Answering Queries Using Views: A Survey

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

Abstract

The problem of answering queries using views is to find efficient methods of answering a query using a set of previously materialized views over the database, rather than accessing the database relations. The problem has recently received significant attention because of its relevance to a wide variety of data management problems. In query optimization, finding a rewriting of a query using a set of materialized views can yield a more efficient query execution plan. To support the separation of the logical and physical views of data, a storage schema can be described using views over the logical schema. As a result, finding a query execution plan that accesses the storage amounts to solving the problem of answering queries using views. Finally, the problem arises in data integration systems, where data sources can be described as precomputed views. This article surveys the state of the art on the problem of answering queries using views, and synthesizes the disparate works into a coherent framework. We describe the different applications of the problem, the algorithms proposed to solve it and the relevant theoretical results.

1 Introduction

The problem of answering queries using views (a.k.a. rewriting queries using views) has recently received significant attention because of its relevance to a wide variety of data management problems: query optimization, maintenance of physical data independence, data integration and data warehouse design. Informally speaking, the problem is the following. Suppose we are given a query Q over a database schema, and a set of view definitions $V_1, \ldots, V_n$ over the same schema. Is it possible to answer the query Q using *only* the answers to the views $V_1, \ldots, V_n$? Alternatively, what is the cheapest query execution plan for Q assuming that in addition to the database relations, the answers to $V_1, \ldots, V_n$ are also available?

The first class of applications in which we encounter the problem of answering queries using views is query optimization and database design. In the context of query optimization, computing a query using previously materialized views can speed up query processing because part of the computation necessary for the query may have already been done while computing the views. In fact, in some cases, certain indices can be modeled as precomputed views (e.g., join indices [Val87]), and deciding which indices to use requires a solution to the query rewriting problem. In the context of database design, view definitions provide a mechanism for supporting the independence of the *physical* view of the data and its *logical* view. This independence enables us to modify the storage schema of the data (i.e., the physical view) without changing its logical schema, and to model more complex types of indices. Hence, several authors proposed to describe the storage schema as a set of views over the logical schema [YL87, TSI96, Flo96]. Given these descriptions of the storage, the problem of computing a query execution plan (which, of course, must access the physical storage) involves figuring out how to use the views to answer the query.

A second class of applications in which our problem arises is data integration. Data integration systems provide a uniform query interface to a multitude of autonomous data sources, which may reside within an enterprise or on the World-Wide Web. Data integration systems free the user from having to locate sources relevant to a query, interact with each one in isolation, and manually combine data from multiple sources. Users of data integration systems do not pose queries in terms of the schemas in which the data is stored, but rather in terms of a *mediated schema*. The mediated schema is a set of relations that is designed for a specific data integration application, and contains the salient aspects of the domain under consideration. The tuples of the mediated schema relations are not actually stored in the data integration system. Instead, the system includes a set of *source descriptions,* that provide semantic mappings between the relations in the source schemas and the relations in the mediated schema.

The data integration systems described in [LRO96, DG97b, KW96, LKG99] follow an approach in which the contents of the sources are described as views over the mediated schema. As a result, the problem of reformulating a user query, posed over the mediated schema, into a query that refers directly to the source schemas becomes the problem of answering queries using views. In a sense, the data integration context can be viewed as an extreme case of the need to maintain physical data independence, where the logical and physical layout of the data sources has been defined in advance. The solutions to the problem of answering queries using views differ in this context because the number of views (i.e., sources) tends to be much larger, and the sources need not contain the *complete* extensions of the views.

The problem of answering queries using views arises in the area of data warehouse and web-site design, where we need to choose a set of views to materialize in the data warehouse [HRU96, TS97, YKL97, GHRU97] or a set of views to precompute in order to improve the performance of a web site [FLSY99]. As a first step in choosing an optimal design for the data warehouse we must ensure that the chosen views can be used to answer the queries we expect to receive over the data warehouse. This problem, again, translates into the view rewriting problem. Finally, answering queries using views plays a key role in developing methods for semantic data caching in client-server systems [DFJ$^+$96, KB96, CR94, ACPS96]. In these works, the data cached at the client is modeled semantically as a set of queries, rather than at the level of particular data pages or tuples. Hence, deciding which data needs to be shipped from the server in order to answer a given query requires an analysis of which parts of the query can be answered by the cached views.

The various applications of the problem of answering queries using views has spurred a flurry of research, ranging from theoretical foundations, algorithm design and implementation in commercial systems. This article surveys the current state of the art in this area, and synthesizes the disparate works into a coherent framework.

The focus of the works in the two classes of applications mentioned above differed because of their expected output. In the case of query optimization and database design, the focus has been on producing a query execution plan that involves the views, and hence the effort has been on extending System-R style optimization to accommodate the presence of views. In the data integration context, the focus has been on translating queries formulated in terms of a mediated schema into queries formulated in terms of data sources. Hence, the output of the algorithm is a query on the data sources, rather than a query execution plan. Furthermore, the works on data integration distinguished the translation problem from the more general problem of finding all the answers to a query given the data in the sources, and showed that the two problems differ in interesting ways.

The survey is organized as follows. Section 2 presents in more detail the applications motivating the study of the problem and Section 3 defines the different aspects of the problem formally. As a basis for the discussion of the different algorithms, Section 4 provides an intuitive explanation of the conditions under which a view can be used to answer a query. Section 5 considers the first class of applications of answering queries using views, and describes how a dynamic-programming style optimizer can be modified in order to incorporate the usage of materialized views. Section 6 describes algorithms for answering queries using views that were developed in the context of data integration, and are especially suited for situations where the number of views is large. Section 7 surveys some theoretical issues concerning the problem of answering queries using views, and Section 8 discusses several extensions to the algorithms in Sections 5 and 6 to accommodate queries over object-oriented databases, queries with grouping and aggregation and views with binding pattern limitations. Finally, Section 9 concludes, and outlines some of the open problems in this area.

We note that this survey is not concerned with the closely related problems of incremental maintenance of materialized views, which is surveyed in [GM99b], and selection of which views to maintain in a data warehouse [HRU96, TS97, GHRU97, YKL97].

2 Motivation and Illustrative Examples

Before beginning the detailed technical discussion, this section motivates the problem of answering queries using views through some of the applications in which it arises. We use the following familiar university schema in our examples throughout the paper. We assume that professors and students are uniquely identified by their names, and courses by their titles.

Prof(name, area)
Teaches(prof, course, quarter)
Registered(student, course, quarter)
Course(title, number)

2.1 Query Optimization

The first and most obvious motivation for considering the problem of answering queries using views is its use for query optimization. If part of the computation needed to answer a query has already been performed in computing a materialized view, then we can use the view to speed up the computation of the query.

Consider the following query, asking for students who attended Ph.D-level classes taught by professors in the Database area: (graduate level classes have numbers 400 and above, and courses exclusively for Ph.D students have course numbers of 500 and above).

```
select   student
from     Teaches, Prof, Registered, Course
where    Prof.name=Teaches.prof and Teaches.course=Registered.course and
         Teaches.quarter=Registered.quarter and Registered.course=Course.title and
         Course.number ≥ 500 and Prof.area="DB".
```

Suppose we have the following materialized view, containing the attendance records of graduate level courses and above.

```
create view Graduate as
select   student, course, number, quarter
from     Registered, Course
where    Registered.course=Course.title and Course.number ≥ 400.
```

The view Graduate can be used in the computation of the above query as follows:

```
select   student
from     Teaches, Prof, Graduate
where    Prof.name=Teaches.prof and
         Teaches.course=Graduate.course and Teaches.quarter=Graduate.quarter and
         Graduate.number ≥ 500 and Prof.area="DB".
```

The resulting evaluation will be cheaper because the view Graduate as already performed the join between Registered and Course, and has already pruned the non-graduate courses (which in many cases actually accounts for most of the activity going on in a typical university). It is important to note the view Graduate is useful for answering the query even though it does not *syntactically* match any of the subparts of the query.

Even if a view has already computed part of the query, it is not necessarily the case that using the view will lead to a more efficient evaluation plan, especially considering the indexes available on the database relations and on the views. For example, suppose the relation Course has an index on the number attribute and the relation Registered has an index on the course attribute. In this case, if the view Graduate does not have any indexes, then evaluating the query directly from the database relations will be cheaper. Hence, the challenge is not only to detect when a view is logically usable for answering a query, but also to make a judicious cost-based decision on when to use the available views.

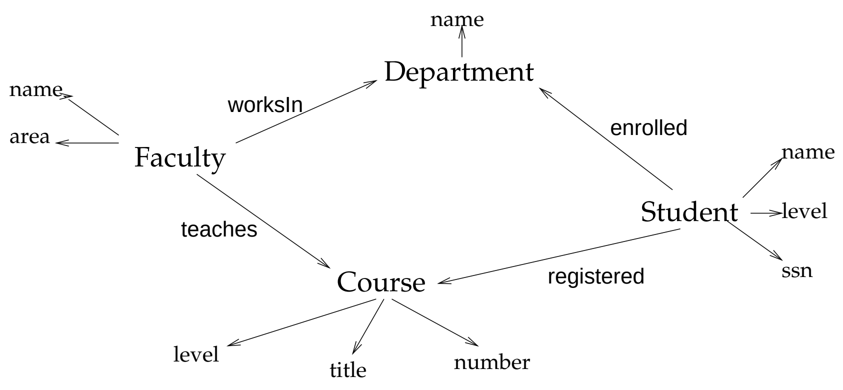

Figure 1: An Entity/Relationship diagram for the university domain.

2.2 Maintaining Physical Data Independence

Several of the works on answering queries using views were inspired by the goal of maintaining physical data independence in relational and object-oriented databases [YL87, TSI96, Flo96]. One of the principles underlying modern database systems is the separation between the logical view of the data (e.g., as tables with their named attributes) and the physical view of the data (i.e., how it is layed out on disk). Relational database systems, even though they provide this separation, are still largely based on a 1-1 correspondence between relations in the schema and files in which they are stored. In object-oriented systems, maintaining the separation is necessary because the logical schema contains significant redundancy, and does not correspond to a good physical layout. Maintaining physical data independence becomes more crucial in applications where the logical model is introduced as an intermediate level after the physical representation has already been determined. This is common in applications of semi-structured data [Bun97, Abi97, FLM98], storage of XML data in relational databases [FK99, SGT$^+$99], and in data integration. In some sense, data integration, discussed in the next section, is an extreme case where there is a separation between the logical view of the data and its physical view.

To maintain physical data independence, several authors proposed to use views as a mechanism for describing the storage of the data. In particular, [TSI96] described the storage of the data using GMAPs *(generalized multi-level access paths)*, expressed over the conceptual model of the database.

To illustrate, consider the entity-relationship model of a slightly extended university domain shown in Figure 1. Figure 2 shows GMAPs expressing the different storage structures for this data.

A GMAP describes the physical organization and indexes of the storage structure. The first clause of the GMAP (the given clause) describes the actual data structure used to store a set of tuples (e.g., a B-tree, hash file etc.) The remaining clauses describe the content of the structure, much like a view definition. The given and select clauses describe the available attributes, where the given clause describes the attributes on which the structure is indexed.

```
def_gmap G1 as btree by
    given Student.name
    select Dept
    where  Student enrolled Dept.
```

```
def_gmap G2 as btree by
    given Student.name
    select Course, Course.level
    where Student Registered Course.
```

```
def_gmap G3 as btree by
    given Course.level
    select Dept, Course
    where Student Registered Course and Student enrolled Dept.
```

Figure 2: GMAPs for the university domain.

The definition of the view, given in the where clause uses infix notation over the conceptual model.

In our example, the GMAP G1 specifies the storage of a set of pairs, containing students and the departments to which they belong, indexed by a B^+-tree on attribute Student.name. The GMAP G2 stores an index from the names of students to the courses in which they are registered, and the levels of these courses. The GMAP G3 stores an index from course levels to courses at that level and the departments from which students have attended these courses. As shown in [TSI96], using GMAPs it is possible to express a large family of data structures, including secondary indexes on relations, nested indexes, collection based indexes and structures implementing field replication.

Given that the data is stored in the structures described by the GMAPs, the question that arises is how to use the structures to answer queries. Since the logical content of the GMAPs are described by views, answering a query amounts to finding a way of rewriting the query using these views. If there are multiple ways of answering the query using the views, we would like to find the most efficient one. Note that in contrast to the query optimization context, since all the data is stored in the GMAPs, we *must* use the views to answer a given query.

Consider the following query in our domain, asking for names of students registered for Ph.D-level courses, and the departments in which these students are enrolled.

```
select    Student.name, Dept
where     Student Registered Course and
          Student enrolled Dept and Course.number=500.
```

The query can be answered in two ways. First, assuming the Student.name uniquely identifies a student, we can take the join of G1 and G2, and then apply a selection on Course.number, and a projection on Student.name and Dept. A second solution would be to also join G3 with G1 and G2. In fact, this solution may even be more efficient because G3 has an index on the course number to facilitate the selection in the query.

2.3 Data Integration

Much of the recent work on answering queries using views has been spurred because of its applicability to data integration systems. A data integration system (a.k.a. a mediator system) provides a *uniform* query interface to a multitude of autonomous heterogeneous data sources. Prime examples of data integration applications include enterprise integration, querying multiple sources on the World-Wide Web, and integration of data from distributed scientific experiments. The sources in such an application may be traditional databases, legacy systems, or even structured files. The goal of a data integration system is to free the user from having to find the data sources relevant to a query, interact with each source in isolation, and manually combine data from the different sources.

To provide a uniform interface, a data integration system exposes to the user a *mediated schema*. A mediated schema is a set of *virtual* relations, in the sense that they are not actually stored anywhere. The mediated schema is designed manually for a particular data integration application. To be able to answer queries, the system must also contain a set of *source descriptions*. A description of a data source specifies the contents of the source, the attributes that can be found in the source, and constraints on the contents of the source.

One of the approaches for specifying source descriptions, which has been adopted in several systems ([LRO96, KW96, FW97, DG97b, LKG99]), is to describe the contents of a data source as a *view* over the mediated schema. This approach facilitates the addition of new data sources and the specification of constraints on contents of sources (see [Ull97, FLM98] for a comparison of different approaches for specifying source descriptions).

In order to answer a query, a data integration system needs to translate a query formulated on the mediated schema into one that refers directly to the schemas in the data sources. Since the contents of the data sources are described as views, the translation problem amounts to finding a way to answer a query using a set of views.

We illustrate the problem with the following example. Continuing with our university domain, suppose the data integration system exposes the following schema, meant to support various queries on a collection of universities.

```
Teaches(prof, course-number, univ)
Course(title, univ, number)
```

Suppose we have the following two data sources. The first provides a listing of all the courses titled "Database Systems" taught in the country and their instructors. This source can be described by the following view definition:

```
create view DB-courses as
select    Course.title, Teaches.prof, Course.number, Course.univ
from      Teaches, Course
where     Teaches.course-number=Course.number and Teaches.univ=Course.univ and
          Course.title="Database Systems".
```

The second source describes the Ph.D level courses being taught at UW, and is described by the following view definition:

```
create view UW-phd-courses as
select    Course.title, Teaches.prof, Course.number, Course.univ
from      Teaches, Course
where     Teaches.course-number=Course.number and
          Course.univ="UW" and Teaches.univ="UW" and Course.number≥500.
```

If we were to ask the data integration system who teaches Database courses at UW, it would be able to answer it by applying a selection on the source DB-courses:

```
select    prof
from      DB-courses
where     univ="UW".
```

On the other hand, suppose we ask for all the graduate-level courses (i.e., number 400 and above) being offered at UW. Given that only these two sources are available, the data integration system cannot find *all* the answers to the query. Instead, the system can attempt to find the maximal set of answers available from the sources. In particular, the system can obtain graduate database courses at UW from the DB-courses source, and the Ph.D level courses at UW from the UW-Phd-courses source. Hence, the following query provides the maximal set of answers that can be obtained from the two sources:

```
select    title, number
from      DB-courses
where     univ="UW" and number≥400
       union
select    title, number
from      UW-phd-courses.
```

Note that courses that are not Ph.D-level courses or database courses will not be returned as answers. Whereas in the contexts of query optimization and maintaining physical data independence the focus is on finding a query expression that is *equivalent* to the original query, here we attempt to find a query expression that provides the *maximal answers* from the views. We formalize both of these notions in Section 3.

Before proceeding, we also note that the problem of answering queries using views arises in the design of data warehouses (e.g., [HRU96, TS97, GHRU97, YKL97]) and in semantic data caching. In data warehouse design, when we choose a set of views to materialize in a data warehouse, we need to check that we will be able to answer all the required queries over the warehouse using only these views. In the context of semantic data caching (e.g., [DFJ$^+$96, KB96, CR94, ACPS96]) we need to check whether the cached results of a previously computed query can be used for a new query, or whether the client needs to request additional data from the server. In [FLSY99] it is shown that precomputing views can significantly speed up the response time from web sites, which again raises the question of view selection.

3 Problem Definition

In this section we define the basic terminology used throughout this paper. We begin by recalling the query languages we consider, defining the concepts of query containment and query equivalence that enable comparing between queries. Finally, we define the problem of answering queries using views, and the problem of extracting *all* the answers to a query from a set of views.

3.1 Queries and Views

The bulk of our discussion will focus on the class of select-project-join queries. We assume the reader is familiar with basic elements of SQL. As we see in Section 8, in some cases answering a query using a set of views may require that the rewriting be recursive. Hence, below we provide a brief reminder of datalog notation and of conjunctive queries [Ull89].

A conjunctive query has the form:

$$q(\bar{X}) :- \ r_1(\bar{X}_1), \ \ldots, \ r_n(\bar{X}_n)$$

where q, and $r_1, \ldots, r_n$ are predicate names. The predicate names $r_1, \ldots, r_n$ refer to database relations. The atom $q(\bar{X})$ is called the *head* of the query, and refers to the answer relation. The atoms $r_1(\bar{X}_1), \ldots, r_n(\bar{X}_n)$ are the *subgoals* in the body of the query. The tuples $\bar{X}, \bar{X}_1, \ldots, \bar{X}_n$ contain either variables or constants. We require that the query be *safe*, i.e., that $\bar{X} \subseteq \bar{X}_1 \cup \ldots \cup \bar{X}_n$ (that is, every variable that appears in the head must also appear in the body).

Queries may also contain subgoals whose predicates are arithmetic comparisons $<, \leq, =, \neq$. In this case, we require that if a variable X appears in a subgoal of a comparison predicate, then X must also appear in an ordinary subgoal.

As an example of expressing an SQL query in datalog, consider the following SQL query asking for the students (and their advisors) who took courses from their advisors after the winter of 1998:

```
select   Advises.prof, Advises.student, Registered.quarter
from     Registered, Teaches, Advises
where    Registered.course=Teaches.course and Registered.quarter=Teaches.quarter and
         Advises.prof=Teaches.prof and Advises.student=Registered.student and
         Registered.quarter > "winter98".
```

In the notation of conjunctive queries, the above query would be expressed as follows:

```
q(prof, student, quarter) :-Registered(student, course, quarter), Teaches(prof, course, quarter),
                            Advises(prof, student), quarter > "winter98".
```

Note that when using conjunctive queries, join predicates of SQL are expressed by multiple occurrences of the same variable in different subgoals of the body (e.g., the variables quarter, course and student above). Unions can be expressed in this notation by allowing a set of conjunctive queries with the same head predicate.

A datalog query is a set of rules, each having the same form as a conjunctive query, except that predicates in the body do not have to refer to database relations. In a datalog query we distinguish EDB predicates that refer to the database relations from the IDB predicates that refer to intermediate relations. Hence, in the rules, EDB predicates appear only in the bodies, whereas the IDB predicates may appear anywhere. We assume that every datalog query has a distinguished IDB predicate called the *query predicate*, referring to the relation of the result.

A predicate p in a datalog program is said to *depend* on a predicate q if q appears in one of the rules whose head is p. The datalog program is said to be *recursive* if there is a cycle in the dependency graph of predicates. A non-recursive datalog program can be equivalently rewritten as a union of conjunctive queries, though possibly with an exponential blowup in the size of the query.

The input to a datalog query Q consists of a database D storing extensions of all EDB predicates in Q. Given such a database D, the answer to Q, denoted by $Q(D)$ is the least fixpoint model of Q and D, which can be computed as follows. We apply the rules of the program in an arbitrary order. An application of a rule may derive new tuples for the relation denoted by the predicate in the head of the rule. We apply the rules until we cannot derive any new tuples. The output of Q is the set of tuples computed for the query predicate. Note that since a datalog program does not contain function symbols, the evaluation is guaranteed to terminate.

3.2 Containment and Equivalence

The notions of query containment and query equivalence enable comparison between different reformulations of queries. They will be used when we test the correctness of a rewriting of a query in terms of a set of views. In the definitions below we assume the answers to queries are sets of tuples. The definitions can be extended in a straightforward fashion to multiset semantics.

Definition 3.1 (**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, the set of tuples computed for Q_1 is a subset of those computed for Q_2, i.e., $Q_1(D) \subseteq Q_2(D)$. The two queries are said to be equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$. $\square$

The problems of query containment and equivalence have been studied extensively in the literature and should be a topic of a specialized survey. Some of the cases which are most relevant to our discussion include: containment of conjunctive queries and unions thereof [CM77, SY81], conjunctive queries with built-in comparison predicates [Klu88, LS93, ZO93, KMT98], and datalog queries [Shm93, Sag88, LS93, CV93, CV94].

3.3 Rewriting of a Query Using Views

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' that refers *only* to the views $V_1, \ldots, V_m$. In SQL, a query refers only to the views if all the relations mentioned in the from clause are views. A datalog query refers only to views if instead of EDB predicates we have predicates referring to views (but we still have arithmetic comparison predicates and IDB predicates). In practice, we may also be interested in rewritings that can also refer to the database relations, but this case does not introduce new difficulties. (Note that rewritings that refer only to the views were called *complete rewritings* in [LMSS95]).

As we saw in our examples, we need to distinguish between two types of query rewritings: *equivalent rewritings* and *maximally-contained rewritings.* For query optimization and maintaining physical data independence we consider equivalent rewritings. Note that formally, in order to determine equivalence or containment between the Q and the rewriting Q', we need to take into consideration the view definitions, i.e., to compare Q with $Q' \cup \mathcal{V}$.

Definition 3.2 (**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 $\mathcal{V}$ if:

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

□

In the context of data integration, we often need to consider maximally-contained rewritings:

Definition 3.3 (**Maximally-contained rewritings**) Let Q be a query, $\mathcal{V} = V_1, \ldots, V_m$ be a set of view definitions, and $\mathcal{L}$ be a query language. The query Q' is a maximally-contained rewriting of Q using $\mathcal{V}$ w.r.t. $\mathcal{L}$ if:

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

□

When a rewriting Q' is contained in Q but is not a maximally-contained rewriting we refer to it as a contained rewriting. Note that the above definitions are independent of the particular query language we consider. Furthermore, we note that algorithms for query containment and equivalence provide methods for *testing* whether a rewriting of a query is an equivalent or contained rewriting, but by themselves do not provide an algorithm for rewriting queries using views.

A more fundamental question we can consider is how to find *all* the possible answers to the query, given a set of view definitions and their extensions. Finding a rewriting of the query using the views and then evaluating the rewriting over the views is clearly one candidate algorithm. If the rewriting is equivalent to the query, then we are guaranteed to find all the possible answers. However, as we see in Section 7, a maximally-contained rewriting of a query using a set of views does not always provide all the possible answers that can be obtained from the views. Intuitively, the reason for this is that a rewriting is maximally-contained only w.r.t. a specific query language, and hence there may sometimes be a query in a more expressive language that may provide more answers.

The problem of finding all the answers to a query given a set of views is formalized below by the notion of *certain answers*. In the definition, we distinguish the case in which the view extensions are complete (closed-world assumption) from the case in which the views may be partial (open-world).

Definition 3.4 (**Certain answers**) Let Q be a query and $\mathcal{V} = V_1, \ldots, V_m$ be a set of view definitions over the database schema $R_1, \ldots, R_n$. Let the sets of tuples $v_1, \ldots, v_m$ be extensions of the views $V_1, \ldots, V_m$, respectively.

The tuple $\bar{a}$ is a *certain answer* to the query Q under the closed-world assumption given $v_1, \ldots, v_m$ if $\bar{a}$ is an answer to Q for any database D such that $V_i(D) = v_i$ for every i, $1 \leq i \leq m$.

The tuple $\bar{a}$ is a *certain answer* to the query Q under the open-world assumption given $v_1, \ldots, v_m$ if $\bar{a}$ is an answer to Q for any database D such that $V_i(D) \supseteq v_i$ for every i, $1 \leq i \leq m$. $\Box$

The intuition behind the definition of certain answers is the following. The extensions of a set of views does not define a unique extension of the database relations. Hence, given the extension of the views we have only partial information about the real state of the database. A tuple is a certain answer of the query Q if it is an answer for *any* of the possible database extensions that are consistent with the given extensions of the views. Section 7.3 considers the complexity of finding certain answers.

4 When is a View Usable for a Query

Before describing the actual algorithms for answering queries using views it is useful to examine a few examples and gain an intuition for the conditions under which a view is usable for answering a query.

Informally, a view can be useful for a query if the set of relations it mentions overlaps with that of the query, and it selects some of the attributes selected by the query. Moreover, if the query applies predicates to attributes that it has in common with the view, then the view must apply either equivalent or logically weaker predicates.

Consider the following query, asking for the triplets of professors, students, and teaching quarters, where the student is advised by the professor, and has taken a class taught by the professor during the winter of 1998 or later. Note that the relations Registered and Teaches also contain an attribute denoting the quarter of the course.

```
select     Advises.prof, Advises.student, Registered.quarter
```

```
from     Registered, Teaches, Advises
where    Registered.course=Teaches.course and Registered.quarter=Teaches.quarter and
         Advises.prof=Teaches.prof and Advises.student=Registered.student and
         Registered.quarter ≥ "winter98".
```

The following view V_1 is usable because it applies the same join conditions to the relations Registered and Teaches. Furthermore, it selects the attributes Registered.student, Registered.quarter and Teaches.prof that are needed for the join with the relation Advises and for the select clause of the query. Finally, the view applies a predicate Registered.quarter > "winter97" which is weaker than that in the query, but since the view selects the attribute Registered.quarter, the stronger predicate can be applied later in the rewriting.

```
create view V1 as
select   Registered.student, Teaches.prof, Registered.quarter
from     Registered, Teaches
where    Registered.course=Teaches.course and Registered.quarter=Teaches.quarter and
         Registered.quarter > "winter97".
```

The views shown in Figure 3 illustrate how minor modifications to V_1 deem the view unusable for answering our query. The problem with view V_2 is that it does not select the attribute Teaches.prof, which is needed in the select clause of the query. Note that if the attributes that were selected would functionally determine Teaches.prof, then V_2 would be usable. However, this dependency does not hold in our example.

```
create view V2 as
select  Registered.student, Registered.quarter
from    Registered, Teaches
where   Registered.course=Teaches.course and
        Registered.quarter=Teaches.quarter and
        Registered.quarter ≥ "winter98".

create view V3 as
select  Registered.student, Teaches.prof, Registered.quarter
from    Registered, Teaches
where   Registered.course=Teaches.course and
        Registered.quarter ≥ "winter98".

create view V4 as
select  Registered.student, Registered.quarter, prof
from    Registered, Teaches, TA
where   Registered.course=Teaches.course and
        Registered.quarter=Teaches.quarter and
        Registered.course=TA.course and
        Registered.quarter ≥ "winter98".

create view V5 as
select  Registered.student, Teaches.prof, Registered.quarter
from    Registered, Teaches
where   Registered.course=Teaches.course and
        Registered.quarter=Teaches.quarter and
        Registered.quarter > "summer98".
```

Figure 3: Examples of unusable views.

The problem with view V_3 is that it does not apply the necessary join predicate Registered.quarter=Teaches.quarter. Since the attribute Teaches.quarter is not selected by V_3, the join predicate cannot be applied later. View V_4 considers only the courses that have at least one teaching assistant, hence it applies an additional condition that does not exist in the query. Without introducing unions into the rewriting, the view V_4 can only be used for an equivalent rewriting of the query if it is known that every course has at least one teaching assistant. Finally, view V_5 applies a stronger predicate than in the query (Registered.quarter > "summer98"), and is therefore not usable for an equivalent rewriting unless we are only interested in contained rewritings.

To summarize, the following conditions need to hold in order for a view V to be usable for a query Q. These conditions can be made formal in the context of a specific query language and/or available integrity constraints (see e.g., [YL87, LMSS95]).

1. There must be a mapping ψ from the tables mentioned in V to the tables mentioned in Q (in the case of bag semantics, this mapping must be a 1-1 mapping).

2. V must either apply the join and selection predicates in Q on the attributes of the tables in the domain of ψ, or must apply to them a logically weaker selection, and select the attributes on which predicates need to still be applied.

3. V must not project out any attributes of the tables in the domain of ψ that are needed in the selection of Q, unless these attributes can be recovered from another view.

5 Incorporating Materialized Views into Query Optimization

We begin our survey of algorithms for answering queries using views by discussing several algorithms whose goal is to extend a System-R style optimizer to accommodate usage of views [CKPS95, TSI96, BDD+98]. The focus of these algorithms is to judiciously decide in a cost-based fashion when to use views to answer a query, and their output is an execution plan for the query. Broadly speaking, these algorithms generate candidate rewritings of the query, check which ones are correct (using variations on query-equivalence techniques), and decide which ones to use based on their expected cost.

To illustrate the changes that the algorithms need to make to a System-R style optimizer we first briefly recall the principles underlying System-R optimization [SAC+79]. System-R takes a bottom-up approach to building query execution plans. In the first phase, it constructs plans of size 1, i.e., that access only one database relation. In phase n, the algorithm considers plans of size n, by combining pairs of plans obtained in the previous phases.[1] The algorithm terminates after constructing plans that cover all the relations in the query.

System-R partitions query execution plans into *equivalence classes*. Two plans are in the same equivalence class if they (1) cover the same set of relations in the query (and therefore are also of the same size), and (2) produce the answers in the same interesting order. In the process of building plans, two plans are combined only if they cover disjoint subsets of

[1]Note that if the algorithm is considering only left-deep plans, it will try to combine plans of size 1 with plans of size $n-1$. Otherwise, it will consider combining plans of size k with plans of size $n-k$.

the relations mentioned in the query. System-R prunes its search space by saving a *single* execution plan for every equivalence class.

In our context the query optimizer builds query execution plans by accessing a set of views, rather than a set of database relations. Hence, in addition to the meta-data that the query optimizer has about the materialized views (e.g., statistics, indexes) the optimizer is also given as input the query expressions defining the views. Recall that a database relation can always be modeled as a view as well.

An algorithm for incorporating materialized views into a System-R style optimizer must consider the following issues. Figure 4 shows a side-by-side comparison of the steps of a traditional optimizer vs. one that exploits materialized views.

1. In the first phase of the optimization the algorithm needs to decide which views are *relevant* to the query. A view is relevant if it is usable in answering the query (illustrated by the conditions in Section 4). The corresponding step in a traditional optimizer is trivial: a relation is relevant to the query if it is mentioned in the from clause.

2. Since the query execution plans involve joins over views, rather than over database relations, plans can no longer be neatly partitioned into equivalence classes which can be explored in increasing size. This observation implies two changes to the traditional algorithm:

   - **Termination testing:** the algorithm needs to distinguish *partial query execution plans* of the query from *complete execution plans* covering the entire query. The enumeration of the possible join orders terminates when there are no more unexplored partial solutions. In contrast, in the traditional setting the algorithm terminates after considering the equivalence classes that include all the relations in the query.
   - **Pruning of plans:** a traditional optimizer compares between pairs of plans *within* one equivalence class and saves only the cheapest one for each class. Here the query optimizer needs to compare between *any pair* of plans generated thus far. A plan p is pruned if there is another plan p' that (1) is cheaper than p and, (2) has greater or equal contribution to the query as p. Informally, a plan p' contributes more to the query than the plan p if it covers more of the relations in the query and selects more of the necessary attributes.

3. The step of combining partial plans is quite a bit more complex. In the traditional setting, when two partial plans are combined, the join predicates that involve both plans are explicit in the query, and the enumeration algorithm need only consider the most efficient way to apply these predicates. However, in our case, the enumeration algorithm may have to try joining the two plans using *any* possible set of join predicates between their respective attributes. This is because it may not be obvious apriori which join predicate will yield a correct rewriting of the query. Fortunately, in practice, the number of join predicates that need to be considered can be significantly pruned using

meta-data about the schema (e.g., there is no point to try joining a string attribute with a numeric one). Finally, after considering all the possible join predicates, the optimizer also needs to check whether the resulting plan is still a partial solution to the query.

Conventional optimizer	**Optimizer using views**
Iteration 1	**Iteration 1**
a) find all possible access paths.	a1) Find all views that are *relevant* to the query.
	a2) Distinguish between partial and complete solutions to the query.
b) Compare their cost and keep the least expensive.	b) Compare all pairs of views. If one has neither greater contribution or a lower cost than the other, prune it.
c) If the query has one relation, stop.	c) If there are no partial solutions, stop.
Iteration 2	**Iteration 2**
For each query join:	
a) Consider joining the relevant access paths found in the previous iteration using all possible join methods.	a1) Consider joining all partial solutions found in the previous iteration using all possible equi-join methods and trying all possible subsets of join predicates.
	a2) Distinguish between complete and partial solutions.
b) Compare the cost of the resulting join plans and keep the least expensive.	b) If any newly generated solution is either not relevant to the query, or dominated by another, prune it.
c) If the query has only 2 relations, stop.	c) If there are no partial solutions, stop.
Iteration 3	**Iteration 3**
...	...

Figure 4: A comparison of a traditional query optimizer with one that exploits materialized views.

Variations on the above algorithm are presented in [TSI94, TSI96] and [CKPS95]. The algorithm in [TSI96] attempts to reformulate a query on a logical schema to refer directly to GMAPs storing the data (see Section 2). They consider select-project-join queries with set semantics. To test whether a solution is complete (i.e., whether it is equivalent to the original query) they use an efficient sufficient query-equivalence condition that also makes use of some inclusion and functional dependencies.

The algorithm described in [CKPS95] describes the use of materialized views for query optimization. In that work they consider select-project-join queries with multiset semantics which may also include arithmetic comparison operators. Under multiset semantics, the ways in which views may be combined to answer a query are more limited. This is due to the fact that two queries are equivalent if and only if there is a 1-1 isomorphism between them [CV93]. Hence, if we ignore the arithmetic comparison operators, a view is usable only if it is isomorphic to a subset of the query. As a result, the join enumeration algorithms become slightly simpler than the one discussed above.

Both [TSI96] and [CKPS95] present experimental results that show that the performance of a query processor does not degrade with the added capability of exploiting views. As we see in Section 7, it can be shown that both algorithms are *complete*, in the sense that they will find a rewriting of the query using the views if one exists.

In [BDD$^+$98] the authors describe the implementation of a view rewriting algorithm in the Oracle DBMS. The algorithm works in two phases. In the first phase, the algorithm applies a set of rewrite rules that attempt to replace parts of the query with references to existing materialized views. The rewrite rules consider the cases in which views satisfy the conditions described in Section 4, and also consider common integrity constraints encountered in practice. The result of the rewrite phase is a query that refers to the views. In the second phase, the algorithm compares the estimated cost of two plans: the cost of the result of the first phase, and the cost of the best plan found by the optimizer that does *not* consider the use of materialized views. The optimizer chooses to execute the cheaper of these two plans. The main advantage of this approach is its ease of implementation, since the capability of using views is added to the optimizer without changing the join enumeration module. On the other hand, the algorithm considers the cost of only one possible rewriting of the query using the views, and hence may miss the cheapest use of the materialized views.

Example

We illustrate the algorithms (in the set-semantics case) with the following example, which includes the following relations and views.

Enrolled(s-name, dept), Registered(s-name, c-name, quarter), Course(title, number).

```
create view V1 as
select    s-name, dept
from      Enrolled
```

```
create view V2 as
select    s-name, c-name, number
from      Registered, Course
where     Registered.c-name=course.c-name.
```

```
create view V3 as
select    dept, c-name
from      Registered, Enrolled
where     Registered.s-name=Enrolled.s-name.
```

Suppose the query below asks for all of the students attending Ph.D level classes, and the departments in which they're enrolled.

```
select    s-name, dept
from      Registered, Enrolled, Course
where     Registered.s-name=Enrolled.s-name and Registered.c-name=Course.title and number≥500.
```

In the first iteration, the algorithm will determine that all three views are relevant to the query, because each of them mentions the relations in the query and applies the same join predicates as in the query. Therefore, the algorithm chooses the best access path to each of the views, depending on the existing index structures.

In the second iteration, the algorithm will consider all of the possible methods to join pairs of plans produced in the first iteration. The algorithm will save the cheapest plan for each of the two-way joins, assuming the result is still a partial or complete solution to the query. In this case, the algorithm will first consider the join of V1 and V2 on the attribute s-name. This join produces a partial result to the query (since applying an additional selection on the number attribute would produce a complete solution to the query). In principle, as explained in (3), the algorithm should also consider joining V1 and V2 on other attributes (e.g., V1.s-name=V2.c-name), but in this case, a simple semantic analysis shows that such a join will not yield a partial solution. The joins of V1 with V3 (on s-name) and of V2 with V3 (on s-name) produce partial solutions to the query, but only if set semantics are considered (because the resulting rewriting will have multiple occurrences of the Enrolled (or Registered) relation, whereas the query has only one occurrence).

In the third iteration, the algorithm tries to join the plans for the partial solutions from the second iteration with a plan from the first iteration. In this example, if we consider set semantics, the algorithm will consider a plan in which the result of joining V1 and V2 is then joined with V3. Even though this plan may seem redundant, depending on the available indexes and relative sizes, it may be cheaper than the one that involves only a join between V1 and V2. Finally, in both cases we need to add a selection on the number attribute.

6 Answering Queries Using Views for Data Integration

The algorithms described in the previous section focused on extending query optimizers to accommodate the use of views. They were designed to handle cases in which the number of views is relatively small (i.e., comparable to the size of the database schema), and cases in which we require an *equivalent* rewriting of the query. In addition, these algorithms did not consider cases in which the resulting rewriting may contain a union over the views.

In contrast, the context of data integration requires that we consider a large number of views, since each data source is being described by one or more views. In addition, in this context the view definitions contain many complex predicates, whose goal is to express fine-grained distinctions between the contents of different sources. As shown in Section 2, we will often not be able to find an equivalent rewriting of the query using the source views, and the best we can do is find the maximally-contained rewriting of the query.

In this section 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 [LRO96] and later studied in [GM99a], and the *inverse-rules algorithm* [Qia96, DG97b] which was implemented in the InfoMaster system [DG97b]. It should be noted that unlike the algorithms described in the previous section, the output of these algorithms is not a query execution plan, but only a query referring to the view relations. For this and the next section, we revert to datalog notation and terminology.

6.1 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). For a query Q, we refer to the subgoals of comparison predicates by $C(Q)$. When the query does not contain arithmetic comparison predicates (but the view definitions still may) the bucket algorithm is guaranteed to return the maximally contained rewriting of the query using the views. As noted in Section 7, the number of possible rewritings of the query using the views is exponential in the size of the query. Hence, the main idea underlying the bucket algorithm is that the 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 that is not in $C(Q)$, 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

a. g and g_1 can be unified, and

b. after applying the unifier to the query and to the view, the predicates in Q and in V are mutually satisfiable. More formally, if ψ is the unification mapping from the variables of Q to the variables of V, and ψ_h is the restriction of ψ to the variables of Q that are mapped to variables in the head of V, then $\psi_h(C(Q))$ and $C(V)$ are mutually satisfiable.

The actual bucket contains the head of the view V with the appropriate variables in the query substituted in the appropriate positions. 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. If not, the algorithm checks whether adding atoms of comparison predicates can make the resulting conjunction contained in the query. If so, the rewriting is added to the answer. Hence, the result of the bucket algorithm is a union of conjunctive rewritings.

Example 6.1 Consider an example including the following views

```
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$\geq$300, Y$\geq$1995.

In the first step the algorithm creates a bucket for each of the relational subgoals in the query in turn. The resulting contents of the buckets are shown in Table 1. The bucket of Enrolled(S,D) includes views V2 and V4, since the following mapping unifies the subgoal in the query with the corresponding Enrolled subgoal in the views (thereby satisfying (a) above):

$\{$ S $\rightarrow$ student, D $\rightarrow$ dept $\}$.

Note that each view head in a bucket only includes variables in the domain of the mapping. Fresh variables (primed) are used for the other head variables of the view.

The bucket of the subgoal Registered(S,C,Y) contains the views V1, V2 and V4 since the following mapping unifies the subgoal in the query with the corresponding Registered subgoal in the views:

$\{$ S $\rightarrow$ student, C $\rightarrow$ course, Y $\rightarrow$ 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: 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 after applying the unification mapping, the predicates Y $\geq$ 1995 and year $\leq$ 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 V4 and in the query are mutually inconsistent, thereby possibly violating rule (b). However, the unification mapping does not include the variable N in its domain, and therefore the variable N is not mapped to the variable number in V4, and the contradiction between the predicates does not arise.

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 view V4 is *not* included in the bucket because the unifier maps N to number, and hence the predicates on the course number would be mutually inconsistent, violating rule (b).

In the second step of the algorithm, we combine elements from the buckets. In our example, we start with a rewriting that includes the top elements of each bucket, i.e., q'(S,D) :- V2(S,D,C'), V1(S, N', Y), V1(S', N, Y').

As can be checked, this rewriting is not contained in the original query. However, by adding predicates N=N' and Y≥1995 we can obtain a rewriting that is contained in the query. Hence, the following is the first rewriting that is obtained by the algorithm (note that Y and Y' have been equated as an additional optimization):

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

In our example, the only other possibly interesting rewriting to consider would involve a join between V1 and V4. Such a rewriting would be 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. □

6.2 The Inverse-rules Algorithm

Like the bucket algorithm, the inverse-rules algorithm was also developed in the context of a data integration system [DG97b]. 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 tuple (dept,name) is a witness in the sense that 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. These function symbols are used in the heads of the inverse rules.

The rewriting of a query Q using the set of views $\mathcal{V}$ is the datalog program that includes

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

As shown in [DG97a], 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 6.2 Suppose a 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 the above 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 [DG97a].

6.3 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 [LFS97, SR92]).

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. As shown in [DG97a, DL97], 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 inverse rules.

Using the inverse rules directly for evaluating queries has a significant drawback, since it attempts to recompute the extensions of the database relations. In our example, evaluating the inverse rules computes tuples for Registered and Enrolled, and the query is then evaluated over these extensions. However, by doing that, we lose the fact that the view already computed the join that the query is requesting. Hence, most of the computational advantage of exploiting the materialized view is lost.

In order to obtain a more efficient rewriting from the inverse rules, we must perform additional optimizations such as rule folding and removing irrelevant rules. 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 [LFS97]). The algorithm does not handle arithmetic comparison predicates, and extending it to handle such predicates turns out to be quite subtle.

A hybrid algorithm that combines ideas of the bucket and inverse-rule algorithms is described in [PL99]. Experimental results in that paper show that the hybrid algorithm performs significantly better than it predecessors, and scales up in the presence of a large number of views.

7 Theory of Answering Queries Using Views

In the previous sections we discussed specific algorithms for answering queries using views. Here we consider several fundamental issues, which cut across all of the algorithms we have discussed thus far, and which have been studied from a more theoretical perspective in the literature.

The first question concerns the *completeness* of the query rewriting algorithms. That is, given a set of views and a query, will the algorithm always find a rewriting of the query using the views if one exists? A related issue is characterizing the complexity of the query rewriting problem. We discuss these issues in Section 7.1.

Completeness of a rewriting algorithm is characterized w.r.t. a specific query language in which the rewritings are expressed (e.g., select-project-join queries, queries with union, recursion). For example, there are cases in which if we do not allow unions in the rewriting of the query, then we will not be able to find an equivalent rewriting of a query using a set of views. The language that we consider for the rewriting is even more crucial when we consider maximally-contained rewritings, because the notion of maximal containment is defined w.r.t. a specific query language. As it turns out, there are several important cases in which a maximally-contained rewriting of a query can only be found if we consider *recursive* rewritings. These cases are illustrated in Section 7.2.

At the limit, we would like to be able to extract all the *certain* answers for a query given a set of views, whether we do it by applying a query rewriting to the extensions of the views or via an arbitrary algorithm. In Section 7.3 we consider the complexity of finding all the certain answers, and show that even in some simple cases the problem is surprisingly NP-hard in the size of the extensions of the views.

7.1 Completeness and complexity of query rewritings

The first question one can ask about an algorithm for rewriting queries using views is whether the algorithm is complete: given a query Q and a set of views $\mathcal{V}$, will the algorithm find a rewriting of Q using $\mathcal{V}$ when one exists. The first answer to this question was given for the class of queries and views expressed as conjunctive queries [LMSS95]. In that paper it was shown that when the views do not contain comparison predicates, and the query Q has n subgoals, then there exists an equivalent-rewriting of Q using $\mathcal{V}$ only if there is a rewriting with at most n subgoals. An immediate corollary of the bound on the size of the rewriting is that the problem of finding and equivalent rewriting of a query using a set of views is in NP, because it suffices to guess a rewriting and check its correctness.[2]

The bound on the size of the rewriting also sheds some light on the algorithms described in the previous sections. In particular, it entails that the search strategy that the GMAP algorithm [TSI96] employs is guaranteed to be complete under the conditions that (1) the we use a sound and complete algorithm for query containment, and (2) when combining two subplans, the algorithm considers all possible join predicates on the combined subplans. The bound also entails that the bucket algorithm is guaranteed to be complete when the queries and the view do not contain comparison predicates, and that the bucket algorithm produces the maximally contained rewriting of the query if we consider rewritings that are unions of conjunctive queries. It is important to emphasize though that the rewriting of the query that produces the most *efficient* plan for answering the query may have *more* conjuncts that the original query.

In [LMSS95] it is also shown that the problem of finding a rewriting is NP-hard for two independent reasons: (1) the number of possible ways to map a single view into the query, and (2) the number of ways to combine the mappings of different views into the query.

[2]Note that checking the correctness of a rewriting is NP-complete, however, the guess of a rewriting can be extended to a guess for containment mappings showing the equivalence of the rewriting and of the query.

The results of [LMSS95] can also be extended to conjunctive queries with comparison predicates. The basic observation underlying this extension is that one only needs to consider rewritings that mention constants that are used in either the views or the query [Klu88, LRU96]. In [RSU95] the authors extend the bound on the size of the rewriting to the case where the views contain binding pattern limitations (discussed in detail in Section 8.3). In [CR97] the authors exploit the close connection between the containment and rewriting problems, and show several polynomial-time cases of the rewriting problems, corresponding to analogous cases for the problem of query containment.

7.2 The need for recursive rewritings

As noted earlier, in cases where we cannot find an equivalent rewriting of the query using a set of views, we consider the problem of finding maximally-contained rewritings. Our hope is that when we apply the maximally-contained rewriting to the extensions of the views, we will obtain the set of *all* certain answers to the query (definition 3.4). Interestingly, there are several contexts where in order to achieve this goal we need to consider recursive datalog rewritings of the query. We recall that a datalog rewriting is a datalog program in which the base (EDB) predicates are the view relations, and there are additional intermediate IDB relations. Except for the obvious case in which the query is recursive [DG97a], other cases include: the presence of functional dependencies on the database relations or binding pattern limitations on the extensions of the views [DL97], and the case where additional semantic information about class hierarchies on objects is expressed using description logics [BLR97]. We illustrate the case of functional dependencies below.

Example 7.1 To illustrate the need for recursive rewritings in the presence of functional dependencies, we temporarily venture into the domain of airline flights. Suppose we have the following database relation

schedule(Airline,Flight_no,Date,Pilot,Aircraft)

which stores tuples describing the pilot that is scheduled for a certain flight, and the aircraft that is used for this flight. Assume we have the following functional dependencies on the relations in the mediated schema

Pilot $\rightarrow$ Airline and
Aircraft $\rightarrow$ Airline

expressing the constraints that pilots work for only one airline, and that there is no joint ownership of aircrafts between airlines. Suppose we have the following view available, which projects the date, pilot and aircraft attributes from the database relation:

v(D,P,C) :- schedule(A,N,D,P,C)

The view v records on which date which pilot flies which aircraft. Now consider a query asking for pilots that work for the same airline as Mike (expressed as the following self join on the Airline attribute of the schedule relation):

```
q(P) :- schedule(A,N,D,'mike',C), schedule(A,N',D',P,C')
```

The view v doesn't record the airlines that pilots work for, and therefore, deriving answers to the above query requires using the functional dependencies in subtle ways. For example, if both Mike and Ann are known to have flown aircraft #111, then, since each aircraft belongs to a single airline, and every pilot flies for only one airline, Ann must work for the same airline as Mike. Moreover, if, in addition, Ann is known to have flown aircraft #222, and John has flown aircraft #222 then the same line of reasoning leads us to conclude that Ann and John work for the same airline. In general, for any value of n, the following conjunctive rewriting is a contained rewriting:

$$\begin{aligned} q_n(P) :- \mathsf{v}(D_1, mike, C_1), \mathsf{v}(D_2, P_2, C_1),\ \mathsf{v}(D_3, P_2, C_2),\ \mathsf{v}(D_4, P_3, C_2),\ \ldots, \\ \mathsf{v}(D_{2n-2}, P_n, C_{n-1}),\ \mathsf{v}(D_{2n-1}, P_n, C_n),\ \mathsf{v}(D_{2n}, P, C_n) \end{aligned}$$

Moreover, for each n, $q_n(P)$ may provide answers that were not given by q_i for $i < n$, because one can always build an extension of the view v that requires n steps of chaining in order to find answers to the query. The conclusion is that we cannot find a maximally-contained rewriting of this query using the views if we only consider non-recursive rewritings. Instead, the maximally-contained rewriting is the following datalog program:

```
relevantPilot("mike").
relevantAirCraft(C)  :- v(D, "mike", C).
relevantAirCraft(C)  :- v(D,P,C), relevantPilot(P).
relevantPilot(P)     :- relevantPilot(P1), relevantAirCraft(C), v(D1, P1, C), v(D2, P, C).
```

In the program above, the relation relevantPilot will include the set of pilots who work for the same airline as Mike, and the relation relevantAirCraft will include the aircraft flown by relevant pilots. Note that the fourth rule is mutually recursive with the definition of relevantAirCraft. □

In [DL97] it is shown how to augment the inverse-rules algorithm to incorporate functional dependencies. The key element of that algorithm is to add a set of rules that simulate the application of a Chase algorithm [MMS79] on the atoms of the database relations.

7.3 Finding the certain answers

If Q' is an *equivalent-rewriting* of a query Q using the set of views $\mathcal{V}$, then it will always produce the same result as Q, independent of the state of the database or of the views. In particular, this means that Q' will always produce all the certain answers to Q for any possible database. Recall that an answer to Q is certain given the extensions $v_1, \ldots, v_n$ of the views in $V_1, \ldots, V_n$, if it would be an answer of Q for *any* database that would give rise to those view extensions.

When Q' is a *maximally-contained rewriting* of Q using the views $\mathcal{V}$ it may produce only a subset of the answers of Q for a given state of the database. The maximality of Q' is defined only w.r.t. the other possible rewritings in a particular query language $\mathcal{L}$ that we consider for

Q'. Hence, the question that remains is how to find all the certain answers, whether we do it by applying some rewritten query to the views or by some other algorithm.

The question of finding all the certain answers is considered in detail in [AD98, GM99a]. In their analysis they distinguish the case of the *open-world assumption* from that of the *closed-world assumption*. With the closed-world assumption, the extensions of the views are assumed to contain *all* the tuples that would result from applying the view definition to the database. Under the open-world assumption, the extensions of the views may be missing tuples (but they may not have incorrect tuples). The open-world assumption is especially appropriate in data integration applications, where the views describe sources that may be incomplete (see [EGW94, Lev96, Dus97] for treatments of complete sources). The closed-world assumption is appropriate for the context of query optimization and maintaining physical data independence, where views have actually been computed from existing database relations.

Under the open-world assumption, [AD98] show that in many practical cases, finding all the certain answers can be done in polynomial time. However, the problem becomes NP-hard as soon as we allow union in the language for defining the views, or allow the predicate $\neq$ in the language defining the query.

Under the closed-world assumption the situation is even worse. Even when both the views and the query are defined by conjunctive queries without comparison predicates, the problem of finding all certain answers is already NP-hard. The following example is the crux of the proof of the NP-hardness result [AD98].

Example 7.2 The following example shows a reduction of the problem of graph 3-colorability to the problem of finding all the certain answers. Suppose the relation edge(X,Y) encodes the edges of a graph, and the relation color(X,Z) encodes which color Z is attached to the nodes of the graph. Consider the following three views:

```
V1(X) :- color(X,Y)
V2(Y) :- color(X,Y)
V3(X,Y) :- edge(X,Y)
```

where the extension of V1 is the set of nodes in a graph, the extension of V2 is the set {red, green, blue}, and the extension of V3 is the set of edges in the graph. Consider the following query:

```
q(c) :- edge(X,Y), color(X,Z), color(Y,Z)
```

In [AD98] it is shown that c is a certain answer to q if and only if the graph encoded by edge is *not* three-colorable. Hence, they show that the problem of finding all certain answers is NP-hard. □

The hardness of finding all the certain answers provides an interesting perspective on formalisms for data integration. Intuitively, the result entails that when we use views to describe the contents of data sources, even if we only use conjunctive queries to describe the sources, the complexity of finding all the answers to a query from the set of sources is

NP-hard. In contrast, using a formalism in which the relations of the mediated schema are described by views over the source relations (as in [GMPQ$^+$97]), the complexity of finding all the answers is always polynomial. Hence, this result hints that the former formalism has a greater expressive power as a formalism for data integration.

It is also interesting to note the connection established in [AD98] between the problem of finding all certain answers and computation with conditional tables [IL84]. As the authors show, the partial information about the database that is available from a set of views can be encoded as a conditional table using the formalism studied in [IL84].

The work in [GM99a] also considers the case where the views may either be incomplete, complete, or contain incorrect tuples. It is shown the case in which either all the views are complete or all of them may contain incorrect tuples, then finding all certain answers can be done in polynomial time in the size of the view extensions. In other cases, the problem is NP-hard.

8 Extensions to the Query Language

In this section we survey the algorithms for answering queries using views in the context of several important extensions to the query languages considered thus far. We consider extensions for Object Query Language (OQL) [FRV96, Flo96], queries with grouping and aggregation [GHQ95, SDJL96, CNS99, GRT99], and views with access pattern limitations [RSU95, KW96, DL97].

8.1 Object Query Language

Florescu et al. [FRV96, Flo96] have studied the problem of answering queries using views in the context of querying object-oriented databases, and have incorporated their algorithm into the Flora OQL optimizer. In object-oriented databases the correspondence between the *logical* model of the data and the *physical* model is even less direct than in relational systems. Hence, as argued in [Flo96], it is imperative for a query optimizer for object-oriented database be based on the notion of physical data independence.

Answering queries using views in the context of object-oriented systems introduces two key difficulties. First, finding rewritings often requires that we exploit some semantic information about the class hierarchy and about the attributes of classes. Second, OQL does not make a clean distinction between the `select`, `from` and `where` clauses as in SQL. `Select` clauses may contain arbitrary expressions, and the `where` clauses also allow path navigation.

The algorithm for answering queries using views described in [Flo96] operates in two phases. In the first phase the algorithm rewrites the query into canonical form, thereby addressing the lack of distinction between the `select`, `from` and `where` clauses. As an example, in this phase, navigational expressions are removed from the `where` clause by introducing new variables and terms in the `from` clause.

In the second phase, the algorithm exploits semantic knowledge about the class hierarchy in order to find a subexpression of the query that is matched by one of the views. When such a match is found, the subexpression in the query is replaced by a reference to the view and appropriate conditions are added in order to conserve the equivalence to the query.

We illustrate the main novelties of the algorithm with the following example from [Flo96], using a French version of our university domain.

Example 8.1 Suppose we have the following view asking for students who are older than their professors, and who study in universities in small cities:

```
create view V1 as
select     distinct [A:=x.name, B:=y.identifier, C:=z]
from       x in Universities, y in x.students, z in union(x.professors, x.adjuncts)
where      x.city.kind="small" and y.age ≥ z.age.
```

Suppose a query asks for Ph.D students in French universities who have the same age as their professors, and study in small town universities:

```
select     distinct [D:=u.name, E:=v.name, F:=t.name]
from       u in France.Universities, v in u.PhDstudents, t in u.professors
where      u.city.kind="small" and v.age=t.age.
```

In the first step, the algorithm will transform the query and the view into their normal form. The resulting expression for the query would be: (note that the variable w was added to the query in order to eliminate the navigation term from the where clause)

```
select     distinct [D:=u.name, E:=v.name, F:=t.name]
from       u in France.Universities, w in City, v in u.PhDstudents, t in u.professors
where      w.kind="small" and v.age=t.age and u.city=w.
```

In the next step, the algorithm will note the following properties of the schema:

1. The collection France.Universities is included in the collection Universities,
2. The collection denoted by the expression u.PhDstudents is included in the collection denoted by x.students. This inclusion follows from the first inclusion and the fact that PhD students are a subset of students.
3. The collection u.professors is included in the collection union(x.professors, x.adjuncts).

Putting these three inclusions together, the algorithm determines that the view can be used to answer the query, because the selections in the view are less restrictive than those in the query. The rewriting of the query using the view is the following:

```
select     distinct [D:=a.A, E:=a.B.name, F:=t.name]
from       a in V1, u in France.Universities, v in u.PhDstudents, t in u.professors
where      u.city.kind="small" and v.age=t.age and
           u.name=a.A and v.name=a.B and t=a.C.
```

Note that the role of the view is only to restrict the possible bindings of the variables used in the query. In particular, the query still has to restrict the universities to only the French ones, the students to only the Ph.Ds, and the range of the variable t to cover only professors. In this case, the evaluation of the query using the view is likely to be more efficient than computing the query only from the class extents. □

8.2 Queries with Grouping and Aggregation

In decision support applications, when queries contain grouping and aggregation, there is even more of an opportunity to obtain significant speedups by reusing the results of materialized views. However, the presence of grouping and aggregation in the queries or the views introduces several new difficulties to the problem of answering queries using views. The first difficulty that arises is dealing with aggregated columns. Recall that for a view to be usable by a query, it must not project out an attribute that is needed in the query (and is not otherwise recoverable). When a view performs an aggregation on an attribute, we lose some information about the attribute, i.e., in a sense *partially* projecting it out. A rewriting algorithm should be careful to consider whether the aggregated column is still sufficient for answering the query. The second difficulty arises due to the loss of multiplicity of values on attributes on which grouping is performed. When we group on an attribute A, we lose the multiplicity of the attribute in the data, thereby losing the ability to perform subsequent sum, counting or averaging operations. In some cases, it may be possible to recover the multiplicity using additional information.

The following simple example illustrates some of the subtleties that arise in the presence of grouping and aggregation. The example uses a single relation:

Teaches(prof, course, year, evaluation)

describing the numeric course evaluations that professors have received for teaching a certain course. Suppose we have the following view available, which considers all the graduate level courses, and for every pair of course and year, gives the maximal course evaluation for that course in the given year, and the number of times the course was offered.

```
create view V as
select    course, year, Max(evaluation) as maxeval, Count(∗) as offerings
from      Teaches
where     course ≥ 400
groupBy   course, year.
```

The following query considers only Ph.D-level courses (i.e., level 500 and higher), and asks for the maximal evaluation obtained for *any* course during a given year, and the number of different course offerings during that year.

```
select    year, count(∗), Max(evaluation)
from      Teaches
```

```
where     course ≥ 500
groupBy   year.
```

The following rewriting uses the view V to answer our query.

```
select    year, sum(offerings), Max(maxeval)
from      V
where     course ≥ 500
groupBy   year.
```

There are a couple of points to note about the rewriting. First, even though the view performed an aggregation on the evaluation attribute, we could still use the view in the query, because the groupings in the query (on year) are more coarse than those in the view (on year and course). Thus, the answer to the query can be obtained by coalescing groups from the view. Second, since the view groups the answers by course and thereby loses the multiplicity of each course, we would have not been able to use it to compute the number of offerings of the course. However, since the view also computed the attribute offerings, there was still enough information in the view to recover the total number of course offerings per year.

Srivastava et al. [SDJL96] describe the conditions required for a view to be usable for answering a query in the presence of grouping and aggregation, and a rewriting algorithm that incorporates these conditions. That paper considers the cases in which the views and/or the queries contain grouping and aggregation. It is interesting to note that when the view contains grouping and aggregation but the query does not, then unless the query removes duplicates in the select clause, the view cannot be used to answer a query. Another important point to recall about this context is that because of the multi-set semantics a view will be usable to answer a query only if there is an isomorphism between the view and a subset of the query [CV93]. A transformational approach to answering queries using views in the context of grouping and aggregation is proposed in [GHQ95]. In this approach, they perform syntactic transformations on the query until the view becomes identical to a subset of the query. As the authors point out, the purely syntactic nature of this approach is a limiting factor in its applicability.

Two papers [CNS99, GRT99] consider the formal aspects of answering queries using views in the presence of grouping and aggregation. They present cases in which it can be shown that a rewriting algorithm is complete, in the sense that it will find a rewriting if one exists. Their algorithms are based on insights into the problem of query containment for queries with grouping and aggregation.

8.3 Binding Pattern Limitations

In the context of data integration, where data sources are modeled as views, we may have limitations on the possible access paths to the data. For example, when querying the Internet Movie Database, we cannot simply ask for all the tuples in the database. Instead, we must supply one of several inputs, (e.g., actor name or director), and obtain the set of movies in which they are involved.

We can model limited access paths by attaching a set of adornments to every data source. If a source is modeled by a view with n attributes, then an adornment consists of a string of length n, composed of the letters b and f. The meaning of the letter b in an adornment is that the source *must* be given values for the attribute in that position. The meaning of the letter f in an adornment is that the source doesn't have to be given a value for the attribute in that position. For example, an adornment bf for a view $V(A, B)$ means that tuples of V can be obtained only by providing values for the attributes A.

Several works have considered the problem of answering queries using views when the views are also associated with adornments describing limited access patterns. In [RSU95] it is shown that the bound given in [LMSS95] on the length of a possible rewriting does not hold anymore. To illustrate, consider the following example, where the binary relation Cites stores pairs of papers X, Y, where X cites Y. Suppose we have the following views with their associated adornments:

CitationDB^{bf}(X,Y) :- Cites(X,Y)
CitingPapers^{f}(X) :- Cites(X,Y)

and suppose we have the following query asking for all the papers citing paper #001:

Q(X) :- Cites(X,001)

The bound given in [LMSS95] would require that if there exists a rewriting, then there is one with at most one atom, the size of the query. However, the only possible rewriting in this case is:

q(X) :- CitingPapers(X), CitationDB(X,001).

[RSU95] shows that in the presence of binding patterns, it is sufficient to consider a slightly larger bound on the size of the rewriting: $n+v$, where n is the number of subgoals in the query and v is the number of variables in the query. Hence, the problem of finding an equivalent rewriting of the query using a set of views is still NP-complete.

The situation becomes more complicated when we consider maximally-contained rewritings. As the following example given in [KW96] shows, there may be *no* bound on the size of the rewriting. In the following example, the relation DBpapers denotes the set of papers in the database field, and the relation AwardPapers stores papers that have received awards (in databases or any other field). The following views are available:

DBSource^{f}(X) :- DBpapers(X)
CitationDB^{bf}(X,Y) :- Cites(X,Y)
AwardDB^{b}(X) :- AwardPaper(X)

The first source provides all the papers in databases, and has no binding-pattern limitations. The second source, when given a paper, will return all the papers that are cited by it. The third source, when given a paper, returns whether the paper is an award winner or not.

The query asks for all the papers that have won awards:

Q(X) :- AwardPaper(X).

Since the view AwardDB requires its input to be bound, we cannot query it directly. One way to get solutions to the query is to obtain the set of all database papers from the view DBSource, and perform a dependent join with the view AwardDB. Another way would be to begin by retrieving the papers in DBSource, join the result with the view CitationDB to obtain all papers cited by papers in DBSource, and then join the result with the view AwardDB. As the rewritings below show, we can follow any length of citation chains beginning with papers in DBSource and obtain answers to the query that were possibly not obtained by shorter chains. Hence, there is no bound on the length of a rewriting of the query using the views.

Q'(X) :- DBSource(X), AwardDB(X)
Q'(X) :- DBSource(V), CitationDB(V,X_1) . . . , CitationDB(X_n,X), AwardDB(X).

Fortunately, as shown in [DL97], we can still find a finite rewriting of the query using the views, albeit a recursive one. The following datalog rewriting will obtain all the possible answers from the above views. The key in constructing the program is to define a new intermediate relation papers whose extension is the set of all papers reachable from papers in databases, and is defined by a transitive closure over the view CitationDB.

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

In [DL97] it is shown that a maximally-contained rewriting of the query using the views can always be obtained with a recursive rewriting. Friedman and Weld [FW97] and Lambrecht et al. [LKG99] describe additional optimizations to this basic algorithm.

8.4 Other Extensions

Several authors have considered additional extensions of the query rewriting problems in various contexts. We mention some of them here.

Extensions to the query language: In [AGK99, Dus98] the authors consider the rewriting problem when the views may contain unions. The presence of inclusion dependencies on the database relations introduces several subtleties to the query rewriting problem, which are considered in [Gry98]. In [Mil98], the author considers the query rewriting for a language that enables querying the schema and data uniformly, and hence, names of attributes in the data may become constants in the extensions of the views.

Semi-structured data: The emergence of XML as a standard for sharing data on the WWW has spurred significant interest in building systems for integrating XML data from multiple sources. The emerging formalisms for modeling XML data are variations on labeled directed graphs, which have also been used to model semi-structured data [Abi97, Bun97].

The model of labeled directed graphs is especially well suited for modeling the irregularity and the lack of schema which are inherent in XML data. Several languages have been developed for querying semi-structured data and XML [AQM$^+$97, FFLS97, BDHS96, DFF$^+$98].

Several works have started considering the problem of answering queries using views when the views and queries are expressed in a language for querying semi-structured data. There are two main difficulties that arise in this context. First, such query languages enable using *regular path expressions* in the query, to express navigational queries over data whose structure is not well known a priori. Regular path expressions essentially provide a very limited kind of recursion in the query language. In [CGLV99] the authors consider the rewriting problem for the restricted case where the views contains a single atom involving a regular path expression. The second problem arises from the rich restructuring capabilities which enable the creation of arbitrary graphs in the output. The output graphs can also include nodes that did not exist in the input data. In [PV99] the authors consider the rewriting problem in the case where the query can create *answer trees*, and queries do not involve regular path expressions with recursion.

Infinite number of views: Two works have considered the problem of answering queries using views in the presence of an *infinite* number of views [LRU96, VP97]. The motivation is that when a data source has the capability to perform *local* processing, it can be modeled by a (possibly infinite) set of views it can supply, rather than a single one. Hence, to answer queries using such sources, one need not only choose which sources to query, but also which query to send to it out of the set of possible queries it can answer. In [LRU96, VP97] it is shown that in certain important cases the problem of answering a query using an infinite set of views is decidable. Of particular note is the case in which the set of views that a source can answer is described by the finite unfoldings of a datalog program.

Description Logics: Description logics are a family of logics for modeling complex hierarchical structures. A description logic enables to define sets of objects by specifying their properties, and then to reason about the relationship between these sets (e.g., subsumption, disjointness). A description logic also enables reasoning about individual objects, and their membership in different sets. One of the reasons that description logics are useful in data management is their ability to describe complex models of a domain and reason about interschema relationships [CL93]. For that reason, description logics have been used in several data integration systems [AKS96, LRO96]. Borgida [Bor95] surveys the use of description logics in data management.

Several works have considered the problem of answering queries using views when description logics are used to model the domain. In [BLR97] it is shown that in general, answering queries using views in this context may be NP-hard, and presents cases in which we can obtain a maximally-contained rewriting of a query in recursive datalog. The complexity of answering queries using views for an expressive description logic (which also includes n-ary relations) is studied in [CGL99].

9 Conclusions and Future Work

As this survey has shown, the problem of answering queries using views raises a multitude of challenges, ranging from theoretical foundations to considerations of a more practical nature. While algorithms for answering queries using views are already being incorporated into commercial database systems [BDD$^+$98], these algorithms will have even more importance in data integration systems and data warehouse design. Furthermore, answering queries using views is a key technique to give database systems the ability of maintaining physical data independence.

There are many issues that remain open in this realm. Although we have touched upon several query languages and extensions thereof, many cases remain to be investigated. Of particular note are studying rewriting algorithms in the presence of a wider class of integrity constraints on both the database and view relations, and studying the effect of restructuring capabilities of query languages (as in OQL or languages for querying semistructured data [BDHS96, AQM$^+$97, FFLS97]).

An important direction for research is to bridge the dichotomy in the treatment of the problem of answering queries using views, as exposed by sections 5 and 6. The challenge is, on the one hand, to extend query optimization algorithms to incorporate a large number of views that contain complex predicates, and on the other hand, to enable data integration systems to choose an *optimal* plan of answering a query using a set of sources.

The context of data warehouse design, when one tries to select a set of views to materialize in the warehouse, raises another challenge. The data warehouse design problem is often treated as a problem of *search* through a set of warehouse configurations. In each one of the states of the space we need to determine whether the workload queries anticipated on the warehouse can be answered using the selected views, and estimate the cost of the configuration. In this context it is important to be able to reuse the results of the computation from the previous state in the search space. In particular, this raises the challenge of developing *incremental* algorithms for answering queries using views, which can compute rewritings more efficiently when only minor changes are made to the set of available views.

Finally, an obvious but relatively overlooked question is the experimental validation of algorithms for answering queries using views, and a careful exploration of their performance.

Acknowledgments

I would like to thank Phil Bernstein, Todd Millstein and Rachel Pottinger for comments on earlier drafts of this paper.

References

[Abi97] Serge Abiteboul. Querying semi-structured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996.

[AD98] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle, WA, 1998.

[AGK99] Foto Afrati, M. Gergatsoulis, and Th. Kavalieros. Answering queries using materialized views with disjunctions. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, 1999.

[AKS96] Yigal Arens, Craig A. Knoblock, and Wei-Min Shen. Query reformulation for dynamic information integration. *International Journal on Intelligent and Cooperative Information Systems*, (6) 2/3:99–130, June 1996.

[AQM$^+$97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[BDD$^+$98] Randall Bello, Karl Dias, Alan Downing, James Feenan, Jim Finnerty, William Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in oracle. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 659–664, 1998.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 505–516, Montreal, Canada, 1996.

[BLR97] Catriel Beeri, Alon Y. Levy, and Marie-Christine Rousset. Rewriting queries using views in description logics. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona., 1997.

[Bor95] Alex Borgida. Description logics in data management. *IEEE Trans. on Know. and Data Engineering*, 7(5):671–682, 1995.

[Bun97] Peter Buneman. Semistructured data. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 117–121, Tucson, Arizona, 1997.

[CGL99] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Answering queries using views in description logics. In *Working notes of the KRDB Workshop*, 1999.

[CGLV99] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Rewriting of regular expressions and regular path queries. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1999.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Taipei, Taiwan, 1995.

[CL93] T. Catarci and M. Lenzerini. Representing and using interschema knowledge in cooperative information systems. *Journal of Intelligent and Cooperative Information Systems*, 1993.

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

[CNS99] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 155–166, 1999.

[CR94] C. Chen and N. Roussopoulos. Implementation and performance evaluation of the ADMS query optimizer. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, March 1994.

[CR97] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[CV93] Surajit Chaudhuri and Moshe Vardi. 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., 1993.

[CV94] Surajit Chaudhuri and Moshe Vardi. 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, 1994.

[DFF$^+$98] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. http://www.research.att.com/ mff/xml/w3c-note.html, 1998.

[DFJ$^+$96] Shaul Dar, Michael J. Franklin, Bjorn Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 330–341, 1996.

[DG97a] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Tucson, Arizona., 1997.

[DG97b] Oliver M. Duschka and Michael R. Genesereth. Query planning in infomaster. In *Proceedings of the ACM Symposium on Applied Computing*, San Jose, CA, 1997.

[DL97] Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1997.

[Dus97] Oliver Duschka. Query optimization using local completeness. In *Proceedings of the AAAI Fourteenth National Conference on Artificial Intelligence*, 1997.

[Dus98] Oliver M. Duschka. *Query Planning and Optimization in Information Integration.* PhD thesis, Stanford University, Stanford, California, 1998.

[EGW94] Oren Etzioni, Keith Golden, and Daniel Weld. Tractable closed world reasoning with updates. In *Proceedings of the Conference on Principles of Knowledge Representation and Reasoning, KR-94.*, 1994. Extended version to appear in *Artificial Intelligence.*

[FFLS97] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, September 1997.

[FK99] D. Florescu and D. Kossmann. Storing and querying xml data using an rdbms. *IEEE Data Engeneering Bulletin*, 22(3):27–34, September 1999.

[FLM98] Daniela Florescu, Alon Levy, and Alberto Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3):59–74, September 1998.

[Flo96] Daniela D. Florescu. *Search Spaces for Object-Oriented Query Optimization.* PhD thesis, Univerisity of Paris VI, France, 1996.

[FLSY99] Daniela Florescu, Alon Levy, Dan Suciu, and Khaled Yagoub. Optimization of run-time management of data intensive web sites. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[FRV96] Daniela Florescu, Louiqa Rashid, and Patrick Valduriez. Answering queries using OQL view expressions. In *Workshop on Materialized Views, in cooperation with ACM SIGMOD*, Montreal, Canada, 1996.

[FW97] M. Friedman and D. Weld. Efficient execution of information gathering plans. In *Proceedings of the International Joint Conference on Artificial Intelligence, Nagoya, Japan*, 1997.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1995.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 208–219, 1997.

[GM99a] Gosta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, 1999.

[GM99b] Ashish Gupta and Interpal Mumick, editors. *Materialized Views: Techniques, Implementations and Applications.* The MIT Press, 1999.

[GMPQ$^+$97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[GRT99] Stephane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. Querying aggregate data. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 174–184, 1999.

[Gry98] Jarek Gryz. Query folding with inclusion dependencies. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Orlando, Florida, 1998.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, 1996.

[IL84] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[KB96] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.

[Klu88] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, pages 35(1): 146–160, 1988.

[KMT98] Phokion Kolaitis, David Martin, and Madhukar Thakur. On the complexity of the containment problem for conjunctive queries with built-in predicates. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle, WA, 1998.

[KW96] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, 1996.

[Lev96] Alon Y. Levy. Obtaining complete answers from incomplete databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[LFS97] Alon Y. Levy, Richard E. Fikes, and Shuky Sagiv. Speeding up inferences using relevance reasoning: A formalism and algorithms. *Artificial Intelligence*, 97(1-2), 1997.

[LKG99] Eric Lambrecht, Subbarao Kambhampati, and Senthil Gnanaprakasam. Optimizing recursive information gathering plans. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1204–1210, 1999.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[LRU96] Alon Y. Levy, Anand Rajaraman, and Jeffrey D. Ullman. Answering queries using limited external processors. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Montreal, Canada, 1996.

[LS93] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 171–181, Dublin, Ireland, 1993.

[Mil98] R. J. Miller. Using schematically heterogeneous structures. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 189–200, Seattle, WA, 1998.

[MMS79] David Maier, Alberto Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):455–469, 1979.

[PL99] Rachel Pottinger and Alon Levy. A scalable algorithm for answering queries using views. Submitted for publication, 1999.

[PV99] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semistructured data. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 1999.

[Qia96] Xiaolei Qian. Query folding. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 48–55, New Orleans, LA, 1996.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

[SAC$^+$79] P. Selinger, M Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in relational database systems. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, Massachusetts, 1979.

[Sag88] Yehoshua Sagiv. Optimizing datalog programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, Los Altos, CA, 1988.

[SDJL96] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering SQL queries using materialized views. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[SGT$^+$99] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[Shm93] Oded Shmueli. Equivalence of datalog queries is undecidable. *Journal of Logic Programming*, 15:231–241, 1993.

[SR92] Divesh Srivastava and Raghu Ramakrishnan. Pushing constraint selections. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Diego, CA., 1992.

[SY81] Y. Sagiv and M. Yannakakis. Equivalence among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1981.

[TS97] Dimitri Theodoratos and Timos Sellis. Data warehouse design. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[TSI94] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 367–378, Santiago, Chile, 1994.

[TSI96] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2):101–118, 1996.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes I, II.* Computer Science Press, Rockville MD, 1989.

[Ull97] Jeffrey D. Ullman. Information integration using logical views. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[Val87] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2):218–246, 1987.

[VP97] Vasilis Vassalos and Yannis Papakonstantinou. 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, 1997.

[YKL97] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Athens, Greece, 1997.

[YL87] H. Z. Yang and P. A. Larson. Query transformation for PSJ-queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 245–254, Brighton, England, 1987.

[ZO93] X. Zhang and M. Z. Ozsoyoglu. On efficient reasoning with implication constraints. In *Proc. of the Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 236–252, 1993.