Automatic View Generation with Deep Learning and Reinforcement Learning

Haitao Yuan, Guoliang Li*, Ling Feng, Ji Sun, Yue Han
Department of Computer Science, Tsinghua University, China
{yht16,sun-j16,han-y19}@mails.tsinghua.edu.cn, {liguoliang,fengling}@tsinghua.edu.cn

***Abstract*—Materializing views is an important method to reduce redundant computations in DBMS, especially for processing large scale analytical queries. However, many existing methods still need DBAs to manually generate materialized views, which are not scalable to a large number of database instances, especially on the cloud database. To address this problem, we propose an automatic view generation method which judiciously selects "highly beneficial" subqueries to generate materialized views. However, there are two challenges. (1) How to estimate the benefit of using a materialized view for a query? (2) How to select optimal subqueries to generate materialized views? To address the first challenge, we propose a neural network based method to estimate the benefit of using a materialized view to answer a query. In particular, we extract significant features from different perspectives and design effective encoding models to transform these features into hidden representations. To address the second challenge, we model this problem to an ILP (Integer Linear Programming) problem, which aims to maximize the utility by selecting optimal subqueries to materialize. We design an iterative optimization method to select subqueries to materialize. However, this method cannot guarantee the convergence of the solution. To address this issue, we model the iterative optimization process as an MDP (Markov Decision Process) and use the deep reinforcement learning model to solve the problem. Extensive experiments show that our method outperforms existing solutions by 28.4%, 8.8% and 31.7% on three real-world datasets.**

I. Introduction

In many OLAP systems, analytical SQL queries share common subqueries and building views on these subqueries can avoid redundant computations and improve the performance. For example, Figure 1 reports the redundant computations on a real-world workload of Alibaba Cloud. Figure 1(a) reports the number of total queries (denoted as total) and the number of queries including redundant computation (denoted as redundant) on six projects. Figure 1(b) illustrates the cumulative percentage of queries including redundant computation among total queries. To address this problem, many studies [20], [3], [31], [34] propose to materialize views for the common subqueries for sharing computations. For instance, `BigSub` [20] models the subquery selection problem as a bipartite labeling problem and then designs an iterative optimization based method to solve it. However, there are still two challenges.

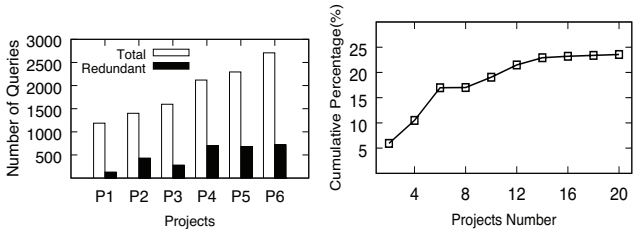
First, it is challenging to evaluate the benefit of using a materialized view to answer a query. Intuitively, the benefit can be computed with the saved cost by using the materialized view for the query. However, it's not realistic to actually rewrite queries with all possible views and execute all rewritten queries to get the actual cost. Therefore, we design a cost estimation model using deep learning. Although there are some deep learning techniques for cost estimation [36], [29], they are designed for estimating the cost of a single query but cannot estimate the cost of a query rewritten by a view. To address this challenge, we first extract significant features from two kinds of information: query/view plans and query/view related tables. We then split these features into *numerical features* and *non-numerical features*. In addition, we design different encoding models to transform different features into hidden representations. For example, we use the sequence encoding model to encode query/view plans, because plans can be regarded as sequences. Finally, to capture the linear and non-linear correlations between the cost and these features, we design an effective model to estimate the cost, which consists of two parts: a linear wide part and a non-linear deep part.

Second, it is hard to automatically select a set of high-quality subqueries to materialize based on their benefits. To address this issue, we model the subquery selection problem as an ILP (Integer Linear Programming) problem. To avoid the heavy overhead for computing actual optimal solutions, we propose an iterative optimization method to get approximate optimal solutions. Inspired by `BigSub` [20], we iteratively compute the probability of selecting a subquery to materialize. However, this method cannot converge to a stable result, because different iterations have no memory ability and cannot share feedback from optimization process, which leads to repeated oscillation of optimization results. To address this problem, `BigSub` [20] forbids turning selected subqueries to unselected when the number of iterations exceeds a certain threshold. However, `BigSub` would degenerate into a greedy method and thus leads to poor results. To address this problem, we propose to a reinforcement learning method, which transforms the optimization process to an MDP (Markov Decision Process) and then applies the deep reinforcement learning model DQN (Deep Q-Network) to get a stable result.

In summary, we make the following contributions.
(1) We formally define the problem of automatically selecting subqueries to materialize for sharing computations (see Section II). Meanwhile, we propose an end-to-end learning-based system for solving the problem (see Section III).
(3) We extract useful features from different perspectives for estimating the benefit of using a view for answering a query. We use different encoding models to encode different features

[1]Guoliang Li is the corresponding author. This work was supported by NSF of China (61925205, 61632016), Huawei, Alibaba, and TAL education.

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00133



(a) total vs redundant (b) cumulative percentage

Fig. 1. Redundant computation on several projects

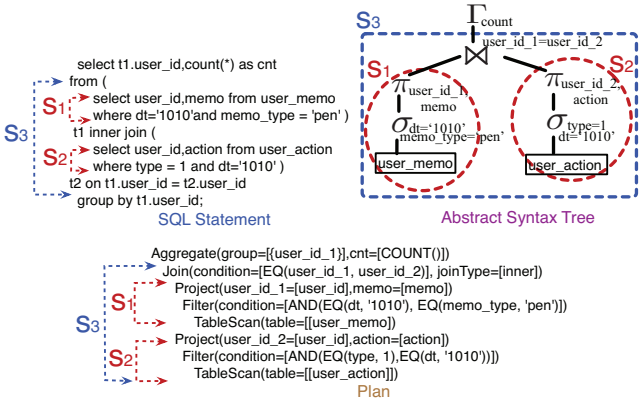


Fig. 2. An example of a query and its subqueries

into hidden representations. We propose a neural network model for cost estimation with views (see Section IV).

(4) We define the subquery selection problem as an ILP problem and design a reinforcement learning method `RLView` to obtain a converged solution (see Section V).

(5) Experimental results show that our methods outperform existing approaches by 28.4%, 8.8% and 31.7% on three real-world datasets (see Section VI).

II. Preliminaries

A. Subquery and Cost

Subquery. A SQL query can be parsed into a syntax tree as shown in Figure 2. We call each subtree a subquery. In the example, we extract three subqueries (e.g., s_1, s_2 and s_3). Each query can be transformed to a logical plan, and each subquery corresponds to a logical subplan. For simplicity, we interchangeably use logical plan and query (subplan and subquery) in the rest of this paper.

Cost. For a query (subquery), we consider its computation cost to answer the query (subquery), such as CPU usage and memory usage, and quantify them together by some pricing strategies (e.g., pricing from cloud vendors). Therefore, the cost of a query (subquery) is defined as follows.

Definition 1 (Cost): Given a query q (subquery s), we use $\mathcal{A}_\beta(q)$ and $\mathcal{A}_\gamma(q)$ ($\mathcal{A}_\beta(s)$ and $\mathcal{A}_\gamma(s)$) to denote the fees of CPU usage and memory usage, respectively. For simplicity, we regard the cost as the summation of these fees, denoted as $\mathcal{A}_{\beta,\gamma}(q) = \mathcal{A}_\beta(q) + \mathcal{A}_\gamma(q)$ ($\mathcal{A}_{\beta,\gamma}(s) = \mathcal{A}_\beta(s) + \mathcal{A}_\gamma(s)$).

B. Materialized View

Overhead of a materialized view. If a subquery is shared by many queries in a query workload, we can materialize a view for this subquery to avoid redundant computations. The overhead of materializing a view for a subquery includes space overhead of the materialized results and the computation cost for this subquery. Therefore, we first define the space overhead and then define the total overhead.

Definition 2 (Space Overhead): Given a materialized view v_s built on subquery s, the byte size of the materialized view is denoted as $u_{sto}(v_s)$. If the fee of storing one byte is α, then the overhead of storing v_s is $\mathcal{A}_\alpha(v_s) = \alpha \cdot u_{sto}(v_s)$.

Definition 3 (Total Overhead): The overhead of a materialized view v_s built on the subquery s is the summation of the space overhead of v_s and the cost of s, which is denoted as $\mathcal{O}_{v_s} = \mathcal{A}_\alpha(v_s) + \mathcal{A}_{\beta,\gamma}(s)$.

Benefit of a materialized view. Using a materialized view to answer a query has a significant benefit, because we can directly get the results of this subquery from the view and avoid re-executing the subquery. The benefit can be calculated by the difference of query cost with/without using the materialized view, which is defined as below.

Definition 4 (Benefit): Given a query q and a materialized view v_s, the cost of executing q is $\mathcal{A}_{\beta,\gamma}(q)$, the cost of executing q using v_s is $\mathcal{A}_{\beta,\gamma}(q|v_s)$, and the benefit is $\mathcal{B}_{q,v_s} = \mathcal{A}_{\beta,\gamma}(q) - \mathcal{A}_{\beta,\gamma}(q|v_s)$.

Benefit of multiple materialized views. Given a set V_S of candidate views and a query q, we want to use the view set to answer query q. However, these views in V_S may not be simultaneously used. For example, if two subqueries have overlaps, we call them *overlapping subqueries* and they cannot be used to answer a query together. For instance, s_3 has overlap with s_1 and s_2 in Figure 2, so we cannot use the views of s_1 or s_2 if we use the view of s_3. Formally, we define *overlapping subqueries* as follows.

Definition 5 (Overlapping Subqueries): Given two subqueries s_i and s_j, s_i and s_j are *overlapping subqueries* if and only if their plan trees have common subtrees.

Given a set V_S of views, let V_S^q denote a subset of V_S and there is no overlapping subquery in V_S^q. We can compute the total benefit of V_S^q by $\mathcal{B}_{q,V_S^q} = \sum_{v_s \in V_S^q} \mathcal{B}_{q,v_s}$. To fully utilize V_S to answer query q, we want to find a *maximal subset* of V_S^q with *the largest benefit* that has no overlapping subqueries.

Utility of multiple materialized views. Given a query workload Q and a set of materialized views V_S, we need to compute the utility of using V_S for Q. The utility is computed by the total benefit of using V_S for Q minus the total overhead of building V_S. Thus, we define U_{Q,V_S} as follows.

Definition 6 (Utility): Suppose we build a set of materialized views V_S for the query workload $Q = \{q_1, q_2, \cdots, q_n\}$. We denote the maximal view subset for the query q as $V_S^q \subseteq V_S$. The utility is $U_{Q,V_S} = \sum_{q \in Q} \sum_{v_s \in V_S^q} \mathcal{B}_{q,v_s} - \sum_{v_s \in V_S} \mathcal{O}_{v_s}$, where $\sum_{v_s \in V_S} \mathcal{O}_{v_s}$ is the total overhead of building V_S and $\sum_{q \in Q} \sum_{v_s \in V_S^q} \mathcal{B}_{q,v_s}$ is the total benefit of using V_S.

Each materialized view is built on an associated subquery, and thus we can estimate its overhead (e.g., query cost and cardinality) with some existing methods [29], [36]. However, it's not realistic to obtain the benefit $\mathcal{B}_{q,v_s}$ due to the cost

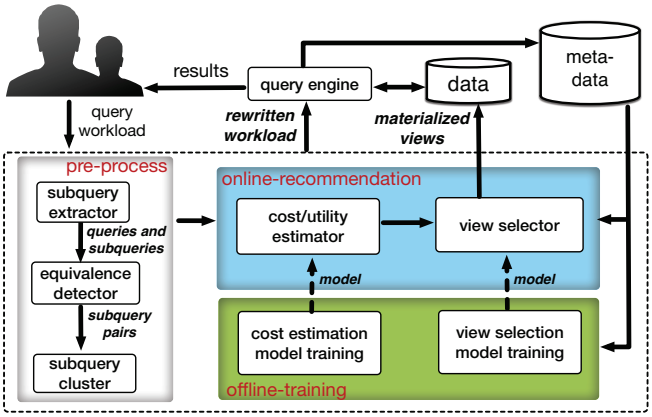


Fig. 3. The overview of system framework

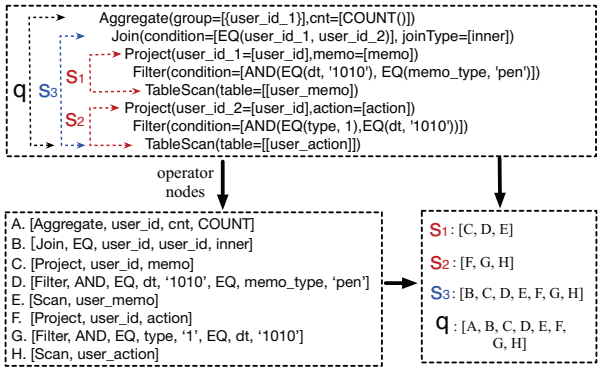


Fig. 4. Feature Extraction from Plans

$\mathcal{A}_{\beta,\gamma}(q|v_s)$ cannot be estimated directly. Thus we propose a deep learning mode to evaluate the cost $\mathcal{A}_{\beta,\gamma}(q|v_s)$.

C. Materialized View Selection

Given a query workload Q, we aim to automatically select the optimal subqueries to materialize. That is, the *Materialized View Selection* (MVS) problem can be modeled as an optimization problem of maximizing the utility. Specifically, the problem contains two optimization objects: the first is to select optimal subqueries to generate materialized views and the second is to select optimal materialized views for each query under the constraint of *overlapping subqueries*. Formally, we define the MVS problem as follows.

Definition 7 (MVS problem): Given a query workload Q and a set of its possible subqueries S_Q, we select subqueries $S \subseteq S_Q$ to build materialized views V_S and then select views $V_S^q \subseteq V_S$ for each query $q \in Q$ as follows:

$$\arg\max_{S \in S_Q, V_S^q \in V_S} \sum_{q \in Q} \sum_{v_s \in V_S^q} \mathcal{B}_{q,v_s} - \sum_{v_s \in V_S} \mathcal{O}_{v_s}$$

$s.t.$ s_i, s_j are not overlapping, $\forall q \in Q, i, j \in [1, |V_S^q|]$.

An intuitive method to solve the MVS problem is to compute the utility for all possible subsets of S_Q. However, this method is not realistic when either S_Q or Q are large. Thus we propose a reinforcement learning model.

III. System Overview

In this section, we present our system. As shown in Figure 3, our system contains three parts: pre-process, online-recommendation and offline-training. First, given a query workload, we extract candidate subqueries from the query workload in the pre-process part. Next, we select some "highly beneficial" subqueries to generate materialized views based on two models in the online-recommendation part, where the two models are trained in the offline-training part and the training data is collected from the metadata database. Later, we rewrite queries using the views. At last, we execute the rewritten workload. Next, we discuss the details of the three parts.

Pre-process. This part contains three components: *subquery extractor*, *equivalence detector* and *subquery cluster*. At first, we use *subquery extractor* to parse query statements into logical plans with the parsing tool in the query engine. For each query, we consider subplans, starting with *Aggregate*, *Join* or *Project*, as subqueries. Later, we use *equivalent detector* to infer whether two subqueries are equivalent and collect equivalent subquery pairs. In particular, we use the method EQUITAS [45] to detect equivalent subqueries. Then, we cluster equivalent subqueries into a *subquery cluster*. For each cluster, we select the subquery with the least overhead as the candidate subquery. Thus, we build a materialized view based on the selected candidate subquery and reuse the materialized result for other queries. To check whether two queries are overlapping queries, we maintain a list of cluster IDs for each query q, which keeps all cluster IDs such that there exists a query s in the cluster and s is a subquery of q. Then two queries are overlapping queries if their lists have overlap.

Online-recommendation. This part contains two components: *cost/utility estimator* and *view selector*.

(1) In *cost/utility estimator*, we aim to compute the utility U_{Q,V_S}. The challenge is to evaluate the cost $\mathcal{A}_{\beta,\gamma}(q|v_s)$. To solve the issue, we design a cost estimation model to estimate $\mathcal{A}_{\beta,\gamma}(q|v_s)$. In particular, we implement the cost estimation model based on the deep learning technology. After that, we can compute the benefit $\mathcal{B}_{q,V_S^q}$ and the utility U_{Q,V_S}.

(2) In *view selector*, we aim to solve the MVS problem. Specifically, we design a reinforcement learning based view selection model to recommend optimal subqueries based on the computed utility for generating materialized views.

Offline-training. We offline train the cost estimation model and the view selection model. Training data are stored in the metadata database. In particular, query plans, view plans, table information (i.e., table size) and the actual cost of rewritten queries can be collected from the query engine and then stored in the metadata database. (1) Cost estimation. For a query and a view, we collect the query plan, the view plan and the associated table information as the features, and collect the actual cost of the rewritten query as the target to train the cost estimation model. (2) View selection. We first compute the actual benefit for a query and a view by the actual cost of the rewritten query, and then use the actual benefit to compute intermediate rewards between different states for fine-tuning the reinforcement learning model.

IV. Utility Estimation

In this section, we propose to estimate the cost $\mathcal{A}_{\beta,\gamma}(q|v_s)$ for a query q and a materialized view v_s. We first describe different features associated with the cost (see Section IV-A) and then introduce our model (see Section IV-B).

A. Feature Extraction

The useful features for evaluating the cost $\mathcal{A}_{\beta,\gamma}(q|v_s)$ contain two parts: query/view plans and associated tables.

Plans. Given a query q and a materialized view v_s, we first extract features from the plans of the query q and the view v_s. In particular, the plan of v_s corresponds to the plan of the subquery s. As shown in Figure 4, each plan is a sequence and each element in the sequence corresponds to an operator (e.g., *Scan, Project, Filter and Join*). For each operator, we extract its associated attributes. Thus, each operator can be regarded as an attribute sequence. In particular, we use prefix notation [1] to represent condition attributes. For example, the *Filter* operator of s_1 is represented as the sequence *[Filter, AND, EQ, dt, '1010', EQ, memo_type, 'pen']*, where *Filter* is the operator type and the rest is the prefix notation of associated attributes. Thus, the plan of q or s can be represented as a two-dimensional sequence. First, each operator contains multiple predicates, and we model each operator as the first layer sequence. Second, each query contains multiple operators, and we model each plan as the second layer sequence. For example, s_1 is the sequence $[C, D, E]$, and C, D and E respectively correspond to three sequences.

Associated Tables. Different tables indicate different query results and thus lead to different cost. Therefore, we consider the metadata information of associated tables for q and v_s, which is collected from metadata database. In particular, the metadata includes two parts: the schema of input tables (e.g., table names, column names and column types) and the statistics of input tables (e.g., the number of tables, the number of columns and the size of records).

In addition, the collected features can be split into *numerical features* and *non-numerical features*. In particular, *numerical features* indicate the statistics of input tables while *non-numerical features* include two kinds of features: plan sequence and table schema.

B. Wide-Deep Model

1) Model Architecture: We adopt a *Wide-Deep* model to predict the cost $\mathcal{A}_{\beta,\gamma}(q|v_s)$. The model contains two parts: a wide linear model and a deep model. The wide model can capture the *linear* relation between input features and results; while the deep model can better capture the *non-linear* relation. As shown in Figure 5, we first collect tables, the query and view plans as input features, which are divided into *numerical features* and *non-numerical features*. We then use the wide model and the deep model to estimate the cost.

In the wide part, we only need to consider *numerical features*. The reason is that *non-numerical features* are discrete

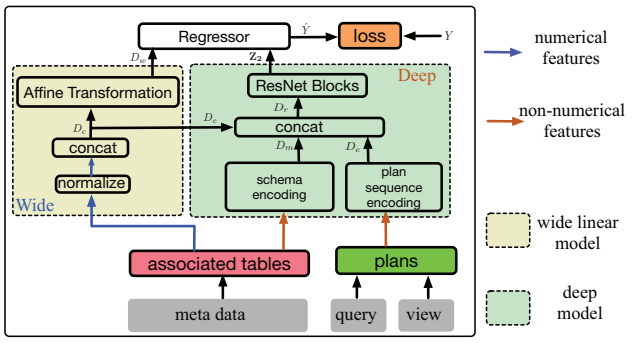


Fig. 5. Wide-Deep Model

and the change of their values are non-linear. Firstly, to eliminate the difference in the magnitude of feature values, we normalize all *numerical features* $f_{c_1}, f_{c_2}, \cdots$, and then concatenate them into a fixed-length vector $D_c = concat(\frac{f_{c_1}-\mu_{c_1}}{\sigma_{c_1}}, \frac{f_{c_2}-\mu_{c_2}}{\sigma_{c_2}}, \cdots)$, where μ_{c_i} and σ_{c_i} are mean value and standard deviation value of the feature f_{c_i}. At last, we use an affine transformation[2] model $\mathcal{M}_w$ to linearly transform D_c into D_w, which is denoted as $D_w = \mathcal{M}_w(D_c)$.

In the deep part, to make it possible to use *non-numerical features*, we first design the schema encoding model ($\mathcal{M}_m$) and the plan sequence encoding model ($\mathcal{M}_e$) to convert table schemas and query/view plans into fixed-length vectors D_m and D_e, respectively. Later, for considering *numerical features*, we concatenate the vector D_c with D_m and D_e, and denote the result as $D_r = concat(D_c, D_m, D_e)$. At last, we implement the deep model ($\mathcal{M}_d$) with two deep residual networks (ResNet [13]) blocks, which is efficient in many real-world applications[17], [39]. In our settings, each ResNet block contains two fully connected layers and two activation layers. We successively calculate their outputs as follows.

$$\mathbf{Z_1} = D_r \oplus ReLU(FC_2(ReLU(FC_1(D_r))))$$
$$\mathbf{Z_2} = \mathbf{Z_1} \oplus ReLU(FC_4(ReLU(FC_3(\mathbf{Z_1}))))$$

where $\mathbf{Z_1}$ and $\mathbf{Z_2}$ are output vectors of the first and second ResNet block respectively, $ReLU$ is the activation function, FC_i represents different fully connected layers, and $\oplus$ denotes the element-wise plus.

Finally, we use a regressor ($\mathcal{M}_r$), which is a two-layer fully connected network and an activation layer, to merge outputs of the wide and deep parts, and get the final predicted cost:

$$\hat{Y} = FC_6(ReLU(FC_5(D_w, \mathbf{Z_2})))$$

2) Encoding Features: We explain the details of the two encoding models $\mathcal{M}_e$ and $\mathcal{M}_m$, which are respectively designed to encode two kinds of *non-numerical features*: query/view plans and input table schemas.

Keyword Embedding. All *non-numerical features* include *keywords* (e.g., table names, column names and operator types). To embed each keyword k_i, we first use a one-hot encoding to transform it into a fixed-length vector $O_{k_i} \in \mathbb{R}^{n_k}$, where the value of one particular dimension is 1 while the rest are 0, and n_k is the number of keywords. However, the one-hot code is too sparse if n_k is too big. To address this issue, we

[1] https://en.wikipedia.org/wiki/Polish_notation

[2] https://en.wikipedia.org/wiki/Affine_transformation

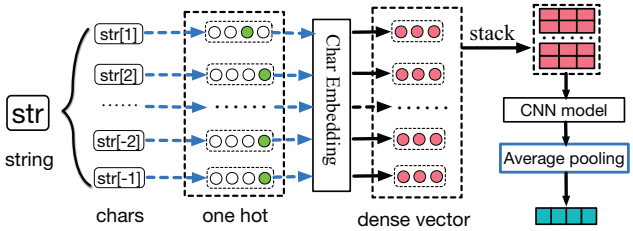
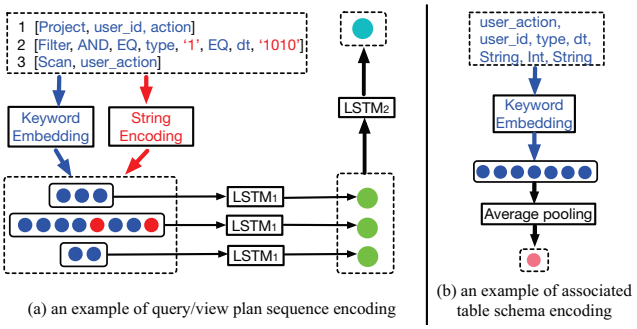


Fig. 6. The overview of String Encoding



(a) an example of query/view plan sequence encoding

(b) an example of associated table schema encoding

Fig. 7. Non-numerical feature encoding examples

use the *Keyword Embedding* model to embed O_{k_i} into a dense vector by the formula $D_{k_i}^{\mathsf{T}} = O_{k_i}^{\mathsf{T}} \mathbf{W_k}$, where $\mathbf{W_k} \in \mathbb{R}^{n_k \times n_d}$ is the matrix parameter of *Keyword Embedding* and $n_d \ll n_k$. We share the *Keyword Embedding* matrix for the two kinds of features as their keywords belong to the same database.

String Encoding. The query/view plans contain *strings*, whose number is not fixed and thus cannot be encoded like *keywords*. To address this issue, as shown in Figure 8, we design a novel model *String Encoding* to encode each string into a fixed-length vector. We first regard each string str as a char array and each char $str[i]$ can be represented as a 128-dimensional one-hot code $O_{str[i]} \in \mathbb{R}^{128}$. Then we use the *Char Embedding* model to transform $O_{str[i]}$ into a dense vector $D_{str[i]}^{\mathsf{T}} = O_{str[i]}^{\mathsf{T}} \mathbf{W_c} \in \mathbb{R}^{n_d}$, where $\mathbf{W_c} \in \mathbb{R}^{128 \times n_d}$ is the matrix parameter of *Char Embedding*. Therefore, we get the dense vector sequence $D_{str} = [D_{str[1]}, D_{str[2]}, \cdots]$. Next, we stack D_{str} into a matrix M_{str} and then use a CNN (Convolutional Neural Network) model to convert the matrix into another matrix $M'_{str} = CNN(M_{str})$. The reason of using CNN is that CNN is able to capture local features and guarantee translation invariance. The CNN model comprises two connected convolution blocks, and each convolution block consists of three layers: $Conv2d \rightarrow BatchNorm2d \rightarrow ReLU$, where $Conv2d$ is the convolution layer, $BatchNorm2d$ is the BatchNorm layer and $ReLU$ is the activation layer. In our setting, the kernel size of $Conv2d$ is 3×1. At last, we use the average pooling technique to get the final fixed-length vector $D_s \in \mathbb{R}^{n_d}$, where $D_s[i] = Avg(M'_{str}[:, i])$ and $Avg(\cdot)$ represents the average function.

Encoding Query/View Plan. As mentioned before, each query/view plan is a two-dimensional sequence, so we utilize the LSTM (Long Short-Term Memory [16]) model, which is popularly applied in natural language processing[7], [41], to capture the sequence structure of plans. Formmaly, each plan can be denoted as a sequence $f_e = [f_{op}^1, f_{op}^2, \cdots]$, where $f_{op}^i = [f_{ks}^{i,1}, f_{ks}^{i,2}, \cdots]$ is also a sequence and $f_{ks}^{i,j}$ is a keyword or string. Thus, as shown in the example of Figure 7 (a), we first use *Keyword Embedding* or *String Encoding* to transform $f_{ks}^{i,j}$ into a code $D_{ks}^{i,j}$ and thus get the sequence $[D_{ks}^{i,1}, \cdots]$ for each operator f_{op}^i. Then, we use an LSTM model $LSTM_1$ to transform each sequence $[D_{ks}^{i,1}, \cdots]$ into a fixed-length vector $D_{op}^i = LSTM_1([D_{ks}^{i,1}, \cdots])$. After that, we use another LSTM model $LSTM_2$ to transform the sequence $[D_{op}^1, D_{op}^2, \cdots]$ into the final vector $D_e = LSTM_2([D_{op}^1, \cdots])$. Intuitively, $LSTM_1$ can capture the local sequence structure for each element of the plan sequence while $LSTM_2$ can capture the sequence structure for the global plan sequence.

Algorithm 1: Wide-Deep Model Training

Input: training features ($X = \{(q_1, v_1, t_1), \cdots\}$), training targets ($Y = \{\mathcal{A}_{\beta,\gamma}(q_1|v_1), \cdots\}$), table schema encoding part $\mathcal{M}_m$, query/view plan encoding part $\mathcal{M}_e$, wide part $\mathcal{M}_w$, deep part $\mathcal{M}_d$, regressor part $\mathcal{M}_r$, learning rate lr, training epochs I, batch size b_s.
Output: parameters $\theta_m, \theta_e, \theta_w, \theta_d, \theta_r$ for $\mathcal{M}_m$, $\mathcal{M}_e$, $\mathcal{M}_w$, $\mathcal{M}_d$ and $\mathcal{M}_r$

1 extract numerical features X^m from $\{t_1, t_2, \cdots\}$;
2 extract non-numerical features X^n from $\{(q_1, v_1, t_1), \cdots\}$;
3 **for** $j \leftarrow 1 \cdots I$ **do**
4 training iterations $I' = \lfloor \frac{|Y|}{b_s} \rfloor$;
5 $shuffle(X^m, X^n, Y)$;
6 **for** $i \leftarrow 1 \cdots I'$ **do**
7 $X_i^m, X_i^n, Y_i \leftarrow$ sample b_s data from X^m, X^n, Y;
8 Normalize and concatenate X_i^m into D_c;
9 $D_m = \mathcal{M}_m$(input table schema in X_i^n);
10 $D_e = \mathcal{M}_e$(query/view plans in X_i^n);
11 $D_r = concat(D_c, D_m, D_e)$;
12 $\hat{Y}_i \leftarrow \mathcal{M}_r(\mathcal{M}_w(D_c), \mathcal{M}_d(D_r))$;
13 $loss_i \leftarrow MSE(Y_i, \hat{Y}_i)$;
14 $\theta_m, \theta_e, \theta_w, \theta_d, \theta_r \leftarrow AdamOpt(loss_i, lr)$;
15 return $\theta_m, \theta_e, \theta_w, \theta_d, \theta_r$;

Encoding Table Schema. We regard the input table schema feature as a set of keywords and formally denote it as $\{k_1, k_2, \cdots\}$. Figure 7 (b) shows an example of encoding the set. We use *Keyword Embedding* to transform each keyword k_i into a dense vector D_{k_i} and thus get a dense vector array $[D_{k_1}, \cdots]$. We then use the average pooling technique to transform the vector array into a vector $D_m \in \mathbb{R}^{n_d}$, where $D_m[i] = Avg([D_{k_1}[i], D_{k_2}[i], \cdots])$.

3) Model Training: The model training process is shown in Algorithm 1. The model contains five parts: the table schema encoding part $\mathcal{M}_m$, the query/view plan encoding part $\mathcal{M}_e$, the wide part $\mathcal{M}_w$, the deep part $\mathcal{M}_d$ and the regressor part $\mathcal{M}_r$. We aim to learn parameters of the five parts by training data, which is composed of input features and output targets. In particular, each input feature includes a query q, a view v and the associated table information t while its corresponding output target is the cost $\mathcal{A}_{\beta,\gamma}(q|v)$. At first, we respectively extract numerical and non-numerical features from input features (lines 1-2). Next, we iteratively train the model with the given epochs I. For each epoch, we first compute the training iterations I' based on given batch size b_s, then shuffle all training data X, Y (lines 4-5). In each iteration, we sample b_s data (line 7) for batch training. After that, we normalize and concatenate *numerical features* (line 8). As for *non-numerical features*, we use $\mathcal{M}_m$ and $\mathcal{M}_e$ to

encode them into vectors (lines 9-11). Later, we use $\mathcal{M}_w$, $\mathcal{M}_d$ and $\mathcal{M}_r$ to compute the estimated cost $\hat{Y}_i$ (line 12). Finally, we use MSE (Mean Squared Error) metric as the loss function, which is defined as $MSE(Y_i, \hat{Y}_i) = \frac{1}{|Y_i|}\sum_{y \in Y_i}(y - \hat{y})^2$. The parameters of all parts are jointly optimized by Adam [23] with the given learning rate lr (lines 13-14).

V. Materialized View Selection

We study the view selection problem. We first model it as an ILP (Integer Linear Programming) problem. Since it is prohibitively expensive to get actual optimal solutions for a large number of queries and views, we propose an iterative optimization method to get approximate solutions in Section V-A. However, this method does not converge to a global optimal solution. To address this issue, we model ILP as an MDP (Markov Decision Process) and use the deep reinforcement learning technique to solve it in Section V-B.

A. ILP Problem and Iterative Optimization

1) Rewriting Problem: Given a query workload Q, assume its candidate subqueries are S_Q. The MVS problem aims to (i) select the optimal subqueries $S \subseteq S_Q$ to materialize and (ii) select the optimal view set $V_S^q \subseteq V_S$ for each query q under the constraint of *overlapping subqueries*, which can be regarded as an ILP problem. We propose a method to address the two subproblems together. Let z_j be a 0-1 variable indicating whether the subquery $s_j \in S_Q$ is selected to materialize or not, x_{jk} be a 0-1 variable indicating whether s_j and s_k are overlapping or not, and y_{ij} be a 0-1 variable indicating whether the query $q_i \in Q$ uses the materialized view v_{s_j} or not. The ILP problem can be defined as follows.

$$\underset{Z,Y}{\text{argmax}} \sum_{i\in[1,|Q|]}\sum_{j\in[1,|S_Q|]} y_{ij}\mathcal{B}_{q_i,v_{s_j}} - \sum_{j\in[1,|S_Q|]} z_j\mathcal{O}_{v_{s_j}}$$

$$s.t.\ y_{ij} + \sum_{k\neq j} x_{jk}y_{ik} \leq 1, \quad \forall i \in [1,|Q|], j \in [1,|S_Q|], \quad (1)$$

$$y_{ij} \leq z_j, \quad \forall i \in [1,|Q|], j \in [1,|S_Q|] \quad (2)$$

where Formula 1 guarantees the constraint of *overlapping subqueries* and Formula 2 guarantees that V_S^q is a subset of V_S. Notably, each x_{jk} is constant and we aim to resolve Z ($\{z_j\}$) for selecting view v_i to materialize and Y ($\{y_{ij}\}$) for using v_j to rewrite q_i.

2) Optimizing Iteratively: The above ILP problem becomes intractable for very large workloads due to the large number of integer variables. One possible method is to separately optimize Z and Y. That is, we would set Y as a constant when optimizing Z, and set Z as a constant when optimizing Y. As shown in the function `IterView`, the above optimization process would be iterated with the given number of iterations.

Initializing Z and Y. For each candidate subquery s_j, we first randomly initialize its associated variable z_j with 0 or 1 and record the overhead $\mathcal{O}_{v_{s_j}}$ of its associated materialized view (if $z_j = 1$) (line 4), and then record the maximum benefit of materializing s_j (line 5). Later, we initialize y_{ij} for each query q_i with 0 or 1. Notably, y_{ij} would be set 0 if the constraints

Function `IterView`

Input: query workflow Q, overhead array $\{\mathcal{O}_{v_{s_j}}\}$, benefit array $\{\mathcal{B}_{q_i,v_{s_j}}\}$, overlapping array $X = \{x_{jk}\}$, iterations n
Output: optimization results Z and Y

1 $Z = \{z_j\}$, $Y = \{y_{ij}\}$, $\mathbb{B}^{max} = \emptyset$, $\mathbb{B}^{cur} = \emptyset$, $\mathcal{O}^{cur} = 0$;
2 /*randomly initialize Z and Y */
3 **for** $j \in [1, |Z|]$ **do**
4 $z_j = \text{random}(0,1)$, $\mathcal{O}^{cur} += z_j\mathcal{O}_{v_{s_j}}$;
5 $\mathbb{B}^{max}[j] = \sum_{q_i \in Q}\mathcal{B}_{q_i,v_{s_j}}$;
6 **for** $i \in [1, |Q|]$ **do**
7 **if** $z_j = 1 \wedge \mathcal{B}_{q_i,v_{s_j}} > 0 \wedge \sum_{k\neq j} x_{jk}y_{ik} = 0$ **then**
 $y_{ij} = \text{random}(0,1)$;
8 **else** $y_{ij} = 0$;
9 $\mathbb{B}^{cur}[j] = \sum_{q_i \in Q} y_{ij} \cdot \mathcal{B}_{q_i,v_{s_j}}$;
10 **for** $iter \in [1, n]$ **do**
11 generate a random probability threshold $\tau \in [0,1]$;
12 $Z, \mathcal{O}^{cur} \leftarrow$ `Z-Opt`$(Z, \{\mathcal{O}_{v_{s_j}}\}, \mathcal{O}^{cur}, \mathbb{B}^{max}, \mathbb{B}^{cur}, \tau)$;
13 $Y, \mathbb{B}^{cur} \leftarrow$ `Y-Opt`$(Y, \mathbb{B}^{cur}, \{\mathcal{B}_{q_i,v_{s_j}}\}, X, Z, Q)$;
14 return Z, Y;

Function `Z-Opt` $(Z, \{\mathcal{O}_{v_{s_j}}\}, \mathcal{O}^{cur}, \mathbb{B}^{max}, \mathbb{B}^{cur}, \tau)$

Output: Z, $\mathcal{O}^{cur}$

$B^{cur} = \sum_{k=1}^{|Z|}\mathbb{B}^{cur}[k]$, $B^{max} = \sum_{k=1}^{|Z|}\mathbb{B}^{max}[k]$;
$\mathcal{O}^{max} = \sum_{j=1}^{|Z|}\mathcal{O}_{v_{s_j}}$;
for $j \in [1, |Z|]$ **do**
 $B_j^{max} = \mathbb{B}^{max}[j]$, $B_j^{cur} = \mathbb{B}^{cur}[j]$;
 computing p_j^{flip} according to Equation 3;
 if $p_j^{flip} \geq \tau$ **then**
 $z_j = 1 - z_j$;
 if $z_j > 0$ **then** $\mathcal{O}^{cur} += \mathcal{O}_{v_{s_j}}$;**else** $\mathcal{O}^{cur} -= \mathcal{O}_{v_{s_j}}$;
return Z, $\mathcal{O}^{cur}$;

Function `Y-Opt` $(Y, \mathbb{B}^{cur}, \{\mathcal{B}_{q_i,v_{s_j}}\}, X, Z, Q)$

Output: Y, $\mathbb{B}^{cur}$

for $i \in [1, |Q|]$ **do**
 $Y[i] \leftarrow \{y_{i1}, \cdots, y_{i|Z|}\}$, $O = \sum_{z_j \in Z} y_{ij}\mathcal{B}_{q_i,v_{s_j}}$, $\mathcal{J} = \emptyset$;
 $\mathcal{J} \leftarrow \{y_{ij} \leq z_j | z_j \in Z\}$;
 $\mathcal{J} \leftarrow \{y_{ij} + \sum_{k=1,k\neq j}^{|Z|} x_{jk} \cdot y_{ik} \leq 1 | z_j \in Z\}$;
 $Y[i] \leftarrow$ ILPSolver$(O, \mathcal{J}, Y[i])$;
for $k \in [1, |Z|]$ **do** $\mathbb{B}^{cur}[k] = \sum_{i=1}^{|Q|} y_{ij} \cdot \mathcal{B}_{q_i,v_{s_j}}$;
return $Y, \mathbb{B}^{cur}$;

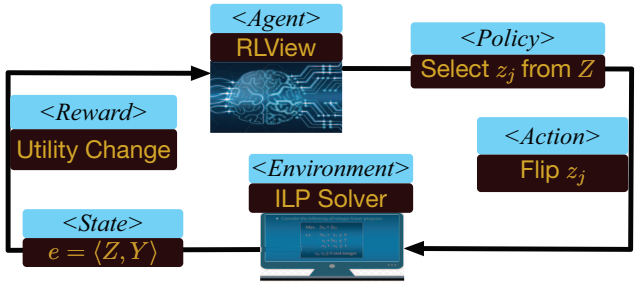
in the ILP problem are not satisfied (lines 6-8). We record the actual current benefit of materializing s_j by considering y_{ij} (line 9). At last, we iteratively optimize Z and Y with the functions `Z-Opt` and `Y-Opt` (lines 10-13).

Optimizing Z. As shown in `Z-Opt`, for each variable z_j, we compute its flipping probability and decide whether to change z_j based on the probability. In particular, we first compute the current benefit summation B^{cur}, the current overhead summation $\mathcal{O}^{cur}$, the maximum benefit summation B^{max}, and the maximum overhead summation $\mathcal{O}^{max}$ of all materialized views. Later, the flipping probability p_j^{flip} for z_j can be computed as follows:

$$p_j^{flip} = p_j^{overhead} \cdot p_j^{benefit} \quad (3)$$

$$p_j^{overhead} = \begin{cases} \mathcal{O}_{v_{s_j}}/\mathcal{O}^{cur} & if\ z_j = 1 \\ 1 - \mathcal{O}^{cur}/\mathcal{O}^{max} & otherwise \end{cases}$$

$$p_j^{benefit} = \begin{cases} 1 - B_j^{cur}/B^{cur} & if\ z_j = 1 \\ \frac{B_j^{max}/\mathcal{O}_{v_{s_j}}}{B^{max}/\mathcal{O}^{max}} & otherwise \end{cases}$$



$Z = \{z_j\}$:z_j is a 0/1 variable indicating whether to materialize the subquery s_j
$Y = \{y_{ij}\}$:y_{ij} is a 0/1 variable indicating whether to use the view v_{s_j} for the query q_i

Fig. 8. The MDP Framework of the ILP Problem

where $B_j^{max} = \mathbb{B}^{max}[j]$ means the maximum potential benefit by setting z_j as 1, $B_j^{cur} = \mathbb{B}^{cur}[j]$ represents the current benefit of using the materialized view v_{s_j}, and $\mathcal{O}_{v_{s_j}}$ means the overhead of v_{s_j}. Therefore, the variable of a 1-labeled materialized view ($z_j = 1$) would be flipped if the view causes more overhead (the first case of $p_j^{overhead}$) and provides less benefit (the first case of $p_j^{benefit}$), and the variable of a 0-labeled materialized view ($z_j = 0$) would be flipped if the view has less overhead (the second case of $p_j^{overhead}$) and is expected to bring more benefit (the second case of $p_j^{benefit}$).

Optimizing Y. In the function `Y-Opt`, we regard Z as a constant, so the materialized view overhead is fixed. Specifically, we optimize $Y[i] = \{y_{i1}, \cdots, y_{i|Z|}\}$ for each query q_i, which is a local ILP optimization problem as follows.

$$\underset{Y[i]}{\text{argmax}} \sum_{j=1}^{|Z|} y_{ij} \mathcal{B}_{q_i, v_{s_j}}$$

$$s.t. y_{ij} + \sum_{k=1, k \neq j}^{|Z|} x_{jk} \cdot y_{ik} \leq 1 \ \& \ y_{ij} \leq z_j, \forall j \in [1, |Z|].$$

where $\mathcal{B}_{q_i, v_{s_j}}$, z_j and x_{jk} are constants, and $Y[i]$ needs to be optimized. Therefore, we can solve the problem efficiently by existing ILP solvers, such as PulP[3] and Gurobi[4].

B. RL based Method

`IterView` has no memory ability and cannot converge to a global optimal solution. The reason is that each iteration in the function can only get a local optimization solution and different iterations cannot share feedback from optimization process, which leads to repeated oscillation of optimization results. To solve the problem, we use the Reinforcement Learning (RL) technique[38], [27], [30] to design an algorithm, which is called `RLView`. In particular, we first explain the optimization process as an MDP (Markov Decision Process) in Section V-B1, and then propose the algorithm in Section V-B2.

1) Markov Decision Process: The reinforcement learning model is proposed to find the best policy for a system to get the most cumulative reward from environment, and it is usually used for decision-making in contexts where a system learns by trial-and-error from rewards and punishment, which is called Markov Decision Process (MDP). In particular, a Markov

[3]https://pythonhosted.org/PuLP/index.html
[4]http://www.gurobi.com/documentation/

Algorithm 2: `RLView`

Input: workload Q, overhead array $\{\mathcal{O}_{v_{s_j}}\}$, benefit array $\{\mathcal{B}_{q_i, v_{s_j}}\}$, overlapping array $X = \{x_{jk}\}$, initial iterations n_1, the number of RL epochs n_2, memory size n_m, reward decay rate γ
Output: Z, Y

/*get optimal results by function `IterView` */
$Z_0, Y_0 \leftarrow$IterView$(Q, \{\mathcal{O}_{v_{s_j}}\}, \{\mathcal{B}_{q_i, v_{s_j}}\}, X, n_1)$;
/*update iteratively based on DQN*/
experience replay memory $\mathcal{M} \leftarrow \emptyset$;
initializing parameters θ of the DQN $\mu(e|\theta)$;
for $ep \in [1, n_2]$ **do**
  $t = 0$, $e_0 = \langle Z_0, Y_0 \rangle$;
  do
    $R_t = \sum_i \sum_j y_{ij}^t \mathcal{B}_{q_i, v_{s_j}} - \sum_j z_j^t \mathcal{O}_{v_{s_j}}$;
    $a_t = argmax_i\{\mathbb{Q}(e_t)[i]\}$, $Z_{t+1}[a_t] = 1 - Z_t[a_t]$;
    $Y_{t+1}, \_ =$`Y-Opt`$(Y_t, \_, \{\mathcal{B}_{q_i, v_{s_j}}\}, X, Z_{t+1}, Q)$;
    $e_{t+1} = \langle Z_{t+1}, Y_{t+1} \rangle$;
    $R_{t+1} = \sum_i \sum_j y_{ij}^{t+1} \mathcal{B}_{q_i, v_{s_j}} - \sum_j z_j^{t+1} \mathcal{O}_{v_{s_j}}$;
    $r_t = R_{t+1} - R_t$, $\mathcal{M} \leftarrow \langle e_t, a_t, r_t, e_{t+1} \rangle$;
    $e_t = e_{t+1}$, $t = t + 1$;
    /*fine-tuning the DQN model*/;
    if $|\mathcal{M}| \geq m$ **then** $\theta \leftarrow$ `DQN`$(\mathcal{M}, \theta, \gamma)$;
  while $t < |Z| \vee r_t > 0$;
return Z, Y;

Function `DQN-offline`

Input: $\mathcal{M}$,θ,γ
Output: θ

$\langle e_t, a_t, r_t, e_{t+1} \rangle \leftarrow$ sample data from $\mathcal{M}$;
calculate Q-value for state e_t: $Q(e_t, a_t) = \mu(e_t, a_t|\theta)$;
calculate Q-values for state e_{t+1}:
$\mathbb{Q}(e_{t+1}) = [\mu(e_{t+1}, a_1|\theta), \cdots, \mu(e_{t+1}, a_n|\theta)]$;
apply Q-learning and obtain the estimated value:
$Q'(e_t, a_t) = \gamma max_i\{\mathbb{Q}(e_{t+1})[i]\} + r_t$;
use the error $||Q(e_t, a_t) - Q'(e_t, a_t)||^2$ to update θ;

Decision Process is composed of a 4-tuple ($\langle E, A, P_a, R_a \rangle$), where E is a finite set of states, A is a finite set of actions, $P_a(e, e') = Pr(e_{t+1} = e' | e_t = e, a_t = a)$ is the probability that action a in state e at time t will go to state e' at time $t + 1$, and $R_a(e, e')$ is the immediate reward (or expected immediate reward) received after transitioning from state e to state e' due to action a. Our iterative optimization process can be modeled as a Markov Decision Process. Thus, we can use the reinforcement learning technique to solve the optimization problem. As shown in Figure 8, we map the optimization process into MDP as follws. The state is defined as the tuple of Z and Y, which is denoted as $e = \langle Z, Y \rangle$, the policy is to select a variable z_j from Z and the action is to flip (select or unselect) the label of the selected variable z_j. In addition, for each state e, we can compute its associated utility, which is denoted as $U(e)$. Once taking an action, we get new labels for Z and then use an ILP solver to get new labels for Y, so the ILP solver is the environment and we get the new state e'. Finally, the immediate reward $R_a(e, e')$ can be regarded as the utility change by the formula $R_a(e, e') = U(e') - U(e)$.

Therefore, the ILP optimization problem becomes a reinforcement learning problem, whose goal is to learn an optimal policy, which is defined as follows:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k} | e_t = e\}, \forall e \in E, \forall t \geq 0.$$

where $\pi : E \times A \rightarrow [0, 1]$ represents a policy function (given a state, the function would select an action), $\mathbb{E}_{\pi}$ denotes the expected value in the policy π, $\gamma \in [0, 1)$ represents the discount rate, k is the time step and r_{t+k} is the intermediate reward in the time $t + k$.

2) RL-Based Method: We apply Q-learning [43] to solve the problem. Q-learning [43] is a value-based reinforcement learning algorithm, In particular, it maintains a Q-table to record all known state-action-value pair $((e, a), Q(e, a))$, where $Q(e, a)$ is the corresponding Q-value of the state e with the action a. Thus, the algorithm would update Q-values based on interacting with the environment. Once we get the Q-table, we can get the best policy based on the Q-table's guide. Hence, the key is to generate Q-values. At first, we randomly initialize each state's Q-values. Then for each state e, we enumerate all possible actions $\mathbb{A}(e) = \{a_1, \cdots, a_n\}$ and then picks the best one a^* by the formula: $a^* = argmax_{a_i}(Q(e, a_i))$, where a_i means flipping z_i ($z_i = 1 - z_i$), $n = |S_Q|$ is the size of candidate subqueries, and $e_i = \langle Z_i, Y_i \rangle$ denotes the new state that the current state e transfers to after taking action a_i. Next, through interacting with the environment, we update $Q(e, a^*)$ for each state e as follows.

$$Q^{new}(e, a^*) = (1 - \epsilon)Q^{old}(e, a^*) + \epsilon(r + \gamma \max_{a_i \in \mathbb{A}(e^*)} Q(e^*, a_i))$$

where r is the intermediate reward from the current e to the new state e^* with the action a^* and ϵ is the learning rate ($0 < \epsilon \leq 1$). We repeatedly update Q-values according to the above equation until we get convergent Q-values for each state. At last, we get the final Q-table, and the best policy function $\pi^*(e)$ can be obtained as follows.

$$\pi^*(e) = \begin{cases} \underset{a_i \in \mathbb{A}(e)}{\text{argmax}}(r_i + \gamma \max_{a_j \in \mathbb{A}(e_i)} Q(e_i, a_j)), \ \exists r_i > 0 \\ terminal, \ \text{otherwise} \end{cases}$$

DQN. However, the state space and action space are both large. For example, the number of total states has an exponential relationship with the number of candidate subqueries, which could cause the problem of dimensional disaster. To solve the issue, we utilize Deep Q-learning Network(DQN)[30], [42]. Specifically, we train a deep neural network to predict the Q-value $Q(e, a)$ for the state e taking the action a, which is denoted as $Q(e, a) = \mu(e, a|\theta)$. Hence, we can get the Q-value vector $\mathbb{Q}(e) = [Q(e, a_1), \cdots, Q(e, a_n)] \in \mathbb{R}^n$ for the state e. We compare values of all dimensions in $\mathbb{Q}(e)$ and select the dimension corresponding to the maximum value as the next action for e. In particular, we implement the prediction model with four fully connected layers, where the number of neurons in these layers are 16, 64, 16 and 1, respectively. In addition, we use *ReLU* as the activation function for each layer.

Algorithm 2 lists the detail of solving the reinforcement learning problem. At first, to efficiently solve the problem, we need to initialize the state with a descent solution, so we get the initial state $e_0 = \langle Z_0, Y_0 \rangle$ by the function `IterView` and randomly initialize the DQN model (line 2-5). Later, we execute the reinforcement learning process with n_2 epochs, where n_2 is given by users. In each epoch, we iteratively get optimal states based on the initial state e_0. In each iterative step, we use the DQN model to update Z and use the function `Y-Opt` to update Y respectively (lines 10-11). In addition, we use a memory pool $\mathcal{M}$ to store experience replay, where each experience is formed with a tuple $\langle e_t, a_t, r_t, e_{t+1} \rangle$. The reward r_t between e_t and e_{t+1} is defined as the difference between their associated utilities, denoted as $R_{t+1} - R_t$ (lines 12-14). It terminates when r_t is not increased and the number of flipping labels is greater than the size of Z (line 17).

We can offline train and online fine-tune the DQN model with the function `DQN`. For offline training, we store the memory pool $\mathcal{M}$ into the metadata database, and then offline train the network DQN by collecting training dataset from the metadata database. For online fine-tuning, we set a threshold n_m. When the size of the memory pool $\mathcal{M}$ is greater than n_m, we also use `DQN` to online fine-tune parameters. In `DQN`, we first sample experience replay data $\langle e_t, a_t, r_t, e_{t+1} \rangle$ from $\mathcal{M}$ (line 1). Then we use the network DQN to predict $Q(e_t, a_t)$ and $\mathbb{Q}(e_{t+1})$ respectively (lines 2-3). Afterwards, we generate estimated value $Q'(e_t, a_t)$ based on $\mathbb{Q}(e_{t+1})$ (line 4). Finally, we use MSE (Mean Squared Error) metric to compute the loss $||Q(e_t, a_t) - Q'(e_t, a_t)||^2$ and update parameters (line 5).

VI. Experiments

A. Experimental Setup

Workloads. We use two kinds of workloads. The first is the real dataset IMDB and the open-source workload `JOB` (Join Order Benchmark), where the size of IMDB is 3.7GB and `JOB` includes 113 multi-table join queries. For making more redundant computation, we generate a new query for each raw query by manually modifying the predicates, and thus get the new workload with 226 queries. The second consists of two real-world SQL workloads, denoted as `WK1` and `WK2`, respectively. In particular, they are both collected from data analysis projects of Ant-Financial(https://www.antfin.com/), and the database sizes of the two workloads are respectively 126GB and 185GB. Table I shows the information.

Firstly, we count the number of projects, the number of tables, the number of queries and the number of extracted subqueries, which are denoted as *#project*, *#table*, *#query* and *#subquery* respectively. Secondly, we use the method EQUITAS [45] to detect equivalent subqueries, and the number of equivalent subquery pairs is denoted as *#equivalent pairs*. After that, we partition subqueries into disjoint groups, and select the subquery with the least overhead as the candidate subquery for each group. We denote the number of candidate subqueries as *#candidate subqueries*, denoted as $|Z|$. Thirdly, we collect queries that can use at least one materialized view built on candidate subqueries Z. We denote the number of these queries as *#associated queries*, which is also represented as $|Q|$. Finally, there are overlapping subqueries, and we denote their number as *#overlapping pairs*.

Baselines. (1) **Cost estimation.** To evaluate the effectiveness of `W-D`, we compare it with existing methods:

a) Traditional estimation: similar to [20], it uses scanning the

TABLE I
WORKLOAD DATASETS

workloads	JOB	WK1	WK2
# project / # table	1/21	21/389	25/435
# query / # subquery	226/398	38.6k/79.6k	157.6k/302.5k
# equivalent pairs	1,312	27,445	98,532
# candidate subquery ($\|Z\|$)	28	2,252	6,871
# associated query ($\|Q\|$)	220	4,642	14,191
# overlapping pairs	74	4,286	5,521

materialized view to replace computing its associated subquery. Thus, we estimate $\mathcal{A}_{\beta,\gamma}(q|v_s) = \mathcal{A}_{\beta,\gamma}(q) - \mathcal{A}_{\beta,\gamma}(s) + \mathcal{A}_{\beta,\gamma}(v_s)$, where $\mathcal{A}_{\beta,\gamma}(v_s)$ is the cost of scanning v_s. In particular, we use two methods to estimate $\mathcal{A}_{\beta,\gamma}(q)$, $\mathcal{A}_{\beta,\gamma}(s)$ and $\mathcal{A}_{\beta,\gamma}(v_s)$. The first is using query optimizers (*Postgres–9.1* for JOB and *MaxCompute–3.2.3* for WK1, WK2) while the second is using the start-of-the-art deep learning method [36]. We denote them as Optimizer and DeepLearn.

b) Linear Regressor (LR): a machine learning approach using a linear function to model cost and computing the loss between the estimated cost and the actual cost with Euclidean distance.

c) Gradient Boosted Machine (GBM): a gradient boosting decision tree based regression method using XGBoost [5].

d) To evaluate the effectiveness of different parts of non-numerical features encodings in W-D (see Figures 8, 7 and 5), we modify W-D by three variations, namely N-Kw, N-Str and N-Exp. In N-Kw, we use one-hot vectors to replace keyword embeddings. In N-Str, we use one-hot vectors to replace char embeddings and remove the CNN model in the string encoding model. In N-Exp, we replace the sequence models (i.e., LSTM1 and LSTM2) with the average pooling of keyword embeddings and string encodings.

(2) **View selection.** We compare RLView with an iterative method BigSub [20] and four greedy methods: **TopkFreq**, **TopkOver**, **TopkBen** and **TopkNorm** [10].

a) BigSub: building a bipartite graph for queries and subqueries, and then regarding the view selection problem as the problem of iteratively labelling vertices and edges of the bipartite graph. For getting a converged solution, BigSub forbids turning selected subqueries to unselected.

b) Greedy methods: Sort candidate subqueries based on different strategies, and then select top-k subqueries to materialize:

- **TopkFreq**: the frequency in the workload. The higher the frequency, the higher the ranking.
- **TopkOver**: the overhead of materializing views. The bigger the overhead, the lower the ranking.
- **TopkBen**: the benefit for the workload. The bigger the benefit, the higher the ranking.
- **TopkNorm**: the ratio between the utility and the overhead. The bigger the ratio, the higher the ranking.

Evaluation metrics for cost estimation methods. We evaluate cost estimation models based on two popular metrics: MAE (Mean Absolute Error) and MAPE (Mean Absolute Percent Error). Specifically, suppose the ground truth is represented as $\mathbf{y} = \{y^i\}$ and the predicted result is denoted as $\hat{\mathbf{y}} = \{\hat{y}^i\}$, where $1 \le i \le N$, these metrics are computed as follows: $MAE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\sum_{i=1}^{N}|y^i - \hat{y}^i|$, $MAPE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\sum_{i=1}^{N}|\frac{y^i - \hat{y}^i}{y^i}|$. In addition, we split queries in each workload into training, validation and test datasets with the ratio 7:1:2. We use Adam [23] as the optimization method.

TABLE II
DEFAULT PARAMETERS

	α, β, γ	I	lr	b_s	n_1, n_2, n_m	γ
JOB	1.67e-5,1e-1,1e-3	50	0.01	8	10, 90, 20	0.9
WK1/WK2		20	0.005	128	10, 990/490, 3k	0.9

TABLE III
EXPERIMENTAL RESULTS ON COST ESTIMATION

Metric	Optimizer	DeepLearn	LR	GBM	N-Exp	N-Str	N-Kw	W-D
MAE (JOB)	4.33	1.69	1.94	1.30	1.41	1.28	1.20	**1.16**
MAPE(%) (JOB)	39.58	26.63	37.32	25.05	26.87	24.40	23.12	**22.77**
MAE (WK1)	4.52	1.59	2.17	1.58	1.20	0.89	0.81	**0.77**
MAPE(%) (WK1)	41.32	27.24	37.04	27.02	17.65	14.39	13.24	**12.96**
MAE (WK2)	4.39	4.07	2.73	2.00	1.65	1.45	1.27	**1.11**
MAPE(%) (WK2)	76.44	63.99	39.45	28.92	22.54	20.11	17.89	**16.98**

Parameter settings. The Parameters of our system can be divided into two parts. The first part is manually set by users, such as the cost parameter α, β, γ, the number of training epochs I for the model W-D and the number of iterations n_1, n_2 for the algorithm RLView. The second part is fine-tuned by the validation dataset, such as the learning rate lr and the batch size b_s in W-D, and the memory size n_m and the reward decay rate γ in RLView. Default settings are listed in Table II. For quantifying costs by pricing strategies, we set the unit of α, β, γ as "*$*/GB*", "*$*/(Core·Minute)" and "*$*/(GB·Minute)". Therefore, we can represent $\mathcal{A}_\alpha$, $\mathcal{A}_\beta$ and $\mathcal{A}_\gamma$ with the same unit "*$*" by the following formulas: $\mathcal{A}_\alpha = \alpha \cdot u_{sto}$, $\mathcal{A}_\beta = \beta \cdot u_{cpu}$ and $\mathcal{A}_\gamma = \gamma \cdot u_{mem}$, where u_{sto}, u_{cpu} and u_{mem} correspond to storage usage (GB), CPU usage (Core·Minute) and memory usage (GB·Minute).

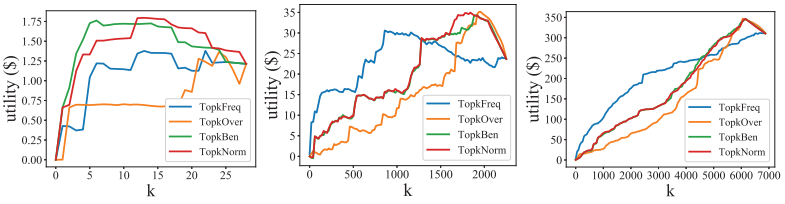
Environment. We use a machine with Intel(R) CPU E5-2630, 128GB RAM and use PyTorch 1.0.

B. Comparison with Baselines

1) Cost Estimation: We compare our W-D model with baselines on JOB, WK1 and WK2 respectively. In particular, for JOB, we rewrite queries with all possible candidate views, and thus get actual computation cost as ground truth by executing rewritten queries. However, it is not realistic to generate ground truth for WK1 and WK2 by rewriting and executing queries with all candidates due to the large overhead. To address this issue, we design a new method RealOpt to get approximate results as ground truth. In RealOpt, for a query q and a view v_s, we first get the actual cost $\mathcal{A}_{\beta,\gamma}(q)$ and $\mathcal{A}_{\beta,\gamma}(s)$ by executing the query q, and then use $\mathcal{A}_{\beta,\gamma}(q) - \mathcal{A}_{\beta,\gamma}(s)$ to represent the ground truth of $\mathcal{A}_{\beta,\gamma}(q|v_s)$. In summary, Table III reports the MAE and MAPE losses of all methods, from which we have the following observations.

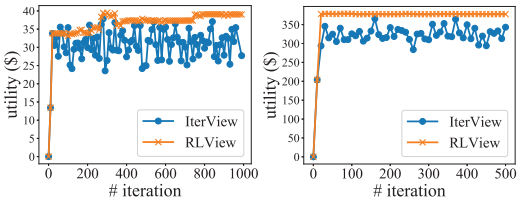
(1) The performance of Optimizer is the worst. For example, its MAPE losses on WK2 is nearly 80%. The main reason is that the error can be accumulated when respectively estimating $\mathcal{A}_{\beta,\gamma}(q)$, $\mathcal{A}_{\beta,\gamma}(s)$ and $\mathcal{A}_{\beta,\gamma}(v_s)$. However, DeepLearn is better than Optimizer on estimating the cost, so DeepLearn has better performance than Optimizer. In addition, the performance of LR is not high. The reason is that computation cost and extracted features are not linearly related.

(2) The neural network based methods outperform other methods in most cases. As mentioned before, N-Exp, N-Str,



(a) JOB (b) WK1 (c) WK2

Fig. 9. Top-k based methods comparisons



(a) WK1 (b) WK2

Fig. 10. Convergence comparisons

N-Kw and W-D are implemented with neural networks. It is well known that deep neural networks can approximately fit any function, so it is reasonable to get better performance using the deep learning technique. However, different deep learning models focus on different features and thus influence the estimation results. For example, W-D focuses on the plan information using the plan sequence encoding model.

(3) The neural network based methods perform better on WK1 and WK2 than JOB. The main reason is that small scale data could cause the overfitting problem for deep neural network models. In contrast, learning on large scale data can improve the generalization performance. However, all methods perform better on WK1 than WK2. The reason is that WK2 includes more complex queries, which causes it difficult to learn effective representation of features.

(4) W-D outperforms other methods. The reason is two-fold. On one hand, we consider more information for estimating cost. On the other hand, we utilize different models to effectively extract significant features from the information. Firstly, we use embeddings to represent keywords and thus could learn the dependency between different keywords. Secondly, we design the string encoding model to capture the char-level and local features of strings by using the char embeddings and the CNN model respectively. Thirdly, we use the sequence model to capture the sequence features of query/view plans.

(5) Comparing three variations of W-D, we can get two findings as below. Firstly, the plan encoding model is the most important (e.g., the performance of N-Exp is worse than both N-Str and N-Kw). Secondly, the string encoding model is more important than the keyword embedding model. The reason is that the keyword embedding is just equivalent to adding a layer of neural network, whose influence is not too big.

(6) We explore some cases where W-D has no good performance. We find that in most cases W-D achieves good performance, and it may not get good performance in some cases. For example, the performance of W-D is not good for a query with 10-table join and a view with 3-table join on JOB.

2) View Selection: In this section, we aim to evaluate the effectiveness of our learning-based method. We compare it with other methods on JOB, WK1 and WK2 respectively.

Finding k for greedy methods. As mentioned before, the four greedy methods (i.e., **TopkFreq**, **TopkOver**, **TopkBen**, **TopkNorm**) compute the optimal solution by using top-k candidate subqueries. To get the best result for each method, we set k with different values. As shown in Figure 9, we illustrate the utility curve with k for the four methods respectively. Because the maximum value of k is equal to the number of candidate subqueries, the range of k for JOB, WK1 and WK2 are $[0, 28]$, $[0, 2252]$ and $[0, 6871]$ respectively. According to Figure 9, we can observe that almost all curves first rise up to the maximum point and then fall down. The reason is two-fold. On one hand, with the increasing of k, queries in the whole workload can reuse more computation and get more benefit, so the utility increases. On the other hand, the overhead for materializing subqueries also increases with the increasing of k. When k is greater than a certain threshold, the overhead would dominate the utility, which results in the decrease of the utility.

TABLE IV
THE OPTIMAL RESULTS FOR DIFFERENT VIEW SELECTION METHODS

	TopkFreq	TopkOver	TopkBen	TopkNorm	BigSub	RLView	OPT
k (JOB)	22	24	6	13	67	44	-
utility($)	1.38	1.36	1.76	1.80	1.78	**1.85**	1.98
ratio(%)	8.97	8.83	11.44	11.70	11.57	12.02	12.86
k (WK1)	860	1.94K	1.9K	1.84K	446	285	-
utility($)	30.60	35.14	34.33	34.96	37.84	39.62	-
ratio(%)	4.44	5.11	4.99	5.08	5.50	5.76	-
k (WK2)	6.7K	6.05K	6.15K	6.15K	167	94	-
utility($)	311.52	346.86	346.61	346.34	365.36	379.25	-
ratio(%)	9.15	10.19	10.18	10.17	10.73	11.14	-

Comparing optimal results. For each method, we collect its associated maximum utility as its optimal result. Table IV shows the results. All methods are split into two kinds: greedy (e.g., **TopkFreq** and **TopkOver**) and iteration based (e.g., BigSub and RLView). For each greedy method, we record the associated k values when getting its maximum utility. As for each iteration based method, we record the number of iteration (also denoted as k for simplicity) for getting its maximum utility. In addition, given a workload Q and a materialized view set V_S, we compute the associated ratio $\frac{U^{max}_{Q,V_S}}{\sum_{q\in Q}\mathcal{A}_{\beta,\gamma}(q)}$ for each method, where U^{max}_{Q,V_S} represents the maximum utility and $\sum_{q\in Q}\mathcal{A}_{\beta,\gamma}(q)$ denotes the total cost of Q. The bigger the associated ratio, the better the method. In addition, we try to use ILP solvers to get actual optimal results for JOB, WK1 and WK2, but they can only give the solution for JOB and fail for WK1 and WK2, because the datasets are too large. Thus, we report the actual solution as **OPT** and demonstrate the optimality gap for JOB. From the table, we have the following observations: (1) Iteration based methods outperform greedy methods on almost all workloads. The reason is two-fold. Firstly, greedy methods cannot guarantee the optimal order of candidate subqueries, and thus they cannot get the actual maximum utility. Secondly, iteration based methods can explore more situations by iteratively selecting candidate subqueries and thus have more possibilities to explore the actual optimal situation. (2) RLView outperforms other methods. The reason is two-fold. Firstly, we rewrite the view selection problem as an ILP problem and propose an iterative optimization function to solve the problem. Secondly, we regard the optimization

TABLE V
END-TO-END RESULTS (**O&B**: Optimizer + BigSub, **O&R**: Optimizer + RLView, **W&B**: W-D + BigSub, **W&R**:W-D + RLView)

Data	JOB			P1			P2		
	$\#q$	c_q(\$)	l_q(s)	$\#q$	c_q(\$)	l_q(s)	$\#q$	c_q(\$)	l_q(s)
	226	15.39	571.16	832	91.27	7.07k	5378	558.19	49.9K
	$\#(q\|v)$	$\#m$	o_m(\$)	$\#(q\|v)$	$\#m$	o_m(\$)	$\#(q\|v)$	$\#m$	o_m(\$)
O&B	182	24	1.04	231	24	1.04	1307	162	34.49
O&R	164	19	0.85	233	24	0.70	1361	156	30.25
W&B	140	17	0.46	224	21	0.79	1345	174	40.79
W&R	148	18	0.53	250	26	0.93	1300	144	22.96
	$b_{q\|v}$(\$)	$l'_q(s)$	$r_c(\%)$	$b_{q\|v}$(\$)	$l'_q(s)$	$r_c(\%)$	$b_{q\|v}$(\$)	$l'_q(s)$	$r_c(\%)$
O&B	2.48	479.12	9.36	8.75	6.04k	8.45	71.86	30.36k	6.69
O&R	2.44	480.61	11.70	8.90	6.02k	8.98	75.31	30.25k	8.07
W&B	2.04	495.47	10.27	8.76	6.02k	8.73	83.20	30.70k	7.60
W&R	2.38	482.86	**12.02**	9.32	5.98k	**9.19**	72.15	30.75k	**8.81**

process as an MDP and apply the deep reinforcement learning technique, which is proven to be effective to get the optimal solution of an MDP problem. (3) JOB and WK2 include more redundant computation than WK1. For example, we can save more than 10% cost by using materialized views for JOB and WK2, but the ratio is only around 5% for WK1.

Evaluating the convergence. To verify the convergence of RLView, we directly use the function IterView to solve the optimization problem on WK1 and WK2. As shown in Figure 10, we record the intermediate utility in each iteration for RLView and IterView, respectively. To make fair comparison, we set the iteration number n in IterView as the summation of n_1 and n_2 in RLView. In conclusion, we have the following observations: (1) IterView cannot converge into global optimal results on both WK1 and WK2. For example, the utility of WK1 first rises to \$33 and then sharply fluctuates between \$25 and \$35. (2) RLView can keep the utility stable on the two workloads. The reason is two-fold. Firstly, IterView only considers local optimal results when selecting subqueries to materialize based on probabilities. Therefore, the method would sharply fluctuate between different local optimal results. Secondly, the DQN technique in our RLView method has the memory ability to store experiences, and thereby eliminates invalid optimization process and avoids the sharp falling of the utility. (3) The benefit or overhead of subqueries in WK1 is more skewed than in WK2. Thus, the intermediate utility of both IterView and RLView for WK1 has wider fluctuation range than WK2. In addition, there even exists a sharply decrease of the utility near the iteration 300 in Figure 10(a), which is caused by selecting some subqueries with heavy overhead or unselecting some subqueries with heavy benefit.

C. End-to-end Experiment

We implement the end-to-end system based on the cost estimation model and the view selection model. To evaluate the effectiveness, we compare four combinations: Optimizer + BigSub, Optimizer + RLView, W-D + BigSub, W-D + RLView, which are denoted as **O&B**, **O&R**, **W&B** and **W&R**, respectively. In addition, we sample two projects, denoted as P1 and P2, respectively from WK1 and WK2 for the experiment that can materialize all high-quality views, because it is expensive to execute the whole query set.

Table V shows the end-to-end results. At first, we report the number ($\#q$), the cost (c_q) and the latency (l_q) of raw queries. Then, for each method, we report the number ($\#m$) and the overhead (o_m) of materialized views, the number ($\#(q|v)$) and the benefit ($b_{q|v}$) of rewriting queries, and the latency (l'_q) of the rewritten workload. At last, we report the associated ratio (r_c), which is computed as $r_c = \frac{b_{q|v}-o_m}{c_q}$. In conclusion, we can find some observations as follows: (1) Our system outperforms other methods. For JOB, **W&R** can save 12.02% cost while **O&B** only save 9.36% cost, so our system improves the performance by $\frac{12.02-9.36}{9.36} \times 100\% = 28.4\%$. Similarly, the improvement for P1 and P2 are $\frac{9.19-8.45}{8.45} \times 100\% = 8.8\%$ and $\frac{8.81-6.69}{6.69} \times 100\% = 31.7\%$, respectively. (2) The more accurate the cost model, the better the solution of the view selection model. For example, **W&B** and **W&R** save more cost than **O&B** and **O&R**, respectively. (3) RLView is more robust than BigSub. Taking JOB for example, the ratio r_c of BigSub is decreased by $10.27\% - 9.36\% = 0.91\%$ while RLView is only decreased by $12.02\% - 11.70\% = 0.32\%$. (4) Building more materialized views doesn't mean saving more cost. In the example of JOB, **O&B** gets the most benefit, but it saves the least cost because of the heavy overhead of views.

VII. RELATED WORK

Cost estimation. Optimizers estimate query execution cost using a mathematical model, which relies heavily on estimation of the cardinality, or the number of tuples [19], [14], [18]. Traditionally, database systems [35], [32] estimate selectivities through fairly detailed statistics on the distribution of values in each column, such as histograms. Benefit from the development of deep learning (e.g. CNN [17], [39], [13], RNN [16], [1], [15], [37]), neural networks have been applied to estimate cost in many domains, such as traffic time prediction [26], [21], [25] and query cost estimation [29], [36], [24], [44]. In this paper, we focus on the benefit of reusing a materialized view for a query. The key point is to estimate the cost of a rewritten query, whose associated subquery is replaced with a materialized view. However, we cannot directly use the above methods to address our problem, since it's not realistic to rewrite all queries before generating materialized views especially when the number of queries is too large. Thus, we take into account useful features and propose a new method, extended from the effective deep learning model Wide&Deep [6], to solve it.

Subquery reusing. Given a set of queries, there are many optimization methods on selecting subqueries to reuse [28], where the target is to minimize a cost function (e.g. space overhead and computation cost) under a set of constraints (e.g. query deadline and space budget). One of the most popular ways is to generate materialized views for selected subqueries. Some studies focus on materialized views selection in the context of data warehouse [11], [12], and several current works aim to improve query latency in analytics clusters by selecting views to materialize. However, these methods cannot detect duplicate computation between different subqueries. Therefore, some related works [20], [3], [31], [34] reuse

common subqueries among different queries. In particular, the authors in [31] study the history-aware query optimization with materialized intermediate views. The authors in [3] use an AND/OR graph representation and an ILP-based solution to select common subqueries to reuse for Pig script. The authors in [34], [20] consider computation reusing in cloud computing.

SQL equivalence. Relational query equivalence is a well studied problem in database theory. The authors in [40] have proven that the first-order logic on the class of all finite models is undecidable. Thus, some works [4], [33] focus on decidable SQL fragments. However, these works are not used in real products. To address this problem, Cossete [9], [8] verifies SQL equivalence by formalizing a substantial fragment of SQL in the *Coq Proof Assistant* and the *Rosette symbolic virtual machine*, but this toolkit is mainly based on syntax but not semantics rewriting. Satisfiability Modulo Theories (SMT [2] and Symbolic Execution [22]) are theory foundations to check whether a first-order predicate is satisfiable. The authors in [45] regard the problem of detecting equivalent subqueries as the problem of detecting whether two first-order predicates are equivalent based on SMT and Symbolic Execution.

VIII. Conclusions

We proposed an end-to-end view selection system by automatically selecting the most beneficial subqueries to materialize. We proposed an effective deep learning model to estimate the cost of a query with a materialized view. We extracted significant features from different kinds of information, such as query/view plans and query/view input tables. We modeled the materialized view selection problem as an ILP problem. We proposed an iterative optimization method to get the approximate optimal solution. We transformed ILP as an MDP and used the reinforcement learning technique DQN to solve it. Extensive experiments showed that our methods outperformed state-of-the-art by 28.4%, 8.8% and 31.7% on three datasets.

References

[1] Y. Bengio and P. Frasconi. Credit assignment through time: Alternatives to backpropagation. In *NeurIPS*, pages 75–82, 1993.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185, 2009.

[3] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. R. Chowdhury. Reuse-based optimization for pig latin. In *CIKM*, pages 2215–2220, 2016.

[4] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[5] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *SIGKDD*, pages 785–794, 2016.

[6] H. Cheng et al. Wide & deep learning for recommender systems. In *DLRS*, pages 7–10, 2016.

[7] K. Cho et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *EMNLP*, pages 1724–1734, 2014.

[8] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *PVLDB*, 11(11):1482–1495, 2018.

[9] S. Chu, C. Wang, K. Weitz, and A. Cheung. Cosette: An automated prover for SQL. In *CIDR*, 2017.

[10] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, pages 75–88, 2010.

[11] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, 1997.

[12] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *TKDE*, 17(1):24–43, 2005.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In *ECCV*, pages 630–645, 2016.

[14] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[15] G. E. Hinton and J. L. McClelland. Learning representations by recirculation. In *NeurIPS*, pages 358–366, 1987.

[16] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[17] F. Huang, J. T. Ash, J. Langford, and R. E. Schapire. Learning deep resnet blocks sequentially using boosting theory. In *ICML*, 2018.

[18] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. *VLDB J.*, 6(2):132–151, 1997.

[19] R. Jin, D. Fuhry, and A. Alali. Cost-based query optimization for complex pattern mining on multiple databases. In *EDBT*, pages 380–391, 2008.

[20] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[21] I. Jindal, X. Chen, et al. A unified neural network approach for estimating travel time and distance for a taxi trip. *CoRR*, 2017.

[22] J. C. King. Symbolic execution and program testing. *Commun*, 19(7):385–394, 1976.

[23] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[24] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *PVLDB*, 2019.

[25] X. Li, G. Cong, A. Sun, and Y. Cheng. Learning travel time distributions with deep generative model. In *WWW*, pages 1017–1027, 2019.

[26] Y. Li, K. Fu, Z. Wang, C. Shahabi, J. Ye, and Y. Liu. Multi-task representation learning for travel time estimation. In *SIGKDD*, pages 1695–1704, 2018.

[27] T. P. Lillicrap et al. Continuous control with deep reinforcement learning. *CoRR*, 2015.

[28] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD Record*, 41(1):20–29, 2012.

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

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, 2013.

[31] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *ICDE*, pages 520–531, 2014.

[32] L. A. Rowe and M. Stonebraker. The POSTGRES data model. In *VLDB'*, pages 83–96, 1987.

[33] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[34] Y. N. Silva, P. Larson, and J. Zhou. Exploiting common subexpressions for cloud query processing. In *ICDE*, pages 1337–1348, 2012.

[35] M. Stonebraker, E. N. Hanson, and C. Hong. The design of the postgres rules system. In *ICDE*, pages 365–374, 1987.

[36] J. Sun and G. Li. An end-to-end learning-based cost estimator. *VLDB*, 2019.

[37] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NeurIPS*, pages 3104–3112, 2014.

[38] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. *TNN*, 9(5):1054–1054, 1998.

[39] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, pages 4278–4284, 2017.

[40] B. A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *DAN SSSR*, 70:569–572, 1950.

[41] O. Vinyals, L. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. E. Hinton. Grammar as a foreign language. In *NeurIPS*, pages 2773–2781, 2015.

[42] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. In *ICML*, pages 1995–2003, 2016.

[43] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

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

[45] Q. Zhou, J. Arulra, S. B. Navathe, W. R. Harris, and D. Xu. Automated verification of query equivalence using satisfiability modulo theories. In *VLDB*, pages 75–88, 2019.