Answering Pattern Queries Using Views

Wenfei Fan [1,2] Xin Wang [3,*] Yinghui Wu [4]
[1]University of Edinburgh [2]RCBD and SKLSDE Lab, Beihang University
[3]Southwest Jiaotong University [4]Washington State University
wenfei@inf.ed.ac.uk, xinwang@swjtu.cn, yinghui@eecs.wsu.edu

Abstract—Answering queries using views has proven effective for querying relational and semistructured data. This paper investigates this issue for graph pattern queries based on graph simulation. We propose a notion of *pattern containment* to characterize graph pattern matching using graph pattern views. We show that a pattern query can be answered using a set of views *if and only if* it is contained in the views. Based on this characterization, we develop efficient algorithms to answer graph pattern queries. We also study problems for determining (minimal, minimum) containment of pattern queries. We establish their complexity (from cubic-time to NP-complete) and provide efficient checking algorithms (approximation when the problem is intractable). In addition, when a pattern query is not contained in the views, we study maximally contained rewriting to find approximate answers; we show that it is in cubic-time to compute such rewriting, and present a rewriting algorithm. We experimentally verify that these methods are able to efficiently answer pattern queries on large real-world graphs.

Index Terms—Graph pattern queries, Pattern containment, Views.

1 Introduction

Answering queries using views has been extensively studied for relational data [27], [33], XML [30], [50], [51] and semistructured data [11], [43], [52]. Given a query Q and a set $\mathcal{V}$ = $\{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ of views, the idea is to find another query A such that A is equivalent to Q, and A only refers to views in $\mathcal{V}$ [27]. If such a query A exists, then given a database D, one can compute the answer $Q(D)$ to Q in D by using A, which uses only the data in the materialized views $\mathsf{V}_i(D)$, *without accessing* D. This is particularly effective when D is "big" and/or distributed. Indeed, views have been advocated for *scale independence*, to query big data "independent of" the size of the underlying data [8], [17]. They are also useful in data integration [33], data warehousing, semantic caching [14], and access control [16].

The need for studying this problem is even more evident for answering graph pattern queries (*a.k.a.* graph pattern matching) [18], [28]. Graph pattern queries have been increasingly used in social network analysis [10], [18], among other things. Real-life social graphs are typically large, and are often distributed. For example, Facebook has more than 1.26 billion users with 140 billion links [46], and the data is geo-distributed to various data centers [26]. One of the major challenges for social network analysis is how to cope with the sheer size of real-life social graphs when evaluating graph pattern queries. Graph pattern matching using views provides an effective method to query such big data.

Example 1: A fraction of a recommendation network is depicted as a graph G in Fig. 1 (a), where each node denotes a person with name and job title (*e.g.,* project manager (PM), database administrator (DBA), programmer (PRG), business analyst (BA) and software tester (ST)); and each edge indicates collaboration/recommendation relation, *e.g.,* (Bob, Dan) indicates that Dan worked well with Bob, on a project led by Bob.

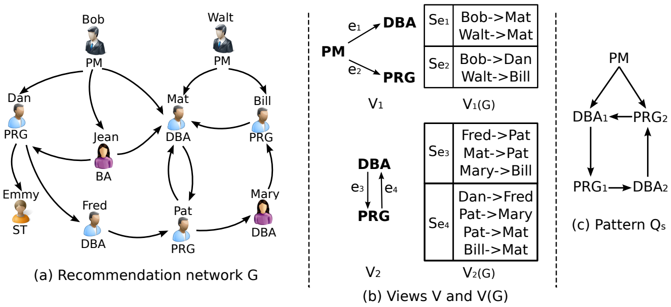


Fig. 1: Data graph, views and pattern queries

To build a team, one issues a pattern query $\mathsf{Q_s}$ depicted in Fig. 1 (c), to find a group of PM, DBA and PRG. It requires that (1) DBA_1 and PRG_2 worked well under the project manager PM; and (2) each PRG (resp. DBA) had been supervised by a DBA (resp. PRG), represented as a collaboration cycle [31] in $\mathsf{Q_s}$. For pattern matching based on *graph simulation* [18], [47], the answer $\mathsf{Q_s}(G)$ to $\mathsf{Q_s}$ in G can be denoted as a set of pairs (e, S_e) such that for each pattern edge e in $\mathsf{Q_s}$, S_e is a set of edges (a match set) for e in G. For example, pattern edge $(\mathsf{PM}, \mathsf{PRG}_2)$ has a match set $S_e = \{(\mathsf{Bob}, \mathsf{Dan}), (\mathsf{Walt}, \mathsf{Bill})\}$, in which each edge satisfies the node labels and connectivity constraint of the pattern edge.

It is known that it takes $O(|\mathsf{Q_s}|^2 + |\mathsf{Q_s}||G| + |G|^2)$ time to compute $\mathsf{Q_s}(G)$ [18], [28], where $|G|$ (resp. $|\mathsf{Q_s}|$) is the size of G (resp. $\mathsf{Q_s}$). For example, to identify the match set of each pattern edge $(\mathsf{DBA}_i, \mathsf{PRG}_i)$ (for $i \in [1, 2]$), each pair of (DBA, PRG) in G has to be checked, and moreover, a number of *join* operations have to be performed to eliminate invalid matches. This is a daunting cost when G is big. One can do better by leveraging a set of *views*. Suppose that a set of views $\mathcal{V} = \{\mathsf{V}_1, \mathsf{V}_2\}$ is defined, materialized and cached ($\mathcal{V}(G) = \{\mathsf{V}_1(G), \mathsf{V}_2(G)\}$), as shown in Fig. 1 (b). As will be shown later, to compute $\mathsf{Q_s}(G)$, (1) we only need to visit views in $\mathcal{V}(G)$, *without* accessing the original big graph G; and (2) $\mathsf{Q_s}(G)$ can be efficiently computed by "merging" views in $\mathcal{V}(G)$. Indeed, the views $\mathcal{V}(G)$ already contains partial answers to $\mathsf{Q_s}$ in G: for each pattern edge e in

[*] *Corresponding author.*

$\mathsf{Q_s}$, the matches of e (*e.g.,* $(\mathsf{DBA}_1, \mathsf{PRG}_1)$) are contained either in $\mathsf{V}_1(G)$ or $\mathsf{V}_2(G)$ (*e.g.,* the matches of e_3 in V_2). These partial answers can be used to construct the complete match $\mathsf{Q_s}(G)$. Hence, the cost of computing $\mathsf{Q_s}(G)$ is in quadratic time in $|\mathsf{Q_s}|$ and $|\mathcal{V}(G)|$, where $\mathcal{V}(G)$ is *much smaller than* $|G|$. □

This example suggests that we conduct graph pattern matching by capitalizing on available views. To do this, several questions have to be settled. (1) How to decide whether a pattern query $\mathsf{Q_s}$ can be answered by a set $\mathcal{V}$ of views? (2) If so, how to efficiently compute $\mathsf{Q_s}(G)$ from $\mathcal{V}(G)$? (3) If not, how to find approximate answers to $\mathsf{Q_s}(G)$ by using $\mathcal{V}(G)$? (4) In both cases, which views in $\mathcal{V}$ should we choose to (approximately) answer $\mathsf{Q_s}$?

Contributions. This paper investigates these questions for answering *graph pattern queries* using *graph pattern views*. We focus on graph pattern matching defined in terms of *graph simulation* [28], since it is commonly used in social community detection [10], biological analysis [35], and mobile network analyses [24]. While conventional subgraph isomorphism often fails to capture meaningful matches, graph simulation fits into emerging applications with its "many-to-many" matching semantics [10], [18], [28]. Moreover, it is more *challenging* since graph simulation is "recursively defined" and has poor data locality [15].

(1) To characterize when graph pattern queries can be answered using views based on graph simulation, we propose a notion of *pattern containment* (Section 3). It extends the traditional notion of query containment [6] to deal with *a set of views*. Given a pattern query $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions, we show that $\mathsf{Q_s}$ can be answered using $\mathcal{V}$ *if and only if* $\mathsf{Q_s}$ is contained in $\mathcal{V}$.

Based on the characterization, we provide an evaluation algorithm for answering graph pattern queries using views (Section 3). Given $\mathsf{Q_s}$ and a set $\mathcal{V}(G)$ of views on a graph G, the algorithm computes $\mathsf{Q_s}(G)$ in $O(|\mathsf{Q_s}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time, *without accessing* G at all when $\mathsf{Q_s}$ is contained in $\mathcal{V}$. It is far less costly than $O(|\mathsf{Q_s}|^2 + |\mathsf{Q_s}||G| + |G|^2)$ for evaluating $\mathsf{Q_s}$ directly on G [18], [28], since G is typically *much larger* than $\mathcal{V}(G)$ in practice.

(2) To decide which views in $\mathcal{V}$ to use when answering $\mathsf{Q_s}$, we identify three fundamental problems for pattern containment (Section 4). Given $\mathsf{Q_s}$ and $\mathcal{V}$, (a) the *containment problem* is to decide whether $\mathsf{Q_s}$ is contained in $\mathcal{V}$, (b) *minimal containment* is to identify a subset of $\mathcal{V}$ that *minimally contains* $\mathsf{Q_s}$, and (c) *minimum containment* is to find a minimum subset of $\mathcal{V}$ that contains $\mathsf{Q_s}$. We show that the first two problems are in cubic-time, whereas the third one is NP-complete and hard to approximate (APX-hard).

The results are also useful for *query minimization*. Indeed, when $\mathcal{V}$ contains a single view, the containment problem becomes the classical query containment problem [6].

These results are a nice surprise. Recall that even for relational SPC (*a.k.a.* conjunctive) queries, the problem of query containment is NP-complete [6]; for XPath fragments, it is EXPTIME-complete or even undecidable [41]. In contrast, (minimal) containment for graph pattern queries is in low PTIME, although the queries may be "recursively defined" (as cyclic patterns).

(3) We develop efficient algorithms for checking (minimal, minimum) pattern containment (Section 5). For containment and minimal containment, we provide cubic-time algorithms in the sizes of *query* $\mathsf{Q_s}$ and *view definitions* $\mathcal{V}$, which are *much smaller* than graph G in practice. For minimum containment, we provide an efficient approximation algorithm with performance guarantees.

(4) When exact answers of a query $\mathsf{Q_s}$ cannot be computed using views $\mathcal{V}$, *i.e.,* when $\mathsf{Q_s}$ is not contained in $\mathcal{V}$, one wants to find the maximal part of $\mathsf{Q_s}$ that can be answered using $\mathcal{V}$. We study the problem of *maximally contained rewriting* (Section 6). A query $\mathsf{Q_s}'$ is a maximally contained rewriting of $\mathsf{Q_s}$ if (a) it is a subquery of $\mathsf{Q_s}$, (b) it is contained in $\mathcal{V}$, and (c) $\mathsf{Q_s}'$ is not a subquery of any larger contained rewriting of $\mathsf{Q_s}$. We show that a maximally contained rewriting $\mathsf{Q_s}'$ of $\mathsf{Q_s}$ *w.r.t.* $\mathcal{V}$ can be found in cubic-time, by presenting such an algorithm. This provides us with *a query-driven approximation scheme*, by treating $\mathsf{Q_s}'(G)$ as approximate query answers to $\mathsf{Q_s}$ in a big graph G. Alternatively, one can compute exact answers $\mathsf{Q_s}(G)$ by using $\mathsf{Q_s}'(G)$ and additionally, accessing a small fraction of G, along the same lines as the scale independence approach suggested in [17].

(5) Using real-life graphs (Amazon, YouTube, Citation and WebGraph), we experimentally verify the effectiveness, efficiency and accuracy of our view-based matching method (Section 7). We find that this method is 23.2 times faster than conventional methods for pattern queries on WebGraph [3], a Web graph with 118.1 million nodes (web pages) and 1.02 billion edges (hyperlinks). In addition, our matching algorithm scales well with data size and pattern size; and our algorithms for (minimal, minimum) pattern containment tests take 0.15 second on complex (cyclic) patterns. We further find that our algorithm can compute maximally contained rewriting $\mathsf{Q_s}'$ efficiently, and that the query results of $\mathsf{Q_s}'$ on $\mathcal{V}(G)$ has accuracy of 0.73 (F-measure) on average on WebGraph.

The work is a first step toward understanding graph pattern matching using views, from theory to practical methods. We contend that the method is effective: one may pick and cache previous query results, and efficiently answer pattern queries using the views, *without* accessing large social graphs directly. If a query $\mathsf{Q_s}$ is not contained in a set of views, one can either adjust the views or approximately answer $\mathsf{Q_s}$ by making use of a maximally contained rewriting of $\mathsf{Q_s}$. Better still, incremental methods are already in place to efficiently maintain cached pattern views (*e.g.,* [20]). The view-based method can be readily *combined* with existing distributed, compression and incremental techniques, and yield a promising approach to querying "big" social data.

Related Work. This work extends [21] by including new proofs, results and experimental study: (1) proofs for the pattern containment characterization (Section 3); (2) proofs of the fundamental problems for pattern containment (Section 4); (3) algorithms contain and minimum (Section 5); (4) results and proofs for maximally contained rewriting for graph pattern matching (Section 6), a topic not studied in [21]; and (5) two sets of new experiments (Section 7): one for evaluating the effectiveness of our approach using graphs with billions of nodes and edges [3], and the other for the efficiency and accuracy of approximate query answering by means of maximally contained rewriting.

We categorize other related work as follows.

Query answering and rewriting. There are two view-based approaches for query processing: query rewriting and query answering [27], [33]. Given a query Q and a set $\mathcal{V}$ of views, (1) query rewriting is to reformulate Q into an equivalent query Q' in a fixed language such that for all D, $Q(D) = Q'(D)$, and moreover, Q' refers only to $\mathcal{V}$; and (2) query answering is to compute $Q(D)$ by evaluating a query A equivalent to Q, while A refers only to $\mathcal{V}$ and its extensions $\mathcal{V}(D)$. While the former requires that Q' is in a fixed language, the latter imposes no constraint on A.

We next review previous work on these issues for relational databases, XML data and general graphs.

(1) Relational data. Query processing using views has been extensively studied for relational data (see [6], [27], [33] for surveys). It is known that for SPC (conjunctive) queries, query answering and rewriting using views are intractable [27], [33]. For the containment problem, the well-known homomorphism theorem shows that an SPC query is contained in another if and only if there exists a homomorphism between the tableaux representing the queries, and it is NP-complete to determine the existence of such a homomorphism [6]. Moreover, the containment problem is undecidable for relational algebra [6].

(2) XML. There has also been a host of work on processing XML queries using views [39], [41], [44]. In [39], the containment of simple XPath queries is shown coNP-complete. When disjunction, DTDs and variables are taken into account, the problem ranges from coNP-complete to EXPTIME-complete to undecidable for various XPath classes [41]. In [7], containment and query rewriting of XML queries are studied under constraints expressed as a structural summary. For tree pattern queries (a fragment of XPath), [30] and [50] have studied maximally contained rewriting.

(3) *Semistructure data.* Views defined in Lorel are studied in, *e.g.,* [52], which are quite different from graph patterns considered here. View-based query rewriting for regular path queries (RPQs) is shown PSPACE-complete in [11], and an EXPTIME rewriting algorithm is given in [43]. The containment problem is shown undecidable for RPQs under constraints [25] and for extended conjunctive RPQs [9].

(4) *RDF.* An EXPTIME query rewriting algorithm is given in [32] for SPARQL. It is shown in [13] that query containment is in EXPTIME for PSPARQL, which supports regular expressions. There has also been work on evaluating SPARQL queries on RDF based on cached query results [14].

Our work differs from the prior work in the following. (1) We study query answering using views for graph pattern queries via graph simulation, which are quite different from previous settings, from complexity bounds to processing techniques. (2) We show that the containment problem for the pattern queries is in PTIME, in contrast to its intractable counterparts for *e.g.,* SPC, XPath, RPQs and SPARQL. (3) We study a more general form of query containment between a query $\mathsf{Q_s}$ and *a set of* queries, to identify an equivalent query for $\mathsf{Q_s}$ that is not necessarily a pattern query. (4) The high complexity of previous methods for query answering using views hinders their applications in the real world. In contrast, our algorithms have performance guarantees and yield a practical method for querying real-life social networks.

Pattern queries on big graphs. There have been a host of techniques for graph pattern queries via simulation on "big" and/or distributed graphs. We next review some of them.

(1) Distributed graph simulation [22], [23], [37]. Several algorithms are in place for distributed graph simulation, by following the synchronized message passing strategy [23] of Pregel [38], scheduling message passing across different fragments, and by integrating (incremental) partial evaluation, partitioned parallelism and message passing [22]; performance guarantees on data shipment and response time are provided in [22].

(2) Graph compression. To query "big" graphs, query-preserving compression [19] and graph summarization [40] have been proposed to reduce the search space by converting a big G to a smaller graph G_c, and evaluate queries on G_c without decompression [19].

(3) Incremental view maintenance. As real-life graphs are updated frequently, techniques for incremental graph simulation have been developed [20] with complexity measured in the size of changes to the input and output, independent of the size of the original big graphs. These allow us to efficiently maintain graph pattern views.

(4) Bounded evaluation. A class of access constraints, a characterization and algorithms have been developed in [12], which allow us to decide whether a pattern query Q can be answered by accessing a small fraction G_Q of a big graph G under the access constraints, and if so, to compute $Q(G)$ by accessing G_Q only. The methods work for both graph simulation and subgraph isomorphism.

This work can be naturally combined with distributed, compression and incremental techniques. For example, view-based techniques can be employed for local evaluation of graph simulation in the distributed algorithm of [22]; views can be cached for simulation-preserving compressed graphs of [19] instead of the original graphs G, which are only 43% of the size of G on average; and the incremental techniques of [20] can be used to efficiently maintain views when graphs are updated. Moreover, maximally contained views can be combined with access constraints [12] to compute exact query answers following [17]. Taken together, these methods yield a promising approach to querying "big" graphs.

The techniques of this work can be readily extended to various revisions of graph simulation such as bounded simulation (reported in [21]), dual simulation and strong simulation [36] (see discussion in Section 8). Due to the space constraints, we only report our findings about graph simulation in this paper.

2 Graphs, Patterns and Views

We first review pattern queries and graph simulation. We then state the problem of pattern matching using views.

2.1 Data Graphs and Graph Pattern Queries

Data graphs. A *data graph* is a directed graph $G = (V, E, L)$, where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$, in which (v, v') denotes an edge from node v to v'; and (3) L is a function such that for each node v in V, $L(v)$ is a set of labels from an alphabet Σ. Intuitively, L specifies the attributes of a node, *e.g.,* name, keywords, blogs and social roles [31].

Pattern queries [18]. A graph pattern query, denoted as $\mathsf{Q_s}$, is a directed graph $\mathsf{Q_s} = (V_p, E_p, f_v)$, where (1) V_p and E_p are the set of *pattern nodes* and the set of *pattern edges*, respectively; and (2) f_v is a function defined on V_p such that for each node $u \in V_p$, $f_v(u)$ is a label in Σ. We remark that f_v can be readily extended to specify search conditions in terms of Boolean predicates [18] (see Figure 8 for examples of search conditions).

Graph pattern matching. We say that a data graph $G = (V, E, L)$ *matches* a graph pattern query $\mathsf{Q_s} = (V_p, E_p, f_v)$ *via simulation*, denoted by $\mathsf{Q_s} \trianglelefteq G$, if there exists a binary relation $S \subseteq V_p \times V$, refereed to as a *match* in G for $\mathsf{Q_s}$, such that

- for each pattern node $u \in V_p$, there exists a node $v \in V$ such that $(u, v) \in S$, referred to as a *match* of u; and
- for each pair $(u, v) \in S$, (a) $f_v(u) \in L(v)$; and moreover, (b) for each pattern edge $e = (u, u')$ in E_p, there exists an edge (v, v') in E, referred to as a *match* of e in S, such that $(u', v') \in S$, *i.e.,* v' is a match of u'.

When $\mathsf{Q_s} \trianglelefteq G$, there exists a *unique maximum* match S_o in G for $\mathsf{Q_s}$ [28]. We derive $\{(e, S_e) \mid e \in E_p\}$ from S_o, where S_e is the set of all matches of e in S_o, referred to as *the match set* of e. Here S_e is *nonempty* for all $e \in E_p$. We define the *result* of $\mathsf{Q_s}$ in G, denoted as $\mathsf{Q_s}(G)$, to be the unique maximum set $\{(e, S_e) \mid e \in E_p\}$ if $\mathsf{Q_s} \trianglelefteq G$, and let $\mathsf{Q_s}(G) = \emptyset$ otherwise.

We define the size of query $\mathsf{Q_s}$, denoted by $|\mathsf{Q_s}|$, to be the total number of nodes and edges in $\mathsf{Q_s}$; we define the size $|\mathsf{Q_s}(G)|$ of $\mathsf{Q_s}(G)$ to be the total edge number of sets S_e for all edges in $\mathsf{Q_s}$.

Example 2: Consider pattern query $\mathsf{Q_s}$ shown in Fig. 1 (c), where each node carries a search condition (job title), and each edge indicates a collaboration relationship. When $\mathsf{Q_s}$ is posed on the network G of Fig. 1 (a), the result $\mathsf{Q_s}(G)$ is shown in the table below:

Edge	Matches
(PM, DBA$_1$)	{(Bob, Mat), (Walt, Mat)}
(PM, PRG$_2$)	{(Bob, Dan), (Walt, Bill)}
(DBA$_1$, PRG$_1$) (DBA$_2$, PRG$_2$)	{(Fred, Pat), (Mat,Pat), (Mary, Bill)}
(PRG$_1$, DBA$_2$) (PRG$_2$, DBA$_1$)	{(Dan, Fred), (Pat, Mary), (Pat, Mat), (Bill, Mat)}

More specifically, (1) both Bob and Walt match pattern node PM as they satisfy its search condition; similarly, Fred, Mat, Mary match DBA, and Dan, Pat, Bill match PRG; (2) pattern edge (PM, DBA$_1$) in $\mathsf{Q_s}$ has two matches in G; and (3) query edges (DBA$_1$, PRG$_1$) and (DBA$_2$, PRG$_2$) (resp. (PRG$_1$, DBA$_2$) and (PRG$_2$, DBA$_1$)) have the same matches. □

To simplify the discussion, we consider *w.l.o.g.* graph patterns $\mathsf{Q_s}$ that are *connected*, as commonly found in real life. That is, for any nodes u and u' in Q, there is an *undirected path* between u and u', by treating $\mathsf{Q_s}$ as an undirected graph. One can easily verify the following, by a straightforward induction on the number of edges in $\mathsf{Q_s}$, based on the definition of graph simulation.

Lemma 1: *For any connected pattern $\mathsf{Q_s}$ and any graph G, if $\mathsf{Q_s}(G) = \emptyset$, then $S_e = \emptyset$ for all e in $\mathsf{Q_s}$.* □

2.2 Graph Pattern Matching Using Views

We next formulate the problem of graph pattern matching using views. We study *views* V defined as a graph pattern query, and refer to the query result $\mathsf{V}(G)$ in a data graph G as the *view extension* for V in G or simply as a *view* [27].

Given a pattern query $\mathsf{Q_s}$ and a *set* $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ of view definitions, *graph pattern matching using views* is to find another query A such that for *all* data graphs G, (1) A only refers to views $\mathsf{V}_i \in \mathcal{V}$ and their extensions $\mathcal{V}(G) = \{\mathsf{V}_1(G), \ldots, \mathsf{V}_n(G)\}$ in G, and (2) A is equivalent to $\mathsf{Q_s}$, *i.e.,* $A(\mathcal{V}(G)) = \mathsf{Q_s}(G)$, where $A(\mathcal{V}(G))$ denotes the matching result of A over $\mathcal{V}(G)$). If such a query A exists, we say that $\mathsf{Q_s}$ *can be answered using views* $\mathcal{V}$.

In contrast to query rewriting using views [27], here A is *not* required to be a pattern query [33]. For example, Figure 1 (b) depicts a set $\mathcal{V} = \{\mathsf{V}_1, \mathsf{V}_2\}$ of view definitions and their extensions $\mathcal{V}(G) = \{\mathsf{V}_1(G), \mathsf{V}_2(G)\}$. To answer the query $\mathsf{Q_s}$ (Fig. 1 (c)), we want to find a query A that computes $\mathsf{Q_s}(G)$ by using only $\mathcal{V}$ and $\mathcal{V}(G)$, where A is not necessarily a graph pattern.

For a set $\mathcal{V}$ of view definitions, we define the size $|\mathcal{V}|$ of $\mathcal{V}$ to be the total size of V_i's in $\mathcal{V}$, and the cardinality $\mathsf{card}(\mathcal{V})$ of $\mathcal{V}$ to be the number of view definitions in $\mathcal{V}$.

The notations of the paper are summarized in Table 1.

Remark. Our techniques also work on graphs and queries with edge labels. Indeed, an edge-labeled graph can be converted to a node-labeled graph: for each edge e, add a "dummy" node carrying the label of e, along with two unlabeled edges.

symbols	notations
$\mathsf{Q_s} = (V_p, E_p, f_v)$	graph pattern query
$\mathsf{Q_s}(G)$	query result of $\mathsf{Q_s}$ in G
$\mathcal{V} = (\mathsf{V}_1, \ldots, \mathsf{V}_n)$	a set of view definitions V_i
$\mathcal{V}(G) = (\mathsf{V}_1(G), \ldots, \mathsf{V}_n(G))$	a set of view extensions $\mathsf{V}_i(G)$
$\mathsf{Q_s} \trianglelefteq G$	simulation
$\mathsf{Q_s} \sqsubseteq \mathcal{V}$	$\mathsf{Q_s}$ is contained in $\mathcal{V}$
$\mathsf{Q_s}' \subseteq \mathsf{Q_s}$	$\mathsf{Q_s}'$ is a subgraph of $\mathsf{Q_s}$
$M_{\mathsf{V}}^{\mathsf{Q_s}}$	view match from a view V to $\mathsf{Q_s}$
$|\mathsf{Q_s}|$ (resp. $|\mathsf{V}|$)	size (total number of nodes and edges) of $\mathsf{Q_s}$ (resp. view definition V)
$|\mathsf{Q_s}(G)|$	total number of edges in sets S_e for all edges e in $\mathsf{Q_s}$
$|\mathcal{V}|$	total size of view definitions in $\mathcal{V}$
$\mathsf{card}(\mathcal{V})$	the number of view definitions in $\mathcal{V}$

TABLE 1: A summary of notations

3 A Characterization

In this section, we propose a characterization of graph pattern matching using views, *i.e.,* a *sufficient and necessary* condition for deciding whether a pattern query can be answered by using *a set of views*. We also provide a quadratic-time algorithm for answering pattern queries using views.

Pattern containment. We first introduce a notion of pattern containment, by extending the traditional notion of query containment to *a set* of views. Consider a pattern query $\mathsf{Q_s} = (V_p, E_p, f_v)$ and a set $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_n\}$ of view definitions, where $\mathsf{V}_i = (V_i, E_i, f_i)$. We say that $\mathsf{Q_s}$ is *contained* in $\mathcal{V}$, denoted by $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, if there exists a mapping λ from E_p to powerset $\mathcal{P}(\bigcup_{i \in [1,n]} E_i)$, such that for all data graphs G, the match set $S_e \subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$ for all edges $e \in E_p$.

The analysis involves query $\mathsf{Q_s}$ and view definitions $\mathcal{V}$, independent of data graphs G and view extensions $\mathcal{V}(G)$.

Example 3: Recall G, $\mathcal{V}$ and $\mathsf{Q_s}$ given in Fig. 1. Then $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. Indeed, there exists a mapping λ from E_p of $\mathsf{Q_s}$ to sets of edges in $\mathcal{V}$, which maps (a) edges (PM, DBA$_1$) and (PM, PRG$_2$) of $\mathsf{Q_s}$ to their counterparts in V_1; (b) both (DBA$_1$, PRG$_1$), (DBA$_2$, PRG$_2$) of $\mathsf{Q_s}$ to e_3, and (c) (PRG$_1$, DBA$_2$) and (PRG$_2$, DBA$_1$) to e_4 in V_2. In any graph G, one may verify that for any edge e of $\mathsf{Q_s}$, its matches are contained in the union of the match sets of the edges in $\lambda(e)$. For instance, the match set of pattern edge (DBA$_1$, PRG$_1$) in G is {(Fred, Pat), (Mat, Pat), (Mary, Bill)}, which is contained in the match set of e_3 of V_2 in G. □

Pattern containment and query answering. The main result of this section is as follows: (1) pattern containment characterizes pattern matching using views; and (2) when $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, for all graphs G, $\mathsf{Q_s}(G)$ can be efficiently computed by using views $\mathcal{V}(G)$ only, *independent of* the size $|G|$ of the underlying graph G. In Sections 4 and 5 we will show how to decide whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by inspecting $\mathsf{Q_s}$ and $\mathcal{V}$ only, also *independent of* $|G|$.

Theorem 2: *(1) A pattern query $\mathsf{Q_s}$ can be answered using $\mathcal{V}$* if and only if *$\mathsf{Q_s} \sqsubseteq \mathcal{V}$. (2) For any graph G, $\mathsf{Q_s}(G)$ can be computed in $O(|\mathsf{Q_s}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$.* □

This suggests an approach to answering graph pattern queries, as follows. Given a pattern $\mathsf{Q_s}$ and a set $\mathcal{V}$ of views, we first efficiently determine whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ (by using the algorithm to be given in Section 5); if so, for all (possibly big) graphs G we compute $\mathsf{Q_s}(G)$ by using $\mathcal{V}(G)$ instead of G, in quadratic-time in the size of $\mathcal{V}(G)$, which is much smaller than G.

Below we prove Theorem 2.

(I) We first prove the **Only If** condition, *i.e.,* if $\mathsf{Q_s}$ can be answered using $\mathcal{V}$, then $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. We show this by contradiction. Assume that $\mathsf{Q_s}$ can be answered using $\mathcal{V}$, while $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$. By the definition of containment, there must exist some data graph G_o such that for all the possible mappings λ, there always exists at least one edge e in $\mathsf{Q_s}$ such that $S_e \not\subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$. Consider the following two cases. (1) When $\mathsf{Q_s}(G_o) = \emptyset$. By Lemma 1, for all e in $\mathsf{Q_s}$, $S_e = \emptyset$ in G_o and hence it contradicts to the assumption that $S_e \not\subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$. (2) When $\mathsf{Q_s}(G_o) \neq \emptyset$. If so, there must exist at least one edge e_o in G_o such that e_o is in S_e for some edge e in $\mathsf{Q_s}$, but it is not in $S_{e'}$ for any $e' \in \lambda(e)$. That is, e_o cannot be included in $S_{e'}$ for any $e' \in \lambda(e)$, for all possible λ. This contradicts the assumption that $\mathsf{Q_s}$ can be answered using only $\mathcal{V}$ and $\mathcal{V}(G_o)$, since at least the edge e_o is missing from $\mathcal{V}(G_o)$ for some graph G_o, no matter how λ is defined. Therefore, $\mathsf{Q_s}$ can be answered using $\mathcal{V}$ *only if* $\mathsf{Q_s} \sqsubseteq \mathcal{V}$.

(II) We next show the **If** condition of Theorem 2(1) with a constructive proof: we give an algorithm to evaluate $\mathsf{Q_s}$ using $\mathcal{V}(G)$, if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. We verify Theorem 2(2) by showing that the algorithm is in $O(|\mathsf{Q_s}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time.

Algorithm. We next present the algorithm that evaluates $\mathsf{Q_s}$ using $\mathcal{V}$. The algorithm, denoted as MatchJoin, is shown in Fig. 2. It takes as input (1) a pattern query $\mathsf{Q_s}$ and a set of view definitions $\mathcal{V} = \{\mathsf{V}_i \mid i \in [1, n]\}$, (2) a mapping λ for $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ (we defer the computation of λ to Section 5); and (3) view extensions $\mathcal{V}(G) = \{\mathsf{V}_i(G) \mid i \in [1, n]\}$. In a nutshell, it computes $\mathsf{Q_s}(G)$ by "merging" (joining) views $\mathsf{V}_i(G)$ as guided by λ. The merge process iteratively identifies and removes those edges that are not matches of $\mathsf{Q_s}$, until *a fixpoint* is reached and $\mathsf{Q_s}(G)$ is correctly computed.

More specifically, MatchJoin works as follows. It starts with empty match sets S_e for each pattern edge e (lines 1-2). MatchJoin sets S_e as $\bigcup_{e' \in \lambda(e)} S_{e'}$, where $S_{e'}$ is extracted from $\mathcal{V}(G)$ (lines 3-4), following the definition of $\lambda(e)$. It then performs a fixpoint computation to remove all invalid matches from S_e (lines 5-10). For each pattern edge $e_p = (u, u')$ with its match set S_{e_p} changed, it checks whether the change propagates to the "parents" (*i.e.,* u'' with edge (u'', u)) of u. That is, it checks whether each match e' of $e = (u'', u)$ still remains to be a match of e (lines 7-10), following the definition of simulation (Section 2.1). More specifically, it checks whether a child u_1 of u'' (resp. a child u_2 of u) has no match as a child v_1 of v'' (resp. a child v_2 of v) in edge $e'_1 = (v'', v_1)$ (resp. $e'_2 = (v, v_2)$). If so, e' is no longer a match of e due to that v'' (resp. v) is invalid match of u'' (resp. u), and is removed from S_e (lines 8,10). In the process, if S_e becomes empty for some edge e, MatchJoin returns $\emptyset$ since $\mathsf{Q_s}$ has no match in G. Otherwise, the process (lines 5-11) proceeds until $\mathsf{Q_s}(G)$ is computed and returned (line 12).

```
Input: A pattern query $\mathsf{Q_s}$, a set of view definitions $\mathcal{V}$
       and their extensions $\mathcal{V}(G)$, a mapping $\lambda$.
Output: The query result $M$ as $\mathsf{Q_s}(G)$.

1.  for each edge $e$ in $\mathsf{Q_s}$ do  $S_e := \emptyset$;
2.  $M := \{(e, S_e) \mid e \in \mathsf{Q_s}\}$;
3.  for each $e \in \mathsf{Q_s}$ do
4.      for each $e' \in \lambda(e)$ do $S_e := S_e \cup S_{e'}$;
5.  while there is change in $S_{e_p}$ for an edge $e_p = (u, u')$ in $\mathsf{Q_s}$ do
6.      for each $e = (u'', u)$ in $\mathsf{Q_s}$ and $e' = (v'', v) \in S_e$ do
7.          if there is $e_1 = (u'', u_1)$ but no $e'_1 = (v'', v_1)$ in $S_{e_1}$ then
8.              $S_e := S_e \setminus \{e'\}$;
9.          if there is $e_2 = (u, u_2)$ but no $e'_2 = (v, v_2)$ in $S_{e_2}$ then
10.             $S_e := S_e \setminus \{e'\}$;
11.         if $S_e = \emptyset$ then return $\emptyset$;
12. return $M = \{(e, S_e) \mid e \in \mathsf{Q_s}\}$, which is $\mathsf{Q_s}(G)$;
```

Fig. 2: Algorithm MatchJoin

Example 4: Consider G, $\mathsf{Q_s}$ and $\mathcal{V}$ shown in Fig. 3. One can verify $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by a mapping λ that maps (AI, Bio), (PM, AI) to e_1, e_2 in V_1, respectively; and (DB, AI), (AI, SE), (SE, DB) to e_3, e_4, e_5 in V_2, respectively. MatchJoin then merges view matches guided by λ, removes $(\mathsf{AI}_1, \mathsf{SE}_1)$ from $S_{(\mathsf{AI},\mathsf{SE})}$, which is an invalid match for (AI, SE) in $\mathsf{Q_s}$. This further leads to the removal of $(\mathsf{SE}_1, \mathsf{DB}_2)$ from $S_{(\mathsf{SE},\mathsf{DB})}$, and $(\mathsf{DB}_2, \mathsf{AI}_2)$ from $S_{(\mathsf{DB},\mathsf{AI})}$. This yields $\mathsf{Q_s}(G)$ shown in the table below, as the final result.

Edge	Matches	Edge	Matches
(PM, AI)	$(\mathsf{PM}_1, \mathsf{AI}_2)$	(AI, Bio)	$(\mathsf{AI}_2, \mathsf{Bio}_1)$
(DB, AI)	$(\mathsf{DB}_1, \mathsf{AI}_2)$	(AI, SE)	$(\mathsf{AI}_2, \mathsf{SE}_2)$
(SE, DB)	$(\mathsf{SE}_2, \mathsf{DB}_1)$		

□

To complete the proof of Theorem 2, we show that (1) MatchJoin correctly evaluates $\mathsf{Q_s}$ using $\mathcal{V}(G)$ if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, and (2) MatchJoin is in $O(|\mathsf{Q_s}||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time.

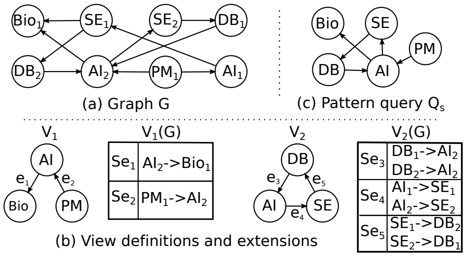
Correctness. For each edge e in $\mathsf{Q_s}$, we denote the match set of e in G as S^*_e when MatchJoin progresses to process $\mathsf{Q_s}$ and $\mathcal{V}(G)$. For the correctness of MatchJoin, it suffices to show the following two invariants it preserves: (1) at any time, for each edge e of $\mathsf{Q_s}$, $S^*_e \subseteq S_e$; and (2) $S_e = S^*_e$ when MatchJoin terminates. For if these hold, then MatchJoin never misses any match or introduces any invalid match when it terminates.

*Proof of **Invariant** (1).* By $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, there exists a mapping λ such that $S_e \subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$. Algorithm MatchJoin takes as input λ, $\mathsf{Q_s}$, $\mathcal{V}$ and $\mathcal{V}(G)$ (Fig. 2). (1) For each edge e in $\mathsf{Q_s}$, it initializes S_e by merging $S_{e'}$ for all $e' \in \lambda(e)$. Hence $S^*_e \subseteq S_e$ due to $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. (2) During the **while** loop (lines 5-10, Fig. 2), MatchJoin repeatedly refines S_e by removing matches that are no longer valid according to the definition of graph simulation. More specifically, for an edge $e_p = (u, u')$ in $\mathsf{Q_s}$ with S_{e_p} changed, the matches $e' = (v'', v) \in S_e$ for all $e = (u'', u)$ in $\mathsf{Q_s}$ become invalid if (a) there is an edge $e_1 = (u'', u_1)$ in $\mathsf{Q_s}$ but there exists no match $(v'', v_1) \in S_{e_1}$ (lines 7-8); or (b) there is an edge $e_2 = (u, u_2)$ in $\mathsf{Q_s}$ but there exists no match $(v, v_2) \in S_{e_2}$ (lines 9-10). Note that (i) both cases indicate that at least a match becomes invalid, and (ii) there exist no other cases that make a match invalid, by the definition of graph simulation. Hence MatchJoin *never* removes a true match, and *never* misses an invalid match by checking the two conditions. Thus, $S^*_e \subseteq S_e$ during the loop (lines 5-10).

*Proof of **Invariant** (2).* When algorithm MatchJoin terminates, either (1) S_e becomes empty (line 11), or (2) no invalid match can be found. Since $S^*_e \subseteq S_e$ during the entire loop (Invariant (1)), if it is case (1), then there exists some edge e such that S^*_e is empty. That is, G does not match $\mathsf{Q_s}$, and MatchJoin returns $\emptyset$ correctly. Otherwise, *i.e.,* in case (2), all invalid matches are removed (lines 7-10), and $S^*_e = S_e$ when MatchJoin terminates.

From the analysis above, the correctness of MatchJoin follows. That is, the **If** condition is verified.

Putting these together, we have shown Theorem 2(1).



(a) Graph G

(c) Pattern query Q_s

(b) View definitions and extensions

Fig. 3: Answering pattern queries using views

Complexity. To complete the proof of Theorem 2(2), we provide a detailed worst-case time complexity analysis for algorithm MatchJoin as follows.

(1) MatchJoin spends $O(|Q_s|)$ time to initialize an empty set M (lines 1-2). It next merges matches in $\mathcal{V}(G)$ via the mapping λ (lines 3-4). Note that the size of $\lambda(e)$ is bounded by $\Sigma_{V \in \mathcal{V}}|V|$. The merge process hence takes in total $O(|Q_s||\mathcal{V}(G)|)$ time.

(2) MatchJoin next iteratively removes invalid matches by conducting a *fixpoint* computation (lines 5-11). Given a match (v'', v) in $\mathcal{V}(G)$, MatchJoin verifies its validity, *i.e.,* whether it carries over to $Q_s(G)$ in the current iteration, in $O(|\mathcal{V}(G)|)$ time; this is because at most $\Sigma_{e_1=(u'',u_1)\in E_p} S_{e_1} + \Sigma_{e_2=(u,u_2)\in E_p} S_{e_2}$ matches have to be inspected, which is bounded by $O(|\mathcal{V}(G)|)$. To speed up the validity checking, MatchJoin employs an index structure $\mathcal{I}$ as a hash-table, which keeps track of a set of key-value pair. Each key is a pair of nodes (u, v), where u is in Q_s and v can match u. Each value corresponding to the key (u, v) is a set of pattern edges and their match set $(e = (u, u_2), S_e)$. The index dynamically maintains the key-value pairs: (1) for each node v, if there exists an edge e emitting from u with $S_e = \emptyset$, then $\mathcal{I}(u, v)$ is set as $\emptyset$, and (2) given a match (v, v_2) of $e = (u, u_2)$, if $\mathcal{I}(u, v)$ or $\mathcal{I}(u_2, v_2)$ is already empty, no further checking is needed, and (v, v_2) can be removed from S_e. Following this, it takes MatchJoin constant time (rather than linear time) to check the validity of a match (lines 7,9).

Observe that when a match is removed from $\mathcal{V}(G)$, it will never be put back, *i.e.,* $\mathcal{V}(G)$ is *monotonically decreasing*. Thus each match in $\mathcal{V}(G)$ is processed at most once. Note that if an edge e in G appears in different match set S_e, each is considered as a distinct edge match. In addition, the index $\mathcal{I}$ can be initialized in $O(|Q_s||\mathcal{V}(G)|)$ time. As a result, the **while** loop (line 5) and **for** loop (line 6) together are bounded by $O(|\mathcal{V}(G)|^2)$ time. Putting these together, MatchJoin is in $O(|Q_s||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time.

These complete the proof of Theorem 2.

Remark. It takes $O(|Q_s|^2 + |Q_s||G| + |G|^2)$ time to evaluate $Q_s(G)$ directly on G by graph simulation [18]. In contrast, MatchJoin is in $O(|Q_s||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time, *without* accessing G. In practice $\mathcal{V}(G)$ is much smaller than G. Indeed, for *WebGraph* in our experiments (Section 7), only 2 to 7 views are needed to answer Q_s, and the overall size of $\mathcal{V}(G)$ is no more than 11% of the size of the entire *WebGraph*.

Optimization. MatchJoin may visit each S_e multiple times. To reduce unnecessary visits, below we introduce an optimization strategy for MatchJoin. The strategy evaluates Q_s by using *ranks* in Q_s as follows. Given a pattern Q_s, the strongly connected component graph G_{SCC} of Q_s is obtained by collapsing each strongly connected component SCC of Q_s into a single node $s(u)$. The rank $r(u)$ of each node u in Q_s is computed as follows: (a) $r(u) = 0$ if $s(u)$ is a leaf in G_{SCC}, where u is in the SCC $s(u)$; and (b) $r(u) = \max\{(1 + r(u'')) \mid (s(u), s(u'')) \in E_{SCC}\}$ otherwise. Here E_{SCC} is the edge set of the G_{SCC} of Q_s. The rank $r(e)$ of an edge $e = (u'', u)$ in Q_s is set to be $r(u)$.

Bottom-up strategy. We revise MatchJoin by processing edges e in Q_s following an ascending order of their ranks (lines 5-11). One may verify that this "bottom-up" strategy guarantees the following for the number of visits.

Lemma 3: *For all edges $e = (u'', u)$ where u'' and u do not reach non-singleton SCC in Q_s, MatchJoin visits its match set S_e at most once using the bottom-up strategy.* □

Indeed, assume that algorithm MatchJoin visits an edge $e = (u'', u)$ at least twice. Then either MatchJoin does not follow a bottom-up strategy in the rank order, or at least u or u'' reaches a non-singleton SCC in Q_s. In particular, when Q_s is a DAG pattern (*i.e.,* acyclic), MatchJoin visits each match set at most once, and the total visits are bounded by the number of the edges in Q_s. As will be verified in Section 7, MatchJoin with optimization strategy runs 1.66 times faster on *WebGraph* than its counterpart without optimization over cyclic patterns.

4 Pattern Containment Problems

In the next two sections, we study how to determine whether $Q_s \sqsubseteq \mathcal{V}$. Our main conclusion is that there are efficient algorithms for these, with their costs as a function of $|Q_s|$ and $|\mathcal{V}|$, which are typically small in practice, and are *independent of* data graphs and materialized views.

We start with three problems in connection with pattern containment, and establish their complexity. In the next section, we will develop effective algorithms for checking $Q_s \sqsubseteq \mathcal{V}$, and computing mapping λ from Q_s to $\mathcal{V}$.

Pattern containment problem. The *pattern containment problem* is to determine, given a pattern query Q_s and a set $\mathcal{V}$ of view definitions, whether $Q_s \sqsubseteq \mathcal{V}$. The need for studying this problem is evident: Theorem 2 tells us that Q_s can be answered by using views of $\mathcal{V}$ if and only if $Q_s \sqsubseteq \mathcal{V}$.

The result below tells us that $Q_s \sqsubseteq \mathcal{V}$ can be efficiently decided (see Table 1 for $|Q_s|$, $|\mathcal{V}|$, $\mathsf{card}(\mathcal{V})$). We will prove the result in Section 5, by providing a checking algorithm.

Theorem 4: *Given a pattern query Q_s and a set $\mathcal{V}$ of view definitions, it is in $O(\mathsf{card}(\mathcal{V})|Q_s|^2 + |\mathcal{V}|^2 + |Q_s||\mathcal{V}|)$ time to decide whether $Q_s \sqsubseteq \mathcal{V}$ and if so, to compute an associated mapping λ from Q_s to $\mathcal{V}$.* □

A special case of pattern containment is the classical *query containment* problem [6]. Given two pattern queries Q_{s1} and Q_{s2}, the latter is to decide whether $Q_{s1} \sqsubseteq Q_{s2}$, *i.e.,* whether for all graphs G, $Q_{s1}(G)$ is contained in $Q_{s2}(G)$. Indeed, when $\mathcal{V}$ contains only a single view definition Q_{s2}, pattern containment becomes query containment. From this and Theorem 4 the result below immediately follows.

Corollary 5: *The query containment problem for graph pattern queries is in quadratic time.* □

Like for relational queries (see, *e.g.,* [6]), query containment is important in minimizing and optimizing pattern queries. Corollary 5 shows that the analysis for graph patterns Q_{s1} and Q_{s2} is in $O(|Q_{s1}|^2 + |Q_{s2}|^2 + |Q_{s1}||Q_{s2}|)$ time, as opposed to the intractability of its counterpart for relational conjunctive queries.

Minimal containment problem. As shown in Section 3, the complexity of pattern matching using views is dominated by $|\mathcal{V}(G)|$. This suggests that we reduce the number of views used for answering $\mathsf{Q_s}$. Indeed, the less views are used, the smaller $|\mathcal{V}(G)|$ is. This gives rise to *the minimal containment problem*. Given $\mathsf{Q_s}$ and $\mathcal{V}$, it is to find a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ that contains $\mathsf{Q_s}$. That is, (1) $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$, and (2) for any proper subset $\mathcal{V}''$ of $\mathcal{V}'$, $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}''$.

The good news is that the minimal containment problem does not make our lives harder. We will prove the next result in Section 5 by developing a cubic-time algorithm.

Theorem 6: *Given* $\mathsf{Q_s}$ *and* $\mathcal{V}$, *it is in* $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ *time to find a minimal subset* $\mathcal{V}'$ *of* $\mathcal{V}$ *containing* $\mathsf{Q_s}$ *and a mapping* λ *from* $\mathsf{Q_s}$ *to* $\mathcal{V}'$ *if* $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. □

Minimum containment problem. One may also want to find a *minimum* subset $\mathcal{V}'$ of $\mathcal{V}$ that contains $\mathsf{Q_s}$. The *minimum containment* problem, denoted by MMCP, is to find a subset $\mathcal{V}'$ of $\mathcal{V}$ such that (1) $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$, and (2) for any subset $\mathcal{V}''$ of $\mathcal{V}$, if $\mathsf{Q_s} \sqsubseteq \mathcal{V}''$, then $\mathsf{card}(\mathcal{V}') \leq \mathsf{card}(\mathcal{V}'')$.

As will be seen shortly (Examples 6 and 7) and verified by our experimental study, MMCP analysis often finds smaller $\mathcal{V}'$ than views found by algorithm minimal.

MMCP is, however, nontrivial: its decision problem is NP-complete and MMCP is APX-hard. Here APX is the class of problems that allow PTIME algorithms with approximation ratio bounded by a constant (see [49] for APX). Nonetheless, we show that MMCP is approximable within $O(\log|E_p|)$ in low polynomial time, where $|E_p|$ is the number of edges of $\mathsf{Q_s}$. That is, there exists an efficient algorithm that identifies a subset $\mathcal{V}'$ of $\mathcal{V}$ with *performance guarantees* whenever $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ such that $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ and $|\mathsf{card}(\mathcal{V}')| \leq \log(|E_p|)|\mathsf{card}(\mathcal{V}_{\mathsf{OPT}})|$, where $\mathcal{V}_{\mathsf{OPT}}$ is a minimum subset of $\mathcal{V}$ that contains $\mathsf{Q_s}$.

Theorem 7: *The minimum containment problem is (1)* NP*-complete (its decision problem) and* APX*-hard, but (2) it is approximable within* $O(\log|E_p|)$ *in* $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}| + (|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ *time.* □

Proof. We first show Theorem 7(1). We defer the proof of Theorem 7(2) to Section 5, where an approximation algorithm is provided as a constructive proof.

(I) We first show that MMCP is NP-complete. The decision problem of MMCP is to decide, given an integer k, whether there exists a subset $\mathcal{V}'$ of $\mathcal{V}$ such that $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ and $\mathsf{card}(\mathcal{V}') \leq k$. It is in NP since there exists an algorithm that guesses and checks $\mathcal{V}'$ in PTIME (Theorem 4). We next show the NP-hardness by reduction from the NP-complete *set cover problem* (SCP) (cf. [42]).

Given a set X, a collection $\mathcal{U}$ of its subsets and an integer B, SCP is to decide whether there exists a B-element subset $\mathcal{U}'$ of $\mathcal{U}$ that covers X, *i.e.,* $\bigcup_{U \in \mathcal{U}'} = X$. Given such an instance of SCP, we construct an instance of MMCP as follows: (a) for each $x_i \in X$, we create a unique edge e_{x_i} with two distinct nodes u_{x_i} and v_{x_i}; (b) we define a pattern query $\mathsf{Q_s}$ as a graph consisting of all edges e_{x_i} defined in (a); (c) for each subset $U_j \in \mathcal{U}$ and $x_i \in U_j$, we define a corresponding view definition V_j that consists of all edges e_{x_i} from U_j; and (d) we set $k = B$.

The construction is obviously in PTIME. We next verify that there exists $\mathcal{U}'$ with size no more than B if and only if there exists $\mathcal{V}'$ of size no more than k that contains $\mathsf{Q_s}$.

(1) Assume that there exists a subset $\mathcal{U}'$ of $\mathcal{U}$ that covers X with size less than B. Let $\mathcal{V}'$ be the set of view definitions V_j corresponding to $U_j \in \mathcal{U}'$. One can verify that $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$, since there exists a mapping λ that maps E_p of $\mathsf{Q_s}$ to powerset $\mathcal{P}(\bigcup_{\mathsf{V}_j \in \mathcal{V}'} E_j)$, such that for any data graph G, $S_e \subseteq \bigcup_{e' \in \lambda(e)} S_{e'}$ for all edges $e \in E_p$. Moreover, $\mathsf{card}(\mathcal{V}') = |\mathcal{U}'| \leq B = k$.

(2) Conversely, if there exists $\mathcal{V}' \subseteq \mathcal{V}$ that contains $\mathsf{Q_s}$ with no more than k view definitions, it is easy to see that the corresponding set $\mathcal{U}'$ is a set cover with at most B elements.

As SCP is known to be NP-complete, so is MMCP.

(II) A problem is APX-hard if every APX problem can be reduced to it by PTIME approximation preserving reductions (AFP-reduction [49]). An AFP-reduction from a (minimization) problem Π_1 to another Π_2 is characterized by a function pair (f, g), where (a) for any instance I_1 of Π_1, $I_2 = f(I_1)$ is an instance of Π_2 such that $\mathsf{opt}_2(I_2) \leq \mathsf{opt}_1(I_1)$, where function $\mathsf{opt}_1()$ (resp. $\mathsf{opt}_2()$) measures the quality of an optimal solution to I_1 (resp. I_2), and (b) for any solution s_2 of I_2, $s_1 = g(I_1, s_2)$ is a solution of I_1 such that $\mathsf{obj}_1(I_1, s_1) \leq \mathsf{obj}_2(I_2, s_2)$, where function $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) measures the quality of a solution to I_1 (resp. I_2). If a problem Π_1 is APX-hard, then Π_2 is APX-hard if there is an AFP-reduction from Π_1 to Π_2.

The APX-hardness of MMCP is verified by AFP-reduction from the minimum set cover (also denoted as SCP), the optimization version of SCP, which is known to be APX-hard (cf. [49]).

(1) We first define a function f. Given an instance I_1 of the SCP as its input, f outputs an instance I_2 of the MMCP following the same transformation in (I). Here $\mathsf{opt}_2(I_2) \leq \mathsf{opt}_1(I_1)$, where $\mathsf{opt}_1()$ (resp. $\mathsf{opt}_2()$) denotes the size of the minimum set cover (resp. the minimum view definition set) that covers X (resp. $\mathsf{Q_s}$). It is easy to see that function f is in PTIME.

(2) We then construct function g. Given a feasible solution $\mathcal{V}'$ for the instance I_2, g outputs a corresponding $\mathcal{U}'$ following the construction given in (1) above. Here $\mathsf{obj}_1()$ (resp. $\mathsf{obj}_2()$) measures the cardinality of the solution $\mathcal{U}'$ to I_1 (resp. $\mathcal{V}'$ to I_2). Note that g is trivially in PTIME.

We now show that (f, g) is an AFP-reduction from the SCP to MMCP. It suffices to show that (a) $\mathsf{opt}_2(I_2) \leq \mathsf{opt}_1(I_1)$, and that (b) $\mathsf{obj}_1(I_1, s_1) \leq \mathsf{obj}_2(I_2, s_2)$. Indeed, the construction guarantees an one-to-one mapping from the elements in a set cover for I_1 to the view definitions in a view definition set for I_2. Thus, $\mathsf{opt}_2(I_2) = \mathsf{opt}_1(I_1)$, and $\mathsf{obj}_1(I_1, s_1) = \mathsf{obj}_2(I_2, s_2)$. Hence, (f, g) is indeed an AFP-reduction. It is known that SCP is APX-hard (cf. [49]); hence MMCP is also APX-hard. □

5 Determining Pattern Containment

In this section, we prove Theorems 4, 6 and 7(2) by providing effective (approximation) algorithms for checking pattern containment, minimal containment and minimum containment in Sections 5.1, 5.2 and 5.3, respectively.

5.1 Pattern Containment

We start with a proof of Theorem 4, *i.e.,* whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ can be decided in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time. To do this, we first propose a *sufficient and necessary* condition to characterize pattern containment. We then develop a cubic-time algorithm based on the characterization.

Sufficient and necessary condition. To characterize pattern containment, we introduce a notion of *view matches*.

Consider a pattern query $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions. For each $\mathsf{V} \in \mathcal{V}$, let $\mathsf{V}(\mathsf{Q_s}) = \{(e_\mathsf{V}, S_{e_\mathsf{V}}) \mid e_\mathsf{V} \in \mathsf{V}\}$, by treating $\mathsf{Q_s}$ as a *data graph*. Obviously, if $\mathsf{V} \trianglelefteq \mathsf{Q_s}$, then S_{e_V} is the nonempty match set of e_V for each edge e_V of V (see Section 2.1). We define the *view match* from V to $\mathsf{Q_s}$, denoted by $M_\mathsf{V}^{\mathsf{Q_s}}$, to be the union of S_{e_V} for all e_V in V.

The result below shows that view matches yield a characterization of pattern containment.

Proposition 8: *For view definitions $\mathcal{V}$ and pattern $\mathsf{Q_s}$ with edge set E_p, $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ if and only if $E_p = \bigcup_{\mathsf{V}\in\mathcal{V}} M_\mathsf{V}^{\mathsf{Q_s}}$.* □

Proof. (I) We first prove the **If** condition. Assume that $E_p = \bigcup_{\mathsf{V}\in\mathcal{V}} M_\mathsf{V}^{\mathsf{Q_s}}$, *i.e.,* the union of all the view matches from $\mathcal{V}$ "covers" E_p. We show that $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by constructing a mapping λ from E_p to the edges in $\mathcal{V}$, such that for all data graphs G and all edges e in $\mathsf{Q_s}$, $S_e \subseteq \bigcup_{e'\in\lambda(e)} S_{e'}$.

We construct a mapping λ as a "reversed" view matching relation: for each edge e_p of $\mathsf{Q_s}$, $\lambda(e_p)$ is a set of edges e' from the view definitions in $\mathcal{V}$, such that for each edge e' of a view definition $\mathsf{V} \in \mathcal{V}$, if $e' \in \lambda(e_p)$, then e_p is a match of e' in the view match $M_\mathsf{V}^{\mathsf{Q_s}}$ of V in $\mathsf{Q_s}$.

We next show that λ ensures $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. For *any* data graph G, (i) if $\mathsf{Q_s}(G) = \emptyset$, then $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by definition; (ii) otherwise, for each pattern edge e_p of $\mathsf{Q_s}$, there exists at least one edge e as a match of e_p in G via simulation. Moreover, for any edge e' (of view V) in $\lambda(e_p)$, e_p is in turn a match of e' via simulation. One can verify that any match e of e_p in G is also a match of $e' \in \lambda(e_p)$ in V. To see this, note that (a) e is a match of e_p; as a result, for any edge e'_p adjacent to e_p, there exists an edge e'' adjacent to e such that e'' is a match of e'_p, by the semantics of simulation (Section 2); and (b) e_p is a match of e'; hence similar to the argument for (a), for any edge e'_a adjacent to e' in a view definition V, one can see that there exists an edge e'_p adjacent to e_p such that e'_p is a match of e'_a, by the semantics of graph pattern matching via graph simulation. From (a) and (b) it follows that e is a match of e' in the view extension. Hence, given *any* match e of e_p from $\mathsf{Q_s}$ in G, there exists an edge e' in $\lambda(e_p)$ from a view definition V, such that e is also a match of e' in view extension $\mathsf{V}(G)$. That it, λ guarantees that $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, by definition.

(II) For the **Only If** condition, assume by contradiction that $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ but $E_p \neq \bigcup_{\mathsf{V}\in\mathcal{V}} M_\mathsf{V}^{\mathsf{Q_s}}$. To simplify the discussion, assume *w.l.o.g.* that each node in $\mathsf{Q_s}$ has a distinct label. Then when taking $\mathsf{Q_s}$ as a data graph G, it can be verified that (i) $\mathsf{Q_s}(G)$ is $\{(e_p,\ S_{e_p} = \{e_p\}) \mid e_p \in E_p\}$; and (ii) there exists a mapping λ from each edge e_p of $\mathsf{Q_s}$ to an edge e_i in some view of $\mathcal{V}$, such that $\{e_p\} \subseteq S_{e_i}$. This is ensured by the definition of $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. On the other hand, (i) $\bigcup_{\mathsf{V}\in\mathcal{V}} M_\mathsf{V}^{\mathsf{Q_s}} \subseteq E_p$, since all the edges in view matches are from E_p; and (ii) $E_p \neq \bigcup_{\mathsf{V}\in\mathcal{V}} M_\mathsf{V}^{\mathsf{Q_s}}$ (by assumption). Hence $E_p \neq \bigcup_{\mathsf{V}\in\mathcal{V}} M_\mathsf{V}^{G}$, *i.e.,* there exists $e_o \in E_p$ such that $e_o \notin M_\mathsf{V}^{G}$ for all $\mathsf{V} \in \mathcal{V}$. Thus, $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ since $\{e_o\} \not\subseteq S_{e_i}$ for all e_i of $\mathsf{V} \in \mathcal{V}$. Hence the contradiction.

This completes the proof of Proposition 8. □

Algorithm. Following Proposition 8, we present an algorithm, denoted as $\mathsf{contain}$ and shown in Fig. 4, to check whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. Given a pattern query $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions, it returns a boolean value ans that is true if and only if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. The algorithm first initializes an empty edge set E to record view matches from $\mathcal{V}$ (line 1) to $\mathsf{Q_s}$. It then checks the condition of Proposition 8 as follows. (1) Compute view match $M_\mathsf{V}^{\mathsf{Q_s}}$ for each V in $\mathcal{V}$, by invoking the simulation evaluation algorithm in [18]. (2) Extend E with $M_\mathsf{V}^{\mathsf{Q_s}}$, since $M_\mathsf{V}^{\mathsf{Q_s}}$ is a *subset* of E_p (lines 2-3). After all view matches are merged, $\mathsf{contain}$ then checks whether $\mathsf{E} = E_p$. It returns true if so, and false otherwise (lines 4-6).

```
Input: A pattern query $\mathsf{Q_s}$, and a set of view definitions $\mathcal{V}$.
Output: A boolean value ans that is true if and only if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$.
1. E := $\emptyset$;
2. for each view definition $\mathsf{V} \in \mathcal{V}$ do
3.     compute $M_\mathsf{V}^{\mathsf{Q_s}}$; E := E $\cup M_\mathsf{V}^{\mathsf{Q_s}}$;
4. if E = $E_p$ then ans := true;
5. else ans := false;
6. return ans;
```

Fig. 4: Algorithm contain

Proof of Theorem 4. Algorithm $\mathsf{contain}$ provides a constructive proof for Theorem 4. To complete the proof, it remains to verify its correctness and complexity.

<u>*Correctness*</u>. It suffices to show that $\mathsf{contain}$ correctly checks the sufficient and necessary condition given in Proposition 8, *i.e.,* whether the union of all the view matches from $\mathcal{V}$ "covers" E_p. Indeed, (1) $\mathsf{contain}$ correctly computes the view match for each view definition in $\mathcal{V}$, by using an algorithm to compute graph simulation relation [18]; and (2) when $\mathsf{contain}$ halts, it determines whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by checking if the union of the view matches covers E_p, following Proposition 8. The correctness of $\mathsf{contain}$ then follows from the proof for Proposition 8.

<u>*Complexity*</u>. Algorithm $\mathsf{contain}$ iteratively computes *view match* $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ for each view definition $\mathsf{V}_i = (V_i, E_i, f_i)$. It takes $O((|V_p| + |V_i|)(|E_p| + |E_i|))$ time for a single iteration [18], [28]. The **for** loop repeats $\mathsf{card}(\mathcal{V})$ times; hence it takes $\mathsf{contain}$ $\Sigma_{\mathsf{V}_i\in\mathcal{V}}((|V_p| + |V_i|)(|E_p| + |E_i|))$ time in total, which equals $\mathsf{card}(\mathcal{V})|V_p||E_p| + \Sigma_{\mathsf{V}_i\in\mathcal{V}}(|V_p||E_i| + |E_p||V_i|) + \Sigma_{\mathsf{V}_i\in\mathcal{V}}(|V_i||E_i|)$ time. Since $|V_p|$ (resp. $|E_p|$) is bounded by $|\mathsf{Q_s}|$, it can be verified that (1) $\mathsf{card}(\mathcal{V})|V_p||E_p|$ is bounded by $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2)$; (2) $\Sigma_{\mathsf{V}_i\in\mathcal{V}}(|V_p||E_i| + |E_p||V_i|)$ is bounded by $O(|\mathsf{Q_s}||\mathcal{V}|)$ since $\Sigma_{\mathsf{V}_i\in\mathcal{V}}(|E_i| + |V_i|) = |\mathcal{V}|$; and (3) $\Sigma_{\mathsf{V}_i\in\mathcal{V}}(|V_i||E_i|)$ is bounded by $\Sigma_{\mathsf{V}_i\in\mathcal{V}}(|\mathsf{V}_i|^2)$, which is further bounded by $O(|\mathcal{V}|^2)$. Thus, algorithm $\mathsf{contain}$ is in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time.

From these and Proposition 8, Theorem 4 follows. □

Example 5: Consider pattern query $\mathsf{Q_s}$ and a set of view definitions $\mathcal{V} = \{\mathsf{V}_i \mid i \in [1,7]\}$ in Fig. 6. The view matches $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ of V_i for $i \in [1,7]$ are shown in the table below.

V_i	$M_{\mathsf{V}_i}^{\mathsf{Q_s}}$	V_i	$M_{\mathsf{V}_i}^{\mathsf{Q_s}}$
V_1	$\{(C,D)\}$	V_2	$\{(B,E)\}$
V_3	$\{(A,B),(A,C)\}$	V_4	$\{(B,D),(C,D)\}$
V_5	$\{(B,D),(B,E)\}$	V_6	$\{(A,B),(A,C),(C,D)\}$
V_7	$\{(A,B),(A,C),(B,D)\}$		

Given $\mathsf{Q_s}$ and $\mathcal{V}$, $\mathsf{contain}$ returns true since $\bigcup_{\mathsf{V}_i\in\mathcal{V}} M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ is the edge set of $\mathsf{Q_s}$. One can verify that $\mathsf{Q_s} \sqsubseteq \mathcal{V}$. □

Remarks. (1) Algorithm $\mathsf{contain}$ can be easily adapted to return a mapping λ that specifies pattern containment (Section 3), to serve as input for algorithm $\mathsf{MatchJoin}$. This can be done by following the construction given in the proof of Proposition 8. (2) In contrast to regular path queries and relational queries, pattern containment checking is in PTIME.

Approach. Using algorithms $\mathsf{contain}$ and $\mathsf{MatchJoin}$ (Fig. 2), we

```
Input: A pattern query $\mathsf{Q_s}$, and a set of view definitions $\mathcal{V}$.
Output: A subset $\mathcal{V}'$ of $\mathcal{V}$ that minimally contains $\mathsf{Q_s}$.
1.  set $\mathcal{V}' := \emptyset$; $\mathsf{S} := \emptyset$; $\mathsf{E} := \emptyset$; index $\mathsf{M} := \emptyset$;
2.  for each view definition $\mathsf{V}_i \in \mathcal{V}$ do
3.      compute $M^{\mathsf{Q_s}}_{\mathsf{V}_i}$;
4.      if $M^{\mathsf{Q_s}}_{\mathsf{V}_i} \setminus \mathsf{E} \neq \emptyset$ then
5.          $\mathcal{V}' := \mathcal{V}' \cup \{\mathsf{V}_i\}$; $\mathsf{S} := \mathsf{S} \cup \{M^{\mathsf{Q_s}}_{\mathsf{V}_i}\}$; $\mathsf{E} := \mathsf{E} \cup M^{\mathsf{Q_s}}_{\mathsf{V}_i}$;
6.          for each $e \in M^{\mathsf{Q_s}}_{\mathsf{V}_i}$ do $\mathsf{M}(e) := \mathsf{M}(e) \cup \{\mathsf{V}_i\}$;
7.          if $\mathsf{E} = E_p$ then break ;
8.  if $\mathsf{E} \neq E_p$ then return $\emptyset$;
9.  for each $M^{\mathsf{Q_s}}_{\mathsf{V}_j} \in \mathsf{S}$ do
10.     if there is no $e \in M^{\mathsf{Q_s}}_{\mathsf{V}_j}$ such that $\mathsf{M}(e) \setminus \{\mathsf{V}_j\} = \emptyset$ then
11.         $\mathcal{V}' := \mathcal{V}' \setminus \{\mathsf{V}_j\}$; update $\mathsf{M}$;
12. return $\mathcal{V}'$;
```

Fig. 5: Algorithm minimal

answer pattern queries using views as follows. Given a pattern $\mathsf{Q_s}$ and a set $\mathcal{V}$ of views, we first determine whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ by using algorithm contain; if so, for all graphs G, we compute $\mathsf{Q_s}(G)$ by using algorithm MatchJoin. If $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$, we compute approximate answers to $\mathsf{Q_s}$, as will be discussed in Section 6. All these are in time determined by $|\mathsf{Q_s}|$, $|\mathcal{V}|$ and $|\mathcal{V}(G)|$, *not by* the size $|G|$.

5.2 Minimal Containment Problem

We now prove Theorem 6 by presenting an algorithm that, given $\mathsf{Q_s}$ and $\mathcal{V}$, finds a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ containing $\mathsf{Q_s}$ in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$.

Algorithm. The algorithm, denoted as minimal, is shown in Fig. 5. Given a pattern query $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions, it returns either a nonempty subset $\mathcal{V}'$ of $\mathcal{V}$ that minimally contains $\mathsf{Q_s}$, or $\emptyset$ to indicate that $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$.

Algorithm minimal initializes (1) an empty set $\mathcal{V}'$ for selected views, (2) an empty set S for view matches of $\mathcal{V}'$, and (3) an empty set E for edges in view matches. It also maintains an index M that maps each edge e in $\mathsf{Q_s}$ to a set of views (line 1). Similar to algorithm contain, minimal first computes $M^{\mathsf{Q_s}}_{\mathsf{V}_i}$ for all $\mathsf{V}_i \in \mathcal{V}$ (lines 2-7). In contrast to contain that simply merges the view matches, it extends S with a new view match $M^{\mathsf{Q_s}}_{\mathsf{V}_i}$ only if $M^{\mathsf{Q_s}}_{\mathsf{V}_i}$ contains a new edge not in E, and updates M accordingly (lines 4-7). The **for** loop stops as soon as $\mathsf{E} = E_p$ (line 7), as $\mathsf{Q_s}$ is already contained in $\mathcal{V}'$. If $\mathsf{E} \neq E_p$ after the loop, it returns $\emptyset$ (line 8), since $\mathsf{Q_s}$ is not contained is $\mathcal{V}$ (Proposition 8). The algorithm then eliminates redundant views (lines 9-11), by checking whether the removal of V_j causes $\mathsf{M}(e) = \emptyset$ for some $e \in M^{\mathsf{Q_s}}_{\mathsf{V}_j}$ (line 10). If no such e exists, it removes V_j from $\mathcal{V}'$ (line 11). After all view matches are checked, minimal returns $\mathcal{V}'$ (line 12).

Proof of Theorem 6. To complete the proof of Theorem 6, we next provide a detailed correctness and complexity analysis of algorithm minimal (Fig. 5).

Correctness. Given a pattern $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions, minimal either returns an empty set indicating $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$, or a subset $\mathcal{V}'$ of $\mathcal{V}$. We show the correctness of minimal by proving that (1) minimal always terminates, (2) it only removes "redundant" view definitions V' from $\mathcal{V}'$, ensuring $\mathsf{Q_s} \sqsubseteq \mathcal{V} \setminus \{\mathsf{V}'\}$ if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, and (3) when it terminates, no redundant view definition is in $\mathcal{V}'$.

(1) Algorithm minimal repeats the **for** loop (lines 2-7, Fig. 5) at most $\mathsf{card}(\mathcal{V})$ times, and in each iteration it computes view matches and adds a view definition V_i to a result set $\mathcal{V}'$. It then performs the redundant checking (lines 9-11) to remove all redundant view definitions, if there exists any. As $\mathcal{V}'$ is a finite set, and its size is monotonically decreasing, the algorithm always terminates.

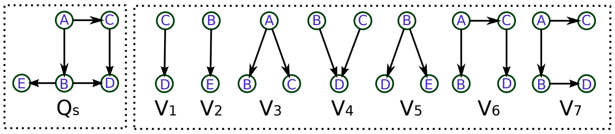


Fig. 6: Containment for pattern queries

(2) We show that minimal only removes "redundant" view definitions. (a) Each time it computes the view match for a view definition V_i (line 3), and it adds V_i to $\mathcal{V}'$ only if the corresponding match set of V_i can cover edges in $\mathsf{Q_s}$ that have not been covered yet (line 4). Hence when the **for** loop terminates, one can verify that either the union of the view matches from $\mathcal{V}'$ covers E_p (line 7), which indicates that $\mathcal{V}'$ contains $\mathsf{Q_s}$, or $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ (line 8), following Proposition 8. (b) A view definition V_j is removed from $\mathcal{V}'$ only when there already exist other view definitions in $\mathcal{V}'$ "covering" every pattern edge $e \in M^{\mathsf{Q_s}}_{\mathsf{V}_j}$ (lines 10-11). Thus, minimal only removes redundant view definitions.

(3) When algorithm minimal terminates with $\mathsf{Q_s} \sqsubseteq \mathcal{V}$, for any view definition V in $\mathcal{V}'$, there exists at least an edge e that can only be introduced by V to cover E_p. By Proposition 8, this indicates that $\mathsf{Q_s} \not\sqsubseteq \mathcal{V} \setminus \{\mathsf{V}\}$ for any $\mathsf{V} \in \mathcal{V}$. Thus minimal returns a minimal set that contains $\mathsf{Q_s}$.

Complexity. Similar to the complexity analysis of contain given above, algorithm minimal takes in total $O(\mathsf{card}(\mathcal{V})\ |\mathsf{Q_s}|^2$ + $|\mathcal{V}|^2$ + $|\mathsf{Q_s}||\mathcal{V}|)$ time to compute all the view matches (line 3, Fig. 5). For each view match, the construction time for the index structure M (line 6) takes in total $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|)$ time (the outer loop is conducted at most $\mathsf{card}(\mathcal{V})$ times). The process for eliminating redundant view definitions (lines 9-11) takes $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|)$ time. Hence, it is in total $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time to find a minimal subset $\mathcal{V}'$ of $\mathcal{V}$ that contains $\mathsf{Q_s}$.

The analysis above completes the proof of Theorem 6. □

Example 6: Consider $\mathsf{Q_s}$ and $\mathcal{V}$ given in Fig. 6. After $M^{\mathsf{Q_s}}_{\mathsf{V}_i}$ ($i \in [1, 4]$) are computed, algorithm minimal finds that E already equals E_p, and breaks the loop, where M is initialized to be $\{((A, B) : \{\mathsf{V}_3\}), ((A, C) : \{\mathsf{V}_3\}), ((B, D) : \{\mathsf{V}_4\}), ((C, D) : \{\mathsf{V}_1, \mathsf{V}_4\}), ((B, E) : \{\mathsf{V}_2\})\}$. As the removal of V_1 does not make any $\mathsf{M}(e)$ empty, minimal removes V_1 and returns $\mathcal{V}' = \{\mathsf{V}_2, \mathsf{V}_3, \mathsf{V}_4\}$ as a minimal subset of $\mathcal{V}$. □

5.3 Minimum Containment Problem

We next prove Theorem 7 (2), *i.e.,* MMCP is approximable within $O(\log |E_p|)$ in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}| + (|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ time. We give such an algorithm following the greedy strategy of the approximation of [49] for the set cover problem. The algorithm of [49] achieves an approximation ratio $O(\log n)$, for an n-element set.

Algorithm. The algorithm is denoted as minimum and shown in Fig. 7. Given a pattern $\mathsf{Q_s}$ and a set $\mathcal{V}$ of views, minimum identifies a subset $\mathcal{V}'$ of $\mathcal{V}$ such that (1) $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ and (2) $\mathsf{card}(\mathcal{V}') \leq \log(|E_p|) \cdot \mathsf{card}(\mathcal{V}_{\mathsf{OPT}})$, where $\mathcal{V}_{\mathsf{OPT}}$ is a minimum subset of $\mathcal{V}$ that contains $\mathsf{Q_s}$. That is, the approximation ratio of minimum is $O(\log |E_p|)$, where $|E_p|$ is typically small.

```
Input: A pattern query $\mathsf{Q_s}$ and a set of view definitions $\mathcal{V}$.
Output: A minimum subset $\mathcal{V}'$ of $\mathcal{V}$ that contains $\mathsf{Q_s}$.
1. set $\mathcal{V}' := \emptyset$, $\mathsf{S} := \emptyset$; $\mathsf{E_c} := \emptyset$;
2. for each view definition $\mathsf{V}_i \in \mathcal{V}$ do
3.     compute $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$; $\mathsf{S} := \mathsf{S} \cup \{M_{\mathsf{V}_i}^{\mathsf{Q_s}}\}$ if $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ is nonempty;
4. while $\mathsf{S} \neq \emptyset$ do
5.     find $\mathsf{V}_i$ with the largest $\alpha(\mathsf{V}_i)$; $\mathsf{S} := \mathsf{S} \setminus \{M_{\mathsf{V}_i}^{\mathsf{Q_s}}\}$;
6.     if $M_{\mathsf{V}_i}^{\mathsf{Q_s}} \setminus \mathsf{E_c} \neq \emptyset$ then
7.         $\mathsf{E_c} := \mathsf{E_c} \cup M_{\mathsf{V}_i}^{\mathsf{Q_s}}$; $\mathcal{V}' := \mathcal{V}' \cup \{\mathsf{V}_i\}$;
8.         if $\mathsf{E_c} = E_p$ then return $\mathcal{V}'$;
9. return $\emptyset$;
```

Fig. 7: Algorithm minimum

The algorithm iteratively finds the "top" view whose view match can cover most edges in $\mathsf{Q_s}$ that are not yet covered. To do this, we define a metric $\alpha(\mathsf{V})$ for a view V, where

$$\alpha(\mathsf{V}) = \frac{|M_{\mathsf{V}}^{\mathsf{Q_s}} \setminus \mathsf{E_c}|}{|E_p|}.$$

Here $\mathsf{E_c}$ is the set of edges in E_p that have been covered by selected view matches, and $\alpha(V)$ indicates the amount of *uncovered* edges that $M_{\mathsf{V}}^{\mathsf{Q_s}}$ covers. We select V with the largest α in each iteration, and maintain α accordingly.

Similar to minimal, algorithm minimum computes the view match $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ for each $\mathsf{V}_i \in \mathcal{V}$, and collects them in a set S (lines 2-3). It then does the following. (1) It selects view V_i with the largest α, and removes $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ from S (line 5). (2) It merges $\mathsf{E_c}$ with $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ if $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ contains some edges that are not in $\mathsf{E_c}$, and extends $\mathcal{V}'$ with V_i (lines 6-7). During the loop, if $\mathsf{E_c}$ equals E_p, the set $\mathcal{V}'$ is returned (line 8). Otherwise, minimum returns $\emptyset$, indicating that $\mathsf{Q_s} \not\sqsubseteq \mathcal{V}$ (line 9).

Proof of Theorem 7 (2). We next provide correctness and complexity analyses of algorithm minimum (Fig. 7).

Correctness. Observe that minimum finds a nonempty $\mathcal{V}'$ with $\mathsf{Q_s} \sqsubseteq \mathcal{V}'$ if and only if $\mathsf{Q_s} \sqsubseteq \mathcal{V}$ (Proposition 8). Its approximation ratio is verified by an approximation-preserving reduction from MMCP to the *set cover problem* [42], by treating each $M_{\mathsf{V}_i}^{\mathsf{Q_s}}$ in S as a subset of E_p. Algorithm minimum extends the algorithm of [49] (with approximation ratio $\log(n)$ for n-element set) to query containment, and preserves approximation ratio $\log|E_p|$.

Complexity. Algorithm minimum computes view matches in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time (lines 1-3). The **while** loop is executed $\min\{|E_p|, \mathsf{card}(\mathcal{V})\} \leq (|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{1/2}$ times. To see this, let's consider following two cases. (1) When $|E_p| \leq \mathsf{card}(\mathcal{V})$, the running time of **while** loop is obviously at most $|E_p|$, since each time we only need to pick a V_i whose view match "covers" at least one distinct edge of E_p. (2) When $\mathsf{card}(\mathcal{V}) \leq |E_p|$, one only needs to check each $\mathsf{V}_i \in \mathcal{V}$, then the loop runs at most $\mathsf{card}(\mathcal{V})$ times. Hence the **while** loop is bounded by $\min\{|E_p|, \mathsf{card}(\mathcal{V})\}$, which is further bounded by $(|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{1/2}$ [29]. Each iteration takes $O(|E_p| \cdot \mathsf{card}(\mathcal{V}))$ time to find a view with the largest α, which is at most $O(|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))$. Thus, minimum is in $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}| + (|\mathsf{Q_s}| \cdot \mathsf{card}(\mathcal{V}))^{3/2})$ time, where $|\mathsf{Q_s}|$ and $\mathsf{card}(\mathcal{V})$ are often smaller than the size $|\mathcal{V}|$.

This completes the proof of Theorem 7 (2). □

Example 7: Given $\mathsf{Q_s}$ and $\mathcal{V} = \{\mathsf{V}_1, \ldots, \mathsf{V}_7\}$ of Figure 6, algorithm minimum selects views based on their α values. More specifically, in the loop it first chooses V_6, as its view match $M_{\mathsf{V}_6}^{\mathsf{Q_s}} = \{(A,B), (A,C), (C,D)\}$ makes $\alpha(\mathsf{V}_6) = 0.6$, the largest one. Then V_6 is followed by V_5, since $\alpha(\mathsf{V}_5) = 0.4$ is the largest one in that iteration. After V_5 and V_6 are picked, algorithm minimum finds that $\mathsf{E_c} = E_p$, and thus $\mathcal{V}' = \{\mathsf{V}_5, \mathsf{V}_6\}$ is returned as a minimum subset that contains $\mathsf{Q_s}$. □

In our approach to answering pattern queries using views described in Section 5.1, we can use minimal or minimum instead of contain to check whether $\mathsf{Q_s} \sqsubseteq \mathcal{V}$.

6 Maximally Contained Rewriting

When a pattern query $\mathsf{Q_s}$ is not contained in a set $\mathcal{V}$ of views, we want to identify a maximal part $\mathsf{Q_s}'$ of $\mathsf{Q_s}$ that can be answered by using $\mathcal{V}$, referred to as a *maximally contained rewriting* of $\mathsf{Q_s}$ using $\mathcal{V}$. As will be seen shortly, given a graph G, $\mathsf{Q_s}'$ helps us approximately answer $\mathsf{Q_s}$ in G, or compute exact answers $\mathsf{Q_s}(G)$ by additionally accessing a small fraction of the data in G.

Maximally contained rewriting. A pattern query $\mathsf{Q_s}'$ is a *subquery* of $\mathsf{Q_s}$, denoted as $\mathsf{Q_s}' \subseteq \mathsf{Q_s}$, if it is an edge induced subgraph of $\mathsf{Q_s}$, *i.e.,* $\mathsf{Q_s}'$ is a subgraph of $\mathsf{Q_s}$ consisting of a subset of edges of $\mathsf{Q_s}$, together with their endpoints as the set of nodes. Query $\mathsf{Q_s}'$ is called a *contained rewriting* of $\mathsf{Q_s}$ using a set $\mathcal{V}$ of view definitions if

- $\mathsf{Q_s}' \subseteq \mathsf{Q_s}$, *i.e.,* $\mathsf{Q_s}'$ is a *subquery* of $\mathsf{Q_s}$, and
- $\mathsf{Q_s}' \sqsubseteq \mathcal{V}$, *i.e.,* $\mathsf{Q_s}'$ can be answered using $\mathcal{V}$.

Such a rewriting $\mathsf{Q_s}'$ is a *maximally contained rewriting* of $\mathsf{Q_s}$ using $\mathcal{V}$ if there exists no contained rewriting $\mathsf{Q_s}''$ such that $\mathsf{Q_s}' \subset \mathsf{Q_s}''$, *i.e.,* there exists no larger contained rewriting $\mathsf{Q_s}''$ with more edges than $\mathsf{Q_s}'$.

Query-driven approximation scheme. When $\mathsf{Q_s}$ is not contained in $\mathcal{V}$, we can still efficiently answer $\mathsf{Q_s}$ in a (possibly big) graph G following two approaches. (1) One may first identify a maximally contained rewriting $\mathsf{Q_s}'$ of $\mathsf{Q_s}$ using $\mathcal{V}$, and then compute $\mathsf{Q_s}'(G)$ as approximate answers to $\mathsf{Q_s}$, by simply invoking the algorithm MatchJoin. (2) Alternatively, one may compute exact answers $\mathsf{Q_s}(G)$ by using $\mathsf{Q_s}'(G)$ and by accessing a small fraction $G_{\mathsf{Q_s}}$ of G, such that $\mathsf{Q_s}(G) = \mathsf{Q_s}'(G) \cup f(G_{\mathsf{Q_s}})$. Here $f(G_{\mathsf{Q_s}})$ first locates the matches of $\mathsf{Q_s}'(G)$ in the original graph G and then verifies the matches for $\mathsf{Q_s}$ by visiting neighborhood of those matches, a small number of nodes and edges in G that constitute $G_{\mathsf{Q_s}}$; this is the approach suggested by [17], referred to as *scale-independent query answering using views* there. Due to the space constraint, we focus on approximate answers $\mathsf{Q_s}'(G)$ in this paper. That is, when limited views are available, we can still approximately answer pattern queries in big graphs by relaxing $\mathsf{Q_s}$ to maximally contained rewriting $\mathsf{Q_s}'$, using those views.

Accuracy. Given a graph G, we measure the quality of the approximate answers $\mathsf{Q_s}'(G)$ versus the true matches in the exact answers $\mathsf{Q_s}(G)$ by following the F-measure [4]:

$$\mathsf{Acc} = 2 \cdot (\mathsf{recall} \cdot \mathsf{precision})/(\mathsf{recall} + \mathsf{precision}),$$

where $\mathsf{recall} = \frac{\#\mathsf{true\_matches\_found}}{\#\mathsf{true\_matches}}$, and $\mathsf{precision} = \frac{\#\mathsf{true\_matches\_found}}{\#\mathsf{matches}}$. Here #matches is the number of all (edge) matches found by $\mathsf{Q_s}'(G)$ using views, #true_matches is the number of all matches in $\mathsf{Q_s}(G)$; and #true_matches_found is the number of all the true matches in both $\mathsf{Q_s}'(G)$ and $\mathsf{Q_s}(G)$.

Intuitively, a high precision means that many matches in $\mathsf{Q_s}'(G)$ are true matches, and a high recall means $\mathsf{Q_s}'(G)$ contains most of the true matches in $\mathsf{Q_s}(G)$. The larger Acc that can be induced by $\mathsf{Q_s}'$, the better. If $\mathsf{Q_s}'$ is equivalent to $\mathsf{Q_s}$, *i.e.,* $\mathsf{Q_s}'(G)$ = $\mathsf{Q_s}(G)$ for all G, Acc takes the maximum value 1.0. Observe that for any edge e in $\mathsf{Q_s}$, if e is covered by $\mathsf{Q_s}'$, then for any G, the match set S_e of e in $\mathsf{Q_s}(G)$ is a subset of the match set S'_e of e in $\mathsf{Q_s}'(G)$; that is, $\mathsf{Q_s}'(G)$ finds all candidate matches of e in G.

Example 8: Recall pattern $\mathsf{Q_s}$ and view definitions V_1–V_7 from Fig. 6. Let $\mathcal{V}' = \{\mathsf{V}_1, \mathsf{V}_3, \mathsf{V}_4, \mathsf{V}_6, \mathsf{V}_7\}$. Then $\mathsf{Q_s}$ is not contained in $\mathcal{V}'$. Nonetheless, a maximally contained rewriting of $\mathsf{Q_s}'$ exists, consisting of four edges $\{(A, B), (A, C), (C, D), (B, D)\}$. Given a graph G, $\mathsf{Q_s}'(G)$ finds matches of these edges, which make a superset of the corresponding edge matches in $\mathsf{Q_s}(G)$. Using $\mathsf{Q_s}'(G)$, one may further verify whether the matches of $\mathsf{Q_s}'(G)$ make true matches in $\mathsf{Q_s}(G)$ by inspecting their neighboring nodes and edges. One may also treat $\mathsf{Q_s}'$ as a "relaxation" of $\mathsf{Q_s}$ by dropping the condition imposed by edge (B, E), and take $\mathsf{Q_s}'(G)$ as approximate answers to $\mathsf{Q_s}$ in G. □

Computing maximally contained rewriting. It is known that finding maximally contained rewriting is intractable for SPC queries [45]. In contrast, maximally contained rewriting can be efficiently found for graph pattern queries.

Theorem 9: *Given a pattern* $\mathsf{Q_s}$ *and a set* $\mathcal{V}$ *of view definitions, it is in* $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ *time to find a maximally contained rewriting of* $\mathsf{Q_s}$ *using* $\mathcal{V}$*.* □

We next prove Theorem 9 by providing an algorithm that computes a maximally contained rewriting for $\mathsf{Q_s}$ using $\mathcal{V}$.

Algorithm. Given a pattern query $\mathsf{Q_s}$ and a set $\mathcal{V}$ of view definitions, the algorithm, denoted as $\mathsf{maximal}$ (not shown) finds a maximally contained rewriting of $\mathsf{Q_s}$ using $\mathcal{V}$ as follows. Similar to algorithm $\mathsf{contain}$, $\mathsf{maximal}$ maintains a set E of all nonempty view matches, initially empty. For each view definition $\mathsf{V} \in \mathcal{V}$, it iteratively computes view match $M_{\mathsf{V}}^{\mathsf{Q_s}}$ and merges it with E, until every view is visited. The difference from $\mathsf{contain}$ is that instead of checking whether E covers all edges in $\mathsf{Q_s}$ as in $\mathsf{contain}$, $\mathsf{maximal}$ simply generates an induced subgraph of $\mathsf{Q_s}$ with edge set E, and returns it as the maximally contained rewriting $\mathsf{Q_s}'$.

Example 9: Given $\mathsf{Q_s}$ of Fig. 6 and $\mathcal{V}'$ from Example 8, $\mathsf{maximal}$ finds a maximally contained rewriting $\mathsf{Q_s}'$ of $\mathsf{Q_s}$ by computing the union of view matches E from each view in $\mathcal{V}'$ to $\mathsf{Q_s}$. One may verify that as E includes a set of edges $\{(A, B), (A, C), (C, D), (B, D)\}$; hence $\mathsf{Q_s}'$, which is a subgraph of $\mathsf{Q_s}$ induced by E, is returned. □

Proof of Theorem 9. Below we prove Theorem 9 by giving a detailed correctness and complexity analysis of $\mathsf{maximal}$.

Correctness. It suffices to show that when algorithm $\mathsf{maximal}$ terminates, $\mathsf{Q_s}'$ is (1) a contained rewriting, and (2) a maximal contained rewriting. Obviously $\mathsf{maximal}$ always terminates since it visits each view in a finite set $\mathcal{V}$ once.

(1) When algorithm $\mathsf{maximal}$ terminates, $\mathsf{Q_s}'$ consists of only the edges of view matches from each view to $\mathsf{Q_s}$. Following Proposition 8, $\mathsf{Q_s}'$, as a graph pattern query, is contained by $\mathcal{V}$, *i.e.,* $\mathsf{Q_s}' \sqsubseteq \mathcal{V}$. Moreover, $\mathsf{maximal}$ preserves the invariant that at any time, $\mathsf{Q_s}'$ contains only the edges from $\mathsf{Q_s}$. Hence $\mathsf{Q_s}' \subseteq \mathsf{Q_s}$. This shows that $\mathsf{Q_s}'$ is a contained rewriting of $\mathsf{Q_s}$. Note that this also holds when $\mathsf{Q_s}'$ is empty.

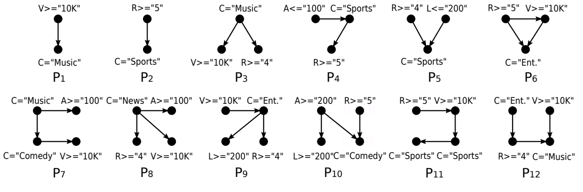


Fig. 8: Youtube views

(2) We next show that $\mathsf{Q_s}'$ is maximal, *i.e.,* there is no contained rewriting $\mathsf{Q_s}''$ of $\mathsf{Q_s}$ using $\mathcal{V}$ such that $\mathsf{Q_s}' \subset \mathsf{Q_s}''$. Assume that such a contained rewriting $\mathsf{Q_s}''$ exists. Then there must exist a view definition V such that $M_{\mathsf{V}}^{\mathsf{Q_s}}$ is in the edge set of $\mathsf{Q_s}''$ but it is not in $\mathsf{Q_s}'$. This cannot happen since algorithm $\mathsf{maximal}$ visits each view in $\mathcal{V}$ including V, and hence $\mathsf{Q_s}'$ includes $M_{\mathsf{V}}^{\mathsf{Q_s}}$ when algorithm $\mathsf{maximal}$ visits V.

Complexity. Algorithm $\mathsf{maximal}$ is the same as algorithm $\mathsf{contain}$ except the last step for constructing $\mathsf{Q_s}'$. It takes $O(\mathsf{card}(\mathcal{V})\ |\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time to compute all the view matches and merge those nonempty matches. The construction of $\mathsf{Q_s}'$ with edge set E takes at most $O(|\mathsf{Q_s}|)$ time. Hence, it is in total $O(\mathsf{card}(\mathcal{V})|\mathsf{Q_s}|^2 + |\mathcal{V}|^2 + |\mathsf{Q_s}||\mathcal{V}|)$ time to compute $\mathsf{Q_s}'$, having the same complexity as $\mathsf{contain}$.

This completes the proof of Theorem 9.

Remark. Note that a mapping from the edges of $\mathsf{Q_s}'$ to views can be readily induced by $\mathsf{maximal}$, to be used as λ in $\mathsf{MatchJoin}$ for answering query using views.

7 Experimental Evaluation

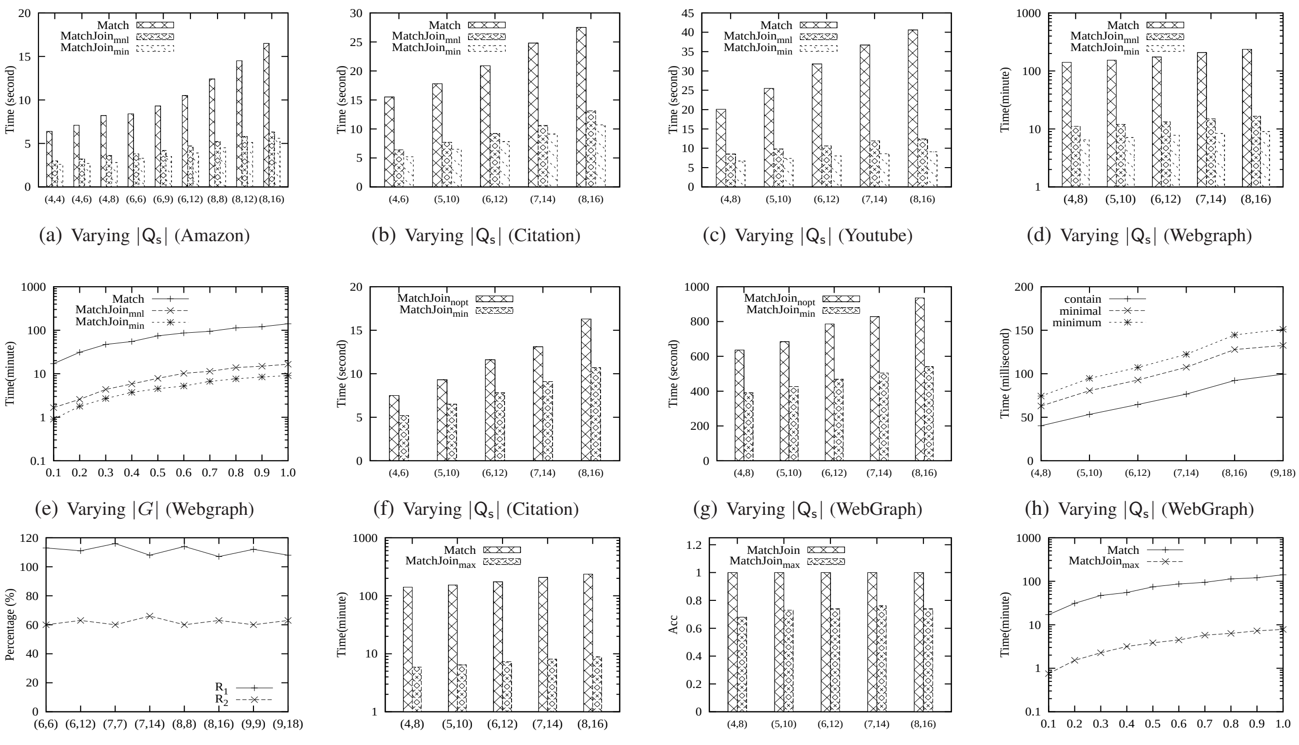
Using real-life data, we conducted four sets of experiments to evaluate (1) the efficiency and scalability of algorithm $\mathsf{MatchJoin}$ for graph pattern matching using views; (2) the effectiveness of optimization techniques for $\mathsf{MatchJoin}$; (3) the efficiency and effectiveness of (minimal, minimum) containment checking; and (4) the efficiency, accuracy and scalability of our query-driven approximation scheme, using maximally contained rewriting.

Experimental setting.

(1) Real-life graphs. We used four real-life graphs: (a) *Amazon* [1], a product co-purchasing network with 548K nodes and 1.78M edges. Each node has attributes such as title, group and sales-rank, and an edge from product x to y indicates that people who buy x also buy y. (b) *Citation* [2], a DAG (directed acyclic graph) with 1.4M nodes and 3M edges, in which nodes represent papers with attributes such as title, authors, year and venue, and edges denote citations. (c) *YouTube* [5], a recommendation network with 1.6M nodes and 4.5M edges. Each node is a video with attributes such as category, age and rate, and each edge from x to y indicates that y is in the related list of x. (d) *WebGraph* [3], a web graph including 118.1M nodes and 1.02B edges, where each node represents a web page with id and domain.

(2) Pattern and view generator. We implemented a generator for graph pattern queries, controlled by three parameters: the number $|V_p|$ of pattern nodes, the number $|E_p|$ of pattern edges, and label f_v from an alphabet Σ of labels taken from corresponding real-life graphs. We use $(|V_p|, |E_p|)$ to denote the size of a pattern query.

We generated a set of 12 view definitions for *each* real-life dataset. (a) For *Amazon*, we generated 12 frequent patterns following [34], where each view extension contains on average $5K$ nodes and edges. The views take 14.4% of the space of the



(a) Varying $|\mathsf{Q_s}|$ (Amazon)

(b) Varying $|\mathsf{Q_s}|$ (Citation)

(c) Varying $|\mathsf{Q_s}|$ (Youtube)

(d) Varying $|\mathsf{Q_s}|$ (Webgraph)

(e) Varying $|G|$ (Webgraph)

(f) Varying $|\mathsf{Q_s}|$ (Citation)

(g) Varying $|\mathsf{Q_s}|$ (WebGraph)

(h) Varying $|\mathsf{Q_s}|$ (WebGraph)

(i) Varying $|\mathsf{Q_s}|$ (WebGraph)

(j) Varying $\mathsf{Q_s}$ (WebGraph)

(k) Varying $\mathsf{Q_s}$ (WebGraph)

(l) Varying $|G|$ (WebGraph)

Fig. 9: Performance evaluation

Amazon dataset. (b) For *Citation*, we designed 12 views to search for papers and authors in computer science. The view extensions account for 12% of the Citation graph. (c) We generated 12 views, shown in Fig. 8, to find videos on *Youtube*, where each node is associated with a Boolean condition, specified by *e.g.,* age (A), length (L), category (C), rate (R) and visits (V). Each view extension has about 700 nodes and edges, accounting for 4% of *Youtube*. (d) On *WebGraph*, we designed 12 views to search Web pages, where the view extensions account for 11% of *WebGraph*.

(3) Implementation. We implemented the following algorithms, all in Java: (1) $\mathsf{contain}$, $\mathsf{minimum}$ and $\mathsf{minimal}$ for checking pattern containment; (2) $\mathsf{maximal}$ for finding the maximally contained rewriting; (3) Match, $\mathsf{MatchJoin_{min}}$ and $\mathsf{MatchJoin_{mnl}}$ for computing matches of patterns in a graph, where Match is the matching algorithm without using views [18], [28]; and $\mathsf{MatchJoin_{min}}$ (resp. $\mathsf{MatchJoin_{mnl}}$) revises $\mathsf{MatchJoin}$ by using a minimum (resp. minimal) set of views; (4) an algorithm $\mathsf{MatchJoin_{max}}$ for approximately answering pattern queries, which invokes $\mathsf{MatchJoin}$ to evaluate maximally contained rewriting using views (Section 6); and (5) a version of $\mathsf{MatchJoin_{min}}$ without using the ranking optimization (Section 3), denoted by $\mathsf{MatchJoin_{nopt}}$.

All the experiments were run on a machine with 2.0GHz Intel Xeon E5-2650 (8-core) CPU and 32GB memory, running windows server 2008 (64bit). Each experiment was run 5 times and the average is reported here.

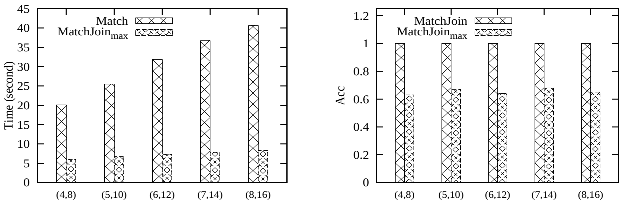
Experimental results. We next present our findings.

Exp-1: Query answering using views. We first evaluated the performance of algorithms $\mathsf{MatchJoin_{min}}$ and $\mathsf{MatchJoin_{mnl}}$, compared to Match [18], [28]. Using real-life data, we studied the efficiency (resp. scalability) of $\mathsf{MatchJoin_{min}}$, $\mathsf{MatchJoin_{mnl}}$ and Match, by varying $|\mathsf{Q_s}|$ (resp. $|G|$).

Efficiency. Figures 9(a), 9(b), 9(c) and 9(d) show the results on *Amazon*, *Citation*, *YouTube* and *WebGraph*, respectively, where the x-axis represents pattern size $(|V_p|, |E_p|)$. The results tell us the following. (1) $\mathsf{MatchJoin_{min}}$ and $\mathsf{MatchJoin_{mnl}}$ substantially outperform Match: they are on average 8.1 and 5.2 times faster than Match over all real-life graphs, respectively. (2) While all the algorithms spend more time on larger patterns, $\mathsf{MatchJoin_{min}}$ and $\mathsf{MatchJoin_{mnl}}$ are less sensitive to patterns than Match, as they reuse previous computation cached in the views. (3) The larger the graphs are, the more substantial improvement of $\mathsf{MatchJoin_{min}}$ and $\mathsf{MatchJoin_{mnl}}$ is over Match. For example, $\mathsf{MatchJoin_{min}}$ (resp. $\mathsf{MatchJoin_{mnl}}$) is 23.2 (resp. 13.3) times faster than Match on *WebGraph*, and 2.7 (resp. 2.3) times faster on smaller *Amazon*.

Scalability. Using *WebGraph*, we evaluated the scalability of $\mathsf{MatchJoin_{min}}$, $\mathsf{MatchJoin_{mnl}}$ and Match. Fixing $|\mathsf{Q_s}| = (4, 6)$, we varied $|G|$ by using scale factors from 0.1 (*i.e.,* 0.1 times of original graph size) to 1.0. The results are reported in Fig. 9(e), from which we can see the following. (1) $\mathsf{MatchJoin_{min}}$ scales best with $|G|$, and is 1.73 times faster than $\mathsf{MatchJoin_{mnl}}$. This verifies that evaluating pattern queries by using less views *significantly reduces* computation time. The results are consistent with the observation of Figures 9(a), 9(b), 9(c) and 9(d).

Exp-2: Optimization techniques. Varying the size of DAG (resp. cyclic) patterns, we evaluated the effectiveness of the optimization strategy given in Section 3, by comparing the performance of $\mathsf{MatchJoin_{min}}$ and $\mathsf{MatchJoin_{nopt}}$ on *Citation* (resp. *WebGraph*). As shown in Figures 9(f) and 9(g), (1) $\mathsf{MatchJoin_{min}}$ is more efficient than $\mathsf{MatchJoin_{nopt}}$ for all the patterns. For example, $\mathsf{MatchJoin_{min}}$ is 1.46 (resp. 1.66) times faster than $\mathsf{MatchJoin_{nopt}}$ for DAG (resp. cyclic) patterns on average. (2) The improvement becomes more substantial when $|\mathsf{Q_s}|$ gets larger. This is because for larger patterns, the bottom-up strategy used in $\mathsf{MatchJoin_{min}}$ can eliminate redundant matches more quickly. (3) The optimization strategy works even better on denser big graphs,



(a) Varying $\mathsf{Q_s}$ (Youtube)

(b) Varying $\mathsf{Q_s}$ (Youtube)

Fig. 10: Performance evaluation

since more invalid matches can be removed by the strategy. This explains why MatchJoin$_{\mathsf{min}}$ works better on *WebGraph* than on *Citation*, since *WebGraph* is denser than *Citation*.

Exp-3: Query containment. We evaluated the efficiency of pattern containment checking *w.r.t.* query complexity.

Efficiency. We generated a set of patterns with size ranging from $(4, 8)$ to $(9, 18)$, and node label from the alphabet Σ of *WebGraph*. Using the same set of views $\mathcal{V}$ as in Fig. 9(d), we evaluated the efficiency of contain, minimal and minimum. As shown in Fig. 9(h), (1) all three algorithms are efficient, *e.g.,* it only takes contain 0.1s to decide whether a pattern with size $(9, 18)$ is contained in $\mathcal{V}$; (2) they all take more time over larger patterns, as expected; and (3) contain accounts for about 68.6% (resp. 59.4%) of the time of minimal (resp. minimum) on average.

Algorithm minimum *vs* minimal. To measure the effectiveness of minimum and minimal, we define and investigate two ratios: $R_1 = |\mathsf{T_{min}}|/|\mathsf{T_{mnl}}|$ as the ratio of the time used by minimum to that of minimal; and $R_2 = |\mathsf{Minimum}|/|\mathsf{Minimal}|$ for the ratio of the size of subsets of views found by minimum to that of minimal. Using the same view definitions and patterns as in Fig. 9(h), we varied the size of patterns from (6,6) to (9, 18). As shown in Fig. 9(i), (1) minimum is efficient on all patterns, *e.g.,* it takes about 0.15s over patterns with size $(9, 18)$; (2) minimum is effective: while minimum takes up to 122% of the time of minimal (R_1), it finds *substantially smaller* sets of views, only about 60%-66% of the size of those found by minimal, as indicated by R_2. Both algorithms take more time over larger patterns, as expected.

Exp-4: Approximate answers. We evaluated the efficiency, scalability and accuracy of MatchJoin$_{\mathsf{max}}$, by using maximally contained rewriting (Section 6) and real-life graphs.

Efficiency. Using the same sets of views as in Figures 9(c) and 9(d), we generated two sets of patterns, where none of them is contained in the corresponding view set. Varying $|\mathsf{Q_s}|$, we evaluated the efficiency of MatchJoin$_{\mathsf{max}}$ and find the following. (1) maximal is efficient. For example, it takes less than 50ms to find a maximally contained rewriting for a pattern with 8 nodes and 16 edges (not shown). (2) As shown in Figures 9(j) and 10(a), MatchJoin$_{\mathsf{max}}$ substantially outperforms Match in running time: it is on average 24.8 (resp. 4.2) times faster than Match on *WebGraph* (resp. *Youtube*). (3) The running time of MatchJoin$_{\mathsf{max}}$ is much less sensitive to $|\mathsf{Q_s}|$ compared to Match.

Accuracy. We report the accuracy (F-measure, Section 6) of MatchJoin$_{\mathsf{max}}$ in Figures 9(k) and 10(b) on *WebGraph* and *Youtube*, respectively. We found the following. (1) MatchJoin$_{\mathsf{max}}$ finds approximate answers with high accuracy. The Acc is 0.73 (resp. 0.65) on *WebGraph* (resp. *Youtube*) on average. (2) The accuracy of MatchJoin$_{\mathsf{max}}$ is not sensitive to the pattern size; instead, it is determined by how much a maximally contained rewriting "covers" the pattern query. For example, we found (not shown) that the accuracy of MatchJoin$_{\mathsf{max}}$ is on average 0.63 when the rewriting "missed" two edges in the pattern query, and it increases to 0.82 when only one query edge is missed.

Scalability. We evaluated the scalability of MatchJoin$_{\mathsf{max}}$ and Match, in the same setting as in Fig. 9(e). As shown in Fig. 9(l), (1) MatchJoin$_{\mathsf{max}}$ scales better with $|\mathsf{Q_s}|$ than Match; and (2) MatchJoin$_{\mathsf{max}}$ takes only 4.4% of the time of Match when the scale factor is 0.1, and the saving is more significant for larger $|G|$.

Summary. From the experimental results we find the following. (1) Answering pattern queries using views is effective in querying large graphs. For example, by using views, pattern matching via graph simulation is 23.2 times faster than computing matches directly on *WebGraph*. (2) Our view-based matching algorithms scale well with the query and data size. Moreover, they are much less sensitive to the size of data graphs. (3) It is efficient to determine whether a pattern query can be answered using views. In particular, our approximation algorithm for minimum containment effectively reduces redundant views. (4) Our optimization strategy further makes the view-based matching up to 1.66 times faster. (5) When patterns are not contained by views, our query-driven approximation scheme evaluates the queries efficiently with reasonable accuracy. For example, MatchJoin$_{\mathsf{max}}$ is 24.8 times faster than Match, with accuracy 0.73 over large Web graph.

8 Conclusion

We have studied graph simulation using views, from theory to algorithms. We have proposed a notion of pattern containment to *characterize* what pattern queries can be answered using views, and provided such an efficient matching algorithm. We have also identified three fundamental problems for pattern containment, established their complexity, and developed effective (approximation) algorithms. When a pattern query is not contained in available views, we have developed efficient algorithms for computing maximally contained rewriting using views to get approximate answers. Our experimental results have verified the efficiency and effectiveness of our techniques. These results extend the study of query answering using views from relational and XML queries to graph pattern queries.

Our techniques can be readily extended to variants of graph simulation. Take strong simulation [36] as example, MatchJoin only needs to check (lines 7-11), for each pattern edge (u', u) and its match (v', v) in S, whether for each pattern edge (u'', u'), there is a match (v'', v'), with time complexity unchanged.

The study of graph pattern matching using views is still in its infancy. One issue is to decide what views to cache such that a set of frequently used pattern queries can be answered by using the views. Techniques such as adaptive and incremental query expansion [48] may apply. Another issue concerns view-based pattern matching via subgraph isomorphism. The third topic is to find a subset $\mathcal{V}'$ of $\mathcal{V}$ such that $\mathcal{V}'(G)$ is minimum for all graphs G. Finally, to find a practical method to query "big" social data, one needs to combine techniques such as view-based, distributed, incremental, and compression methods.

Acknowledgments. Fan is supported in part by NSFC 61133002, 973 Program 2012CB316200 and 2014CB340302, Shenzhen Peacock Program 1105100030834361, Guangdong Innovative Research Team Program 2011D005, EPSRC EP/J015377/1 and EP/M025268/1, and a Google Faculty Research Award. Wang is supported in part by NSFC 61402383 and 71490722, Sichuan

Provincial Science and Technology Project 2014JY0207, and Fundamental Research Funds for the Central Universities, China. Wu is also supported in part by Shenzhen Peacock Program 1105100030834361.

REFERENCES

[1] Amazon dataset. *http://snap. stanford.edu/ data/ index.html.*

[2] Citation. *http://www.arnetminer.org/citation/.*

[3] WebGraph. *http://law.di.unimi.it/datasets.php.*

[4] Wikipedia. F-measure. *http://en.wikipedia.org/wiki/F-measure.*

[5] Youtube dataset. *http://netsg.cs.sfu.ca/youtubedata/.*

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[8] M. Armbrust, E. Liang, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson. Generalized scale independence through incremental precomputation. In *SIGMOD*, 2013.

[9] P. Barceló, C. A. Hurtado, L. Libkin, and P. T. Wood. Expressive languages for path queries over graph-structured data. In *PODS*, 2010.

[10] J. Brynielsson, J. Högberg, L. Kaati, C. Martenson, and P. Svenson. Detecting social positions using simulation. In *ASONAM*, 2010.

[11] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based query processing and constraint satisfaction. In *LICS*, 2000.

[12] Y. Cao, W. Fan, and R. Huang. Making pattern queries bounded in big graphs. In *ICDE*, 2015.

[13] M. W. Chekol, J. Euzenat, P. Genevès, and N. Layaida. PSPARQL query containment. Technical report, 2011.

[14] C.-M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *EDBT*, 1994.

[15] F. Chung, R. Graham, R. Bhagwan, S. Savage, and G. M. Voelker. Maximizing data locality in distributed systems. *JCSS*, 72(8):1309–1316, 2006.

[16] W. Fan, C. Y. Chan, and M. N. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.

[17] W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *PODS*, 2014.

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

[19] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, pages 157–168, 2012.

[20] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.

[21] W. Fan, X. Wang, and Y. Wu. Answering graph pattern queries using views. In *ICDE*, pages 184–195, 2014.

[22] W. Fan, X. Wang, and Y. Wu. Distributed graph simulation: Impossibility and possibility. *PVLDB*, 7(12):1083–1094, 2014.

[23] A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz. A distributed vertex-centric approach for pattern matching in massive graphs. In *Big Data*, 2013.

[24] J. C. Godskesen and S. Nanz. Mobility models and behavioural equivalence for wireless networks. In *Coordination Models and Languages*, pages 106–122, 2009.

[25] G. Grahne and A. Thomo. Query containment and rewriting using views for regular path queries under constraints. In *PODS*, 2003.

[26] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39, 2008.

[27] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[28] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.

[29] J. Hromkovic. *Algorithmics for hard problems*. Springer, 2001.

[30] L. V. S. Lakshmanan, W. H. Wang, and Z. J. Zhao. Answering tree pattern queries using views. In *VLDB*, 2006.

[31] T. Lappas, K. Liu, and E. Terzi. A survey of algorithms and systems for expert location in social networks. In *Social Network Data Analytics*. 2011.

[32] W. Le, S. Duan, A. Kementsietsidis, F. Li, and M. Wang. Rewriting queries on SPARQL views. In *WWW*, 2011.

[33] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

[34] J. Leskovec, A. Singh, and J. M. Kleinberg. Patterns of influence in a recommendation network. In *PAKDD*, 2006.

[35] J. J. Luczkovich, S. P. Borgatti, J. C. Johnson, and M. G. Everett. Defining and measuring trophic role similarity in food webs using regular equivalence. *Journal of Theoretical Biology*, 220(3):303–321, 2003.

[36] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo. Capturing topology in graph pattern matching. *PVLDB*, 5(4), 2011.

[37] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *WWW*, pages 949–958, 2012.

[38] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[39] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[40] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD*, 2008.

[41] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.

[42] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[43] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD*, pages 455–466, 1999.

[44] D. Park and M. Toyama. XML cache management based on XPath containment relationship. In *ICDE Workshops*, 2005.

[45] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *VLDB*, 2000.

[46] C. Smith. By the numbers: 98 amazing facebook statistics. DMR, Mar. 2014.

[47] L. G. Terveen and D. W. McDonald. Social matching: A framework and research agenda. In *ACM Trans. Comput.-Hum. Interact.*, 2005.

[48] M. Theobald, R. Schenkel, and G. Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*, 2005.

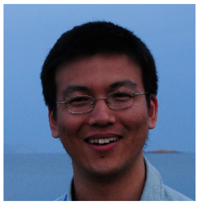
[49] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[50] J. Wang, J. Li, and J. X. Yu. Answering tree pattern queries using views: A revisit. In *EDBT*, 2011.

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

[52] Y. Zhuge and H. Garcia-Molina. Graph structured views and their incremental maintenance. In *ICDE*, 1998.

Wenfei Fan is Chair Professor of Web Data Management in the School of Informatics, University of Edinburgh, UK. He is a Fellow of the Royal Society of Edinburgh, UK, a fellow of the ACM, a National Professor of the Thousand-Talent Program and a Yangtze River Scholar, China. He received his PhD from the University of Pennsylvania, and his MS and BS from Peking University. He is a recipient of the Roger Needham Award in 2008 (UK), the Alberto O. Mendelzon Test-of-Time Award of ACM PODS 2010 and 2015, and several Best Paper Awards (VLDB 2010, ICDE 2007, Computer Networks 2002), and the Career Award in 2001 (USA). His current research interests include database theory and systems, in particular big data, data quality, data integration, distributed query processing, query languages, recommender systems, social networks and Web services.



Xin Wang is an associate professor at Southwest Jiaotong University, China. He received his PhD degree from the University of Edinburgh in 2013. His current research interests include graph queries, and big social data analysis.

Yinghui Wu is an assistant professor at School of EECS, Washington State University, US. He was a research scientist at University of California Santa Barbara, and a member of Network Science Collaborative Technique Alliance. His current research mainly focus on big data, graph databases and network science, with applications in social and information network analytics, and network security. He receives his Ph.D. from the University of Edinburgh in 2010.