An End-to-End Learning-based Cost Estimator

Ji Sun Guoliang Li[*]
Department of Computer Science, Tsinghua University
sun-j16@mails.tsinghua.edu.cn,liguoliang@tsinghua.edu.cn

ABSTRACT

Cost and cardinality estimation is vital to query optimizer, which can guide the query plan selection. However traditional empirical cost and cardinality estimation techniques cannot provide high-quality estimation, because they may not effectively capture the correlation between multiple tables. Recently the database community shows that the learning-based cardinality estimation is better than the empirical methods. However, existing learning-based methods have several limitations. Firstly, they focus on estimating the cardinality, but cannot estimate the cost. Secondly, they are either too heavy or hard to represent complicated structures, e.g., complex predicates.

To address these challenges, we propose an effective end-to-end learning-based cost estimation framework based on a tree-structured model, which can estimate both cost and cardinality simultaneously. We propose effective feature extraction and encoding techniques, which consider both queries and physical operations in feature extraction. We embed these features into our tree-structured model. We propose an effective method to encode string values, which can improve the generalization ability for predicate matching. As it is prohibitively expensive to enumerate all string values, we design a patten-based method, which selects patterns to cover string values and utilizes the patterns to embed string values. We conducted experiments on real-world datasets and experimental results showed that our method outperformed baselines.

PVLDB Reference Format:
Ji Sun, Guoliang Li. An End-to-End Learning-based Cost Estimator. *PVLDB*, 13(3): 307-319, 2019.
DOI: https://doi.org/10.14778/3368289.3368296

1. INTRODUCTION

Query optimizer is a vital component of database systems, which aims to select an optimized query plan for a SQL query. However, recent studies show that the classical query optimizer [13, 14, 20] often generates sub-optimal plans due to poor cost and cardinality estimation. First, traditional empirical cost/cardinality estimation techniques cannot capture the correlation between multiple columns, especially for a large number of tables and columns. Second, the cost model requires to be tuned by DBAs.

Recently, the database community attempts to utilize machine learning models to improve cardinality estimation. MSCN [12] adopts the convolutional neural network to estimate the cardinality. However, this method has three limitations. Firstly, it can only estimate the cardinality, but cannot estimate the cost. Secondly, the deep neural network with average pooling is hard to represent complicated structures, e.g., complex predicates and tree-structured query plan, and thus the model is hard to be generalized to support most of SQL queries. Thirdly, although MSCN outperforms PostgreSQL in cardinality estimation, its performance can be further improved. For example, on the JOB-LIGHT workload, the mean error is over 50 and the max error is over 1,000. We can improve them to 24.9 and 289 respectively.

There are four challenges to design an effective learning-based cost estimator. First, it requires to design an end-to-end model to estimate both cost and cardinality. Second, the learning model should capture the tree-structured information of the query plan, e.g., estimating the cost of a plan based on its sub-plans. Third, it is rather hard to support predicates with string values, e.g., predicate "not LIKE '%(co-production)%'", if we don't know which values contain pattern '(co-production)'. As the string values are too sparse, it is rather hard to embed the string values into the model. Fourth, the model should have a strong generalization ability to support various of SQL queries.

To address these challenges, we propose an end-to-end learning-based cost estimation framework by using deep neural network. We design a tree-structured model that can learn the representation of each sub-plan effectively and can replace traditional cost estimator seamlessly. The tree-structured model can also represent complex predicates with both numeric values and string values.

In summary, we make the following contributions.
(1) We develop an effective end-to-end learning-based cost estimation framework based on a tree-structured model, which can estimate both cost and cardinality simultaneously. (see Section 3).
(2) We propose effective feature extraction and encoding techniques, which consider both queries and physical execution in feature extraction. We embed these features into

[*]Guoliang Li is the corresponding author. This work was supported by 973 Program (2015CB358700), NSF of China (61632016, 61521002, 61661166012), Huawei, and TAL.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368296

our tree-structured model, which can estimate the cost and cardinality utilizing the tree structure (see Section 4).

(3) For predicates with string values, we propose an effective method to encode string values for improving the generalization ability. As it is prohibitively expensive to enumerate all possible string values, we design a pattern-based method, which selects patterns to cover string values and utilizes the patterns to embed the string values (see Section 5).

(4) We conducted experiments on real-world datasets, and experimental results showed that our method outperformed existing approaches (see Section 7).

2. RELATED WORK

Traditional Cardinality Estimation. Traditional cardinality estimation techniques can be broadly classified into three classes. The first is *histogram-based* methods [11]. The core idea is to divide the cell values into equal depth or equal width buckets, keep the cardinality of each bucket, and estimate the cardinality according to the buckets. The method is easy to implement and has been widely used in commercialized databases. However, it is not effective to estimate the correlations between different columns. The second is *sketching*, which aims to solve distinct cardinality estimation problem, including FM [7], MinCount [1], LinearCount [28], LogLog [5], HyperLogLog [6]. The basic idea first maps the tuple values to bitmaps, then counts the continuous zeros or the number of hitting for each position, and finally infers the approximate number of distinct values. These methods can estimate the distinct number of rows for each dataset effectively. However, they are not suitable for estimating range query. The third is *sampling-based* methods [18, 26, 30, 14]. These methods utilize the data samples to estimate the cardinality. In order to address the sample vanishing problem (valid samples decrease rapidly for joins), [14] proposed index-based sampling. Sampling methods improve the accuracy of cardinality estimation, but they bring space overhead and only be adopted by in-memory database like HyPer [25]. Another limitation of this method is 0-tuple problem, i.e., when a query is sparse, if the bitmap equals to 0, the sample is invalid.

Traditional Cost Model. Traditional cost estimation is estimated by combining multiple factors like cost of sequential page fetch, cost of random page fetch, cost of CPU cost of processing a tuple and cost of performing operation. Firstly, these factors are highly correlated to the cardinality of data affected by the query. Secondly, the weight of each factor has to be tuned. There are some works focusing the cost model tuning [29, 19, 13], and [13] conducted experiments on the IMDB dataset to show that cardinality estimation is much more crucial than the cost model for cost estimation and query optimization.

Learning-based Cardinality Estimation. The database community starts to solve this problem by using learning-based method like statistic machine learning or deep neural network. The first learning based work on cardinality estimation [21] first classifies queries according to the query structure (join condition, attributes in predicates etc.), and then trains a model on the values of the predicates, but the model is ineffective to train on unknown structured query. The state-of-the-art method [12] trains a multi-set convolutional network on queries, but this method is not suitable for query optimization, because the query-based encoding is too tricky when optimizing on a tree structure, and the generalization is limited. [27] proposed a vision of training representation for the join tree with reinforcement learning. Yang et al [31] proposed deep likelihood models to capture the data distribution of multiple attributes, but it focused on cardinality estimation on single tables. Marcus et al [22] proposed an end-to-end learning-based optimizer, but their focus is not to estimate the cost and they utilize the cost to select a good query plan.

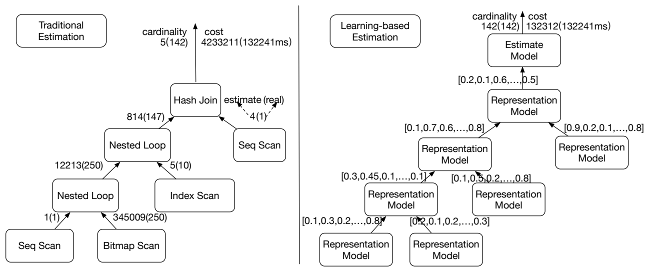


Figure 1: Comparison of Traditional Cost Estimation and Learning-based Cost Estimation.

Machine Learning for Database. Many machine learning techniques have recently proposed for optimizing databases [15], e.g., learned join order selection [32], knob tuning [16, 33], performance prediction [2, 29, 17, 8, 34], but they all require experts to select the features according to operation properties. A deep learning based approach [23] is also proposed.

3. OVERVIEW OF COST ESTIMATOR

Cost estimation is to estimate the execution cost of a query plan, and the estimated cost is used by the query optimizer to select physical plans with low cost. Cardinality estimation is to estimate the number of tuples in the result of a (sub)query. In this section we propose the system overview of cost and cardinality estimation.

Traditional databases estimate the cost and cardinality using statistics. For filter operations, cardinality estimator (e.g., PostgreSQL, DB2) estimates the cardinality using the histograms; for join operations, the cardinality is estimated by empirical functions with selectivity of joined tables (nodes) as variables. In Figure 1, the numbers on top of each node are estimated cardinality and real cardinality. We find that there exist large errors in traditional methods.

In general, we can effectively estimate the cardinality for leaf nodes (like Scan) by using the histogram; however, the error would be very large for joins because of the correlations between tables. Usually the more joins are, the larger error is. Unlike traditional cost estimation methods, our learning-based model can learn the correlation among multiple columns and tables, and the representation can retain accurate information on distribution of results even for the queries with dozens of operations.

Moreover, the query plan is a tree structure, and the plan is executed in a bottom-up manner. Intuitively, the cost/-cardinality of a plan should be estimated based on its sub-plans. To this end, we design a tree-structured model that matches the plan naturally, where each model can be composed of some sub-models in the same way as a plan is made up of sub-plans. We use the tree-structured model to estimate the cost/cardinality of a plan in a bottom-up manner.

Learning-based Cost Estimator. The end-to-end learning-based tree-structured cost estimator includes three

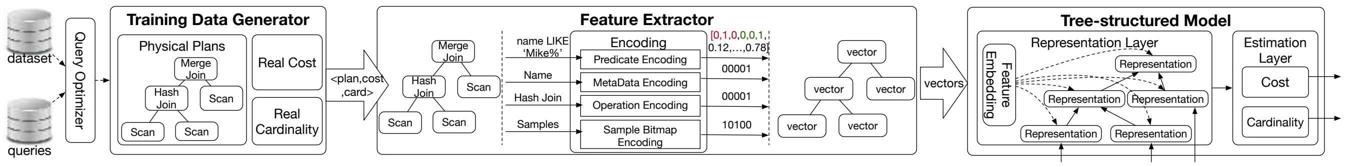
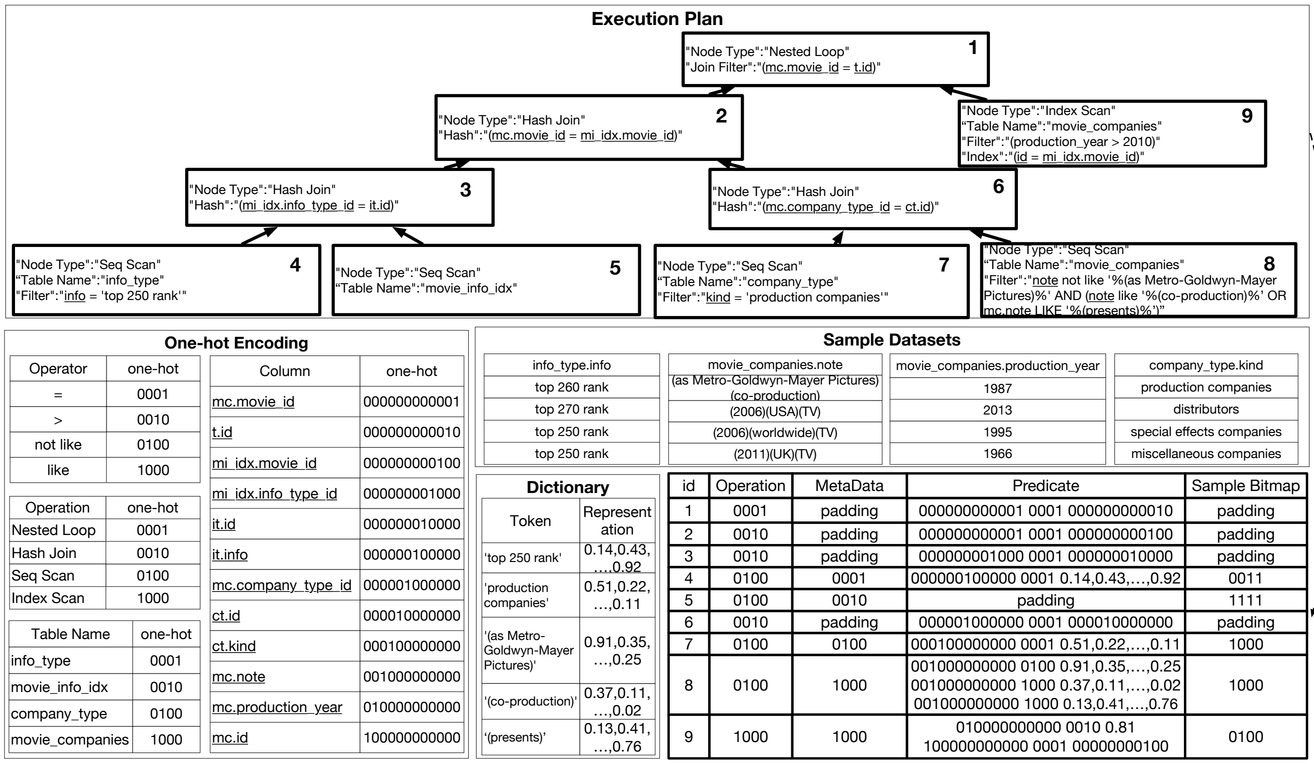


Figure 2: Architecture of learning-based cost estimator


Execution Plan
1
"Node Type":"Nested Loop"
"Join Filter":"(mc.movie_id = t.id)"
2
"Node Type":"Hash Join"
"Hash":"(mc.movie_id = mi_idx.movie_id)"
9
"Node Type":"Index Scan"
"Table Name":"movie_companies"
"Filter":"(production_year > 2010)"
"Index":"(id = mi_idx.movie_id)"
3
"Node Type":"Hash Join"
"Hash":"(mi_idx.info_type_id = it.id)"
6
"Node Type":"Hash Join"
"Hash":"(mc.company_type_id = ct.id)"
4
"Node Type":"Seq Scan"
"Table Name":"info_type"
"Filter":"info = 'top 250 rank'"
5
"Node Type":"Seq Scan"
"Table Name":"movie_info_idx"
7
"Node Type":"Seq Scan"
"Table Name":"company_type"
"Filter":"kind = 'production companies'"
8
"Node Type":"Seq Scan"
"Table Name":"movie_companies"
"Filter":"note not like '%(as Metro-Goldwyn-Mayer Pictures)%' AND (note like '%(co-production)%' OR mc.note LIKE '%(presents)%')"

One-hot Encoding

Operator	one-hot
=	0001
>	0010
not like	0100
like	1000

Operation	one-hot
Nested Loop	0001
Hash Join	0010
Seq Scan	0100
Index Scan	1000

Table Name	one-hot
info_type	0001
movie_info_idx	0010
company_type	0100
movie_companies	1000

Column	one-hot
mc.movie_id	000000000001
t.id	000000000010
mi_idx.movie_id	000000000100
mi_idx.info_type_id	000000001000
it.id	000000010000
it.info	000000100000
mc.company_type_id	000001000000
ct.id	000010000000
ct.kind	000100000000
mc.note	001000000000
mc.production_year	010000000000
mc.id	100000000000

Sample Datasets

info_type.info	movie_companies.note	movie_companies.production_year	company_type.kind
top 260 rank	(as Metro-Goldwyn-Mayer Pictures)(co-production)	1987	production companies
top 270 rank	(2006)(USA)(TV)	2013	distributors
top 250 rank	(2006)(worldwide)(TV)	1995	special effects companies
top 250 rank	(2011)(UK)(TV)	1966	miscellaneous companies

Dictionary

Token	Representation
'top 250 rank'	0.14,0.43,…,0.92
'production companies'	0.51,0.22,…,0.11
'(as Metro-Goldwyn-Mayer Pictures)'	0.91,0.35,…,0.25
'(co-production)'	0.37,0.11,…,0.02
'(presents)'	0.13,0.41,…,0.76

id	Operation	MetaData	Predicate	Sample Bitmap
1	0001	padding	000000000001 0001 000000000010	padding
2	0010	padding	000000000001 0001 000000000100	padding
3	0010	padding	000000001000 0001 000000010000	padding
4	0100	0001	000000100000 0001 0.14,0.43,…,0.92	0011
5	0100	0010	padding	1111
6	0010	padding	000001000000 0001 000010000000	padding
7	0100	0100	000100000000 0001 0.51,0.22,…,0.11	1000
8	0100	1000	001000000000 0100 0.91,0.35,…,0.25 001000000000 1000 0.37,0.11,…,0.02 001000000000 1000 0.13,0.41,…,0.76	1000
9	1000	1000	010000000000 0010 0.81 100000000000 0001 00000000100	0100

Figure 3: Running Example of query plan encoding (padding means filling up the corresponding blocks with zeros)

main components, including training data generator, feature extractor, and tree-structured model, as shown in Figure 2.

1) **Training Data Generator** generates training data as follows. It first generates some queries according to the potential join graph of the dataset and the predicates in the workload (see Section 4.3 for details). Then for each query, it extracts a physical plan by the optimizer and gets the real cost/cardinality. A training data is a triple ⟨a physical plan, the real cost of the plan, the real cardinality of the plan⟩.

2) **Feature Extractor** extracts useful features from the query plan, e.g., query operation and predicates. Each node in the query plan is encoded into feature vectors and each vector is organized into tensors. Then the tree-structured vectors are taken as input of the training model. For simple features, we can encode them by using one-hot vector or bitmap. While for complicated features, e.g., LIKE predicate, we encode each tuple ⟨column, operator, operand⟩ into vectors, by using a one-to-one mapping (see Section 4.1).

3) **Tree-structured Model** defines a tree-structured model which can learn representations for the (sub)plans, and the representations can be used in cost and cardinality estimation. The model is trained based on the training data, stores the updated parameters in the model, and estimates cost and cardinality for new query plans.

4) **Representation Memory Pool** stores the mapping from sub-plans to their representations when processing a query. For example, when the optimizer optimizes query A ⋈ B ⋈ C ⋈ D by using dynamic programming, it must know the cost of $\{A, B, C, D, A \bowtie B, B \bowtie C, C \bowtie D, A \bowtie (B \bowtie C), (A \bowtie B) \bowtie C, (B \bowtie C) \bowtie D, B \bowtie (C \bowtie D), \cdots\}$. When evaluating $A \bowtie (B \bowtie C)$, the representations of sub-plans $B \bowtie C$ and A can be extracted from the memory pool directly without re-calculating. Note that we only keep the mappings of the current query and the mappings will be removed when the query is processed.

Workflow. For offline training, the training data are generated by *Training Data Generator*, which are encoded into tensors by *Feature Extractor*. Then the training data is fed into the *Training Model* and the model updates weights by back-propagating based on current training loss. The details of model training is discussed in Section 4.3.

For online cost estimation, when the query optimizer asks the cost of a plan, *Feature Extractor* encodes it in a up-down manner recursively. If the sub-plan rooted at the current node has been evaluated before, it extracts representation from *Representation Memory Pool*, which stores a mapping from a query plan to its estimated cost. If the current sub-plan is new, *Feature Extractor* encodes the root and goes to its children nodes. We input the encoded plan vector into *Tree-structured Model*, and then the model evaluates the cost and cardinality of the plan and returns them to the query optimizer. Finally, the estimator puts all the representations of 'new' sub-plans into *Representation Memory Pool*.

Table 1: Main Plan Operations

Operation	Features
$Aggregate$	$[Operator, Name_{keys}]$
$Sort$	$[Operator, Name_{keys}]$
$Join$	$[Operator, Predicate_{join}]$
$Scan$	$[Operator, Name_{table}, Name_{index}, Predicate_{filter}, SampleBitmap]$

Figure 3 shows a running example of feature encoding for a plan extracted from the JOB workload. The plan is encoded as a vector using the one-hot encoding scheme, which considers both query and database samples. Then the vectors are taken as an input of the training model.

4. TREE-BASED LEARNING MODEL

In this section, we introduce a tree-structured deep neural network based solution for end-to-end cost estimation. We first introduce feature extraction in Section 4.1, and then discuss model design in Section 4.2. Finally we present the model training in Section 4.3.

4.1 Feature extraction and encoding

We first encode a query node as a node vector, and then transform the node vectors into a tree-structured vector.

There are four main factors that may affect the query cost, including the physical query operation, query predicate, meta data, and data. Next we discuss how to extract these features and encode them into vectors.

Operation is the physical operation used in the query node, including *Join* operation (e.g., Hash Join, Merge Join, Nested Loop Join); *Scan* operation (e.g., Sequential Scan, Bitmap Heap Scan, Index Scan, Bitmap Index Scan, Index Only Scan); *Sort* operation (e.g., Hash Sort, Merge Sort); *Aggregation* operation (e.g., Plain Aggregation, Hash Aggregation). These operations significantly affect the cost. Each operation can be encode as a one-hot vector. Figure 3 shows the one-hot vectors of different operations.

Predicate is the set of filter/join conditions used in a node. The predicate may contain join conditions like 'movie.movie_id = mi_idx.movie_id' or filter conditions like 'production_year $>$ 1988'. Besides the *atomic predicates* with only one condition, there may exist *compound predicates* with multiple conditions, like 'production_year $>$ 1988 AND production_year $<$ 1993'. The predicates affect the query cost, because the qualified tuples will change after applying the predicates.

Each atomic predicate is composed of three parts, *Column*, *Operator*, and *Operand*. *Operator* and *Column* can be encoded as one-hot vectors. For *Operand*, if it is a numeric value, we encode it by a normalized float; if it is string value, we encode it with a string representation (see Section 5 for details). Then the vector of an atomic predicate is the concatenation of the vectors for column, operator and operand. Table 2 shows the vector of each predicate.

For a compound predicate, we first generate a vector for each atomic predicate and then transfer multiple vectors into a vector using a one-to-one mapping strategy. There are multiple ways to transfer a tree structure to a sequence in a one-to-one mapping, and here we take the depth first search (DFS) as an example. We first transfer the nodes to a sequence using DFS, and then concatenate the vectors following the sequence order. Figure 4 shows an example of encoding a compound predicate. We transfer the nodes into a sequence in the visited order, where the solid lines with arrow represent forward search, and each dotted line represents one step backtracking. We append an empty node to the end of sequence for each dotted line. Thus, we can encode each distinct complex predicate tree as a unique vector and the compound predicate can be encoded as a tensor.

Table 2: Features of Condition Operators

Operators	Features
and/or/not	$[Operator]$
=/!=/>/</LIKE/IN	$[Operator, Column, Operand]$

MetaData is the set of columns, tables and indexes used in a query node. We use a one-hot vector for columns, tables, and indexes respectively. Then the meta node vector is a concatenation of column vectors, table vectors and index vectors. (Note that a node may contain multiple tables, columns and indexes, and we can compute the unions of different vectors using the OR semantic.) We encode both meta data and predicate, because some nodes may not contain predicates. Figure 3 shows an example.

Sample Bitmap is a fix-sized 0-1 vector where each bit denotes whether the corresponding tuple satisfies the predicate of the query node. If the data tuple matches the predicate, the corresponding bit is 1; 0 otherwise. This feature is only included in *single nodes* with predicates. As it is expensive to maintain a vector for all tuples, we select some samples for each table and maintain a vector for the samples. Figure 3 shows an example of encoding sample data.

After encoding each node in the plan, we need to encode the tree-structured plan into a vector using a one-to-one mapping strategy. We also adopt the DFS method in the same ways as encoding compound predicates.

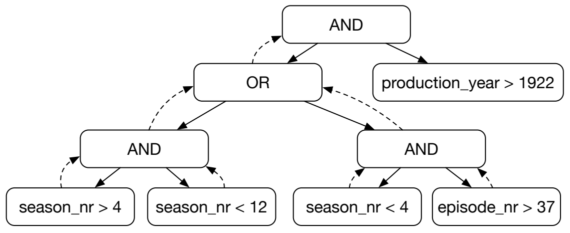
4.2 Model Design

Our model is composed of three layers, *embedding layer*, *representation layer* and *estimation layer* as shown in Figure 5. Firstly, feature vectors in each plan node are large and sparse, we should condense them and extract high-dimensional information of features, and thus the embedding layer embeds the vector for each plan node. Secondly, the *representation layer* employs a tree-structured model, where each node is a representation model and the tree structure is the same as the plan. Each representation model learns two vectors (global vector and local vector) for the corresponding sub-plan, where the global vector captures the information of the sub-plan rooted at the node and the local vector captures the information of the node. Node that each representation model learns the two vectors based on the vectors of its two children and the feature vector of the corresponding node. Finally, based on the vectors of the root node, *estimation layer* estimates the cost and cardinality.

4.2.1 Embedding Layer

The embedding layer embeds a sparse vector to a dense vector. As discussed in Section 4.1, there are 4 types of features, *Operation*, *Metadata*, *Predicate* and *Sample Bitmap*. *Operation* is a one-hot encoding vector, and *Metadata* and *Sample Bitmap* are bitmap vectors. We use a one-layer fully connected neural network with ReLU activator to embed these three vectors. However, the structure of the *Predicate* vector is complicated because it contains multiple AND/OR semantics, and we design an effective model to learn the representation of predicates.

Our goal is to estimate the number of tuples that satisfy a predicate. For an atomic predicate, we can directly use the



Node	Bool Operator	Predicate Column	Predicate Operator	Predicate Operand
episode_nr > 37	00	00010000	0010	0.0004
season_nr < 12	00	00001000	0001	0.0059
AND	01	padding	padding	padding
OR	10	padding	padding	padding

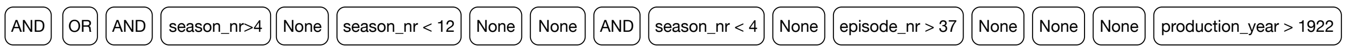
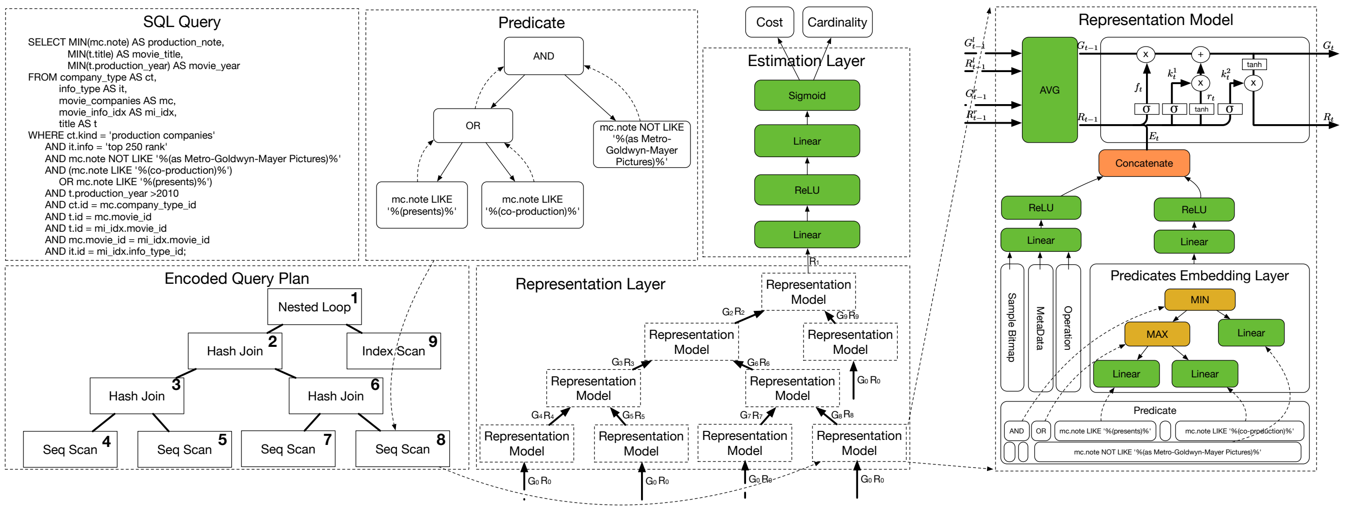


Figure 4: Predicates Encoding (padding means filling up the corresponding blocks with zeros)



```
SELECT MIN(mc.note) AS production_note,
         MIN(t.title) AS movie_title,
         MIN(t.production_year) AS movie_year
FROM company_type AS ct,
       info_type AS it,
       movie_companies AS mc,
       movie_info_idx AS mi_idx,
       title AS t
WHERE ct.kind = 'production companies'
    AND it.info = 'top 250 rank'
    AND mc.note NOT LIKE '%(as Metro-Goldwyn-Mayer Pictures)%'
    AND (mc.note LIKE '%(co-production)%')
       OR mc.note LIKE '%(presents)%')
    AND t.production_year >2010
    AND ct.id = mc.company_type_id
    AND t.id = mc.movie_id
    AND t.id = mi_idx.movie_id
    AND mc.movie_id = mi_idx.movie_id
    AND it.id = mi_idx.info_type_id;
```

Figure 5: Two Level Tree Model

vector. But for a compound predicate with multiple conditions, we need to learn the semantic of the predicates and the distribution of the results after applying the predicates on the dataset.

Consider a compound predicate with two atomic predicates using the AND semantic. We can estimate the number of results satisfying the predicate by the minimum number of estimated results satisfying the atomic predicates. Thus we use the *min* pooling layer to combine the two atomic predicates. Consider a compound predicate with two atomic predicates using the OR semantic. We can estimate the number of results satisfying the predicate by the maximum number of estimated results satisfying the atomic predicates. Thus we use the *max* pooling layer to combine the two atomic predicates.

In this way, we use a tree pooling to encode a predicate, where the tree structure is the same as the predicate tree structure. Particularly, the leaf node is a fully connected neural network, the OR semantic is replaced with the max pooling layer and the AND semantic is replaced with the min pooling layer. The advantages of this model are two folds. The first is that only the leaf nodes need to be trained so that it's easy to do efficient batch training. The second is that this model converges faster and performs better.

Figure 4 shows a compound predicate and its embedded model. For leaf nodes, we use a fully connected neural network. For conjunction nodes, we use max pooling layer for 'OR' and min pooling layer for 'AND' which meet the semantic of 'AND' and 'OR'.

Predicate Embedding Layer: LSTM vs Min-Max-Pooling. We can also use LSTM as the predicate embedding layer. However, LSTM cannot represent the semantic of predicates explicitly. We compare them in Section 7.2.

Embedding Formulation. We denote features *Operation*, *Metadata*, *Predicate* and *Sample Bitmap* of $node_t$ as O_t, M_t, P_t, B_t respectively, and we denote each node of feature *Predicate* as P_t with P_t^l as its left child and P_t^r right child. *Embedding Model* can be formalized as below. E is the embedding output. W is the weight of a fully connected neural network. b is a bias.

$$E = [embed(O_t), embed(M_t), embed(B_t), embed(P_t)]$$
$$embed(O_t) = ReLU(W_o \cdot O_t + b_o)$$
$$embed(M_t) = ReLU(W_m \cdot M_t + b_m)$$
$$embed(B_t) = ReLU(W_b \cdot B_t + b_b)$$
$$embed(P_t) = \begin{cases} min(embed(P_t^l), embed(P_t^r)) & P_t = and, \\ max(embed(P_t^l), embed(P_t^r)) & P_t = or, \\ W_p \cdot P_t + b_p & P_t = expr. \end{cases}$$

where $type(P_t)$ is the type of a node, which includes AND, OR, and a predicate expression.

4.2.2 *Representation Layer*

Cost estimation has two main challenges – information vanishing and space explosion. First, it is easy to estimate the cost for simple operations, e.g., estimating the cost of a filtering predicate on a single table, but it is rather hard to estimate the cost of joining multiple tables, because the join space is large and the joined tuples are sparse. In other words, in leaf nodes, we can capture much information for single table processing. But for upper nodes in the query plan, the correlation among nodes may be lost and this is the information vanishing problem. Second, to retain enough information to capture the correlations among tables, it requires to store much more space and intermediate results. But the space grows exponentially and becomes prohibitively expensive for a large number of tables. This is the space explosion problem.

Representation layer aims to address these two problems, by capturing the global cost information from leaf nodes to the root and avoiding information loss (e.g., correlation between columns). The representation layer trains representation for sub-plan recursively, instead of using data sketch to represent intermediate results of sub-plans. The representation layer uses the representation vector learned from features to represent results of sub-plans. As Figure 5 shows, all the units in this layer are neural networks in the same structure and share common parameters and we call these units as representation models. Each representation model has three inputs, the *embedding vector* E, the representation vector $[G^l_{t-1}, R^l_{t-1}]$ of its left child, and the representation vector $[G^r_{t-1}, R^r_{t-1}]$ of its right child. (Note that for leaf nodes, we use zero vectors as their children vectors.) The output is $[G_t, R_t]$.

The most important design issue in this layer is the choice of recurrent neural network. As a joint network of the tree-structured model, it decides which information to be passed over. A naive implementation is using fully connected neural network which takes as input the concatenate of local transformed features and its children's output. However, the lost information would never be utilized by upper nodes any more. Therefore, representation model with naive neural networks suffers from gradient vanishing and gradient explosion problems. Compared to deep neural network, LSTM cell can efficiently address these problems by using an extra information channel. The structure of LSTM cell is shown in Figure 5, where G_t is the channel for long memory, and f_t controls which information should be forgot in the long memory. k^1_t controls which information should be added into the long memory channel. k^2_t controls which information in the memory channel should be taken as the representation of the sub-plans. As G_t can be a path without any multiplication, LSTM avoids gradient vanishing. The forget gate of Sigmoid can help LSTM to address gradient explosion problem. The representation layer can be formalized as below:

$$\begin{aligned}
&x_t = E\\
&G_{t-1} = (G^l_{t-1} + G^r_{t-1})/2\\
&R_{t-1} = (R^r_{t-1} + R^r_{t-1})/2\\
&f_t = Sigmoid(W_f \cdot [R_{t-1}, x_t] + b_f)\\
&k^1_t = Sigmoid(W_{k_1} \cdot [R_{t-1}, x_t] + b_{k_1})\\
&r_t = tanh(W_r \cdot [R_{t-1}, x_t] + b_r)\\
&k^2_t = Sigmoid(W_{k_2} \cdot [R_{t-1}, x_t] + b_{k_2})\\
&G_t = f_t \times G_{t-1} + k^1_t \times r_t\\
&R_t = k^2_t \times tanh(G_t)
\end{aligned}$$

where E is the vector, W is a weight and b is a bias.

Representation Layer: LSTM vs Fully Connected Neural Network. We can use a two-layer fully connected neural network (TNN) as the representation layer. The network accepts the concatenation of children representations as input and outputs the representation for the current plan. However, TNN may suffer from gradient vanishing and explosion for complicated queries (see Section 7.2).

4.2.3 *Estimation Layer*

The *Estimation Layer* is a two-layer fully connected neural network, and the activator is a ReLU function and the output layer is a sigmoid function to predict the normalized cardinality and cost. The output layer should be able to evaluate cost or cardinality for any sub-plan by its representation vector. This layer takes the representation R_t of the upper model in *Representation Layer* as input, and can be formalized as below:

$$\begin{aligned}
cost' &= ReLU(W_{cost'} \cdot R_t + b_{cost'})\\
card' &= ReLU(W_{card'} \cdot R_t + b_{card'})\\
cost &= Sigmoid(W_{cost} \cdot cost' + b_{cost})\\
card &= Sigmoid(W_{card} \cdot card' + b_{card})
\end{aligned}$$

As we need to estimate both cost and cardinality, we use multitask learning [4] in order to improve the generalization by using the domain information contained in the training signals of related tasks as an inductive bias. Parameter sharing is one of the most common implementation strategies for multitask learning, which means that we train several tasks simultaneously, and the models of these tasks share some network layers. Since the cost of the query plan is correlated to the cardinality of each node in the plan tightly, we train the cost estimation task and cardinality estimation task simultaneously. These two tasks share common *embedding layer* and *representation layer* and we expect these two layers to learn general representations for both tasks, and the *estimation layer* can extract cost and cardinality from representations respectively. In this way, the generality of the learning model is better than training on cost only task, and thus the model can achieve much better performance.

Estimation Layer: Single vs Multiple. We can either learn cost only (Single) or cost-cardinality together (Multi). Multi-learning is better because the model can learn general representations for both tasks (see Section 7.3).

4.3 Training Model

Training Data Generation. First, given a dataset, we generate a join graph based on the primary keys (PK) and foreign keys (FK), where the nodes are tables and edges are PK-FK between two tables. Then according to the join graph, we select some joined tables. Next, we add some predicates for the selected jointed tables. To generate the predicates, we generate numeric expressions and string expressions as follows. For numeric expressions, we randomly select some columns with numeric type in the chosen tables and select a value for each column. Then we pick operators from '$>, <, =, ! =$' for each column. While for expressions with string values, we first pick operators from '$=, ! =, LIKE, NOT\ LIKE, IN$', and then select strings from datasets. After generating expressions for each table, we aggregate these expressions into complex predicates by using the 'AND'/'OR' semantics. Next we obtain all the training SQL queries, and we get physical plans from DBMS by using plan analysis tools.

Loss Function. Our model trains cost and cardinality simultaneously. The loss function can be a linear combination of cost loss and cardinality loss, and the weight can be regarded as hyper-parameters. We try different loss weights in $\{0.1, 0.2, 0.5, 1, 2, 5, 10\}$, and pick the one with the lowest validation error by cross validation. In order to achieve high quality and accelerate convergence speed, we take the normalized true cardinality/cost as the targets. The loss functions are formalized as below:

$$loss = \frac{1}{n}\sum_i (\omega \cdot qerror(cost_i, \hat{cost_i}) + qerror(card_i, \hat{card_i}))$$

$$qerror(cost_i, \hat{cost_i}) = \frac{\max(cost_i, \hat{cost_i})}{\min(cost_i, \hat{cost_i})}$$

$$qerror(card_i, \hat{card_i}) = \frac{\max(card_i, \hat{card_i})}{\min(card_i, \hat{card_i})}$$

where n is the number of training queries, $cost_i$ and $card_i$ are respectively the real cost and cardinality, and $\hat{cost_i}$ and $\hat{card_i}$ are respectively the estimated cost and cardinality.

Batch Training. In a batch of training samples with size $\mathcal{N}$, let $\mathcal{D}$ denote the maximum depth of the tree structure. We change the depth-first encoding into width-first encoding, where the input of each batch is organized hierarchically which means the first dimension of the nodes tensor equals to $\mathcal{D}$ while not $\mathcal{N}$. An extra tensor is needed to represent the edge of different layers of trees where each element indicates positions of the left child and right child. For each time, the model batches nodes in certain level of encoded plan trees and trains them all at once. In this way, the model only needs to run LSTM cell for each level of trees ($\mathcal{D}$ times) instead of running for each node, which reduces the training and evaluating time by $\mathcal{O}(\frac{2^{\mathcal{D}}\mathcal{N}}{\mathcal{D}})$ times.

5. STRING EMBEDDING

The distribution of numeric values can be learned, even though some values do not appear in the training data, because most numeric values are in a continuous space. For example, predicate $production\_year \in [1990, 2000)$ can be inferred from $production\_year \in [1990, 1995)$ and $production\_year \in [1995, 2000)$. However, string values are sparse and discrete, and thus hard to learn. So it is much harder to learn the predicates with string values than predicates with numeric values.

There are four intuitive ways to represent a predicate with string values, including *one-hot*, *selectivity*, *sample bitmap*, *hash bitmap*. The *one-hot* embedding maps a string to a bit in the vector. However it cannot estimate an approximate result for unseen string values. The *selectivity* embedding first translates the selectivity of a string value into a numerical value, and then utilizes the numeric value to embed the string. However, it cannot reflect the details on which tuples satisfy the predicate. The *sample bitmap* embedding uses samples to embed a string value, i.e., the string value is 1 if the sample contains it; 0 otherwise. However, it suffers from the 0-tuple problem for sparse predicates. The *hash bitmap* embedding first initializes a zero vector $\mathcal{V}$, and then for each character in the string, calculates its hash value $\mathcal{H}$ and set position $\mathcal{H}\%|\mathcal{V}|$ in the bitmap as 1. The number of character types is small (e.g., only 60 distinct characters in the JOB workload, including numbers, characters, punctuations, and other special characters). Therefore, the embedding vectors with hundreds of bits could effectively avoid hash collision even for unseen strings. The hash-bitmap embedding carries the characters contained in a string, so we can know the approximate overlap of two strings by taking 'AND' on their hash bitmap. However, the hash bitmap embedding can only capture the similarity between two strings but cannot reveal the co-occurrences of two strings.

Embedding-based Method. In order to represent the distribution of the string values in a dataset, we need to learn representations for string values in the predicates, and

Table 3: Example of substring extraction

Predicate	Extraction
title LIKE 'Din%'	"Dinos in Kas" → "Din"
title LIKE 'Sch%'	"Schla in Tra" → "Sch"
title LIKE '%06%'	"(2002-06-29)" → "06"
title LIKE '%08%'	"(2014-08-26)" → "08"

the representations can capture the co-occurrences of strings that co-exist in the same tuple. There are two kinds of possible string predicates in a query. The first is exact matching predicate (e.g., using operators '=' and 'IN'), and query strings in exact matching predicates will match the tuple values. The second is pattern matching predicate (e.g., using operator 'LIKE'), and the keywords in pattern matching predicates match substrings of data values in the dataset. In this section, our goal is to pre-train all the substrings in the predicates so that each keywords in predicates can be encoded by coexistence-aware representation. There are two challenges. The first is how to build a dictionary covering all the keywords in both current and future workloads and the size of the dictionary is bounded. The second is how to maintain all the keywords in the dictionary to efficiently get the encoding of each keyword with little space overhead. We first give the overview of the embedding method in Section 5.1, and then address these two challenges in Sections 5.2 and 5.3 respectively.

5.1 String Embedding Overview

Given a dataset and a query workload, we aim to encode all the keywords that either are used in the current workload or will be used in the future workload. Thus we do not just encode the plain strings appearing in the query workload. Instead, we generalize the strings/substrings and generate some important rules which could extract all the keywords in the query workload from the dataset. Then we extract all substrings that satisfy the rules and train a model to learn a representation for each of them. We take a collection of (sub)strings with the key values in one tuple as a sentence and use the skip-gram model [24] (a kind of model for word2vec) to train the string embedding. To efficiently get the embedding of each string in order to encode the queries online, we build an index with small space overhead and support fast prefix/suffix searching. Thus we address two challenges in string embedding.

Rule Generation. We generate some rules to generalize the keywords in the query workload, where each rule is similar to a regular expression, e.g., the prefix of a string with length 3. The rules are used for string embedding.

String Indexing. We use the rules to extract all the keywords, and then we encode them. To efficiently get the code of each substring, we construct trie indexes for storing all these keywords with their vectors.

5.2 Rule Generation

Rule Definition. Each rule can be expressed as a program. We borrow the idea from domain specific language (DSL) proposed by Gulwani [9, 10] to define a rule, which is composed of three parts, *pattern, string function and size*. The pattern matches substrings of tuples in the dataset. The string function decides which keyword in the substrings should be extracted. The size indicates the length of the keyword to be extracted. The pattern includes capital letters $\mathcal{P}_C$, lowercase letters $\mathcal{P}_l$, numerical values $\mathcal{P}_n$, white spaces $\mathcal{P}_s$ and exact matching token $\mathcal{P}_t(T)$ which can only match a specific substring T. The string function includes

Table 4: Candidate rules for "Dinos in Kas" → "Din%"

	Rules
"Dinos" → "Din"	$\langle Prefix, P_t(\text{“}D\text{”})P_l, 3\rangle$ $\langle Prefix, P_C P_l, 3\rangle$ $\langle Prefix, P_C P_t(\text{“}i\text{”})P_l, 3\rangle$ $\langle Prefix, P_C P_t(\text{“}in\text{”})P_l, 3\rangle$ $\langle Prefix, P_t(\text{“}Din\text{”})P_l, 3\rangle$
"Dinos in" → "Din"	$\langle Prefix, P_t(\text{“}D\text{”})P_l P_s P_l, 3\rangle$ $\langle Prefix, P_C P_l P_s P_l, 3\rangle$ $\langle Prefix, P_C P_t(\text{“}i\text{”})P_l P_s P_l, 3\rangle$ $\langle Prefix, P_C P_t(\text{“}in\text{”})P_l P_s P_l, 3\rangle$ $\langle Prefix, P_t(\text{“}Din\text{”})P_l P_s P_l, 3\rangle$
"Dinos in Kas" → "Din"	$\langle Prefix, P_t(\text{“}D\text{”})P_l P_s P_l P_s P_C P_l, 3\rangle$ $\langle Prefix, P_C P_l P_s P_l P_s P_C P_l, 3\rangle$ $\langle Prefix, P_C P_t(\text{“}i\text{”})P_l P_s P_l P_s P_C P_l, 3\rangle$ $\langle Prefix, P_C P_t(\text{“}in\text{”})P_l P_s P_l P_s P_C P_l, 3\rangle$ $\langle Prefix, P_t(\text{“}Din\text{”})P_l P_s P_l P_s P_C P_l, 3\rangle$

two types, *Prefix* and *Suffix*, which respectively extract the prefix/suffix of the string. The rule is formalized as below:

$$
\begin{aligned}
&rule = \langle \mathcal{F}, \mathcal{P}, \mathcal{L} \rangle, \quad \mathcal{F} \in \{\text{Prefix, Suffix}\} \\
&\mathcal{P} \in combination\{\mathcal{P}_C, \mathcal{P}_l, \mathcal{P}_s, \mathcal{P}_n, \mathcal{P}_t(T)\} \\
&\qquad \mathcal{P}_C = [A\text{–}Z]+; \mathcal{P}_l = [a\text{–}z]+; \mathcal{P}_s = whitespace+; \\
&\qquad \mathcal{P}_n = [0\text{–}9]+; \mathcal{P}_t(T) = T;
\end{aligned}
$$

where $\mathcal{P}$ is a pattern, $\mathcal{F}$ is a string function, and $\mathcal{L}$ is the length of a substring.

Rule Candidate Set. Given a keyword in the predicate and a string value in the dataset, we first find all substrings of the string value that match the keyword. Then for each matched substring, we generate all possible patterns that map the keyword to the substring, by enumerating all possible combinations of patterns in $\mathcal{P}_C$, $\mathcal{P}_l$, $\mathcal{P}_n$, $\mathcal{P}_s$, $\mathcal{P}_t(T)$. If the predicate is *prefix search* (e.g., LIKE “Din%”), then for each possible pattern *p*, we generate a rule (prefix, *p*, size of the keyword). If the predicate is *suffix search*, we generate a rule (suffix, *p*, size of the keyword). If the predicate is *substring search*, we generate a rule (prefix/suffix, *p*, size of the keyword), based on how the keyword matches the substring. All the possible rules will form a *Rule Candidate Set*. For example, Table 4 shows an example on how to generate rules for keywords in prefix searching. Pattern ‘Din%’ will select value ‘Dinos in Kas’ in the dataset, and the value ‘Dinos in Kas’ should be able to generate keyword ‘Din’ by using some general rules. The size of keyword ‘Din’ is 3, and it can be the prefix of different substrings of value ‘Dinos in Kas’. Different substrings (‘Dinos’, ‘Dinos in’, $\cdots$) may have different pattern sets. The pattern set is a set of patterns matching a substring. For example, in the domain of patterns, ‘Dinos’ can only be matched by 5 possible patterns, and thus it has 5 ways (rules) to generate keyword ‘Din’ in Table 4. Suffix searching is similar to prefix searching but uses suffix functions. Table 5 shows an example for the containment searching. Different from affix search, it considers both prefix and suffix functions, e.g., ‘06’ is the prefix of substring ‘06-29’ and it’s also the suffix of ‘2002-06’.

Rule Selection. Based on the candidate rules, we aim to find an *optimal* set of rules, which finds the minimum number of rules to cover the keywords in query workload. However, if we select those too general rules, the number of substrings would be too large. Therefore, we set an upper bound for the total number of extracted substrings.

Let $\mathcal{R}$ denote a subset of candidate rule $C_{\mathcal{R}}$ which could cover the keywords in the workload, $S_{\mathcal{R}}$ be the set of substrings which are extracted by rules in $\mathcal{R}$ from the datasets, and $S_{\mathcal{W}}$ is the set of keywords in the workload. We aim to minimize the size of $\mathcal{R}$ with an upper bound $\mathcal{B}$ where $S_{\mathcal{R}}$ contains all strings in $S_{\mathcal{W}}$. The problem is formalized below:

Table 5: Candidate rules for "(2002-06-29)" → "%06%"

	Rules
"06-" → "06"	$\langle Prefix, P_t(\text{“06”})P_t(\text{“}-\text{”}), 2\rangle$ $\langle Prefix, P_n P_t(\text{“6”})P_t(\text{“}-\text{”}), 2\rangle$ $\langle Prefix, P_n P_t(\text{“}-\text{”}), 2\rangle$ $\langle Prefix, P_t(\text{“0”})P_n P_t(\text{“}-\text{”}), 2\rangle$
"06-29" → "06"	$\langle Prefix, P_t(\text{“06”})P_t(\text{“}-\text{”})P_n, 2\rangle$ $\langle Prefix, P_n P_t(\text{“6”})P_t(\text{“}-\text{”})P_n, 2\rangle$ $\langle Prefix, P_n P_t(\text{“}-\text{”})P_n, 2\rangle$ $\langle Prefix, P_t(\text{“0”})P_n P_t(\text{“}-\text{”})P_n, 2\rangle$
"06-29)" → "06"	$\langle Prefix, P_t(\text{“06”})P_t(\text{“}-\text{”})P_n P_t(\text{“)”}), 2\rangle$ $\langle Prefix, P_n P_t(\text{“6”})P_t(\text{“}-\text{”})P_n P_t(\text{“)”}), 2\rangle$ $\langle Prefix, P_n P_t(\text{“}-\text{”})P_n P_t(\text{“)”}), 2\rangle$ $\langle Prefix, P_t(\text{“0”})P_n P_t(\text{“}-\text{”})P_n P_t(\text{“)”}), 2\rangle$
"-06" → "06"	$\langle Suffix, P_t(\text{“}-\text{”})P_t(\text{“06”}), 2\rangle$ $\langle Suffix, P_t(\text{“}-\text{”})P_n P_t(\text{“6”}), 2\rangle$ $\langle Suffix, P_t(\text{“}-\text{”})P_n, 2\rangle$ $\langle Suffix, P_t(\text{“}-\text{”})P_t(\text{“0”})P_n, 2\rangle$
"2002-06" → "06"	$\langle Suffix, P_n P_t(\text{“}-\text{”})P_t(\text{“06”}), 2\rangle$ $\langle Suffix, P_n P_t(\text{“}-\text{”})P_n P_t(\text{“6”}), 2\rangle$ $\langle Suffix, P_n P_t(\text{“}-\text{”})P_n, 2\rangle$ $\langle Suffix, P_n P_t(\text{“}-\text{”})P_t(\text{“0”})P_n, 2\rangle$
"(2002-06" → "06"	$\langle Suffix, P_t(\text{“(”})P_n P_t(\text{“}-\text{”})P_t(\text{“06”}), 2\rangle$ $\langle Suffix, P_t(\text{“(”})P_n P_t(\text{“}-\text{”})P_n P_t(\text{“6”}), 2\rangle$ $\langle Suffix, P_t(\text{“(”})P_n P_t(\text{“}-\text{”})P_n, 2\rangle$ $\langle Suffix, P_t(\text{“(”})P_n P_t(\text{“}-\text{”})P_t(\text{“0”})P_n, 2\rangle$

$$
\mathcal{R} = arg \min_{\mathcal{R} \subseteq C_{\mathcal{R}}} (|\mathcal{R}|) \qquad \text{s.t.} \quad |S_{\mathcal{R}}| < \mathcal{B}, S_{\mathcal{W}} \subseteq S_{\mathcal{R}}
$$

This is an NP-hard problem by a reduction from a classical set cover problem (SCP)[3]. Now the universe is $S_{\mathcal{W}}$, and the subset is $S_r \cap S_{\mathcal{W}}$ where $r \in C_{\mathcal{R}}$. We also have that the union of subsets $\sum_{r \in C_{\mathcal{R}}} S_r \cap S_{\mathcal{W}}$ equals to the universe $S_{\mathcal{W}}$. Our target is to find the minimum number of subsets to cover the universe. We propose a greedy solution to address this problem approximately. We add a rule r to the rule set $\mathcal{R}$ covering the most substrings in $S_{\mathcal{W}}$ each time. If the total size of $S_{\mathcal{R}}$ exceeds the bound $\mathcal{B}$, we remove the rule r with the largest S_r and repeat. For example, consider three rules in the Table 5, $\langle Prefix, P_t(\text{“06”}), 2\rangle$, $\langle Prefix, P_n, 2\rangle$ and $\langle Suffix, P_t(\text{“(”})P_n P_t(\text{“}-\text{”})P_n, 2\rangle$. The first rule can only extract ‘06’. The second rule can extract 5 substrings including {‘20’, ‘06’, ‘29’, ‘08’, ‘26’}. The third rule can extract {‘06’, ‘08’}. The third rule would be selected in our algorithm, because it’s general and will not extract too many substrings. Based on the selected rules, we generate all the string values in the dataset and store the extracted substrings in the dictionary.

5.3 String Indexing

There could be a large number of strings and it is expensive to maintain all strings in a dictionary. In order to avoid storing a huge number of duplicate tokens, we build a trie index to store the mapping from a string to its code.

String Indexing. We use both prefix trie and suffix trie as string index. Substrings extracted by the *prefix* function are stored into prefix trie and substrings extracted by the *suffix* function are stored in suffix trie. So each string in the dictionary must have one or two paths in the index. Leaf nodes of the trie index are representation vectors of strings.

Online Searching. When a new query comes, there may be some query strings which do not exist in the dictionary. These query strings may be in prefix searching (LIKE s%), suffix searching(LIKE %s), keyword searching(=) or containment searching(LIKE %s%). For prefix search, we search the longest prefix of the query string. For suffix search, we search the longest suffix of the query string. For

Table 6: Methods on JOB workload

Methods	Represent Network	Predicate Network	Estimate Network	String Encoding	Sample Bitmap
`PostgreSQL`	No	No	No	No	No
`MySQL`	No	No	No	No	No
`Oracle`	No	No	No	No	No
`MSCN`	No	CNN	`Single`	No	Yes/No
`TNN`	NN	LSTM	`Multi`	Rule+Embed	Yes/No
`TLSTM`	LSTM	LSTM	`Multi`	Rule+Embed	Yes/No
`TPool`	LSTM	Pool	`Multi`	Rule+Embed	Yes/No

other searches, we search both the longest prefix and longest suffix of the query string, and then pick the longest one as the representation. Considering "title LIKE 'Dino%'", we search the prefix trie and take the representation of 'Din' as the representation of 'Dino'.

6. SUPPORTING DATABASE UPDATES

Our model can be extended to support database changes, including tuple update, column update and table update.

(1) Tuple update. The estimator reserves (e.g., 500 bits) in each sample bitmap for tuple insertion. When tuples are added, the estimator samples new tuples in the same ratio with original data sampling, and fills the reserved bits with 0 or 1 based on the effect that whether the samples satisfy the query. When tuples are deleted, we just update the sample bits. The updated sample bitmap for each query can be input into the model for evaluation directly. When tuples are updated, samples would also be updated locally, and the estimator will recalculate the sample bitmap for each query. The updated sample bitmap for each query can be input into the model for evaluation directly. If many tuples are updated, we fine-tune the model online by training some queries related to the updated tuples incrementally.

(2) Column update. The estimator reserves several bits in one-hot column vector for each table. When a column is inserted, the estimator selects a bit representing the new column and calculates the sample bitmap online. Besides, the estimator fine-tunes the updated table and its ancestors in the query tree incrementally. For column deletion, the model does not need to make any change online.

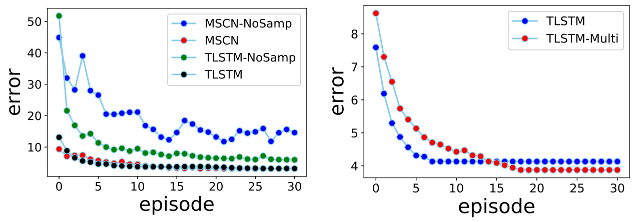
(3) Table update. The estimator reserves several bits in the one-hot table vector and column vector. When a table is inserted, the estimator selects a bit representing the new table, selects bits representing the columns in the new table and calculates the sample bitmap online. Besides, the estimator also fine-tunes the created table and its ancestors in the query tree incrementally. For table deletion, the model does not need to make any change online.

7. EXPERIMENTS

7.1 Experiment Setting

Datasets. We use the real datasets IMDB and JOB [12]. It is much harder to estimate the cardinality and cost on the IMDB dataset than TPC-H, because of the correlations and skew distributions of the IMDB dataset. The IMDB dataset includes 22 tables, which are joined on primary keys and foreign keys. We build indexes on primary and foreign keys. We use two types of query workloads for tesing.

(1) The first workload type contains predicates with numeric attributes only [12]. We use three widely-used workloads in previous works: (*i*) easy workload *Synthetic*, (*ii*) medium workload *Scale*, and (*iii*) hard workload *JOB-light*[1]. *Synthetic* contains queries with 0-2 joins and there are 5000



(a) Card Validation Error **(b)** Cost Validation Error

Figure 6: Validation Error on Numeric Workload

queries. *Scale* contains queries with 0-4 joins and there are 500 queries. *JOB-light* contains queries with 1-4 joins and there are 70 queries. We generate 100K queries with 0-3 joins. We take 90% as training data and 10% as validation data, and the three workloads as the test data.

(2) The second workload type contains complex predicates with string attributes. We generate training data based on the join graph of IMDB and predicates used in the JOB workloads. We generate 50,000 queries on single tables, 50,000 queries with 0-4 joins, and 50,000 queries with more than 5 joins. We obtain the plans for these queries from PostgreSQL and use them to train our model. We take 90% of generated queries as training data and 10% as validation data. The 113 JOB queries[2] are taken as the test workload.

Methods. Table 6 shows the baseline methods. We compare with three traditional cost estimators from two open sourced databases (`PostgreSQL` 9.5.19 and `MySQL` 5.7.26) and a commercial database (`Oracle` 18.4.0.0.0). `MSCN` uses a multi-set convolutional neural network to learn the cardinality of SQL queries, but it doesn't support string predicates and does not support cost estimation. To extend `MSCN` to estimate the cost, we utilize `PostgreSQL` by taking the estimated cardinality of `MSCN` as input. `TNN` uses fully connected neural network (NN) as the representation layer and LSTM as the predicate embedding layer. `TLSTM` uses a LSTM as the representation layer and LSTM as the predicate layer. `TPool` uses LSTM as the representation layer and min-max pooling (Pool) as the predicate layer.

Environment. We use a machine with Intel(R) Xeon(R) CPU E5-2630 v4, 128GB Memory, and GeForce GTX 1080.

7.2 Numeric predicates only

As existing methods only support the predicates with numerical values, we report the results of different methods on the workload with numeric predicates only. After models are converged, we test them on the workload including *Synthetic*, *Scale* and *JOB-light*. Figure 6 shows the validation errors, and Tables 7, 8 show the test errors.

Sample vs Non-Sample. The methods with sample bitmap (e.g., `TLSTM` and `MSCN`) outperform the methods without bitmap (e.g., `TLSTM-NoSamp` and `MSCN-NoSamp`) on both the validation and test workloads, because sample bitmap reveals both the distribution of data and the semantic of predicates explicitly, and the model easily extracts the approximate intermediate result distribution from the bitmap.

Tree-structured Model vs CNN. `TLSTM` outperforms `MSCN`, because the tree-structured model could capture the query predicate and learn the semantic of them much better than convolutional network. However, `MSCN` and `TLSTM` make no much difference on validation queries because the validation workload contains only easy queries like the training

[1]https://github.com/andreaskipf/learnedcardinalities

[2]https://github.com/gregrahn/join-order-benchmark

Table 7: Cardinality on numeric workloads(test errors)

Synthetic	median	90th	95th	99th	max	mean
`PostgreSQL`	1.69	9.57	23.9	465	373901	154
`MySQL`	2.07	22.6	50.6	625	458835	353
`Oracle`	1.97	12.4	40.1	473	545912	378
`MSCN-NoSamp`	2.14	6.72	11.5	114	1870	23.6
`TLSTM-NoSamp`	1.97	5.53	9.13	81.5	988	10.3
`MSCN`	1.19	3.32	6.84	30.51	1322	2.89
`TNN`	1.40	5.51	10.7	43.1	441	3.57
`TLSTM`	1.20	3.21	6.12	25.2	357	2.87
`TPool`	**1.18**	**3.19**	**6.05**	**24.5**	**323**	**2.81**
Scale	median	90th	95th	99th	max	mean
`PostgreSQL`	2.59	200	540	1816	233863	568
`MySQL`	3.08	90.1	329	7534	54527	426
`Oracle`	2.43	114	482	3412	102833	397
`MSCN-NoSamp`	2.33	96.1	257	1110	4013	131
`TLSTM-NoSamp`	2.06	69	176	931	3295	78.2
`MSCN`	**1.42**	37.4	140	793	3666	35.1
`TNN`	1.59	58.7	141	573	2238	31.3
`TLSTM`	1.43	38.8	139	469	1892	28.1
`TPool`	**1.42**	**37.3**	**125**	**345**	**1813**	**26.3**
JOB-light	median	90th	95th	99th	max	mean
`PostgreSQL`	7.93	164	1104	2912	3477	174
`MySQL`	9.55	303	685	2256	2578	149
`Oracle`	8.32	374	976	2761	3331	157
`MSCN-NoSamp`	5.43	126	978	1310	2020	100
`TLSTM-NoSamp`	5.18	97.3	613	864	1541	72.3
`MSCN`	3.82	78.4	362	927	1110	57.9
`TNN`	**2.95**	76.8	275	799	902	49.8
`TLSTM`	3.73	50.8	157	256	289	24.9
`TPool`	3.51	**48.6**	**139**	**244**	**272**	**24.3**

workload, and the sample bitmap of length 1000 is sufficient to represent distribution of each intermediate result and brings little error for the cardinality estimation of queries. Thus the advantage of tree-structured model is not significant on easy queries for cardinality estimation. However, if we apply the model to harder queries like the JOB-light workload, then small bias or 0-tuple problem of the sample bitmap would lead to large errors of the estimated cardinality. On the JOB-light workload, `TLSTM` outperforms `MSCN` by 20% on mean error for cardinality estimation, and 40% for cost estimation. On the Scale workload, `TLSTM` outperforms `MSCN` by 2 times on max error for cardinality estimation and cost estimation. On Synthetic workload, `TLSTM` outperforms `MSCN` by 4 times on max error for cardinality estimation and 3 times for cost estimation. The main reasons are three-fold. (1) Tree-structured model is good at representing complicated plans and predicates. (2) Tree-structured model captures more correlations for complex queries. (3) Cost model used by `MSCN` is hard to be tuned.

LSTM vs TNN. On the JOB-light workload. For cardinality estimation, `TLSTM` outperforms `TNN` by 3 times on max error and 2 times on the mean error. While for cost estimation, `TLSTM` outperforms `TNN` by 3 times on mean error and max error. On the Synthetic workload, `TLSTM` outperforms `TNN` on all the cardinality errors. On the Scale workload, `TLSTM` outperforms `TNN` on all the cardinality errors and cost errors. `TLSTM` outperforms `TNN` on harder queries for cardinality estimation and cost estimation. These results show that LSTM uses the representation model to learn more robust representations for sub-plans, as LSTM has an extra channel to avoid information vanishing for complex queries.

Multi-Learning vs Cost Only Learning. On the JOB-light workload, multitask technique can help the model to achieve 3 times improvement on 90-99th, mean and max error for cost estimation. On the Synthetic workload, `TLSTM-Multi` outperforms `TLSTM` on all the cost errors. On the Scale workload, `TLSTM-Multi` outperforms `TLSTM` by more than 2 times on max error for cost estimation. The reason is that multitask learning can improve the generalization ability of the model for complex queries.

Table 8: Cost on numeric workloads (test errors)

Synthetic	median	90th	95th	99th	max	mean
`PostgreSQL`	15.1	65.1	173	1200	8040	62.7
`MySQL`	4.51	39.7	94.7	449	7203	32.4
`Oracle`	6.72	41.1	124	796	6674	56.1
`MSCN-NoSamp`	10.3	24.7	234	569	2110	31.6
`TLSTM-NoSamp`	5.34	21.2	153	328	1345	19.8
`MSCN`	3.14	7.43	18.1	65.8	739	10.3
`TNN`	1.49	4.50	10.6	61.5	718	4.35
`TLSTM`	1.56	4.47	10.7	57.7	689	4.45
`TLSTM-Multi`	1.49	4.33	10.2	55.8	624	4.16
`TPool`	**1.48**	**4.12**	**10.1**	**47.6**	**532**	**3.99**
Scale	median	90th	95th	99th	max	mean
`PostgreSQL`	13.3	38.9	81.1	718	1473	35.7
`MySQL`	4.25	37.4	131	577	5157	40.7
`Oracle`	6.49	27.7	61.4	623	3612	31.5
`MSCN-NoSamp`	3.32	20.9	30.5	274	1173	21.2
`TLSTM-NoSamp`	2.19	13.4	21.7	228	1162	14.9
`MSCN`	1.79	10.6	27.1	88.8	1027	8.22
`TNN`	1.61	5.37	13.5	72.7	714	5.53
`TLSTM`	1.58	5.51	14.4	70.1	611	5.21
`TLSTM-Multi`	1.56	5.56	12.2	68.6	**254**	4.41
`TPool`	**1.54**	**5.29**	**11.9**	**67.6**	**254**	**4.39**
JOB-light	median	90th	95th	99th	max	mean
`PostgreSQL`	26.8	332	696	2740	3020	173
`MySQL`	9.47	102	342	1293	2228	84.5
`Oracle`	12.3	157	278	1366	1825	102.1
`MSCN-NoSamp`	12.4	152	231	1071	1553	62.7
`TLSTM-NoSamp`	10.4	103	217	986	1271	38.3
`MSCN`	4.75	11.3	40.1	563	987	27.4
`TNN`	2.06	25.5	134	293	401	19.1
`TLSTM`	3.66	32.1	80.3	445	583	17
`TLSTM-Multi`	**1.85**	13.2	22.9	**95**	**123**	5.81
`TPool`	**1.85**	**11.1**	**20.3**	101	125	**5.76**

Traditional vs Learning-based. Learning-based methods (`MSCN`, `TLSTM`, `TPool`) outperform traditional approaches (`PostgreSQL`, `MySQL`, `Oracle`), because traditional methods rely on the independent assumption among different columns while learning-based methods can capture the correlations between columns/tables. Our model achieves the best performance, as our model captures more correlations.

7.3 Both string and numeric predicates

To investigate our string encoding techniques, we divide the training data into two parts and train on them respectively. The first is workload without join, and the second is workload with multiple joins. All these training queries contain complicated predicates on both numeric and string values. Firstly, we train our models on the first workload and test them on single table workload, and this can compare the performance of different predicate embedding techniques directly. Secondly, we train our models on the second workload and evaluate them on 113 JOB queries, and this can compare the effects of different predicate embedding techniques on complicated queries estimation.

7.3.1 Evaluation on single table workload

The predicates in the workload contain string equal search, string pattern search, range query and numeric equal search. For conjunction predicates, the complex predicates are composed of expressions with 'AND' and 'OR' semantics. The most complex predicate in the workload has 4 boolean conjunctions and 5 expressions. We set the batch

Table 9: Test errors on the JOB workload with strings

Cardinality	median	90th	95th	99th	max	mean	Cost	median	90th	95th	99th	max	mean
PostgreSQL	184	8303	34204	106000	670000	10416	PostgreSQL	4.90	80.8	104	3577	4920	105
MySQL	104	28157	213471	1630689	2487611	60229	MySQL	7.94	691	1014	1568	1943	173
Oracle	119	55446	179106	697790	927648	34493	Oracle	6.63	149	246	630	1274	55.3
TLSTM-Hash	11.1	207	359	824	1371	83.3	TLSTM-Hash	4.47	53.6	149	239	478	24.1
TLSTM-Emb	11.6	181	339	777	1142	70.2	TLSTM-Emb	4.12	18.1	44.1	105	166	10.3
TLSTM-EmbRule	10.9	136	227	682	904	55.0	TLSTM-EmbRule	4.28	13.3	22.5	104	126	8.6
TPool	**10.1**	**74.7**	**193**	**679**	**798**	**47.5**	TPool	**4.07**	**11.6**	**17.5**	**63.1**	**67.3**	**7.06**

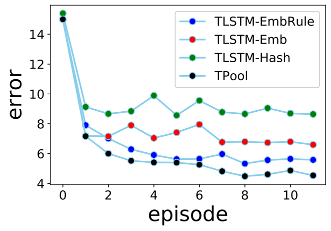


Figure 7: Cardinality Error on Single Table Queries

size as 64 and divide all 50,000 queries into 782 batches. We take the first 700 batches as the training data, and the remainders as test data. We use the Adam optimizer and the learning rate is 0.001. Since the semantic of predicates has no much effect on execution cost on single table queries (Scan operation on the same table takes similar time no matter what predicate is.), we only report the error for cardinality estimation in Figure 7.

Hash Bitmap vs String Embedding. TLSTM-Hash performs the worst for cardinality estimation on test queries and its convergence speed is the slowest. TLSTM-Emb outperforms TLSTM-Hash by 30% on cardinality error, because string embedding can capture coexistence relation among different strings to improve the performance on the test workload.

Rule vs No-Rule. As the rules can pre-train many more strings and encode them with more accurate distributed representations, TLSTM-EmbRule can cover more strings and thus outperforms TLSTM-Emb by around 15%.

Tree-Pooling Predicate vs Tree-LSTM Predicate. TPool outperforms TLSTM-EmbRule by 20% on test workload, because the tree structure with Min-Max pooling is more capable of representing compound predicate in semantic, and it can learn a better predicate representation with stronger generalization ability.

7.3.2 *Evaluation on the JOB workload*

We train the representation and output layers on 100,000 queries with multiple joins. We take 90% of multi-table join queries as training data and 10% of them as validation data. We train the model until the validation cardinality error will not decrease anymore, and then we evaluate the trained model on the JOB queries. The estimation errors are shown in Table 9.

Traditional vs Learning-based. Experimental results show that traditional methods (PostgreSQL,MySQL,Oracle) have large errors on cardinality estimation for harder queries in the JOB workload. The reason is that distribution information passed to the root of the plan is not accurate using statistical or sampling methods, and traditional methods estimate cardinality of most of queries as 1 (the true value is from 0 to millions). The cost estimation error is less than cardinality estimation, and the learning-based methods still outperform traditional methods by 1-2 orders of magnitude.

Hash Bitmap vs String Embedding. TLSTM-Emb outperforms TLSTM-Hash on 90-99th errors, max error and mean error for cardinality estimation, because the string embedding not only represents the correlations among columns but the correlations among tables, and the methods with string embedding perform better for cardinality estimation on complex queries with multiple joins. Moreover, the performance difference for cost estimation is even larger. TLSTM-Emb outperforms TLSTM-Hash by 2 times on mean and 99th cost errors, and 3 times on max, 95th and 90th cost errors. The improvement on complex queries for cost estimation is due to the correlations carried by string embedding.

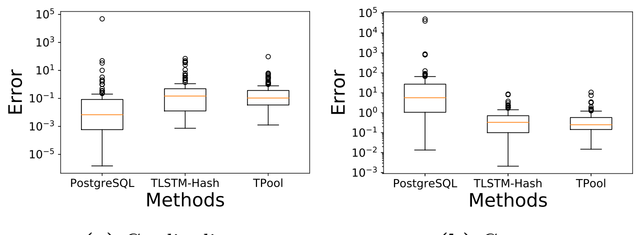
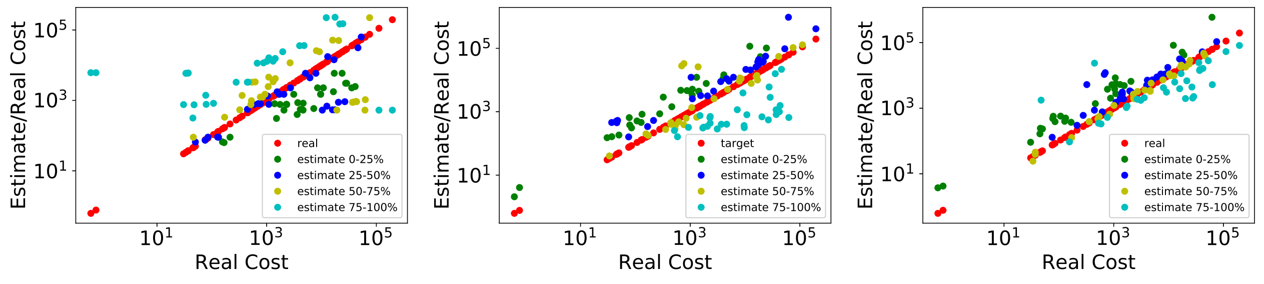


Figure 8: Estimation errors on the JOB workload. The box boundaries are at the 25th/50th/75th percentiles

Rule vs No-Rule. TLSTM-EmbRule outperforms TLSTM-Emb on all the errors for cardinality estimation, especially on 90-99th errors, max error and mean error. The methods with rules achieve better performance, because the rules extract lots of substrings from datasets so that all the keywords in the workload are trained and the distribution representations of strings contain more coexistence relations. Complex queries tend to gain more benefit from rules, because errors would accumulate rapidly for complex queries. Similar to cardinality estimation, TLSTM-EmbRule also outperforms TLSTM-Emb on 90-95th, max and mean errors on cost errors.

Tree-Pooling Predicate vs Tree-LSTM Predicate. The difference between TLSTM-EmbRule and TPool is the structure of the predicate embedding model, and TPool outperforms TLSTM-EmbRule on all the cardinality errors and cost errors. This is because that the tree model with Min-Max Pooling can represent the compound predicate better and train a more robust model for cardinality and cost estimation. On 99th and max errors for cost estimation, TPool outperforms TLSTM-EmbRule by 1.5-2 times, because the predicates representation trained by the Min-Max Pooling structure still keeps accurate for complex queries.

Distribution of errors. Figure 8 shows the error variance on the JOB workload. For PostgreSQL, we have tuned the factor of page IO so that the unit of the estimated cost equals to the unit of time (milliseconds). However, it still overestimates the cost and underestimates the cardinality, and the maximal error is very large. Our methods underestimate both cost and cardinality of queries, but errors of our methods are more concentrated and smaller. We draw the real time and estimated cost together in Figure 9, which is more intuitive and comprehensive. The cost estimated by TPool fits the real time very well. PostgreSQL can not



(a) `PostgreSQL` **(b)** `TLSTM-Emb` **(c)** `TPool`

Figure 9: Distribution of Estimated Cost

Table 10: Time and Errors on Varying Training Sizes

Training Size (K)	20	40	60	80	100	120	140
Cardinality Error	134	101	72.3	55.1	50.1	47.5	49.0
Time (Minutes)	103	149	208	251	302	352	406

Table 11: Efficiency on estimation of 113 queries (ms)

`PostgreSQL`	`MySQL`	`Oracle`	`MSCN`	`TLSTM`	`TPool`	`TPoolBatch`
18.9	22.1	17.6	14.2	70.3	47.3	**3.63**

estimate the small cost at all and the estimated cost is very scattered. The error variance of `TLSTM-Hash` is between the `TPool` and `PostgreSQL`.

7.4 Efficiency

Training Time. We show the training time on the queries with 0-12 joins and test errors on the JOB workload with varying training sizes in Table 10. We can see that our method takes about 5 hours to train 10K queries.

Estimation Time. We evaluate the online estimation time. Table 11 shows the efficiency of different methods on 113 queries on the JOB workload. The *Batch* indicates whether the batch technique described in Section 4.3 is used. The estimation time of PostgreSQL/MySQL/Oracle is obtained from the planning time without query optimization. `TLSTM` and `TPool` estimate these queries one by one, and `TPoolBatch` estimates these queries in batch. `TPoolBatch` outperforms `TPool` by one order of magnitude, because the batch evaluation reduces the number of times computing the model, and increases the parallelism as discussed in Section 4.3. `TPool` outperforms `TLSTM` by 50%, because tree-pooling replaces neural networks in the predicate model and has less computation cost in estimation.

7.5 Database Update

We evaluate our method on database update. As table *cast-info* has the most join relations in the JOB join graph (it has 3.6 million tuples and can join with 12 tables), we use this table for tuples update. As table *movie-keyword* has lots of related queries in the test workload (over a half in *Synthetic*, *Scale* and *JOB-light*), we use this table for table update. As the attribute *role-id* is frequent in the predicate, we use this column for column update.

Firstly, we remove table *movie-keyword* from the database, and remove column *role-id* from table *cast-info*. Secondly, we extract 3,000,000 tuples (around 10%) from table *cast-info* as inserted tuples, and train the model on the remainder tuples. For tuple insertions, we insert 1,000,000 tuples each time and insert 3 times to table *cast-info*. For column insertion, we insert column *role-id* to table *cast-info*. For table insertion, we insert table *movie-keyword*. Table 12 shows the affected queries by the updates.

After tuple update, we directly estimate our model using the model in Section 6. After column/table update, we fine tune our method by online training some queries, where

Table 12: Affected queries by database updates

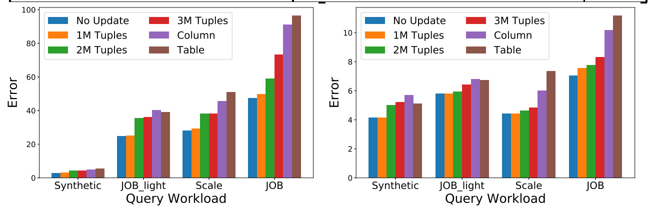
# Affected queries	Synthetic	Scale	JOB-light	JOB
total	5000	500	70	113
insert tuples to `cast-info`	1614	209	35	57
insert column `role-id`	508	57	15	26
insert table `movie-keyword`	1628	231	42	75

Table 13: Number of queries used by fine-tuning

# Queries	Synthetic	Scale	JOB-light	JOB
insert column `role-id`	100	100	100	200
insert table `movie-keyword`	150	150	150	300

Table 14: Time of fine-tuning

Time (seconds)	Synthetic	Scale	JOB-light	JOB
insert column `role-id`	2.31			4.12
insert table `movie-keyword`	4.17			7.68



(a) Cardinality Errors **(b)** Cost Errors

Figure 10: Errors with Database Changes

Table 13 shows the number of queries used for fine tuning. Table 14 shows the running time for fine tuning. We can see that for tuple updates, we do not need to online tuning the model. For column and table update, our method only takes several seconds for fine tuning.

After database updates and fine tuning, we run the test workloads. Figure 10 shows the estimation errors. We can see that the errors are similar with those without updates, because the sample bitmap and our fine-tuning method can capture the plan semantics and data correlations.

8. CONCLUSION

In this paper, we proposed an end-to-end learning-based tree-structured cost estimator for estimating both cost and cardinality. We encoded query operation, meta data, query predicate and some samples into the model. The model contained embedding layer, representation layer and estimation layer. We proposed an effective method to encode string values into the model to improve the model generalization. Extensive results on real datasets showed that our method outperformed existing techniques.

9. REFERENCES

[1] Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406 – 427, 2009.

[2] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2012.

[3] A. Caprara, P. Toth, and M. Fischetti. Algorithms for the set covering problem. *Annals of Operations Research*, 98(1):353–371, 2000.

[4] R. Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, 1997.

[5] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *Algorithms - ESA*, pages 605–617, 2003.

[6] P. Flajolet, E. Fusy, O. Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, 2007.

[7] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[8] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009.

[9] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[10] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[11] Y. E. Ioannidis. The history of histograms (abridged). In *PVLDB*, pages 19–30, 2003.

[12] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[13] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[14] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[15] G. Li, X. Zhou, and S. Li. Xuanyuan: An ai-native database. *Data Engineering*, page 70, 2019.

[16] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *PVLDB*, 12(12):2118–2130, 2019.

[17] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[18] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. *SIGMOD Rec.*, 19(2):1–11, May 1990.

[19] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *SoCC*, pages 153–166, 2015.

[20] G. Lohman. Is query optimization a "solved" problem?, 2014.

[21] T. Malik, R. C. Burns, and N. V. Chawla. A black-box approach to query cardinality estimation. In *CIDR*, pages 56–67, 2007.

[22] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *CoRR*, abs/1904.03711, 2019.

[23] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *CoRR*, abs/1902.00132, 2019.

[24] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[25] T. Neumann, V. Leis, and A. Kemper. The complete story of joins (in hyper). In *BTW*, pages 31–50, 2017.

[26] F. Olken and D. Rotem. Random sampling from database files: A survey. In *Statistical and Scientific Database Management*, 1990.

[27] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *DEEM@SIGMOD*, pages 4:1–4:4, 2018.

[28] K. Whang, B. T. V. Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, 1990.

[29] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigumus, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.

[30] W. Wu, J. F. Naughton, and H. Singh. Sampling-based query re-optimization. In *SIGMOD*, pages 1721–1736, 2016.

[31] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Selectivity estimation with deep likelihood models. *CoRR*, abs/1905.04278, 2019.

[32] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. *ICDE*, 2020.

[33] J. Zhang, Y. Liu, K. Zhou, and G. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*, pages 415–432, 2019.

[34] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing xml queries. In *PVLDB*, pages 289–300, 2005.