Efficient Learning with Pseudo Labels for Query Cost Estimation

Shuncheng Liu
University of Electronic Science and Technology of China
Chengdu, China
liushuncheng@std.uestc.edu.cn

Xu Chen
University of Electronic Science and Technology of China
Chengdu, China
xuchen@std.uestc.edu.cn

Yan Zhao
Aalborg University
Aalborg, Denmark
yanz@cs.aau.dk

Jin Chen
University of Electronic Science and Technology of China
Chengdu, China
chenjin@std.uestc.edu.cn

Rui Zhou
Huawei Technologies Co., Ltd.
Chengdu, China
zhourui24@huawei.com

Kai Zheng*
University of Electronic Science and Technology of China
Chengdu, China
zhengkai@uestc.edu.cn

ABSTRACT

Query cost estimation, which is to estimate the query plan cost and query execution cost, is of utmost importance to query optimizers. Query plan cost estimation heavily relies on accurate cardinality estimation, and query execution cost estimation gives good hints on query latency, both of which are challenging in database management systems. Despite decades of research, existing studies either over-simplify the models only using histograms and polynomial calculation that leads to inaccurate estimates, or over-complicate them by using cumbersome neural networks with the requirements for large amounts of training data hence poor computational efficiency. Besides, most of the studies ignore the diversity of query plan structures. In this work, we propose a plan-based query cost estimation framework, called *Saturn*, which can eStimate cardinality and latency accurately and efficiently, for any query plan structures. *Saturn* first encodes each query plan tree into a compressed vector by using a traversal-based query plan autoencoder to cope with diverse plan structures. The compressed vectors can be leveraged to distinguish different query types, which is highly useful for downstream tasks. Then a pseudo label generator is designed to acquire all cardinality and latency labels with the execution part of the query plans in the training workload, which can significantly reduce the overhead of collecting the real cardinality and latency labels. Finally, a chain-wise transfer learning module is proposed to estimate the cardinality and latency of the query plan in a pipeline paradigm, which further enhances the efficiency. An extensive empirical study on benchmark data offers evidence that *Saturn* outperforms the state-of-the-art proposals in terms of accuracy, efficiency, and generalizability for query cost estimation.

*Corresponding author: Kai Zheng.

CIKM '22, October 17–21, 2022, Atlanta, GA, USA

ACM ISBN 978-1-4503-9236-5/22/10…$15.00
https://doi.org/10.1145/3511808.3557305

CCS CONCEPTS

• **Information systems** → **Data management systems**; **Query optimization**.

KEYWORDS

Query Plans; Cardinality Estimation; Latency Estimation

ACM Reference Format:
Shuncheng Liu, Xu Chen, Yan Zhao, Jin Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Learning with Pseudo Labels for Query Cost Estimation. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3511808.3557305

1 INTRODUCTION

Query optimizer is a vital component of modern Database Management Systems (DBMS), which aims to select an optimal query plan for a given SQL query. However, the recent studies [11, 12, 17, 33, 40] show that traditional query optimizers often select sub-optimal plans from the candidate query plans due to inaccurate cost estimation. Traditional cost models manually derive useful combinations of factors (e.g., cardinality, page fetch, and CPU) and polynomially calculate the expected costs [43] that may be proportional to the execution latency [36]. But the traditional histogram-based cardinality estimator simply builds a histogram on each attribute and assumes that all attributes are mutually independent [30]. Moreover, the cost model needs to be carefully tuned by Database Administrators (DBAs), leading to poor generalizability with the increasing complexity of the DBMS.

In this work, we divide the query cost estimation into the *query plan cost estimation* and the *query execution cost estimation*. Considering that the query plan cost heavily relies on accurate cardinality estimation [43], and the query execution cost is the execution latency of a query plan [19], we focus on the cardinality and latency estimation in this work.

Recently, the database community attempts to utilize learning-based models to estimate the cardinality and latency. For example, MSCN [10] adopts a CNN model to estimate the cardinality. Ortiz et al. [24] evaluate the effect of different deep learning architectures (e.g., Multi-layer Perceptron and RNN) on cardinality estimation. For the latency estimation, TPool [33] uses a tree-structured model to simultaneously estimate the cardinality and latency, and

QPPNet [19] introduces a plan-structured deep neural network to estimate the latency. However, they share some common limitations. Firstly, they can only estimate either cardinality or latency, or simply modify the output layer to estimate both cardinality and latency. The correlations between these two tasks are not well considered, which can affect the estimation accuracy especially when the training data is insufficient. Secondly, the training process of the aforementioned methods is usually time-consuming due to the large number of parameters and hyper-parameters to be tuned. Furthermore, they need to be fed with a huge amount of training data with the real cardinality or latency labels, which is quite expensive in terms of both time and economics as it requires the DBMS to actually execute each SQL query or query plan. Thirdly, it is hard to generalize those models to diverse query plan structures or scenarios. For example, TPool and QPPNet can only deal with binary-plan trees, and MSCN can only encode the SQL query, which is unavailable for query plans from the DBMS query optimizers.

To overcome these limitations, we face three main challenges: (**1**) variety of the query plan structures; (**2**) lack of methods that can significantly reduce the overhead for acquiring the real cardinality and latency labels; (**3**) lack of methods that can estimate both cardinality and latency accurately and efficiently. We aim to address the above challenges and propose a framework that fulfills accuracy, efficiency and generalizability for query cost estimation.

In this work, we propose a novel plan-based query cost estimation framework, called ***Saturn***, which can estimate the cardinality and latency accurately and efficiently, while easily adapted to any query plan structures. In particular, *Saturn* firstly encodes each query plan tree into a compressed vector by using a traversal-based query plan autoencoder that can deal with diverse query plan structures. The traversal methods (e.g., pre-order, level-order, and post-order traversal methods) convert each query plan tree into a sequence, and then a self-supervised autoencoder uses the sequence to generate a compressed vector. Using the compressed vector, the framework can not only obtain the latent features of a query plan, but also benefit a variety of downstream tasks (e.g., label generation and estimation). Secondly, a novel pseudo label generator is proposed to get all cardinality and latency labels with the execution part of the query plans in the training workload. Finally, we develop a chain-wise transfer learning module to estimate the cardinality and latency of query plans. The estimation chains, performed by transfer learning, can capture the latent correlations between cardinality and latency estimations, thus making estimates more accurate. Moreover, a lightweight network structure improves the training efficiency considerably.

To the best of our knowledge, this is first study to estimate both cardinality and latency by considering the latent correlations between tasks. In summary, we make the following contributions:

- We propose a traversal-based query plan autoencoder that can encode diverse query plans into fixed-size compressed vectors.
- We propose a pseudo label generator that can obtain all cardinality and latency labels in the training workload with minimal overhead.
- We develop a chain-wise transfer learning module to capture the latent correlations between cardinality and latency estimations.
- We conduct extensive experiments on two real datasets, i.e., TPC-H and IMDB, to demonstrate the superiority of our framework in terms of accuracy, efficiency, and generalizability.

2 PROBLEM DEFINITION

For a given SQL query q, the DBMS aims to find the optimal query plan with the lowest cost, from a group of candidate query plans by using its cost-based query optimizer. Specifically, the query optimizer first estimates the cardinality for each candidate query plan, and then calculates the cost value of the query plan by considering multiple factors, such as the cost of page fetch, CPU, tuple processing, and performing physical operations [36]. Finally, the query optimizer chooses the lowest-cost query plan as the execution query plan, which is expected to have a low execution latency.

Our purpose is to estimate the cardinality and latency of a query plan before its actual execution, which is crucial for generating high-quality query plans. Next, we formally define our estimation tasks, and analyze their potential characteristics in traditional query optimizers, based on which we summarize the key findings that inspire our work.

Query Plan. Given a SQL query q, the query optimizer can generate a query plan p. The query plan is naturally in a tree-based structure, and the number of child nodes is not fixed. Many DBMS expose this information through APIs, such as `EXPLAIN` [36], and thus we can get the query plan p conveniently. In the rest of the paper we use plan, query plan, and query plan tree interchangeably when no ambiguity is caused.

Query Cost Estimation. Given a query plan p, our problem is to achieve the following goals: (**1**) *Cardinality estimation goal:* estimate the number of tuples, $card(p)$, produced by the query plan p. (**2**) *Latency estimation goal:* estimate p's latency, $late(p)$, prior to execution.

In the following, we give two characteristics of the query cost estimation in traditional DBMS query optimizers.

(**1**) *Sequentiality.* Generally, the estimation pipeline of a DBMS query optimizer is: $card(p) \rightarrow cost(p)$, where $cost(p)$ denotes the query plan's cost estimates that is proportional to the execution latency in theory. Thereafter, the DBMS query optimizer can accordingly choose the best plan to execute.

(**2**) *High efficiency.* Given a query plan, the histogram-based cardinality estimator and the polynomial cost calculator can infer the cardinality and cost in time (generally less than $1ms$).

Inspirations. Although the cardinality and cost estimates of the DBMS query optimizer are biased, the sequentiality of the estimates reflects the latent correlations between cardinality and latency estimations, and the estimates can be quickly fed back to the DBMS. This leads to our key hypothesis: if we could, in a sequential manner, apply both cardinality and cost estimates of the DBMS query optimizer, would it be possible to estimate the cardinality and latency more accurately and efficiently? We answer this question affirmatively with a novel framework, called *Saturn*, which can perform cardinality and latency estimations by the following estimation chains:

$$\begin{aligned} card_D(p) &\rightarrow card(p) \\ card_D(p) &\rightarrow cost_D(p) \rightarrow late(p) \end{aligned} \tag{1}$$

where $card_D(p)$ denotes the estimated cardinality of the DBMS query optimizer, $card(p)$ denotes the real cardinality, $cost_D(p)$ denotes the estimated cost of the DBMS query optimizer, and $late(p)$ denotes the real latency. The cardinality and cost estimates generated by the DBMS query optimizer will guide our query cost estimation tasks.

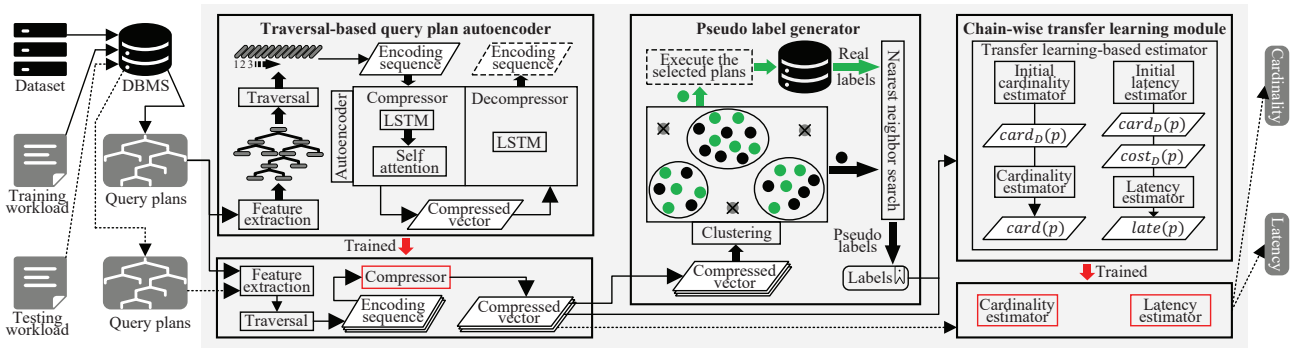

Figure 1: *Saturn* Framework Overview

3 FRAMEWORK OVERVIEW

Figure 1 shows the architecture of our framework *Saturn*, which consists of three components as follows:

Traversal-based Query Plan Autoencoder. The traversal-based query plan autoencoder is designed to encodes each query plan tree into a compressed vector, which can deal with diverse plan structures. Given a query plan tree, we first encode each node via feature extraction, and traverse the encoded query plan tree to form an encoding sequence. Secondly, we use an autoencoder equipped with a compressor and a decompressor to learn the latent features of each encoding sequence. After training the autoencoder, the compressor can be directly used to generate the compressed vectors of query plans for the downstream tasks (detailed in Section 4).

Pseudo Label Generator. The pseudo label generator is proposed to obtain all cardinality and latency labels by execution of a small number of query plans in the training workload, which can reduce the overhead of collecting the real cardinality and latency labels. We first uses DBSCAN to cluster the compressed vectors of query plans in the training workload, while deleting outliers automatically (marked by gray dots in Figure 1). Then we sample the query plans from each cluster to execute and obtain the real labels (marked in green). Afterwards we search for the most similar executed plans and applies their labels as the pseudo labels of the remaining query plans (which are not actually executed). Finally, all query plans in the training workload are labeled with both cardinality and latency, which can be used to train an estimator (detailed in Section 5).

Chain-wise Transfer Learning Module. The chain-wise transfer learning module is used to estimate the cardinality and latency of query plans, which can capture the latent correlations between cardinality and latency estimations. We use a transfer learning-based estimator to perform the estimation chains and make estimates. For cardinality estimation, an initial estimator first learns to estimate $card_D(p)$, and then learns to estimate $card(p)$ using fine-tuning strategies. For latency estimation, an initial estimator first learns to estimate $card_D(p)$, and then learns to estimate $cost_D(p)$, and finally learns to estimate $late(p)$ using fine-tuning strategies. After training the transfer learning-based estimator, the trained cardinality estimator estimates the cardinality and the trained latency estimator estimates the latency of query plans (detailed in Section 6).

4 QUERY PLAN AUTOENCODER

As an upstream component of *Saturn*, the traversal-based query plan autoencoder is used to encode each query plan tree into a fixed-size compressed vector. We first perform feature extraction on the query plan tree, then we traverse the tree to form an encoding sequence and input it to an autoencoder. The autoencoder will reduce the dimensionality of the encoding sequence and then try to restore it, thus forming an end-to-end training process. Among them, the traversal methods combined with LSTMs make the autoencoder general, the query plans with any structures and sizes can be compressed without changing the network. Furthermore, the autoencoder can automatically learn to retain important features while eliminating redundant features, and thus it can better extract the latent features of the query plan tree.

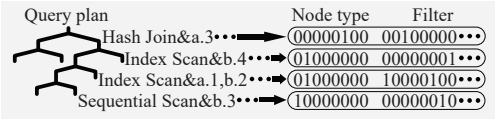

Figure 2: Example of Plan-based Feature Extraction

4.1 Plan-based Feature Extraction

Given a query plan tree p, we encode each node to a vector. Existing methods, for instance, use bottom-up encoding to encode the plan [18] or use selectivity to encode each predicate in the nodes [33]. However, they introduce some uncertainties in the encoding results: the features of the root node may not be the bit-wise sum of its child nodes' features, and the selectivities given by the query optimizers are inaccurate, especially in join operations. These uncertainties will spread throughout the entire encoding and affect the network performance. Our purpose is to encode the query plan tree without any uncertainty, helping the autoencoder obtain effective features. There are two main factors that can identify a node, including the node type and the filter. Next we discuss how to extract these features and encode them into vectors.

Node Type Encoding. For each node in a query plan, the query optimizer will specify its node type, and we use one-hot encoding to represent the node type. We mainly consider 8 node types, which can be divided into scan types and join types. The scan types include 'Sequential Scan', 'Index Scan', 'Bitmap Index Scan', 'Bitmap Heap Scan' and 'Index Only Scan', and the join types include 'Hash Join', 'Merge Join' and 'Nested Loop Join' [36].

Filter Encoding. In addition to the node types, the query optimizer will specify the filtered attributes in each node. We use one-hot encoding to represent these filtered attributes in each node.

Figure 2 shows an example of encoding node types and filters of a given plan. In this simple but effective way, a node in the query

plan tree is transformed into a vector of size, denoted as v, which is the sum of the number of node types and the number (A) of all the attributes in the dataset, i.e., $8 + A$. Our feature extraction is unbiased, where '1'*(s)* in the vector represents the existence of a certain node type or filtered attribute in the plan node, while '0'*(s)* represents the absence of them. This unbiased feature extraction allows the autoencoder to get 'pure' inputs which means that the inputs only consist of 1 and 0.

4.2 Plan Traversal Methods

A query plan tree generated by query optimizers, is an m-ary tree [38]. It is a rooted tree in which each node has no more than m children. Previous studies over-simplified the tree structure as a binary tree [19, 33] or a left-deep tree [18], so that they can use the structural neural network for representation learning. However, they cannot handle the m-ary query plan tree, leading to limited generalizability. In order to deal with the m-ary plan tree, we use the traversal method to serialize the node vectors that is convenient for training with LSTMs. For the m-ary tree, there are three traditional traversal methods, i.e., pre-order, level-order and post-order traversal [41].

Specifically, give a query plan tree p where each node is encoded into a vector v. Using a traversal method, we can convert these vectors into an encoding sequence $\widetilde{p} = \{v_1, v_2, \cdots, v_n\}$, where n denotes the number of nodes in the plan. Next we discuss three traversal methods, and analyze their availability.

(**1**) *Pre-order Traversal.* The root node is visited first, then we recursively traverse its leftmost subtree to the rightmost subtree.

(**2**) *Level-order Traversal.* We visit every node on a level (from the leftmost node to the rightmost node) before going to a lower level.

(**3**) *Post-order Traversal.* We recursively traverse the leftmost subtree to the rightmost subtree and finally visit the root node.

Analysis. We need to choose a traversal method that best matches the order of the query plan's execution. We find that the order in which the DBMS executes a query plan is most similar to the order generated by the post-order traversal. Therefore, we use the post-order traversal to traverse the query plan tree.

4.3 Autoencoder Architecture and Workflow

After traversing the query plan tree, we get a sequence of vectors (namely encoding sequence). Intuitively, the simplest approach is to use a recurrent neural network to learn latent features of the encoding sequence and then make estimates. However, different tasks need to train different deep learning models, leading to poor efficiency. Moreover, the sparse inputs will affect the convergence performance of the models, and even reduce the accuracy of the estimation. In summary, a representation model that can compress the encoding sequence into a dense vector, and be suitable for different tasks, is desired.

We propose an autoencoder equipped with a compressor and a decompressor, to learn the latent features of the query plan more wisely. The compressor is designed to reduce the dimensionality of the encoding sequence. Conversely, the decompressor needs to restore the compressed vector to the encoding sequence. The autoencoder can learn to retain typical features while eliminating redundant features in the encoding sequence. Furthermore, to recover different encoding sequences, the model should learn to distinguish different query types to avoid confusion. Therefore, using the compressed vectors, we can distinguish different types of queries, which can improve the performance of downstream tasks. Next, we will introduce the compressor and decompressor of the autoencoder, and then present the workflow.

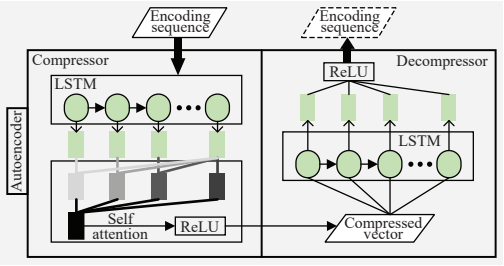

Figure 3: Autoencoder Architecture

Compressor. As shown on the left side in Figure 3, our compressor is composed of an LSTM and a self-attention mechanism. The LSTM learns the latent features of the encoding sequence and then uses the self-attention mechanism to enhance the memory ability of historical information while aggregating the features into a compressed vector.

To be specific, the encoding sequence $\widetilde{p} = \{v_1, v_2, \cdots, v_n\}$ is firstly fed into the LSTM that outputs the hidden state vector of each step as follows:

$$h_t = LSTM(v_t, h_{t-1}; W_l) \tag{2}$$

where $t \in \{1, 2, \cdots, n\}$, h_{t-1} is the hidden state vector of step $t-1$, and W_l denotes the parameters of LSTM.

Then the self-attention mechanism is designed for aggregating the hidden states along the sequential steps while different steps have different importance scores. Unlike the simple usage of LSTM's hidden state vectors, we introduce the self-attention mechanism [16, 39] to enhance the memory ability of the compressor, which can deal with the complicated query plans. The last hidden state vector h_n output by the LSTM, containing the information of all historical steps, will be used to calculate the importance of each step. For example, to get the importance score of a step in the encoding sequence, we calculate how much h_n pays attention to it. This attention represents the weight assigned to this step during the aggregation process. Following the standard procedure [31, 39], we can obtain a query vector q of the last hidden state vector, and a key matrix K of all hidden state vectors, as follows:

$$q = h_n \times W_q + b_q, K = H \times W_K + b_K \tag{3}$$

where W_q and W_K are the weights of two fully connected neural networks for h_n and H, respectively, b_q and b_K denote biases of W_q and W_K, respectively, and H refers to all hidden state vectors, i.e., $H = [h_1, h_2, ..., h_n]$. Using q and K, we can calculate the importance scores $\boldsymbol{s}$ of all steps using $Softmax(q \times K)$. Thereafter, we can aggregate all hidden state vectors of the LSTM into a vector h as follows:

$$h = \sum_{t \in \{1,2,\cdots,n\}} s_t \cdot h_t \tag{4}$$

where s_t ($s_t \in \boldsymbol{s}$) represents the importance score of step t.

Finally, the output of the compressor, i.e., compressed vector v_c, can be formulated as follows:

$$v_c = ReLU((h \times W_{c1} + b_{c1}) \times W_{c2} + b_{c2}) \tag{5}$$

where W_{c1} and W_{c2} are the weights of two fully connected neural networks, and b_{c1} and b_{c2} are biases of W_{c1} and W_{c2}, respectively.

Decompressor. After using the compressor to get the compressed vector v_c, we utilize a decompressor to restore it to the encoding sequence $\widetilde{p}$. As shown on the right side of Figure 3, we use an LSTM to recover the encoding sequence. Unlike the compressor, the input of each step of decompressor's LSTM is the compressed vector, thus the neural network can decode the compressed vector on demand.

To be specific, the input compressed vector v_c is firstly fed into the LSTM that outputs the hidden state vector of each step as follows:

$$h_t^{de} = LSTM(v_c, h_{t-1}^{de}; W_l^{de}) \tag{6}$$

where $t \in \{1, 2, \cdots, n\}$, h_{t-1}^{de} is the hidden state vector of step $t-1$, and W_l^{de} denotes the parameters of LSTM. This calculation will be executed n times to get the matrix $H^{de} = [h_1^{de}, h_2^{de}, \cdots, h_n^{de}]$.

Finally, the output of the decompressor is obtained by using non-linear activation as follows:

$$\widetilde{p}^{de} = ReLU((H^{de} \times W_{d1} + b_{d1}) \times W_{d2} + b_{d2}) \tag{7}$$

where $\widetilde{p}^{de}$ denotes the decompressed encoding sequence with length n, W_{d1} and W_{d2} are the weights of two fully connected neural networks, and b_{d1} and b_{d2} denote biases of W_{d1} and W_{d2}, respectively.

Workflow. In the offline training phase, the autoencoder is trained in an end-to-end manner. Benefiting from the well-designed network, different length of encoding sequences can be compressed and decompressed effectively. Our autoencoder needs to minimize the loss function as follows:

$$Loss = \frac{1}{n}\sum_{i=1}^{n}(\widetilde{p}_i - \widetilde{p}_i^{de})^2 \tag{8}$$

where $\widetilde{p}_i$ denotes i-th vector in the encoding sequence $\widetilde{p}$, and $\widetilde{p}_i^{de}$ denotes i-th vector in the decompressed encoding sequence $\widetilde{p}^{de}$.

Thereafter, in the online phase, the decompressor is put aside, and we use the trained compressor in the autoencoder to compress the encoding sequence (obtained by using the feature extractor and the traversal method in turn) into a compressed vector. We note that the online phase of the autoencoder can be used in both training workload and testing workload. For the training workload, the compressed vectors of the training plans are the inputs of the pseudo label generator and the chain-wise transfer learning module in the offline training phase. For the testing workload, the compressed vectors of the testing plans are the inputs of the chain-wise transfer learning module in the online inference phase.

5 PSEUDO LABEL GENERATOR

In order to estimate cardinality and latency, we need to get the ground truths of all query plans in the training workload, in the context of supervised learning. Intuitively, after using the autoencoder to get compressed vectors, we can learn the mapping relationship between the latent features and real labels. Existing studies [10, 19, 24, 33] use exposed APIs, e.g., `EXPLAIN ANALYZE` [36] and `pg_hint_plan` [5], to get the real cardinality and latency labels of SQL queries or query plans, so as to train their deep learning models. However, it is time-consuming for `DBMS` to execute a large number of SQL queries or query plans. In many production deployments, the large time cost makes the model unable to be updated in time, affecting the performance of the system. To reduce the overhead of getting cardinality and latency labels, we propose a pseudo label generator, which can obtain all labels of the training plans with minimal overhead.

In general, we use $k\%$ (<1) of query plans with real labels to label the other plans, and the `DBMS` only needs to execute the $k\%$ of query plans in the training workload. There are two goals during the generation process. The first one is to generate labels as accurate as possible. The second one is to ensure the diversity of the pseudo labels and real labels, that is, in different types of query plans, the proportions of real labels and pseudo labels are consistent. The first goal is to make the deep learning model learn accurate knowledge, while the second goal is to make the model find different knowledge and learn to distinguish them. For example, if the query plans with `2 Joins` have the real labels in the training workload, the model can only learn this limited knowledge, leading to severe underfitting. What we expect is that different types of query plans have the same proportions of real labels and pseudo labels, so as to improve the generalizability of the model.

To achieve the goals, our generator first groups the query plans based on a clustering method. Then we randomly sample $k\%$ of the plans from each cluster, and they will be executed to obtain real labels. In each cluster, the remaining $1-k\%$ of plans use a nearest neighbor algorithm to search for their nearest executed plans (i.e., the most similar plan with real label) and use the real labels as the pseudo labels. To this end, all of the plans in the training workload are labeled with both cardinality and latency. Next, we will introduce the process of clustering and nearest neighbor search of our pseudo label generator.

Clustering. In order to automatically distinguish different types of query plans, we use DBSCAN [29] to cluster the query plans. However, simply clustering the one-hot encoding of the plan will fail, because the input vectors are sparse and the high-dimensional vectors will suffer from the curse of dimensionality. Benefiting from the autoencoder, the compressed vectors are able to effectively represent the query plans in a low-dimensional space. Moreover, the compressed vectors are distinguishable due to the restorability of the autoencoder. Thus, we use the compressed vectors as the input of DBSCAN, to adaptively group the query plans while finding some outliers.

To be specific, for all query plans in the training workload, we first use the trained compressor to obtain the compressed vectors. Then we perform DBSCAN clustering for the compressed vectors. Finally, we can obtain some clusters and outliers, and the outliers will be treated as the noises. We delete the outliers and use the valid clusters to perform the search process.

Nearest Neighbor Search. After clustering all query plans, in each cluster, we randomly select $k\%$ of the plans to execute, so as to obtain real cardinality and latency. Thereafter, for each unlabeled query plan, we should search for its nearest-labeled neighbor that has the minimum distance (e.g., euclidean distance) with the unlabeled query plan, and use the real label as the pseudo label. Intuitively, we can search for the nearest neighbor from all labeled query plans by performing the 1-Nearest Neighbour (1-NN) algorithm. However, when the search space is large, the 1-NN algorithm will be time-consuming. To reduce the search space, we can only

search for the nearest neighbor from each cluster. Each cluster represents a type of query plan, so the nearest neighbor of a plan must exist in its corresponding cluster.

Specifically, for an unlabeled query plan, we first perform 1-NN to search for its nearest-labeled neighbor in the cluster. Then we use the real label of the neighbor to be the pseudo label of the unlabeled query plan. Through this process, the remaining $1 - k\%$ of unlabeled plans in each cluster will be labeled without executing.

6 CHAIN-WISE TRANSFER LEARNING

After obtaining the trained autoencoder and the (real and pseudo) labels of query plans in the training workload, we can build the neural network to learn the mapping between the compressed vector of the plan and its labels (i.e., cardinality and latency labels). Existing studies use multi-layer perceptron (MLP) to learn the mapping. For example, TPool [33] uses a 4-layer fully connected neural network to simultaneously estimate the latency and cardinality. MSCN [10] uses a 4-layer perceptron to estimate the cardinality, and QPPNet [19] uses a 6-layer fully-connected neural network to estimate the latency. However, they have some common flaws as follows. Firstly, the training process of the aforementioned networks is usually time-consuming due to a large number of parameters and hyperparameters. Their networks are composed of representation layers and estimation layers, which are trained together. Furthermore, they ignore the correlations between cardinality and latency estimations. TPool [33] uses a 2-unit output layer to perform the multi-task learning, and MSCN [10] and QPPNet [19] can only estimate cardinality or latency. As we analyze in Section 2, there exists some latent correlations between cardinality and latency estimations. It is difficult for the deep learning model to directly learn to estimate the cardinality or latency. Also, the aforementioned networks may suffer from low convergence efficiency or sub-optimal solution. In summary, an estimation module that is accurate and efficient is desired.

We propose a chain-wise transfer learning module to estimate cardinality and latency more accurately and efficiently. We use the cardinality and cost estimates of the DBMS query optimizer to guide our estimation, and borrow the idea from inductive transfer learning [25, 49] to perform the estimation chains in Equation (1). Next, we will introduce the estimation chains, the transfer learning-based estimator, and the workflow of the module.

Estimation Chains. We propose the estimation chains that have two main ideas. First, we utilize the cardinality and the cost values estimated by DBMS query optimizer as the prior knowledge. Although these estimates deviate from the real cardinality and latency, they give good hints on our goals and can be easily obtained. Secondly, we design two estimation chains, which first imitate DBMS query optimizer and then surpass it. The prior knowledge can guide our estimator to imitate DBMS query optimizer, then the real cardinality and latency enable our estimator to surpass DBMS query optimizer. As shown on the left side in Figure 4, to estimate the cardinality of the query plan, we first learn to estimate the cardinality obtained by DBMS query optimizer, and then we learn to estimate the real cardinality. Similarly, to estimate the latency, we first estimate the cardinality generated by DBMS query optimizer, and then we estimate the cost obtained by DBMS. Finally, we learn to estimate the real latency. Since the first step of the two chains are to estimate $card_D(p)$, we can merge them into one step.

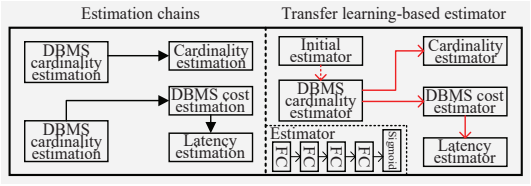

Figure 4: Estimation Chains and Transfer Learning-based Estimator

For the inputs of all steps on the chains, we find that all the estimates are strongly related to the query plan. Therefore, we unify the inputs of all steps using the compressed vectors generated by the trained autoencoder, which are low dimensional and informative. Specifically, given a query plan p, we use the autoencoder to obtain its compressed vector v_c and input v_c to all the steps on the estimation chains. For the estimation labels of all steps on the chains, $card_D(p)$ and $cost_D(p)$ can be quickly obtained by using APIs (e.g., `EXPLAIN` [36]), while $card(p)$ and $late(p)$ can be obtained via the pseudo label generator. Therefore, we can effectively obtain the training labels of all steps on the chains.

Transfer Learning-based Estimator. To perform the estimation chains, we utilize inductive transfer learning [25], which can improve the learning of the target task using the learned knowledge from the source task. As shown on the right side in Figure 4, the red solid arrows point from the source task estimator to the target task estimators, and the dashed red arrow represents the training process of the initial network, which follows the traditional supervised learning. To be specific, we use a 4-layer Fully Connected (FC) neural network with a Sigmoid activator as an estimator (i.e., using three fully connected layers and one fully connected layer with the Sigmoid activation to estimate the normalized results). The initial estimator will be trained to be the DBMS cardinality estimator to estimate $card_D(p)$, then the DBMS cardinality estimator will be copied to accomplish two estimation tasks including cardinality and latency estimation. For cardinality estimation, our proposed cardinality estimator aims to estimate $card(p)$ combining the DBMS cardinality estimator and fine-tuning strategies. For latency estimation, the DBMS cardinality estimator is first fine-tuned to estimate $cost_D(p)$, being DBMS cost estimator, and then we fine tune the DBMS cost estimator to generate a latency estimator for estimating $late(p)$.

We mainly consider two fine-tuning strategies [15, 49]. The first one is a semi-fixed strategy, which fixes the first two layers of the estimator, and only updates the parameters of the last two layers. The second one is an unfixed strategy, which updates all the parameters of the estimator. The latter performs better because the network can get closer to the global optimal solution.

Workflow. For offline training, the input vectors are generated by the trained autoencoder which can encode training plans into compressed vectors. Then the compressed vectors are fed into the estimator and perform the estimation chains. In each estimation step on the chains, the estimator needs to minimize the loss functions as follows: $LOSS_a = \frac{1}{N}\sum_{p\in\mathbb{P}_T} Q(card_D(p), \widehat{card_a(p)})$, $LOSS_b = \frac{1}{N}\sum_{p\in\mathbb{P}_T} Q(card(p), \widehat{card_b(p)})$, $LOSS_c = \frac{1}{N}\sum_{p\in\mathbb{P}_T} Q(cost_D(p), \widehat{cost(p)})$, and $LOSS_d = \frac{1}{N}\sum_{p\in\mathbb{P}_T} Q(late(p), \widehat{late(p)})$[1], where $LOSS_a$, $LOSS_b$, $LOSS_c$, and

[1] $Q(variable_1, variable_2) = \frac{max(variable_1, variable_2)}{min(variable_1, variable_2)}$

$LOSS_d$ denote different losses of DBMS cardinality estimator, cardinality estimator, DBMS cost estimator, and latency estimator, respectively, $\mathbb{P}_T$ denotes a set of query plans in the training workload, and $N = |\mathbb{P}_T|$. The labels of $card_D(p)$ and $cost_D(p)$ can be obtained by DBMS query optimizers, while the labels of $card(p)$ and $late(p)$ can be obtained by our pseudo label generator. Next, $\widehat{card_a(p)}$, $\widehat{card_b(p)}$, $\widehat{cost(p)}$ and $\widehat{late(p)}$ are the estimated cardinality by the DBMS cardinality estimator, the estimated cardinality by the cardinality estimator, the estimated cost and the estimated latency of the query plan p, respectively. Further, Q represents the q-error function [21], which is a relative factor between an estimate and the ground truth.

For online estimation, when the query optimizer requires cardinality or latency of a query plan, the trained autoencoder will encode the query plan to a compressed vector. We input the compressed vector into the trained cardinality estimator to estimate cardinality or into the trained latency estimator to estimate latency.

7 EXPERIMENT

7.1 Experimental Settings

Datasets and Workloads. We conduct experiments using two datasets: (1) TPC-H, a decision support benchmark with 8 relations [27], where the scale factor is set to 10 (i.e., the size of TPC-H is 10GB). (2) IMDB, a real-world dataset that contains a wide variety of information about actors, movies, etc. IMDB has 21 relations, based on the 3.6GB snapshot [12].

For query workloads, we simulate the scenario with insufficient training queries, which is similar to the real-world engineering. In many production deployments, preparing the query in advance is time-consuming. In general, it is not possible to prepare enough queries (more than 10K), so it is necessary to simulate the environment in which the training workload is insufficient.

For TPC-H, we use the standard query generator [26] to generate 2200 queries with 1–8 joins from 22 TPC-H query templates. We randomly take 1800 queries as the training workload, 200 queries as the validation workload, and 200 queries as the testing workload.

For IMDB, we use the provided query generator [33] to generate 2000 queries with 4–12 joins, and we randomly select 1800 queries as the training workload and 200 queries as the validation workload. We use two widely-used workloads for testing: the first one is the JOB-light [10] workload with 70 queries (1–4 joins), and the second one is the JOB [11] workload with 113 queries (3–16 joins). The 70 JOB-light queries and 113 JOB queries are taken as the testing workloads for the IMDB dataset.

Implementation Details. For the traversal-based query plan autoencoder, we traverse the query plan tree using the post-order traversal, which outperforms other traversal methods in terms of the end-to-end estimation accuracy. The LSTMs in the compressor and decompressor have 128 units. The two fully connected layers for calculating the q vector and the K matrix are comprised of 64 units. The two fully connected layers for obtaining the compressed vector are comprised of 64 units (for W_{c1}) and 32 units (for W_{c2}), respectively. Thus, the compressed vector has a fixed dimension of 32. The two fully connected layers for restoring the encoding sequence are comprised of 100 units for W_{d1} and 71 (or 116) units for W_{d2}, respectively. W_{d2} of 71 units is used for TPC-H and W_{d2} of 116 units is used for IMDB since the dimensionalities of node vectors in TPC-H and IMDB are 71 and 116, respectively.

For the pseudo label generator, we try several clustering methods to cluster the compressed vectors, including k-means++ [1], Mean shift[4], and DBSCAN [29]). We use DBSCAN for clustering because its effect is stable in this work. The DBSCAN clustering has two key hyperparameters, i.e., ϵ (set to 0.5 and 2 in TPC-H and IMDB, respectively), and *Min samples* (set to 6 in both datasets). These two hyperparameters are selected via grid search. The proportion parameter k is set to 50 (i.e., we only execute 50% of query plans in the training workload to get the real labels) [32], which can sufficiently evaluate the pseudo label generator.

For the chain-wise transfer learning module, all the estimators share the same structure, i.e., a 4-layer fully connected neural network with Sigmoid activator. Specifically, the number of neurons from the first layer to the fourth layer is set to 128, 64, 32, and 1, respectively, and the Sigmoid activator is set in the fourth layer to output the normalized estimation. In addition, we use the unfixed strategy (i.e., updating all the parameters of the estimator) as the default fine-tuning strategy, which outperforms the semi-fixed strategy in terms of the end-to-end estimation accuracy.

Finally, for the offline training phase of the autoencoder and the transfer learning-based estimator, we use the Adam optimizer [8] for updating the parameters with a scheduled learning rate of 0.0001. The bath size of training the autoencoder is set to 1, because the length of the input encoding sequence is not fixed [16]. Although the batch size of the autoencoder is small, the training process is efficient because the model is lightweight and the size of the training workload is small. While the bath size of the transfer learning-based estimator is set to 32, due to the fixed-size inputs (i.e., compressed vectors). The above hyperparameters (except the batch size of autoencoder) are tuned on the validation workloads by using the grid search. In addition, we use Early Stopping [2] to avoid the overfitting of the neural networks. Our experimental results are reported based on the above parameter settings, unless expressly specified.

Competitors and Variants. We compare *Saturn* with the following baselines:

(**1**) PostgreSQL 12.11 [36]: an open source DBMS which uses a histogram-based cardinality estimator and a manual cost calculator to estimate the cardinality and cost values, respectively, for a given plan. We can obtain the estimates of cardinality and cost value of a plan through the EXPLAIN command.

(**2**) MSCN: a query-driven cardinality estimation algorithm [10]. To extend MSCN to estimate the latency, we use the same network structure to train an additional latency estimator, based on the open source implementation [9].

(**3**) QPPNet: a plan-based latency estimation algorithm [19]. To extend QPPNet to estimate the cardinality, we use the same network structure to train an additional cardinality estimator, based on the open source implementation [20]. Since the network can only adapt to binary-plan trees, we filter out the input plans that are not binary.

(**4**) TPool: a state-of-the-art plan-based estimator which can estimate cardinality and latency simultaneously [33]. We reference the open source implementation [34] to reproduce the estimator. Since the network can only adapt to binary-plan trees, we filter out the input plans that are not binary.

Table 1: q-error Distribution of Baselines and Ours (*Saturn*)

Tasks			Cardinality estimation						Latency estimation					
Datasets	**Workloads**	**Methods**	**Median**	**90th**	**95th**	**99th**	**Max**	**Mean**	**Median**	**90th**	**95th**	**99th**	**Max**	**Mean**
TPC-H	TPC-H (Test)	PostgreSQL	2.35	42.2	257	478	1542	96.4	-	-	-	-	-	-
		MSCN	2.52	14.4	26.1	57.2	102	18.7	6.38	25.7	54.3	97.3	164	7.20
		QPPNet	2.13	7.46	18.5	30.4	87.3	8.46	**1.18**	1.34	1.66	2.05	8.89	**1.47**
		TPool	2.25	33.2	42.6	84.9	108	13.5	1.73	8.56	23.3	36.5	87.9	4.63
		Saturn	**1.26**	**3.17**	**15.4**	**22.5**	**26.7**	**1.80**	1.21	**1.33**	**1.64**	**1.97**	**7.63**	1.52
IMDB	JOB-light	PostgreSQL	5.79	60.4	857	2536	3380	163	-	-	-	-	-	-
		MSCN	3.37	67.4	345	884	963	54.8	16.4	68.9	115	139	314	27.2
		QPPNet	4.65	34.2	97.8	184	358	26.7	2.80	13.5	79.8	146	385	7.48
		TPool	3.17	39.7	129	216	263	20.4	1.74	9.7	20.2	93.5	124	10.6
		Saturn	**1.48**	**2.33**	**4.75**	**8.32**	**14.6**	**1.73**	**1.29**	**2.73**	**5.64**	**9.95**	**22.4**	**1.65**
	JOB	PostgreSQL	182	6529	24381	85649	423854	8957	-	-	-	-	-	-
		MSCN	13.8	76.8	645	1254	1736	86.0	39.0	136	228	295	554	36.4
		QPPNet	8.55	96.0	242	336	564	42.7	5.4	35.7	62.5	143	179	18.7
		TPool	9.40	70.8	183	626	748	46.8	4.95	32.7	56.2	138	166	18.3
		Saturn	**3.68**	**45.2**	**69.3**	**184**	**274**	**33.1**	**3.06**	**18.5**	**26.0**	**38.6**	**69.4**	**13.5**

We also compare *Saturn* with the following *Saturn*-Variants:
(**1**) *Saturn*-NoSA: our proposed framework without the self-attention mechanism in the autoencoder. We directly use the last hidden state vector h_n of LSTM in the compressor as the compressed vector.
(**2**) *Saturn*-NoDe: our proposed framework without the decompressor in the autoencoder. After deleting the decompressor, the autoencoder cannot be trained, so the pseudo label generator is unavailable. We simply combine the compressor and the estimator to train the framework in an end-to-end manner.
(**3**) *Saturn*-NoPLG: our proposed framework without the pseudo label generator. We randomly select 50% of query plans in the training workload to get the real labels and use them to train our estimator in the chain-wise transfer learning module.
(**4**) *Saturn*-NoChain: our proposed framework without the estimation chains performed by transfer learning. We directly train two initial estimators to estimate cardinality and latency, respectively.
Environment. We implement all algorithms in Python, and run the experiments on an Ubuntu Server with an Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, and NVIDIA GeForce RTX 3080 GPU. We use the `PostgreSQL` 12.11 [36] as the default `DBMS`, to obtain query plans, real labels of cardinality and latency [5], and estimates of cardinality and cost value, using its exposed APIs.

7.2 End-to-End Evaluation of *Saturn*

Estimation Accuracy. We adopt the widely-used q-error metric [10, 21, 33] and its optimal value is 1. We report the q-error distribution on the testing workloads in Table 1. The accuracy of different estimation algorithms can be ranked as: *Saturn* >TPool ≈ QPPNet >MSCN »`PostgreSQL`. The details are as follows:
(1) *Cardinality Estimation.* For all the testing workloads, *Saturn* achieves the highest accuracy for the cardinality estimation. On the TPC-H testing workload, *Saturn* outperforms TPool by 7×, QPPNet by 4×, MSCN by 10× and `PostgreSQL` by 53× on mean q-error. While our *Saturn* outperforms TPool by 4×, QPPNet by 3×, MSCN by 3×, and `PostgreSQL` by 57× on max q-error. On the JOB-light workload, *Saturn* outperforms TPool by 11×, QPPNet by 15×, MSCN by 31×, and `PostgreSQL` by 94× in terms of mean q-error. On the JOB workload, which is harder due to more joins, *Saturn* outperforms TPool by 29%, QPPNet by 22%, MSCN by 2×, and `PostgreSQL` by 270× on mean q-error. These results demonstrate that *Saturn* can estimate the cardinality accurately.

Table 2: Average Training and Inference Time of Baselines and Ours

Average training time (min)					
Workloads	PostgreSQL	MSCN	QPPNet	TPool	***Saturn***
TPC-H	**1.21**	10.6	20.4	23.5	4.74
IMDB	**0.88**	18.9	25.2	27.2	5.68

Average inference time (ms)					
Workloads	PostgreSQL	MSCN	QPPNet	TPool	***Saturn***
TPC-H (Test)	**0.13**	12.8	44.5	43.7	3.17
JOB-light	**0.14**	12.4	45.7	45.3	3.26
JOB	**0.16**	13.6	58.4	46.1	3.52

(2) *Latency Estimation.* For all the testing workloads, the latency estimation accuracy of *Saturn* is very high. On the TPC-H testing workload, *Saturn* outperforms TPool by 3× and MSCN by 4× on mean q-error. Besides, *Saturn* outperforms TPool by 11×, QPPNet by 14%, and MSCN by 21× on max q-error. The accuracy of QPPNet is comparable to *Saturn*, which is marginally better than *Saturn* on the TPC-H dataset. On the JOB-light workload, *Saturn* outperforms TPool by 6×, QPPNet by 4×, and MSCN by 16× on mean q-error. On the JOB workload, *Saturn* outperforms TPool by 26%, QPPNet by 27%, and MSCN by 2× on mean q-error. These results demonstrate the effectiveness of *Saturn* when estimating the latency.

It should be noted that *Saturn* only uses 50% of training plans with real labels, and 50% of training plans with the generated pseudo labels. Even under such a harsh setting, our model can achieve higher accuracy than other methods.
Training Efficiency. We record the average training time of all methods. For `PostgreSQL`, we record the latency of executing the `ANALYZE` [36] command on the TPC-H and IMDB datasets. For other methods, we record the average training time taken by both cardinality and latency estimation models to converge on the training workloads on TPC-H and IMDB. As shown in Table 2, the training time of `PostgreSQL` is the lowest since the histogram-based estimator does not require iterative gradient updates. *Saturn* is more efficient than other deep learning models in the training phase.
Inference Time. We also record the average inference time of all methods. As shown in Table 2, `PostgreSQL` runs the fastest, which requires around 0.1ms for each query plan. *Saturn* requires around 3.1 ~ 3.5ms for each query plan, which is much faster than other deep learning models. This is due to the fact that the number of training parameters of *Saturn* is smaller than others, resulting in high inference efficiency.
Generalizability Analysis. We count the types of plan structures that each method can handle. The more plan structures that can

Table 3: Generalizability Analysis of Baselines and Ours

Methods	Binary tree	Ternary tree	Quad tree
PostgreSQL	✓	✓	✓
MSCN	-	-	-
QPPNet	✓	-	-
TPool	✓	-	-
Saturn	✓	✓	✓

Table 4: Mean q-error of *Saturn* and *Saturn*-Variants

Mean q-error of cardinality estimation					
Workloads	*Saturn* -NoSA	*Saturn* -NoDe	*Saturn* -NoPLG	*Saturn* -NoChain	***Saturn***
TPC-H (Test)	2.32	3.46	3.74	2.87	**1.80**
JOB-light	2.54	3.51	4.92	3.49	**1.73**
JOB	38.6	42.6	55.3	41.8	**33.1**
Mean q-error of latency estimation					
Workloads	*Saturn* -NoSA	*Saturn* -NoDe	*Saturn* -NoPLG	*Saturn* -NoChain	***Saturn***
TPC-H (Test)	1.96	2.34	2.86	2.03	**1.52**
JOB-light	2.04	2.69	3.48	2.58	**1.65**
JOB	17.2	18.3	22.9	17.7	**13.5**

be processed by a method means that the generalizability of the method is higher. Table 3 reports their generalizability for query plans in the testing workloads. We can see that PostgreSQL and *Saturn* can deal with all structures of query plans, while TPool and QPPNet can only process binary trees. Since MSCN is a query-driven method, its generalizability is the poorest.

7.3 Ablation Study of *Saturn*

Effectiveness of Autoencoder. We compare *Saturn* against two variants, including *Saturn*-NoSA and *Saturn*-NoDe, and we report their mean q-error of cardinality and latency estimation in Table 4. We can see that *Saturn*-NoSA has decent estimation accuracy on TPC-H testing and JOB-light workloads. However, it performs poorly on JOB workload which contains complicated query plans with more joins. This clearly shows that the self-attention mechanism is helpful for long-range features memorization. In addition, *Saturn*-NoDe performs even worse than *Saturn*-NoSA on all testing workloads, since its autoencoder cannot be trained in advance.

Effectiveness of Pseudo Label Generator. We compare *Saturn* against *Saturn*-NoPLG, and report their mean q-error of cardinality and latency estimation in Table 4. We can see that *Saturn*-NoChain performs noticeably worse than *Saturn* since it is trained with fewer query plans and labels.

Effectiveness of Chain-wise Transfer Learning Module. We compare *Saturn* against *Saturn*-NoChain and report their mean q-error of cardinality and latency estimation in Table 4. We can see that *Saturn*-NoChain performs worse than *Saturn*. The main reason is that the estimation chains with the prior knowledge of DBMS query optimizer can guide the initial estimator to learn to estimate the cardinality or latency step by step, instead of trying to estimate the cardinality or latency directly.

8 RELATED WORK

Cardinality Estimation Methods. Traditional cardinality estimation methods can be divided into two categories [47], histogram [28] and sampling [22]. However, histogram-based methods can only handle the data distribution of each attribute, and sampling-based methods cannot work well for complicated SQL queries due to the 0-tuple problem [47]. Recently, the database community attempts to utilize deep learning models to estimate the cardinality. Existing deep learning models for cardinality estimation can be broadly classified into three categories, i.e., query-based, plan-based and data-based methods. The query-based approaches [10, 24] learn functions mapping a SQL query to its number of tuples. The state-of-the-art method [10] trains a Multi-Set Convolutional Network (MSCN) on queries, but this method is not suitable for query optimization, because that the query-based encoding is too tricky when optimizing on a tree structure. The plan-based approaches [3, 14, 18, 23, 33] learn functions mapping a query plan to its cardinality. A state-of-the-art method [33] builds an end-to-end estimator (i.e., TPool) based on a tree-structured model, which can estimate both cardinality and latency simultaneously. However, the plan-based representation layer can only deal with binary-plan trees, lacking scalability for other query plan structures (e.g., nested-query plans). The data-based approaches [44, 45, 48] learn unsupervised models of the joint probability density function of attributes in the datasets. The state-of-the-art method [48] proposes an unsupervised graphical model to learn the joint PDF of attributes in one table or multi tables, so as to obtain the selectivity of a given predicate. However, the data-based approaches can only estimate the cardinality, while the query-based and plan-based methods can be easily extended to deal with the latency estimation.

Latency Estimation Methods. Latency estimation is important for a wide variety of data management tasks, e.g., admission control [37] and resource management [35]. A number of studies leveraging statistical analysis to address the problem of latency estimation [6, 7, 13, 42, 43, 46]. However, they require human experts to analyze the properties of an operator or query plan and determine how they should be transformed into factors. Recently, the database community attempts to utilize deep learning models to estimate the latency. Existing deep learning models for latency estimation are mainly based on query plans. Sun et al. [33] propose a tree-structured model to learn the mapping between the query plan and execution latency. The state-of-the-art study [19] proposes a plan-structured neural network (QPPNet) that can estimate the latency from the bottom up. However, the existing networks can only estimate either of cardinality and latency, or simply modify the output layer to estimate both cardinality and latency.

9 CONCLUSION

We propose a plan-based query cost estimation framework, called *Saturn*, to estimate cardinality and latency accurately and efficiently, for any query plan structures. Overall, this study pioneers to estimate cardinality and latency by considering the latent correlations between estimation tasks, encode the query plan by using the autoencoder, and reduce the overhead of acquiring the training labels via the pseudo label generator. Extensive experiments offer insight into the effectiveness and efficiency of the proposed solutions.

ACKNOWLEDGMENTS

This work is partially supported by NSFC (No. 61972069, 61836007 and 61832017), and Shenzhen Municipal Science and Technology R&D Funding Basic Research Program (JCYJ20210324133607021).

REFERENCES

[1] David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding*. Technical Report. Stanford.

[2] Rich Caruana, Steve Lawrence, and Lee Giles. 2000. Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. In *NIPS*. 381–387.

[3] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-Based Representation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 97–107.

[4] Dorin Comaniciu and Peter Meer. 2002. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence* 24, 5 (2002), 603–619.

[5] NTT OSS Center DBMS Development and Support Team. 2022. Github repository: pg_hint_plan. https://github.com/ossc-db/pg_hint_plan

[6] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 337–348.

[7] Jennie Duggan, Olga Papaemmanouil, Ugur Cetintemel, and Eli Upfal. 2014. Contender: A resource modeling approach for concurrent query performance prediction.. In *EDBT*. 109–120.

[8] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv e-prints* (2014).

[9] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Github repository: Learned Cardinalities in PyTorch. https://github.com/andreaskipf/learnedcardinalities

[10] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.

[11] 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.

[12] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*.

[13] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries Using Statistical Techniques. *Proc. VLDB Endow.* 5, 11 (2012), 1555–1566.

[14] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation. *Proc. VLDB Endow.* (2021).

[15] Shuncheng Liu, Han Su, Yan Zhao, Kai Zeng, and Kai Zheng. 2021. Lane Change Scheduling for Autonomous Vehicle: A Prediction-and-Search Framework. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3343–3353.

[16] Shuncheng Liu, Zhi Xu, Huimin Ren, Tianfu He, Boyang Han, Jie Bao, Kai Zheng, and Yu Zheng. 2022. Detecting Loaded Trajectories for Hazardous Chemicals Transportation. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 3294–3306.

[17] Guy Lohman. 2014. Is query optimization a "solved" problem. In *Proc. Workshop on Database Query Optimization*, Vol. 13. 10.

[18] 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.

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

[20] Ryan Marcus and Olga Papaemmanouil. 2020. Github repository: QPPNet in PyTorch. https://github.com/rabbit721/QPPNet

[21] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proc. VLDB Endow.* 2, 1 (2009), 982–993.

[22] Magnus Müller, Guido Moerkotte, and Oliver Kolb. 2018. Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses. *Proc. VLDB Endow.* 11, 9 (2018), 1016–1028.

[23] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* (2021).

[24] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2019. An Empirical Analysis of Deep Learning for Cardinality Estimation. *arXiv e-prints* (2019).

[25] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *TKDE* 22, 10 (2010), 1345–1359.

[26] David Phillips. 2011. Github repository: TPC-H dbgen. https://github.com/electrum/tpch-dbgen

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

[28] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. *SIGMOD Rec.* 25, 2 (June 1996), 294–305.

[29] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017).

[30] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. 23–34.

[31] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. 464–468.

[32] Han Su, Shuncheng Liu, Bolong Zheng, Xiaofang Zhou, and Kai Zheng. 2020. A survey of trajectory distance measures and performance evaluation. *The VLDB Journal* 29, 1 (2020), 3–32.

[33] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-Based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.

[34] Ji Sun and Guoliang Li. 2021. Github repository: Learning-based-cost-estimator. https://github.com/greatji/Learning-based-cost-estimator

[35] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, and David DeWitt. 2016. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *SoCC*. 388–400.

[36] The PostgreSQL Global Development Group. 2022. PostgreSQL 12.11 Documentation. https://www.postgresql.org/docs/12/index.html.

[37] Sean Tozer, Tim Brecht, and Ashraf Aboulnaga. 2010. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *ICDE*. 397–408.

[38] Dominique Roelants van Baronaigien. 2000. A Loopless Gray-Code Algorithm for Listing k-ary Trees. *Journal of Algorithms* 35, 1 (2000), 100–107.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *NIPS*. 6000–6010.

[40] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation?. In *Proc. VLDB Endow.*

[41] Wikipedia contributors. 2021. Tree traversal — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Tree_traversal&oldid=1031403202.

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

[43] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*. 1081–1092.

[44] 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.

[45] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment* (2019), 279–292.

[46] Ning Zhang, Peter J Haas, Vanja Josifovski, Guy M Lohman, and Chun Zhang. 2005. Statistical learning techniques for costing XML queries. In *Proc. VLDB Endow.* 289–300.

[47] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and JI SUN. 2020. Database Meets Artificial Intelligence: A Survey. *TKDE* (2020).

[48] 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.

[49] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2021. A Comprehensive Survey on Transfer Learning. *Proc. IEEE* 109, 1 (2021), 43–76.