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
Tel. (650) 723 0587
Fax (650) 725 2588

This manuscript includes a clearly marked appendix, which has been added for the purpose of completeness.

Abstract

We consider the problem of rewriting semistructured queries using semistructured views. Given a client semistructured positive non-recursive query and a set of semistructured views, the algorithm finds a rewriting query that is equivalent to the client query and accesses only the views. Our solution is based on appropriately generalizing well understood techniques such as containment mappings and the chase. Furthermore, we develop an equivalence testing algorithm for checking the equivalence of the client query against the composition of the rewriting query with the views. We prove that our algorithm is sound and complete. Finally we investigate a specialized polynomial algorithm for composing the rewritten query with the views, based on which we can show that the semistructured query rewriting problem is NP-complete; i.e., it is no harder than the rewriting problem for conjunctive queries.

*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.

1 Introduction

Recently, many semistructured data models, query and view definition languages have been proposed [GM+97, MAG+97, BDHS96, AV97, MM97, KS95, PGMU96, PAGM96, AGM+97, Suc96].[1] A main motivation and use is the integration of heterogeneous information [Wie93, LSS96] using an "on-demand" mediator [PGMW95, FFLS97, KS95, MMM96] or a semistructured database/warehouse where semistructured materialized views are stored [MAG+97, BDHS96]. Semistructured query rewriting algorithms will significantly enhance the performance and functionality of semistructured mediators and repositories. The importance of rewriting algorithms in mediators and repositories of relational systems, as described below, is a witness to the the many applications they'll have in the semistructured world.

Materialized views In the relational world, rewriting algorithms have been developed for answering queries using materialized views [LMSS95, LY85]. Furthermore, query rewriting is used to improve the use of the query cache [KB96].

On-demand mediators The source contents are often described via views. Furthermore, the different and limited query capabilities of the sources are often described by "views" where the constants are parameterized. Then a client query over the source data sets is rewritten to correctly use the contents and capabilities of the sources, i.e., to correctly use the available views [RSU95, LRU96, VP97, VP, HKWY97, PGH96, LRO96].

We express queries and views using MSL [PGMU96, PAGM96], a logic-based language with minimal model semantics for querying the OEM semistructured data model [PGMW95]. The syntactic and semantic similarity of MSL to Datalog clarifies the similarities as well as the fundamental differences between the rewriting of relational and semistructured queries.

In this paper, we focus on queries and views that are positive, conjunctive, single-rule MSL programs without arithmetic; a generalization to positive non-recursive MSL programs without arithmetic is straightforward. We solve the rewriting problem with a novel sound and complete algorithm which extends well-understood techniques from the relational world into the semistructured one.

First the algorithm finds a containment mapping [CM77] from a set of views to the client query. Note that we extend mappings to deal with object nesting. Furthermore, the algorithm also uses the *chase* technique [Ull89] to cope with object identity issues. The chase is as well extended to deal with set-valued attributes.

Then the rewriting query is computed and our algorithm checks whether the composition of the rewriting query and the views is equivalent to the client query. In particular, we present syntactic conditions for the equivalence of MSL queries that are an extension of the conditions for equivalence of unions of conjunctive queries [CM77, SY80].

We also propose a polynomial algorithm for composing the rewritten query with the views and based on this algorithm we conjecture that the semistructured query rewriting problem is NP-complete; i.e., it is no harder than the rewriting problem for conjunctive queries.

The following section discusses related work. Section 3 introduces the OEM data model and the MSL query language for semistructured data. Section 4 formally states the rewriting problem, defines mappings and describes our algorithm. Section 5 presents an algorithm for equivalence testing of MSL queries. Finally, Section 6 proves the correctness of our proposed algorithm for

[1] By semistructured views (queries) we mean views (queries) evaluated on data conforming to a semistructured data model.

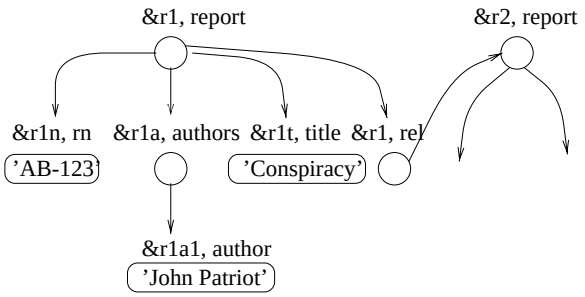

Figure 1: Example OEM objects

MSL rewriting and discusses the complexity of the rewriting problem. For the sake of completeness, in Appendix A we present query composition for MSL and in Appendix B a mapping discovery algorithm.

2 Related work

The problem of query rewriting for conjunctive relational views is discussed in [LMSS95, Qia96, LRU96, VP97, DL97, VP] and for recursive relational 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 rewriting of semistructured queries. Furthermore, the relational rewriting work cannot offer a straightforward solution to the non-recursive MSL rewriting problem because (i) non-recursive MSL queries reduce to (a restricted form of) *recursive*[2] Datalog programs, as described in [Pap97], hence making inapplicable the conjunctive query rewriting results and (ii) the special form of the restricted recursion leads to decidability and complexity results which do not hold for arbitrary recursive Datalog programs.

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.[3] 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, note that 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 bisimulation [BDHS96].

3 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 (usually starting with a `&`). Figure 1 illustrates some bibliographic data represented in OEM. *Atomic* objects have an atomic value (e.g., `Conspiracy`) while the value of the other objects (called *set objects*) is the set of objects pointed by the outgoing

[2] A restricted form of recursion is needed to deal with the unbounded nesting of the semistructured objects.

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

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 can be terms. Indeed meaningful term object-id's can facilitate the integration tasks [PAGM96].

A MSL query consists of *rules* that define the query result using minimal model semantics. 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 an atomic term (variable, constant, or term) or a *set* value pattern which contains zero or more object patterns. We illustrate the semantics with the following example. See [PAGM96, Pap97] for more information.

(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
then the query result has
  a `female` object, with object-id `fem`(p),
  the `fem`(p) object has a y subobject with value z and object id x.

Note that a MSL query may involve 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. Finally, we only use *normal form* queries.

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. □

We can easily convert non normal form queries into normal form ones. For example, 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
```

4 MSL Query Rewriting

Given a MSL query Q and views $\mathcal{V} = V_1, \ldots, V_n$, the problem is to find a query Q' such that, for all OEM databases D, the result $Q(D)$ of Q applied on D is *identical* to the result $Q'(V_1(D), \ldots, V_n(D))$ of Q' applied on $V_1(D), \ldots, V_n(D)$ (where $Q(D), V_1(D), \ldots, V_n(D)$ are of course OEM databases.) The query Q and each of the views $V_1, \ldots, V_n$ is a single, positive MSL rule without arithmetic. We call Q' the *rewriting* query. In general, there may be more than one rewriting queries.

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

4.1 Rewriting of Queries with Single Path Condition

Our first algorithm decides whether a query Q having one single path condition can be rewritten using a single view V. This algorithm, though a special case of the complete rewriting algorithm (see Section 4.3), demonstrates some key aspects of the rewriting problem in MSL and highlights important similarities and differences between rewriting of conjunctive and semistructured queries. The algorithm's steps are:

1. Find, if it exists, the *mapping* from V to Q. Our mappings extend [CM77] to cope with object nesting. They are formally defined in Section 4.2. 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. 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. Furthermore, the mapping indicates which conditions of Q do not appear in V.

2. 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'.

3. Check whether the composition of V and Q', denoted by $V \circ Q'$ is equivalent to Q. A straightforward way to perform the check is to compute the composition $V \circ Q'$ by using the resolution and unification based algorithm in [PAGM96, Pap97][5] and then use the equivalence testing algorithm presented in Section 5.

Example 4.1 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, and the query (Q3) which asks whether the value `leland_stanford` appears in the database.

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

The existence of mapping (M2) is a necessary condition for the "relevance" of view (V1) to the rewriting of (Q3). (M2) is similar to containment mappings. However, our mappings are extended to map a variable to a set pattern.

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>}>
```

We test whether (Q4) is a valid rewriting query by putting it in normal form, composing it with (V1) and comparing the resulting query (V1)$\circ$(Q4)$_{norm}$ to (Q3). Indeed, (V1)$\circ$(Q4)$_{norm}$ *is* equivalent to (Q3) because the heads are identical and there are mappings in both directions.

[5] Indeed, the query composition algorithm is implemented in the TSIMMIS mediator system.

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

□

Example 4.2 Consider query (Q5). The body of view (V1) maps to the body of (Q5) by the mapping (M3). Notice the set pattern appearing in the mapping. Notice also that despite the existence of the mapping the rewriting query (Q6) is not correct.

(Q5) <P stanford yes> :- <P person {<X name {<Z last stanford>}>}>@db
(M3) [P' $\mapsto$ P, X' $\mapsto$ X, Y' $\mapsto$ name, Z' $\mapsto$ {<Z last stanford>}]
(Q6) <P stanford yes> :- <P person {<prop(P,name) property name>
<X value {<Z last stanford>}>}>@V_1

□

As we will show in Section 6, the above procedure is sound and complete. In the following subsection, we formally define mappings and describe an algorithm for mapping composition. Subsections 4.3 and 4.5 present a general algorithm for query rewriting. Subsection 4.4 extends the chase for set variables.

4.2 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 $\mapsto$. □

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. □

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 $rhs' = \theta_2(rhs)$. □

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

4.3 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 to do the rewriting. The method used generalizes trivially to sets of views of any size; the algorithm described in subsection 4.5 covers the general case.

Example 4.3 Consider the following view (V4). 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 (Q7).

```
(V4) <P' person {<prop(X') property Y'> <val(X') value Z'>}> :-
                <P' person {<X' Y' Z'>}>
(Q7) <P stan_student {<X Y Z>}> :-
                <P person {<U university stanford>}>@db
                AND <P person {<X Y Z>}>
```

Intuitively, (Q7) can be answered using (V4) as follows: First use (V4) to find the `P`'s that have a "university" subobject with value "stanford". The mapping (M5) from the body of (V4) to the first condition of (Q7) implies that this is possible. Then for every `P` that qualifies, pick all its subobjects `<X Y Z>`. Mapping (M6) from the body of (V4) to the second condition of (Q7) implies that this is also possible. Then, the head of the rewriting query (Q8) is the head of (Q7) and the body of (Q8) is the conjunction of $\theta_5(head(V4))$ and $\theta_6(head(V4))$.

```
(M5) θ5 = [P' ↦ P, X' ↦ U, Y' ↦ university, Z' ↦ stanford]
(M6) θ6 = [P' ↦ P, X' ↦ X, Y' ↦ Y, Z' ↦ Z]
(Q8) <P stan_student {<X Y Z>}> :-
        <P person {<prop(X) property Y> <val(X) value Z>}>@V4
        <P person {<prop(U) property university> <val(U) value stanford>}>@V4
```

Let us now check whether (Q8) is a valid rewriting query. That means transforming (Q8) into normal form and checking whether (Q9)=(V4)$\circ$(Q8)$_{norm}$ is equivalent to (Q7).

```
(Q9) <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 $Oid \rightarrow LabelValue$ there is no mapping from the body of the query (Q7) to the body of (Q9). By *chasing* (Q9), we infer that $\mathtt{Y} \equiv \mathtt{Y}'$, $\mathtt{Z} \equiv \mathtt{Z}'$, $\mathtt{Y}'' \equiv \mathtt{university}$, and $\mathtt{Z}'' \equiv \mathtt{stanford}$. □

4.4 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 4.4 Consider (Q7) and (Q10) below.

```
(Q10) <P stan_student V> :- <P person {<U university stanford>}>@db
                            AND <P person V>@db
```

(Q10) is equivalent to (Q7) 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 (Q7) to the second condition of (Q10). Using the functional dependencies for objects, we can infer that `V` is a set variable and transform (Q10) to (Q7). □

4.5 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 rewriting query Q'.
Step 1: 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 2: construct candidate rewriting queries Q'
- $head(Q')$ is $head(Q)$
- $body(Q')$ is any combination of length $1 \leq l \leq k$ of $\theta_{i_j}(head(V_i))$

if the resulting query is unsafe [Ull89], then continue with next candidate
chase Q'
Step 3: test whether the constructed Q' is correct. Specifically,
construct $Q'(V_1, \ldots, V_n)$
chase it
if $Q'(V_1, \ldots, V_n)$ is equivalent to Q (see Section 5) declare success
else continue with the next candidate.

5 Equivalence of MSL queries

Two queries Q_1, Q_2 are equivalent *if and only if* for all OEM databases D, $Q_1(D)$ and $Q_2(D)$ are *identical*. In this section, we will develop a compile-time, syntactic 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. We start with the simpler case of MSL queries with *identical* heads.

Theorem 5.1 Consider two MSL queries Q and Q' which are unions of multiple rules $Q_1, \ldots, Q_n$ and $Q'_1, \ldots, Q'_n$ correspondingly. Also assume that all the rules of Q and Q' have the same head. Q is equivalent to Q' *if and only if* (i) for every rule Q_i there is mapping from the body of Q_i to the body of some $Q'_{k(i)}$ and (ii) for every rule Q'_i there is mapping from the body of Q'_i to the body of some $Q_{k'(i)}$.

The main point is that a set of variable bindings satisfying Q will also satisfy a query of Q'. Therefore, since the heads are equivalent, every object produced by Q will also be produced by Q' (and vice versa).

In the case of queries with arbitrary heads, the problem is more complicated because different rules can contribute different parts of a connected part of the graph. Hence we need to make sure that all the components of the result graph are the same. To do that, 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]. Note that `member` and `top` depart from the MSL syntax to emphasize the connection to Datalog [Pap97].

Example 5.2 Consider the following query:

(Q11) `<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>}>}>
```

□

The described decomposition is in the same spirit as normal form decomposition for query bodies (see Section 3) but goes one step further. The condition for equivalence of the resulting graph component queries is easily derived:

Theorem 5.3 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 5.3 is a straightforward generalization of the containment theorem for unions of relational conjunctive queries. Moreover, the following is easy to see:

Theorem 5.4 (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.

6 Soundness, Completeness and Complexity

In the previous section we have shown that query rewriting can be done in three steps. In the first step, we find mappings from the body of the views to the body of the query. In the second step, we use "instantiated" view heads to construct the rewriting query and in the third, we check the correctness of the rewriting. The third 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 6.1 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.

[6] Remember that OEM graphs are rooted.

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

Lemma 6.2 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 6.3 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 to Lemmas 6.2 and 6.3. 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 MSLs case 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 6.4 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 this, 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 6.5 The rewriting algorithm proposed in subsection 4.5 is sound and complete.

Crux: 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. □

6.1 Complexity of MSL rewriting

The algorithm described in Section 4.3 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]. We now present a claim that the composition of the rewriting query and the participating views can be done in polynomial time and, assuming this, we show that MSL rewriting is NP-complete. The main observation is that query composition of arbitrary

[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.

[8] See also Appendix A.

queries finds all possible unifiers for each single path condition of the query Q' to be resolved. We *claim* that, for the rewriting query Q', only one of all the possible unifiers for each condition is necessary and sufficient to correctly compute the composition. To see this, let us consider the following example.

Example 6.6 Let the query and the (single) view be

(Q12) `<P p {<Q L W> <M R S>}> :- <Q L W> AND <P person {<M R S>}>`
(V7) `<P p {<Q L W> <M R S>}> :- <Q L W> AND <P person {<M R S>}>`

It should be intuitively obvious that we can answer (Q12) using just the data provided by (V7).

There is just one mapping from the body of the view to the body of the query. The rewriting query is (before and after normal form transformation):

`<P' p {<Q' L' W'> <M' R' S'>}> :- <P' p {<Q' L' W'> <M' R' S'>}>@V`
(Q13) `<P' p {<Q' L' W'> <M' R' S'>}> :- <P' p {<Q' L' W'>}>@V AND <P' p {<M' R' S'>}>@V`

Notice that it is obvious which single path condition in the view head each subgoal of (Q13) "originates" from.

To test the rewriting, we compose (Q13) with (V7). The composition is done as follows[9]: We can resolve the first subgoal of (Q13) in two different ways, resulting in two MSL rules:

(a) `<P p {<Q L W> <M' R' S'>}> :- <Q L W> AND <P person {<M R S>}>`
`AND <P p {<M' R' S'>}>@V`
(b) `<P p {<M R S> <M' R' S'>}> :- <Q L W> AND <P person {<M R S>}>`
`AND <P person {< M' R' S'>}>@V`

Notice that (a) is produced by a unifier that unifies the condition C:`<P' p {<Q' L' W'>}>` with the single path condition from which it originated; we call such unifiers *natural* unifiers. In contrast, (b) is produced using a unifier that unifies C with `<P p {<M R S>}>`. Resolving the last remaining unresolved condition gives us 4 MSL rules:

(a_1) `<P p {<Q L W> <M' R' S'>}> :- <Q L W> AND <P person {<M R S>}>`
`AND <Q'' L'' W''> AND <P person {<M' R' S'>}>`
(a_2) `<P p {<Q L W> <M' R' S'>}> :- <Q L W> AND <P person {<M R S>}>`
`AND <M' R' S'> AND <P person {<M'' R'' S''>}>`
(b_1) `<P p {<M R S> <M' R' S'>}> :- <Q L W> AND <P person {<M R S>}>`
`AND <Q1 L1 W1> AND <P person {<M' R' S'>}>`
(b_2) `<P p {<M R S> <M' R' S'>}> :- <Q L W> AND <P person {<M R S>}>`
`AND <M' R' S'> AND <P person {<M1 R1 S1>}>`

Notice the following: a_1 is generated by only using natural unifiers. Moreover, it turns out that a_2, b_1, b_2 are irrelevant to the correctness of the rewriting query: they all map[10] into a_1. Thus, we can use this limited and "focused" form of composition for the rewriting query: only consider natural unifiers during composition.

The general claim is that all the MSL rules produced during the full-fledged query composition of Q' with the view set $\mathcal{V}$ can be mapped into the (single) rule that is produced when composing using only natural unifiers. □

[9] As mentioned earlier, the order in which the subgoals are resolved is not important.
[10] It is a small technical issue to extend Section 5 to deal with containment of MSL queries, instead of equivalence.

That specialized query composition scheme of Q' with $\mathcal{V}$ takes polynomial time, since there is at most one natural unifier per single path condition in the rewriting query. Thus, assuming the truth of the above claim, we have the following:

Lemma 6.7 Query composition of the rewriting query Q' with the views $\mathcal{V}$ takes polynomial time.

We can then show that the query rewriting problem in MSL is NP-complete.

Theorem 6.8 Query rewriting for MSL is NP-complete.

Crux: NP-hardness is easily established by reducing relational query rewriting (which is NP-complete [LMSS95]) to rewriting for MSL. Membership in NP is proven as follows: We guess a rewriting of length $\leq k$, where k is the number of simple path conditions in the view and we also guess containment mappings involving a limited set of variables. (We may need fresh variables for reusing a view but, in the absence of recursion, the number of fresh variables is limited.) We "translate" the rewriting query Q' into a query on the source data using the polynomial query composition described above and we test if the mappings establish the equivalence between the composition and the original query. □

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.

[AV97] S. Abiteboul and V. Vianu. Queries and computation on the Web. In *Proc. ICDT 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.

[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.

[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.

[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 as www-cse.ucsd.edu/~yannis/papers/save.ps.

[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.

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

[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.

[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.

A Query composition

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, using *unification* (which generalizes [GN88, Ull89].) A unifier in MSL is defined as follows:

Definition: [Pap97] 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 A. □

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 A.1 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

(V8) `<f(A',B) p {<A' l V'> <B Y 'abc'>}> :-`
`<X label1 {<A' label2 V'> <B Y W> <C label3 T>}>`
(V9) `<F L E> :- <G l {<F L E>}>`

There exist two unifiers for the first condition of (Q14) and the head of (V8):

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

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

There exists one unifier for the first condition of (Q14) and the head of (V9):

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

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 (V9):

$$\theta_4 = [\mathtt{Q} \mapsto \mathtt{F}, \mathtt{L} \mapsto \mathtt{q}, \mathtt{E} \mapsto \{\mathtt{<D\ m\ V'>}\}]$$
$$\theta_5 = [\mathtt{Q} \mapsto \mathtt{F}, \mathtt{L} \mapsto \mathtt{q}, \mathtt{E} \mapsto \{\mathtt{<D\ m\ 'abc'>}\}]$$
$$\theta_6 = [\mathtt{Q} \mapsto \mathtt{F}, \mathtt{L} \mapsto \mathtt{q}, \mathtt{E} \mapsto \{\mathtt{<D\ m\ V>}\}]$$

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.)

Theorem A.2 (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 a more detailed description and proof of correctness of the unification process in MSL, see [Pap97].

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 $\{c_1^1, \ldots, c_n^1\}$ and $\{c_1^2, \ldots, c_m^2\}$ of single path conditions
OUTPUT All mappings θ such that for every condition $c_i^1, i = 1, \ldots, n$ there is a $c_j^2, j = 1, \ldots, m$ such that $\theta(c_i^1) \equiv c_j^2$
METHOD Run the function `mapmany`$(\{c_1^1, \ldots, c_n^1\}, \{c_1^2, \ldots, c_m^2\})$

function `mapmany`$(\{c_1^1, \ldots, c_n^1\}, \{c_1^2, \ldots, c_m^2\})$
  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 $\theta_i =$ `map1`$(c_i^1, c_{f(i)}^2, \theta_{i-1})$
        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.