Constraint-Based XML Query Rewriting for Data Integration

Cong Yu [*] [†]
Department of EECS, University of Michigan
congy@eecs.umich.edu

Lucian Popa
IBM Almaden Research Center
lucian@almaden.ibm.com

ABSTRACT

We study the problem of answering queries through a target schema, given a set of mappings between one or more source schemas and this target schema, and given that the data is at the sources. The schemas can be any combination of relational or XML schemas, and can be independently designed. In addition to the source-to-target mappings, we consider as part of the mapping scenario a set of target constraints specifying additional properties on the target schema. This becomes particularly important when integrating data from multiple data sources with overlapping data and when such constraints can express data merging rules at the target. We define the semantics of query answering in such an integration scenario, and design two novel algorithms, *basic query rewrite* and *query resolution*, to implement the semantics. The basic query rewrite algorithm reformulates target queries in terms of the source schemas, based on the mappings. The query resolution algorithm generates additional rewritings that merge related information from multiple sources and assemble a coherent view of the data, by incorporating target constraints. The algorithms are implemented and then evaluated using a comprehensive set of experiments based on both synthetic and real-life data integration scenarios.

1. INTRODUCTION

The data inter-operability problem arises from the fact that data, even within a single domain of application, is available at many different sites, in many different schemas, and even in different data models (e.g., relational and XML). The integration and transformation of such data has become increasingly important for many modern applications that need to support their users with informed decision making. As a rough classification, there are two basic forms of data inter-operability: *data exchange* and *data integration*. Data exchange (also known as data translation) is the problem of moving and restructuring data from one (or more) source schema(s) into a target schema. It appears in many tasks that require data to be transferred between independent applications that do not necessarily agree on a common data format. In contrast, data integration is the problem of uniformly querying many different sources through one common interface (target schema). There is no need to materialize a target instance in this case. Instead, the emphasis is on answering queries over the common schema [18, 20]. In both cases, of data exchange and data integration, relationships or *mappings* must first be established between the source schemas and the target schema. Mappings are often specified as high-level, declarative, assertions that state how groups of related elements in a source schema correspond to groups of related elements in the target schema. Mappings can be given by a human user or they can be derived semi-automatically [23, 26] based on the outcome of schema matching algorithms [27]. Mappings have been used for query rewriting in relational data integration systems, in the form of GAV (global-as-view) [20], LAV (local-as-view) [21] or, more generally, GLAV (global-and-local-as-view) assertions [16]. They have also been used to formally specify relational data exchange systems [14]. A more general form of GLAV that accounts for XML-like structures, and which we will use here as well, has been used to give semantics for mappings between XML schemas and to generate the data transformation scripts (in SQL, XQuery or XSLT) that implement the desired data exchange [26].

In this paper we study the data integration problem associated with mappings. More concretely, we study the problem of efficient answering of queries through a target schema, given a set of mappings between the source schema(s) and the target schema, and given that the data is at the source(s). While most of the preceding work has been focused on the relational case, we consider queries and mappings over both relational and XML schemas. We also consider, as part of the mapping scenario, a set of constraints on the target schema. The presence of such target constraints is an important requirement for us: they can be used to express *data merging* rules that arise when integrating data from multiple sources with overlapping information. Data merging is notoriously hard for data integration and often not dealt with. Integration of scientific data, however, offers many complex scenarios where data merging is required. For example, proteins (each with a unique protein id) are often stored in multiple biological databases, each of which independently maintains different aspects of the protein data (e.g., structures, biological functions, etc.). When querying on a given protein through a target schema, it is important to merge all its relevant data (e.g., structures from one source, functions from another) given the constraint that protein id identifies all components of the protein.

When target constraints are present, it is not enough to consider only the mappings for query answering. The target instance that a query should "observe" must be defined by the interaction between all the mappings from the sources and all the target constraints. This interaction can be quite complex when schemas and mappings are nested and when the data merging rules can enable each other, possibly, in a recursive way. Hence, one of the first problems that we study in this paper is what it means, in a precise sense, to answer the target queries in the "best" way, given that the target instance is specified, indirectly, via the mappings and the target constraints. The rest of the paper will then address how to compute the correct answers without materializing the full target instance, via two novel algorithms that rewrite the target query into a set of corresponding source queries.

Summary of results: Our main contributions are the following:

[*]Supported in part by NSF under grant IIS-0219513.

[†]Work partially done while at IBM Almaden Research Center.

SIGMOD 2004, June 13-18, 2004, Paris, France.

- **semantics of query answering:** we define what it means to answer a target query in the best way, given a set of mappings between the source schemas and the target schema, and given a set of target constraints. We define a *canonical target instance* that satisfies all the requirements (mappings and target constraints) with respect to the given source instances, and we take the semantics of query answering to be the result of evaluating the query on this canonical instance. While building on recent work on relational data exchange [14], this semantics captures not only relational settings but nested (XML) settings as well. It then becomes a requirement that we impose on the subsequent basic query rewriting and query resolution algorithms.
- **basic query rewriting algorithm:** this algorithm rewrites the target query into a set of source queries. Evaluating the union of these queries on the data sources has essentially the same effect as running the target query on the canonical target instance, provided that there are no target constraints. This algorithm extends earlier relational techniques for rewriting queries using views [13, 3, 17], with novel techniques for XML query rewriting that are based on XML mappings between XML schemas. Dealing with XML is a significant extension over the previous work as it requires handling of a variety of complex types and queries as well as the hierarchical structure that the relational model does not have. Furthermore, we prove that the algorithm is *complete*, in the sense that the resulting rewritings retrieve *all* the answers, according to the semantics given by the canonical target instance. To the best of our knowledge, this is the first complete algorithm to perform query rewriting based on mappings by operating directly on nested structures. The class of queries that we consider is a considerable fragment of XQuery [7] that includes nested subqueries.
- **query resolution algorithm:** this algorithm extends the above one by taking into account target constraints to generate additional source queries to produce merged results. Such merged results are among those that would be obtained by running the target query on the target canonical instance, which is constructed based on the mappings *and* the target constraints. The constraints that we consider are *nested equality-generating dependencies (NEGDs)* and they include functional dependencies in relational or nested schemas, XML Schema key constraints, and more general constraints stating that certain tuples/elements in the target must satisfy certain equalities. To the best of our knowledge, the resolution techniques that we give in this paper are entirely new.

The target constraints are specified solely based on the target schema, which is assumed to be designed independently of the source schemas. In fact, the target constraints are often part of the target schema (e.g., key constraints in XML Schema). We do not deal in this paper with the detection and resolution of conflicts that may arise due to target constraints. A conflict is a case where two (or more) source values are supposed to be equal due to a target constraint but they are not. The rewritings that we generate will include all such conflicting values, when they exist. Handling of conflicts is an additional problem in data integration that we believe is complementary to the techniques described in this paper and which we plan to address in future work.

Related work: There has been considerable work on XML and semistructured query rewriting. [24] and [6] focus on query optimization by rewriting semistructured queries in terms of materialized views. MARS [10, 11] employs a powerful method for reformulating XML queries in publishing scenarios that are based on mixed (i.e., XML and relational) and redundant (i.e., cached views, indexes) storage. XPeranto [28] and SilkRoute [15] address the problem of publishing SQL data in XML by rewriting XML queries into SQL queries. In fact, there is a vast amount of work on XML-to-SQL translation [19]. In most of the above cases, the source (materialized views, relational store, etc.) to target (XML logical schema, XML view, etc.) mapping is *lossless*: it consists of statements (whether explicit or implicit) each asserting that some portion of the XML data is *equal* to some portion of the relational (store) data. Hence, query rewriting is equivalence-preserving. In contrast, the query rewriting that we consider involves *lossy* or *incomplete* mappings, where each statement asserts that some portion of a source is a *subset* of some portion of the target. Thus, the data sources and their mappings offer an incomplete, partial, view of the world. As a consequence of this incompleteness, the goal of query rewriting is on obtaining *contained* rewritings (and, if possible, maximally-contained rewritings [17]) instead of equivalent rewritings (which may not exist). For the design of scalable data integration systems, having lossy mappings is a real-life necessity [17, 20]. Usually, each source to target mapping is defined independently of other mappings or data sources and involves only a part of the target schema. The advantage of this design is its *modularity* and *scalability*: new mappings (and target constraints) can be easily added into the system without affecting other mappings and constraints. It is the run-time (i.e., the query answering system) that takes the responsibility of assembling a coherent view of the world out of the many mappings and constraints on the target. This modular design is also adopted by the Agora [22] system. However, their query rewriting is performed via a translation first to a generic relational schema, and by employing then relational techniques for answering queries using views. The mappings themselves are required to be defined in terms of this generic relational schema rather than in terms of the XML target. Because of the translation, queries and mappings can be quite complex and hard to understand/define by a human user. On the other hand, our techniques operate directly at the XML level and form the basis for an integrated solution for XML query rewriting in the presence of both lossy mappings and target constraints.

Capturing target constraints in XML data integration or query rewriting has been recently studied in [2, 5, 10, 11]. Translation of relational data into XML with predefined type and key constraints is the focus of [5]. It is an example of data exchange and hence orthogonal to the work that we describe here. The work in [10] and [11] offers powerful techniques for query reformulation in the presence of constraints. However, their techniques are equivalence-preserving and we found them inapplicable for the scenarios that we consider. An approach that is similar in spirit to ours is the one reported in [2], addressing the problem of query rewriting in the presence of incomplete (LAV-style) mappings and target constraints (keys), while allowing for nested structures. Their work, however, deals with simpler classes of mappings, queries and constraints, does not attempt an analysis of the properties (e.g., completeness) of the algorithms and does not contain an experimental validation of the approach. Finally, target constraints in the context of query answering in relational LAV data integration systems have been considered in [12]. Their algorithm produces a recursive query plan that incorporates the target constraints via a set of recursive rules in the spirit of logic programming and chase. In contrast, our focus is on obtaining non-recursive rewritings that can be efficiently optimized and executed by SQL or XQuery engines.

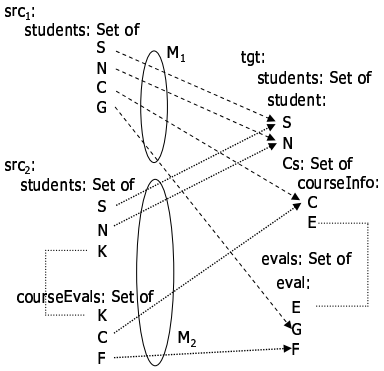
The rest of the paper is organized as following: Section 2 provides an overview of our proposed solutions. Section 3 explains *what* are the query answering requirements (semantics). Section 4 and 5 address *how* this semantics can be achieved through the basic query rewriting and query resolution algorithms, respectively. Experimental evaluation is provided in Section 6 and finally we conclude in Section 7.

2. OVERVIEW OF OUR SOLUTIONS

This section gives an overview of our solutions to the problem of query answering over a virtual target. We start with an example of *mappings* between two relational source schemas and a nested target schema. We then give an example of a target *query* and source instances and show what the intuitive result to the query should be. Next, we add *constraints* on the target to show how this impacts the expected answer to the (same) query. Using the examples, we formulate what the semantics of query answering should be.

2.1 Schemas and Mappings

Consider the mapping scenario in Figure 1. It illustrates two source relational schemas ($\texttt{src}_1$ and $\texttt{src}_2$), a target nested schema (tgt),



M_1: foreach s in src$_1$.students
exists s' in tgt.students, c' in s'.student.Cs,
e' in tgt.evals
where c'.courseInfo.E = e'.eval.E
with s'.student.S = s.S and s'.student.N = s.N and
c'.courseInfo.C = s.C and e'.eval.G = s.G

M_2: foreach s in src$_2$.students, c in src$_2$.courseEvals
where s.K = c.K
exists s' in tgt.students, c' in s'.student.Cs,
e' in tgt.evals
where c'.courseInfo.E = e'.eval.E
with s'.student.S = s.S and s'.student.N = s.N and
c'.course.C = c.C and e'.eval.F = c.F

Figure 1: Schemas and mappings.

and two mappings between them. The schemas are shown in a nested relational representation (described shortly) that is used as a common data model to represent both relational schemas and XML Schema[1]. The symbols S, N, C, G, F represent, respectively, "student id", "student name", "course", "grade" (only in src_1), and "file evaluation" (a written evaluation that a student receives for a course; only in src_2). Information in the two sources may overlap: the same student may appear in both sources. The attribute K in src_2 is used to link students with the courses they take. The target schema consists of a root element `tgt` containing two subelements: `students` and `evals`. The first one contains zero or more `student` elements; we make here use of the keyword SetOf to denote that the value of `students` is a *set* of `student` elements. A `student` is a complex element containing atomic elements S, N, as well as a set element, Cs, containing course related entries (`courseInfo`). A course entry contains a course (C), while the grade (G) and file evaluation (F) for that course are stored separately under `evals`. The element E plays a "linkage" role: it is, in XML Schema terms, a *key* for `eval`, and a *keyref* in `courseInfo`.

Schemas and Types In general, a *schema* is a set of labels (called roots), each with an associated *type* τ, defined by:

τ ::= String | Int | SetOf τ | Rcd$[l_1 : \tau_1, \ldots, l_n : \tau_n]$
| Choice$[l_1 : \tau_1, \ldots, l_n : \tau_n]$.

Types Int and String are called atomic types, SetOf is a set type, and Rcd and Choice are complex types. With respect to XML Schema, we use SetOf to model repeatable elements (or repeatable groups of elements)[2], while Rcd and Choice are used to represent the "all" and "choice" *model-groups*. We do not consider order, SetOf represents unordered sets. "Sequence" model-groups of XML Schema are also represented as (unordered) Rcd types. Although the mappings, queries, and algorithms that we implemented all handle Choice types we will ignore this aspect from now on, due to lack of space.

Two mappings have been defined, in Figure 1, from the two source schemas to the target schema. Graphically, the mappings are described by the arrows that go between the "mapped" schema elements. However, the precise semantics of these mappings is embedded in M_1 and M_2, which are also called logical mappings. Each of them is, essentially, a constraint of the form $Q^S \rightsquigarrow Q^T$, where Q^S (the foreach clause and its associated where clause) is a query over the sources and Q^T (the exists clause and its associated where clause) is a query over the target. These mappings specify a containment assertion: for each tuple returned by Q^S, there must exist a corresponding tuple in Q^T. The with clause makes explicit how the source and the target elements relate to each other.

Mapping M_1 specifies how student tuples in the first source relate to `student`, `courseInfo`, and `eval` elements in the target. The exists and the where clauses specify how the target elements themselves relate to each other (and to the root of the schema). For example, the generator c' in s'.student.Cs asserts that each `courseInfo` element (denoted by the variable c') must be an element of the set s'.student.Cs. Also, the target join condition c'.courseInfo.E = e'.eval.E specifies how a `courseInfo` element relates to an `eval` element. Similarly, mapping M_2 specifies how student information in the second source relate to `student`, `courseInfo`, and `eval` elements in the target. The difference from the first mapping is that the student and course information is split across two tables and a source join condition must be used (s.K = c.K).

Mapping language Let an expression be defined by the grammar $e ::= S \mid x \mid e.l$, where x is a variable, S is a schema root, l is a label, and $e.l$ is record projection. Then a mapping is a statement (constraint) of the form:

$$\begin{array}{l}\mathcal{M} ::= \text{foreach } x_1 \text{ in } g_1, \ldots, x_n \text{ in } g_n \\ \quad\quad \text{where } B_1 \\ \quad\quad \text{exists } y_1 \text{ in } g'_1, \ldots, y_m \text{ in } g'_m \\ \quad\quad \text{where } B_2 \\ \quad\quad \text{with } e'_1 = e_1 \text{ and } \ldots \text{ and } e'_k = e_k\end{array}$$

where each x_i in g_i (y_j in g'_j) is called a *generator* and each g_i (g'_j) is an expression e with type SetOf τ; the variable x_i (y_j) binds to individual elements of the set e. The mapping is well-formed if the variable (if any) used in g_i (g'_j) is defined by a previous generator within the same clause. Any schema root used in the foreach or exists clause must be a source or target schema root, respectively. The two where clauses (B_1 and B_2) are conjunctions of equalities between expressions over $x_1, \ldots, x_n$, or $y_1, \ldots, y_m$, respectively. They can also include equalities with constants (i.e., selections). Finally, each equality $e'_i = e_i$ in the with clause involves a target expression e'_i and a source expression e_i, of the same atomic type.

The mapping language allows for partial specifications. For example, mapping M_1 specifies a grade, but not a file evaluation, while for M_2 the opposite is true. Also, the value of E is not specified by either M_1 or M_2, although this value plays an important correlation role and it appears in the target where clause. The advantage of such mappings is that the system can be defined, incrementally, from incomplete mappings that are independent of each other and that do not attempt to (or cannot) fully specify the target. This becomes more important when we add target constraints to mappings. We note that when restricted to the relational model the above mapping language coincides with the language of sound (but not necessarily complete) GLAV assertions [16, 20]. Finally, the above mapping language is the one used to represent XML-Schema mappings in Clio [26].

2.2 Queries

Queries use *instances* as input and create *values* as output. Hence, we first define instances and values. Given a schema, an instance is defined as a set of values for the roots, with the requirement that the types must be respected. A value of type Rcd$[l_1 : \tau_1, \ldots, l_k : \tau_k]$, called a record value, is an unordered tuple of label-value pairs: $[A_1 = a_1, \ldots, A_k = a_k]$, where $a_1, \ldots, a_k$ are of types τ_1, ..., τ_k, respectively. A value of type SetOf τ is a special value called a *SetID*. Each such *SetID* can be associated with a set $\{v_1, \ldots, v_n\}$

[1] http://www.w3.org/TR/xmlschema-0/

[2] In Figure 1, only set elements are marked with their type. Complex and atomic elements are represented implicitly as elements with subelements and, respectively, as leaves.

(q_1) for s in tgt.students, c in s.student.Cs, e in tgt.evals
where c.courseInfo.E = e.eval.E
return [name = s.student.N, course = c.courseInfo.C,
grade = e.eval.G, file = e.eval.F]
(q_2) for s in tgt.students
return [name = s.student.N,
results = for s' in tgt.students, c in s'.student.Cs,
e' in tgt.evals
where c'.courseInfo.E = e'.eval.E and
s'.student.N = s.student.N
return [result = [grade = e'.eval.G,
file = e'.eval.F]]

Figure 2: Target queries

of values of type τ (these are the elements of the set, in the classical sense). This representation of sets (using *SetID*s) is consistent with the graph or tree-based models of XML, where elements are identified by their node id rather than by contents. The association between a *SetID* S and its elements $\{v_1, \ldots, v_n\}$ is denoted by *facts* of the form: $S(v_1), \ldots, S(v_m)$. The following are two source instances, src_1 and src_2, for the source schemas in Figure 1: *src*$_1$ = [students = S_1], and *src*$_2$ = [students = S_2, courseEvals = C], where S_1, S_2, and C are *SetID*s with the following associated facts (for simplicity, we drop the field labels and the angle brackets surrounding the record values):

$S_1(001, Mary, CS120, A)$, $S_1(005, John, CS500, B)$
$S_2(001, Mary, K7)$, $S_2(001, Mary, K4)$
$C(K7, CS120, file01)$, $C(K4, CS200, file07)$

Figure 2 shows two queries over the target schema of Figure 1. The queries are written in an XQuery-like notation, using the `for-where-return` style of syntax; queries with `let` clauses are not in the syntax, although they can be represented using subqueries in the for clause and then rewritten to eliminate these subqueries. This notation is the internal *nested relational* language into which external queries in XQuery syntax are translated. Query q_1 asks for all tuples with name, course, grade and file for that course. Record expressions (e.g., [name = s.student.N,...]) are used to construct and to group elements together in the return clause. Query q_2 constructs nested output, by using a subquery that returns a set of results (with grade and file) for each student.

Core query language (CQ) The queries that we consider have the following general form:

q ::= for x_1 in $g_1, \ldots, x_n$ in g_n
where B
return r

where r is defined by r ::= $[A_1 = r_1, \ldots, A_k = r_k] \mid e \mid q$, and g_i, e, B are defined as with mappings. Both source and target schema elements can be used in q and in fact, during rewriting, a partially rewritten query will mix target and source schema elements. We call this language CQ (core queries). Additionally, CQ allows for *Skolem functions* wherever an expression e can be used. Skolem functions, which we introduce in Section 4.1, play an important role in describing "unknown" values as well as *SetID*s. In our implementation, features such as user functions, arithmetic operations, inequalities, etc., are allowed in the query, though they are not part of CQ. An important subclass of CQ is one in which queries can not have subqueries. We call this fragment CQ_0.

2.3 Target Query Answering

We address first the following question: *given a set of source instances (e.g., src_1 and src_2 shown before), and given a set of mappings (e.g., M_1 and M_2), what should the answers to a target query such as q_1 be?* One possibility that has been considered in the relational literature on query answering using views [17, 20] as well as in query answering in incomplete databases [29], is to consider the set of *all* possible target instances J that are consistent with the mappings and the source instances and then take the intersection of $q_1(J)$ over all such J. This intersection is called the set of the *certain answers*. We take a somewhat different approach, which has a more constructive flavor and can be generalized to the cases of XML queries. This approach, described next, extends recent work on semantics of relational data exchange and query answering based on universal solutions [14]. Moreover, we show in Section 3 that this approach is more inclusive than the semantics by certain answers, even in the relational case.

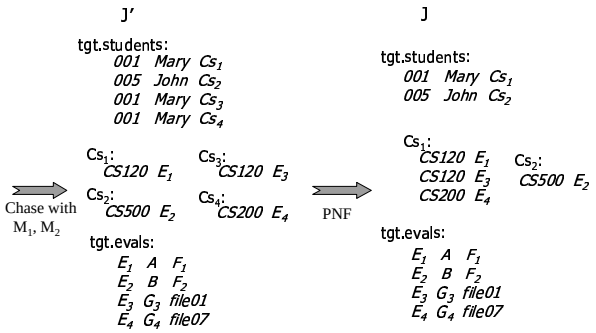


Figure 3: The canonical target instance is J.

Canonical instance We define the semantics of target query answering by constructing a *canonical* target instance J based on the source instances and the mappings. Intuitively, this is what a user would see if we are to materialize a target instance that is consistent with the given sources and mappings. We then define the result of a target query to be the result of evaluating it on J. The construction of the canonical target instance is described below.

For each combination of tuples in the sources that matches the foreach clause and its where clause of a mapping we add tuples in the target so that the exists clause and its where clause are satisfied. The atomic values being added are either equal to source values (if the with clause specifies this equality) or they are created as new, "unknown", values (called *nulls*). Every generated null is different from the other nulls and source values unless the mapping or some target constraint specifies otherwise. In addition to creating nulls, we also generate *SetID*s. For our example source instance src_1, the tuple $[001, Mary, CS120, A]$ matches M_1; we thus add three related tuples in tgt: a `student` tuple $[001, Mary, Cs_1]$ under `tgt.students`, a `courseInfo` tuple $[CS120, E_1]$ under Cs_1, and an `eval` tuple $[E_1, A, F_1]$ under `tgt.evals`. Here, Cs_1 is a newly generated *SetID*, while E_1 and F_1 are nulls. Note that the same null E_1 is used to fill in the two E elements in `courseInfo` and `eval` as required by the mapping via the condition c'.courseInfo.E = e'.eval.E in the exists clause. The target instance that results after filling in all the necessary target tuples is the instance J' of Figure 3. The process described informally here is called the *chase*.

Finally, we take the view that the mapping also specifies some implicit grouping conditions that the target must satisfy. In particular, we require that in any set we cannot have two distinct tuples that agree on all the atomic valued elements, component wise, but do not agree on the set-valued elements (if any). For example, the three tuples under `tgt.students` that involve Mary all have the same atomic values but different *SetID*s: Cs_1, Cs_3, and Cs_4. We do not allow such instance. Instead, we collapse the three tuples into one by identifying the three *SetID*s. Hence, the three singleton sets containing `courseInfo` for Mary are merged into one set (identified by Cs_1; see the instance J in Figure 3). The resulting instance J is in Partitioned Normal Form (or PNF) [1]. PNF is a basic form of data merging that is consistent with most user requirements. The instance J is then our canonical instance.

The evaluation of q_1 on J produces the following set of tuples: $\{[Mary, CS120, A, F_1], [John, CS500, B, F_2], [Mary, CS120, G_3, file01], [Mary, CS200, G_4, file07]\}$. We view this as being the right result for q_1. We adopt this semantics as a *formal requirement on query answering*, although not necessarily as an implementation strategy. In

fact, we show in Section 4 how to implement this semantics, without materializing the canonical instance, but instead by query rewriting. For q_1, we would generate the following rewritings:

(r_1) for s in src$_1$.students
return [name = s.N, course = s.C, grade = s.G, file = null]
(r_2) for s in src$_2$.students, e in src$_2$.courseEvals
where s.K = e.K
return [name = s.N, course = e.C, grade = null, file = e.F]

Evaluating $r_1 \cup r_2$ on the sources (src_1 and src_2) gives precisely the above four tuples that would be obtained by evaluating q_1 on J. This is modulo the fact that the null values for grade and file have all been replaced by one single `null` value. In general, "unknown" values can be replaced by `null` at the end of the rewriting process, as long as the corresponding schema element is nullable in the target schema. This would not be the case if we were to return one of the two E elements: E plays an integrity role (key/keyref) and cannot be nullable. The rewritings would then include Skolem terms to explicitly construct values for E.

The relational chase [4] has been used to study the semantics of data exchange as well as query answering in [14]. Their canonical universal solution would correspond to our canonical target instance, if we restrict to the relational model. We use here a nested extension [25] of the chase.

2.4 Target Constraints

Next, we consider the following question: *under which conditions, will the answers to the target query include "merged" tuples that, for example, fuse the grade and the file for Mary and CS120 in our running example?* The answer will rely on the use of the target constraints to specify data merging rules. For example, the following constraints can be specified on the target instance for the mapping scenario of Figure 1:

(c_1) for s_1 in tgt.students, c_1 in s_1.student.Cs,
s_2 in tgt.students, c_2 in s_2.student.Cs
[s_1.student.S = s_2.student.S and
c_1.courseInfo.C = c_2.courseInfo.C
→ c_1.courseInfo.E = c_2.courseInfo.E]
(c_2) for e_1 in tgt.evals, e_2 in tgt.evals
[e_1.eval.E = e_2.eval.E
→ e_1.eval.G=e_2.eval.G and e_1.eval.F=e_2.eval.F]

The constraint c_1 asserts that there must be at most one evaluation id (E) for a given pair of student id (S) and course (C), while c_2 asserts that E must be a primary key of the set `tgt.evals`. Both constraints are functional dependencies; however c_1 is a functional dependency that goes across nested sets. The constraints that we use generalize the relational equality-generating dependencies of [4] and are the same fragment as the EGDs introduced in [25] for a slightly richer data model. We call these constraints NEGDs (nested equality-generating dependencies). The general form is:

for x_1 in $g_1, \ldots, x_n$ in g_n [$B_1 \rightarrow B_2$]

where g_i, B_1, B_2 are defined as with mappings and queries. NEGDs capture XML Schema key constraints as well as forms that are not expressible by XML Schema (e.g., c_1).

Recall the source instances given earlier for our example, and the canonical instance J constructed in Figure 3. Given the additional specification with c_1 and c_2, J is no longer an accurate view of the sources. In particular, the constraints c_1 and c_2 are not satisfied (there are two distinct E values for $Mary$ and $CS120$). We define the new canonical instance to be the result of additional chasing of J with the target constraints. Figure 4 shows how J is transformed into J_1 via this chase. Concretely, whenever two distinct nulls are required to be equal by some constraint, we enforce the equality by identifying one with the other. Similarly, if a null is required to be equal to a source value, we enforce it by replacing the null with the source value. At the end we may need to reapply the PNF procedure. The result, J_1, is now the "correct" view of the sources. It now includes only one

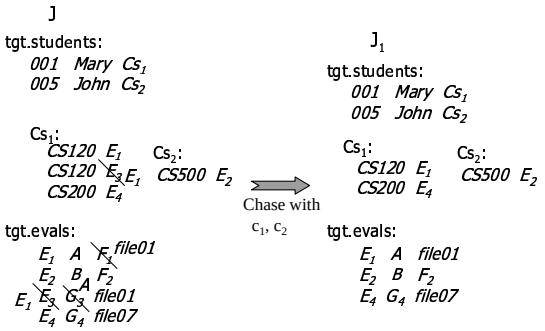


Figure 4: Chase with target constraints.

tuple containing all the information (grade and file) about $Mary$ and $CS120$. We take the semantics of target query answering to be the result of evaluating the query on this canonical target instance.

For our example, if we evaluate q_1 on J_1, we obtain: $\{[Mary, CS120, A, file01], [John, CS500, B, F_2], [Mary, CS200, G_4, file07]\}$. The first tuple correctly "fuses" data from the two sources. One of our main contributions is the technique to produce such tuples that are consistent with the mappings *and* the target constraints, without materializing a canonical instance, but instead by additional transformation of the query (via *resolution*, Section 5). In particular, we obtain an additional rewriting for q_1 that joins the two sources on student id and course, and produces the "fused" tuple:[3]

(r_3) for s in src$_1$.students, s' in src$_2$.students,
e' in src$_2$.courseEvals
where s'.K = e'.K and s.S = s'.S and s.C = e'.C
return [name = s.N, course = s.C, grade = s.G, file = e'.F]

3. QUERY REWRITING REQUIREMENTS

In this section, we give the formal details regarding the semantics for target query answering, and articulate in a precise way the requirements on query rewriting. Let I be a source instance, Σ_{st} be a set of mappings, and Σ_t be a set of target constraints (NEGDs), all arbitrary. In general, there can be multiple canonical instances J for I due to the fact that the chase may choose different names for the nulls, or the sequence in which the mappings and constraints are applied may be different. However, the important fact about canonical instances is that they behave essentially the same when queries in CQ$_0$ are evaluated on them. We call a target instance K a *solution* with respect to I, Σ_{st}, and Σ_t (or solution, in short), if K satisfies the constraints of $\Sigma_{st} \cup \Sigma_t$ for the given I.

DEFINITION 3.1. Let q be a CQ$_0$ query. Then the *set of the PNF-certain answers of q with respect to I, Σ_{st} and Σ_t*, denoted *PNF-certain*$_{\Sigma_{st} \cup \Sigma_t}(q, I)$, is the set of all tuples t such that $t \in q(K)$ for *every* solution K.

PROPOSITION 3.2. *Let J be a canonical target instance for I, Σ_{st} and Σ_t. Then for every CQ$_0$ query q, we have that $q(J)_{\downarrow}$ = PNF-certain$_{\Sigma_{st} \cup \Sigma_t}(q, I)$. Here $q(J)_{\downarrow}$ is the result of evaluating q on J and then removing all tuples with nulls.*

Thus, CQ$_0$ query evaluation on a canonical instance is essentially the same as computing the certain answers. In general, we would like a rewriting algorithm to produce a rewriting r of q such that $r(I)_{\downarrow} = q(J)_{\downarrow} (= \textit{PNF-certain}_{\Sigma_{st} \cup \Sigma_t}(q, I))$. Typically, r must be a union of rewritings. We call such r a *sound and complete rewriting*. When the exact equality is not guaranteed by the algorithm but we have that $r(I)_{\downarrow} \subseteq q(J)_{\downarrow}$ (= *PNF-certain*$_{\Sigma_{st} \cup \Sigma_t}(q, I)$), we say that r is a *sound rewriting*. An algorithm that always produces sound rewritings is sound. An algorithm that always produces sound and

[3] We also assume here, for simplicity of presentation, that student id (S) functionally determines student name (N). Otherwise, the rewriting is slightly more complex, in order to account for all the different names for a given student id.

complete rewritings is sound and complete. We are interested in rewriting algorithms that are sound and, if possible, complete.

For the larger class CQ of queries, the classical notion of the certain answers is no longer sufficient (because the answers are no longer flat tuples[4]). However, we will continue to use the canonical target instance to define the semantics of query answering, thus going beyond the notion of the certain answers and beyond CQ_0. Furthermore, in practice, we do not remove tuples that contain nulls during evaluation, that is, we do not compute $r(I)_{\downarrow}$ (or $q(J)_{\downarrow}$) but rather $r(I)$ (or $q(J)$). In that case, we cannot require the exact equality $r(I) = q(J)$ (or $r(I) \subseteq q(J)$, for that matter), since the nulls (as well as the *SetID*s in the case of queries that return nested sets) may be named differently. Instead, for sound rewritings, we require a relaxed version of containment: $r(I) \leq q(J)$ if there exists a function h mapping the nulls occurring in $r(I)$ into values (null or non-null) of $q(J)$ and mapping the *SetID*s occurring in $r(I)$ into *SetID*s of $q(J)$, such that the facts of $r(I)$ are mapped into facts of $q(J)$ (when $r(I)$ and $q(J)$ are viewed as nested instances). For sound and complete rewritings, we will require that $r(I) \leq q(J)$ and $q(J) \leq r(I)$.

4. BASIC QUERY REWRITING

We describe next the basic algorithm that rewrites a target query into a set of source queries, based on the mappings. There are four phases in the algorithm: *rule generation*, *query translation*, *query optimization*, and *query assembly*. Rule generation creates a set of *mapping rules* based on the schemas and the given set of mappings. The mapping rules are then used in the translation phase to reformulate target queries into (unions of) source queries. If the target query has nested subqueries in the return clause, the translation phase also decorrelates the query into a set of stand-alone queries and translates them one by one. The optimization phase removes unsatisfiable source queries and minimizes the satisfiable ones. The assembly phase re-assembles the decorrelated source queries back into queries with nested subqueries, if the original target query is nested. In Section 5 we extend this basic algorithm to handle target constraints, by inserting a *query resolution* phase between the translation and the optimization phases.

4.1 Rule Generation

Mappings are often incomplete and may specify only a subset of the target elements. The goal of the rule generation phase is to turn this incomplete specification into a set of mapping rules that fully specify the target in terms of the sources, so that target expressions can be substituted by source expressions later. We illustrate the algorithm for generating mapping rules using our running example in Figure 1. The algorithm starts by generating a rule for each root of the target schema. For the root `tgt` of our target schema we generate:

(R_0) tgt = [students = $SK_{tgt.0}$(), evals = $SK_{tgt.1}$()]

$SK_{tgt.0}$ and $SK_{tgt.1}$ are 0-ary *Skolem functions* that are used to generate the *SetID*s for the set elements `students` and `evals`, respectively. The functions are 0-ary because only one instance of each of these sets must exist, according to the schema. In general, we associate each set type in the target schema with a unique Skolem function that can generate instances (i.e., *SetID*s) at that type. The Skolem function depends, in general, on the atomic elements from the current nesting level as well as from the parent and ancestor levels. Once we finished with the roots, the algorithm looks at the mappings. For each mapping and for each generator in the exists clause we construct a rule. For the generator s' in tgt.students in M_1, we obtain:

(R_1) $SK_{tgt.0}$() ←
for s in src$_1$.students
return [student = [S = s.S, N = s.N,
Cs = $SK_{tgt.0.0.2}$(s.S, s.N)]]

$SK_{tgt.0}$() is the head of the rule, while the body of the rule is a query that constructs `student` elements for the set denoted by $SK_{tgt.0}$() (i.e., `students`). The foreach and associated where clause of the mapping become the for and where clauses of the query, while the source expressions in the with clause are used to fill in values for the atomic elements (s.S and s.N, in this case) in the return clause. For the set type Cs, we again use a Skolem function to generate corresponding *SetID*s. In this case, a new *SetID* (and, accordingly, a new set of `courseInfo` elements) is generated for each different combination of s.S and s.N. The name of the Skolem function, $SK_{tgt.0.0.2}$, is generated based on the position of the element Cs in the schema (e.g., it is the 2nd child of the 0th child of the 0th child of `tgt`). Continuing with c' in s'.student.Cs of M_1, we produce:

(R_2) $SK_{tgt.0.0.2}$(s.S, s.N) ←
for s in src$_1$.students
return [courseInfo=[C=s.C,
E=SK_{125}(s.S,s.N,s.C,s.G)]]

This rule populates, with elements, the *SetID*s that were created by R_1. The head of this rule is a Skolem function with arguments: s.S and s.N. Hence, the rule generates `courseInfo` elements under a different set $SK_{tgt.0.0.2}$(s.S, s.N), for each different pair of s.S and s.N. Note that courses for the same student (i.e., same id and name) are grouped together under the same set. This is in accordance with the requirement that the target instance must be in partitioned normal form (PNF). Since M_1 does not specify a value for the atomic element E, we use an atomic type Skolem function (SK_{125} is a system generated name) to create a value for it. This function depends on all the source atomic elements that are mapped into the target via the mapping. Finally, one more rule is generated for M_1:

(R_3) $SK_{tgt.1}$() ←
for s in src$_1$.students
return [eval=[E=SK_{125}(s.S,s.N,s.C,s.G), G=s.G,
F=SK_{126}(s.S,s.N,s.C,s.G)]]

Note that the same function SK_{125} as in R_2 is used to generate an E-value. This is because the mapping requires the two E values to be equal, as specified in the target where clause. In a similar fashion, three more rules are generated from the second mapping, M_2. We list below two of them:

(R_4) $SK_{tgt.0}$() ←
for s in src$_2$.students, c in src$_2$.courseEvals
where s.K = c.K
return [student = [S = s.S, N = s.N, Cs = $SK_{tgt.0.0.2}$(s.S, s.N)]]

(R_5) $SK_{tgt.0.0.2}$(s.S, s.N) ←
for s in src$_2$.students, c in src$_2$.courseEvals
where s.K = c.K
return [courseInfo = [C = c.C, E = SK_{127}(s.S, s.N, c.C, c.F)]]

The same Skolem function, $SK_{tgt.0.0.2}$, is used in R_4 and R_5 as in R_1 and R_2 (as mentioned, such Skolem function is associated with the set type in the schema and not with a mapping). Hence, if the same student id and name occur in both sources, the course related information is *merged* under the same set. Again, this reflects the PNF requirement on the target. Such PNF-based merging is a form of data merging across sources that we achieve even without additional target constraints.

After iterating through all the mappings, the algorithm merges all the rules with the same Skolem function in the head, to obtain a single definition for each target set type. Essentially, this amounts to taking a union and is straightforward for $SK_{tgt.0}$() and $SK_{tgt.1}$(). However, in the case of combining rules R_2 and R_5, we cannot just take the union since the head is parameterized. Instead we describe the combined effect of the two rules by defining (see rule R_{25} in Figure 5) $SK_{tgt.0.0.2}$ as a function (lambda), with two arguments, l_1 and l_2, denoting possible student id and name. The function returns a set of associated course entries by combining the bodies of the rules R_2 and R_5. Note that the two queries used inside R_{25} are now parameterized by l_1 and l_2. If the same student id and name appear in both sources then both these queries are non-empty and their union is non trivial. Figure 5 gives all the mapping rules for our example. We note that evaluating the generated mapping rules on any source instance would give us a canonical target instance as described in Section 2.3.

4.2 Query Translation

The next phase is to translate the target query into a set of source queries based on the mapping rules. The *QueryTranslate* algorithm

[4] An agreed upon definition of the certain answers in this case does not exist, as far as we know.

(R_0) tgt = [students = $SK_{tgt.0}$(), evals = $SK_{tgt.1}$()]
(R_{14}) $SK_{tgt.0}$ = λ() . {
for s in src$_1$.students
return [student = [S = s.S, N = s.N,
Cs = $SK_{tgt.0.0.2}$(s.S, s.N)]]
∪
for s in src$_2$.students, c in src$_2$.courseEvals
where s.K = c.K
return [student = [S = s.S, N = s.N,
Cs = $SK_{tgt.0.0.2}$(s.S, s.N)]] }
(R_{25}) $SK_{tgt.0.0.2}$ = λ (l_1, l_2) . {
for s in src$_1$.students
where l_1 = s.S and l_2 = s.N
return [courseInfo = [C = s.C,
E = SK_{125}(s.S,s.N,s.C,s.G)]]
∪
for s in src$_2$.students, c in src$_2$.courseEvals
where s.K = c.K and l_1 = s.S and l_2 = s.N
return [courseInfo = [C = c.C,
E = SK_{127}(s.S,s.N,c.C,c.F)]] }
(R_{36}) $SK_{tgt.1}$ = λ () . {
for s in src$_1$.students
return [eval = [E = SK_{125}(s.S,s.N,s.C,s.G), G = s.G,
F = SK_{126}(s.S,s.N,s.C,s.G)]]
∪
for s in src$_2$.students, c in src$_2$.courseEvals
where s.K = c.K
return [eval = [E=SK_{127}(s.S,s.N,c.C,c.F),
G=SK_{128}(s.S,s.N,c.C,c.F), F=c.F]] }

Figure 5: Mapping rules

(shown in Figure 6) achieves this by iteratively substituting generators in the for clause of the target query with source generators of the matching mapping rules. The set of **transformation rules** (also in Figure 6) are applied to transform the rest of the query after each substitution. We describe next both the transformation rules and the *QueryTranslate* algorithm. We will use the notation $E[x \rightarrow y]$ to denote the result of substituting all occurrences of x in E with y and then recursively applying the transformation rules.

The transformation rules describe the steps involved in transforming a query when an expression in the query is replaced by another expression. There are a total of four rules: 1) Lambda substitution rule: this rule extracts the union of queries (E) from the body of the lambda definition and replaces the lambda variables with the actual arguments. 2) Union separation rule: this rule divides a query whose for clause contains a union of queries into a set of queries without union in the for clause. 3) De-Nesting rule: this rule applies when a query contains an inner query in its for clause. It replaces the generator (g in $\mathcal{Q}$) with the for clause of $\mathcal{Q}$ and appends the original where clause to the where clause of $\mathcal{Q}$. Every occurrence of g throughout the query is then substituted with $\mathcal{Q}$'s return clause. 4) Record projection rule: intuitively, this rule applies when a record value is projected on one of its labels. The inner value matching the label is returned as the result of the projection. It can be applied multiple times until no projection step is left.

The algorithm first substitutes the target root element using the root rule (e.g., R_0 in Figure 5). The result is marked as a top query to differentiate it from a subquery nested inside another query. The query is then added to L, a list of all partially rewritten queries. In these partial queries, whenever a Skolem function occurs in a generator in the for clause, it is substituted with the lambda definition of the corresponding mapping rule. After each substitution step, the algorithm applies all the applicable transformation rules and adds the resulting query or queries back to L. Eventually, the for clause will contain only source generators and this marks the completion of the rewriting for this query (without considering its subqueries). If the query contains subqueries, then the algorithm decorrelates the parent query from the subqueries (line 10-14) before adding it to the output. Decorrelation serves two purposes: first, it simplifies and improves the performance of the algorithm by avoiding unnecessary overhead of examining the parent query when child queries are being processed; second, it simplifies the *QueryOptimization* and *QueryResolution* algorithms (Sections 4.3 and 5) by relieving them of the complexity of dealing with nested subqueries. During decorrelation, each subquery is assigned a unique *QryID*; its occurrence in the parent subquery is then substituted by a *Skolem query term* $SQ_{QryID}(\vec{E})$, where SQ_{QryID} is a fresh Skolem function, and $\vec{E}$ are all the expressions in the subquery that refer to variables in the for clause of the parent query. The subquery itself is added to the working list L, after $\vec{E}$ is replaced by a set of fresh variables $l_0, \ldots, l_k$. These variables will be substituted back with the actual expressions in the assembly phase.

Input: target query q, set of mapping rules $\mathcal{R}$
1. Initialize list L and Q_s
2. $q_0 = q[root \rightarrow body(\mathcal{R}_{root})]$
3. Mark q_0 top query, and add it to L
4. *while* L is not empty
5. q_c = L.**removeFirst**
6. *if* exists $\{g_i$ in $SK_{SetID}(\ldots)\}$ in q_c.for:
7. $Q_c = q_c[SK_{SetID} \rightarrow body(\mathcal{R}_{SetID})]$
8. Add all queries in Q_c to L and **continue**
9. *else if* q_c.return contains subqueries:
10. *for* each subquery q_{sub}:
11. Assign a unique *QryID* to q_{sub}
12. *let* $\vec{E}$ be the sequence of all distinct expressions in q_{sub} that refer to any variable in q_c.for
13. $q_{sub} = q_{sub}[\vec{E} \rightarrow (l_0, \ldots, l_k)]$ and add q_{sub} to L
14. $q_c = q_c[q_{sub} \rightarrow SQ_{QryID}(\vec{E})]$
15. Add q_c to Q_s and **continue**
16. *else* Add q_c to Q_s and **continue**
Output: the set Q_s of translated and decorrelated queries

Lambda Substitution Rule:
$\{\lambda(l_1, \ldots, l_n).\{E\}\}(e_1, \ldots, e_n) \Rightarrow E[l_1 \rightarrow e_1, \ldots, l_n \rightarrow e_n]$

Union Separation Rule:
for $\ldots, g_i$ in $\{\mathcal{Q}_1 \cup \ldots \cup \mathcal{Q}_n\} \ldots$
where B return r
⇓
for $\ldots, g_i$ in $\{\mathcal{Q}_1\}, \ldots$ where B return r $\cup \ldots \cup$ for $\ldots, g_i$ in $\{\mathcal{Q}_n\}, \ldots$ where B return r

De-Nesting Rule:
for x_1 in $X_1, \ldots, x_n$ in X_n,
g in {for y_1 in $Y_1, \ldots, y_k$ in Y_k where B return r},
z_1 in $Z_1, \ldots, z_m$ in Z_m
where B_0 return r_0
⇓
for x_1 in $X_1, \ldots, x_n$ in X_n, y_1 in $Y_1, \ldots, y_k$ in Y_k,
z_1 in $Z_1[g \rightarrow r], \ldots, z_m$ in $Z_m[g \rightarrow r]$
where $B_0[g \rightarrow r]$ and B return $r_0[g \rightarrow r]$

Record Projection Rule:
$[\ldots, l_i = r_i, \ldots].l_i \Rightarrow r_i$

Figure 6: Algorithm QueryTranslate

Figure 7 sketches several steps during the rewriting of query q_1 of Figure 2. The first query shown is the result of substituting the target root *tgt* using the mapping rule R_0 in Figure 5 and applying record projection afterwards. The target generators are substituted with source generators one by one, until the query becomes a source query[5]. Figure 8 uses the running query q_2 to illustrate decorrelation. After step 1, which substitutes $SK_{tgt.0}$ using rule R_{14}, the translation of the top query itself is completed. Since there is a subquery in the return clause, step 2 decorrelates the subquery from the top query by replacing it with SQ_{201}(s_1.N), where SQ_{201} is a new Skolem function associated with this subquery and s_1.N is the expression in the subquery that refers to the parent variable s_1. The subquery is marked with the $QryID$ (i.e., 201) and all occurrences of s_1.N are replaced with the variable l_0.

4.3 Source Query Optimization

The optimization phase consists of two components: *compilation* and *minimization*. Compilation eliminates the equalities involving Skolem terms that occur in the where clause of a rewriting. Any

[5]Due to space limitation, we ignore here the second term of the union, in R_{14}, as well as in the rest of the rules.

for s in $SK_{tgt.0}()$, c in s.student.Cs, e in $SK_{tgt.1}()$
where c.courseInfo.E = e.eval.E
return [name=s.student.N, course=c.courseInfo.C,
grade=e.eval.G, file=e.eval.F]

(step 1) $\Downarrow$ Substitute $SK_{tgt.0}$

for s' in src_1.students, c in $SK_{tgt.0.0.2}(s'.\text{S}, s'.\text{N})$,
e in $SK_{tgt.1}()$
where c.courseInfo.E = e.eval.E
return [name=s'.N, course=c.courseInfo.C,
grade=e.eval.G, file=e.eval.F]

(steps 2 and 3) $\Downarrow$ Substitute $SK_{tgt.0.0.2}$ and $SK_{tgt.1}$

for s' in src_1.students, c' in src_1.students, e' in src_1.students
where $SK_{125}(c'.\text{S}, c'.\text{N}, c'.\text{C}, c'.\text{G}) = SK_{125}(e'.\text{S}, e'.\text{N}, e'.\text{C}, e'.\text{G})$
and s'.S=c'.S and s'.N=c'.N
return [name=s'.N, course=c'.C, grade=e'.G,
file=$SK_{126}(e'.\text{S}, e'.\text{N}, e'.\text{C}, e'.\text{G})$]

Figure 7: Example steps during translation of q_1

(top) for s in $SK_{tgt.0}()$
return [name=s.student.N,
results= for s' in tgt.students, c in s'.student.Cs,
e' in tgt.evals
where c'.courseInfo.E = e'.eval.E and
s'.student.N = s.student.N
return [result = [grade = e'.eval.G,
file = e'.eval.F]]

(step 1) $\Downarrow$ (substitution)

(top) for s_1 in src_1.students
return [name=s_1.N,
results= for s' in tgt.students, c in s'.student.Cs,
e' in tgt.evals
where c'.courseInfo.E = e'.eval.E and
s'.student.N = s_1.N
return [result = [grade = e'.eval.G,
file = e'.eval.F]]

(step 2) $\Downarrow$ (decorrelation)

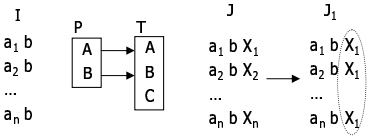
(top) for s_1 in src_1.students
return [name=s_1.N, results= $SQ_{201}(s_1.\text{N})$]
+
$(SQ_{201}(l_0))$
for s' in tgt.students, c in s'.student.Cs, e' in tgt.evals
where c'.courseInfo.E = e'.eval.E and s'.student.N = l_0
return [result = [grade = e'.eval.G, file = e'.eval.F]]

Figure 8: Example steps during translation of q_2

equality between two Skolem terms that have the *same* Skolem function, that is, of the form $F(t_1, \ldots, t_k) = F(t'_1, \ldots, t'_k)$, is replaced by the equalities of their arguments: $t_1 = t'_1$ and $\ldots$ and $t_k = t'_k$. In contrast, if a rewriting contains an equality involving two Skolem terms with *different* Skolem functions or a Skolem term and a non-Skolem term, then the rewriting is marked unsatisfiable (i.e., returning the empty set) and eliminated. This procedure preserves completeness of the algorithm (see Section 4.5) as long as there are no target constraints. Minimization is applied afterwards; it eliminates redundant generators by searching for a minimal subset of generators and, hence, minimal query, that can yield the same answers. The actual procedure is a case of the minimization introduced in [9]. For our query q_1, if we take the rewriting obtained after the translation steps shown in Figure 7 and apply compilation and minimization, we obtain the following minimal rewriting:

for s' in src_1.students
return [name=s'.N, course=s'.C, grade=s'.G,
file=$SK_{126}(s'.\text{S}, s'.\text{N}, s'.\text{C}, s'.\text{G})$]

This is the same as the rewriting r_1 shown in Section 2, provided that we replace the above Skolem function by `null`, which is done whenever the value is nullable. We point out that the queries that result after rewriting often contain redundancies, and minimization is a key component for the efficient evaluation of such queries.



M: foreach p in P
exists t in T
with t.A = p.A and t.B = p.B

c: for t_1 in T, t_2 in T [t_1.B = t_2.B $\rightarrow$ t_1.C = t_2.C]

Figure 9: Mapping scenario with target constraints.

4.4 Query Assembly

After optimization, the set of minimized, decorrelated source queries are assembled back into nested queries in the query assembly phase. First, the top queries are identified; there can be a set of top queries since one top query can be rewritten into a set of queries, all of which are top queries. Each Skolem query term with a given $QryID$ in the return clause of a top query is substituted with the union of all queries that are marked with the same *QryID*. A query is fully assembled when its return clauses (including those inside its subqueries) no longer contain any Skolem query term.

4.5 Soundness, Completeness, & Complexity

The following theorem is a statement of the correctness and completeness of the basic rewriting algorithm with respect to the semantics given by the canonical target instance. We use here the notation and terms introduced in Section 3.

THEOREM 4.1. *Let Σ_{st} be a set of mappings. For every core query q over the target, the basic rewriting algorithm generates r such that: whenever I is a source instance and J is a canonical target instance for I and Σ_{st}, then $r(I) \leq q(J)$ (soundness) and $q(J) \leq r(I)$ (completeness). In particular, $r(I)_{\downarrow} = q(J)_{\downarrow}$ = PNF-certain$_{\Sigma_{st}}(q, I)$, if q is in CQ_0.*

In the above r is a union of rewritings. We next give a bound on the number of rewritings generated by *QueryTranslate* for a given CQ_0 query (either the input query or a query that results after decorrelation).

PROPOSITION 4.2. *If k is the size (number of variables) of an input CQ_0 query and n is the number of mappings then the number of rewritings generated by QueryTranslate is $O(n^k)$.*

5. QUERY RESOLUTION

When target constraints are part of the input, the basic rewriting algorithm becomes incomplete. This means that the condition $q(J) \leq r(I)$ is no longer true, that is, $q(J)$ may contain answers that are not reflected in $r(I)$. We show next how to extend the basic rewriting algorithm in order to handle target constraints. We illustrate the deficiencies of the basic algorithm when it comes to target constraints, and show how a new algorithm called *query resolution* addresses these deficiencies. For the simplicity of discussion, we will use throughout this section a scenario (Figure 9) that is simpler than our previous example.

Rewriting revisited In Figure 9, the A and B columns of a source relation P are mapped via the mapping M into the A and B columns of a target relation T. The C column represents an identifier that exists at the target but not at the source. A target constraint c asserts that B functionally determines C. Figure 9 also shows a possible source instance I, as well as the canonical instance J obtained from I based on the mapping M alone, and the canonical instance J_1 based on M *and* c. In particular, J_1 reflects the fact that C-values are functionally determined by the B-values. The following target query:

(q) for t_1 in T, t_2 in T
where t_1.C = t_2.C
return [$\text{A}_1 = t_1$.A, $\text{A}_2 = t_2$.A]

asks for all pairs of A-values that have the same identifier (C-value). We consider first the case of rewriting q based on M alone. According to the basic query rewriting algorithm, we generate a mapping rule R for M, and then rewrite q by using R into a source query q_s:

(R) T = for p in P return [A=p.A, B=p.B, C=F(p.A, p.B)]

(q_s) for p_1 in P, p_2 in P
where F(p_1.A, p_1.B) = F(p_2.A, p_2.B)
return [A_1 = p_1.A, A_2 = p_2.A]

The equality of the C-values has thus been replaced by an equality of two Skolem terms. In effect, the query q_s incorporates reasoning about incomplete information in the form of equalities involving Skolem functions. In the absence of target constraints, the query compilation algorithm described in Section 4.3 is applied and the equality of the Skolem terms is replaced by the equalities of the arguments. We say that this replacement *resolves* the Skolem term equality. After minimization we obtain:

(r_1) for p_1 in P return [A_1 = p_1.A, A_2 = p_1.A]

The result is a rewriting that, when applied to the source instance I, generates all the "identity" pairs $(a_1, a_1), \ldots, (a_n, a_n)$. This is consistent with the semantics given by the canonical instance J, which is obtained in the absence of c. However, in the presence of c, this set of answers is incomplete. All pairs of the form (a_i, a_j) with $i \neq j$ are also correct answers (consistent with the semantics given by J_1). To obtain these additional answers we make the following crucial observation. When we resolve the equality of the Skolem terms in q_s by replacing it with the equalities of the arguments, we in fact generate a *contained* rewriting, by making use of the fact that (p_1.A = p_2.A and p_1.B = p_2.B) implies F(p_1.A, p_1.B) = F(p_2.A, p_2.B). This implication is a trivial one (i.e. always true). However, in general, it is possible that *additional* contained rewritings exist, because of additional conditions that might imply the equality of the two Skolem terms. We next show how to discover such additional conditions, based on the target constraints. (The completeness part of Theorem 4.1 implies that such rewritings do not exist, if there are no target constraints.)

Rewriting of target NEGDs We rewrite target NEGDs by applying the same translation algorithm (Section 4.2) that we use for queries. We do not apply the compilation algorithm; hence, Skolem term equalities may occur in the left-hand side of the implication, in the resulting constraints. We do, however, apply minimization in order to limit the size of the constraints. The result is a set of constraints over the source schemas that imply additional equalities between Skolem terms. In our example, the NEGD c is rewritten as a constraint that gives us an additional condition (besides the trivial one) under which the two Skolem terms in q_s can be considered equal:

(c_s) for p_1 in P, p_2 in P
[p_1.B = p_2.B → F(p_1.A, p_1.B) = F(p_2.A, p_2.B)]

Resolution step The rewritten constraints are then applied to generate additional ways of resolving equalities involving Skolem functions. For the query q_s and constraint c_s, we can simply add the precondition p_1.B = p_2.B from c_s to the where clause of q_s. We obtain the following rewriting (also contained in q_s, because we only added an extra condition):

for p_1 in P, p_2 in P
where p_1.B = p_2.B and F(p_1.A, p_1.B) = F(p_2.A, p_2.B)
return [A_1 = p_1.A, A_2 = p_2.A]

But then we can immediately drop F(p_1.A, p_1.B) = F(p_2.A, p_2.B), since it is implied by the constraint. We obtain the rewriting r_2 shown below. We call the process of generating r_2 from q_s a *resolution step*. The result of the rewriting algorithm is now $r_1 \cup r_2$ (and not just r_1). We note that $(r_1 \cup r_2)(I)$ includes pairs of the form (a_i, a_j) with $i \neq j$, and in fact $(r_1 \cup r_2)(I) = q(J_1)$. This is true for all instances I. Thus, $r_1 \cup r_2$ is sound and *complete* (according to Section 3).

(r_2) for p_1 in P, p_2 in P
where p_1.B = p_2.B
return [A_1 = p_1.A, A_2 = p_2.A]

For this example, r_1 happens to be contained in r_2 and, hence, it is redundant. In our system, at the end of the resolution phase, we remove redundant rewritings by performing containment checks.

In general, to apply a resolution step, the constraint need not match the query exactly. Hence, the general resolution step is slightly more complicated than explained on this example. We sketch the complete resolution step in Figure 10. We use the notation **p** in **P** to denote a set of generators p_1 in P_1,...,p_n in P_n, and **p** to denote a set of variables $p_1, \ldots, p_n$ (same for **r**). Essentially, the algorithm generates a "join" between the query q_s and the constraint c_s (where the join condition ensures that the two Skolem term equalities in the query and, respectively, constraint, match). Then the Skolem term equality in the query becomes implied, given that the constraint is true, and hence eliminated. Among the newly introduced generators and conditions, not all may be necessary. In the final step we apply a minimization procedure that removes the redundant ones. There, an induced subquery is a query obtained by eliminating a subset of generators from the parent query and then "projecting" the parent where clause to contain all the conditions that involve only remaining variables. Checking the equivalence of r^m with r is performed by checking the existence of containment mappings. The minimization procedure and the equivalence check follow the techniques described in [9]. Although minimization is an exponential procedure, it is important to apply it in order to limit the size of the resulting rewriting. Furthermore, the exponent in our case is given only by the number of generators **r** in **R** and we expect this to be relatively small in practice. The performance of resolution is measured experimentally in the next section.

Input: source query q_s:
for **p** in **P**
where B_1(**p**) and $F(t_1, \ldots, t_l) = G(t'_1, \ldots, t'_k)$
return O(**p**)
source constraint c_s:
for **r** in **R** [B_2(**r**) → $F(e_1, \ldots, e_l) = G(e'_1, \ldots, e'_k)$]
1. Compute equalities to unify the two Skolem term equalities:
Θ(**p**, **r**) = $(t_1 = e_1)$ and ... and $(t_l = e_l)$ and $(t'_1 = e'_1)$ and ... and $(t'_k = e'_k)$
2. Generate contained rewriting r by "unioning" q_s and c_s, and then eliminating the Skolem term equality
for **p** in **P**, **r** in **R**
where B_1(**p**) and B_2(**r**) and Θ(**p**, **r**)
return O(**p**)
3. Find minimal subset (**r_0** in **R_0**) of (**r** in **R**) so that the induced subquery r^m (see below) is equivalent to r.
for **p** in **P**, **r_0** in R_0
where B_1(**p**) and B'_2(**r_0**) and Θ'(**p**, **r_0**)
return O(**p**)
Output: Rewriting r^m.

Figure 10: Algorithm Resolution Step

The above resolution step is applied to eliminate an equality involving Skolem terms in the where clause. Similarly, a resolution step can be used to substitute Skolem terms in the return clause with non-Skolem terms. This style of resolution (not shown here due to space limitation) is used, in addition to the one described above, to generate the rewriting r_3 of Section 2.4 for the query q_1 of our running example.

Resolution phase For a given query, there can be multiple ways in which one resolution step can be applied, for different constraints, and for different Skolem term equalities in the where clause (or the different Skolem terms in the return clause). The *query resolution phase* (*QueryResolution*) is the one in which we explore exhaustively all possible ways of applying resolution steps. For each query q_s that results after query translation, we start by applying one resolution step in all possible ways, and generate multiple rewritings. Each rewriting resolves either one Skolem term equality in the where clause or one Skolem term in the return clause of the given query. We then continue with the new rewritings to generate more rewritings. The computation can be viewed as a tree, whose branching factor is given by the multiple ways in which a resolution step can apply to a query. All the generated rewritings (all the nodes in the tree) are contained rewritings of the original q_s. Some of these queries may be redundant; also, the same query may be generated twice. We eliminate all redundant rewritings. The remaining ones enter the query optimization and assembly phase, as in the basic rewriting algorithm.

Termination of resolution; acyclic constraints Each resolution step eliminates at least one Skolem term equality in the where clause or one Skolem term in the return clause. After the resolution step, it is possible that new Skolem term equalities appear in the where clause of the resulting query. This happens when the applied constraint contains such Skolem term equality in the left-hand side of the implication. The newly introduced Skolem term equalities can then enable additional resolution steps (in order to resolve them). Thus, this process can be recursive in the sense that constraints can enable each other. In order to guarantee termination of resolution, we impose a natural acyclicity condition on the shape of the constraints. Let $\mathcal{F}$ be the set of constraints over the source schema that are obtained by translating the set of target NEGDs, based on the mapping rules. If f_1 and f_2 are constraints in $\mathcal{F}$, we say that f_1 *enables* f_2 and write $f_1 \rightarrow f_2$ if f_1 and f_2 are of the following form:

(f_1) for ... [... F(...) = G(...) ... → ...]
(f_2) for ... [... → ... F(...) = G(...) ...]

where F, G are two Skolem functions (possibly the same). Intuitively, if f_1 is applied in a resolution step to a query, then f_2 becomes applicable in a subsequent resolution step (even though it may not have been applicable before). We construct a directed graph whose nodes are the constraints in $\mathcal{F}$ and whose edges are given by the "enables" relation. We then say that $\mathcal{F}$ is acyclic if the corresponding directed graph is acyclic. In the following, we will restrict ourselves to target NEGDs that, for given schemas and mappings, give rise to an acyclic set $\mathcal{F}$. A typical example of target NEGDs that may violate this condition is the set of two functional dependencies $\{B \rightarrow C, C \rightarrow B\}$ on a target relation T. If $\mathcal{F}$ is acyclic then we can immediately prove that resolution is terminating.

Soundness & complexity The next theorem is a statement of the correctness (soundness) of the rewriting algorithm extended to include the resolution phase.

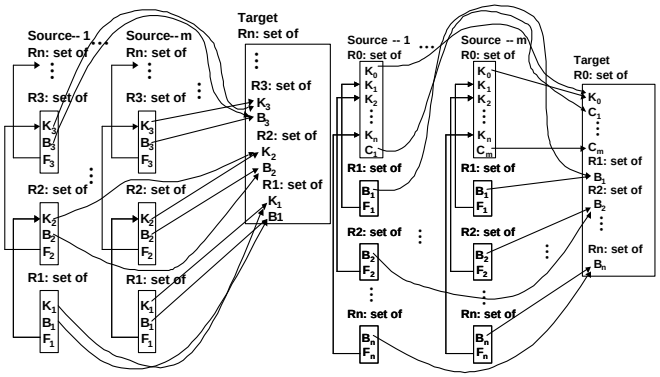
THEOREM 5.1. *Let Σ_{st} and Σ_t be a set of mappings and target NEGDs, respectively. Assume that the set $\mathcal{F}$ of translated constraints is acyclic. For every target CQ query q, the extended algorithm generates r such that: whenever I is a source instance and J is a canonical target instance for I, Σ_{st} and Σ_t, then $r(I) \leq q(J)$ (soundness). In particular, $r(I)_{\downarrow} \subseteq q(J)_{\downarrow}$ = PNF-certain$_{\Sigma_{st} \cup \Sigma_t}(q, I)$, if q is in CQ_0.*

We point out that the extended algorithm may still be incomplete. More precisely, there are examples of schemas, mappings, target functional dependencies, and target CQ query q for which the rewriting r that is generated by the extended algorithm does not satisfy $q(J) \leq r(I)$. This follows from a result from [12] that showed that, for relational settings, one needs recursion in order to obtain the complete set of the certain answers, in the case when functional dependencies are allowed in the target. This holds in our case as well, since the scenarios that we consider generalize the relational LAV scenario of [12]. We note that we did not consider recursion as an acceptable choice for our language of rewritings, as one of the main goals of this work was to generate rewritings that can be expressed using the core fragments of SQL or XQuery and, hence, can be efficiently optimized and executed. Although incomplete in general, we believe that the rewritings generated by the extended algorithm perform well in practice. There are many examples of settings with target NEGDs for which the generated rewritings are in fact complete (the examples shown in this paper, and others). Furthermore, our current experience with the implemented system tells us that even for the cases where the generated rewritings are incomplete, the answers that we get are a pretty close approximation to the complete set of answers. We plan to further validate this claim, experimentally, in our future work. Also, it remains to be seen whether there is a way to bridge the completeness gap in an efficient way.

The following gives a bound on the number of rewritings explored by the extended algorithm, for a given CQ_0 query.

PROPOSITION 5.2. *Let k be the size (number of variables, number of conditions in the* where *clause, and number of expressions in the* return *clause) of an input CQ_0 query. Let M be the maximum size of a mapping (number of variables in the* for *clause), let n be the number of mappings, let s be the maximum size of a target NEGD (number of variables) and let r be the number of target NEGDs. Moreover, assume that the set $\mathcal{F}$ of translated constraints is acyclic. In that case, let f and h be the maximal fan-out and, respectively, the maximum length of a path in the directed acyclic graph associated to $\mathcal{F}$. Then the number of rewritings generated and explored by the extended rewriting algorithm is $O(n^k \times (Mkn^s r)^{Mkf^h})$.*

This number is still a polynomial in the number of mappings, if we consider the other parameters fixed. This is the same as in the case of using *QueryTranslate* alone (although the degree is higher now). In terms of the input query size, the complexity is higher now (exponential in $k \log k$, as opposed to just exponential in k).



Chain Query 1: a single-variable query

for $\$r$ in doc()/Target/R_n return $\$r/B_n$

Chain Query 2: a three-variable query

for $\$r_n$ in doc()/Target/R_n, $\$r_{n-1}$ in $\$r_n/R_{n-1}$, $\$r_{n-2}$ in $\$r_{n-1}/R_{n-2}$
return $\$r_{n-2}/B_{n-2}$

Authority Query 1: an m-way merging query

for $\$r$ in doc()/Target/R_0
return <result> {$\$r/C_1$} {$\$r/C_2$} ... {$\$r/C_m$} </result>

Authority Query 2: a three-variable query

for $\$r_0$ in doc()/Target/R_0, $\$r_1$ in $\$r_0/R_1$, $\$r_n$ in $\$r_0/R_n$,
return <result> {$\$r_1/B_1$} {$\$r_n/B_n$} </result>

Authority Target Constraint:

every $\$r_1$ in doc()/Target/R_0, $\$r_2$ in doc()/Target/R_0
satisfies if $\$r_1/K_0 = \$r_2/K_0$
then $\$r_1/C_1 = \$r_2/C_1$ and ... and $\$r_1/C_m = \$r_2/C_m$

Figure 11: The synthetic "chain" (left) and "authority" (right) scenarios, with queries for both and a target constraint for "authority" scenario. Queries and constraints are written in XQuery syntax (as they are implemented).

6. EXPERIMENTAL EVALUATION

We evaluated the performance of our query rewriting system using a comprehensive set of experiments, including two synthetic mapping scenarios and one real world scenario. We show that the system scales well with the increasing mapping and query complexities in the synthetic scenarios and is capable of efficiently rewriting a set of common queries in the real scenario. The system is implemented in Java and all experiments are performed on a PC-compatible machine, with a single 2.0GHz P4 CPU and 512MB RAM, running Windows XP (SP1) and JRE 1.4.1. Each experiment is repeated five times and the average of the five is used as the measurement.

6.1 The Synthetic Scenarios

We designed two scenarios (shown in Figure 11), **chain** and **authority**, to evaluate the performance of the system along two major dimensions: the *mapping complexity*, measured by the number of elements and referential relationships in a single source schema and the number of independent sources that are mapped to the target schema, and the *query complexity*, measured as the number of levels of nested subqueries (and indirectly, the number of variables) in the

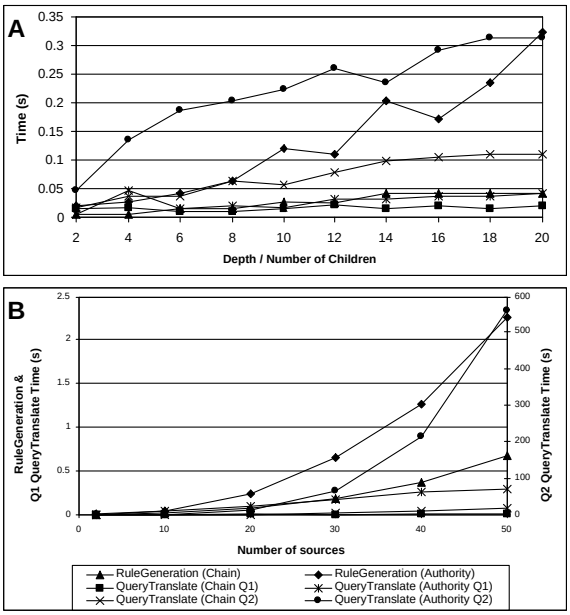


Figure 12: Performance of *RuleGeneration* and *QueryTranslate* with (A) number of sources = 3 and (B) depth or number of children = 3 (Note that in B, the costs for *QueryTranslate* of both Q2 are given on the right y axis).

query. The chain scenario simulates the case where multiple interlinked relational tables are mapped into an XML target with large number of nesting levels (depth). The authority scenario simulates the case where multiple relational tables referencing a central table are mapped into a shallow XML target with a large branching factor (number of children). Three queries were designed for each scenario: two of them (Q1 and Q2) have different number of variables in the for clause (shown in the figure), and one (Q3) has adjustable level of nested subqueries (Q3 is not shown due to space limitation). A target constraint is defined on the authority scenario to be used in evaluating the performance of the *QueryResolution* algorithm. In addition to these two scenarios, we also designed a scenario that is the reverse of the chain scenario, i.e., the XML schemas are used as the sources and the relational schema is used as the target. The results for this scenario are similar to those for the chain scenario and therefore not reported here.

Mapping complexity: Figure 12 shows the performance of *RuleGeneration* and *QueryTranslate* on rewriting Q1 and Q2 in both scenarios with the increasing mapping complexity. The upper limit on the depth or number of children (also called single schema complexity) is set to 20 as we believe this is a reasonably high estimate for real schemas (the system can easily scale up to 40 children or levels deep, costing under 10 seconds for either *RuleGeneration* or *QueryTranslate*). As shown in the figure, the *RuleGeneration* algorithm scales well with both the increasing single schema complexity and the increasing number of sources in the mapping scenario (it takes less than 2.5 seconds with 50 sources and depth/number of children equal to 3). *QueryTranslate* scales well for both queries with the increasing single schema complexity and for the single-variable query with increasing number of sources. For the three-variable query, the cost of *QueryTranslate* increases quickly (but is still acceptable) with the increasing number of sources (around 10 minutes with 50 sources for authority Q2). This is due to the large number of possible ways to substitute a target generator, which produces many potential source queries that are invalidated later. As part of the future work, we are exploring ways of improving the performance of *QueryTranslate* through early detection of unsatisfiable source queries. We also evaluated *QueryOptimization* and *QueryAssemble* algorithms, which show similar performance patterns to the *QueryTranslate* algorithm on a per produced query basis (not shown due to space limitation).

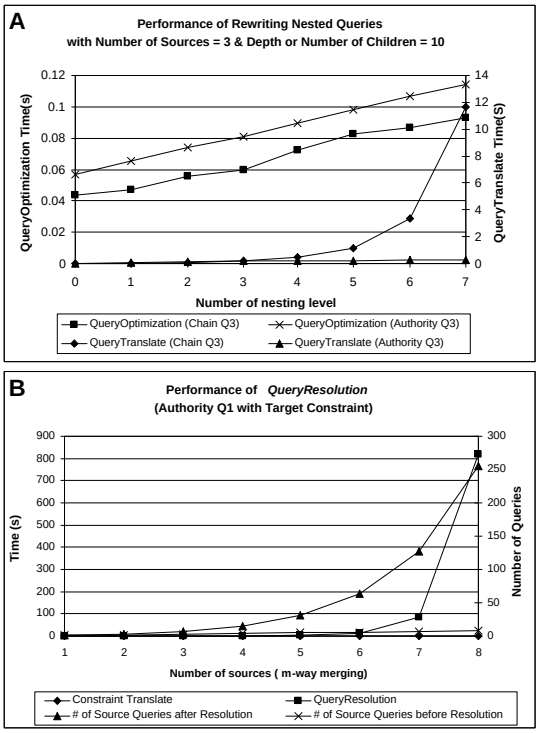


Figure 13: Performance of rewriting nested query (A) and *QueryResolution* (B) (The cost of *QueryOptimization* is measured as per produced query).

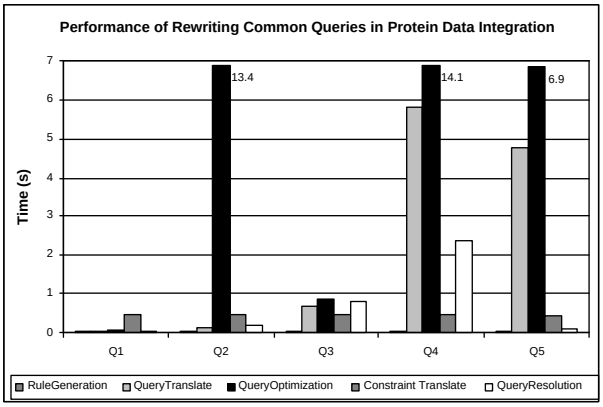
Query complexity: The panel A of Figure 13 shows the performance of rewriting queries with an increasing nesting level (Q3). The subquery at each level contains a single-variable for clause in both chain and authority scenarios. The *QueryTranslate* algorithm scales well with the increasing levels of nesting, taking less than 12 seconds for a 7-level nested query in the chain scenario. The cost is mainly affected by the number of produced (decorrelated) queries: the largest number of produced queries is 9840 for chain Q3 and 840 for authority Q3, which account for the performance differences between the two (y axis on the right). The performance of *RuleGeneration* is not affected by the query complexity.

Performance of QueryResolution: We further evaluated the performance of *QueryResolution* algorithm using the authority scenario with a single child in each source schema and the target constraint shown in Figure 11. The results of rewriting the merging query Q1 are shown in panel B of Figure 13. As expected, the number of valid source queries being produced increases with number of sources. The total time cost for *QueryResolution* algorithm increases accordingly. As a subject of our future work, the cost can be reduced if we know that certain sources can not have complementary data of the same object: any merged source query involving at least two of those non-overlapping sources can not be valid.

6.2 The Life Sciences Scenario

Although synthetic scenarios can help us analyze the behavior of the system, real world examples are necessary to test its practicality. In this regard, we measured the performance of the system using a real data integration scenario from the Life Sciences domain. In a recent project [8], two prominent protein-protein interaction databases, BIND and DIP[6], are integrated into a single centralized database. The mapping scenario (from BIND and DIP to the central target schema) is extracted and five queries that are representatives of those commonly asked by the biologists are chosen for rewriting (shown in Figure 14 is one of the five queries). The three schemas (all are XML schemas) contain a total of about 500 elements with a maximum depth of 17 and a maximum fanout of 13. The mapping scenario

[6] http://www.bind.ca/; http://dip.doe-mbi.ucla.edu/



Q5: Find "NFKB" interaction chains with length 3

```
for   $o in doc()/root/org, $a in $o/obj, $ai in $a/obj_inter,
      $an in $a/obj_name, $b in $o/obj, $bi in $b/obj_inter
where  $an/name = "NFKB" and $ai/inter_with/id = $b/id
return <result>
       { for  $bn in $b/obj_name return $bn/name }
       { for  $c in $o/obj
         where  $bi/inter_with/id = $c/id
         return <names>
                  { for $cn in $c/obj_name return $cn/name }
                </names>
       </result>
```

Figure 14: Rewriting common protein interaction queries

contains 4 logical mappings with 3-10 variables in both foreach and exists clauses for each mapping, and two target key constraints identifying components of a protein (or its interaction partner) given the id. Figure 14 shows the time cost for each component of our system to rewrite the five representative queries. Each of those queries is rewritten into 4-8 valid source queries, while the total number of produced (decorrelated) queries, valid or invalid, ranges from 16 to 2560. Each of the five queries is rewritten by the system in a total of about 60 seconds, with query 4 being the longest one to finish, taking 69.5 seconds.

7. CONCLUSION

This paper presents the first mapping and constraint based XML query rewriting system. Our techniques can be applied in various XML or relational data integration scenarios for answering queries through a virtual target schema. The semantics of such query answering in the presence of both *mappings* and *target constraints* is defined and used as the basis for the system. Mappings can be incomplete, and this gives flexibility to the design of the data integration system. The incorporation of target constraints ensures that various parts of the same data entity, though residing at different sources, are merged and presented to the user as a whole. Two novel algorithms are implemented: the *basic query rewriting algorithm* transforms target queries into source queries based on mappings, while the *query resolution algorithm* generates additional source queries to merge related data based on the constraints. Experimental evaluation shows that the system scales well with increasing complexities of the mapping scenario and the target query, and is practical in a real data integration scenario drawn from the Life Sciences domain. Some of the open questions that remain to be answered are: identifying classes of mappings and target constraints for which resolution is guaranteed to be complete, further performance improvement of the resolution algorithm, and extension to a larger class of queries that would include aggregation and more complex predicates.

8. REFERENCES

[1] S. Abiteboul and N. Bidoit. Non-first Normal Form Relations: An Algebra Allowing Data Restructuring. *JCSS*, 33:361–393, 1986.

[2] B. Amann, C. Beeri, I. Fundulaki, and M. Scholl. Querying XML sources using an ontology-based mediator. In *CoopIS*, 2002.

[3] C. Beeri, A. Y. Levy, and M.-C. Rousset. Rewriting queries using views in description logics. In *PODS*, 1997.

[4] C. Beeri and M. Y. Vardi. A Proof Procedure for Data Dependencies. *J. ACM*, 31(4):718–741, 1984.

[5] M. Benedikt, C. Y. Chan, W. Fan, J. Freire, and R. Rastogi. Capturing both types and constraints in data integration. In *SIGMOD*, 2003.

[6] D. Calvanese, G. D. Giacomo, and M. Lenzerini. View-based query processing for regular path queries with inverse. In *PODS*, 2000.

[7] D. Chamberlin. XQuery: An XML query language. *IBM Systems Journal*, 41(4):597–615, 2002.

[8] A. Chapman, C. Yu, and H. V. Jagadish. Effective integration of protein data through better data modeling. *OMICS: A Journal of Integrative Biology*, 7(1):101–102, 2003.

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

[10] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[11] A. Deutsch and V. Tannen. Reformulation of XML queries and constraints. In *ICDT*, 2003.

[12] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

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

[14] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, 2003.

[15] M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A framework for publishing relational data in XML. *TODS*, 27(4):438–493, 2002.

[16] M. Friedman, A. Levy, and T. Millstein. Navigational Plans for Data Integration. In *AAAI*, 1999.

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

[18] V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: a new flavor of federated query processing for DB2. In *SIGMOD*, 2002.

[19] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL Query Translation Literature: The State of the Art and Open Problems. In *XSym*, 2003.

[20] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, 2002.

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

[22] I. Manolescu, D. Florescu, and D. Kossman. Answering XML queries over heterogeneous data sources. In *VLDB*, 2001.

[23] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, 2000.

[24] Y. Papakonstantinou and V. Vassalos. Rewriting queries using semistructured views. In *SIGMOD*, 1999.

[25] L. Popa and V. Tannen. An equational chase for path-conjunctive queries, constraints, and views. In *ICDT*, 1999.

[26] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *VLDB*, 2002.

[27] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001.

[28] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *VLDB*, 2001.

[29] R. van der Meyden. Logical Approaches to Incomplete Information: A Survey. In *Logics for Databases and Information Systems*, pages 307–356. Kluwer, 1998.