Answering Graph Pattern Queries using Compact Materialized Views

Michael Lan
New Jersey Institute of Technology
New Jersey, USA
mll22@njit.edu

Xiaoying Wu*
School of Computer Science, Wuhan University
Wuhan, China
xiaoying.wu@whu.edu.cn

Dimitri Theodoratos
New Jersey Institute of Technology
New Jersey, USA
dth@njit.edu

ABSTRACT

We address the problem of evaluating graph pattern queries involving reachability (edge-to-path mapping) and direct (edge-to-edge mapping) relationships under homomorphisms on data graphs using materialized graph pattern views. We propose an original approach for view materialization which materializes views as summary graphs, an approach that records, in a compact way, all the homomorphisms of the view to the data graph. In this context, we characterize view usability in terms of query edge coverage and provide necessary and sufficient conditions for answering queries using views. We design algorithms for deciding whether a query can be answered using a set of views, for generating the summary graph of a query from the view materializations, and for producing a minimal view set capable of answering a query. Our experimental evaluation demonstrates that our approach outperforms, by several orders of magnitude, a state-of-the-art approach which does not use materialized views, and substantially improves upon its scalability.

ACM Reference Format:
Michael Lan, Xiaoying Wu, and Dimitri Theodoratos. 2022. Answering Graph Pattern Queries using Compact Materialized Views. In *Proceedings of ACM Conference, Edinburgh, UK, March 29, 2022 (DOLAP '22),* 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graphs model complex relationships between entities in a multitude of modern applications. A fundamental operation for querying, exploring and analyzing graphs is graph matching, which consists of finding the matches of a query graph pattern in the data graph. Graph matching is crucial in many application domains, such as social network analysis [8], protein interaction analysis [27], cheminformatics [28], knowledge bases [1, 30], and road network management [3].

Existing approaches are characterized by: (a) the type of edges the patterns have, and (b) the type of morphism used to map the pattern to the data graph. An edge in a query pattern can be either a child edge, which represents a parent-child relationship in the data graph (edge-to-edge mapping) [4, 6, 10, 24, 25, 31, 33], or a descendant edge, which represents a node reachability relationship in the data graph (edge-to-path mapping) [7, 13, 22]. The morphism determines how a pattern is mapped to the data graph and, in this context, it can be an isomorphism (injective mapping) [6, 25, 31, 33] or a homomorphism (general mapping) [4, 7, 13, 22, 24]. Graph simulation [17] and its variants [12, 23] are another way to match patterns to data graphs.

*The research of this author was supported by the National Natural Science Foundation of China under Grant No. 61872276.

Earlier contributions considered isomorphisms and edge-to-edge mappings, while more recent ones focus on homomorphic mappings. By allowing edge-to-path mapping on graphs, patterns with descendent edges are able to extract matches "hidden" deeply within large graphs which might be missed by patterns with only child edges. On the other hand, the patterns with child edges can discover important parent-child relationships in the data graph which can be missed by patterns with only descendant edges. We adopt, in this paper, a general framework that considers patterns which allow both child and descendant edges. This framework incorporates the benefits from both types of edges.

Graph pattern matching is an NP-hard problem, even for isomorphic matching of patterns with only child edges [15]. Finding the homomorphic matches of query patterns which involve descendant edges on a data graph is more challenging. Descendant edges in a query pattern increase the number of results since they are offered more chances to be matched to the data graph compared to child edges. Furthermore, finding matches of descendant edges to the data graph is an expensive operation and requires the use of a node readability index [9, 18, 29]. Despite the use of reachability indexes, evaluating descendant edges remains a costly operation. Existing approaches for evaluating pattern queries with reachability relationships produce a huge number of intermediate results (that is, results for subgraphs of the query graph which do not appear in any result for the query). As a consequence, existing approaches do not scale satisfactorily when the size of the data graph increases.

Answering queries using materialized views is a well known technique for improving the performance of query evaluation and for evaluating queries without accessing the base data, in particular in a distributed environment [11, 14, 16, 20, 40]. The idea is to pre-compute and store the answers of views and to rewrite an incoming query using exclusively the view materializations, if the query language is closed [16], or to otherwise provide a process for computing the query answer from the view materializations [20]. Materialized views can also be effectively used for addressing the data scalability problem of queries.

In this paper we adopt a novel approach for materializing graph pattern views over data graphs: a view materialization is a graph, called a *summary graph* of the view, which is a compact representations of the view answer. A summary graph constitutes a search space for the view answer and the view results can be enumerated by applying multiway joins while traversing the graph.

Contribution. The main contributions of the paper are as follows:

- We consider hybrid queries (i.e., queries involving parent-child and reachability relationships) to be mapped against large data graphs using homomorphisms. In this context, we address the problem of answering graph pattern queries using materialized views. This problem has not been addressed before for this type of queries and views.

- We suggest an original way for representing materialized views as summary graphs. A summary graph of a view compactly encodes all the homomorphisms of the view to the data graph in a structure which is, typically, much smaller than the view answer (Section 3).
- We characterize answering a query using one or multiple views in terms of query edge coverage from view edges. We provide necessary and sufficient conditions for answering a query using materialized views (Section 4).
- We design an algorithm which identifies the views from a pool of materialized views that can be used for answering a query, and computes the summary graph of a query from the summary graphs of these views (Section 5).
- Not all available views might be needed for answering a query. We provide an algorithm which finds a minimal set of views (this is a set of views which does not include redundant views) from the view pool (Section 5).
- We run extensive experiments to evaluate the efficiency and scalability of our approach for answering queries using views. We also compare it with a previous state-of-the-art approach which does not use materialized views. Our results show that our view-based approach outperforms that approach by orders of magnitude in terms of execution time and displays better scalability (Section 6).

2 DATA GRAPH AND GRAPH PATTERN QUERIES

In this section, we present the data model, graph pattern queries, edge-to-path mappings and homomorphisms. We also present related concepts that are needed for the results presented later.

Data Graph. We assume that the data is presented in the form of a data graph defined below.

Definition 2.1 (Data Graph). A *data graph* is a directed node-labeled graph $G = (V, E)$ where V denotes the set of nodes and E denotes the set of edges (ordered pairs of nodes). Let $\mathcal{L}$ be a finite set of node labels. Each node v in V has a label $label(v) \in \mathcal{L}$ associated with it.

Given a label a in $\mathcal{L}$, the inverted list I_a is the list of nodes in G whose label is a. Figure 1(a) shows a data graph G with labels a, b, c, d and e. Label subscripts are used to distinguish nodes with the same label. The inverted list of label a in G is $I_a = \{a_1, a_2, a_3, a_4, a_5\}$

Definition 2.2 (Node reachability). A node u is said to *reach* node v in G, denoted by $u \prec v$, if there exists a path from u to v in G. Clearly, if $(u, v) \in E$, then $u \prec v$. Abusing tree notation, we refer to v as a *child* of u (or u as a *parent* of v) if $(u, v) \in E$, and v as a *descendant* of u (or u is an *ancestor* of v) if $u \prec v$.

Given two nodes u and v in G, in order to efficiently check whether $u \prec v$, graph pattern matching algorithms use some kind of reachability indexing scheme. In most reachability indexing schemes the data graph node labels are the entries in the index for the data graph [29]. Our approach can flexibly use any labeling scheme to check node reachability. In order to check if v is a child of u, the basic access information of the graph G can be used; for example, adjacency lists.

Queries. We consider graph pattern queries that involve child and/or descendant edges.

Definition 2.3 (Graph Pattern Query). A query is a graph Q. Every node x in Q has a label $label(x)$ from $\mathcal{L}$. There can be two types of edges in Q. A *child* (resp. *descendant*) edge denotes a child (resp. descendant) structural relationship between the respective two nodes. A graph pattern that contains both child and descendant edges is a *hybrid* graph pattern.

Intuitively, a child edge represents an edge in the data graph G. A descendant edge represents a path of edges in G. Figure 1(b) shows a query Q. Single line arrows denote child edges while double line arrows denote descendant edges.

The *match set* $ms(x)$ of a node x in Q is the inverted list $I_{label(x)}$ of the label of node x. A *match* of an edge $e = (x, y)$ in Q is a pair (u, v) of nodes in G such that $label(x) = label(u)$, $label(y) = label(v)$ and: (a) $u \prec v$ if e is a descendant edge, while (b) (u, v) is an edge in G if e is a child edge. The *match set* $ms(e)$ of e is the set of all the matches of e in G.

The match set $ms(e)$ of an edge $e = (x, y)$ on a data graph G can be computed using the match sets $ms(x)$ and $ms(y)$ along with reachability information on the nodes of G (if e is a descendant edge), or the adjacency lists for the nodes of G (if e is a child edge).

The notion of node reachability provided in Definition 2.2 for nodes in a data graph is extended to nodes in a graph pattern in a natural way.

Homomorphisms. Queries are matched to the data graph using homomorphisms.

Definition 2.4 (Graph Pattern Homomorphism to a Data Graph). Given a graph pattern Q and a data graph G, a *homomorphism* from Q to G is a function h mapping the nodes of Q to nodes of G, such that: (1) for any node $x \in Q$, $label(x) = label(h(x))$; and (2) for any edge $(x, y) \in Q$, if (x, y) is a child edge, $(h(x), h(y))$ is an edge of G, while if (x, y) is a descendant edge, $h(x) \prec h(y)$ in G.

Figure 1(a,b) shows a homomorphism h of query Q to the data graph G. Query edges (A_1, B_2) and (C_1, B_2) which are child edges, are mapped by h to an edge in G. The other edges of Q which are descendant edges are mapped by h to a path of edges in G (possibly consisting of a single edge).

Homomorphisms can be also defined between query graph patterns as follows.

Definition 2.5 (Homomorphism between Graph Patterns). Given a graph pattern V and another graph pattern Q, a *homomorphism* from V to Q is a function h mapping the nodes of V to nodes of Q, such that: (1) for any node $x \in V$, $label(x) = label(h(x))$; and (2) for any edge $(x, y) \in V$, if (x, y) is a child edge, $(h(x), h(y))$ is a child edge of Q, while if (x, y) is a descendant edge, $h(x) \prec h(y)$ in Q.

Note that if (x, y) is a descendant edge in V, the path (of child and/or descendant edges) in Q from $h(x)$ to $h(y)$, can be a single child or descendant edge. Figures 2(a) and (b) show a query Q and a query (view) V_1 with a homomorphism from V_1 to Q.

Query Answer. We call an *occurrence* of a pattern query Q on a data graph G a tuple indexed by the nodes of Q whose values are the images of the nodes in Q under a homomorphism from Q to G.

Definition 2.6 (Query Answer). The *answer* of Q on G, denoted as $Q(G)$, is a relation whose schema is the set of nodes of Q, and whose instance is the set of occurrences of Q under all possible homomorphisms from Q to G.

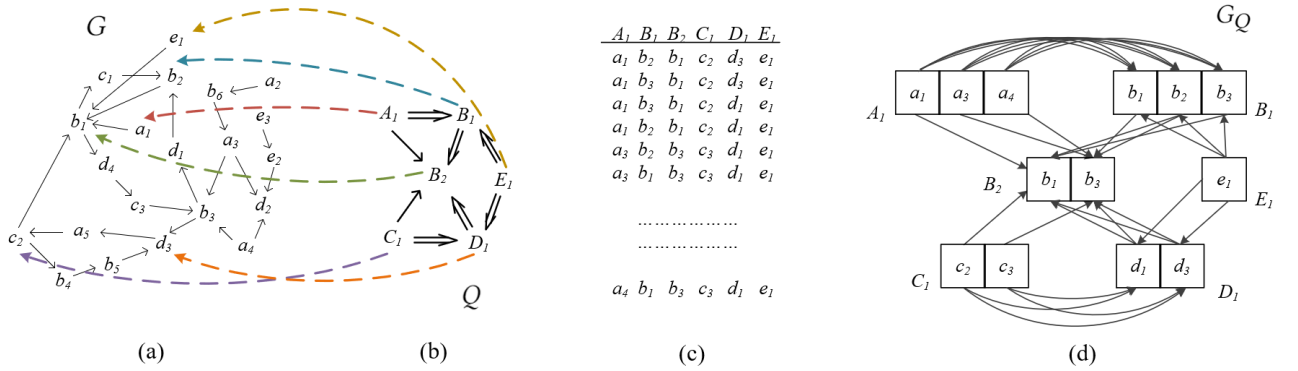


Figure 1: (a) A data graph G, (b) A graph pattern query Q and a homomorphism from Q to G, (c) The answer of Q on G, (d) A summary graph G_Q of Q on G.

Figure 1(c) shows the answer of a query Q on a data graph G. If x is a node in Q labeled by label a, an *occurrence* of x in G is the image $h(x)$ of x in G under a homomorphism h from Q to G. The *occurrence set of x on G*, denoted as $os(x)$, is the set of all the occurrences of x on G. This is a subset of the match set $ms(x)$ containing only those nodes that occur in the answer of Q on G for x (that is, nodes that occur in the column x of the answer). For instance, the occurrence set of node A_1 of query Q in Figure 1 is $\{a_1, a_3, a_4\}$.

If $e = (x, y)$ is an edge in Q, an *occurrence* of e in G is a pair (u, v) of nodes from G such that $u = h(x)$ and $v = h(y)$, where h is a homomorphism from Q to G. The *occurrence set of e on G*, denoted as $os(e)$, is the set of all the occurrences of e on G. This is the set of pairs (u, v) of nodes in G such that there is an occurrence t of Q on G with $t.x = u$ and $t.y = v$ (that is, $os(e)$ is in the projection of the answer of Q on G on the columns x and y). Clearly, $os(e) \subseteq ms(e)$. In the example of Figure 1, the occurrence set of the edge (A_1, B_2) of query Q is $\{(a_1, b_1), (a_3, b_3), (a_4, b_3)\}$.

3 SUMMARY GRAPHS, VIEWS AND VIEW MATERIALIZATIONS

A Compact Representation for Query Answers. The number of homomorphic matches of a graph pattern query on a data graph can very large. Therefore, we use *summary graphs* to compactly encode all possible homomorphisms of a query to a data graph.

Definition 3.1 (Query Summary Graph). The summary graph G_Q of a pattern query Q is a k-partite graph where k is the number of nodes in Q. Graph G_Q has an independent node set, denoted $cos(q)$, for every node $q \in Q$ such that $os(q) \subseteq cos(q) \subseteq ms(q)$. Every node in $cos(q)$ is incident to an edge in G_Q if q is incident to an edge in Q. The set $cos(q)$ is called the *candidate occurrence set* of q in G_Q. For every edge $e_q = (x, y)$ in Q, the set of edges $cos(e_q)$ between the data graph nodes in the sets $cos(x)$ and $cos(y)$ satisfies the inclusion relationships: $os(e_q) \subseteq cos(e_q) \subseteq ms(e_q)$. The set $cos(e_q)$ is called the *candidate occurrence set* of e_q in G_Q.

Figure 1(d) shows a summary graph G_Q for the query Q of Figure 1(a), and Figure 2(c) shows a summary graph for the query (view) V_1 of Figure 2(b). A summary graph G_Q losslessly summarizes all the occurrences of Q on G. Similarly to factorized representations of query results studied in the context of classical databases and probabilistic databases [26], G_Q exploits computation sharing to reduce redundancy in the representation and computation of query results. Besides recording candidate occurrences sets for the edges of query Q, a summary graph also records how the edges in the candidate occurrence sets can be joined to form occurrences for query Q. A summary graph G_Q represents a search space for the answer of Q on G. We later present an algorithm for enumerating the results of Q on G from a summary graph G_Q.

We define a partial order $\prec$ on the summary graphs of a query Q. Let G_Q^1 and G_Q^2 be two summary graphs for Q. Then $G_Q^1 \prec G_Q^2$ iff for every edge e in Q, the candidate occurrence set for e in G_Q^1 is a subset of the candidate occurrence set for e in G_Q^2. Partial order $\prec$ has a least element G_Q^a called the *answer graph* of Q on G, and a greatest element G_Q^m called the *match graph* of Q on G. One can see that for any edge e in Q, the candidate occurrence set for e in G_Q^a is the occurrence set $os(e)$, while the candidate occurrence set for e in G_Q^m is the match set $ms(e)$.

Views and View Materializations. A *view* is a named query. The class of views is not restricted. Any type of query can be a view. We materialize views on a data graph by storing a summary graph of this view.

Definition 3.2 (View Materialization). The *materialization of a view V* on a data graph G is a summary graph of V on G. A view is characterized as *materialized* if it has a materialization.

Figure 2(c) shows the materialization of view V_2 of the same figure. One can see that this summary graph is the answer graph of V_2.

4 MATERIALIZED VIEW USABILITY IN GRAPH PATTERN QUERY ANSWERING

We define now when a view is usable in answering a graph pattern query and we provide necessary and sufficient conditions for answering a query using materialized views.

View Usability in Graph Pattern Query Answering. Graph pattern queries can be evaluated by computing the match sets of their edges on a data graph G and then joining them on their common query nodes. Let e_q be an edge in a query Q. The match set of e_q is $ms(e_q)$ and its occurrence set is $os(e_q)$ (recall that $os(e_q) \subseteq ms(e_q)$). If there is a materialized view V which has an edge e_v such that $os(e_q) \subseteq os(e_v) \subseteq ms(e_q)$ for every data graph, then V can be used for evaluating Q since $os(e_v)$ can be used instead of $ms(e_q)$ in the join. That is, e_v "covers" e_q. In

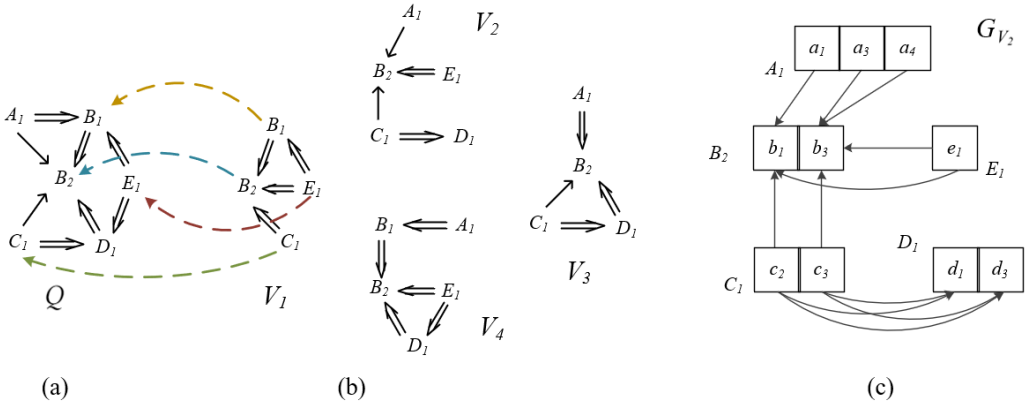


Figure 2: (a) A graph pattern query Q, (b) Views V_1, V_2, V_3, V_4, and a homomorphism from V_1 to Q, (c) A summary graph G_{V_2} of V_2 on data graph G of Figure 1(a).

addition, as $os(e_v)$ is not bigger than $ms(e_q)$, this option is, in general, beneficial in the evaluation of Q. We define view usability in query answering based on this remark. As we will see later, when this happens, other edges of view V might cover an edge in Q as well, in which case, their occurrence sets can also be exploited in evaluating query Q. We now formalize query edge coverage:

Definition 4.1. An edge e_q of a query Q is *covered* by an edge e_v of a view V if $os(e_q) \subseteq os(e_v) \subseteq ms(e_q)$ on any data graph G.

In the example of Figure 2, one can see that the edge (B_1, B_2) of view V_1 covers the edge (B_1, B_2) of query Q_1 since for every mapping m of Q to G, there is a mapping of V_1 to G which is a restriction of m. We can now define view usability.

Definition 4.2. A view V is *usable* in answering a query Q if there is an edge in Q which is covered by an edge in V.

View Usability Conditions. We characterize query edge coverage in terms of homomorphisms from a view to the query. We say that a homomorphism h from a view V to a query Q *maps* an edge $e = (x, y)$ in V to an edge $e = (u, v)$ in Q if $h(x) = u$ and $h(y) = v$.

THEOREM 4.3. *Let e_q be an edge in a graph pattern query Q and e_v be an edge in a view V. Edge e_q in Q is covered by edge e_v in V iff there is a homomorphism from V to Q that maps e_v to e_q such that if e_q is a child edge then e_v is also a child edge.*

The proof can be found in the full version of the paper [2]. In the example of Figure 2, the edge (B_1, B_2) of view V_1 covers the edge (B_1, B_2) of query Q_1. In contrast, (C_1, B_2) in Q is not covered by (C_1, B_2) in V_1 since the former is a child edge and the latter is a descendant edge, and (E_1, B_2) in V_1 does not cover any edge in Q since it cannot be mapped to any edge in Q by a homomorphism from V_1 to Q.

Redundant Query Edges. Two graph pattern queries are *equivalent* if they have the same answer on any data graph. A graph pattern query can have *redundant* edges. An edge in a query Q is redundant if its removal from Q results in a query which is equivalent to Q. A descendant edge $e = (x, y)$ in a query Q is *transitive* if there is a path from x to y in Q other than edge e. Clearly, a transitive edge is redundant. Therefore, transitive edges can be removed from Q without altering the answer of Q.

Answering a Graph Pattern Query Using Multiple Views. In the presence of one or multiple materialized views, it is possible that the answer of query Q can be computed using only the answers of the materialized view(s).

Definition 4.4. Let Q be a query and $\mathcal{V}$ be a set of materialized views which can be used for answering Q. Query Q *can be answered using the views in* $\mathcal{V}$ if, for every data graph, the answer of Q can be computed from a relational algebra expression in $\{\sigma, \pi, \bowtie, \cup\}$ involving exclusively the answers of the views in $\mathcal{V}$.

The following theorem provides necessary and sufficient conditions for answering a query using exclusively a set of materialized views.

THEOREM 4.5. *Let Q be a query and $\mathcal{V}$ be a set of usable views. Query Q* can be answered using the views in $\mathcal{V}$ *if and only if every non-redundant edge in Q is covered by an edge of a view in $\mathcal{V}$.*

The proof can be found in the full version of the paper [2]. In the example of Figure 2, one can see that query Q can be answered using the view set $\mathcal{V} = \{V_1, V_2, V_3, V_4\}$ as all its edges are covered by edges of the views in $\mathcal{V}$.

Given an edge e in a query Q and a view V, the *covering set of e in V*, denoted $cov(e, V)$, is the set of edges in V which cover e. Given a set of views $\mathcal{V}$, the *covering set of e in $\mathcal{V}$*, denoted $cov(e, \mathcal{V})$, is defined as $cov(e, \mathcal{V}) = \bigcup_{V \in \mathcal{V}} cov(e, V)$. Based on Theorem 4.5, Q can be answered using $\mathcal{V}$ if $cov(e, \mathcal{V}) \neq \emptyset$ for every non-redundant edge e of Q.

Minimal Set of Views. A query edge can be covered by multiple view edges of the same and/or different views. However, it is possible that not all of the usable views are needed for answering the query.

Definition 4.6. Let Q be a query and let $\mathcal{V}$ be a set of views such that Q can be answered using the views in $\mathcal{V}$. Set $\mathcal{V}$ is *minimal* if there is no proper subset $\mathcal{V}'$ of $\mathcal{V}$ such that Q can be answered using the views in $\mathcal{V}'$.

Set $\mathcal{V}'$ does not have redundant views. In the example of Figure 2, query Q can be answered using the view set $\{V_1, V_2, V_3\}$ which is minimal. We present in the next section an algorithm which computes a minimal set of views for answering a query.

5 ALGORITHMS

In this Section, we present an algorithm called *SumGraphBuild* which computes a summary graph for a pattern query Q using the

materializations (summary graphs) of the views in a view set $\mathcal{V}$. Algorithm *SumGraphBuild* uses another algorithm, called *FindQCover*, which computes the covering set $cov(e, V)$ of a view V for each query edge e. Therefore, Algorithms *SumGraphBuild* and *FindQCover* can be used to check if a query can be answered using the view set $\mathcal{V}$. Finally, we present an algorithm called *FindMinimalVSet* which finds a minimal set of views for answering a query from a view pool.

Computing the Covering View Edges for a Query Edge. Algorithm *FindQCover*, shown in Algorithm 1, takes as input a query Q and a view V and returns the covering sets of the edges and nodes of Q in V through a function cov on the nodes and edges of V. The covering set of a query node in a view is defined analogously to the covering set of a query edge in a view. Algorithm *FindQCover* first calls procedure *homEnumerate* to enumerate all the homomorphisms from V to Q that satisfy the condition of Theorem 4.3 (line 5). It encodes homomorphisms as n-ary tuples, where n is the number of nodes in V (lines 1,2). The homomorphisms found are stored in set H (line 4). $cov(e)$ denotes the covering set of query edge e in V and $cov(q)$ denotes the covering nodes of query node q in V (line 6). Procedure *homEnumerate* performs a recursive backtracking search to find (candidate) matches in Q for the nodes of V iteratively, one at a time, according to the chosen order (line 1) before returning any generated homomorphism. Finding homomorphisms of graphs to graphs is an NP-hard problem but this is not an issue in this context since the number of nodes and edges of queries and views is restricted. Using set H, Algorithm *FindQCover* calls procedure *findCover* to compute the covering nodes and edges of Q in V.

Algorithm *SumGraphBuild* on the query Q and the view V_1 of Figure 2 will return $cov((B_1, B_2)) = \{(B_1, B_2)\}$, $cov((C_1, B_2)) = \emptyset$, and $cov((D_1, B_2)) = \emptyset$, as there is only one homomorphism from V_1 to Q.

Computing a Query Summary Graph from the Summary Graphs of the Materialized Views. Algorithm *SumGraphBuild*, shown in Algorithm 2, takes as input a query Q and a set of materialized views (summary graphs) $\mathcal{V}$, and produces a summary graph for Q in the form of a function cov on the nodes and edges of Q representing their candidate occurrence sets. The algorithm consists of two phases: the first phase initializes the candidate occurrence sets (cos) of the nodes and edges of Q (line 1) and the second phase builds a summary graph by iteratively refining the candidate occurrence sets generated in the first phase until a fixed point is reached (lines 2-4).

To initialize function cos for the node and edges of Q, *SumGraphBuild* begins by computing the covering sets of the nodes and edges of Q with respect to each view V in $\mathcal{V}$ using algorithm *FindQCover* (Algorithm 1) (lines 3-4 in Procedure initializeCos()). Then, for every node q in Q, the algorithm intersects the occurrence sets $cos(v)$ of the covering nodes $v \in cov(q)$ to obtain the candidate occurrence set $cos(q)$ (lines 5-6). Similarly, for every edge e_q in Q, it intersects the occurrence sets $cos(e_v)$ of the covering edges $e_v \in cov(e_q)$ to obtain the candidate occurrence set $cos(e_q)$ (lines 7-11).

In the second phase, *SumGraphBuild* refines function cos using two procedures, which iterate on the edges of Q in different directions. The first procedure, called *forwardPrune()*, visits each edge $e_q = (q_i, q_j) \in Q$ from the tail node q_i to the head node q_j, and removes node n_{q_i} and its associated outgoing edges from $cos(q_i)$ and $cos(e_q)$, respectively, if there is no $n_{q_j} \in cos(q_j)$ such that

Algorithm 1 Algorithm *FindQCover*.

Input: Graph pattern query Q, and graph pattern view V.
Output: Function cov on the nodes and edges of Q.

1. Pick an order $v_1, \ldots, v_n$ for the nodes of V;
2. Let t be a n-tuple initialized so that $t[i]$ is $null$ for $i \in [1, n]$;
3. Let S_i be the set of nodes of Q having the same label as view node v_i;
4. $H := \emptyset$ /* set H records the homomorphisms from V to Q */
5. homEnumerate(1, t);
6. For every node q in Q and for every edge e in Q, $cov(q) = \emptyset$ and $cov(e) = \emptyset$;
7. findCover();
8. **return** cov;

Procedure homEnumerate(index i, tuple t)

1. **if** (i=n+1) **then**
2. add t to H and return;
3. $N_i := \{v_j \mid (v_i, v_j) \in V \text{ or } (v_j, v_i) \in V, j \in [1, i-1]\}$
4. $S'_i := S_i$;
5. **for** (every $v_j \in N_i$) **do**
6. $S'_i := \{q \in S'_i \mid q \prec t[j] \text{ or } t[j] \prec q\}$;
7. **for** (every $q \in S'_i$) **do**
8. **if** ((v_j, v_i) is a child edge in V and $(t[j], q)$ is not a child edge in Q) or ((v_i, v_j) is a child edge in V and $(q, t[j])$ is not a child edge in Q) **then**
9. Remove q from S'_i;
10. **for** (every node $q \in S'_i$) **do**
11. $t[i] := q$;
12. homEnumerate($i + 1$, t);

Procedure findCover()

1. **for** (every tuple $t \in H$) **do**
2. **for** (every node $v \in V$) **do**
3. add v to $cov(t[v])$;
4. **for** every edge (v_i, v_j) in V **do**
5. **if** $e = (t[v_i], t[v_j])$ is an edge in Q which is a child edge if (v_i, v_j) is a child edge **then**
6. add (v_i, v_j) to $cov(e)$;

(n_{q_i}, n_{q_j}) is an occurrence of e_q in $cos(e_q)$. The second procedure, called *backwardPrune()*, visits each edge $e_q = (q_i, q_j) \in Q$ from the head node q_j to the tail node q_i and removes n_{q_j} and its associated incoming edges from $cos(q_j)$ and $cos(e_q)$, respectively, if there is no $n_{q_i} \in cos(q_i)$ such that (n_{q_i}, n_{q_j}) is an occurrence in $cos(e_q)$. The above process is repeated until function cos becomes stable, i.e., no further removals can be applied to it.

Finally, the refined function cos representing the summary graph of Q is returned to the user (line 5).

Consider the query Q and the views V_1, V_2, V_3 and V_4 in the example of Figure 2. Algorithm *SumGraphBuild* on the answer graph for V_1 of Figure 2(c) and the answer graphs for the views V_2, V_3 and V_4 (not shown in figure) will return the summary graph of Figure 1(d) which is, in fact, the answer graph of Q.

Note that the candidate occurrence sets of the query node and edges can be stored as bitmaps on data graph nodes resulting not only in space savings but also in substantial performance savings as all candidate occurrence set intersection operations can be implemented as bit-wise AND operations.

Finding a Minimal View Set. Algorithm *FindMinimalVSet*, shown in Algorithm 3, takes as input a set of views $\mathcal{V}$ which can be used for answering Q and returns a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ which can be used for answering Q. The algorithm begins with an empty set of views $\mathcal{V}'$. It adds a view to $\mathcal{V}'$ as long as this view covers at least one query edge not covered by the set of views already

Algorithm 2 Algorithm *SumGraphBuild*.

Input: Graph pattern query Q and set $\mathcal{V}$ of materialized views on G which can be used for answering Q.
Output: A summary graph of Q on G (represented by function cos on the nodes and edges of Q).

1. initializeCos();
2. **while** (cos has changes) **do**
3. forwardPrune();
4. backwardPrune();
5. **return** cos;

Procedure initializeCos()

1. For every node $q \in Q$, initialize $cos(q)$ to be $ms(q)$.
2. For every edge $e_q \in Q$, initialize $cos(e_q)$ to be $\emptyset$
3. **for** (every view $V \in \mathcal{V}$) **do**
4. $cov := FindQCover(Q, V)$;
5. **for** (every node $q \in Q$) **do**
6. $cos(q) := cos(q) \cap_{v \in cov(q)} cos(v)$;
7. **for** (every edge $e \in Q$) **do**
8. **if** ($cos(e) = \emptyset$) **then**
9. $cos(e) := \cap_{e_v \in cov(e)} cos(e_v)$;
10. **else**
11. $cos(e) := cos(e) \cap_{e_v \in cov(e)} cos(e_v)$;

Procedure forwardPrune()

1. **for** (each edge $e_q = (q_i, q_j) \in Q$ and each $n_{q_i} \in cos(q_i)$) **do**
2. **if** (there is no $n_{q_j} \in cos(q_j)$ such that (n_{q_i}, n_{q_j}) is an occurrence in $cos(e_q)$) **then**
3. Remove n_{q_i} and its associated outgoing edges from $cos(q_i)$ and $cos(e_q)$, respectively;

Procedure backwardPrune()

1. **for** (each edge $e_q = (q_i, q_j) \in Q$ and each $n_{q_j} \in cos(q_j)$) **do**
2. **if** (there is no $n_{q_i} \in cos(q_i)$ such that (n_{q_i}, n_{q_j}) is an occurrence in $cos(e_q)$) **then**
3. Remove n_{q_j} and its associated incoming edges from $cos(q_j)$ and $cos(e_q)$, respectively;

Algorithm 3 Algorithm *FindMinimalVSet*.

Input: Graph pattern query Q and a set $\mathcal{V}$ of views which can be used for answering Q.
Output: A minimal set $\mathcal{V}' \subseteq \mathcal{V}$ of views which can be used for answering Q.

1. $\mathcal{V}' := \emptyset$;
2. findViews();
3. removeRedundant();
4. **return** $\mathcal{V}'$;

Procedure findViews()

1. $U := edges(Q)$; /* the set of uncovered edges of Q */
2. **while** ($U \neq \emptyset$) **do**
3. Select an edge e in U;
4. Find a view V in $\mathcal{V}$ which has an edge covering e;
5. Let C be the set of edges in Q which are covered by V;
6. $\mathcal{V}' := \mathcal{V}' \cup \{V\}$;
7. $U := U - C$;

Procedure removeRedundant()

1. **for** (every view $V \in \mathcal{V}'$) **do**
2. **if** (Q can be answered using exclusively $\mathcal{V}' - \{V\}$) **then**
3. Remove V from $\mathcal{V}'$;

selected in $\mathcal{V}'$. After all the query edges are covered, the algorithm eliminates redundant views by checking if the removal of that view would cause a query edge to be uncovered by the set of views in $\mathcal{V}'$.

In the example of Figure 2, Algorithm 3 will initially add to $\mathcal{V}'$ all the views V_1, V_2, V_3 and V_4 if the views are considered in the order V_1, V_2, V_3, V_4. It will subsequently identify the view V_3 as redundant and it will remove it from $\mathcal{V}'$ to return the minimal view set $\{V_1, V_2, V_4\}$.

As our experiments show, considering additional views for answering a query Q beyond a set of views that cover all the edges of Q does not significantly reduce the query evaluation cost. Thus, a minimal set of views from the materialized view pool constitutes a reasonable choice for answering a query.

6 EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our materialized view approach in terms of time performance and scalability.

6.1 Experimental Setting

Algorithms in comparison. We implemented our approach for answering queries using materialized views. In our implementation of Algorithm *SumGraphBuild*, we used bitmaps to represent query and view node occurrence sets and adjacency lists and bit-wise AND operations for intersecting sets. We refer to this approach in this section as *MatView*.

We compare *MatView* with the approach presented in [37] for evaluating hybrid graph pattern queries using homomorphisms over a large graph. This approach employs an algorithm called *FltSim* to construct a summary graph for the input query on a data graph. Therefore, it can be directly compared with *MatView*, which also constructs a summary graph for the input query. Algorithm *FltSim* first applies a filtering technique to prune nodes and edges from the data graph that do not participate in the query answer, and uses the pruned data graph to construct an initial summary graph. It then refines this summary graph using double simulation to exclude nodes and edges that are unlikely to be part of the query answer before returning it to the user.

The main difference between *FltSim* and *SumGraphBuild* lies in the summary graph edge construction: *FltSim* needs to access a reachability index on G in order determine the existence of reachability relationships between nodes in the candidate occurrence sets and connect them by edges. In contrast, *SumGraphBuild* obtains edges for the candidate occurrence sets of the query edges from the candidate occurrence sets of the covering view edges. This is much cheaper than accessing a reachability index and gives the upper hand to *MatView* which benefits from the materialized views. We refer to the base approach that does not use materialized views as *FltSim*.

We do not compare *MatView* with other approaches as *FltSim* is shown in [37] to outperform previous state-of-the art approaches [7, 24, 39, 42] for this type of query patterns on data graphs.

Datasets. We ran experiments on two real-world graph datasets which have been used in previous works [25, 31]. The datasets have different structural properties and come from different application domains, such as the web and social networks. Table 1 lists the properties of the datasets. Its last column displays the average number of incident edges (both incoming and outgoing) per node.

For our scalability experiments we vary the number of nodes and edges of the data graphs and their number of distinct labels.

Table 1: Key statistics of the graph datasets used.

Domain	Dataset	# of nodes	# of edges	Avg #incident edges
Web	BerkStan (bs)	685K	402K	11.76
Social	DBLP (db)	317K	1049K	6.62

Queries. We generated 10 graph pattern query templates, shown in Figure 3. These hybrid query templates involve child and descendant edges. They have various and complex structures and many of them were used in previous work [7, 24]. The number associated with each node of a query template denotes the node id. Query instances are generated by assigning labels to nodes.

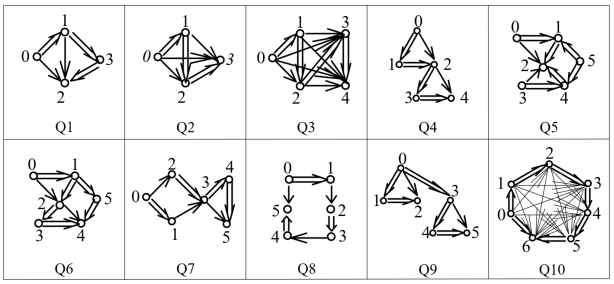


Figure 3: Graph pattern query templates used in the evaluation.

Views. For every run using *MatView*, a query Q was run with a set of views $\mathcal{V}$. Each view was randomly generated from the query graph.

A query edge e_q can be covered by more than one view edge. Algorithm *SumGraphBuild* initially intersects the candidate occurrence sets of its covering view edges in order to compute the candidate occurrence set of a query edge. The more covering edges on e_q are intersected, the smaller their resulting candidate occurrence set would be when this is computed by algorithm *SumGraphBuild*.

The average number of covering edges for a query edge e_q in Q in a view set $\mathcal{V}$ is:

$$cov_{avg}(Q, \mathcal{V}) = \sum_{V \in \mathcal{V}} cov(e_q, V)/|E(Q)|$$

When $cov_{avg}(Q, \mathcal{V}) = 1$, each query edge is covered by exactly one view edge. We expect that the higher $cov_{avg}(Q, \mathcal{V})$ is, the smaller the summary graph G_Q will be. The lowest value for $cov_{avg}(Q, \mathcal{V})$ is produced by a minimal set $\mathcal{V}$.

For each query computation, we used a set of views $\mathcal{V}$ with the same number of edges. With the exception of the experiment where $cov_{avg}(Q, \mathcal{V})$ is varied, $cov_{avg}(Q, \mathcal{V})$ is manitained within a fixed range: $1 \leq cov_{avg}(Q, \mathcal{V}) \leq 2$. For the experiments in sections 6.2, 6.3, and 6.6, all sets of views $\mathcal{V}$ used by *MatView* contained views with mixed edges and exactly two edges, and were chosen to be minimal using Algorithm 3.

Metrics. We measured the evaluation time of the queries in a query set in seconds (sec). In the case of *FltSim*, this includes the preprocessing time (i.e., the time spent on filtering data graph nodes and edges). Given that the number of query results can be very large, we terminated the evaluation of a query after finding 10^7 matches.

Our implementation was coded in Java. All the experiments reported were performed on a 64-bit Linux machine equipped with an Intel Xeon 6240 @ 2.60 Hz processor and 768GB RAM.

6.2 Benefit of Using Materialized Views

Figure 4 displays the elapsed time of *FltSim* versus *MatView* for all the queries of Figure 3 on a bs data graph with 350K nodes and 5 labels. The scale of the y-axis is logarithmic. We observe that for all queries, *MatView* is several orders of magnitude better than *FltSim*; in most cases, *MatView* is approximately three orders of magnitude better than *FltSim*.

Figure 5 displays the elapsed time of *FltSim* versus *MatView* for all the queries of Figure 3 on a dblp data graph with 250K nodes and 20 labels. The scale of the y-axis is logarithmic. As in the bs data graph, for all queries, *MatView* is several orders of magnitude better than *FltSim*. In the case of Q_{10}, which has many descendant edges, *MatView* is five order of magnitude better than *FltSim*.

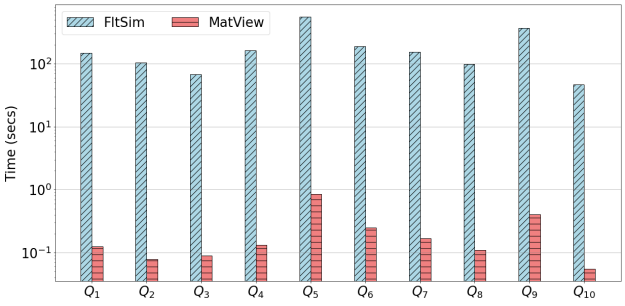


Figure 4: Elapsed time of *FltSim* and *MatView* for various queries on a bs data graph with 350K nodes and 5 labels.

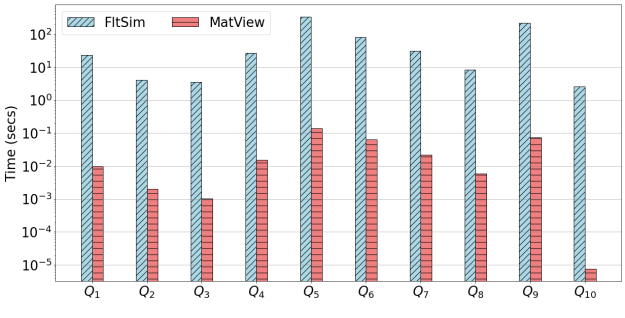
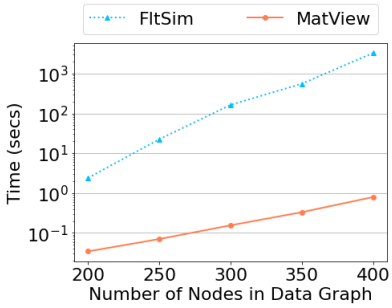


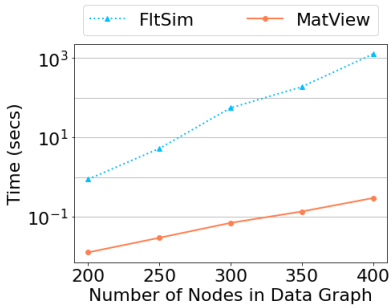
Figure 5: Elapsed time of *FltSim* and *MatView* for various queries on dblp data graph with 250K nodes and 20 labels.

6.3 Data Graph Size Scalability

In this experiment, we evaluated the performance of the two algorithms as the data set size grows. We ran queries on increasingly larger randomly chosen subsets of a data graph, such that each increasingly larger subset is a superset of the previous subset, and recorded the elapsed time. Figure 6 shows the results, on a logarithmic scale for y-axis, for queries Q_5 and Q_6 on the bs data graph with 5 labels. Figure 7 shows the results, on a logarithmic scale for the y-axis, for queries Q_7 and Q_9 on the dblp data graph with 20 labels.

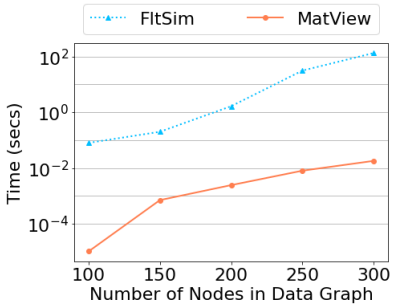


(a) Q_5

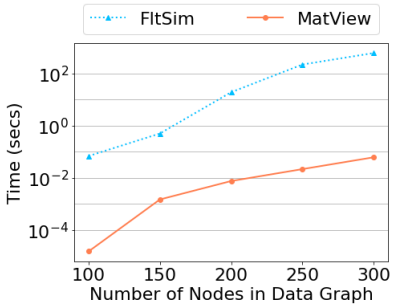


(b) Q_6

Figure 6: Elapsed time of *FltSim* and *MatView* on increasingly larger number of data subsets of the bs data subset with 5 labels.



(a) Q_7



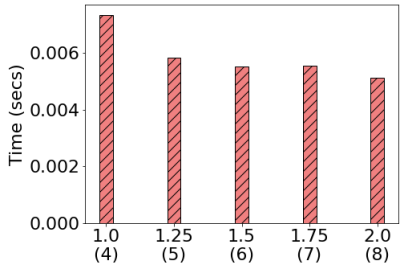
(b) Q_9

Figure 7: Elapsed time of *FltSim* and *MatView* on increasingly larger number of data subsets of the dblp data subset with 20 labels.

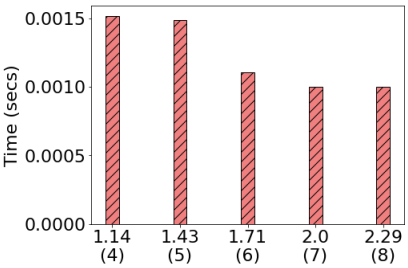
In all cases, the execution time for all algorithms increased when the total number of graph nodes increased. *MatView* provided significantly better performance than *FltSim* for evaluating the two queries. In addition, the slope of $FltSim$ is much steeper than that of $MatView$.

We also observed that in Figure 7, for the data point with 100K nodes, the evaluation time for *MatView* was very small. This is because, in contrast to the other data points, there were no matches for query Q_7; while *FltSim* had to spend time to filter out irrelevant nodes and edges from the data graph before it discovered that the query has an empty answer, *MatView* was able to quickly discover that this query has empty answer.

6.4 Varying the Number of Covering View Edges



(a) Q_5



(b) Q_7

Figure 8: Elapsed time of *MatView* on queries run with varying $cov_{avg}(Q,\mathcal{V})$ (top label in x-axis) and a different number of covering views (bottom label in x-axis) on a bs data set with 20 labels and 350K nodes.

We ran experiments comparing the performance of *FltSim* and *MatView* varying $cov_{avg}(Q,\mathcal{V})$ (using a different number of views). All views had mixed edges and exactly two edges. We started by evaluating a query using a minimal set of views, where each query edge is covered by only one covering edge, and gradually added one view at a time. Each time a new view was added, $cov_{avg}(Q,\mathcal{V})$ increased slightly. In Figure 8 we plotted the value of $cov_{avg}(Q,\mathcal{V})$ for each new set of views V on the top row label of the X-axis, and plotted the number of views in $\mathcal{V}$ on the bottom row label of the X-axis.

The results for two of these queries, Q_5 and Q_7, that are run on the bs data graph with 20 labels and 350K nodes are shown in Figure 8. We observed that for sets of views with a higher $cov_{avg}(Q,\mathcal{V})$, the summary graphs obtained were only smaller, but the differences did not have much impact on the evaluation times. Thus, selecting a minimal view set for evaluating query Q is a viable solution.

6.5 Varying the Number of Edges per View

We compared the performance of *MatView* and *FltSim* using views with two edges versus views with three edges. Both the set of views with two edges and the set of views with three edges met the condition where $1 < cov_{avg}(Q, \mathcal{V}) < 2$; this was achieved by varying the number of views within $\mathcal{V}$ such that, for each query Q, the set with three-edge views contained less views than the set with two-edge views.

The results for all 10 queries evaluated on the bs data graph with 20 labels and 350K nodes are shown in Figure 9. Overall, for nine out of ten queries, we found that using views with three edges obtained better evaluation times than using views with two edges, while for one of the queries (Q_3), they obtained approximately the same evaluation time.

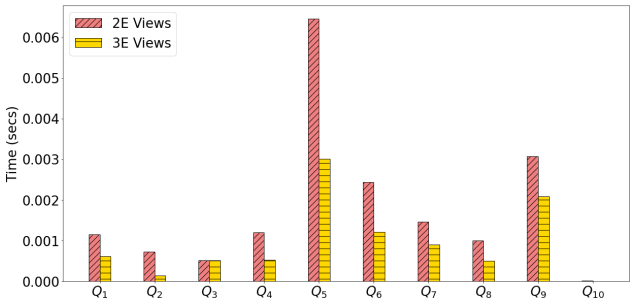


Figure 9: Elapsed time of *MatView* using *2-edge views* and *3-edge views* for various queries on a bs data subset with 20 labels and 350K nodes.

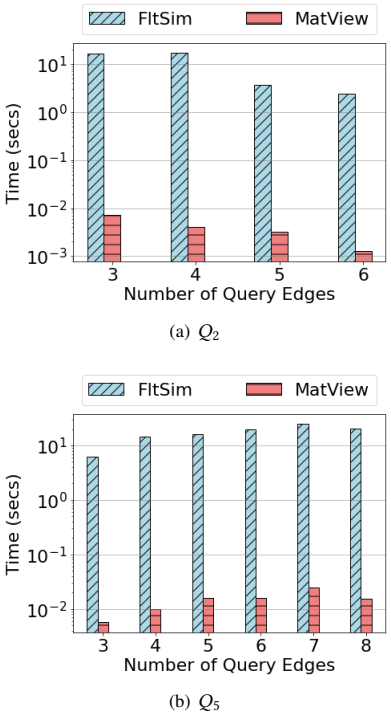


Figure 10: Elapsed time of *FltSim* and *MatView* on queries with a different edges on a bs data subset with 20 labels and 350K nodes.

6.6 Varying the Number of Query Edges

We measured the execution time of the two appoaches varying the number of edges in the queries. To obtain these queries, we started with the original query, then removed one edge at a time.

The results for two of these queries, Q_2 and Q_5, on the bs data graph with 20 labels and 350K nodes using a logarithmic scale are shown in Figure 10. We can see that the execution time does not follow a specific pattern as adding on more edge to a query can increase or decrease the number of query results.

6.7 Summary

The experiments reported here have examined the performance of pattern query evaluation algorithms on graphs. The results can be summarized as follows:

- The performance of a graph pattern matching algorithm is affected significantly by costly computations using the reachability index.
- The view materialization approach *MatView* significantly reduces evaluation times by using the view materializations instead of accessing the reachability index.
- *MatView* shows the best efficiency and scalability performance between the two algorithms, while displaying a negligible occurrence set intersection time cost. This demonstrates the effectiveness of the view materialization approach.

7 RELATED WORK

Answering queries using views has been extensively studied for relational data (see [16] for a survey) and tree data [5, 32, 35, 40, 41]. Due to the importance of graph pattern matching in many application domains and the need to improve pattern matching time on large graph data, there have recently been quite a few contributions [11, 14, 21, 34, 36, 38] addressing the problem of answering graph pattern queries using views.

Fan et al. [14] investigate this problem for graph pattern queries based on graph simulation and study its complexity. Under this setting, they characterize graph pattern matching using graph pattern views based on pattern containment, and provide algorithms for answering graph pattern queries using a set of materialized views. This work was extended to address answering graph queries using views in terms of subgraph isomorphism [36]. Another extension [21] studies the approximation of graph pattern queries using views based on both graph simulation and subgraph isomorphism.

More recently, Trindade et al. [11] presented a graph query optimization framework called Kaskade which materializes graph views to enable efficient query evaluation. Kaskade considers two types of views: path views which match to a path of data nodes with bounded length, and relational counterparts which are filters and aggregates. Kaskade only supports query rewriting/answering using a single view. Unlike previous work, it focuses on leveraging structural properties of graphs and queries to enumerate views and to select the best views to materialize based on a budget constraint.

To speed up graph query processing, Wang et al. [34] proposed to acquire and utilize knowledge from the results of previously executed queries, which are essentially materialized views. Views considered for answering a new query are subgraphs or supergraphs of the query. Unlike previous approaches this approach considers the framework of a collection of small data graphs and aims at minimizing the number of isomorphism tests that need to be performed to find the data graphs that contain the query pattern.

Wu et al. [38] studied the problem of using materialized views for homomorphic pattern matching on data graphs, but considered only tree-pattern queries. Le et al. [19] studied the problems of rewritting SPARQL queries using views, but did not consider materializing these views.

The problem we address in this paper is different than those addressed by existing graph view approaches. We consider general graph patterns and not simply paths or trees. Our patterns contain child and descendant edges, allowing for both edge-to-edge and edge-to-path matches to the data graph. Patterns are mapped to the data graph using homomorphisms which relax the strict one-to-one mapping entailed by isomorphisms and, unlike graph simulation, preserve the topology of the data graph. We adopt the concept of a summary graph to encode all possible homomorphisms from a query pattern to the data graph, and materialize views as summary graphs. By generating a summary graph for a query pattern using the summary graphs of multiple materialized views, our approach greatly reduces the time to find the homomorphic matches of the query.

8 CONCLUSION

We have addressed the problem of answering graph pattern queries using graph pattern materialized views to efficiently evaluate such queries on large data graphs under homomorphisms. We considered a broad class of pattern queries that involve both node reachability and direct relationships. We suggested an original approach which materializes views as summary graphs, therein compactly representing the homomorphic matches of the views. In this context, we characterized the view usability problem in terms of query edge coverage, and provided necessary and sufficient conditions for answering graph pattern queries using views. We designed algorithms for deciding whether a query can be answered from materialized views, for computing query summary graphs from the summary graphs of the views, and for producing minimal sets of views for answering a query. Our experimental results showed that our approach outperforms, by several orders of magnitude, approaches that do not use materialized views, and provides much better scalability.

We are currently working on scale-independently answering queries using views based on the framework set in this paper.

REFERENCES

[1] DBpedia. https://wiki.dbpedia.org/.

[2] Full version of the paper. https://drive.google.com/drive/folders/1MwxsgrrKGM_t4IzWHdtgfYJzwlauFs6h?usp=sharing.

[3] Network Repository. http://networkrepository.com/.

[4] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, pages 431–446, 2016.

[5] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured materialized views for XML queries. In *VLDB*, 2007.

[6] B. Bhattarai, H. Liu, and H. H. Huang. CECI: compact embedding cluster index for scalable subgraph matching. In *SIGMOD*, pages 1447–1462, 2019.

[7] J. Cheng, J. X. Yu, and P. S. Yu. Graph pattern matching: A join/semijoin approach. *IEEE Trans. Knowl. Data Eng.*, 23(7):1006–1021, 2011.

[8] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[10] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[11] J. M. F. da Trindade, K. Karanasos, C. Curino, S. Madden, and J. Shun. Kaskade: Graph views for efficient graph analytics. In *ICDE*, pages 193–204, 2020.

[12] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. *PVLDB*, 3(1):264–275, 2010.

[13] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3(1):1161–1172, 2010.

[14] W. Fan, X. Wang, and Y. Wu. Answering pattern queries using views. volume 28, pages 326–341, 2016.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[16] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[17] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, 1995.

[18] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.

[19] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Rewriting queries on SPARQL views. In *Proc. of the Intl. Conf. on World Wide Web*, pages 655–664, 2011.

[20] M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 233–246. ACM, 2002.

[21] J. Li, Y. Cao, and X. Liu. Approximating graph pattern queries using views. In *CIKM*, pages 449–458, 2016.

[22] R. Liang, H. Zhuge, X. Jiang, Q. Zeng, and X. He. Scaling hop-based reachability indexing for fast graph pattern query processing. *IEEE Trans. Knowl. Data Eng.*, 26(11):2803–2817, 2014.

[23] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Strong simulation: Capturing topology in graph pattern matching. *ACM Trans. Database Syst.*, 39(1):4:1–4:46, 2014.

[24] A. Mhedhbi, C. Kankanamge, and S. Salihoglu. Optimizing one-time and continuous subgraph queries using worst-case optimal joins. *ACM Trans. Database Syst.*, 46(2):6:1–6:45, 2021.

[25] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[26] D. Olteanu and M. Schleich. Factorized databases. *SIGMOD Record*, 45(2):5–16, 2016.

[27] N. Przulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. *Bioinform.*, 22(8):974–980, 2006.

[28] A. M. Smalter, J. Huan, Y. Jia, and G. H. Lushington. GPD: A graph pattern diffusion kernel for accurate graph classification with applications in cheminformatics. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 7(2):197–207, 2010.

[29] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: Can it be even faster? *IEEE Trans. Knowl. Data Eng.*, 29(3):683–697, 2017.

[30] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[31] S. Sun and Q. Luo. In-memory subgraph matching: An in-depth study. In *SIGMOD*, pages 1083–1098, 2020.

[32] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple materialized view selection for XPath query rewriting. In *ICDE*, 2008.

[33] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[34] J. Wang, N. Ntarmos, and P. Triantafillou. Indexing query graphs to speedup graph query processing. In *EDBT*, pages 41–52, 2016.

[35] J. Wang and J. X. Yu. XPath rewriting using multiple views. In *DEXA*, 2008.

[36] X. Wang. Answering graph pattern matching using views: A revisit. In *DEXA*, pages 65–80, 2017.

[37] X. Wu and D. Theodoratos. Evaluating hybrid graph pattern queries using runtime index graphs. https://arxiv.org/abs/2112.08638.

[38] X. Wu, D. Theodoratos, D. Skoutas, and M. Lan. Evaluating mixed patterns on large data graphs using bitmap views. In *DASFAA*, pages 553–570, 2019.

[39] X. Wu, D. Theodoratos, D. Skoutas, and M. Lan. Efficient in-memory evaluation of reachability graph pattern queries on data graphs. In *DASFAA*, 2022.

[40] X. Wu, D. Theodoratos, and W. H. Wang. Answering XML queries using materialized views revisited. In *CIKM*, 2009.

[41] X. Wu, D. Theodoratos, W. H. Wang, and T. Sellis. Optimizing XML queries: Bitmapped materialized views vs. indexes. *Inf. Syst.*, 38(6):863–884, 2013.

[42] Q. Zeng, X. Jiang, and H. Zhuge. Adding logical operators to tree pattern queries on graph-structured data. *PVLDB*, 5(8):728–739, 2012.