Paper Number 311

Query Rewriting using Semistructured Views*

Yannis Papakonstantinou†
Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093
yannis@cs.ucsd.edu

Vasilis Vassalos‡
Computer Science Department
Stanford University
Stanford, CA 94305
vassalos@cs.stanford.edu

Abstract

We address the problem of query rewriting for MSL, a semistructured language developed at Stanford in the TSIMMIS project for information integration. We develop and present an algorithm that, given a semistructured query q and a set of semistructured views $\mathcal{V}$, finds *rewriting* queries, i.e., queries that access the views and produce the same result as q. Our algorithm is based on appropriately generalizing *containment mappings*, the *chase*, and *unification* – techniques which were developed for structured, relational data. At the same time we develop an algorithm for equivalence checking of MSL queries.

We show that the rewriting algorithm is sound and complete, i.e., it always finds every conjunctive MSL rewriting query of q, and we discuss its complexity. We currently incorporate the algorithm in the TSIMMIS system.

1 Introduction

Recently, many semistructured data models, query and view definition languages have been proposed [GM+97, MAG+97, BDHS96, AV97a, MM97, KS95, PGMU96, PAGM96, AGM+97, Suc96, FFLS97] and are used for querying and management of Web data [FFLS97, AV97a, MM97, KS95], biological databases [TMD92], integration of heterogeneous data [GM+97, LYV+98], etc. As it has been the case in the relational world, rewriting of semistructured queries using views is a fundamental query processing and optimization tool for semistructured queries. In this section we first present an abstract version of the rewriting problem and consequently we describe its applications including a rewriter we build for the TSIMMIS system [GM+97].

*Contact author: Vasilis Vassalos. Tel. (650) 723 0587

†Research partially sponsored by NSF, under Award Number IRI-9712239, and the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339.

‡Research partially supported by NSF grant IRI–96–31952, ARO grant DAAH04–95–1–0192, Air Force contract F33615–93–1–1339 and the L. Voudouri Foundation.

The Rewriting Problem At a sufficient level of abstraction the problem faced by the applications listed below is as follows: Given a query q accessing a semistructured database D[1] and a set of views $\mathcal{V} = \{V_1, \ldots, V_n\}$ over D find *rewriting* queries, where a rewriting query of q given $\mathcal{V}$ is a query that accesses at least one view of $\mathcal{V}$ and returns the same result as q.[2] If the rewriting query uses views only (i.e., it does not access directly the database D) then it is called a *total rewriting* query.

Applications of the Rewriting Algorithm Semistructured models have been used by repositories that store semistructured data [MAG+97, BDHS96] and by mediators that integrate heterogeneous information [Wie93, LSS96, PGMW95, FFLS97, KS95, MMM96].

The importance of rewriting algorithms in mediators and repositories of relational systems, as described below, is a witness to the many applications they'll have in the semistructured world.

1. Relational query rewriting algorithms are used for answering queries using materialized views [LMSS95, LY85] and the query cache [KB96].

2. Views have been used in mediator systems to describe the source contents [LRO96]. Furthermore, the different and limited query capabilities of the sources are often described by "views" where the constants are parameterized. For example, the parameterized view

   `SELECT * FROM R WHERE R.A=$X` where `R` resides at source S declares that S can answer queries that pick all attributes of `R` and have `R.A` be bound to a constant. Then a query over the source data has to be rewritten to use correctly the contents and capabilities of the sources, i.e., to correctly use the available views [RSU95, LRU96, VP97, VP, HKWY97, PGH96, DG97, LRO96]. Indeed, in that case the query has to access *only* views and hence we need a total rewriting query.

The above points highlight the importance of rewriting algorithms in relational databases and mediators. We believe that rewriting algorithms will be equally important for semistructured databases and mediators.

Use of the Rewriting Algorithm in the TSIMMIS System: Capabilities Based Rewriting and Cached Queries A TSIMMIS mediator integrates semistructured data from multiple heterogeneous information sources into a virtual view V_m — not to be confused with the views used by the rewriting algorithm. The general integration architecture is shown in Figure 1.

For example, a bibliographic mediator may combine the data of multiple bibliographic sources into a single "union" view. These bibliographic sources are accessible through interfaces that have varying query capabilities. At run time, given a user query, the mediator decomposes it into multiple queries which refer to the source data. A more detailed description of the mediator architecture is shown in Figure 2. For example, if the user query requests all "SIGMOD 97" publications, the

[1]The database may be distributed over multiple sites.

[2]We formalize the concept of "same result" and the definition of a rewriting query in Section 3.

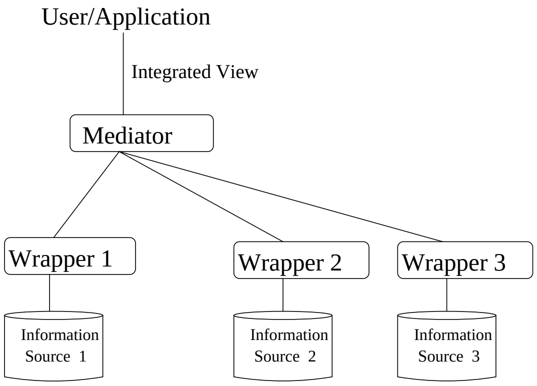

Figure 1: The TSIMMIS integration architecture

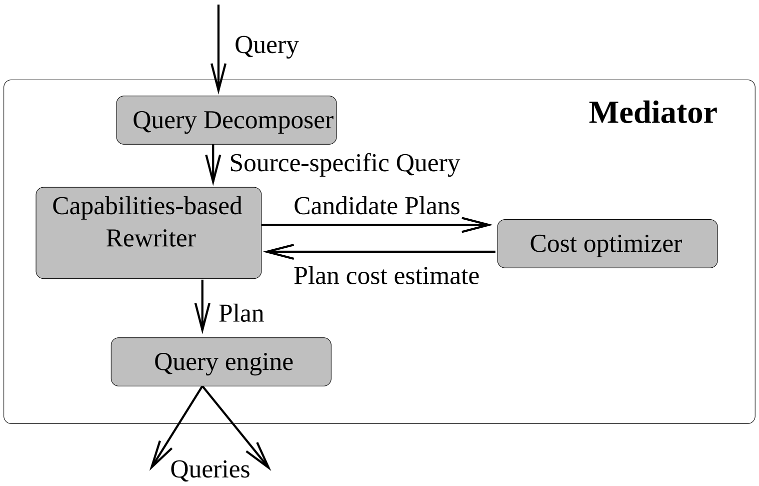

Figure 2: Mediator architecture

mediator will decompose the user query into multiple "SIGMOD 97" queries where each one of them refers to one source only. To do that correctly and efficiently the mediator must figure out, using the capabilities of the underlying sources, how to extract the necessary information from the sources. This decision is made in the Capabilities Based Rewriter module shown in Figure 2. After the decision is made, and the mediator formulates a query plan that respects the query capabilities of the sources, each query is sent to a wrapper, where it is translated into the native query language of the corresponding source. Then the query results are collected and appropriately combined by the mediator.

The TSIMMIS system uses parametrized views to describe query capabilities. The mediator employs a simplified limited version of our rewriting algorithm to accomplish its task [LYV+98]. We are in the process of implementing the fully general MSL query rewriting algorithm presented in this paper in the TSIMMIS system. The existence of parameters in the views does not seriously affect the complexity of the problem [RSU95, VP].

Moreover, our rewriting algorithm can enhance the system's performance and availability by using cached results of previous queries to answer the current query. For example, if a cached query result contains all "SIGMOD" publications our rewriting algorithm can create a rewriting query where "SIGMOD 97" publications are obtained by filtering the cached query for "1997' publications. Notice that the rewriting algorithm only needs the query and the cached query statements - it does not need to examine the source data. The cached queries play in this case the role of views.[3]

Use of the Rewriting Algorithm in Lore: Our algorithm can be used to answer queries using materialized views defined over repositories that store semistructured data, such as Lore [MAG+97]. This was the main original motivation for relational query rewriting [YL87, CKPS95] and we believe it is as important for semistructured databases.

Data Model and Query and View Definition Language In TSIMMIS we model semistructured data using the OEM data model [PGMW95, QRS+95] which is a labeled graph model where the nodes are uniquely identified by object-id's. We express queries and views using MSL [PGMU96, PAGM96], a logic-based language with path expressions. The resemblance of MSL to Datalog facilitates the presentation of similarities and differences between the rewriting of relational conjunctive queries and semistructured MSL queries.

In this paper we focus on queries and views defined by conjunctive MSL without arithmetic. Furthermore, the rewriting queries under consideration are in the same fragment of MSL. In the remainder of this paper by MSL we mean conjunctive MSL without arithmetic unless we specify otherwise.

Algorithm The described algorithm uses as input (i) a MSL query q where all conditions refer to a database D and (ii) MSL views also referring exclusively to D, and outputs a set $\mathcal{Q}$ of rewriting queries, i.e., queries equivalent to q that have at least one condition referring to one of the views, and furthermore every rewriting query q_r appears in $\mathcal{Q}$ or there is a $q_r' \in \mathcal{Q}$ that is equivalent to q_r. From now on, we will say that the algorithm returns *all* rewriting queries though we actually mean that it returns a set of rewriting queries $\mathcal{Q}$ with the above properties.

The algorithm operates in two steps: First it finds candidate rewriting queries. Then it retains the candidate rewriting queries which are equivalent to the original query. During the first phase, the algorithm uses a generalization of containment mappings [CM77] from the views to the original query q in order to narrow the space of candidate queries. Note that we extend mappings to deal with (potentially cyclic) path expressions in the body as well as the head of the query and the

[3] Of course, given the autonomy of the bibliographic sources and the mediator, the rewriting query may deliver a stale result to the user. Nevertheless this result may still be very useful to the user. For example, assuming that the sources do not delete information, the cached result contains correct answers to the user's query, and it can be quickly delivered. Then, while the user inspects the cached query, the system can collect the current result from the sources [S+]. Furthermore, if an update propagation system is in place it can account for the "deltas" between the cache and the sources [GL95, ZGMHW95, AMR+98]. In this paper we will not deal any further with these consistency issues. Instead we focus on the rewriting algorithm.

views. Furthermore, the algorithm uses the *chase* technique [Ull89] to cope with the functional dependencies that hold because of object identity.

Testing the equivalence of the candidate rewriting query with the original query is also accomplished in two steps. First, the algorithm "composes" the rewriting query and the views to obtain an "expanded" query q_r' that is equivalent to the rewriting query and does not refer to the views any more. Then q_r' is tested for equivalence with the original query. To do that, we develop conditions for the equivalence of MSL queries that extend the conditions for equivalence of unions of conjunctive queries [CM77, SY80] to queries with path expressions.

Lessons and Formal Results Our rewriting algorithm is based on extending containment mappings, the chase, and unification from the relational into the semistructured world. In doing so we benefit from a vast body of knowledge on relational query rewriting. Furthermore, we obtain insight on how to interface with the optimizer of the TSIMMIS system (see Figure 2).

The reader may wonder whether, given an appropriate reduction of semistructured data to relations, the conjunctive MSL rewriting problem can be reduced to the well-understood relational conjunctive query rewriting problem. The answer is negative because MSL queries can not be reduced into conjunctive relational queries. MSL queries and views are actually reducible into Datalog with function symbols and with a special form of recursion [Pap97]. The main factors that make rewriting of semistructured queries harder than that of relational queries are the lack of schema and the ability of MSL to construct arbitrary objects through the use of semantic object ids. In the absence of a schema the composition of an MSL condition with the head of an MSL view, which is an essential step of our rewriting algorithm, takes exponential time in the size of the view head — while it is polynomial for a relational query and view.

Contents The following section introduces the OEM data model and the MSL query language for semistructured data. Section 3 states the rewriting problem, describes our algorithm and describes mappings and query composition in MSL. Section 4 presents an algorithm for equivalence testing of MSL queries. Finally, Section 5 proves the correctness of our proposed algorithm for MSL rewriting and discusses the complexity of the rewriting problem.

2 The OEM Data Model and the MSL Query Language

In the OEM data model, the data are represented as a rooted graph with labeled nodes (also called *objects*) that have unique object-id's.

Figure 3 illustrates some bibliographic data represented in OEM. *Atomic* objects have an atomic value (e.g., `SIGMOD`) while the value of the other objects (called *set objects*) is the set of objects pointed by the outgoing edges. The roots of the graph are illustrated as top level objects. They are the starting points for querying the sources. The object-id's are typically atomic data. In the general case they are terms from the Herbrand universe composed from (i) a set of atomic data, which includes but is not necessarily confined to, the atomic data appearing as labels and values and

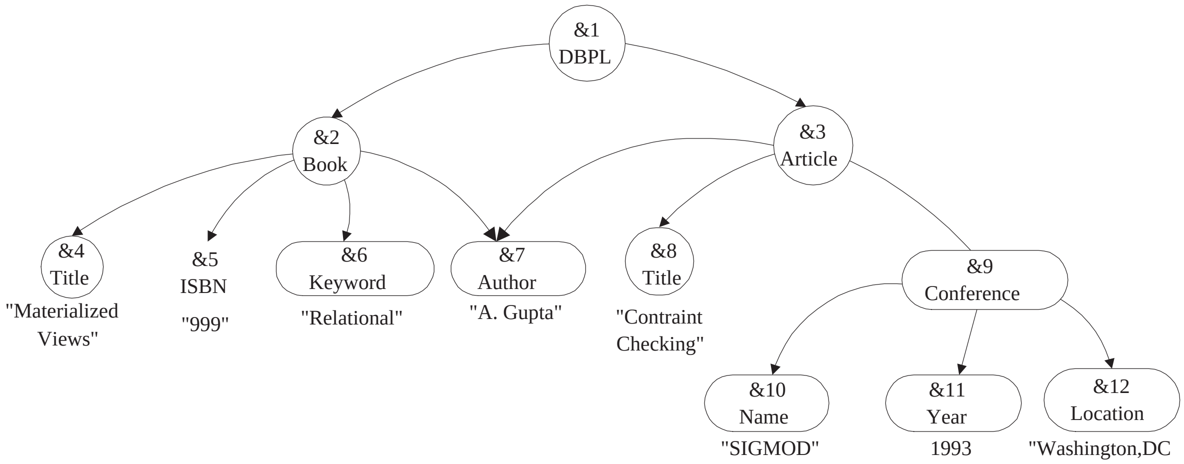

Figure 3: Example OEM objects

(ii) an arbitrary set of uninterpreted function symbols. For example, `f(&10, ashish)` is a possible object-id. Object-id's may be symbols with no particular meaning. In other cases object-id's may have a semantic meaning. For example, if the object is a Web page then it is typically a good idea to have the URL be the object-id. Furthermore, meaningful term object-id's can facilitate the integration tasks [PAGM96].

In general an MSL query consists of *rules* that define the query result using minimal model semantics. Note that in this paper we consider single rule MSL queries only. Each rule consists of a *head* followed by a `:-` and a *body*, in the style of Datalog [Ull89]. Intuitively, the head describes result objects, whereas the body describes one or more conditions that must be satisfied by the source objects. The head and the body conditions are based on *object patterns* of the form <*object-id label value*>. The *value* field can be either a term (variable, atomic constant, or function symbol followed by a term list) or a *set* value pattern which contains zero or more object patterns. We illustrate the semantics with the following example. Note that we do not include arithmetic or negation in our example since we do not consider them in this paper. See [PAGM96, Pap97] for more examples.

(Q1) `<fem(P) female {<X Y Z>}> :- <P person {<G gender female> <X Y Z>}>@db`

The semantics of the above query are

if there is a tuple of *bindings* p, g, x, y and z for the variables `P`, `G`, `X`, `Y`, and `Z` such that
the data source `db` contains a `person` top-level (root) object identified by p,
the p object has a `gender` subobject with value `female` and object id g, and
the p object has a y subobject with value z and object id x
the object p may also have sub-objects other than the g and the x

then the query result has

a `female` object, with object-id $\texttt{fem}(p)$,

the $\texttt{fem}(p)$ object has a y subobject with value z and object id x.

the object $\texttt{fem}(p)$ *may have subobjects other than* y *because the result of another rule may "fuse" more subobjects into the object* $\texttt{fem}(p)$

Note that the answer to query (Q1 is an object with a new, unique object-id and the structure denoted by the query head. In general a MSL query can construct answer objects that are arbitrary restructurings of source data. Note also that a MSL query may refer to more than one data sources, e.g., one condition may refer to `db1` and a second one to `db2`. Finally, note the following restrictions on the MSL that is used in this paper. First, we do not allow variables that appear in object-id fields of body conditions to also appear in a *label* or a *value* field of another condition[4]. Second, we only consider single rule, positive MSL queries without arithmetic. To simplify the presentation we focus on normal form queries, defined next. Every MSL query can be easily converted into normal form and hence the focus on normal form does not limit the power of the algorithm.

Definition: Normal Form MSL Queries are the MSL queries in whose body all set-valued *value* fields contain *at most* one object pattern. Additionally, a normal form query with just one condition in its body is called a *single path* query. □

The query (Q1) can be easily transformed into the following normal form query:

```
(Q2) <fem(P) female {<X Y Z>}> :- <P person {<G gender female>}>@db
                                  AND <P person {<X Y Z>}>@db
```

Safe MSL queries An MSL query is safe if every variable appearing in the query head also appears in the query body. Thus, the same simple syntactic test that is used by [Ull89] to define safety of conjunctive queries can be used to define safety in MSL. In the remainder of this paper we are only discussing safe MSL queries.

3 MSL Query Rewriting

Given a MSL query Q referring to an OEM database D and conjunctive views $\mathcal{V} = V_1, \ldots, V_n$, also referring to D, the rewriting problem is to find a MSL query Q' such that (i) Q' refers to at least one of $V_1, \ldots, V_n$ and optionally on D and (ii) for all OEM databases D, the result of Q is equivalent to the result of Q'. (See definition of equivalence below.)

We call Q' the *rewriting* query. In general, there may be more than one rewriting queries. We start our discussion with a straightforward definition of equivalence of OEM databases.

Equivalence of two OEM databases D_1 and D_2 Two OEM databases D_1 and D_2 are equivalent if they are *identical*, i.e., they have the same set of object-id's and for every object-id x the

[4]Our algorithm will work in the absence of this condition but it will not be complete.

two objects identified by x in D_1 and D_2 (i) have the same label l (ii) both of them have an atomic value or both of them have a set value (iii) if they are atomic objects they have the same atomic value v and (iv) if they are set objects they have identical sets of subobjects.

Apparently the above definition carries to equivalence of query results and views. It is possible to define OEM database equivalence up to object-id renaming. We are discussing such issues in Section 6.

3.1 Rewriting of Queries with Single Path Condition

We informally present an algorithm which decides whether a query Q having one single path condition can be rewritten using a single view V that has one or more path conditions. This algorithm, though a special case of the complete rewriting algorithm (see Section 3.3), illustrates the basic steps of our technique. The general algorithm is proven sound and complete in Section 5.

Step 1: Find Candidate Queries We first find mappings from the view to the condition and then we develop a candidate query for each mapping. Note that for the special case of queries with a single path condition there may be at most one mapping and consequently at most one candidate query. However, we let the step titles to refer to the general case.

Step 1A: Find Mappings Find, if it exists, the *mapping* from the body of V to the body of Q. Our mappings extend [CM77] to cope with object nesting and are formally defined in Section 3.2. If a mapping exists, then we can be sure that, if there is a variable binding that satisfies the body of Q, then there is also a binding that satisfies the body of V. Hence mappings are a necessary condition for the relevance of the view to the query condition. Furthermore, the mapping indicates which conditions of Q do not appear in V. Notice that there can be *at most* one mapping from the body of V to the one single path condition in the body of Q. However, in the general case (Section 3.3) we may have multiple mappings.

Example 3.1 Consider the view (V1), which restructures the `person` objects of `db` into objects that "group" their labels in `property` subobjects and their values in `value` subobjects. Notice that (V1) "loses" information in the sense that it only shows the labels and values that appear in `db` but the label-value correspondence has disappeared. Queries such as (Q3), which asks whether the value `leland_stanford` appears in the database, can be answered using the view (V1) because they do not need information on the label-value correspondence – unlike query (Q7) of Example 3.3. The example shows how our algorithm finds a rewriting query for (Q3).

```
(V1) <P' person {<prop(P',Y') property Y'> <X' value Z'>}> :-
                 <P' person {<X' Y' Z'>}>@db
(Q3) <P stanford yes> :- <P person {<X Y leland_stanford>}>@db
```

The only mapping from the body of (V1) to the body of (Q3) is (M2). Intuitively, (M2) indicates that the condition `Z' = leland_stanford` must be enforced on the view in order to get objects relevant to the query.

```
(M2) [P' ↦ P, X' ↦ X, Y' ↦ Y, Z' ↦ leland_stanford]
```

□

Step 1B: Generate Candidate Queries Apply the mapping to V, resulting in an "instantiation" of V, namely V'. Then build the rewriting query Q' as follows: The head of Q' is identical to the head of Q. The body of Q' is the head of V'.

Example 3.1 continued The only candidate rewriting query (Q4) is created from the head of (Q3) and the result of applying (M2) to the head of (V1).

```
(Q4) <P stanford yes> :- <P person {<prop(P,Y) property Y>
                                     <X value leland_stanford>}>
```

Step 2: Test Correctness of Candidate Query Check whether the composition of V and Q', denoted by $V \circ Q'$ is equivalent to Q. Step 2 is accomplished in two sub-steps:

Step 2A: Computation of Composition The composition $V \circ Q'$ of the rewriting query with the view is computed. We compute $V \circ Q'$ using the TSIMMIS query-view composition algorithm [PAGM96, Pap97], which is based on extending resolution and unification for semistructured data. This algorithm takes exponential time in the size of the query.

Step 2B: Testing Equivalence of $Q' \circ V$ with Q The general idea of equivalence testing is to find (1) a mapping that maps $V \circ Q'$ into Q, i.e., (i) it maps the head of $V \circ Q'$ into the head of Q and every condition of $V \circ Q'$ is mapped into a condition of Q and (2) a mapping that maps Q into $V \circ Q'$. Note that the $V \circ Q'$ and Q have to be in normal form in order to test equivalence as described above[5].

Example 3.1 continued We test whether (Q4) is a valid rewriting query by first transforming it in the normal form $(\mathrm{Q4})_{norm}$, then composing it with (V1), and finally comparing the resulting query $(\mathrm{V1})\circ(\mathrm{Q4})_{norm}$ to (Q3). Indeed, $(\mathrm{V1})\circ(\mathrm{Q4})_{norm}$ *is* equivalent to (Q3) because (i) the mapping (M3) maps $(\mathrm{V1})\circ(\mathrm{Q4})_{norm}$ to (Q3) and (ii) the mapping (M4) maps (Q3) to $(\mathrm{V1})\circ(\mathrm{Q4})_{norm}$).

```
(Q4)norm <P stanford yes> :- <P person {<prop(P,Y) property Y>}>
                              AND <P person {<X value leland_stanford>}>
(V1)∘(Q4)norm <P stanford yes> :- <P person {<X' Y Z'>}>
```

[5]The general equivalence testing algorithm is actually more intricate, because of the existence of object-id's. For a full description of the equivalence testing algorithm for MSL see Section 4.

```
                                          AND <P person {<X'' Y'' leland_stanford>}>
(M3) [P ↦ P, X' ↦ X, Y ↦ Y, Z' ↦ leland_stanford, X'' ↦ X, Y'' ↦ Y]
(M4) [P ↦ P, X ↦ X', Y ↦ Y'']
```

Set Mappings The rewriting query may have to apply a "subobject membership" condition on a value variable. To handle this case, our mappings are extended to map a variable to a set pattern.

Example 3.2 Consider the query (Q5) and the view (V1). It is clear that `Z'` must bind to set values which contain a `<Z last stanford>` subobject. The algorithm captures this intuition by developing the mapping (M5) from the body of (V1) to the body of (Q5). Notice that `Z'` is mapped to `{<Z last stanford>}`.

```
(Q5) <P stanford yes> :- <P person {<X Y {<Z last stanford>}>}>@db
(M5) [P' ↦ P, X' ↦ X, Y' ↦ Y, Z' ↦ {<Z last stanford>} ]
(Q6) <P stanford yes> :- <P person {<prop(P,Y) property Y>
                                    <X value {<Z last stanford>}>}>@V1
```

(Q6) is the candidate query created from the head of (Q5) and the result of applying (M5) to the head of (V1). □

Mappings are necessary but not sufficient for the existence of a rewriting query as the following example illustrates.

Example 3.3 Consider the query (Q7) and the view (V1). Intuitively there is no rewriting query for (Q7) because the view "loses" the correspondence between labels and values. Hence, if the database contains a `name` attribute and a value v containing the `<last stanford>` subobject it is impossible for the rewriting query to discover whether there is a `name` object with value v or `name` and v appear in different objects of the database. Notice that despite the non-existence of a rewriting query there is the mapping (M6). Based on this mapping the algorithm derives the candidate rewriting query (Q8). However, the composition of the candidate rewriting query with the view results in the query (Q9) which is not equivalent to the original query (Q7). Notice that `name` is the label of the object `X'` while `<last stanford>` is a subobject of another object `X''`.

```
(Q7) <P stanford yes> :- <P person {<X name {<Z last stanford>}>}>@db
(M6) [P' ↦ P, X' ↦ X, Y' ↦ name, Z' ↦ {<Z last stanford>} ]
(Q8) <P stanford yes> :- <P person {<prop(P,Y) property name>
                                    <X value {<Z last stanford>}>}>@V1
(Q9) <P stanford yes> :- <P person {<X' name Z'>}>@db
                                    AND <P person {<X'' Y'' {<Z last stanford>}>}>@db
```

□

In the following subsection, we formally define mappings and describe an algorithm for mapping composition. Subsections 3.3 and 3.5 present a general algorithm for query rewriting. Subsection 3.4 extends the chase for set variables.

3.2 Mappings

In this section, we formally define mappings and mapping composition.

Definition: A Mapping is a set of structures of the form *variable* $\mapsto$ *rhs* that transforms a MSL object pattern (or a conjunction of MSL object patterns or a MSL rule) to another. *Rhs* can be a term or a set of MSL object patterns $\{pattern_1, \ldots, pattern_n\}$. The right hand sides cannot contain variables that appear in the left hand sides of $\mapsto$. $\Box$

Definition: Valid Application of a Mapping on an OEM Object Pattern The result of applying a mapping on an OEM object pattern is a pattern where every variable which appears in the left hand side is replaced by the corresponding right hand side. The mapping is applicable to the object pattern if (i) the resulting pattern has valid OEM syntax, i.e., set patterns do not appear in object-id or label positions, (ii) is compatible with functional dependencies imposed by the object-id's. $\Box$

The composition of two mappings θ_1 and θ_2, denoted by $\theta_1 \circ \theta_2$, is derived by applying θ_2 on the variables and set patterns on the right hand side of θ_1 and subsequently concatenating them.

Definition: Mapping Composition The composition $\theta_1 \circ \theta_2$ is a mapping θ consisting of (i) all $V \mapsto rhs$ structures of θ_2 and (ii) for every $V \mapsto rhs$ structure of θ_1, θ includes a structure of the form $V \mapsto rhs'$ and $rhs' = \theta_2(rhs)$. $\Box$

The complete algorithm for discovering mappings from a set of single path conditions to another set of single path conditions appears in Appendix B.

3.3 General case of query rewriting

We now treat the general case of the query rewriting problem, with any number of views in $\mathcal{V}$ and any number of conditions in the body of the query Q. Note that object identity introduces a functional dependency (from the object id to the label and value) which needs to be taken into consideration. We extend and apply the *chase* technique to take the functional dependency into account. For the sake of simplicity, the following example uses a view set with only one view. The method used generalizes trivially to sets of views of any size; the algorithm described in subsection 3.5 covers the general case.

Example 3.4 Consider the following view (V7). Notice that the semantic object-ids of `property` and `value` objects retain information about the object that originally had that property and value. Then consider query (Q10).

```
(V7) <P' person {<prop(X') property Y'> <val(X') value Z'>}> :-
                                    <P' person {<X' Y' Z'>}>
(Q10) <P stan_student {<X Y Z>}> :- <P person {<U university stanford>}>@db
                                    AND <P person {<X Y Z>}>@db
```

Intuitively, (Q10) can be answered using (V7) as follows: First use (V7) to find the P's that have a "university" subobject with value "stanford". The mapping (M8) from the body of (V7) to the first condition of (Q10) implies that this is possible. Then for every P that qualifies, pick all its subobjects <X Y Z>. Mapping (M9) from the body of (V7) to the second condition of (Q10) implies that this is also possible. Then, the head of the rewriting query (Q11) is the head of (Q10) and the body of (Q11) is the conjunction of $\theta_8(\mathit{head}(V7))$ and $\theta_9(\mathit{head}(V7))$.

(M8) θ_8 = [P' $\mapsto$ P, X' $\mapsto$ U, Y' $\mapsto$ university, Z' $\mapsto$ stanford]
(M9) θ_9 = [P' $\mapsto$ P, X' $\mapsto$ X, Y' $\mapsto$ Y, Z' $\mapsto$ Z]
(Q11) <P stan_student {<X Y Z>}> :-
<P person {<prop(X) property Y> <val(X) value Z>}>@V_7
<P person {<prop(U) property university> <val(U) value stanford>}>@V_7

Let us now check whether (Q11) is a valid rewriting query. That means transforming (Q11) into normal form and checking whether (Q12)=(V7)$\circ$(Q11)$_{norm}$ is equivalent to (Q10).

(Q12) <P stan_student {<X Y Z>}> :-
<P person {<X Y Z'>}>@db AND <P person {<X Y' Z>}>@db
<P person {<U university Z''>}>@db AND <P person {<U Y'' stanford>}>@db

Notice that unless we make use of the functional dependency *Oid* $\rightarrow$ *Label Value* there is no mapping from the body of the query (Q10) to the body of (Q12). By *chasing* (Q12), we infer that Y $\equiv$ Y$'$, Z $\equiv$ Z$'$, Y$''$ $\equiv$ university, and Z$''$ $\equiv$ stanford. □

3.4 Extending the chase for set variables

As noted in the previous section, the chase has to be extended for the case of variables that can bind to sets. The following example motivates the need and presents our extension to chase. Notice how the "set" variable is transformed into a set pattern.

Example 3.5 Consider (Q10) and (Q13) below.

(Q13) <P stan_student V> :- <P person {<U university stanford>}>@db
AND <P person V>@db

(Q13) is equivalent to (Q10) since V is a set variable. However, our algorithm, as described so far, will erroneously not discover a rewriting query because there is no mapping from the condition of (Q10) to the second condition of (Q13). Using the functional dependencies for objects, we can infer that V is a set variable and transform (Q13) to (Q10). □

3.5 Rewriting Algorithm

The following algorithm generates a rewriting query if one exists. The query bodies are converted into *normal form* and are chased before we apply the algorithm.

Input:A MSL query Q with k single path conditions in the body
and a set of MSL views $\mathcal{V} = \{V_1, \ldots, V_n\}$.

Output: A set of rewriting queries.

Step 1A: Find the mappings θ_{i_j} from the body of each $V_i \in \mathcal{V}$ to the body of Q
using the mapping discovery algorithm of Appendix B.

Step 1B: construct candidate rewriting queries Q'

- $head(Q')$ is $head(Q)$
- $body(Q')$ is any conjunction of l conditions, $1 \leq l \leq k$,

where each condition is either a view "instantiation" $\theta_{i_j}(head(V_i))$ or a condition of Q
if the resulting query is unsafe, then continue with next candidate
chase Q'

Step 2: test whether each constructed Q' is correct. Specifically,
construct the composition of Q' with $V_1, \ldots, V_n$ $Q'(V_1, \ldots, V_n)$ (see Section 3.6
chase it
if $Q'(V_1, \ldots, V_n)$ is equivalent to Q (see Section 4) include Q' in the output
else continue with the next candidate.

Notice that the above algorithm constructs candidate queries (in Step 1B) basically at random. The efficiency of the algorithm can be substantially improved with the use of simple heuristics. A particularly effective heuristic is the following:

- keep track of which conditions of the query body each instantiated view $\theta_{i_j}(head(V_i))$ maps into. These are the conditions that are "covered" by $\theta_{i_j}(head(V_i))$.
- only construct candidate queries Q' such that the views and conditions in the body of Q' "cover" all the conditions in the body of Q.

A variation of the above heuristic is implemented in the capabilities-based rewriting module of the TSIMMIS system [LYV+98].

The next subsection formally defines the composition of MSL queries. Composition of MSL queries is performed in Step 2 of the rewriting algorithm presented in Section 3.5.

3.6 Composition of MSL queries

The *composition* of MSL queries Q and V is a query $Q_c = V \circ Q$, such that for any OEM database D, $Q_c(D) = Q(V(D))$. Query composition is accomplished by *resolving* each condition in the body of Q with the head of V *in all possible ways*, using *unification* (which generalizes [GN88, Ull89].) A unifier in MSL is defined as follows:

Definition: [PAGM96] Unifier θ from a single path condition e_1 to a general condition e_2 θ is a unifier from e_1 to e_2 if the pattern $\theta(e_1)$ *is included in* the pattern $\theta(e_2)$, as described by Definition 3.6. □

Definition: Object pattern inclusion A single path pattern e_1 is included in a pattern e_2 *if and only if*

(a) e_1 has identical object-id and label fields as e_2
(b) *if* the value field of e_1 is of the form $\{e'_1\}$
then the value field of e_2 is of the form $\{e_2^1, \ldots, e_2^m\}$ and
there is a pattern $e_2^j, j = 0, \ldots, m$ such that e'_1 is included in e_2^j.
else if the value field of e_1 is of the form $\{\}$
then the value field of e_2 is of the form $\{e_2^1, \ldots, e_2^m\}$ (m may be 0)
else e_1 and e_2 have the same value field.

□

Query composition is easily generalized to multiple queries $V_1, \ldots, V_n$. Let us look at the following detailed example:

Example 3.6 Let us consider the following query

```
(Q14) <P ans {<D m V>}> :- <P p {<A l V>}> AND <Q q {<D m V>}>
```

and two views

```
(V10) <f(A′,B) p {<A′ l V’> <B Y ‘abc’>}> :-
                  <X label1 {<A′ label2 V′> <B Y W> <C label3 T>}>
(V11) <F L E> :- <G l {<F L E>}>
```

There exist two unifiers for the first condition of (Q14) and the head of (V10):

$$\theta_1 = [\mathtt{P} \mapsto \mathtt{f(A',B)}, \mathtt{A} \mapsto \mathtt{A'}, \mathtt{V} \mapsto \mathtt{V'}]$$

$$\theta_2 = [\mathtt{P} \mapsto \mathtt{f(A',B)}, \mathtt{A} \mapsto \mathtt{B}, \mathtt{Y} \mapsto \mathtt{l}, \mathtt{V} \mapsto \mathtt{'abc'}]$$

There exists one unifier for the first condition of (Q14) and the head of (V11):

$$\theta_3 = [\mathtt{P} \mapsto \mathtt{F}, \mathtt{L} \mapsto \mathtt{p}, \mathtt{E} \mapsto \{\texttt{<A l V>}\}]$$

That means the result of resolving the first condition of (Q14) with the views gives 3 MSL queries:

```
(Q15) <f(A′,B) ans {<D m V′>}> :-  <X label1 {<A′ label2 V′> <B Y W> <C label3 T>}>
                                   AND <Q q {<D m V′>}>
(Q16) <f(A′,B) ans {<D m ’abc’>}> :-  <X label1 {<A′ label2 V′> <B l W> <C label3 T>}>
                                      AND <Q q {<D m ’abc’>}>
(Q17) <F ans {<D m V>}> :-  <G l {<F p {<A l V>}>}> AND <Q q {<D m V>}>
```

For the second condition of each one of (Q15,Q16,Q17), there exists one unifier with (V11):

$$\theta_4 = [\mathtt{Q} \mapsto \mathtt{F}, \mathtt{L} \mapsto \mathtt{q}, \mathtt{E} \mapsto \{\texttt{<D m V’>}\}]$$

$$\theta_5 = [\mathtt{Q} \mapsto \mathtt{F}, \mathtt{L} \mapsto \mathtt{q}, \mathtt{E} \mapsto \{\texttt{<D m ’abc’>}\}]$$

$$\theta_6 = [\mathtt{Q} \mapsto \mathtt{F}, \mathtt{L} \mapsto \mathtt{q}, \mathtt{E} \mapsto \{\texttt{<D m V>}\}]$$

Therefore, Q_c consists of 3 MSL rules:

```
(Q18) <f(A′,B) ans {<D m V′>}> :-  <X label1 {<A′ label2 V′> <B Y W> <C label3 T>}>
                                   AND <G l {<F q {<D m V′>}>}>
(Q19) <f(A′,B) ans {<D m 'abc'>}> :-  <X label1 {<A′ label2 V′> <B l W> <C label3 T>}>
                                      AND <G l {<F q {<D m 'abc'>}>}>
(Q20) <F ans {<D m V>}> :-  <G l {<F p {<A l V>}>}> AND <G l {<F q {<D m V>}>}>
```

□

Notice that in MSL there are multiple *mgu*s or most general unifiers. The practical consequence is that the result of $V \circ Q$, where V, Q are single rule conjunctive MSL queries, could be a *union* of conjunctive MSL queries. In other words, Q_c could consist of multiple rules. In particular, Q_c could consist of an exponential number of rules (of at most polynomial length.) This observation gives us the following theorem.

Theorem 3.7 (Composition Complexity) Query composition in MSL can take exponential time.

Notice that the order of resolving query conditions with view heads does not matter. It is obvious that query composition "implements" view dereferencing: it transforms a query that refers to the object patterns in a view head to a query that refers to the "source" objects that the view is defined over.

For the complete unification algorithm see Appendix A. For a more detailed description and proof of correctness of the unification process in MSL, see [Pap97].

4 Equivalence of MSL queries

Two queries Q_1, Q_2 are equivalent *if and only if* for all OEM databases D, their results $Q_1(D)$ and $Q_2(D)$ are equivalent. In this section, we will develop a compile-time test of equivalence of MSL queries, based on an extension of containment mappings [CM77]. We assume that the chase has already been applied to the queries.

The problem of MSL equivalence is complicated because of the restructuring capabilities of MSL: query heads construct arbitrary answer graphs and different rules can contribute different parts of the same answer graph. Hence we need to make sure that all the components of the result graph are the same. The required decomposition is in the same spirit as normal form decomposition for query bodies (see Section 2) but it has to go one step further by decomposing a MSL query into finer grain rules. In particular, we decompose a MSL query into *graph component* queries that correspond to the components of the result graph: edges, nodes and *root*, i.e., top-level objects[6]. Every MSL rule Q is decomposed into three types of finer grain rules:

- one `top` rule corresponding to the top level condition of the head of Q (this query corresponds to the *root* of the OEM graph constructed by the head of Q)

[6] Remember that OEM graphs are rooted.

- as many `member` rules as there are object-subobject relationships in the head of Q (these queries correspond to the *edges* of the OEM graph constructed by the head of Q, and specify their start and end objects) and

- one `object` type rules as object conditions in the query head of Q (corresponding to the *objects* of the OEM graph constructed by the head of Q and describing their labels and values).

The decomposition is illustrated by the following example. The reduced rules are essentially MSL: set values are allowed in the object "predicates". Indeed it is the possibility of sets including objects of unbounded depth that prevents conjunctive MSL from being reducible to non-recursive Datalog. Note that `member` and `top` "predicates" depart from MSL syntax to emphasize the connection to Datalog [Pap97].

Example 4.1 Consider the following query:

```
(Q21) <l(X) l {<Y m {<n(Z) n V>}>}> :- <X a {<Y b {<Z c V>}>}>
```

Its decomposition in graph component queries is as follows:

```
top(l(X)) :- <X a {<Y b {<Z c V>}>}>
member(l(X),Y) :- <X a {<Y b {<Z c V>}>}>
member(Y,n(Z)) :- <X a {<Y b {<Z c V>}>}>
<l(X) l {}> :- <X a {<Y b {<Z c V>}>}>
<Y m {}> :- <X a {<Y b {<Z c V>}>}>
<n(Z) n V> :- <X a {<Y b {<Z c V>}>}>
```

□

For an example that illustrates why normal form decomposition of query heads is *not* enough to determine equivalence, see Section 4.1. The condition for equivalence of the resulting graph component queries is easily derived:

Theorem 4.2 Two sets $S_1 = \{P_1, \ldots, P_n\}$ and $S_2 = \{T_1, \ldots, T_m\}$ of graph component queries are equivalent if and only if for each P_i there exists a *mapping* to it from some T_j and for each T_i there exists a mapping to it from some P_j. Mappings include the heads of the component queries.

The proof of Theorem 4.2 is a straightforward generalization of the containment theorem for unions of relational conjunctive queries. Moreover, the following theorem holds:

Theorem 4.3 (MSL query equivalence) Two MSL queries are equivalent if and only if their decompositions into graph component queries are equivalent.

From the above, it is straightforward to derive a simple equivalence test for MSL queries.

4.1 Normal form decomposition is not enough for MSL equivalence

We will demonstrate through an example why it is necessary to decompose MSL queries to their graph component queries in order for our simple equivalence test checking to work correctly. As explained in Section 4, if two single-rule queries Q_1 and Q_2 then $Q_1 \equiv Q_2$ only if there is a mapping from Q_2 to Q_1 and vice versa. As explained in Section 4, we need to strengthen the equivalence condition to address the issues introduces by the existence of object-id's and the restructuring capabilities of MSL queries. Our generalization is based on normalization of the query heads. However, normalizing query heads into single path conditions is not sufficient, as the following example shows.

Example 4.4 Consider the following queries; each one consists of two rules and in order to make our point clear we have let all the queries have the same body. Notice that the first rule of query (Q21) creates `l` objects for every `a` object and `n'` for every `c`, while the second rule creates an `l'` for every `a` and an `n` for every `c`. Then, notice that query (Q22), though it has different single path conditions than query (Q21), creates exactly the same result. The intuition is that the different query heads create different "parts" of the same answer graph for (Q22) and (Q23).

```
(Q22) <l(X) l {<Y m {<n'(Z) n' V>}>}> :- <X a {<Y b {<Z c V>}>}>
      <l'(X) l' {<Y m {<n(Z) n V>}>}> :- <X a {<Y b {<Z c V>}>}>
(Q23) <l(X) l {<Y m {<n(Z) n V>}>}> :- <X a {<Y b {<Z c V>}>}>
      <l'(X) l' {<Y m {<n'(Z) n' V>}>}> :- <X a {<Y b {<Z c V>}>}>
```

□

To solve problems such as those posed by the above example we have to further normalize MSL rules, as explained in Section 4. The following rules are the graph component queries of (Q22).

```
top(l(X)) :- <X a {<Y b {<Z c V>}>}>
member(l(X),Y) :- <X a {<Y b {<Z c V>}>}>
member(Y,n'(Z)) :- <X a {<Y b {<Z c V>}>}>
<l(X) l {}> :- <X a {<Y b {<Z c V>}>}>
<Y m {}> :- <X a {<Y b {<Z c V>}>}>
<n'(Z) n' V> :- <X a {<Y b {<Z c V>}>}>
```

5 Soundness, Completeness, and Complexity

In the previous section we have shown that query rewriting can be done in two steps. In the first step, we find mappings from the body of the views to the body of the query and we use "instantiated" view heads to construct candidate rewriting queries. In the second step we check the correctness of the rewriting. The second step establishes the soundness of our rewriting algorithm. To prove the completeness of the algorithm, we first observe that if there is no mapping from a view body to the query body, then the view is not "relevant" to the query.

Lemma 5.1 Let Q and V be MSL queries. There is a rewriting query Q' of Q using view V only if there is a mapping from the body of V to the body of Q.

Moreover, we can bound both the number of conditions and the variables appearing in the rewriting.

Lemma 5.2 Let Q be a MSL query and $\mathcal{V}$ be a set of MSL views. If there exists a rewriting of Q using $\mathcal{V}$, then there exists such a rewriting consisting of at most k view heads, where k is the number of *single path* conditions in the body of the query.[7]

Lemma 5.3 If there exists a rewriting of query Q using the set of views $\mathcal{V}$, then there exists a rewriting of Q using $\mathcal{V}$ that doesn't use variables that don't exist in Q.

The above lemmata demonstrate that the theory of relational query rewriting, presented in [LMSS95], can be generalized for MSL. Notice that the object-id functional dependencies do not create a problem in Lemmas 5.2 and 5.3. In particular, our algorithm would not be complete if there are FDs other than the FD of value and label on object-id. This FD is handled by our generalized chase but, as [DL97] has showed, the chase can not always guarantee the completeness of rewriting. Our algorithm is complete, since we disallow the existence of extra FDs, which in the case of MSL can be created by placing a variable in the object-id and label or value field simultaneously. For example, we disallow the condition `<X Y {<Y Z W>}>`, which creates the extra FD from `X` to `Z` and `W`.

The following lemma justifies why completeness is not compromised by only constructing rewriting queries Q' that have a head identical to the head of the query Q. Notice, this is an issue that is particular to semistructured and nested models while it is trivial in the relational model (Q' must have an identical, up to variable renaming, head to Q.)

Lemma 5.4 If there exists a valid rewriting query Q'' such that $head(Q'')$ is not the same as $head(Q)$, then there exists a valid rewriting query Q' such that $head(Q') = head(Q)$.

To see that Lemma 5.4 holds, notice that if there exists such a query Q'', then we can always apply our rewriting algorithm to it, to derive a query Q' equivalent to Q'' (and therefore to Q) whose head is identical to the head of Q.

Theorem 5.5 The rewriting algorithm proposed in subsection 3.5 is sound and complete.

Proof: *(Sketch)* It is obviously sound, because the last step of the algorithm is a correctness test. It is complete because of the above lemmata, because the query composition algorithm is correct [Pap97], and finally because the rewriting algorithm exhaustively searches the space of rewritings defined by the above lemmata. □

[7] Notice that, since view heads do not have to be single path, the number of single paths in the rewriting *can* be greater than k.

5.1 Complexity of MSL rewriting

The algorithm described in Section 3.3 takes exponential time. First, *Step 1* can generate an exponential in the size of the view bodies number of mappings. Then *Step 2* can generate an exponential number of candidate rewritings. Finally the construction of $Q'(V_1, \ldots, V_n)$ using a generic query composition algorithm, i.e., an algorithm which can compose two arbitrary queries Q_1 and Q_2, takes exponential time[8].

6 Related work

The problem of query rewriting for conjunctive relational views is discussed in [LMSS95, Qia96, LRU96, VP97, DL97, VP] and for recursive queries (but not recursive views) in [DG97]. It is also related to the problems of query containment and query equivalence [CM77, CV92].

To the best of our knowledge, there is no work on the problem of rewriting semistructured queries using views. The only relevant work we are aware of is [FLS] on the problem of query containment in StruQL (a semistructured language similar to MSL). This work does not deal with the restructuring capabilities of the StruQL language, whereas our work indeed deals with the restructuring capabilities of MSL. On the other hand, [FLS] deals with queries and views containing "wildcards" and regular path expressions. Our work is restricted to conjunctive MSL without regular path expressions.

The relational rewriting work cannot offer a straightforward solution to the non-recursive MSL rewriting problem because non-recursive MSL queries reduce to (a restricted form of) *recursive*[9] Datalog programs, as described in [Pap97], hence making inapplicable the conjunctive query rewriting results. The special form of the restricted recursion leads to decidability and complexity results which are known not to hold for arbitrary recursive Datalog programs [DG97].

Since our data model supports object oriented features, our work is relevant to the problem of object oriented query rewriting. Previous work on the problem of containment and equivalence of object oriented queries [Cha92, LR96] relies on the existence of a static class hierarchy.[10] Work on the containment of queries on complex objects has been presented most recently in [LS97]. However, [LS97] considers the problem of query containment and equivalence, while we are solving the query rewriting problem. Furthermore, the language used in [LS97] is typed – as opposed to MSL which is semistructured. Finally, Abiteboul and Vianu in [AV97b] address the problem of containment for regular path conditions in the presence of path constraints.

Isomorphism In the OEM data model every node of the semistructured graph has an object identity — unlike [BDHS96] and [LS97]. Furthermore, we require that the original and the rewritten queries compute identical graphs (i.e., same OID's) as opposed to graphs equivalent under

[8] See also Section 3.6.

[9] A restricted form of recursion is needed to deal with the lack of schema and the unbounded nesting of the semistructured objects.

[10] Notice the difference with the unbounded nesting depth of the semistructured objects.

bisimulation [BDHS96] or bisimulation's close relative, isomorphism. Following the isomorphism approach two OEM databases D_1 and D_2 would be equivalent if for every object x_1 of D_1 we can find an object x_2 of D_2 such that x_1 and x_2 have the same label, same value if atomic, or equivalent (i.e. isomorphic) sets of subobjects if they have set values. The isomorphism approach is based on the fact that typically we do not care for the object-id symbol; we only care for the object subobject relationships the object-id's create. For example, the URL names are not important; it is the hypertext structure created by the links that is important.

We believe that from the point of view of the rewriting algorithm it is not important whether the rewriting query Q' produces results identical to the original query Q or it produces isomorphic results. The reason is that we conjecture that if there is no rewriting query Q' with a result identical to Q then there is no rewriting query Q'' returning a result isomorphic to Q.

7 Conclusions and Future Work

We presented an algorithm that given a semistructured query q expressed in conjunctive MSL and a set of semistructured views $\mathcal{V}$, finds *rewriting* queries, i.e., queries that access the views and are equivalent to q. Our algorithm is based on appropriately generalizing *containment mappings*, the *chase*, and *unification*. The first step uses containment mappings to produce candidate rewriting queries. The second step composes each candidate rewriting query with the views and checks whether the composition is equivalent to the original query. Though the algorithm is similar to the one for the rewriting of conjunctive queries there are many challenges stemming from the semistructured nature of the data and the queries. For example, the composition of the rewriting query and the views is harder (from a complexity point of view) because of the lack of schema and of the rich restructuring capabilities of MSL views.

We currently incorporate our algorithm into the TSIMMIS system for use as a capabilities based rewriter. We will soon adapt its interfaces to the TSIMMIS system so that it will be able to serve as a rewriter of queries using cached views. Furthermore we plan to enhance the class of MSL queries we can handle by including queries that contain regular path expressions.

References

[AGM+97] Serge Abiteboul, Roy Goldman, Jason McHugh, Vasilis Vassalos, and Yue Zhuge. Views for semistructured data. In *SIGMOD Workshop on Management of Semistructured Data*, pages 83–90, Tuscon, Arizona, May 1997.

[AMR+98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data, 1998. Stanford University Technical Report.

[AV97a] S. Abiteboul and V. Vianu. Queries and computation on the Web. In *Proc. ICDT Conf.*, 1997.

[AV97b] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proc. PODS Conf.*, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. ACM SIGMOD*, 1996.

[Cha92] E.P. Chan. Containment and minimization of positive conjunctive queries in OODB's. In *Proc. PODS Conf.*, 1992.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. ICDE*, 1995.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[CV92] S. Chaudhuri and M. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proc. PODS Conf.*, 1992.

[DG97] O. Duschka and M. Genesereth. Answering queries using recursive views. In *Proc. PODS Conf.*, 1997.

[DL97] O. Duschka and A. Levy. Recursive plans for information gathering. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[FFLS97] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language and processor for a web-site management system. In *Workshop on Management of Semistructured Data, ACM SIGMOD Conf.*, 1997.

[FLS] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. Accepted for publication, PODS 98. Available from www.research.att.com/~suciu/ under Publications.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. SIGMOD Conference*, pages 328–339, 1995.

[GM+97] H. Garcia-Molina et al. The TSIMMIS approach to mediation: data models and languages. *Journal of Intelligent Information Systems*, 8:117–132, 1997.

[GN88] M.R. Genesereth and N.J. Nillson. *Logical Foundations of Artificial Intelligence.* Morgan Cauffman, 1988.

[HKWY97] L. Haas, D. Kossman, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. VLDB*, 1997.

[KB96] A.M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5:35–47, January 1996.

[KS95] D. Konopnicki and Oded Shmueli. W3QS: A query system for the World Wide Web. In *Proc. VLDB Conf.*, pages 54–65, Zürich, Switzerland, September 1995.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS Conf.*, pages 95–104, 1995.

[LR96] A. Levy and M.-C. Rousset. CARIN: a representation language integrating rules and description logics. In *Proceedings of the European Conference on Artificial Intelligence*, Budapest, Hungary, 1996.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, pages 251–262, 1996.

[LRU96] A. Levy, A. Rajaraman, and J. Ullman. Answering queries using limited external processors. In *Proc. PODS*, pages 227–37, 1996.

[LS97] A. Levy and D. Suciu. Deciding containment for queries with complex objects. In *Proc. PODS Conf.*, 1997.

[LSS96] L. Lakshmanan, F. Sadri, and I. N. Subramanian. Schemasql - a language for interoperability in relational multi-database systems. In *Proc. VLDB Conf.*, pages 239–250, 1996.

[LY85] P.A. Larson and H.Z. Yang. Computing queries from derived relations. In *Proc. VLDB Conf.*, pages 259–69, 1985.

[LYV+98] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, and Y. Papakonstantinou. Capability based mediation in tsimmis, 1998. Accepted for demonstration, SIGMOD98.

[MAG+97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. Technical report, Stanford University Database Group, February 1997.

[MM97] A. Mendelzon and T. Milo. Formal models of the Web. In *Proc. PODS Conf.*, 1997.

[MMM96] A. Mendelzon, G. Mihaila, and T. Milo. Querying the World Wide Web. In *Proc. PDIS Conf.*, 1996.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. VLDB Conf.*, 1996.

[Pap97] Y. Papakonstantinou. Query processing in heterogeneous information sources. Technical report, Stanford University Thesis, 1997. Available from www-cse.ucsd.edu/`~yannis`/papers/.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. PDIS Conf.*, 1996.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications. In *Proc. ICDE Conf.*, pages 132–41, 1996.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. ICDE Conf.*, pages 251–60, 1995.

[Qia96] Xiaolei Qian. Query folding. In *Proc. ICDE*, pages 48–55, 1996.

[QRS+95] D. Quass, A. Rajaraman, S. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proc. DOOD*, pages 319–44, 1995.

[RSU95] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. PODS Conf.*, pages 105–112, 1995.

[S+] V.S. Subrahmanian et al. HERMES: A heterogeneous reasoning and mediator system. http://www.cs.umd.edu/projects/hermes/overview/paper.

[Suc96] D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In *Proc. 22nd VLDB Conf.*, pages 227–238, 1996.

[SY80] S. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *JACM*, 27:633–55, 1980.

[TMD92] J. Thierry-Mieg and R. Durbin. Syntactic definitions for the acedb data base manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, 1992.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies.* Computer Science Press, New York, NY, 1989.

[VP] V. Vassalos and Y. Papakonstantinou. Expressive capabilities description languages and query rewriting algorithms. Stanford Technical Report.

[VP97] V. Vassalos and Y. Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources. In *Proc. VLDB Conf.*, pages 256–266, 1997.

[Wie93] G. Wiederhold. Intelligent integration of information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 434–437, 1993.

[YL87] H. Z. Yang and P. . Larson. Query transformation for PSJ-queries. In *Proc. VLDB Conf.*, pages 245–254, 1987.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. SIGMOD Conference*, pages 316–327, 1995.

A Query composition

See Figure 4

B Mappings

The following algorithm returns the mapping, if there is one, of a single path condition c_1 to a single path condition c_2.

INPUT Two single path conditions c_1 and c_2
OUTPUT A mapping θ, if there is one, such that $\theta(c_1) \equiv c_2$
METHOD Run the function $\theta = \mathtt{map1}(c_1, c_2, [])$

function $\mathtt{map1}(c_1, c_2, \theta^i)$
  apply θ^i to $c_1.oid$ and $c_2.oid$
    %Mappings of oid's and labels are not described in detail because they are well-known
  if there is no mapping θ_{oid} of $c_1.oid$ to $c_2.oid$
    return no mapping
  else apply $\theta^i \circ \theta_{oid}$ on $c_1.label$ and $c_2.label$
  if there is no mapping θ_{label} from $c_1.label$ to $c_2.label$
    return no mapping
  else apply $\theta^i \circ \theta_{oid} \circ \theta_{label}$ on $+c_1.value$ and $c_2.value$

if $c_1.value$ and $c_2.value$ are terms and they have a mapping θ_{value}
  return $\theta^i \circ \theta_{oid} \circ \theta_{label} \circ \theta_{value}$
else if $c_2.value$ is a set of the form $\{r_1, \ldots, r_m\}$ and $c_1.value$ is a variable then
  if there is already a mapping of $c_1.value$ to a set $\{s_1, \ldots, s_n\}$
    return $\theta^i \circ \theta_{oid} \circ \theta_{label} \circ [c_1.value \mapsto \{s_1, \ldots, s_n, r_1, \ldots, r_m\}]$
  else
    return $\theta^i \circ \theta_{oid} \circ \theta_{label} \circ [c_1.value \mapsto \{r_1, \ldots, r_m\}]$
else return no mapping

Next, we describe a brute force algorithm for discovering the mappings from a set of single path conditions to another set of single path conditions.

INPUT Two sets $\{c^1_1, \ldots, c^1_n\}$ and $\{c^2_1, \ldots, c^2_m\}$ of single path conditions
OUTPUT All mappings θ such that for every condition $c^1_i, i = 1, \ldots, n$
there is a $c^2_j, j = 1, \ldots, m$ such that $\theta(c^1_i) \equiv c^2_j$
METHOD Run the function $\texttt{mapmany}(\{c^1_1, \ldots, c^1_n\}, \{c^2_1, \ldots, c^2_m\})$
function $\texttt{mapmany}(\{c^1_1, \ldots, c^1_n\}, \{c^2_1, \ldots, c^2_m\})$
  for every function f from $\{1, \ldots, n\}$ to $\{1, \ldots, m\}$ do
    $\theta_0 \leftarrow []$
    for $i = 1, \ldots, n$
      if there is not a $\theta_i = \texttt{map1}(c^1_i, c^2_{f(i)}, \theta_{i-1})$
        exit inner loop
    if $theta_n$ was found, add θ_n to mappings
  return

Notice that there may be up to m^n mappings between the two sets. The algorithm described above takes no more than exponential time.

INPUT A single path condition c, and a normal-form rule head r

OUTPUT A set S of unifiers θ such that $\theta(r)$ contains $\theta(c)$

METHOD Run the function $S =$ `unify(`c`,`r`,`$[]$`)`

function `unify(`c`,`r`,`θ`)` *returns sets of unifiers*

  apply θ to $c.oid$ and $r.oid$

  if term unification of $c.oid$ and $r.oid$ results in θ_{oid} *unify object-id's*

    apply $\theta oid \circ \theta$ to labels of c and r

  else return empty set

  if unification of $c.label$ and $r.label$ results in θ_{label} *unify labels*

    apply $\theta_{label} \circ \theta_{oid} \circ \theta$ to values of c and r

  else return empty set

  if $c.value$ and $r.value$ are terms and their term unification results in θ_{value}

if atomic objects then unify the term values

    return $\theta_{value} \circ \theta_{label} \circ \theta_{oid} \circ \theta$

  else if $c.value$ is a variable and $r.value$ is a set

    return $[c.value \mapsto r.value] \circ \theta_{label} \circ \theta_{oid} \circ \theta$

  else if $c.value$ is a set $\{c_1 \ldots c_d\}$ and $r.value$ is a variable

    if there is already definition of the form $r.value \mapsto \{s_1\ ldotss_p\}$

      return $[r.value \mapsto \{c_1, \ldots, c_d, s_1, \ldots, s_p\}]$¡$rc\theta_{label} \circ \theta_{oid} \circ \theta$

    else

      return $[r.value \mapsto \{c_1, \ldots, c_d\}] \circ \theta_{label} \circ \theta_{oid} \circ \theta$

  else if $c.value$ is the empty set and $r.value$ is a set

    return $\theta_{label} \circ \theta_{oid} \circ \theta$

  else if $c.value$ has a subobject condition c' and $r.value$ is a set

    for each subobject r_i or $r.value$

      $S_i =$ `unify(`c'`, `r_i`, `$\theta_{label} \circ \theta_{oid}\ \ circ\theta$`)`

    return the union $\cup_i S_i$ of the results of the `unify` calls above

  else return empty set

Figure 4: The unification algorithm