Querying DAG-shaped Execution Traces Through Views

Maya Ben-Ari Tova Milo Elad Verbin

Tel Aviv University

{mayaben,milo,eladv}@cs.tau.ac.il

ABSTRACT

The question whether a given set of views, defined by queries, can be used to answer another query, arises in several contexts such as query optimization, data integration and semantic caching [24, 10, 12]. This paper studies a specific instance of this problem, where the queried data has the shape of a *DAG* (Directed Acyclic Graph) and the query language uses *DAG patterns* to retrieve portions of the data graph that are of interest. Our study is motivated by a particular application domain concerning the analysis of Web-based Business Processes (BPs for short). Such DAGs / DAG patterns are the standard way to model / query BP execution traces [3].

Previous research considered tree-shaped XML data and (general) graph-shaped Semi-Structured data. We show that the particular DAG shape of BP execution traces makes the problem easier than for general graphs, yet harder than for XML trees. Specifically, we show which combinations of DAG classes and query features allow for PTIME query answering algorithms and which lead to NP-complete problems.

1. INTRODUCTION

The question whether a given set of views, defined by queries, can be used to answer another query, arises in several contexts such as data integration, query optimization and semantic caching [24, 10, 12]. In this work we study a specific instance of this problem, where the queried data has the shape of a *DAG* (Directed Acyclic Graph) and the query language uses *DAG patterns* to retrieve portions of the data graph that are of interest.

Our study is motivated by a particular application domain concerning the analysis of Web-based Business Processes (BPs for short). A BP is a collection of logically related activities that, when combined in a flow, achieve a business goal. An execution flow of a BP can be viewed as a DAG, containing nodes that represent the business activities that took place and edges that describe their flow and causal relationship [3]. As a simple example, consider the BP of an on-line Web-based travel agency. The execution DAG here may include activities (nodes) such as login, flight/hotel search, reservation and payment, and edges that describe their (possibly parallel) execution order.

The analysis of execution flows is extremely valuable for companies. It allows to optimize business processes, reduce operational costs, and ultimately increase competitiveness. Execution flows are thus often *traced*, and the recorded execution traces are stored in repositories. In a typical BP analysis, the repository is first queried to select portions of the execution traces that are of particular interest. These then serve as input for a finer analysis that further queries and mines the subtraces to derive critical business information [11]. The subtraces of interest are selected using DAG patterns, an adaptation of the tree- and graph-patterns offered by existing query languages for XML and graph-shaped data to execution DAGs.

As usual in query processing, good performance is critical. The reuse of previously computed query answers has proved to be useful for query optimization in general, and for the optimization of queries of XML and graph-shaped Web data in particular [12, 23, 19]. To enable such reuse, one needs to determine whether the given query can be answered based on the set of answers to previous queries. This paper studies this problem, for the first time, for an important class of DAG-shaped data that describes execution traces of BPs. Particular attention is payed here to a family of DAGs, called *series-parallel DAGs* [15], which captures execution traces of BPs defined using the BPEL (Business Process Execution Language) standard [5, 3].

The question whether a set of query answers (views) can be used to answer another query has been extensively studied previously for relational data [20, 17, 14], tree-shaped XML data [2, 19, 23], and (general) graph-shaped semi-structured data [8, 6]. We show here that the DAG shape of execution traces makes the problem easier than for general graphs, yet harder than for XML trees. Specifically, we consider queries with and without projection. For queries without projection, we prove that the problem is NP-complete for arbitrary DAGs, (in contrast to the PTIME complexity of the analogues problem for XML trees). Nevertheless, we show that it can be solved in PTIME for the class of series-parallel DAGs. Our solution is based on a syntactic characterization of *answerable queries*, along with a dedicated dynamic programming algorithm that allows to efficiently test the fulfillment of the syntactic requirements. For queries with projection, we show that the above syntactic

Twelfth International Workshop on the Web and Databases (WebDB 2009), June 28, 2009, Providence, Rhode Island, USA.

characterization no longer holds and the problem becomes NP-hard even for series-parallel DAGs. However, we present a stricter syntactic characterization of *answerable projection queries*, and use it (1) to present an algorithm (of Σ_2 time complexity) for the case of general DAGs, and (2) to show that, under plausible restrictions on the queries, PTIME complexity may still be achieved for series-parallel DAGs.

Paper organization. Section 2 describes our data model and query language. Section 3 considers queries without projection and Section 4 queries with projection. We conclude in Section 5 with an overview of related work.
For space constraints, the proofs are omitted and can be found in the full version of the paper [4].

2. PRELIMINARIES

In this section we present the data model and query language that we consider, and formally define the problem studied in the paper. To get a handle on the difficulty of the problem we start by considering a very simple query language, then extend it in the following sections.

DAGs and DAG patterns. A BP execution trace describes a set of activities and the order in which they occurred, and is abstractly modeled as a labeled DAG [3]. In the sequel, let $\mathcal{N}$ be an infinite domain of graph nodes, each having a unique id, and let $\mathcal{A}$ be an infinite domain of activity names.

DEFINITION 2.1. *A* labeled Directed Acyclic Graph *(abbr. DAG) is a triplet (N, E, λ) where $N \subset \mathcal{N}$ is a finite set of nodes, $E \subseteq N \times N$ is a non empty set of directed edges, and $\lambda : N \to \mathcal{A}$ is a labeling function for the nodes. The graph is required to be acyclic.*
A DAG pattern *(abbr. PDAG) is a pair (D, E_t) where $D = (N, E, \lambda)$ is a DAG and $E_t \subseteq E$ is a subset of the edges of D called* transitive *edges. The remaining edges in E are called* direct.

A PDAG describes partial information about the execution trace. A transitive edge between two nodes n_1, n_2 of a PDAG d represents knowledge about the existence of a path from n_1 to n_2, without details on which specific nodes are on the path. Note that if d contains some other path between n_1 and n_2, then such a transitive edge is redundant as it brings no additional information. We assume in the sequel that the PDAGs contain no redundant transitive edges (if such edges exist they are automatically removed).

EXAMPLE 2.1. *The DAG in Figure 1(a) describes a possible execution trace of the Web-based travel agency from the Introduction. The user logins and then searches (in parallel) for flights and for combined deals that include a flight and a hotel reservation. She pays (separately) by credit card for the flight and the combined deal, and the full reservation is confirmed. The a_i labels represent the node ids (to be used in the sequel). The PDAG in Figure 1(b) details part of this execution trace. (Ignore for now the darker/lighter background of the nodes). The solid (resp. dashed) arrows are direct (transitive) edges. We can see here that after login, the user performs (at some point) hotel and flight search. Then, (possibly after a sequence of other operations) she pays and the reservation is next confirmed.*

We will pay here particular attention to a family of (P)DAGs, called *series-parallel* [15], which captures execution traces of

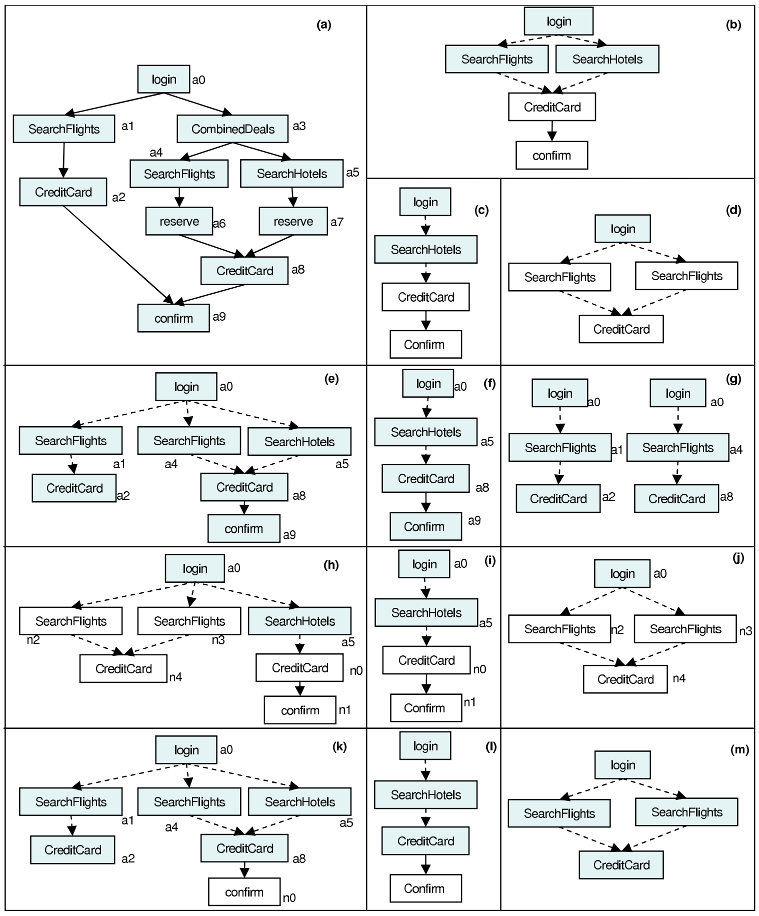

Figure 1: DAG and PDAGs

BPs defined using the BPEL (Business Process Execution Language) standard [5, 3]. The definition is standard:

DEFINITION 2.2. *A (P)DAG d is* series-parallel *if it has a single* `start` *node without incoming edges, a single* `end` *node without outgoing edges, and one of the following holds.*

1. `[base]` *d consists of a pair of nodes connected by a single (direct or transitive) edge.*
2. `[series composition]` *d_1, d_2 are two series-parallel (P)DAGs where the end node of d_1 and the start node of d_2 have the same label, and d is obtained from d_1, d_2 by merging the two nodes into a single one.*
3. `[parallel composition]` *d_1, d_2 are two series-parallel (P)DAGs whose start (resp. end) nodes have the same label and d is obtained from d_1, d_2 by merging their two start (end) nodes into a single start (end) node.*

For example, the DAG in Figure 1(a) is series-parallel. The same DAG with an additional edge from node $a1$ to $a4$ is no longer series-parallel.

Queries and answers. PDAGs will play two roles here. First, they are used as *queries*, to select parts of the execution traces (DAGs) whose shape matches that of the query PDAG. Thus, whenever we use in the following the term *query* we mean a PDAG. Second, they are used to model the *partial information* about the execution traces that is retrieved by queries.

To optimize the processing of a new query, it is useful to reuse the (partial) information obtained by previously computed queries [12, 23, 19]. In our setting this means evaluating the query on the PDAGs obtained by previous queries, rather than on the original full DAG. To define this formally, we define below the semantic of queries when evaluated on (P)DAGs. We use for that an auxiliary notion of query *embedding.*

Definition 2.3. *For a (P)DAG d and a query q, an* embedding *of q into d is a homomorphism ψ from the nodes of q to the nodes of d, that (1) preserves the node labels and (2) for each direct (transitive) edge in q from a node m to a node n, there exists a direct edge (path consisting of direct or transitive edges) in d from $\psi(m)$ to $\psi(n)$.*

Definition 2.4. *For a (P)DAG d and an embedding ψ of q into d, the* answer *defined by ψ is the image of q under ψ. Namely, each node n in q is assigned the id of $\psi(n)$, and if a node id occurs several times in the image, its multiple occurrences are merged into a single node having the same id as the original one.*

For a (P)DAG d, the answer of q on d, denoted $q(d)$, consists of the set of answers defined by all the possible embeddings of q into d.

Example 2.2. *For example, the answer of the query in Figure 1(c) on the DAG in Figure 1(a) consists of the single PDAG depicted in Figure 1(f). (Ignore for now the darker/lighter background of some of the query nodes). The answer of the query in Figure 1(d) on the same DAG consists of two PDAGs, depicted in Figure 1(g). In each of the two embeddings that yielded these PDAGs, the two* `SerachFlight` *nodes of the query were mapped to one node ($a1$ and $a4$ respectively). Thus the two answers consist each of a single (merged) occurrence of* `SearchFlights`.

Answerable queries.

Given the answers of a set of queries, the partial information that they reveal, together, about the structure of the original DAG, is realized by gluing together nodes with identical ids. Formally,

Definition 2.5. *Given a set Q of queries and a (P)DAG d, the* partial view *of d (given by Q), denoted $Partial(Q, d)$, is the PDAG obtained from $\cup_{q_i \in Q} q_i(d)$ by merging multiple occurrences of nodes having the same id into a single node with that id.*

Example 2.3. *Continuing our example, for Q consisting of the two queries in Figures 1(c) and 1(d), and the DAG d in Figure 1(a), $Partial(Q, d)$ is the PDAG in Figure 1(e). It is obtained from the query answers in Figures 1(f) and 1(g), by merging the multiple occurrences of the $a0$ nodes (*`Login`*) and the $a8$ nodes (*`CreditCard`*).*

We are now ready to formally define when a query q is answerable, *for every PDAG d*, using the partial information obtained by a given set Q of queries.

Definition 2.6. *A query q is* answerable *by a set Q of queries iff for all PDAGs d, $q(d) = q(Partial(Q, d))$.*

Given a set Q of queries and a query q, we call the problem of testing whether q is answerable by Q the *query answering* problem.

3. QUERY ANSWERING

We next consider the complexity of the *query answering* problem. We first present a syntactic criterion that allows to solve the problem, then use it for complexity analysis.

Onto Embedding.

Given a set Q of queries and a query q, we consider a particular class of embeddings of queries in Q into q which we call an *onto embedding.* We then show that the existence of such an embedding provides a necessary and sufficient condition for q to be answerable by Q.

We have defined in the previous section (Definition 2.3) an embedding of a *single* query q to a (P)DAG d. The definition naturally extends to a set of queries, with the homomorphism ψ now applying to the nodes of all queries, satisfying the same requirements as before. We use this extended notion of embedding below.

Definition 3.1. *An* onto embedding *ψ from a set Q of queries to a query (PDAG) q is an embedding of Q to q s.t. for each direct (resp. transitive) edge $[n, m]$ of q there exists a direct (transitive) edge $[n', m']$ of some $q' \in Q$ s.t. $\psi(n') = n$ and $\psi(m') = m$.*

Example 3.1. *The set Q consisting of the two queries in Figures 1(c) and 1(d) has an onto embedding to the query in Figure 1(b): It maps each node in 1(c) and 1(d) to the (unique) node in 1(b) of the same label. On the other hand, a set Q consisting of just one of these queries does not have such an onto embedding as it cannot "cover" all of 1(b).*

An onto embedding requires that each edge in q has a corresponding edge in Q of the same type (direct or transitive). Note that Q may contain multiple isomorphic copies of the same PDAG (differing only in their node ids) which can be used to "cover" distinct parts of q having similar shapes.

The notion of onto embedding is utilized to provide a necessary and sufficient condition for query answerability. Given two sets of queries Q, Q', we use below $Q' \subseteq Q$ to denote that every query $q' \in Q'$ has some query $q \in Q$ that is identical to it up to node isomorphism. (A query $q \in Q$ may have several such isomorphic copies in Q'.)

Theorem 3.1. *Given a set Q of queries and a query q, q is* answerable *by Q iff there exists a set of queries $Q' \subseteq Q$ that has an onto embedding to q.*

For space constraints we omit the proof (see [4] for details) and illustrate the theorem with an example.

Example 3.2. *We saw above that the set Q consisting of the two queries in Figures 1(c) and 1(d) has an onto embedding to the query q in Figure 1(b). q is thus answerable by Q. E.g., q's answer, for the DAG d in Figure 1(a), is the same as its answer for $Partial(Q, d)$ in Figure 1(e).*

We can now use this theorem to provide an NP algorithm for the *query answering* problem. First note that to "cover" all the edges of q it suffices to consider a set Q' of queries whose size is bounded by the size of q. Thus the NP algorithm simply guesses $|q|$ queries from Q (possibly with multiple choices of the same query), as well as a homomorphisms from their nodes to those of q, and checks if it satisfies the onto embedding requirements. We next show that, unless P=NP, a PTIME algorithm is unlikely to exist.

Theorem 3.2. *The query answering problem is NP-hard even when Q and q contain no transitive edges.*

The proof is by reduction from the *clique* problem, known to be NP-complete [18].

Series-parallel PDAGs. While *query answering* is NP-complete in general, it turns out to be solvable in PTIME for the class of series-parallel PDAGs.

THEOREM 3.3. *If the queries in Q are series-parallel, the query answering problem for a (not necessarily series-parallel) query q is solvable in time polynomial in the size of Q and q.*

Algorithm. [Sketch] Our PTIME algorithm is based on a subroutine called `FindEmbeddings`. Given a series-parallel query $\hat{q}$, a (general) query q and an edge e in q, `FindEmbeddings` returns (1) all pairs of nodes n, m in q s.t. there exists an embedding of $\hat{q}$ to q, mapping the start and end nodes of $\hat{q}$ to n and m, resp., and (2) annotates each such n, m pair by "true" if any of its corresponding embeddings "covers" e (in the sense of Definition 3.1), and "false" otherwise. We first describe how `FindEmbeddings` works, then explain how it is used to solve the *query answering* problem.

FindEmbeddings. As shown in [22, 25], one can determine whether a given graph is series-parallel in linear time, and a tree describing its series-parallel construction steps may also be built in linear time. We denote this tree for $\hat{q}$ by $\hat{q}_{tree}$. `FindEmbeddings` is a dynamic programming algorithm that works on this tree bottom up. Each node in the tree represents a series-parallel sub-query q' of $\hat{q}$. For every such node it computes a boolean matrix $A_{q'}$ of size $|q| \times |q|$ s.t. $A_{q'}[n, m] = 1$ if there exists an embedding of q' to q mapping the start and end nodes of q' to nodes n and m in q, resp., and 0 otherwise. For the leaves of the tree (representing the base single-edge queries) the matrix is computed by simply matching the edge into q. The matches that cover the edge e are annotated by "true". For higher tree nodes, the matrix $A_{q'}$ is computed (using boolean matrix multiplication) from the matrices of the node's children (representing the sub-queries from which q' is composed), and the "true" values are propagated. We omit the details.

In total, `FindEmbeddings` works in time $O(|\hat{q}||q|^{\omega})$, where $\omega = 2.376$ is the matrix multiplication constant [9].

Query answering. Given a set Q of series-parallel queries and a general query q, we can solve the query answering problem by running `FindEmbeddings` for each query $\hat{q} \in Q$ and each edge e of q. q is answerable iff each of its edges is covered by (at least) one of the embeddings of the queries in Q. In total, `FindEmbeddings` is invoked here $|Q||q|$ times, yielding an overall time complexity of $O(|Q|^2|q|^{1+\omega})$.

4. PROJECTION QUERIES

We next consider queries with projection. We will see that the *query answering* problem becomes harder here.

Definitions. We first extend the definition of queries by denoting some of the PDAG nodes as output nodes.

DEFINITION 4.1. *A PDAG (query)* with projection *is a pair (d, O) where d is a PDAG and O is a subset of the nodes of d called the* output nodes. *The remaining nodes of d are called* non-output nodes.

Definition 2.4 of *query answers* is naturally adapted to this context: When a projection query q is applied on a PDAG d with projection, only embeddings that map the output nodes of q to output nodes of d are considered. When q's answer, for an embedding ψ, is constructed, only the output nodes of q are assigned the id of their image in d. Non-output nodes are assigned arbitrary new ids which only record the fact that *some* nodes of d were matched to them in the embedding (but not which ones). Two answers here are considered identical if they are isomorphic up to the new ids assigned to the non-output nodes.

The (P)DAGs from the previous section can be thought of as projection ones with all nodes being output nodes.

EXAMPLE 4.1. *The PDAGs in Figures 1(b)-1(d), with the darker background denoting the output nodes, are projection queries. The answers of the projection queries in Figures 1(c) and 1(d), on the DAG d in Figure 1(a), are depicted in Figures 1(i) and 1(j) resp. The a_i nodes (with darker background) come from d, whereas the n_i's are fresh new ids. Note that the two distinct embeddings of query 1(d) into d yield here just one answer, as they differ only in the mappings of non-output nodes.*

As before (Definition 2.5), given a set Q of queries and a (P)DAG d, the partial view of d given by Q, denoted $Partial(Q, d)$, is the PDAG obtained from the query answers by merging multiple occurrences of nodes having the same id. Note that here only output nodes may be merged (since non-output nodes all have distinct new ids).

EXAMPLE 4.2. *Continuing with our example, for Q consisting of the two projection queries in Figures 1(c) and 1(d), and the DAG d in Figure 1(a), $Partial(Q, d)$ is the PDAG in Figure 1(h) (obtained from the answers in Figures 1(i) and 1(j)).*

Answerable projection queries are now defined using the above refined definitions of *query answer* and $Partial(Q, d)$:

DEFINITION 4.2. *A projection query q is* answerable *by a set Q of projection queries iff for all PDAGs d, $q(d)$ and $q(Partial(Q, d))$ are the same up to isomorphism on the new node ids not appearing in d.*

Minimal queries. To solve the *query answering* problem for projection queries, we introduce the notion of *minimal queries.* Given a query q, our algorithm will first compute a minimized version q' of q, then test for the answerability of q'. We will show that this suffices to solve the problem for the original query q. For brevity, we omit from now on the word 'projection' and, unless stated otherwise, whenever we use the term query we mean a projection one.

To define minimal queries we use an auxiliary notion of query containment.

DEFINITION 4.3. *A query q' is* contained *in a query q, denoted $q' \preceq q$, if there exists an embedding ψ from q' to q s.t. (1) each* output *node of q' is mapped to an* output *node of q and (2) each* output *node of q has at least one* output *node of q' that is mapped to it. Two queries q' and q are* equivalent *(denoted $q' \doteq q$) if $q' \preceq q$ and $q \preceq q'$.*

A query q is minimal *if there is no query $q' \doteq q$ which can be obtained from q by removing one or more edges or non-output nodes and possibly replacing previously existing paths by transitive edges.*

EXAMPLE 4.3. *The query q', obtained from the query q in Figure 1(d) by removing one* `SearchFlights` *node and its incoming/outgoing edges, is contained in q. q is also contained in q' (with ψ mapping each node in q to the unique node in q' of the same label). Thus $q' \doteq q$. q' is minimal.*

Observe that given a query q, one can obtain an equivalent minimal query by, iteratively, trying to remove non-output nodes (replacing then by corresponding transitive edges) and edges, and testing for equivalence to the resulting query. The complexity of such a minimization process is determined by the following proposition.

PROPOSITION 4.1. *Given a query q, the problem of testing if q is minimal is coNP-complete. For series-parallel queries, the problem can be solved in PTIME.*

To conclude, the following proposition guarantees that to determine whether a query q is answerable by a set Q of queries it suffices to examine a minimized version q' of q.

THEOREM 4.1. *For every two equivalent queries q, q' and a set Q of queries, q is answerable by Q iff so is q'.*

Query answering. We now show how the *query answering* problem is solved for minimal queries. In Section 3 we saw a syntactic characterization that allowed to decide whether a query is answerable by a given set of queries. We start by showing that for queries with projection this syntactic characterization no longer holds. We then present a stricter characterization and use it to solve the problem.

First let us illustrate that Theorem 3.1 no longer holds for projection queries.

EXAMPLE 4.4. *Consider the set Q consisting of the two projection queries in Figures 1(c) and 1(d). We saw in Example 3.1 that Q has an* onto embedding *to the query q in Figure 1(b). However the answer of q when evaluated (as projection query) on the DAG d in Figure 1(a) differs from its answer when evaluated on $Partial(Q, d)$ in Figure 1(h).*

There are two reasons for the difference here. The first is caused by the fact that the `SearchFlights` output node of q does not have a corresponding output node in Q. Thus the relevant node ids of d do not appear in $Partial(Q, d)$. But even if the `SearchFlights` nodes had been output nodes in Q, the answers would still be different: to merge properly the results of the two queries, the common `CreditCard` nodes must retain a common id. Hence the corresponding query nodes must be output nodes (which is not the case here).

Following the above discussion we define a stricter notion of *strong* onto embedding, which assures that the answers of the queries in Q contain *all the information* required by q and can be *assembled* properly together.

DEFINITION 4.4. *An onto embedding ψ from a set Q of queries to a query q is called* strong *if there exists a homomorphism $\hat{\psi}$ that is a restriction of ψ to a subset of the nodes in Q, satisfying the following.*

- *Each direct (resp. transitive) edge $e = [n, m]$ in q has exactly one corresponding direct (transitive) edge $e' = [n', m']$ in Q s.t. $\hat{\psi}(n') = n$ and $\hat{\psi}(m') = m$, and if n (m) is an output node so is n' (m').*
- *For every pair e_1, e_2 of adjacent edges in q with common endpoint n, in their corresponding edges e'_1, e'_2 in Q the two endpoints that are mapped to n are either the same node (i.e. e'_1, e'_2 are also adjacent in Q) or are both output nodes.*

The following theorem shows that the existence of a strong onto embedding provides a necessary and sufficient condition for query answerability of *minimal* queries.

THEOREM 4.2. *Given a set Q of queries and a minimal query q, q is* answerable *by Q iff there exists a set of queries $Q' \subseteq Q$ that has a* strong *onto embedding to q.*

We omit the proof for space consideration and only illustrate things with an example.

EXAMPLE 4.5. *The set Q consisting of the two queries in Figures 1(l) and 1(m) has a strong onto embedding to the query q in Figure 1(b): For $Q' = Q$, the embedding ψ maps the nodes in Q' to the nodes of q having the same label, and $\hat{\psi}$ simply ignores one of the* `SearchFlights` *nodes. Indeed the result of q when evaluated on the DAG d in Figure 1(a) is the same as its result when evaluated on $Partial(Q, d)$ depicted in Figure 1(k). In contrast, the set Q consisting of the queries in Figures 1(c) and 1(d) does not have a strong onto embedding to q. Indeed we saw above that in this case q's answer differs for d and $Partial(Q, d)$ of Figure 1(h).*

Complexity. The above theorem, together with Proposition 4.1 and Theorem 4.1, yields an algorithm for the *query answering* problem: Given Q and q, guess a minimal equivalent query q' of q, then guess an onto embedding from Q to q' (as done for queries without projection), and check if it is a strong one. The complexity class is Σ_2 (an NP algorithm with a coNP oracle).

Let us consider now series-parallel queries. While minimization can be done for them in PTIME, the search for strong onto embedding cannot. Indeed, we can show:

THEOREM 4.3. *The query answering problem for projection queries is NP-hard even if only series-parallel queries are considered.*

The proof is by reduction to the *Perfect 3-Dimensional Matching (P3DM)* problem [18].

Intuitively, this high complexity is caused by the large number of possible embeddings from Q to q that need to be examined (an exponential number in the worst case). In practice, their number is unlikely to reach this worst case upper bound. For instance, if each label appears in a query at most once, the possible embedding is uniquely determined. We next show that if the number of embeddings is bounded, the problem becomes tractable. The crux is to avoid examining, for each embedding ψ, each of its possible restrictions $\hat{\psi}$ individually (since there is an exponential number of such possible restrictions) and instead factorize their analysis.

THEOREM 4.4. *Given a set Q of series-parallel queries, a series-parallel query q, and a bound b on the number of possible embeddings of queries from Q into q, the query answering problem can be solved in time polynomial in the size of Q, q, and b.*

Algorithm. [Sketch] W.l.o.g. assume that q is minimal (otherwise it can be minimized in PTIME). The algorithm first computes all the possible embeddings ψ of queries $\hat{q} \in Q$ into q. (We show that for series-parallel queries this can be done in time polynomial in the queries size and the number of embeddings). For each embedding ψ, we construct an isomorphic copy $\hat{q}_\psi$ of $\hat{q}$, and denote the new set of queries by $\hat{Q}$. To solve the problem it suffices to determine if there exists a subset Q' of $\hat{Q}$ that satisfies the conditions of the strong onto embedding w.r.t. q.

We construct, as in the algorithm of Theorem 3.3, a series-parallel decomposition tree for q, denoted q_{tree}. Each node in the tree represents a series-parallel sub-query q' of q. Our algorithm works on this tree bottom up. It computes for each node a boolean matrix $A_{q'}$ of size $(N+1) \times (N+1)$, where N is the number of non output nodes in $\hat{Q}$. Intuitively, an entry $A_{q'}[n,m] = 1$, for $n, m \leq N$, represents the fact that there exists a strong onto embedding from $\hat{Q}$ to q', mapping n (m) to the start (end) node of q'; an entry $A_{q'}[n,m] = 1$, $n \leq N$, $m = N+1$, (resp. $n = N+1, m \leq N$) represents the fact that there exists such a strong onto embedding with n (m) being mapped to the start (end) node of q' and where some output node is mapped to the end (start) node of q'; an entry $A_{q'}[n,m] = 1$, $n = m = N+1$ represents the fact that some strong onto embedding to q' exists with the nodes mapped to the start and end nodes of q' both being output nodes.

For the leaves of the tree (representing the base single-edge queries) the matrix is computed by simply matching the edges in $\hat{Q}$ to the edges of the corresponding base queries. For higher nodes in the tree, $A_{q'}$ is computed from the matrices of the node's children. (We omit the details). At the end of the computation we return 'true' iff some entry in the matrix A_q, of the root of q_{tree}, is marked by 1.

5. RELATED WORK AND CONCLUSION

This paper studies the question whether a given set of views, defined by queries, can be used to answer another query, for a particular case where the queried data has the shape of a *DAG* and the query language uses *DAG patterns*. We showed which combinations of DAG classes and queries allow for PTIME algorithms and which lead to NP-complete problems. DAGs and DAG patterns are the standard way to model and query BP execution traces [3]. Specifically, the (P)DAGs that are used here are a simplified abstraction of the data model and query language employed by the BPQL system [3, 11] for BP analysis. BPQL further allows to use *nested DAGs* to model compound BP activities, and queries may zoom-in, recursively, inside them. Queries may also test data values used in activities. Extending our results to the full-fledged BPQL model is a challenging future research.

The reuse of previously computed query answers (views) has been studied extensively in the literature and proved useful in many contexts such as query optimization, data integration and semantic caching [24, 10, 12]. The closest to our setting are works on tree-shaped (XML) data and general graph-shaped (semi-structured) data.

Relative to XML, our model and query language are a generalization, to DAGs, of the XPath fragment denoted $XP^{\{/,//,[]\}}$, which contains child-axis (/), descendants-axis(//), and branching ([ ]) [1, 21]. While for $XP^{\{/,//,[]\}}$ the problem can be solved in PTIME, we proved it to be NP-hard for arbitrary DAGs and in PTIME only for series-parallel DAGs and queries without projection. As for $XP^{\{/,//,[]\}}$, it is easy to show that extending the language with wildcards (*) makes the problem NP-hard even for series-parallel DAGs. Recall that our algorithm employs query minimization. Here too, while $XP^{\{/,//,[]\}}$ queries can be minimized in PTIME [1], we showed coNP-hardness for general PDAGs, and PTIME complexity for series-parallel ones.

Relative to works on general graph-shaped semi-structured data, our query language is a restriction of *Conjunctive Regular Path Queries (CRPQ)* [16, 13] that enrich the classic conjunctive queries by regular path expressions. The problem in this general setting is EXPSPACE complete [7, 16] (and NP-complete even for classical conjunctive queries.)

Our algorithms determine if the result of previously computed queries suffices to answer a new query. When the answer is negative, it is desirable to determine what additional (minimal) information is needed. We intend to study this problem in future work. Identifying additional cases where processing can be done in PTIME is another challenge.

6. REFERENCES

[1] S. Amer-yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[2] A. Balmin, F. Özcan, K. S. Beyer, R. J. Cochrane, and H. Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB*, 2004.

[3] C. Beeri, A. Eyal, T. Milo, and A. Pilberg. Monitoring business processes with queries. In *VLDB*, 2007.

[4] M. Ben-Ari. Answering DAG queries using partial views. MSc Thesis, Tel Aviv University. http://www.cs.tau.ac.il/∼milo/projects/bpq/papers/DagsFull.pdf.

[5] Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/library/ws-bpel/.

[6] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Answering regular path queries using views. In *ICDE*, 2000.

[7] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, 2000.

[8] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. *J. Comput. Syst. Sci.*, 64(3):443–465, 2002.

[9] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC*, 1987.

[10] S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, 1996.

[11] D. Deutch and T. Milo. Type inference and type checking for queries on execution traces. In *VLDB*, 2008.

[12] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, 1999.

[13] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *DBPL*, 2001.

[14] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.

[15] D. Eppstein. Parallel recognition of series-parallel graphs. *Inf. Comput.*, 98(1):41–55, 1992.

[16] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS*, 1998.

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

[18] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*. Plenum Press, 1972.

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

[20] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.

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

[22] K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *J. ACM*, 29(3):623–641, 1982.

[23] N. Tang, J. Yu, M. T. Ozsu, B. Choi, and K. Wong. Multiple materialized view selection for xpath query rewriting. *ICDE*, 2008.

[24] J. D. Ullman. Information integration using logical views. In *ICDT*, 1997.

[25] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel digraphs. In *STOC*, 1979.