# An Autonomous Materialized View Management System with Deep Reinforcement Learning

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

***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 deep reinforcement learning model to select high-quality MVs, which enriches the state representation with query and MVs' embedding. 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.**

## I. 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 [10], [13], [14], 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.

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], [12] 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 use a recurrent neural network (RNN) model, `Encoder-Reducer`, to estimates queries and views and embed them as embedding vectors. 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 propose a reinforcement model to select MVs for materialization, and integrate the embedding vectors of queries and MVs into the model.

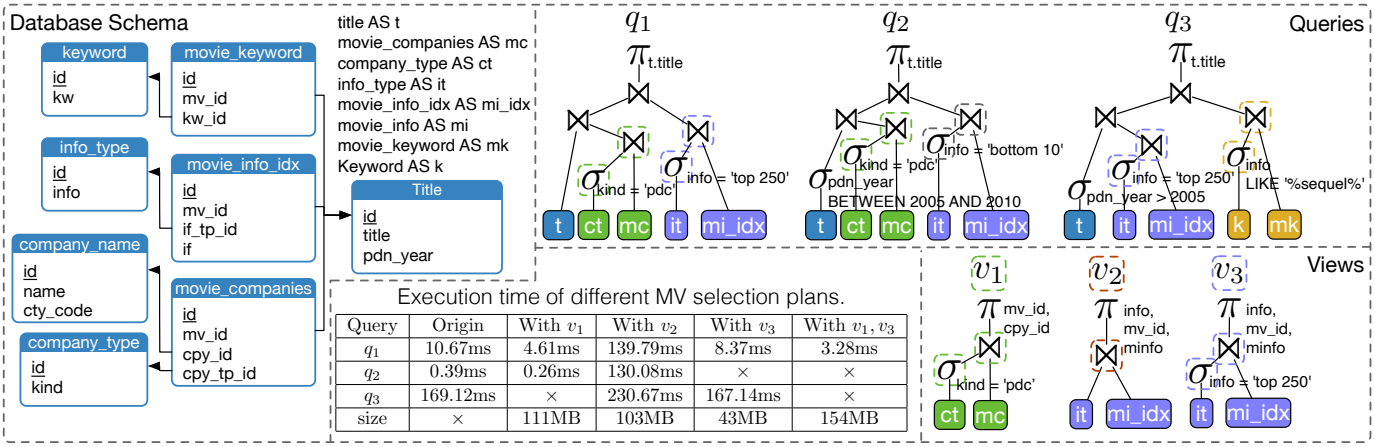
(3) Our experimental results on real datasets showed that our method significantly outperformed existing solutions.

## II. `AutoView` OVERVIEW

### A. 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

*Guoliang Li is the corresponding author. This work was supported by NSF of China (61632016,61925205,62041204), National Key R&D Program of China (2020AAA0104500), Huawei, BNRist, and TAL education.




| Query | Origin | With $v_1$ | With $v_2$ | With $v_3$ | With $v_1, v_3$ |
|---|---|---|---|---|---|
| $q_1$ | 10.67ms | 4.61ms | 139.79ms | 8.37ms | 3.28ms |
| $q_2$ | 0.39ms | 0.26ms | 130.08ms | × | × |
| $q_3$ | 169.12ms | × | 230.67ms | 167.14ms | × |
| size | × | 111MB | 103MB | 43MB | 154MB |

Fig. 1. MV selection example.

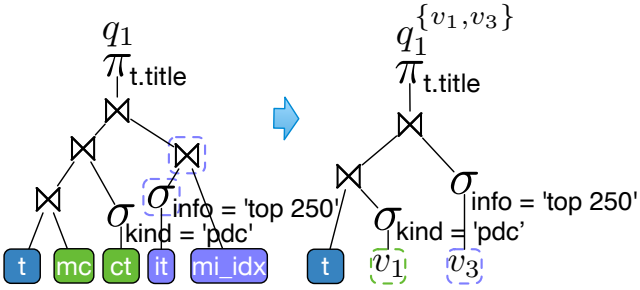



Fig. 2. Query rewriting example.

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.

### B. 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.

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

**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 [2], [3], [9]. 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 III.

**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:

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

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

There are several ways to estimate the benefit. The most straightforward way is utilizing the cost estimation of optimizer. The difference of the COST of a query and the rewritten query can be the estimation of benefit. Due to that optimizer has a large error on the estimation, we can use deep learning model [8] as cost estimation. Furthermore, we propose an RNN model, `Encoder-Reducer`, to estimate the benefit and embed queries and MVs. `Encoder-Reducer` will be introduced in future works.

**MV Selection.** Given a space budget $\tau$, 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 IV.

**MV-aware Query Rewriting.** Given a query, if the query can be optimized using the MVs, we use our estimation model

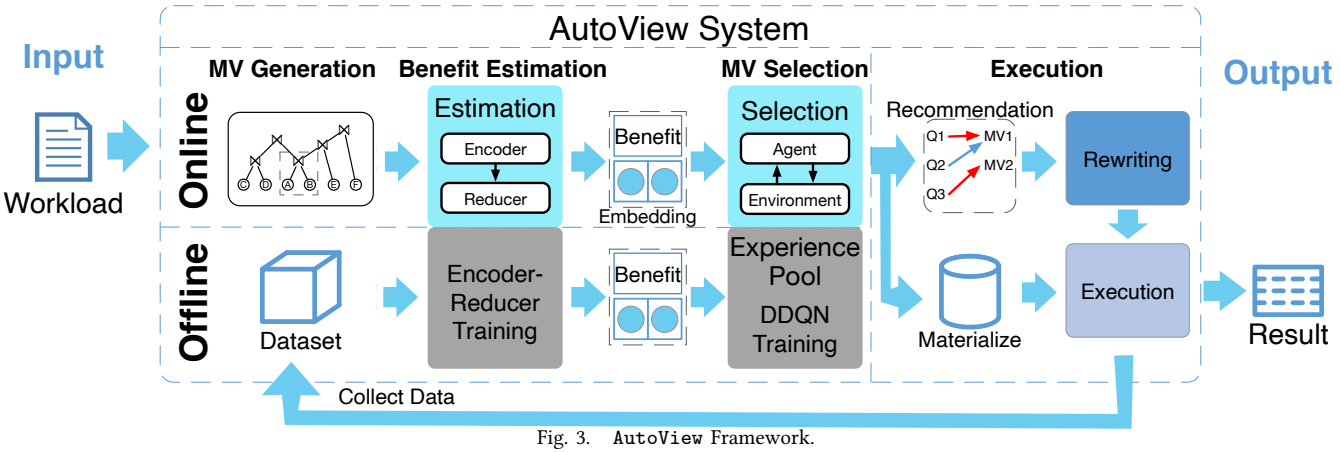



Fig. 3. `AutoView` Framework.

to select the most appropriate views and rewrite the query using 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 $v_3$.

### C. Related Work

Traditional MV selection methods are usually heuristic methods, while reinforcement learning (RL) methods are introduced in recent researches. Jindal et al. [4] propose an iteration based heuristic method, `BigSubs`, to select MVs. However, it cannot use the experience of history workloads and result in unstable results. Liang et al. [6] propose an RL method, DQM, to predict the benefit and learn a policy for view creation and eviction. This method observes real runtimes of queries in the DBMS instead of using benefit estimation or heuristics. Thus, when the distribution of data or workload changes, they rerun the workloads and retrain the model that results in an expensive model training cost in the cold-start step. Yuan et al. [12] propose an RL method to select and create MVs for workloads. They regard the whole MVs selection state as a fixed-length state of the DQN model [7]. Thus, it retrains the DQN model to fit new workloads. Moreover, they use the assumption of infinity 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.

## III. 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 an explicit subexpression.

The second method represents each SQL query as a tree-structured query plan and finds high-beneficial subtrees. Query plans in tree structure provide more choices for selecting view candidates. However, each query may have many available query plans for different join order. This results in a large number of different subtrees. We keep the query plan which is recommended by the optimizer, because it has high possibility to contain a high-beneficial MV. However, there are still a large number of subtrees. To address this issue, 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. 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. An 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 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. 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) [11]. 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. 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 in the query graph (i.e., the number of queries that merge into this node), and take the node (i.e., the corresponding subtree rooted at the node) with high benefit as the MV candidates.

## IV. 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. 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 $\tau$ be the space constraint. We optimize:

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

### A. 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 III, and 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 [15]. $w_v$ 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), $t_v$ is the execution time (estimated by `Encoder-Reducer`), $|v|$ is the estimated size of the result of the subtree, $t_{scan_v}$ 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.

### B. 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 4. The model has six main parts: environment, agent, state, reward, action and policy.

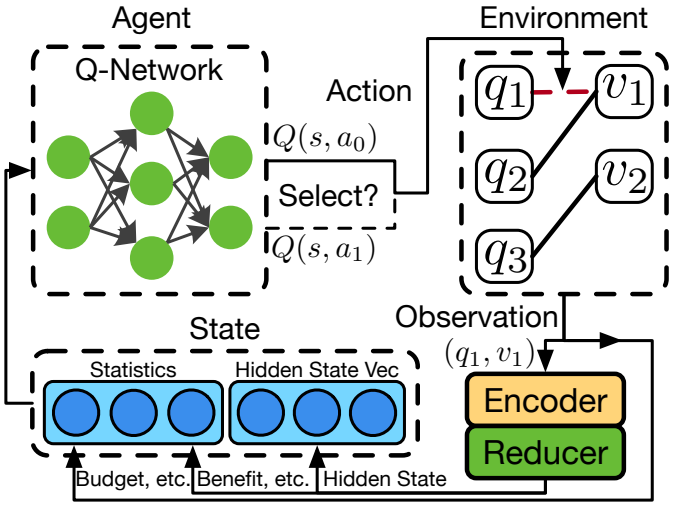



Fig. 4. Encoder-Reducer DDQN Model.

**Environment.** It stores the global MV selection state and calculates the total benefit during the iterative solving procedure. Environment models the global selection state 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 for the agent module.

**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 $W(s,a)$, where $W(s,a)$ equals the expected 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 $G$.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \quad (2)$$

where $r$ is the reward, $\gamma$ is the decay and $t$ is the rounds of the iteration.

**Reward.** To make the feedback positively correlated to the final total benefit, we define the reward as the change of total benefit after each action. $G_0$ will be the max total benefit we can achieve without the decay. With the decay, the more

steps we take in the solving procedure, the lower feedback we get. Therefore, the model will achieve the optimal solution as soon as possible to obtain higher feedback.

**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.

**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. Note that traditional methods cannot handle the conflict and cooperation problems because they regard each MV as individual one. We estimate the MV size by multiplying its estimated cardinality and its row width. We input MVs into `Encoder` and obtain the estimated cardinality.

**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:

$$W^*(s_t, a_t) = W^*(s_t, a_t) + lr[r_{t+1} + \gamma \max_a W^*(s_{t+1}, a) - W^*(s_t, a_t)] \quad (3)$$

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 $W^*(s_t, a_t)$ 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 decreases from 0.9 to 0.1 during iterations.

## V. Experiment

We have conducted a set of experiments to evaluate our `AutoView` from two aspects. (1) The effectiveness of our MV selection model, `ERDDQN` model. (2) The efficiency of our query rewrite method.

### A. Experimental Setting

**Datasets.** We use the real 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 [5] for model training and evaluation. The JOB workload contains 113 queries with 16 joins at most. Given workloads, we generate MV candidates and sample query-MVs pairs, and 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.

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

### B. MVs Training Data

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 provide optimization possibility for MVs, e.g., complex queries should contain 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 $V_k$ optimize $q_i$ with no performance improvement, $(q_i, V_k)$ is a negative sample; otherwise $(q_i, V_k)$ is a positive sample. A robust dataset contains both positive and negative samples.

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

### C. 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 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` [4]: An iterative method. It optimizes views and queries separately in each iteration. In each iteration, it first flips selection state of views by a specified probability, then select views to optimize queries using an integer linear programming solver. (5) `AutoView-NS`: Our `ERDDQN` model without the semantic vector in state representation from `Encoder-Reducer` model. (6) `AutoView`: Our `ERDDQN` model with the semantic vector from the `Encoder-Reducer` model. All methods are based on our benefit/cost estimation.

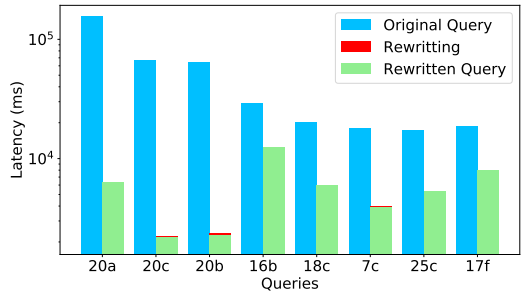



Fig. 5. Example queries rewritten in JOB workload.

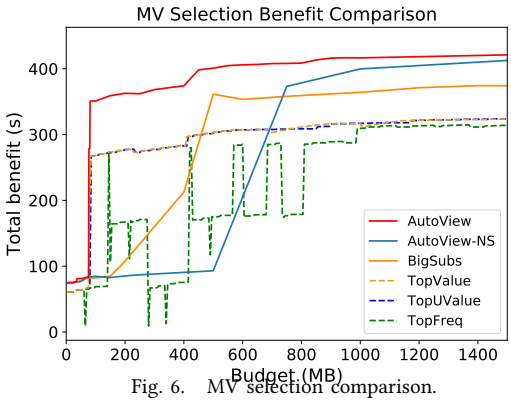



Fig. 6. MV selection comparison.

We compare the effectiveness of these methods on optimizing the JOB workload under different budget size from 5MB to 2000MB. The result is shown in Figure 6.

**AutoView vs Heuristics.** `AutoView` outperforms `TopValue`, `TopUValue` and `TopFreq`. The reason is two-fold. Firstly, `AutoView` can estimate the benefit of utilizing multiple MVs while the greedy methods cannot because it is expensive to enumerate all the combination, which is exponential. Thus, greedy methods approximate the benefit by summing up the individual benefit which is not accurate. Secondly, 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. While `AutoView` adjusts the earlier selection in the subsequent iterations. When an MV results in local optimum, `ERDDQN` model will prefer not to select this MV.

**AutoView vs Bigsubs.** `AutoView` outperforms `BigSubs` by 12.5%. 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. 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 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 with each other so that some MVs become useless and lost its benefit due to other MVs. However, `AutoView` can capture the correlation between MVs, benefiting from the semantic vectors of query and views in the state representation. `AutoView` learns from the semantic vectors and tends to avoid selecting MVs that are in conflict with existed ones.

**Summary.** Our `ERDDQN` model and semantic vector can improve the quality of MV section.

### D. 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 5 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.

## VI. Conclusion

We proposed an autonomous materialized view generation system `AutoView`. We proposed the DQN model with the semantic information of queries and MVs' embedding that leaded to higher and stable performance on MVs selection. Experimental result showed that `AutoView` outperformed existing methods.

## References


[1] R. Ahmed, R. G. Bello, A. Witkowski, and P. Kumar. Automated generation of materialized views in oracle. *VLDB*, 2020.

[2] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[3] 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.

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

[5] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 2015.

[6] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic view materialization with deep reinforcement learning. *CoRR*, abs/1903.01363, 2019.

[7] 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.

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

[9] Y. Tao, Q. Zhu, and C. Zuzarte. Exploiting common subqueries for complex query optimization. In *Collaborative Research*, page 12, 2002.

[10] S. Tian, S. Mo, L. Wang, and Z. Peng. Deep reinforcement learning-based approach to tackle topic-aware influence maximization. *Data Science and Engineering*, 5(1):1–11, 2020.

[11] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, 1997.

[12] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, 2020.

[13] X. Zhou, C. Chai, G. Li, and J. SUN. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2020.

[14] X. Zhou, J. Sun, G. Li, and J. Feng. Query performance prediction for concurrent queries using graph embedding. *VLDB*, 13(9):1416–1428, 2020.

[15] D. C. Zilio et al. Recommending materialized views and indexes with IBM DB2 design advisor. In *ICAC*, pages 180–188, 2004.