Computing Full Disjunctions*

(Extended Abstract)

Yaron Kanza
School of Computer Science and Engineering
The Hebrew University of Jerusalem
yarok@cs.huji.ac.il

Yehoshua Sagiv
School of Computer Science and Engineering
The Hebrew University of Jerusalem
sagiv@cs.huji.ac.il

ABSTRACT

Under either the OR-semantics or the weak semantics, the answer to a query over semistructured data consists of maximal rather than complete matchings, i.e., some query variables may be assigned null values. In the relational model, a similar effect is achieved by computing the full disjunction (rather than the natural join or equijoin) of the given relations. It is shown that under either the OR-semantics or the weak semantics, query evaluation has a polynomial-time complexity in the size of the query, the database and the result. It is also shown that the evaluation of full disjunctions is reducible to query evaluation under the weak semantics and hence can be done in polynomial time in the size of the input and the output. Complexity results are also given for two related problems. One is evaluating a projection of the full disjunction and the other is evaluating the set of all tuples in the full disjunction that are non-null on some given attributes. In the special case of γ-acyclic relation schemes, both problems have polynomial-time algorithms in the size of the input and the output. In the general case, such algorithms do not exist, assuming that P $\neq$ NP. Finally, it is shown that the weak semantics can generalize full disjunctions by allowing tuples to be joined according to general types of conditions, rather than just equalities among attributes.

1. INTRODUCTION

Most query language return *complete* answers, that is, answers that provide bindings to all the variables of the query. If the database has incomplete information, the user has to formulate the query differently, e.g., by using outerjoins or conditions that check whether some attributes are null. The need to be aware of the possibility of incomplete information is further complicated by the fact that the semantics of outerjoins is problematic, since outerjoins are not associative.

*This research was supported by The Israel Science Foundation (Grant 96/01).

PODS 2003, June 9-12, 2003, San Diego, CA.

Galiando-Legaria [11] proposed full disjunctions, which are commutative and associative, as an alternative to outerjoins. Essentially, the full disjunction of a set of relations is obtained by taking all joins of any subset of relations, excluding joins that involve a Cartesian product, and removing subsumed tuples. Rajaraman and Ullman [23] enunciated full disjunctions as a mechanism for integrating information from the Web, since the user cannot have any a priori knowledge of where information might be incomplete.

For queries over semistructured data, several semantics that differ in their degree of incompleteness were proposed in [15, 16]. Query formulation remains the same under all these semantics, but the user can choose the degree of incompleteness that suits her best.

An important issue is the complexity of evaluating incomplete answers. Traditionally, complexity of query evaluation is measured in terms of either data complexity or query complexity [28]. But neither one can capture the differences in time complexity between evaluation of queries under complete semantics vs. the incomplete semantics of [15, 16]. For all these semantics, data complexity is polynomial and query complexity is exponential.

The move from the traditional complete semantics to an incomplete semantics could potentially have two opposing effects. On one hand there are more answers, but on the other hand finding some answer could be easier, since it is not necessary to satisfy all the conditions of the query.

Another issue is subsumptions among answers. There is no need to keep an answer if there is another answer that has all the bindings of the first answer plus some additional bindings. For example, if the tuple (a, b, c) is an answer, then there is no need to include also the tuple $(a, b, \perp)$ as an answer. If subsumption is not handled judiciously during query evaluation, then it could add another exponential factor to the evaluation time under query complexity.

A more suitable measure of complexity is *input-output* complexity, in which the time to evaluate a query is measured as a function of the size of the input (i.e., both the query and the database) and the output. Yannakakis [30] used this complexity measure to show that there is a polynomial-time algorithm for evaluating the natural join of n relations if the relation schemes are α-acyclic. In the general case, deciding non-emptiness of the natural join of n relations is

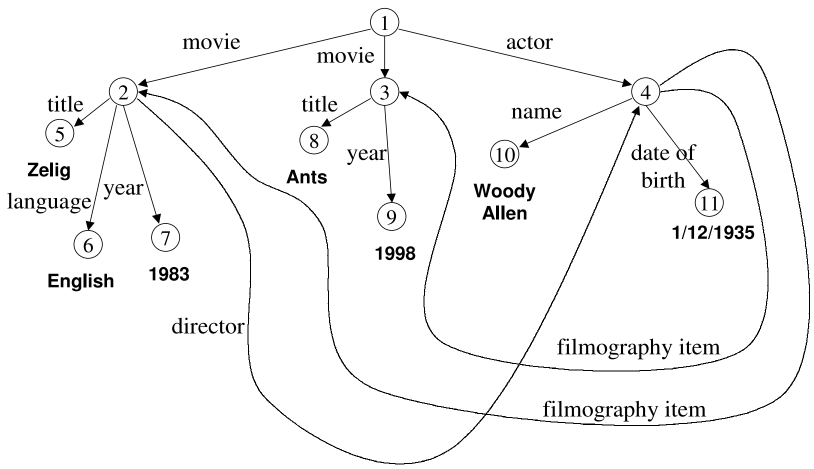

Figure 1: A movie database.

NP-complete [20].

In [15, 16], it was shown that for both the OR-semantics and the weak semantics, tree as well as DAG queries (without selections and projections) can be evaluated in polynomial time under input-output complexity. In this paper we extend this result to queries that may have cycles. We also show that computing full disjunctions is reducible to evaluating queries under the weak semantics. Thus, it follows that full disjunctions can always be computed in polynomial time in the size of the input and the output. Note that for the special case of relations with γ-acyclic schemes, this result is a direct corollary of the result of Rajaraman and Ullman [23] that the full disjunction of n relations can be computed by a sequence of natural outerjoins if and only if the relation schemes form a connected and γ-acyclic hypergraph. We also give complexity results for two related problems. One is evaluation of a projection of the full disjunction. The other is evaluation of all tuples of the full disjunction that are not null on a given set of attributes.

When integrating information, full disjunctions are rather limited, since they are based solely on equalities among attributes as in natural joins and equijoins. In many cases, there is a need to integrate information based on conditions that are more general than merely equalities among attributes. We show how our approach leads to a method of integrating information by means of general types of conditions and not just equality conditions.

Section 2 introduces the data model, patterns (i.e., queries without selections and projections) and matchings of patterns to a given semistructured database. In addition, maximal OR-matchings and maximal weak matchings are defined in Section 2. In Section 3, the algorithm for computing maximal OR-matchings is presented, along with a proof of correctness and an analysis of the time complexity. In Section 3 it is also shown how to modify the algorithm in order to compute maximal weak matchings. In Section 4, full disjunctions are defined and it is shown how to reduce the evaluation of full disjunctions to the evaluation of maximal weak matchings. Section 5 presents a generalization of full disjunctions that facilitates integration of relational data by means of join conditions that are more general than just equalities. Finally, in Section 6, we survey some related work and conclude.

2. DATA MODEL AND QUERIES

In this work, we use the *semistructured data model*, which is based on the Object Exchange Model (OEM) of [21]. The data are represented by rooted labeled directed graphs. Each node represents an *object* and has an identifier (*oid*). Values are attached to *atomic nodes* (i.e., nodes without outgoing edges). For simplicity, we assume that all atomic nodes are of type PCDATA (i.e., *string*). Nodes with outgoing edges represent *complex objects*. A database has a *root* and every node in the database is reachable from the root.

Formally, a semistructured database D is a 4-tuple $D = \langle O, E_D, r_D, \alpha\rangle$, where O is a finite set of objects, E_D is a set of labeled edges, r_D is the root and α is a function that maps each atomic node to a value.

EXAMPLE 2.1. Figure 1 shows a movie database. Nodes are depicted as circles and the oid of each node appears inside the circle. The values that are attached to atomic nodes are shown below those nodes, in boldface. Labels appear next to the edges.

2.1 Patterns and Matchings

The queries considered in this paper are called *patterns* and they are a simplified form of the queries discussed in [15]. Patterns are represented as rooted labeled directed graphs. The nodes of a pattern represent variables rather than objects. Formally, a *pattern* is a 3-tuple $P = \langle V, E_P, r_P\rangle$, where V is a finite set of *variables*, E_P is a set of labeled edges and r_P is the root of the pattern. We use ulv to denote an edge with label l from node u to node v.

Let $P = \langle V, E_P, r_P\rangle$ be a pattern and $D = \langle O, E_D, r_D, \alpha\rangle$ be a database. An *assignment* of P with respect to (abbreviated w.r.t.) the database D is a mapping $\mu\colon V \to O \cup \{\perp\}$, such that each variable of P is mapped either to an object

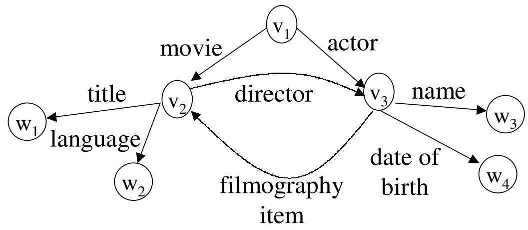

Figure 2: A pattern over the movie database.

of D or to a special value that is called *null* and is denoted as $\perp$.

An *assignment graph* G of a pattern $P = \langle V, E_P, r_P \rangle$ w.r.t. a database $D = \langle O, E_D, r_D, \alpha \rangle$ consists of nodes of the form (v, o), where v is a node of P and o is a node of D. The assignment graph G cannot have any *repeated variable,* i.e., it cannot have any variable v that appears in two distinct nodes (v, o_1) and (v, o_2) $(o_1 \neq o_2)$. The assignment graph G may (but does not necessarily) have an edge from (v_1, o_1) to (v_2, o_2), denoted $(v_1, o_1)l(v_2, o_2)$, provided that v_1lv_2 is an edge of P and o_1lo_2 is an edge of D. We say that $(v_1, o_1)l(v_2, o_2)$ is a *potential edge* of G if v_1lv_2 is in P and o_1lo_2 is in D. If the edge $(v_1, o_1)l(v_2, o_2)$ appears in G, then it is called an *actual edge* of G.

An assignment graph G corresponds to an assignment μ that is defined as follows. If (v, o) is a node of G, then $\mu(v) = o$, and if G has no node with v as its first component (i.e., no node of the form (v, o) for some o), then $\mu(v) = \perp$. The assignment μ is well defined, since G does not have any repeated variable.

Conversely, if μ is an assignment, then an assignment graph for μ comprises all nodes of the form $(v, \mu(v))$, where $\mu(v)$ is non-null. The assignment μ may have several assignment graphs. All those graphs have the same set of nodes, but they may have different sets of edges. However, the set of actual edges is always a subset of the set of potential edges.

Let G be an assignment graph of a pattern P with respect to a database D. The *closure* of G is obtained by adding to G all potential edges that are not already in G. We say that G is *closed* if G is equal to its closure.

A *matching* is an assignment that satisfies some additional conditions. In this paper, we define two types of matchings. The definitions are equivalent to those in [15].

An assignment μ of a pattern $P = \langle V, E_P, r_P \rangle$ w.r.t. a database $D = \langle O, E_D, r_D, \alpha \rangle$ is an OR-*matching* if μ has an assignment graph M that satisfies the following conditions.

1. M has the node (r_P, r_D) (i.e., $\mu(r_P) = r_D$) and that node is designated as the root of M; and
2. Each node of M is reachable from the root.

An assignment graph M of a pattern P w.r.t. a database D *preserves* the edges of P if it satisfy the following condition. For each two nodes (v_1, o_1) and (v_2, o_2) of M, if v_1lv_2 is an edge of P, then $(v_1, o_1)l(v_2, o_2)$ is an edge of M (and thus, o_1lo_2 is an edge of D). An assignment μ is a *weak matching* of P if it has an assignment graph M that satisfies the conditions of an OR-matching and, in addition, M preserves the edges of P.

An assignment graph that satisfies the conditions of an OR-matching is called an OR-*matching graph.* Similarly, an assignment graph that satisfies the conditions of a weak matching is called a *weak-matching graph.* Note that a weak-matching graph is always closed.

Essentially, the result of applying a pattern to a database is a set of matchings. Under the OR-semantics, this set comprises all the OR-matchings; under the weak semantics, this set comprises all the weak matchings.

2.2 Subsumption and Maximal Matchings

Let $\mathcal{A}$ be a set of assignments of a pattern P w.r.t. a database D. Given two assignments μ and μ' of $\mathcal{A}$, we say that μ *subsumes* μ', denoted $\mu' \sqsubseteq \mu$, if $\mu(x) = \mu'(x)$ whenever $\mu'(x)$ is non-null. The assignment $\mu \in \mathcal{A}$ is a *maximal* element of $\mathcal{A}$ if no element in $\mathcal{A}$, other than μ itself, subsumes μ. Intuitively, if $\mu' \sqsubseteq \mu$, then μ has more information than μ'. Therefore, all non-maximal elements can be discarded from $\mathcal{A}$ without any loss of information.[1]

Subsumption is defined for assignment graphs in the natural way. Let G_1 and G_2 be assignment graphs that correspond to the assignments μ_1 and μ_2, respectively. We say that G_1 subsumes G_2 if μ_1 subsumes μ_2. We say that G_1 is maximal if μ_1 is maximal. Note that subsumption among assignment graphs depends only on the nodes, but not on the edges. Therefore, two assignment graphs could subsume each other even if they are not identical.

Given a pattern P and a database D, the set of all maximal OR-matchings of P w.r.t. D is denoted as $\mathcal{M}_\vee(P, D)$. Similarly, the set of all maximal weak matchings of P w.r.t. D is denoted as $\mathcal{M}_w(P, D)$. Note that both $\mathcal{M}_\vee(P, D)$ and $\mathcal{M}_w(P, D)$ can be viewed as relations with columns that are labeled with the variables of P and tuples that are filled with oid's and nulls.

EXAMPLE 2.2. Figure 2 shows a pattern over the movie

[1] This may not be true if bag semantics has to be assumed (e.g., in order to evaluate aggregate queries). However, the issue of bag semantics is beyond the scope of this paper.

Semantics	v_1	v_2	v_3	w_1	w_2	w_3	w_4
OR	1	2	4	5 (Zelig)	6 (English)	10 (Woody Allen)	11 (1/12/1935)
	1	3	4	8 (Ants)	$\perp$	10 (Woody Allen)	11 (1/12/1935)
Weak	1	2	4	5 (Zelig)	6 (English)	10 (Woody Allen)	11 (1/12/1935)
	1	3	$\perp$	8 (Ants)	$\perp$	$\perp$	$\perp$

Figure 3: $\mathcal{M}_\vee(P,D)$ and $\mathcal{M}_w(P,D)$ for the P and D of Figures 1 and 2.

database of Figure 1. Intuitively, the pattern presents a query that looks for an actor (along with her name and date of birth) and a movie (along with its title and language). The pattern specifies two relationships between the movie and the actor. First, the edge labeled with "filmography item" requires that the actor indeed acted in the movie. Secondly, the edge labeled with "director" requires that the actor was also the director of the movie.

The answers to the query differ according to the semantics of the query. The set of maximal OR-matchings and the set of maximal weak matchings are shown in Figure 3. Each matching is shown as a tuple of oid's and nulls. For atomic nodes, the value is shown next to the oid, inside parenthesis. Formally, however, a matching is always an assignment of oid's and not of values.

In a maximal weak matching, the edges of the pattern are preserved. Thus, an answer to the query includes a movie and an actor only if the actor acted in the movie and was also the director of the movie. The first weak matching in Figure 3 finds Woody Allen as both an actor and the director of the movie Zelig. An actor will be in an answer without a movie (i.e., null in v_2) if none of the movies of the actor was directed by her. A movie will be in an answer without an actor (i.e., null in v_3) if none of the actors in the movie was also a director of the movie. The second weak matching in Figure 3 includes the movie Ants without an actor, since no actor in Ants was also the director of this movie.

In a maximal OR-matching, the edges of the pattern are not necessarily preserved. Thus, an OR-matching can include an actor and a movie even if the actor neither acted in the movie nor was the director of the movie. The second OR-matching in Figure 3 gives a movie and an actor, such that the actor acted in the movie but was not the director. There are no other maximal OR-matchings in this example, since a movie always has an actor and an actor (or a director) always has a movie in the given database.

3. COMPUTING MAXIMAL MATCHINGS

Algorithms for computing the sets $\mathcal{M}_\vee(P,D)$ and $\mathcal{M}_w(P,D)$ were given in [16] for patterns that are acyclic graphs. The time complexity of these algorithms was shown to be polynomial in the size of the input (i.e., the pattern and the database) and the output (i.e., the set of all maximal matchings). In this section, we generalize this result to patterns that may have cycles.

3.1 The Product Graph

Consider a pattern $P = \langle V, E_P, r_P\rangle$ and a database $D = \langle O, E_D, r_D, \alpha\rangle$. The *product* of P and D, denoted $P{\times}D$, is the graph $\mathcal{P} = \langle \mathcal{V}, \mathcal{E}, r_{\mathcal{P}}\rangle$, where

1. The root of $\mathcal{P}$, denoted $r_{\mathcal{P}}$, is the pair (r_P, r_D);
2. The set of nodes of $\mathcal{P}$, denoted $\mathcal{V}$, consists of all nodes of the form (v,o), where v is a variable of P and o is an object of D, i.e., $\mathcal{V} = V \times O$; and
3. The set of edges of $\mathcal{P}$, denoted $\mathcal{E}$, consists of all edges of the form $(v_1,o_1)l(v_2,o_2)$, where $v_1 l v_2$ is an edge of P and $o_1 l o_2$ is an edge of D.

EXAMPLE 3.1. The product of the pattern from Figure 2 and the database from Figure 1 is depicted in Figure 4. Note that only nodes that are reachable from the root are shown.

G is a *subgraph* of $P{\times}D$ if G comprises some subset of the nodes of $P{\times}D$. The edges of G may be some or all the edges of $P{\times}D$ that connect nodes appearing in G. It is rather obvious that if M is either an OR-matching graph or a weak-matching graph, then M is a subgraph of $P{\times}D$. The next proposition states when the converse is also true.

PROPOSITION 3.2. *Consider the following four conditions on a subgraph G of $P{\times}D$.*

1. *G has no repeated variables; that is, G does not have two nodes (v,o_1) and (v,o_2), such that $o_1 \neq o_2$.*
2. *G contains the root of $P{\times}D$.*
3. *In G, each node is reachable from the root.*
4. *G preserves the edges of P.*

A subgraph G of $P{\times}D$ is an OR-matching graph if and only if it satisfies the first three conditions. G is a weak-matching graph if and only if it satisfies all four conditions.

PROOF. Condition 1 is needed so that G will be the assignment graph of an assignment μ that is defined as follows. If (v,o) is a node of G, then $\mu(v) = o$; otherwise, $\mu(v) = \perp$. Conditions 2 and 3 are identical to the two conditions in the definition of an OR-matching. Condition 4 is the additional condition in the definition of a weak matching. $\square$

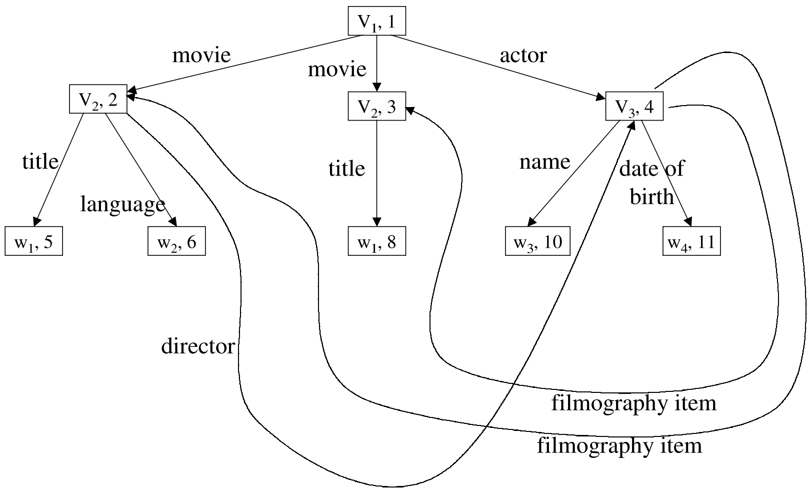

Figure 4: The product of the pattern and the database from Figures 1 and 2.

3.2 Computing Maximal OR-Matchings

In this section, we will show how to compute the maximal OR-matchings of a pattern P w.r.t. a database D. The main idea is to compute all maximal subgraphs G of $P \times D$, such that G satisfies Conditions 1–3 of Proposition 3.2. (Recall that several maximal OR-matching graphs could subsume each other without being equal. Obviously, the algorithm would compute just one graph for each maximal OR-matching.)

The computation is incremental; it starts with small subgraphs and expands them by adding edges. Initially, the root is the only current subgraph. In every iteration, each current subgraph is expanded by adding one more edge from $P \times D$. The algorithm works efficiently if the edges are added in the "right order." If P is a DAG (directed acyclic graph), a topological sort of P gives the right order and, in this case, there is really no need to consider the product $P \times D$. Instead, it is sufficient to apply topological sort just to P; see the details in [15, 16].

3.2.1 The General Idea

When P has cycles, a different approach is needed. Instead of P itself, the product $P \times D$ should be considered. Furthermore, when adding edges in the "right order," it is not sufficient to consider each edge just once. To formalize the notion of the "right order," we say that an edge $m_1 l m_2$ of $P \times D$ has a *depth* d if there is a path consisting of d edges that starts at the root of $P \times D$ and ends with the edge $m_1 l m_2$. Two observations should be noted. First, an edge can have more than one depth when $P \times D$ is not a tree. Second, an edge with depth 1 emanates from the root of $P \times D$.

We divide the edges of $P \times D$ into strata, such that stratum k contains all the edges at depth k (and k is called the *depth* of the stratum). Note that an edge may belong to more than one stratum. A *stratum traversal* of $P \times D$ is an ordered list T that starts with the edges of stratum 1, followed by the edges of stratum 2, and so on. Notice that the order of the edges within each stratum is unimportant and can be determined arbitrarily, but the edges of stratum k must appear before the edges of stratum $k+1$. Clearly, a stratum traversal may have multiple occurrences of the same edge, but in each stratum a given edge may appear at most once.

An important part of the algorithm for computing maximal OR-matchings is to create a stratum traversal T in polynomial time. However, as defined, a stratum traversal could be infinite. We will now show how to construct a finite stratum traversal T in polynomial time.

An occurrence of an edge e in stratum k is *extraneous* if e is not the kth edge of any path that emanates from the root and consists of distinct edges. In principle, all extraneous occurrences of edges can be removed from a stratum traversal, but doing so is computationally hard. We will use a stratum traversal with some extraneous occurrences of edges. However, these extraneous occurrences neither affect the correction of the algorithm nor add an exponential blowup to the time complexity.

Obviously, if there are m edges in $P \times D$, then only the first m strata may have non-extraneous occurrences of edges. Actually, it is sufficient for a stratum traversal to include only the first n strata, where n is the number of nodes in P. The reason for this is that a matching graph cannot include two nodes (v, o_1) and (v, o_2), such that $o_1 \neq o_2$. Therefore, the number of nodes in a matching graph cannot exceed the number of nodes in P.

The algorithm for computing maximal OR-matchings uses a stratum traversal T that consists of the first n strata, where n is the number of nodes in the pattern P, and is constructed as follows. Stratum 1 contains all the edges that emanate from the root of $P \times D$. Stratum k contains all edges $(v_1, o_1) l_1 (v_2, o_2)$ of $P \times D$, such that stratum $k-1$ has an edge that enters node (v_1, o_1). The size of the stratum traversal T is $O(p^2 d)$, where p is the size of the pattern P and d is the size of the database D. T can be constructed in $O(p^2 d)$ time.

Let T be the stratum traversal that was constructed as described above. The maximal OR-matchings are generated by an incremental algorithm that iterates over the edges in T. Initially, the set of current subgraphs of $P \times D$ has only one member, namely, the root of $P \times D$. In the ith iteration, each current subgraph is expanded, if possible, by adding the edge that appears in position i of T.

3.2.2 The Various Cases of Adding An Edge

Let G be a subgraph of $P \times D$, such that G satisfies Conditions 1–3 of Proposition 3.2; that is,

- G has no repeated variables;
- G contains the root of $P \times D$; and
- All the nodes of G are reachable from the root (via edges that belong to G).

Consider an attempt to add an edge $(v_1, o_1)l(v_2, o_2)$ of $P \times D$ to G. Several cases are possible and we now consider them one by one.

CASE 1. *The node (v_2, o_2) satisfies $v_2 = r_P$ and $o_2 \neq r_D$.* In this case, the edge $(v_1, o_1)l(v_2, o_2)$ is not added to G, since the node (v_2, o_2) cannot belong to a subgraph of $P \times D$ that satisfies Conditions 1–3 of Proposition 3.2. Thus, G remains unchanged.

CASE 2. *The node (v_1, o_1) is not in G.* In this case, the edge $(v_1, o_1)l(v_2, o_2)$ cannot be added to G and G remains unchanged.

CASE 3. *The nodes (v_1, o_1) and (v_2, o_2) as well as the edge that connects them are all in G.* In this case, G remains unchanged.

CASE 4. *Both (v_1, o_1) and (v_2, o_2) are in G, but the edge that connects them is not in G.* In this case, the edge $(v_1, o_1)l(v_2, o_2)$ is added to G. Notice that the result is a subgraph G' of $P \times D$ that satisfies Conditions 1–3 of Proposition 3.2. The new graph G' replaces G.

CASE 5. *The node (v_1, o_1) is in G and v_2 does not appear in any node of G.* In this case, the edge $(v_1, o_1)l(v_2, o_2)$ as well as the node (v_2, o_2) are added to G. Notice that the result is a subgraph G' of $P \times D$ that satisfies Conditions 1–3 of Proposition 3.2. The new graph G' replaces G.

CASE 6. *The node (v_1, o_1) is in G, $v_2 \neq r_P$ and G also has a node (v_2, o_3), where $o_2 \neq o_3$.* In this case, the addition of the edge $(v_1, o_1)l(v_2, o_2)$ to G forces the deletion of the node (v_2, o_3). Thus, a new subgraph G' is created from G as follows. First, the node (v_2, o_2) and the edge $(v_1, o_1)l(v_2, o_2)$ are added to G. Second, the node (v_2, o_3) is deleted from G'. Third, all the nodes that are not reachable from the root, due to the deletion of (v_2, o_3), are deleted from G'; edges that are incident on deleted nodes are also deleted. The result is a subgraph G' of $P \times D$ that satisfies Conditions 1–3 of Proposition 3.2. Notice that G is not subsumed by G', since the former includes the node (v_2, o_3), but the latter does not. However, it could be that G subsumes G' (that may happen if the edge $(v_1, o_1)l(v_2, o_2)$ that was initially added to G is eventually deleted). In summary, both G and G' should be retained unless G' is subsumed by G.

3.2.3 The Algorithm

The algorithm computes a set $\mathcal{M}$ as follows.

1. Initially, $\mathcal{M}$ has one graph that consists of the root of $P \times D$.
2. Repeat the next two steps for each edge e in the stratum traversal T, starting with the first edge of T.
3. For each graph $G \in \mathcal{M}$, replace G with the result of adding e to G, according to the above six cases. Recall that G is replaced either with G itself, with a new graph G' that subsumes G or with G and a new graph G', such that neither one subsumes the other.
4. Remove subsumed graphs from $\mathcal{M}$, i.e., a graph $G \in \mathcal{M}$ is removed if it is subsumed by some other graph of $\mathcal{M}$ (and this step is repeated until there are no subsumptions among the graphs of $\mathcal{M}$).

The next theorem shows that the algorithm is correct and gives its time complexity.

THEOREM 3.3. *Given a pattern P and a database D, the algorithm terminates with $\mathcal{M} = \mathcal{M}_\vee(P, D)$. The running time of the algorithm is $O(p^3 dm^2)$, where p is the size of P, d is the size of D and m is the size of $\mathcal{M}_\vee(P, D)$.*

3.2.4 Correctness of the Algorithm

The algorithm iterates over all the edges in the stratum traversal T. In each iteration, Steps 3 and 4 are executed once. The ith iteration is for the edge that occurs in position i of T. The value of $\mathcal{M}$ at the end of the ith iteration is denoted as $\mathcal{M}^i$.

Consider a maximal OR-subgraph M of $P \times D$. Intuitively, we would like to prove that for all i, there is a graph $G \in \mathcal{M}^i$, such that G subsumes the restriction of M just to the edges that appear in the first i positions of T. However, the minimal depth of an edge in M may be greater than its minimal depth in $P \times D$. Therefore, a more elaborate definition is needed.

Consider a position i in the stratum traversal T and suppose that position i occurs in the stratum that has depth d. Let M be a maximal OR-matching subgraph of $P \times D$. The *i-portion* of M, denoted M^i, consists of all the edges $m_1 l m_2$ of M, such that the minimal depth of $m_1 l m_2$ in M is not greater than d and there is at least one occurrence of $m_1 l m_2$ among the first i positions of T. Note that, by definition, the 0-portion of M consists just of the root of M. Also note that M^i is an OR-matching subgraph of $P \times D$.

Next, we will prove the following lemma that shows the correctness of the algorithm.

Lemma 3.4. *When the algorithm complete its execution,* $\mathcal{M} = \mathcal{M}_{\vee}(P, D)$.

Proof. The proof follows from the following two claims.

Claim 3.5. *During the evaluation of the algorithm, all the graphs in* $\mathcal{M}$ *are* OR*-matchings graphs.*

This claim is true, since all the graphs in $\mathcal{M}$ satisfy Conditions 1–3 of Proposition 3.2 throughout the evaluation of the algorithm.

Claim 3.6. *When the algorithm terminates, the following holds. For every maximal* OR*-matching graph* M*, the set* $\mathcal{M}$ *has a graph* G*, such that* G *subsumes* M.

Together, the two claims imply that at the end of the algorithm, $\mathcal{M}$ is exactly the set of all maximal OR-matching graphs, since the set $\mathcal{M}$ is subsumption free.

Claim 3.6 is proven by induction, using the following inductive hypothesis: If M is a maximal OR-graph, then M^i is subsumed by some $G \in \mathcal{M}^i$.

For the basis of the induction, we should consider the initialization of $\mathcal{M}$ just prior to the first iteration. Initially, $\mathcal{M}$ contains one subgraph G of $P \times D$ that comprises the root of $P \times D$. The 0-portion of M is just the root, so the induction hypothesis is true.

Now suppose that the inductive hypothesis holds at the end of the ith iteration. We will show that it also holds at the end of the $(i+1)$st iteration. Let M be a maximal OR-matching graph. By the inductive hypothesis, there is a graph G in $\mathcal{M}^i$, such that M^i is subsumed by G.

Since $G \in \mathcal{M}^i$, there is a $G' \in \mathcal{M}^{i+1}$, such that G' subsumes G. Therefore, if M^i and M^{i+1} are identical, then G' also subsumes M^{i+1} and the inductive hypothesis is proven.

Let $(v_1, o_1)l(v_2, o_2)$ be the $(i+1)$st edge in the stratum traversal T. If M^i and M^{i+1} are different, then the edge $(v_1, o_1)l(v_2, o_2)$ appears in M^{i+1} but not in M^i.

Next, we need to consider the six cases of adding the edge $(v_1, o_1)l(v_2, o_2)$ to G during the $(i+1)$st iteration, as described in Section 3.2.2.

In the first case, the edge $(v_1, o_1)l(v_2, o_2)$ cannot be in M and, thus, M^i and M^{i+1} are identical.

In the second case, node (v_1, o_1) is not in G and, hence, it cannot be in M^i. Thus, the edge $(v_1, o_1)l(v_2, o_2)$ is not in M^{i+1} either. Consequently, M^i and M^{i+1} are identical.

In the third case, G already has the edge $(v_1, o_1)l(v_2, o_2)$. Since G subsumes M^i, it follows that it also subsumes M^{i+1}. Therefore, G' subsumes M^{i+1}.

In the fourth case, the situation is similar to that in the third case, since subsumption between graphs depends only on the nodes of each graph, but not on the edges.

In the fifth case, the graph G' has all the nodes and edges of G and also the added edge $(v_1, o_1)l(v_2, o_2)$. G subsumes M^i and, hence, G' subsumes M^{i+1}.

Finally, consider the sixth case. First note that if M includes the node (v_2, o_3), then M^i and M^{i+1} are identical and, as argued earlier, the inductive hypothesis is proven. So, we will assume that M does not include the node (v_2, o_3). In the sixth case, a new subgraph G' is created from G by adding the edge $(v_1, o_1)l(v_2, o_2)$ and then deleting the node (v_2, o_3) as well as all the nodes that are not reachable from the root. Let $\bar{G}$ be the subgraph that is obtained immediately after the addition of $(v_1, o_1)l(v_2, o_2)$ and just before the deletion of any node. Since M does not include (v_2, o_3), all the nodes of M^{i+1} are reachable from the root via paths that do not include (v_2, o_3). By the induction hypothesis, M^i is subsumed by G and, consequently, all the nodes of M^{i+1} also appear in $\bar{G}$. Therefore, in $\bar{G}$, all the nodes of M^{i+1} are reachable from the root via paths that do not include (v_2, o_3). Thus, none of the nodes of M^{i+1} is deleted from $\bar{G}$ and, so, G' subsumes the graph M^{i+1}.

In summary, in all of the above cases, the inductive hypothesis holds at the end of the $(i+1)$st iteration. $\square$

3.2.5 Time Complexity

In this section, we will prove the part of Theorem 3.3 that refers to the time complexity. The proof hinges on the fact that the number of graphs in $\mathcal{M}$ does not decrease from one iteration to the next, as shown by the following lemma.

Lemma 3.7. $|\mathcal{M}^i| \leq |\mathcal{M}^{i+1}|$.

Proof. The lemma is proved by showing that each $G' \in \mathcal{M}^{i+1}$ subsumes at most one $G \in \mathcal{M}^i$. To derive a contradiction, we assume that G_1 and G_2 are two graphs in $\mathcal{M}^i$ that are subsumed by some $G' \in \mathcal{M}^{i+1}$. Suppose that position i of the stratum traversal T occurs in a stratum that has a depth d. Consider the graph $\hat{G}$ that consists of all the edges that appear in either G_1 or G_2, as well as all the nodes connected by those edges. (If there are no edges, then $\hat{G}$ consists just of the root of $P \times D$.) Clearly, each edge of $\hat{G}$ has a depth (in $\hat{G}$) that is not greater than d and has an occurrence among the first i positions of the stratum traversal T. Furthermore, $\hat{G}$ has no repeated variables, since it is subsumed by another OR-matching subgraph, G'. Thus, $\hat{G}$ is an OR-matching subgraph of $P \times D$, and $\hat{G}^i$ (the i-portion of $\hat{G}$) is the same as $\hat{G}$. By the inductive hypothesis in the proof of Lemma 3.4, there is a $G \in \mathcal{M}^i$, such that G subsumes $\hat{G}$. Consequently, G subsumes both G_1 and G_2, in contradiction to the fact that $\mathcal{M}^i$ is subsumption free. $\square$

We will now complete the proof of the time complexity. By Lemma 3.7, at any time during the evaluation of the algorithm, $\mathcal{M}$ has at most twice as many graphs as the final value of $\mathcal{M}$. In Step 3 of the algorithm, adding an edge to a graph $G \in \mathcal{M}$ takes $O(p)$ time. Thus, each execution of Step 3 takes $O(pm)$ time. Each execution of Step 4 takes $O(pm^2)$ time. The size of the stratum traversal is $O(p^2d)$ and it can be constructed in $O(p^2d)$ time. Hence, Steps 3 and 4 are repeated $O(p^2d)$ times and the whole algorithm takes $O(p^3dm^2)$ time.

3.3 Computing Maximal Weak Matchings

To modify the algorithm so that it will compute the set of all maximal weak matchings, Cases 5 and 6 of Section 3.2.2 should be changed. In each of these two cases, after creating the subgraph G', there is a need to test and possibly modify G' so that it will preserve the edges of P. This is done as follows.

Let $(v_1, o_1)l(v_2, o_2)$ be the edge that was added to G'. For each node (v_i, o_i) in G', if there is a label h, such that v_ihv_2 is an edge of P, but o_iho_2 is not an edge of D, then delete node (v_i, o_i) from G'. Similarly, for each node (v_j, o_j) in G', if there is a label h, such that v_2hv_j is an edge of P, but o_2ho_j is not an edge of D, then delete node (v_j, o_j) from G'. Nodes that become non-reachable from the root should also be deleted. Edges that are incident on deleted nodes are deleted as well. As a result of these deletions, G' may no longer subsume G and consequently, in Case 5, both G' and G should be added to $\mathcal{M}$ (or just G should be added if it subsumes G'). The same is done in Case 6.

THEOREM 3.8. *Given a pattern P and a database D, the set $\mathcal{M}_w(P, D)$ of all maximal weak matchings can be computed in $O(p^3dm^2)$ time, where p is the size of P, d is the size of D and m is the size of $\mathcal{M}_w(P, D)$.*

In proving the correctness of the algorithm for computing the maximal weak matchings, Claim 3.5 should be replaced with the following claim.

CLAIM 3.9. *During the evaluation of the algorithm, the closure of each graph in $\mathcal{M}$ is a weak matching.*

Claim 3.6 remains the same, except for changing "maximal OR-matching graph M" to "maximal weak-matching graph M." The proofs of correctness and time complexity are similar to the ones in the case of computing the maximal OR-matchings.

4. FULL DISJUNCTIONS

4.1 Preliminaries

Let $r_1, \ldots, r_n$ be relations with relation schemes $R_1, \ldots, R_n$, respectively. The jth tuple of r_i is denoted as t_{ij}. Given a subset $X \subseteq R_i$, the projection of the tuple t_{ij} on X is denoted as $t_{ij}[X]$.

Two distinct relation schemes R_i and R_j are *connected* if $R_i \cap R_j$ is non-empty. This definition is generalized to m ($m \geq 1$) distinct relation schemes as follows. The *scheme graph* of $R_{i_1}, \ldots, R_{i_m}$ consists of a node for each R_{i_j} and edges in both directions between R_{i_h} and R_{i_k} ($1 \leq h < k \leq m$) if R_{i_h} and R_{i_k} are connected, i.e., $R_{i_h} \cap R_{i_k}$ is non-empty. We say that the relation schemes $R_{i_1}, \ldots, R_{i_m}$ are connected if their scheme graph is connected. (There is an equivalent definition of connectivity that is based on the notion of the hypergraph of $R_{i_1}, \ldots, R_{i_m}$.) We say that tuples $t_{i_1j_1}, \ldots, t_{i_mj_m}$ from m ($m \geq 1$) distinct relations are connected if their relation schemes are connected. Note that a single tuple $t_{i_1j_1}$ is connected.

Two connected tuples $t_{i_1j_1}$ and $t_{i_2j_2}$ are *join consistent* if $t_{i_1j_1}[R_{i_1} \cap R_{i_2}] = t_{i_2j_2}[R_{i_1} \cap R_{i_2}]$. More generally, m tuples $t_{i_1j_1}, \ldots, t_{i_mj_m}$ from m distinct relations are join consistent if every pair of connected tuples $t_{i_hj_h}$ and $t_{i_kj_k}$ is join consistent. The natural join of $t_{i_1j_1}, \ldots, t_{i_mj_m}$, denoted $\bowtie_{k=1}^{m} t_{i_kj_k}$, is either empty or has one tuple over the attributes of $\cup_{k=1}^{m} R_{i_k}$. It has one tuple if and only if $t_{i_1j_1}, \ldots, t_{i_mj_m}$ are join consistent.

A *universal* tuple is defined over the set of all the attributes, namely $\cup_{i=1}^{n} R_i$, and its columns are filled with nulls as well as non-null values. If u is non-null exactly on the attributes of Z, then $u[Z]$ is called the *non-null portion* of u and is denoted as $\hat{u}$. The universal tuple u is called an *integrated* tuple if there are $m \geq 1$ connected and join-consistent tuples $t_{i_1j_1}, \ldots, t_{i_mj_m}$, such that $\hat{u} = \bowtie_{k=1}^{m} t_{i_kj_k}$. The tuples $t_{i_1j_1}, \ldots, t_{i_mj_m}$ are called *generators* of u.

A universal tuple v *subsumes* a universal tuple u, denoted $u \sqsubseteq v$, if $v[Z] = u[Z]$, where u is non-null exactly on Z. Given a set $\mathcal{D}$ of universal tuples, $u \in \mathcal{D}$ is *maximal* if it is not subsumed by any other tuple of $\mathcal{D}$.

The *full disjunction* of the relations $r_1, \ldots, r_n$ is the set of all maximal integrated tuples that can be generated from m ($1 \leq m \leq n$) tuples of $r_1, \ldots, r_n$.

EXAMPLE 4.1. In Figure 5, the full disjunction of four relations is depicted. The first tuple of the full disjunction is created as a join of the the first tuples of the relations R_1 and R_2. The second tuple of the full disjunction is the join of the second tuple of R_1, the first tuple of R_3 and the second tuple of R_4. The third tuple is a join of the first tuples of R_3 and R_4. Finally, the fourth tuple is created from the second tuple of R_3.

4.2 Computing Full Disjunctions

In order to compute the full disjunction, we need to view tuples as objects and relation schemes as variables. By a slight abuse of notation, t_{ij} denotes both the tuple and its oid. Similarly, R_i denotes both the relation scheme and its corresponding variable.

We construct a database $D = (O, E_D, r_D, \alpha)$ and n patterns $P_i = (V, E_{P_i}, r_{P_i})$ ($1 \leq i \leq n$) as follows. D has a root r_D and an object for each tuple t_{ij}. There are edges in both directions between every pair of connected and join-consistent tuples. There is also an edge from the root to every tuple. An edge that enters a tuple of r_k ($1 \leq k \leq n$) is labeled with l_k. The values of the atomic objects can be chosen arbitrarily.

The pattern $P_i = (V, E_{P_i}, r_{P_i})$ is constructed from the sch-

A	B	C
1	2	3
2	3	4

Relation R_1

A	B	D
1	2	4

Relation R_2

A	E
2	3
3	2

Relation R_3

C	D	E
3	3	3
4	3	3

Relation R_4

A	B	C	D	E
1	2	3	4	$\perp$
2	3	4	3	3
2	$\perp$	3	3	3
3	$\perp$	$\perp$	$\perp$	2

The Full Disjunction of the Relations R_1, R_2, R_3, R_4

Figure 5: The full disjunction of four relations.

eme graph of $R_1, \ldots, R_n$ by adding a root r_{P_i} and an edge from the root to R_i. An edge that enters R_k $(1 \le k \le n)$ is labeled with l_k. Note that only one connected component of the scheme graph of $R_1, \ldots, R_n$ can be reached from the root of P_i.

Consider a weak matching μ of some P_i w.r.t. D. Recall that μ assigns to each variable of P_i either a tuple or null. Let $g(\mu)$ denote the tuples in the image of μ. Since μ is a weak matching, the tuples of $g(\mu)$ are connected and join consistent. Therefore, the tuples of $g(\mu)$ generate an integrated tuple that is denoted as $\bowtie(\mu)$.

PROPOSITION 4.2. *The tuple u is an integrated tuple of $r_1, \ldots, r_n$ if and only if there is a weak matching μ of some P_i w.r.t. D, such that $u = \bowtie(\mu)$.*

PROOF. As noted earlier, if μ is a weak matching of P_i w.r.t. D, then $\bowtie(\mu)$ is an integrated tuple. For the other direction, consider an integrated tuple u that is generated by the tuples $t_{i_1 j_1}, \ldots, t_{i_m j_m}$. Let μ be the assignment for P_{i_1} w.r.t. D that is defined as follows: $\mu(r_{P_{i_1}}) = r_D$, $\mu(R_{i_k}) = t_{i_k j_k}$ $(1 \le k \le m)$, and $\mu(R_i) = \perp$ for any R_i that is not among $R_{i_1}, \ldots, R_{i_m}$. Note that all the edges that enter R_{i_k} and all the edges that enter $t_{i_k j_k}$ $(1 \le k \le m)$ have the same label l_{i_k}. Moreover, $t_{i_1 j_1}, \ldots, t_{i_m j_m}$ are connected and join consistent. Thus, μ is a weak matching. By the construction of μ, we have that $u = \bowtie(\mu)$. This proves the proposition. $\square$

The full disjunction $\mathcal{F}$ of $r_1, \ldots, r_2$ can be computed as follows. First, compute the set $\mathcal{M}_w(P_i, D)$ for $i = 1, 2, \ldots, n$. Let $\mathcal{M}$ be obtained from $\cup_{i=1}^{n} \mathcal{M}_w(P_i, D)$ by removing subsumed[2] matchings. $\mathcal{F} = \{\bowtie(\mu) \mid \mu \in \mathcal{M}\}$.

THEOREM 4.3. *The set $\mathcal{F}$ is the full disjunction of the relations $r_1, \ldots, r_n$ and it can be computed in $O(n^5 s^2 f^2)$ time, where n is the number of relations, s is the total size of all the relations and f is the size of $\mathcal{F}$.*

[2]Technically, $\cup_{i=1}^{n} \mathcal{M}_w(P_i, D)$ has matchings of all the P_i, but all these matchings are for the same set of variables (ignoring the root) and therefore, subsumptions among them are well defined.

PROOF. From Proposition 4.2, it follows that $\mathcal{F}$ consists of integrated tuples. To complete the proof that $\mathcal{F}$ is the full disjunction, it should be observed that if μ_1 and μ_2 are two assignments of P_i w.r.t. D, such that $\mu_1 \sqsubseteq \mu_2$, then $\bowtie(\mu_1) \sqsubseteq \bowtie(\mu_2)$. Therefore, it is sufficient to compute only the maximal weak matchings in order to generate all the maximal integrated tuples.

In order to show that the running time of the algorithm is polynomial in the size of the input and the output, it is necessary to show that $\mathcal{F}$ has only maximal integrated tuples. We will derive a contradiction by assuming that there are two distinct integrated tuples u_1 and u_2 in $\mathcal{F}$, such that $u_1 \sqsubseteq u_2$. Let μ_1 and μ_2 be two maximal weak matchings of P_i and P_j, respectively, such that $u_1 = \bowtie(\mu_1)$ and $u_2 = \bowtie(\mu_2)$. Essentially, it can be shown that there is a weak matching μ of P_j w.r.t. D, such that $u_2 = \bowtie(\mu)$ and μ subsumes both μ_1 and μ_2. This contradicts the fact that μ_1 and μ_2 are maximal in $\mathcal{M}$.

The size of each pattern P_i is $O(n^2)$ and the size of the database D that is constructed from the relations is s^2. The size of each $\mathcal{M}_w(P_i, D)$ is no more than the size of $\mathcal{F}$. In general, the size of the product graph is $O(pd)$, where p is the size of the pattern and d is the size of the database. However, in the computation of the full disjunction, for each node t_{ij} of D, the product graph $P_k \times D$ $(1 \le k \le n)$ has at most one node $(R_i, t_{ij}) \in P_k \times D$ that is reachable from the root. Consequently, the size of the portion of $P_k \times D$ that is reachable from the root is $O(d)$. Therefore, the size of the stratum traversal is $O(pd)$ and the time to compute each $\mathcal{M}_w(P_i, D)$ is $O(n^4 s^2 f^2)$. Hence, $\mathcal{M}$ is computed in $O(n^5 s^2 f^2)$ time. Removing subsumed matchings from $\cup_{i=1}^{n} \mathcal{M}_w(P_i, D)$ takes $O(n^3 f^2)$ time. The final step of constructing $\mathcal{F}$ from $\mathcal{M}$ takes $O(f)$ time. Thus, the running time is $O(n^5 s^2 f^2)$. $\square$

4.3 Projections and Restrictions of Full Disjunctions

One way of generalizing the result of Theorem 4.3 is by considering the complexity of evaluating a projection of the full disjunction. Formally, the *projection problem* is the problem of computing the projection of the full disjunction on a given set of attributes X.

In some cases, it might be desirable to compute only those tuples of the full disjunction that are non-null on all the attributes of a given set X. Formally, the *restriction problem* is the problem of computing just those tuples of the full disjunction that are non-null on X. The following theorem shows that the restriction problem cannot be computed in polynomial time in the size of the input and the output, even if the relation schemes are α-acyclic and X has only two attributes.

THEOREM 4.4. *Let $r_1, \ldots, r_n$ be relations with α-acyclic relation schemes and let X be a set of two attributes. Deciding whether the full disjunction has a tuple that is non-null on all the attributes of X is NP-complete.*

PROOF. NP-hardness is shown by a reduction from the Hamiltonian-path problem. Let $G = (V, E)$ be a given directed graph, and let s and t be two nodes in G. The problem is to decide whether G has a directed path from s to t that goes through all the nodes of G, visiting each node exactly once.

Suppose that G has k nodes, denoted $n_1, \ldots, n_k$, and m edges. We assume that $s = n_1$ and $t = n_k$. We construct the following $m + 2$ relations:

1. A relation r_s that has the relation scheme $S(A, N_1)$ and contains the single tuple $(1, 1)$.
2. A relation r_t that has the relation scheme $T(N_k, B)$ and contains the single tuple (k, k).
3. For each edge $e_l = (n_i, n_j)$ $(1 \leq l \leq m)$, we construct a relation r_l that has the relation scheme $E_l(N_i, N_j)$ and contains the $k - 1$ tuples: $(1, 2), (2, 3), \ldots, (k - 1, k)$.

In addition to the above $m + 2$ relations, we also construct an empty relation r_α that has the set of all the attributes as its relation scheme. The sole purpose of r_α is to ensure that the relation schemes of the $m + 3$ relations are α-acyclic.

Let $\mathcal{F}$ denote the full disjunction of the $m + 3$ relations and let q be the projection of $\mathcal{F}$ on the attributes A and B.

It is easy to see that only the tuples $(1, k)$, $(1, \perp)$ and $(\perp, k)$ may appear in q. Furthermore, there is a Hamiltonian path in G from s to t if and only if the tuple $(1, k)$ is in the result of q. This shows NP-hardness.

To witness membership in NP, note that we can guess an assignment of values and nulls to all the attributes of all the relation schemes, such that the attributes of X are assigned non-null values. We can then verify in polynomial time that the guess creates an integrated tuple. $\square$

Essentially, the reduction that is used in the proof of the above theorem also shows that the projection problem cannot be computed in polynomial time under input-output complexity if P $\neq$ NP.

It is easy to see that the projection problem and the restriction problem have polynomial-time algorithms if each relation scheme either contains X or is disjoint from X. Note that, in particular, this condition is satisfied if X is a singleton.

Rajaraman and Ullman [23] showed that the full disjunction of $r_1, \ldots, r_n$ can be evaluated by a natural outerjoin sequence if and only if the relation schemes are connected and γ-acyclic. Their result can be generalized as follows.

THEOREM 4.5. *The projection problem and the restriction problem have polynomial-time algorithms under input-output complexity if the relation schemes are γ-acyclic.*

5. GENERALIZING FULL DISJUNCTIONS

Full disjunction are based either on equijoins [11] or natural joins [23]. Quite frequently, however, the process of integrating information involves more general conditions than merely equality of attributes. Consider the scenario of piecing together information about people, in the absence of a unique ID for each person. Joining two records, simply because both carry the same name, might be error prone. A safer approach could be to join two records if the names are the same or similar (e.g., one has a middle initial and the other does not) and either the address or one of the phone numbers (e.g., home, office or mobile phone) are the same. A similar scenario is joining two records, such that one has a complete address while the other only contains the city and state. In this case the appropriate condition is that the city and state from the second record appear in the address of the first record.

The approach of Section 4 can be easily generalized to the above scenarios. We use the same patterns as in Section 4 and use the same database, but the edges should reflect the new conditions. Thus, there are edges in both directions between R_i and R_j if some condition is specified for this pair of relation schemes. Similarly, there are edges in both directions between the tuples $t_{i_1 j_1}$ and $t_{i_2 j_2}$ if these two tuples satisfy the condition that is specified for their relation schemes, R_{i_1} and R_{i_2}.

Conditions of general types could be used for joining tuples. Essentially, we may assume that queries are formulated using a SELECT-FROM-WHERE clause and the WHERE clause is a conjunction of conditions $C_1 \wedge \ldots \wedge C_k$, where each C_h involves either one or two relations. If C_h involves just one relation r_i, then it is a selection that can be applied to r_i before starting the evaluation of the full disjunction. If C_h involves two relations r_i and r_j, then it may be any condition that can be evaluated for pairs of tuples from these two relations.

Thus, weak semantics provides the means to generalize full disjunctions and thereby facilitating integration of information from the Web, as well as from other heterogeneous sources, in a more general manner than earlier work.

6. RELATED WORK AND CONCLUSIONS

The main issues in incomplete information are representation of missing information and query evaluation. In logical databases, one could use either a proof-theoretic approach [24] or a model-theoretic approach [29]. If there is some partial knowledge about the missing information, then it could be represented in the form of conditions that restrict the possible values of unknown data items [14]. The complexity of query evaluation in these approaches is investigated in [1, 14, 29].

The outerjoin [6, 7, 17] was introduced in order to preserve tuples that are lost during an ordinary join. Since outerjoins are not associative, it is rather difficult to formulate queries that use them. Moreover, the lack of associativity restricts the ability to optimize queries with outerjoins. Efficient evaluation methods for outerjoin queries is the focus of several papers [4, 5, 10, 25].

Full disjunctions were proposed by Galiando-Legaria [11] as an alternative to outerjoins, since full disjunctions are commutative and associative. Full disjunctions bear some similarity to the weak instances that were investigated extensively in the framework of the universal-relation model [12, 13, 26, 27]. In both approaches, all the tuples of the base relations are preserved. In full disjunctions, tuples are joined together whenever possible. The joins should be connected in order to avoid combinations of tuples that are evidently not related. In weak instances, tuples are joined together when it is implied by the dependencies and queries can be evaluated by taking the union of all lossless joins [26, 27].

Different notions of acyclicity for hypergraphs were investigated in [8], including α-acyclicity and γ-acyclicity. Properties of acyclic database schemes were investigated in [3]. (Note that a set of relation schemes can be represented as a hypergraph by viewing the attributes as nodes and viewing each relation scheme as a hyperedge.) Acyclicity leads to another form of query evaluation over universal relations [9].

Rajaraman and Ullman [23] enunciated the importance of full disjunctions in the context of integrating information from the Web. They showed how full disjunction could be used to find maximal join-consistent connections in trees of the OEM model of [21, 22]. Their main result is that a full disjunction can be evaluated by a natural outerjoin sequence if and only if the relation schemes are connected and γ-acyclic. Their algorithm was implemented in the Information Manifold System [18, 19] that integrates information from the Web.

For α-acyclic relation schemes, Yannakakis [30] showed that any projection of the natural join of all the relations can be computed in polynomial time in the size of the input and the output. Theorem 4.5 builds on the results of [23, 30]. We have also shown that when the relation schemes are α-acyclic, there are no polynomial-time algorithms, under input-output complexity, for computing projections and restrictions of full disjunctions.

This paper follows the work of [15, 16] on queries with incomplete answers over semistructured data. In [15, 16], three semantics, AND, OR and weak, were presented and the complexity of query evaluation was investigated. In particular, it was shown that for DAG queries, both the OR-semantics and the weak semantics have polynomial-time evaluation algorithms under input-output complexity. In this paper, we have shown that this result carries over to cyclic queries. We have also shown that evaluation of full disjunctions is reducible to evaluation of queries under the weak semantics. Thus, full disjunctions can be computed in polynomial time in the size of the input and the output. Previously, the only known algorithm to compute full disjunctions in the general case was by evaluating all connected joins and removing subsumed tuples. That algorithm runs in exponential time in the size of the input and the output.

The weak semantics provides a substantial generalization of full disjunctions, by allowing for general types of join conditions (provided that those conditions can be computed efficiently). This is achieved without increasing the time complexity of evaluating full disjunctions, in stark contrast to other approaches for querying incomplete information, where complex join conditions quickly lead to intractability of query evaluation.

The OR-semantics can provide another, more general form of full disjunctions. In the OR-semantics, an answer should be connected, but it does not have to satisfy all the join conditions among its parts; instead, it is sufficient to satisfy only a subset of the conditions, provided that the satisfied join conditions are connected. In traditional databases, this would not be an answer at all, but in the realm of the Web this might still be of interest to the user, although it would probably be ranked lower than answers that are obtained under the weak semantics.

A different approach for querying XML with incomplete information is presented in [2]. Their goal is to split a query into two parts, such that one part can be evaluated from answers to previous queries whereas the second part requires additional querying of the information sources. The approach of [2] is not applicable to integration of information from heterogeneous sources, such as those found on the Web, since they assume that the information sources provide persistent node id's and their query language does not have data joins.

Future work should address the issue of optimizing the evaluation of full disjunctions, in particular, and the evaluation of queries under the OR-semantics and the weak semantics, in general.

7. REFERENCES

[1] S. Abiteboul, P.Kanellakis, and G. Grahne. On the representation and querying sets of possible worlds. *Theoretical Computer Science*, 78(1):158–187, 1991.

[2] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. In *Proceedings of the 20th ACM Symposium on Principles of Databae Systems*, Santa Barbara (California, USA), May 2001.

[3] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

[4] G. Bhargava, P. Goel, and B. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *Proceedings of the of 1995 ACM SIGMOD International Conference on Management of*

Data, pages 304–315, San Jose (California, USA), 1995.

[5] A. L. P. Chen. Outer join optimization in multidatabase systems. In *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems*, pages 211–218, Dublin, (Ireland), July 1990.

[6] E. F. Codd. Extending the relational database model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[7] C. J. Date. The outer join. In *Proceedings of the 2nd International Conference on Databases*, pages 76–106, Cambridge, 1983.

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

[9] R. Fagin, O. Mendelzon, and J. D. Ullman. A simplified universal relation assumption and its properties. *ACM Transactions on Database Systems*, 7(3):343–360, 1982.

[10] C. Galiando-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–73, 1997.

[11] C. Galindo-Legaria. Outerjoins as disjunctions. In *Proceedings of the of 1994 ACM SIGMOD International Conference on Management of Data*, pages 348–358, Minneapolis (Minnesota, USA), May 1994.

[12] M. Graham and M. Yannakakis. Independent database schemas. *Journal of Computer and System Sciences*, 28(1):121–141, 1984.

[13] P. Honeyman. Testing satisfaction of functional dependencies. *Journal of the ACM*, 29(3):668–677, 1982.

[14] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31:761–791, 1984.

[15] Y. Kanza, W. Nutt, and Y. Sagiv. Queries with incomplete answers over semistructured data. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems*, pages 227–236, Philadelphia, (Pennsylvania), May 1999.

[16] Y. Kanza, W. Nutt, and Y. Sagiv. Querying incomplete information in semistructured data. *Journal of Computer and System Sciences*, 64(3):655–693, 2002.

[17] M. Lacroix and A. Pirotte. Generalized joins. *ACM SIGMOD Record*, 8(3):14–15, 1976.

[18] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Query-answering algorithms for information agents. In *Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference*, pages 40–47, Portland (Oragon), August 1996.

[19] A. Y. Levy, A. Rajaraman, and J. J. Ordille. The world wide web as a collection of views: Query processing in the information manifold. In *Workshop on Materialized Views: Techniques and Applications*, pages 43–55, Monteal, (Canada), June 1996.

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

[21] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the 11th International Conference on Data Engineering*, pages 251–260, Taipei, Mar. 1995.

[22] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: A lightweight object repository for semistructured data. In *Proceedings of the of 1996 ACM SIGMOD International Conference on Management of Data*, Montreal (Canada), June 1996.

[23] A. Rajaraman and J. D. Ullman. Integrating information by outerjoins and full disjunctions. In *Proceedings of the 15th ACM Symposium on Principles of Database Systems*, Montreal, (Canada), June 1996.

[24] R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *Journal of the ACM*, 33(2):349–370, 1986.

[25] A. Rosenthal and D. Reiner. Extending the algebric framework of query processing to handle outerjoins. In *Proceedings of the 10th International Conference on Very Large Data Bases*, pages 334–343, Singapore, 1984.

[26] Y. Sagiv. A characterization of globally consistent databases and their correct access paths. *ACM Transactions on Database Systems*, 8(2):266–286, 1983.

[27] Y. Sagiv. Evaluation of queries in independent database schemes. *Journal of the ACM*, 38(1):120–161, 1991.

[28] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 137–146, San Francisco (California, USA), May 1982.

[29] M. Y. Vardi. On the integrity of databases with incomplete information. In *Proceedings of the 5th ACM Symposium on Principles of Database Systems*, pages 252–266, Cambridge (Massachusetts, USA), 1986.

[30] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 82–94, Cannes (France), September 1981.