ACM DIGITAL LIBRARY Association for Computing Machinery acm open

Latest updates: https://dl.acm.org/doi/10.1145/3769806

PDF Download
3769806.pdf
03 February 2026
Total Citations: 0
Total Downloads: 171

Published: 05 December 2025

Citation in BibTeX format

RESEARCH-ARTICLE

MAVIS: Materialized View for Subgraph Matching

LISHENG CAO, Peking University, Beijing, China

XIANGYANG GOU, UNSW Sydney, Sydney, NSW, Australia

LEI ZOU, Peking University, Beijing, China

WENJIE ZHANG, UNSW Sydney, Sydney, NSW, Australia

Open Access Support provided by:

UNSW Sydney

Peking University

Proceedings of the ACM on Management of Data, Volume 3, Issue 6 (December 2025)
https://doi.org/10.1145/3769806
EISSN: 2836-6573

MAVIS: Materialized View for Subgraph Matching

LISHENG CAO, Peking University, China
XIANGYANG GOU*, The University of New South Wales, Australia
LEI ZOU, Peking University, China
WENJIE ZHANG, The University of New South Wales, Australia

Subgraph matching is a fundamental task in graph analysis systems. In real-world applications, graph query engines often need to process a large number of subgraph matching queries, many of which share common substructures. Materializing the results of these shared subqueries as view patterns can enable computation reuse and significantly improve query efficiency. However, existing view materialization techniques suffer from either high memory usage or limited acceleration benefits. This paper introduces MAVIS, a novel view-based subgraph matching algorithm. MAVIS partitions view patterns into connected subgraphs called super-nodes and performs super-node-oriented materialization to balance memory consumption and processing speed. To further improve efficiency, it proposes a tree-based super-node partitioning method that avoids generating invalid candidates during materialization. Additionally, a customized query answering algorithm is designed to leverage the materialized views for faster query execution. Extensive experiments on real-world datasets demonstrate that MAVIS achieves a superior trade-off between memory usage and acceleration, and it outperforms existing approaches.

CCS Concepts: • **Information systems** → **Graph-based database models**.

Additional Key Words and Phrases: Graph; Subgraph Matching; Materialized View

ACM Reference Format:
Lisheng Cao, Xiangyang Gou, Lei Zou, and Wenjie Zhang. 2025. MAVIS: Materialized View for Subgraph Matching. *Proc. ACM Manag. Data* 3, 6 (SIGMOD), Article 341 (December 2025), 26 pages. https://doi.org/10.1145/3769806

1 INTRODUCTION

Subgraph matching is a fundamental building block of graph analysis, which requires finding all subgraphs in a data graph that are isomorphic or homomorphic to a given query graph. It is widely used in various domains, including knowledge graphs [2], social networks [16], software engineering [34], bioinformatics [39], chemistry [53], etc. This problem is known to be NP-hard [25]. Extensive research has been conducted to address this problem [4, 23, 25, 30, 45].

The majority of existing research focuses on solving a single query. However, in real-world applications, graph query engines need to handle a large number of queries. Many of these queries share common subqueries [7, 33]. Therefore, using the correlation among queries to enable computation sharing is a worthwhile optimization direction. Some studies have also been conducted in this area,

*Corresponding author

Authors' addresses: Lisheng Cao, leeson@pku.edu.cn, Peking University, Beijing, China; Xiangyang Gou, xiangyang.gou@unsw.edu.au, The University of New South Wales, Sydney, Australia; Lei Zou, zoulei@pku.edu.cn, Peking University, Beijing, China; Wenjie Zhang, wenjie.zhang@unsw.edu.au, The University of New South Wales, Sydney, Australia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2025/12-ART341
https://doi.org/10.1145/3769806

such as computation sharing among multiple queries [41] and building views to cache historical query results [14].

This paper explores the problem of accelerating subgraph matching with views. Consider a set of subqueries $\{\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_n\}$, where each $\mathcal{V}$ denotes one subquery that is frequently shared in the query workloads. We build materialized views based on them in the offline phase and use these views to accelerate future online queries that contain one or more $\mathcal{V}$. We call $\mathcal{V}$ **view patterns**. Note that the selection of view patterns $\mathcal{V}$, known as the view selection problem [37], is not the focus of this paper. These patterns can be chosen based on frequent historical queries or according to rules specific to particular applications. Additionally, there are view-selection methods in relational databases that are transferable to subgraph matching [37].

Popular view materialization methods can be divided into two classes. The first kind is called **query-based materialization (QM)**. It directly stores all the matching results of a given view pattern $\mathcal{V}$ in the data graph [14]. Figure 1(d) illustrates the view built with the QM method based on the view pattern in Figure 1(a) and the data graph in Figure 1(c). When answering an online query Q containing $\mathcal{V}$, we can retrieve the results of $\mathcal{V}$ and extend them to obtain the results of Q. However, this type of method is limited in scalability. The number of query results of subgraph matching is $O(|V_G|^{|V_\mathcal{V}|})$, where $|V_G|$ and $|V_\mathcal{V}|$ are the number of vertices in the data graph and the view pattern, respectively. Moreover, these results need to be stored in memory for fast retrieval. When the data graph is large, the QM method requires significant space and time overhead to build views and may even be infeasible due to memory limitations. The second kind of methods is called **vertex-based materialization (VM)** [5, 6, 31]. It only stores candidates for each vertex in the view pattern and maintains the connection between these candidates. The candidates in a VM view are filtered according to their neighborhood. v is a candidate of u if it has the same label as u, and has neighbors in the candidate set of each u' that connects to u. Building candidate sets with such a filtering rule takes polynomial time and memory cost $O(|V_\mathcal{V}||V_G|)$. However, we cannot guarantee that every candidate appears in the match result. Namely, some candidates will be invalid. Figure 1(e) illustrates the view built with the VM method. Note that v_{101} is an invalid candidate of u_1; it has neighbors in the candidate sets of u_0, u_2, and u_3, but does not appear in any match results. Similar to v_{102}. Due to the polynomial building cost, many subgraph matching algorithms build VM views on-the-fly for online queries to filter the candidate set and accelerate the matching process. However, extracting query results from VM views requires considerable time. As a result, when applied as offline built views, their acceleration effect is not as significant as the QM method. The invalid candidates further induce invalid computations in the match result extraction.

To address this problem, we proposed a materialized view framework for solving subgraph matching called **MAVIS**. MAVIS strikes a balance between the QM method and the VM method. It partitions a given view pattern into subgraphs called **super-nodes**. MAVIS stores the match results of each super-node in the data graph as its candidates and filters these candidates according to inter-super-node connections. Figure 1(f) illustrates the super-node partition of MAVIS, denoted as $P(\mathcal{V})$, where vertices enclosed in the same rectangle are partitioned into a single super-node. Figure 1(g) shows the materialized view built with MAVIS. The memory cost to store the candidates of one super-node s is $O(|V_G|^{|V_s|})$, where $|V_s|$ is the number of vertices in s. The total memory cost is $\sum_{s \in P(\mathcal{V})} |V_G|^{|V_s|}$, where $\sum_{s \in P(\mathcal{V})} |V_s| = |V_\mathcal{V}|$. Such a cost is much smaller than that of the QM method. Moreover, unlike the VM method, MAVIS joins the candidates of super-nodes rather than individual vertices, resulting in faster query answering.

Moreover, we propose an observation that guides the super-node partition: **the invalid candidates of VM methods are all induced by cycles**, as proven in Section 3. Similar to VM methods, MAVIS preserves super-node connections to filter candidates. Therefore, this observation also

holds for super-node candidates. These invalid candidates will not appear in the match results of $\mathcal{V}$. Storing them wastes memory and slows down query extraction. Therefore, we propose to partition the super-nodes in a tree-structure manner, which can eliminate cycles as well as the invalid candidates. Moreover, we propose a novel algorithm for tree partition that can minimize the maximum super-node size, and thus minimize the memory cost of views.

We also developed a query-answering algorithm for MAVIS that fully utilizes materialized views and seamlessly integrates with the subgraph matching order and pruning strategies.

In summary, we made the following contributions in this paper:

(1) We propose a subquery-based view materialization method that strikes a balance between memory usage and acceleration effect.
(2) We observe that tree-structured partitioning eliminates invalid candidates in view materialization. Based on this, we propose a tree partitioning algorithm that minimizes super-node sizes and reduces memory cost.
(3) We propose a view-based query answering method that fully leverages materialized views and seamlessly integrates with order selection methods and pruning strategies of existing subgraph matching algorithms.
(4) We conducted extensive experiments comparing MAVIS with baseline QM and VM methods, as well as more advanced counterparts including factorization [1, 38], tree decomposition [42], and MQO [41]. The query results demonstrate that MAVIS outperforms competing algorithms in terms of materialization efficiency and query processing efficiency.

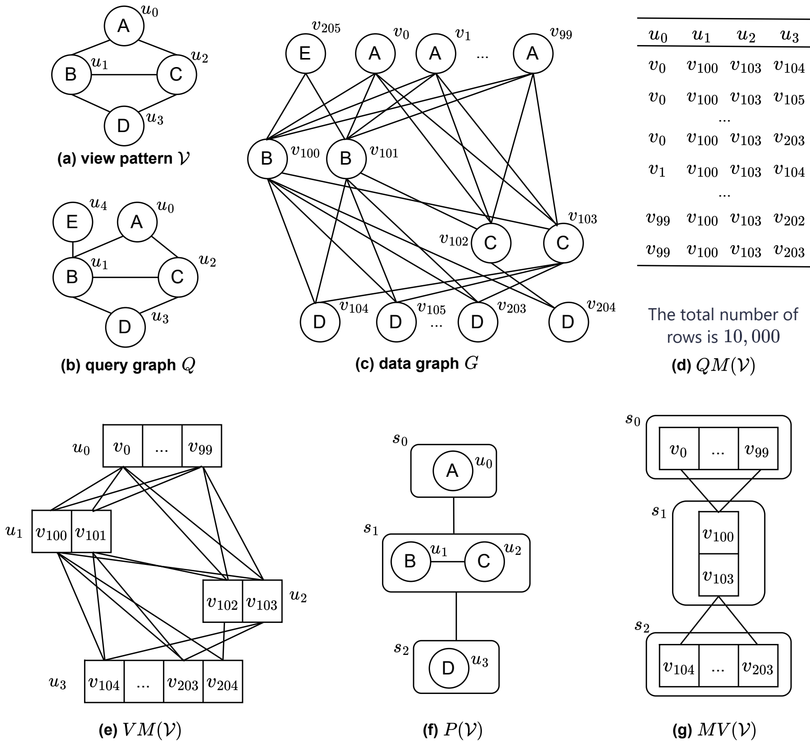


Fig. 1. A running example of materialized views

2 BACKGROUND

2.1 Preliminary

Table 1. Notations.

Notations	Descriptions
$g, Q, \mathcal{V}$ *and* G	graph, query graph, view pattern and data graph
V_g, E_g *and* L_g	vertex set, edge set and label function of g
Σ	label set
$g[S]$	subgraph of g induced by $S \subseteq V(g)$
$P(\mathcal{V})$	super-node partitioning of $\mathcal{V}$
$MV(\mathcal{V})$	materialized view of $\mathcal{V}$
$R(Q)$	rewritten query of Q

This paper focuses on undirected vertex-labeled graphs, following prior works [30, 45]. A graph g is defined as a tuple $(V(g), E(g), l_g)$, where $V(g)$ is the vertex set, $E(g)$ is the edge set, and $l_g : V(g) \mapsto \Sigma$ maps each vertex to a label. As in prior works [30, 45], we assume each vertex has a single label. If a vertex has multiple labels, it can be split into multiple single-labeled copies. Definitions of subgraph homomorphism and isomorphism follow.

Definition 2.1 (Subgraph Homomorphism (resp. Isomorphism)). Given a data graph $G(V_G, E_G, l_G)$ and a query graph $Q(V_Q, E_Q, l_Q)$, a subgraph homomorphism (resp. isomorphism) is a mapping (resp. injection) $M : V_Q \mapsto V_G$ that satisfies: 1) $\forall u \in V_Q$, $l_Q(u) = l_G(M(u))$; 2) $\forall (u, u') \in E_Q$, $(M(u), M(u')) \in E_G$.

We refer to such a mapping instance M as a match. In this paper, we consider a match to be represented as a set of tuples, where each tuple $\langle u, v \rangle$ represents the mapping from a query vertex u to a data vertex v. We call M' a partial match of M if and only if $M' \subseteq M$. Moreover, given a match M and a query vertex set S, we denote $M[S]$ to represent the partial match $\{\langle u, v \rangle | u \in S \wedge \langle u, v \rangle \in M\}$.

For example, $M = \{\langle u_0, v_0 \rangle, \langle u_1, v_{100} \rangle, \langle u_2, v_{103} \rangle, \langle u_3, v_{104} \rangle\}$ is a match for the view pattern illustrated by Figure 1(a) in the data graph depicted by Figure 1(c). Given $S = \{u_0, u_3\}$, $M[S]$ denotes the partial match $\{\langle u_0, v_0 \rangle, \langle u_3, v_{104} \rangle\}$.

Definition 2.2 (Subgraph Matching Problem). Given a data graph G and a query graph Q, find all distinct matches of Q in G.

In this paper, we explore the problem of view-based subgraph matching. A view pattern $\mathcal{V}$ is a query graph derived from the query workloads, which is usually selected from frequent subqueries [37]. A materialized view of $\mathcal{V}$, denoted as $MV(\mathcal{V})$, is a data structure that records matches of $\mathcal{V}$. We can use $MV(\mathcal{V})$ to aid in the matching of queries Q where $\mathcal{V}$ is a subgraph of Q, because matches of Q can be extended from matches of $\mathcal{V}$.

Note that we store the match results of homomorphisms in views. The isomorphism results are a subset of homomorphisms filtered by the injection requirement. Therefore, storing homomorphism results enables our views to support both semantics. We can further filter the results in the online matching phase for isomorphism queries. As the view needs to be stored in memory for fast retrieval, we require the materialized view to be efficient in both memory usage and match result extraction time.

2.2 Related work

Subgraph matching The subgraph matching problem has been extensively investigated [4, 6, 10, 23, 25, 30, 35, 45, 54]. Generally, these algorithms can be categorized into three main stages: filtering, ordering, and enumeration [54]. The filtering stage involves constructing candidate sets for each vertex or edge. The ordering stage determines an optimal matching sequence. Finally, the enumeration stage performs the matching search according to this predetermined order. Our framework also encompasses these three processes and can be well integrated with existing subgraph matching techniques during the query-answering process.

Materialized view Views are extensively utilized in relational databases and have been shown to significantly reduce redundant computations, thereby optimizing query performance [20–22]. Recently, the growing emphasis on graph data structures has spurred interest in graph views. However, substantial differences in storage, data access, and computation between relational and graph data necessitate the redesign of views to align with graph structures. Several graph-specific view frameworks have emerged: Kaskade introduced a comprehensive framework for view selection and query rewriting tailored to traversal queries [11]; Han and Ives proposed an implementation strategy for property graph views [24]; and Le et al. developed SPARQL rewriting techniques for RDF graphs [32]. For subgraph matching, materialization faces significant challenges due to the enormous solution space and computational complexity. Consequently, research has focused on specialized materialized views: Fan et al. optimized graph simulation queries using views [17] – a relaxation of subgraph isomorphism/homomorphism, though extracting exact matches from simulations remains NP-hard. Wang extended this approach to subgraph isomorphism [49] but only addressed query rewriting, not materialization. Lan et al. proposed summary graphs, a kind of VM view, as compressed materialized views [31], while VM views have been shown to be inefficient.

Factorization Factorization techniques [1, 26, 27, 38] compress conjunctive query results by identifying and reusing shared components, thereby addressing the redundancy induced by the Cartesian product. Our materialization method also falls under this paradigm. However, our method is distinct in that it employs a super-node-based materialization strategy. In Sections 7 and 8, we compare our method with other factorization-based approaches from both theoretical and experimental perspectives.

Tree partition and tree decomposition Tree partition is a technique that divides a graph into a tree-structured partitioning [15, 51]. It has been applied in graph drawing and graph coloring [3, 9, 12]. Tree decomposition is a similar technique [42, 47], widely used in query optimization, combinatorial optimization, and graph indexing [19, 29, 50]. Several works discuss the differences between these two techniques [13, 51]. We employ a variant of tree partition to construct materialized views. In Section 7, we will discuss in detail the distinctions between tree partition and tree decomposition, as well as the advantages of choosing tree partition.

3 OVERVIEW

Limitation of existing methods: Given a view pattern $\mathcal{V}$, there are two kinds of methods to build views for $\mathcal{V}$:

Query-based materialization (QM) directly stores all match results of $\mathcal{V}$. This method is very costly, as the number of match results is exponential ($O(|V_G|^{|V_{\mathcal{V}}|})$). However, we can directly retrieve match results of $\mathcal{V}$ from QM, which makes it very helpful for future online queries.

Vertex-based materialization (VM) only stores candidates for each view vertex in $\mathcal{V}$ and the connections among them. It enumerates the match results by joining the candidates of each view

vertex. In prior art [5, 31], the candidates of vertices are filtered using dual-simulation [36] rules. These rules require that a data vertex v is a candidate of u if

(1) v has the same label as u.
(2) For each neighbor u' of u, there is a candidate v' of u' that is the neighbor of v.

VM views can be built with polynomial time and memory costs ($O(|V_G||V_{\mathcal{V}}|)$). Some variants of VM views [46] contain edge candidates. Edge (v, v') is stored as a candidate of $(u, u') \in \mathcal{V}$ if v and v' are candidates of u and u', respectively. Edge candidates support fast retrieval of connected candidates, at an additional materialization cost of ($O(|E_G||E_{\mathcal{V}}|)$), but the total cost remains polynomial. However, extracting matches in VM requires costly multi-way joins. Additionally, VM views may include invalid candidates that are not in any match result, leading to unnecessary computation and reduced efficiency in query answering.

We propose the following theorem regarding the invalid candidates of the VM method:

Theorem 3.1. *When the view pattern is tree-shaped, there will be no invalid candidates.*

Proof. Consider a tree-shape view pattern $\mathcal{V}$, for an arbitrary candidate v of a view vertex u, we can find a match result of $\mathcal{V}$ containing v with the following procedure:

First, add u to set S and add mapping $\langle u, v \rangle$ to a partial match M.

Then, find a vertex $x \in S$ with at least one neighbor out of S. Suppose x is mapped to data vertex y in partial match M. For each neighbor x' of x that is not in S yet, according to the filtering rule of VM view, there must be at least one neighbor y' of y that is a candidate of x'. Add $\langle x', y' \rangle$ to the partial match M, and add x' to S. Because $\mathcal{V}$ is a tree, x' only has one neighbor x in S. Therefore, we do not need to check other edge connections to guarantee that M is a match of $\mathcal{V}[S]$.

Repeat the second step until S contains all vertices in $\mathcal{V}$, and M is a match of $\mathcal{V}$ now.

□

However, if $\mathcal{V}$ is not a tree, in the second step, a newly added vertex x' may have multiple neighbors in S, and we cannot guarantee that there is a candidate y' connected to the matched data vertices of all these neighbors. Therefore, we cannot necessarily obtain a match with the above process for $\mathcal{V}$ with cycles. In other words, invalid candidates are induced by cycles. Unfortunately, most real-world queries contain cycles. As a result, invalid candidates lead to a significant degradation in efficiency.

Figure 1(d) and (e) illustrate the materialized views constructed using the QM and VM methods, respectively. The query graph in Figure 1(b) contains a subgraph that matches the view pattern in Figure 1(a), enabling view-based optimization. For the QM view, we join the pre-computed matches of view patterns with vertices labeled 'E' based on topology. For the VM view, it is primarily used to accelerate the filtering process by initializing the candidate sets of query vertices with the candidate sets of the corresponding view vertices. We need to further enumerate matches with these candidates. The QM method is efficient at query time due to precomputed results; however, it incurs high construction and storage costs (e.g., 10,000 rows, exceeding the data graph in Figure 1(d)). The VM method is more compact but offers limited acceleration and may retain invalid candidates in cyclic queries, such as v_{101} and v_{102} in Figure 1(e).

There are also some methods like factorization [38] or answer graph [1] that can be used to build more effective materialized views, but they also fall into these two categories and suffer from similar drawbacks. We will discuss them in detail in Section 7.

Overview of MAVIS: To balance memory and retrieval costs, we propose partitioning the view pattern $\mathcal{V}$ into subgraphs called super-nodes. We store the match results of these super-nodes and apply dual-simulation rules for filtering. This subgraph-based materialization keeps memory usage

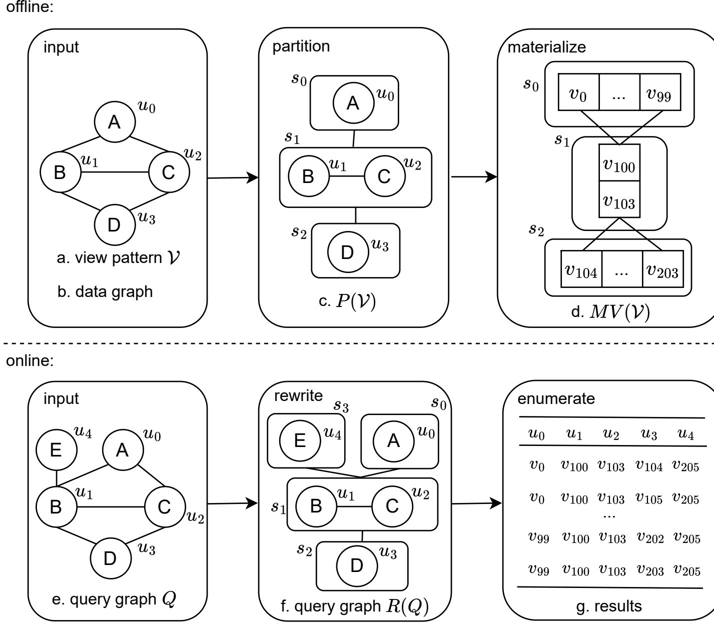


Fig. 2. Pipeline of MAVIS

lower than QM—since each super-node is size-bounded—and reduces retrieval costs compared to VM, as there are fewer super-nodes than vertices and, thus, fewer joins.

Specifically, we define super-nodes, super-edges, and super-node partitioning as follows:

Definition 3.2 (Super-node partitioning). Given a graph $\mathcal{V}$, a super-node partitioning of $\mathcal{V}$ is a graph $P(\mathcal{V})$. $P(\mathcal{V})$ contains a super-node set V^S and a super-edge set E^S, where:

(1) Each super-node represents a non-empty subset of $V_{\mathcal{V}}$;
(2) The vertex sets corresponding to different super-nodes in V^S are disjoint, and their union equals the entire vertex set of the original graph;
(3) There exists a super-edge between two super-nodes s and s' if and only if there is at least one edge (u, u') in the original edge set $E_{\mathcal{V}}$ such that $u \in s$ and $u' \in s'$.

For each super-node s, we find the match results of the induced subgraph $\mathcal{V}[s]$ in the data graph G. Each match result M is stored as a candidate of s. In the following sections, we will directly refer to M as the match result of the super-node s for convenience. Moreover, these candidates are further filtered according to super-edges. We define a **candidate super-edge** as follows:

Definition 3.3 (Candidate super-edge). Given a super-node partitioning $P(\mathcal{V})$ of a view pattern $\mathcal{V}$, a candidate super-edge (M, M') is a match result of a super-edge (s, s') if:

(1) M is a match of s and M' is a match of s'.
(2) For each $(u, u') \in E(\mathcal{V})$ and $u \in s$, $u' \in s'$, there is a data edge $(v, v') \in G$ where $\langle u, v\rangle \in M$ and $\langle u', v'\rangle \in M'$.

We define a match M of s **connected** to match M' of s' if and only if there is a candidate (M, M') of super-edge (s, s'). In this case, $M \cup M'$ is the match result of subgraph $\mathcal{V}[s \cup s']$. The match result M is stored as a candidate of super-node s if and only if, for each neighbor s' of s, there is a candidate

M' of s' connected to M. This condition is similar to the dual simulation rule. As a result, Theorem 3.1 also applies to MAVIS. If there are cycles in the super-node partition, invalid candidates will be induced. **To solve this problem, we propose to partition super-nodes into a tree shape.** Moreover, the cost of storing candidates of each super-node is exponential, namely $O(|V_G|^{|V_s|})$, where $|V_s|$ is the number of view vertices in the super-node. Therefore, large super-nodes will significantly increase the memory consumption. To balance correctness and efficiency, we aim to minimize super-node sizes while maintaining a tree structure. We thus propose an algorithm to compute the minimum connected tree partition.

The MAVIS pipeline (Figure 2) materializes view pattern matches offline to accelerate online query processing. Given a view pattern, MAVIS first partitions it into a tree-shaped super-node structure (Figure 2c). Then, it enumerates matches for each super-node, connects candidate super-edges, and filters invalid candidates using the dual simulation rule (Figure 2d). In Figure 2d, candidates are grouped by super-node and organized in a column-wise layout. For instance, the column with v_{100} and v_{103} represents one match result of s_1, with v_{100} mapped to u_1 and v_{103} mapped to u_2. Edges between super-node candidates represent candidate super-edges. At query time, if a query graph contains at least one view pattern, MAVIS rewrites it into a super-node structure based on the view patterns (Figure 2f) and enumerates matches based on the materialized results. Compared to QM and VM methods, MAVIS has several advantages. First, the materialized view contains no invalid candidates. The tree-shaped super-node partitioning eliminates cycles, allowing for effective filtering. For instance, invalid candidates v_{101} and v_{102} are removed in Figure 2d, as the match $\{\langle u_1, v_{101}\rangle, \langle u_2, v_{102}\rangle\}$ of s_1 does not connect to any match of s_2. Second, the materialized view occupies a reasonable amount of space. MAVIS only stores matches of subgraphs rather than the entire pattern, significantly reducing space consumption. In this example, the MAVIS view contains only 202 data vertices and 200 data edges, which is much smaller than the QM view in Figure 1(d).

4 TREE-SHAPE SUPER-NODE PARTITIONING

As stated in the above section, partitioning the super-nodes into a tree can avoid invalid candidates. This is known as a **tree partitioning** problem [15]. Besides, the vertex count of the largest super-node is defined as the width of the tree partitioning [15, 51]. As the memory cost is exponential with the width, we aim to minimize the width of the tree partitioning. Furthermore, it is crucial to keep the vertices within each super-node connected. If a super-node comprises multiple disconnected components, the candidate set for this super-node becomes the Cartesian product of the candidate sets of these components. Without edge constraints, this Cartesian product can become prohibitively large, significantly increasing materialization costs.

Definition 4.1 (Minimum Connected Tree Partitioning). Given a view pattern $\mathcal{V}$, a connected tree partitioning is a super-node partitioning that contains no cycles, with each super-node encompassing one connected component. The minimum connected tree partitioning is the one with the minimum width among all connected tree partitionings.

[15] proved that the minimum tree partitioning problem is NP-complete. When adding a constraint of inter-super-node connectivity, the problem remains NP-complete, which can be proven using an identical proof as [15]. [15] proposed a method to select a minimal tree partitioning instead, where splitting any super-node causes the graph partitioning to no longer be a tree [1]. However, a minimal tree partitioning may not be a minimum-width one and can even be far from the optimum.

We propose a search-based method to find the minimum connected tree partitioning, as detailed in Algorithm 1. Starting from an empty partition, we iteratively add vertices of $\mathcal{V}$ following an

[1]the original paper calls it a maximal quotient tree as the number of super-nodes is maximal. We call it minimal in respect of tree-width to keep consistent with our definition

order ϕ (lines 1–2). In our implementation, ϕ is the depth-first search (DFS) order from an arbitrary root, although any connected order, where each vertex connects to at least one previously added vertex, is valid. Each search state st represents a partition over the subset $\{\phi[0], \phi[1]...\phi[\|st\| - 1]\}$, where $\|st\|$ is the number of vertices in the current state. To explore successors of st, we add the next vertex u to st, either by inserting it into an existing super-node (lines 13–20) or by creating a new one (lines 21–23). To ensure correctness, we enforce two constraints at each step: the partition must remain a tree, and each super-node must be connected or potentially connectable in future steps (as will be discussed in the following). Invalid states are pruned accordingly (lines 16 and 22).

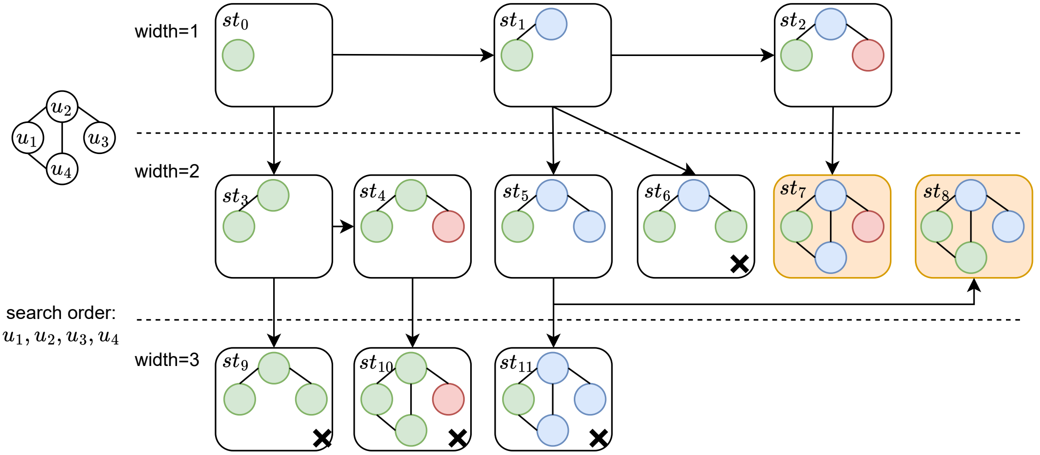


Fig. 3. Search process

Example 4.2. Figure 3 illustrates the search process, where each state is enclosed by a rounded rectangle and the state transition is represented by an arrow. Vertices of the same color are grouped into the same super-node. Despite there being two possible answers, st_7 and st_8, MAVIS terminates the partitioning process upon identifying the first solution in our experiment.

An exhaustive search can be implemented based on state transitions. However, it is inefficient to search over all states indiscriminately. Therefore, our algorithm incorporates various optimizations. The following are two notable optimizations:

Iterative Enlarging Width Limit Search. This pruning strategy is inspired by iterative deepening search (IDS). When $\mathcal{V}$ is sparse, the minimum tree width tends to be small, making the exploration of high-width states unnecessary. To leverage this, we divide the partitioning process into multiple iterations with increasing width limits (lines 5–9 in Algorithm 1). In each iteration, states exceeding the current width limit are not expanded. Instead, they are marked as held and saved for later (lines 17–18). If a complete partition meeting the current limit is found, it is returned as the optimal solution (line 8). Otherwise, the limit is incremented, and the next iteration begins from the previously stored held states (line 9), effectively avoiding redundant exploration.

Connectivity Pruning. After generating a new state st, the algorithm examines each super-node in the state. We cannot directly prune st upon finding a disconnected super-node. Such super-nodes may become connected after adding new vertices in subsequent transitions. If we prune st without further checks, we may miss its legal child states and thus miss the correct answer. Therefore, we employ an index fc. For each vertex pair u, u' in $\mathcal{V}$ and each integer $k \in [1, |V_{\mathcal{V}}|]$, we use

Algorithm 1: Minimum connected tree partitioning

Input: view pattern $\mathcal{V}$
Output: tree partitioning $P(\mathcal{V})$

1 $\phi \leftarrow DfsOrder(\mathcal{V})$
2 $st \leftarrow \emptyset$ /* Start search from an empty partition */
3 $InitialStates \leftarrow \{st\}$,
4 $P(\mathcal{V}) \leftarrow \emptyset$
5 **for** $limit \leftarrow 1$ **to** $|V_{\mathcal{V}}|$ **do**
6   **foreach** $st \in InitialStates$ **do**
7         $P(\mathcal{V}) \leftarrow$ Search(st, ϕ, $\mathcal{V}$, $limit$, $HeldStates$)
8   **if** $P(\mathcal{V}) \neq \emptyset$ **then return** $P(\mathcal{V})$
9     $InitialStates \leftarrow HeldStates$, $HeldStates \leftarrow \emptyset$
10 **Function** *Search(st, ϕ, $\mathcal{V}$, $limit$, $HeldStates$)*:
11   **if** $\|st\| = |V_{\mathcal{V}}|$ **then return** st
12     $u \leftarrow \phi_{\|st\|}$
13   **foreach** *super-node* $s \in st$ **do**
14         $s' \leftarrow s \cup \{u\}$ /* add u into s */
15         $st' \leftarrow st \setminus s \cup \{s'\}$ /* state transition */
16   **if** *CheckState(st', $\mathcal{V}$)* **then**
17   **if** *the width of st' exceeds limit* **then**
18                 $HeldStates \leftarrow HeldStates \cup \{st'\}$
19   **else**
20   **return** Search(st', ϕ, $\mathcal{V}$, $limit$, $HeldStates$)
21     $st' \leftarrow st \cup \{\{u\}\}$ /* Another transition possibility: add u as a new super-node */
22   **if** *CheckState(st', $\mathcal{V}$)* **then**
23   **return** Search(st', ϕ, $\mathcal{V}$, $limit$, $HeldStates$)
24   **return** $\emptyset$

$fc[u][u'][k]$ to indicate whether vertices u and u' can be connected through paths composed of the last k vertices in ϕ. This data structure can be preprocessed using a variant of the Floyd-Warshall algorithm [18]. For each super-node in state st, if it contains a vertex pair (u, u') that is not yet connected and cannot be connected by the remaining unpartitioned vertices (namely, $fc[u][u'][|V_{\mathcal{V}}| - \|st\|]$ is false), we can prune this state.

Example 4.3. In Figure 3, the state st_6 was skipped due to connectivity pruning, as u_1 and u_3 cannot connect through the remaining vertex u_4. The states st_9, st_{10} and st_{11} were pruned, since their widths are three, while the results have already been found in the two-width stage.

5 VIEW MATERIALIZATION

5.1 Basic Version

After the super-node partition, we can build the materialized view $MV(\mathcal{V})$, which consists of the following components:

(1) Candidate table of super-nodes: each candidate M of super-node s is a match result of s in the data graph G, which contains view vertex-data vertex mapping pairs $\langle u, v\rangle$ for each $u \in s$. Besides, for each neighbor s' of s, M should be connected to at least one candidate of s'.
(2) Materialized candidate super-edges: We store each candidate super-edge (M, M') as a connection between M and M'.

Theorem 5.1. *The materialized view of MAVIS satisfies the following two conditions:*
(1) Completeness: We can retrieve all the match results of $\mathcal{V}$ with the materialized view.
(2) Tightness: There is no invalid candidate in our materialized view.

Proof. We prove the completeness by contradiction. Assume that there exists a match M that cannot be retrieved. We decompose M into several partial matches M_i, each corresponding to a super-node s_i. Since M cannot be retrieved, there must exist at least one partial match M_k absent from the candidate set of s_k. Recall that the candidate set of a super-node is constructed by enumerating all possible matches of $\mathcal{V}[s_k]$ and filtering out invalid candidates based on the dual simulation rule. However, M_k is a valid match of s_k, and the dual simulation rule cannot eliminate M_k because for every neighbor s_t of s_k, M_t is a valid match of s_t and is adjacent to M_k. Therefore, the assumption leads to a contradiction.

Since the super-nodes are organized as trees, the tightness of the materialized view can be proved in the same method as Theorem 3.1; we omit it here due to space limitations. □

Moreover, we propose two additional techniques to further improve the efficiency of our materialized view: candidate edge factorization and oversized super-node splitting. We will introduce these two techniques in the following subsections.

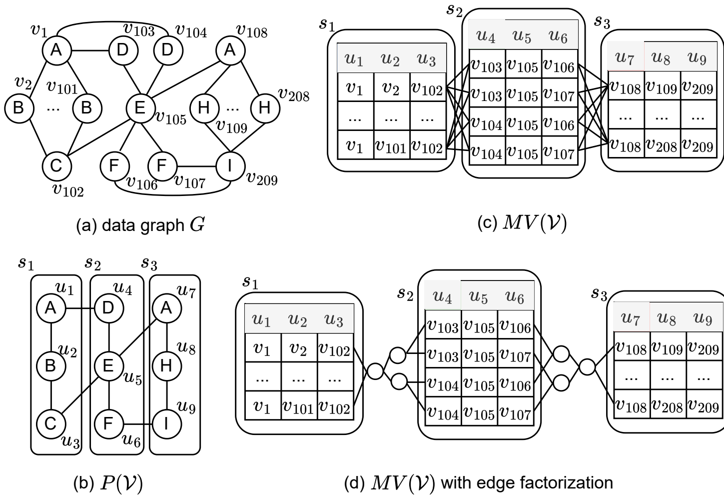


Fig. 4. An example of edge factorization

5.2 Candidate edge factorization

In the basic version, for each super-edge (s, s'), we store its candidates as connections among the candidates of super-node s and s'. The time and memory cost to materialize these connections is $O(|C(s)| \times |C(s')|)$, where $C(s)$ and $C(s')$ denote the candidate sets of s and s', respectively.

The cost can be significant, especially for large super-nodes that may have numerous candidates. Therefore, MAVIS utilizes edge factorization to reduce the redundancy of the edges. We present the definition of the port vertex, which is a key concept in edge factorization.

Definition 5.2 (port vertex). Given a super-edge (s, s'), u is a port vertex in s (resp. s') if and only if it has at least one edge connecting to a vertex in s' (resp. s). The set of port vertices in s for super-edge (s, s') is denoted by $s.pv(s')$.

According to the definition of a candidate super edge, if two candidates M_1 and M_2 of super-node s have the same mapping on port vertices $s.pv(s')$, they share the same neighbor set in the candidate set of s'. Thus, when building super-edge (s, s'), we can group candidates of s and s' by their mappings on $s.pv(s')$ and $s'.pv(s)$, and connect these groups via **group edges**. With such factorization, we reduce the memory and time costs to $O(|C(s.pv(s'))| \times |C(s'.pv(s))|)$, where $C(s.pv(s'))$ and $C(s'.pv(s))$ denote the match results of $s.pv(s')$ and $s'.pv(s)$, respectively.

Example 5.3. Figure 4 presents an example of edge factorization. Specifically, Figure 4(c) and (d) depict the materialized view without and with edge factorization, respectively. Matches are arranged row-wise for clarity. For the super-edge (s_1, s_2), the port vertex sets are $s_1.pv(s_2) = \{u_1, u_3\}$ and $s_2.pv(s_1) = \{u_4, u_5\}$. Candidates are grouped based on the mapping of port vertices, and group edges connect these groups accordingly. Edge factorization decreases the number of edges needed to materialize (s_1, s_2) from 400 to 106 (including edges that link super-node candidates to their groups).

While the cost of materializing a super-edge is reduced by edge factorization, it remains high if the super-edge connects two super-nodes with a large number of candidates. To further control this cost, we introduce a threshold δ_E on the number of candidates for each super-edge. Given a super-edge (s, s'), MAVIS incrementally materializes group edges for each candidate group pair and terminates materialization if the group edge count exceeds δ_E. The remaining candidate groups whose connections are not materialized are labeled with an "unmaterialized" tag.

Candidate super-edges accelerate the search process (see Section 6) by enabling efficient computation of local candidates. For a super-node s adjacent to an already enumerated super-node s', MAVIS retrieves the local candidates of s directly via the candidate super-edges of (s, s'). However, if the match of s' belongs to a group marked as unmaterialized, the corresponding super-edges are unavailable and cannot be used in this step.

5.3 Oversized Super-node Splitting

Even with the minimum-width tree partitioning, dense subgraphs may still produce large super-nodes with many candidates, leading to high memory and time costs. To mitigate this, we introduce a threshold δ_V on the number of candidates per super-node and split any super-node that exceeds this threshold.

Specifically, for each super-node s, we estimate the match result of $\mathcal{V}[s]$ in the data graph G using an approximate algorithm similar to [6], where $\mathcal{V}[s]$ denotes the subgraph of view pattern $\mathcal{V}$ induced by the vertices in s. We use the VM method to filter candidates for each view vertex in $\mathcal{V}[s]$. Then, we extract a spanning tree from $\mathcal{V}[s]$ and utilize a bottom-up dynamic programming approach to estimate the number of matches of the spanning tree in G. The detailed algorithm can be found in [6]. This estimation is overestimated, thus ensuring that the actual value of the remaining super-nodes will not exceed δ_V.

We split all super-nodes whose estimated candidate size exceeds δ_V. Since the original partition $P(\mathcal{V})$ is a minimum-width tree partition, splitting will break its tree structure, compromising view tightness. To minimize this impact, we select the split that minimizes $|E^S| - |V^S|$, where $|E^S|$ and

$|V^S|$ are the numbers of super-edges and super-nodes, respectively. A smaller value implies closer proximity to a tree structure (with $|E^S| - |V^S| = 1$ indicating a tree). While splitting may introduce some invalid candidates, it is a necessary trade-off between query efficiency and memory usage. δ_V can be tuned based on available memory. Higher values avoid splitting and preserve a tighter view when memory allows.

5.4 Complexity Analysis

The space and time cost of materializing views in MAVIS can be divided into 2 parts: the cost for super-nodes and the cost for super-edges.

MAVIS enumerates and stores all possible matches to compute the candidates of a super-node s. Consequently, the cost is $O(|V_G|^{|V_s|})$, where $|V_s|$ is the number of query vertices in s. The total cost for all super-nodes is $O(|V^S||V_G|^w)$, where $|V^S|$ is the number of super-nodes, and w is the width of $P(\mathcal{V})$, namely the maximum $|V_s|$. When adding a constraint δ_V on the super-node candidate size, the cost is reduced to $O(|V^S|\delta_V)$.

As for super-edges, the cost is $O(|V_G|^{|V_s|+|V_{s'}|})$ when materializing a super-edge (s, s'). Since the maximum super-node size is w, the total cost for materializing super-edges is $O(|E^S||V_G|^{2w})$, where $|E^S|$ is the number of super-edges. When adding a constraint δ_E on the super-edge candidate size, the cost decreases to $O(|E^S|\delta_E)$.

In conclusion, without constraints, the space and time cost of materializing views in the original MAVIS is $O(|V^S||V_G|^w + |E^S||V_G|^{2w})$. With constraints δ_V and δ_E, the cost is limited to $O(|V^S|\delta_V + |E^S|\delta_E)$. Though the complexity of original MAVIS is larger than the VM method ($O(|V_G||V_{\mathcal{V}}| + |E_G||E_{\mathcal{V}}|)$), adding constraints δ_V and δ_E makes this complexity manageable. In case of memory shortage, we can even decrease δ_V to a sufficiently small value, which prevents MAVIS from grouping any super-nodes. MAVIS will degrade to a VM in this case. According to our experiments, given a large δ_V and δ_E, the memory cost of MAVIS is larger than VM, but is still limited and much smaller than QM. Moreover, MAVIS is more effective than VM in query answering, with an acceleration of at most 31 times. In real-world applications, δ_E is recommended to be set to approximately $d \times \delta_V$, where d is the average degree of the data graph. Then, we can estimate δ_V and δ_E with the memory complexity formula $O(|V^S|\delta_V + |E^S|\delta_E)$ and the available memory of the application.

6 QUERY ANSWERING

This section details how MAVIS accelerates online query answering with a set of materialized views. The answering process consists of two steps: First, we transform a query graph Q into a rewritten query graph $R(Q)$ using view patterns. Next, we perform a rewritten query-based matching to enumerate all possible matches.

6.1 Rewriting

The first step is to construct a rewritten query $R(Q)$, which indicates how materialized views can be utilized during enumeration. Specially, a rewritten query is defined as follows:

Definition 6.1 (Rewritten Query). A rewritten query $R(Q)$ contains two components:

(1) A group of isomorphic functions $\mathcal{F}$, where each $F \in \mathcal{F}$ is an isomorphism from a subgraph of Q to a view pattern $\mathcal{V}$. Moreover, for each $u \in V_Q$, there is at most one $F \in \mathcal{F}$ that contains u.
(2) A super-node partitioning $P(Q)$. Each super-node s in $P(Q)$ contains either one vertex u not belonging to any F, or a set of vertices $\{u_1, u_2 ... u_i\}$ mapped to the same view pattern $\mathcal{V}$ by $F \in \mathcal{F}$, and $\{F(u_1), F(u_2) ... F(u_i)\}$ belongs to the same super-node s' in $\mathcal{V}$. For the second case, we call s materialized and build a mapping H between s and s'. Moreover, if two

materialized super-nodes s_1 and s_2 are mapped to the same view pattern $\mathcal{V}$, and the edges between them are all matched by $\mathcal{V}$. Then the super-edge between them is also materialized, and mapped to super-edge $(H(s_1), H(s_2)) \in \mathcal{V}$.

Example 6.2. Figure 5 illustrates a rewritten query graph $R(Q)$ utilizing $\mathcal{V}_1$ and $\mathcal{V}_2$. In this graph, the solid-line boxes within $R(Q)$ denote the materialized super-nodes, which are mapped to super-nodes in the view pattern. Additionally, only the vertex labeled 'H' is not mapped to any view pattern. We group it into a singleton super-node s_6 and map it to a null value. The super-edges denoted by bold lines represent the materialized super-edge.

We have the following theorem:

THEOREM 6.3. *If a super-node $s \in P(Q)$ is materialized, for any match M of Q in the data graph G, $M[s]$ is contained in the candidate set of $H(s)$ in the materialized view.*

Proof sketch: According to Definition 6.1, there is a subgraph of Q that contains the vertices of s and is isomorphic to a view pattern $\mathcal{V}$. Thus $M[s]$ is in a match of $\mathcal{V}$. According to the completeness proof of Theorem 5.1, $M[s]$ is included in the candidate set of $H(s)$. A more detailed proof is provided in the technical report [8] due to space limitations.

Based on this theorem, we aim to maximize the number of query vertices that are grouped into materialized super-nodes, so that we can utilize the materialized view rather than computing their candidates from scratch. This target relies on the selection of isomorphic functions $\mathcal{F}$ that maximize the number of $u \in V_Q$ involved in $F \in \mathcal{F}$. However, this problem is NP-hard, as it can be reduced from the exact cover problem [28] (see technical report [8]). To avoid the rewriting time dominating the query answering cost, we employ a greedy algorithm to build a sub-optimal $\mathcal{F}$. We iteratively select the maximum view pattern that can be matched to a subgraph of Q, and the matched subgraph does not overlap with the matches of previously chosen view patterns, and store its isomorphism to $\mathcal{F}$. After selecting the isomorphic functions $\mathcal{F}$, we can compute the super-node partitioning according to the definition.

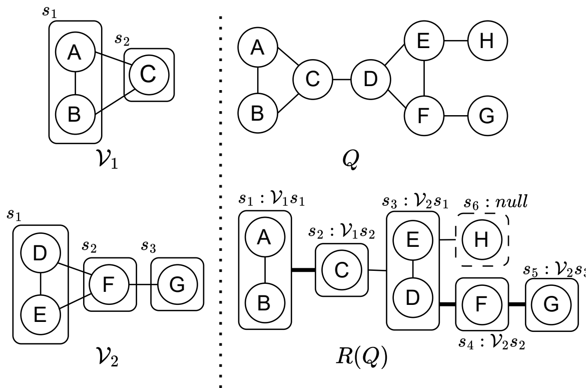


Fig. 5. An example of rewritten query

6.2 Rewritten query-based matching

In this section, we discuss how to answer the rewritten query with the materialized views in MAVIS. The query answering procedure of MAVIS is similar to existing subgraph matching algorithms, except that we enumerate matches in the granularity of super-nodes, and the candidates of materialized super-nodes are stored in the materialized views.

A general subgraph matching framework comprises three key components: filtering, ordering, and enumeration. Given a query graph, the framework first constructs a VM view on-the-fly to filter candidates for each query vertex based on vertex labels and neighbor connectivity. While specific algorithms may adjust the VM view differently, the core principle remains consistent. Next, a search order is determined by reordering the query vertices. The enumeration phase then explores all possible matches following this order, typically using a depth-first search (DFS) strategy. A partial match M is maintained and incrementally extended by adding mappings of query vertices to it, following the search order. For each query vertex u_i, a local candidate set is derived based on the current partial match and initial candidate sets. Each candidate v leads to a new DFS branch by adding $\langle u_i, v\rangle$ to M. Then we further explore mappings of u_{i+1} following this new branch. A branch ends when a full match is found or the local candidate set is empty. In either case, the algorithm backtracks by removing the last mapping and explores alternative branches.

MAVIS applies a similar framework, which also contains these three parts. However, MAVIS performs the matching process on the rewritten query composed of super-nodes and super-edges, which induces several differences.

Filtering: The objective of the filtering phase is to compute the candidate set for each super-node in the rewritten query $R(Q)$. Some super-nodes in $R(Q)$ have been mapped to super-nodes in view patterns, and we can use super-node candidates in the materialized views as the initial candidate set. However, these candidates are only filtered according to the topology of subqueries mapped by views, not the entire query graph. Therefore, we need to further filter these candidates. MAVIS builds a VM view for the entire query graph Q. Any initial candidate of super-node s containing mappings (u, v) that do not appear in the VM view needs to be excluded. In addition, a super-node s in $R(Q)$ may contain additional internal edges compared to $H(s)$ in the view pattern. In such cases, MAVIS eliminates candidates that fail to match these edges. For unmaterialized super-nodes $s' = \{u\}$ that are not mapped to any view patterns, their candidate sets are directly set to the candidate sets of the query vertex u in the VM view. Besides, MAVIS builds an additional VM view for each view pattern in the offline phase. The candidate sets in these VM views can be used to initialize the VM view of Q in this step, so that the building process can be accelerated. As VM views are lightweight, materializing them will not bring much memory cost.

Ordering: As the following enumeration is carried out in the granularity of super-nodes, we need to generate an ordering of super-nodes rather than query vertices in this step. Any existing ordering algorithm for subgraph matching can be utilized. We just need to apply it to the rewritten query graph $R(Q)$ made up of super-nodes and super-edges. In our experiments, we use the ordering algorithm of Rapidmatch [45] to generate a search order.

Enumeration: Enumeration is usually the most computationally intensive phase in the subgraph matching. This step generates an exponential number of search states relative to the query graph size. MAVIS clusters multiple query vertices into super-nodes and enumerates match results at the granularity of super-nodes. Consequently, the search space is significantly reduced, leading to enhanced search efficiency.

Algorithm 2 illustrates the enumeration process of MAVIS, which closely resembles the general subgraph matching framework. We enumerate the match results by iteratively extending the partial match M in a DFS manner. We add matches of each super-node to M following the search order

Algorithm 2: Match enumeration of MAVIS

```
    Input: A rewritten query graph R(Q), a search order φ and candidates of each super-node C
    Output: All the matches of Q into G
 1  Search(∅, R(Q), φ, 0, C)
 2  Function Search(M, R(Q), φ, i, C):
 3      if i = |V^R| then
 4          output M and return
 5      s ← φ[i]
 6      C_M[s] ← GetLocalCandidates(s, M, R(Q), φ, C)
 7      for M_s ∈ C_M[s] do
            /* Check conflict if in isomorphic semantics */
 8          M' ← M ∪ M_s
 9          Search(M', R(Q), φ, i + 1, C)
10  Function GetLocalCandidates(s, M, R(Q), φ, C):
11      {s'_j} ← GetMatchedNeighbors(s, M, R(Q), φ)
12      if All the super-edges (s'_j, s) are materialized then
13          return ⋂_{s'∈N^-(s)} N(M(s'))
14      else
15          res ← C(s)
16          foreach port vertex u of s do
17              lc(u) ← GetLocalCandidate(M, u)
18              res ← {M | M ∈ res ∧ M[u] ∈ lc(u)}
19          return res
```

ϕ, and output M when it contains matches of all super-nodes (lines 3-4 of Algorithm 2). When enumerating the matches of a super-node s, we need to compute the local candidates of s, which are selected from the candidate set generated in the filtering phase according to the current partial match M (line 6 of Algorithm 2). Specifically, there are two methods to compute local candidates in MAVIS. Suppose the neighbor set of s that is already matched in M is $N^-(s)$. For each $s' \in N^-(s)$, we denote its match result as $M(s')$. If the candidate super-edges that connect $M(s')$ with candidates of s are materialized in views, we can retrieve the candidates of s connected with $M(s')$, denoted as $N(M(s'))$, through these candidate super-edges. If all $N(M(s'))$ can be retrieved through candidate super-edges, we compute the local candidate of s as $\bigcap_{s' \in N^-(s)} N(M(s'))$ (lines 12-13 of Algorithm 2). However, some super-edges are not materialized, either due to space limitation (see Section 5.2) or because they are not mapped to any super-edges in view patterns. In this case, we use the second method, which computes local candidates by set intersection at the query-vertex level (lines 14-19 of Algorithm 2). This approach first determines the local candidate set $lc(u)$ for each port vertex u within s. The local candidates of u can be computed by joining the neighborhood of all $M(u')$ where u' is a matched neighbor vertex of u. Then we select local candidates of s by joining the candidate set $C(s)$ generated in the filtering phase with each $lc(u)$.

7 COMPARISON WITH RELATED METHODS

In this section, we will give a detailed discussion about approaches similar to MAVIS: **factorization-based methods [1, 26, 27, 38], tree decomposition [42], and MQO algorithm [41]**.

Factorization: Factorization techniques [1, 26, 27, 38] compress the results of conjunctive queries by identifying and reusing shared components, avoiding redundancy from Cartesian products. Our materialization method falls under this category but differs by adopting a super-node-based materialization strategy.

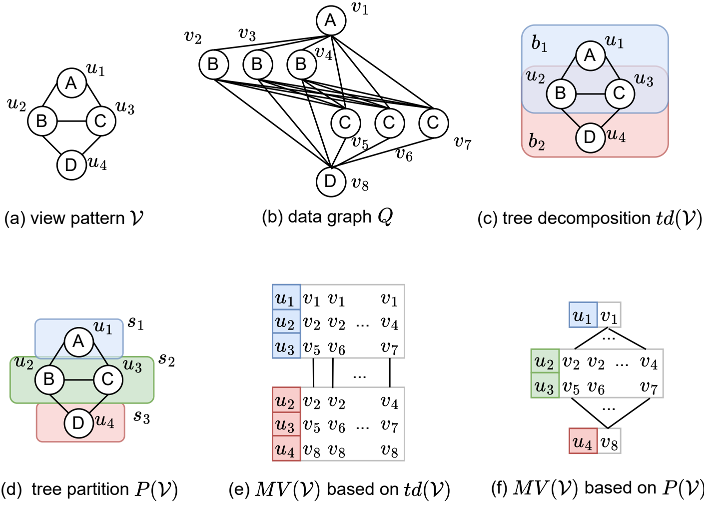


Fig. 6. Comparison between tree decomposition and tree partition

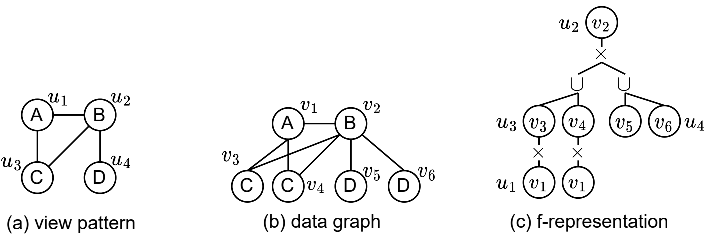


Fig. 7. An example of f-representation

The classic approach in [38] uses a trie-like f-representation to compress matches of a view pattern. An example is shown in Figure 7, where the matches can be generated in a top-down manner with the union and Cartesian product operators. A more compressed version of f-representation, named d-representation, further merges common subtrees and organizes the match results in a more compressed DAG manner. However, constructing the f-representation or d-representation remains exponential in terms of time and space complexity concerning the view pattern size, limiting its practicality. In contrast, MAVIS controls construction cost via cardinality estimation and constraints on super-node/edge candidates. Furthermore, vain cost may be induced by decompressing invalid

matches from the f-representation in the query answering phase. A match result of the view pattern is not necessarily in a match of the query graph, but we cannot effectively recognize such invalid matches in f-representation. On the other hand, MAVIS solves this by supporting flexible match ordering and early filtering at the super-node granularity level, as detailed in Section 6.2.

Subsequent methods extend factorization to dynamic scenarios [26, 27], but they are limited to acyclic queries, which differ from our static and cyclic setting.

Another factorization method, the answer graph [1], can be viewed as a refined VM view that decomposes the view pattern into triangles by adding chord edges, and filters candidates by checking their participation in triangle matches. This improves query answering efficiency over the baseline VM by reducing invalid candidates. However, it still enumerates matches one vertex at a time, limiting performance. In contrast, MAVIS enumerates one super-node at a time, significantly improving enumeration efficiency (see Section 6.2). Moreover, constructing the answer graph requires enumerating triangle matches, leading to a significant amount of construction time. In comparison, MAVIS achieves efficient construction by leveraging cardinality estimation and restricting super-node and super-edge candidate sets.

Tree decomposition: Tree decomposition [42] can be used as an alternative to the tree partition algorithm discussed in Section 4. The most obvious difference between tree decomposition and tree partition is that **tree partition generates non-overlapping vertex subsets, while tree decomposition produces vertex subsets that are necessarily overlapping**. This difference also makes tree decomposition not suitable for view materialization.

Tree decomposition is defined as follows: Given a graph G, construct a tree T where each node of T is a subset b_i (called a bag) of V_G, satisfying: 1) The union of all b_i is V_G; 2) For any edge (u, u'), there exists a bag containing both u and u'; 3) For any vertex u, overlapping bags that contain u form a connected subtree. Tree composition has also been extended to hypergraphs [43], where it generates overlapping bags of hypergraph vertices and edges. On the other hand, a tree partition generates a group of non-overlapping subsets (called super-nodes) of vertices. Super-nodes are connected by edges in E_G and form a tree shape.

We can design a materialization method based on tree decomposition rather than tree partition. Specifically, we enumerate the matches of each bag and join the match sets of bags based on their common vertices in the materialization phase. We can also materialize the relation among the matches with links. Figure 6(c) presents the tree decomposition of $\mathcal{V}$. Figure 6(e) shows the materialized view based on it.

However, materialized views based on tree decomposition suffer from high cost due to the overlap between bags. In Figure 6(e), the matches of u_2 and u_3 are repeated in the two bags. This problem does not exist in the materialized view based on the tree partition, as shown in Figure 6(f). Moreover, the overlapping leads to larger partitions. In the experiment, we observed that the width of tree decomposition is typically greater than that of tree partition [2]. A larger partition with more matches increases the computational cost of concatenating and filtering those matches.

MQO: [41] designed a data structure for multi-query optimization to store the match results of common subqueries. For simplicity, we denote it as MQO. It can also be applied to view materialization. While it shares the idea of partitioning the view pattern into super-nodes and materializing their candidate sets, it differs from MAVIS in three key aspects:

Tree partitioning strategy: MAVIS uses a minimum-connect-tree-partition, ensuring minimum tree partition width and inter-super-node connectivity. On the other hand, MQO adopts the MNL method with an additional step to split large super-nodes, which only achieves a minimal

[2] In this paper, the width of tree decomposition represents the size of the maximum bag without minus one, which is more straightforward but different from conventional definition [40].

tree partition. Moreover, it does not enforce connectivity within a super-node. MNL method is experimentally proved to be less efficient than our tree partitioning strategy (Section 8.4) in terms of query answering efficiency.

Materialization strategy: MQO materializes the entire view by enumerating all matches and then projecting them into each super-node, leading to a time complexity of $O(|V_G|^{|V_Q|})$, similar to QM views. In contrast, MAVIS materializes the matches of each super-node and connects them based on the super-edges. Its complexity is $O(|V^S||V_G|^w + |E^S||V_G|^{2w})$, where w represents the width of the tree partition. The materialization cost of MAVIS is much lower than MQO, as experimentally proved in Section 8.2.

Query answering strategy: [41] proposes a multi-query answering method that can also be used to answer a single query based on materialized views. It enumerates query results by joining matches from views and uncovered vertices. In contrast, MAVIS supports more flexible match ordering and filtering at the level of super-nodes, as detailed in Section 6.

We experimentally compared MAVIS with factorization-based view [38], answer graph [1], tree-decomposition based view [42] and MQO method [41] in Section 8.2 and 8.3. The results show that MAVIS outperforms these methods in both materialization efficiency and query-answering efficiency.

8 EXPERIMENTS

In this section, we evaluate MAVIS through experiments. We will conduct experiments on materialized view construction and subgraph matching optimization.

8.1 Experimental Setup

Data Graphs. We used four real-world graphs for the experimental evaluation. These datasets have been used in previous subgraph matching studies [6, 10, 30, 45, 54]. The statistical data for these datasets are shown in Table 2, where d denotes the average degree of vertices, and $|\Sigma|$ denotes the number of labels. The datasets Human, YouTube, and WordNet were obtained from [44], while Twitter was sourced from [48]. Here, Human is a biological information-related dataset. YouTube and Twitter relate to social networks. WordNet is a dataset that describes the relationships between words. Twitter is originally a directed graph. To ensure uniform comparison, we treated Twitter as an undirected graph by ignoring edge directions and removing duplicates. Human and WordNet come with predefined labels. For the remaining datasets, vertex labels were assigned by uniformly sampling from a label set of appropriate size.

Table 2. Statistics of data graphs.

Dataset	$\|V\|$	$\|E\|$	$\|\Sigma\|$	d
Human	4,674	86,282	44	36.9
YouTube	1,134,890	2,987,624	25	5.3
WordNet	76,853	120,399	5	3.1
Twitter	41,652,230	1,202,513,046	100	57.74

Query Graphs To comprehensively evaluate the performance of MAVIS, we generated a substantial number of query graphs with varying scales and densities. Query graphs with an average degree of 3 or higher are classified as dense, while others are classified as sparse. We denote each query type using its size and the first letter of its density classification. For example, '16d' refers to the query type in which the query graph contains exactly 16 vertices and is classified as dense. Since

materialized views rely on the similarity between queries for query optimization, to generate a set of similar queries, we employed a query graph generation approach analogous to the one proposed in [41]. We divided the query graphs into several groups. Within each group, a core query graph was initially constructed using a random walk on the data graph. The remaining query graphs in the group were then generated by transforming this core query graph. During the transformation, we randomly select 75% vertices from the core graph. These vertices and edges among them are fixed. The other parts were modified by adding or removing vertices and edges. These modifications were also performed via random walking to ensure that the resulting query graphs remained subgraphs of the data graph. Moreover, the size and density remain the same during modification. For instance, query graphs generated from a core graph classified into type '16d' also have 16 vertices and an average degree no smaller than 3. Figure 8 presents an example of query generation. The core query graph q_0 is constructed through random walking on the data graph. Query graphs q_1 and q_2 are transformed from q_0. The vertices and edges that remain unchanged are highlighted in bold. In our experiments, 10 groups of connected query graphs were generated for each query type, where each group contains 10 queries.

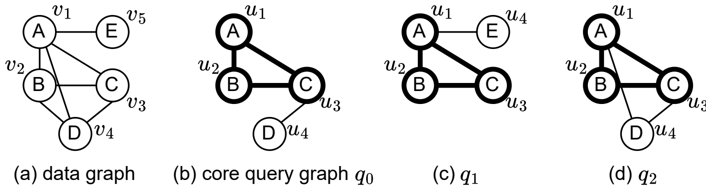


Fig. 8. An example of query generation

View Patterns Selecting views from the query workload is referred to as the view selection problem [37]. Typically, the view selection algorithm identifies a set of candidate views that frequently occur in the query workload. The importance of each view is evaluated based on its query acceleration effect and materialization cost, and the algorithm selects the most important views to materialize. In our experiments, we identified frequent subgraphs as candidate views based on [52]. Thereafter, the importance of each view is estimated by its size and the number of queries that can benefit from it. We chose 10 view patterns with the highest importance for each query type.

Metrics During the materialization phase, we configured a memory limit of 100 GB and a time limit of one hour for each query type. The execution time and memory consumption were recorded as efficiency metrics for each materialization algorithm. In the query answering experiment, since the prior subgraph matching work [6, 10, 30, 45] conducted the experiment on isomorphic semantics, we also configured all the view-based methods to generate isomorphic matches accordingly. To be consistent with prior work, the query processing algorithms are terminated for each query when any of the following conditions are satisfied: 1) all matches have been found; 2) the number of matches amounts to 10^5; and 3) the algorithm has executed for 300 seconds. The execution time is recorded as the efficiency metric.

Environment and Implementation. The experiments were conducted on a machine equipped with two 2.60GHz Intel(R) Xeon(R) E5-2640 v3 CPUs featuring hyperthreading (32 cores, 64 hyperthreads), along with 128GB of RAM, running the 64-bit CentOS Linux release 7.9.2009. We implemented MAVIS in C++. Our source code is available at [8]. All C++ code was compiled with g++ 10.2.1 and optimized with the -O3 flag. δ_V and δ_E (the limits on the number of matches for each

super-node and the number of candidates for each super-edge) are set to 10^5 and 10^7 for Human, YouTube, and WordNet, and to 10^6 and 10^8 for Twitter.

8.2 Materialization

We implemented VM, QM, factorization-based view (FDB), answer graph (AG) [1], tree-decomposition-based view (TD) [42, 47], and MQO-based view (MQO) [41]. Among them, FDB adopts d-representation [38]. The tree decomposition component of TD is implemented using a state-of-the-art algorithm [47]. MQO limits the size of each super-node to no more than 3, and any oversized super-node will be randomly divided into super-nodes that do not exceed the limit. This is consistent with [41]. Figure 9 and 10 illustrate the total space usage of materialized views and the total time cost of the materialization process. Among these, we mark cases of materialization failure (exceeding space or time limits) with "•". Methods that failed due to memory limits (e.g., QM and TD on WordNet) have no bars in the time cost figure, as they cannot be executed, and vice versa. VM completed materialization with modest space and time overhead. However, its acceleration effect in subsequent experiments was generally mediocre, as will be shown in Section 8.3. QM exceeded the space limit in all cases. FDB timed out on WordNet and Twitter due to its exponential time complexity. AG timed out on WordNet and Twitter because maintaining matches for query triangles incurred prohibitively high time costs. Nevertheless, the materialized views built by AG were smaller than those of VM. TD exceeded the space limit on WordNet and Twitter, caused by its large super-node (bag) size, leading to excessive matching results. We observed that the width of TD is generally larger than that of MAVIS. MQO passed only Human-16s and 16d; it timed out on all others due to an excessive number of view matches. We attempted materialization for one view in Human-24s by extending MQO's time limit to 12 hours, but it still timed out. Therefore, MQO is not suitable for view materializing in case of complex views or large datasets. At last, MAVIS was able to materialize all cases within the given constraints.

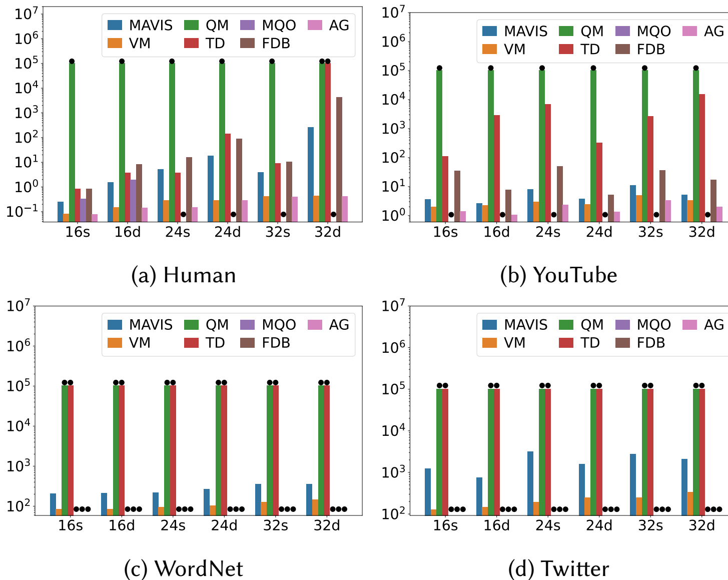


Fig. 9. Space occupation (MB) of materialization

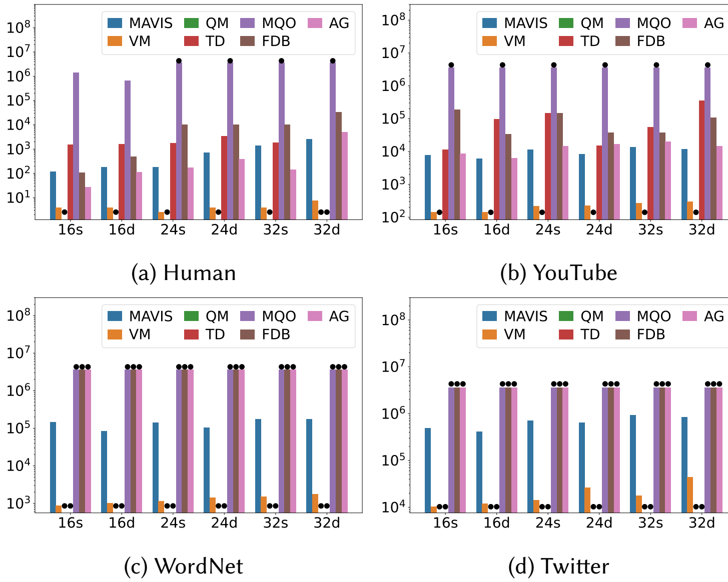


Fig. 10. Processing time (ms) of materialization

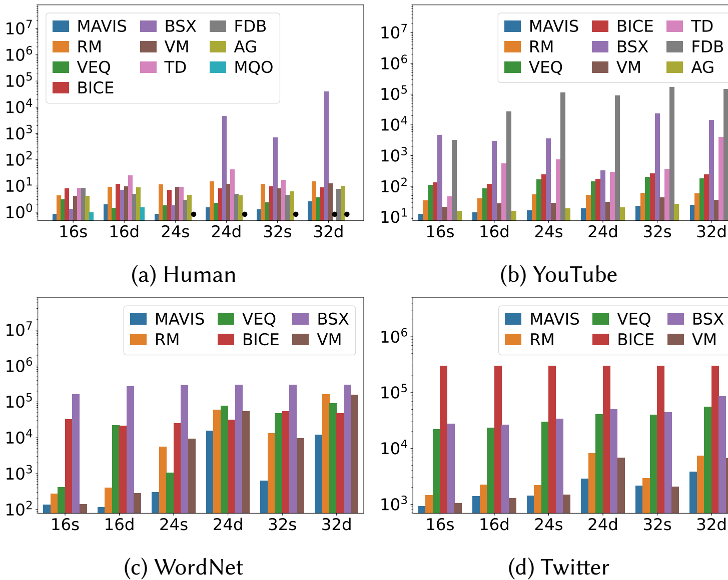
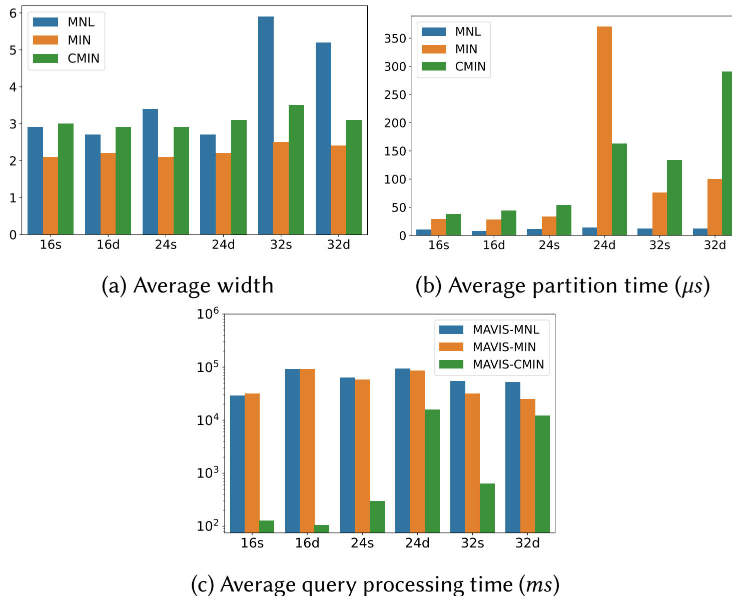


Fig. 11. Query processing time (ms)

8.3 Query Answering

In this section, we evaluate the query-answering effectiveness of the materialized view algorithms. For comprehensive comparison, we selected the following subgraph matching algorithms as baselines: RapidMatch (RM) [45], VEQ [30], BICE [10], and BSX [35]. We implement these algorithms using their respective open-source code from their GitHub repositories. In terms of materialized

views, for AG and VM, we accelerated the filtering step of subgraph matching using their materialized views; other steps followed the RM implementation. For FDB's materialized view, we extracted the matches from it and augmented them to full matches for the query. For TD's materialized view, we constructed a rewritten query for it, following the method described in Section 7. Specifically, a super-edge exists between two super-nodes if and only if they are adjacent bags in the tree decomposition or a query edge exists between them. Other steps followed the same procedure as MAVIS. For MQO's materialized view, we used its original answering method [41]. Figure 11 presents the experimental results. Cases of materialization failure are either omitted from the figure or marked with "●". VM and AG demonstrate significant optimization benefits in some scenarios, such as YouTube. Furthermore, AG outperforms VM in optimization effectiveness due to its ability to filter out more invalid candidates. However, since they primarily optimize the filtering step, their impact is limited in datasets where enumeration dominates the workload (e.g., WordNet). FDB's materialized view exhibits poor query efficiency on YouTube. This occurs because matches found in the view are not necessarily matches for the query, and FDB lacks mechanisms to filter out such cases. TD's enumeration efficiency is also suboptimal for two reasons: (1) The large super-node size in TD's materialized views leads to numerous matches, forcing the filtering phase to scan excessive candidates, thereby incurring high overhead. (2) The connections between super-nodes combine repeated vertex connections and edge connections, which distorts the rewritten query and thus hinders the algorithm from generating an optimal search order. MQO achieves efficiency comparable to MAVIS, but incurs a significantly higher materialization cost. Finally, MAVIS consistently achieves significant query speedups across all evaluated scenarios.



(a) Average width (b) Average partition time (μs)

(c) Average query processing time (ms)

Fig. 12. Performance of partition methods on WordNet

8.4 Evaluation of Tree Partitioning

In this section, we evaluate the performance of each tree partitioning algorithm. We selected the minimal tree partitioning method (MNL), as described in [15, 41], as our baseline approach.

Specifically, this algorithm first constructs a breadth-first search tree and subsequently partitions it in a bottom-up manner, layer by layer. We refer to our proposed method of minimum connected tree partitioning as CMIN. To investigate the impact of connectivity constraints, we also implemented the minimum tree partitioning method (MIN) by disabling the connectivity pruning step in CMIN. These experiments are conducted on WordNet using the view settings generated previously. We evaluated the average tree partitioning width and computational time for each algorithm across all configurations. The experimental results are summarized in Figure 12a and 12b.

In terms of partition time, MNL is more efficient due to its polynomial time complexity concerning the size of the view pattern, whereas CMIN and MIN involve exponential time complexity. Nevertheless, partitioning a view pattern using CMIN typically takes less than one millisecond, which is negligible within the overall pipeline. Compared to MIN, CMIN generally incurs higher computational costs due to the additional connectivity checking. However, in certain cases, this checking effectively prunes a large number of search states, thereby leading to substantial time savings. Regarding partition width, MNL typically produces a larger width than the other two methods. This increased width can significantly enhance the degradation effectiveness of the materialized view, as demonstrated in later experiments. In some cases, CMIN produces the highest average width due to the guarantee of connectivity. However, the difference is usually negligible (≤ 1) compared to MIN. Moreover, preserving connectivity is generally more important than marginal width reduction.

To further investigate the impact of tree partition, we conducted the following experiments. We replaced the tree partition algorithm of MAVIS with MNL and MIN. We denote these algorithms, including the original MAVIS, as MAVIS-MNL, MAVIS-MIN, and MAVIS-CMIN. We materialized views using them and processed queries with the materialized views. The queries and views are consistent with Section 8.3. Figure 12c presents the experimental results. Due to the super-edge factorization technique and the oversized super-node splitting technique, each algorithm was able to materialize successfully. However, MAVIS-MNL and MAVIS-MIN demonstrated significant performance degradation. This indicates that ensuring the connectivity within super-nodes and minimizing the partition width are crucial.

9 CONCLUSION

In this paper, we propose a subgraph matching materialization framework named MAVIS. This framework is capable of materializing match results of views and optimizing query processing by leveraging these materialized views. MAVIS employs a novel technique, referred to as tree-shape super-node partitioning, to materialize the results compactly. To maximize the effectiveness of this partitioning, we introduce the minimum connected tree partitioning problem and design an innovative search algorithm to solve it. Furthermore, we develop both the materialization method and the answering method, thereby constructing the complete pipeline of MAVIS. Extensive experimental evaluations demonstrate that MAVIS achieves significant performance improvements through the effective utilization of the view. For future work, we will try to migrate MAVIS to view materialization in relational databases.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under grant 2023YFB4502303, NSFC under grant 62532001, and ARC under grant DP230101445 and FT210100303. Xiangyang Gou is the corresponding author of this work.

REFERENCES

[1] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Stanley Clark, and Mark Chignell. 2020. Answer graph: Factorization matters in large graphs. *arXiv preprint arXiv:2011.04838* (2020).

[2] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2022. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal* (2022), 1–26.

[3] Noga Alon, Guoli Ding, Bogdan Oporowski, and Dirk Vertigan. 2003. Partitioning into graphs with only small components. *Journal of Combinatorial Theory, Series B* 87, 2 (2003), 231–243.

[4] Junya Arai, Yasuhiro Fujiwara, and Makoto Onizuka. 2023. Gup: Fast subgraph matching by guard-based pruning. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–26.

[5] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. 1447–1462.

[6] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[7] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2 (2020), 655–679.

[8] Lisheng Cao. 2025. *Repository for MAVIS*. https://github.com/Leeson63/MAVIS

[9] Paz Carmi, Vida Dujmović, Pat Morin, and David R Wood. 2008. Distinct distances in graph drawings. *arXiv preprint arXiv:0804.3690* (2008).

[10] Yunyoung Choi, Kunsoo Park, and Hyunjoon Kim. 2023. BICE: Exploring Compact Search Space by Using Bipartite Matching and Cell-Wide Verification. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2186–2198.

[11] Joana MF da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. 2020. Kaskade: Graph views for efficient graph analytics. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 193–204.

[12] Emilio Di Giacomo, Giuseppe Liotta, and Henk Meijer. 2005. Computing straight-line 3D grid drawings of graphs in linear volume. *Computational Geometry* 32, 1 (2005), 26–58.

[13] Marc Distel and David R Wood. 2024. Tree-partitions with bounded degree trees. In *2021-2022 MATRIX Annals*. Springer, 203–212.

[14] Chi Thang Duong, Trung Dung Hoang, Hongzhi Yin, Matthias Weidlich, Quoc Viet Hung Nguyen, and Karl Aberer. 2020. Fast and Accurate Efficient Streaming Subgraph Isomorphism. In *47th International Conference on Very Large Data Bases (VLDB)*.

[15] Anders Edenbrandt. 1986. Quotient tree partitioning of undirected graphs. *BIT Numerical Mathematics* 26 (1986), 148–155.

[16] Wenfei Fan. 2012. Graph pattern matching revised for social network analysis. In *Proceedings of the 15th international conference on database theory*. 8–21.

[17] Wenfei Fan, Xin Wang, and Yinghui Wu. 2015. Answering pattern queries using views. *IEEE Transactions on Knowledge and Data Engineering* 28, 2 (2015), 326–341.

[18] Robert W Floyd. 1962. Algorithm 97: shortest path. *Commun. ACM* 5, 6 (1962), 345–345.

[19] Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. 2012. Size and treewidth bounds for conjunctive queries. *Journal of the ACM (JACM)* 59, 3 (2012), 1–35.

[20] Ashish Gupta, Inderpal Singh Mumick, et al. 1995. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[21] Ashish Gupta, Inderpal Singh Mumick, et al. 1998. *Materialized views: techniques, implementations, and applications*. MIT press Cambridge, MA.

[22] Alon Y Halevy. 2001. Answering queries using views: A survey. *The VLDB Journal* 10 (2001), 270–294.

[23] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[24] Soonbo Han and Zachary G Ives. 2024. Implementation Strategies for Views over Property Graphs. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[25] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 337–348.

[26] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1259–1274.

[27] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2019. Efficient query processing for dynamically changing datasets. *ACM SIGMOD Record* 48, 1 (2019), 33–40.

[28] Richard M Karp. 2009. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*. Springer, 219–241.
[29] Ali Khanafer, François Clautiaux, and El-Ghazali Talbi. 2012. Tree-decomposition based heuristics for the two-dimensional bin packing problem with conflicts. *Computers & Operations Research* 39, 1 (2012), 54–63.
[30] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile equivalences: Speeding up subgraph query processing and subgraph matching. In *Proceedings of the 2021 International Conference on Management of Data*. 925–937.
[31] Michael Lan, Xiaoying Wu, and Dimitri Theodoratos. 2022. Answering Graph Pattern Queries using Compact Materialized Views.. In *DOLAP*. 51–60.
[32] Wangchao Le, Songyun Duan, Anastasios Kementsietsidis, Feifei Li, and Min Wang. 2011. Rewriting queries on SPARQL views. In *Proceedings of the 20th international conference on World wide web*. 655–664.
[33] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. 2012. Scalable multi-query optimization for SPARQL. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 666–677.
[34] Jingyue Li and Michael D Ernst. 2012. CBCD: Cloned buggy code detector. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 310–320.
[35] Yujie Lu, Zhijie Zhang, and Weiguo Zheng. 2025. BSX: Subgraph Matching with Batch Backtracking Search. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.
[36] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. 2014. Strong simulation: Capturing topology in graph pattern matching. *ACM Transactions on Database Systems (TODS)* 39, 1 (2014), 1–46.
[37] Imene Mami and Zohra Bellahsene. 2012. A survey of view selection methods. *Acm Sigmod Record* 41, 1 (2012), 20–29.
[38] Dan Olteanu and Maximilian Schleich. 2016. Factorized databases. *ACM SIGMOD Record* 45, 2 (2016), 5–16.
[39] N Pržulj, Derek G Corneil, and Igor Jurisica. 2006. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics* 22, 8 (2006), 974–980.
[40] Bruce A Reed. 2003. Algorithmic aspects of tree width. In *Recent advances in algorithms and combinatorics*. Springer, 85–107.
[41] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment* 10, 3 (2016), 121–132.
[42] Neil Robertson and Paul D. Seymour. 1986. Graph minors. II. Algorithmic aspects of tree-width. *Journal of algorithms* 7, 3 (1986), 309–322.
[43] André Schidler and Stefan Szeider. 2020. Computing optimal hypertree decompositions. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 1–11.
[44] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.
[45] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapidmatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.
[46] Shixuan Sun, Xibo Sun, Bingsheng He, and Qiong Luo. 2022. Rapidflow: An efficient approach to continuous subgraph matching. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2415–2427.
[47] Hisao Tamaki. 2022. Heuristic computation of exact treewidth. *arXiv preprint arXiv:2202.07793* (2022).
[48] twitter. 2010. *Twitter follower network*. https://snap.stanford.edu/data/twitter-2010.html
[49] Xin Wang. 2017. Answering graph pattern matching using views: a revisit. In *Database and Expert Systems Applications: 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part I 28*. Springer, 65–80.
[50] Fang Wei-Kleiner. 2016. Tree decomposition-based indexing for efficient shortest path and nearest neighbors query answering on graphs. *J. Comput. System Sci.* 82, 1 (2016), 23–44.
[51] David R Wood. 2009. On tree-partition-width. *European Journal of Combinatorics* 30, 5 (2009), 1245–1253.
[52] Xifeng Yan and Jiawei Han. 2002. gspan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE, 721–724.
[53] Xifeng Yan, Philip S Yu, and Jiawei Han. 2004. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 335–346.
[54] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–29.

Received April 2025; revised July 2025; accepted August 2025