Chapter 1

LOGIC-BASED TECHNIQUES IN DATA INTEGRATION

Alon Y. Levy
Department of Computer Science and Engineering
University of Washington
Seattle, WA, 98195
alon@cs.washington.edu

Abstract

The data integration problem is to provide uniform access to multiple heterogeneous information sources available online (e.g., databases on the WWW). This problem has recently received considerable attention from researchers in the fields of Artificial Intelligence and Database Systems. The data integration problem is complicated by the facts that (1) sources contain closely related and overlapping data, (2) data is stored in multiple data models and schemas, and (3) data sources have differing query processing capabilities.

A key element in a data integration system is the language used to describe the contents and capabilities of the data sources. While such a language needs to be as expressive as possible, it should also enable to efficiently address the main inference problem that arises in this context: to translate a user query that is formulated over a mediated schema into a query on the local schemas. This paper describes several lanaguages for describing contents of data sources, the tradeoffs between them, and the associated reformulation algorithms.

Keywords:

Data integration, description logics, views.

1. INTRODUCTION

The goal of a data integration system is to provide a *uniform* interface to a multitude of data sources. As an example, consider the task of providing information about movies from data sources on the World-Wide Web (WWW). There are numerous sources on the WWW concerning movies, such as the Internet Movie Database (providing comprehensive listings of movies, their casts, directors, genres, etc.), MovieLink (providing playing times of movies in US cities), and several sites providing reviews of selected movies. Suppose we want to find which movies, directed by Woody Allen, are playing tonight in Seattle, and their respective reviews. None of these data sources *in isolation* can answer this query. However, by combining data from multiple sources, we can answer queries like this one, and even more complex ones. To answer our query, we would first search the Internet Movie Database for the list of movies directed by Woody Allen, and then feed the result into the MovieLink database to check which ones are playing in Seattle. Finally, we would find reviews for the relevant movies using any of the movie review sites.

The most important advantage of a data integration system is that it enables users to focus on specifying *what* they want, rather than thinking about *how* to obtain the answers. As a result, it frees the users from the tedious tasks of finding the relevant data sources, interacting with each source in isolation using a particular interface, and combining data from multiple sources.

The main characteristic distinguishing data integration systems from distributed and parallel database systems is that the data sources underlying the system are *autonomous*. In particular, a data integration system provides access to *pre-existing* sources, which were created independently. Unlike multidatabase systems (see Litwin et al., 1990 for a survey) a data integration system must deal with a large and constantly changing set of data sources. These characteristics raise the need for richer mechanisms for describing our data, and hence the opportunity to apply techniques from knowledge representation. In particular, a data integration system requires a flexible mechanism for describing contents of sources that may have overlapping contents, whose contents are described by complex constraints, and sources that may be incomplete or only partially complete.

This paper describes the main languages considered for describing data sources in data integration systems (Section 4.), which are based on extensions of database query languages. We then discuss the novel challenges arising in designing the appropriate reasoning algorithms (Section 5.). Finally, we consider the advantages and challenges of accompanying these languages with a richer knowledge representation formalism based on Description Logics (Section 6.).

This paper is not meant to be a comprehensive survey of the work on data integration or even on languages for describing source descriptions. My goal is simply to give a flavor of the main issues that arise and of the solutions that have been proposed. Pointers to more detailed papers are provided throughout.

2. NOTATION

Our discussion will use the terminology of relational databases. A *schema* is a set of relations. Columns of relations are called attributes, and their names are part of the schema (traditionally, the type of each attribute is also part of the schema but we will ignore typing here).

Queries can be specified in a variety of languages. For simplicity, we consider the language of conjunctive queries (i.e., single Horn rule queries), and several variants on it. A conjunctive query has the form:

$$q(\bar{X}) \text{ :- } e_1(\bar{X}_1), \ldots, e_n(\bar{X}_n),$$

where $e_1, \ldots, e_n$ are database relations, and $\bar{X}_1, \ldots, \bar{X}_n$ are tuples of variables or constants. The atom $q(\bar{X})$ is the head of the query, and the result of the query is a set of tuples, each giving a binding for every variable in $\bar{X}$. We assume the query is safe, i.e., $\bar{X} \subseteq \bar{X}_1 \cup \ldots \cup \bar{X}_n$. Interpreted predicates such as $<, \leq, \neq$ are sometimes used in the query. Queries with unions are expressed by multiple rules with the same head predicate. A *view* refers to a named query, and it is said to be *materialized* if its results are stored in the database.

The notions of query containment and query equivalence are important in order to enable comparison between different formulations of queries.

Definition 1: Query containment and equivalence. A query Q_1 is said to be contained in a query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for any database D, $Q_1(D) \subseteq Q_2(D)$. The two queries are said to be equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.

The problems of query containment and equivalence have been studied extensively in the literature. Some of the cases which are most relevant to our discussion include: containment of conjunctive queries and unions thereof Chandra and Merlin, 1977; Sagiv and Yannakakis, 1981, conjunctive queries with built-in comparison predicates Klug, 1988, and datalog queries Shmueli, 1993; Sagiv, 1988; Levy and Sagiv, 1993; Chaudhuri and Vardi, 1993; Chaudhuri and Vardi, 1994.

3. CHALLENGES IN DATA INTEGRATION

As described in the introduction, the task of a data integration system is to provide a uniform interface to a collection of data sources. The data sources can either be full-fledged database systems (of various flavors:

relational, object-oriented, etc.), legacy systems, or structured files hidden behind some interface program. For the purposes of our discussion we model data sources as containing relations. In this paper (as in most of the research) we only consider data integration systems whose goal is to query the data, and not to perform updates on the sources.

Even though this paper focuses on the problem of modeling data sources and query reformulation, it is worthwhile to first highlight some of the challenges involved in building data integration systems. To do so, we briefly compare data integration systems with traditional database management systems.
Figure 1.1 illustrates the different stages in processing a query in a data integration system.

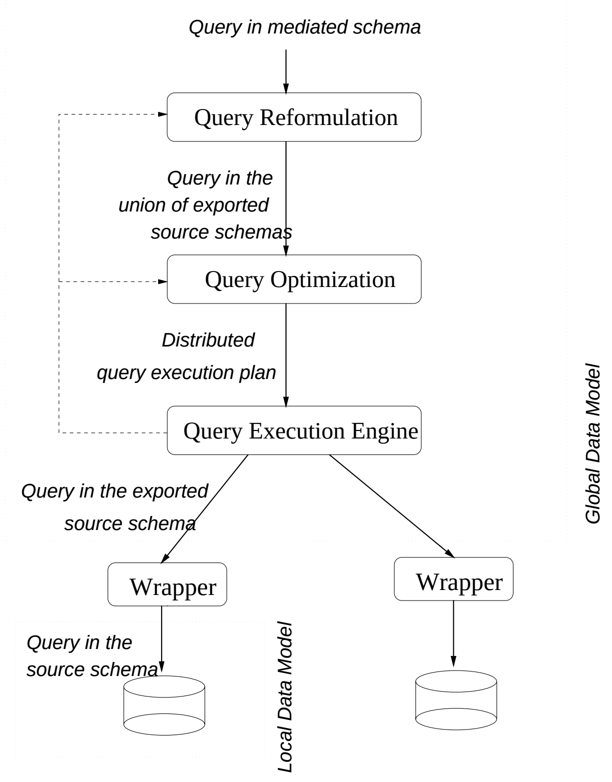
Query in mediated schema
Query Reformulation
Query in the union of exported source schemas
Query Optimization
Distributed query execution plan
Query Execution Engine
Query in the exported source schema
Wrapper
Wrapper
Query in the source schema
Local Data Model
Global Data Model

Figure 1.1 Prototypical architecture of a data integration system

Data modeling:. in a traditional database application one begins by modeling the requirements of the application, and designing a database schema that appropriately supports the application. As noted earlier, a data integration application begins from a set of pre-existing data sources. Hence, the first step of the application designer is to develop a *mediated schema* (often referred to as a *global* schema) that describes the data that exists in the sources, and exposes the aspects of this data that may be of interest to users. Note that the mediated schema does not necessarily contain all the relations and attributes modeled in each of the sources. Users pose queries in terms of the mediated schema, rather than directly in terms of the source schemas. As such, the mediated schema is a set of *virtual* relations, in the sense that they are not actually stored anywhere. For example, in the movie domain, the mediated schema may contain the relation MOVIE(TITLE, YEAR, DIRECTOR, GENRE) describing the different properties of a movie, the relation MOVIEACTOR(TITLE, NAME), representing the cast of a movie, MOVIEREVIEW(TITLE, REVIEW) representing reviews of movies, and AMERICAN(DIRECTOR) storing which directors are American.

Along with the mediated schema, the application designer needs to supply *descriptions* of the data sources. The descriptions specify the relationship between the relations in the mediated schema and those in the local schemas at the sources. The description of a data source specifies its contents (e.g., contains movies), attributes (e.g., genre, cast), constraints on its contents (e.g., contains only American movies), completeness and reliability, and finally, its query processing capabilities (e.g., can perform selections, or can answer arbitrary SQL queries).

The fact that data sources are pre-existing requires that we be able to handle the following characteristics in the language for describing the sources:

1. *Overlapping* and even *contradictory* data among different sources.

2. Semantic mismatches among sources: since each of the data sources has been designed by a different organization for different purposes, the data is modeled in different ways. For example, one source may store a relational database in which all the attributes of a particular movie are stored in one table, while another source may spread the attributes across several relations. Furthermore, the names of the attributes and of the tables will be different from one source to another, as will the choice of what should be a table and what should be an attribute.

3. Different naming conventions for data values: sources use different names or formats to refer to the same object. Simple examples in-

clude various conventions for specifying addresses or dates. Cases in which persons are named differently in the sources are harder to deal with (e.g., one source contains the full name, while another contains only the initials of the first name). A recent elegant treatment of this problem is presented in Cohen, 1998.

Query reformulation:. a user of a data integration system poses queries in terms of the mediated schema, rather than directly in the schema in which the data is stored. As a consequence, a data integration system must contain a module that uses the source descriptions in order to *reformulate* a user query into a query that refers directly to the schemas of the sources. Such a reformulation step does not exist in traditional database systems. Clearly, as the language for describing data sources becomes more expressive, the reformulation step becomes harder. Clearly, we would like the reformulation to be sound, (i.e., the answers to the reformulated query should all be correct answers to the input query), and complete (i.e., all the answers that can be extracted from the data sources should be in the result of applying the reformulated query). In addition, we want the reformulation to produce an efficient query, e.g., to ensure that we do not access irrelevant sources (i.e., sources that cannot contribute any answer or partial answer to the query).

Wrappers:. the other layer of a data integration system that does not exist in a traditional system is the wrapper layer. Unlike a traditional query execution engine that communicates with a local storage manager to fetch the data, the query execution plan in a data integration system must obtain data from remote sources. A wrapper is a program which is specific to a data source, whose task is to translate data from the source to a form that is usable by the query processor of the system. For example, if the data source is a web site, the task of the wrapper is to translate the query to the source's interface, and when the answer is returned as an HTML document, it needs to extract a set of tuples from that document. (Clearly, the emergence of XML as a standard for data exchange on the WWW will alleviate much of the wrapper building problem).

Query optimization and execution:. a traditional relational database system accepts a *declarative* SQL query. The query is first parsed and then passed to the *query optimizer*. The role of the optimizer is to produce an efficient *query execution plan*, which is an imperative program that specifies exactly how to evaluate the query. In particular, the plan specifies the *order* in which to perform the different operations in the query (join, selection, projection), a specific algorithm to use for each operation (e.g., sort-merge join, hash-join), and the scheduling of the different operators. Typically, the optimizer selects a query execution plan by searching a space of possible plans, and comparing their estimated cost. To evaluate the cost of a query execution plan the optimizer relies on extensive statistics about the underlying data, such as sizes of relations, sizes of domains and the selectivity of predicates. Finally, the query execution plan is passed to the query execution engine which evaluates the query.

The main differences between the traditional database context and that of data integration are the following:

- Since the sources are autonomous, the optimizer may have no statistics about the sources, or unreliable ones. Hence, the optimizer cannot compare between different plans, because their costs cannot be estimated.

- Since the data sources are not necessarily database systems, the sources may appear to have different processing capabilities. For example, one data source may be a web interface to a legacy information system, while another may be a program that scans data stored in a structured file (e.g., bibliography entries). Hence, the query optimizer needs to consider the possibility of exploiting the query processing capabilities of a data source. Note that query optimizers in distributed database systems also evaluate where parts of the query should be executed, but in a context where the different processors have identical capabilities.

- Finally, in a traditional system, the optimizer can reliably estimate the time to transfer data from the disc to main memory. But in a data integration system, data is often transferred over a wide-area network, and hence delays may occur for a multitude of reasons. Therefore, even a plan that appears to be the best based on cost estimates may turn out to be inefficient if there are unexpected delays in transferring data from one of the sources accessed early on in the plan.

4. MODELING DATA SOURCES AND QUERY REFORMULATION

As described in the previous section, one of the main differences between a data integration system and a traditional database system is that users pose queries in terms of a mediated schema. The data, however, is stored in the data sources, organized under local schemas. Hence, in order for the data integration system to answer queries, there must be some description of the relationship between the source relations and the mediated schema. The query processor of the integration system must be able to reformulate a query posed on the mediated schema into a query against the source schemas.

In principle, one could use arbitrary formulas in first-order logic to describe the data sources. But in such a case, sound and complete reformulation would be practically impossible. Hence, several approaches have been explored in which restricted forms of first-order formulas have been used in source descriptions, and effective accompanying reformulation algorithms have been presented. We describe two such approaches: the *Global as view* approach (GAV) Garcia-Molina et al., 1997; Papakonstantinou et al., 1996; Adali et al., 1996; Florescu et al., 1996, and the *Local as view* approach (LAV) Levy et al., 1996b; Kwok and Weld, 1996; Duschka and Genesereth, 1997a; Duschka and Genesereth, 1997b; Friedman and Weld, 1997; Ives et al., 1999.

Global As View:. In the GAV approach, for each relation R in the mediated schema, we write a query over the source relations specifying how to obtain R's tuples from the sources.

For example, suppose we have two sources DB_1 and DB_2 containing titles, actors and years of movies. We can describe the relationship between the sources and the mediated schema relation MovieActor as follows:

$$DB_1(id, title, actor, year) \Rightarrow \textsc{MovieActor}(title, actor)$$
$$DB_2(id, title, actor, year) \Rightarrow \textsc{MovieActor}(title, actor).$$

If we have a third source that shares movie identifiers with DB_1 and provides movie reviews, the following sentence describes how to obtain tuples for the MovieReview relation:

$$DB_1(id, title, actor, year) \wedge DB_3(id, review) \Rightarrow \textsc{MovieReview}(title, review)$$

In general, GAV descriptions are Horn rules that have a relation in the mediated schema in the consequent, and a conjunction of atoms over the source relations in the antecedent.

Query reformulation in GAV is relatively straightforward. Since the relations in the mediated schema are defined in terms of the source relations, we need only unfold the definitions of the mediated schema relations. For example, suppose our query is to find reviews for movies starring Marlon Brando:

$$\begin{aligned} q(title, review) :- \ & \text{MovieActor}(title, \text{`Brando'}), \\ & \text{MovieReview}(title, \text{review}). \end{aligned}$$

Unfolding the descriptions of MovieActor and MovieReview will yield the following queries over the source relations: (the second of which will obviously be deemed redundant)

$$\begin{aligned} q(title, review) :- \ & DB_1(id, title, \text{`}Brando', year), DB_3(id, review) \\ q(title, review) :- \ & DB_1(id, title, \text{`}Brando', year), \\ & DB_2(id, \text{`}Brando', year), DB_3(id, review) \end{aligned}$$

Local As View:. The LAV approach the descriptions of the sources are given in the opposite direction. That is, the contents of a data source are described as a query over the mediated schema relation.

Suppose we have two sources: (1) V_1, containing titles, years and directors of American comedies produced after 1960, and (2) V_2 containing movie reviews produced after 1990. In LAV, we would describe these sources by the following formulas (variables that appear only on the right hand sides are assumed to be existentially quantified):

$$\begin{aligned} S_1 : V_1(title, year, director) \Rightarrow \ & \text{Movie(title,year, director,genre)} \wedge \\ & \text{American(director)} \wedge \\ & \text{year} \geq 1960 \wedge \text{genre=Comedy.} \end{aligned}$$

$$\begin{aligned} S_2 : V_2(title, review) \Rightarrow \ & \text{Movie(title, year, director, genre)} \wedge \text{year} \geq 1990 \wedge \\ & \text{MovieReview(title, review).} \end{aligned}$$

Query reformulation in LAV is more tricky than in GAV, because it is not possible to simply unfold the definitions of the relations in the mediated schema. For example, suppose our query asks for reviews for comedies produced after 1950:

$$\begin{aligned} q(title, review) :- \ & \text{Movie}(title, year, director, Comedy), year \geq 1950, \\ & \text{MovieReview}(title, review). \end{aligned}$$

The reformulated query on the sources would be:

$$q'(title, review) :- V_1(title, year, director), V_2(title, review).$$

Note that in this case, the reformulated query is not equivalent to the original query, because it only returns movies that were produced after 1990. However, given that we only have the sources S_1 and S_2, this is the best reformulation possible. In the next section we define precisely the reformulation problem in LAV and present several algorithms for solving it.

A Comparison of the Approaches:. The main advantage of the GAV approach is that query reformulation is very simple, because it reduces to rule unfolding. However, adding sources to the data integration system is non-trivial. In particular, given a new source, we need to figure out all the ways in which it can be used to obtain tuples for each of the relations in the mediated schema. Therefore, we need to consider the possible interaction of the new source with each of the existing sources, and this limits the ability of the GAV approach to scale to a large collection of sources.

In contrast, in the LAV approach each source is described in isolation. It is the system's task to figure out (at query time) how the sources interact and how their data can be combined to answer the query. The downside, however, is that query reformulation is harder, and sometimes requires recursive queries over the sources. An additional advantage of the LAV approach is that it is easier to specify rich constraints on the contents of a source (simply by specifying more conditions in the source descriptions). Specifying complex constraints on sources is essential if the data integration system is to distinguish between sources with closely related and overlapping data. Finally, Friedman et al., 1999 described the GLAV language that combines the expressive power of GAV and LAV, and query reformulation complexity is the same as for LAV.

5. ANSWERING QUERIES USING VIEWS

The reformulation problem for LAV can be intuitively explained as follows. Because of the form of the LAV descriptions, each of the sources can be viewed as containing an *answer* to a query over the mediated schema (an answer to the query expressed by the right hand side of the source description). Hence, sources represent materialized answers to queries over the virtual mediated schema. A user query is also posed over the mediated schema. The problem is therefore to find a way of answering the user query using only the answers to the queries describing the sources.

The problem of answering a query using a set of previously materialized views has received significant attention because of its relevance to other database problems, such as query optimization Chaudhuri et al., 1995, maintaining physical data independence Yang and Larson, 1987; Tsatalos et al., 1996, and data warehouse design.

Formally, suppose we are given a query Q and a set of view definitions $V_1, \ldots, V_m$. A rewriting of the query using the views is a query expression Q' whose subgoals use *only* view predicates or interpreted predicates. We distinguish between two types of query rewritings: *equivalent rewritings* and *maximally-contained rewritings.*

Definition 2: (Equivalent rewritings). Let Q be a query and $\mathcal{V} = V_1, \ldots, V_m$ be a set of view definitions. The query Q' is an equivalent rewriting of Q using V if:

- Q' refers only to the views in $\mathcal{V}$, and
- Q' is equivalent to Q.

Definition 3: (Maximally-contained rewritings). Let Q be a query and $\mathcal{V} = V_1, \ldots, V_m$ be a set of view definitions in a query language $\mathcal{L}$. The query Q' is a maximally-contained rewriting of Q using $\mathcal{V}$ w.r.t. $\mathcal{L}$ if:

- Q' refers only the views in $\mathcal{V}$,
- Q' is contained in Q, and
- there is no rewriting Q_1, such that $Q' \subseteq Q_1 \subseteq Q$ and Q_1 is not equivalent to Q'.

Equivalent rewritings of a query are needed when materialized views are used for query optimization and physical data independence. In the data integration context, we usually need to settle for maximally-contained rewritings, because, as we saw in the previous section, the sources do not necessarily contain all the information needed to provide an equivalent answer to the query.

Recent research has considered many variants of the problem of answering queries using views (see Levy, 1999 for a survey). The problem has been shown to be NP-complete even when the queries describing the sources and the user query are conjunctive and don't contain interpreted predicates Levy et al., 1995. In fact, Levy et al., 1995 shows that in the case of conjunctive queries, we can limit the candidate rewritings that we consider to those that have at most the number of subgoals in the query. Importantly, the complexity of the problem is polynomial in the number of views (i.e., the number of data sources in the context of data integration).

In what follows we describe two algorithms for answering queries using views that were developed specifically for the context of data integration. These algorithms are the *bucket algorithm* developed in the context of the Information Manifold system Levy et al., 1996b, and the *inverse-rules algorithm* Qian, 1996; Duschka and Genesereth, 1997b; Duschka et al., 1999 which was implemented in the InfoMaster system Duschka and Genesereth, 1997b.

5.1 THE BUCKET ALGORITHM

The goal of the bucket algorithm is to reformulate a user query that is posed on a mediated (virtual) schema into a query that refers directly to the available data sources. Both the query and the sources are described by select-project-join queries that may include atoms of arithmetic comparison predicates (hereafter referred to simply as predicates). The bucket algorithm returns the maximally contained rewriting of the query using the views.

The main idea underlying the bucket algorithm is that the (possibly exponential) number of query rewritings that need to be considered can be drastically reduced if we first consider each subgoal in the query in isolation, and determine which views may be relevant to each subgoal. Given a query Q, the bucket algorithm proceeds in two steps. In the first step, the algorithm creates a bucket for each subgoal in Q, containing the views (i.e., data sources) that are relevant to answering the particular subgoal. More formally, a view V is put in the bucket of a subgoal g in the query if the definition of V contains a subgoal g_1 such that

- g and g_1 can be unified, and
- after applying the unifier to the query and to the variables of the view that appear in the head, the predicates in Q and in V are mutually satisfiable.

The actual bucket contains the head of the view V after applying the unifier to the head of the view. Note that a subgoal g may unify with more than one subgoal in a view V, and in that case the bucket of g will contain multiple occurrences of V.

In the second step, the algorithm considers query rewritings that are conjunctive queries, each consisting of one conjunct from every bucket. Specifically, for each possible choice of element from each bucket, the algorithm checks whether the resulting conjunction is contained in the query Q, or whether it can be made to be contained if additional predicates are added to the rewriting. If so, the rewriting is added to the answer. Hence, the result of the bucket algorithm is a union of conjunctive rewritings.

Example 5..1: Consider an example including views over the following schema:

Enrolled(student, dept) Registered(student, course, year)
Course(course, number)

V1(student,number,year) :- Registered(student,course,year),

```
                                  Course(course,number),
                                  number≥500, year≥1992.
V2(student,dept,course) :- Registered(student,course,year),
                                  Enrolled(student,dept)
V3(student,course) :- Registered(student,course,year), year ≤ 1990.
V4(student,course,number) :- Registered(student,course,year),
                                  Course(course,number),
                                  Enrolled(student,dept), number≤100
```

Suppose our query is:

q(S,D) :- Enrolled(S,D), Registered(S,C,Y), Course(C,N), N≥300, Y≥1995.

In the first step of the algorithm we create a bucket for each of the relational subgoals in the query in turn. The resulting contents of the buckets are shown in Table 1.1. The bucket of Enrolled(S,D) will include the views V2 and V4, since the following mapping maps the atom in the query into the corresponding Enrolled atom in the views:

{ S → student, D → dept }.

Note that the view head in the bucket only includes the variables in the domain of the mapping, and fresh variables (primed) for the other head variables of the view.

The bucket of the subgoal Registered(S,C,Y) will contain the views V1, V2 and V4 since the following mapping maps the atom in the query into the corresponding Registered atom in the views:

{ S → student, C → course, Y → year }.

Enrolled(S,D)	Registered(S,C,Y)	Course(C,N)
V2(S,D,C')	V1(S,N',Y)	V1(S',N,Y')
V4(S,C',N')	V2(S,D',C)	
	V4(S,C,N')	

Table 1.1 Contents of the buckets. The primed variables are those that are not in the domain of the unifying mapping.

The view V3 is not included in the bucket of Registered(S,C,Y) because the predicates Y ≥ 1995 and year ≤ 1990 are mutually inconsistent. On the other hand, one may wonder why V4 *is* included in the bucket, since the predicates on the course number in the view and in the query are mutually inconsistent. However, the point to note is that the mapping from the query to the view does not map the variable D to the variable number in V4. Hence, the contradiction does not arise.[1]

Next, consider the bucket of the subgoal Course(C,N). The view V1 will be included in the bucket because of the mapping

$\{$ C $\rightarrow$ course, N $\rightarrow$ number $\}$.

Note that in this case the view V4 is *not* included in the bucket because the unifier does map N to number, and hence the predicates on the course number would be mutually inconsistent.

In the second step of the algorithm, we combine elements from the buckets. In our example, the only combination that would yield a solution is joining V1 and V2 as follows:

q'(S,D) :- V1(S, N, Y), V2(S,D,C), Y$\geq$1995.

The only other option that would not involve a redundant view subgoal in the rewriting, would involve a join between V1 and V4 and is dismissed because the two views contain disjoint numbers of courses (greater than 500 for V1 and less than 100 for V4). In this case, the views V1 and V4 are relevant to the query in *isolation*, but, if joined, produce the empty answer. Finally, the reader should also note that in this example, as usually happens in the data integration context, the algorithm produced a *maximally-contained* rewriting of the query using the views, and not an equivalent rewriting.

5.2 THE INVERSE-RULES ALGORITHM

Like the bucket algorithm, the inverse-rules algorithm was also developed in the context of a data integration system Duschka and Genesereth, 1997b. The key idea underlying the algorithm is to construct a set of rules that *invert* the view definitions, i.e., rules that show how to compute tuples for the database relations from tuples of the views. We illustrate inverse rules with an example.

Suppose we have the following view:

V3(dept, c-name) :- Enrolled(s-name,dept), Registered(s-name,c-name).

We construct one inverse rule for every conjunct in the body of the view:

Enrolled(f_1(dept,X), dept) :- V3(dept,X)
Registered(f_1(Y, c-name), c-name) :- V3(Y,c-name)

Intuitively, the inverse rules have the following meaning. A tuple of the form (dept,name) in the extension of the view V3 is a witness of tuples in the relations Enrolled and Registered. The witness provides only partial information, and hides the rest. In particular, it tells us two things:

- the relation Enrolled contains a tuple of the form (Z, dept), for some value of Z.
- the relation Registered contains a tuple of the form (Z, name), for the *same* value of Z.

In order to express the information that the unknown value of Z is the same in the two atoms, we refer to it using the functional term f_1(dept,name). Note that there may be several values of Z in the database that cause the tuple (dept,name) to be in the join of Enrolled and Registered, but all that matters is that there exists at least one such value.

In general, we create one function symbol for every existential variable that appears in the view definitions, and these function symbols are used in the heads of the inverse rules.

The rewriting of a query Q using the set of views V is the datalog program that includes

- the inverse rules for V, and
- the query Q.

As shown in Duschka and Genesereth, 1997a, the inverse-rules algorithm returns the maximally contained rewriting of Q using V. In fact, the algorithm returns the maximally contained query even if Q is an arbitrary datalog program.

Example 5..2: Suppose our query asks for the departments in which the students of the "Database" course are enrolled,

q(dept) :- Enrolled(s-name,dept), Registered(s-name, "Database")

and the view V3 includes the tuples:

{ (CS, "Database"), (EE, "Database"), (CS, "AI") }

The inverse rules would compute the following tuples:

Registered: { (f_1(CS,"Database"), CS), (f_1(EE,"Database"), EE),
 (f_1(CS,"AI"), CS) }
Enrolled: { (f_1(CS,"Database"),"Database"),
 (f_1(EE,"Database"),"Database",), (f_1(CS,"AI"),"AI") }

Applying the query to these extensions would yield the answers CS and EE.

In this example we showed how functional terms are generated as part of the evaluation of the inverse rules. However, the resulting rewriting can actually be rewritten in such a way that no functional terms appear Duschka and Genesereth, 1997a.

5.3 COMPARISON OF THE ALGORITHMS

There are several interesting similarities and differences between the bucket and inverse rules algorithms that are worth noting. In particular, the step of computing buckets is similar in spirit to that of computing the inverse rules, because both of them compute the views that are relevant to single atoms of the database relations. The difference is that the bucket algorithm computes the relevant views by taking into consideration the *context* in which the atom appears in the query, while the inverse rules algorithm does not. Hence, if the predicates in a view definition entail that the view cannot provide tuples relevant to a query (because they are mutually unsatisfiable with the predicates in the query), then the view will not end up in a bucket. In contrast, the query rewriting obtained by the inverse rules algorithm may result in containing views that are not relevant to the query. However, the inverse rules can be computed once, and be applicable to any query. In order to remove irrelevant views from the rewriting produced by the inverse-rules algorithm we need to apply a subsequent constraint propagation phase (as in Levy et al., 1997; Srivastava and Ramakrishnan, 1992).

The strength of the bucket algorithm is that it exploits the predicates in the query to prune significantly the number of candidate conjunctive rewritings that need to be considered. Checking whether a view should belong to a bucket can be done in time polynomial in the size of the query and view definitions when the predicates involved are arithmetic comparisons. Hence, if the data sources (i.e., the views) are indeed distinguished by having different comparison predicates, then the resulting buckets will be relatively small. In the second step, the algorithm needs to perform a query containment test for every candidate rewriting. The testing problem is π_2^p-complete, but only in the size of the query and the view definition, and hence quite efficient in practice. The bucket algorithm also extends naturally to unions of conjunctive queries, and to other forms of predicates in the query such as class hierarchies. Finally, the bucket algorithm also makes it possible to identify opportunities for interleaving optimization and execution in a data integration system in cases where one of the buckets contains an especially large number of views.

The inverse-rules algorithm has the advantage of being modular. The inverse rules can be computed ahead of time, independent of a specific query. As shown in Duschka and Genesereth, 1997a; Duschka and Levy, 1997, extending the algorithm to handle functional dependencies on the database schema, recursive queries or the existence of binding pattern limitations can be done by adding another set of rules to the resulting rewriting. Unfortunately, the algorithm does not handle arithmetic comparison predicates, and extending it to handle such predicates turns out to be quite subtle. The algorithm produces the maximally-contained rewriting in time that is polynomial in the size of the query and the views (but the complexity of removing irrelevant views has exponential time complexity Levy et al., 1997).

MiniCon Algorithm Pottinger and Levy, 1999 builds on the ideas of both the bucket algorithm and the inverse rules algorithm. Extensive experiments described in Pottinger and Levy, 1999 show that the MiniCon algorithm significantly outperforms both of its predecessors, and scales up gracefully to hundreds and even thousands of views.

5.4 THE NEED FOR RECURSIVE REWRITINGS

An interesting phenomenon in several variants of LAV descriptions is that the reformulated query may actually have to be *recursive* in order to provide the maximal answer. The most interesting of these variants is the common case in which data sources can only be accessed with particular patterns.

Consider the following example, where the first source provides papers (for simplicity, identified by their title) published in AAAI, the second source records citations among papers, and the third source stores papers that have won significant awards. The superscripts in the source descriptions depict the access patterns that are available to the sources. The superscripts contain strings over the alphabet $\{b, f\}$. If a b appears in the i'th position, then the source requires a binding for the i'th attribute in order to produce provide answers. If an f appears in the i'th position, then the i'th attribute may be either bound or not. From the first source we can obtain all the AAAI papers (no bindings required); to obtain data from the second source, we first must provide a binding for a paper and then receive the set of papers that it cites; with the third source we can only query whether a *given* paper won an award, but not ask for all the award winning papers.

$AAAIdb^{f}(X) \Rightarrow AAAIPapers(X)$
$CitationDB^{bf}(X,Y) \Rightarrow Cites(X,Y)$

$AwardDB^b(X) \Rightarrow AwardPaper(X)$

Suppose our query is to find all the award winning papers:

$Q(X) : -AwardPaper(X)$

As the following queries show, there is no finite number of conjunctive queries over the sources that is guaranteed to provide *all* the answers to the query. In each query, we can start from the AAAI database, follow citation chains of length n, and feed the results into the award database. Since we cannot limit the length of a citation chain we need to follow apriori (without examining the data), we cannot put a bound on the size of the reformulated query.

$Q'(X) : -AAAIdb(X), AwardDB(X)$
$Q'(X) : -AAAIdb(V), CitationDB(V, X_1), \ldots, CitationDB(X_n, X),$
$\quad AwardDB(X).$

However, if we consider recursive queries over the sources, we can obtain a finite concise query that provides all the answers, as follows (note that the newly invented relation *papers* is meant to represent the set of all papers reachable from the AAAI database):

$papers(X) : -AAAIdb(X)$
$papers(X) : -papers(Y), CitationDB(Y, X)$
$Q'(X) : -papers(X), AwardDB(X).$

Other cases in which recursion may be necessary are in the presence of functional dependencies on the mediated schema Duschka and Levy, 1997, when the user query is recursive Duschka and Genesereth, 1997a, and, as we show in the next section, when the descriptions of the sources are enriched by description logics Beeri et al., 1997.

Finally, it turns out that slight changes to the form of source descriptions in LAV can cause the problem of answering queries to become NP-hard in the size of the data in the sources Abiteboul and Duschka, 1998. One example of such a case is when the query contains the predicate $\neq$. Other examples include cases in which the sources are known to be *complete* w.r.t. their descriptions (i.e., where the $\Rightarrow$ in the source description is replaced by $\Leftrightarrow$).

6. THE USE OF DESCRIPTION LOGICS

Thus far, we have described the mediated schema and the local schemas as flat sets of relations. However, in many applications we would like to present users with an interface that includes a rich model of the underlying domain, and be able to pose the integration queries over such a model. Furthermore, providing the mapping between source relations and a mediated schema is considerably facilitated when the the mediated schema is part of a rich domain model. Several works have considered the use of a richer representation based on description logics in the mediated schema Arens et al., 1996; Catarci and Lenzerini, 1993; Levy et al., 1996a; Lattes and Rousset, 1998.

Description logics are a family of logics for modeling complex hierarchical structures. In a description logic it is possible to intensionally define sets of objects, called *concepts*, using descriptions built from a set of constructors. For example, the set $\mathsf{Person} \sqcap (\leq 3\ \mathsf{child}) \sqcap (\forall\ \mathsf{child}\ \mathsf{smart})$ describes the set of objects belonging to the class person who have at most 3 children, and all of their children belong to the class smart. In addition to defining concepts, it is also possible to state that an object belongs to a concept. In fact, it is possible to say that Fred belongs to the above concept, *without* specifying who his children are, and how many there are.

As originally suggested in Catarci and Lenzerini, 1993, when the source descriptions and mediated schema both use description logics, it is possible to use the reasoning service of the underlying logic to detect when a source is relevant to a query. However, desciption logics in themselves are not expressive enough to model arbitrary relational data. In particular, they are unable to express arbitrary joins of relations. Hence, several hybrid languages combining the expressive power of Horn rules and description logics have been investigated Levy and Rousset, 1998; Donini et al., 1991; MacGregor, 1994; Cadoli et al., 1997 with associated reasoning algorithms.

Answering queries using views in the context of these hybrid languages is still largely an open problem. As the following example shows, in the presence the rewriting of a query using a set of views may need to be recursive.

Example 6..1: Consider the following two views:

$v_1(X,Y) : -\mathsf{child}(X,Y), \mathsf{child}(X,Z), (\leq\ 1\,\mathsf{child})(Z)$
$v_2(X) : -(\geq\ 3\,\mathsf{child})(X).$

The atom $(\leq\ 1\mathsf{child})(X)$ denotes the set of objects that have at most one child. Similarly, $(\geq\ 3\,\mathsf{child})(X)$ denotes the objects with 3 children or more. Suppose the query Q is to find all individuals who have at least 2 children, i.e.:

$q(X) : -(\geq\ 2\,\mathsf{child})(X).$

For any n, the following conjunctive query is a rewriting that is contained in Q:

$$q'_n(X) \quad :- \quad v_1(X,Y_1), v_1(Y_1,Y_2), v_1(Y_n,U), v_2(U).$$

To see why, consider the variable Y_n. The view v_1 entails that it has one filler for the role child that has less than one child (the variable Z in the definition of v_1) while the view v_2 says that its child U has at least 3 children. Therefore, Y_n has at least 2 children. The same line of reasoning can be used to see that Y_{n-1} has at least 2 children, and continuing in the same way, we get that X has at least two children, i.e., $q(X)$ holds. The point is that inferring $q(X)$ required a chain of inference that involved an arbitrary number of constants, and the length of the chain does not depend on the query or the views. Furthermore, for any value of n, we can build a database such that the union of $q'_1, \ldots, q'_n$ is *not* the maximally-contained rewriting, and therefore we cannot find a maximally-contained rewriting that is a union of conjunctive queries.

We can still find a maximally-contained rewriting in this example if we allow the rewriting to be a recursive datalog program. The following is a maximally-contained rewriting:

$q'(X) : - v_2(X)$
$q'(X) : - v_1(X,Y), q'(Y).$

This recursive program essentially mimics the line of reasoning we described above.

The problem of answering queries using views in the context of description logics is further discussed in Beeri et al., 1997; Calvanese et al., 1999. In Beeri et al., 1997 it is shown that under very tight restrictions, it is always possible to find a maximally-contained rewriting of the query using a set of views, and otherwise, it may not be possible to find a maximally contained rewriting in datalog.

7. CONCLUDING REMARKS

In this article I touched upon some of the problems of data integration in which logic-based methods provided useful solutions. In particular, I showed how semantic relationships between the contents of data sources and relations in a mediated schema can be specified using limited forms of logic, and illustrated the associated reasoning algorithms. There are additional areas in which logic based techniques have been applied, including descriptions of source *completeness* Etzioni et al., 1994; Levy, 1996; Duschka, 1997, and descriptions of query processing capabilities of data sources Levy et al., 1996c; Vassalos and Papakonstantinou, 1997.

There are two areas in which I believe significant future research is required. The first area concerns the role of more expressive knowledge representation languages in data integration. Currently, relatively little is known about the query reformulation problem in cases where the mediated schema and/or the data source schemas are described using a richer knowledge representation language. In addition, we need to carefully study which extensions to the expressive power are really needed in practice.

The second area, which is of wider applicability than the problem of data integration, concerns extending logical reasoning techniques to deal with non-logical extensions that have been needed in database systems. In order to deal with real world applications, commercial database systems provide support for dealing with issues such as bag (multiset) semantics, grouping and aggregation, and nested structures. In order to deal with many real world domains, we need to develop reasoning techniques (extensions of query containment algorithms and algorithms for answering queries using views) to settings where the queries involve bags, grouping and aggregation, and nested structures (see Cohen et al., 1999; Levy and Suciu, 1997; Chaudhuri and Vardi, 1993; Srivastava et al., 1996 for some work in these areas). Another interesting question is whether some of these features can be incorporated into knowledge representation languages.

Notes

1. However, if we also knew that the course name functionally determines its number, then we could prune V4 from this bucket.

References

Abiteboul, S. and Duschka, O. (1998). Complexity of answering queries using materialized views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle, WA.

Adali, S., Candan, K., Papakonstantinou, Y., and Subrahmanian, V. (1996). Query caching and optimization in distributed mediator systems. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Montreal, Canada.

Arens, Y., Knoblock, C. A., and Shen, W.-M. (1996). Query reformulation for dynamic information integration. *International Journal on Intelligent and Cooperative Information Systems*, (6) 2/3:99–130.

Beeri, C., Levy, A. Y., and Rousset, M.-C. (1997). Rewriting queries using views in description logics. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona.

Cadoli, M., Palopoli, L., and Lenzerini, M. (1997). Datalog and description logics: Expressive power. In *Proceedings of the International Workshop on Database Programming Languages.*

Calvanese, D., Giacomo, G. D., and Lenzerini, M. (1999). Answering queries using views in description logics. In *Working notes of the KRDB Workshop.*

Catarci, T. and Lenzerini, M. (1993). Representing and using interschema knowledge in cooperative information systems. *Journal of Intelligent and Cooperative Information Systems.*

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

Chaudhuri, S., Krishnamurthy, R., Potamianos, S., and Shim, K. (1995). Optimizing queries with materialized views. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Taipei, Taiwan.

Chaudhuri, S. and Vardi, M. (1993). Optimizing real conjunctive queries. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 59–70, Washington D.C.

Chaudhuri, S. and Vardi, M. (1994). On the complexity of equivalence between recursive and nonrecursive datalog programs. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 55–66, Minneapolis, Minnesota.

Cohen, S., Nutt, W., and Serebrenik, A. (1999). Rewriting aggregate queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 155–166.

Cohen, W. (1998). Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA.

Donini, F. M., Lenzerini, M., Nardi, D., and Schaerf, A. (1991). A hybrid system with datalog and concept languages. In Ardizzone, E., Gaglio, S., and Sorbello, F., editors, *Trends in Artificial Intelligence*, volume LNAI 549, pages 88–97. Springer Verlag.

Duschka, O. (1997). Query optimization using local completeness. In *Proceedings of the AAAI Fourteenth National Conference on Artificial Intelligence.*

Duschka, O., Genesereth, M., and Levy, A. (1999). Recursive query plans for data integration. *Journal of Logic Programming, special issue on Logic Based Heterogeneous Information Systems.*

Duschka, O. M. and Genesereth, M. R. (1997a). Answering recursive queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona.

Duschka, O. M. and Genesereth, M. R. (1997b). Query planning in infomaster. In *Proceedings of the ACM Symposium on Applied Computing*, San Jose, CA.

Duschka, O. M. and Levy, A. Y. (1997). Recursive plans for information gathering. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence.*

Etzioni, O., Golden, K., and Weld, D. (1994). Tractable closed world reasoning with updates. In *Proceedings of the Conference on Principles of Knowledge Representation and Reasoning, KR-94.* Extended version to appear in *Artificial Intelligence.*

Florescu, D., Raschid, L., and Valduriez, P. (1996). Answering queries using OQL view expressions. In *Workshop on Materialized Views, in cooperation with ACM SIGMOD*, Montreal, Canada.

Friedman, M., Levy, A., and Millstein, T. (1999). Navigational plans for data integration. In *Proceedings of the National Conference on Artificial Intelligence.*

Friedman, M. and Weld, D. (1997). Efficient execution of information gathering plans. In *Proceedings of the International Joint Conference on Artificial Intelligence, Nagoya, Japan.*

Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., and Widom, J. (1997). The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132.

Ives, Z., Florescu, D., Friedman, M., Levy, A., and Weld, D. (1999). An adaptive query execution engine for data integration. In *Proc. of ACM SIGMOD Conf. on Management of Data.*

Klug, A. (1988). On conjunctive queries containing inequalities. *Journal of the ACM*, pages 35(1): 146–160.

Kwok, C. T. and Weld, D. S. (1996). Planning to gather information. In *Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence.*

Lattes, V. and Rousset, M.-C. (1998). The use of the CARIN language and algorithms for information integration: the PICSEL project. In *Proceedings of the ECAI-98 Workshop on Intelligent Information Integration.*

Levy, A. and Rousset, M.-C. (1998). Combining Horn rules and description logics in carin. *Artificial Intelligence*, 104:165–209.

Levy, A. Y. (1996). Obtaining complete answers from incomplete databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India.

Levy, A. Y. (1999). Answering queries using views: A survey. Submitted for publication.

Levy, A. Y., Fikes, R. E., and Sagiv, S. (1997). Speeding up inferences using relevance reasoning: A formalism and algorithms. *Artificial Intelligence*, 97(1-2).

Levy, A. Y., Mendelzon, A. O., Sagiv, Y., and Srivastava, D. (1995). Answering queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA.

Levy, A. Y., Rajaraman, A., and Ordille, J. J. (1996a). Query answering algorithms for information agents. In *Proceedings of AAAI.*

Levy, A. Y., Rajaraman, A., and Ordille, J. J. (1996b). Querying heterogeneous information sources using source descriptions. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India.

Levy, A. Y., Rajaraman, A., and Ullman, J. D. (1996c). Answering queries using limited external processors. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Montreal, Canada.

Levy, A. Y. and Sagiv, Y. (1993). Queries independent of updates. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 171–181, Dublin, Ireland.

Levy, A. Y. and Suciu, D. (1997). Deciding containment for queries with complex objects and aggregations. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona.

Litwin, W., Mark, L., and Roussopoulos, N. (1990). Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22 (3):267–293.

MacGregor, R. M. (1994). A description classifier for the predicate calculus. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 213–220.

Papakonstantinou, Y., Abiteboul, S., and Garcia-Molina, H. (1996). Object fusion in mediator systems. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India.

Pottinger, R. and Levy, A. (1999). A scalable algorithm for answering queries using views. Submitted for publication.

Qian, X. (1996). Query folding. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 48–55, New Orleans, LA.

Sagiv, Y. (1988). Optimizing datalog programs. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, Los Altos, CA.

Sagiv, Y. and Yannakakis, M. (1981). Equivalence among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655.

Shmueli, O. (1993). Equivalence of datalog queries is undecidable. *Journal of Logic Programming*, 15:231–241.

Srivastava, D., Dar, S., Jagadish, H. V., and Levy, A. Y. (1996). Answering SQL queries using materialized views. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India.

Srivastava, D. and Ramakrishnan, R. (1992). Pushing constraint selections. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Diego, CA.

Tsatalos, O. G., Solomon, M. H., and Ioannidis, Y. E. (1996). The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2):101–118.

Vassalos, V. and Papakonstantinou, Y. (1997). Describing and using query capabilities of heterogeneous sources. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 256–265, Athens, Greece.

Yang, H. Z. and Larson, P. A. (1987). Query transformation for PSJ-queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 245–254, Brighton, England.