Answering graph pattern queries using views

Citation for published version:

Digital Object Identifier (DOI):
10.1109/ICDE.2014.6816650

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Answering Graph Pattern Queries Using Views

Wenfei Fan 1,2 Xin Wang 3 Yinhui Wu 4
1University of Edinburgh 2RCBD and SKLSDE Lab, Beihang University 3Southwest Jiaotong University 4UC Santa Barbara
{wenfei@inf, x.wang-36@sms}.ed.ac.uk, yinhui@cs.ucsb.edu

Abstract—Answering queries using views has proven an effective technique for querying relational and semistructured data. This paper investigates this issue for graph pattern queries based on (bounded) simulation, which have been increasingly used in, e.g., social network analysis. We propose a notion of pattern containment to characterize graph pattern matching using graph pattern views. We show that a graph pattern query can be answered using a set of views if and only if the query is contained in the views. Based on this characterization we develop efficient algorithms to answer graph pattern queries. In addition, we identify three problems associated with graph pattern containment. We show that these problems range from quadratic-time to NP-complete, and provide efficient algorithms for containment checking (approximation when the problem is intractable). Using real-life data and synthetic data, we experimentally verify that these methods are able to efficiently answer graph pattern queries on large social graphs, by using views.

I. INTRODUCTION

Answering queries using views has been extensively studied for relational queries [19], [20], [25], XML [22], [36], [37] and semistructured data [11], [32], [38]. Given a query Q and a set V = {V1, . . . , Vn} of views, the idea is to find another query A such that A is equivalent to Q, and A only refers to views in V [19]. This yields an effective technique for evaluating Q: 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 V(D), without accessing D. This is particular 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]. They are also useful in data integration [25], data warehousing, semantic caching [13], and access control [14].

The need for studying this problem is even more evident for answering graph pattern queries (a.k.a. graph pattern matching) [16], [21]. Graph pattern queries have been increasingly used in social network analysis [10], [16], among other things. Real-life social graphs are typically large, and are often distributed. For example, Facebook currently has more than 1 billion users with 140 billion links [3], and the data is geo-distributed to various data centers [18]. One of the major challenges for social network analysis is how to cope with the sheer size of real-life social data when evaluating graph pattern queries. Graph pattern matching using views provides an effective method to query such 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, e.g., (Bob, Dan) indicates Dan worked well with Bob on a project led by Bob.

To build a team, a human resource manager issues a pattern query [23]. The query, expressed as Qs in Fig. 1 (c), is to find a group of PM, DBA and PRG. It requires that (1) DBA1 and PRG2 worked well under the project manager PM; (2) each PRG (resp. DBA) had been supervised by a DBA (resp. PRG), represented as a collaboration cycle [23] in Qs. For pattern matching based on graph simulation [16], [34], the answer Qs(G) to Qs in G can be denoted as a set of pairs (e, Sε) such that for each pattern edge e in Qs, Sε is a set of edges (a match set) for e in G. For example, pattern edge (PM, PRG2) has a match set Sε = {Bob, (Walt, Bill)}, in which each edge matches the node labels and satisfies the connectivity constraint of the pattern edge (PM, PRG2).

It is known that it takes O(|Q1|2 + |Q1| · |G| + |G|2) time to compute Qs(G) [16], [21], where |G| (resp. |Q1|) is the size of G (resp. Q1). This is a daunting cost when G is big. For example, to identify the match set of each pattern edge (DBA1, PRG1) (for i ∈ {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.

One can do better by leveraging a set of views. Suppose that a set of views V = {V1, V2} is defined, materialized and cached (V(G) = {V1(G), V2(G)}), as shown in Fig. 1 (b). Then as will be shown later, to compute Qs(G), (1) we only need to visit views in V(G), without accessing the original big graph G; and (2) Qs(G) can be efficiently computed by “merging” views in V(G). Indeed, V(G) already contains partial answers to Qs in G; for each query edge e in Qs, the matches of e (e.g., (DBA, PRG)) are contained either in V1(G) or V2(G) (e.g., the matches of e1 in V2). These partial answers can be used to construct Qs(G). As a result, the cost of computing Qs(G) is quadratic in |Qs| and |V(G)|, where |V(G)| is typically 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 \( Q_s \) can be answered by a set \( V \) of views? (2) If so, how to efficiently compute \( Q \_s (G) \) from \( V (G) \)? (3) Which views in \( V \) should we choose to answer \( Q \_s \)?

Contributions. This paper investigates these questions for answering graph pattern queries using graph pattern views. We focus on pattern matching defined in terms of graph simulation \([21]\) and bounded simulation \([16]\), which are particularly useful in detecting social communities and positions \([10]\).

(1) To characterize when graph pattern queries can be answered using views based on graph simulation, we propose a notion of pattern containment (Section III). It extends the traditional notion of query containment \([6]\) to deal with a set of views. Given a pattern query \( Q_s \) and a set \( V = \{V_1, \ldots, V_n\} \) of view definitions, we show that \( Q_s \) can be answered using \( V \) if and only if \( Q_s \) is contained in \( V \).

We also provide an evaluation algorithm for answering graph pattern queries using views (Section III). Given \( Q_s \) and a set \( V (G) \) of views on a graph \( G \), the algorithm computes \( Q \_s (G) \) in \( O(|Q_s| |V (G)| + |V (G)|^2) \) time, without accessing \( G \) at all when \( Q_s \) is contained in \( V \). It is far less costly than \( O(|Q_1|^2 + |Q_2| G + |G|^2) \) for evaluating \( Q_2 \) directly on \( G \) \([16], [21]\), since \( G \) is typically much larger than \( V (G) \) in practice.

(2) To decide which views in \( V \) to use when answering \( Q \_s \), we identify three fundamental problems for pattern containment (Section IV). Given \( Q_s \) and \( V \), (i) the containment problem is to decide whether \( Q_s \) is contained in \( V \); (ii) the minimal containment problem is to identify a subset of \( V \) that minimally contains \( Q_s \), and (iii) the minimum containment problem is to find a minimum subset of \( V \) that contains \( Q_s \).

We establish the complexity of these problems. We show that the first two problems are in quadratic-time, whereas the last one is NP-complete and approximation-hard. These results are not only useful in answering pattern queries using views, but are also interesting for query minimization. Indeed, when \( V \) contains a single view, the containment problem becomes the classical query containment problem \([6]\).

These results are a nice surprise. Note that even for relational conjunctive queries, the problem of query containment is NP-complete \([6]\); for XPath fragments, it is EXPTIME-complete or even undecidable \([30]\). In contrast, the (minimal) containment problem for graph pattern queries is in low PTIME, although graph pattern matching via (bounded) simulation may be “recursively defined” (for cyclic patterns).

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

(4) We show that all these results carry over to bounded simulation \([16]\) (Section VI). More specifically, the notion of pattern containment, the algorithm for answering pattern queries using views, the three containment problems, and the (approximation) algorithms for checking (minimal, minimum) pattern containment can be extended to bounded simulation, with the same or comparable complexity.

(5) Using real-life data (Amazon, YouTube and Citation) and synthetic data, we experimentally verify the effectiveness and efficiency of our view-based matching method (Section VII). We find that this method reduces 94% of the time used by prior methods for bounded pattern queries on large datasets on average \([16]\). Moreover, our matching algorithm scales well with both the data size and pattern size; and our algorithms for (minimal, minimum) pattern containment checking take less than 0.5 second on complex (cyclic) patterns. Furthermore, we find that our optimization methods by identifying minimal (minimum) containment effectively reduce redundant views and improve the performance by 46% on average.

This 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 these views without accessing the large social graphs. Better still, incremental methods are already in place to efficiently maintain cached pattern views (e.g., \([15]\)). The view-based method can be readily combined with existing distributed, compression and incremental techniques for graphs, and yield a promising approach to querying “big” social data.

The proofs of the results of this work can be found in \([4]\).

Related Work. There are two view-based approaches for query processing: query rewriting and query answering \([20], [25]\). Given a query \( Q \) and a set \( V \) of views, (1) query rewriting is to reformulate \( Q \) into an equivalent query \( Q' \) in a fixed language (i.e., for all \( D \), \( Q(D) = Q'(D) \)), such that \( Q' \) refers only to \( V \); and (2) query answering is to compute \( Q(D) \) by evaluating an equivalent query \( A \) of \( Q \), while \( A \) refers only to \( V \) and its extensions \( V(D) \). While the former requires that \( Q' \) is in a fixed language, the latter imposes no constraint on \( A \). We study answering graph pattern queries using pattern views.

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

Relational data. Query answering using views has been extensively studied for relational data (see \([6], [20], [25]\) for surveys). It is known that for conjunctive queries, query answering and rewriting using views are already intractable \([20], [25]\). For the containment problem, the homomorphism theorem shows that one conjunctive 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 for conjunctive queries is NP-complete, and is undecidable for relational algebra \([6]\).

XML queries. There has been a host of work on processing XML queries using views \([29], [30], [33]\). In \([29]\), 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 \([30]\). In \([7]\), pattern containment and query rewriting of XML are studied under constraints expressed as a structural summary. For tree pattern queries (a fragment of XPath), \([22], [36]\) study maximally contained rewriting instead of equivalent rewriting.
Semistructured data and RDF. There has also been work on view-based query processing for semistructured data and RDF, which are also modeled as graphs.

(1) Semistructured data. Views defined in Lorel are studied in, e.g., [38], 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 [32]. The containment problem is shown undecidable for RPQs in the presence of path constraints [17] and for extended conjunctive RPQs [9].

(2) RDF. An EXPTIME query rewriting algorithm is given in [24] for SPARQL. It is shown in [12] that query containment is in EXPTIME for SPARQL, which supports regular expressions. There has also been work on evaluating SPARQL queries on RDF based on cached query results [13].

Our work differs from the prior work in the following. (1) We study query answering using views for graph pattern queries via (bounded) 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., XPath, regular path queries and SPARQL. (3) We study a more general form of query containment between a query Q_s and a set of queries, to identify an equivalent query for 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.

We focus on (bounded) simulation in this work as it is widely used in social data analysis [10], [16]. Nonetheless, the techniques can be extended to revisions of simulation such as dual and strong simulation [28] (see Section VIII).

II. GRAPHS, PATTERNS AND VIEWS

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

A. 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 ⊆ V × 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, social roles [23].

Pattern queries [16]. A graph pattern query, denoted as Q_s, is a directed graph 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 [16] (see Fig. 7 for examples).

Graph pattern matching via simulation. We say that a data graph G = (V, E, L) matches a query Q_s = (V_p, E_p, f_v) via simulation, denoted by Q_s ≤_{sim} G, if there exists a binary relation S ⊆ V_p × V, referred to as a match in G for Q_s, such that

- for each node u ∈ V_p, there exists a node v ∈ V such that (u, v) ∈ S, referred to as a match of u; and
- for each pair (u, v) ∈ S, f_v(u) ∈ L(v); 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) ∈ E.

When Q_s ≤_{sim} G, it is known that there exists a unique maximum match S_m in G for Q_s [21]. We derive {e, S_m} | e ∈ E_p} from S_m, where S_m is the set of all matches of e in S_m, called the match set of e. Here S_m is nonempty for all e ∈ E_p.

We define the result of Q_s in G, denoted as Q_s(G), to be the unique maximum set \{e, S_m | e ∈ E_p\} if Q_s ≤_{sim} G, and let Q_s(G) = \emptyset otherwise. We denote the size of query Q_s by |Q_s|, and the size of result Q_s(G) by |Q_s(G)| (see Table I).

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

<table>
<thead>
<tr>
<th>Edge</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PM, DBA)</td>
<td>(Bob, Mat), (Walt, Mat)</td>
</tr>
<tr>
<td>(PM, PRG)</td>
<td>(Bob, Dan), (Walt, Bill)</td>
</tr>
<tr>
<td>(DBA, PRG)</td>
<td>(Fred, Pat), (Mat, Pat), (Mary, Bill)</td>
</tr>
<tr>
<td>(PRG, DBA)</td>
<td>(Dan, Fred), (Pat, Mary), (Bill, Mat)</td>
</tr>
</tbody>
</table>

Here (1) both Bob and Walt are matches of pattern node PM as they satisfy the search condition of PM; similarly, Fred, Mat, Mary match DBA, and Dan, Pat, Bill match PRG; (2) query edge (PM, DBA) has two matches in G; and (3) query edges (DBA, PRG) and (DBA, PRG) (resp. (PRG, DBA) and (PRG, DBA)) have the same matches.

B. 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 V(G) in a data graph G as the view extension for V in G or simply as a view [19].

Given a pattern query Q_s and a set V = \{V_1, . . . , V_n\} of view definitions, graph pattern matching using views is to find another query A such that (1) A is equivalent to Q_s, i.e., A(G) = Q_s(G) for all data graphs G; and (2) A only refers to views V_i ∈ V and their extensions V_i(G) = \{V_i(G), V_i(G)\} in G, without accessing G. If such a query A exists, we say that Q_s can be answered using V.

In contrast to query rewriting using views [19] but along the same lines as query answering using views [25], here A is not required to be a pattern query. For example, Fig. 1 (b) depicts a view definition set V = \{V_1, V_2\} and their extensions V(G) = \{V_1(G), V_2(G)\}. To answer the query Q_s (Fig. 1 (c)), we want to find a query A that computes Q_s(G) by using only V and V(G), where A is not necessarily a graph pattern.

For a set V of view definitions, we define the size |V| of V to be the total size of V_i’s in V, and the cardinality card(V) of V to be the number of view definitions in V.

The notations of the paper are summarized in Table I.
TABLE I: A summary of notations

Remark. (1) We assume without loss of generality that graph patterns are connected, since isolated pattern nodes can be easily handled using the same matching semantic. (2) Our techniques can be readily extended to graphs and queries with edge labels. Indeed, an edge-labeled graph can be transformed to a node-labeled graph: for each edge e, add a “dummy” node carrying the edge label of e, along with two unlabeled edges.

III. PATTERN CONTAINMENT: 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 introduce a notion of pattern containment, by extending the traditional notion of query containment to a set of views. Consider a pattern query $Q_s = (V_s, E_s, f_s)$ and a set $V = \{V_1, \ldots, V_n\}$ of view definitions, where $V_i = (V_{i1}, E_{i1}, f_{i1})$. We say that $Q_s$ is contained in $V$, denoted by $Q_s \subseteq V$, if there exists a mapping $\lambda$ from $E_s$ to powerset $P(\bigcup_{i=1}^n E_{i1})$, such that for all data graphs $G$, the match set $S_e \subseteq \bigcup_{i \in \lambda(e)} S_{e_i}$ for all edges $e \in E_s$.

Example 3: Recall $G$, $V$ and $Q_s$ in Fig. 1. Then $Q_s \subseteq V$. Indeed, there exists a mapping $\lambda$ from $E_s$ of $Q_s$ to sets of edges in $V$, which maps edges $(PM, DBA_1)$, $(PM, PRG_3)$ of $Q_s$ to their counterparts in $V_1$; both $(DBA_1, PRG_1)$, $(DBA_2, PRG_2)$ of $Q_s$ to $e_3$, and $(PRG_1, DBA_2)$, $(PRG_2, DBA_2)$ to $e_4$ in $V_2$. One may verify that for any graph $G$ and any edge $e$ of $Q_s$, its matches in $G$ are contained in the union of the match sets of the edges in $\lambda(e)$, e.g., 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 $e_3$ in $V_2$.

Pattern containment and query answering. The main result of this section is as follows: (1) pattern containment indeed characterizes pattern matching using views; and (2) when $Q_s \subseteq V$, for all graphs $G$, $Q_s(G)$ can be efficiently computed by using views $V(G)$ only, independent of $G$. In Sections IV and V we will show how to decide whether $Q_s \subseteq V$ by inspecting $Q_s$ and $V$ only, also independent of $G$.

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

Proof sketch: Below we outline the proof (see [4] for details).

Input: A pattern query $Q_s$, a set of view definitions $V$, and their extensions $V(G)$, a mapping $\lambda$.

Output: The query result $M$ as $Q_s(G)$.

1. for each edge $e$ in $Q_s$, do $S_e := \emptyset$;
2. $M := \{(e, S_e) \mid e \in Q_s\}$
3. for each $e \in Q_s$ do
4. for each $e' \in \lambda(e)$ do $S_e := S_e \cup S_{e'}$;
5. while there is change in $S_e$ for an edge $e_p = (u, u')$ in $Q_s$ do
6. for each $e = (u, u') \in Q_s$ and $e' = (v, v') \in S_{e_p}$ do
7. if $e_1 = (u', u_1)$ but not $e_1 = (e', v_1)$ in $S_{e_1}$ then
8. $S_e := S_e \setminus \{e'\}$;
9. if $e_2 = (u, u_2)$ but not $e_2 = (v, v_2)$ in $S_{e_2}$ then
10. $S_e := S_e \setminus \{e'\}$;
11. if $S_e = \emptyset$ then return $\emptyset$;
12. return $M = \{(e, S_e) \mid e \in Q_s\}$, which is $Q_s(G)$;

Fig. 2: Algorithm MatchJoin

(I) We prove the Only If condition in Theorem 1(1) by contradiction. Assume that $Q_s$ can be answered using $V$ and $Q_s \not\subseteq V$. By $Q_s \not\subseteq V$, there exists at least an edge $e$ of $Q_s$ that cannot be mapped to any edge in the match sets from $V$. We show that for such an edge $e$, one may always construct a data graph $G$ that matches $Q_s$, but no match in $G$ can be identified by using only $V$. This contradicts the assumption that $Q_s$ can be answered using $V$ by definition (Section II-B).

(II) We show the If condition of Theorem 1(1) by a constructive proof: we next present an algorithm to evaluate $Q_s$ using $V(G)$, if $Q_s \subseteq V$. We verify Theorem 1(2) by showing that the algorithm is in $O(|Q_s| |V(G)| + |V(G)|^2)$ time. □

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

More specifically, MatchJoin works as follows. It first initializes $M$ 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 $V(G)$ (lines 3-4), following the definition of $\lambda(e)$ (Section II). It then performs a fixpoint computation to remove all invalid matches from $S_e$ (lines 5-10). Specifically, for a pattern edge $e_p = (u, u')$ in $Q_s$ with changed match set $S_e$, it checks whether each match $e'$ of a pattern edge $e = (u', u)$ still remains to be a match of $e$ (lines 7-10), by the definition of simulation (Section II-A). If $e'$ is no longer a match, it is removed from $S_e$ (lines 8, 10). In the process, if $S_e$ becomes empty for some edge $e$, MatchJoin returns $\emptyset$ since $Q_s$ has no match in $G$. Otherwise, the process (lines 5-11) proceeds until $M=Q_s(G)$ is computed and returned (line 12).

Example 4: Given $Q_s$, $V$, $V(G)$ of Fig. 1, and the mapping $\lambda$ of Example 3, MatchJoin evaluates $Q_s$ using $V$ and $V(G)$. For each edge $e$ of $Q_s$, its match set $S_e$ is exactly $\bigcup_{e' \in \lambda(e)} S_{e'}$, which yields the same $Q_s(G)$ as given in Example 2.

Consider another example shown in Fig. 3. One can verify $Q_s \not\subseteq V$ by a mapping $\lambda$ that maps $(Al, Bio), (PM, Al)$ 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 $\lambda$. It next removes $(AI, SE)$ from $S_1$ when MatchJoin terminates. This further leads to the removal of $(SE, DB)$ from $S_2$ and $(DB, AI)$ from $S_3$, and yields $Q_3(G)$ shown in the table below.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Matches</th>
<th>Edge</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(PM, AI)$</td>
<td>$(PM_1, AI_2)$</td>
<td>$(AI, BI_0)$</td>
<td>$(AI, BI_0)$</td>
</tr>
<tr>
<td>$(DB, AI)$</td>
<td>$(DB, AI_2)$</td>
<td>$(AI, SE)$</td>
<td>$(AI, SE)$</td>
</tr>
<tr>
<td>$(SE, DB)$</td>
<td>$(SE_2, DB_1)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Correctness & Complexity.** Denote the match set in $G$ as $S_e^*$ for each edge $e$ in $Q_3$. One may verify that MatchJoin preserves two invariants: (1) at any time, for each edge $e$ of $Q_3$, $S_e^* \subseteq S_e$; and (2) $S_e = S_e^*$ when MatchJoin terminates. Indeed, $S_e$ is initialized with $\bigcup_{e' \in \lambda}(S_{e'})$, hence $S_e \subseteq S_e^*$ due to $Q_3 \subseteq V$. During the while loop (lines 5-10), MatchJoin repeatedly refines $S_e$ by removing invalid matches only (lines 8-10) until $S_e$ can no longer be refined. Thus, $S_e = S_e^*$ when the algorithm terminates. From these the correctness of MatchJoin follows.

For the complexity, it takes $O(|Q_3|)$ time to initialize $M$ (lines 1-2), and $O(|Q_3||V(G)|)$ time to initialize $S_e$ (lines 3-4). MatchJoin then removes invalid matches by using a dynamically maintained index, which maps pattern edges to their possible matches (see [4] for details). The while loop (lines 5-11) is bounded by $O(|V(G)|^2)$ time. Putting these together, MatchJoin is in $O(|Q_3||V(G)| + |V(G)|^2)$ time.

**Remark.** (1) It takes $O(|Q_3|^2 + |Q_3||G| + |G|^2)$ time to evaluate $Q_3(G)$ directly on $G$ [16]. In contrast, MatchJoin is in $O(|Q_3||V(G)| + |V(G)|^2)$ time, without accessing $G$. As will be seen in Section VII, $V(G)$ is much smaller than $G$, and MatchJoin is more efficient than the algorithm of [16]. Indeed, for Youtube graph in our experiments, only 3 to 6 views are used to answer $Q_3$, and the overall size of $V(G)$ is no more than 4% of the size of the Youtube graph.

**Optimization.** MatchJoin may visit each $S_e$ multiple times. To reduce unnecessary visits, below we introduce an optimization strategy for MatchJoin. The strategy evaluates $Q_3$ by using ranks in $Q_3$ as follows. Given a pattern $Q_3$, the strongly connected component graph $G_{SCC}$ of $Q_3$ is obtained by collapsing each strongly connected component SCC of $Q_3$ into a single node $s(u)$. The rank $r(u)$ of each node $u$ in $Q_3$ 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 $Q_3$. The rank $r(e)$ of an edge $e = (u', u)$ in $Q_3$ is set to be $r(u)$.

**Bottom-up strategy.** We revise MatchJoin by processing edges $e$ in $Q_3$, 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 2:** For all edges $e = (u', u)$, where $u'$ and $u$ do not reach any non-singleton SCC in $Q_3$, MatchJoin visits its match set $S_e$ at most once, using the bottom-up strategy. □

In particular, when $Q_3$ 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 $Q_3$. As will be verified in Section VII, the optimization strategy improves the performance by at least 46% over (possibly cyclic) patterns, and is even more effective over denser data graphs.

**IV. Pattern Containment Problems**

We have seen that given a pattern query $Q_3$ and a set $V$ of views, we can efficiently answer $Q_3$ by using the views when $Q_3 \subseteq V$. In the next two sections, we study how to determine whether $Q_3 \subseteq V$. Our main conclusion is that there are efficient algorithms for these, with their costs as a function of $|Q_3|$ and $|V|$, which are typically small in practice, and are independent of data graphs and materialized views.

In this section we study three problems in connection with pattern containment, and establish their complexity. In the next section, we will develop effective algorithms for checking $Q_3 \subseteq V$ and computing mapping $\lambda$ from $Q_3$ to $V$.

**Pattern containment problem.** The pattern containment problem is to determine, given a pattern query $Q_3$ and a set $V$ of view definitions, whether $Q_3 \subseteq V$. The need for studying this problem is evident: Theorem 1 tells us that $Q_3$ can be answered by using views of $V$ if and only if $Q_3 \subseteq V$.

The result below tells us that $Q_3 \subseteq V$ can be efficiently decided, in quadratic-time in $|Q_3|$ and $|V|$. We will prove the result in Section V, by providing such an algorithm.

**Theorem 3:** Given a pattern query $Q_3$ and a set $V$ of view definitions, it is in $O(\text{card}(V)|Q_3|^2 + |V|^2 + |Q_3||V|)$ time to decide whether $Q_3 \subseteq V$ and if so, to compute a mapping $\lambda$ from $Q_3$ to $V$, where $|V|$ is the size of view definitions. □

A special case of pattern containment is the classical query containment problem [6]. Given two pattern queries $Q_{31}$ and $Q_{32}$, the latter is to decide whether $Q_{31} \subseteq Q_{32}$, i.e., whether for all graphs $G$, $Q_{31}(G)$ is contained in $Q_{32}(G)$. Indeed, when $V$ contains only a single view definition $Q_{32}$, pattern containment becomes query containment. From this and Theorem 3 the result below immediately follows.

**Corollary 4:** The query containment problem for graph pattern queries is in quadratic time. □

Like for relational queries (see, e.g., [6]), the query containment analysis is important in minimizing and optimizing
pattern queries. Corollary 4 shows that the analysis can be efficiently conducted for graph patterns, as opposed to the intractability of its counterpart for relational conjunctive queries.

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

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

**Theorem 5:** Given \(Q_s\) and \(V\), it is in \(O(\text{card}(V)\|Q_s\|^2 + |V|^2 + |Q_s|\|V\|)\) time to find a minimal subset \(V'\) of \(V\) containing \(Q_s\) and a mapping \(\lambda\) from \(Q_s\) to \(V'\) if \(Q_s \subseteq V\). \(\Box\)

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

As will be seen shortly (Examples 6 and 7) and verified by our experimental study, MMCP analysis often finds smaller \(V'\) than that found by minimal containment checking.

MMCP is, however, nontrivial: its decision problem is NP-complete and it is APX-hard. Here APX is the class of problems that allow PTIME algorithms with approximation ratio bounded by a constant (see [35] 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 \(Q_s\). That is, there exists an efficient algorithm that identifies a subset \(V'\) of \(V\) with performance guarantees whenever \(Q_s \subseteq V\) such that \(Q_s \subseteq V'\) and \(\text{card}(V') \leq \log(|E_p|)/\text{card}(\text{OPT})\), where \(\text{OPT}\) is a minimum subset of \(V\) that contains \(Q_s\).

**Theorem 6:** 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 \(O(\text{card}(V)\|Q_s\|^2 + |V|^2 + |Q_s|\|V\|)\) time. \(\Box\)

**Proof sketch:** (I) The decision problem of MMCP is to decide whether there exists a subset \(V'\) of \(V\) such that \(Q_s \subseteq V'\) and \(\text{card}(V') \leq k\), where \(k\) is an integer bound. It is in NP since there exists an NP algorithm that first guesses \(V'\) and then checks whether \(Q_s \subseteq V'\) and \(\text{card}(V') \leq k\) in PTIME. The lower bound is verified by reduction from the NP-complete set cover problem (cf. [31]). Given a set \(X\), a collection \(U\) of its subsets and an integer \(B\), the latter is to decide whether there is a \(B\)-element subset of \(U\) that covers \(X\). We show that there exists a set cover of size \(k\) if and only if there exists a \(k\)-element subset \(V'\) of \(V\) that contains \(Q_s\).

(II) The APX-hardness of MMCP is verified by approximation preserving reduction [35] from the minimum set cover problem, which is APX-hard (cf. [35]; see [4] for details). \(\Box\)

We defer the proof of Theorem 6(2) to Section V, where an approximation algorithm is provided.

V. **Determining Pattern Containment**

We next prove Theorems 3, 5 and 6(2) by providing effective (approximation) algorithms for checking pattern containment, minimal containment and minimum containment, in Sections V-A, V-B and V-C, respectively.

A. **Pattern Containment**

We start with a proof of Theorem 3, i.e., whether \(Q_s \subseteq V\) can be decided in \(O(\text{card}(V)\|Q_s\|^2 + |V|^2 + |Q_s|\|V\|)\) time. To do this, we first propose a sufficient and necessary condition to characterize pattern containment. We then develop a quadratic-time algorithm based on the characterization.

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

**Proposition 7:** For view definitions \(V\) and pattern \(Q_s\) with edge set \(E_p\), \(Q_s \subseteq V\) if and only if \(E_p = \bigcup_{V \in \mathcal{V}} \mathcal{M}_{\lambda}^Q\). \(\Box\)

**Proof sketch:** We outline the proof below (see [4] for details).

(1) Assume \(E_p = \bigcup_{V \in \mathcal{V}} \mathcal{M}_{\lambda}^Q\). We construct a mapping \(\lambda\) from \(E_p\) to the edges of the views in \(V\), as a “reversed” view match relation. The mapping \(\lambda\) ensures that for any data graph \(G\), if \(e_o \in G\) is a match of \(e\) in \(Q_s\) (\(e_o \in S_o\)), there must exist an edge \(e_i \in \lambda(e)\) such that \(e_o \subseteq S_{e_i}\). Thus, \(Q_s \subseteq V\).

(2) Assume by contradiction \(Q_s \not\subseteq V\) but \(E_p \neq \bigcup_{V \in \mathcal{V}} \mathcal{M}_{\lambda}^Q\). Then by \(E_p \neq \bigcup_{V \in \mathcal{V}} \mathcal{M}_{\lambda}^Q\), there exists an edge \(e\) in \(E_p\) but not in \(\bigcup_{V \in \mathcal{V}} \mathcal{M}_{\lambda}^Q\). Since \(Q_s \subseteq V\), if an edge \(e_o \in G\) matches \(e\) in \(Q_s\), then \(e_o \in S_{e_i}\) of \(V(G)\) for some \(V \in \mathcal{V}\). These together lead to the contradiction, since if such an \(e\) exists, we can expand mapping \(\lambda(e)\) by including \(e_i\) of \(V\); thus \(e\) is “covered” by \(\mathcal{M}_{\lambda}^Q\). Therefore, if \(Q_s \not\subseteq V\), \(E_p = \bigcup_{V \in \mathcal{V}} \mathcal{M}_{\lambda}^Q\). \(\Box\)

**Algorithm.** Following Proposition 7, we present an algorithm, denoted as contain (not shown) to check whether \(Q_s \subseteq V\). Given a pattern query \(Q_s\) and a set \(V\) of view definitions, it returns a boolean value \(ans\) that is true if and only if \(Q_s \subseteq V\). The algorithm first initializes an empty edge set \(E\) to record view matches from \(V\) to \(Q_s\). It then checks the condition of Proposition 7 as follows.

(1) Compute view match \(\mathcal{M}_{\lambda}^Q\), for each \(V \in \mathcal{V}\), by invoking the simulation evaluation algorithm in [16]. (2) Extend \(E\) with \(\mathcal{M}_{\lambda}^Q\) by union, since \(\mathcal{M}_{\lambda}^Q\) is a subset of \(E_p\). After all view matches are merged, contain then checks whether \(E = E_p\). It returns true if so, and false otherwise.

**Example 5:** Recall the pattern query \(Q_s\) and views \(V = \{V_1, V_2\}\) given in Fig. 1. As remarked earlier, \(Q_s \not\subseteq V\). Indeed, one can verify that \(\bigcup_{V \in \mathcal{V}} \mathcal{M}_{\lambda}^Q = E_p\).
Correctness & Complexity. The correctness of the algorithm follow from Proposition 7. For each \( \mathcal{V} \in \mathcal{V} \), it takes \( O(|\mathcal{Q}_s||\mathcal{V}|^2 + |\mathcal{V}|^2) \) time to compute \( M^Q_{\mathcal{V}} \), [16], and \( O(1) \) time for updating the edges. The \( \textbf{for} \) loop (lines 2-3) has \( \text{card} (\mathcal{V}) \) iterations, and it takes \( O(\text{card} (\mathcal{V})|\mathcal{Q}_s|^2 + |\mathcal{V}|^2 + |\mathcal{Q}_s||\mathcal{V}|) \) time in total, since both \( \text{card} (\mathcal{V}) \cdot |\mathcal{V}| \) and \( \mathcal{V} \) are bounded by \( |\mathcal{V}| \).

From these and Proposition 7, Theorem 3 follows.

Remarks. (1) Algorithm \textsc{contain} can be easily adapted to return a mapping \( \lambda \) that specifies pattern containment (Section III), to serve as input for algorithm \textsc{MatchJoin}. This can be done by following the construction given in the proof of Proposition 7. (2) In contrast to regular path queries and relational queries, pattern containment checking is in PTIME.

B. Minimal Containment Problem

We now prove Theorem 5 by presenting an algorithm that, given \( \mathcal{Q}_s \) and \( \mathcal{V} \), finds a minimal subset \( \mathcal{V}' \) of \( \mathcal{V} \) containing \( \mathcal{Q}_s \) in \( O(\text{card}(\mathcal{V})|\mathcal{Q}_s|^2 + |\mathcal{V}|^2 + |\mathcal{Q}_s||\mathcal{V}|) \) time if \( \mathcal{Q}_s \subseteq \mathcal{V} \).

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

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

Example 6: Consider \( \mathcal{Q}_s \) and \( \mathcal{V} \) given in Fig. 4. After \( M^Q_{\mathcal{V}_i} \) (i.e., \( \mathcal{V}_i \in [1, 4] \)) are computed, algorithm \textsc{minimal} finds that \( \mathcal{E} \) already equals \( E_p \), and breaks the loop, where \( \mathcal{M} \) is initialized to be \{\( (A, B) : \{V_3\} \), \( (A, C) : \{V_5\} \), \( (B, D) : \{V_4\} \), \( (C, D) : \{V_1, V_4\} \), \( (B, E) : \{V_2\} \)\}. As the removal of \( \mathcal{V}_1 \) does not make \( M(e) \) empty, minimal removes \( \mathcal{V}_1 \) and returns \( \mathcal{V}' = \{V_2, V_3, V_4\} \) as a minimal subset of \( \mathcal{V} \).

Correctness & Complexity. To see the correctness of minimal, observe the following: (1) \( \mathcal{Q}_s \subseteq \mathcal{V}' \) if \( \mathcal{V}' \neq \emptyset \); indeed, \( \mathcal{V}' \) is returned only if the union of the view matches in \( \mathcal{S} \) equals \( E_p \), i.e., \( \mathcal{Q}_s \subseteq \mathcal{V}' \) by Proposition 7; and (2) \( \mathcal{Q}_s \not\subseteq \mathcal{V}' \) for any \( \mathcal{V}' \subset \mathcal{V}' \). To see this, note that by the strategy of minimal for reducing redundant views in \( \mathcal{V}' \) (lines 9-11), for any \( \mathcal{V}' \subset \mathcal{V}' \), \( \bigcup_{\mathcal{V}_i \in \mathcal{V}' \setminus \mathcal{V}_i} M^Q_{\mathcal{V}_i} \) is not equal to \( E_p \), the edge set of \( \mathcal{Q}_s \). Hence again by Proposition 7, \( \mathcal{Q}_s \not\subseteq \mathcal{V}' \).

The algorithm and its analyses Theorem 5 follows. Again algorithm \textsc{minimal} can be readily extended to return a mapping \( \lambda \) that specifies containment of \( \mathcal{Q}_s \) in \( \mathcal{V}' \).

C. Minimum Containment Problem

We next prove Theorem 6 (2), i.e., MMCP is approximable within \( O(\log(|E_p|)) \) in \( O(\text{card}(\mathcal{V})|\mathcal{Q}_s|^2 + |\mathcal{V}|^2 + |\mathcal{Q}_s||\mathcal{V}| + (|\mathcal{Q}_s| \cdot \text{card}(\mathcal{V}))^{1/2}) \) time. We give such an algorithm for MMCP, following the greedy strategy of the approximation of [35] for the set cover problem. The algorithm of [35] achieves an approximation ratio \( O(\log n) \), for an \( n \)-element set.

Algorithm. The algorithm is denoted as minimum (not shown). Given a pattern \( \mathcal{Q}_s \) and a set \( \mathcal{V} \) of view definitions, minimum identifies a subset \( \mathcal{V}' \) of \( \mathcal{V} \) such that (1) \( \mathcal{Q}_s \subseteq \mathcal{V}' \) if \( \mathcal{Q}_s \subseteq \mathcal{V} \) and (2) \( \text{card}(\mathcal{V}') \leq \log(|E_p|) \cdot \text{card}(\mathcal{V}_\text{OPT}) \), where \( \mathcal{V}_\text{OPT} \) is a minimum subset of \( \mathcal{V} \) that contains \( \mathcal{Q}_s \). In other words, minimum approximates MMCP with approximation ratio \( O(\log |E_p|) \). Note that \( |E_p| \) is typically small.

Algorithm minimum iteratively finds the “top” view whose view match can cover most edges in \( \mathcal{Q}_s \) that are not covered.
To do this, we define a metric $\alpha(V)$ for a view $V$, where

$$\alpha(V) = \frac{|M_{Q_{\text{in}}}^V \setminus E_c|}{|E_p|}.$$  

Here $E_c$ 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 $M_{Q_{\text{in}}}^V$ covers. We select $V$ with the largest $\alpha$ in each iteration, and maintain $\alpha$ accordingly.

Similar to minimal, algorithm minimum computes the view match $M_{Q_{\text{in}}}^V$ for each $V \in V$, and collects them in a set $S$. It then does the following. (1) It selects view $V_i$ with the largest $\alpha$, and removes $M_{Q_{\text{in}}}^V$ from $S$. (2) It merges $E_c$ with $M_{Q_{\text{in}}}^V$ if $M_{Q_{\text{in}}}^V$ contains some edges that are not in $E_c$, and extends $\gamma'$ with $V_i$. During the loop, if $E_c$ equals $E_p$, the set $\gamma'$ is returned. Otherwise, minimum returns $\emptyset$, indicating that $Q_b \not\subseteq \mathcal{V}$.

Example 7: Given $Q_b$ and $\mathcal{V} = \{V_1, \ldots, V_7\}$ of Fig. 4, minimum selects views based on their $\alpha$ values. More specifically, in the loop it first chooses $V_5$, since its view match $M_{Q_{\text{in}}}^V = \{(A, B), (A, C), (C, D)\}$ makes $\alpha(V_5) = 0.6$, the largest one. Then $V_6$ is followed by $V_5$, as $\alpha(V_5) = 0.4$ is the largest one in that iteration. After $V_5$ and $V_6$ are selected, algorithm minimum finds that $E_c = E_p$, and thus $\gamma' = \{V_5, V_6\}$ is returned as a minimum subset that contains $Q_b$. \qed

Correctness & Complexity. Observe that minimum finds a nonempty $\gamma'$ such that $Q_b \subseteq \gamma'$ if and only if $Q_b \subseteq \mathcal{V}$ (Proposition 7). The approximation ratio of minimum can be verified by an approximation-preserving reduction from MMCP to the set cover problem [31], by treating each $M_{Q_{\text{in}}}^V$ in $S$ as a subset of $E_p$. Algorithm minimum extends the algorithm of [35] (with approximation ratio $\log(n)$ for $n$-element set) to query containment, and preserves approximation ratio $\log|E_p|$.

For the complexity, minimum computes view match times in $O(|\mathcal{V}| |Q_b|^2 + |\mathcal{V}|^2 + |Q_b||\mathcal{V}|)$ time (lines 1-3). The while loop is executed $O((|Q_b| \cdot |\mathcal{V}|)^{1/2})$ times. Each iteration takes $O(|Q_b| \cdot |\mathcal{V}|)$ time to find a view with the largest $\alpha$. Thus, minimum is in $O(|\mathcal{V}|^2 |Q_b|^2 + |\mathcal{V}|^2 + |Q_b||\mathcal{V}| + (|Q_b| \cdot |\mathcal{V}|)^{1/2})$ time, where $|Q_b|$ and $|\mathcal{V}|$ are often smaller than $|\mathcal{V}|$. This completes the proof of Theorem 6 (2).

VI. BOUNDED PATTERN MATCHING USING VIEWS

In this section, we show that the results of the previous sections carry over to bounded pattern queries, which extend patterns with distance constraints on pattern edges, and have been verified effective in social network analysis [16].

Bounded pattern queries [16]. A bounded pattern query, denoted as $Q_b$, is a directed graph $(V_p, E_p, f_v, f_e)$, where (1) $V_p$, $E_p$ and $f_v$ are the same as in a pattern query $Q$ (Section II), and (2) $f_e$ is a function defined on $E_p$ such that for all $(u, u')$ in $E_p$, $f_e(u, u')$ is either a positive integer $k$ or a symbol $\ast$.

A data graph $G = (V, E, L)$ matches $Q_b$ via bounded simulation, denoted by $Q_b \subseteq_{\text{sim}} G$ (Table 1), if there exists a binary relation $S \subseteq V_p \times V$ such that (1) for each node $u \in V_p$, there exists a match $v \in V$ such that $(u, v) \in S$, and (2) for each pair $(u, v) \in S$, $f_u(u, v) \in L(v)$, and for each pattern edge $e = (u, u')$ in $E_p$, there exists a nonempty path from $v$ to $v'$ in $G$, with its length bounded by $k$ if $f_e(u, u') = k$. When $f_e(u, u') = \ast$, there is no constraint on the path length.

Intuitively, $Q_b$ extends pattern queries by mapping an edge $(u, u')$ in $E_p$ to a nonempty path from $v$ to $v'$ in data graph $G$, such that $v$ can reach $v'$ within $f_e(u, u')$ hops.

It is known that when $Q_b \subseteq_{\text{sim}} G$, there exists a unique maximum match $S_e$ in $G$ for $Q_b$ [16]. Along the same lines as Section II, we define the query result $Q_b(G)$ to be the maximum set $\{e, S_e\} | e \in E_p \}$ derived from $S_e$, where $S_e$ is a set of node pairs for $e = (u, u')$ such that (1) $v$ (resp. $v'$) is a match of $u$ (resp. $u'$), and (2) the distance $d$ from $v$ to $v'$ satisfies the bound specified in $f_e(e)$, i.e., $d \leq k = f_e(e)$.

Example 8: Consider $Q_b = (V_p, E_p, f_v, f_e)$, a bounded pattern in which (1) $V_p$, $E_p$ and $f_v$ are the same as in $Q$ of Fig. 3; and (2) $f_e(A, B) = 2$, and $f_e(e) = 1$ for all the other edges $e$. The result $Q_b(G)$ in graph $G$ of Fig. 3 (a) is:

<table>
<thead>
<tr>
<th>Edge</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK, AI</td>
<td>(PK, AL1, PK, AI2)</td>
</tr>
<tr>
<td>DB, AI1</td>
<td>(DB, BI1, DB, AI1)</td>
</tr>
<tr>
<td>SE, DB</td>
<td>(SE, DB2, SE, DB)</td>
</tr>
</tbody>
</table>

Note that the pattern edge (AI, Bio) has a match (AI1, Bi01), which denotes a path ((AI1, SE1), (SE1, Bio1)) of length 2. \qed

Observe that pattern queries (Section II) are a special case of bounded patterns when $f_e(e) = 1$ for all edges $e$. While bounded patterns are more expressive, they do not incur extra complexity when it comes to query answering using views (Section VI-A) and their containment analysis (Section VI-B).

A. Answering Bounded Pattern Queries

Given a bounded pattern query $Q_b$ and a set $\mathcal{V}$ of view definitions (expressed as bounded pattern queries), the problem of answering queries using views is to compute $Q_b(G)$ by only referring to $\mathcal{V}$ and their extensions $\mathcal{V}(G)$.

Pattern containment for $Q_b$ is defined in the same way as for pattern queries. That is, $Q_b$ is contained in $\mathcal{V}$, denoted as $Q_b \subseteq \mathcal{V}$, if there exists a mapping $\lambda$ that maps each $e \in E_p$ to a set $\lambda(e)$ of edges in $\mathcal{V}$, such that for any data graph $G$, the match set $S_e \subseteq \bigcup_{e' \in \lambda(e)} S_e'$ for all edges $e$ of $Q_b$. Along the same lines as Theorem 1, one can readily verify that pattern containment also characterizes whether bounded pattern queries can be answered using views.

Theorem 8: A bounded pattern query $Q_b$ can be answered using views $\mathcal{V}$ if and only if $Q_b$ is contained in $\mathcal{V}$. \qed

Better still, answering bounded pattern queries using views is no harder than its counterpart for pattern queries.

Theorem 9: Answering bounded pattern query $Q_b$ on graph $G$ using views $\mathcal{V}$ is in $O(|Q_b||\mathcal{V}(G)| + |\mathcal{V}(G)|^2)$ time. \qed

To prove Theorem 9, we outline an algorithm for computing $Q_b(G)$ by using $\mathcal{V}$ and $\mathcal{V}(G)$ when $Q_b \subseteq \mathcal{V}$. To cope with edge-to-path mappings, it uses an auxiliary index $I(\mathcal{V})$ such that for each match $(v, v')$ in $\mathcal{V}(G)$ of some edge in $\mathcal{V}$, $I(\mathcal{V})$
Algorithm. The algorithm, denoted by BMatchJoin (not shown), takes as input Q_b, V, V(G), I(V) and a mapping \( \lambda \) from the edges of Q_b to edge sets in V. Similar to algorithm MatchJoin (Fig. 2), it evaluates Q_b by (1) “merging” views in \( V(G) \) to \( M \) according to \( \lambda \), and (2) removing invalid matches. It differs from MatchJoin in the following: for an edge \( e_p = (u, u') \) of Q_b with \( \text{changed} \) \( S_{e_p} \), it reduces match set \( S_e \) of a “parent” edge \( e = (u', u) \) in Q_b by getting the distance \( d \) (by querying I(V) in O(1) time) from \( v' \) to \( v \) (resp. \( v \) to \( v_2 \)), checking whether \( \{(v', v_1) \} \subset S_{e_1} \) (resp. \( \{(v, v_2) \} \subset S_{e_2} \) for pattern edge \( e_1 = (u, u_1) \) (resp. \( e_2 = (u, u_2) \)) such that distance \( d \) is no greater than \( f_\lambda(u', u_1) \) (resp. \( f_\lambda(u, u_2) \)), and removing \( \{v', v\} \) from \( S_e \) if no \( \{v, v_1\} \) (resp. \( \{v, v_2\} \)) exists. The removal of \( \{v', v\} \) may introduce more invalid matches in \( M \), which are removed repeatedly by BMatchJoin until a fixpoint is reached. Then \( M \) is returned as the answer.

The correctness of BMatchJoin follows from Theorem 8. One can verify that BMatchJoin takes \( O(|Q_b| |V(G)| + |V(G)|^2) \) time, the same as the complexity of MatchJoin.

Remarks. (1) Evaluating \( Q_b \) directly in a graph \( G \) takes cubic-time \( O(|Q_b||G|^2) \) [16]. In contrast, it takes \( O(|Q_b| |V(G)| + |V(G)|^2) \) time using views, and \( V(G) \) is much smaller than \( G \) in practice. (2) The optimization strategy in Section III can be naturally incorporated into BMatchJoin (see details in [4]).

B. Bounded Pattern Containment

We next show that the containment analysis of bounded pattern queries is in cubic-time, up from quadratic-time.

Theorem 10: Given a bounded pattern query \( Q_b \) and a set \( V \) of view definitions, (1) it is in \( O(|Q_b|^2 |V|) \) time to decide whether \( Q_b \subseteq V \); (2) the minimal containment problem is also in \( O(|Q_b|^2 |V|) \) time; and (3) the minimum containment problem (denoted as BMMCP) is (i) NP-complete (decision version) and APX-hard, but (ii) approximable within \( O(|E_p|) \) in \( O(|Q_b|^2 |V| + (|Q_b| \cdot \text{card}(V))^3/2) \) time.

To prove Theorem 10, we extend the notion of view matches (Section IV) to bounded pattern queries. Given a bounded pattern query \( Q_b = (V_p, E_p, f_v, f_e) \) and a view definition \( V = (V^V, E^V, f^V_v, f^V_e) \), we define the view match from \( V \) to \( Q_b \) as follows. (1) We treat \( Q_b \) as a weighted data graph in which each edge \( e \) has a weight \( f_e(e) \). The distance from node \( u \) to \( u' \) in \( Q_b \) is given by the minimum sum of the edge weights on shortest paths from \( u \) to \( u' \). (2) We define \( V(Q_b) = \{(eV, S_{eV}) \mid eV \in V \} \) as its counterpart for \( Q_b \), except that for each edge \( eV \), the distance from \( u \) to \( u' \) in \( Q_b \) is bounded by \( k \) if \( f^V_e(eV) = k \). One may verify that there exists a unique, nonempty maximum set \( V(Q_b) \) if \( V \subseteq \text{Sim}(Q_b) \). The view match \( M_{Q_b}^Q \) from \( V \) to \( Q_b \) is the union of \( S_{eV} \) for all \( eV \) in \( V \).

Example 9: Consider \( Q_b \) and \( V = \{V_1, \ldots, V_7\} \) shown in Fig. 6. One may verify that \( M_{Q_b}^Q = \{(A, B), (B, E)\} \), where the corresponding node pairs in \( Q_b \) satisfies the length constraints imposed by \( V_3 \). As another example, it can be shown that the view match \( M_{Q_b}^Q \) from \( V_7 \) to \( Q_b \) is \( \emptyset \), since the distance from \( C \) to \( D \) in \( Q_b \) is greater than 2.

Similar to Proposition 7, the result below gives a sufficient and necessary condition for \( Q_b \) containment checking.

Proposition 11: For view definitions \( V \) and bounded pattern query \( Q_b \), \( Q_b \subseteq V \) if and only if \( E_p = \bigcup_{V \in V} M_{Q_b}^V \).

Bounded pattern containment. To prove Theorem 10 (1), we give an algorithm for checking bounded pattern containment following Proposition 11, denoted by Bcontain (not shown). Bcontain is the same as contain (Section III) except that it computes view matches differently. More specifically, it extends the algorithm for evaluating bounded pattern queries [16] to weighted graphs. It can be easily verified that it is in \( O(|Q_b|^2 |V|) \) time to find all view matches for \( V \). Thus Bcontain decides whether \( Q_b \) is contained in \( V \) in \( O(|Q_b|^2 |V|) \) time, from which Theorem 10 (1) follows.

Minimal bounded containment. To show Theorem 10 (2), we give an algorithm for minimal containment checking, denoted by Bminimal (not shown). Similar to minimal (Fig. 5), Bminimal first computes view matches for each \( V \in V \), in \( O(|Q_b|^2 |V|) \) time, and unions view matches until \( E \) equals the edge set \( E_p \) of \( Q_b \) as described above. Bminimal then follows the same strategies as minimal to eliminate redundant views \( V_i \) whose removal will not cause any \( M(e) = 0 \) for each \( e \in M_{V_i} \). Thus Bminimal is in \( O(|Q_b|^2 |V|) \) time.

Minimum bounded containment. Theorem 10 (3) (i) follows from Theorem 6 (1), since MMCP is a special case of BMMCP when \( f(e) = 1 \) for all edges. To show Theorem 10 (3) (ii), we give an algorithm for minimum containment checking, denoted by Bminimal (not shown). It is similar to minimal, except that it computes view matches differently. Bminimal takes \( O(|Q_b|^2 |V|) \) time to find all view matches of \( V \). Thus, it takes \( O(|Q_b|^2 |V| + (|Q_b| \cdot \text{card}(V))^{3/2}) \) time to return a subset of \( V \) no larger than \( \log(|E_p|) \cdot \text{card}(V_{\text{OPT}}) \), where \( V_{\text{OPT}} \) is a minimum subset of \( V \) that contains \( Q_b \).

VII. EXPERIMENTAL EVALUATION

Using real-life and synthetic data, we conducted four sets of experiments to evaluate (1) the efficiency and scalability of algorithm MatchJoin for graph pattern matching using views; (2) the effectiveness of optimization techniques for MatchJoin; (3) the efficiency and effectiveness of (minimal, minimum) containment checking algorithms; and (4) the counterparts of the algorithms in (1) and (3) for bounded pattern queries.

Experimental setting. We used the following data.
(1) **Real-life graphs.** We used three 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], 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 \( x \).

(2) **Synthetic data.** We designed a generator to produce random graphs, controlled by the number \( |V| \) of nodes and the number \( |E| \) of edges, with node labels from an alphabet \( \Sigma \).

(3) **Pattern and view generator.** We implemented a generator for bounded pattern queries controlled by four parameters: the number \( |V| \) of pattern nodes, the number \( |E| \) of pattern edges \( |E_p| \), label \( f_v \) from \( \Sigma \), and an upper bound \( k \) for \( f_{x}(e) \) (Section VI), which draws an edge bound randomly from \([1, k]\). When \( k = 1 \) for all edges, bounded patterns are pattern queries. We use \((|V_p|, |E_p|)\) (resp. \((|V_p|, |E_p|, k)\)) to present the size of a (resp. bounded) pattern query.

We generated a set \( \mathcal{V} \) of 12 view definitions for each real-life dataset. (a) For Amazon, we generated 12 frequent patterns following [27], where each of the view extensions contains in average 5\( K \) nodes and edges. The views take 14.4\% of the physical memory of the entire 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 for Youtube, shown in Fig. 7, where each node specifies videos with Boolean search conditions specified by \( e.g., \) age (A), length (L), category (C), rate (R) and visits (V). Each view extension has about 700 nodes and edges, and put together they take 4\% of the memory for Youtube.

For synthetic graphs, we randomly constructed a set \( \mathcal{V} \) of 22 views with node labels drawn from a set \( \Sigma \) of 10 labels. We cached their view extensions (query results), which take in total 26\% of the memory for the data graphs.

(4) **Implementation.** We implemented the following algorithms, all in Java: (1) contain, minimum and minimal for checking pattern containment; (2) Bcontain, Bminimun and Bminimal for bounded pattern containment; (3) Match, MatchJoin\_\text{min} and MatchJoin\_\text{nnl}, where Match is the matching algorithm without using views [16], [21]; and MatchJoin\_\text{min} (resp. MatchJoin\_\text{nnl}) revises MatchJoin by using a minimum (resp. minimal) set of views; (4) BMATCH, BMATCHJoin\_\text{min} and BMATCHJoin\_\text{nnl}, where BMATCH evaluates bounded pattern queries without using views [16], and BMATCHJoin\_\text{min} and BMATCHJoin\_\text{nnl} are the counterparts of MatchJoin\_\text{min} and MatchJoin\_\text{nnl} for bounded pattern queries, respectively; and (5) a version of MatchJoin (resp. BMATCHJoin) without using the ranking optimization (Section III), denoted by MatchJoin\_\text{nopt} (resp. BMATCHJoin\_\text{nopt}).

All the experiments were run on a machine powered by an Intel Core(TM)2 Duo 3.00GHz CPU with 4GB of memory, using scientific Linux. 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 graph pattern matching using views, \( i.e., \), algorithms MatchJoin\_\text{min} and MatchJoin\_\text{nnl}, compared to Match [16], [21]. Using real-life data, we studied the efficiency of MatchJoin\_\text{min}, MatchJoin\_\text{nnl} and MatchJoin, by varying the size of the queries. We also evaluated the scalability of these three algorithms with large synthetic datasets.

**Efficiency.** Figures 8(a), 8(b) and 8(c) show the results on Amazon, Citation and YouTube, respectively. The \( x \)-axis represents pattern size \((|V_p|, |E_p|)\). The results tell us the following. (1) MatchJoin\_\text{min} and MatchJoin\_\text{nnl} substantially outperform Match, taking only 45\% and 57\% of its running time on average over all real-life datasets. (2) All three algorithms spend more time on larger patterns. Nonetheless, MatchJoin\_\text{min} and MatchJoin\_\text{nnl} are less sensitive than Match, since they reuse previous computation cached in the views.

**Scalability.** Using large synthetic graphs, we evaluated the scalability of MatchJoin\_\text{min}, MatchJoin\_\text{nnl} and Match. Fixing pattern size with \(|V_p| = 4\) and \(|E_p| = 6\), we varied the node number \(|V|\) of data graphs from 0.3M to 1M, in 0.1M increments, and set \(|E| = 2|V|\). As shown in Fig. 8(d), (1) MatchJoin\_\text{min} scales best with \(|G|\), consistent with the complexity analysis of MatchJoin; and (2) MatchJoin\_\text{min} accounts for about 49\% of the time of MatchJoin\_\text{nnl}. This verifies that evaluating pattern queries by using less view extensions significantly reduces computational time, which is consistent with the observation of Figures 8(a), 8(b) and 8(c).

To further evaluate the impact of pattern sizes on the performance of MatchJoin\_\text{min}, we generated four sets of patterns with \((|V_p|, |E_p|)\) ranging from \((4, 8)\) to \((7, 14)\), kept \(|E_p| = 2|V_p|\), and varied \(|G|\) as in Fig. 8(d). The results are reported in Fig. 8(e), which tell us the following. (1) MatchJoin\_\text{min} scales well with \(|Q|\), which is consistent with Fig. 8(d). (2) The larger \(Q_p\) is, the more costly MatchJoin\_\text{min} is. For larger \(Q_p\), more views may be needed to “cover” \(Q_p\); and MatchJoin\_\text{min} takes longer time, using the selected views.

**Exp-2:** **Optimization techniques.** We also evaluated the effectiveness of the optimization strategy given in Section III, by comparing the performance of MatchJoin\_\text{min} and MatchJoin\_\text{nopt} using patterns of size \((4, 6)\) and same set of views. The synthetic graphs are generated following the densification law [26]: \(|E| = |V|^p\). Fixing \(|V| = 200K\), we varied \(p\) from 1 to 1.25 in 0.05 increments. As shown in Fig. 8(f), MatchJoin\_\text{min} is more efficient than MatchJoin\_\text{nopt} over all the datasets. Indeed, the running time of MatchJoin\_\text{min} is on average 54\% of that of MatchJoin\_\text{nopt}. The improvement becomes more evident when \(p\) increases. This is because
when graphs become dense, more redundant edges can be removed by the bottom-up strategy used in MatchJoin\textsubscript{min}. The results for BMatchJoin\textsubscript{min} and BMatchJoin\textsubscript{opt} are consistent with Fig. 8(f) and are hence not shown.

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

**Efficiency of containment.** We generated two sets of DAG and cyclic patterns, denoted by $Q_{\text{DAG}}$ and $Q_{\text{cyclic}}$, respectively. Fixing a set of synthetic views $\mathcal{V}$, we varied the pattern size from (6, 6) to (10, 20), where each size corresponds to a set of patterns with different structures and/or node labels. As shown in Figure 8(g), (1) contain is efficient, e.g., it takes only 39 ms to decide whether a cyclic pattern with $|V_p| = 10$ and $|E_p| = 20$ is contained in $\mathcal{V}$; (2) contain takes more time over larger DAG and cyclic patterns, as expected; and (3) when pattern size is fixed, cyclic patterns cost more than DAG patterns for contain, due to a more time-consuming fixpoint process.

**minimum vs minimal.** To measure the performance of minimum and minimal, we define $R_1 = |T_{\text{min}}|/|T_{\text{ml}}|$ as the ratio of the time used by minimum to that of minimal; and $R_2 = |\text{Minimal}|/|\text{Minimal}|$ for the ratio of the size of subsets of views found by minimum to that of minimal. Using the same view definitions and cyclic patterns as in Figure 8(g), we varied the size of patterns from (6, 6) to (10, 20). As shown in Fig. 8(h), (1) minimum is efficient on all patterns used, e.g., it takes about 0.4s over patterns with 10 nodes and 20 edges; (2) minimum is effective; while minimum takes up to 120% of the time of minimal ($R_1$), it finds substantially smaller sets of views, only about 40%–55% 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: Efficiency and scalability of BMatchJoin.** In this set of experiment we evaluated (1) the efficiency of BMatchJoin\textsubscript{min} vs. BMatchJoin\textsubscript{opt} and BMatchJoin over the real-life datasets, by varying the size of pattern queries, and (2) the scalability of BMatchJoin\textsubscript{min} over large synthetic graphs, by varying the size of patterns and data graphs.

**Efficiency.** We used the same patterns as for MatchJoin in Exp-1, except that the edge bounds of the patterns were set to be $f_e(e) = 2$ (resp. $f_e(e) = 3$) for queries over Amazon (resp. Citation). Figure 8(i) shows the results on Amazon in which the x-axis ($|V_p|, |E_p|, f_e(e)$) indicates the size of patterns $Q_s = (V_p, E_p, f_e)$. From the results we find that BMatchJoin\textsubscript{min} and BMatchJoin\textsubscript{opt} perform much better than BMatch: (1) BMatchJoin\textsubscript{min} (resp. BMatchJoin\textsubscript{opt}) needs only 10% (resp. 14%) of the time of BMatch; (2) when pattern size increases, running time of BMatchJoin\textsubscript{min} (resp. BMatchJoin\textsubscript{opt}) grows slower than that of BMatch; and (3) BMatchJoin\textsubscript{opt} always outperforms BMatchJoin\textsubscript{opt}. These are consistent with the result for Citation, shown in Fig. 8(j).

Fixing pattern size with $|V_p| = 4$ and $|E_p| = 8$, we varied $f_e(e)$ from 2 to 6. As shown in Fig. 8(k), (1) BMatchJoin\textsubscript{min} substantially outperforms BMatch; when $f_e(e) = 3$, for example, BMatchJoin\textsubscript{min} accounts for only 3% of the computational time of BMatch; (2) the larger $f_e(e)$ is, the more costly BMatch is, as it takes longer for BFS to identify ancestors or descendants of a node within the distance bound $f_e(e)$; and (3) BMatchJoin\textsubscript{opt} is more efficient than BMatchJoin\textsubscript{opt}, as it uses less views.

**Scalability.** Fixing $|V_p| = 4, |E_p| = 6$ and $f_e(e) = 3$, we varied $|V|$ from $0.3M$ to $1M$ in $0.1M$ increments, while letting $|E| = 2|V|$. As shown in Fig 8(l), (1) BMatchJoin\textsubscript{min} scales best with $|G|$; this is consistent with its complexity analysis; and


(2) BMatchJoin_{\text{min}} takes only 6% of the computation time of BMatch, and the saving is more evident when $G$ gets larger.

Summary. We find the following. (1) Answering (bounded) pattern queries using views is effective in querying large social graphs. For example, by using views, matching via bounded simulation takes only 3% of the time needed for computing matches directly in YouTube, and 6% on synthetic graphs. For graph simulation, the improvement is over 51% at least. (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 (bounded) pattern query can be answered using views. In particular, our approximation algorithm for minimum containment effectively reduces redundant views, which in turn improves the performance of matching by 55% (resp. 94%) for (resp. bounded) pattern queries. (4) Better still, our optimization strategy further improves the performance of pattern matching using views by 46%.

VIII. Conclusion

We have studied graph pattern matching 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. 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. Moreover, our techniques can be readily extended to strong simulation [28], retaining the same complexity.

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 may apply. Another issue is to develop efficient algorithms for computing maximally contained rewriting using views, when a pattern query is not contained in available views [25]. A third problem concerns view-based pattern matching via subgraph isomorphism. The fourth 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, Wang and Wu are supported in part by 973 Programs 2012CB316200 and 2014CB340302, NSFC 6113002, Guangdong Innovative Research Team Program 2011D005 and Shenzhen Peacock Program 1105100030834361 of China, as well as EPSRC EP/J015377/1, UK.

References
