Flexible Queries over Semistructured Data[*]

Yaron Kanza
Institute of Computer Science
The Hebrew University
Jerusalem 91904, Israel
yarok@cs.huji.ac.il

Yehoshua Sagiv
Institute of Computer Science
The Hebrew University
Jerusalem 91904, Israel
sagiv@cs.huji.ac.il

ABSTRACT

Flexible queries facilitate, in a novel way, easy and concise querying of databases that have varying structures. Two different semantics, flexible and semiflexible, are introduced and investigated. The complexity of evaluating queries under the two semantics is analyzed. Query evaluation is polynomial in the size of the query, the database and the result in the following two cases. First, a semiflexible DAG query and a tree database. Second, a flexible tree query and a database that is any graph. Query containment and equivalence are also investigated. For the flexible semantics, query equivalence is always polynomial. For the semiflexible semantics, query equivalence is polynomial for DAG queries and exponential when the queries have cycles. Under the semiflexible and flexible semantics, two databases could be equivalent even when they are not isomorphic. Database equivalence is formally defined and characterized. The complexity of deciding equivalences among databases is analyzed. The implications of database equivalence on query evaluation are explained.

1. INTRODUCTION

Semistructured databases are aimed at representing data that do not conform to a strict schema, due to frequent changes in the structure of the data. The lack of a schema may also be typical of an environment where many users contribute data, in a variety of forms, to the database. The World-Wide Web is such an environment. For an overview of semistructured data see [1, 2, 9].

Traditional query languages and traditional querying methods are not well suited for semistructured data. For one, the semistructured data model is based on a labeled directed graph. More importantly, traditional query languages are not geared to data having no schema at all, or a schema that may change considerably over time or from one data instance to another. An additional consideration is the size of the schema, which could be quite large compared to the size of schemas of structured data.

[*]This work was supported in part by Grant 9481-3-00 from the Israeli Ministry of Science.

When the schema is large and complicated, querying the data could be rather difficult. An initial phase of querying the schema might be needed before the query can be formulated. Even with this additional step, the query could be quite large and hard to phrase, due to the need to cover many structurally similar, but not structurally identical data instances.

The size of the schema is not the only source of difficulties. A case in point is an XML repository of documents with DTDs designed for machine interchange (as well as DTDs that were actually generated by machines). In the eyes of a layperson, such DTDs might not be easy to grasp, regardless of the size.

Traditional query languages are based on the concept of *rigid matchings;* that is, there should be a perfect match between the conditions specified in the query and the data. Rigid matchings are very sensitive to variations in the schema. A query that is posed having a specific schema in mind, is likely to yield only some of the answers or no answer at all, even when only minor changes occur in the structure of the data while the data itself does not change at all.

In recent years, a considerable amount of work has been done on querying semistructured data, in general, and XML data, in particular. Many query languages for semistructured data [3, 5, 6, 11, 20, 22] and for XML [2, 7, 8, 15] have been proposed. In order to deal with the special characteristics of semistructured data, some of these languages use regular expressions. Issues related to the evaluation and optimization of query languages with regular expressions are discussed in [4, 24]. A language that is more expressive than the combination of first-order logic and regular expressions is described in [25]. Additional work has been done on constraining semistructured data [10, 12], describing the structure of the data for query formulation and optimization [17], and extracting information on changes in the data [13].

The query languages cited above are based on rigid matchings, and some of these languages use regular expressions to express possible variations in the structure of the data. But the approach of relying on regular expressions is problematic in at least two aspects. First, the ounce of responsibility is

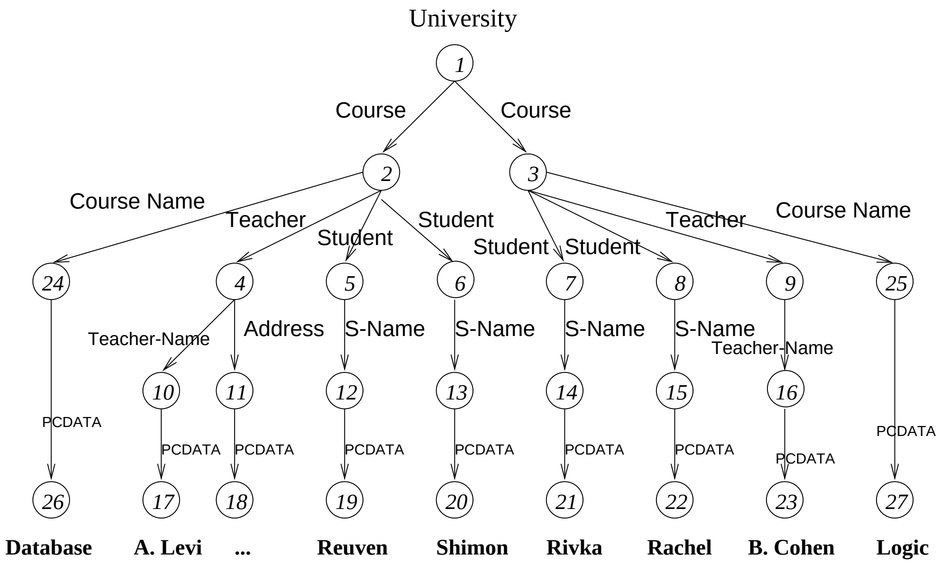


Figure 1: A university database with information on courses, students and teachers.

on the user, who should phrase the regular expressions in a way that will cover the possible variations in the structure of the data and yet will *not* cover too much. Secondly, the tasks of query evaluation and optimization become more complex in the presence of regular expressions.

An alternative approach is that of extracting the *ontology* from the DTDs (or schemas). The ontology consists of the list of names given to the elements and to the attributes of the DTDs, and is much easier to comprehend than the full structural details of the DTDs. In many cases, queries can be phrased simply and succinctly using just the ontology.

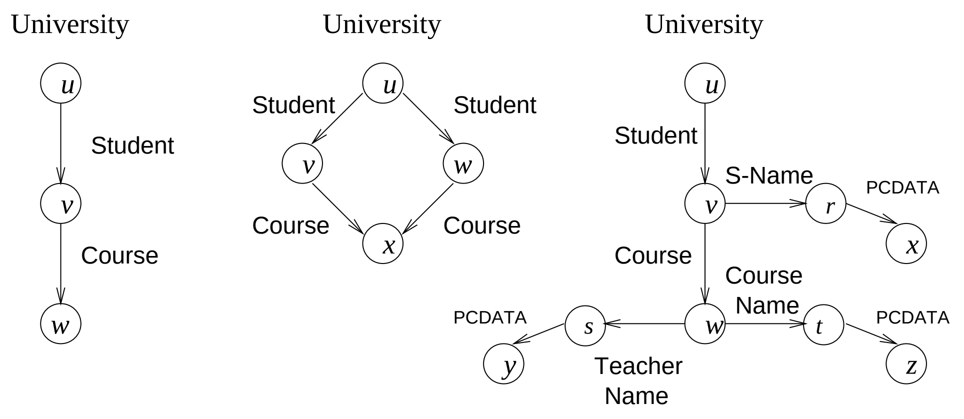
In this paper, we present two semantics that are suitable for ontology-based querying of semistructured data. Similarly to existing query languages, our queries are represented as labeled directed graphs. Thus, preserving the similarity of queries and data. However, instead of a semantics that is based on rigid matchings, we introduce two new semantics that are based on *semiflexible matchings* and *flexible matchings*. We believe that these semantics capture the intended meaning of many common queries. A user only needs to be familiar with the ontology of the database in order to phrase a query, and hence query formulation is more intuitive and simpler compared to query languages that use regular expressions.

In a rigid-matching semantics, the *implicit* relationships between objects are expressed by labeled edges. More complicated relationships have to be constructed explicitly by formulating queries in terms of labeled directed graphs. The semiflexible semantics, in comparison, also uses labeled directed graphs to exhibit relationships between objects, but however, assumes that database objects are implicitly related if they are connected by a directed path (and not just by an edge). Thus, a path π of the query can match a path ϕ of the database if ϕ includes all the labels of π; however, the inclusion need not be contiguous or in the same order as in π. The flexible semantics further extends this idea by applying transitivity to the implicit relationships that hold under the semiflexible semantics.

Section 2 presents the essentials of the data model. Section 3 defines the new semantics and discusses some basic properties of the semiflexible semantics. Section 4 provides complexity results for query evaluation under the semiflexible semantics. Section 5 gives similar results for the flexible semantics. The running times of query-evaluation algorithms are cast as functions of the combined size of the query, the database and the result. This approach of analyzing the combined complexity (rather than the more common data complexity) can discern between cases that are computationally hard, even when the result is small, and cases where the running time is governed merely by the size of the result. Section 6 gives characterizations and complexity results for containment and equivalence of queries. Under the semiflexible and flexible semantics, two databases could be equivalent (even in nontrivial cases) in the sense that they give the same result for all queries. Section 7 characterizes equivalence of two databases. It also characterizes when a given database is equivalent to a tree and gives an algorithm for transforming the database to a tree. This result is important in the context of query evaluation, since queries over tree databases can be evaluated more efficiently, as shown in Section 4.

2. DATA MODEL

Our data model is a simplified version of the Object Exchange Model (OEM) of [26, 3]. The data is represented by labeled directed graphs. Figure 1 depicts a database (which is essentially an XML document). Each node represents an *object* and has an *oid*. The *atomic nodes* (i.e., nodes without outgoing edges) also have values, which are shown below the nodes. For simplicity, we assume that all atoms are of type PCDATA (i.e., *string*). Nodes with outgoing edges represent *complex objects*. A database has a *root* and every node in



(a) Query 1 (b) Query 2 (c) Query 3

Figure 2: Three queries over the university database.

the database is reachable from the root.

Formally, a database D is a 4-tuple $D = \langle O, E^D_{\mathcal{L}}, r_D, \alpha\rangle$, where O is a finite set of objects, $E^D_{\mathcal{L}}$ is a set of labeled edges, r_D is the root and α is a function that maps each atom to a value. The set $\mathcal{L}$ of labels of $E^D_{\mathcal{L}}$ is called the *ontology* of D. For example, the ontology of the database of Figure 1 is the set: {*Course*, *Course Name*, *Teacher*, *Teacher-Name*, *Address*, *Student*, *S-Name*, PCDATA}.

3. QUERIES

Queries are also represented as labeled directed graphs and are similar to those defined in [19]. Nodes of queries represent variables rather than objects, and no value is associated with any node. Formally, a *query* is 3-tuple $Q = \langle V, E^Q_{\mathcal{L}}, r_Q\rangle$, where V is a finite set of *variables*, $E^Q_{\mathcal{L}}$ is a set of labeled edges and r_Q is the root of the query.

An edge from node u to node v that is labeled with l is denoted as ulv. A path from node u to node v that ends with a label l is denoted as $u{*}lv$. This notation is generalized in the natural way. For example, $ulv{*}kw$ denotes a path that starts at u, continues to v through an edge labeled with l, and then continues to w through a path that ends with the label k.

Let $Q = \langle V, E^Q_{\mathcal{L}}, r_Q\rangle$ be a query and $D = \langle O, E^D_{\mathcal{L}}, r_D, \alpha\rangle$ be a database. A *satisfying assignment* (or a *matching*) of Q w.r.t. D is a mapping $\mu \colon V \to O$ that satisfies the *constraints* imposed by Q. There are several types of constraints. One type is a *root constraint* (rc) that requires the root of Q to be mapped to the root of D. A second type is an *edge constraint* (ec) that is written as ulv, where ulv is an edge of Q. The ec ulv is satisfied by the mapping μ if D has an edge labeled with l from $\mu(u)$ to $\mu(v)$. A third type is a *weak edge constraint* (wec) that is written as lv, where variable v of Q has an incoming edge labeled with l. The wec lv is satisfied by μ if $\mu(v)$ has an incoming edge labeled with l. A fourth type is a *quasi edge constraint* (qec) that is written as ulv, where u and v are variables of Q and l is the label of an edge from u to v. The qec ulv is satisfied by the mapping μ if D either has a path from $\mu(u)$ to $\mu(v)$ or vice-versa.

Other types of constraints also include *filtering constraints* (fc) which are essentially selections. Considering fc's is beyond the scope of this paper. Some of the techniques developed in [19] could be used to extend our results also to queries with fc's.

The usual semantics of queries is defined in terms of rigid matchings. A *rigid matching* is one that satisfies the rc and all the ec's of the query Q. The set of all rigid matchings of Q w.r.t. D is denoted as $Mat^r_D(Q)$.

In this paper we introduce two new notions of matchings. The assignment μ is a *semiflexible matching* if it satisfies the rc of Q as well as the following two conditions. (1) For each finite path $v_0 l_1 v_1 l_2 v_2 \cdots l_n v_n$ of Q, where v_0 is the root, the images $\mu(v_i)$ $(0 \le i \le n)$ lie on some path ϕ of D, such that for all i $(1 \le i \le n)$, the edge of ϕ that enters $\mu(v_i)$ is labeled with l_i. Note that $\mu(v_0)$ is the root of D and, hence, is the first node on ϕ, but the rest of the $\mu(v_i)$ are not necessarily arranged on ϕ in the order $\mu(v_1), \ldots, \mu(v_n)$. Moreover, other nodes could be on ϕ between the $\mu(v_i)$. (2) For every strongly connected component C in Q, the image of C is a strongly connected component in D. Recall that a strongly connected component in a graph is a set of nodes in which between each two nodes there is a path. The preservation of the strongly connected components is needed in order to deal with paths that have cycles. We also give next a more formal definition.

Formally, a path $v_0 l_1 v_1 l_2 v_2 \cdots l_n v_n$ of Q satisfies the *SF-Condition* w.r.t. μ if there is a permutation σ of $0, \ldots, n$, such that $\sigma(0) = 0$ and D has a path of the following form: $\mu(v_{\sigma(0)}){*}l_{\sigma(1)}\mu(v_{\sigma(1)}){*}l_{\sigma(2)}\mu(v_{\sigma(2)}) \cdots {*}l_{\sigma(n)}\mu(v_{\sigma(n)})$.

Now, we formally define an assignment μ to be a semiflexible matching if (1) it satisfies the rc of Q, and (2) every finite path $v_0 l_1 v_1 l_2 v_2 \cdots l_n v_n$ of Q, where v_0 is the root, satisfies the SF-Condition w.r.t. μ, and (3) for every strongly connected component $C = \{v_{i_1}, \ldots, v_{i_m}\}$ in Q, the set $\{\mu(v_{i_1}), \ldots, \mu(v_{i_m})\}$ is a strongly connected component in D. We denote the set of all semiflexible matchings of Q w.r.t. D as $Mat^{sf}_D(Q)$.

Matchings	u	v	w	r	s	t	x (student)	y (teacher)	z (course)
	1	5	2	12	10	24	Reuven (19)	A. Levi (17)	Database (26)
	1	6	2	13	10	24	Shimon (20)	A. Levi (17)	Database (26)
	1	7	3	14	16	25	Rivka (21)	B. Cohen (23)	Logic (27)
	1	8	3	15	16	25	Rachel (22)	B. Cohen (23)	Logic (27)

Table 1: The flexible matchings for Query 3 of Figure 2 and the database of Figure 1.

A *flexible matching* satisfies the rc, all the wec's and all the qec's of Q. We denote the set of all flexible matchings of Q w.r.t. D as $Mat^f_D(Q)$.

Intuitively, in both the semiflexible and flexible semantics, we assume that nodes of D that are connected by a path are semantically related. The difference is that in the flexible semantics, we also assume that this relationship is transitive. This difference entails another important difference between the semiflexible and the flexible semantics. In the flexible semantics, cycles in the query are not necessarily mapped to cycles in the database. Semiflexible matchings, however, require strongly connected components in the query to be mapped to strongly connected components in the database.

Proposition 3.1. *For a database D and a query Q, the following holds:*

$$Mat^r_D(Q) \subseteq Mat^{sf}_D(Q) \subseteq Mat^f_D(Q)$$

Some languages for semistructured data use regular expressions as a tool for formulating queries when the structure is not completely known to the user. In principle, the expressive power of regular expressions subsumes the semiflexible and flexible semantics. However, in addition to regular expressions, one may also need unions and joins in order to fully express these semantics. In any case, the semiflexible and flexible semantics facilitate a much more succinct and easier style of writing queries. Moreover, we will show that evaluation and optimization of semiflexible and flexible queries are easier than evaluation and optimization of queries with regular expressions.

3.1 Examples

A university database that holds information on courses, students and teachers is depicted in Figure 1. Suppose that Alice wants to query the data; she is familiar with the ontology of the database, but does not know how the information is organized. We examine a few cases.

Example 3.2. *Alice tries to look in the database for information on courses. She assumes that the information on each student includes the courses that the student is taking. Thus, she formulates Query 1 of Figure 2. There are no rigid matchings of Query 1 w.r.t. the university database. However, when the query is evaluated under the semiflexible semantics, variable w will be assigned to the desired course nodes (Node 2 and Node 3).*

Example 3.3. *Alice wants to find two students who are taking the same course. She formulates Query 2 of Figure 2. Once again, there are no rigid matchings in this case. However, under either the semiflexible or the flexible semantics, Alice will find, for example, Rivka and Rachel as two students who are taking the same course (Logic). This is due to the fact that assigning u, v, w, x to $1, 7, 8, 3$, respectively, is a semiflexible (and, hence, also a flexible) matching of the query w.r.t. the university database.*

Example 3.4. *When Query 3 of Figure 2 is posed to the university database as either a rigid or a semiflexible query, there are no matchings of the query w.r.t. the database. Nevertheless, under the flexible semantics, we get the assignments that are shown in Table 1. Atomic nodes where replaced with their value.*

3.2 More on Semiflexible Matchings

It is not necessarily easy to check whether an assignment to the variables of a query Q is a semiflexible matching. The reason is that all paths of Q have to be considered. If Q is a dag (directed acyclic graph), the number of paths could be exponential, and if Q has a cycle, then there are infinitely many paths. In this section, we show that if the given database D is a dag, then checking whether an assignment is a semiflexible matching could be done in polynomial time in the size of Q and D. We also show that in the case of a cyclic query, it is sufficient to consider only simple paths and cycles in order to check whether a given assignment is a semiflexible matching. Note that as a matter of terminology, a cyclic query (database) has at least one cycle. Thus, the class of cyclic queries (databases) does not include any dag query (database). A tree query (database), however, is a special case of a dag query (database).

Lemma 3.5. *Suppose that Q is a dag query and D is a dag database. The assignment $\mu\colon V \to O$ is a semiflexible matching of Q w.r.t. D if and only if the following condition is satisfied.*

- *For all labels l, all nodes v and the root r of Q, if Q has a path of the form $r{*}lv$, then D has a path of the form $\mu(r){*}l\mu(v)$; and*
- *For all labels l and k, and all nodes u, v and w of Q, if Q has a path of the form $ulv{*}kw$, then D either has a path of the form $\mu(v){*}k\mu(w)$ or a path of the form $\mu(w){*}l\mu(v)$.*

Proof. It is easy to see that if μ is a semiflexible matching, then it satisfies the condition of the lemma. For the other direction, suppose that the condition of the lemma holds and consider a path $\pi = v_0 l_1 v_1 l_2 v_2 \cdots l_n v_n$ of Q. Since D is a dag, its nodes can be sorted topologically. Let σ be a permutation of $0, \ldots, n$, such that $\sigma(0) = 0$ and the sequence of objects $\mu(v_{\sigma(0)}), \mu(v_{\sigma(1)}), \ldots, \mu(v_{\sigma(n)})$ conforms to the topological ordering on D. Note that $v_{\sigma(0)}$ and $\mu(v_{\sigma(0)})$ are the roots of Q and D, respectively. Moreover, if both $\mu(v_{\sigma(i-1)})$ and $\mu(v_{\sigma(i)})$ $(1 \leq i \leq n)$ lie on the same path of D, then the path must be from $\mu(v_{\sigma(i-1)})$ to $\mu(v_{\sigma(i)})$.

We claim that D has a path of the form

$$\mu(v_{\sigma(0)})*l_{\sigma(1)}\mu(v_{\sigma(1)})*l_{\sigma(2)}\mu(v_{\sigma(2)})\cdots *l_{\sigma(n)}\mu(v_{\sigma(n)}).$$

To prove it, we show that for all $1 \leq i \leq n$, there is a path of the form $\mu(v_{\sigma(i-1)})*l_{\sigma(i)}\mu(v_{\sigma(i)})$ in D. For $i = 1$, node $v_{\sigma(0)}$ is the root of Q, and so, Q has a path of the form $v_{\sigma(0)}*l_{\sigma(1)}v_{\sigma(1)}$. Thus, by the first part of the condition of the lemma, D has a path of the form $\mu(v_{\sigma(0)})*l_{\sigma(1)}\mu(v_{\sigma(1)})$.

For $i > 1$, there are two cases to be considered, depending on whether $v_{\sigma(i-1)}$ appears before or after $v_{\sigma(i)}$ on the path π of Q. If $v_{\sigma(i-1)}$ appears first, then Q has a path of the form $v_{\sigma(i-1)}*l_{\sigma(i)}v_{\sigma(i)}$. Thus, according to the second part of the condition of the lemma, D must have a path of the form $\mu(v_{\sigma(i-1)})*l_{\sigma(i)}\mu(v_{\sigma(i)})$. Alternatively, if $v_{\sigma(i)}$ appears first, then there is a j $(0 \leq j \leq n-2)$, such that Q has a path of the form $v_j l_{\sigma(i)} v_{\sigma(i)} * l_{\sigma(i-1)} v_{\sigma(i-1)}$. Thus, according to the second part of the condition of the lemma, D must have a path of the form $\mu(v_{\sigma(i-1)})*l_{\sigma(i)}\mu(v_{\sigma(i)})$. $\square$

Next, we will consider cyclic queries. Since such queries have infinitely many paths, a naive test based on the definition of a semiflexible matching does not terminate. The following lemma alleviates this difficulty.

The path $v_0 l_1 v_1 l_2 v_2 \cdots l_n v_n$ (where v_0 is not necessarily the root) is a *simple path* if all the v_i are distinct. It is a *cycle* if $v_0 = v_n$, and it is a *simple cycle* if $v_0 = v_n$ and $v_1, v_2, \ldots, v_n$ are distinct.

LEMMA 3.6. *Let Q be a cyclic query and D be a cyclic database. The assignment μ is a semiflexible matching of Q w.r.t. D if and only if the following conditions are satisfied. (1) The assignment μ satisfies the rc of Q. (2) All simple paths of Q that start at the root satisfy the SF-Condition w.r.t. μ. (3) For all simple cycles $u_0 l_1 u_1 \cdots l_k u_k$ of Q (where u_0 is not necessarily the root), D has a cycle (which is not necessarily simple) of the form $\mu(u_0)*l_1\mu(u_1)\cdots *l_k\mu(u_k)$.*

PROOF. Obviously, if μ is a semiflexible matching, then Conditions (1) and (2) hold. Next, we will show the necessity of Condition (3).

Let μ be a semiflexible matching of Q w.r.t. D, and let $C = u_0 l_1 u_1 \cdots l_k u_k$, where $u_0 = u_k$, be a simple cycle of Q.

The set $\{\mu(u_0), \ldots, \mu(u_k)\}$ is contained in a strongly connected component of D, because μ is a semiflexible matching and $\{u_0, \ldots, u_k\}$ is contained in a strongly connected component of Q. Therefore, there is a path from $\mu(u_i)$ to $\mu(u_{i+1})$ $(0 \leq i \leq k-1)$.

The second observation is that D has a path of the form $\mu(u_i)*l_i\mu(u_i)$ $(1 \leq i \leq k)$. In proof, for each node u_i $(1 \leq i \leq k)$ in C, there is a path that starts at the root of Q, continues to a node in C and then goes to u_i through the edge $u_{i-1}l_iu_i$. This path can be extended by going around the cycle C from u_i back to itself. Thus, Q has a path of the form $r_Q*l_iu_i*l_iu_i$, where r_Q is the root of Q. This path satisfies the SF-Condition, since μ is semiflexible matching. Therefore, D must have a path of the form $\mu(u_i)*l_i\mu(u_i)$ $(1 \leq i \leq k)$.

By combining the path from $\mu(u_{i-1})$ to $\mu(u_i)$ with the path $\mu(u_i)*l_i\mu(u_i)$ $(1 \leq i \leq k)$, it follows that D has a path of the form $\mu(u_{i-1})*l_i\mu(u_i)$ $(1 \leq i \leq k)$. Therefore, D also has a path of the form $C_D = \mu(u_0)*l_1\mu(u_1)\cdots *l_k\mu(u_k)$. Since $u_0 = u_k$, it follows that $\mu(u_0) = \mu(u_k)$ and, thus, C_D is a cycle in D. Thus, we have shown that Condition (3) is satisfied.

For the other direction, suppose that μ is a mapping of the variables of Q to objects of D, such that Conditions (1)–(3) are satisfied. By Condition (3), μ maps strongly connected components of Q to strongly connected components of D. Next, we will show that paths of Q satisfy the SF-Condition w.r.t. μ.

Let $\pi = v_0 l_1 v_1 \cdots l_n v_n$ be a path of Q, where v_0 is the root of Q. We will show by induction on n that π satisfies the SF-Condition w.r.t. μ.

For $n = 1$, π is either a simple path (if $v_0 \neq v_1$) or a simple cycle (if $v_0 = v_1$). In the first case, π satisfies the SF-Condition by Condition (2), and in the second case — by Condition (3).

Next, suppose that the claim holds for paths of length less that n, where $n > 1$. We have to show that the claim holds for the path π.

Once again, if π is a simple path or a simple cycle, then either Condition (2) or Condition (3) implies that π satisfies the SF-Condition.

If π is neither a simple path nor a simple cycle, then it must have variables v_i and v_k $(i < k)$, such that the path $\pi_c = v_i l_{i+1} v_{i+1} \cdots l_k v_k$ is a simple cycle. By Condition (3), D has a cycle of the form $\phi_c = \mu(v_i)*l_{i+1}\mu(v_{i+1})\cdots *l_k\mu(v_k)$.

By the induction hypothesis, the path

$$\pi_p = v_0 l_1 v_1 \cdots l_i v_i l_{k+1} v_{k+1} \cdots l_n v_n$$

satisfies the SF-Condition. Thus, there is a permutation σ_p of $0, 1, \ldots, i, k+1, \ldots, n$, such that $\sigma_p(0) = 0$ and D has a path of the following form.

$$\begin{aligned}\phi_p \quad = \quad & \mu(v_{\sigma_p(0)})*l_{\sigma_p(1)}\mu(v_{\sigma_p(1)})\cdots \\ & *l_{\sigma_p(i)}\mu(v_{\sigma_p(i)})*l_{\sigma_p(k+1)}\mu(v_{\sigma_p(k+1)})\cdots \\ & *l_{\sigma_p(n)}\mu(v_{\sigma_p(n)})\end{aligned}$$

We will now show that the cycle ϕ_c and the path ϕ_p can be combined into a single path ϕ that shows that π satisfies the SF-Condition. Intuitively, this is done by inserting the cycle ϕ_c into the path ϕ_p starting at an occurrence of v_i in ϕ_p. To do that, we need to define a permutation σ of $0, 1, \ldots, n$ that

Database / Query	path query	tree query	dag query	cyclic query
path database	PTIME	PTIME	PTIME	there are no semiflexible matchings
tree database	PTIME	PTIME	PTIME	there are no semiflexible matchings
dag database	PTIME	PTIME	PTIME	there are no semiflexible matchings
cyclic database	PTIME	PTIME	coNP	coNP

Table 2: The complexity of checking whether μ is a semiflexible matching.

combines the effect of the permutation σ_p with the effect of the identity permutation of $i+1,\dots,k$, which relates the simple cycle π_c to the cycle ϕ_c.

The permutation σ_p is defined over $0,1,\dots,i,k+1,\dots,n$, i.e., it has a gap between $i+1$ and k. However, this gap is not necessarily the place to put the cycle ϕ_c. Therefore, we first have to shift some positions in order to create the gap in the right place.

Formally, there is a j, such that $\sigma_p(j)=i$. Note that j is the position of $\mu(v_i)$ in the path ϕ_p. There are two cases for creating the gap, starting at position $j+1$, and for each case σ is defined differently.

If $j+1 \le i$, then the gap has to be created by shifting right; that is, for $j+1 \le m \le i$, we define $\sigma(m+k-i)=\sigma_p(m)$.

If $k+1 \le j$, then the gap has to be created by shifting left; that is, for $k+1 \le m \le j$, we define $\sigma(m-(k-i))=\sigma_p(m)$.

Now, we fill the gap with the cycle ϕ_c by defining $\sigma(j+m)=i+m$ for $1 \le m \le k-i$. Finally, we complete the definition of σ by defining $\sigma(m)=\sigma_p(m)$, for all other values of m.

It thus follows that D has a path ϕ of the form

$$\mu(v_{\sigma(0)}) * l_{\sigma(1)} \mu(v_{\sigma(1)}) * l_{\sigma(2)} \mu(v_{\sigma(2)}) \cdots * l_{\sigma(n)} \mu(v_{\sigma(n)})$$

and this path shows that π satisfies the SF-Condition. □

It should be noted that a query with cycles cannot be satisfied by a database without cycles, under either the rigid or semiflexible semantics. However, for the flexible semantics, this is not necessarily the case.

COROLLARY 3.7. *Deciding if a given assignment μ of a cyclic query w.r.t. a cyclic database is a semiflexible matching is in* coNP.

PROOF. We will describe a nondeterministic polynomial-time algorithm for testing whether μ is not a semiflexible matching. If μ does not map the root of the query to the root of the database, then the algorithm returns yes. The algorithm guesses a simple path $v_0 l_1 v_1 l_2 v_2 \cdots l_n v_n$ of Q and returns yes if this path does not satisfy the SF-Condition (in the full paper, we will describe how to test that a given path satisfy the SF-Condition in polynomial time). The algorithm also guesses a set of nodes $\{v_{i_1},\dots,v_{i_m}\}$ and returns yes if this set is strongly connected in Q, but the set $\{\mu(v_{i_1}),\dots,\mu(v_{i_m})\}$ is not strongly connected in D. □

In summary, if the database is a dag, then Lemma 3.5 implies that testing whether μ is a semiflexible matching has a running time that is polynomial in the size of the query and the database. If the database is cyclic and the query is a tree, the test can still be done in polynomial time, since a tree query has only a linear number of paths from the root to a leaf. However, for a cyclic database and a query that is either a dag or cyclic, we only know that the test is in coNP. Table 2 gives the complexity results, in terms of the size of the query and the database, for the various cases.

4. QUERY EVALUATION IN SEMIFLEXIBLE SEMANTICS

Query evaluation has a polynomial-time data complexity under either the semiflexible or flexible semantics. If both the query and the data are considered as input, then the complexity is certainly exponential, since the size of the result could be exponential in the size of the query and the data. A better approach is to analyze the *combined complexity* that is measured in terms of the size of the query, the database and the result. Recall that in the relational case, merely checking whether a join of n relations is not empty is NP-Complete [21]; hence, the combined complexity is exponential. However, for the important case of acyclic joins, the combined complexity is polynomial [28]. In this section, we will discuss the combined complexity of query evaluation under the semiflexible semantics.

4.1 Path Queries

In this section we consider *path queries*. A path query has the form $v_0 l_1 v_1 l_2 v_2 \dots l_n v_n$, where v_0 is the root and all the v_i are distinct. First, we will discuss the case of evaluating a path query over a database that is also a path. Note that even in this case, the result could be exponential in the size of the query and the database (provided that some labels are repeated along paths of the database). However, the combined complexity is polynomial.

For a query node v, the *correspondence set* of v, denoted C_v, is the set of all database objects o, such that o satisfies some wec of v (i.e., there is a label l, such that both o and v have incoming edges labeled with l). The correspondence set of the root r_Q of Q consists of the root r_D of the database.

PROPOSITION 4.1. *Consider a path query Q of the form $v_0 l_1 v_1 l_2 v_2 \dots l_n v_n$, and a path database D over a set of objects O. The set $Mat_D^{sf}(Q)$ of the semiflexible matchings of Q w.r.t. D is $\{\mu : V \to O \mid \mu(v_i) \in C_{v_i}$ for $0 \le i \le n$, and $\mu(v_i) \ne \mu(v_j)$ for $i \ne j\}$. Computing $Mat_D^{sf}(Q)$ has a linear-time combined complexity.*

PROOF. Follows from the definition of the SF-Condition. In particular, note that this condition implies that if the

Database / Query	path query	tree query	dag query	cyclic query
path database	PTIME	PTIME	PTIME	the result is always empty
tree database	PTIME	PTIME	PTIME	the result is always empty
dag database	NP-Complete	NP-Complete	NP-Complete	the result is always empty
cyclic database	NP-Complete	NP-Complete	NP-Hard (in Σ_2^{P})	NP-Hard (in Σ_2^{P})

Table 3: The complexity of checking nonemptiness under the semiflexible semantics.

database has no cycles, then μ must map variables that are connected by a path to distinct objects. □

Next, we consider the case of a tree database D. For each path π of D from the root to a leaf, we compute the result of the path query Q w.r.t. π, and then take the union of all these results. Thus, we get the following.

THEOREM 4.2. *Query evaluation for a path query Q and a tree database D has a combined complexity of $O(|Q| \cdot |D| + |M|)$ time, where $|Q|$, $|D|$ and $|M|$ are the sizes of the query, the database and the result, respectively.*

The next theorem shows that query evaluation is not likely to have a polynomial-time combined complexity when the database is a dag.

THEOREM 4.3. *Given a path query Q and a dag database D, deciding whether $Mat_D^{sf}(Q)$ is not empty is NP-Complete (when the input is the query and the data).*

The NP-Hardness is shown by a reduction of 3SAT. The problem is in NP, since one can guess an assignment μ and verify that it is a semiflexible matching in polynomial time (follows from Lemma 3.5).

Note that for a path query Q and a database D that may have cycles, query evaluation has a combined complexity of $O(|D|^{|Q|+1} + |Q|)$ time.

4.2 Tree, DAG, and Cyclic Queries

In this section, we show that evaluation of tree and dag queries is not significantly harder than evaluation of path queries, except for the case of a cyclic database.

Note that checking nonemptiness is NP-Complete when both the query and the database are dag's, since NP-Hardness follows from Theorem 4.3 and membership in NP follows from Lemma 3.5. The result of Theorem 4.2 can be generalized to dag queries and tree databases, as stated by the next theorem that gives the main result about evaluation of semiflexible queries. The algorithm mentioned in the theorem is rather intricate and its description cannot be given here due to space limitations.

THEOREM 4.4. *Let Q be a dag query and let D be a tree database. There is an algorithm that computes the set of semiflexible matchings $Mat_D^{sf}(Q)$ in $O(|Q|^4 \cdot |D|^2 + |M|^2 \cdot |Q|^3 \cdot |D|)$ time, where $|Q|$, $|D|$ and $|M|$ are the sizes of the query, the database and the result, respectively.*

For the case of cyclic databases, query evaluation is NP-Complete provided that there is a polynomial-time test for verifying that an assignment is a semiflexible matching, i.e., when the query is either a path or a tree. For dag queries, however, we only know that verifying that an assignment is a semiflexible matching is in coNP. Therefore, checking nonemptiness is in Σ_2^{P} (and is still NP-Hard).

Now consider cyclic queries. Since the result of evaluating a cyclic query over a dag database is always empty, NP-Hardness does not follow from Theorem 4.3. However, a reduction similar to that of Theorem 4.3 shows that nonemptiness of a cyclic query over a cyclic database is NP-Hard (and is in Σ_2^{P}).

The complexity results for the various cases are shown in Table 3. Note that when testing emptiness is polynomial-time (in the size of the query and the data), query-evaluation has a polynomial-time combined complexity. When testing emptiness is not polynomial, query-evaluation cannot have a polynomial-time combined complexity either.

5. EVALUATING FLEXIBLE QUERIES

First, we will show that query evaluation under the flexible semantics can be reduced to query evaluation under the rigid semantics.

Let $D = \langle O, E, r_D, \alpha \rangle$ be a database over the set of nodes O. The *reachability graph* of D is a rooted labeled directed graph, denoted $RG(D) = \langle O, E_R, r_D, \alpha \rangle$, that is obtained from D by adding edges as follows. If object o' of D has an incoming edge labeled with l and there is either a path from o' to another object o or vice-versa, then E_R has an edge labeled with l from o to o'.

THEOREM 5.1. *Let Q be a query and let D be a database with a reachability graph $RG(D)$. The set of flexible matchings of Q w.r.t. D is equal to the set of rigid matchings of Q w.r.t. $RG(D)$.*

Evaluating all the rigid matching of Q w.r.t. $RG(D)$ can be done as follows. For each edge $e = ulv$ of Q, we create a binary *edge relation* r_e that has the attributes u and v, and the following set of tuples: $\{(o, o') \mid RG(D)$ has an edge $olo'\}$. In other words, r_e contains all edges of $RG(D)$ that have the same label as e. The join of all the edge relations yields all the rigid matchings of $RG(D)$. When the join is acyclic, Yannakakis's algorithm [28] can be applied and, hence, we get the following corollary.

COROLLARY 5.2. *Query evaluation under the flexible semantics has a polynomial-time combined complexity when the query Q is a tree and the database D is any graph.*

The next theorem shows that if Q is not a tree, then a query-evaluation algorithm with a polynomial-time combined complexity is not likely to exist.

THEOREM 5.3. *Given a dag query Q and a database D, deciding whether the flexible semantics yields a nonempty result is* NP*-complete.*

NP-hardness follows from a reduction of 3SAT. Membership in NP follows because a flexible matching can be guessed and verified in polynomial time.

6. QUERY CONTAINMENT AND QUERY EQUIVALENCE

In the case of relational conjunctive queries, the final step of evaluation is a projection of the matchings onto the distinguished variables. Consequently, containment is defined in terms of those projections. In the case of queries over semistructured data, the final step is a construction of the result from the matchings. A discussion of this step, however, is beyond the scope of the paper. Thus, in this paper, the semantics of queries is defined in terms of matchings over all the variables, and containment (equivalence) is defined as containment (equality) of the corresponding sets of matchings.

Formally, given two queries Q_1 and Q_2 over the same set of node variables, we say that Q_1 is *contained* in Q_2 under the semantics s, denoted $Q_1 \subseteq_s Q_2$, if for all database D, $Mat^s_D(Q_1) \subseteq Mat^s_D(Q_2)$. The queries Q_1 and Q_2 are *equivalent* if for all database D, $Mat^s_D(Q_1) = Mat^s_D(Q_2)$.

Deciding equivalence and containment of queries is useful for optimization techniques. The next theorem provides a characterization of containment for semiflexible queries.

THEOREM 6.1. *Let Q_1 and Q_2 be queries over the same set of variables V. $Q_1 \subseteq_{sf} Q_2$ if and only if the identity mapping μ over V is a semiflexible matching of Q_2 w.r.t. Q_1.*

COROLLARY 6.2. *Deciding if $Q_1 \subseteq_{sf} Q_2$ is in coNP when Q_1 is a cyclic graph and Q_2 is either a dag or a cyclic graph. In all other cases, it is in polynomial time.*

A characterization of containment for flexible queries is given in the next theorem.

THEOREM 6.3. *Let Q_1 and Q_2 be queries over the same set of variables. $Q_1 \subseteq_f Q_2$ if and only if the following two conditions hold:*

1. *For each wec lv in Q_2, there is a wec lv in Q_1.*
2. *For each ec ulv in Q_2, where $u \neq r_{Q_2}$ (r_{Q_2} is the root of Q_2), the query Q_1 contains either the ec $ul'v$ or the ec $vl'u$ for some label l'.*

To decide containment of Q_1 in Q_2 we just need to check that every wec (ec) in Q_1 has a suitable wec (ec) in Q_2.

COROLLARY 6.4. *Deciding if $Q_1 \subseteq_f Q_2$ is in $O(|Q_1|\cdot|Q_2|)$ time.*

If we sort the ec's and wec's of Q_2 deciding containment can be done in $O(|Q_2| \cdot \log|Q_2| + |Q_1| \cdot \log|Q_2|)$.

7. DATABASE EQUIVALENCE

Given two databases D and D' over the same set of objects O, we say that D and D' are *equivalent* under the semantics s if for every query Q, the set of s-matchings of Q w.r.t. D is equal to the set of s-matchings of Q w.r.t. D'. Under the classical (i.e., rigid) semantics, two databases are equivalent if and only if they are isomorphic (i.e., have the same root and the same set of labeled edges). In the case of the semiflexible and flexible semantics, however, two databases can be equivalent even if they are not isomorphic.

One reason for investigating database equivalence is that in some cases it is more efficient to evaluate queries over databases that have a certain form than over databases that have a different form. For example, we showed that it is more efficient to evaluate queries over a tree database than over a dag database. Thus, it is important to be able to characterize equivalence of databases, and to be able to transform a database of a given form (e.g., dag) to an equivalent database that has a different form (e.g., tree).

Let D and D' be two databases that have the same set of objects and the same root. We say that a path $\phi = o_0 l_1 o_1 l_2 o_2 \cdots l_n o_n$ of D, where o_0 is the root, is *included* in a path ϕ' of D' if there is a permutation σ of $1, \ldots, n$, such that ϕ' has the form $o_0{*}l_{\sigma(1)}o_{\sigma(1)}{*}l_{\sigma(2)}o_{\sigma(2)} \cdots {*}l_{\sigma(n)}o_{\sigma(n)}$.

We say that there is a *semiflexible path inclusion* of D in D' if for every path ϕ in D that starts at the root, there is a path ϕ' in D', such that ϕ' includes ϕ.

THEOREM 7.1. *Consider two databases D and D' over the same set of objects and with the same root. D and D' are equivalent under the semiflexible semantics if and only if (1) there is a semiflexible path inclusion of D in D' and vice-versa, and (2) each strongly connected component in D is a strongly connected component in D' and vice-versa.*

Conditions (1) and (2) of the above theorem are essentially equivalent to the condition that the identity mapping (of the objects of D to the objects of D') is a semiflexible matching. Therefore, we get the following corollary.

COROLLARY 7.2. *Deciding equivalence of databases, under the semiflexible semantics, is in polynomial time if the databases are dags and is in* coNP *if the databases have cycles.*

For the flexible semantics, we have the following theorem.

```
Algorithm TransformingDatabaseToTree($D$);
Input a tree-transformable database $D$;
Output a tree database $T$ that is equivalent to $D$ under the semiflexible semantics;
let $\mathcal{H}$ be the path hypergraph of $D$;
let $\mathcal{B} = BD(\mathcal{H})$ be the Bachman diagram of $\mathcal{H}$;
let $S_0$ be the least node of $\mathcal{B}$ ($S_0$ is the intersection of all nodes of $\mathcal{B}$);
(* by chosing $S_0$ to be the root, $\mathcal{B}$ can be viewed as a tree *)
$\mathcal{S} \leftarrow \{S_0\}$;
create from the objects of $S_0$ a path $P_0$ with $r_D$ (the root of $D$) as the first node;
while $\mathcal{S}$ is not empty
    remove some node $S_j$ from $\mathcal{S}$;
    add to $\mathcal{S}$ all the children of $S_j$ in $\mathcal{B}$;
    let $S_i$ be the parent of $S_j$ in $\mathcal{B}$;
    create a simple path $P_j$ from the objects of $S_j$ that are not in $S_i$;
    (* Since $S_i$ is the parent of $S_j$, a path $P_i$ has already been created for $S_i$ *)
    add an edge from the last node of $P_i$ to the first node of $P_j$;
let $T$ be the database that was produced in the previous steps;
label $T$ by attaching to each edge the label that corresponds to its target in $D$;
return $T$;
```

Figure 3: Creating a tree database T from a database D.

THEOREM 7.3. *Consider two databases D and D' over the same set of objects O and with the same root. D and D' are equivalent under the flexible semantics if and only if their reachability graphs are isomorphic, i.e., have the same set of labeled edges.*

It is not necessary to solve the general case of graph isomorphism in order to decide equivalence of databases under the flexible semantics. Instead, it is sufficient to check that the identity mapping is an isomorphism. Thus, deciding whether two databases D and D' are equivalent under the flexible semantics is in $O(|D|^2 \log |D| + |D'|^2 \log |D'|)$.

7.1 Removing Redundancies

One application of database equivalence is the removal of *redundant parts* from databases. A part of a database is redundant if removing it from the database has no effect on the result of query evaluation.

For example, given a database D, we say that an edge $o_1 l o_2$ in D is *redundant* w.r.t. the semiflexible semantics if D has a path of the form $o_1{*}l o_2$ that does not include the edge $o_1 l o_2$. If a redundant edge is removed, the result is a database that is equivalent to the original one under the semiflexible semantics.

7.2 Testing for Equivalence to a Tree

In addition to testing database equivalence, it is also possible to transform a database D to a tree, provided that D is indeed equivalent to some tree database. Note that under the semiflexible semantics, a cyclic database cannot be equivalent to a tree database. However, there are dag databases that are equivalent to tree databases. Under the flexible semantics, even a cyclic database could be equivalent to a tree database.

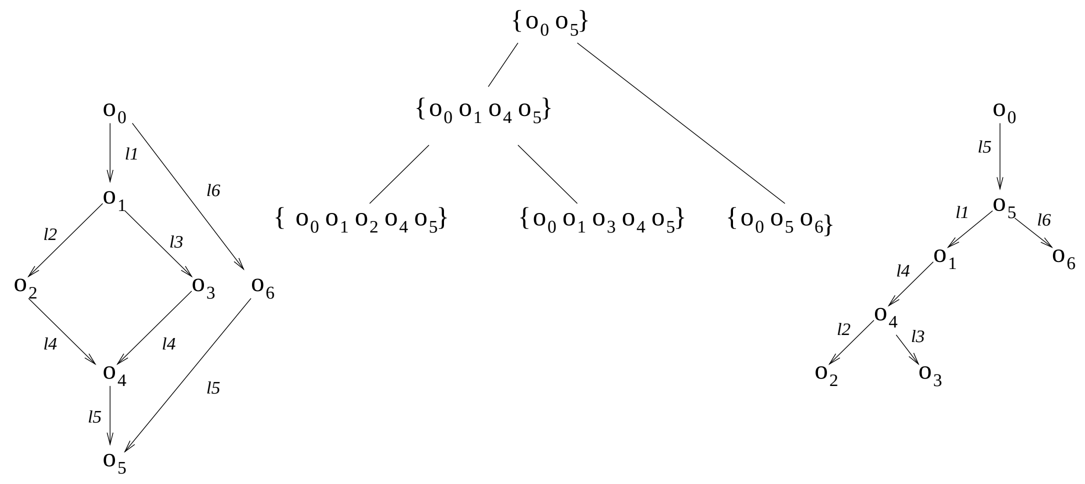
Transforming a database to an equivalent tree is important for several reasons. First, evaluation of queries is more efficient when the database is a tree, as was shown earlier. Secondly, in a graphical user interface, presenting trees is easier than presenting dags. Thirdly, storing data as a document (e.g., in XML) is easier when the data is a tree, since references and identifiers need not be used.

Given a database D, we say that D is *tree transformable* under the semantics s if there exists a tree database T that is equivalent to D under the semantics s. In this section, we characterize tree transformable databases under the semiflexible and the flexible semantics, and give algorithms for the transformations.

We will start with transformation to a tree under the semiflexible semantics. Given a database D that has no redundant edges, the following are necessary conditions for D to be tree transformable under the semiflexible semantics.

The first necessary condition is that D does not include a node that has incoming edges with different labels. The reason for this condition is that in a tree database only one edge enters each node (except for the root that has no incoming edge) and this edge holds exactly one label. The second necessary condition is that D is acyclic. The reason for the second condition is that in a finite tree all paths are finite, while a cyclic graph contains an infinite path. The third necessary condition is that the number of paths in D is not greater than the number of nodes in D. Recall that in a tree, the number of paths, from the root to a leaf, never exceeds the number of nodes.

In order to have conditions that are both necessary and sufficient, we need additional definitions. Given a database D, the *path hypergraph* of D is the hypergraph $\mathcal{H}$ that has the same nodes as D and has as hyperedges all sets of nodes S,



(1) A given database D. (2) The Bacman diagram of D. (3) The equivalent tree database.

Figure 4: Transformation to a tree under the semiflexible semantics.

such that S is the set of all nodes on some path of D from the root to a leaf. Let $\mathcal{E}$ be the set that contains all the hyperedges of $\mathcal{H}$ and all the nonempty intersections of two or more hyperedges of $\mathcal{H}$. The *Bachman diagram* of $\mathcal{H}$, denoted $BD(\mathcal{H})$, is the following graph. For each element of $\mathcal{E}$, there is a node in $BD(\mathcal{H})$. There is an edge between E_1 and E_2 in $BD(\mathcal{H})$ if (1) $E_1 \subseteq E_2$, and (2) there is no element E' in $\mathcal{E}$, such that $E_1 \subseteq E' \subseteq E_2$ and E' is different from both E_1 and E_2. We say that $BD(\mathcal{H})$ is acyclic if it contains no cycle (as an undirected graph). A discussion of acyclic Bachman diagrams and their usage in the characterization of γ-acyclic hypergraphs is given in [16]. In [27], Bachman diagrams are used to characterize full disjunctions.

THEOREM 7.4. *Let D be a database with no redundant edges and with the path hypergraph $\mathcal{H}$. The database D is tree transformable under the semiflexible semantics if and only if the following three conditions hold. (1) D is acyclic; (2) There is no node in D that has two different incoming labels; and (3) $BD(\mathcal{H})$, the Bachman diagram of $\mathcal{H}$, is acyclic. Deciding if D is tree transformable under the semiflexible semantics is in polynomial runtime in the size of D.*

Figure 3 gives a polynomial-time algorithm that actually transforms a given database D to a tree, provided that D is tree transformable. The algorithm starts by creating the path hypergraph $\mathcal{H}$ of D. Note that in a tree transformable database, the number of paths from the root to the leaves cannot exceeds the number of nodes, and thus, the size of the path hypergraph is polynomial in the size of the database.

The second step of the algorithm is the creation of the Bachman diagram $\mathcal{B} = BD(\mathcal{H})$. This can be done in polynomial time in the size of $\mathcal{H}$. Recall that the nodes of $\mathcal{B}$ are sets of objects of the database D. The intersection of all the nodes of $\mathcal{B}$ always contains the root of D. Thus, there is a *least node* of $\mathcal{B}$ that is contained in all the other nodes. In the algorithm, we view $\mathcal{B}$ as a tree whose root is the least node.

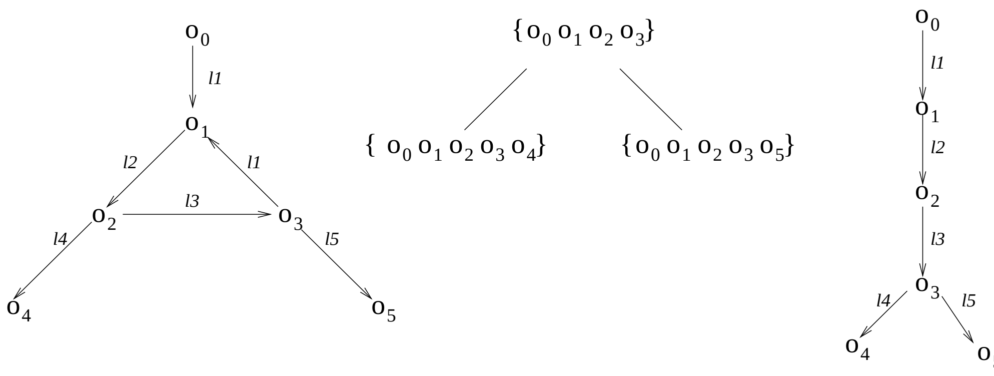
The algorithm creates a tree database T by visiting the nodes of $\mathcal{B}$ in a topological order, starting with the least node. Initially, T is empty. For each visited node E of $\mathcal{B}$, the algorithm adds to T the objects of $E - E'$, where E' is the parent of E. The newly added objects are connected by new edges to form a simple path. The order of the objects along this path is not important, except for the root of D that must be the first object on the path created for the least node of $\mathcal{B}$. Thus, each node E of $\mathcal{B}$ is associated with some path in T, and that path has a first object and a last object. A new edge is also added from the last object of the parent of E to the first object of E.

The final step of the algorithm is to label the edges of T. Each edge is labeled with the unique label associated with the object that it enters; that is, if an edge enters an object o, then in the original database D, all the edges that enter o are labeled with the same label l, and l is also the label of the single edge that enters object o in the tree database T. Note that the root has no incoming edges in T. It is easy to see that the returned graph is indeed a tree.

We now consider transformation to a tree under the flexible semantics. A necessary condition for the existence of a transformation under the flexible semantics is as follows. The database should not have a node o, such that two edges with different labels enter o. Moreover, there should not be any edge that enters the root.

Next, we give some additional definitions. The *augmented reachability graph* of a database D is the reachability graph of D augmented with an unlabeled edge from each node to the root. An edge between two nodes, in the augmented reachability graph, reflects the existence of a path in D between these nodes. Since each node in the database is reachable from the root, there is an edge in the augmented reachability graph from each node to the root.

The *maximal-clique hypergraph* of a database D is the hypergraph that has the same nodes as D and has, as hyperedges,



(1) A given database D. (2) The Bacman diagram. (3) The equivalent tree database.

Figure 5: Transformation to a tree under the flexible semantics.

the maximal cliques in the augmented reachability graph of D. Note that in a directed graph, a clique is a set of nodes, such that every two nodes are connected by edges in both directions. A clique is maximal if it is not contained in any other clique.

Theorem 7.5. *Consider a database D. Let the hypergraph $\mathcal{H}$ be the maximal-clique hypergraph that is created from the augmented reachability graph of D. The database D is tree transformable under the flexible semantics if and only if the following three conditions hold. (1) There is no edge in D that enters the root; (2) There is no node in D that has two different incoming labels; and (3) The Bachman diagram $BD(\mathcal{H})$ of $\mathcal{H}$ is acyclic.*

Given that a database D is tree transformable under the flexible semantics, the creation of the equivalent tree T is the same as in the case of the semiflexible semantics, except for the following. In the algorithm of the transformation (Figure 3), the hypergraph $\mathcal{H}$ that is used is the maximal-clique hypergraph of the augmented reachability graph of D (instead of the path hypergraph of D). The creation of the maximal-clique hypergraph is not in polynomial time. Thus, the transformation to a tree under the flexible semantics is not in polynomial runtime.

An example of a transformation is depicted in Figure 5. Note that Figure 5 does not show the augmented reachability graph of the database D. However, it is easy to see that the augmented reachability graph contains two maximal cliques. One clique is the set $\{o_0, o_1, o_2, o_3, o_4\}$ and the other clique is the set $\{o_0, o_1, o_2, o_3, o_5\}$. These cliques are the leaves of the Bachman diagram that is created from the maximal-clique hypergraph of the augmented reachability graph.

8. CONCLUSIONS

The semiflexible and flexible semantics facilitate easy and intuitive querying of semistructured data. Meaningful queries can be formulated even when the user is oblivious to the structural details of the data and is only familiar with the ontology. Moreover, queries are insensitive to common variations in the schemas of semantically similar data instances. For both semantics, query evaluation and optimization have favorable complexities. Tree queries can be evaluated in polynomial time in the size of the query, the data and the result. In the case of the semiflexible semantics, even evaluation of dag queries has the same complexity provided that the database is a tree. Equivalence of queries is decidable in polynomial time, except for the case of semiflexible cyclic queries; in this case, there is an exponential-time algorithm for testing equivalence.

A novel feature of the new semantics is the possibility of transforming a given database D to a tree database that is equivalent to D. For the semiflexible semantics, testing whether D is equivalent to a tree database and actually transforming D to a tree database (if the test is positive) are both polynomial. Since queries can be evaluated more efficiently over tree databases, this result is of practical importance.

More research is needed in order to determine which of the two semantics is more appropriate in common applications. A second interesting subject for future research is extending the semiflexible and flexible semantics to handle incomplete information, as was done in [19] for the rigid semantics.

9. REFERENCES

[1] S. Abiteboul. Querying semi-structured data. In *International Conference on Database Theory*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18, Delphi (Greece), Jan. 1997. Springer-Verlag.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 2000.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[4] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. 16th Symposium on Principles of Database Systems*, pages 122–133, Tucson (Arizona, USA), May 1997. ACM Press.

[5] P. Atzeni, G. Mecca, and P. Merialdo. To weave the Web. In *Proc. 23nd International Conference on Very*

Large Data Bases, pages 206–215, Athens (Greece), Aug. 1997. Morgan Kaufmann Publishers.

[6] Z. Bar-Yossef, Y. Kanza, Y. Kogan, W. Nutt, and Y. Sagiv. Querying semantically tagged documents on the world-wide web. In *Proc. 4th International Workshop on Next Generation Information Technologies and Systems*, Zichron Yaakov (Israel), July 1999. Springer-Verlag.

[7] C. Baru, A. Gupta, B. Ludacher, R. Marciano, Y. Papakonstantinou, and P. Valikhov. Xml-base information mediation with mix. In *SIGMOD System Demonstration*, 1999.

[8] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, editors. *The World Wide Web Consortium (W3C)'s XML web page.* `http://www.w3c.org/XML`, 1998.

[9] P. Buneman. Semistructured data. In *Proc. 16th Symposium on Principles of Database Systems*, pages 117–121, Tucson (Arizona, USA), May 1997. ACM Press.

[10] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *International Conference on Database Theory*, pages 336–350, Delphi (Greece), Jan. 1997. Springer-Verlag.

[11] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, pages 505–516, Montreal (Canada), June 1996.

[12] P. Buneman, W. Fan, and S. Weinstein. Path constraints in semistructured and structured databases. In *Proc. 17th Symposium on Principles of Database Systems*, pages 129–138, Seattle (Washington, USA), June 1998. ACM Press.

[13] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. In *Proc. 14th International Conference on Data Engineering*, pages 4–13, Orlando (Florida, USA), Feb. 1998. IEEE Computer Society.

[14] I. Cruz, A. Mendelzon, and P. Wood. A graphical query language supporting recursion. In *Proc. 1982 International Conference on Management of Data*, pages 323–330, Orlando (Florida, USA), June 1982.

[15] A. Deutsch, M. Fernandez, D. Florescu, A. Levi, and D. Suciu. A query language for xml. In *Proc. of the World Wide Web Conference*, 1999.

[16] R. Fagin. Degree of acyclicity for hypergraphs and relational database schemas. *J. ACM*, 7(3):343–360, 1983.

[17] R. Goldman and J. Widom. Dataguides: enabling query formulation and optimization in semistructured databases. In *Proc. 23nd International Conference on Very Large Data Bases*, Athens (Greece), Aug. 1997. Morgan Kaufmann Publishers.

[18] G. Grahne and L. V. S. Lakshmanan. On the difference between navigating semistructered data and querying it. In *8th International Workshop on Database Programming Languages*, Kinloch Rannoch (Scotland), Sept. 1999.

[19] Y. Kanza, W. Nutt, and Y. Sagiv. Queries with incomplete answers over semistructured data. In *pods99*, pages 227–236, Philadelphia (Pennsylvania, USA), June 1999. ACM Press.

[20] L. Lakshmanan, F. Sadri, and I. Subramanian. A declarative language for querying and restructuring the web. In *Proc. 6th International Workshop on Research Issues on Data Engineering - Interoperability of Nontraditional Database Systems*, pages 12–21, New Orleans (Louisiana, USA), Feb. 1996. IEEE Computer Society.

[21] D. Maier, Y. Sagiv, and M. Yannakakis. On the complexity of testing implications of functional and join dependencies. *J. ACM*, 28(4):680–695, 1981.

[22] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. The Araneus web-base management system. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pages 544–546, Seattle (Washington, USA), June 1998. ACM Press.

[23] A. Mendelzon and T. Milo. Formal models of web queries. In *Proc. 16th Symposium on Principles of Database Systems*, pages 134–143, Tucson (Arizona, USA), May 1997. ACM Press.

[24] A. Mendelzon and P. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(5), 1995.

[25] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. 19th Symposium on Principles of Database Systems*, pages 145–156, Dallas (Texas, USA), May 2000. ACM Press.

[26] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange acroos heterogeneous information sources. In *Proc. 11th International Conference on Data Engineering*, pages 251–260, Taipei, Mar. 1995. IEEE Computer Society.

[27] A. Rajaraman and J. Ullman. Integrating information by outerjoins and full disjunctions. In *Proc. 15th Symposium on Principles of Database Systems*, pages 238–248, Montreal (Canada), June 1996. ACM Press.

[28] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. 7th International Conference on Very Large Data Bases*, pages 82–94, Cannes (France), Sept. 1981. IEEE Computer Society.