AutoView: An Autonomous Materialized View Management System with Encoder-Reducer

Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun

Abstract—Materialized views (MVs) can significantly optimize the query processing in databases. However, it is hard to generate MVs for ordinary users because it relies on background knowledge, and existing methods rely on DBAs to generate and maintain MVs. However, DBAs cannot handle large-scale databases, especially cloud databases that have millions of database instances and support millions of users. Thus it calls for an autonomous MV management system. In this paper, we propose an autonomous materialized view management system, AutoView. It analyzes query workloads, estimates the costs and benefits of materializing queries as views, and selects MVs to maximize the benefit within a space budget. We propose a serialization and encoding method that can encode the features of both queries and views into vectors. Then we design a sequence-to-sequence model, Encoder-Reducer, to estimate MVs' cost/benefit by taking the encoding vectors as input. Next, we propose a deep reinforcement learning model to select high-quality MVs, which enriches the state representation with Encoder-Reducer's output. Experimental results show that our method outperforms existing studies in terms of MV selection quality.

Index Terms—materialized views, database, deep learning, deep reinforcement learning.

1 Introduction

Materialized views (MVs) are very important in DBMS that utilize views to improve the query performance based on the space-for-time trade-off principle. Specifically for online analytical processing (OLAP), many queries share equivalent sub-queries and there are many redundant computations among these queries. MVs can alleviate this problem by utilizing views to avoid such redundant computations.

However, it is hard to automatically generate MVs for ordinary users, because it relies on background knowledge. Existing methods rely on DBAs to generate and maintain MVs. However, DBAs cannot handle large-scale databases, especially cloud databases that have millions of database instances and support millions of users. Therefore, it calls for an autonomous MVs management system, which, given a query workload, selects potential queries (subqueries) as views and uses the views to answer subsequent queries [35], [38], [43], [44].

MV management systems have four main modules. (1) MV candidate generation. It analyzes the query workload, selects common sub-queries, and takes them as candidates to generate MVs. (2) MV Cost/Benefit estimation. It estimates the cost and benefit of materializing subqueries as views, where the cost includes the space/time overhead and the benefit is the saved execution time using the view to optimize queries. (3) MV selection. It selects high-quality MV candidates to generate MVs based on the estimation model, aiming to maximize the benefit within a given cost budget. (4) MV-aware query rewriting. Given a new query, it selects appropriate views and rewrites the query based on the selected views. There are several challenges in these four modules. First, MV selection relies on benefit estimation of using a view to optimize a query, and existing methods [1], [2], [8], [11], [16], [17], [36], [41] do not consider the complicated effect of views on queries and cannot capture the correlation between views and queries. Second, traditional MV selection methods model it as the *knapsack problem* and use greedy algorithms to choose which MVs to materialize. However, the knapsack problem relies highly on the estimation model and cannot find high-quality views. Third, MV rewriting also relies on the estimation model, but existing models depend on the cost model of optimizers and cannot effectively estimate the cost and benefit of using an MV to answer a query.

To address these challenges, we propose an end-to-end autonomous MV management system, AutoView. It first analyzes the query workloads, extracts common subqueries, and selects the subqueries with high frequency as MV candidates. Then it estimates the benefits of MV candidates and selects the candidates with the highest benefits as MVs. We propose a benefit/cost estimation model to estimate MV candidates. The estimation model serializes, encodes, and estimates queries and views. A Recurrent Neural Network (RNN) model, Encoder-Reducer, is designed to embed queries and views as embedding vectors and predict the benefit of views. In Encoder-Reducer, the encoder *encodes* a query into a semantic vector, and the reducer *reduces* views from a query and predicts the benefit. Next, to effectively select the MVs, we propose a reinforcement learning (RL) model, Encoder-Reducer Double Deep Q-learning Network (ERDDQN), to select MVs. Finally, for MV rewriting, we use the ERDDQN model to select MVs to rewrite queries.

Contributions. We make the following contributions.
(1) We propose an autonomous materialized view management system, AutoView, with deep reinforcement learning.
(2) We design Encoder-Reducer to estimate the cost and benefit of using MVs to answer a query. We propose a serialization and encoding method to encode query's features.
(3) We propose a reinforcement model to select MVs for materialization, and integrate the vectors outputted by the Encoder-Reducer model into the model.
(4) Our experimental results on real datasets showed that our method significantly outperformed existing solutions.

• *Y. Han, G. Li, H. Yuan and J. Sun were with the Department of Computer Science,Tsinghua University, Beijing, China. Guoliang Li is the corresponding author. {han-y19@mails.,liguoliang@,yht16@mails.,sun-j16@mails.}tsinghua.edu.cn*

Manuscript received Feb. 28, 2021; revised Jan. 22, 2022; accepted Mar. 19 2022.

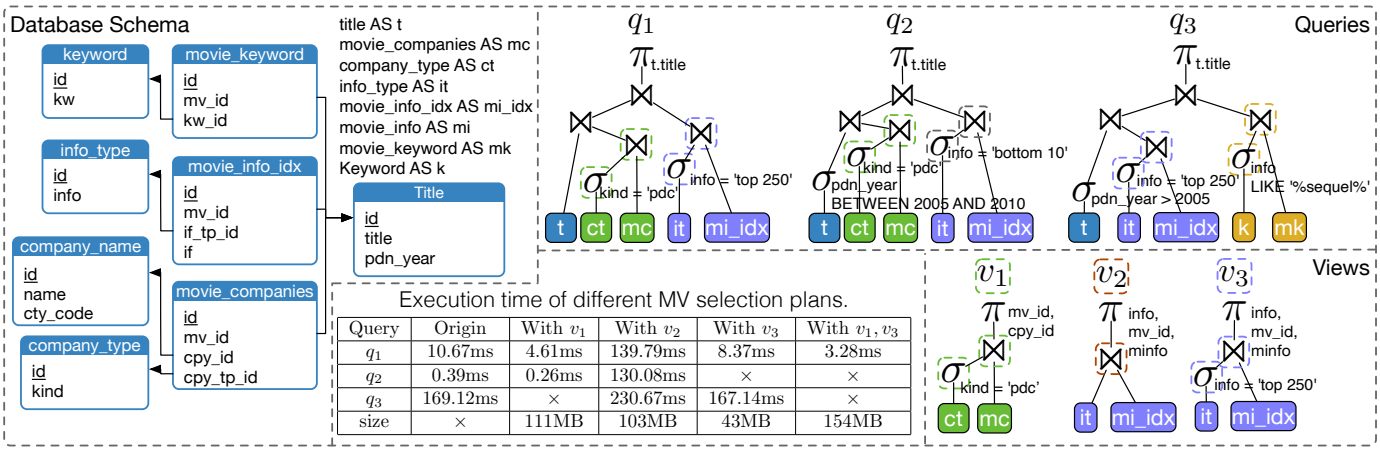


Fig. 1. MV selection example.

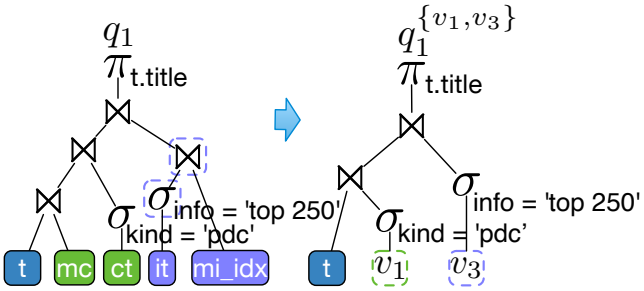


Fig. 2. Query rewriting example.

2 AutoView Overview

2.1 Problem Formulation

MV Selection. Given a set of SQL queries, $Q = \{q_i\}$, we aim to generate a set of views $V = \{v_j\}$, such that (1) the total size of views in V is within a space budget[1], and (2) the performance of using views in V to answer queries in Q is optimized. Figure 1 shows an example with three queries $Q = \{q_1, q_2, q_3\}$ and three views $V = \{v_1, v_2, v_3\}$. The execution time of different optimization plans are also shown in Figure 1. The spaces occupied by v_1, v_2, v_3 are 111MB, 103MB and 43MB respectively. If the MV space budget is 50MB, we will materialize $\{v_3\}$ and utilize it to optimize q_3 with a benefit of (10.67-8.37)+(169.12-167.14)=4.28ms. If the budget is 120MB, we will materialize $\{v_1\}$ and get a benefit of (10.67-4.61)+(0.39-0.26)=6.19ms. If the budget is 200MB, we will materialize $\{v_1, v_3\}$ and get a benefit of (10.67-3.28)+(0.39-0.26)+(169.12-167.14)=9.50ms. We do not materialize v_2, because it does not improve the performance.

Query Rewriting with MVs. Given a set of views $V = \{v_j\}$ and a query q, we select a subset of views, $V^k \subseteq V$, and use the views in V_k to answer query q, such that the performance of answering q with MVs in V is optimized. For example, given three MVs v_1, v_2, v_3, query q_1 can be optimized using v_1 and v_3, and the optimized plan is shown in Figure 2.

2.2 System Overview

To address the MV generation and query rewriting problem with MVs, we propose an autonomous MV management system as shown in Figure 3. The goal of `AutoView` is to automatically generate MVs by analyzing the query workload and utilize the MVs to optimize queries. The system includes four modules, MV candidate generation, MV cost/benefit estimation, MV selection, and MV-aware query rewriting.

MV Candidate Generator. We analyze the workload to find common subqueries for MV candidate generation, where a subquery is a subtree of the syntax tree for relational algebra. Common subqueries are the equivalent or similar rewritten subqueries among different queries. Common subqueries with a high quality will be selected as MV candidates. Equivalent subqueries will be rewritten in the same form [8], [11], [36]. And subqueries that have similar selection conditions will be merged into a large one. For example, "WHERE country IN ('Sweden', 'Norway') GROUP BY country" and "WHERE country IN ('Bulgaria') GROUP BY country" will be merged into "WHERE country IN ('Sweden', 'Norway', 'Bulgaria') GROUP BY country". We discuss the details of MV candidate generation in Section 3.

MV Estimation. Let $V = \{v_j\}$ denote the set of MV candidates. This module estimates the saved execution time (called `benefit`) from executing $q_i \in Q$ by making use of a set of views $V_k \subseteq V$. The `benefit` of using V_k to answer q_i can be calculated by the formula below:

$$\mathcal{B}(q_i, V_k) = t_{q_i} - t_{q_i}^{V_k} \quad (1)$$

where t_{q_i} is the execution time of q_i without using views and $t_{q_i}^{V_k}$ is the execution time of executing q_i using V_k.

We propose an `Encoder-Reducer` model to predict t_{q_i} and $t_{q_i}^{V_k}$ using the features of queries and MVs in Section 4. Note that it is expensive to enumerate and predict all pairs of (q_i, V_k) with exponential time complexity. Instead, we prune unmatched pairs, e.g., a view cannot be used to answer a query, using join and selection conditions.

We estimate the cost of a view v_j including the space cost $|v_j|$ and the view generation time cost t_{v_j} using our `Encoder-Reducer` model. Finally, the estimated benefits, cardinalities and hidden states outputted by the neural network are passed to the MVs selection module. The details are in Section 4.

MV Selection. Given a space budget τ, this module selects a subset of MV candidates to maximize the total benefit of answering queries in Q within the space budget. We model this selection problem as an integer programming problem and propose a reinforcement learning (RL) model to address it. The details of MV selection are presented in Section 5.

MV-aware Query Rewriting. Given a query, if the query can be optimized using the MVs, we use our estimation model to select the most appropriate views and rewrite the query using

1. Our method can also support the case that the total time of generating views in V is within a time constraint.

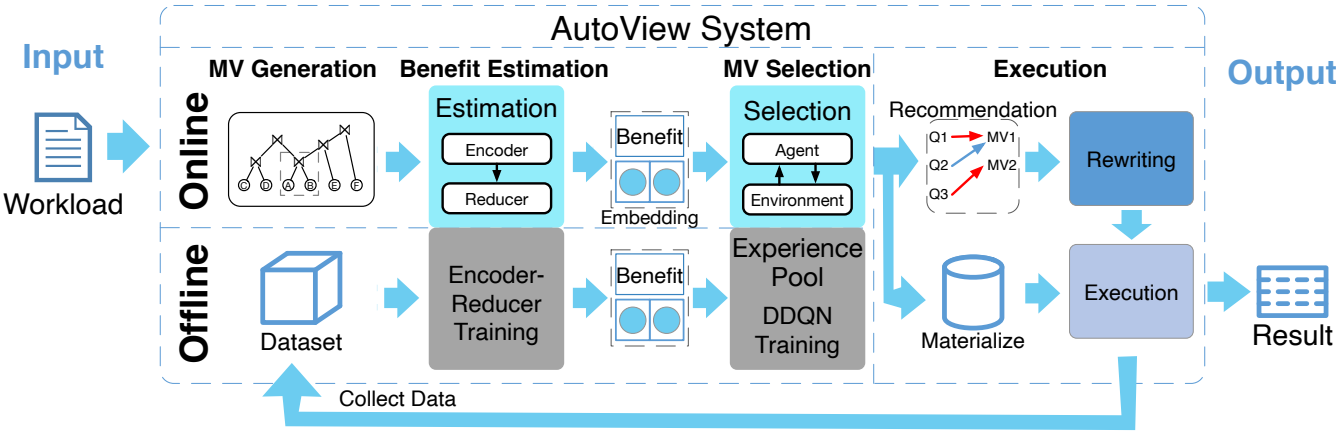


Fig. 3. AutoView Framework.

the views. Subqueries in the query are replaced by the MVs. An example of utilizing v_1 and v_3 to rewrite q_1 is shown in Figure 2. $\sigma_{info='top\ 250'}(it) \bowtie mi\_idx$ is replaced by v_3 and the predicate "info = 'top 250'" is appended in case that v_3 is a superset of "info = 'top 250'". The join order is also reordered. $(t \bowtie mc) \bowtie ct$ is reordered into $t \bowtie (mc \bowtie ct)$, and $mc \bowtie \sigma_{kind='pdc'}(ct)$ is replaced by v_3.

2.3 Related Work

MV Candidate Generation. Traditional MV methods generate MV candidates by exploiting common subqueries that appear frequently in the workload [1], [7], [11], [16], [17], [29], [30], [31], [36]. To make MVs more general, some studies rewrite subqueries to find equivalent sub-queries [1] and merge subqueries [45]. To apply MVs on queries, query equivalence is checked and queries are rewritten based on the MVs. There are two main methods for query equivalence checking and rewriting, using graphs [6], [11], [14], [17], [27], [36] and rules [5], [8], [12]. We propose an efficient query plan-based method to improve the performance of MV candidate generation.

MV Estimation. Existing studies estimate MV benefit by checking whether the MVs appear in the optimal query plans of queries [1]. Traditional methods estimate benefit and cost of MVs based on cardinality estimation. Horng et al. [15] assign query and update frequency to each MV candidate and use cardinality for approximating cost of query and MVs. Ahmed et al. [2] prune MV candidates by heuristics such as join number and cardinality ratio. Traditional cardinality methods, e.g., sampling and histogram, are not accurate, because they cannot capture the correlations between queries and views. Recently, deep learning-based methods [19], [22], [24], [25], [28] are proposed to estimate cost and cardinality more accurately, but they are not aware of the correlation between queries and MVs. Therefore, these methods still perform not well on MV estimation. Thus, we propose a new model that can capture the correlation for a better estimation of answering queries with MVs.

MV Selection. There are many heuristic methods for MV selection. Horng et al. [15] present a genetic algorithm to select MVs by encoding the query plan and MV states in chromosome representation. However, the length of chromosome representation is linearly dependent on workload size and the genetic algorithm becomes slow when the workload size grows. Sohrabi et al. [33] present a frequent itemset mining method to select MVs from queries. However, they do not take the MV maintenance cost into consideration. Gosain et al. [13] design a penalty function for MV disk space and maintenance cost to make heuristic methods better aware of MV cost. Azgomi et al. [3] propose coral reefs optimization algorithm for MV selection. Kumar et al. [20] propose another heuristic method, particle swarm optimization. Jindal et al. [16] propose BigSubs, which is a combination of heuristic algorithm and integer linear programming, to select MVs. However, the heuristic methods rely on some assumptions on the data/workload distribution and adapt to the distribution changes. Therefore, deep learning methods address this by learning from the experience. Liang et al. [23] introduce the reinforcement learning (RL) method, DQM, to learn MV benefits from historical runtime statistics and create/evict MVs. However, DQM still needs a long model training time to adapt to the data/workload distribution or environment changes. Yuan et al. [41] propose RLView to select MVs for workloads. The DQN [26] model they used can only deal with a fixed number of MV candidates because they represent MV states by a fixed-length vector. Moreover, they cannot select MVs within a limited space budget and cannot handle multiple views rewriting. To address these problems, AutoView learns to estimate benefits with query plans and has a better generalization ability.

MV-aware Query Rewriting. Query rewriting using given MVs to answer SQL queries is necessary. Chaudhuri et al. [8] use rules to identify sub-queries in the given query that can be safely substituted by an MV. They extend the join enumeration algorithm, with safe guarantee. However, the application of their method is limited to cost-based join enumeration algorithm. For ML-based cost estimation and optimization algorithm such as [40], their method are not well applicable.

3 View Candidates Generation

The first challenge of autonomous MV management is to obtain *important* view candidates in the workload. Here we take the benefit of a subquery as the importance of materializing the subquery as a view candidate. Intuitively, the benefit of a subquery is positively correlated to its frequency and computational cost. There are two methods to find high-beneficial subqueries. The first method searches for subexpressions of the SQL text as MV candidates, but this method has two limitations. (1) A subexpression of a SQL may not be a valid subquery. (2) A good MV may not be a subexpression. For example, in SQL "SELECT t.title FROM ct, mc, t WHERE ct.kind = 'pdc' AND ct.id = mc.ct_id AND t.id = mc.mv_id", a subquery "SELECT * FROM ct, mc WHERE ct.kind = 'pdc' AND ct.id = mc.ct_id" is a good candidate, but it is not a subexpression.

The second method represents each SQL query as a tree-structured query plan and finds high-beneficial subtrees. As the

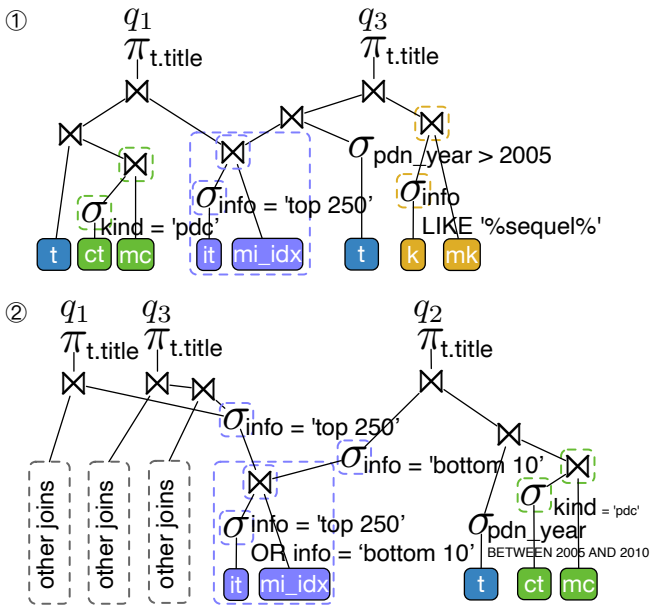


Fig. 4. Merged query plans example.

subtrees in the query plan are valid latent subqueries, we can select subtrees and take their expressions as view candidates. Thus, it is easier to search for view candidates in the tree-structured query plan. Query plans in tree structure provide more choices for selecting view candidates. However, each query may have exponential available query plans according to the different join order. This results in a large number of different subtrees. Yu et al. [40] use RL to search and recommend a low cost query plan in query optimizer. PostgreSQL optimizer uses dynamic programming and heuristics to recommend query plans. Query plans in PostgreSQL can have left-deep, right-deep and bushy join order. We use the query plan recommended by the optimizer, because it has high possibility to contain a high-beneficial MV. Our generator can be extended to support multiple plans of a query from the database optimizer to consider other potential join orders. To address this issue, that there are still many subtrees, we propose an efficient method to extract common sub-expressions and generate MV candidates.

MV Candidate Generation Framework. For each query, we first extract the tree-structured physical query plan from optimizers. We then detect the *common subtrees*, where two nodes (subtrees) in the query plan tree are equivalent if the two nodes have similar join/selection/projection conditions and their children are equivalent. We can merge the two nodes and a merging example is shown in Figure 4. Next, we calculate the benefit of each subtree, which is the product of the number of queries that contain common subtrees and the estimated benefit. Finally, we take the top beneficial subtree as MV candidates. The challenge here is to efficiently detect the common subtrees because it is rather expensive to enumerate every subtree and check every subtree pair.

Merging Similar Nodes. To efficiently detect common subtrees and the corresponding MV for these subtrees, we merge two subtrees into a new subtree once we detect them. A merging example is shown in Figure 1. First, we try to merge q_1 and q_3 and detect common subtrees between them. The purple subtrees, $\sigma_{info='top\ 250'}(it) \bowtie mi\_idx$, have same join condition which is " it.id = mi_idx.it_id". And their children are both "it where

Algorithm 1: JoinQueryGeneration

Input: Q: A set of queries
Output: S: Merged join graph

```
S ← ∅; // initialize the merged join graph
for q_k in Q do
    Queue ← ∅: initialize the to-merge queue;
    T ← Query plan graph of q_k; Add T into S;
    for node_g ∈ S and node_q ∈ T with same table do
        Queue ← (node_g, node_q);
    while Queue ≠ ∅ do
        (node_g, node_q) ← Queue.front
        Queue pops (node_g, node_q)
        if node_g.children ≡ node_q.children and
          node_g ≡ node_q then
            Queue ← {node_g.father, node_q.father}
            node_g ← Merge(node_g, node_q)
```

info='top 250')" and "mi_idx", which are equivalent respectively. Thus, we merge q_1 and q_3 so that they share the same common subtrees. Second, we merge q_2 into the graph of q_1 and q_3. Note that the purple subtree in q_2 is not completely equal to that in q_1 and q_3, but it is worth merging them because using one MV to optimize three queries is beneficial for saving the storage, especially in the situation with a "GROUP BY" clause. We measure the commonality in two subtrees [45], and merge them if the overlap between their predicates is higher than a threshold. The result of the merged subtree should be a union set of the two subtrees. Thus, the selection condition will be "info='top 250' OR info='bottom 10'". Then, additional filters, $\sigma_{info='top\ 250'}$ and $\sigma_{info='bottom\ 10'}$ are appended to the corresponding queries to ensure correct execution results.

Merging Query Plan Trees. To find the high beneficial subtrees, we merge all the query plans among the workload into a Multiple View Processing Plan (MVPP) [15], [32], [39], [42]. MVPP is a directed acyclic graph that merges all the query plan trees. In the graph, equivalent subtrees with the same structure are merged into one subtree so that we can easily find the common subtrees. We adopt a bottom-up manner to merge the equivalent subtrees as shown in Algorithm 1. First, two leaves are merged if they use the same tables and have the same selection/projection conditions. Second, we merge two internal nodes if (1) the two nodes have the same selection/projection conditions and (2) their children are equivalent, i.e., for each child of a node, we can find an equivalent child of the other node and vice versa. Iteratively, we merge the queries into the query graph. Finally, we count the frequency of each node (i.e., the number of queries merged into this node), and take the node (i.e., the corresponding subtree rooted at the node) with high benefit as the MV candidates, for example, the purple common subtree in Figure 4.

4 Benefit Estimation Model

The goal of MV selection is to maximize the benefit of selecting MVs to answer queries. Thus, it is important to estimate the benefit $\mathcal{B}(q, V_k)$ of using a set of views V_k to answer a query q. According to Equation 1, we can estimate t_q and $t_q^{V_k}$ to calculate $\mathcal{B}(q, V_k)$. A rough estimation of $\mathcal{B}(q, V_k)$ is the MV's

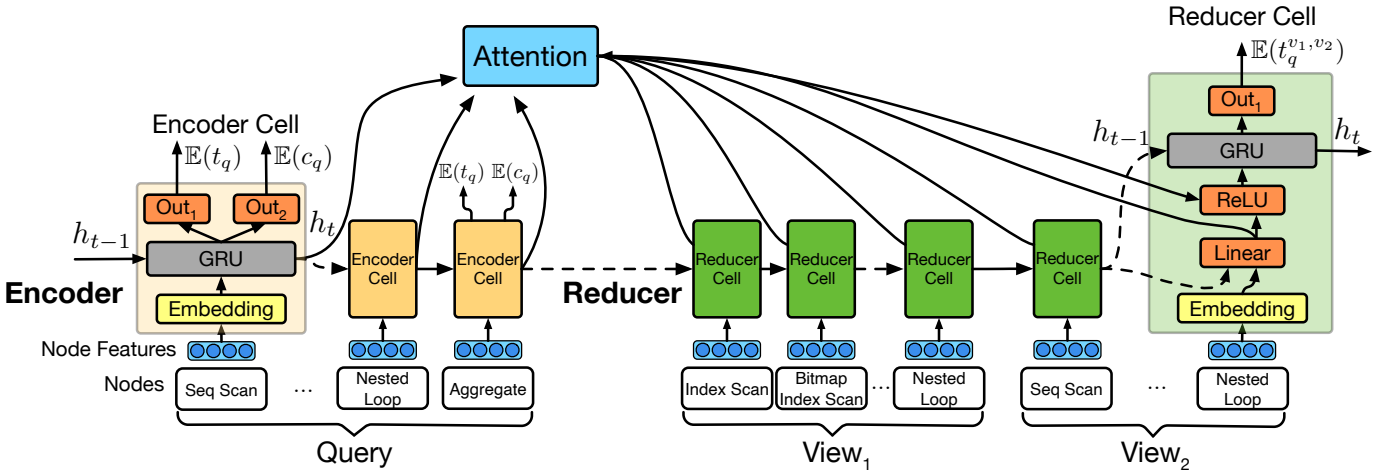


Fig. 5. Encoder-Reducer Model.

generation time t_{V_k} for that the query q can read the result of the MV instead of executing the corresponding subquery repeatedly. Meanwhile, reading the result of MV from the disk instead of memory brings the overhead of disk scanning, $t_{scan_{V_k}}$. Thus, the rough estimation of $\mathcal{B}(q, V_k)$ is $t_{V_k} - t_{scan_{V_k}}$. However, this estimation is very inaccurate, because MVs have complicated effects on queries, e.g., reordering joins due to MVs, and different join orders dramatically affect the execution time of the query [40]. For example, Figure 2 shows that using v_1 changes the original join order of the query plan. Thus we cannot estimate the benefit by estimating the execution time of the query and the view separately using the learning-based single query cost estimator [34] and conduct linear computations.

To estimate the benefit, we propose a learning-based query-view model. There are two main challenges. (1) How to input an SQL query into the tensor model? different query plan trees have a varied number of nodes, and they cannot be simply concatenated and input into the neural networks with fixed input size. Thus, we serialize and encode queries and views, and propose a Recurrent Neural Network (RNN) model, `Encoder-Reducer`, to estimate the benefit. We use the RNN model because it is suitable for encoding varied-length sequences of nodes. `Encoder-Reducer` model is different from the traditional `Encoder-Decoder` model [9]. `Encoder-Decoder` model translate sentences from a language to another language. Instead, our encoder encodes q into a semantic vector and our reducer "reduces" q with V_k and predicts the benefit. (2) How to capture the correlation between queries and views? Experimental results show that two simple RNN models cannot predict benefit/cost accurately enough. After deeply analyzing the RNN structure and node embedding, we found that the co-occurrence of the same or correlated predicates and join orders between a query and views affects the benefit. Therefore, we define three relationships, similarity, conflict and reordering, between a query and views at query plan node level, and use Multi-head Attention [37] in `Encoder-Reducer` to better capture the correlations between nodes of query trees and view trees. Moreover, we propose two-step training to accelerate model convergence and improve the performance.

4.1 Overview of The Estimation Model

We use the `Encoder-Reducer` model to estimate the execution time t_q of answering query q, the execution time $t_q^{V_k}$ of using V_k to answer query q, the time t_v of executing v and the cardinality c_v of view v. The `Encoder-Reducer` model consists of two sub-models, encoder and reducer, as shown in Figure 5. We first extract node features from the query plan tree. We then input these query features into the encoder using an embedding model. The encoder predicts t_q and encodes the query into a semantic vector. Next, we input the semantic vector and views into the reducer to predict $t_q^{V_k}$. With t_q and $t_q^{V_k}$, we derive the benefit. To obtain the space/execution cost of views, we input each view into the encoder to predict the corresponding c_v and t_v. To improve the accuracy of prediction, we use the multi-head attention in `Encoder-Reducer` to better capture the correlation between query and views.

4.2 Encoding

Deep neural networks only accept tensor models and SQL queries cannot be directly inputted into neural networks. To address this problem, we use a serialization and encoding method that transforms SQLs into tensors while keeping its structural and semantic information. Given an SQL query, we serialize the query plan tree by a postorder traversal which is the executing order of the query execution engine. After serialization, we encode the elements (tree nodes) in the sequence into fixed-length tensors by using an existing encoding method [34]. For a node in the query plan, which contains join type, index type, predicates, and so on, we assign an encoding on them using one-hot encoding and then concatenate them. Meanwhile, we sample the data of the relation and append the sample bitmap [19] (e.g., 500 bits) to the encoding, to utilize the sample bitmap to capture the selectivity and distribution of the data.

4.3 Encoder-Reducer Model

We use the `Encoder-Reducer` model to estimate the benefit and obtain the query-view semantic vector. The query-view semantic vector outputted by the reducer implies the meaning of the rewritten query, q^{V_k}. Reducer estimates the execution time where the query is optimized by MV without regard to the order of inputted MVs. Intuitively, MVs reduce the execution time by replacing redundant computations in queries. Query rewritten can be seen as subtraction of views from a query on the SQL level, and the semantic vector can be seen as subtraction of views from a query on the semantic level, and the prediction of the model can be seen as subtraction of views from a query on the execution time level.

Encoder. The encoder cell contains an embedding layer, a gated recurrent unit [10] (GRU) cell and two output layers as shown in Figure 5. The embedding layer is to embed the node features into a dense vector called embedding. The GRU cell gets the passed

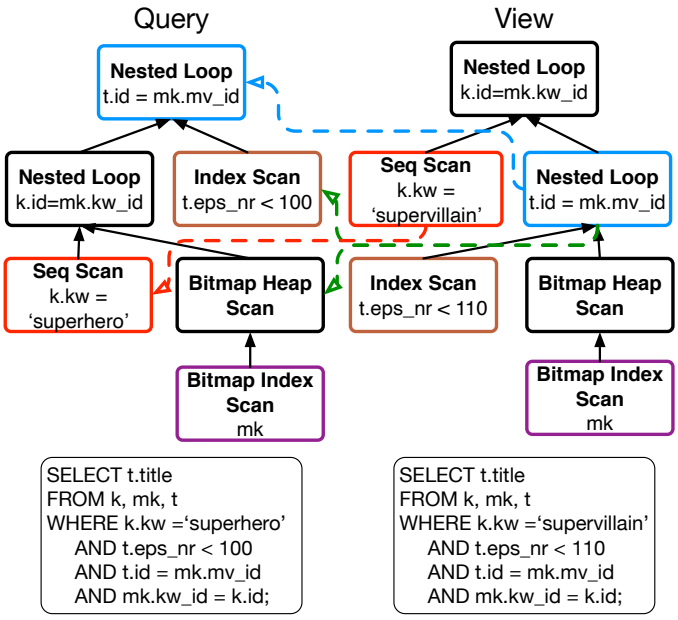


Fig. 6. Attention among nodes.

hidden state, h_{t-1}, and the current node embedding. It remembers some new information about the current node embedding and forgets some useless information in the hidden state. The two output layers are designed to output the estimated execution time and cardinality at the current node. The new hidden state, h_t, which contains the information of the past hidden state and the current node, is passed to the next recurrence of the encoder cell. At the last recurrence of the encoder cell of the encoder, the outputted hidden state is regarded as the encoding of the whole query. The outputted values are the estimated execution time and cardinality of the whole query.

Reducer. The reducer cell contains an embedding layer, a GRU cell, a linear layer, a ReLU layer and an output layer as shown in Figure 5. The linear layer combines the hidden state with the embedded input to be an input of the attention module. The ReLU layer combines the attention with the input. The output layer outputs the estimated execution time of the rewritten query. The cardinality of the rewritten query is the same as the query, so it is not necessary to predict cardinality repeatedly. As the inputs of the encoder and reducer are in the same semantic space, the parameters of the embedding layer and the output layer are shared between the encoder and reducer. The difference between hidden states h_{t-1} and h_t contains the information of the current node operation.

4.4 Multi-head Attention

We use the Multi-head Attention [4], [37] to capture the relationship between nodes in query plan tree and view plan trees because it is hard for GRU to deliver long-term information in the hidden state. There are three types of relationships – node similarity, node conflict, node reordering, which have a significant influence on the benefit of views. For example, Figure 6 shows these relationships between nodes in a query and a view.

Node Similarity. The green dashed line shows the relationship of node similarity, e.g., a node in view outputs same result as the node in the query. We find that if an MV can be utilized to answer a query, there will be some similar sub-plan appearing in the MV and the query. Capturing this information will help the model understanding on how much computation the query can replace with the MV. For example, on the "Nested Loop(t.id=mk.mv_id)" node in the view's plan, the model needs to attend to the "Index Scan(t)" node and "Bitmap Scan(mk)" node in the query's plan. This helps the model know that the query can reuse the computation performed by the "Nested Loop" node although "t" and "mk" are not joined directly in the query. Note that the predicate of "Index Scan(t)" nodes in the view and query are a little different, but the predicate can still be reused because the scan result of predicate "episode_nr<110" contains the result of predicate "episode_nr<100". We just attach an additional predicate "episode_nr<100" when using the view.

Node Conflict. The red dashed line shows the relationship of node conflict, e.g., the predicates on the query and view conflict. For example, on the "Seq Scan(kw, kw='supervillain')" node in the view plan, the model needs to attend to the "Seq Scan(kw, kw='superhero')" node in the query plan. These two nodes have different predicates that lead to completely different results. Furthermore, these results will be further joined with other parts and make the whole view not usable. Capturing the conflicted relationship between nodes helps the model to capture the negative influences of the views.

Node Reordering. The blue dashed line shows the relationship of node reordering. One of the most important reasons that $t_q^{V_k}$ cannot be predicted by $t_{q_i} - t_{v_i} + t_{scan_{v_j}}$ is that the using of a view may change the join order from the original optimal plan to a sub-optimal plan. For example, on the "Nested Loop(t.id = mk.mv_id)" node in the view's plan, the model needs to attend to the "Nested Loop(t.id = mk.mv_id)" node in the query's plan. In the sub-plan represented by that node in the view, the join order is $t \bowtie mk$; while in the query, the join order is $mk \bowtie kw \bowtie t$. If the query wants to use this sub-plan, it needs to adjust its join order to $t \bowtie mk \bowtie kw$, which has a different execution time. Paying close attention to this structure difference relationship helps the model to predict an accurate benefit even when the query needs to change the plan structure drastically to use views.

In a reducer cell, the embedding of the input and the hidden state will be compared with the hidden state of nodes in the encoder to find the correlated subqueries. Then the hidden state of correlated subqueries will be used to enhance rewritten query estimation directly without long-distance information delivering.

To capture different relationships between nodes, we transform the hidden state of a query node with multiple projection matrices W_i^Q, and an MV node with W_i^K. With different projection matrices, the model can capture information of query nodes from different representation subspace. The process can be formulated as below, where $u \in \mathbb{R}^d$ is the output vector of the linear layer in each reducer cell; $K \in \mathbb{R}^{n \times d}$ is the hidden state vectors outputted by the encoder cells where n is the number of the encoder cells. We then compute the similarity scores between nodes by dot production, which is the Ku in Equation 2. We use a $softmax$ function to normalize the scores as a weight. The attention value is the weighted sum of each K. The attention value of each head will be then concatenated and projected by matrix W^O, i.e., the final attention value in Equation 4.

$$Attention(u, K) = softmax(\frac{Ku}{\sqrt{d}})K \tag{2}$$

$$head_i = Attention(W_i^Q u, KW_i^K) \tag{3}$$

$$MultiHeadAttn(u, K) = Concat(head_1, ..., head_h)W^O \tag{4}$$

4.5 Loss Function

We choose the metric of mean absolute percentage error (MAPE) as the judgment of the model performance. Let $\mathbb{E}(t_q^{V_k})$ be the estimated time and $t_q^{V_k}$ be the real execution time. For estimating $t_q^{V_k}$, we optimize:

$$\begin{aligned}&\min \sum_{q_i \in Q, v_j \in V} \frac{\left|\mathbb{E}(t_q^{V_k}) - t_q^{V_k}\right|}{t_q^{V_k}} = \min \sum_{q_i \in Q, v_j \in V} \left|\frac{\mathbb{E}(t_q^{V_k})}{t_q^{V_k}} - 1\right| \\ &= \min \sum_{q_i \in Q, v_j \in V} \left|\exp\left[\ln \mathbb{E}(t_q^{V_k}) - \ln t_q^{V_k}\right] - 1\right|\end{aligned} \tag{5}$$

We transform the expression with "ln" because we apply a logarithm operation to the data before the model training. The execution time of queries in the workload shows a long-tail distribution which is not friendly to the neural network. After applying the logarithm operation and data standardization, it shows a normal distribution which is easier for the neural network to converge. To smooth the gradient values and accelerate the model training, we use the Smooth L1 function as an alternate of the absolute value,

$$Smooth_{L_1}(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < 1 \\ |x| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{6}$$

where x denotes the original L1 distance loss value.

The loss functions for q^{V_k}, q_i and c_{q_i} are:

$$\mathcal{L}_{t_q^{V_k}} = Smooth_{L_1}\left(\ln \mathbb{E}(t_q^{V_k}) - \ln t_q^{V_k}\right) \tag{7}$$

$$\mathcal{L}_{t_{q_i}} = Smooth_{L_1}\left(\ln \mathbb{E}(t_{q_i}) - \ln t_{q_i}\right) \tag{8}$$

$$\mathcal{L}_{c_{q_i}} = Smooth_{L_1}\left(\ln \mathbb{E}(c_{q_i}) - \ln c_{q_i}\right) \tag{9}$$

The total loss function is the sum of the three loss functions:

$$\mathcal{L} = \mathcal{L}_{t_q^{V_k}} + \mathcal{L}_{t_{q_i}} + \mathcal{L}_{c_{q_i}} \tag{10}$$

4.6 Model Training

Training Data. The training data consists of a dataset, e.g. IMDB, and a set of queries $Q = \{q_i\}$. Given the queries and the dataset, we generate MV candidates $V = \{v_j\}$ by the method in Section 3. We can also add some manually written views to simulate the real environment that users may provide their own materialized views. Given Q and V, we first select out the usable views set, V_{q_i}, for each query, q_i. To judge whether v_j can be utilized to optimize q_i, we check whether the relation set of v_j is a subset of the table set of q_i. And for each table, we check whether the selection condition of the view is a superset of the selection condition of the same relation of the query. Given q_i and corresponding V_{q_i}, we sample usable pairs, $\{(q_i, V_k)\}$, where $V_k \subseteq V_{q_i}$ and q_i can be rewritten with V_k and the rewritten query is q^{V_k}. We also generate some negative pairs from $V - V_{q_i}$, where q_i cannot be rewritten with V_k, to enhance the model generalization ability. For these pairs, the benefit is $-t_{scan_{V_k}}$ which is a punishment for scanning the MVs in V_k from the disk. To obtain the ground truth – the execution time t_{q_i} and the result size c_{q_i} of the query q_i, the execution time $t_q^{V_k}$ and the result size of the rewritten query $c_q^{V_k}$, we shuffle the queries and rewritten queries and get the real query plans and the running log from optimizers. We extract the real execution time t_{q_i}, $t_q^{V_k}$ and cardinality c_{q_i} from the query plans. Note $c_q^{V_k}$ is equal to c_{q_i} because the query and the rewritten query have the same results. By the way, a query plan contains all the sub-plans' execution time and cardinality, we also extract this information for training the Encoder part.

Two-step training. It is hard for the encoder and reducer to converge if we train both the two models from random initialization. To accelerate the model convergence we separate the training process into two steps. The encoder model is supervised by the query data Q and the corresponding ground truth t_{q_i}, c_{q_i}. In the first step, we train the encoder model with the loss $\mathcal{L}_{t_{q_i}}$ and $\mathcal{L}_{c_{q_i}}$ only. When the encoder model predicts t_{q_i}, c_{q_i} from query data Q, we go to the second step, which trains the reducer model and fine-tunes the encoder model with training data.

Offline training. We apply incremental training on the model to handle the evolution of workloads. As shown in Figure 3, we update the dataset and train the model offline periodically.

5 MV Selection

Given a set of MV candidates and a query workload, we select a subset of MV candidates to maximize the total benefit while not exceeding a space constraint, where the benefit and cost can be estimated by our benefit/cost estimation model in Section 4. We model this problem as an integer programming problem. Let $e_{ij} \in \{0, 1\}$ denote whether we use v_j to optimize q_i, $x_j \in \{0, 1\}$ denote whether v_j will be materialized, and τ be the space constraint. We optimize:

$$\begin{gathered}\arg\max_{e_{ij}} \sum_{i=1}^{|Q|} \mathcal{B}(q_i, V_i), s.t., \ (\sum_{j=1}^{|V|} x_j |v_j|) \leq \tau, where \\ e_{ij} \in \{0, 1\}, \forall i \in [1, |Q|], j \in [1, |V|], \\ V_i = \{v_j | e_{ij} = 1, j \in [1, |V|]\}, \forall i \in [1, |Q|], \\ x_j = \max\left\{e_{ij} | i \in [1, |Q|]\right\}, \forall j \in [1, |V|]\end{gathered}$$

5.1 Benefit/Cost Estimation of MV Candidates

We estimate the execution time and space cost of each MV candidate generated from the query plan graph in Section 3. We estimate MV candidates using the benefit estimation model introduced in Section 4 by inputting MV candidates' plan tree into the encoder to predict the cardinality c_v and execution time t_v. The space cost, $|v|$, equals to the product of the row size and cardinality. However, if we estimate all the MV candidates one by one, the time complexity of the estimation is $\mathcal{O}(n^2)$ where n is the number of internal nodes in the query plan graph. When the n is large, it is expensive.

Thus, we design an efficient method to estimate all the MV candidates with time complexity of $\mathcal{O}(n)$. We add a super root which connects to all the root nodes of each query so that we can traverse the query plan graph once by the postorder and save all the intermediate outputs of the model on each internal node. The intermediate outputs of an internal node is actually t_v and $|v|$ of the MV candidates subtree rooted at this node.

Moreover, we prune low benefit MV candidates to optimize the MV selection problem. Given the estimation results of MV candidates, we attach a score, w_v, to each MV candidate [45]. w_v can be calculated by $w_v = \frac{f_v \times (t_v - t_{scan_v})}{|v|}$ where f_v is the appearance frequency of the subtree (counted in the merged query plan graph), t_{scan_v} is the time of scanning the result from the disk (calculated by multiplying the size and the unit time of disk accesses). We retain MV candidates with higher w_v until the space cost exceeds the budget.

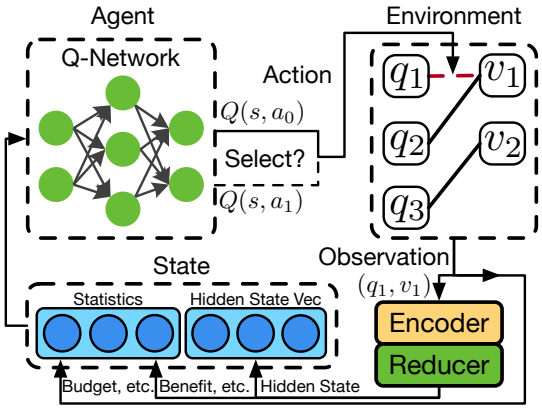


Fig. 7. Encoder-Reducer DDQN Model.

5.2 DDQN Model for MV Selection

It is expensive to use an ILP solver to solve the MV selection problem because there are a large number of queries and MVs. Thus, we utilize the deep reinforcement model to address this problem. `DDQN` model is an effective model among RL models, but there are two challenges: (1) The number of variables of the MV selection problem is dynamic varied among different workloads and it is hard for state representation. (2) It is hard to encode the relation between MVs, i.e., whether two MVs can be used jointly for optimizing a query. To address the first challenge, we design an iterative method for selecting MVs so that we can split the global state into many fixed sub-states. To address the second challenge, we design a new state representation method which can include the rich information in queries and MVs. DDQN model contains two parts, the agent and the environment. The agent plays the "game" of solving the MV selection problem, and the environment provides the simulation of the problem-solving process and gives the rewards. Our `Encoder-Reducer DDQN` model (`ERDDQN`), is shown in Figure 7. The model has six main parts: environment, agent, state, reward, action and policy.

Environment. The MV selection state is modeled as a bipartite graph where nodes at the left side represent queries in Q, nodes at the right side represent views in V, and the edges between them represent $\{e_{ij}\}$. The environment module provides observations, e.g. selection state and total benefit, for the agent.

State. We propose a state representation that includes the semantic vector outputted by the `Encoder-Reducer` model besides the features extracted from the environment observation. This semantic vector provides rich information about the pair (q_i, v_j) such as MV's structure which can be used to judge whether two MVs conflict on one query. We estimate the MV size by multiplying its estimated cardinality and its row width. The `Encoder-Reducer` model will also estimate the cardinality along with the execution time by inputting its expression into the encoder.

Agent. It consists of two neural networks and the experience replay mechanism. The two neural networks can be seen as a function $W^*(s, a)$ which approximates the action-value function $W(s, a)$. $W(s, a)$ equals to the maximum feedback, G, after choosing action a (changing e_{ij} to 0 or 1) at state s. G is positively correlated to the final total benefit. We maximize the final total benefit by obtaining a high G_t at each turn, where r is the reward, γ is the decay and t is the rounds of the iteration.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \tag{11}$$

Reward. To make the G positively correlated to the final total benefit, we define the reward as the change of total benefit after each action. With the decay, the more steps we take in the solving procedure, the smaller G we get. Therefore, the model will achieve the optimal solution as soon as possible.

Action. At each iteration, environment takes an edge (q_i, v_j), i.e., using v_j to optimize q_i, from all edges and asks agent whether to use this edge. The answer is the action. If yes, e_{ij} for this pair will be set to 1, i.e., putting this pair in the global selection state. If not, e_{ij} for this pair will be set to 0, i.e., removing this pair from the global selection state.

Policy. Agent acts based on the policy of obtaining higher feedback. Agent chooses the action with the highest feedback.

Solving Procedure. Environment and agent work iteratively. Agent takes actions and changes the global selection state according to the policy and observation of the environment. Once the selection state converges, or we reach the maximum iterations, the global selection state with the highest benefit will be saved as the final solution.

Training. `ERDDQN` model trains with the experience replay mechanism, which builds an experience pool and samples experience tuples $(s_t, a_t, s_{t+1}, r_{t+1})$ for training, where s_t denotes the current state, a_t denotes the action it chooses at s_t, s_{t+1} denotes the next state after applying a_t, and $W^*(s_t, a_t)$ denotes the estimated action-value of the Q-network. Let lr be the learning rate. We update the Q-network parameters by the iteration:

$$W^*(s_t, a_t) = W^*(s_t, a_t) + lr[r_{t+1} + \gamma \max_a W^*(s_{t+1}, a) - W^*(s_t, a_t)] \tag{12}$$

For each experience $(s_t, a_t, s_{t+1}, r_{t+1})$, we use the estimated action-value $(r_{t+1} + \gamma \max_a W^*(s_{t+1}, a))$ at $(t+1)$-th round to update the estimated action-value at t-th round. Iteratively, the Q-network approximates to the true action-value. To make the experience pool cover more possible states, we let the model choose random action to take a random walk in the state space at the early stage of training. The probability of taking random action is 0.9. After hundreds of iterations, it decays to 0.1.

6 Query Rewrite

This section presents how to utilize a set of given MVs to rewrite a query. Given a query q, and the set of MVs, $V = \{v_j\}$, we select a subset of views, $V_k \subseteq V$, to answer q such that the performance of answering q with V is optimized. For example, as shown in Figure 2, given q_1 and corresponding MVs $\{v_1, v_2, v_3\}$, we select $\{v_1, v_3\}$ to optimize q_1, and the rewritten query is $q_1^{\{v_1, v_3\}}$. There are two main challenges: (1) The first is how to select a good V_k which is valid and the benefit is maximum. The conflicts between MVs will make V_k invalid. For example, v_2 and v_3 are in conflict with each other, because they cannot be used to rewrite the query at the same time. We need to avoid invalid views when searching possible V_k. We propose a selection method that first prunes invalid MVs, and then reuse the MVs selection model to estimate the benefits and select MVs for the query. (2) The second is how to reduce the latency of selecting MVs for the query. The latency of a query is sensitive to the MVs selection time. However, there are $2^{|V|}$ view subsets, V_k, for the query. Estimating all of them is expensive. Thus, we optimize the benefit estimation in the selection method with fewer redundant computations.

MVs Selection for a Query. We reuse the MVs selection model to select MVs for a query. The selection problem in this section

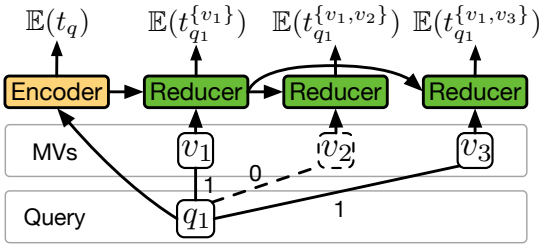


Fig. 8. MV selection for a query.

is a special case of the MVs selection problem in Section 5. As shown in Figure 8, there is only one query in the query set, and the view set consists of MVs instead of candidates. The budget is infinity because the views are already materialized. Accordingly, the workload total benefit, which is the optimization target of the MVs selection model, is equal to the single query benefit. Thus, MVs selection model gives the solution that maximizes the benefit of the rewritten query. Moreover, invalid solutions will be avoided because useless MVs give negative benefit as the punishment of MVs scanning time.

Removing Redundant Estimation. We cache the hidden state vectors and attention values among the `Encoder-Reducer` model during the estimation of benefit to reduce redundant computations. For example, in Figure 8, we consider three actions: estimating the benefit of using $\{v_1\}$ and selecting v_1; estimating $\{v_1, v_2\}$ and discarding v_2; estimating $\{v_1, v_3\}$ and selecting v_3. If we estimate these three actions separately, q will be input into Encoder model for three times, and v_1 will be input into Reducer model for three times. Instead, by caching the hidden state vectors of Encoder, we just input q once for the whole round. Similarly, the hidden state vectors of Reducer on v_1 can be cached and reused at later estimation, and thus v_1 just needs to be input once. Thus the estimation time will be reduced to about $\frac{1}{n}$ of the original, where n is the number of MV candidates.

7 Experiment

We have conducted a set of experiments to evaluate our `AutoView` from three aspects. (1) The effectiveness of our benefit estimation model, `Encoder-Reducer` model. (2) The effectiveness of our MV selection model, `ERDDQN` model. (3) The efficiency of our query rewrite method.

7.1 Experimental Setting

Datasets. We use the real world dataset IMDB with several workloads. IMDB is designed in snowflake schema with three tables, title (movie title), name (person name) and movie_companes. IMDB has a size of 3.7GB with 21 tables. The largest table has a size of 1.4GB and 36 millions rows.

We use four query workloads–JOB [21], Extended JOB, Synthesis, and Scale [19], as shown in Table 1. The JOB workload contains 113 queries with 16 joins at most. The Extend JOB dataset contains 35K queries. It is a generalization of the JOB dataset. By modifying the predicates of the queries, we generate more different queries. We use it to pre-train the encoder model for fine-tuning and help the model to converge in the training process. The synthesis workload contains 5000 queries with 2 joins at most and only numeric predicates. This workload has lower potential optimization chances for MVs. The scale workload contains 500 queries with 4 joins at most and only numeric predicates. It is more complex than the synthesis workload. The TPC-H dataset contains 10K queries generated from 22 templates with 7 joins at most. The TPC-DS dataset contains 990 queries generated from 99 templates with 17 joins at most. Given

TABLE 1
Workloads Dataset.

Dataset	Queries	Join Number
JOB	113	3-16
Extended JOB	35615	3-16
Synthesis	5000	0-2
Scale	500	0-4
TPC-H	10000	0-7
TPC-DS	990	0-17

workloads, we generate MV candidates and sample query-MVs pairs as introduced in Section 4. We execute these workloads to obtain ground truth of execution time and cardinality. We split queries in each workload into training and test dataset with the ratio 8:2. We split the training dataset into training and validation dataset with the ratio 9:1 in the 10-fold cross-validation.

Evaluation metrics. We evaluate benefit estimation models based on the error of prediction. We use Mean Absolute Percentage Error (MAPE) as the metrics. Specifically, for predicted values $\hat{\boldsymbol{y}} = \{\hat{y}^i\}$ and ground true values $\boldsymbol{y} = \{y^i\}$, $MAPE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{\hat{y}^i - y^i}{y^i}\right|$. We evaluate MV selection models by the sum of queries' latency, $t_{q_i}^{V_k}$, in query workload. For convenience, we use benefit, $\mathcal{B}(q_i, V_k) = t_{q_i} - t_{q_i}^{V_k}$, in comparison.

Environment. We use a machine with Intel(R) Xeon(R) CPU E5-2630, 128GB RAM, and GeForce RTX 2080.

7.2 MVs Training Data

The method of generating training data is introduced in Section 4. To improve the model generalization ability and better benchmark the model's performance. The training data should satisfy the properties as follows:

(1) Containing complex queries. The dataset should contain enough complex queries which can be optimized by MVs, for example, queries with at least 5 tables and 3 filter conditions.

(2) Containing queries with similar structure. To provide redundant computation that MVs optimize, queries should have 3 or more other queries with similar structure.

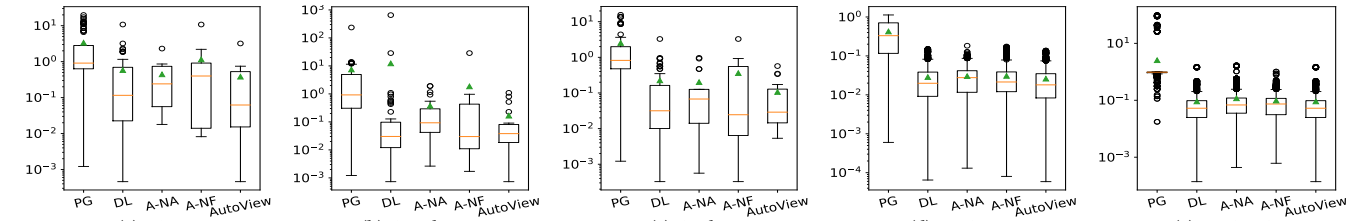
(3) Containing positive and negative samples. If V_k can be used to optimize q_i but cannot improve the performance, (q_i, V_k) is a negative sample; otherwise (q_i, V_k) is a positive sample. A robust dataset should contain both positive and negative samples.

(4) Containing multiple optimization choices. To train the selection ability of `ERDDQN`, the dataset should contain queries that can use different MVs, i.e., a query can be optimized by multiple MVs and multiple MVs can be jointly used.

According to the properties, we choose JOB and it's derived workloads as the dataset to evaluate the model performance.

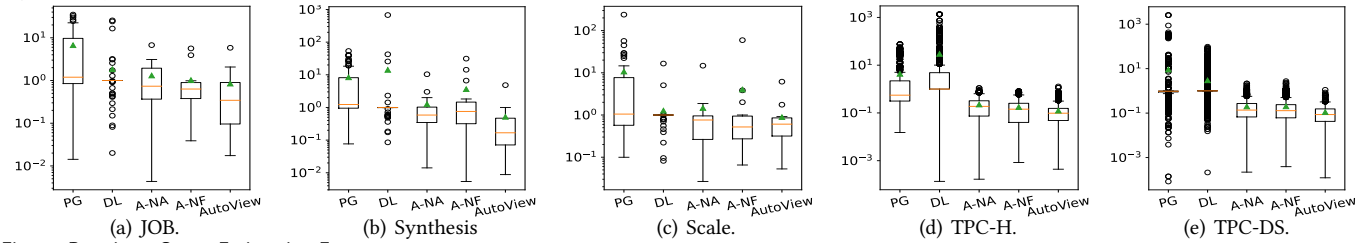
7.3 Effectiveness on Benefit Estimation

Methods. We evaluate and compare the following five methods. (1) `PG`: PostgreSQL's optimizer gives the estimation of execution time based on the unit of disk page fetches. We transform it to execution time by multiplying the ratio: $\frac{Mean\{real\ time\}}{Mean\{estimated\ time\}}$. (2) `DL`: We use the Encoder model in Encoder-Reducer as the DL model to estimate the execution time of queries and MVs and use $t_q - t_{V_k} + t_{scan_{V_k}}$ as the rewritten query time. (3) `AutoView-NA`: Our `Encoder-Reducer` model without multi-head attention. (4) `AutoView-NF`: Our `Encoder-Reducer` model without fine-tuning. (5) `AutoView`: Our `Encoder-Reducer` model with multi-head attention and fine-tuning. The encoder model is pre-trained on the Extended JOB workload, then fine-tuned on other workloads. We train and evaluate the model by 10-fold cross-validation. We split the dataset into 10 partitions.



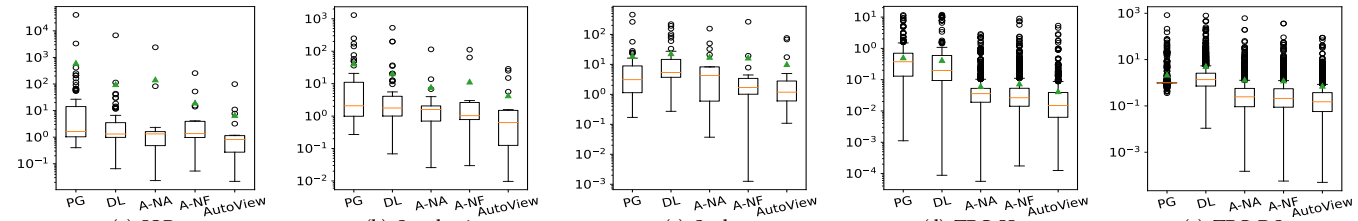
(a) JOB. (b) Synthesis. (c) Scale. (d) TPC-H. (e) TPC-DS.

Fig. 9. Query Estimation Error.



(a) JOB. (b) Synthesis (c) Scale. (d) TPC-H. (e) TPC-DS.

Fig. 10. Rewritten Query Estimation Error.



(a) JOB. (b) Synthesis. (c) Scale. (d) TPC-H. (e) TPC-DS.

Fig. 11. Benefit Estimation Error (AutoView-NA (A-NA): AutoView without multi-head attention; AutoView-NF (A-NF): AutoView without fine-tuning; The box boundaries are at the 25th/50th/75th percentiles; The traingles are mean values).

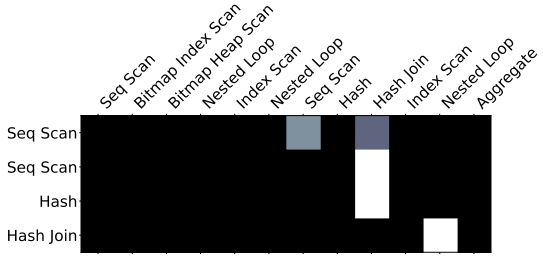
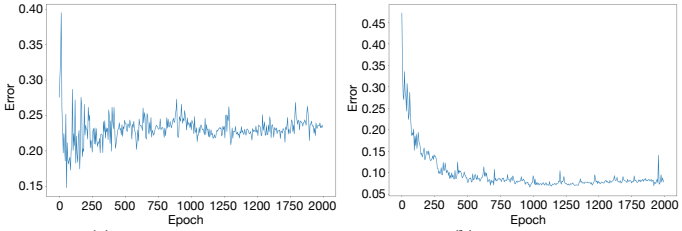


Fig. 12. Attention Weight Visualization.



(a) AutoView-NF. (b) AutoView.

Fig. 13. Converging Time.

Each time choose 9 partitions for training and use the last one for evaluation. We evaluate the execution time prediction on queries, rewritten queries and benefits on test dataset using the MAPE metric.[2] The results are shown in Figure 9 10 11. We use Adam [18] as the optimizer for model training.

Attention vs Non-Attention. `AutoView` outperforms `AutoView-NA` on the estimation of rewritten query and benefit. The results show that multi-head attention in the reducer model alleviates the problem of long-term information delivering and improves the model's ability to capture the correlation between query and views. As shown in Figure 12, attention captures the similarity between an MV and the corresponding subquery in a query and has a higher weight between the nodes they are rooted.

Fine-tune vs Non-Fine-tune. In Figure 9(a)(b)(c), `AutoView` outperforms `AutoView-NF`, especially on the mean estimation error of query execution time because pre-training on encoder model improves its generalization ability. Moreover, `AutoView` converges faster and is more stable than `AutoView-NF` as shown in Figure 13, because the encoder is pre-trained instead of randomly initialized which helps the training of reducer and reduces the possibility of overfitting.

AutoView vs PG. In Figure 11, `PG` performs the worst on estimation of query, rewritten query and benefit. For more intuitive comparison, we draw the distribution of estimated execution time and real time in Figure 14. We can see that `PG` usually gives same estimation on queries even they have very different execution time. This is because `PG` performs bad when analyzing some complicated predicates such as strings. Ignoring these predicates will not affect the estimation result but largely affects the performance of the queries.

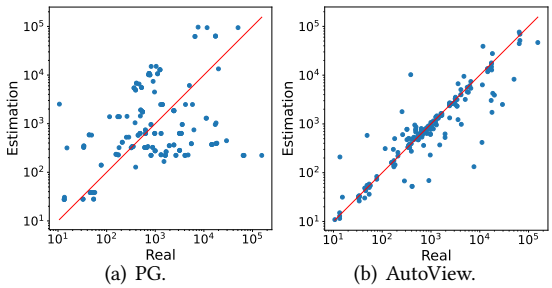
AutoView vs DL. In Figure 10 11, `AutoView` outperforms `DL` on estimation of rewritten queries and benefits. When estimating rewritten queries, `DL` predicts an execution time of 0 on more than half cases, because `DL` tends to predict higher benefit on the using of MVs, but, in fact, MVs do not always optimize the queries. This result shows that capturing the correlation between MVs and queries significantly improves the estimation.

Summary. Our `Encoder-Reducer` model, multi-head attention, and fine-tuning can improve the quality of benefit estimation.

7.4 Effectiveness on MV Selection

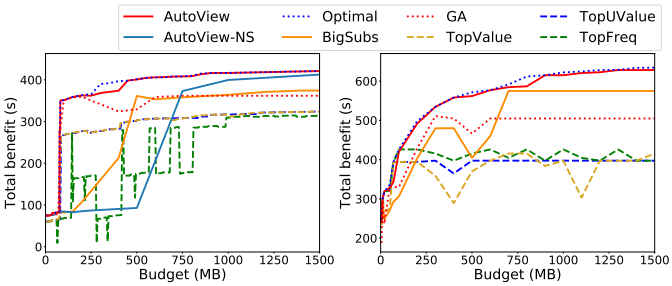
To evaluate the performance of our MVs selection module, we evaluate the module on different budgets and compare our `ERDDQN` model with `BigSubs` and traditional algorithms. (1) `TopValue`: A greedy algorithm, using the metric of sum benefit

2. For a proper comparison, we filtered out entries whose $t_{q_i} < 10ms$ or $t_{q_i,v_j} < 10ms$ or $\mathcal{B}(q_i, v_j) < 10ms$ that greatly impact the MAPE metric but have less influence on the workload execution time.



(a) PG. (b) AutoView.

Fig. 14. Distribution of Estimated Execution Time.



(a) JOB (b) TPC-H

Fig. 15. MV selection comparison.

for each MV. The sum benefit of an MV is the sum of the benefit of using this MV answering each query. MVs with top sum benefit will be selected within the budget. (2) `TopUValue`: A greedy algorithm, using the metric of unit benefit, sum benefit/size, for each MV. (3) `TopFreq`: A greedy algorithm, using the metric of frequency for each MV. (4) `BigSubs` [16]: An iterative method that optimizes views and queries with heuristic and ILP respectively. (5) `GA` [15]: Applying genetic algorithm to choose MVs. (6) `Optimal`: Using ILP solver to get an optimal solution. (7) `AutoView-NS`: Our `ERDDQN` model without the semantic vector in state representation from `Encoder-Reducer` model. (8) `AutoView`: Our `ERDDQN` model with the semantic vector from the `Encoder-Reducer` model.

We compare the effectiveness of these methods on optimizing the JOB workload under different budget size from 5MB to 1500MB. The result is shown in Figure 15.

AutoView vs Heuristics. `AutoView` outperforms `TopValue`, `TopUValue` and `TopFreq`. The reason is two-fold. First, `AutoView` can estimate the benefit of utilizing multiple MVs while the greedy methods approximate the benefit by summing up the individual benefit which is not accurate. Second, the performances of greedy methods are not stable during the increase of budget while the `AutoView` grows stable. The reason is that greedy methods are more likely to fall in local optimum, and they select MVs with higher benefit or unit benefit, but higher benefit leads to larger size that waste the budget. `AutoView` adjusts the earlier selection in the subsequent iterations. When an MV results in local optimum, `ERDDQN` will prefer not to select it.

AutoView vs GA. `AutoView` outperforms `GA` by 17.7%. We observe that `GA` performs well on small budgets. However, on large budgets, which mean large problem scales, it is not stable similar to heuristic methods. Moreover, to get a comparable result, `GA` consumes more time than `AutoView` due to more iterations and large populations.

AutoView vs Bigsubs. `AutoView` outperforms `BigSubs` by 13.8%. The reason is two-fold. Firstly, `BigSubs` flips a view by the probability that relies on the benefit and cost of this view. The probability cannot well reflect the correlation between views.

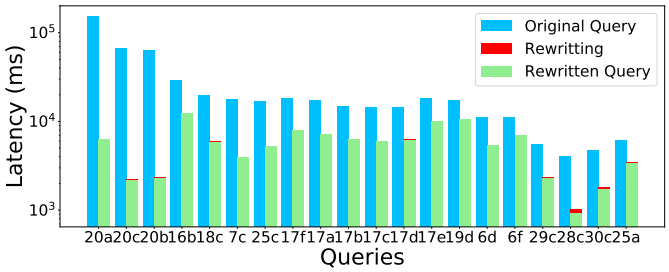


Fig. 16. Example queries rewritten in JOB workload.

While `AutoView` can capture the correlation between views. Secondly, `BigSubs` may fall in local optimal. While `AutoView` learns to select views, and avoids local optimal solution as low rewards make the model to change the action.

AutoView vs Optimal. We use an ILP solver to get an optimal solution for MV selection problem formulated in Section 5. The results show that in most cases `AutoView` outputs the optimal solution. On space budgets of 300, 350 and 400MB, `AutoView` has a little gap with the optimal solution. It shows that `ERDDQN` needs more exploration to cover more cases.

AutoView vs AutoView-NS. `AutoView` outperforms `AutoView-NS` 4 times on total benefit under the budget of 500MB, because `AutoView-NS` selects MVs that are in conflict and results in waste of budget. However, `AutoView` learns from the semantic vectors and captures the conflict relationship.

Summary. `ERDDQN` and semantic vector can improve the quality.

7.5 Efficiency on Query Rewriting

We evaluate the latency of our query rewriting method. We rewrite the queries in the JOB workload and compare the latency of original queries with the total latency of rewritten queries and rewriting. Figure 16 shows the result of 20 queries. We observe that the query rewriting latency is nearly a constant because it relies on the size of query/MV plans and the number of available MVs which vary little among queries/MVs. The average query rewriting latency in JOB workload is 64.75ms which is small compared to the slow queries. The slowest query in JOB workload is "20a" with 9 joins and a latency of 154,902.81ms. It is optimized to 6303.36ms with a query rewriting latency of 65.28ms. Thus, it is beneficial to rewrite slow queries and our query rewriting method is efficient and has a low latency.

8 Conclusion

We proposed an autonomous materialized view generation system `AutoView`. We devised the `Encoder-Reducer` model for benefit/cost estimation. To obtain accurate benefit/cost estimation, the `Encoder-Reducer` estimation model serialized and encoded the queries and views. We adopted an RNN model to embed them as semantic vectors by capturing the semantic information of queries and views. The encoder *encoded* a query into a semantic vector, and the reducer *reduced* views from a query and predicted the benefit. The `Encoder-Reducer` was very important to estimate the cost/benefit of using a view to answer a query. Moreover, we proposed a double deep Q-learning network to select MVs based on the estimate of cost/benefit. Finally, we used the `ERDDQN` model to select views for MV-aware query rewriting. Experimental result showed that `AutoView` outperformed existing methods, and semantic information significantly improved the performance and stability of `ERDDQN` model in MVs selection problem.

Acknowledgments

This paper was supported by NSF of China (61925205, 62072261), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB 2000*, pages 496–505. Morgan Kaufmann, 2000.
[2] R. Ahmed, R. G. Bello, A. Witkowski, and P. Kumar. Automated generation of materialized views in oracle. *Proc. VLDB Endow.*, 13(12):3046–3058, 2020.
[3] H. Azgomi and M. K. Sohrabi. A novel coral reefs optimization algorithm for materialized view selection in data warehouse environments. *Appl. Intell.*, 49(11):3965–3989, 2019.
[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In Y. Bengio and Y. LeCun, editors, *ICLR 2015, May 7-9, 2015, Conference Track Proceedings*, 2015.
[5] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *VLDB*, pages 659–664, 1998.
[6] A. Boukorca, L. Bellatreche, and A. Cuzzocrea. SLEMAS: an approach for selecting materialized views under query scheduling constraints. In *COMAD 2014, Hyderabad, India, December 17-19, 2014*, pages 66–73. Computer Society of India, 2014.
[7] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. R. Chowdhury. Reuse-based optimization for pig latin. In *CIKM*, pages 2215–2220. ACM, 2016.
[8] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.
[9] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *ACL*, pages 1724–1734, 2014.
[10] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.355, 2014.
[11] T. Dökeroglu, M. A. Bayir, and A. Cosar. Robust heuristic algorithms for exploiting the common tasks of relational cloud database queries. *Appl. Soft Comput.*, 30:72–82, 2015.
[12] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. *SIGMOD Rec.*, 30(2):331–342, 2001.
[13] A. Gosain and K. Sachdeva. Handling constraints using penalty functions in materialized view selection. *Int. J. Nat. Comput. Res.*, 8(2):1–17, 2019.
[14] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.
[15] J. Horng, Y. Chang, and B. Liu. Applying evolutionary algorithms to materialized view selection in a data warehouse. *Soft Comput.*, 7(8):574–581, 2003.
[16] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.
[17] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *SIGMOD*, pages 191–203, 2018.
[18] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2015.
[19] 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.
[20] A. Kumar and T. V. V. Kumar. Materialized view selection using self-adaptive perturbation operator-based particle swarm optimization. *Int. J. Appl. Evol. Comput.*, 11(3):50–67, 2020.
[21] 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.
[22] G. Li, X. Zhou, S. Li, and B. Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.*, 12(12):2118–2130, 2019.
[23] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic view materialization with deep reinforcement learning. *CoRR*, abs/1903.01363, 2019.
[24] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.
[25] R. C. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, 2019.
[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.
[27] M. C. Mouna, L. Bellatreche, and B. Narhimene. HYRAQ: optimizing large-scale analytical queries through dynamic hypergraphs. In B. C. Desai and W. Cho, editors, *IDEAS 2020, August 12-14*, pages 17:1–17:10. ACM, 2020.
[28] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *SIGMOD*, pages 4:1–4:4, 2018.
[29] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, editors, *IEEE*, pages 520–531. IEEE Computer Society, 2014.
[30] T. Phan and W. Li. Dynamic materialization of query views for data warehouse workloads. In G. Alonso, J. A. Blakeley, and A. L. P. Chen, editors, *ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 436–445. IEEE Computer Society, 2008.
[31] Y. N. Silva, P. Larson, and J. Zhou. Exploiting common subexpressions for cloud query processing. In A. Kementsietsidis and M. A. V. Salles, editors, *IEEE*, pages 1337–1348. IEEE Computer Society, 2012.
[32] M. K. Sohrabi and H. Azgomi. Evolutionary game theory approach to materialized view selection in data warehouses. *Knowl. Based Syst.*, 163:558–571, 2019.
[33] M. K. Sohrabi and V. Ghods. Materialized view selection for a data warehouse using frequent itemset mining. *J. Comput.*, 11(2):140–148, 2016.
[34] J. Sun and G. Li. An end-to-end learning-based cost estimator. In *VLDB*, 2019.
[35] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang. Learned cardinality estimation: A design space exploration and A comparative evaluation. *Proc. VLDB Endow.*, 15(1):85–97, 2021.
[36] Y. Tao, Q. Zhu, and C. Zuzarte. Exploiting common subqueries for complex query optimization. In *Collaborative Research*, page 12, 2002.
[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.
[38] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A normalizing flow based cardinality estimator. *Proc. VLDB Endow.*, 15(1):72–84, 2021.
[39] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.
[40] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. pages 1297–1308. IEEE, 2020.
[41] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, pages 1501–1512, 2020.
[42] C. Zhang and J. Yang. Genetic algorithm for materialized view selection in data warehouse environments. In *DWKD*, pages 116–125, 1999.
[43] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *TKDE*, 2020.
[44] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *PVLDB*, 2022.
[45] D. C. Zilio et al. Recommending materialized views and indexes with IBM DB2 design advisor. In *ICAC*, pages 180–188, 2004.

Yue Han is currently a PhD student in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests include materialized views and machine learning for database.

Guoliang Li is currently working as a professor in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests mainly include learning models for database system, data cleaning and integration, spatial databases and crowdsourcing.

Haitao Yuan is currently a PhD student in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests include spatial-temporal data manangement, data mining and machine learning for database.

Ji Sun is currently a PhD student in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests include query processing and machine learning for database.