A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies

Yue Zhao
Nanyang Technological University
zhao0342@e.ntu.edu.sg

Zhaodonghui Li
Nanyang Technological University
g220002@e.ntu.edu.sg

Gao Cong
Nanyang Technological University
gaocong@ntu.edu.sg

ABSTRACT

Query plan is widely used as input in machine learning for databases (ML4DB) research, with query plan representation as a critical step. However, existing studies typically focus on one task, and propose a novel design to represent query plans along with a ML4DB framework, without comparing with other representation methods designed for a different task. This raises a critical question: How do we select a query plan representation method in a ML4DB system?

To address this question, we perform a comparative study on ten representation methods on three distinct ML4DB tasks: cost estimation, index selection and query optimization. Our extensive experiments not only verify the interchangeability of representation methods across different tasks, but also identify consistently high-performing models. Further, we dissect the query plan representation into two core components: feature encoding and tree model, and evaluate the impact of design choices for each in different scenarios. Our results show that the findings for tasks optimizing absolute errors are different from findings for tasks optimizing relative errors. Some findings challenge widely-held assumptions, i.e., one finding shows that tree models do not significantly impact cost estimation results, but only play a significant role to optimize relative performance. Practical guidelines and future directions are provided based on the findings of the study.

PVLDB Reference Format:
Yue Zhao, Zhaodonghui Li, and Gao Cong. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. PVLDB, 17(4): 823 - 835, 2023.
doi:10.14778/3636218.3636235

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

1 INTRODUCTION

In machine learning for databases (ML4DB) research, query plans are widely utilized as inputs for various tasks such as cost estimation [17, 28, 36], index selection [11], join order selection [27, 44], query optimization [25, 26], view selection [45] and more [33, 46]. A fundamental aspect of these endeavors involves representing query plans in a format suitable for machine learning models [46].

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636235

A physical query plan is a detailed description of the sequence of steps to process a query and retrieve the results. It is represented in the form of a directed tree, in which each node describes a unit operation, and each edge describes the dependencies between two nodes. Specifically, each node is responsible for processing a part of the query passing the results to its parent node.

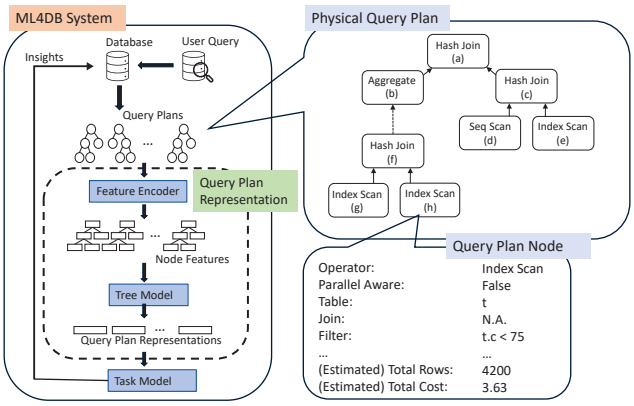


Figure 1: Left: An example workflow of a query-plan based ML4DB system. Top Right: An example query plan. Bottom Right: An example query plan node.

Figure 1 illustrates an example workflow of a typical ML4DB system which takes query plans as input. First, a *feature encoder* module takes the query plans generated from a database system as input, extracts the important information from each node in a query plan, and featurizes the information into a unified format, such as fixed-sized vectors. Next, the trees of vectors will be passed to a *tree model*, i.e., long short-term memory (LSTM) [45], tree convolutional neural networks (TreeCNN) [25], tree-structured LSTM (TreeLSTM) [44], or Transformer [46], to aggregate the node features. The tree model will output vector representations for the input query plans, and finally, they will be passed to a *task-specific machine learning model*, such as a regressor or classifier, which provides insights to the database system and user. Despite targeting on different tasks, these ML4DB systems taking query plans as input rely on the representations of query plans to learn the correlations between query plan properties and the targeted outputs [46]. They may employ different task models, but they all need the feature encoder component and tree model component to learn the representation of physical query plans.

Most of the existing studies on ML4DB using query plans focus on one task, such as cost estimation, and typically propose a new approach to represent query plans along with task models to serve

its task. However, they do not empirically compare with the query plan representation methods which are designed for a different task. One natural question arises: 'Given the diversity of tasks and datasets in ML4DB work, how can we effectively evaluate and select query plan representation methods?' More specifically, there are four open problems: **Unknown Interchangeability:** Query plan representation methods are often designed for a specific task, which may or may not perform well in other tasks. **Task and Dataset Dependency:** The performance of a representation method can vary on the specifications of tasks and datasets, making finding an effective method difficult. **Lack of Comparative Analysis:** There is lacking a comprehensive comparative study of the different representation methods across tasks and datasets. **Optimal Design:** Query plan representation methods use different feature encoding and tree models. It is unclear that what is the determining component of the performance of a method.

To address the problems, we design three research questions and answer each of them with comprehensive experiments.

RQ1 **How interchangeable are representation methods across various ML4DB tasks, and for each task, what representation method is the most effective?** To answer the question, we collate and implement ten query plan representation methods. We experiment them on three different ML4DB tasks, namely cost estimation [36], index selection [11] and query optimization [25].

RQ2 **What is the best tree model design?** As the existing methods use different feature encodings and tree models, the best choice of tree model designs is unknown. To answer the question, we standardize the feature encoding, and perform controlled experiment on tree models in existing works.

RQ3 **What is the best feature encoding?** We disentangle feature encoding from tree models in representation methods. Given the vast options of encoding elements (please refer to Section 2.2 for details), we prune the exploration space and primarily focus on cardinality features, as they are the most varied part among representation methods. Next, we study both their individual performances, and the impact of their combinations in different scenarios.

In summary, our work makes the following contributions and key findings:

- We identify a fundamental step of query plan representation in three ML4DB tasks that utilize query plans as input.
  - We confirm the interchangeability of query plan representation methods for different tasks.
  - We find some methods excel for the tasks that they are not designed for
- We break down query plan representation methods into two components: feature encoding and tree model. Through comprehensive controlled experiments, we conclude that the optimal design choice is intrinsically tied to the task's optimization target, to minimize absolute error or to identify correct relative ranking.
  - **Feature Encoding.** We identify the difference in feature encoding lies in cardinality-related features. For tasks aiming to reduce absolute error, a straight-forward strategy is to include as many candidate features as possible. Conversely, when the focus is the relative rankings, it is more beneficial to select a subset of features, and adding other features usually damage the robustness of the learned model.
  - **Tree Model.** To tasks optimizing absolute error, surprisingly, the choice of tree model is insignificant. However, on tasks to minimize relative errors, TreeCNN is preferred for in-distribution workloads, and LSTM performs the best for out-of-distribution workloads.

2 QUERY PLAN REPRESENTATION IN ML4DB

This section first gives a problem definition. Next, we break down a query plan representation into two stages: feature encoding and tree model, and summarize the existing techniques used in each stage. Finally, we describe each representation method.

2.1 Problem Definition

In this study, we focus on query plan representation in a single database application and dataset. We leave learning a general purpose model and transfer learning opportunities to future work. Consider a ML4DB system, query plan representation is the procedure which takes an arbitrary query plan as input, and output a vector representation for the query plan as the input to a machine learning model of the ML4DB system. The vector has to encode the important features, so that the machine learning model can leverage it to generate insights for the database system or the database user.

2.2 Feature Encoding

A query plan consists of information of heterogeneous types and correlated properties that affect its execution performance such as latency and throughput [36]. Hence, a necessary step in all query plan representation methods is to select and encode important features. A common practice is to select a subset of features for every node and concatenate the representation vector of each component as the node representation [46]. We summarize all the features that are used in query plan representation techniques below.

Operator. Operator specifies the physical operation performed by each node, such as *Hash Join* or *Sequential Scan*. Because of its importance, it is encoded in all representation methods. As a categorical variable, it is encoded either using one-hot encoding or a learned embedding vector.

Table. Table usually appears in scan nodes, specifying the input of an operation, and is a commonly included feature.

Column. Column is usually present in nodes that involve join or predicate. The size and underlying data distribution of a column helps to determine input and output sizes.

Join. Join specifies the two columns that the join operation works on. Some methods use the column representations to infer their interactions. Other methods assign a learned embedding vector to each possible join.

Predicate. A predicate is described by a $(column, operator, value)$ triplet. Predicates are commonly represented by concatenating the encoding for the three components, i.e., $|column, operator, value|$ [36, 46]. Other variants exist. For example, AVGDL uses LSTM to connect the triplet components [45].

Estimates. Cardinality and cost estimates are produced (and used) by cost-based database optimizers [5]. These estimators are usually built from database statistics with independence assumptions and magic numbers. Although not always accurate, machine

learning models can learn to make use of them [25]. We evaluate the impact of its inaccuracies in Section 6.1.

Histogram and Sample. Some methods incorporate other database statistics besides the estimates. For example, they encode raw database statistics like materialized samples for each table or histograms for each column [36, 46]. A bitmap is used to represent the satisfiability of each sample point or a histogram bin.

We note that a representation method uses only a subset of the above features, as some features may be considered redundant. For example, methods that encode predicates typically do not encode estimates, as these methods can ideally learn the distribution of data and infer the estimates [46].

2.3 Tree Model

Query plan representation methods have to transform the tree structures of query plans into a single fixed-size vector for machine learning models, such as a multi-layer perceptron (MLP). We summarize the techniques used in existing work below:

LSTM. LSTM is a recurrent neural network (RNN) with shortcut connections [16]. As there is no natural order to traverse a query plan with the tree structure, depth first search (DFS) is employed to first flatten a query plan [45]. The hidden state vector from the last node is treated as the query plan representation.

TreeRNN. TreeRNN models maintain the tree structure when aggregating node information. Some work use TreeLSTM, which generalizes the traditional LSTM cell by accepting inputs from multiple channels [38]. Other work proposes novel RNN-units to aggregate features with a tree structure [28].

TreeCNN. TreeCNN is a variant of the traditional convolutional neural networks (CNN), designed to handle tree-structured data [30]. It consists of convolutional layers with triangular shapes (parent-child-child) to slide over all the sub-trees. Dynamic pooling is used to aggregate all features to a single vector [26].

Transformer. Transformer utilize a stack of attention layers to aggregate information from a sequence [40]. QueryFormer generalizes to the tree structure of a query plan by modifying the positional encoding and attention formulation [46].

Feature Vector. We name the methods that do not use learnable parameters 'Feature Vector'. They directly encode the important features into a pre-defined size vector. To incorporate tree structures with varied shape and size, zero-padding is usually applied [11, 27].

2.4 Representation Methods

In this section, we will describe each representation method in ML4DB literature, focusing on feature encoding and the tree model.

AVGDL [45]. Yuan et al. proposed a view selection framework based on reinforcement learning called Automatic View Generation with Deep Learning (AVGDL in short). AVGDL extracts features from each node, including operator, tables, columns, and predicates. Numerical features are normalized, and non-numerical features are encoded as embedding vectors. AVGDL then employs an LSTM model to transform the features of a node into a fixed-length vector as the node encoding. Last, the query plan tree is flattened into a sequence using DFS, and a second LSTM is used to aggregate the node encodings into a vector representation.

Implementation: We reproduce the representation method based on the paper. We simplified *String* predicate encoding with a learned embedding. We employ similar treatment on all methods, as string predicate representation itself is a challenging research question [35] and may confound the findings of this study.

AIMEETSAI [11]. AIMEETSAI proposes an index selection framework, with a key insight of employing a classifier to predict the relative performance of query plans from different index configurations. It defines a set of feature channels related to each operator, and then incorporates structural information by defining additional channels with *weighted_sum* suffix, where the weights are the heights of the nodes in a query plan. To represent a query plan, it collects values for all feature channels from each node, and sums the values of all nodes by channels (assign zero to absent entries). This essentially encodes any query plan into a fixed-size vector. Note that this method does not have learnable parameters.
Implementation: We reproduce AIMEETSAI with five feature channels: *est_node_cost*, *est_rows*, *est_bytes*, *est_rows_weighted_sum* and *est_node_cost_weighted_sum*.

ReJOIN [27]. ReJOIN proposes a proof-of-concept reinforcement learning framework for join order selection. It represents a (full or partial) query plan by concatenating three components: tree structure, join and predicate. Tree structure is represented as a row vector, where each entry is the height of each table in a plan. Join and predicate are represented using multi-hot encoding, where each entry indicates the existence of predicate or join.
Implementation: We reproduce ReJOIN based on its paper.

Neural Optimizer (NEO) [26]. NEO proposes a reinforcement learning-based optimizer that builds a query plan end-to-end. It encodes a query plan in two parts: query encoding and plan encoding. Query encoding is plan-agnostic, which only includes join and predicate. Join is represented using a adjacency matrix and predicate encoding has three variants: one-hot, histogram, and R-Vector. The join and predicate encodings are concatenated and passed through linear layers to form the final query encoding. Next, plan encoding concatenates the operator and table encoding for each node, which is then appended with query encoding to form an augmented tree. The tree of vectors is passed to a TreeCNN network to produce the final query plan representation.
Implementation: We reproduce the representation method by adapting the open source code of BAO [2]. We use the histogram variant for predicate encoding. We do not use the R-Vector variant because it requires separate training using the tuples in a database, which can be unfair to other methods.

Bandit Optimizer (BAO) [25]. BAO is a successor of NEO, which introduces a reinforcement learning framework to improve an existing optimizer. BAO only includes the operator type, estimated cardinality, and estimated cost for each node in its query plan representation, and uses the same TreeCNN model as NEO.
Implementation: We use the open source implementation [2].

Prestroid [17]. Prestroid proposes a cost estimation model for large query plans. It concatenates the encoding of operator, table and predicate for each node. Operator and table are encoded as one-hot vectors, and predicate is encoded using a word2vec model [29]. Next, Prestroid employs a custom TreeCNN model with a sub-tree sampling algorithm to aggregate information and output a final representation.

Table 1: Summary of Existing Representation Methods in ML4DB works.

Method	Original Task	Tree Model	Encoding Information			
			Operator	Table	Predicate	Database Statistics
AVGDL [45]	View Selection	LSTM	Yes	Yes	Yes	-
AIMeetsAI [11]	Index Selection	Feature Vector	Yes	-	-	Estimates
ReJOIN [27]	Join Order Selection	Feature Vector	Yes	Yes	Column	-
BAO [25]	Optimizer	TreeCNN	Yes	-	-	Estimates
NEO [26]	Optimizer	TreeCNN	Yes	Yes	Column	Estimates
Prestroid [17]	Cost Estimation	TreeCNN	Yes	Yes	Yes	-
E2E-Cost [36]	Cost/Card Estimation	TreeLSTM	Yes	Yes	Yes	Sample
RTOS [44]	Join Order Selection	TreeLSTM	Yes	Yes	Yes	-
Plan-Cost [28]	Cost Estimation	TreeRNN	Yes	-	-	Estimates
QueryFormer [46]	General Purpose	Transformer	Yes	Yes	Yes	Sample & Histogram

Implementation: We reproduce the method based on the paper. We exclude the word2vec module. Similar to NEO, this simplification is to avoid separate training for fairness.

Plan-Cost [28]. Plan-Cost proposes a cost estimation model for query plans using a novel TreeRNN model. It defines two base type units called *neural units*. A leaf neural unit takes the operator type, estimated cost, and estimated number of rows as input, and outputs a vector representing data and estimated latency. An intermediate neural unit has an additional input channel from its child nodes. Plan-cost recursively applies the corresponding neural unit from the leaf nodes to the root node to obtain the final vector representation.
Implementation: We implement Plan-Cost based on a third-party reproduction [3]. Additionally, we implement a sampling module that groups query plan templates to support batch training.

E2E-Cost [36]. Sun and Li propose a cost estimation framework which estimates the cost and cardinality of query plans simultaneously. It encodes each node's operator, table, column and index in one-hot encoding. It represents predicates with concatenation in $|column, operator, value|$ format, and enriches it with a sample bitmap. It employs a string predicate embedding scheme with data mining techniques [13] and a word2vec model [29]. E2E-Cost uses a custom TreeLSTM model to aggregates information from all nodes.
Implementation: We used the original open-source codes [1], and exclude the string encoding as it requires separate training.

RTOS [44]. RTOS proposes a join order selection framework using reinforcement learning. RTOS first associates all columns with a learned embedding. RTOS then augments the column representation by multiplying it with a vector encoding predicate conditions. Next, a table is encoded by concatenating and pooling all its columns. Last, RTOS employs a custom TreeLSTM model to aggregate information from nodes.
Implementation: We reproduce the method based on its paper. We implement the TreeLSTM using the Deep Graph Library (dgl) [41].

QueryFormer [46]. QueryFormer proposes a general purpose query plan representation model for various database tasks. It first extracts features including tables, predicates, operators and encodes them in learned embeddings, and enriches with database statistics including samples and histograms. QueryFormer then aggregates the individual node encoding using a custom tree-structured Transformer model.
Implementation: We use the open-source implementation [4].

3 COST ESTIMATION

Cost estimation is a fundamental task essential to many database applications, such as query optimization [7, 12], index and view selection [6], scheduling [22], and more [10]. The objective is to predict the execution cost of a physical query plan, which is reflected in its execution latency. We note that cost estimation includes many other aspects, such as CPU, I/O, memory utilization, etc. We can adapt the framework by changing them as the ground truth labels, and the results can potentially be different. We scope this experiment to latency prediction to align with common practice in recent work [17, 28, 36]. To evaluate the performance of representation methods, we use the setting from E2E-Cost [36]. E2E-Cost encodes a query plan to a single vector using a novel encoding scheme and a TreeLSTM model. The representation vector is then passed to a MLP network as the estimation model. We replace the query plan representation component while keeping other parts the same. In this manner, we can compare the effectiveness of the representation methods from the cost estimation accuracy.

3.1 Experimental Setup

Datasets. We use four datasets from both industrial standard benchmarks and real-life databases to evaluate cost estimation. We summarize the datasets properties of the workloads in Table 2.

IMDB [24] is a movie database often used in work for join order selection, cardinality estimation [20], cost estimation [36], etc. It is challenging due to its high skewness and correlation between columns [20]. We use the same query workloads and splitting as they are used in E2E-Cost, i.e., 100,000 training queries are generated and deduplicated from templates with zero to two joins. We use two test workloads: 'Synthetic' consists of 5,000 queries generated from the same program with a different seed to test for in-distribution (ID) performance. 'Job-Light' consists of 70 handcrafted queries with meaningful semantics. It is more complex, with up to four joins and close-range predicates. Hence, it can test for out-of-distribution (OOD) performance.

TPC-H [34] and TCP-DS [31] are commonly used benchmarks to evaluate the performance of cost estimators [28] and index recommenders [23]. We generate and execute 50 queries for each query template in TPC-H, and 5 queries for each query template in TPC-DS. 20% of the queries are held out as test set. TPC-H and TPC-DS benchmarks contain much larger query plans as compared to IMDB.

Table 2: Dataset properties.

Dataset	Max #	Avg #	Skewness	Correlation
TPC-H	26	16.8	Uniform	Uncorrelated
TPC-DS	143	44.4	Uniform	Uncorrelated
Synthetic	10	4.9	Moderate	Moderate
JOB-Light	14	8.44	Moderate	Moderate
STATS	16	9.49	High	High
Job-Extend	73	21.2	Moderate	Moderate

STATS [14] is a real-world dataset originally used in cardinality estimation. It is challenging because it comes with more relations, columns, higher skewness and correlation between columns, and more complex topology, i.e., it contains a mix of star joins and chain joins [14]. Following the original work, we use 70,142 generated queries for training, and 146 hand-crafted queries as the test set, which test for OOD performance.

Evaluation Metrics. We use Q-Error to evaluate the accuracy of estimated latency, following E2E-Cost. It measures the multiplicative error as: $\frac{\max(estimated\_cost, actual\_cost)}{\min(estimated\_cost, actual\_cost)}$.

Implementation. We connect the vector output of query plan representations to an MLP model to predict the cost, following E2E-Cost [36]. The MLP models for each representation method have the same structure and layers, but their hidden sizes are tuned individually as hyper-parameters. All learnable parameters in the representation methods and the MLP model are trained end-to-end. Finally, we include the estimations of Postgres as baseline. Since the cost estimates from Postgres has no unit, we use linear regression to map the estimates from Postgres to the actual latency.

Model Training. In model training, we employ ADAM [19] optimizer coupled with a step learning rate scheduler for all methods. Hyper-parameters are individually tuned for each method using a random search approach. Specifically, we jointly permute the learning rate, the hidden size of the MLP, and other method-specific hyper-parameters. Training is halted based on an early stopping criterion: we cease training if the validation loss fails to decrease by at least 0.1% or upon reaching 200 epochs, whichever comes first.

3.2 Overall Cost Estimation Performance

We evaluate the cost estimation performances to answer (1) how interchangeable are representation methods to database tasks, (2) what is the best performing methods to the task. We present the cost estimation results in Table 3.

Performance Summary. We rank the representation methods based on their scores for each dataset below. Specifically, each '>' symbol indicates a significant difference in performance, with the methods on the left outperforming those on the right.

- **TPC-H:** {Other Methods} > {Plan-Cost, E2E-Cost}.
- **TPC-DS:** {Other Methods} > {AIMEETSAI} > {Plan-Cost, E2E-Cost}.
- **Synthetic:** {QueryFormer, BAO, AIMEETSAI, E2E-Cost, AVGDL, RTOS} > {Plan-Cost} > {NEO, Prestroid, ReJOIN}.
- **Job-Light:** {QueryFormer, E2E-Cost, Plan-Cost, BAO, AIMEETSAI} > {ReJOIN, NEO, Prestroid, AVGDL, RTOS}.
- **STATS:** {QueryFormer, BAO} > {Other Methods} > {ReJOIN, Plan-Cost}.

Observation: Most representation methods can handle large query plans. Most methods have near perfect performances for TPC-H and TPC-DS as shown in Table 3.This attributes to the uniform distribution in the database columns, hence letting the prediction layer to accurately capture the correlation between input features to output costs. There are a few exceptions. E2E-Cost and Plan-Cost have manually implemented or designed recurrent model design, making them struggle for a large number of nodes [32]. AIMeetsAI has high tail error in TPC-DS. This is because TPC-DS has a few extremely large query plans, which may mess up the 'height' term in its formulation. This essentially demonstrates an intrinsic limitation to rule-based representation methods.

Observation: Performances among representation methods differ on real-life datasets, especially on OOD workloads. On Synthetic (ID) workload, representation methods demonstrate larger differences in scores compared to on TPC- benchmarks. Around half of the representation methods (QueryFormer, BAO, AIMEETSAI, E2E-Cost, AVGDL, and RTOS) perform similarly well, i.e., their median Q-Error deviate by about 7%. However, other methods like NEO and Prestroid have 88% higher median Q-Error and max Q-Error up to 4 digits. The larger discrepancy suggests that representation methods have different capacity in their designs to model the complex data distribution. Furthermore, the differences in scores for the Job-Light and STATS (OOD) workloads are much larger. For example, among the best performing models (QueryFormer and BAO), the differences in median Q-Error exceed 50%, and between the highest and the lowest scores are up to 570%.

3.3 Analysis of Tree Models

Our experiments illustrate that representation methods, each made up of a different feature encoding strategy and tree model, have different performances. This makes it challenging to attribute performances to design choices. To this end, we standardize the feature encoding in all these methods, so that we can rigorously assess and compare the performance of different tree models. There are five tree models: TreeCNN, TreeLSTM, LSTM, Transformer and Feature Vector, as summarized in Section 2.3. For Feature Vector, we use AIMeetsAI's formulation instead of ReJOIN due to its higher performance. We use the feature encoding from BAO as the standardized encoding because of its high performance. We perform the same suite of experiments and present the results in the second block of Table 3.

Observation: The performance differences among tree models are generally marginal! In ID and most OOD workloads, all tree models show practically identical performances. This unexpected finding suggests that the choice of tree model may not be as consequential as previously assumed! There are two minor exceptions: (1) Feature Vector shows large tail errors on TPC-DS, JOB-Light, and STATS, similar to the original AIMEETSAI's performance. (2) RNN-based methods (TreeLSTM and LSTM) slightly underperform in the tail regions for OOD workloads. This could be attributed to the difficulty in training for RNN models [32].

3.4 Analysis on Feature Encoding

We investigate the impact of feature encoding methods. There are two challenges. First, there is a large candidate pool of possible

Table 3: Cost Estimation Accuracy (Q-Error) on five test workloads. The first black is the original methods that we evaluate, followed by PostgreSQL as baseline. The subsequent blocks are the tree models (Section 3.3) and feature encodings (Section 3.4). The highest scores in each block are marked bold. The overall best scores across blocks are underlined.

Dataset	TPC-H				TPC-DS				Synthetic				JOB-Light				STATS			
	50th	90th	99th	Max	50th	90th	99th	Max	50th	90th	99th	Max	50th	90th	99th	Max	50th	90th	99th	Max
AVGDL	1.010	1.067	1.222	1.297	1.045	1.925	4.563	5.754	1.284	4.065	22.62	242.8	3.962	47.22	412.5	427.2	3.702	22.11	533.7	824.9
AIMEETSAI	1.044	1.142	1.268	1.392	1.167	3.127	616.6	1329	1.204	3.250	14.22	124.2	2.033	51.92	1085	1561	2.106	22.60	17930	1.2e5
ReJOIN	1.052	1.964	11.37	14.93	**1.044**	1.756	4.301	5.326	2.851	36.83	490.9	1639	3.276	38.49	1016	2746	8.941	726.7	43540	96690
BAO	**1.008**	1.095	1.285	1.331	1.085	1.650	**3.621**	**3.956**	1.142	3.169	13.42	89.14	2.242	**18.20**	268.0	**316.0**	2.028	32.81	336.5	457.2
NEO	1.027	1.122	1.845	2.240	1.034	1.437	4.388	5.669	2.106	21.76	476.7	1009	3.924	37.76	483.7	716.0	2.622	50.22	2657	34630
Prestroid	1.011	1.457	1.792	8.196	1.212	2.344	6.267	9.333	2.012	20.68	382.8	999.2	4.324	47.7	343.2	584.4	2.695	30.01	1213	17600
Plan-Cost	2.711	13.56	72.91	79.52	2.016	6.597	113.9	295.3	1.633	4.492	18.04	93.82	2.070	45.38	182.8	189.5	8.035	283.9	11570	30380
RTOS	1.071	1.414	1.750	1.961	1.064	1.934	5.154	6.615	1.321	3.497	13.78	235.7	4.013	105.1	1689	1853	3.793	48.57	2304	32650
E2E-COST	3.253	6.023	10.92	13.46	2.100	3.827	18.79	50.18	1.257	2.748	14.12	296.2	1.771	29.68	417.0	433.0	3.831	44.37	685.1	1270
QueryFormer	1.028	**1.052**	**1.137**	**1.259**	1.053	**1.296**	5.058	7.499	**1.065**	**1.718**	**13.05**	**20.53**	**1.408**	20.28	**176.6**	380.0	**1.322**	**8.354**	**60.41**	**132.1**
POSTGRES	2.064	8.547	85.85	1131	2.972	13.88	27010	116900	3.775	16.48	202.8	1362	2.742	20.90	213.6	455.9	6.431	518.6	5205	8565
TreeLSTM	1.028	**1.053**	**1.134**	**1.256**	1.031	1.608	4.110	4.477	1.236	3.444	14.70	110.7	1.913	32.10	225.1	319.7	**1.803**	36.48	950.3	12030
Transformer	1.029	1.177	1.371	1.394	1.127	1.847	4.047	5.345	1.241	3.461	14.56	143.7	**1.621**	22.36	201.3	**247.5**	2.363	**20.61**	704.6	3858
LSTM	1.024	1.065	1.198	1.332	**1.024**	**1.282**	5.085	6.562	1.168	**3.110**	**12.80**	134.7	2.108	78.49	704.6	758.6	2.022	41.77	1606	70800
TreeCNN	**1.008**	1.095	1.285	1.331	1.085	1.650	**3.621**	**3.956**	**1.142**	3.169	13.42	89.14	2.242	**18.20**	268.0	316.0	2.028	32.81	**336.5**	**457.2**
Feature Vector	1.027	1.084	1.315	1.448	1.081	1.763	33.87	72.27	1.185	3.218	13.61	**81.51**	2.012	34.21	**138.0**	284.3	2.129	24.31	1085	12510
Empty	**1.007**	1.084	1.404	1.464	1.027	1.773	3.996	4.195	2.013	20.52	380.5	1042	4.197	31.65	420.6	687.2	2.598	25.38	789.6	33070
Pred	**1.007**	1.046	1.430	1.493	**1.026**	**1.429**	4.629	6.025	**1.141**	**2.674**	**11.5**	764.3	2.985	62.54	**154.7**	**258.9**	**1.584**	**8.734**	**278.8**	**3172**
Hist	**1.007**	1.051	1.387	1.447	1.030	1.709	4.677	6.315	1.760	12.61	220.2	3386	4.622	**28.47**	466.6	676.7	2.367	45.44	3488	55780
Sample	1.010	**1.041**	1.450	1.512	1.041	1.731	4.386	5.051	1.160	6.670	118.7	2004	4.490	67.19	468.9	779.5	3.236	202.6	11480	65770
Est	1.015	1.046	**1.372**	**1.419**	1.033	1.680	**3.307**	**4.188**	1.169	3.393	13.70	**87.90**	**2.577**	29.16	249.5	675.5	2.016	16.29	707.5	18070
Pred_Est	**1.007**	**1.034**	1.144	1.285	1.022	1.357	**2.751**	4.194	1.084	**2.050**	10.37	782.2	2.383	24.06	304.3	514.6	1.579	11.57	76.90	541.9
Pred_Hist	1.008	1.050	1.399	1.459	1.031	1.797	3.986	**4.181**	1.155	2.836	16.68	470.5	4.033	50.84	715.8	1564	1.605	14.29	465.0	16550
Pred_Sample	1.019	1.065	1.505	1.569	1.032	1.787	4.001	4.204	1.105	2.937	15.15	767.4	3.489	27.09	**260.1**	**466.7**	1.739	14.32	560.0	3117
Pred_Est_Hist	1.009	1.036	1.147	1.250	**1.017**	1.272	2.858	4.196	1.086	2.062	**10.28**	739.0	2.772	32.05	1192	1324	1.615	10.26	226.8	3409
Pred_Hist_Sample	1.013	1.053	1.382	1.441	1.030	1.781	4.217	4.311	1.097	2.962	18.78	**470.3**	4.464	45.00	308.8	636.2	1.833	9.486	1080	1824
Pred_Est_Sample	1.010	1.035	1.180	1.316	1.025	1.386	4.118	4.196	1.064	2.195	11.02	632.5	2.321	23.92	788.6	796.4	**1.383**	10.46	**28.00**	**37.32**
ALL	1.016	1.047	**1.139**	**1.246**	1.021	**1.250**	4.547	5.774	**1.060**	2.180	12.16	737.1	**2.314**	**22.20**	290.1	590.9	1.461	**6.605**	239.4	332.7

features. Since each representation method typically encodes a different subset of features, the number of possible combinations for features is exponential. Second, existing works use various encoding formulations, such as concatenation, LSTM, or rule-based designs like in AIMEETSAI or RTOS. This adds more options to the possibility of feature encodings.

To make the study feasible, we make two simplifications. First, to select features to study from the candidate pools, we observe three commonalities in almost all existing encoding methods: (1) All methods include *Operator* in the encoding. (2) Almost all methods include *Table*, because it provides the information of the input to a node. (3) All methods include some encoding parts that are related to cardinality. This is because they determine the output size of a node. Based on this observation, we primarily focus on the part where most representation methods are different on: *Cardinality_Features*. Second, we unify the encoding using concatenation, as it is the most commonly used (five out of ten) approach, and can easily accommodate any feature in the experiment. In other words, we formulate the encoding in the form of $|Operator, Table, Cardinality\_Features|$.

The candidate pool for *Cardinality_Features* includes two categories: predicate and database statistics. Predicate (denote as *Pred*) encodes the filter conditions. Database statistics include *Sample*, histogram (denoted as *Hist*) and estimates (denoted as *Est*). Please refer to Section 2.2 for details. We use *Tree-CNN* as the tree model due to its consistent performance in previous studies.

Our study is twofold. First, we compare the single feature performance to identify the most effective feature. For comparison, we include a simple baseline without any *Cardinality_Feature*, denoted as *Empty* (it still has *Operator* and *Table* in the encoding). Next, we examine the combinations of the above features, because a feature encoding scheme is not restricted to using only one feature. We present the results on the bottom two blocks of Table 3 respectively.

Performance Summary. For the TPC- benchmarks, all *Cardinality_Features* and their combinations have similarly good performances, which align with previous observations.

In Synthetic (ID) workload, *Pred* and *Est* significantly outperform other *Cardinality_Features*, achieving less than a 20% median error from the perfect score. Among them, *Est* has the best performance, evident from its lowest tail error. For the combination of features, despite the individual differences, all combinations show similar performance and are usually better than single element feature.

In JOB-Light and STATS (OOD) workloads, *Pred* is the most effective feature, closely followed by *Est*. This is evident from both median and max Q-Errors. For feature combinations, those incorporating both *Pred* and *Est* perform better than others.

Observation: *Pred* and *Est* are the most impactful features, and their combination can yield superior results. Existing works have never included both of them together! *Pred* and *Est* features prove to be highly effective on datasets with skewness and correlation, and for OOD queries. As discussed in Section 2, *Pred* and *Est* are typically treated as mutually exclusive in representation method design. Different from assumption, our experiments show that their combination can lead to better performance. This is because machine learning models can leverage these complementary features for different query plans. In fact, adding so-called 'bad' features, i.e., *Hist* and *Sample*, which underperform as a single feature, can enhance performance in most cases as well. For example, *Pred_Est_Sample* consistently outperform *Pred_Est*. This finding

suggests that in practice, a good first attempt when applying query plan representation in cost estimation tasks can be encoding all potentially useful features.

3.5 Key Findings

The main findings are as follows:

- QueryFormer and BAO consistently demonstrate superior performance across all scenarios.
- Tree models have relatively minor impact to performance.
- Feature encoding is crucial to performance. *Pred* and *Est* are the most effective features, and including as many features as possible can be a default choice.

4 INDEX SELECTION

In this section, we evaluate the effectiveness of query plan representation methods on the task of index selection, following the setting of AIMEETSAI [11]. Index selection task refers to the process of picking a index configuration with the least total cost for a query or a workload on a given database [9]. To select the most beneficial index configuration, a index tuner utilizes an optimizer with the "what-if" API [8] to obtain query plans for hypothetical index configurations. In this way, the task of selecting better index configurations is transferred into selecting better query plans.

Conceptually, the cost estimators as evaluated in Section 3 can be directly applied to solve the task. AIMEETSAI demonstrated that cost estimation task is inherently difficult, and the inaccuracies in cost estimators lead to large prediction errors. On the other hand, a classification-based model that is trained on pairs of plans directly minimizes comparison errors, leading to improved accuracy [11]. Specifically, AIMEETSAI uses a representation technique (as discussed in Section 2) , and constructs a 'plan pair' by taking the differences of two query plan representations. The 'plan pair' representation is fed to a classifier with three labels: *IMPROVE*, *REGRESS*, or *NO_DIFF*. Using the classifier, an index tuner can select an optimal index configuration for a given query.

In line with AIMEETSAI's approach, we replace the query plan representation module and choose MLP as the classifier. The original paper discussed various classifier models such as random forest, logistic regression and etc. We selected MLP because most query plan representation methods require training parameters, and using MLP allows the framework to be trained in an end-to-end manner.

4.1 Experimental Setup

Evaluation Metrics. We use the average F1 score instead of accuracy to evaluate the classification performances. It defined as: $F1 = 2 \times (precision \times recall)/(precision + recall)$. F1 score is advantageous over imbalanced datasets. For example, only a few index configurations may lead to improvements for some queries, where most configurations have insignificant effect. In this case, F1 score takes into account of all three class labels, not favoring a model that only predicts accurately for a specific class, such as *NO_DIFF*.

Dataset. We use TPC-H, TPC-DS (the original datasets used in AIMEETSAI), and STATS for index selection. For each query, we generate potential index configurations using the toolkit from [23], and execute the queries with each index configuration to collect the actual execution time. We generate on average 6.4 distinct index configurations (and hence, query plans) for each query in TPC-H, 71.9 for TPC-DS and 10.4 for STATS. We split the datasets using 3 strategies following AIMEETSAI with increasing difficulty:

(1) **Pair:** The train and test pairs are randomly split.
(2) **Plan:** The plan pairs in test set contain at least one unseen query plan. This simulates the real-life index tuner setting.
(3) **Query:** The model is tested on unseen queries.

Model Training. We use similar training and hyper-parameter tuning strategy as used in cost estimation task.

4.2 Overall Index Selection Performance

Performance Summary. The average F1 scores of all methods on the index selection task are presented in Table 4. We rank the representation methods based on their scores, with methods on the left performing better than the right:

- **TPCH:**
  Pair: {Other Methods} > {ReJOIN}.
  Plan: {Other Methods} > {AIMEETSAI}.
  Query: {QueryFormer} > {RTOS, BAO, NEO} > {ReJOIN, AIMEETSAI} > {Plan-Cost, Prestroid, AVGDL}.
- **TPC-DS:**
  Pair & Plan: {QueryFormer,AVGDL, NEO, BAO, RTOS} > {Prestroid, AIMEETSAI} > {ReJOIN, E2E-Cost, Plan-Cost}.
  Query: {QueryFormer, BAO} > {NEO, AVGDL, AIMEETSAI, Prestroid, ReJOIN, RTOS } > {E2E-Cost, Plan-Cost}.
- **STATS:**
  Pair: {AVGDL, QueryFormer, E2E-Cost, BAO} > {RTOS, NEO, Prestroid} > {AIMEETSAI, Plan-Cost, ReJOIN}.
  Plan: Same as **Pair** except E2E-Cost drops to tier 2.
  Query: Plan-Cost promotes to tier 1, and AVGDL, RTOS, E2E-Cost drop to tier 3.

Observation: The relative performance rankings on Pair and Plan splittings are mostly consistent, but deviates on Query splitting. This suggests different levels of robustness among methods. We observe that the performance rankings for Plan splitting tend to mirror Pair splitting (with a few exceptions). It can be seen that most methods score a few points lower when moving on to Plan splitting due to the increased difficulty. This suggests that interchangeability of these metrics to evaluate representation method performance. However, this does not hold in Query splitting. For example, AVGDL, E2E-Cost and Prestroid show significant performance decline from Plan to Query splitting, particularly evident on TPC-H. Their scores drop to the level of a random guess. This observation shows overfitting among these methods, suggesting they are learning patterns overly specific to training data that fail to generalize to unseen queries.

A different type of inconsistency arises due to 'underfitting'. This phenomenon is observed in the performances of Plan-Cost on STATS and ReJOIN on TPC-H. These methods underperform on Pair splitting. This implies that they do not capture the task-related features effectively. To explain, ReJOIN as a simplistic design does not encode information regarding index, thus relying on join order to distinguish query plans. Plan-Cost, an early RNN-based method, does not incorporate LSTM components like other RNN-based approaches, limiting its effectiveness. Interestingly, the underfitting

Table 4: Index Selection Performance.

Method	TPC-H			TPC-DS			STATS		
	Pair	Plan	Query	Pair	Plan	Query	Pair	Plan	Query
AVGDL	0.992	0.866	0.255	0.940	0.888	0.629	**0.928**	**0.865**	0.536
AIMEETSAI	0.989	0.734	0.549	0.854	0.801	0.613	0.787	0.673	0.549
ReJOIN	0.777	0.861	0.561	0.824	0.787	0.593	0.672	0.623	0.505
BAO	0.991	0.932	0.68	0.920	0.843	0.705	0.900	0.808	0.566
NEO	0.964	0.929	0.636	0.927	0.865	0.664	0.870	0.795	0.536
Prestroid	0.970	0.911	0.374	0.902	0.838	0.596	0.868	0.775	0.572
E2E-Cost	0.984	0.888	0.238	0.824	0.753	0.528	0.907	0.755	0.520
PLAN-Cost	0.954	0.863	0.412	0.767	0.736	0.527	0.737	0.712	**0.613**
RTOS	0.991	**0.961**	0.684	0.918	0.877	0.588	0.886	0.846	0.531
QueryFormer	**0.993**	0.920	**0.770**	**0.950**	**0.907**	**0.710**	0.919	0.857	0.609

in these methods may serve as a form of implicit regularization, which enhances their performance on more challenging splittings.

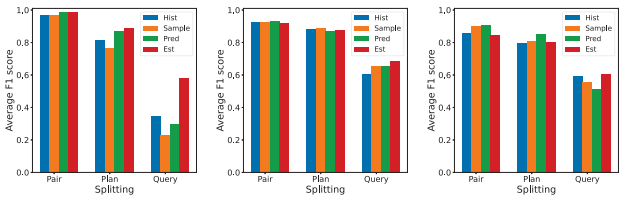
4.3 Analysis on Feature Encoding

We isolate the tree model designs and examine feature encoding performances. We fix the tree model using LSTM and compare the encoding elements. Similar to cost estimation, we formulate the node encoding in $|Operator, Table, Cardinality\_Features|$ format and focus on comparing *cardinality_Features*. We compare the single element encoding performance, and present the results in Figure 2. Next, we present the combination of encodings in Figure 3. Due to space constraint, we only present the encoding combinations on STATS, as the general trend on other datasets is similar.

Observation: *Est* is the most effective feature in generalization. Different from cost estimation, adding more features in the combination does not help. In easy splittings, such as Pair and Plan, the rankings of performance among encoding elements are mostly consistent. This is shown in Figure 2, where the patterns of the Pair and Plan groups are similar. This observation is seen in the original methods as well. However, on Query splitting which is the most challenging scenario, *Est* is clearly the most effective feature, as shown in Figure 2. In TPC-H, it is the only encoding that has a reasonable F1 score for the ternary classification task, whereas other encodings fall to the level of random guess due to overfitting. This is because *Est* is directly derived from the optimizer, where other features require the model to learn the relationship between them to the execution cost. These relationships learned on a set of queries may not be transferable to completely different query templates.

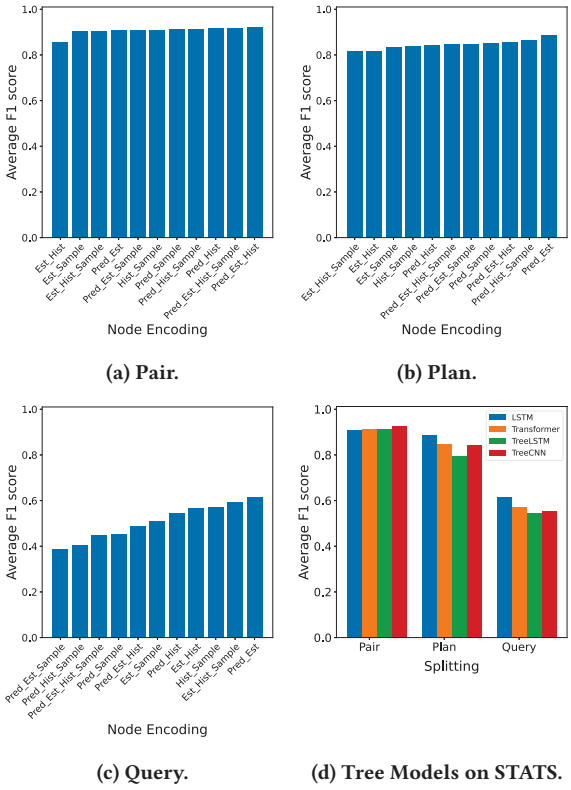
Moreover, different from the cost estimation experiments where encoding as many features as possible is beneficial, the opposite is observed in Figure 3. In Query splitting, all combinations cannot outperform *Est* alone. Intuitively, those under-performing features may be helpful to roughly gauge the execution cost. However, they are not useful in comparing query plans from the same query, because the absolute time does not matter.

Observation: *Pred* is effective in easy splittings, especially in real-life dataset. *Pred* has the highest score in Pair and Plan splitting in STATS, and is close to highest score in TPC- benchmarks, as shown in Figure 2. This is because *Pred* can effectively capture the relationship between query regions and output cardinality [20]. In combination of features as shown in Figure 3, we observe that those with *Pred* generally perform better in Pair and Plan splittings.



(a) TPCH. **(b) TPCDS.** **(c) STATS.**

Figure 2: Single Element Encoding Performance.



(a) Pair. **(b) Plan.**

(c) Query. **(d) Tree Models on STATS.**

Figure 3: Comparison of Encoding Combinations (in a, b, c) and Tree Models (in d) Performances on STATS.

4.4 Analysis on Tree Models

We compare the performance of different Tree Models. We fix the feature encoding using *Pred_Est*, because this combination shows universally good performance across different splittings. We present the tree model performance for STATS dataset in Figure 3d.

Observation: LSTM is the most discriminative tree model, especially in hard cases. Although LSTM has the lowest score in splitting by Pair (where most tree models have similar score), it outperforms other tree models in Plan and Query splittings. This suggests its effectiveness in generalization. TreeLSTM, on the other hand is the least effective model in both Plan and Query splittings.

4.5 Key Findings

Index selection task requires the representation methods to distinguish better query plans for the same query. The findings are indeed very different from cost estimation.

- QueryFormer and BAO perform the best among all representation methods.
- Est is the most important feature. Unlike in cost estimation, more information in the encoding does not benefit performance.
- LSTM is the most effective model to differentiate query plans, especially on OOD scenarios.

5 QUERY OPTIMIZATION

In this section, we evaluate the performance of query plan representation methods on the task of query optimization. We use the framework of BAO, which is proposed to enhance an existing query optimizer by offering per-query hints through reinforcement learning [25]. Specifically, it frames the optimization task as a bandit optimization problem, and proposes to use a Thompson Sampling method to solve the problem. A key component in the framework is comparing the execution cost of a set of query plans derived from a pre-defined set of hints. We evaluate the representation methods by substituting them in the value network. To avoid any confusion, we henceforth refer to the task as 'optimization', and the original query plan representation used as 'BAO'.

Although the value network in optimization seemingly resembles cost estimation, the performances of representation methods can vary significantly, because the task has three distinct requirements: (1) **Data:** collecting training data (or exploration) is at the cost of executing queries with suboptimal query plans, which impacts the overall execution time. Sufficient training data as used in Section 3 is a luxury to afford. (2) **Metric:** similar to index selection, the correct relative rankings of query plans are more important than the absolute errors. To illustrate, we present a case study in Section 6.3. (3) **Efficiency:** both training and inference need to be efficient as they constitute to the overall runtime.

5.1 Experimental Setup and Results

Dataset. For this experiment, we utilize the same IMDB database as in the cost estimation scenario, but generate a new workload, referred to as JOB-extend, in accordance with the specifications laid out by the original work [25]. It comprises 1697 distinct queries derived from the same templates as JOB. Please refer to Table 2 for the dataset properties. The queries are arranged based on a time-series split strategy: the system always evaluate the next unseen query, and new templates are introduced periodically.

Evaluation Metrics. We evaluate the optimization performance by the total time taken executing a workload, which comprises both the model training and inference (referred to as model time) and the query execution time. The query execution time depends on the quality of query plans selected, indicates model accuracy, and the model time indicates the efficiency of representation methods.

Implementation. Following the original paper, we connect the representation methods to a MLP model to estimate the rewards of query plans. The experiment has three system hyper-parameters: retrain frequency f, training size N, and experience size P. f denotes to the interval at which we retrain the advisor model after a certain number of queries have been executed. A higher f and N can ideally improve the model's accuracy, but leads in larger training overhead. P refers to the number of most recent k query executions in the sample pool, where older executions are discarded. This is because the system cannot retain past executions indefinitely. We set f=100, N=100, P=800. For fairness, these system hyper-parameters are held constant across all representation methods.

We introduce two baselines for comparison: query execution time of Postgres and query execution time when the best possible query plan is selected for every query (refer as Best Possible).

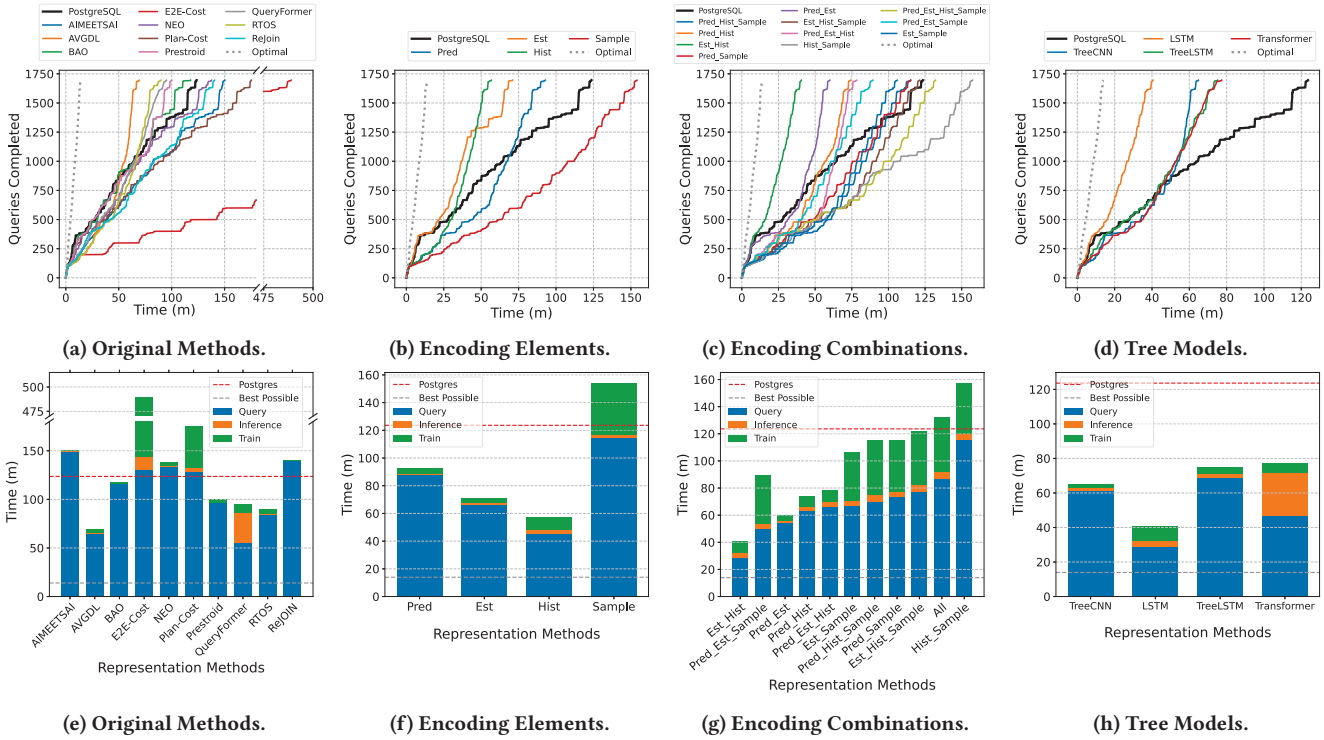
Results. We present the query optimization performance in Figure 5c. The goal is to complete the query workload as fast as possible: ideally approximating the 'Best Possible' curve on the far left. Additionally, Figure 4e presents the composition of time taken on model training, inference (model time), as well as query execution time.

Performance Summary. The overall performance ranking of representation methods is as follows: {AVGDL, RTOS, QueryFormer, Prestroid, BAO} > {NEO, ReJOIN, AIMEETSAI} > {Plan-Cost, E2E-Cost}. As shown in Figure 5c, half of the representation methods can outperform Postgres, with AVGDL demonstrating the most significant improvement by almost reducing the workload execution time by 44%. Next, RTOS, QueryFormer, Prestroid and BAO can reduces the workload time by 27%, 23%, 19% and 4.5% respectively. Other methods have negative impact to the workload execution. In particular, Plan-Cost and E2E-Cost performs the worst, which increase the total time by 40% and 380% respectively.

In terms of model efficiency, E2E-Cost and Plan-Cost have high training overhead as seen in Figure 4e, rendering them unsuitable for the task. They can be potentially optimized through re-implementation with new modern tools, such as the Deep Graph Library (dgl) [41] as we used in the implementation of RTOS. QueryFormer has high inference overhead. Through investigation, the inference time comes from computing the relative distances between nodes during pre-processing. Other methods have relatively insignificant overheads compared to query execution time.

Observation: Good performing methods in the optimization task can improve the system early with limited training data. As an RL-based framework, a 'warm-up' phase of exploration is required to gather diverse training data, and obtain knowledge of query plans from other hints as queries are executed. Hence, during this phase, a system is not expected to provide better predictions. Indeed, as seen in Figure 5c, none of the methods surpass Postgres for the first 700 queries. However, certain representation methods, such as AVGDL, starts generating better query plans than Postgres after processing merely 800 queries, as seen in Figure 5c. This rapid adaptability makes AVGDL superior compared to other representation methods. In contrast, methods such as AIMEETSAI demonstrate effective predictions for the final 300 queries, as shown from the steep gradient (indicating less time taken executing the queries) in Figure 5c. However, the late-stage efficacy cannot offset the time invested in the earlier queries.

Observation: Balancing accuracy and efficiency is important in optimization task. As shown in Figure 4e, QueryFormer has the lowest query execution time, demonstrating its high accuracy in the task. However, it falls behind RTOS and AVGDL due to its high inference time. Conversely, AIMEETSAI, BAO and ReJOIN



(a) Original Methods. **(b) Encoding Elements.** **(c) Encoding Combinations.** **(d) Tree Models.**

(e) Original Methods. **(f) Encoding Elements.** **(g) Encoding Combinations.** **(h) Tree Models.**

Figure 4: Performance of Representation Methods on Optimization Task. Top row is query execution progress over time. Bottom row is the overall compositions of the time taken.

are the most efficient methods in both training and inference time. However, as their inability to select the optimal query plans, they are beaten by more accurate alternatives.

5.2 Analysis on Feature Encoding

We examine the efficacy of node encodings with the fixed tree model. Similar to previous sections, we format the encoding as $|Operator, Table, Cardinality\_Features|$, and compare the performance of *Cardinality_features*. We standardize the tree model as LSTM, because AVGDL with LSTM demonstrated the best overall performance in the experiments. Results for single *Cardinality_features* comparison are presented in Figure 4b and 4f, and their combinations are presented in Figure 4c and 4g.

Performance Summary. *Est* and *Hist* are the best as single feature encoding, and their combined encoding yield the best overall score, reducing the total time by 67% as compared to Postgres. *Sample* performs the poorest, increasing the total time by 25%. In feature combinations, a general pattern is the inclusion of *Est* or *Hist* enhances performance, while *Sample* hampers it.

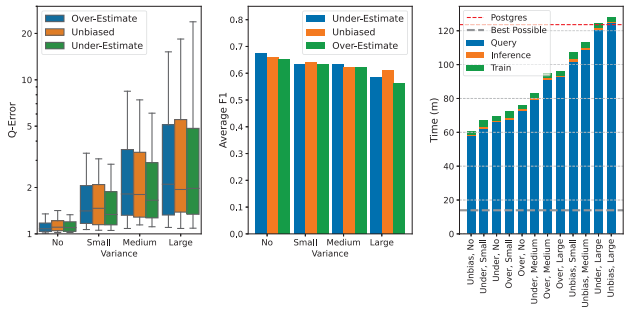
In terms of efficiency, *Pred* and *Est* perform similarly well, by having less than 5 minutes in model overhead. *Hist* has 2x training time and 2.5x inference time compared to *Est* and *Pred*. However, it selects better query plans as reflected in the query time. *Sample* is the least efficient, with 9x training time and 3x inference time. The trend in efficiency largely holds for combinations of elements as well: the overheads are about the sum of the ingredients.

Observation: Different encoding elements (except *Sample*) lead to similar exploitation effectiveness, but differ in exploration efficiency. Interestingly, the query optimization performances for the final 500 queries are similar among *Pred*, *Est*, *Hist* and their combinations, as seen from the similar shape of their curves at the last part in Figure 4b and 4c. However, their differences in exploration are more prominent. *Est*, for instance, begins to outperform Postgres after about 1,000 queries, and performs similarly to Postgres before that point (as seen in Figure 4b). *Pred* and *Hist* start to outperform Postgres after running about 400 queries.This is because *Est* is directly derived from Postgres, so it is reasonable for a model built on the *Est* feature to mirror Postgres' in early stages. In contrary, a model needs to learn the correlation between *Pred* or *Hist* and query latency from scratch, which lead to inevitable early-stage mistakes. As a result, the combination of *Est* and *Hist* showcases the inherent benefits of both: it avoids serious mistakes in the early stage and makes better predictions earlier, making it the best choice for this task.

5.3 Analysis On Tree Models

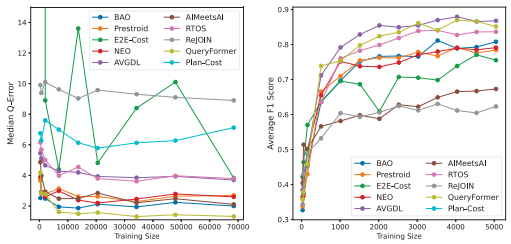
We fix the encoding using the most performing combinations from the last experiment (*Est_Hist*), and compare the tree models in Figure 4d and 4h. It can be seen that all tree models can outperform Postgres by a large margin (at least 37% time reduction). The ranking for these tree models are as follows:

- **Performance:** {LSTM} > {TreeCNN} > {TreeLSTM, Transformer}.



(a) Cost Estimation. (b) Index Selection. (c) Optimization.

Figure 5: Performance under CE noises.



(a) Cost Estimation. (b) Index Selection.

Figure 6: Accuracy with different training size. (a) and (b) are for cost estimation, (c) and (d) are for index selection.

- **Efficiency:** {TreeCNN} > {TreeLSTM} > {LSTM} > {Transformer}.

Observation: There is a trade-off between accuracy and overhead time in tree model selection. Despite that LSTM has 3x overhead than TreeCNN, it selects superior query plans, resulting in a substantial reduction in time. On the other hand, the Transformer selects better query plans than both TreeLSTM and TreeCNN, it ranks the last due to its high inference time.

6 OTHER EXPERIMENTS

6.1 Feature Noise Analysis

Cardinality estimates (CE) are shown to be an effective feature in previous experiments. However, since cardinality estimation is inherently challenging, the estimates are noisy and erroneous [42]. In fact, the median Q-Error is about 1.4, and the max Q-Error is more than four digits in the datasets. To this end, we study the impact of the feature noises in CE to downstream tasks performance.

We manually set the CE of all plan nodes to study their impacts. We consider three scenarios: under-estimation, unbiased estimation, and over-estimation, and scale each with different variances. We use LSTM model and feature encoding with $|Operator, Table, Estimates|$, the same format as previous experiments for feature encoding analysis. We present the results in Figure 5.

We observe that the two types of biases do not significantly affect task performance in all three tasks. However, the magnitude of variance directly impacts the accuracy across three tasks. We conjuncture that machine learning models, unlike traditional methods, can automatically learn to adjust to the biases. The uncertainties from the random errors are more detrimental to the performances.

6.2 Training Size Analysis

To demonstrate that all models are sufficiently trained and examine the potential issues of under/over-fitting, we experiment all representation methods with different training sizes in cost estimation and index selection. We exempted query optimization from this analysis due to its inherent challenge with limited training data in its task setting.

We present the cost estimation and index selection errors with respect to training size in STATS dataset as an example in Figure 6. We truncate the cost error plot because E2E-Cost is unstable. We observe that most methods converge at about 20000 queries for cost estimation, and 3000 queries for index selection. Beyond this point, further training improvements were minimal. This shows that the most methods can be sufficiently trained with large amount of data.

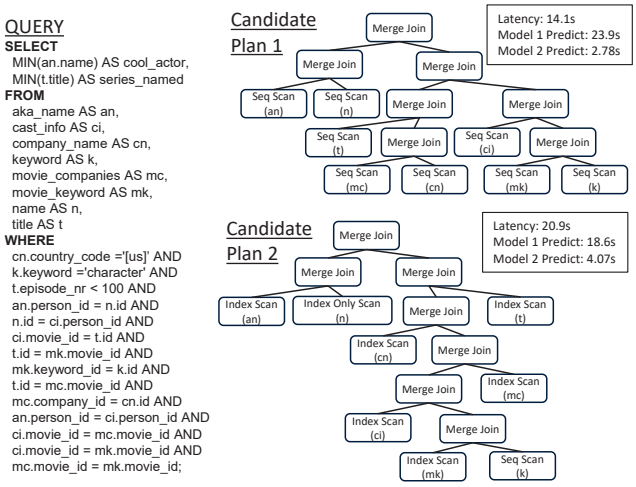


Figure 7: Case study of two query plans generated from different hint set. Model 1 (BAO) has higher accuracy than Model 2 (QueryFormer), but predicts the relative order wrongly.

6.3 Connections between Tasks

Cost estimation can be seen as a building block in downstream tasks like index selection and query optimization. Intuitively, a query plan representation method with higher accuracy in cost estimation is expected to have better downstream tasks performance. However, we show that this does not hold in previous experiments. To illustrate what happens, we provide an actual example extracted from the optimization experiment in Figure 7. For the given query, although the predictions from Model 1 have smaller Q-Errors (or other regression metrics like MSE or MAE) for both candidate query plans compared to Model 2, Model 1 chooses a sub-optimal query plan because it predicts the relative ranking wrongly. This shows that the accuracy across the entire domain in some downstream tasks is less important than in the decision margins. Because of this, a better cost estimator does not mean better downstream task performance.

Table 5: Experiment Summary for all representation methods and their components. ID indicates in-distribution workloads and OOD indicates out-of-distribution workloads.

Method/ Component	Cost (ID)	Cost (OOD)	Index (ID)	Index (OOD)	Optimizer
AVGDL	++	-	++	-	++
AIMEETSAI	++	-	+	-	-
ReJOIN	-	-	-	+	- -
BAO	++	++	++	+	+
NEO	-	+	+	+	-
Prestroid	-	+	+	-	+
E2E-Cost	++	+	-	-	- -
RTOS	++	-	+	+	+
Plan-Cost	+	-	-	-	-
QueryFormer	++	++	++	+	+
Pred	++	+	-	+	+
Est	++	+	++	++	++
Hist	-	-	+	+	++
Sample	+	-	+	-	-
TreeCNN	++	+	++	+	+
TreeLSTM	++	+	+	-	-
LSTM	++	+	+	++	++
Transformer	++	+	+	+	-

7 RELATED WORK

ML4DB research gains popularity in the last decade. Zhou et. al [47] and Zou et. al [48] survey comprehensively on approaches to enhance database systems with ML techniques. A growing body of evaluation work has emerged to analyze the behavior compare the effectiveness to specific database tasks. For instance, there are several works on cardinality estimation [14, 18, 37, 39, 42], focusing on method designs, error analysis, prediction interval analysis, practicality in deployment, and impacts on query optimizers, respectively. However, there is no evaluation work focusing on query plan representation, which is a building block of many ML4DB applications.

8 CONCLUSIONS AND FUTURE WORK

8.1 Summary and Key Takeaways

We score the performances of query plan representation methods, encoding features, and the tree models in different scenarios, and collate them in Table 5 for a more direct comparison. We summarize the key findings to answer the research questions as follows.

First, regarding **interchangeability**, this experimental study confirms that query plan representation methods can be applied to other tasks for which they are not originally designed, and some methods are able to achieve state of the art (SOTA) performance for the tasks that they are not designed for. In general, the performance of a method depends on task and dataset specifications.

To summarize the overall ranking of representation methods, BAO and QueryFormer have excellent performances across different task scenarios. We note that QueryFormer has high accuracy at the cost of efficiency. The second tier methods are AVGDL and RTOS, which perform well for in-distribution (ID) cases, but are less robust for out-of-distribution (OOD) cases. The rest of the methods are less competitive in comparison. We also find that the performance for cost estimation is not indicative of the performance on other tasks. The reason is that cost estimation focuses on absolute errors while other tasks are more focused on relative rankings among representation methods.

Second, regarding the optimal design of **tree model**, surprisingly, it is insignificant to cost estimation task. Different tree model designs matter in tasks like index selection and query optimization, where the relative rankings among query plans are more important. TreeCNN performs the best for ID cases, whereas LSTM is the most robust for OOD cases. Transformer has low efficiency (due to inference overhead), and TreeLSTM is the least effective.

Third, the optimal set of **feature encoding** depends on the task requirement. In cost estimation where the optimization target is the absolute error, concatenating as much candidate features as possible lead to higher or at least equal accuracy. However, for tasks where the relative ranking is more important, only using a selective subset of features (particularly Est and Hist) is more effective to predict better query plans for a query; adding other features usually damage the robustness of the learned model for such tasks.

Our findings on tree models and feature encoding will provide useful guidance for the future development of machine learning-based methods for database tasks. Among others, when we design models and features for tasks optimizing absolute error (such as cost estimation), and tasks optimizing relative ranking (such as index section), we would need different strategies.

8.2 Future Directions

This study evaluates the representation methods on a single dataset and task at a time. To improve their generalizability, we identify three research directions:

(1) **Generalizing datasets.** Current representation methods often latch onto dataset-specific patterns and require retraining when working across datasets. New approaches can be designed to account for transferable features, incorporate diverse training data, develop sampling techniques, and balance the trade-offs between individual dataset performance and broad adaptability.
(2) **Transferring across tasks.** Meta-learning techniques can be explored to extract task-agnostic knowledge, so that representation models learned remain relevant and effective across different database tasks [15, 33, 43].
(3) **Understanding tail performance.** A common challenge with representation methods is they have high tail errors, which are difficult to understand and debug for practicioners. Exploring the field of interpretable machine could provide potential solutions [21]. The integration of these methods with query plan representations is an interesting future direction.

ACKNOWLEDGMENTS

This research is supported in part by MOE Tier-2 grant MOE-T2EP20221-0015, and the Alibaba Talent Programme. We would also like to thank the reviewers for their constructive comments.

REFERENCES

[1] 2019. *Learning-based-cost-estimator.* https://github.com/greatji/Learning-based-cost-estimator [Last accessed: 2023-12-14].

[2] 2020. *Bao, a learned query optimizer. For PostgreSQL.* https://github.com/learnedsystems/BaoForPostgreSQL [Last accessed: 2023-12-14].

[3] 2020. *QPPNet.* https://github.com/rabbit721/QPPNet [Last accessed: 2023-12-14].

[4] 2021. *QueryFormer.* https://github.com/zhaoyue-ntu/QueryFormer [Last accessed: 2023-12-14].

[5] 2023. *Documentation PostgreSQL 12 71.1. Row Estimation Examples.* https://www.postgresql.org/docs/12/row-estimation-examples.html [Last accessed: 2023-12-14].

[6] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, Vol. 2000. 496–505.

[7] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *SIGACT-SIGMOD-SIGART*. 34–43.

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

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

[10] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1 (2010), 48–57.

[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 *ICDM*. 1241–1258.

[12] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[13] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1, 317–330.

[14] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765.

[15] Benjamin Hilprecht and Carsten Binnig. 2021. One model to rule them all: towards zero-shot learning for databases. *arXiv:2105.00642* (2021).

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[17] Johan Kok Zhi Kang, Gaurav, Sien Yi Tan, Feng Cheng, Shixuan Sun, and Bingsheng He. 2021. Efficient Deep Learning Pipelines for Accurate Cost Estimations Over Large Scale Query Workload. In *SIGMOD*. ACM, 1014–1022.

[18] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned cardinality estimation: An in-depth study. In *ICDM*. 1214–1227.

[19] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv:1412.6980*.

[20] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv:1809.00677* (2018).

[21] Pang Wei Koh and Percy Liang. 2017. Understanding black-box predictions via influence functions. In *ICML*. PMLR, 1885–1894.

[22] Donald Kossmann and Konrad Stocker. 2000. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Trans. Database Syst.* 25, 1 (2000), 43–82.

[23] 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.

[24] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[25] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proc. Int. Conf. Manag. Data 2021*. 1275–1288.

[26] 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.

[27] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM@SIGMOD 2018*. ACM, 3:1–3:4.

[28] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746.

[29] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR 2013, Workshop Track Proceedings*.

[30] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI, 2016*. AAAI Press, 1287–1293.

[31] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS.. In *VLDB*, Vol. 6. 1049–1058.

[32] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *ICML*. 1310–1318.

[33] Debjyoti Paul, Jie Cao, Feifei Li, and Vivek Srikumar. 2021. Database Workload Characterization with Query Plan Encoders. *Proc. VLDB Endow.* 15, 4 (2021), 923–935.

[34] Meikel Pöss and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Rec.* 29, 4 (2000), 64–71.

[35] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning. *Proc. VLDB Endow.* 14, 4 (2020), 471–484.

[36] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *PVLDB* 13, 3 (2019), 307–319.

[37] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.

[38] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proc. ACL 2015*. 1556–1566.

[39] Saravanan Thirumuruganathan, Suraj Shetiya, Nick Koudas, and Gautam Das. 2022. Prediction Intervals for Learned Cardinality Estimation: An Experimental Evaluation. In *ICDE 2022*. 3051–3064.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *arXiv:1706.03762* (2017).

[41] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv:1909.01315* (2019).

[42] 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.

[43] Ziniu Wu, Peilun Yang, Pei Yu, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2021. A Unified Transferable Model for ML-Enhanced DBMS. *arXiv:2105.02418* (2021).

[44] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1297–1308.

[45] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *ICDE 2020*. 1501–1512.

[46] 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.

[47] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2022. Database Meets Artificial Intelligence: A Survey. *IEEE Trans Knowl Data Eng* 34, 3 (2022), 1096–1116.

[48] Benyuan Zou, Jinguo You, Quankun Wang, Xinxian Wen, and Lianyin Jia. 2022. Survey on Learnable Databases: A Machine Learning Perspective. *Big Data Res.* 27 (2022).