Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

USING MATERIALIZED VIEWS FOR ANSWERING GRAPH PATTERN QUERIES

by
Michael Lan

Discovering patterns in graphs by evaluating graph pattern queries involving direct (edge-to-edge mapping) and reachability (edge-to-path mapping) relationships under homomorphisms on data graphs has been extensively studied. Previous studies have aimed to reduce the evaluation time of graph pattern queries due to the potentially numerous matches on large data graphs.

In this work, the concept of the summary graph is developed to improve the evaluation of tree pattern queries and graph pattern queries. The summary graph first filters out candidate matches which violate certain reachability constraints, and then finds local matches of query edges. This reduces redundancy in the representation of the query results and allows for computation sharing during the generation of these results. Methods using materialized graph pattern views are developed to improve the efficiency of graph pattern query evaluation. A view is materialized as a summary graph, which compactly records all the homomorphisms of the view to the data graph. View usability is characterized in terms of query edge coverage to provide necessary and sufficient conditions for answering queries using views, and algorithms are developed for determining view usability and for summary graph construction.

Experimental evaluation shows that the methods using summary graphs and its related concepts outperform previous state-of-the-art approaches. It also demonstrates that the view materialization method outperforms, by several orders of magnitude, a state-of-the-art approach which does not use materialized views, and substantially improves upon its scalability.

USING MATERIALIZED VIEWS FOR ANSWERING GRAPH PATTERN QUERIES

by
Michael Lan

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

December 2022

Copyright © 2022 by Michael Lan

ALL RIGHTS RESERVED

APPROVAL PAGE

USING MATERIALIZED VIEWS FOR ANSWERING GRAPH PATTERN QUERIES

Michael Lan

Dr. Dimitri Theodoratos, Dissertation Advisor — Date
Associate Professor, Department of Computer Science, NJIT

Dr. James Geller, Committee Member — Date
Professor, Department of Computer Science, NJIT

Dr. Vincent Oria, Committee Member — Date
Professor, Department of Computer Science, NJIT

Dr. Usman Roshan, Committee Member — Date
Associate Professor, Department of Computer Science, NJIT

Dr. Yi Chen, Committee Member — Date
Professor, Martin Tuchman School of Management, NJIT

BIOGRAPHICAL SKETCH

Author:	Michael Lan
Degree:	Doctor of Philosophy
Date:	December 2022

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 2022
- Master of Science in Statistics,
  Rutgers University, New Brunswick, NJ, 2017
- Bachelors in Biomathematics,
  Rutgers University, New Brunswick, NJ, 2015

Major: Computer Science

Presentations and Publications:

M. Lan, X. Wu, and D. Theodoratos, "Optimizing graph pattern queries using materialized views," *Journal Paper In Preparation*, 2022.

X. Wu, D. Theodoratos, N. Mamoulis, and M. Lan, "Evaluating hybrid graph pattern queries using runtime index graphs," *Conference Paper Under Review*, 2022.

X. Wu, D. Theodoratos, D. Skoutas, and M. Lan, "Efficient in-memory evaluation of reachability graph pattern queries on data graphs," *International Conference on Database Systems for Advanced Applications*, pp. 55–71, 2022.

M. Lan, X. Wu, and D. Theodoratos, "Answering graph pattern queries using compact materialized views," *International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data*, pp. 51–60, 2022.

X. Wu, D. Theodoratos, D. Skoutas, and M. Lan, "Exploring citation networks with hybrid tree pattern queries," *International Workshop on Assessing Impact and Merit in Science*, pp. 311–322, 2020.

X. Wu, D. Theodoratos, D. Skoutas, and M. Lan, "Leveraging double simulation to efficiently evaluate hybrid patterns on data graphs," *International Conference on Web Information Systems Engineering*, pp. 255–269, 2020.

X. Wu, D. Theodoratos, D. Skoutas, and M. Lan, “Evaluating mixed patterns on large data graphs using bitmap views,” *International Conference on Database Systems for Advanced Applications*, pp. 553–570, 2019.

X. Wu, D. Theodoratos, D. Skoutas, and M. Lan, “Efficiently computing homomorphic matches of hybrid pattern queries on large graphs,” *International Conference on Data Warehousing and Knowledge Discovery*, pp. 279–295, 2019.

As Above, So Below.

The Monad

ACKNOWLEDGMENT

I would like to thank my advisor Dr. Dimitri Theodoratos for his continuous guidance and support for this research.

I would like to thank Dr. James Geller, Dr. Vincent Oria, Dr. Usman Roshan, and Dr. Yi Chen for taking time to serve in my committee.

I am appreciative of the Department of Computer Science at NJIT for years of funding support, in particular to Dr. Baruch Schieber for additional funding support in terms of a stipend.

Finally, I am thankful to Dr. Xiaoying Wu for providing technical support and guidance for this research, and to collaborator Dr. Dimitrios Skoutas.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

Chapter **Page**

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Motivation

Graphs model complex relationships between entities in a multitude of modern applications. A fundamental operation for querying, exploring and analyzing graphs is graph pattern matching, which consists of finding the matches of a query graph pattern in the data graph. Graph pattern matching is crucial in many application domains, such as social network analysis [1, 2], protein interaction analysis [3], and cheminformatics [4].

Existing approaches are characterized by: (a) the type of edges the query patterns have, and (b) the type of morphism used to map the query pattern to the data graph. An edge in a query pattern can be either a direct edge, which represents a parent-child relationship in the data graph (edge-to-edge mapping) [5, 6, 7, 8, 9, 10, 11, 12, 13], or a reachability edge, which represents a node reachability relationship in the data graph (edge-to-path mapping) [14, 15, 16]. The morphism determines how a pattern is mapped to the data graph and, in this context, it can be an isomorphism (an injective mapping) or a homomorphism (a mapping more general than an isomorphism). Earlier contributions considered isomorphisms and edge-to-edge mappings, while more recent ones also focus on homomorphic mappings.

By allowing edge-to-path mapping on graphs, patterns with reachability edges are able to extract matches "hidden" deeply within large graphs which might be missed by patterns with only direct edges. On the other hand, the patterns with direct edges can discover important parent-child relationships in the data graph which can be missed by patterns with only reachability edges. In our work, we employ a general framework that considers patterns which allow both direct and reachability edges. This framework incorporates the benefits from both types of edges.

To demonstrate an example of the applicability of using reachability edges in graph pattern queries, consider a large financial graph database which consists of customers, companies, money transfers, relationships between individuals, and more. There are different ways to characterize companies, such as by their trustworthiness. A shell company is a company that may serve as a vehicle for business transactions without itself having any significant assets or operations. Money laundering often uses multiple transfers between shell companies and individuals to hide the origin of their funds. Figure 1.1 shows a pattern query which can be used for fraud detection over this financial graph database. The single-lined arrows in the query pattern (direct edges) represent edges in the data graph, while the double-lined arrows in the query pattern (reachability edges) represent paths in the data graph.

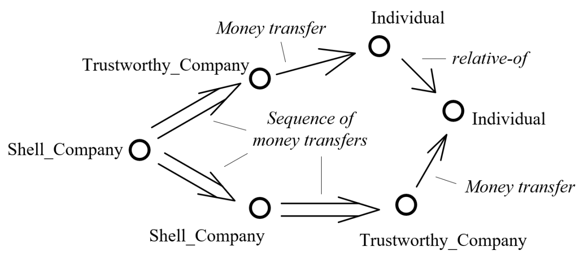


Figure 1.1 A query pattern for fraud detection over a financial graph database.

Graph pattern matching is an NP-hard problem, even for isomorphic matching of patterns with only direct edges [17]. Finding the homomorphic matches of query patterns which involve reachability edges on a data graph is more challenging. Reachability 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 direct edges, and finding matches of reachability edges to the data graph is an expensive operation which requires the use of a node readability index [18, 19, 20]. Furthermore, most of the algorithms that allow edge-to-path mapping for graphs [21, 22, 15, 23, 16] are tied to a specific reachability index, thus being unable to utilize recent, more efficient, reachability indexing schemes like [20, 24, 25]. Despite the use of reachability indexes, evaluating reachability 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. Thus, the objective of this dissertation is to improve upon existing methods for evaluating pattern queries with reachability relationships.

1.2 Overview of Research

In this dissertation, we develop the following approaches to improve upon the field of graph pattern matching:

- We aim to speed up the computation of *homomorphic* matches of *hybrid* tree patterns on large graphs. This task lies at the core of graph pattern matching techniques since a common approach followed by many graph pattern matching methods [26, 27, 28, 29, 30, 31, 7, 13] is to first decompose or transform the given graph pattern into one or more tree patterns using various methods, and then use them as the basis for processing.

- We address the problem of evaluating graph pattern queries with reachability edges (edge-to-path mapping) using homomorphisms over a data graph. This is a general setting for graph pattern matching. We develop a new graph pattern matching framework, which consists of two phases: (a) the *summarization phase*, where a query dependent summary graph is built on-the-fly, serving as a compact search space for the given query, and (b) the *enumeration phase*, where query solutions are produced using the summary graph.

- We adopt a novel approach for materializing graph pattern views over data graphs and for evaluating graph pattern queries that have all their edges covered by views. A view materialization is a graph, in particular the view's *summary graph*, which is a compact representations of the view's homomorphic match results. A summary graph constitutes a search space for the view's homomorphic match results, and the view results can be enumerated by applying multiway joins while traversing the graph.

- We extend our materialized views approach to cases of graph pattern queries partially covered using views. We define *node coverage* and *weakly usable views* to consider views which may help speed up graph pattern query evaluation time, but do not have to cover graph pattern query edges. These views can be used for optimizing the evaluation of graph pattern queries.

Originality and Significance. This dissertation introduces novel approaches to improve the performance times of graph pattern matching. These approaches involve compactly summarizing enumeration results as a summary graph representation, and materializing graph pattern views over data graphs, both which can significantly speed up the computation of query matches to the data graph.

1.3 Organization of the Dissertation

We will begin by conducting a literature review, introducing important terminology and theory. Here, we will discuss key concepts and issues in graph pattern matching that are necessary to understand our modeling, along with current approaches for addressing those issues. Next, we present our work by starting with our novel approach to using compact summary graphs for tree pattern matching and graph pattern matching. We build upon previous results to design algorithms for answering graph pattern queries using exclusively materialized views. These results are extended to provide algorithms for optimizing graph pattern queries using materialized views. In each chapter, we present an experimental evaluation and we analyze the results of our methods. Finally, we will conclude and discuss future work.

CHAPTER 2

LITERATURE REVIEW

We review related work on graph pattern query evaluation algorithms. Our discussion focuses on in-memory algorithms that find all occurrences of a graph pattern in a single large data graph. We categorize the related work by the type of morphism used to map the patterns to the data structure. The morphism determines how a pattern is mapped to the data graph: a homomorphism is a function from pattern nodes to data graph nodes, while an isomorphism is a one-to-one function from pattern nodes to data graph nodes.

2.1 Isomorphic Mapping Algorithms

A recent survey [9] studied the performance of representative in-memory isomorphic mapping algorithms [32, 33, 34, 35, 30, 7, 31]. It compared their methods for node filtering, matching order, result enumeration and algorithm optimization. Several recent algorithms [29, 30, 31, 7] first generated a breadth-first-search tree of the query, and used the tree as the processing unit for building an auxiliary data structure (ADS) to maintain edge sets between candidate nodes. Then, they used this ADS to generate a good matching order. Finally, they enumerated query results with the assistance of the ADS along with the matching order. The majority of algorithms for isomorphic mapping adopted a backtracking method to enumerate the query answer [5]. This method recursively extended partial matches by mapping query nodes to data graph nodes. Many optimization techniques designed for isomorphic mapping algorithms do not apply to homomorphisms since they focused on reducing the search space for the case of injective functions and edge-to-edge mapping.

2.2 Homomorphic Mapping Algorithms

Homomorphisms for mapping graph patterns similar to those considered in this paper were introduced in [14] (called p-hom), which did not address the problem of efficiently computing graph pattern matches; instead, the paper used the notion of p-hom to resolve a graph similarity problem between two graphs.

Existing homomorphic graph pattern matching algorithms mainly employed the *edge-join* approach (*EJ*). Given a graph pattern query Q, *EJ* first computes binary relations corresponding to the edges of the query by matching the edges against the data graph. The query is then evaluated by joining together these individual matches. Homomorphic mapping algorithms such as R-Join [15] and database management systems such as PostgreSQL, MonetDB and Neo4j use the edge-join approach. The more recent approaches including EmptyHeaded [13] and Graphflow [8, 11] roughly fall in this category.

The algorithm R-Join was proposed by Cheng et al. [15]. An important challenge for join-based algorithms is finding a good join order. To optimize the join order, R-Join used dynamic programming to exhaustively enumerate left-deep tree query plans. Due to the large number of potential query plans, R-Join was efficient only for small queries that had less than 10 nodes. As was typical with join-based algorithms (algorithms that join smaller matches together to compute matches), R-Join suffered from the problem of computing a large number of intermediate results. As a consequence, its performance degraded rapidly when the graph became larger [16]. R-Join was adopted as the underlying pattern matching method in D-join [36] for evaluating graph patterns whose edges carried the same connectivity constraint (a constraint that bounds the number of nodes in the image's data paths).

The EmptyHeaded system [13] decomposed the input query into a tree of subqueries, computed each subquery using a multiway join algorithm, and combined subquery occurrences using Yannakakis' algorithm [37]. Graphflow [8, 11] was a more recent join-based homomorphic mapping algorithm. Like EmptyHeaded, Graphflow pruned relations based on labels. But unlike EmptyHeaded, which picked a join order using a simple heuristic, Graphflow designed a cost model to pick an optimal join plan. Both EmptyHeaded and Graphflow considered only *edge-to-edge* homomorphic mappings.

Another well known approach for evaluating tree-pattern queries over data graphs is the *path-join* approach (*PJ*) [22, 26, 29, 30]. Unlike *EJ*, *PJ* first generates occurrences to each root-to-leaf path of the tree pattern, and then generates the final answer by merge-joining the path occurrences. A typical representative of a *PJ* approach is TwigStackD [38, 22], a tree-based pattern matching algorithm on directed acyclic graphs (dags). TwigStackD used stacks for identifying candidate nodes and stored (partial) solutions in array-like data structures. Specifically, it traversed the tree pattern Q_T in a top-down manner, and put a data node v to the stack of the corresponding query node q if v was in a solution of the subtree rooted at q in Q_T. It generateed the final answer by merge-joining the path occurrences of the root-to leaf paths of Q_T. Like edge-join algorithms, TwigStackD may produce a large number of unnecessary intermediate results. Similar to TwigStackD, the evaluation process of the isomorphic graph matching algorithms [29, 30] also involved enumerating and merge-joining occurrences of query paths of a spanning tree of the input graph query.

Zeng et al. [23, 16] proposed a proprietary graph matching method called TPQ-3Hop. The matching process was tightly coupled with a specific reachability indexing scheme to support reachability edge-only tree patterns. However, this approach could not be effectively applied to hybrid pattern query evaluation.

A graph pattern matching algorithm called DagStackD was developed in [22]. DagStackD implemented a tree-based approach. Given a graph pattern query Q, DagStackD first found a spanning tree Q_T of Q, then evaluated Q_T, filtering out tuples that violated the reachability relationships specified by the edges of Q missing in Q_T. To evaluate Q_T, a tree pattern evaluation algorithm was presented. This algorithm decomposed the tree query into a set of root-to-leaf paths, evaluated each query path, and merge-joined their results to generate the tree-pattern query answer.

2.3 Graph Simulation-Based Algorithms

Simulation-based pruning was proposed recently as a powerful node pruning technique in graph matching [39, 40]. It used subgraph simulation or some variants to filter unnecessary tuples before answering queries. A number of approaches used graph simulation-based semantics to match the graph pattern queries against the data graph [41, 42, 43]. Unlike morphism-based graph pattern matching, which is NP-complete, graph simulation-based graph pattern matching algorithms could be performed in cubic-time. However, simulation and its extensions [41, 42] could not preserve the structural properties of the graph pattern. One consequence of this was that they they may have returned an excessive number of undesirable matches. To address this issue, [44] leveraged simulation to compute the query answer without producing any redundant intermediate results.

2.4 Graph Pattern Query Evaluation Using Views

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 [45, 46, 47, 48, 49]. The idea is to pre-compute and store the matches of views and to rewrite an incoming query using exclusively the view materializations, if the query language is closed [45], or to otherwise provide a process for computing the query's match results from the view materializations [46].

Using materialized views for answering queries has been extensively studied for relational data and tree data [50, 51, 52, 47, 53]. 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 [54, 48, 55, 56, 57, 49] addressing the problem of answering graph pattern queries using views.

Fan et al. [48] investigated this problem for graph pattern queries based on graph simulation and studied its complexity. Under this setting, they characterized graph pattern matching using graph pattern views based on pattern containment, and provided 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 [56]. Another extension [55] studied the approximation of graph pattern queries using views based on both graph simulation and subgraph isomorphism.

More recently, Trindade et al. [49] presented a graph query optimization framework called Kaskade which materialized graph views to enable efficient query evaluation. Kaskade considered two types of views: path views which matched to a path of data nodes with bounded length, and relational counterparts which were filters and aggregates. Kaskade only supported query rewriting/answering using a single view. Unlike previous work, it focused 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. [54] 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 were subgraphs or supergraphs of the query. Unlike previous approaches, this approach considered the framework of a collection of small data graphs and aimed at minimizing the number of isomorphism tests that needed to be performed to find the data graphs that contained the query pattern.

Wu et al. [57] studied the problem of using materialized views for homomorphic pattern matching on data graphs, but considered only tree-pattern queries. Le et al. [58] studied the problems of rewritting SPARQL queries using views, but did not consider materializing these views.

The problem we address in this dissertation is different than those addressed by existing graph view approaches. We consider general graph patterns and not simply paths or trees. Our patterns contain direct and reachability 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.

CHAPTER 3

EVALUATION OF HYBRID TREE PATTERN QUERIES ON LARGE DATA GRAPHS

We begin by discussing our work on evaluating tree pattern queries on large data graphs. The techniques we apply to studying tree pattern queries will later be extended to graph pattern queries in Chapter 4. The concepts and algorithms introduced in this chapter lay a foundation to understand the methods discussed in later chapters of this dissertation.

3.1 Preliminaries and Problem Definition

Given a labeled data graph G and a tree pattern query Q, our goal is to efficiently find the matches of Q on G. We begin by introducing definitions for pattern matching. We describe the data model, which is in the form of a data graph. Next, we discuss edge-to-path mappings, which is necessary to describe the concept of reachability edges in graph pattern queries.

Definition 3.1.1 (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.

For each label $l \in \mathcal{L}$, the *inverted list* I_a contains the nodes in G with label a.

Definition 3.1.2 (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 a reachability indexing scheme. In most reachability indexing schemes, the data graph node labels are the entries in the index for the data graph [20]. Our approach can flexibly use any labeling scheme to check node reachability. In order to check if v is a child of u, the adjacency lists of the graph G can be used.

Queries. We consider tree pattern queries that involve direct and/or reachability edges.

Definition 3.1.3 (Hybrid Tree Pattern Query)**.** A hybrid tree pattern query is a tree $Q = (V_Q, E_Q)$, where V_Q is the set of nodes of Q and $E_Q = E_Q^c \cup E_Q^d$ is the set of edges of Q. Every node $x \in V_Q$ has a label *label*$(x) \in \mathcal{L}$. There can be two types of edges in E_Q. A *direct* edge $e_c \in E_Q^c$ denotes a child relationship between the respective two nodes, whereas a *reachability* edge $e_d \in E_Q^d$ denotes a descendant relationship.

Intuitively, a direct edge represents an edge in the data graph G. A reachability edge represents a path of edges in G. Figure 5.1(b) shows a query Q. Single line arrows denote direct edges while double line arrows denote reachability edges. Direct edges can also be called *child* edges, and reachability edges can also be called *descendant* edges.

Homomorphisms. Queries are matched to the data graph using an extension of homomorphism called *ep-homomorphism* (for edge-to-path homomorphism). In this dissertation, we use the terms of homomorphism and ep-homomorphism interchangeably, as we do not refer to any other type of homomorphism other than ep-homomorphism.

Definition 3.1.4 (Pattern Homomorphism)**.** Given a hybrid tree pattern query Q and a labeled data graph G, a *homomorphism* from Q to G is a function m mapping the nodes of Q to nodes of G, such that: (1) for each node $x \in V_Q$, *label*$(x) =$

label$(m(x))$; (2) for any edge $(x, y) \in E_Q^c$, $(m(x), m(y))$ is an edge of G; (3) for any edge $(x, y) \in E_Q^d$, $m(x) \prec m(y)$ in G.

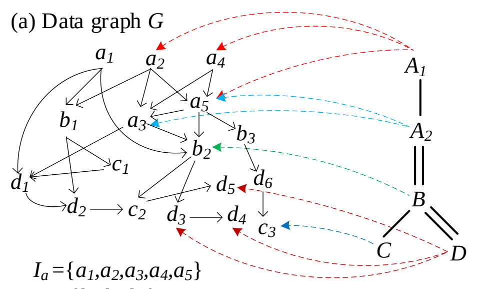
Clearly, a reachability edge in the query can be mapped by a homomorphism to a direct edge in the data graph.

Definition 3.1.5 (Pattern Occurrence)**.** An *occurrence* of a pattern query Q on a data graph G is 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 3.1.6 (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.

If x is a node in Q labeled by a, the *occurrence set of* x *on* G is a subset S_x of the inverted list I_a 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). The elements of S_x are called *occurrences* of x on G.

Let (q_i, q_j) be a query edge in Q, and v_i and v_j be two nodes in G, such that $label(q_i) = label(v_i)$ and $label(q_j) = label(v_j)$. The pair (v_i, v_j) is called a *match* of the query edge (q_i, q_j) if: (a) (q_i, q_j) is a direct edge in Q and (v_i, v_j) is an edge in G, or (b) (q_i, q_j) is a reachability edge in Q and $v_i \prec v_j$ in G. The pair (v_i, v_j) is called an *occurrence* of the query edge (q_i, q_j) if there is a homomorphism from Q to G which maps q_i to v_i and q_j to v_j.



$I_a = \{a_1, a_2, a_3, a_4, a_5\}$
$I_b = \{b_1, b_2, b_3\}$
$I_c = \{c_1, c_2, c_3\}$
$I_d = \{d_1, d_2, d_3, d_4, d_5, d_6\}$

(b) Inverted lists of G

A_1	A_2	B	C	D
a_2	a_3	b_2	c_2	d_3
a_2	a_3	b_2	c_2	d_4
a_2	a_3	b_2	c_2	d_5
a_2	a_5	b_2	c_2	d_3
		...		
a_4	a_5	b_2	c_2	d_3
		...		
a_5	a_3	b_2	c_2	d_5

(d) Answer of Q on G

$S_{A1} = \{a_2, a_4, a_5\}$
$S_{A2} = \{a_3, a_5\}$
$S_B = \{b_2\}$
$S_C = \{c_2\}$
$S_D = \{d_3, d_4, d_5\}$

(e) Occurrence sets of Q on G

Figure 3.1 A data graph G and its inverted lists, a query Q and its answer on G, and the occurrence sets of Q's nodes.

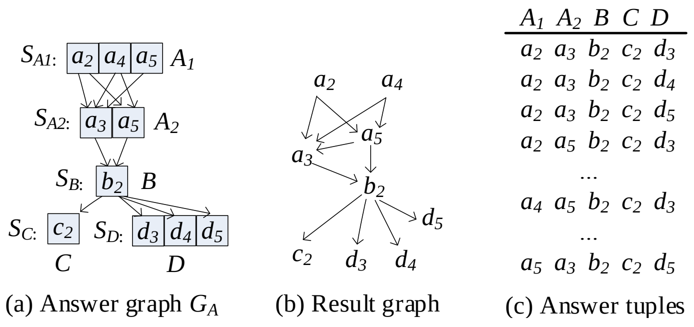
3.2 Query Answer Summarization, Counting and Enumeration

In this section, we propose the concept of the *answer graph* to compactly and losslessly encode all possible homomorphisms of a query in a graph. We also provide algorithms for counting and enumerating query results. Answer graph construction algorithms are presented in Sections 3.3 and 3.4.

Definition 3.2.1 (Answer Graph)**.** The answer graph G_A of a pattern query Q on a data graph G is a k-partite graph where k is the number of nodes in Q. Graph G_A has k disjoint sets of data graph nodes, one for every node $q \in V_Q$. The node set for node q in G_A is the occurrence set S_q of q. Moreover, there is an edge (v_x, v_y) in G_A between a node $v_x \in S_x$ and a node $v_y \in S_y$ if and only if there is an edge $(x, y) \in E_Q$ and (v_x, v_y) is an occurrence of (x, y) in G (i.e., there is a homomorphism from Q to G which maps x to v_x and y to v_y).

The answer graph reduces redundancy in the representation of the query results and allows for computation sharing during the generation of these results. The concept has analogies to the factorized representation of query results studied in the context of relational databases and probabilistic databases [59].

A different way of compacting graph pattern query results uses a concept called the *result graph* [41]. Unlike the answer graph, which is a k-partite graph that


S_{A1}: a_2 a_4 a_5 A_1
S_{A2}: a_3 a_5 A_2
S_B: b_2 B
S_C: c_2 S_D: d_3 d_4 d_5
C D
(a) Answer graph G_A
a_2 a_4 a_5 a_3 b_2 d_5 c_2 d_3 d_4
(b) Result graph
(c) Answer tuples

A_1	A_2	B	C	D
a_2	a_3	b_2	c_2	d_3
a_2	a_3	b_2	c_2	d_4
a_2	a_3	b_2	c_2	d_5
a_2	a_5	b_2	c_2	d_3
		...		
a_4	a_5	b_2	c_2	d_3
		...		
a_5	a_3	b_2	c_2	d_5

Figure 3.2 The answer graph of Q on G (Figure 3.1) and the corresponding result graph.

encodes query homomorphisms, a result graph is a subgraph of the data graph which represents pattern matchings defined in terms of (an extension of) graph simulation [39]. As such, it is not capable of compactly encoding homomorphic matches. Variations of the result graph data were used in [29, 30, 31, 7]. These variants of result graphs are different than answer graphs as: (a) they serve as a search space for subgraph isomorphisms, (b) they are designed for undirected graphs, and (c) they are built for a subgraph of the original query graph, and thus may contain redundant nodes.

Figure 3.2(a) shows the answer graph G_A of query pattern Q on the data graph G of Figure 3.1. As a comparison, Figure 3.2(b) shows the result graph of Q on G as defined in [41]. Clearly, the answer graph representation of a query answer is much more compact than the relational representation of the query answer. As an example, the query answer in Figure 3.2(c) (repeated here from Figure 3.1(d)) has 75 data graph node labels (15 five-label tuples), while the corresponding answer graph in Figure 3.2(a) only has 10 node labels.

3.2.1 Query Answer Counting

Given the answer graph G_A of Q on G, we can calculate the total number of solutions of Q on G from G_A without explicitly enumerating the tuples. Algorithm 1 describes the method. Its core procedure, called *count*, calculates the number of occurrences of the subquery of Q rooted at $q \in V_Q$ when q is matched to $n_q \in S_q$ (S_q is the independent node set of G_A for q). In particular, when q is a leaf node, *count* returns one. For other query nodes q, Procedure *count* exploits dynamic programming techniques to obtain the result based on the results of the children of q and n_q (Lines 2-5).

Algorithm 1: Query counting using a query answer graph.

```
Input  : Query pattern $Q$, and answer graph $G_A$ of $Q$ on data graph $G$.
Output: The number of solutions of $Q$ on $G$.

1  $cnt := 0$ ;
2  $q :=$ root($Q$) ;
3  for ($n_q \in S_q$) do
4      $cnt := cnt +$ count($q$, $n_q$) ;
5  end
6  return $cnt$

7  Procedure count($q$, $n_q$):
8      $c := 1$ ;
9      for ($q_i \in$ children($q$)) do
10         $c_i := 0$ ;
11         for (every $n_{q_i} \in S_{q_i}$ that is a child of $n_q$ in $G_A$) do
12             $c_i := c_i +$ count($q_i$, $n_{q_i}$) ;
13         end
14         $c := c * c_i$ ;
15     end
16     return $c$;
```

As an example, in Figure 3.2(a), *count* returns 1 for the nodes in the occurrence sets $\{c_2\}$ and $\{d_3, d_4, d_5\}$ of the query nodes C and D, respectively, since these two query nodes are leaves in Q. The occurrence set of B is $\{b_2\}$. Therefore, *count* returns 3, on B and b_2 as the occurrence sets of the children C and D of B have 1 and 3 nodes, respectively. On A_2, *count* returns 3 for both a_3 and a_5. Finally, *count* returns 6, 6, and 3 on the nodes a_2, a_4 and a_5 in the occurrence set of A_1. Hence, the total number of solutions of Q is 15.

Algorithm 2: DP: A dynamic programming method for tuple enumeration from a query answer graph.

```
Input  : Pattern query $Q$, and answer graph $G_A$ of $Q$ on data graph $G$.
Output: The answer of $Q$ on $G$.

1  $OC := \emptyset$ ;
2  $q :=$ root($Q$) ;
3  for ($n_q \in S_q$) do
4      $OC := OC \cup$ enumerate-D($q$, $n_q$) ;
5  end
6  return $OC$ ;

7  Procedure enumerate-D($q$, $n_q$):
8      if ($q$ is a leaf node of $Q$) then
9          return $\{n_q\}$ ;
10     end
11     $OC := \emptyset$ ;
12     for ($q_i \in$ children($q$)) do
13         $OC_i := \emptyset$ ;
14         for (every $n_{q_i} \in S_{q_i}$ which is a child of $n_q$ in $G_A$) do
15             $OC_i := OC_i \cup$ enumerate-D($q_i$, $n_{q_i}$) ;
16         end
17         $OC := OC \times OC_i$ ;
18     end
19     $OC := \{n_q\} \times OC$ ;
20     return $OC$ ;
```

3.2.2 Query Answer Enumeration

Given query pattern Q and its answer graph G_A, we present two different algorithms for enumerating result tuples of Q from G_A. The first one leverages dynamic programming techniques to avoid redundant computations, while the second one focuses on minimizing memory consumption.

A Dynamic Programming Method (DP). Let q be a node of Q and Q_q be the subquery of Q rooted at q. The first algorithm (Algorithm 2) generates the query answer by dynamic programming. For each $q \in V_Q$, it invokes procedure *enumerate-D* to generate the result tuples (the answer) of Q_q. When q is a leaf node, the answer of Q_q is a single-column relation containing the data nodes in the occurrence set S_q. When q is an internal node of Q, *enumerate-D* is recursively called on every child node of q and then computes the Cartesian product of the relations returned. The answer of query pattern Q is the union of the results obtained for the nodes in the occurrence set of the query root.

Similar to Algorithm 1, we can speed up the computation by caching expressions, the memoization technique. Consider, for instance, the example of Figure 3.1. We can store the answer for the subpattern rooted at node B when it is first generated while traversing the data graph G from a_3 in S_{A_2} and refer to it using a pointer when every subsequent node in S_{A_2} is considered. When generating the query answer, a check is performed to examine whether the answer of the subpattern we are about to compute has already been computed. If this is the case, a reference to it in the cache is returned. If not, the answer to the subpattern is computed and saved in the cache.

Through subquery answer caching, Algorithm 2 avoids repeatedly generating intermediate results. It has worst-case complexity linear in the sum of input and output sizes but independent of the size of intermediate results. Therefore, it is optimal among all sequential enumeration algorithms that read the entire input (i.e., G_A). This contrasts with many existing join-based graph pattern evaluation methods

[60, 15, 36, 61, 8]. These methods evaluate the given query by executing a sequence of binary joins, which may generate more redundant intermediate results than the actual query results.

Nevertheless, Algorithm 2 expands subquery occurrences aggressively and must maintain results generated for each subquery. As a result, its memory consumption can be very high for dense queries, i.e., queries having a large number of occurrences.

A Top-Down Method (TD). We now present another enumeration method (Algorithm 3) which has small memory footprints. It is a top-down method which recursively extends the prefix of a query occurrence by mapping the next query node to a data graph node. More concretely, the algorithm first picks a topological order $q_1, \ldots, q_n$ for the query nodes. Then, it calls procedure *enumerate-T*, which performs an iterative search on the query node occurrences sets according to the selected query node order to construct occurrences for the query.

Let q_i be the current query node under consideration, and S_i be the independent node set of q_i in the answer graph G_A (which is the occurrence set of q_i). Let also t be a tuple of length n, where $t[1], \ldots, t[i-1]$ contains an occurrence of the subquery of Q defined by the query nodes $q_1, \ldots, q_{i-1}$. Procedure *enumerate-T* identifies a subset S'_i of S_i which is computed as follows: If $i = 1$, q_i is the root of Q and S'_i is just S_i. When $i > 1$, S'_i is computed by intersecting S_i with *adjacency-list*($t[j]$), where $j \in [1, i-1]$ is the index of the parent, q_j, of q_i in the topological query node order and *adjacency-list* is a function that returns the forward adjacency list of a given node in G_A (lines 4-6).

Next, *enumerate-T* iterates over the occurrences of q_i in S'_i (line 7). In every iteration, it assigns the occurrence at hand to $t[i]$ (line 8) and then proceeds to the next recursive step (line 9). When $i = n+1$, tuple t contains one occurrence of query Q which is outputted as a result tuple of Q (line 2).

Algorithm 3: TD: A top-down method for tuple enumeration from a query answer graph

```
Input  : Pattern query $Q$, and answer graph $G_A$ of $Q$ on data graph $G$.
Output: The answer of $Q$ on $G$.
1  Pick a topological order $q_1, \ldots, q_n$ for the nodes of $Q$, where $n = |V_Q|$ ;
2  Let $t$ be a tuple where $t[i]$ is initialized to be $null$, $i \in [1, n]$ ;
3  Let $S_i$ be the independent node set of $q_i$ in $G_A$ (the occurrence set of $q_i$) ;
4  enumerate-T(1, $t$) ;
5  Procedure enumerate-T(index $i$, tuple $t$):
6      if ($i = |V_Q| + 1$) then
7          return $t$ ;
8      end
9      $S'_i := S_i$; if ($i > 1$) then
10         Let $j$ be the index of the parent of $q_i$ in $Q$, $j \in [1, i-1]$ ;
11         $S'_i := S_i \cap$ adjacency-list($t[j]$) ;
12     end
13     for (every node $v_i \in S'_i$) do
14         $t[i] := v_i$ ;
15         enumerate-T($i+1$, $t$)) ;
16     end
```

DP vs. TD. Both algorithms have different memory consumption during execution. DP decomposes the given query into a set of subqueries and computes the result tuples for each subquery by joining the result tuples of the subqueries rooted at its children. In contrast, TD executes in a pipelined fashion, returning one tuple of the given query at a time, thus avoiding keeping large intermediate results. When tuples in the query result are dumped to the secondary storage upon need, TD has very low memory consumption, which is linear on the size of the largest label inverted list. Hence, it is very well suited for in-memory evaluation. Yet, storing and reusing intermediate results can offer a substantial performance improvements in the presence of significantly skewed data [62].

So far, we have shown how the answer graph can be used for query result counting and enumeration. In the next two sections, we present methods for efficiently constructing the answer graph of an input query.

3.3 Answer Graph Construction—A Bottom-up Approach

In this section, we present a bottom-up dynamic programming algorithm which incrementally builds the answer graph for the pattern query, based on the answer graphs constructed for the sub-patterns rooted at the children of the node under consideration. We first present the main algorithm, and then introduce several optimizations.

3.3.1 Main Algorithm

Algorithm Overview. The algorithm presented in Algorithm 4, called BUP, takes as input a data graph G and a tree-pattern query Q, and builds the answer graph G_A of Q on G by performing a postorder traversal on Q.

We first define the concept of *candidate* occurrence set. A candidate occurrence set CS_q of a query node q in Q is the occurrence set of the root of the subquery of Q rooted at q. Clearly, when q is a leaf node in Q, $CS_q = I_{label(q)}$. In general, we have $CS_q \subseteq I_{label(q)}$ and $S_q \subseteq CS_q$.

Algorithm BUP proceeds in two phases. In the first phase, it generates the candidate occurrence sets for the query nodes in Q and links their nodes with edges. In this phase, it essentially applies multiway structural joins. A data node $v \in I_{label(q)}$ is added to the candidate occurrence set CS_q of node $q \in V(Q)$ if there exist data nodes $v_1, \ldots, v_k$ for the child query nodes $q_1, \ldots q_k$ of q such that: (a) for every $i \in [1, k]$, $v_i \in CS_{q_i}$, and (b) for every $i \in [1, k]$, (v, v_i) is an occurrence in G of the edge $(q, q_i) \in Q$. For every $i \in [1, k]$, an edge is added to G_A from node $v \in CS_q$ to node $v_i \in CS_{q_i}$ if and only if (v, v_i) is an occurrence of the edge $(q, q_i) \in Q$. Due to the bottom up traversal of Q, the candidate occurrence sets of the child nodes q_i of a node q are available from the previous iteration of the algorithm when q is considered.

In the second phase, BUP eliminates nodes in the candidate occurrence sets CS_q which are not in occurrence sets S_q, by executing a top-down traversal of the

Algorithm 4: Algorithm BUP for building a query answer graph.

```
Input   : Data graph $G$, and pattern query $Q$
Output: Answer graph $G_A$ of $Q$ on $G$

1  Initialize $G_A$ to be a k-partite graph without edges having one data node
     set $CS_q$ for every node $q \in V(Q)$ ;
2  $CS_q := \emptyset, \forall q \in Q$;
3  traverse($root(Q)$) ;
4  for (every $q \in Q$, $q \neq root(Q)$, in a top-down manner) do
5      Remove the data nodes of $CS_q$ which do not have an incoming edge ;
6  end

7  Procedure traverse($q$):
8      if ($isLeaf(q)$) then
9          $CS_q := I_{label(q)}$ ;
10         return
11     end
12     for ($q_i \in children(q)$) do
13         traverse($q_i$) ;
14     end
15     for ($v_q \in I_{label(q)}$) do
16         expand($q$, $v_q$) ;
17     end

18 Procedure expand($q$, $v_q$):
19     Append $v_q$ to $CS_q$ ;
20     for ($q_i \in children(q)$) do
21         for ($v_{q_i} \in CS_{q_i}$) do
22             if ($(v_q, v_{q_i})$ is an occurrence of the query edge $(q, q_i)$) then
23                 Add the edge $(v_q, v_{q_i})$ to $G_A$ ;
24             end
25         end
26         if (no occurrence of $(q, q_i)$ is found in $G$) then
27             remove $v_q$ from $CS_q$ ;
28             return
29         end
30     end
```

answer graph G_A under construction. In this phase, the algorithm eliminates nodes (and their outgoing edges) which do not have incoming edges.

Detailed Description. The bottom-up processing phase in BUP is realized by procedure *traverse*. Let q be the current query node under consideration. For each node v_q in $I_{label(q)}$, procedure *traverse* evokes procedure *expand* to potentially expand G_A by putting v_q into the candidate occurrence set CS_q and by adding incident edges to G_A (lines 6-7 in *traverse*).

Initially, CS_q is empty. Line 1 of Procedure *expand* puts v_q temporarily to CS_q. Then, line 2 iterates over every child q_i of q. For each node v_{q_i} in the candidate occurrence set CS_{q_i} of q_i, line 4 determines whether (v_q, v_{q_i}) is an occurrence of the query edge (q, q_i). If this is the case, the edge (v_q, v_{q_i}) is added to G_A. The following two cases are considered: (a) (q, q_i) is a reachability edge, and (b) (q, q_i) is a direct edge.

For case (a), a reachability index is used to check whether $v_q \prec v_{q_i}$. For case (b), the adjacency lists of the graph G can be used to check the child/parent relationship between v_q and v_{q_i}. This edge verification operation requires a runtime proportional to the degree of v_q or v_{q_i}, whichever is the smaller. When the adjacency list is sorted (by node id), the runtime reduces to log of that degree. While an adjacency matrix representation for the data graph, like the one adopted by CFLMatch [30] with size $|V| \times |V|$, would overcome this overhead, it can only be used with small data graphs. We will later present a more efficient way to check child constraints.

After the last element of CS_{q_i} is accessed, if for some (q, q_i) no match is found, v_q is removed from CS_q, and the procedure terminates (lines 6-8 of Procedure *expand*).

When Procedure *traverse* terminates after processing the root of Q, we have $CS_{root(Q)} = S_{root(Q)}$. The candidate occurrence lists CS_q for other nodes q of Q might contain data nodes that are not in S_q. To eliminate these nodes, a breadth first traversal of Q is performed. For every node q of Q encountered (other than the root

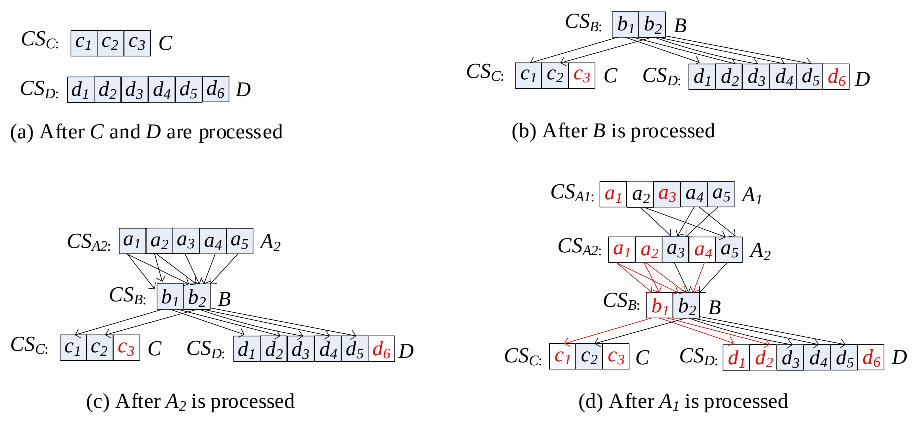


Figure 3.3 Snapshots of the answer graph during its construction using BUP on the query Q and data graph G of Figure 3.1.

node), all the data nodes which do not have an incoming edge are removed from G_A along with their incident outgoing edges (lines 6-7 in the main procedure). The resulting graph is the answer graph G_A of Q.

Figure 3.3 shows different snapshots from the answer graph construction when running BUP on the query Q and the data graph G of Figure 3.1. The final answer graph G_A is shown in Figure 3.2(a). In each snapshot, we mark in red those nodes which are not in the answer of the corresponding subquery of Q. A subsequent top-down traversal of the answer graph under construction recursively eliminates these marked nodes.

The following lemma is useful for showing the output of Algorithm BUP is the answer graph.

Lemma 1. Algorithm BUP adds a data node $v_q \in I_q$ in node set CS_q if and only if it is in the candidate occurrence set of the root of the subquery of Q rooted at q.

Proof. The "only-if" direction is obvious by the bottom-up processing nature of the algorithm. We now prove the "if" direction of the lemma, that is, we prove that if a data node $v \in I_q$ is in the candidate occurrence list of the root of the subquery of Q

rooted at q, it will be put into CS_q. We prove it by induction on the height d of the query nodes in Q (the height of leaf nodes is 0).

Base Case: $d = 0$. If q is a leaf query node, then every data node $v_q \in I_q$ is in the candidate occurrence set of q and is put into CS_q by BUP.

Inductive Case: Assuming that the claim holds for $d \leq k - 1$, let the height of the query node q under consideration by BUP be k. Let also v_q be a node in I_q which is in the occurrence set of the root of the subquery of Q rooted at q. By the inductive assumption, all the data nodes which are in the candidate occurrence sets of the roots of the subqueries of Q rooted at child query nodes q_i of q (their height is $\leq k - 1$) have been put into CS_{q_i}. Then, there exist nodes v_{q_i} in each CS_{q_i}, such that (v_q, v_{q_i}) is an occurrence of the query edge (q, q_i). Consequently, node v_q will be put in CS_q by algorithm BUP. $\square$

Theorem 1. The graph returned by Algorithm BUP when a query Q and a data graph G is given as input is the answer graph G_A for Q on G.

The proof follows from Lemma 1 by letting node q be the root node of the query pattern Q.

3.3.2 Optimizing BUP

We now discuss techniques to further improve the performance of BUP on large data graphs.

Early expansion termination. When expanding G_A with a node $v_q \in I_{label(q)}$, it is not always necessary to scan the entire set CS_{q_i} for every child node q_i of q. When the input data graph G is a dag, we can associate each node u in G with an interval label, which is an integer pair (*begin*, *end*) denoting the first discovery time of u and its final departure time in a depth-first traversal of G. Nodes in $I_{label(q)}$ are accessed in ascending order of the *begin* value of their interval labels, and nodes satisfying the query q are appended to CS_q in the same order. Interval labelling guarantees that node v_q does not reach node v_{q_i} if $v_q.end < v_{q_i}.begin$. Therefore, once such a node v_{q_i} is encountered, the expansion over CS_{q_i} can be safely terminated since all the subsequent nodes have a *begin* value which is larger than $v_{q_i}.begin$.

Intersection-based child relationship checking. As aforementioned, in order to find all the elements v_{q_i} in CS_{q_i} having a child relationship with v_q, the edge verification-based method needs to verify the existence of an edge between v_q and v_{q_i}. This can be time costly. We now present an efficient method which can find all the v_{q_i}s in CS_{q_i} having a child relationship with v_q in one step.

The child relationship checking is converted into a set intersection operation. More concretely, let A_{v_q} denote the adjacency list of v_q; every element v_{q_i} in the intersection $A_{v_q} \cap CS_{q_i}$ is a child of v_q in CS_{q_i}. We store A_{v_q} and CS_{q_i} as bit vectors, and implement the intersection using a bitwise AND operation. Our method is able to obtain all the nodes in CS_{q_i} satisfying a child relationship with v_q in one step.

Node pre-filtering. The performance of BUP can be improved by pruning redundant nodes from the inverted lists. A node pre-filtering technique was proposed in [22, 26] to filter out nodes not participating in the query answer before the query evaluation starts. The technique conducts two graph traversals and maintains data structures that record, for each data node, whether it has ancestors or descendants matching a particular query node. Our experimental results in Section 3.5 show that the pre-filtering technique can improve the time performance of graph matching algorithms. However, as this technique is designed for reachability edge-only tree patterns, it has limited pruning power for queries involving also direct edges.

3.4 Building the Answer Graph Using Graph Simulation

The BUP algorithm reduces the generation of redundant intermediate results by applying multi-way structural joins between the candidate occurrence set of a node and those of its child nodes in the query. However, it may still do redundant computations since it is unable to check structural join conditions globally. In this section, we introduce a method which exploits graph simulation in order to build the answer graph, extended to account for reachability edges. Our method is holistic and does not generate *any* redundant intermediate results.

3.4.1 The Double Simulation Relation

Different types of simulation have been implemented in different graph database applications [63, 41, 43, 40] As opposed to a homomorphism, which is a function, a simulation is a binary relation on the node sets of two directed graphs. It provides one possible notion of structural equivalence between the nodes of two graphs.

Double simulation. Since the structure of a node in a graph is determined by its incoming and outgoing paths, we define a type of simulation called *double simulation*, which handles both the incoming and the outgoing paths of the graph nodes. Double simulation is an extension of dual simulation [43] to allow edge-to-path mappings. In this paper, we consider the double simulation of hybrid tree pattern queries on graph data.

Definition 3.4.1 (Double Simulation)**.** The *double simulation* of a tree pattern query $Q = (V_Q, E_Q)$ by a directed data graph $G = (V_G, E_G)$ is the largest binary relation $S \subseteq V_Q \times V_G$ such that, whenever $(q, v) \in S$, the following conditions hold:

1. $label(q) = label(v)$.

2. For each $(q, q') \in E_Q$, there exists $v' \in V_G$ such that $(q', v') \in S$ and (v, v') is a match of the edge (q, q') .

3. If $q \neq root(Q)$ and $(q', q) \in E_Q$, there exists $v' \in V_G$ such that $(q', v') \in S$, and (v', v) is a match of the edge (q', q).

Recall that a pair (v, v') of data graph nodes is a match of an edge (q, q') in Q if: (a) (q, q') is a direct edge in Q and (v, v') is an edge in G, or (b) (q, q') is a reachability edge in Q and $v \prec v'$ in G.

Table 3.1 Forward ($\mathcal{F}$), Backward ($\mathcal{B}$), and Double ($\mathcal{FB}$) Simulation of the Query Q on the Graph G of Figure 3.1

q	$\mathcal{F}(q)$	$\mathcal{B}(q)$	$\mathcal{FB}(q)$
A_1	$\{a_2, a_4, a_5\}$	$\{a_1, a_2, a_3, a_4, a_5\}$	$\{a_2, a_4, a_5\}$
A_2	$\{a_1, a_2, a_3, a_4, a_5\}$	$\{a_3, a_5\}$	$\{a_3, a_5\}$
B	$\{b_1, b_2\}$	$\{b_2, b_3\}$	$\{b_2\}$
C	$\{c_1, c_2, c_3\}$	$\{c_2\}$	$\{c_2\}$
D	$\{d_1, d_2, d_3, d_4, d_5, d_6\}$	$\{d_3, d_4, d_5, d_6\}$	$\{d_3, d_4, d_5\}$

Just as in Chapter 3, the double simulation of Q by G is unique, since there is exactly one largest binary relation S satisfying the above three conditions. This can be proved by the fact that, whenever we have two binary relations S_1 and S_2 satisfying the three conditions between Q and G, their union $S_1 \cup S_2$ also satisfies those conditions.

We call the largest binary relation that satisfies conditions 1 and 2 *forward simulation* of Q by G, and we call the largest binary relation that satisfies conditions 1 and 3 *backward simulation.* We denote the forward, backward, and double simulation by $\mathcal{F}$, $\mathcal{B}$, and $\mathcal{FB}$, respectively. For $q \in V_Q$, $\mathcal{F}(q)$, $\mathcal{B}(q)$, and $\mathcal{FB}(q)$ denote the set of all nodes of V_G that forward, backward, and double simulate q, respectively. $\mathcal{FB}$ preserves both incoming and outgoing edge types (direct or reachability) between Q and G, whereas $\mathcal{F}$ and $\mathcal{B}$ preserve only outgoing and incoming edge types, respectively.

Table 3.1 shows the simulations $\mathcal{F}$, $\mathcal{B}$, and $\mathcal{FB}$ of the query Q on the graph G of Figure 3.1. In particular, for the reachability query edge (B, D) of Q, the matches considered for double simulation are $(b_2, d_3), (b_2, d_4), (b_2, d_5)$.

The following theorem shows the significance of double simulation in graph pattern matching: all graph data nodes captured by the double simulation participate in the query's final answer.

Theorem 2. Given a tree pattern query Q and a data graph G, there exists a homomorphism from Q to G that maps node $q \in V_Q$ to node $v \in V_G$ if and only if $v \in \mathcal{FB}(q)$.

Proof. The "only if" direction is obvious since the structural constraints imposed by a homomorphism from Q to G imply those imposed by the $\mathcal{FB}$ simulation of Q by G. We prove next the "if" direction of the theorem.

Let Q_q denote the subquery of Q that consists of the path from the root of Q to q and the subtree rooted at q in Q. We first prove the following claim: if $v \in \mathcal{FB}(q)$, there exists a homomorphism from Q_q to G that maps node $q \in V_Q$ to node $v \in V_G$. We prove this by induction on the height k of q in Q. The height of a node in Q is the length of the longest path from that node to a leaf node; a leaf has height 0, and the root of Q has the largest height. First, let $k = 0$ (in this case, q is a leaf in Q_q). Let $q_r, \ldots, q_1, q_0 = q$ be the path from the root q_r of Q to q. Since $v \in \mathcal{FB}(q)$, there is a path of nodes $v_r, \ldots, v_1, v_0 = v$ in G such that $v_{i+1} \in \mathcal{FB}(q_{i+1})$, and $(v_{i+1}, v_i) \in E_G$ (resp. $v_{i+1} \prec v_i$ in G) if (q_{i+1}, q_i) is a direct (resp. reachability) edge in E_Q, for every $i \in [0, r-1]$. Clearly, the mapping that maps the query nodes $q_r, \ldots, q_1, q$ to the data graph $v_r, \ldots, v_1, v$, respectively, is a homomorphism from Q_q to G.

Assuming that the claim holds for $k \leq i$, we next show that it also holds for $k = i + 1$. Let $c_1, c_2, \ldots, c_n$ be the child nodes of q in Q_q. Since $v \in \mathcal{FB}(q)$, for each c_j ($1 \leq j \leq n$), there exists a node $v_{c_j} \in V_G$ such that $v_{c_j} \in \mathcal{FB}(c_j)$ and $(v, v_{c_j}) \in E_G$ (resp. $v \prec v_{c_j}$ in G) if (q, q_{c_j}) is a direct (resp. reachability) edge in E_Q. As the height of c_j is $\leq i$, by the inductive assumption, for each Q_{c_j} there exists a homomorphism h_{c_j} from Q_{c_j} to G that maps node $c_j \in V_Q$ to node $v_{c_j} \in V_G$. We can construct a mapping h from the nodes of Q to nodes in G such that for every $j \in [1, n]$, h coincides with h_{c_j} on the nodes of the subtree of Q rooted at c_j. Further, $h(q) = h(v)$. Finally, the nodes in the path $root(Q) = q_r, \ldots, q_1, q_0 = q$ in Q are mapped by h to nodes $v_r, \ldots, v_1, v_0 = v$, respectively, in G such that $v_{i+1} \in \mathcal{FB}(q_{i+1})$, and $(v_{i+1}, v_i) \in E_G$ (resp. $v_{i+1} \prec v_i$ in G) if (q_{i+1}, q_i) is a direct (resp. reachability) edge in E_Q, for every $i \in [0, r-1]$. The later node mapping is possible because $v \in \mathcal{FB}(q)$. Clearly, h is a homomorphism from Q_q to G. □

Computing double simulation. The computation of the double simulation of a graph pattern query by a data graph is the algorithmic basis of our simulation-based method. Note that, for every node $q \in V_Q$, $\mathcal{FB}(q) \subseteq \mathcal{F}(q)$ and $\mathcal{FB}(q) \subseteq \mathcal{B}(q)$, but, in general, $\mathcal{FB}(q) \neq \mathcal{F}(q) \cap \mathcal{B}(q)$. An example of this inequality can be seen in Table 3.1. Therefore, we cannot compute $\mathcal{FB}(q)$ simply by intersecting $\mathcal{F}(q)$ and $\mathcal{B}(q)$.

We develop a 2-pass algorithm called *FBSim* to compute $\mathcal{FB}$ for query Q and data graph G by traversing the nodes of Q two times. *FBSim* leverages the acyclic nature of the rooted tree pattern. It first computes the forward simulation $\mathcal{F}$ (considering outgoing query edges), and then refines $\mathcal{F}$ by computing a subset of the backward simulation $\mathcal{B}$ (considering the incoming query edge) which is equal to $\mathcal{FB}$.

Algorithm 5 shows the pseudocode for *FBSim*. The algorithm first invokes procedure *getFSim()* to compute $\mathcal{F}$ by traversing nodes of Q in a bottom-up way (line 2). When $q \in V_Q$ is a leaf node in Q, variable F_q is equal to $I_{label(q)}$ ($I_{label(q)}$ is the inverted list of nodes in G having the label of query node q). When q is an internal node, a node $v \in I_{label(q)}$ is added to F_q if there are nodes $v_1, \ldots, v_k \in V_G$ for the child query nodes $q_1, \ldots q_k$ of q such that: for every $i \in [1, k]$, $v_i \in F_{q_i}$, and (v, v_i) is an occurrence of the query edge (q, q_i) in G. Due to the bottom-up traversal, the set F_{q_i} for each child node q_i of $q \in V_Q$ is available from the previous iteration.

Based on the forward simulation relation $\mathcal{F}$, *FBSim* proceeds to compute the forward and backward simulation relation $\mathcal{FB}$ (by computing node sets FB_q for every $q \in V_q$) using procedure *getBSim()* (line 3). The procedure computes FB_q by traversing nodes q of Q in a top-down manner. When q is the root node of Q, $FB_q = F_q = \mathcal{F}(q)$. Otherwise, let q' be the parent of q in Q. A node $v \in F_q$ is in FB_q if there exists node $v' \in FB_{q'}$, such that (v', v) is an occurrence of the edge $(q', q) \in Q$ in G .

As we show below in Theorem 3, after procedure *getBSim()* terminates, for every node $q \in V_Q$, $FB_q = \mathcal{FB}(q) \subseteq F_q = \mathcal{F}(q)$. Based on Theorem 2, F_q is equal to the occurrence set L_q for every q in Q.

Algorithm 5: Algorithm *FBSim* for computing double simulation.

```
Input   : Data graph $G$, pattern query $Q$
Output: Data node set $FB_q$ for every $q \in V_Q$ (the double simulation $\mathcal{FB}$
         of $Q$ by $G$)

1  $F_q := \emptyset$ and $FB_q := \emptyset$, $\forall q \in V_Q$ ;
2  getFSim() ;
3  getBSim() ;
4  return $FB_q$ for every $q \in V_q$ ;

5  Procedure getFSim():
6      $F_q := I_{label(q)}$, for all leaf nodes $q \in V_Q$ ;
7      for (every internal node $q$ in $V_Q$ in a bottom-up order) do
8          for ($v_q \in I_{label(q)}$) do
9              if ($\forall$ child $q_i$ of $q$ in $Q$, $\exists v_{q_i} \in F_{q_i}$ s.t. $(v_q, v_{q_i})$ is an occurrence of
                 $(q, q_i) \in E_Q$) then
10                 $F_q := F_q \cup \{v_q\}$ ;
11             end
12         end
13     end

14 Procedure getBSim():
15     $r := root(Q)$ ;
16     $FB_r := F_r$ ;
17     for (every non-root node $q_i$ in $V_Q$ in a top-down order) do
18         $q := parent(q_i)$ ;
19         for ($v_{q_i} \in F_{q_i}$) do
20             if ($\exists v_q \in FB_q$ s.t. $(v_q, v_{q_i})$ is an occurrence of $(q, q_i) \in E_Q$) then
21                 $FB_{q_i} := FB_{q_i} \cup \{v_{q_i}\}$ ;
22             end
23         end
24     end
```

Consider again the example of Figure 3.1. Using procedure *getFSim()*, we obtain the forward simulation of Q by G, $\mathcal{F}$, shown in Table 3.1. Based on $\mathcal{F}$, we call *getBSim()* to compute $\mathcal{FB}$.

Note that the order of execution of the two phases in *FBSim* is significant. Algorithm *FBSim* first computes $\mathcal{F}$, then based on $\mathcal{F}$, it computes a subset of $\mathcal{F}$ and $\mathcal{B}$ to return $\mathcal{FB}$. Applying first a top-down traversal on Q to compute the backward simulation $\mathcal{B}$ and then a bottom-up traversal on Q to obtain the forward simulation $\mathcal{F}$ might erroneously add a data graph node to FB_q. Consider the following scenario where edge (p, q) of query Q has only one match (u, v) in graph G, node u is in $\mathcal{B}(p)$ but not in $\mathcal{F}(p)$ and node v is in $\mathcal{F}(q)$. In this setting, based on definition 3.4.1, v is not in $\mathcal{FB}(q)$. Assume that the algorithm computes $\mathcal{B}$ first by a top-down traversal on Q. Since p is a parent of q, p is processed before q. Given that $u \in \mathcal{B}(p)$ and (u, v) is a match of (p, q), the algorithm derives that $v \in \mathcal{B}(q)$. Once $\mathcal{B}$ is computed, the algorithm computes $\mathcal{F}$ using $\mathcal{B}$ with a bottom-up traversal on Q. Since q is a child of p, q is processed before p. As v will be discovered to be in $\mathcal{F}(q)$, and v has been added to $\mathcal{B}(q)$ in the top-down processing phase, v will be wrongly added to FB_q.

An example is data graph node d_6 in Figure 3.1 which, as Table 3.1 shows, appears in $\mathcal{F}(D)$ and $\mathcal{B}(D)$ but not in $\mathcal{FB}(D)$. Node d_6 will be (incorrectly) outputted as an element of $\mathcal{FB}(D)$ if a top-down traversal of query Q precedes a bottom-up traversal.

Theorem 3. Algorithm *FBSim* correctly computes the double simulation $\mathcal{FB}$ of Q by G.

Proof. Algorithm *FBSim* first incrementally constructs $\mathcal{F}$ in a bottom-up way on the query pattern by considering all pairs of query nodes q and data graph nodes from $I_{label(q)}$ and by disqualifying pairs violating the conditions of the definition of forward simulation (Definition 3.4.1). All the nodes in I_q for leaf nodes $q \in V_Q$

trivially satisfy the forward simulation conditions. For internal nodes q in Q, nodes in $I_{label(q)}$ are eliminated when they do not have the required child/descendant nodes in V_G. Since all possible pairs of nodes are considered and the elimination is done in a bottom-up way the computed relation is indeed the forward simulation $\mathcal{F}$ of Q by G.

Based on the computed forward simulation relation $\mathcal{F}$, *FBSim* incrementally computes $\mathcal{FB}$ by considering nodes in the query pattern in a top-down manner and by disqualifying pairs of nodes violating the conditions of the definition of backward simulation (Definition 3.4.1). Nodes in $\mathcal{F}(q)$ for the root q of Q trivially satisfy the backward simulation conditions. For other query nodes $q \in V_Q$, data graph nodes in $\mathcal{F}(q)$ are discarded only when they do not have the required parent/ancestor node in V_G. Since Q is a tree, each internal node q has only one parent in Q. Therefore, data graph node removal from $\mathcal{F}(q)$ will not make other data graph nodes violate the forward simulation conditions. Thus, the resulted binary relation is the largest one satisfying both the forward and backward simulation conditions, hence it is equal to $\mathcal{FB}$. □

Analogy to a full reducer. We can draw an analogy between *FBSim* and the full reducer problem for semi-joins on relational databases [64]. Given an acyclic query Q, a full reducer aims at removing all the *dangling tuples* from relations of Q, defined as tuples that do not participate in the result of Q. A full reducer as realized by [37], first forms a join tree of Q, and then applies semi-joins in a bottom-up and a top-down phase. It have been shown that the two semi-join sequences remove all dangling tuples.

Like full reducer, *FBSim* has a bottom-up phase and a top-down phase. However, the two methods work with different data structures and have different outputs: a full reducer works on relations, removes dangling tuples and returns reduced relations of the query, whereas *FBSim* works on the data graph, prunes all the nodes from the candidate occurrence sets of the query nodes that are not in the query answer and returns occurrence sets of the query nodes.

3.4.2 The Simulation-Based Algorithm

We now present *SIM*, our holistic pattern matching algorithm that builds the answer graph for a given pattern query in two phases. First, it computes the double simulation relation to find all the answer graph nodes, filtering out those that do not participate in the final answer. Then, it links the answer graph nodes with edges to construct the final answer graph.

The outline of *SIM* is presented in Algorithm 6. The algorithm takes as input a data graph G and a pattern query Q. The two phases of the construction of the answer graph G_A of Q on G are: (a) the *node selection* phase, and (b) the *node linking* phase.

Algorithm 6: Algorithm *SIM* for building a query answer graph

```
Input   : Data graph $G$, pattern query $Q$
Output: Answer graph $G_A$ of $Q$ on $G$

1  Use Algorithm FBSim to compute $\mathcal{FB}$ of $Q$ by $G$ ;
2  Initialize $G_A$ as a $k$-partite graph without edges having one independent
     set for every node $q \in V_Q$ which is the occurrence set $S_q = \mathcal{FB}(q)$ ;
3  for ( $q \in V_Q$ in a top-down order) do
4      for ($v_q \in S_q$) do
5          expand($q$, $v_q$) ;
6      end
7  end
8  return $G_A$ ;

9  Procedure expand($q$, $v_q$)):
10     for ($q_i \in$ children($q$)) do
11         for ($v_{q_i} \in S_{q_i}$) do
12             if (($v_q, v_{q_i}$) is an occurrence of edge $(q, q_i) \in E_Q$) then
13                 Add the edge $(v_q, v_{q_i})$ to $G_A$ ;
14             end
15         end
16     end
```

The node selection phase computes the double simulation relation of Q by G using algorithm *FBSim* of Algorithm 5 (line 1). As shown in Section 3.4.1, after relation $\mathcal{FB}$ is computed, the occurrence set S_q for each node q of Q is available (line 2). The node linking phase traverses Q in a top-down manner and links nodes in the occurrence sets S_q with edges to produce the answer graph G_A (lines 3-5). This is implemented by procedure *expand*. Let q be the current query node under consideration. For each node $v_q \in S_q$, procedure *expand* expands G_A by adding incident edges to v_q. More concretely, it iterates over every child q_i of q (line 1). For each node $v_{q_i} \in S_{q_i}$ of q_i, it determines whether (v_q, v_{q_i}) is an occurrence of the query edge (q, q_i) (line 4). If so, it adds the edge (v_q, v_{q_i}) to G_A (line 5).

Checking query edge occurrence. Checking whether (v_q, v_{q_i}) is an occurrence of the query edge (q, q_i) is a core operation needed by two different processes in SIM: for the computation of $\mathcal{FB}$, and for linking nodes with edges. For this, we apply the intersection-based child relationship checking method described in Section 3.3.2.

Consider building the answer graph of the pattern query Q on the data graph G of Figure 3.1 with SIM. With the node selection phase, the relation $\mathcal{FB}$ is computed (Table 3.1). Using $\mathcal{FB}$ as input, the node linking phase of SIM is executed and returns the answer graph G_A shown in Figure 3.2(a).

3.5 Experimental Evaluation

In this section, we present a thorough evaluation of our proposed approach.

3.5.1 Experimental Setup

Evaluated methods. We implemented four variants of our approach. The optimizations described in Subsection 3.3.2 were applied to the algorithms in the implementation:

- BUP-DP and BUP-TD: the bottom-up algorithm (BUP) for constructing the query answer graph (Section 3.3), combined with the dynamic-programming (DP) and the top-down method (TD), respectively, for enumerating answer tuples (Section 3.2.2).

- SIM-DP and SIM-TD: the simulation-based algorithm (SIM) for constructing the query answer graph (Section 3.4), combined with the dynamic-programming (DP) and the top-down method (TD), respectively, for enumerating answer tuples (Subsection 3.2.2).

Existing pattern matching algorithms can be broadly classified into the edge-join approach (EJ) and the path-join approach (PJ). EJ first computes the occurrences for each edge of the input query on the data graph, then finds an optimized join plan for joining these binary relations, and finally uses this plan to evaluate the query [15, 13, 8, 11]. PJ produces the query answer in two phases. It first generates solutions to each root-to-leaf path of the given query, and then generates the query answer by joining the path solutions [22, 26, 29, 30].

We implemented both approaches for finding homomorphisms of graph pattern queries on data graphs. In our implementation of EJ, in order to evaluate the binary joins, an optimized left-deep join plan is found through dynamic programming as suggested in [15]. Finding the occurrences of the reachability edges of the query pattern was implemented using a recent efficient reachability scheme called *Bloom Filter Labeling* (BFL) [20], which was shown to greatly outperform most existing schemes. For PJ, the solutions of each root-to-leaf path of the given query are merge-joined in order to generate the query answer as suggested in [26], and Bloom Filter Labeling was used for this approach too. As a preprocessing step, we often applied to both approaches (EJ and PJ) the node pre-filtering technique described in [26].

Our implementation was coded in Java. All the experiments reported here were performed on a workstation running Ubuntu 16.04 with 32GB memory and 8 cores

Table 3.2 Key Statistics of the Real-World Graph Datasets

Domain	Dataset	# of nodes	# of edges	# of labels	Avg #incident edges
Citation Networks	Citation[1]	1,397K	3,021K	16,4421	4.32
	Citeseerx[2]	6.3M	14.3M	1K∼10K	4.59
Social Networks	Epinions	76K	509K	20	6.87
Communication Networks	Email	265K	420K	20	2.6

of Intel(R) Xeon(R) processor (3.5GHz). The Java virtual machine memory size was set to 16GB.

Data graphs. We used both real-world and synthetic datasets to evaluate the algorithms in comparison. We provide their details below.

Real-world datasets. We selected four real-world graph datasets which have been used extensively in previous works [8, 31, 9, 65]. The datasets have different structural properties and come from a variety of application domains: citation networks, social networks, and communication networks. Table 6.1 lists the properties of the datasets. The average number of incident edges listed in the last column refers to both incoming and outgoing edges per node. It should be noted that the original Citeseerx graph does not have labels. Therefore, we randomly assigned distinct labels to nodes. This is a technique commonly used for generating labeled data graphs [30, 31, 9]. We generated ten labeled Citeseerx graphs whose numbers of labels range from 1K to 10K.

Synthetic datasets. We implemented a random graph generator to generate synthetic datasets. Given three input parameters n, m, and l, the generator first creates a random graph with n nodes and m edges; then, it randomly assigns l distinct labels to the nodes. Using this graph generator, we generated five graphs

[1] www.aminer.cn/citation, Last accessed on 2022/09/20.

[2] citeseerx.ist.psu.edu, Last accessed on 2022/09/20.

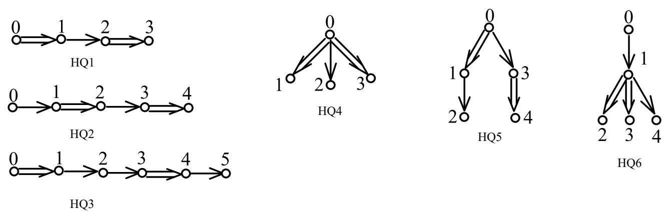


Figure 3.4 Hybrid pattern templates used for evaluation.

Table 3.3 Parameters for Query Generation

Parameters	**Range**	**Description**
Q	300 to 3300	Number of queries
D	6 to 16	Maximum depth of queries
DS	0 to 1	Probability for an edge to be a reachability edge ('//')
NP	1 to 3	Number of branches per query node

varying the number of nodes n from 300K to 1.5M, setting the number of edges m to $2 \times n$, and the number l of labels to $5K$.

Queries. We designed three query sets for each of the two data graphs Epinions and Email, which are distinguished by the type of queries contained: direct edge-only, reachability edge-only, and hybrid. Each query set contains 3 path queries and 3 tree queries, respectively. The templates of the hybrid query set are shown in Figure 3.4. Double line edges of a query template denote reachability edges, while single line edges denote direct edges. The number associated with each node of a query template denotes the node id. Query occurrences are generated by assigning labels to nodes. The direct-only and reachability-only query sets contain queries having the same structure as the hybrid queries, with their edges being exclusively direct edges and reachability edges, respectively.

For the Citation, Citeseerx and synthetic datasets, we used randomly generated queries. We implemented a query generator that creates a set of tree pattern queries based on the parameters listed in Table 3.3. For each data graph, we first generated a number of queries (with cardinality ranging from 300 to 3300) using different value combinations of the parameters listed in Table 3.3, and then formed a query set by randomly selecting 10 of these queries. The number of nodes of each query ranges from 3 to 22.

Metrics. We measured the runtime of individual queries in a query set. For query listing, this includes two parts: (1) the matching time, which consists of the time spent on filtering data graph nodes, building auxiliary data structures such as answer graphs and generating query plans (for EJ only), and (2) the tuple enumeration time, which is the time spent on enumerating results. The number of matches for a given query on a data graph can be quite large. Following usual practice [31, 9], we terminated the evaluation of a query after finding 10^7 matches covering as much of the search space as time allowed. We stopped the execution of a query if it did not complete within 10 minutes, so that the experiments could be completed in a reasonable amount of time. We refer to these queries as unsolved. For tuple counting, we measured the time spent on counting the result tuples instead of the time spent on enumerating tuples. To evaluate an algorithm on a query set instead of an individual query, we reported the average runtime of the queries in the query set.

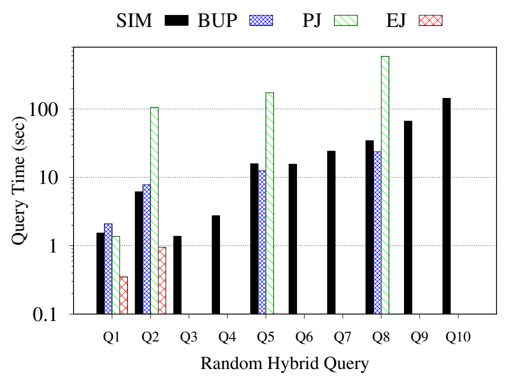
3.5.2 Results on Query Counting

We first compare the four algorithms SIM, BUP, PJ and EJ on query counting. Given a query, the first three algorithms build the answer graph, then use the answer graph to compute the number of result tuples without enumerating them, as shown in Section 3.2. Our experimental results show that calculating the result number using the answer graph takes almost no time for all the queries. The main difference among the three algorithms lies in the query matching process. In contrast, EJ does not construct the answer graph. It executes a sequence of binary joins to evaluate the query. We implemented binary joins using the classical hash join operator. For the query counting task, EJ generates tuples for each intermediate binary join. For the last binary join, EJ only counts the number of matching tuples in the hash table for each probing tuple, without computing the join. The sum of these matching numbers is returned as the total number of query results.

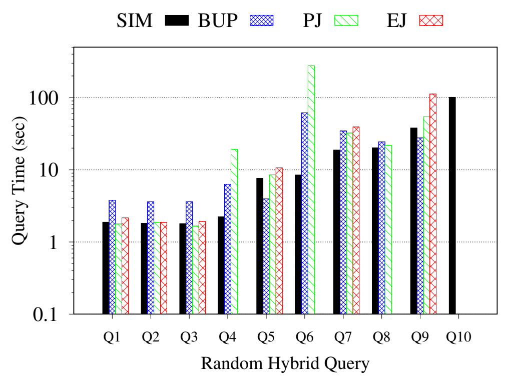
Performance Comparison on Citation Graph We compare the performance of SIM, BUP, EJ and PJ running 10 randomly generated queries over Citation with and without applying node pre-filtering. The average number of nodes per query is 7. Among the ten queries, Q_0, Q_1 and Q_2 are very sparse, with less than 10^3 result tuples; the number of results of Q_4 and Q_6 ranges from 10^4 to 10^5; Q_3, Q_5, Q_7 and Q_8 are dense, and their result numbers range from 10^{10} to 10^{12}. Query Q_9 has the largest number of results which lies in the order of 10^{17}. Figure 3.5 shows the experimental results. A general observation is that the pre-filtering technique can improve the time performance of graph matching algorithms, especially for PJ, for which this technique was originally designed.

Overall, in Figure 3.5, the best performer is SIM, followed by BUP, PJ and EJ in this order. SIM solves 100% of the queries both with or without pre-filtering. Both BUP and PJ solve 90% of the queries with pre-filtering, but without pre-filtering this percentage drops to 40% due to timeout, so they only solve Q_1, Q_2, Q_5, and Q_8. Without pre-filtering, EJ only solves Q_1 and Q_2, getting a timeout warning for the other eight queries. With pre-filtering, EJ solves four more queries. It gets an out-of-memory error when computing Q_4, Q_6, and Q_8.

The experiment confirms the benefits of our technique which uses query answer graphs for the query counting task. It also shows that the matching method of both SIM and BUP greatly outperforms that of PJ.



(a) No Pre-filtering



(b) With Pre-filtering

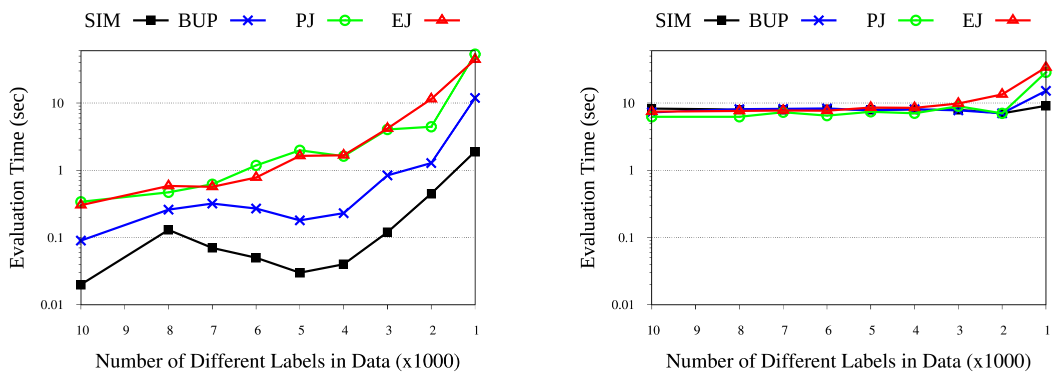
Figure 3.5 Performance comparison on Citation.

Scalability Comparison on Citeseerx and Random Graphs

Varying data labels. In this experiment, we examine the impact of the total number of distinct graph labels on the query counting performance of the algorithms in comparison. We used the aforementioned labeled Citeseerx graph, to produce 10 versions of it where the number of labels increases from 1K to 10K. For each labeled graph, we generated a query set of 10 distinct queries whose average number of nodes per query ranges from 3 to 4. Queries in the query sets are sparse in general. Figure 3.6 reports on the execution time of the four algorithms with and without applying node pre-filtering. In this experiment, the four algorithms are able to solve all the queries in all the cases.

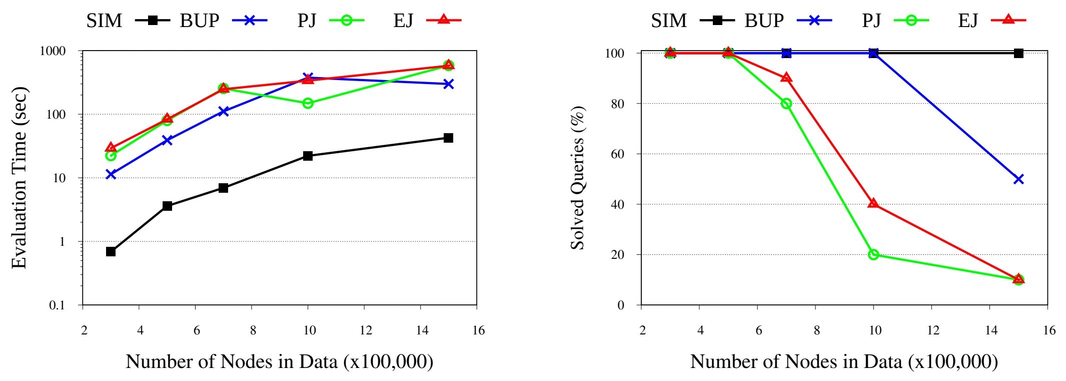
As can be seen in Figure 3.6, the execution time of the algorithms tends to increase while decreasing the total number of graph labels. In particular, the increase rate becomes steeper when the number is close to 1K. This is reasonable since the average cardinality of the input label inverted lists in a graph increases when the number of distinct labels in the graph decreases.

As with the experiment on Citation, SIM has the best overall performance and BUP comes next, while PJ and EJ show similar performance when the cardinality of the input inverted lists is high.



(a) No Pre-filtering (b) With Pre-filtering

Figure 3.6 Performance comparison on Citeseerx.



(a) Time (b) Solved Queries

Figure 3.7 Performance comparison on the randomly generated graphs.

In contrast to the results on Citation, in the experiment on Citeseerx we observe an increase in the execution time of the algorithms when the pre-filtering technique is applied to graphs with a large (>1K) number of labels. The reason is that when the number of graph labels is large, the cardinality of the inverted lists is relatively small. In this case, the potential benefit of reducing intermediate results during graph matching can be offset by the overhead of node pre-filtering (the node pre-filtering needs to traverse the whole data graph two times [26]).

Varying random graph sizes. In this experiment, we evaluate the performance of the algorithms on randomly generated graphs. We used the aforementioned five random graphs where the number of nodes was increased from 300K to 1,500K, while the total number of labels is fixed to 5,000. For each random graph, we generated a query set with 10 distinct queries. The average number of nodes per query ranges from 4 to 7. The average numbers of solution tuples per query ranges from 1K to 265K. Prefiltering was applied in all cases.

Figure 3.7 shows the results. As expected, the execution time for all algorithms increases with the increase of the total number of graph nodes. SIM shows significantly better time performance than the other algorithms across the entire range of random graphs (Figure 3.7(a)). In terms of solved queries, SIM solves 100% of queries in all the cases. BUP solves only 50% queries when the number of graph nodes increases to 1500K. Both PJ and EJ show especially poor scalability. When the number of nodes increases from 700K to 1500K, the number of queries solved by these algorithms decreases sharply from 80% to merely 10% due to timeout.

3.5.3 Results on Query Answer Enumeration

We now compare the three algorithms SIM, BUP, and EJ on query enumeration. We omit PJ here since we have shown in Subsection 3.5.2 that the performance of its matching method (the construction of the answer graph) is greatly outperformed by both SIM and BUP. For the query listing task, both SIM and BUP first build the answer graph for the given query, then call either algorithm DP (dynamic-programming) or TP (top-down) to enumerate the result tuples from the answer graph. In contrast, EJ first generates a query plan, then executes a sequence of binary joins to generate the result tuples for the query. We conduct the experiment on two data graphs, Epinions and Email, evaluating query instances of 6 query templates. Figure 3.4 shows the templates of hybrid queries.

Query Time. Figure 3.8 shows the elapsed time of the different algorithms on evaluating direct-edge-only, hybrid, and reachability-edge-only queries on Email and

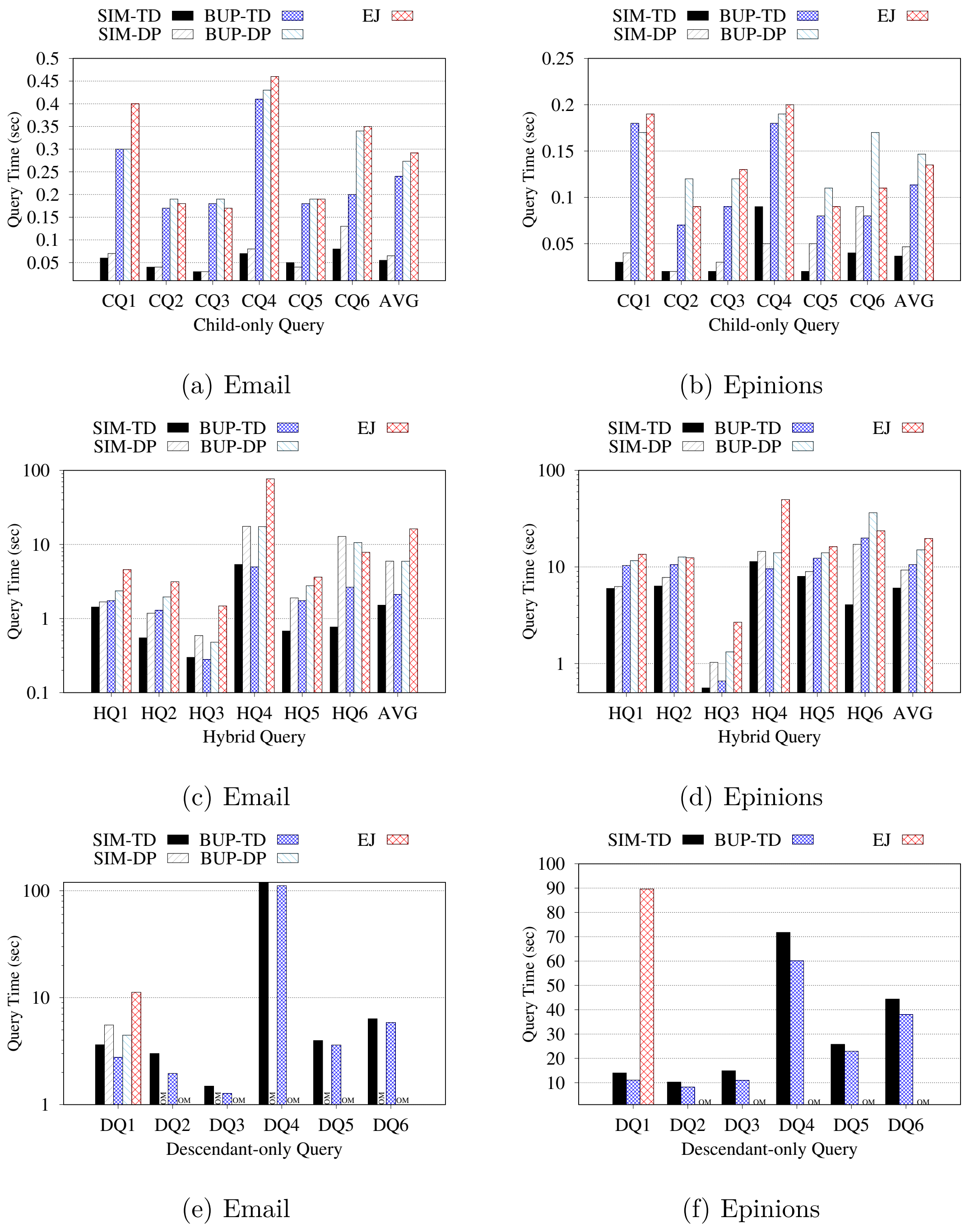


Figure 3.8 Evaluating different types of queries on Email and Epinions.

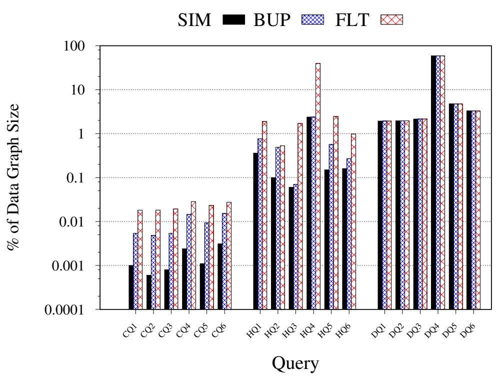
Epinions. Pre-filtering is applied in all cases except for the SIM algorithm on query patterns with direct-only edges where it is not beneficial.

The general trends observed in Figure 3.8 are as follows: (1) As the percentage of reachability edges in a query increases, the query evaluation time of all the algorithms increases; and (2) TD is in general more efficient than DP on tuple enumeration, in particular for dense queries, both on BUP and SIM.

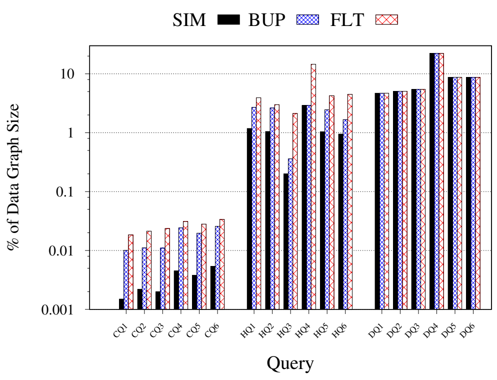
More specifically, in Figures 3.8(a) and 3.8(b), we observe that all the algorithms can solve child edge-only queries very efficiently, within 0.5 and 0.2 seconds per query on Email and Epinions, respectively. This can be explained by the fact that the child edge-only queries are usually sparse, with the total number of result tuples per query being less than 1.9K on each data graph. Algorithm SIM outperforms the other algorithms by far followed by BUP and then by EJ.

For hybrid queries, the best performer is again SIM, and then BUP and EJ follow in this order. As shown in Figures 3.8(c) and 3.8(d), the max speedup of SIM-TD over EJ is around 13× (for HQ_4) and 5× (for HQ_6) on Email and Epinions, respectively. BUP-TD outperforms EJ on average elapsed time by around 7× and 90% on Email and Epinions, respectively. The average speedup of SIM-TD over SIM-DP is around 3× and 50% on Email and Epinions, respectively, while the average speedup of BUP-TD over BUP-DP is around 2× and 40% on Email and Epinions, respectively.

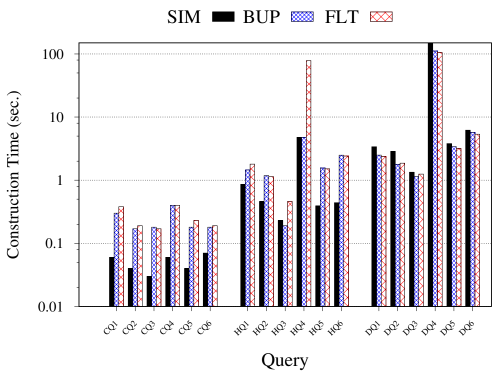
For reachability edge-only queries, BUP-TD slightly outperforms SIM-TD, in general, but the performance gap between SIM-TD and EJ is much more prominent (Figures 3.8(e) and 3.8(f)). The average elapsed time is not shown in figures 3.8(e) and 3.8(f) as most queries are unsolved by some algorithms. EJ solves only DQ_1 on Email and Epinions, and is unable to finish the rest of the queries due to an out-of-memory error. The enumeration method DP performs bad as well. Like EJ, both SIM-DP and BUP-DP solve only DQ_1 on Email, and are unable to solve any query on Epinions due to an out-of-memory error. The main reason is that as the reachability-edge-only queries are very dense, both Algorithm EJ and the DP method have to generate a large number of intermediate results which exceed the allocated memory.



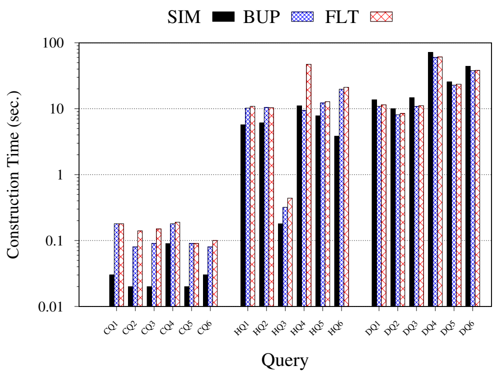
(a) Email



(b) Epinions



(c) Email



(d) Epinions

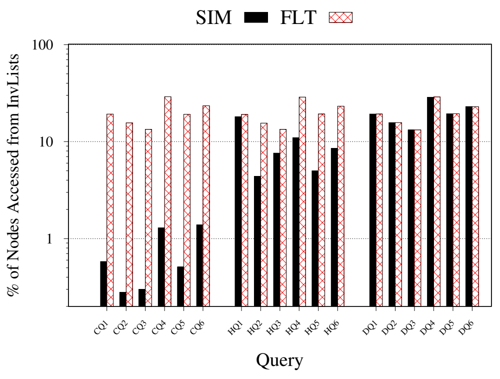
Figure 3.9 Size and construction time of (candidate) answer graphs by different algorithms on Email and Epinions.

Answer Graph Size and Construction Time. In this experiment, we examine the size and construction time of the answer graphs for different queries on Email and Epinions. As usual, the size of a graph is measured by the total number of nodes and edges. We compare three different algorithms. For the first algorithm, SIM, we have shown in Subsection 3.4.1 that the double simulation produced by SIM for a given query contains *exactly* the nodes appearing in the query answer. For the second algorithm, BUP, recall also that the graph built by BUP may contain redundant nodes (nodes not appearing in the query answer). We call this graph a *candidate answer graph*. For comparison with the candidate answer graph built by PJ, we used a process which first runs the node pre-filtering procedure to prune nodes from the inverted lists, then builds a candidate answer graph based on the pruned inverted lists in a top-down manner. This process, the third algorithm, is denoted as FLT. Figure 3.9 reports on the results. The Y-axis of Figures 3.9(a) and 3.9(b) shows the (candidate) answer graph size as a percentage of the input data graph size. Figures 3.9(c) and 3.9(d) report on the construction time of (candidate) answer graphs by SIM, BUP and FLT on Email and Epinions, respectively. The reported time for each query includes the time for node pre-filtering which is applied before the execution of the graph construction algorithms on hybrid and reachability-edge only queries.

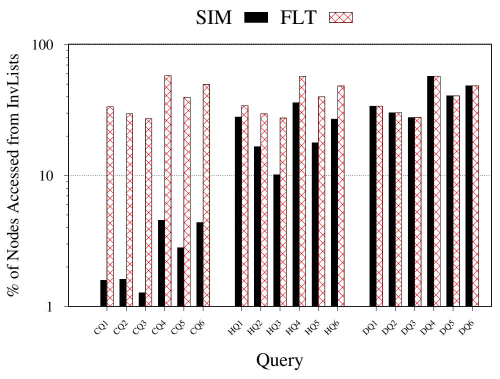
By design, for a given query, SIM always generates the smallest graph, followed by BUP, then FLT which generally produces the largest graph among the three. This is confirmed by the results shown in Figures 3.9(a) and 3.9(b) The size differences are more prominent on direct-edge-only and hybrid queries. We also observe that the sizes of graphs generated by each method increase significantly when the percentage of reachability edges in a query increases. In all cases, the size of (candidate) answer graphs is significantly below the theoretical bound of $|Q| \times |I_{max}|^2$ (cf. Section 3.2). The results also confirm that the answer graph size is affected by the corresponding query density.

In Figures 3.9(c) and 3.9(d), we observe that for reachability-edge-only queries, the graph construction time of BUP is slightly faster than that of FLT, while SIM takes a little bit longer than the other two, due to a small overhead in computing double simulation. However, for hybrid queries and reachability-edge only queries, SIM takes the smallest time on constructing answer graphs, outperforming BUP and FLT (on hybrid queries by up to 4.6× and 15×, respectively). This demonstrates the efficiency of the double simulation technique on pruning redundant nodes.

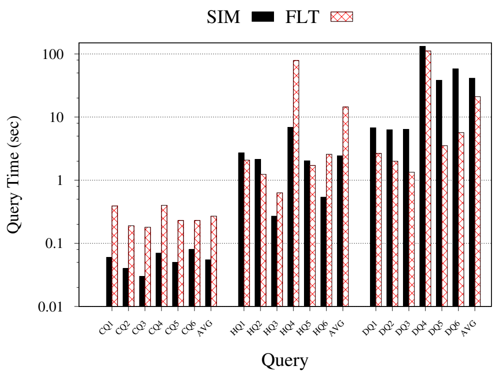
Pre-filtering vs. Simulation. In this experiment, we compare the pruning power of the two node pruning techniques, namely double simulation and pre-filtering, and



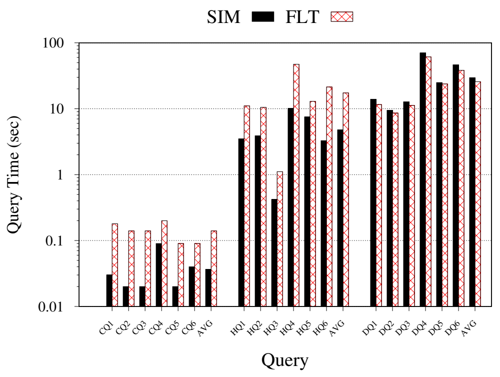
(a) Email



(b) Epinions



(c) Email



(d) Epinions

Figure 3.10 Comparison of pruning techniques SIM and FLT on Email and Epinions.

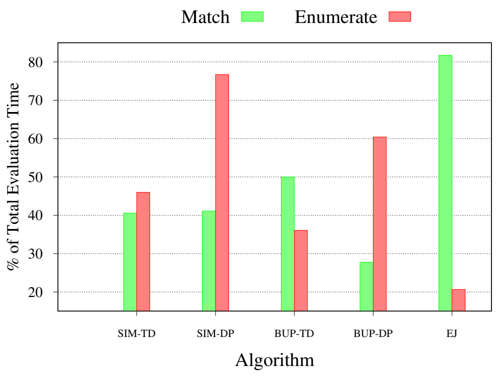
examine their effect on the graph matching performance. To this end, we run the procedures for node pre-filtering and double simulation to prune nodes from the inverted lists used by the queries. After the filtering, both procedures proceed in the same way, building a (candidate) answer graph based on the pruned inverted lists in a top-down manner. As in the previous experiments, they are referred to as FLT and SIM, respectively. Figure 3.10 reports on the results.

Figures 3.10(a) and 3.10(b) show the percentage of nodes from the inverted lists left for evaluating each query after the pruning processes of SIM and FLT are applied to the input inverted lists of Email and Epinions, respectively. We observe that for direct-only (resp. hybrid) queries, FLT leaves up to 54× (resp. 3×) and 20×

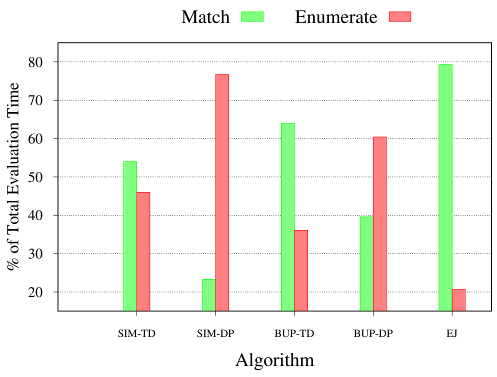
(resp. 2×) more nodes than SIM on Email and Epinions, respectively. This confirms that the node pruning power of double simulation is significantly higher than that of pre-filtering, for direct edge-only and hybrid queries. Both procedures leave almost the same number of nodes for evaluating reachability edge-only queries. The reason for the inferior performance of pre-filtering is that as it is designed to prune nodes for reachability edge-only queries, it misses nodes violating the child relationships in the data graph.

Figures 3.10(c) and 3.10(d) report on the elapsed time of the two algorithms on evaluating different types of queries on Email and Epinions, respectively. We observe that for direct edge-only (resp. hybrid) queries, SIM outperforms FLT by around 4× (resp. 5×) and 2.8× (resp. 2.6×) on the average on Email and Epinions, respectively. On reachability-only queries, FLT outperforms SIM by about 0.96× and 0.15× on Email and Epinions, respectively. This is due to the fact that the pre-filtering procedure identifies redundant nodes on any type of query by traversing the input data graph two times and does not require a reachability index. In contrast, the method based on double simulation needs a reachability index to check the connectivity between two data graph nodes whose corresponding query nodes are linked with a reachability edge. A reachability index like the BFL indexing scheme we used in our implementation might require traversing the data graph. Hence double simulation may incur more overhead than pre-filtering in identifying redundant nodes when queries with reachability edges are involved. However, the double simulation method has higher pruning power for direct edge-only and hybrid queries and this speeds up the answer graph construction process for this type of queries. This explains the time differences between SIM and FLT for query evaluation.

Figures 3.10(c) and 3.10(d) report on the elapsed time of the two algorithms on evaluating different types of queries on Email and Epinions, respectively. We observe that for direct edge-only (resp. hybrid) queries, SIM outperforms FLT by



(a) Email



(b) Epinions

Figure 3.11 Percentage of elapsed time of matching and enumeration over total time evaluating hybrid queries on Email and Epinions.

around 4× (resp. 5×) and 2.8× (resp. 2.6×) on the average on Email and Epinions, respectively. On reachability-only queries, FLT outperforms SIM by about 0.96× and 0.15× on Email and Epinions, respectively. This is due to the fact that the pre-filtering procedure identifies redundant nodes on any type of query by traversing the input data graph two times and does not require a reachability index. In contrast, the method based on double simulation needs a reachability index to check the connectivity between two data graph nodes whose corresponding query nodes are linked with a reachability edge. A reachability index like the BFL indexing scheme we used in our implementation might require traversing the data graph. Hence double simulation may incur more overhead than pre-filtering in identifying redundant nodes when queries with reachability edges are involved. However, the double simulation method has higher pruning power for direct edge-only and hybrid queries and this speeds up the answer graph construction process for this type of queries. This explains the time differences between SIM and FLT for query evaluation.

Matching Time vs. Result Tuple Enumeration time. In this experiment, we evaluate the six hybrid queries on Email and Epinions using the five algorithms SIM-TD, SIM-DP, BUP-TD, BUP-DP, and EJ. Node prefiltering was applied in all cases. Unlike previous experiments, we did not limit the query evaluation time by restricting the number of result tuples returned. We break down the evaluation time of each query into two parts: the matching time (pruning inverted lists, constructing the candidate answer graph and generating query plans—the latter for EJ only) and the result tuple enumeration time, and compare their respective ratio over the total evaluation time. Figure 3.11 reports on the the average matching time and the average enumeration time for each algorithm under comparison. Note that not all the six queries can be solved by EJ, BUP-DP, and SIM-DP due to an out-of-memory error. For instance, on Email, EJ failed on HQ_1, HQ_4 and HQ_6.

A general trend in Figure 3.11 is as follows: (1) on all the queries that EJ can solve, EJ spends most of its time on the matching phase, which computes matches for each query edge; and (2) both SIM and BUP generally take more time on the enumeration phase when using the DP method.

Comparison with Neo4j. In this experiment, we compared SIM-TD with the most popular graph DBMS Neo4j on evaluating pattern queries. Cypher, Neo4j's query language, uses patterns to match desired graph structures. However, Cypher does not offer node reachability semantics for reachability edges in a query pattern. Instead, Cypher adopts path finding semantics: consider a query pattern Q with a reachability edge (p,q). Cypher finds not only the matches of Q but also all the paths from the image of p to the image of q in each query match. This, of course, penalizes Cypher in pattern matching compared to algorithms that adopt node reachability semantics. For a fair comparison of the pattern evaluation times, we implement the node reachability semantics in Cypher using the Neo4j APOC (Awesome Procedures On Cypher) procedures. An APOC procedure expands to subgraph nodes reachable from the start node following relationships to max-level adhering to specified label filters [66].

Like EJ, Neo4j uses binary joins to evaluate pattern queries. We used Neo4j v.4.2.1 and expressed the queries of the aforementioned three query sets (see Section 3.5.1) in Cypher. Both SIM-TD and Neo4j were configured to find all the query matchings. The timeout was set to be 1 hour.

Table 3.4 shows the results on the Email graph. We denote by CQ_i, DQ_i, and HQ_i, direct-edge-only, reachability-edge-only, and hybrid queries, respectively (Figure 3.4). As Neo4j was unable to solve any hybrid or reachability edge-only query within 1 hour on the original Email graph, we used only 30k-sized and 3k-sized Email graphs, respectively, to compare the query time of the two approaches on evaluating these types of queries. As one can see in Table 3.4, SIM-TD is significantly faster than Neo4j, even by orders of magnitude in many cases. This result is even more pronounced for dense queries or queries with reachability relationship constraints. This clearly demonstrates the advantage of our proposed new framework for evaluating different types of pattern queries on graphs as well as the need to extend Neo4j with efficient reachability query evaluation support [67]. We nevertheless note that Neo4j is a full-fledged graph database system with many functions that our prototype implementation does not support, and this may also account for the performance gap between our framework and Neo4j.

Table 3.4 Runtime (sec) of SIM-TD and Neo4j Evaluating Different Types of Queries on the Email Graph

		CQ_1	CQ_2	CQ_3	CQ_4	CQ_5	CQ_6
Email	SIM-TD	0.05	0.09	0.04	0.07	0.06	0.07
($\|V\|$=265k)	Neo4j	0.32	0.35	0.56	1.02	0.33	12.35
		HQ_1	HQ_2	HQ_3	HQ_4	HQ_5	HQ_6
Email	SIM-TD	0.33	0.1	0.12	0.51	0.08	1.2
($\|V\|$=30k)	Neo4j	83	5.68	37.6	738.3	72.1	793
		DQ_1	DQ_2	DQ_3	DQ_4	DQ_5	DQ_6
Email	SIM-TD	0.06	0.16	0.32	0.16	0.52	1.83
($\|V\|$=3k)	Neo4j	11.74	126.9	677	656.6	1779	>1 hour

CHAPTER 4

EVALUATING HYBRID GRAPH PATTERN QUERIES USING SUMMARY GRAPHS

In Chapter 3, we considered tree pattern queries on large data graphs. In Chapter 4, we extend our approaches for tree pattern queries to graph pattern queries on large data graphs, with the difference between a tree and a graph being that in a graph, there is no unique node designated as the root, thus allowing for loop edges. Given a large directed graph G and a pattern query Q, our goal is to efficiently find the answer of Q on G. We present a novel approach for efficiently finding homomorphic matches for hybrid graph patterns, where each pattern edge may be mapped either to an edge or to a path in the input data, thus allowing for higher expressiveness and flexibility in query formulation. A key component of our approach is a lightweight index structure that leverages graph simulation to compactly encode the query answer search space. The index can be built on-the-fly during query execution and does not have to persist on the disk. Using the index, we design a multi-way join algorithm to enumerate query solutions without generating an exploding number of intermediate results.

4.1 Preliminaries and Problem Definition

We consider graph pattern queries that involve direct and/or reachability edges. The concepts of data graphs, inverted lists, node reachability, homomorphism, answer, and occurrence provided in Chapter 3, Section 3.1, are extended to graph pattern queries.

Definition 4.1.1 (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 *direct* (resp. *reachability*) edge denotes a child (resp. descendant) structural relationship between

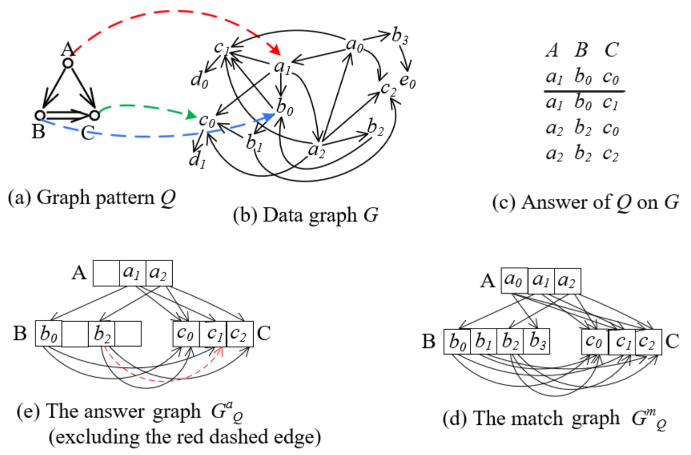


Figure 4.1 A hybrid graph pattern query Q, a data graph G, a homomorphism from Q to G, the answer of Q on G, and two summary graphs of Q on G.

the respective two nodes. A graph pattern that contains both direct and reachability edges is a *hybrid* graph pattern.

Definition 4.1.2 (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 direct edge, $(h(x), h(y))$ is an edge of G, while if (x, y) is a reachability edge, $h(x) \prec h(y)$ in G.

Figure 4.1(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 direct edges, are mapped by h to an edge in G. The other edges of Q which are reachability edges are mapped by h to a path of edges in G (which can possibly consist of just a single edge).

4.2 A Lightweight Index as Compact Search Space

4.2.1 Summary Graph

Given a pattern query Q and a data graph G, we propose the concept of a *summary graph* to serve as a search space of the answer of Q on G. The summary graph differs from the answer graph concept presented in Chapter 3 in that the answer graph is the smallest possible summary graph, such that the answer graph only contains edges and nodes which participate in the answer, while the more general concept of the summary graph may contain nodes and edges which do not participate in the answer.

Match Sets. 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 reachability edge, while (b) (u, v) is an edge in G if e is a direct 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 reachability edge), or the adjacency lists for the nodes of G (if e is a direct edge).

Definition 4.2.1 (Summary Graph)**.** A *summary graph* of pattern query Q over data graph G is a k-partite graph G_Q where k is the number of nodes in Q. For every node $q \in Q$, graph G_Q has an independent node set, denoted $cos(q)$, such that $os(q) \subseteq cos(q) \subseteq ms(q)$. Set $cos(q)$ is called the *candidate occurrence set* of q in G_Q. For every edge $e_Q = (p, q)$ in Q, graph G_Q has a set $cos(e_Q)$ of edges from nodes in $cos(p)$ to nodes in $cos(q)$ such that $os(e_Q) \subseteq cos(e_Q) \subseteq ms(e_Q)$. Set $cos(e_Q)$ is called the *candidate occurrence set* of e_Q in G_Q.

By definition, we can have many summary graphs of a given query Q on data graph G. Among them, the largest one is called the *match* summary graph of Q on G, denoted as G_Q^m, and the smallest one is called the *answer* summary graph of Q on G, denoted as G_Q^a. The *answer* summary graph can also be called the answer graph, as it is referred to in Chapter 3. 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 $ms(e)$. Figures 4.1(d) and (e), respectively, show the match summary graph and the answer summary graph for query Q on the data graph G in Figure 4.1(b).

A summary graph G_Q losslessly summarizes all the occurrences of Q on G as shown by the proposition below.

Proposition 1. Let G_Q be a summary graph of a pattern query Q over a data graph G. Assume that there is a homomorphism from Q to G which maps nodes p and q of Q to nodes v_p and v_q, respectively, of G. Then, if (p, q) is an edge in Q, (v_p, v_q) is an edge of G_Q.

By Proposition 1, G_Q encodes all the homomorphisms from Q to G. Thus, it represents a search space of the answer of Q on G. Similarly to factorized representations of query results studied in the context of classical databases and probabilistic databases [59], 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. We later present an algorithm for enumerating the results of Q on G from a summary graph G_Q.

Summary graph vs. other query related auxiliary data structures. A number of recent graph pattern matching algorithms also use query related auxiliary data structures to represent the query answer search space [41, 29, 30, 31, 7, 49]. These auxiliary data structures are designed to support searching for (an extension of) graph simulation [41] or subgraph isomorphisms [29, 30, 31, 7, 49]. Unlike summary graph, they are subgraphs of the data graph, hence they do not contain reachability information between data nodes, and consequently, they are not capable of compactly encoding edge-to-path homomorphic matches.

4.2.2 Refining a Summary Graph using Double Simulation

A summary graph G_Q can contain redundant nodes and edges, that is, nodes and edges that are not in the query answer. To further reduce the query answer search space, we would like to refine a summary graph G_Q as much as possible by pruning redundant nodes and edges. Ideally, we would like to build the answer summary graph G_Q^a before computing the query answer. However, when Q is a graph which is not a tree, finding G_Q^a is a NP-hard problem.

Most existing data node filtering methods are either simply based on query node labels [13, 11], or apply an approximate subgraph isomorphism algorithm [33] on query edge matches, or use one or more subtrees of the query to filter out data nodes violating children or parent structural constraints of subtrees [31, 7, 10]. They are unable to prune nodes violating ancestor/descendant structural constraints of the input query. While the recent node pre-filtering method [68] can prune nodes violating ancestor/descendant structural constraints, it is unable to prune data nodes violating children or parent structural constraints. Moreover, that pruning technique does not capture the specific structure among those ancestors and descendants.

Inspired by the graph simulation technique used in [69, 63] which constructs a covering index for queries over graph data, we propose to leverage an extension of traditional graph simulation [39] to construct a refined summary graph. The refined summary graph can serve as a compact search space for queries over graphs.

Double simulation. In Chapter 3, we discussed double simulation in the context of tree pattern queries; now, we will extend double simulation to graph pattern queries. This definition for double simulation is mostly similar to the definition previously defined, with slight modifications.

Definition 4.2.2 (Double Simulation)**.** The *double simulation* $\mathcal{FB}$ of a graph pattern query $Q = (V_Q, E_Q)$ by a directed data graph $G = (V, E)$ is the largest binary relation $S \subseteq V_Q \times V$ such that, whenever $(q, v) \in S$, the following conditions hold:

Table 4.1 Forward ($\mathcal{F}$), Backward ($\mathcal{B}$), and Double ($\mathcal{FB}$) Simulation of the Query Q on the Graph G of Figure 4.1.

q	$\mathcal{F}(q)$	$\mathcal{B}(q)$	$\mathcal{FB}(q)$
A	$\{a_1, a_2\}$	$\{a_0, a_1, a_2\}$	$\{a_1, a_2\}$
B	$\{b_0, b_1, b_2\}$	$\{b_0, b_2, b_3\}$	$\{b_0, b_2\}$
C	$\{c_0, c_1, c_2\}$	$\{c_0, c_1, c_2\}$	$\{c_0, c_1, c_2\}$

1. $label(q) = label(v)$.

2. For each edge $e_Q = (q, q') \in E_Q$, there exists $v' \in V$ such that $(q', v') \in S$ and $(v, v') \in ms(e_Q)$.

3. For each edge $e_Q = (q', q) \in E_Q$, there exists $v' \in V$ such that $(q', v') \in S$ and $(v', v) \in ms(e_Q)$.

For $q \in V_Q$, let $\mathcal{FB}(q)$ denote the set of all nodes of V that double simulate q. In Section 4.2.5, we will show how to use $\mathcal{FB}$ to construct a refined summary graph of Q on G.

The double simulation of Q by G is unique, since there is exactly one largest binary relation S satisfying the three conditions of Definition 4.2.2. This can be proved by the fact that, whenever two binary relations S_1 and S_2 satisfy the three conditions, their union $S_1 \cup S_2$ also satisfies these conditions.

We call the largest binary relation which satisfies the conditions 1 and 2 of Definition 4.2.2 above *forward simulation* of Q by G, while the largest binary relation which satisfies conditions 1 and 3 of Definition 4.2.2 above is called *backward simulation*. While the double simulation preserves both incoming and outgoing edge types (direct or reachability) between Q and G, the forward and the backward simulation preserve only outgoing and incoming edge types, respectively.

Table 4.1 shows the simulations $\mathcal{F}$, $\mathcal{B}$, and $\mathcal{FB}$ of the query Q on the graph G of Figure 4.1. In particular, for the reachability query edge (B, C) of Q, the matches considered for double simulation are (b_0, c_0), (b_0, c_1), (b_1, c_0), (b_1, c_2), (b_2, c_0), (b_2, c_1), (b_2, c_2).

4.2.3 A Basic Algorithm for Computing Double Simulation

To compute $\mathcal{FB}$, we present first a basic algorithm called *FBSimBas* (Algorithm 7). Algorithm *FBSimBas* is based on an extension of a naive evaluation strategy originally designed for comparing graphs of unknown sizes [39, 43]. While the original method works for edge-to-edge mappings between the given two graphs, *FBSimBas* allows edge-to-path mappings from a reachability edge in the pattern graph to a path in the data graph.

Given a query Q and a data graph G, *FBSimBas* implements the following strategy: starting with the largest possible relation between the node sets of Q and G, it incrementally disqualifies pairs of nodes violating the conditions of Definition 4.2.2. The process terminates when no more node pairs can be disqualified.

More concretely, *FBSimBas* works as follows. Let FB be an array structure indexed by the nodes of Q. The algorithm initializes FB by setting $FB(q)$ to be equal to the match set $ms(q)$ of q, for each $q \in V_Q$. The main process consists of two procedures which iterate on the edges of Q and check the conditions of Definition 4.2.2 in different directions. The first procedure, called *forwardPrune*, checks the satisfaction of the forward condition in Definition 4.2.2 by visiting each edge $e_Q = (q_i, q_j) \in E_Q$ from the tail node q_i to the head node q_j. Specifically, *forwardPrune* removes each v_{q_i} from $FB(q_i)$ if there exists no $v_j \in FB(q_j)$ such that (v_i, v_j) is in $ms(e_Q)$. The second procedure, called *backwardPrune*, checks the satisfaction of the backward condition in Definition 4.2.2 by visiting each edge in the opposite direction. The above process is repeated until FB becomes stable, i.e., no more changes can be made to FB.

The table of Figure 4.2(b) shows the node pruning steps performed by Algorithm *FBSimBas* for the query Q of Figure 4.1(a) on the graph G_2. We assume that the

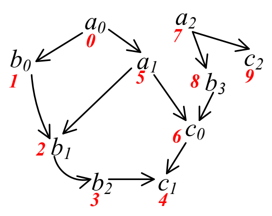
Algorithm 7: Algorithm *FBSimBas* for computing double simulation.

```
Input  : Data graph $G$, pattern query $Q$
Output: Double simulation $\mathcal{FB}$ of $Q$ by $G$

1  Let $FB$ be an array indexed by the nodes of $Q$ ;
2  Initialize $FB(q)$ to be $ms(q)$ for every node $q$ in $V_Q$ ;
3  repeat
4      forwardPrune() ;
5      backwardPrune() ;
6  until $FB$ has no changes;
7  return $FB$ ;

8  Procedure forwardPrune():
9      for (each edge $e_Q = (q_i, q_j) \in E_Q$ and each node $v_{q_i} \in FB(q_i)$) do
10         if (there is no $v_{q_j} \in FB(q_j)$ such that $(v_{q_i}, v_{q_j}) \in ms(e_Q)$) then
11             delete $v_{q_i}$ from $FB(q_i)$ ;
12         end
13     end

14 Procedure backwardPrune():
15     for (each edge $e_Q = (q_i, q_j) \in E_Q$ and each node $v_{q_j} \in FB(q_j)$) do
16         if (there is no $v_{q_i} \in FB(q_i)$ such that $(v_{q_i}, v_{q_j}) \in ms(e_Q)$) then
17             delete $v_{q_j}$ from $FB(q_j)$ ;
18         end
19     end
```



(a) Data graph G_2

Step	$FB(A)$			$FB(B)$				$FB(C)$		
	a_0	a_1	a_2	b_0	b_1	b_2	b_3	c_0	c_1	c_2
1	×									
2				×					×	×
3			×		×	×				
4							×			
5		×								
6								×		

(b) *FBSimBas*

Step	$cs(A)$			$cs(B)$				$cs(C)$		
	a_0	a_1	a_2	b_0	b_1	b_2	b_3	c_0	c_1	c_2
1	×									
2				×					×	×
3		×	×		×	×				
4							×	×		

(c) *FBSimDag*

Figure 4.2 Node pruning of *FBSimBas* and *FBSimDag* for the query Q of Figure 4.1(a) on the graph G_2.

edges of Q are considered in the order: $(A, B), (A, C)$, and (B, C). The first column shows the step number. Odd numbers correspond to Procedure *forwardPrune* while even numbers correspond to Procedure *backwardPrune*. The other three columns show the nodes pruned at each step from the candidate FB sets for the query nodes A, B, and C. An '×' symbol indicates that the corresponding node is pruned. Notice that Q has an empty answer on G. Algorithm *FBSimBas* detects and prunes all the redundant nodes and hence Q has an empty summary graph.

4.2.4 Efficiently Computing Double Simulation by Exploiting the Pattern Structure

Recall that *FBSimBas* picks an arbitrary order to process/evaluate the query edges. It has been shown in [40], and also verified by our experimental study, that the order in which the edges are evaluated has an impact on the overall runtime similar to the impact of join order on join query evaluation. We would like to explore the pattern structure to design a more efficient algorithm for computing relation $\mathcal{FB}$ for Q by G, which not only converges faster because of a reduced number of iteration passes of *FBSimBas* but also reduces the computation cost. In order to do so, we first describe an algorithm for computing $\mathcal{FB}$ for dag pattern queries.

An algorithm for computing $\mathcal{FB}$ for dag patterns. Leveraging the acyclic nature of the dag query pattern, we develop a multi-pass algorithm called *FBSimDag* which is based on dynamic programming. As with *FBSimBas*, FB is initially set to be the largest possible relation between the nodes sets V_Q and V. Unlike *FBSimBas* which visits each edge of Q in two directions in each pass, *FBSimDag* traverses nodes of Q by their topological order two times, first bottom-up (reverse topological order) and then top-down (forward topological order). During each traversal, nodes of G violating the conditions of Definition 4.2.2 are removed from FB. As we will show later, the bottom-up traversal computes a forward simulation of Q by G, while the top-down traversal computes a backward simulation of Q by G. In contrast, *FBSimBas* traverses pattern edges in an arbitrary order. The algorithm terminates when no more nodes can be removed from FB.

Algorithm 8 shows the pseudocode of *FBSimDag*. The algorithm first invokes procedure *forwardSim* to check for nodes $v_q \in FB(q)$ which satisfy the forward simulation condition. Procedure *forwardSim* considers outgoing query edges of q by traversing nodes of Q in a bottom-up way. When $q \in V_Q$ is a sink node in Q, v_q trivially satisfies the forward simulation condition. Otherwise, if edge $e_Q = (q, q_i) \in E_Q$ but there is no $v_i \in FB(q_i)$ such that (v, v_i) is in $ms(e_Q)$, v_q is removed from $FB(q)$.

When the bottom-up traversal terminates, *FBSimDag* proceeds to do a top-down traversal of Q using procedure *backwardSim*. This procedure checks whether nodes $v_q \in FB(q)$ satisfy the backward simulation condition by considering incoming query edges of q. When $q \in V_Q$ is a source node in Q, v_q trivially satisfies the

Algorithm 8: Algorithm *FBSimDag* for computing double simulation.

```
Input  : Data graph $G$, dag pattern query $Q$
Output: Double simulation $\mathcal{FB}$ of $Q$ by $G$

1  Lines 1-2 in Algorithm FBSimBas ;
2  repeat
3      forwardSim() ;
4      backwardSim() ;
5  until $FB$ has no changes;
6  return $FB$ ;

7  Procedure forwardSim():
8      for (each $q \in V_Q$ in a reverse topological order and each $v_q \in FB(q)$) do
9          if ($\exists e_Q = (q, q_i) \in E_Q$ s.t. for no $v_{q_i} \in FB(q_i)$, $(v_q, v_{q_i}) \in ms(e_Q)$)
            then
10             delete $v_q$ from $FB(q)$ ;
11         end
12     end

13 Procedure backwardSim():
14     for (each $q \in V_Q$ in a topological order and each $v_q \in FB(q)$) do
15         if ($\exists e_Q = (q_i, q) \in E_Q$ s.t. for no $v_{q_i} \in FB(q_i)$, $(v_{q_i}, v_q) \in ms(e_Q)$)
            then
16             delete $v_q$ from $FB(q)$ ;
17         end
18     end
```

backward simulation condition. Otherwise, if edge $e_Q = (q_i, q) \in E_Q$ but there is no $v_i \in \mathcal{FB}(q_i)$ such that (v_i, v) is in $ms(e_Q)$, v_q is removed from $FB(q)$.

The above process is repeated until FB stabilizes, i.e., no $FB(q)$, $q \in V_Q$, can be further reduced. When Q is a tree pattern, a single pass is sufficient for FB to stabilize [44].

The main difference of the two algorithms is that *FBSimDag* considers query nodes in a (forward and backward) topological order, whereas *FBSimBas* considers query nodes in an arbitrary order. The table of Figure 4.2(c) shows the node pruning steps performed by Algorithm *FBSimDag* for the query Q of Figure 4.1(a) on the graph G_2. Comparing the table with that of Figure 4.2(a), one can see that it takes *FBSimDag* fewer steps than *FBSimBas* to converge.

Dag+Δ: an efficient $\mathcal{FB}$ algorithm. Based on *FBSimDag*, we design a new $\mathcal{FB}$ algorithm called *FBSim*. The algorithm first decomposes the input graph pattern Q into a dag Q_{dag} and a set E_{bac} of back edges (Δ). The main body of the algorithm has two phases: it first calls *FBSimDag* to compute *FB* on Q_{dag}. After that, it calls *FBSimBas* on E_{bac} to update *FB*. The above process is repeated until *FB* becomes stable.

4.2.5 Efficiently Building the Refined Summary Graph

We now present Algorithm *BuildSummaryGraph* (Algorithm 9) for building a refined summary graph in two phases: in the *node selection* phase (line 1), all the summary graph nodes are obtained by pruning redundant data nodes. This is achieved by computing the double simulation relation. In the *node expansion* phase (lines 2-3), the summary graph nodes are expanded with incident edges to construct the final summary graph graph. During the summary graph construction, once node $v_q \in cos(q)$ has been expanded, the outgoing and incoming edges of v_q are indexed by the parents and children of query node q. This allows efficient intersection operations of adjacency lists of selected nodes in the summary graph graph. These efficient

Algorithm 9: Algorithm *BuildSummaryGraph* for building a refined summary graph

```
Input  : Data graph $G$, pattern query $Q$
Output: Summary Graph $G_Q$ of $Q$ on $G$

1  select() ;
2  for (each edge $(q_i, q_j) \in E_Q$) do
3      expand($q_i$, $q_j$) ;
4  end
5  return $G_Q$ ;

6  Procedure select():
7      Use Algorithm FBSimBas or FBSim to compute $\mathcal{FB}$ of $Q$ by $G$ ;
8      Initialize $G_Q$ as a $k$-partite graph without edges having one
         independent set $cos(q)$ for every node $q \in V_Q$, where $cos(q) = \mathcal{FB}(q)$ ;

9  Procedure expand($p$, $q$):
10     for (each $v_p \in cos(p)$) do
11         for (each $v_q \in cos(q)$) do
12             if (($v_p, v_q) \in ms(e_Q)$, where $e_Q = (p, q) \in E_Q$) then
13                 Connect $v_p$ to $v_q$ with a directed edge ;
14             end
15         end
16     end
```

intersection operations are useful in the phase of query occurrence enumeration as we will show in Section 4.3.

As an example, consider building a refined summary graph for query Q on graph G in Figure 4.1 using Algorithm 9. After the first phase, we obtain the following $\mathcal{FB}$ relation: $FB(A) = \{a_1, a_2\}$, $FB(B) = \{b_0, b_2\}$ and $FB(C) = \{c_0, c_1, c_2\}$. The summary graph generated from the second phase is shown in Figure 4.1(e). The summary graph has one more edge than the answer summary graph (shown by a red dashed line), but it has fewer nodes and edges than the match summary graph (Figure 4.1(d)).

Speedup convergence for simulation computation. As described in Section 4.2.2, the computation of $\mathcal{FB}$ terminates only when no more nodes can be pruned from the candidate occurrence sets of the query nodes during the multi-pass process. This process can be costly since we need to repeatedly check the candidate occurrence sets of the query nodes. We describe below optimizations to speedup the convergence of the process.

First, if no change is made to candidate occurrences corresponding to a subquery of Q in the last pass, then the computation on that subquery for the current pass can be skipped. To achieve this, we associate with each query node q a flag indicating whether nodes were pruned from its candidate occurrence set $FB(q)$ during the last pass. The flags are consulted in the current pass to decide whether the computation can be skipped.

Second, as aforementioned, the node selection enforces the existence semantics. A data node v is retained in its candidate occurrence set as long as there exist nodes in the parent and child node lists that make v satisfy the conditions of Definition 4.2.2. Checking node v in the current pass can be skipped if the nodes guaranteeing its existence are not removed in the last pass. We therefore design an index on the nodes in the candidate occurrence sets of the query nodes. Specifically, the index records for each data node $v \in FB(q)$ of query node q those nodes in the candidate sets of q's parent and child nodes in Q that guarantee v's existence in $FB(q)$. The index structure is maintained throughout the multi-pass process.

4.3 A Multiway Intersection-based Enumeration Algorithm

We now present our graph pattern answer enumeration algorithm, called *MJoin*, which is shown in Algorithm 10.

High level idea. Given a query Q and data graph G, relation $cos(e)$ contains the candidate occurrences of query edge e on G. Conceptually, *MJoin* produces occurrences of Q by joining multiple such relations at the same time. Instead of using standard query plans that join one relation (i.e., query edge) at a time, *MJoin* considers a new style of multi-way join plans which join one join key (i.e., query node in graph terms) at a time. A query-node-at-a-time style join plan considers

Algorithm 10: Algorithm *MJoin*

```
Input  : Graph pattern query $Q$ and the summary graph $G_Q$ of $Q$ on $G$
Output: The answer of $Q$ on $G$.
1  Pick an order $q_1, \ldots, q_n$ for the nodes of $Q$, where $n = |V(Q)|$ ;
2  Let $t$ be a tuple where $t[i]$ is initialized to be $null$ for $i \in [1, n]$ ;
3  Enumerate(1, $t$) ;
4  Procedure Enumerate(index $i$, tuple $t$):
5      if ($i = |V(Q)| + 1$) then
6          return $t$;
7      end
8      $N_i := \{q_j \mid (q_i, q_j) \in E_Q \ or \ (q_j, q_i) \in E_Q, j \in [1, i-1]\}$ ;
9      $cos_i := cos(q_i)$ ;
10     for (every $q_j \in N_i$) do
11         $cos_i^j := \{v_i \in cos_i \mid (v_i, t[j]) \in E(G_Q) \ or \ (t[j], v_i) \in E(G_Q)\}$ ;
12         $cos_i := cos_i \cap cos_i^j$ ;
13     end
14     for (every node $v_i \in cos_i$) do
15         $t[i] := v_i$ ;
16         Enumerate($i + 1$, $t$) ;
17     end
```

only the distinct join key values if a specific join key value occurs in multiple tuples. Also, all joins are executed in a pipeline to avoid materializing intermediate results. Hence, it can avoid enumerating large intermediate results that typically occur with Selinger-style binary-joins (query-edge-at-a-time joins in graph terms) [70].

This new style multi-way joins are called worst case optimal joins [71] and have been exploited in recent graph matching algorithms [72, 13, 65]. The main difference between *MJoin* and those algorithms lies in the implementation of the new style multi-way joins. *MJoin* exploits the summary graph G_Q to perform multi-way joins. We show below how this can be done by multi-way intersecting node adjacency lists of G_Q.

The algorithm. Algorithm *MJoin* first picks a *search order* to search solutions. This is a linear order of the query nodes. A search order heavily influences the query evaluation performance. We will discuss how to choose a good search order later. Then, *MJoin* performs a recursive backtracking search to find occurrences of the query nodes iteratively, one at a time by the given order, before returning any query occurrences.

More concretely, let's assume that the chosen search order is $q_1, \ldots, q_n$. Let Q_i denote the subquery of Q induced by the nodes $q_1, \ldots, q_i$, $i \in [1, n]$. Algorithm *MJoin* calls a recursive function *enumerate* which searches for potential occurrences of a single query node q_i in each recursive step. The index i of the current query node is passed as a parameter to *enumerate*. When $i > 0$, the backtracking nature of *enumerate* entails that a specific occurrence for the subquery Q_{i-1} has already been considered in the previous recursive steps. The second parameter of *enumerate* is a tuple t of length n, where $t[1 : i]$ is an occurrence of Q_i. Initially, i is set to 0 and all the values of t are set to *null*.

At a given recursive step i, function *enumerate* first determines query nodes that have been considered in a previous recursive step and are adjacent to the current node q_i. These nodes are collected in the set N_i. Let cos_i be a node set of q_i in G_Q, initially set to be equal to $cos(q_i)$. To reduce the size of cos_i, for each $q_j \in N_i$, *enumerate* intersects cos_i with the forward adjacency list of $t[j]$ in G_Q when (q_i, q_j) is an edge of Q, or with the backward adjacency list of $t[j]$ when (q_j, q_i) is an edge of Q (lines 5-7). If after this process cos_i is not empty, function *enumerate* iterates over the nodes in cos_i (line 8). In every iteration step, a node of cos_i is assigned to $t[i]$ (line 9) and *enumerate* proceeds to the next recursive step (line 10). If cos_i is empty or all the nodes in cos_i have been considered, *enumerate* backtracks to the last matched query node q_{i-1}, reassigns an unmatched node (if any) from cos_{i-1} to $t[i-1]$, and recursively calls *enumerate*. In the final recursive step, when $i = n+1$, tuple t contains one specific occurrence for all the query nodes and is returned as an occurrence of Q (line 2).

In our running example, let G_Q be the refined summary graph, i.e., the graph of Figure 4.1(e) including the red dashed edge. Assume the search order of Q is

A, B, C. When $i = 1$, Algorithm *MJoin* first assigns a_1 from $cos(A)$ to tuple $t[1]$, then recursively calls *Enumerate*(2, t). The intersection of a_1's adjacency list with $cos(B)$ is $\{b_0\}$. Node b_0 is then assigned to $t[2]$. When $i = 3$, since the intersection of forward adjacency lists of a_1 and b_0 with $cos(D)$ is $\{c_0, c_1\}$, *MJoin* assigns c_0 and c_1 to $t[4]$, and returns two tuples $\{a_1, b_0, c_0\}$ and $\{a_1, b_0, c_2\}$ in that order. Then, *MJoin* backtracks till $i = 1$, assigns the next node a_2 from $cos(A)$ to $t[1]$ and proceeds in the same way. It finally returns another two tuples $\{a_2, b_2, c_0\}$ and $\{a_1, b_0, c_2\}$. Note that edge (b_2, c_1) (the red dashed edge in Figure 4.1(e)) is not filtered out by the double simulation pruning process and its redundancy is detected only after *MJoin* is executed.

Search order. A search order σ is a permutation of query nodes that is chosen for searching query solutions. The performance of a query evaluation algorithm is heavily influenced by the search order [73]. As the number of all possible search orders is exponential in the number of query nodes, it is expensive to enumerate all of them.

The search order σ for query Q is essentially a left-deep query plan [33]. The traditional dynamic programming technique would take $O(2^{|V_Q|})$ time to generate an optimized join order. This is not scalable to large graph patterns, as verified by our experimental evaluation in Section 4.4.

Therefore, we use a greedy method to find a search order for Q leveraging statistics of G_Q. Our greedy method is based on the join ordering strategy proposed in [33]. We refer to this method as *JO*. *JO* selects as a start node of σ a node q in V_Q with the smallest candidate occurrence set $cos(q)$ in G_Q among the nodes in V_Q. Subsequently, *JO* iteratively selects as the next node in σ a node q' of Q which satisfies the following two conditions: (a) q' is a new node adjacent in Q to some node in σ, and (b) $cos(q')$ is the smallest among all the nodes q' satisfying condition (a). The rationale here is to enforce connectivity in order to reduce unpromising intermediate results caused by redundant Cartesian products [30] as well as to minimize (estimated) join costs. Different from the original method which uses the cardinality of the inverted lists of the data graph G [33], *JO* uses the cardinality of the candidate occurrence sets of a refined summary graph G_Q, which provide a better cost estimation for generating an effective search order.

In our experiments, we also implemented a well known ordering method called *RI* [34]. Unlike *JO*, *RI* generates σ based purely on the topological structure of the given query, independently of any target data graph. The rationale of *RI* is to introduce as many edge constraints as possible and as early as possible in the ordering. Roughly speaking, vertices that are highly connected with vertices previously present in the ordering tend to come early in the final ordering. In our *enumerate* procedure, edge constraints will translate into intersection operations to produce candidate occurrence sets for the query nodes under consideration. Intuitively, the search order chosen by *RI* is likely to reduce the computation cost, since it tends to ensure the search space of *enumerate* would be reduced significantly after each iteration. We examine this intuition and compare the effectiveness of *RI* with *JO* for different workloads in the experiments.

4.4 Experimental Evaluation

We conduct extensive performance studies to evaluate the effectiveness and efficiency of our proposed summary graph-based graph pattern matching approach.

4.4.1 Experimental Setting

Setup. We compare the performance of our approach, abbreviated as *GM*, with the join-based approach (*JM*) [15, 13, 11], and the tree-based approach (*TM*) [26, 30, 31, 7]. Among all the existing algorithms in *JM* and *TM*, only the contributions [15] and [26] are capable of directly finding homomorphisms of hybrid graph pattern queries on data graphs. As the source code of [15] and [26] are not publicly available, we implemented the algorithms described in [15] and [26], abbreviated as *JM* and

TM, respectively, in the plots. For the TM approach, we implemented the recent algorithm for evaluating tree patterns on graphs described in [44], which has been shown to outperform other existing algorithms. In our implementation, we applied the node pre-filtering technique described in [22, 26] to both approaches, *JM* and *TM*.

For checking node reachability in the data graph, all three algorithms under comparison use a recent efficient indexing scheme, called *Bloom Filter Labeling* (BFL) [20], which was shown to greatly outperform most existing schemes [20].

Our implementation was coded in Java. All the experiments reported were performed on a 64-bit Linux workstation equipped with an Intel(R) Xeon(R) processor (3.5GHz) and 32GB RAM.

Datasets. We ran experiments on six real-world graph datasets from the Stanford Large Network Dataset Collection which have been used extensively in previous works [8, 31, 9, 65]. The datasets have different structural properties and come from a variety of application domain: biology, social networks, and communication networks. Table 4.2 lists the properties of the datasets.

Queries. For the three biology datasets *hu, hp* and *yt* in Table 4.2, we used randomly generated queries that were originally used in [9] for finding subgraph isomorphisms. We modified those queries by turning query edges with 50% probability into reachability edges. The number of nodes of the queries ranges from 4 to 20 for *hu*, and from 4 to 32 for *hp* and *yt*.

For the other three datasets of Table 4.2, we used designed queries. We generated 20 graph pattern query templates, shown in Figure 4.3. These query templates involve direct and reachability edges. They have various and complex structures. Instances (with only reachability or only direct edges) of many of them were used in previous work [15, 11]. The number associated with each node of a query template denotes the node id. Query instances are generated by assigning labels to nodes. We group the 20 query templates into four classes: acyclic, cyclic, clique, and

Table 4.2 Key Statistics of the Graph Datasets Used

Domain	Dataset	$\|V\|$	$\|E\|$	$\|L\|$	d_{avg}
Biology	Yeast (*yt*)	3.1K	12K	71	8.05
	Human (*hu*)	4.6K	86K	44	36.9
	HPRD (*hp*)	9.4K	35K	307	7.4
Social	Epinions (*ep*)	76K	509K	20	6.87
	DBLP (*db*)	317K	1049K	20	6.62
Communication	Email (*em*)	265K	420K	20	2.6

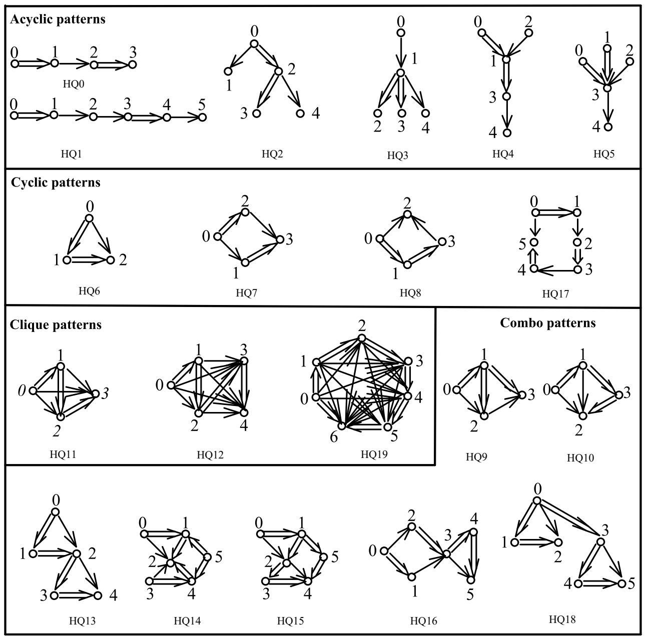


Figure 4.3 Categorized graph pattern queries.

combo patterns. We call a graph pattern query *acyclic* if its corresponding undirected graph is acyclic, and *cyclic* otherwise. A pattern is called *combo* if its undirected graph contains more than two cycles. A pattern is called *clique* if its undirected graph is complete.

Metrics. We measured the runtime of individual queries in a query set. For query listing, this includes two parts: (1) the matching time, which consists of the time spent on filtering vertices, building auxiliary data structures such as summary graphs (summary graphs), and generating query plans (or search order), and (2) the result enumeration time, which is the time spent on enumerating occurrences. The number of occurrences for a given query on a data graph can be quite large. Following usual practice [31, 10, 9], we terminated the evaluation of a query after finding a limited number of matches (this number was set to 10^7 in the experiments) covering as much of the search space as time allowed. We stopped the execution of a query if it did not complete within 10 minutes, so that the experiments could be completed in a reasonable amount of time. We recorded the elapsed time of these stopped queries as 10 minutes.

4.4.2 Time Performance

We run this experiment to compare the time performance of the three algorithms *JM, TM,* and *GM* on evaluating categorized query instances of pattern templates from Figure 4.3 as well as larger random graph pattern queries on realworld datasets.

Categorized graph patterns. Figure 4.4(a) and 4.4(b) shows the performance of *GM* against *JM* and *TM* for categorized graph pattern queries on *em* and *ep* data graphs, respectively. We omit the other bigger data graphs because *JM* and *TM* cannot compute the queries either due to an out-of-memory error or due to an extensively long execution time (hours). The figures show the results of three queries from each of the acyclic, cyclic, clique, and combo pattern classes.

The overall best performer is our approach *GM*, which outperforms *JM* and *TM* by up to two and three orders of magnitude, respectively. Both *TM* and *JM* were unable to solve all the queries either because of timeout or out-of-memory errors. In particular, *TM* has the worst performance on the acyclic query HQ_5 and the combo patterns. Unlike *GM* which evaluates the graph pattern directly, *TM* works by evaluating a tree query of the original graph query. For each tuple of the tree query, it checks the non-tree edges for satisfaction. Hence, its performance is enormously affected when the number of tree solutions is very large.

JM performs badly especially on cyclic patterns and clique patterns. It could not compute HQ_{14} on *em* because of an out-of-memory error (Figure 4.4(a)). For these types of queries, *JM* will typically perform a large amount of computations generating redundant intermediate results.

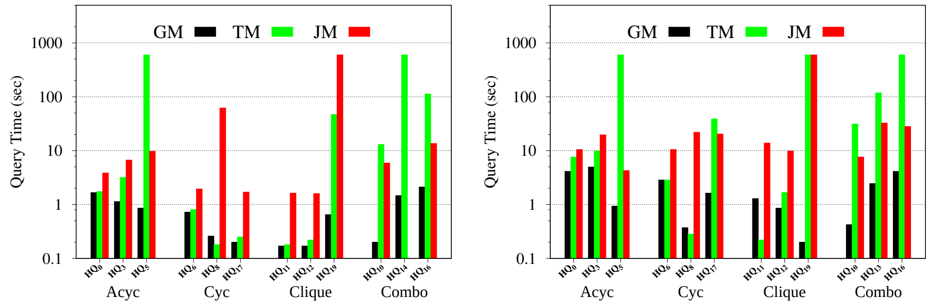
Larger graph patterns. Figure 4.5(a), 4.5(b), and 4.5(c) show the results of evaluating five random hybrid queries on the graphs *hp*, *yt* and *hu*, respectively. The x-axis represents the number of nodes of each query, and ranges from 4 to 32.

GM again significantly outperforms *TM* and *JM*. It is able to solve all the queries, whereas both *TM* and *JM* fail on several queries. *JM* had a high percentage of unresolved cases on queries with more than 10 nodes. In three cases, *JM* reports an out-of-memory error due to the large number of redundant intermediate results generated during the query evaluation. Another reason of the inefficiency of *JM* is the join plan selection. As described in [15], in order to select an optimized join plan, *JM* uses dynamic programming to exhaustively enumerate left-deep tree query plans. For queries with more than 10 nodes, the number of enumerated query plans can be huge.

TM has relatively good performance on *hp*, because it has small candidate tuples to compute; but it failed for more than half of the times on the dense dataset *hu* whose average node degree is 36.9 (Table 4.2).

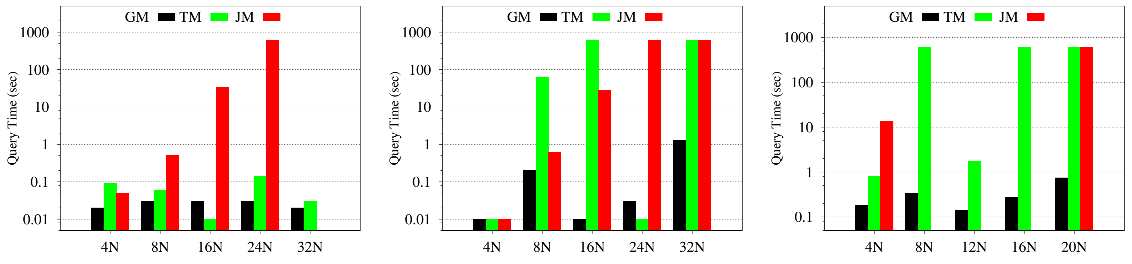
4.4.3 Scalability

Varying data labels. In this experiment, we examine the impact of the total number of distinct graph labels on the performance of the algorithms in comparison. We produced four versions of the *em* graph where the number of labels increases from 5 to 20 (the size of the graph is fixed). On these versions of *em*, we evaluated one set of 20 hybrid query instances of the query templates shown in Figure 4.3.



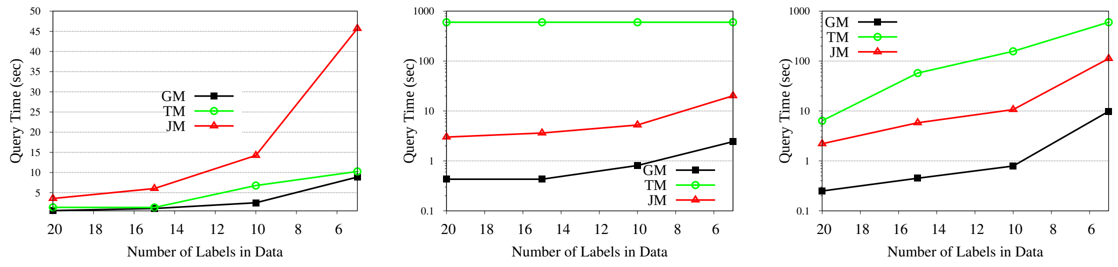
(a) *em* (b) *ep*

Figure 4.4 Performance comparison of *GM* with *TM* and *JM* using the categorized graph pattern queries.



(a) *hp* (b) *yt* (c) *hu*

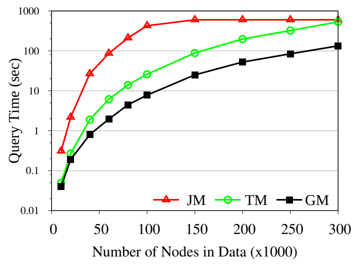
Figure 4.5 Performance comparison of *GM* with *TM* and *JM* using larger graph pattern queries.



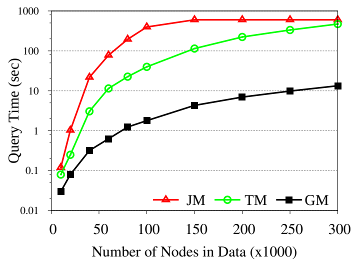
(a) HQ_2 (b) HQ_4 (c) HQ_7

Figure 4.6 Elapsed time of queries on *em* when increasing the number of labels.

Figure 4.6 reports on the query time of the three algorithms on the queries HQ_2, HQ_4, and HQ_7. The other queries gave similar results in our experiments. We observe that the execution time of the algorithms tends to increase while decreasing



(a) HQ_8



(b) HQ_{12}

Figure 4.7 Elapsed time on increasingly larger subsets of *dp*.

the total number of graph labels. This is reasonable since the average cardinality of the input label inverted lists in a graph increases when the number of distinct labels in the graph decreases.

GM has the best performance in all the cases. While *TM* has comparable performance with *GM* on the tree pattern query HQ_2, it is outperformed by *GM* by several orders of magnitude compared to the other graph pattern queries. In particular, it could not complete within 10 minutes the evaluation of HQ_4 on all four versions of *em*. *JM* is up to 13× slower than *GM* for HQ_2, HQ_4, and HQ_7.

Varying graph sizes. In this experiment, we evaluated the scalability of the three algorithms as the data set size grows. We recorded the elapsed query time on increasingly larger randomly chosen subsets of the DBLP data. Figure 4.7 shows the results of the three algorithms evaluating instantiations of the query templates HQ_8 and HQ_{12} shown in Figure 4.3. The other queries gave similar results in our experiments. As expected, the execution time for all algorithms increases when the total number of graph nodes increases. *GM* scales smoothly compared to both *TM* and *JM* for evaluating the two queries.

4.4.4 Effectiveness of New Framework

In this subsection, we evaluate the effectiveness of our proposed techniques and strategies for reducing the overall querying time.

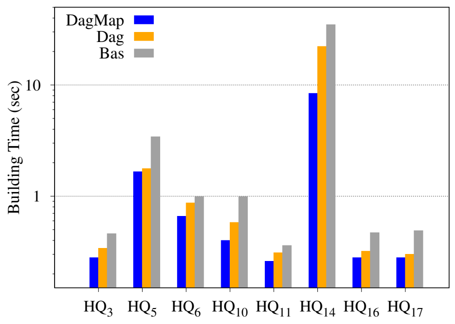
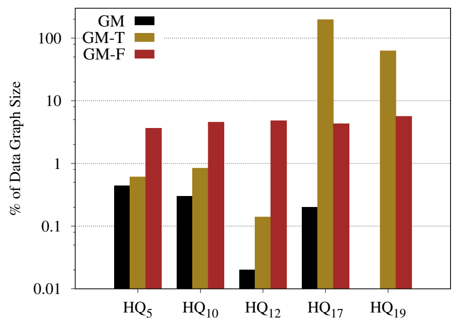


Figure 4.8 Simulation relation building time on *em*.

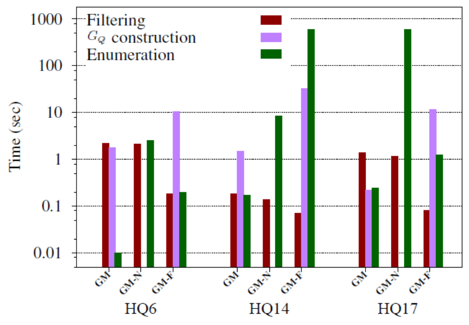
Simulation relation construction. We compare three methods to construct the double simulation relation $\mathcal{FB}$. The first one, denoted here as *Bas*, is the basic algorithm described in Subsection 4.2.3. The second one, denoted as *Dag*, is described in Subsection 4.2.4. This method explores the pattern structure to construct $\mathcal{FB}$, thus it can converge faster by reducing the number of iteration passes of *Bas*. The third one, called *DagMap*, applies the optimization techniques described in Subsection 4.2.5 to achieve further speedup on top of *Dag*.

Figure 4.8 shows the running time of the three methods when evaluating hybrid queries from different categories using *GM* on the data graph *em*. As we can see, *DagMap* consistently outperforms the other two and *Dag* comes the next.

Summary graph size. In this experiment, we examine the size of summary graph graphs achieved by different query evaluation methods. As usual, the size of a graph is measured by the total number of nodes and edges—the smaller the size of the summary graph achieved, the better. We design two variants of *GM* referred to here as *GM-F* and *GM-T*. Unlike *GM*, *GM-F* does not compute the double simulation $\mathcal{FB}$, but only applies node pre-filtering to prune nodes from inverted lists. Then, it builds a summary graph based on the pruned inverted lists. *GM-T* first transforms the graph pattern query into a tree query and builds a refined summary graph for the tree query. A similar procedure for evaluating pattern queries is also used in [30, 31].



(a) Summary Graph Size



(b) Query time breakdown

Figure 4.9 Summary graph size and query time breakdown on *ep*.

We compare the sizes of the summary graph graphs generated by *GM*, *GM-T* and *GM-F* for different queries on data graphs. Figure 4.9(a) reports on the results for evaluating queries on the data graph *ep*. The Y-axis shows the summary graph graph size as a percentage of the input data graph size.

Filtering and refinement reduce the size of summary graph significantly. In all cases, *GM* generates the smallest summary graph graph, with the average percentage over all 20 queries on *ep* being around 0.4%. Notice that query HQ_{19} has an empty answer on *ep*, and this case is detected by *GM* at an early stage of the query evaluation. The average percentage is around 4.2% for *GM-F*. This demonstrates that the double simulation technique has much better pruning power than the node pre-filtering. On HQ_{17} and HQ_{19}, the summary graph size of *GM-T* is around 45× and 10× that of *GM-F*, respectively. In the rest of the cases, however, the former is much smaller than the latter.

Summary graph benefit and overhead. The overhead for constructing and using a refined summary graph for query result enumeration is insignificant compared to its benefits. To experimentally verify this, we design two variants of *GM*. One is *GM-F* introduced above. The other one is called *GM-N*. Variant *GM-N* does not construct a summary graph but uses the simulation relation $\mathcal{FB}$ to directly compute the query occurrences. As shown in Figure 4.9(b), query result enumeration by *GM* with a refined summary graph (including the computation of the simulation relation and the overhead for the construction of the summary graph), is up to 3 orders of magnitude faster than query result enumeration by *GM-F* with an unrefined summary graph as well as query result enumeration by *GM-N* using directly a simulation relation. This speedup comes from several factors including reduced search space, filtering and refinement, and computation sharing provided by the summary graph.

Search order. In this experiment, we compare the effectiveness of four search ordering strategies for homomorphic pattern matching: JO, BJ, RI, and TD. JO is described in Section 4.3. BJ finds an optimal left-deep join plan of the given query through dynamic programming. Unlike JO and BJ, RI [34] generates a search order based only on the topological structure of the given query. Roughly speaking, vertices that are highly connected with vertices previously present in the ordering tend to come early in the final ordering produced by RI. TD works on dag queries. It considers the query nodes according to their topological order in the query graph. An ordering strategy similar to this one is adopted in [31]. We integrate JO, RI, BJ and TD with Algorithm *MJoin* which is used by the approach *GM* for enumerating query occurrences (Section 4.3).

Table 4.3 shows the experimental results comparing the four ordering strategies for evaluating hybrid queries on graph *em* and *ep*. Except for query HQ_{15} on graph *em*, JO gives the best performance and BJ comes next. BJ is able to find a good query plan, but it does not scale to large queries with tens of nodes [73]. RI does not perform well on most of these homomorphically matched hybrid queries, even though it is found to be an effective technique in recent research on subgraph isomorphism matching [9]. Comparative results between JO and RI for evaluating direct-edge only queries are shown in Subsection 4.4.5. Unlike JO which uses cardinalities of node sets in summary graph graphs to do cost estimation, RI produces a search order based

Table 4.3 Effectiveness of Search Ordering Methods.

Query	*em*				*ep*			
	RI	*JO*	*BJ*	*TD*	*RI*	*JO*	*BJ*	*TD*
HQ_2	3.64	**1.88**	2.45	4.43	7.00	**2.02**	2.09	13.54
HQ_4	3.06	**1.05**	**1.05**	5.95	4.67	**0.67**	0.88	7.98
HQ_{15}	**1.33**	7.32	1.79	13.91	**6.33**	6.66	**6.33**	272.23
HQ_{18}	7.07	**0.99**	1.36	7.97	441.94	**30.18**	38.15	420.96

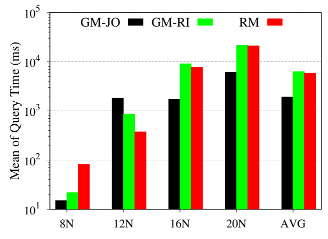
exclusively on the topological structure of the given query, independently of the input data graph. TD gives the worst performance in most cases.

This experimental result demonstrates that an effective search ordering strategy for homomorphic pattern matching should take into account both the query graph structure and the data graph statistics.

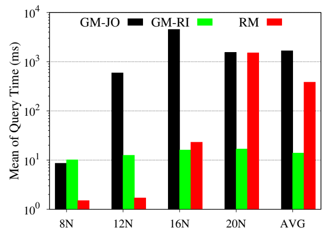
4.4.5 Comparison to Systems and Engines

We compare the performance of *GM* with the graph query engine RapidMatch [10] and the graph DBMS Neo4j. All these engines/systems were designed to process graph pattern queries whose edges are mapped with homomorphisms to edges in the data graph (therefore, they do not need a reachability index). Our approach is more general since at allows edge-to-edge and edge-to-path homomorphic mappings.

Comparison with RapidMatch. RapidMatch [10] (abbreviated as *RM* here) is a recent graph query engine which outperforms the state of the art graph matching approaches CFL [30], DAF [31] and GraphFlow [8, 11]. The source code of *RM*[1]is publicly available at GitHub. *RM* is a tree-based graph query engine that adopts worst-case optimal (WCO) style joins in its result enumeration. It proposes a sophisticated search order method based on the nucleus decomposition of query



(a) Dense query sets



(b) Sparse query sets

Figure 4.10 Performance comparison of *GM* with *RM* for large queries on the undirected Human data graph.

graphs. To improve its enumeration efficiency, *RM* adopts several optimizations, including advanced set intersection methods [72, 74], the intersection caching [8, 11], and the failing set pruning [31].

We compare *RM* with two variants of *GM*, denoted as *GM-JO* and *GM-RI*, which use *JO* and *RI* as their respective search order method. We use the recommended configuration for *RM*[1] to compute homomorphic matches. We follow also the experimental setting described in [10] by setting the time limit at 5 minutes and the max. number of matches at 10^5. We run the experiment on the data graphs and query workloads used in [10]. *RM* considers undirected graphs. Our approach, *GM* is more general as it considers directed graphs. In order to compare with *RM*, we replace each edge of the undirected data graph by two directed edges in opposite direction and we use this graph as input to *GM* for the experiment. Each data graph has one dense query set (the degree of each query node is at least 3) and one sparse query set (the degree of each query node is less than 3). Each set contains 200 connected query graphs with the same number of nodes.

Figures 4.10(a) and 4.10(b) present the mean of query time on different query sizes of dense and sparse query sets, respectively, on the Human data graph. We choose the Human data graph since it is a real dataset and it is very dense and most of its nodes have the same label making the graph matching particularly challenging

[1]https://github.com/RapidsAtHKUST/RapidMatch, Last accessed on 2022/09/20.

[9, 10]. The number of nodes for queries on the graph varies from 8 to 20. Query sets with i nodes are denoted as iN. For the dense query sets, *GM-JO* has, overall, the best performance. It runs more than 2 times faster than *RM* on average. *GM-RI* runs slightly slower than *RM* on average. In contrast, for the sparse query sets, *GM-JO* gives the worst performance, while *GM-RI* achieves up to two order of magnitude average speedup over the other two algorithms.

When our algorithm runs much slower than competing algorithms, it is because it generates ineffective search orders for a number of queries. Both *RI* and the search order method of *RM* prioritize the dense sub-structure of a query Q. When this assumption in the heuristic search rule does not hold on the workloads, they can generate ineffective searching orders. *JO* performs well on dense queries, but worse on sparse ones because the cardinality differences of candidate occurrence sets among query nodes tend to be very small for sparse queries, thus making it difficult to choose an effective search order. Advanced subgraph cardinality estimation methods [75] can help *JO* to improve its search order quality.

In summary, our experimental results on the Human graph demonstrate that *GM* with two simple search order methods beats the highly optimized *RM* that comes with a sophisticated search order method, in most tested workloads, despite the fact that it is more general since it considers directed data graphs and allows also for edge-to-path mapping.

Neo4j comparison. In this experiment, we compared *GM* with the most popular graph DBMS Neo4j on evaluating hybrid pattern queries. Cypher, Neo4j's query language, uses patterns to match desired graph structures. The pattern matching in Cypher adopts the path finding semantics, whereas we adopt the path existence semantics for pattern matching. For a fair comparison on pattern evaluation, we enforce the path existence semantics in Cypher using Neo4j APOC (Awesome Procedures On Cypher) procedures [66].

Table 4.4 Runtime (sec) of Neo4j and *GM* for Hybrid Pattern Queries on a Fragment of *em* Graph with 30K Nodes

Alg.	Acyc			Cyc			Clique			Combo		
	HQ_0	HQ_3	HQ_5	HQ_6	HQ_8	HQ_{17}	HQ_{11}	HQ_{12}	HQ_{19}	HQ_{10}	HQ_{13}	HQ_{16}
Neo4j	51.952	457.034	> 3600	60.119	35.86	118.709	54.104	> 3600	> 3600	319.064	> 3600	476.426
GM	0.29	0.22	0.32	0.09	0.05	0.02	0.02	0.02	0.04	0.04	1.31	0.16

We used Neo4j v.4.2.1 and expressed queries in Cypher. Both *GM* and Neo4j were configured to find all the query matchings. The timeout was set to one hour. The results on a fragment of *em* graph with 30K nodes are shown in Table 4.4. We note that Neo4j was unable to finish within 1 hour for all the tested queries on the original *em* graph. *GM* is significantly faster than Neo4j, in most cases by orders of magnitude. These results demonstrate the advantage of our proposed approach for evaluating pattern queries on graphs as well as the need to extend Neo4j with efficient reachability query evaluation support [67]. We however note that Neo4j is a full-fledged graph database system with many functions that our prototype implementation does not support, and this may also account for the performance gap between our framework and Neo4j.

CHAPTER 5

ANSWERING GRAPH PATTERN QUERIES USING COMPACT MATERIALIZED VIEWS

In Chapter 4, we introduced the concept of a summary graph. In this chapter, we use the summary graph as a compact representation of an intermediate result for query answers, which is stored for use in future query cases. Specifically, we refer to this intermediate result as a view materialization.

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 [45, 46, 47, 48, 49]. 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 [45], or to otherwise provide a process for computing the query answer from the view materializations [46]. Materialized views can also be effectively used for addressing the data scalability problem of queries. In this dissertation, we adopt a novel approach for materializing graph pattern views over data graphs, which builds off algorithms that we introduced in Chapter 4.

5.1 Preliminaries and Problem Definition

In Chapter 4, we defined homomorphisms between a graph pattern and a data graph. To define which views can be used for a query, homomorphisms can be also defined between query graph patterns as follows.

Definition 5.1.1 (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 $x \in V$, $label(x)$

$= \mathit{label}(h(x))$; and (2) for any edge $(x, y) \in V$, if (x, y) is a direct edge, $(h(x), h(y))$ is a direct edge of Q, while if (x, y) is a reachability edge, $h(x) \prec h(y)$ in Q.

Note that if (x, y) is a reachability edge in V, the path (of direct and/or reachability edges) in Q from $h(x)$ to $h(y)$, can be a single direct or reachability edge.

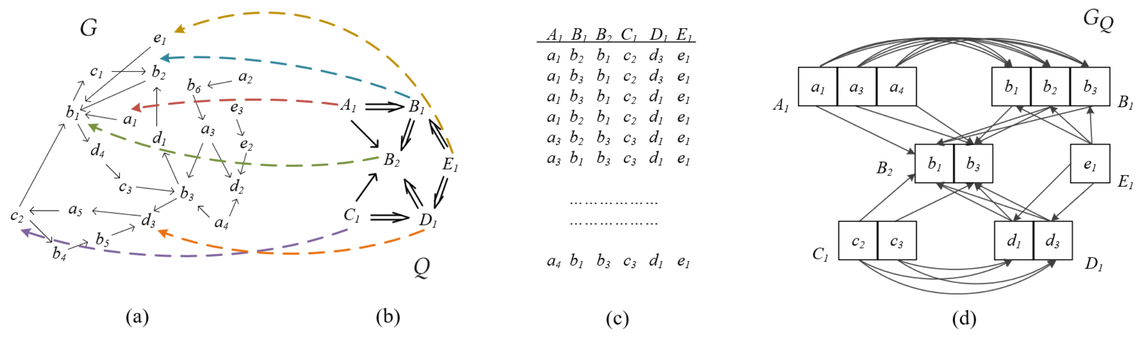
Views and View Materializations. A *view* is a named query. The class of views is not restricted. Any type of query can be a view. We materialize views on a data graph by storing a summary graph of this view.

Definition 5.1.2 (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.

A summary graph constitutes a search space for the view answer and its matches can be enumerated by applying multiway joins while traversing the graph.

To show how views can have a homomorphism to a query, and get a sense of how view materializations are related to query summary graphs, we first show an example of a query with a homomorphism to a data graph. Figure 5.1(a,b) depicts a homomorphism h of query Q to the data graph G. Figures 5.2(a) and (b) show a query Q and a view V_1 with a homomorphism from V_1 to Q.

Figure 5.1(c) displays the answer of a query Q on a data graph G, and Figure 5.1(d) shows a summary graph G_Q for the query Q of Figure 5.1(a). Then Figure 5.2(c) shows *a summary graph for the query (view)* V_2 of Figure 5.2(b). The summary graph of Figure 5.2(c) is also the materialization of view V_2. One can see that this summary graph is the answer graph of V_2. Figure 5.2(c) bears many similarities to Figure 5.1(d), the summary graph of the query. Later in this chapter, we will show how a view's materializations, for a view which has a homomorphism(s) to a query, can be used to construct a query's summary graph.



A_1	B_1	B_2	C_1	D_1	E_1
a_1	b_2	b_1	c_2	d_3	e_1
a_1	b_3	b_1	c_2	d_3	e_1
a_1	b_3	b_1	c_2	d_1	e_1
a_1	b_2	b_1	c_2	d_1	e_1
a_3	b_2	b_3	c_3	d_1	e_1
a_3	b_1	b_3	c_3	d_1	e_1
……	……	……	……	……	……
……	……	……	……	……	……
a_4	b_1	b_3	c_3	d_1	e_1

Figure 5.1 (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.

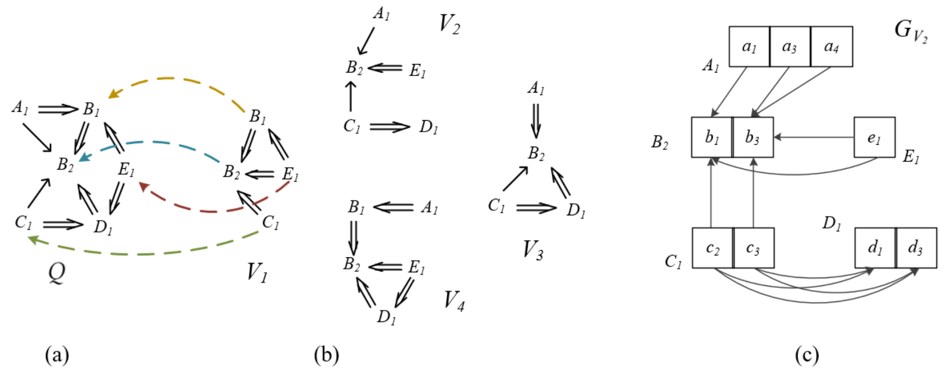


Figure 5.2 (a) A graph pattern query Q, (b) Views V_1, V_2, V_3, V_4, and a homomorphism from V_1 to Q, (c) A summary graph G_{V_2} of V_2 on data graph G of Figure 5.1(a).

5.2 Materialized View Usability

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 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 5.2.1. 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 5.2, 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 5.2.2. 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. 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 direct edge then e_v is also a direct edge.

Proof. *If part:* Assume there is a homomorphism h from V to Q that maps e_v to e_q such that if e_q is a direct edge then e_v is also a direct edge. Let G be a data graph and let there be homomorphism h' from Q to G that maps e_q in Q to an occurrence (a, b) in G. By homomorphism composition, (a, b) is also an occurrence of e_v. Therefore,

$oc(e_q) \subseteq oc(e_v)$. Since e_v and e_q have the same labels, edge direction and are both direct or reachability edges, $ms(e_v) = ms(e_q)$. As $oc(e_v) \subseteq ms(e_v)$, $oc(e_v) \subseteq ms(e_q)$. Thus, e_q is covered by e_v.

Only if part: (1) Let's assume that edge e_q in Q is covered by edge e_v in V and there is no homomorphism from V to Q. Let G be a data graph that is the same as query Q except the reachability edges in Q are replaced by regular edges. Then Q has a homomorphism to G but V does not; that is, e_q has an occurrence in G while e_v does not have any. As $os(e_q) \not\subseteq os(e_v)$, our assumption that edge e_q in Q is covered by edge e_v in V is contradicted.

(2) Let's now assume that edge e_q in Q is covered by edge e_v in V and there is a homomorphism from V to Q but no homomorphism that matches e_v to e_q. Let G be a data graph that is the same as query Q except that reachability edges in Q are replaced by regular edges. Let also h_1 be a homomorphism which maps every node of Q to its corresponding node in G. Then the occurrence of e_q produced by h_1 on G is not an occurrence of e_v on G, that is $os(e_q) \not\subseteq os(e_v)$. This contradicts our assumption that edge e_q is covered by edge e_v.

(3) Let's finally assume that edge e_q in Q is covered by edge $e_v = (x, y)$ in V and there is a homomorphism from V to Q which matches e_v to e_q but e_q is a direct edge while e_v is a reachability edge. Let G be a graph which is the same as view V except that reachability edges other than e_v in V are replaced by regular edges in G, and e_v is replaced by a path of two edges where the internal node is labeled by a label not appearing in Q. Then, since e_q is a direct edge and there is no edge from x to y in G, (x, y) is an occurrence of e_v on G but not a match of e_q in G. This is a contradiction since as e_v covers e_q, $os(e_v) \subseteq ms(e_q)$ on any data graph. $\square$

In the example of Figure 5.2, 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 direct edge and the latter is a reachability 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 reachability 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 5.2.3. 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 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 $\mathcal{V}$.

Proof. *If part:* Clearly, if every edge e_q of Q is covered by an edge e_v of a view in $\mathcal{V}$, the answer of Q can be obtained by joining the occurrence sets $oc(e_v)$ of the covering view edges on their common query nodes. As $oc(e_q) \subseteq oc(e_v)$ no query occurrence in the answer of Q is missed.

Only if part: Let's assume that Q can be answered using the views in $\mathcal{V}$ and there is a non-redundant edge e_q in Q which is not covered by any edge of the views in $\mathcal{V}$. Since Q can be answered using $\mathcal{V}$, it can be computed by a relational algebra expression E which joins the answers of the views in $\mathcal{V}$. Since, e_q is not redundant, it is not a transitive edge in Q. Let G be a data graph which has the nodes of Q and an edge for every edge in the transitive closure of Q except for e_q. The answer of Q on G is empty while E returns a non empty answer on G. This contradicts our assumption that Q can be answered using the views in $\mathcal{V}$. $\square$

In the example of Figure 5.2, 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 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 5.2.4. 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 5.2, query Q can be answered using the view set $\{V_1, V_2, V_4\}$ which is minimal. We present in Section 5.3 an algorithm which computes a minimal set of views for answering a query.

5.3 Algorithms

In this section, we present an algorithm called *SumGraphBuildViews* 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 *SumGraphBuildViews* uses another algorithm, called *FindQCover*, which computes the covering set $cov(e, V)$ of a view V for each query edge e. Therefore, Algorithms *SumGraphBuildViews* 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 11, 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 (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 *SumGraphBuildViews* on the query Q and the view V_1 of Figure 5.2 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 *SumGraphBuildViews*, shown in Algorithm 12,

Algorithm 11: 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 $Q$ */
    homEnumerate(1, $t$);
5  For every node $q$ in $Q$ and for every edge $e$ in $Q$, $cov(q) = \emptyset$ and $cov(e) = \emptyset$;
6  findCover();
7  return $cov$;

8  Procedure homEnumerate($i, t$):
9      if ($i$=$n$+1) then
10         add $t$ to $H$ and return;
11     end
12     $N_i := \{v_j \mid (v_i, v_j) \in V \text{ or } (v_j, v_i) \in V, j \in [1, i-1]\}$;
13     $S'_i := S_i$;
14     for (every $v_j \in N_i$) do
15         $S'_i := \{q \in S'_i \mid q \prec t[j] \text{ or } t[j] \prec q\}$;
16         for (every $q \in S'_i$) do
17             if (($v_j, v_i$) is a direct edge in $V$ and ($t[j], q$) is not a direct edge
                 in $Q$) or (($v_i, v_j$) is a direct edge in $V$ and ($q, t[j]$) is not a
                 direct edge in $Q$ ) then
18                 Remove $q$ from $S'_i$;
19             end
20         end
21     end
22     for (every node $q \in S'_i$) do
23         $t[i] := q$;
24         homEnumerate($i + 1$, $t$);
25     end

26 Procedure findCover():
27     for (every tuple $t \in H$) do
28         for (every node $v \in V$) do
29             add $v$ to $cov(t[v])$;
30         end
31         for every edge $(v_i, v_j)$ in $V$ do
32             if $e = (t[v_i], t[v_j])$ is an edge in $Q$ which is a direct edge if
                 $(v_i, v_j)$ is a direct edge then
33                 add $(v_i, v_j)$ to $cov(e)$;
34             end
35         end
36     end
```

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, *SumGraphBuildViews* 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 11) (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, *SumGraphBuildViews* 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 (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.

Algorithm 12: Algorithm SumGraphBuildViews

```
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 $Q$).

initializeCos();
while (cos has changes) do
    forwardPrune();
    backwardPrune();
end
return $cos$;

Procedure initializeCos():
    For every node $q \in Q$, initialize $cos(q)$ to be $ms(q)$.;
    For every edge $e_q \in Q$, initialize $cos(e_q)$ to be $\emptyset$;
    for (every view $V \in \mathcal{V}$) do
        $cov := FindQCover(Q, V)$;
        for (every node $q \in Q$) do
            $cos(q) := cos(q) \cap_{v \in cov(q)} cos(v)$;
        end
        for (every edge $e \in Q$) do
            if ($cos(e) = \emptyset$) then
                $cos(e) := \cap_{e_v \in cov(e)} cos(e_v)$;
            else
                $cos(e) := cos(e) \cap_{e_v \in cov(e)} cos(e_v)$;
            end
        end
    end

Procedure forwardPrune():
    for (each edge $e_q = (q_i, q_j) \in Q$ and each $n_{q_i} \in cos(q_i)$) do
        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
            Remove $n_{q_i}$ and its associated outgoing edges from $cos(q_i)$ and
              $cos(e_q)$, respectively;
        end
    end

Procedure backwardPrune():
    for (each edge $e_q = (q_i, q_j) \in Q$ and each $n_{q_j} \in cos(q_j)$) do
        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
            Remove $n_{q_j}$ and its associated incoming edges from $cos(q_j)$ and
              $cos(e_q)$, respectively;
        end
    end
```

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 5.2. Algorithm *SumGraphBuildViews* on the answer graph for V_1 of Figure 5.2(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 5.1(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.

Algorithm 13: 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 $\mathcal{V}'$;

5  Procedure findViews():
6      $U := edges(Q)$;        /* the set of uncovered edges of $Q$ */ ;
7      while ($U \neq \emptyset$) do
8          Select an edge $e$ in $U$;
9          Find a view $V$ in $\mathcal{V}$ which has an edge covering $e$;
10         Let $C$ be the set of edges in $Q$ which are covered by $V$;
11         $\mathcal{V}' := \mathcal{V}' \cup \{V\}$;
12         $U := U - C$;
13     end

14 Procedure removeRedundant():
15     for (every view $V \in \mathcal{V}'$) do
16         if ($Q$ can be answered using exclusively $\mathcal{V}' - \{V\}$ ) then
17             Remove $V$ from $\mathcal{V}'$ ;
18         end
19     end
```

Finding a Minimal View Set. Algorithm *FindMinimalVSet*, shown in Algorithm 13, 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 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 5.2, Algorithm 13 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.

5.4 Experimental Evaluation

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

5.4.1 Experimental Setting

Algorithms in comparison. We implemented our approach for answering queries using materialized views. In our implementation of Algorithm *SumGraphBuildViews*, 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 algorithm Algorithm *BuildSummaryGraph*, the approach presented in Chapter 4 for evaluating hybrid graph pattern queries using homomorphisms over a large graph. In this section, we refer to the algorithm from Chapter 4 as *FltSim*. Since *FltSim* constructs a summary graph for the input query on a data graph, 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 *MatView* 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, *MatView* 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 using 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* was shown in Chapter 4 to outperform previous state-of-the art approaches [15, 23, 11, 68] 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 [8, 9]. The datasets have different structural properties and come from different application domains, such as the web and social networks.

Table 5.1 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 5.1 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 5.3. These hybrid query templates involve direct and reachability edges. They have various and complex structures and many of them were used in previous work [15, 11]. The number associated with each node of a query template denotes the node id. Query instances are generated by assigning labels to nodes.

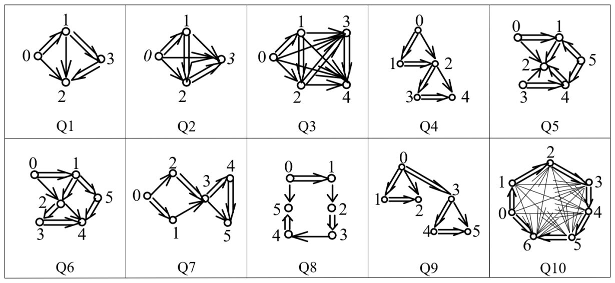


Figure 5.3 Graph pattern query templates used in the evaluation.

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 *SumGraphBuildViews* 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 *SumGraphBuildViews*.

Let $cov(e_q, \mathcal{V})$ be the number of view edges in view set $\mathcal{V}$ that cover query edge e_q from the edge set $E(Q)$ of query Q. The average number of covering edges for a query edge e_q in Q in a view set $\mathcal{V}$ is calculated using the following equation:

$$cov_{avg}(Q, \mathcal{V}) = \frac{\sum_{e_q \in E(Q)} cov(e_q, \mathcal{V})}{|E(Q)|} \tag{5.1}$$

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 "Benefit of Using Materialized Views", "Data Graph Size Scalability", and "Varying the Number of Query Edges"s, 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 13.

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.

5.4.2 Benefit of Using Materialized Views

Benefit of Using Materialized Views. Figure 5.4 displays the elapsed time of *FltSim* versus *MatView* for all the queries of Figure 5.3 on a bs data graph with 350K nodes and five 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 5.5 displays the elapsed time of *FltSim* versus *MatView* for all the queries of Figure 5.3 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 reachability edges, *MatView* is five order of magnitude better than *FltSim*.

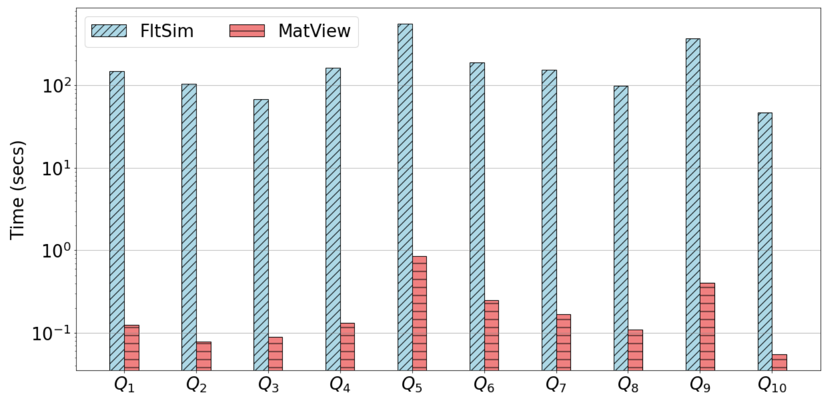


Figure 5.4 Elapsed time of *FltSim* and *MatView* for various queries on a bs data graph with 350K nodes and five labels.

5.4.3 Data Graph Size Scalability

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. The x-axis shows the size of the data graph in terms of 1k nodes (for instance, 100 means 100k). Figure

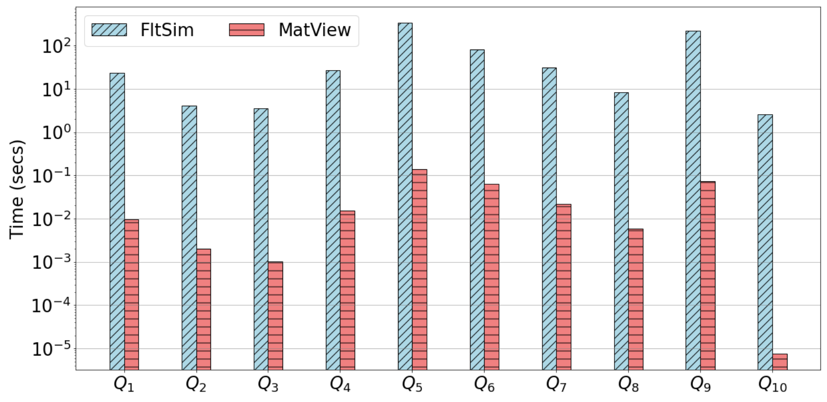


Figure 5.5 Elapsed time of *FltSim* and *MatView* for various queries on dblp data graph with 250K nodes and 20 labels.

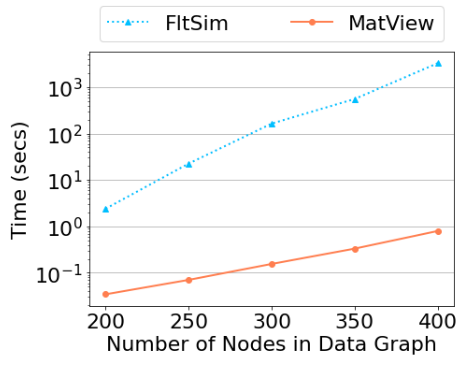
5.6 shows the results, on a logarithmic scale for y-axis, for queries Q_5 and Q_6 on the bs data graph with five labels. Figure 5.7 shows the results, on a logarithmic scale for the y-axis, for queries Q_7 and Q_9 on the dblp data graph with 20 labels.

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 $MatView$.

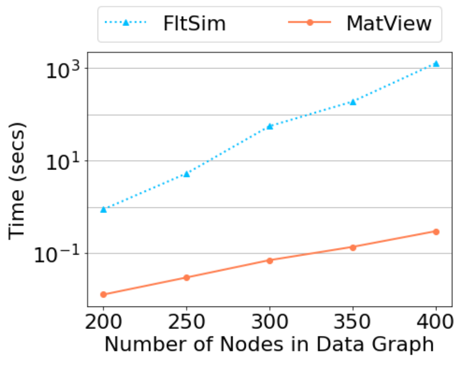
We also observed that in Figure 5.7, for the data point with 100K nodes, the evaluation time for *MatView* was very small. This is because, in contrast to the other data points, there were no matches for query Q_7; while *FltSim* had to spend time to filter out irrelevant nodes and edges from the data graph before it discovered that the query has an empty answer, *MatView* was able to quickly discover that this query has empty answer.

5.4.4 Performances on View and Query Variations

Varying the Number of Covering View Edges. We ran experiments comparing the performance of *MatView* when 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

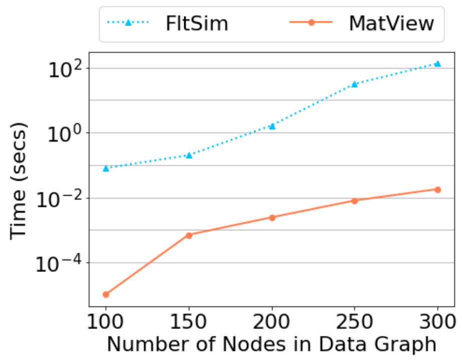


(a) Q_5

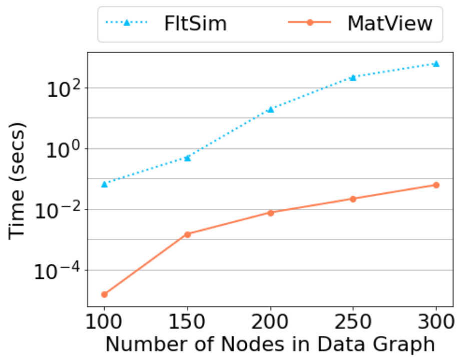


(b) Q_6

Figure 5.6 Elapsed time of *FltSim* and *MatView* on increasingly larger number of data subsets of the bs data subset with five labels.



(a) Q_7

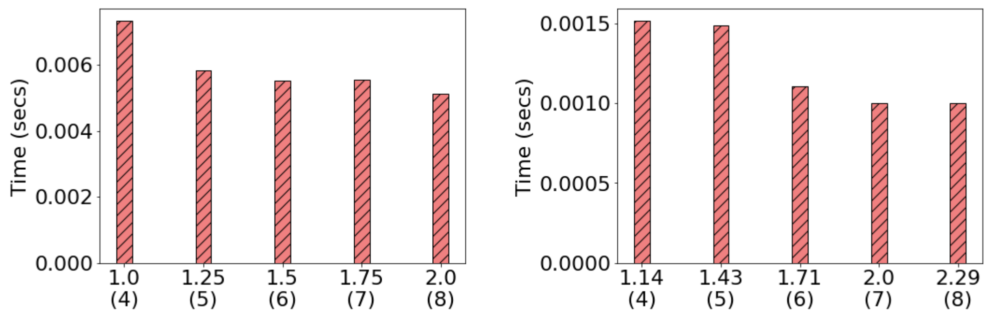


(b) Q_9

Figure 5.7 Elapsed time of *FltSim* and *MatView* on increasingly larger number of data subsets of the dblp data subset with 20 labels.

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 5.8, we plotted the value of $cov_{avg}(Q, \mathcal{V})$ for each new set of views $\mathcal{V}$ 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 5.8. 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.



(a) Q_5 (b) Q_7

Figure 5.8 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.

Varying the Number of Edges per View. We compared the performance of *MatView* 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 5.9. 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.

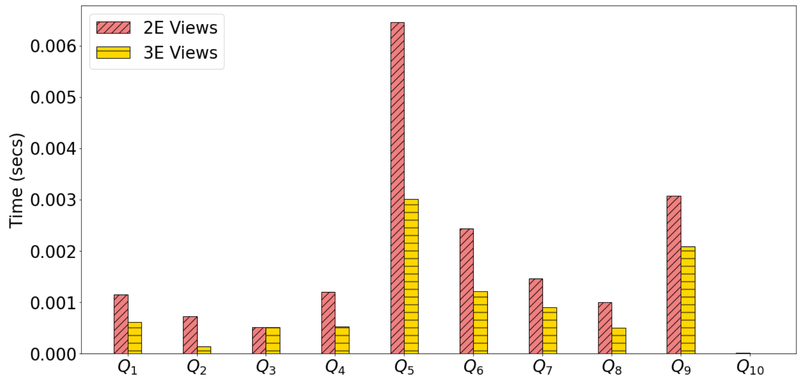


Figure 5.9 Elapsed time of *MatView* using *2-edge views* and *3-edge views* for various queries on a bs data subset with 20 labels and 350K nodes.

Varying the Number of Query Edges. We measured the execution time of the two approaches 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 5.10. 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.

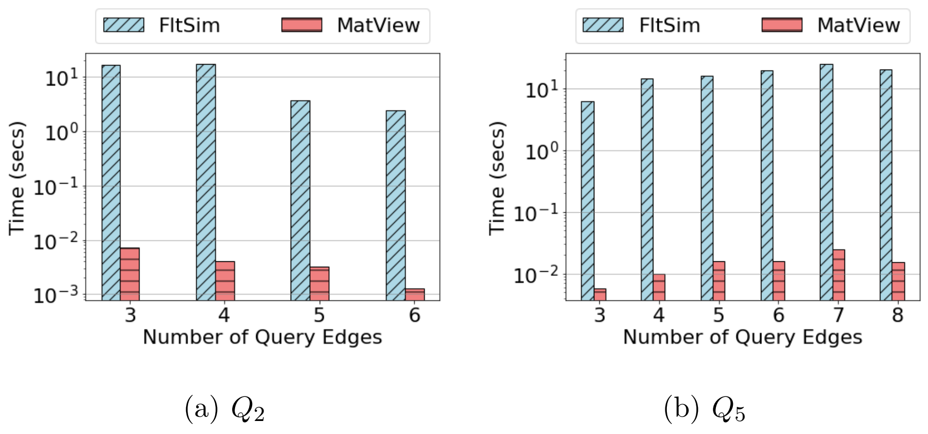


Figure 5.10 Elapsed time of *FltSim* and *MatView* on queries with a different edges on a bs data subset with 20 labels and 350K nodes.

CHAPTER 6
OPTIMIZING GRAPH PATTERN QUERIES WITH MATERIALIZED VIEWS

In Chapter 5, we focused on views which cover every edge of a query, such that a query can be answered exclusively using the views without needing to use a reachability index. However, views do not have to cover every edge of a query to be used by a query; we can use views whose edges only cover some of the edges of a query to improve the performance times of algorithms which find the query's answer. In this chapter, we investigate views which only need to partially cover a query, meaning only at least one edge of a query is covered by a view. Additionally, we can also use views that just cover query nodes, even if they do not cover any edges, as the query nodes that are covered can reduce the candidate occurrence set sizes of the query's summary graph.

6.1 Preliminaries and Problem Definition

We now formally define what it means for a view node to cover a query node.

Definition 6.1.1 (Node Coverage)**.** A node x_q of a query Q is *covered* by an node x_v of a view V if $os(x_q) \subseteq os(x_v) \subseteq ms(x_q)$ on any data graph G.

To show how views are used to cover a query and partially contribute to summary graph construction, we first show an example of a query with a homomorphism to a data graph. Figure 6.1 shows a homomorphism h of query Q to the data graph G, along with its answer and summary graph. Then in Figure 6.2, one can see that node B_1 of query Q from 6.1 is covered by the node B_1 of query V_4 since for every homomorphism of Q to G, there is a homomorphism of V_4 to G. Furthermore, the node A_1 of view V_1 covers the node A_1 of query Q and the edge (B_1, C_1) of view V_1 covers the edge (B_1, C_1) of query Q since for every homomorphism

h of Q to G, there is a homomorphism of V_1 to G which is a restriction of h. Later on in Section refsec:optAlgo, it will be shown how the materializations of views V_1 and V_2 are used in the summary graph construction of query Q.

We can now define view usability. A *Usable View* is defined in Chapter 5, defined as a view with having at least one edge of Q covered by an edge of V. We will now formally define a Weakly Usable View, as used for optimizing summary graph construction times.

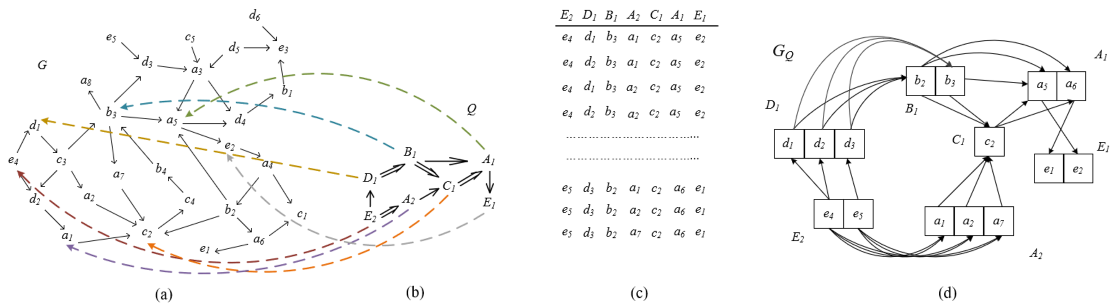
Definition 6.1.2 (Weakly Usable View)**.** A view V is *weakly usable* for optimizing a query Q if at least one node of Q is covered by a node of V.

We characterize query node coverage in terms of homomorphisms from a view to the query. In this chapter, we refer to homomorphisms as 'ep-homomorphisms', to more specifically refer to them as 'edge-path' homomorphisms.

Theorem 6. Let x_q be a node in a graph pattern query Q and x_v be a node in a view V. Node x_q in Q is covered by node x_v in V iff there is a ep-homomorphism from V to Q that maps x_v to x_q.

Proof. *If part:* Assume there is an ep-homomorphism h from V to Q that maps x_v to x_q. Let G be a data graph and let there be an ep-homomorphism h' from Q to G that maps x_q in Q to an node u in G. The composition of h and h', $h \circ h'$, is a homomorphism from V to G which maps x_v to u. Therefore, $os(x_q) \subseteq os(x_v)$. Since x_v and x_q have the same labels, $ms(x_v) = ms(x_q)$. As $os(x_v) \subseteq ms(x_v)$, $os(x_v) \subseteq ms(x_q)$. Therefore, $os(x_q) \subseteq os(x_v) \subseteq ms(x_q)$. Thus, x_q is covered by x_v.

Only if part: (1) Let's assume that node x_q in Q is covered by node x_v in V and there is no homomorphism from V to Q. Let G be a data graph that is the same as query Q except that reachability edges in Q are replaced by regular edges. Then Q has a homomorphism to G but V does not; that is, x_q has an occurrence in G while



E_2	D_1	B_1	A_2	C_1	A_1	E_1
e_4	d_1	b_3	a_1	c_2	a_5	e_2
e_4	d_2	b_3	a_1	c_2	a_5	e_2
e_4	d_1	b_3	a_2	c_2	a_5	e_2
e_4	d_2	b_3	a_2	c_2	a_5	e_2
………						
………						
e_5	d_3	b_2	a_1	c_2	a_6	e_1
e_5	d_3	b_2	a_2	c_2	a_6	e_1
e_5	d_3	b_2	a_7	c_2	a_6	e_1

Figure 6.1 (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.

x_v does not have any, and thus $os(x_q) \not\subseteq os(x_v)$, contradicting our assumption that node x_q in Q is covered by node n_v in V.

(2) Let's now assume that node x_q in Q is covered by node x_v in V and there is a homomorphism from V to Q but no homomorphism that matches x_v to x_q. Let G be a data graph that is the same as query Q except that reachability edges in Q are replaced by regular edges. Let also h_1 be a homomorphism which maps every node of Q to its corresponding node in G. Then the occurrence of x_q produced by h_1 on G is not an occurrence of x_v on G, that is $os(x_q) \not\subseteq os(x_v)$. This contradicts our assumption that node x_q is covered by node x_v.

Therefore if x_q is covered by x_v, there is an ep-homomorphism from V to Q that maps x_v to x_q.

$\square$

In the example of Figure 6.2, V_4 is a weakly usable view because it has a homomorphism to Q and while it does not cover any edges of Q, its node E_2 covers the node E_2 in Q. The edge (B_1, A_1) of view V_3 covers the edge (B_1, A_1) of query Q_1. In contrast, (C_1, B_2) in Q does not cover (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 by a homomorphism from V_1 to Q.

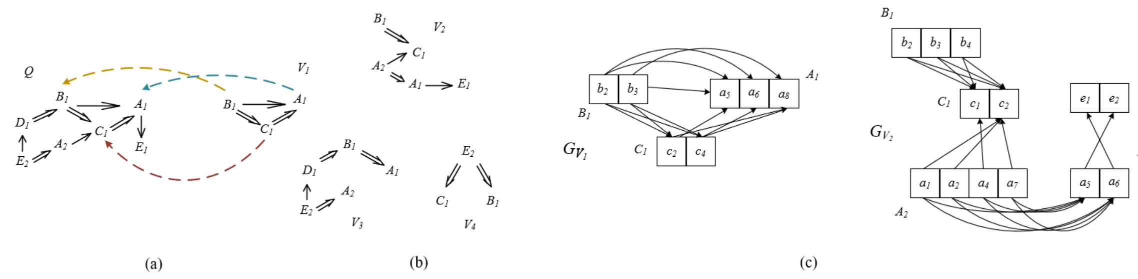


Figure 6.2 (a) A graph pattern query Q, (b) Views V_1, V_2, V_3, and a homomorphism from V_1 to Q, (c) Summary graphs G_{V_1} of V_1 and G_{V_2} of V_2 on data graph G of Figure 6.1(a).

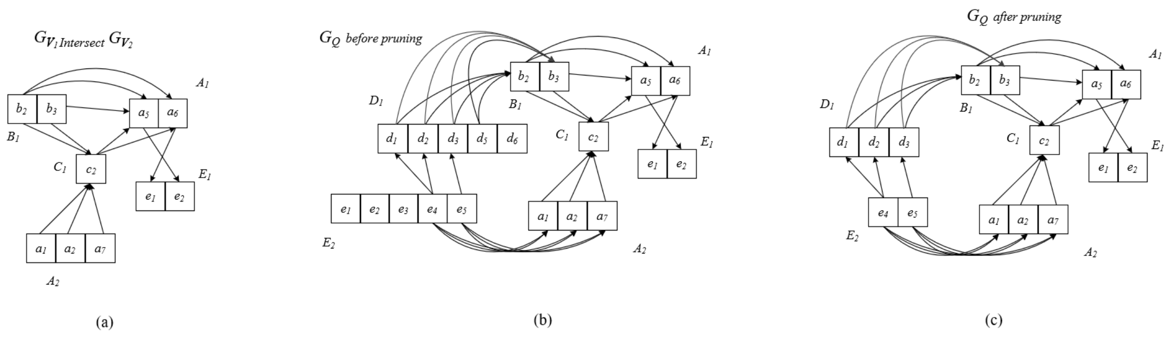


Figure 6.3 (a) The intersection of summary graphs G_{V_1} and G_{V_2}, (b) Merging the match sets of the uncovered query nodes and edges with Figure 6.3(a), (c) The summary graph G_Q obtained after pruning Figure 6.3(b).

6.2 Algorithms

We present an algorithm called *SumGraphBuildOptimize* 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 *SumGraphBuildOptimize* uses Function *Cov*, which computes the covering set $cov(e, V)$ of a view V for each query edge e. These algorithms are very similar to Algorithms *FindQCover* and *SumGraphBuild* in Chapter 5, with the main difference being that this algorithm can handle uncovered nodes and edges.

Figure 6.3 demonstrates steps performed by Algorithm *SumGraphBuildOptimize*. This example uses views V_1 and V_2 to cover query Q, leaving the node sets D_1 and E_2, along with the edges (D_1, B_1), $(E2_D1)$, and (E_2, A_2) as uncovered. Figure 6.3(a) shows a summary graph obtained by intersecting the materializations of views V_1 and V_2. Then Figure 6.3(b) shows the summary graph obtained after combining the intersection of the summary graphs of V_1 and V_2 with the uncovered node sets and edges. Finally, Figure 6.3(c) gives the summary graph after performing the procedures *forwardPruning()* and *backwardPruning()*. After pruning, the nodes which do not have candidate occurrence edges to nodes they should have edges to, according to the query Q, are eliminated. For instance, node d_5 does not have an edge to any node in node set E_2, but Q requires there to be an edge $(E2_D1)$, so d_5 cannot belong to an answer and is thus eliminated.

Algorithm 14: Function *Cov*

```
Input   : Graph pattern query $Q$ and a set $\mathcal{V}$ of materialized views
Output: Function $Cov$ on the nodes and edges of $Q$.

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

11 Procedure homEnumerate(index $i$, tuple $t$):
12     if ($i$=$n$+1) then
13         add $t$ to $H$ and return;
14     end
15     $N_i := \{v_j \mid (v_i, v_j) \in V \ or \ (v_j, v_i) \in V, j \in [1, i-1]\}$;
16     $S'_i := S_i$;
17     for (every $v_j \in N_i$) do
18         $S'_i := \{q \in S'_i \mid q \prec t[j] \ or \ t[j] \prec q\}$;
19         for (every $q \in S'_i$) do
20             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
21                 Remove $q$ from $S'_i$;
22             end
23         end
24     end
25     for (every node $q \in S'_i$) do
26         $t[i] := q$;
27         homEnumerate($i + 1$, $t$);
28     end

29 Procedure findCover():
30     for (every tuple $t \in H$) do
31         for (every node $v \in V$) do
32             add $v$ to $cov(t[v])$;
33         end
34         for every edge $(v_i, v_j)$ in $V$ do
35             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
36                 add $(v_i, v_j)$ to $cov(e)$;
37             end
38         end
39     end
```

Algorithm 15: Algorithm SumGraphBuildOptimize

```
Input  : Data Graph $G$, graph pattern query $Q$ and set $\mathcal{V}$ of materialized
         views on $G$
Output: A summary graph of $Q$ on $G$ (represented by function $cos$ on the
         nodes and edges of $Q$)

1  Procedure cosInitialization(); repeat
2      forwardPruning();
3      backwardPruning();
4  until (until $cos$ has no changes);
5  return $cos$;

6  Procedure cosInitialization():
7      for every node $q \in Q$ do
8          if $Cov(q) \neq \emptyset$ then
9              $cos(q) := \cap_{v \in Cov(q)} cos(v)$ ;
10         else
11             $cos(q) := ms(q)$ ;
12         end
13     end
14     for every edge $e = (q_i, q_j) \in Q$ do
15         if $Cov(e) \neq \emptyset$ then
16             $cos(e) := \cap_{e_v \in Cov(e)} cos(e_v)$ ;
17         else
18             $cos(e) := \emptyset$ ;
19             for every node $n_i \in cos(q_i)$ do
20                 for every node $n_j \in cos(q_j)$ do
21                     if $ReachIndex(n_i, n_j) = true$ then
22                         $cos(e) := cos(e) \cup \{(n_i, n_j)\}$ ;
23                     end
24                 end
25             end
26         end
27     end

28 Procedure forwardPruning():
29     for (each edge $e_q = (q_i, q_j) \in Q$ and each $n_i \in cos(q_i)$) do
30         if (there is no $n_j \in cos(q_j)$ such that $(n_i, n_j) \in cos(e_q)$) then
31             Remove $n_i$ and its associated outgoing edges from $cos(q_i)$ and
                 $cos(e_q)$, respectively;
32         end
33     end

34 Procedure backwardPruning():
35     for (each edge $e_q = (q_i, q_j) \in Q$ and each $n_j \in cos(q_j)$) do
36         if (there is no $n_i \in cos(q_i)$ such that $(n_i, n_j) \in cos(e_q)$) then
37             Remove $n_j$ and its associated incoming edges from $cos(q_j)$ and
                 $cos(e_q)$, respectively;
38         end
39     end
```

6.3 Experimental Evaluation

Algorithms in comparison. We examine the impact that views have on improving query pattern matching algorithm performance times. Just as in Chapter 5, in our implementation of Algorithm *SumGraphBuildOptimize* 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 algorithm *FltSim*, used by the approach presented in Chapter 4 for evaluating hybrid graph pattern queries using homomorphisms over a large graph. Our approach differs from before as we do not fully cover a query using views.

Datasets. We ran experiments on two real-world graph datasets which have been used in previous works [8, 9]. The datasets have different structural properties and come from different application domains, such as the web and social networks. Table 6.1 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 6.1 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 8 graph pattern queries, shown in Figure 6.4. These hybrid query templates involve child and descendant edges. The number associated with each node of a query template denotes the node id. Query instances are generated by assigning labels to nodes.

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 *SumGraphBuildOptimize* 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 *SumGraphBuildOptimize*.

The average number of covering edges for a query edge e_q in Q in a view set $\mathcal{V}$, denoted as $cov_{avg}(Q, \mathcal{V})$, is computed using Equation 5.1 from Chapter 5. 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 all the experiments in this section, 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 13.

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.

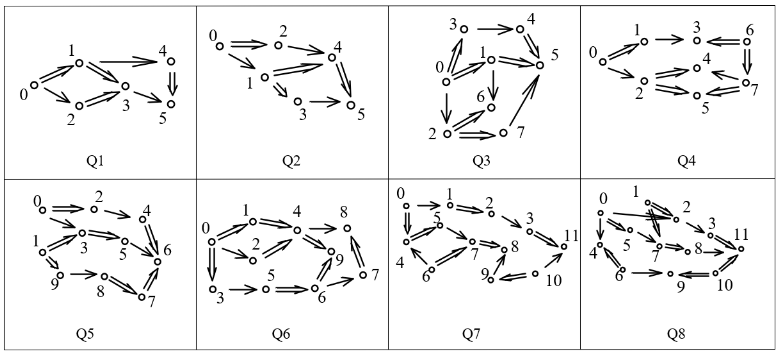


Figure 6.4 Graph pattern queries used in the evaluation.

6.3.1 Benefit of Using Materialized Views

Figure 6.5 displays the elapsed time of *FltSim* versus *MatView* for all the queries of Figure 6.4 on a bs data graph with 450K nodes and 20 labels. For each query, between 70% to 80% of query edges are covered by view edges. The percentage of query edges covered are given at the bottom line of the x-axis in the plots. We observe that for all queries, our algorithm *MatView* is better than the current state of the art algorithm, *FltSim*. In most cases, our algorithm *MatView* has at least a 3x better performance compared to *FltSim*. In the best case, we see *MatView* has an approximately 4x better performance compared to *FltSim*.

For the results run on the dblp data graph with 300K nodes and 20 labels in Figure 6.6, we observe that in most cases, we see our algorithm *MatView* has a 2x better performance compared to *FltSim*. In the best case, we see *MatView* has an approximately 4x better performance compared to *FltSim*

6.3.2 Scalability of Query Coverage using Views

We analyzed the performance of our algorithm *MatView* as we increase the percentage of query edges covered by view edges. Figure 6.7 shows the results for queries Q_4 and Q_5 on the bs data graph with 350K nodes and 20 labels, with the first data point showing the performance on an uncovered graph pattern query. demonstrating that the performance of the algorithm improves as more edges of the query are covered by

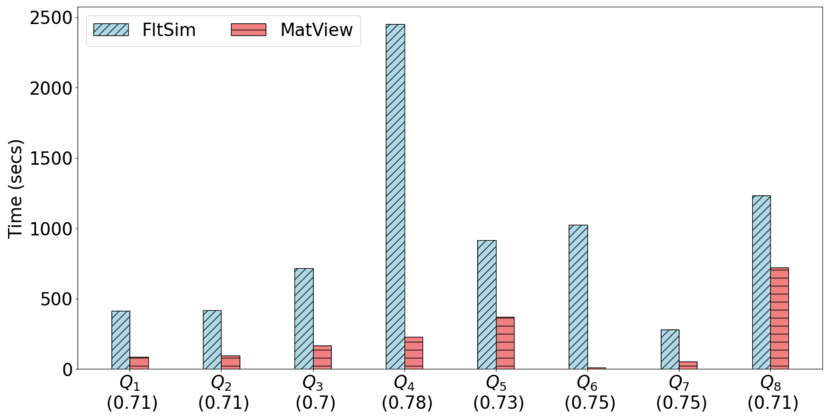


Figure 6.5 Elapsed time of *FltSim* and *MatView* for various queries on a bs data graph with 450K nodes and 20 labels.

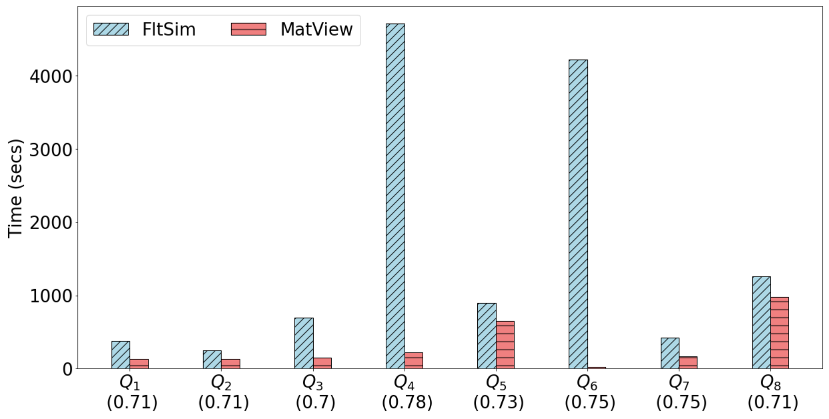
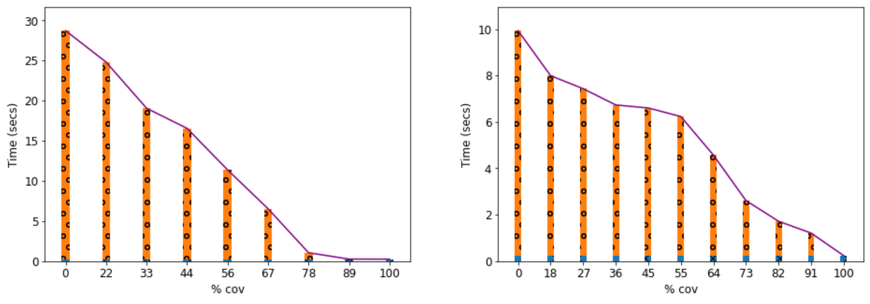
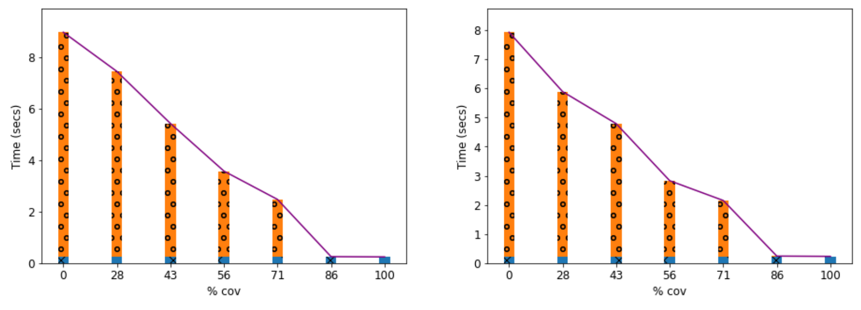


Figure 6.6 Elapsed time of *FltSim* and *MatView* for various queries on a dblp data graph with 300K nodes and 20 labels.



(a) Q_4 (b) Q_5

Figure 6.7 Elapsed time of *FltSim* and *MatView* on query and view sets, each view with 2E edges, that increasingly cover more edges of the query. Run on the bs data graph subset with 350K nodes and 20 labels

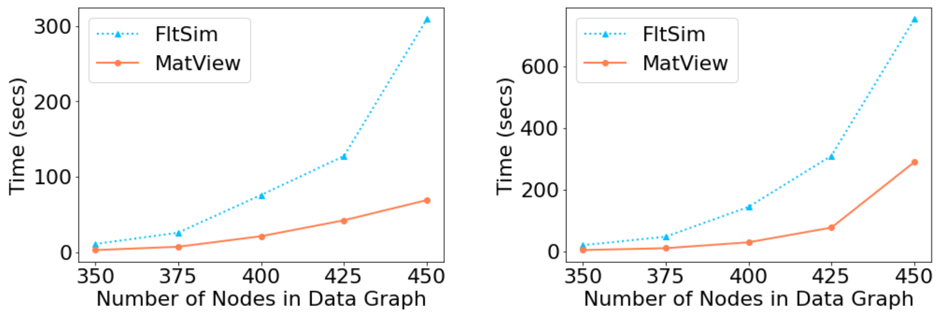


(a) Q_1 (b) Q_2

Figure 6.8 Elapsed time of *FltSim* and *MatView* on query and view sets, each view with 2E edges, that increasingly cover more edges of the query. Run on the dblp data graph subset with 250K nodes and 20 labels

view edges. In Figure 6.8, we also see that the performance of the algorithm improves as more edges are covered by view edges in the case of the dblp data graph with 250K nodes and 20 labels for queries Q_1 and Q_2.

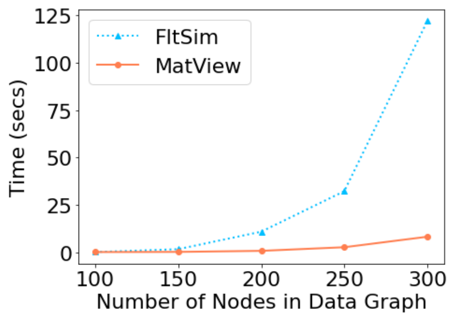
6.3.3 Data Graph Size Scalability



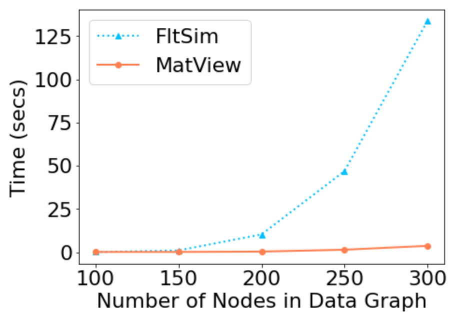
(a) Q_1 (b) Q_5

Figure 6.9 Elapsed time of *FltSim* and *MatView* on increasingly larger number of data subsets of the bs data subset with 20 labels.

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. We expect that since the size of the inverted



(a) Q_4



(b) Q_6

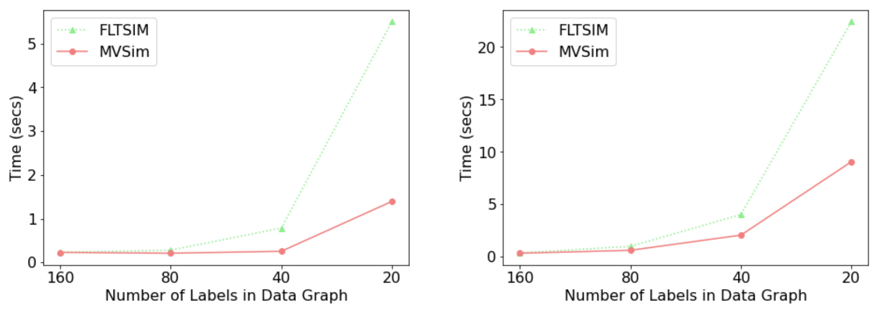
Figure 6.10 Elapsed time of *FltSim* and *MatView* on increasingly larger number of data subsets of the dblp data subset with 20 labels.

list grows as the data graph gets larger, the evaluation time would also increase. The x-axis of our plots shows the size of the data graph in terms of 1k nodes (eg., 100 means 100k).

Figure 6.9 shows the results for queries Q_1 and Q_5 on the bs data graph with 20 labels. Q_1 has 0.71% of its query edges covered, and Q_5 has 0.73% of its query edges covered. Figure 6.10 shows the results for queries Q_4 and Q_6 on the dblp data graph with 20 labels. Q_4 has 0.78% of its query edges covered, and Q_6 has 0.75% of its query edges covered. In both cases of the bs and dblp data graphs, as the execution time for all algorithms increased when the total number of graph nodes increased, *MatView* demonstrated better performance than *FltSim* for evaluating the two queries.

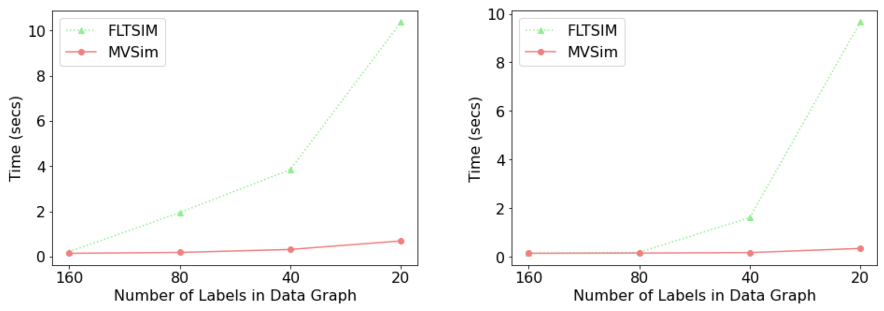
6.3.4 Data Graph Label Scalability

For this experiment, we examine the impact that the total number of distinct data graph labels has on the performance of the algorithms. We first randomly added a specified number of distinct labels to a data graph. To increase the number of labels in the same data graph, we randomly chose half of the existing labels to turn into new labels. Thus, each new data point doubles the number of labels from the previous data point.



(a) Q_7 (b) Q_8

Figure 6.11 Elapsed time of *FltSim* and *MatView* on increasingly fewer number of labels of the bs data subset with 350K nodes.



(a) Q_4 (b) Q_6

Figure 6.12 Elapsed time of *FltSim* and *MatView* on increasingly fewer number of labels of the dblp data subset with 200K nodes.

For these experiments, we used queries with 20 labels at all data points. We expect that the more labels there are in a data graph, the less matches there are, so evaluation time would be faster.

In Figure 6.11, we analyze the performance of the algorithms on increasingly fewer number of data graph labels for the bs data graph subset with 350K nodes for queries Q_7 and Q_8. Q_7 has 0.75% of its query edges covered, and Q_8 has 0.71% of its query edges covered. As the number of labels decreased, *MatView* demonstrated better performance and scalability than *FltSim*.

In Figure 6.12, we analyze the performance of the algorithms on increasingly fewer number of data graph labels for the dblp data graph subset with 200K nodes for queries Q_4 and Q_6. Q_4 has 0.78% of its query edges covered, and Q_6 has 0.75% of its query edges covered. As in the case with the bs data graph, as the number of labels decreased, *MatView* demonstrated better performance and scalability than *FltSim*.

CHAPTER 7

CONCLUSION

This dissertation demonstrates that our methods based on using the summary graph and view materializations outperform state-of-the-art methods for hybrid graph pattern query evaluation. The results show that further work should be developed in the direction of storing local results for use in later cases.

We have addressed the problem of evaluating different types of tree-pattern queries over a large graph. The core of our framework lies at the concept of answer graph proposed to compactly represent pattern matching results. Using the answer graph, we designed efficient algorithms for query counting and query listing tasks. We further designed two holistic algorithms, one that exploits multi-way structural joins on inverted lists of data graph nodes, and one that leverages graph simulation to fully prune redundant data graph nodes, to efficiently build the answer graph. An extensive experimental evaluation verified the efficiency and scalability of our proposed approach, showing that it largely outperforms the state of the art methods.

We have also addressed the problem of efficiently evaluating hybrid graph patterns using homomorphisms over a large data graph. By allowing *edge-to-path* mappings, homomorphisms can extract matches "hidden" deep within large graphs which might be missed by *edge-to-edge* mappings of subgraph isomorphisms. We introduced the concept of the summary graph to compactly encode the pattern matching search space. To further reduce the search space, we designed a novel graph simulation-based node-filtering technique to prune nodes that do not contribute to the final query answer. We have also designed a novel join-based query occurrence enumeration algorithm which leverages multi-way joins realized as intersections of adjacency lists and node sets from the summary graph. Extensive experimental evaluation were conducted to verify the efficiency and scalability of our approach and showed that it largely outperforms state-of-the-art approaches.

We have addressed the problem of answering graph pattern queries exclusively 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 exclusively using views. We designed algorithms for deciding whether a query can be exclusively 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, an approach that does not use materialized views, and provides much better scalability.

Lastly, we have addressed the problem of answering partially graph pattern queries using graph pattern materialized views to efficiently evaluate such queries on large data graphs under homomorphisms. We extended our approach for answering exclusively graph pattern queries using graph pattern materialized views by characterizing the view usability problem in terms of query edge coverage, and provided necessary and sufficient conditions for answering graph pattern queries partially using views. We then designed algorithms for optimizing queries using materialized views. Our approach, for different levels of coverage, was experimentally shown to outperform a state of the art approach that does not use materialized views.

We are currently working on the problem of selecting materialized views for optimizing the graph pattern query evaluation and view maintenance cost in the presence of different types of constraints (e.g., space constraints, view maintenance cost constraints). A variety of algorithms have been used for this problem in the past in the context of relational databases. We are interested to see if neural network algorithms, such as deep reinforcement learning techniques, may be exploited to outperform existing approaches in the framework of graph pattern queries.

REFERENCES

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *International Conference on Very Large Data Bases*, pp. 1804–1815, 2015.

[2] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. Lin, "Real-time twitter recommendation: Online motif detection in large dynamic graphs," *International Conference on Very Large Data Bases Endowment*, vol. 7, no. 13, pp. 1379–1380, 2014.

[3] N. Przulj, D. G. Corneil, and I. Jurisica, "Efficient estimation of graphlet frequency distributions in protein-protein interaction networks," *Bioinformatics*, vol. 22, no. 8, pp. 974–980, 2006.

[4] 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," *Institute of Electrical and Electronics Engineers Association for Computing Machinery Transactions on Computational Biology and Bioinformatics*, vol. 7, no. 2, pp. 197–207, 2010.

[5] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the Association for Computing Machinery*, vol. 23, no. 1, pp. 31–42, 1976.

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *Institute of Electrical and Electronics Engineers Transactions on Pattern Analytics and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[7] B. Bhattarai, H. Liu, and H. H. Huang, "CECI: compact embedding cluster index for scalable subgraph matching," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 1447–1462, 2019.

[8] A. Mhedhbi and S. Salihoglu, "Optimizing subgraph queries by combining binary and worst-case optimal joins," *International Conference on Very Large Data Bases*, pp. 1692–1704, 2019.

[9] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 1083–1098, 2020.

[10] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, "Rapidmatch: A holistic approach to subgraph query processing," *International Conference on Very Large Data Bases*, pp. 176–188, 2020.

[11] A. Mhedhbi, C. Kankanamge, and S. Salihoglu, "Optimizing one-time and continuous subgraph queries using worst-case optimal joins," *Association for Computing Machinery Transactions on Database Systems*, vol. 46, no. 2, pp. 6:1–6:45, 2021.

[12] H. Kim, Y. Choi, K. Park, X. Lin, S. Hong, and W. Han, "Versatile equivalences: Speeding up subgraph query processing and subgraph matching," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 925–937, 2021.

[13] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré, "Emptyheaded: A relational engine for graph processing," *Association for Computing Machinery Transactions on Database Systems*, vol. 42, no. 4, pp. 20:1–20:44, 2017.

[14] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu, "Graph homomorphism revisited for graph matching," *International Conference on Very Large Data Bases*, pp. 1161–1172, 2010.

[15] J. Cheng, J. X. Yu, and P. S. Yu, "Graph pattern matching: A join/semijoin approach," *Institute of Electrical and Electronics Engineers Transactions on Knowledge and Data Engineering*, vol. 23, no. 7, pp. 1006–1021, 2011.

[16] R. Liang, H. Zhuge, X. Jiang, Q. Zeng, and X. He, "Scaling hop-based reachability indexing for fast graph pattern query processing," *Institute of Electrical and Electronics Engineers Transactions on Knowledge and Data Engineering*, vol. 26, no. 11, pp. 2803–2817, 2014.

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

[18] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *The Society for Industrial and Applied Mathematics Journal on Computing*, vol. 32, no. 5, pp. 1338–1355, 2003.

[19] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 813–826, 2009.

[20] J. Su, Q. Zhu, H. Wei, and J. X. Yu, "Reachability querying: Can it be even faster?," *Institute of Electrical and Electronics Engineers Transactions on Knowledge and Data Engineering*, vol. 29, no. 3, pp. 683–697, 2017.

[21] Z. Vagena, M. M. Moro, and V. J. Tsotras, "Twig query processing over graph-structured XML data," *International Workshop on the Web and Databases*, pp. 43–48, 2004.

[22] L. Chen, A. Gupta, and M. E. Kurul, "Stack-based algorithms for pattern matching on dags," *International Conference on Very Large Data Bases*, pp. 493–504, 2005.

[23] Q. Zeng, X. Jiang, and H. Zhuge, "Adding logical operators to tree pattern queries on graph-structured data," *International Conference on Very Large Data Bases*, pp. 728–739, 2012.

[24] J. Zhou, J. X. Yu, Y. Qiu, X. Tang, Z. Chen, and M. Du, "Fast reachability queries answering based on rcn reduction," *Institute of Electrical and Electronics Engineers Transactions on Knowledge and Data Engineering*, pp. 1–1, 2021.

[25] Z. Su, D. Wang, X. Zhang, L. Cui, and C. Miao, "Efficient reachability query with extreme labeling filter," *International Conference on Web Search and Data Mining*, pp. 966–975, 2022.

[26] Q. Zeng and H. Zhuge, "Comments on "stack-based algorithms for pattern matching on dags"," *International Conference on Very Large Data Bases*, pp. 668–679, 2012.

[27] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *International Conference on Very Large Data Bases*, pp. 788–799, 2012.

[28] J. Cheng, X. Zeng, and J. X. Yu, "Top-k graph pattern matching over large graphs," *International Conference on Data Engineering* , pp. 1033–1044, 2013.

[29] W. Han, J. Lee, and J. Lee, "Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 337–348, 2013.

[30] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 1199–1214, 2016.

[31] M. Han, H. Kim, G. Gu, K. Park, and W. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 1429–1446, 2019.

[32] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *International Conference on Very Large Data Bases*, pp. 364–375, 2008.

[33] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 405–418, 2008.

[34] V. Bonnici, R. Giugno, A. Pulvirenti, D. E. Shasha, and A. Ferro, "A subgraph isomorphism algorithm and its application to biochemical data," *BioMed Central Bioinformatics*, vol. 14, no. S-7, p. S13, 2013.

[35] V. Carletti, P. Foggia, and M. Vento, "VF2 plus: An improved version of VF2 for biological graphs," *International Workshop on Graph-Based Representations in Pattern Recognition*, pp. 168–177, 2015.

[36] L. Zou, L. Chen, M. T. Özsu, and D. Zhao, "Answering pattern match queries in large graph databases via graph embedding," *International Conference on Very Large Data Bases*, pp. 97–120, 2012.

[37] M. Yannakakis, "Algorithms for acyclic database schemes," *International Conference on Very Large Data Bases*, pp. 82–94, 1981.

[38] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal XML pattern matching," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 310–321, 2002.

[39] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," *The Institute of Electrical and Electronics Engineers Symposium on Foundations of Computer Science*, pp. 453–462, 1995.

[40] S. Mennicke, J. Kalo, D. Nagel, H. Kroll, and W. Balke, "Fast dual simulation processing of graph database queries," *International Conference on Data Engineering* , pp. 244–255, 2019.

[41] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: From intractable to polynomial time," *International Conference on Very Large Data Bases*, pp. 264–275, 2010.

[42] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu, "Adding regular expressions to graph reachability and pattern queries," *International Conference on Data Engineering* , pp. 39–50, 2011.

[43] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Strong simulation: Capturing topology in graph pattern matching," *Association for Computing Machinery Transactions on Database Systems*, vol. 39, no. 1, pp. 4:1–4:46, 2014.

[44] X. Wu, D. Theodoratos, D. Skoutas, and M. Lan, "Leveraging double simulation to efficiently evaluate hybrid patterns on data graphs," *International Conference on Web Information Systems Engineering*, pp. 255–269, 2020.

[45] A. Y. Halevy, "Answering queries using views: A survey," *International Conference on Very Large Data Bases*, pp. 270–294, 2001.

[46] M. Lenzerini, "Data integration: A theoretical perspective," *Association for Computing Machinery's Special Interest Group on Algorithms and Computation Theory - Association for Computing Machinery's Special Interest Group on Management of Data - Association for Computing Machinery's Special Interest Group on Artificial Intelligence Symposium on Principles of Database Systems*, pp. 233–246, 2002.

[47] X. Wu, D. Theodoratos, and W. H. Wang, "Answering XML queries using materialized views revisited," *The Conference on Information and Knowledge Management*, pp. 475–484, 2009.

[48] W. Fan, X. Wang, and Y. Wu, "Answering pattern queries using views," *Institute of Electrical and Electronics Engineers Transactions on Knowledge and Data Engineering*, vol. 28, no. 2, pp. 326–341, 2016.

[49] J. M. F. da Trindade, K. Karanasos, C. Curino, S. Madden, and J. Shun, "Kaskade: Graph views for efficient graph analytics," *International Conference on Data Engineering* , pp. 193–204, 2020.

[50] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou, "Structured materialized views for XML queries," *International Conference on Very Large Data Bases*, pp. 87–98, 2007.

[51] J. Wang and J. X. Yu, "XPath rewriting using multiple views," *International Conference on Database and Expert Systems Applications*, pp. 493–507, 2008.

[52] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong, "Multiple materialized view selection for XPath query rewriting," *International Conference on Data Engineering*, pp. 873–882, 2008.

[53] X. Wu, D. Theodoratos, W. H. Wang, and T. Sellis, "Optimizing XML queries: Bitmapped materialized views vs. indexes," *Information Systems*, vol. 38, no. 6, pp. 863–884, 2013.

[54] J. Wang, N. Ntarmos, and P. Triantafillou, "Indexing query graphs to speedup graph query processing," *International Conference on Extending Database Technology*, pp. 41–52, 2016.

[55] J. Li, Y. Cao, and X. Liu, "Approximating graph pattern queries using views," *The Conference on Information and Knowledge Management*, pp. 449–458, 2016.

[56] X. Wang, "Answering graph pattern matching using views: A revisit," *International Conference on Database and Expert Systems Applications*, pp. 65–80, 2017.

[57] X. Wu, D. Theodoratos, D. Skoutas, and M. Lan, "Evaluating mixed patterns on large data graphs using bitmap views," *International Conference on Database Systems for Advanced Applications*, pp. 553–570, 2019.

[58] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang, "Rewriting queries on SPARQL views," *The World Wide Web Conference*, pp. 655–664, 2011.

[59] D. Olteanu and M. Schleich, "Factorized databases," *Association for Computing Machinery's Special Interest Group on Management of Data Conference Record*, vol. 45, no. 2, pp. 5–16, 2016.

[60] T. Neumann and G. Weikum, “The RDF-3X engine for scalable management of RDF data,” *The Very Large Data Bases Journal*, pp. 91–113, 2010.

[61] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, “A distributed graph engine for web scale RDF data,” *International Conference on Very Large Data Bases Endowment*, pp. 265–276, 2013.

[62] O. Kalinsky, Y. Etsion, and B. Kimelfeld, “Flexible caching in trie joins,” *International Conference on Extending Database Technology*, pp. 282–293, 2017.

[63] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, “Covering indexes for branching path queries,” *Association for Computing Machinery’s Special Interest Group on Management of Data Conference*, pp. 133–144, 2002.

[64] P. A. Bernstein and D. W. Chiu, “Using semi-joins to solve relational queries,” *Journal of the Association for Computing Machinery*, vol. 28, no. 1, pp. 25–40, 1981.

[65] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, “Adopting worst-case optimal joins in relational database systems,” *International Conference on Very Large Data Bases*, pp. 1891–1904, 2020.

[66] “Awesome Procedures On Cypher (apoc).” https://neo4j.com/labs/apoc/, last accessed on 10/27/2022.

[67] M. Hotz, T. Chondrogiannis, L. Wörteler, and M. Grossniklaus, “Experiences with implementing landmark embedding in neo4j,” *International Workshop on Graph Data Management Experiences and Systems*, pp. 7:1–7:9, 2019.

[68] X. Wu, D. Theodoratos, D. Skoutas, and M. Lan, “Efficient in-memory evaluation of reachability graph pattern queries on data graphs,” *International Conference on Database Systems for Advanced Applications*, pp. 55–71, 2022.

[69] T. Milo and D. Suciu, “Index structures for path expressions,” *The International Conference on Database Theory*, pp. 277–295, 1999.

[70] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra, “Join processing for graph patterns: An old dog with new tricks,” *International Workshop on Graph Data Management Experiences and Systems*, pp. 2:1–2:8, 2015.

[71] H. Q. Ngo, C. Ré, and A. Rudra, “Skew strikes back: new developments in the theory of join algorithms,” *Association for Computing Machinery’s Special Interest Group on Management of Data Conference Rec.*, vol. 42, no. 4, pp. 5–16, 2013.

[72] T. L. Veldhuizen, “Triejoin: A simple, worst-case optimal join algorithm,” *The International Conference on Database Theory*, pp. 96–106, 2014.

[73] T. Neumann and B. Radke, "Adaptive optimization of very large join queries," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 677–692, 2018.

[74] S. Han, L. Zou, and J. X. Yu, "Speeding up set intersections in graph algorithms using SIMD instructions," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 1587–1602, 2018.

[75] Y. Park, S. Ko, S. S. Bhowmick, K. Kim, K. Hong, and W. Han, "G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching," *Association for Computing Machinery's Special Interest Group on Management of Data Conference*, pp. 1099–1114, 2020.