Dynamic Materialized View Management using Graph Neural Network

Yue Han[1], Chengliang Chai[1], Jiabin Liu[1], Guoliang Li[1], Chuangxian Wei[2], Chaoqun Zhan[2]
[1]*Department of Computer Science, Tsinghua University,* [2]*Alibaba Group*
{han-y19@mails.,ccl@mail.,liujb19@mails.,liguoliang@}tsinghua.edu.cn, {chuangxian.wcx,lizhe.zcq}@alibaba-inc.com

***Abstract*—Materialized views (MVs) are vital in DBMS to improve the query efficiency by reducing redundant computations of shared subqueries in a workload. Traditional methods focus on static MV management, which assumes that MVs will not be added or evicted. However, in real scenarios, query workloads usually dynamically change, and thereby the previously maintained MVs cannot be well adapted to future workloads due to the possible shift in query distribution. Therefore, it is important to study the dynamic MV management problem where workloads dynamically change, but there are several challenges. First, it is challenging to estimate the benefit of using an MV to answer a query (i.e., the execution time reduction of answering the query using the view) in order to select high-quality MVs, especially for dynamic workloads, where queries and MVs dynamically change. Second, it is challenging to efficiently maintain the set of MVs to immediately utilize the new views to answer the newly coming queries. However, existing methods either have low accuracy (traditional methods) or low efficiency (learning-based methods).**

To address these challenges, we propose a novel framework `GnnMV` that leverages the graph neural network (GNN) to estimate the benefit for efficient and effective dynamic MVs management. First, we maintain the dynamic query workloads as a query graph, extract and encode key features of queries to model a GNN. Second, we design a feature aggregation function for neighbor nodes in the graph to achieve high accuracy. Third, we propose to extract a small subgraph for efficient benefit estimation, when the graph becomes larger and larger due to continuously coming queries. Experimental results show that our method significantly outperforms state-of-the-art approaches.

I. INTRODUCTION

In OLAP databases, a query workload contains many complex SQL queries, which usually share many common time-consuming subqueries, and building materialized views (MVs) for time-consuming and frequently-used subqueries is a commonly-used technique to improve the query performance by using MVs to answer queries.

Dynamic MV Management. Traditional static MV management methods generate a fixed set of MVs and use these MVs to answer queries. Obviously, these methods are not effective for dynamical workloads, because the fixed MVs cannot be well adapted to the newly coming queries. For example, the advertising business of Alibaba faces huge analytical queries from e-commerce store owners every day. In addition to the daily batch workloads, there are many other analytical queries, which have variable patterns and, most importantly, change with different seasons or fashion trends. Hence, when the dynamic workload patterns change, the MVs set should be updated in time to keep high hit ratio of MVs. Therefore, dynamic MV management is an important problem, which consists of two key steps as shown in Figure 1. (1) *Query rewrite.* Given a new query q, query rewrite aims to rewrite q with some MVs from currently maintained MVs. To this end, we have to estimate the benefit of using view v to answer q, *i.e.,* the execution time reduction of answering q after using v, and select the MVs that can bring the most benefit to rewrite, so as to maximize the query performance. (2) *MV set maintenance.* As new queries are continuously coming, we need to maintain the set of MVs to cope with the continuously coming queries with possible distribution shift. To this end, we also need benefit estimation of using existing MVs to answer queries in dynamic workloads. Then we generate new MVs that can mostly optimize the dynamic workloads within a limited space budget.

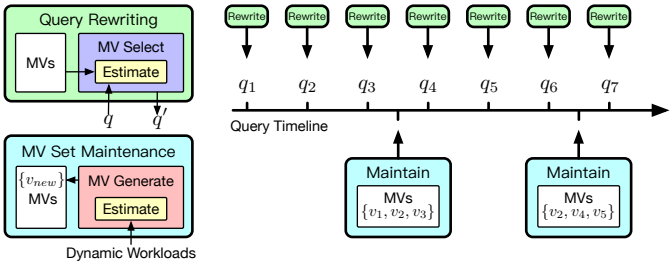


Fig. 1. Working mechanism of MVs for dynamic workloads.

Limitation of existing methods. MVs management is a vital problem that has been studied for decades. Traditional methods [1], [2], [8], [10], [15]–[17], [22], [24], [31]–[35] leverage the optimizer in DBMS to estimate the benefit of MVs efficiently, based on which they can iteratively select the proper MVs to materialize. Hence, they can achieve high efficiency even if the MVs are built from scratch given new query workloads, but low-quality MVs are generated since purely using the optimizer to conduct the benefit estimation is inaccurate [25], [36]. To address this issue, learning-based methods [14], [29], [45] are proposed to accurately estimate the benefit. However, to handle the dynamic scenario, they have to build the MVs from scratch. Although high-quality MVs can be derived, they are prohibitively expensive because

Chengliang Chai and Guoliang Li are the corresponding authors. This work is supported by NSF of China (62232009, 61925205, 62102215, 62072261), Huawei, TAL education, Alibaba, China National Postdoctoral Program for Innovative Talents (BX2021155), China Postdoctoral Science Foundation (2021M691784), Shuimu Tsinghua Scholar.

the deep neural architectures cause a slow benefit/cost estimation. Hence, they cannot meet the high-efficiency requirement under the frequent workload changing scenarios.

Challenges. To achieve the ultimate goal of minimizing the total execution time of the query workloads in the dynamic scenario, there are two essential challenges. First, given a large number of queries, how to estimate the benefit of each MV to each query accurately is challenging, which is rather significant to generate high-quality MVs (**C1**). Second, to enable that newly coming queries can utilize the generated MVs in time, MV set maintenance in the dynamic scenario should be very efficient. But with the number of queries becoming larger and larger, how to conduct the benefit estimation efficiently while keeping the high accuracy is challenging (**C2**).

Our proposed methods. To address the above challenges, we propose a graph neural network (GNN) based dynamic materialized view management framework, `GnnMV`, which maintains the set of MVs efficiently and effectively. Initially, `GnnMV` first builds a graph on the current workload. Each query in the workload can be represented as a query plan tree, and multiple trees constitute the graph. Next, we train a GNN model on the graph, which takes as input the node features and outputs the benefit estimation of using an MV to answer a query (an MV/subquery corresponds to a node in the graph). We design a GNN model to judiciously capture the benefit from different types of neighbor nodes through several iterations of feature propagations, and thereby provide accurate benefit estimation (for **C1**). Besides, the graph structure makes the benefit estimation in parallel for each iteration, which is efficient for query rewrite and MV set maintenance. Furthermore, as the graph grows larger (more queries), we discover a small subgraph, on which we conduct an efficient and accurate inference, instead of on the entire graph (for **C2**).

Contributions. We make the following contributions.

(1) We propose a GNN-based framework for efficient and effective dynamic MVs management.

(2) We build a well-designed graph model by encoding key features and designing an aggregate function for capturing neighbor nodes in a graph, to capture the MV characteristics.

(3) As dynamic workloads come, we maintain the graph incrementally, and discover a subgraph for efficient and accurate query-rewrite and MV set maintenance.

(4) Experimental results on five datasets show that our method outperforms state-of-the-art approaches.

II. PRELIMINARIES

A. Problem Definition

Query tree. Given a SQL query q, we model it as a query tree. Each subtree rooted at a node corresponds to a subquery, and each node indicates an operator.

Example 1: As shown in Figure 2, suppose q_1 : `SELECT R3.title FROM R1, R2, R3 WHERE R2.age > 30 AND R3.year > 2005 AND R2.id = R1.a_id AND R3.id = R1.bk_id;` and the optimizer produces a tree with 8 nodes, including 3 table scan operators, 2 filtering operators, 2 join operators, as well as a project operator, *i.e.,* the root node.

Since each node corresponds to a subquery rooted at the node in the query tree, we use the two terms (*i.e.,* subquery and node) interchangeably if the context is clear.

MVs Generation for Static Query Workload. Given a query workload, if a subquery with a high computational cost is shared by many complex queries, it is vital to avoid redundant computation by materializing a view for the subquery. Thus, when a query contains this subquery, we may rewrite the query and use the view to optimize it.

Example 2: As shown in Figure 2, if we materialize a set of views $V = \{v_1, v_3\}$ (*i.e.,* v_1 is a subquery that joins R_1 and R_2, and v_3 is a subquery that applies a filtering operator on R_3), both q_1 and q_2 can be optimized by using the views.

Given a query workload, there exist a large number of subqueries, *i.e.,* a set $\mathcal{V}$ of candidate MVs, it is necessary to judiciously materialize a subset $V^* \subseteq \mathcal{V}$ such that the total execution time of the query workload is minimized, where the total size of views in V^* is within a given space budget.

Remark. As discussed above, all subqueries (queries) except the leaves in the query tree can be materialized as views, so we call them *view nodes*. We call root nodes of queries as *query nodes*. Note that *query nodes* can also be *view nodes*.

Dynamic Workloads. We consider a common scenario that queries are coming continuously in a long period of time, also known as dynamic workloads. Apparently, it is ineffective that MVs are generated just based on an existing static workload, and the method using these fixed MVs to optimize the subsequent queries is not optimal, as the queries distribution is likely to shift. However, it is reasonable that the MVs built based on certain workloads can benefit the workloads arriving in the near future, because the query distributions are likely not to shift much. Therefore, we propose to keep updating the MVs iteratively with new workloads arriving, such that all queries can be well optimized over a period of time.

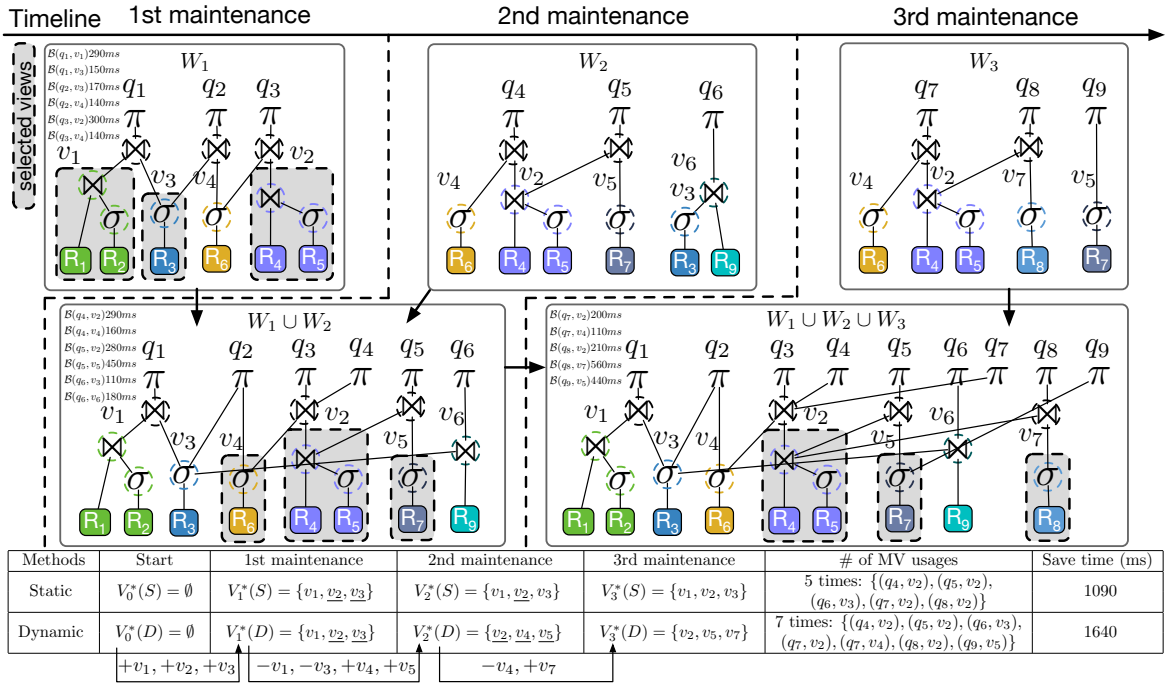
Next, we formally define the problem as below.

Definition 1 (Dynamic MVs Management): Given the current MVs set V_c^* selected based on the existing workload $\mathcal{W}_c$, when a new workload $\mathcal{W}_n$ arrives, the dynamic MVs management problem is to efficiently update V_c^* to a new set V_n^* such that V_n^* can minimize the total execution time of the future workload. Afterwards, we keep updating the MVs when new query workloads arrive, so as to utilize these up-to-date MVs to optimize the continuously coming queries.

Furthermore, given a workload $\mathcal{W}_c$, each MV set maintenance consists of two steps.

(1) [*Benefit estimation.*] As there are many subqueries (queries) in the workload, each of which can be a view, we can derive the MVs candidate set $\mathcal{V}$. To obtain a proper MVs set, we should know the benefit $\mathcal{B}(q, v)$ of utilizing an MV $v \in \mathcal{V}$ to answer a query $q \in \mathcal{W}_c$. The benefit is the execution time reduction of answering q with v, *i.e.,* $\mathcal{B}(q, v) = t_q - t_q^v$, where t_q is the execution time of the query plan of q, and t_q^v is the execution time of the query plan rewritten by v.

(2) [*MVs generation.*] Afterwards, we can build a bipartite graph to generate MVs, where one side includes the set of queries in the workload and the other side includes the MV



Methods	Start	1st maintenance	2nd maintenance	3rd maintenance	# of MV usages	Save time (ms)
Static	$V_0^*(S) = \emptyset$	$V_1^*(S) = \{v_1, \underline{v_2}, \underline{v_3}\}$	$V_2^*(S) = \{v_1, \underline{v_2}, v_3\}$	$V_3^*(S) = \{v_1, v_2, v_3\}$	5 times: $\{(q_4, v_2), (q_5, v_2), (q_6, v_3), (q_7, v_2), (q_8, v_2)\}$	1090
Dynamic	$V_0^*(D) = \emptyset$	$V_1^*(D) = \{v_1, \underline{v_2}, \underline{v_3}\}$	$V_2^*(D) = \{\underline{v_2}, \underline{v_4}, \underline{v_5}\}$	$V_3^*(D) = \{v_2, v_5, v_7\}$	7 times: $\{(q_4, v_2), (q_5, v_2), (q_6, v_3), (q_7, v_2), (q_7, v_4), (q_8, v_2), (q_9, v_5)\}$	1640

Fig. 2. An Example of Dynamic MVs Management.

candidates. Each edge is associated with $\mathcal{B}(q, v)$. Next, We aim to select and materialize a subset of MV candidates $V^* \subseteq \mathcal{V}$ within a space budget τ such that the total benefit of all queries with properly selected MVs is maximized.

Remark. We consider adding or evicting MVs from currently maintained views. We leave the MVs refreshing (for data update) in a future work.

Example 3: As shown in Figure 2, suppose that we have three dynamic workloads, *i.e.,* $[W_1 = \{q_1, q_2, q_3\}$, $W_2 = \{q_4, q_5, q_6\}$, $W_3 = \{q_7, q_8, q_9\}]$ arriving in a chronological order. And we compare the query execution time of using the *static MVs* and our proposed *dynamic MVs management* approach. We use $V_i^*(S)$ to denote the generated MVs after the i-th maintenance using the static strategy, and use $V_i^*(D)$ to denote the generated MVs with dynamic strategy.

Initially, since the MVs set is empty, we start with $V_i^*(S) = V_i^*(D) = \emptyset$. Then given W_1 (for the 1st maintenance), both methods initialize an MVs set as follows. First, we can derive an MVs candidate set $\mathcal{V}_1 = \{v_1, v_2, v_3, \cdots\}$, and then we estimate each *benefit* $\mathcal{B}(q, v), q \in W_1, v \in \mathcal{V}_1$. The estimated benefits are shown in Figure 2. In the first step, for each view, we add up its benefit on all queries that can use the view. For example, $\mathcal{B}(v_1) = 290ms$, $\mathcal{B}(v_2) = 300ms$, $\mathcal{B}(v_3) = 150 + 170 = 320ms$, $\mathcal{B}(v_4) = 140 + 140 = 280ms$. As the space budget is not enough for materializing all the views, we use the view selection algorithm to generate an MVs set (*i.e.,* $V_1^*(D) = \{v_1, v_2, v_3\}$) with the highest total benefit ($290 + 300 + 320 = 910ms$). Note that the static method does not change the set of views, so $V_i^*(S) = \{v_1, v_2, v_3\}, \forall i \geq 1$.

Next, $W_2 = \{q_4, q_5, q_6\}$ arrives, which can benefit from MVs in $V_1^*(S)(V_1^*(D))$. For example, $\mathcal{B}(q_4, v_2) = 290ms$, $\mathcal{B}(q_5, v_2) = 280ms$, $\mathcal{B}(q_6, v_3) = 110ms$. Next, for the second workload, our dynamic method will maintain the set of MVs. Given the workload $\mathcal{W}_c = W_1 \cup W_2$, we re-calculate the benefits of each view. For example, $\mathcal{B}(v_2) = 300+290+280 = 570ms$, $\mathcal{B}(v_3) = 150 + 170 + 110 = 430ms$, $\mathcal{B}(v_4) = 140+140+160 = 440ms$, $\mathcal{B}(v_5) = 450ms$, $\mathcal{B}(v_6) = 180ms$. We then generate an MVs set (*i.e.,* $V_2^*(D) = \{v_2, v_4, v_5\}$) with the highest total benefit ($870 + 440 + 450 = 1760ms$) within the space budget limit. v_1, v_3 will be evicted to make space for v_4, v_5. Based on this, $W_3 = \{q_7, q_8, q_9\}$ can be optimized using $\{v_2, v_4, v_5\}$. v_2 can be utilized to rewrite q_7 and q_8, while v_4, v_5 can respectively rewrite q_7, q_9. But if we use the static method, only v_2 can benefit q_7 and q_8. Then let $\mathcal{W}_c = W_1 \cup W_2 \cup W_3$ and we compute $V_3^*(D) = \{v_2, v_4, v_5\}$, which is utilized to optimize the next workload.

Overall, we can observe that compared with the static method, our dynamic method can optimize 2 more queries using MVs, which improves the efficiency.

In Section V, we discuss how to efficiently solve the dynamic MVs problem using the graph neural network.

III. GnnMV FRAMEWORK

In this section, we first summarize the overall framework of GnnMV, and then discuss how to train a GNN model and use it to achieve efficient and effective MV set maintenance.

A. Overview of GnnMV

Motivation of using GNN. Recap that one of the core parts of the MVs working mechanism in the dynamic scenario is the MV benefit estimation, which is the key in both query rewrite and MV set maintenance. More concretely, the estimation accuracy is rather important because it directly influences the quality of generated MVs. Besides, the efficiency of benefit

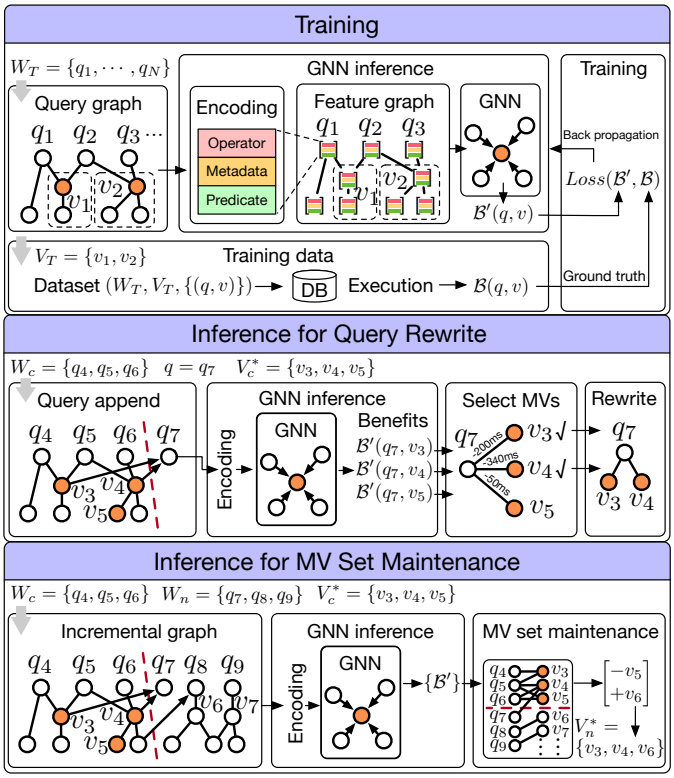


Fig. 3. Framework of GnnMV.

estimation is the bottleneck for dynamic MVs utilization with high-efficiency requirements.

To address the above issues, as the queries in a workload can be naturally represented as a graph, we propose to (1) efficiently detect common subqueries on the graph and (2) use a graph neural network to conduct the benefit estimation. At a high level, GNN has the advantage of these learning-based methods [14], [29], [45] that use the neural network with powerful learning ability to infer the benefit. Moreover, compared to RNN [14], GNN is rather efficient because it captures the information of the entire graph through several rounds of feature propagation among nodes in parallel. Compared to Wide & Deep learning [45] and RL [29], GNN is rather effective because it can capture the correlations among queries through the graphical structure.

The GnnMV framework. As shown in Figure 3, GnnMV consists of three modules, *i.e.,* offline GNN model training, online GNN inference for query rewrite, and GNN inference for MV set maintenance. A high-level workflow is as follows. Initially, we take as input a historical workload, based on which a query graph is built, and train a GNN model on the graph. The output of the model is the benefit estimation of the views to the queries in the workload.

Next, when queries arrive continuously online, the trained model is used for query rewrite and MV set maintenance. Given each new arriving query, the query rewrite module leverages the GNN model to estimate the benefit, and select MVs from currently maintained MVs to rewrite the query.

For a new workload, we generate new MVs, where the GNN model is also utilized to estimate the benefit, and then a bipartite graph is built for MVs selection.

Challenges. There are two challenges in using GnnMV. The first is how to encode the queries and views as features, and aggregate these features for training (Section IV). The second is that when maintaining the set of MVs, how to better leverage the knowledge of historical workloads to adapt to the new workload based on the GNN model, for keeping up-to-date high-quality MVs efficiently (Section V).

Remark. Besides the above issues, the time for MV set maintenance is also a challenge. In this paper, we can set a fixed number of incoming queries n. Suppose $n = 100$, which means that we should start to maintain the set of MVs every 100 new queries. More adaptive methods of determining when to maintain the set of MVs will be discussed in Section III-C.

B. GNN-based training

In this part, we overview the training process of the GNN model for benefit estimation (see Section IV for details).

Given a workload W_T that we collect for offline training, we build a query graph by merging query trees of queries in the workload. Next, we extract and encode the features (e.g., operator types, metadata, predicates) for each node in the graph to produce a feature graph. We design an aggregation function to aggregate the features from each node's neighbors and generate the node embedding. Afterwards, we use embeddings of a query q and a view v to compute the benefit $\mathcal{B}'(q, v)$ with a neural network. Obviously, we also need training data to construct a GNN model. But materializing all possible view is too expensive, we sample views and materialize them (*i.e.,* the set V_T), use them to optimize the queries in W_T, and compute the true benefit $\mathcal{B}(q, v)$. Then we compute the loss with $\mathcal{B}(q, v)$ and $\mathcal{B}'(q, v)$ to train the model.

C. GNN-based inference

In this part, we overview the inference process of the GNN model, which is used for both query rewrite and MV set maintenance (see Section V for details).

1) Inference for query rewrite: Once a new query arrives, e.g., q_7 in Figure 3, we select MVs to rewrite the query in the query optimizing process. First we detect available MVs (if exist) that can be used to rewrite the query by sub-graph matching, which ensures that the rewritten query is correct. Then, to select MVs that maximize the benefit of answering the query, we need to accurately estimate the benefit. Thanks to the graphical structure, we can easily add the query tree of q_7 into the current graph. Then we can encode q_7 into the trained GNN model, infer the benefits of views of current MVs, e.g., v_3, v_4, v_5, and select the proper view(s), e.g., v_3, v_4 to rewrite q_7 jointly. Given the set of views (*i.e.,* $\{v_3, v_4, v_5\}$) that can be used to rewrite a query (*i.e.,* q_7), the intuitive idea is to select the ones with high benefits to rewrite, but some views may have conflict and thereby cannot be used simultaneously. Hence, it is an NP-hard problem that selects an optimal subset with the highest benefit, where every two views do not conflict.

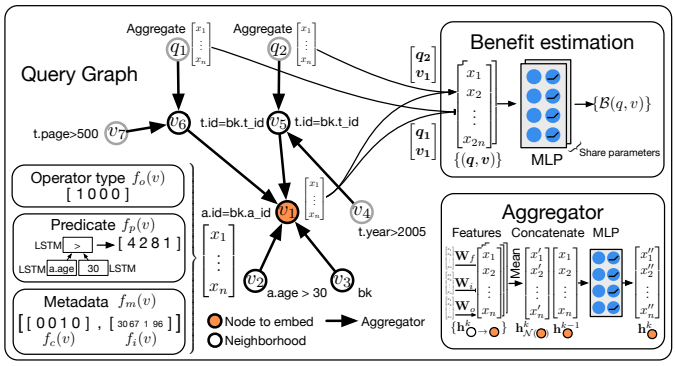


Fig. 4. The model in GnnMV.

2) Inference for MV set maintenance: Periodically, we need to maintain the set of MVs, which can further adapt to the subsequent workloads. Generally, we trigger the MVs set maintenance according to the performance of existing MVs. We monitor the accumulated benefit of existing MVs to recent queries in query rewrite in Section V-A because the benefit directly reflects the workload performance. When the accumulated benefit drops below a threshold, we trigger the MVs set maintenance. And, the queries arrive after the last maintenance form the new query workload. Given the new query workload, we can naturally incrementally update the graph by inserting the queries in the workload. Intuitively, the information of previous nodes, e.g., nodes of q_4, q_5, q_6, can be propagated to views (queries) in the new workload, such that the previous knowledge can help to refine their embeddings and capture the correlations among queries. Next, we use the GNN model to estimate the benefit for each new pair of (query, view), and compute the size of each view. Finally, we conduct the MVs generation step, *i.e.,* building a bipartite graph to generate a new set of MVs, e.g., $V_n^* = \{v_3, v_4, v_6\}$.

As the number of workloads grows, the graph becomes larger, leading to inefficient inference. To address this, we propose to extract a subgraph to infer with keeping efficient yet accurate benefit estimation (see Section V).

IV. GnnMV Training

We introduce how to design the GNN model, including graph feature encoding and graph design and training.

Query graph. To discover commonly-used subqueries and choose proper MVs, it is natural to build a query graph for a query workload [9], [18], [42]. Generally speaking, the graph is constructed by merging the query trees in the workload. Note that the graph is not restricted to be connected, i.e., some query trees can be independent if they do not share any nodes. For ease of representation, we will abuse the notations for the workload a little to denote the corresponding graph.

Example 4: In Figure 2, q_1, q_2 and q_3 constitute the graph corresponding the workload W_1, by merging q_1, q_2 on v_3, and q_2, q_3 on v_4. Then the graph W_2 comes, and by merging W_1 and W_2, we obtain the current graph $\mathcal{W}_c$.

Remark. The optimizer that does not take into consideration the affect of MVs may not generate best query plans for using MVs. Thus, we need to change the query plan of a query with different join orders to find high benefit MVs. In fact, the optimizer searches multiple query plans in the optimizing process, corresponding to multiple query trees. We can merge these redundant query trees into the query graph to find more shared MVs. Note that when counting MVs' benefit to the workload in the MVs generation step, MVs should not receive redundant benefits from all these redundant query trees. For ease of representation, we only use a single tree per query as example. However, merging multiple query trees of a query into the query graph brings overhead including more memory footprint. If the graph size exceeds the memory size, it needs more time for model training. Thus, we limit the number of redundant query trees with consideration of the memory size.

Next, we show the details of training the model.

A. Feature encoding

To build a GNN model, the first thing is to extract the features of each node in the graph and encode them. More concretely, as shown in Figure 4, there are three significant types of features for MVs management as follows.

Operator type. Each node v corresponds to an operator, including join operators (*e.g.,* Merge Join, Hash Join), scan operators or aggregation operators, etc. We also encode edge information related attributes such as "Parent Relationship" (e.g., Inner, Outer) and "Parallel Aware". We use a multi-hot vector with length O (the total number of operator types and attributes types), denoted by $f_o(v)$, to represent this feature.

Metadata. We also need to incorporate the metadata related to each node, including the table columns and the index information. Suppose that there are C columns in total among all tables in the database. We use a vector $f_c(v)$ (with length C) to specify which table columns are related to the node v. Each element (*i.e.,* a table column) of $f_c(v)$ is either 1 or 0, which means that the operator corresponding to v uses the column or not. Based on the column encoding, the data distribution is learned as the column embedding when training the model with historical query workloads. Similarly, we use $f_i(v)$ to encode information including estimated start-up cost, total cost, rows and width which are obtained by the optimizer of the database when generating the physical query plan. The start-up cost is the time expended before the output phase begins, *e.g.,* time to start the sorting in a sort node. The total cost is the time that the node completes its job, *e.g.,* all rows sorted and be output to its father node. The "rows" is used to capture the number of rows output by the node. The width is the width of rows output by the node (in bytes). Then the feature vector of the metadata, $f_m(v)$, is denoted by concatenating $f_c(v)$ and $f_i(v)$.

Predicate. Generally speaking, a predicate consists of table columns, computation operators (*e.g.,* $>, =, <$), and literals (*e.g.,* numeric, string, and category values). We can construct the predicate as an expression tree, use one-hot encoding to represent these table columns, computation operators and category values. Numeric values are encoded by the normalized float. String's character sequence is encoded to a vector. We use a tree-LSTM model [36] to encode the entire predicate.

The output $f_p(v)$ is represented as a vector with length P, which is a hyper-parameter. Afterwards, each feature vector of a node $f(v)$ is represented by concatenating $f_o(v)$, $f_m(v)$ and $f_p(v)$. Next, we use an example to better illustrate this.

Example 5: As shown in Figure 4, v_2 is associated with a sequential scan operator that scans the author table with the predicate `a.age>30`. We encode the operator type `Seq Scan` into a one-hot vector $f_o(v)$, in which the corresponding position for `Seq Scan` has the value 1. The predicate `a.age>30` can be represented as an expression tree, where the root node $>$ has two children `a.age` and `30`. We encode this expression tree into an embedding vector $f_p(v)$ by the tree-LSTM model. The metadata contains `author.age` column, and scanning cost, rows and width. We encode it into a vector $f_m(v)$ which contains a one-hot vector where corresponding position for the column `a.age` has the value 1, and other statistics.

B. Training for benefit estimation

The ultimate goal of our GNN model is to accurately estimate the benefit of using an MV v to answer a query q (*i.e.,* $\mathcal{B}(q, v)$) in a graph. To this end, we should first create the training data. Second, given the feature encodings, we also need proper representations (or embeddings) for both query and MV nodes so that the model can well capture their relationship for an accurate estimation of each $\mathcal{B}(q, v)$.

Training data. As a supervised learning task, we create the training data using a historical workload, denoted by W_T. As shown in Figure 3, we first build a query graph of W_T. Next, we sample a set V_T of MV nodes in the graph and materialize them because materializing all possible views is too expensive. We uniformly sample MV nodes rather than purely sampling some MVs with high benefits. This is because we request the training samples to be balanced, *i.e.,* covering different execution latency and sizes, in order to produce a model with strong generalization ability. Afterwards, for each $q \in W_T$, we discover the corresponding $v \in V_T$ that can be utilized to optimize q, compute the ground truth $\mathcal{B}(q, v)$ by executing q with/without v and regard this pair as one training instance. Note that the benefit can be negative because MVs are not always beneficial to queries. However, using only positive samples in the training will make the model have positive bias on benefit estimation. Therefore, carefully constructing negative samples is important to improve the model generalization ability [40]. Thus, we sample (q, v) pairs that have negative benefits by negative sampling technique [14]. Moreover, for each dataset, we try different ratio of positive samples to negative samples in the training data and use validation data to identify the best ratio for estimation accuracy. The empirical ratio of positive and negative samples is 8:2 [14].

Feature propagation. Besides the encoded features and training data, we need to learn a feature embedding for each MV (query) node, which can well capture the characteristics of the node for accurate benefit estimation. To this end, the key challenge is that the benefit of a view to a query does not only rely on its own features, but is also related to its neighbors in the query graph. For example, the cost of a join operation is highly related to the selectivity of its children's predicates. And the performance of a query is greatly correlated by the former query because of the buffer. Hence, the neighborhood information can definitely help the representation of a node.

Considering the above issues, we use GNN to capture the correlation among different query plans as well as plan nodes through the graph structure. We propose to propagate the features on the graph [12] in iterations. In each iteration, for each node, we aggregate the features of its neighbors in parallel. For example, given a feature graph as shown in Figure 4, at the first propagation, v_1 receives the features of v_2, v_3, v_5, v_6. Meanwhile, features of q_1, v_7 are also propagated to v_6, and features of q_2, v_4 are propagated to v_5. Then for the second propagation, v_2, v_3, v_5, v_6 are propagated to v_1 again, but at this time, v_1 can capture the information of q_1 because in the first iteration, the features of q_1 has been propagated to v_6. In this way, each node can well capture the information of its neighbors through several iterations.

Aggregate function. After propagation, an aggregate function is needed to compute the embedding of each node, considering the features of itself as well as its neighbors. One straightforward method is to just adopt the typical `GCN` aggregator [12] as the aggregation function. To be specific, for each node v at the k-th propagation iteration, the aggregator computes the element-wise mean of the embeddings of v and its neighbors at $(k-1)$-th iteration. Then we then apply a multi-layer perception (MLP) with an activation function ReLU on the aggregated embedding, producing the new embedding of v at the k-th iteration. Formally,

$$\boldsymbol{h}_i^k \leftarrow \sigma\left(\boldsymbol{W}_{gcn} \cdot \texttt{MEAN}\left(\{\boldsymbol{h}_i^{k-1}\} \cup \{\boldsymbol{h}_j^{k-1}, \forall v_j \in \mathcal{N}(v_i)\}\right)\right) \quad (1)$$

where $\boldsymbol{h}_i^0 = f(v_i)$ and $\mathcal{N}(v_i)$ denotes the node set of neighbors of v_i (e.g., $\mathcal{N}(v_1) = \{v_2, v_3, v_5, v_6\}$).

However, the above method treats each neighbor equivalently without considering the different characteristics of different types of nodes and edges in the query graph, which is different from other typical graphs like social networks. More concretely, in a query tree, data flows from bottom to top during the execution, and thus the features of father nodes should be distinguished from child nodes in the aggregate function. In addition, the difference of child nodes should also be identified. To be specific, we observe that if one of a node v's children has an index, the execution of the operator in v will be more efficient than not using the index, especially for the join operator. Therefore, child nodes with/without index will significantly influence the query execution. After analysis on the influence of different types of nodes on benefit estimation, we find that father nodes and indexed nodes should be treated specifically for better estimation.

Considering the above issues, we design an *MV aggregator* for feature aggregation on our query graph, which handles the father node, indexed node and others with different linear transformation, denoted by $(\boldsymbol{W}_f, \boldsymbol{b}_f), (\boldsymbol{W}_i, \boldsymbol{b}_i), (\boldsymbol{W}_o, \boldsymbol{b}_o)$ respectively. As shown in Figure 4, relying on $\boldsymbol{W}_f, \boldsymbol{W}_i, \boldsymbol{W}_o$,

neighbors will propagate different features to the node in aggregation. Formally, we define the propagated features as:

$$\boldsymbol{h}_{j\to i}^{k} = \begin{cases} \sigma(\boldsymbol{W}_f \boldsymbol{h}_j^{k-1} + \boldsymbol{b}_f) & \text{if } v_j \text{ is the father of } v_i \\ \sigma(\boldsymbol{W}_i \boldsymbol{h}_j^{k-1} + \boldsymbol{b}_i) & \text{if } v_i \text{ accesses } v_j \text{ by index} \\ \sigma(\boldsymbol{W}_o \boldsymbol{h}_j^{k-1} + \boldsymbol{b}_o) & \text{otherwise} \end{cases} \quad (2)$$

The MV aggregate function is:

$$\boldsymbol{h}_{\mathcal{N}(i)}^{k} \leftarrow \texttt{MEAN}\left(\{\boldsymbol{h}_{j\to i}^{k}, \forall v_j \in \mathcal{N}(v)\}\right) \quad (3)$$

$$\boldsymbol{h}_i^k \leftarrow \sigma(\boldsymbol{W}_{MV} \cdot \texttt{Concat}(\boldsymbol{h}_i^{k-1}, \boldsymbol{h}_{\mathcal{N}(i)}^{k})) \quad (4)$$

The number of propagation iterations is denoted by $\boldsymbol{K}$, the final embedding of each node is $e(v_i) = \boldsymbol{h}_i^{\boldsymbol{K}}$.

Benefit estimation. Given the embedding of each node, we can estimate the benefit of each MV node (a view) to each query node (a query). Inspired by the link prediction task in social network, we capture the relationship between a query q and an MV v by concatenating their embeddings, followed by an MLP, and then compute the benefit $\mathcal{B}'(q, v)$, as shown in Figure 4. Suppose that q and v are represented by v_i and v_j respectively. Formally,

$$\mathcal{B}'(v_i, v_j) = \boldsymbol{W}_2 \cdot \sigma\left(\boldsymbol{W}_1 \cdot \texttt{Concat}(e(v_i), e(v_j)) + \boldsymbol{b}_1\right) + \boldsymbol{b}_2 \quad (5)$$

Then we train the GNN model by minimizing the loss function, with the goal of optimizing parameters $\boldsymbol{W}$ and $\boldsymbol{b}$.

$$\mathcal{L} = \sum_{\mathcal{P}=\{(q,v)\}} \texttt{Smooth}_{L_1}(\ln \mathcal{B}'(q, v) - \ln \mathcal{B}(q, v)) \quad (6)$$

$$\texttt{Smooth}_{L_1}(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < 1 \\ |x| - \frac{1}{2} & \text{otherwise} \end{cases} \quad (7)$$

where $\mathcal{P} = \{(q, v)\}$ is a set of pairs of (q, v) such that q is the ancestor of v on the graph. This indicates that if q is not the ancestor of v, the view has no chance to benefit q, and thus we do not need to compute the benefit.

V. GnnMV INFERENCE

At a high level, given a node v, the trained model learns an aggregate function to leverage v's own features and its neighbors to produce an embedding for the benefit estimation. Thus, given some new nodes (a new query), no matter whether these nodes connect to the query graph, we can encode the node features and use the learned *aggregate function* to propagate features among the nodes and infer the embeddings. Hence, GnnMV can well support the benefit estimation in the dynamic scenario.

Advantages. To summarize, the advantages of using GNN to estimate the benefit are from two aspects. On the one hand, for each node on the query graph, GNN can well capture the information of its nearby nodes through a few iterations of propagation, which provides a more accurate embedding to each node. On the other hand, the features are propagated in parallel for each iteration, and the simple yet expressive neural architecture makes the inference step efficient.

Challenge. Hence, when a new query comes for rewrite, or furthermore, given a workload and the set of MVs need to be maintained, we have to use the graph model to infer the benefit for high-quality MVs generation. A challenge here is with the entire graph growing larger, how to ensure the inference efficiency while keeping high quality (Section V-B).

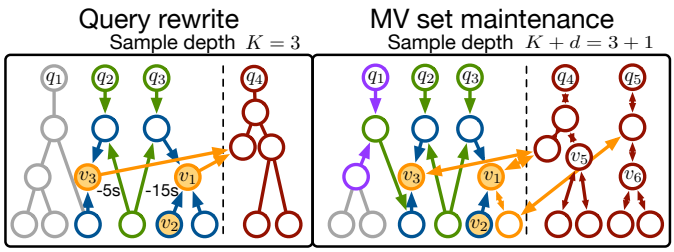


Fig. 5. Inference for query rewrite and MV set maintenance.

Next, we present how to use GnnMV model to conduct query rewrite and MV set maintenance effectively and efficiently.

A. Query rewrite using GNN

As discussed before, given a new query q, we first append it to the query graph and discover the view set $R(q) \subseteq V_c^*$, where $R(q)$ is a set of MVs that are available to rewrite q (*e.g.,* $\{v_1, v_2, v_3\}$ in Figure 5). Note that a query usually has multiple join orders, and different orders may have different benefits of using MVs. Thus, we enumerate multiple join orders to discover more latent available MVs, and try to find the best rewritten query. If the new query does not have any shared nodes with existing MVs even after the enumeration, this query cannot be rewritten with existing MVs. After, discovering available MVs, other MVs, whose corresponding queries are not sub-queries of q but have an partial overlap with q, are ignored because they are less possible to rewrite q and need to be further evaluated for query rewrite. We then estimate $\mathcal{B}(q, v)(\forall v \in R(q))$, and select a set of views, $R^*(q) \subseteq R(q)$ that has the largest benefit to rewrite q. Here, we define $\mathcal{B}(q, R^*(q))$ as the benefit of a set of views to a query. Specifically, it can be approximately computed by $\mathcal{B}(q, R^*(q)) = \sum_{v \in R^*(q)} \mathcal{B}(q, v)$ [17], where every two views in $R^*(q)$ do not conflict with each other. Here, the conflict means that the two MVs cannot be used to rewrite the query simultaneously (e.g., v_1 and v_2). Next, we estimate the benefit and select $R^*(q)$.

Benefit inference for query rewrite. It is easy to find $R(q)$, and mainly discuss the benefit estimation in this part, using an intuitive example. As shown in Figure 5, when q_4 comes, we append it on the graph and $R(q_4) = \{v_1, v_2, v_3\}$. Then we aim to compute $\mathcal{B}(q_4, v), \forall v \in R(q_4)$. To this end, since nodes in V_c^* have learned embeddings previously, we compute embeddings for each new node. Taking the query node q_4 as an example, first, we extract and encode the features of q_4, *i.e.,* $f(q_4)$ (Section IV). Then we leverage the Eqs. 2,3,4 to compute $e(q_4)$ by capturing the features of its neighbors. Suppose $\boldsymbol{K} = 3$, q_4 can capture the information of the nodes 3 hops away from it, *i.e.,* red nodes of q_4 and yellow nodes v_1 and v_3 of the query q_3 and q_2 respectively. Since q_2, q_3 are related to q_4 and may have been just executed (the buffer can make q_4 more efficient), their information is

likely to be helpful. Similarly, all the new nodes (in red) can simultaneously derive their embeddings with a low latency.

Select MVs for query rewrite. Given $\forall v \in R(q_4), \mathcal{B}(q_4, v)$, we start to select $R^*(q) \subseteq R(q)$ to rewrite the query. At a high level, it is a non-trivial problem because two MVs may conflict with each other. Hence, it can in fact be formulated as a weighted maximum independent set problem with node weights as MV benefits, which is NP-hard. Specifically, we regard each $v \in R(q)$ as a node (with weight $\mathcal{B}(q, v)$), and if two nodes conflict with each other, there should be an edge between them. Then we aim to select an independent set with the largest weights on the graph. Since it is an NP-hard problem, we use an approximate algorithm [19] to solve it. To be specific, the key idea is to first initialize $R^*(q)$ as an empty set, and then select $v \in R(q)$ one by one iteratively. In each iteration, it selects the node with the minimum weighted degree (the sum of v's neighbors weight/v's weight).

B. MV set maintenance using GNN

Benefit estimation for MV set maintenance. As the number of new arrived queries grows, we derive a new workload $\mathcal{W}_n$, append it to the current query graph $\mathcal{W}_c$ and we start to maintain the set of MVs. To this end, we estimate views benefit and view materialization cost (*i.e.,* generation cost of MV candidates and update cost of existing MVs). We estimate the generation cost by adding the time of executing the view and writing the result, which can be calculated with the information extracted from the subtrees of executed query plan trees where the views are generated from. We consider the update cost as MV recomputing cost (*e.g.,* `REFRESH` in `PostgreSQL`) which is approximately equal to the cost of recreate the view. The estimation of MV update cost with incremental view maintenance is not supported yet and will be studied in the future work.

To estimate the benefit of MV candidates, the node embeddings should be obtained. In addition, as a number of new queries have been added, some of the node embeddings in $\mathcal{W}_c$ should also be verified. A straightforward method is to encode the features of new queries (subqueries), propagate the features on the entire graph ($\mathcal{W}_c \cup \mathcal{W}_n$), and infer the embeddings. However, as the graph becomes larger and larger with a number of workloads arriving, inference over the entire graph is time-consuming because a large graph has to be inferred in multiple batches. Also, it is not necessary to do this because the new nodes have little impact on the embeddings of the nodes far away from them on the graph.

Therefore, we propose to compute the embeddings on an extracted subgraph, which covers the necessary feature propagation. First, this subgraph definitely contains $\mathcal{W}_n$ because the embeddings of nodes contained in $\mathcal{W}_n$ are unknown. Second, the subgraph should also contain some nodes that are close to $\mathcal{W}_n$ because they will be greatly affected, and thus their embeddings should be updated.

d-hops nodes. We define the set of nodes that have a distance within d to at least one of nodes in $\mathcal{W}_n$, denoted by $\mathcal{W}_n^d$. For example, in Figure 5, $\mathcal{W}_n^{d=1}$ is the set of yellow nodes.

Consider the number K of iterations for feature propagation. As discussed above, we are not only going to derive the embeddings of $\mathcal{W}_n$, but also have to update the embeddings of nodes close to $\mathcal{W}_n$, *i.e.,* nodes in $\mathcal{W}_n^d$ where d is small. Hence, in this situation, we propose that we only need to extract the subgraph containing $\mathcal{W}_n \cup \mathcal{W}_n^{d+\mathrm{K}}$. In this way, the features of $\mathcal{W}_n^{d+\mathrm{K}}$ can be fully propagated to W_n^d within K iterations, which achieves the same result of the updated embeddings of K computed on the entire graph.

For example, as shown in Figure 5, where $K = 3, d = 1$ and q_4, q_5 are newly appended queries. The extracted subgraph contains the red, yellow, blue, green, and purple colored nodes. When aggregating the features, the new nodes can capture the information of yellow, blue, and green nodes. Besides, the yellow nodes can also be updated by aggregating the features from red, blue, green, and purple nodes.

In this way, the updated embeddings make the benefit estimation more accurate, and the subgraph improves the efficiency even more and more workloads arrive.

MVs generation. After a new workload arrives, the new current workload becomes $\mathcal{W}_c = \mathcal{W}_c \cup \mathcal{W}_n$. Also, the MV candidates set varies, and we still use $\mathcal{V}$ to denote it for ease of representation. To generate $V_n^* \subseteq \mathcal{V}$ for optimizing $\mathcal{W}_c$ within a space budget τ, we build a bipartite graph where nodes of one side are from $\mathcal{W}_c$ and nodes of the other side are from $\mathcal{V}$. Each edge denotes the benefit of a $v \in \mathcal{V}$ to $q \in \mathcal{W}_c$ and labeled by $\mathcal{B}(q, v)$. To take into consideration the view materialization costs, for creating a new view, we subtract the view materialization costs from the benefit of view creation.

Note that some views conflict with each other, when rewriting a query, we use a matrix $T_{|\mathcal{V}|\times|\mathcal{V}|\times|W|}$ to indicate the relationship between views, where $T_{ijk} = 1(0)$ indicates that v_i and v_j conflict (not conflict) on q_k. To quickly find views in conflict instead of slowly enumerating every two views, we traverse the bipartite graph to identify 1-hop neighbors and 2-hops neighbors of each $v \in \mathcal{V}$. The 1-hop neighbors of v are the queries set $\{q\}$ that v can rewrite. The 2-hops neighbors of v are the views set that v may conflict with. We use $x_{ik} = 1(0)$ to indicate whether we select v_i to rewrite q_k or not. We denote the space of v as $|v|$. To obtain the maximum total benefit, we optimize this integer programming problem:

$$\begin{aligned} &\arg\max_{\{x_{ik}\}} \sum_{q_k \in \mathcal{W}_c, v_i \in \mathcal{V}} \mathcal{B}(q_k, v_i) x_{ik}, \\ s.t., &\sum_{v_i \in \mathcal{V}} |v_i| \max\{x_{ik} | \forall k \in [1, |\mathcal{W}_c|]\} \leq \tau, \\ &T_{ijk} + x_{ik} + x_{jk} < 3 \end{aligned} \tag{8}$$

Since it is an NP-hard problem [11], we use the greedy algorithm [5] with an approximation ratio of 2 to select V_n^*. We denote the benefit with considering other selected MVs as $\mathcal{B}(q_k, v_i | V_n^*)$ where $V_n^* = \{v_{n_1}, v_{n_2}, \cdots, v_{n_m}\}$ and $\mathcal{B}(q_k, v_i | V_n^*) = 0, if \ \exists T_{ijk} = 1, \forall v_j \in V_n^*$. We derive the marginal benefit of a view v as $\mathcal{B}(v | V_n^*) = \sum_{(q,v)\in E} \mathcal{B}(q, v | V_n^*)$. Thus, we derive the total benefit of V_n^*:

$$\mathcal{B}(W_m, V_n^*) = \sum_{v_{n_i} \in V_n^*} \mathcal{B}(v | \{v_{n_1}, \cdots v_{n_{i-1}}\}) \tag{9}$$

TABLE I
WORKLOADS AND QUERY GRAPH.

Workload	# of queries	# of nodes	# of edges
Extended JOB	226	3086	8284
Scale	500	3787	7362
Synthesis	5000	29782	56444
TPC-H	10000	136215	293800
XuetangX	35749	164384	328534

We first initialize selected view set $V_n^* \leftarrow \emptyset$ and view's benefits $\mathcal{B}(v_i|\emptyset) \leftarrow \sum_{q_k \in W} \mathcal{B}(q_k, v_i), \forall v_i \in \mathcal{V}$. We then choose a view v_{n_1} with the highest benefit $\mathcal{B}(v_{n_1}|\emptyset)$ and insert it into $V_n^* \leftarrow V_n^* \cup \{v_{n_1}\}$. We then update the benefit of rest views to queries and view's benefits, *i.e.,* $\mathcal{B}(q, v|\{v_{n_1}\}), \mathcal{B}(v|\{v_{n_1}\})$ respectively. Iteratively, we choose the next view $v_{n_2}, v_{n_3}, \cdots$ until the total size of selected views exceeds the budget τ. At last, we output the selected view V_n^* for MV set maintenance.

Remark. Note that the time complexity of the above greedy algorithm is $O(|\mathcal{V}|^2)$, which is too expensive as the number of queries grows. Considering the fact that recent queries would not shift much in distribution, in practice, we keep the union of recent query workloads as $\mathcal{W}_c$, from which the corresponding $\mathcal{V}$ are derived. In this way, the complexity is greatly reduced without much influencing the MVs' quality.

Incremental training at runtime. As the number of workloads accumulating at runtime, the workload distribution shifts. Therefore, we periodically incrementally train the model and substitute it for the online model to keep the benefit estimation accuracy. Note that the incremental training will not suspend the online model. The trigger of incremental training follows two strategies: (1) periodically train the model at system idle time; (2) sample online queries and evaluate the prediction error of benefit estimation model and train the model when the prediction error is greater than a threshold.

VI. EXPERIMENT

We have conducted extensive experiments to evaluate our GnnMV mainly from three aspects: (1) Overall dynamic MV management performance. (2) The efficiency of MV set maintenance and query rewrite. (3) The effectiveness of the GNN model on MV benefit estimation.

A. Experimental Settings

Dataset. We used three datasets including TPC-H benchmark and two real-world datasets IMDB, which is widely used with join order benchmark (JOB) [25], [36], [44], and XuetangX [41], [50] (a time-series query workload). We split each dataset into 10 equal length consecutive workloads and maintain the set of MVs at the end of each workload dynamically, and rewrite each query using existing MVs.

As shown in Table I. (1) XuetangX is a real-world benchmark containing 14 tables and we extracted 35749 slow queries (execution time more than 1s) among three months as the workload. Each query in XuetangX was naturally assigned with a timestamp, so we ordered the queries by the timestamp before splitting it into 10 dynamic workloads. (2) TPC-H is a widely used synthetic benchmark including 22 query templates. We generate 10GB of data with 10K queries according to the standard of TPC-H. We shuffled the workload before splitting it to avoid a too large distribution gap between adjacent workloads. (3) IMDB is a movie dataset that has a size of 3.7GB with 21 tables, in which the largest table has a size of 1.4GB with 36 million rows. We use three query workloads: JOB [25], Synthesis [20], and Scale [20], The JOB workload contains 113 queries with at most 16 joins. Since the number of 113 queries is too small for MVs evaluation, we extended it to 226 queries by modifying the predicates of queries. The Scale workload has 500 queries with at most 4 joins and only numeric predicates in queries. The Synthesis workload has 5000 queries with at most 2 joins. The query tree structure is simpler than the Scale workload and has fewer potential rewrite chances for MVs. We also split these three workloads into consecutive dynamic workloads.

Baselines. We compared GnnMV with the following methods:
(1) Static: It generated MVs using the first workload and kept the MVs fixed for all incoming workloads.
(2) DynaMat [22]: It is a score-based method that measured the quality of MV based on frequency, execution time, and size. It selects MVs according to their quality.
(3) HAWC [31]: It maintain the set of MVs based on the cost model in the optimizer according to recent historical queries.
(4) DQM [29]: It is an RL-based method that learned MV scores from database runtime statistics and update the MV scores after each query execution.
(5) AutoView [14]: It is an RNN and RL-based method that learned MV quality and MV selection strategy from query workload. We modified AutoView to select new MVs set after each dynamic workload arrives.

Hyper-parameter settings. We built our GNN model with 3 aggregate layers, all of them with a hidden size of 256. We split the workload into training, validation and test set with the ratio 3:1:6. We trained the model up to 500 epochs with early stopping and the learning rate was 1e-3.

Evaluation metrics. We evaluated the methods from three aspects: effectiveness, efficiency, and accuracy. For effectiveness, we used the total reduced execution time over the period of all workloads as the metric for MV generation performance. For model efficiency, we used the MV set maintenance latency as the metric. For MV benefit estimation accuracy, we used mean absolute percentage error (MAPE) as the metric, *i.e.,* $\texttt{MAPE} = \texttt{MEAN}_{(q,v)\in\mathcal{P}} \left(\left\{ \left| \frac{\mathcal{B}'(q,v) - \mathcal{B}(q,v)}{\mathcal{B}(q,v)} \right| \right\} \right)$.

B. Overall MV performance evaluation

1) Comparison of MV performance: We executed these dynamic workloads in each dataset and evaluated the accumulated reduced workload execution time of different methods, as shown in the left y-axis in Figure 6. The right y-axis showed the time reduction as a percentage of the total execution time of workloads without MVs.

Generally speaking, GnnMV outperformed other methods on all five datasets. For reduced workload execution time in JOB workload, GnnMV outperformed the second best solution by 1.75 times, mainly because (1) GnnMV estimated MVs' benefit accurately which produces high-quality MVs (Sec VI-C), and

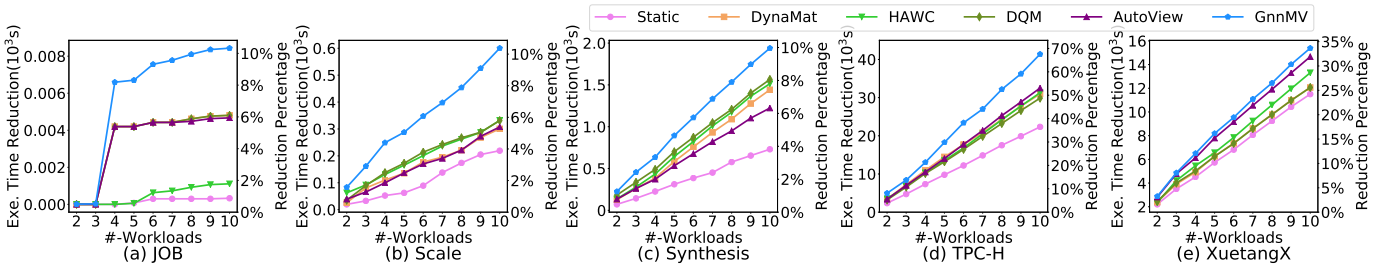


Fig. 6. Accumulated execution time reduction (the higher, the better).

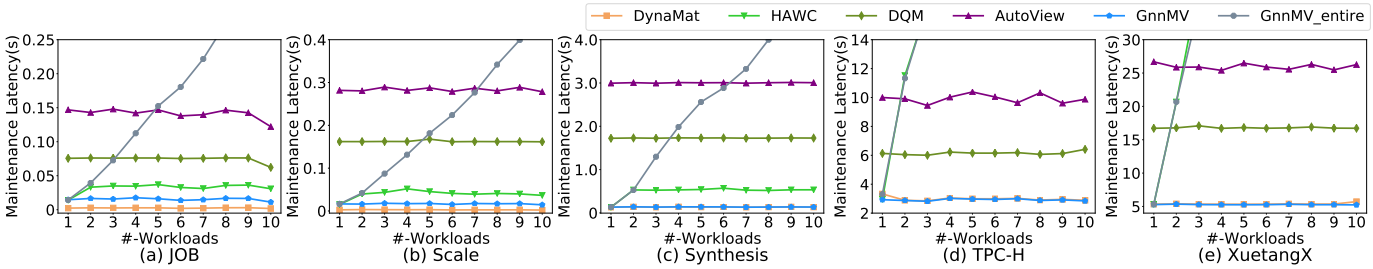


Fig. 7. MV set maintenance latency (the lower, the better).

(2) GnnMV rewrote query and maintained the set of MVs with a low latency (Sec VI-B2). On the five workloads, GnnMV reduced the average execution time of each query by 10.4%, 10.3%, 9.9%, 67.0%, and 34.0% respectively as shown in Figure 6. The training time of model was between 6.46s (JOB) and 6.82s (XuetangX). The accumulated benefit estimation time of GnnMV model on five workloads were 0.027, 0.023, 0.024, 0.036 and 0.039s respectively. The accumulated time of building query graph and and extracting features on five workloads were 0.05, 0.06, 0.41, 2.4 and 7.4s respectively. The accumulated time of generating the MV candidates were 0.25, 0.24, 4.6, 13.6, 26.8ms respectively.

Dynamic vs Static. Dynamic MVs methods (*i.e.,* GnnMV, DynaMat, HAWC, DQM and AutoView) outperformed static MVs (*i.e.,* Static) over the entire period of workloads. For example, as shown in Figure 6(a), on JOB dataset, the total reduced workload time (after 10 dynamic workloads) of GnnMV was 8.43s, which was 25 times more than that of Static (0.33s). On XuetangX, in Figure 6(e), GnnMV saved 1.3 times more than Static. The results showed that MV set maintenance significantly improved the performance of the dynamic workloads. The reason why GnnMV improved more on dataset JOB than XuetangX was that XuetangX had more uniform queries and smaller distribution shifts along with the dynamic workloads.

Learning-based method vs Traditional method. GnnMV outperformed traditional methods (*i.e.,* HAWC and DynaMat) on the total reduced execution time. For example, in Figure 6(a), on JOB dataset, GnnMV saved 8.43s, which was 1.75 times more than that of DynaMat (4.81s) and 7.6 times more than that of HAWC (1.11s). In Figure 6(b)(c), DQM also outperformed HAWC and DynaMat on datasets Scale and Synthesis. In Figure 6(d)(e) AutoView outperformed HAWC and DynaMat on datasets TPC-H and XuetangX. The reason why the learning-based methods performed better was twofold. One reason was that learning-based methods (*e.g.,* GnnMV and AutoView) used a neural networks to accurately estimate MV benefit while HAWC and DynaMat used the cost model of the optimizer. For example, on XuetangX dataset, GnnMV saved 15385s of total workload execution time and AutoView saved 14664s while HAWC and DynaMat saved 13311s and 12062s respectively. Another one was that learning-based methods (*e.g.,* GnnMV and DQM) leverage the powerful ability of the ML model to adapt to different dynamic workload distributions, but traditional methods tend to fall into the local optimum once the distributions vary.

GnnMV vs DQM. To summarize first, GnnMV outperformed DQM, for example, on Scale dataset, GnnMV reduced 600.4s total workload time, which was 1.8 times more than DQM (331.1s). The reason was that Scale contained many complex and time-consuming queries, but DQM just learned from the runtime statistics in DBMS rather than features of queries. In contrast, GnnMV extracted and encoded MVs features (e.g. join conditions, predicates, and index) from the query plan nodes which helped estimate the query execution time and benefit of MVs. Moreover, GnnMV inferred on query graph, and thereby can capture the influence among queries which greatly affects the query execution time by buffer using.

GnnMV vs AutoView. GnnMV outperformed AutoView among all five workloads. For example, on Synthesis, in Figure 6(c), GnnMV saved 1939s for total workload execution time, which was 1.59 times more than AutoView (1223s). The reason was that AutoView focused on single workload MV generation and could not capture the relationship between queries in dynamic workloads, but GnnMV estimated queries and MVs on a graph incrementally, and thus it could capture the workload distribution change along with the dynamic workload to maintain the set of MVs for subsequent queries. Moreover, AutoView focused on MV benefit estimation without consideration of the context of previous executed queries, while GnnMV learned from the query plan graph, and thus performed more accurate

estimation. In short, simply applying AutoView to the scenario of dynamic workloads by generating MVs multiple times would result in low performance.

2) Comparison of MV set maintenance latency: We evaluated the latency of MV set maintenance for each method. In GnnMV, we kept the recent 2 workloads as $\mathcal{W}_c$ over which we maintain the set of MVs. We did not evaluate Static baseline because it did not maintain the set of MVs except for the first dynamic workload. We additionally compared GnnMV with a modified version GnnMV-entire: infer the entire query graph to estimate MV benefit instead of only the nodes newly appended to the graph. The evaluation result was shown in Figure 7. Generally speaking, GnnMV outperformed other learning-based methods and the traditional method HAWC, and also was efficient in a large workload (*e.g.,* XuetangX).

GnnMV vs AutoView. GnnMV outperformed AutoView on all five datasets over the entire period of workloads. For example, as shown in Figure 7(a), on JOB dataset, the average maintenance latency of GnnMV was 0.015s which was 9.4 times more efficient than AutoView (0.142s). The reason was that GnnMV estimated MV benefits by inferring all the new nodes in the query graph in parallel, while AutoView spent a longer time on sequentially tackling the query plan nodes.

GnnMV vs DQM. GnnMV also outperformed DQM on all the datasets. For example, as shown in Figure 7(c), on Synthesis dataset, the average maintenance latency of GnnMV was 0.14s, which was 12.4 times more efficient than DQM (1.73s). The reason was that the RL model used in DQM estimated MVs after each query execution and was trained online, while the GNN model in GnnMV was trained offline and was used to estimate MVs online efficiently.

GnnMV vs GnnMV-entire. GnnMV outperformed the modified method GnnMV-entire on all the datasets. For example, in Figure 7(c), on Synthesis, GnnMV (0.14s) was 24 times more efficient than GnnMV-entire (grew to 3.33s at the 7th workload). One reason was that, as the number of queries grew, the inference time on all nodes in the incremental graph increased because the number of batches of nodes also grew. Thus, for GnnMV, we only inferred the newly appended nodes and the around old ones. Another reason was that the growing number of MV candidates in the incremental graph also slowed down the MV generation algorithm. Fortunately, considering the fact that recent queries would not shift much in distribution, in practice, we kept the union of recent 2 query workloads. This greatly reduces the complexity without much influencing the MVs's quality.

GnnMV vs HAWC & DynaMat. GnnMV outperformed HAWC on the average maintenance latency on all the datasets. For example, as shown in Figure 7(b), on Scale dataset, GnnMV (0.02s) was 2 times more efficient than HAWC (0.04s). HAWC kept a historical query and view pool for MVs generation. As the number of queries grows, the maintenance latency grew with the pool size. GnnMV had a competitive maintenance latency with DynaMat on large datasets (*e.g.,* TPC-H and XuetangX) as shown in Figure 7(d)(e). This was because DynaMat had to estimate every existing MV for eviction so as to add a new MV. However, the number of existing MVs would be large on the XuetangX dataset. GnnMV had slightly higher latency than DynaMat on JOB and Scale workloads, because JOB and Scale had relatively complex query plans where GnnMV takes more time to detect MVs conflicts in higher query tree.

TABLE II
COMPARE ESTIMATION RELATIVE ERROR ON DATASETS (MAPE).

JOB	median	90th	95th	99th	max	mean
RLView	0.575	2.61	4.93	27.0	151	1.78
AutoView	0.264	0.910	1.41	3.74	93.6	0.488
GnnMV	**0.120**	**0.416**	**0.544**	**1.24**	**39.0**	**0.207**
Scale	median	90th	95th	99th	max	mean
RLView	0.726	2.80	5.33	31.7	101	1.96
AutoView	0.280	0.870	1.44	4.90	48.2	0.520
GnnMV	**0.212**	**0.540**	**0.712**	**1.66**	**38.4**	**0.308**
Synthesis	median	90th	95th	99th	max	mean
RLView	0.445	1.67	2.91	11.6	65.0	0.937
AutoView	0.233	0.632	0.870	1.93	**21.3**	0.325
GnnMV	**0.186**	**0.487**	**0.590**	**1.10**	25.7	**0.243**
TPC-H	median	90th	95th	99th	max	mean
RLView	0.219	0.603	0.928	6.24	**251**	0.360
AutoView	0.095	0.346	0.434	1.17	405	0.171
GnnMV	**0.037**	**0.153**	**0.185**	**0.261**	458	**0.071**
XuetangX	median	90th	95th	99th	max	mean
RLView	0.275	0.640	0.745	2.20	**80.6**	0.353
AutoView	0.109	0.311	0.442	0.884	120	0.155
GnnMV	**0.060**	**0.158**	**0.187**	**0.317**	90.1	**0.088**

3) Query rewrite overhead: Finding MVs that were available to rewrite a query was fast because we only traverse the nodes of the plan tree of the query instead of all nodes. The inference time overhead of GNN for a query and its MVs was 0.0025s, which was much lower than RNN (0.0979s).

C. Benefit estimation accuracy

1) Evaluate benefit estimation accuracy: We evaluated the GnnMV on MV benefit estimation accuracy by comparing the following methods:

(1) RLView [45]: It used the Wide & Deep learning to estimate the benefit of MVs.

(2) AutoView [14]: It used the state-of-the-art double RNN model, Encoder-Reducer, to estimate the benefit of MVs.

Table II showed the MAPE of different estimation methods. The result could be ranked as GnnMV >AutoView >RLView. Next, we explained the results. Generally speaking, the GNN model performed best on errors at median, 90th, 95th, and 99th on all the datasets, because GnnMV could well capture the relationship between operator nodes which was help full for benefit estimation. For example, considering the mean error on TPC-H dataset, GnnMV reduced the MV benefit estimation error (MAPE) by 58.5% against the the second-best solution. It was harder to keep a low max error, because small benefit values (*e.g.,* several milliseconds) that could result in a large percentage error. Moreover, learning-based methods (*e.g.,* RLView, AutoView, and GnnMV) performed better on workloads with uniform distribution, *e.g.,* TPC-H and XuetangX.

GnnMV performed better than AutoView. For example, on TPC-H, the MAPE of GnnMV (0.071) was 58.5% lower than AutoView (0.171). The reason was that GnnMV captured the relationship between queries along with the workload.

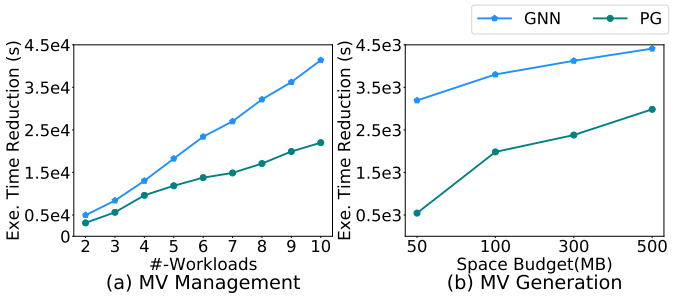


Fig. 8. Evaluate different benefit estimation (the higher, the better).

GnnMV performed better than RLView. For example, on TPC-H dataset, The MAPE of GnnMV (0.071) was 80.3% lower than RLView (0.360). The reason was that RLView could not well distinguish the correlation between nodes in the query tree. In contrast, GnnMV aggregated the information of neighbors. Furthermore, GnnMV handled father nodes and indexed nodes with different transformations on embeddings.

2) Evaluate different benefit estimation methods: We evaluated the effectiveness of accurate benefit estimation on MV generation on TPC-H dynamic workloads. We compared GnnMV with PG (*i.e.,* cost estimator of the PostgreSQL optimizer) by evaluating the performance of generated MV. The experimental result was shown in Figure 8. Generally speaking, more accurate MV benefit estimation significantly improved the MV generation performance because high-quality MVs are generated. For example, in Figure 8(a), the total reduced query execution time of GNN method was 41379s, which was 1.9 times more than PG method (21997s). We also evaluated the MV generation on a single workload with different space budgets (from 50MB to 500MB), as shown in Figure 8(b). At 500MB of space budget, GNN method saved 4410s which was 1.5 times more than PG method (2984s). Moreover, we observed that accurate benefit estimation significantly improved MV performance at a smaller space budget, because PG method with inaccurate estimation was more likely to miss high benefit MVs at small space budget.

D. Additional experiments

We also evaluated and discussed the effect of several hyperparameters in GnnMV, including the different aggregators in GNN, $\boldsymbol{K}$ hops in aggregators, ablation study for feature encoding, and MVs set maintenance window size. (We put it in our technical report [13]).

VII. Related Work

MV generation for static workloads. MV generation for static workload has been studied over decades. Finding the best MV set for a workload within a space budget has been proven to be NP-hard [11]. The view selection problem can be represented as a 0-1 integer linear programming (0-1 ILP) problem, and ILP solvers are proposed to select MVs [17], [42]. However, the complexity of the 0-1 ILP approach is $\mathcal{O}(2^n)$ which is too expensive for large workloads. Thus, many heuristics methods [1], [2], [8], [10], [15]–[17], [24], [35] are proposed, including genetic algorithm [15], [16], reefs optimization algorithm [2], and particle swarm optimization algorithm [24]. However, heuristic methods have some assumptions on the data and query distributions. Thus, these heuristic methods cannot be well generalized to other workloads and datasets. To address this issue, learning-based methods [14], [29], [45] are proposed. Han et al. [14] propose AutoView to encode pairs of (MV, query) as features, and feed them to the RNN model [4], based on which the benefit of the MV to the query can be precisely estimated. Similarly, Yuan et al. [45] propose RLView to encode the features and utilize the Wide & Deep model [3] for estimation. Reinforcement learning (RL), e.g., RLView [45] and AutoView [14], are proposed to use RL to select MVs.

MV generation for dynamic workloads. Statically generated MVs cannot serve dynamic workloads, because the query pattern and data distribution shift among workloads. Therefore, the set of MVs should be maintained during the processing of workloads. Traditional approachs propose to use heuristic algorithms to select MVs. WATCHMAN [34], DynaMat [22] and HAWC [31] measure the quality of MVs and manage MVs by scores. The scores basically depend on the cost-model of the optimizer. However, the inaccurate of cost estimation results in low MV quality. DQM [29] uses RL to choose which MVs to create and uses scores to evict MVs by learning the MV scores on real runtime statistics in the DBMS. However, they have no query feature encoding and do not capture latent patterns in the workload which results in unstable MV scores estimation.

Graph Neural Network (GNN). GNN [12] is proposed to learn from a network where partial nodes are labeled (with ground truth) and others are not. GNN learns the feature of nodes and can be used to solve the tasks such as cardinality estimation and query performance prediction. In this paper, we build our graph model based on GraphSAGE [12] that improves GCN [7], [21], [38] by enabling the model to generalize to unseen nodes such that they can also be inferred.

Learning models for database.. There exist many works that leverage ML methods to solve database problems (see [48] for a survey). For instance, there are learning-based approach like reinforcement learning for join order selection [30], [44], [51], knob tuning [27], [46] and query generation [47]. There are also many deep learning methods for cardinality estimation [36], [37], [39], [43], query rewrite [49], index construction [6], [23], database systems [26], [28] etc.

VIII. Conclusions

We have studied the problem of dynamic MVs management and proposed a GNN-based model. First, we built a query graph on the dynamic workloads. Then we proposed to encode and propagate features of the nodes (subqueries) on the graph. We designed an aggregate function to aggregate the features of node neighbors for accurate benefit estimation. Finally, we proposed a subgraph extraction method to accelerate the graph inference for query rewrite and MV set maintenance while keeping high accuracy. Experimental results on real datasets showed that our method achieved higher quality than state-of-the-art methods.

REFERENCES

[1] 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.

[2] 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.

[3] H. Cheng, L. Koc, J. Harmsen, and etc. Wide & deep learning for recommender systems. In *DLRS@RecSys 2016*, pages 7–10. ACM, 2016.

[4] 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.

[5] G. Dantzig. *26. Discrete-Variable Extremum Problems*. Princeton University Press, 2016.

[6] H. Dong, C. Chai, Y. Luo, J. Liu, J. Feng, and C. Zhan. Rw-tree: A learned workload-aware framework for r-tree construction. In *ICDE 2022*, pages 2073–2085. IEEE, 2022.

[7] Z. Gharibshah, X. Zhu, A. Hainline, and M. Conway. Deep learning for user interest and response prediction in online display advertising. *Data Science and Engineering*, 5(1):12–26, 2020.

[8] A. Gosain and K. Sachdeva. Handling constraints using penalty functions in materialized view selection. *Int. J. Nat. Comput. Res.*, 8(2):1–17, 2019.

[9] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[10] H. Gupta. Selection of views to materialize in a data warehouse. In F. N. Afrati and P. G. Kolaitis, editors, *ICDT*, volume 1186 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 1997.

[11] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In C. Beeri and P. Buneman, editors, *ICDT 1999*, volume 1540, pages 453–470. Springer, 1999.

[12] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *NIPS 2017*, pages 1024–1034, 2017.

[13] Y. Han, C. Chai, J. Liu, G. Li, C. Wei, and C. Zhan. A technical report on dynamic materialized view management using graph neural network. http://dbgroup.cs.tsinghua.edu.cn/ligl/papers/gnnmv-tr.pdf.

[14] Y. Han, G. Li, H. Yuan, and J. Sun. An autonomous materialized view management system with deep reinforcement learning. In *ICDE*, pages 2159–2164. IEEE, 2021.

[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] J.-T. Horng, Y.-J. Chang, B.-J. Liu, and C.-Y. Kao. Materialized view selection using genetic algorithms in a data warehouse system. In *Computation-CEC99*, volume 3, pages 2221–2227. IEEE, 1999.

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

[18] 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.

[19] A. Kako, T. Ono, T. Hirata, and M. M. Halldórsson. Approximation algorithms for the weighted independent set problem. In D. Kratsch, editor, *Graph-Theoretic Concepts in Computer Science*, volume 3787, pages 341–350. Springer, 2005.

[20] 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.

[21] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*. OpenReview.net, 2017.

[22] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999*, pages 371–382. ACM Press, 1999.

[23] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *SIGMOD Conference 2018*, pages 489–504. ACM, 2018.

[24] 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.

[25] 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.

[26] G. Li, X. Zhou, and S. Li. Xuanyuan: An ai-native database. *IEEE Data Eng. Bull.*, 42(2):70–81, 2019.

[27] 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.

[28] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. opengauss: An autonomous database system. *Proc. VLDB Endow.*, 14(12):3028–3041, 2021.

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

[30] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In R. Bordawekar and O. Shmueli, editors, *aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018.

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

[32] 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*, pages 436–445. IEEE Computer Society, 2008.

[33] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In H. V. Jagadish and I. S. Mumick, editors, *SIGMOD*, pages 447–458. ACM Press, 1996.

[34] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN : A data warehouse intelligent cache manager. In *VLDB 1996*, pages 51–62. Morgan Kaufmann, 1996.

[35] M. K. Sohrabi and V. Ghods. Materialized view selection for a data warehouse using frequent itemset mining. *J. Comput.*, 11(2):140–148, 2016.

[36] J. Sun and G. Li. An end-to-end learning-based cost estimator. *PVLDB*, 13(3):307–319, 2019.

[37] J. Sun, G. Li, and N. Tang. Learned cardinality estimation for similarity queries. In *SIGMOD*, pages 1745–1757, 2021.

[38] 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.

[39] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A normalizing flow based cardinality estimator. *Proc. VLDB Endow.*, 15(1):72–84, 2021.

[40] Z. Wang, J. Zhang, J. Feng, and Z. Chen. Knowledge graph embedding by translating on hyperplanes. *Proceedings of the AAAI Conference on Artificial Intelligence*, 28(1), Jun. 2014.

[41] XuetangX. https://www.xuetangx.com/global/.

[42] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.

[43] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, 2019.

[44] X. Yu, G. Li, C. chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *ICDE 2020*, pages 196–207, 2019.

[45] 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.

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

[47] L. Zhang, C. Chai, X. Zhou, and G. Li. Learnedsqlgen: Constraint-aware SQL generation using reinforcement learning. In *SIGMOD Conference 2022*, pages 945–958. ACM, 2022.

[48] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *IEEE Trans. Knowl. Data Eng.*, 34(3):1096–1116, 2022.

[49] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *Proc. VLDB Endow.*, 15(1):46–58, 2021.

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

[51] R. Zhu, Z. Wu, C. Chai, A. Pfadler, B. Ding, G. Li, and J. Zhou. Learned query optimizer: At the forefront of ai-driven databases. In *EDBT*, pages 1–4, 2022.