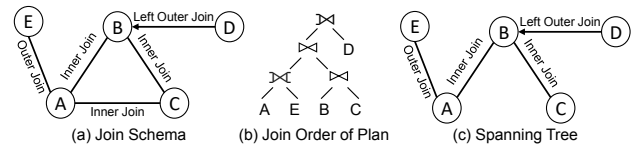
Figure 5: An example of join schema and join order. (a) The join schema of a query. (b) The plan of the query. (c) The spanning tree of the join schema corresponding to the plan.

values are concatenated with the encoding of the plan nodes. As QueryFormer focuses on encoding joins (edges), we call it an edge-oriented encoding method. However, it has several disadvantages:

(1) According to join conditions, there are two kinds of joins. One kind of joins is based on the relationships between foreign keys and primary keys, which can be figured out from the database schema. Of course, these joins can be mapped to categorical values in advance. The other kind of joins are specified by users according to query semantics and cannot be known in advance if these joins do not appear in the workload. For unseen user-specified joins, there are no corresponding categorical values, making them unable to be properly understood by the model trained on the workload that does not contain these unseen joins.

(2) For a query plan, QueryFormer only encodes the join order in this plan. The join orders that can be derived from other spanning trees of the join schema are all excluded, so they are missed by the model. Although encoding only one join order is sufficient for some tasks such as cardinality estimation, it is not enough for detecting changes in query plans because the model is unaware of all possible join orders except the encoded one.

To address the above issues, we propose a vertex-oriented encoding method to encode the whole join schema instead of one of its spanning trees. In this method, the information of the join schema is encoded into the embeddings of vertices instead of edges. Our method is designed based on the attention mechanism [32] with information flow control. It works as follows:

First, each vertex (table) T_i in the join schema is mapped to a categorical value and is then encoded as a fixed-length vector $E(T_i)$. Then, the encoding of the tables $E(T_i)$ are input into the attention module, and the attention module fuse $E(T_i)$ with $E(T_j)$ of all tables T_j that are joinable with T_i (T_j is a neighbor of T_i in the join schema). We refer to the output corresponding to table T_i as the table fusion embedding of T_i, denoted by E_i^J. In this way, the information of the join schema is encoded into the table fusion embedding of all its vertices (tables).

As depicted in Figure 4, our attention module adopts a new structure. An introduction to the attention mechanism can be found in [32]. As with the standard attention module, the matrices Q and K are used to compute the attention values A_{ij} between any pairs of tables T_i and T_j in the join schema, which represents the influence of T_j on T_i. However, the influence of a table on another table is not arbitrary but is restricted by the edges (the join relationships between the tables) in the join schema. To restrict such influence, we create the join-bias matrix J, where each element J_{ij} represents whether there is an edge between two tables T_i and T_j in the join schema, that is, whether T_i and T_j are joinable. We have $J_{ij} = 0$

if T_i and T_j are not joinable, so the influence between T_i and T_j is masked. We have $T_{ij} = j_1, j_2, j_3, j_4$ if T_i and T_j are joined by an inner join, a left outer join, a right outer join, or a full outer join, respectively, where j_1, j_2, j_3 and j_4 are learnable scalars. Finally, we add the join-bias matrix J to the attention score matrix A.

Because our join schema encoding method is vertex-oriented, we associate the query plan P_q with the join schema by associating the encoding of the vertices (tables) in the join schema with the encoding of the related plan nodes in P_q. In particular, if a plan node N is a join on two input tables or intermediate results L and R, the table fusion embedding of L and R are concatenated with the encoding of N. If L (or R) is a table T_i, the embedding of L (or R) is certainly the table fusion embedding E_i^J of T_i. If L (or R) is an intermediate relation, the embedding of L (or R) is $\mathbf{0}$.

5.5 Index Encoding

The index encoder encodes each index as a fixed-length vector. Let $E(a)$ denote the embedding vector of an attribute a obtained as described in subsection 5.3 and let $k_{\max}$ be the maximum number of attributes composing an index key. An index E_j^I with index key $(a_1, a_2, \ldots, a_k)$ is encoded by

$$E_j^I = \left(\bigoplus_{i=1}^{k} E(a_i) \right) \oplus \left(\bigoplus_{i=k+1}^{k_{\max}} \mathbf{0} \right), \qquad (10)$$

where $\mathbf{0}$ is a vector of zeros having the same length as $E(a_i)$.

There are several methods to integrate the information of the index configuration $\mathcal{I}$ into the model, e.g., encoding it together with each node of the query plan P_q similar to join schema encoding. However, through experimentation, we chose the simplest but the most effective approach: encoding $\mathcal{I}$ independent of P_q.

Let $l_{\max}$ be the maximum number of indexes that the model can handle and suppose the index configuration $\mathcal{I}$ contains $|\mathcal{I}| \leq l_{\max}$ indexes encoded as $E_1^I, E_2^I, \ldots, E_{|\mathcal{I}|}^I$, respectively. We concatenate the encoding vectors of all these indexes and input them into an MLP to obtain the representation vector R_I of $\mathcal{I}$, that is,

$$R_I = \mathrm{MLP}\left(\left(\bigoplus_{i=1}^{|\mathcal{I}|} E_i^I \right) \oplus \left(\bigoplus_{i=|\mathcal{I}|+1}^{l_{\max}} \mathbf{0} \right) \right). \qquad (11)$$

Recall that $\mathcal{I}$ only contain the enumerated indexes that are relevant to q. Thus, it is not necessary to set $l_{\max}$ to a very large number.

5.6 Model Training

Training Dataset. To train ChangeFormer, we first prepare a dataset $D = \{(P_{q_i}, \mathcal{I}_i, P'_{q_i}) | i = 1, 2, \ldots, n\}$, where P_{q_i} is the execution plan of a query q_i under the current index configuration $\mathcal{I}_0$, $\mathcal{I}_i$ is an index configuration, and P'_{q_i} is the execution plan of q_i after replacing $\mathcal{I}_0$ with $\mathcal{I}_i$. To get D, we acquire a set Q of queries and a collection $\mathbb{I}$ of index configurations during the latest index tuning process, where Q consists of all or a fraction of historical queries, and $\mathbb{I}$ consists of all or a fraction of enumerated index configurations. For each $q \in Q$ and each $\mathcal{I} \in \mathbb{I}$, we obtain q's execution plans P_q and P'_q under $\mathcal{I}_0$ and $\mathcal{I}$, respectively, and compose a triple $(P_q, \mathcal{I}, P'_q)$ in D.

The training set of ChangeFormer can be easily derived from on D. For each triple $(P_{q_i}, \mathcal{I}_i, P'_{q_i}) \in D$, we create a training record

$(P_{q_i}, \mathcal{I}_i, y_i)$, where $y_i \in \{0, 1\}$ is the class label. Particularly, $y_i = 1$ if P_{q_i} and P'_{q_i} have different tree structures, and $y_i = 0$ otherwise.

In addition, D can be used to train the linear model formulated by Eq. (5), particularly, the coefficients r_N and b_N. For a specific operator type t such as SeqScan, we observe that the hardware (or the DBMS configuration) affects various plan nodes of type t in various queries almost to the same extent. It implies that all plan nodes N of type t can have the same coefficients r_N and b_N. Hence, we denote them by r_t and b_t, respectively. To train r_t and b_t, we prepare a training set as follows: For each triple $(P_{q_i}, \mathcal{I}_i, P'_{q_i}) \in D$, we find a node N in P_{q_i} with $t_N = t$ and a node N' in P'_{q_i} with $I_{N'} = I_N$, $O_{N'} = O_N$ and $t_{N'} \neq t$. In other words, N is speed up by some indexes in $\mathcal{I}$. Then, we compute the selectivity $\theta(N)$ of N and the multiplicative factor $\alpha = (c(t_N, I_N) - c(t_{N'}, I_{N'})) / c(t_N, I_N)$ and create a training record $(\theta(N), \alpha)$. Finally, based on the training set, we use the least squares method to find the best coefficients r_t and b_t for operator type t.

Training Loss. To train a binary classification model, the *binary cross-entropy loss* is often adopted. In our problem, false positives and false negatives of ChangeFormer affect the time and the quality of index tuning, respectively. In general, we prefer to the quality. Therefore, we adopt the *weighted binary cross-entropy loss*:

$$L = -\frac{1}{N} \sum_{i=1}^{N} (\alpha \cdot y_i \log p_i + \beta \cdot (1 - y_i) \log(1 - p_i)), \qquad (12)$$

where α and β are the scaling factors of the false negative cost and the false positive cost, respectively. To reduce false negatives, we can set $\alpha = 2$ and $\beta = 1$, which work well in many situations.

6 EVALUATION

6.1 Experiment Setup

Databases & Workloads. The evaluation was conducted on three famous database benchmarks: TPC-H, TPC-DS and JOB [17]. The databases for TPC-H and TPC-DS were generated with the scale factor (SF) of 10. The workloads were generated as follows.

TPC-H: In this workload, we generated 30 queries at random based on each template in TPC-H excluding templates 2, 17 and 20. As with Leis et al. [15], templates 2, 17 and 20 were excluded because queries in these templates attain several orders of magnitude longer execution time than queries in other templates. If not excluded, the queries in these templates will dominate the cost of the entire workload, thereby creating an index to decrease the costs of the queries in one of these templates would always be better than creating indexes for other queries.

TPC-DS: In this workload, we randomly generated 30 queries based on each template in TPC-DS excluding templates 4, 6, 9, 10, 11, 32, 35, 41, and 95 due to the same reason given above.

JOB: This workload contains 30 queries randomly generated based on each template in JOB.

To increase the complexity of index tuning, we added some synthetic queries to these workloads as the complexity of index tuning depends on the number of possible index configurations, the number of queries in the workload and the correlations between index configurations. Synthetic queries may not comply with any known templates, so they can increase the number of possible index configurations. Synthetic queries were generated according

Table 1: Summary of databases and workloads.

Benchmark	DB SF/Size	# Queries	# Templates	# Synthetic Queries	Avg. # Plan Nodes
TPC-H	SF = 10	570	19	0	14
TPC-DS	SF = 1	2700	90	0	29
JOB	13GB	3390	113	0	24
TPC-H+	SF = 10	950	19	380	10
TPC-DS+	SF = 1	4500	90	1800	19
JOB+	13GB	5650	113	2260	16

to the context-free grammar of SQL that supports most advanced SQL syntax including joins and nested queries. The constants in a synthetic query were randomly chosen from the values of their corresponding attributes to ensure nonempty query results. The obtained workloads are denoted as **TPC-H+**, **TPC-DS+** and **JOB+**, respectively. Table 1 summarizes the databases and the workloads.

Hardware & Software. The experiments were carried out on a Ubuntu server with two Intel Xeon 4210R CPUs (10 cores, 2.40GHz), 256GB of main memory, and one NVIDIA GeForce RTX 3060 GPU. The DBMS is PostgreSQL 12.13.

Implementation. Leis et al. [15] shows that no index configuration enumeration algorithm can achieve the best performance in all situations. In the experiments, we used AutoAdmin [6] as it is usually effective for larger index configuration search spaces. RIBE also works well with other enumeration algorithms, e.g. DTA [5].

6.2 End-to-End Evaluation of RIBE

6.2.1 Baselines. RIBE was compared with five baseline algorithms that follow the traditional index tuning framework described in section 2. For the sake of fairness, these baselines also use AutoAdmin [6] to enumerate candidate index configurations. These baselines are different in how they compress the input workload.

Full: Workload compression is disabled.

GSUM: It uses GSUM [10] to compress the workload.

ISUM: It uses ISUM [28] to compress the workload.

ISUM-S: A variant of ISUM is used to compress the workload, which avoids pairwise comparisons between queries. It often compresses a workload faster with a minor decline in performance.

Sample: The workload is compressed by randomly sampling a specified number of queries from the workload.

Let n be the number of queries in the workload. Similar to the evaluation in [28], the compressed workload contains $\sqrt{n}/2$ queries.

DISTILL [29] is a new index tuning method. However, it is not open-sourced. Due to various possible ways to implement DISTILL and various heuristic rules adopted by DISTILL, it is not easy to reproduce the implementation in the original paper. Therefore, we did not compare RIBE with DISTILL in the experiments.

6.2.2 Evaluation Metrics. We use the following metrics to evaluate the performance of index tuning.

Cost Improvement. Given a workload W, let $\mathcal{I}_0$ be the initial set of indexes and $\mathcal{I}$ be the set of selected indexes. The quality of replacing $\mathcal{I}_0$ with $\mathcal{I}$ is measured by the relative improvement in the cost of the workload W which is defined as

$$\max\left(0, \frac{B(W, \mathcal{I}_0, \mathcal{I})}{c(W, \mathcal{I}_0)}\right) = \max\left(0, 1 - \frac{c(W, \mathcal{I})}{c(W, \mathcal{I}_0)}\right). \quad (13)$$

In some occasions, the selected indexes in $\mathcal{I}$ may inversely increase the cost and must be discarded, so the cost improvement is 0.
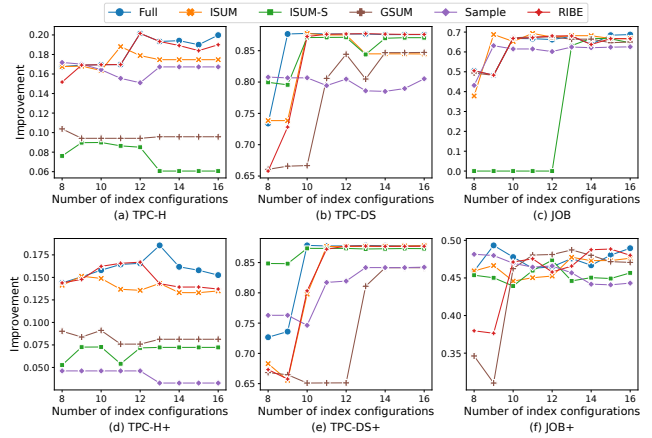
Figure 6: Cost improvements achieved by different index tuning methods.

Table 2: Rankings of cost improvements achieved by different index tuning methods.

Method	# 1st Place	# 2nd Place	# 3rd Place	# Worst
RIBE	16	17	10	1
Full	27	14	6	0
ISUM	6	11	23	3
ISUM-S	2	4	5	16
GSUM	4	3	2	12
Sample	4	5	4	22

Index Tuning Time. The efficiency of index tuning is measured by the wall clock time from the acceptance of the workload to the time when the "optimal" index configurations are returned. The index tuning time mainly includes the time for workload compression, model loading and index configuration enumeration.

6.2.3 Experimental Results. In this experiment, we vary the budget on the number of selected indexes from 8 to 16 because the number of selected indexes affects the quality and the efficiency of index tuning. We obtain the following experimental results.

Quality of Index Tuning. The cost improvements achieved by the tested index tuning methods are depicted in Figure 6, and the detailed statistics of cost improvements are shown in Table 2. We have the following observations:

(1) The index tuning quality of these methods follows the order: Full > RIBE > ISUM > GSUM ≥ ISUM-S > Sample. However, no single method outperforms the others on all workloads.

(2) The indexes selected by RIBE often lead to higher cost improvements on various database benchmarks especially when more indexes are selected. As shown in Table 2, out of 54 tests across 6 benchmarks and 9 budgets (8–16) on the number of selected indexes, RIBE gets the first place in cost improvement in 16 tests (29.6%), the second place in 17 tests (31.5%) and the third place in 10 tests (18.5%). It performs the worst in only one test. RIBE achieves index tuning quality closest to Full.

This observation can be explained as follows: GSUM, ISUM, ISUM-S, and Sample retain a subset of queries in the workload with high costs after workload compression, which allows us to
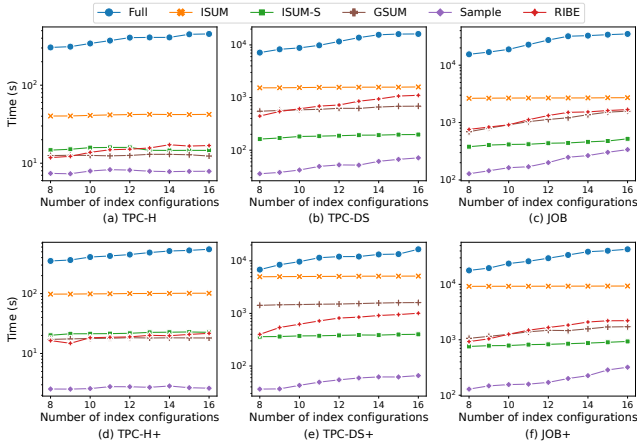
Figure 7: Running time of different index tuning methods.

select the most beneficial indexes when the desired number of indexes is small. However, as the number of indexes increases, the discarded queries affect index tuning more significantly, leading to a decrease in cost improvement after building the selected indexes. Unlike the baselines, RIBE only removes redundant queries from the workload, which enables us to find more appropriate indexes.

(3) The cost improvement led to by every index tuning method is not monotonically increasing with the number of selected indexes. Similar to the observations in [28], creating more indexes may even slow down the execution of some queries. This is mainly caused by the intrinsic errors in the query optimizer's cost estimator and the fact that the index configuration enumeration algorithms are often greedy and cannot find the optimal solution.

Efficiency of Index Tuning. The index tuning time of the tested methods is shown in Figure 7. Their efficiency follows the order: Sample > ISUM-S ≥ RIBE ≈ GSUM > ISUM ≫ Full.

Although Full has the best index tuning quality, it is the slowest because it handles the full workload. All the other methods are faster than Full because their index tuning is performed on compressed workloads. RIBE achieves index tuning quality very close to Full and is 1–2 orders of magnitude faster than Full.

Among the methods based on workload compression, ISUM is the slowest because it requires comparing all pairs of queries, resulting in a much higher time complexity of $O(kn^2)$ than the other compression methods. Sample is the fastest as it randomly selects a subset of queries from the workload without complex processing. However, its tuning quality is the worst. RIBE is superior to ISUM in both tuning quality and efficiency. The tuning efficiency of RIBE is close to GSUM but sometimes lower than ISUM-S.

6.3 Evaluation of ChangeFormer

Here, we evaluate the accuracy and efficiency of ChangeFormer.

6.3.1 Baselines. ChangeFormer represents a query plan as a vector which is essential for predicting if the plan will be changed after building an index. In this evaluation, we compare ChangeFormer with three models adapted from ChangeFormer by substituting its query representation component with the following ones.

QueryFormer: QueryFormer is a transformer-based query representation model [38]. We adopt the processing methods described in the original paper, that is, using sampling and histograms to encode the statistical information in a query plan and employing an edge-oriented scheme to encode join schemas.

Tree-CNN: This model is used in the learning-based query optimizers BAO [18] and NEO [19]. It is designed based on convolution neural networks (CNN) and utilizes triangular-shaped filters to handle the tree structures of query plans.

GCN: A query is represented using a graph neural network [12, 40]. Based on the query's tree representation obtained in subsection 5.3 (viewed as an undirected graph here), 4–16 layers of graph convolutional networks (GCNs) [14] are applied, and a mean pooling is finally used to obtain the representation of the plan.

6.3.2 Experimental Results. The evaluation on the classification performance and the efficiency of ChangeFormer is as follows.
Classification Performance. The classification performance of ChangeFormer is evaluated by its accuracy and the F2-score. We use F2-score because of the higher misclassification costs caused by false negatives in query plan change prediction. We used 10-fold cross-validation to evaluate the accuracy and the F2-score.

As shown in Table 3, ChangeFormer attains higher classification performance than all the other models in most cases. The design of ChangeFormer is inspired by QueryFormer, so they have many common points in design. However, ChangeFormer always outperforms QueryFormer due to two distinct design decisions. First, ChangeFormer adopts the vertex-oriented encoding of join schemas which results in a better generalization ability. Second, using actual statistics in ChangeFormer leads to better classification accuracy than using sampling and histograms in QueryFormer.

GCN generally achieves better performance than the other models except ChangeFormer. It is attributed to the stacking of multiple GCN layers that can learn high-order information of a graph.

QueryFormer generally outperforms Tree-CNN, especially in more complex workloads that include synthetic queries. Its superior performance is due to the self-attention mechanism which captures more useful information and effectively tackles the issue of information loss resulting from long paths in query plans.
Time Efficiency. We evaluated the efficiency of the models by the training time per epoch and the inference time per batch (consisting of 1024 queries). Since the hyperparameters can significantly affect the efficiency of the models, we tested the per-epoch training time and the per-batch inference time averaged over 20 diverse sets of hyperparameters. As shown in Table 4, the efficiency of ChangeFormer is comparable to that of QueryFormer. These two models are somewhat less efficient than Tree-CNN because they encode more query-related information than Tree-CNN.

GCN is 2–7× slower than the other models in all circumstances. This is because the eigendecomposition of an $N \times N$ matrix within GCN entails a time complexity of $O(N^3)$. Moreover, to accomplish superior accuracy, GCN must be composed of more neurons than the other models, thereby further degrading to the efficiency.

6.4 Accuracy of Cost Estimation

In this subsection, we compare the accuracy of our proposed cost estimation method with the query optimizer's cost estimator.

Table 3: Classification performance of different query plan change prediction models. Green color highlights the highest evaluation metric values, and red color highlights the second highest evaluation metric values.

Model	TPC-H		TPC-DS		JOB		TPC-H+		TPC-DS+		JOB+	
	Accuracy	F2-Score	Accuracy	F2-Score	Accuracy	F2-Score	Accuracy	F2-Score	Accuracy	F2-Score	Accuracy	F2-Score
ChangeFormer	0.963	0.954	0.950	0.972	0.963	0.984	0.981	0.972	0.941	0.951	0.972	0.987
QueryFormer	0.941	0.945	0.932	0.952	0.929	0.968	0.949	0.929	0.883	0.951	0.937	0.960
Tree-CNN	0.950	0.909	0.960	0.972	0.952	0.975	0.837	0.766	0.873	0.878	0.877	0.909
GCN	0.955	0.936	0.955	0.972	0.956	0.967	0.957	0.951	0.957	0.962	0.952	0.964

Table 4: Training time and inference time of different query plan change prediction models.

Model	TPC-H		TPC-DS		JOB		TPC-H+		TPC-DS+		JOB+	
	Training Time (s)	Inference Time (ms)	Training Time (s)	Inference Time (ms)	Training Time (s)	Inference Time (ms)	Training Time (s)	Inference Time (ms)	Training Time (s)	Inference Time (ms)	Training Time (s)	Inference Time (ms)
ChangeFormer	76.7	6.92	872.1	7.43	219.5	6.15	153.8	6.28	1012.6	6.08	472.9	5.17
QueryFormer	69.2	6.71	716.3	8.45	321.5	6.47	182.7	4.03	749.9	6.82	767.15	4.11
Tree-CNN	21.2	4.56	291.6	3.15	219.3	3.88	82.5	2.89	746.6	2.82	757.3	2.39
GCN	146.1	13.79	1520.3	15.20	862.1	17.85	309.2	15.20	2117.2	12.57	1046.8	13.78

Table 5: Cost estimation errors of different cost estimation methods.

Estimator	TPC-H			TPC-DS			JOB			TPC-H+			TPC-DS+			JOB+		
	50th	95th	99th	50th	95th	99th	50th	95th	99th	50th	95th	99th	50th	95th	99th	50th	95th	99th
Optimizer	0.077	0.287	0.959	0.018	0.529	3.087	0.109	0.543	0.836	0.070	0.324	0.959	0.019	0.567	1.538	0.076	0.533	0.998
Matrix-D	0.076	0.288	0.959	0.020	0.292	3.121	0.107	0.539	0.783	0.071	0.379	0.959	0.021	0.500	1.615	0.077	0.534	0.998
Matrix-R	0.077	0.308	0.557	0.021	0.292	3.015	0.155	0.438	0.649	0.088	0.323	0.608	0.022	0.500	1.515	0.089	0.441	0.673

6.4.1 Experiment Design. The experiment is designed as follows: First, we collect the queries in the workload whose plan tree structures are not changed by the enumerated index configurations. Then, we compute the estimated costs of these queries after building the selected indexes using three methods.

Optimizer: We create hypothetical indexes and use PostgreSQL's optimizer to compute the estimated costs of the query plans.

Matrix-D: The method proposed in subsection 4.2 is used to compute the estimated costs of the query plans. For all plan nodes N, the parameters r_N and b_N are set to 1 and 0, respectively.

Matrix-R: The method is the same as Matrix-D except that r_N and b_N are adjusted with regard to the hardware and the software.

For each collected query q, we evaluate the accuracy of the estimated cost of q after building a set of indexes $\mathcal{I}$ by the absolute error between the actual benefit and the estimated benefit of building $\mathcal{I}$ with respect to q, that is,

$$\left| \frac{c(q, \mathcal{I}) - c(q, \mathcal{I}_0)}{c(q, \mathcal{I}_0)} - \frac{\hat{c}(q, \mathcal{I}) - c(q, \mathcal{I}_0)}{c(q, \mathcal{I}_0)} \right| = \left| \frac{c(q, \mathcal{I}) - \hat{c}(q, \mathcal{I})}{c(q, \mathcal{I}_0)} \right|,$$

where $\mathcal{I}_0$ is the initial set of indexes created on the database.

6.4.2 Experimental Results. Table 5 shows the 50th, 95th and 99th percentiles of the errors in the estimated costs of the collected queries. The minimum error in each column is highlighted in green. We have the following observations:

(1) On all benchmarks, Matrix-D achieves cost estimation accuracy comparable to that of Optimizer. This is because the cost estimation formula used by Matrix-D effectively simulates the cost calculation process of the query optimizer's cost estimator. Additionally, in some cases, Matrix-D achieves significant improvements in accuracy, such as the 95th percentile of errors on TPC-DS. These improvements are because the estimated statistics used by

Optimizer significantly deviate from the actual statistics in some situations, while Matrix-D can always make accurate estimation based on the actual statistics stored in the workload matrices.

(2) Matrix-R adjusts the parameters r_N and b_N based on the historical data. It leads to a substantial reduction in the tail distribution of errors on all benchmarks, up to 40% compared with both Matrix-D and Optimizer.

(3) In some occasions, cost estimation errors can exceed 1. This is because index creation increases query execution time. Both the query optimizer and our method are unable to handle such extreme situations. We will address this issue in our future work.

7 CONCLUSION

RIBE tackles both workload redundancy and frequent costly what-if calls that make the traditional index tuning framework sub-optimal in terms of both tuning efficiency and quality. For the former issue, the queries in the workload are clustered according to their operator-level actual statistical features stored in the workload matrices. This clustering-based workload compression reduces workload redundancy more effectively than the existing methods. For the latter issue, the estimated benefit of creating indexes with respect to a query is computed based on the actual statistics stored in the workload matrices if the query's plan structure will not be changed by the indexes to be created. It is faster and more accurate than using the what-if tool. With the encoding schemes for query plans, indexes and join schemas, ChangeFormer can accurately predict if the structure of a plan will be changed by the enumerated indexes.

REFERENCES

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

[2] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 227–238. https://doi.org/10.1145/1066157.1066184

[3] S. Chaudhuri, M. Datar, and V. Narasayya. 2004. Index selection for databases: a hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering* 16, 11 (2004), 1313–1323. https://doi.org/10.1109/TKDE.2004.75

[4] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "What-If" Index Analysis Utility. *SIGMOD Rec.* 27, 2 (jun 1998), 367–378. https://doi.org/10.1145/276305.276337

[5] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server. (June 2020). https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/

[6] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (Eds.). Morgan Kaufmann, 146–155.

[7] Sunil Choenni, Henk M. Blanken, and Thiel Chang. 1993. Index Selection in Relational Databases. In *Computing and Information - ICCI'93, Fifth International Conference on Computing and Information, Sudbury, Ontario, Canada, May 27-29, 1993, Proceedings*, Osman Abou-Rabia, Carl K. Chang, and Waldemar W. Koczkodaj (Eds.). IEEE Computer Society, 491–496.

[8] Douglas Comer. 1978. The Difficulty of Optimum Index Selection. *ACM Trans. Database Syst.* 3, 4 (1978), 440–445. https://doi.org/10.1145/320289.320296

[9] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *Proc. VLDB Endow.* 4, 6 (2011), 362–372. https://doi.org/10.14778/1978665.1978668

[10] Shaleen Deep, Anja Grueneheid, Paraschos Koutris, Jeffrey Naughton, and Stratis Viglas. 2020. Comprehensive and Efficient Workload Compression. *Proc. VLDB Endow.* 14, 3 (2020), 418–430. https://doi.org/10.14778/3430915.3430931

[11] 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* (New York, NY, USA, 2019-06-25) (SIGMOD '19). Association for Computing Machinery, 1241–1258. https://doi.org/10.1145/3299869.3324957

[12] Jianling Gao, Nan Zhao, Ning Wang, Shuang Hao, and Haoyan Wu. 2022. Automatic index selection with learned cost estimator. *Information Sciences* 612 (2022), 706–723. https://doi.org/10.1016/j.ins.2022.08.051

[13] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, Not from Queries! *Proc. VLDB Endow.* 13, 7 (mar 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[14] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 10. https://openreview.net/forum?id=SJU4ayYgl

[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. *Proc. VLDB Endow.* 13, 12 (2020), 2382–2395. https://doi.org/10.14778/3407790.3407832

[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* (New York, NY, USA, 2020-10-19) (CIKM '20). Association for Computing Machinery, 2105–2108. https://doi.org/10.1145/3340531.3412106

[17] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[18] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event China, 2021-06-09). ACM, 1275–1288. https://doi.org/10.1145/3448016.3452838

[19] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[20] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746. https://doi.org/10.14778/3342263.3342646 arXiv:1902.00132 [cs]

[21] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[22] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2021. DBA Bandits: Self-Driving Index Tuning under Ad-Hoc, Analytical Workloads with Safety Guarantees. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (Chania, Greece, 2021-04). IEEE, 600–611. https://doi.org/10.1109/ICDE51399.2021.00058

[23] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2022. HMAB: Self-Driving Hierarchy of Bandits for Integrated Physical Database Design Tuning. *Proc. VLDB Endow.* 16, 2 (2022), 216–229. https://www.vldb.org/pvldb/vol16/p216-perera.pdf

[24] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-Attribute Index Selection Using Recursive Strategies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (Macao, Macao, 2019-04). IEEE, 1238–1249. https://doi.org/10.1109/ICDE.2019.00113

[25] Vishal Sharma and Curtis Dyreson. 2022. Indexer++: Workload-Aware Online Index Tuning with Transformers and Reinforcement Learning. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (New York, NY, USA, 2022-04-25) (SAC '22). Association for Computing Machinery, 372–380. https://doi.org/10.1145/3477314.3507691

[26] Vishal Sharma, Curtis Dyreson, and Nicholas Flann. 2021. MANTIS: Multiple Type and Attribute Index Selection Using Deep Reinforcement Learning. In *Proceedings of the 25th International Database Engineering & Applications Symposium* (New York, NY, USA, 2021-09-07) (IDEAS '21). Association for Computing Machinery, 56–64. https://doi.org/10.1145/3472163.3472176

[27] Jiachen Shi, Gao Cong, and Xiaoli Li. 2022. Learned Index Benefits: Machine Learning Based Index Performance Estimation. *Proc. VLDB Endow.* 15, 13 (2022), 3950–3962. https://www.vldb.org/pvldb/vol15/p3950-shi.pdf

[28] 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* (New York, NY, USA, 2022-06-10) (SIGMOD '22). Association for Computing Machinery, 660–673. https://doi.org/10.1145/3514221.3526152

[29] 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. *Proc. VLDB Endow.* 15, 10 (2022), 2019–2031. https://doi.org/10.14778/3547305.3547309

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

[31] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 101–110. https://doi.org/10.1109/ICDE.2000.839397

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2017-12-04) (NIPS'17). Curran Associates Inc., 6000–6010.

[33] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (2021), 1640–1654. https://doi.org/10.14778/3461535.3461552 arXiv:2012.06743

[34] Zilong Wang, Qixiong Zeng, Ning Wang, Haowen Lu, and Yue Zhang. 2023. CEDA: Learned Cardinality Estimation with Domain Adaptation. *Proc. VLDB Endow.* 16, 12 (2023), 3934–3937. https://doi.org/10.14778/3611540.3611589

[35] Sai Wu, Ying Li, Haoqi Zhu, Junbo Zhao, and Gang Chen. 2022. Dynamic Index Construction with Deep Reinforcement Learning. *Data Sci. Eng.* 7, 2 (2022), 87–101. https://doi.org/10.1007/S41019-022-00186-4

[36] 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* (New York, NY, USA, 2022-06-10) (SIGMOD '22). Association for Computing Machinery, 1528–1541. https://doi.org/10.1145/3514221.3526128

[37] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73. https://doi.org/10.14778/3421424.3421432

[38] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670. https://doi.org/10.14778/3529337.3529349

[39] 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) (Kuala Lumpur, Malaysia, 2022-05). IEEE, 2196–2208. https://doi.org/10.1109/ICDE53745.2022.00210

[40] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries Using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (may 2020), 1416–1428. https://doi.org/10.14778/3397230.3397238

[41] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.* 14, 9 (2021), 1489–1502.