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 maximal set of answers for Q that we can obtain from the views? If we can access both the views and the database relations, what is the cheapest query execution plan for answering Q?

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. Such savings are especially significant in decision support applications when the views and queries contain grouping and aggregation. Furthermore, in some cases, certain indices can be modeled as precomputed views (e.g., join indices [Val87]),[1] 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.

[1] Strictly speaking, to model join indices we need to extend the logical model to refer to row IDs.

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 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 [LRO96b, 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.

In the area of data warehouse design we need to choose a set of views (and indexes on the views) to materialize in the warehouse [HRU96, TS97, YKL97, GHRU97, ACN00, CG00]. Similarly, in web-site design, the performance of a web site can be significantly improved by choosing a set of views to materialize [FLSY99]. In both of these problems, a first step in determining the utility of a choice of views is to ensure that the views are sufficient for answering the queries we expect to receive over the data warehouse or the web site. 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 physically at the physical level as a set 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 many applications of the problem of answering queries using views has spurred a flurry of research, ranging from theoretical foundations to algorithm design and implementation in several commercial systems. This article surveys the current state of the art in this area, and classifies the works into a coherent framework based on a set of dimensions along which the treatments of the problem differ. The dimensions include the query and view definition languages, the output of the algorithm (whether it's a query execution plan or a query rewriting), and whether the rewriting is equivalent to the original query or only contained in it.

The treatments of the problem differ mainly depending on whether they are concerned with

query optimization and database design or with data integration. 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 query optimizers to accommodate the presence of views. In this context, it is also necessary the result be equivalent to the original query. 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. Because the data sources may not entirely cover the domain, we sometimes need to settle for a contained query rewriting, rather than an equivalent one. 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 the dimensions along which we can study the problem. Section 3 defines 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 describes how materialized views have been incorporated into query optimization. Section 6 describes algorithms for answering queries using views that were developed in the context of data integration. 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 and queries access-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], selection of which views to maintain in a data warehouse [HRU96, TS97, GHRU97, Gup97, YKL97, GM99c, CG00] or automated selection of indexes [CN98b, CN98a].

2 Motivation and Illustrative Examples

Before beginning the detailed technical discussion, we motivate the problem of answering queries using views through some of its applications. In particular, this section serves to illustrate the wide and seemingly disparate range of applications of the problem. We end the section by enumerating a set of dimensions along which we will classify the treatments of the problem.

We use the following familiar university schema in our examples throughout the paper. We assume that professors and students and departments are uniquely identified by their names, and courses are uniquely identified by their numbers. The Registered relation describes the students' registration in classes, while the Major relation describes in which department a particular student is majoring (we assume for simplicity that every department has a single major program).

Prof(name, area)
Course(c-number, title)
Teaches(prof, c-number, quarter,evaluation)
Registered(student, c-number, quarter)
Major(student, dept)
WorksIn(prof, dept)
Advises(prof, student).

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 registered in Ph.D-level classes taught by professors in the Database area: (in our example university graduate level classes have numbers 400 and above, and Ph.D-level courses numbers of 500 and above).

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

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

```
create view Graduate as
select   student, title, c-number, quarter
from     Registered, Course
where    Registered.c-number=Course.c-number and Course.c-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.c-number=Graduate.c-number and Teaches.quarter=Graduate.quarter and
         Graduate.c-number ≥ 500 and Prof.area="DB".
```

The resulting evaluation will be cheaper because the view Graduate has 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 university). It is important to note that 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 relations Course and Registered have indexes on the c-number attribute. In this case, if the view Graduate does not have any indexes, then evaluating the query directly from the database relations may 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.

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

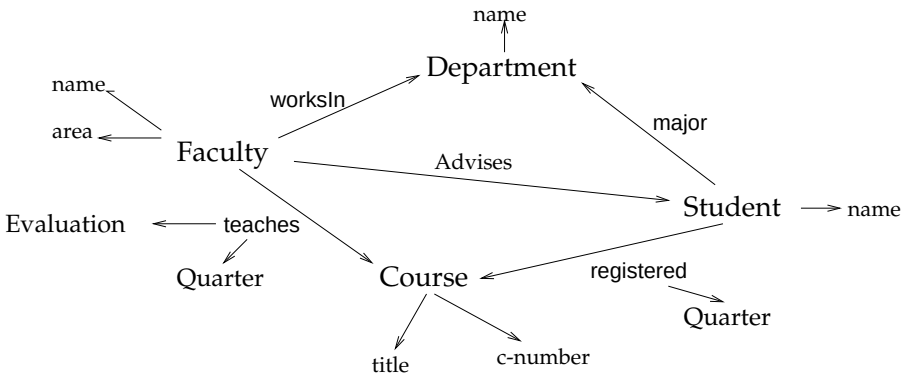

Figure 1: An Entity/Relationship diagram for the university domain. Note that quarter is an attribute of the relationships registered and teaches.

```
def_gmap G1 as b+-tree by
    given Student.name
    select Dept
    where Student major Dept.
```

```
def_gmap G2 as b+-tree by
    given Student.name
    select Course.c-number
    where Student registered Course.
```

```
def_gmap G3 as b+-tree by
    given Course.c-number
    select Dept
    where Student registered Course and Student major Dept.
```

Figure 2: GMAPs for the university domain.

the data (e.g., as tables with their named attributes) and the physical view of the data (i.e., how it is laid out on disk). With the exception of horizontal or vertical partitioning of relations into multiple files, relational database systems 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, DFS99], 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 as 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. 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 in which they major, indexed by a B^+-tree on attribute Student.name. The GMAP G2 stores an index from the names of students to the numbers of the courses in which they are registered. The GMAP G3 stores an index from course numbers to departments whose majors are enrolled in the course. 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 these 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 cheapest one. Note that in contrast to the query optimization context, we *must* use the views to answer a given query, because all the data is stored in the GMAPs,

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

select Student.name, Dept
where Student registered Course **and** Student major Dept **and** Course.c-number≥500.

The query can be answered in two ways. First, since Student.name uniquely identifies a student, we can take the join of G1 and G2, and then apply a selection Course.c-number≥500, and a projection on Student.name and Dept. A second solution would be to join G3 with G2 and select Course.c-number≥500. In fact, this solution may even be more efficient because G3 has an index on the course number and therefore the intermediate joins may be much smaller.

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 [Wie92]) 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 the constraints on the contents of the source.

One of the approaches for specifying source descriptions, which has been adopted in several systems ([LRO96b, 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, where the mediated schema exposed to the user is our university schema, except that the relations Teaches and Course have an additional attribute identifying the university at which a course is being taught:

Teaches(prof, c-number, quarter, evaluation, univ)
Course(c-number, title, univ)

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

create view DB-courses as
select Course.title, Teaches.prof, Course.c-number, Course.univ
from Teaches, Course
where Teaches.c-number=Course.c-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.c-number, Course.univ
from Teaches, Course
where Teaches.c-number=Course.c-number **and**
Course.univ="UW" **and** Teaches.univ="UW" **and** Course.c-number≥500.

If we were to ask the data integration system who teaches courses titled "Database Systems" at UW, it would be able to answer the query 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, c-number
from DB-courses
where univ"UW" **and** c-number≥400

```
        UNION
select  title, c-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.

Other applications: 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, YFIV00] it is shown that precomputing views can significantly speed up the response time from web sites, which again raises the question of view selection.

2.4 Dimensions of the problem

As illustrated by the examples, there are several dimensions along which we can classify the treatments of the problem of answering queries using views. The following are the main dimensions:

- **The query and view languages:** the algorithms and results regarding answering queries using views depend crucially on the language used for expressing the views and the queries. Much of the work has concentrated on select-project-join queries, but numerous extensions have been considered for queries with grouping, aggregation and multiple SQL blocks (Section 5.3), recursive queries (Section 7.2), views with access-pattern limitations (Section 8.2), queries over object-oriented databases 8.1 and queries over semi-structured data and description logics (Section 8.3). In addition to the query language, treatments differ on whether they consider set or bag semantics.

- **Output of the algorithm:** Given a query Q, and a set of views $\mathcal{V}$, there are three possible outputs an algorithm may produce: (1) an expression Q' that references the views and is either equivalent to or contained in Q, (2) a query execution plan for answering Q using the views (and possibly the database relations), or (3) the answers to Q. In the latter case, we assume the extensions of the views $\mathcal{V}$ are given as well. The works on data integration have focused on (1), while the works on query optimization and database design have focused on (2) (which clearly yields an expression Q' as a side effect). We consider several works that address (3) in Section 7.3.

- **Equivalent versus maximally-contained rewritings:** in the contexts of query optimization and maintenance of physical data independence (Section 5), the focus has been on finding equivalent rewritings of the query, while the focus of the work in data integration (Section 6) has focused on finding maximally-contained rewritings.

- **Completeness of the views:** in the context of query optimization, the materialized views are assumed to contain *all* the tuples in their definition. In contrast, data integration applications often assume that the views (i.e., data sources) may contain only a subset of the tuples that satisfy their definition. As we see in Section 7.3, assuming completeness often makes the problem of answering queries using views harder.

3 Problem Definition

In this section we define the basic terminology used throughout this paper. We define the concepts of query containment and query equivalence that provide a semantic basis for comparing between queries, and then define the problem of answering queries using views. Finally, we define the problem of extracting *all* the answers to a query from a set of views (a.k.a. the set of certain answers).

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. We will distinguish between queries that involve arithmetic comparison predicates (e.g., $\leq, <, \neq$) and those that do not. Our discussion of answering queries using views in the context of data integration systems will require considering recursive datalog queries. We recall the basic concepts of datalog in Section 6.

In our discussion, we denote the result of computing the query Q over the database D by $Q(D)$. We often refer to queries that reference named views (e.g., in query rewritings). In that case, $Q(D)$ refers to the result of computing Q after the views have been computed from D.

3.1 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 bag semantics. In the context of our discussion it is important to note that the definitions below also apply to queries that may reference named views.

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 all databases 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 $Q_2 \sqsubseteq Q_1$. □

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 select-project-join queries and unions thereof [CM77, SY81], queries with arithmetic comparison predicates [Klu88, LS93, ZO93, KMT98], and recursive queries [Shm93, Sag88, LS93, CV92, CV94].

3.2 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 $V_1, \ldots, V_m$.[2] In SQL, a query refers only to

[2]Note that rewritings that refer only to the views were called *complete rewritings* in [LMSS95].

the views if all the relations mentioned in the from clause are views. In practice, we may also be interested in rewritings that can also refer to the database relations. Conceptually rewritings that refer to the database relations do not introduce new difficulties, because we can always simulate the previous case by inventing views that mirror precisely the database tables.

As we saw in Section 2, 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.

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' is equivalent to Q.

□

In the context of data integration, we often need to consider maximally-contained rewritings. Note that maximality of a rewriting is defined w.r.t. a particular query language:

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

□

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 candidate rewriting of a query is an equivalent or contained rewriting. However, by themselves, these algorithms do not provide a solution to the problem of answering 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*, originally introduced in [AD98]. In the definition, we distinguish the case in which the view extensions are assumed to be 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 $\bar{a} \in Q(D)$ for all databases 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 $\bar{a} \in Q(D)$ for all databases D such that $V_i(D) \supseteq v_i$ for every i, $1 \leq i \leq m$. □

The intuition behind the definition of certain answers is the following. The extensions of a set of views do 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 instructive to examine a few examples and gain an intuition for the conditions under which a view is usable for answering a query. In this section we consider select-project-join queries under set semantics. Note that in some cases a view may be usable in maximally-contained rewritings but not in equivalent rewritings.

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 in order to be part of an equivalent rewriting. If the view applies a logically stronger predicate, it may be part of a contained rewriting.

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.

```
select   Advises.prof, Advises.student, Registered.quarter
from     Registered, Teaches, Advises
where    Registered.c-number=Teaches.c-number 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 the predicate Registered.quarter ≥ "winter98" in the query. However, since the view selects the attribute Registered.quarter, the stronger predicate can be applied in the rewriting.

```
create view V1 as
select   Registered.student, Teaches.prof, Registered.quarter
from     Registered, Teaches
where    Registered.c-number=Teaches.c-number 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. The only way to find the matching professor names would be to join V_2 with the relations Registered and Teaches.

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

```
create view V4 as
select  Registered.student, Registered.quarter, prof
from    Registered, Teaches, Advises
where   Registered.c-name=Teaches.c-name and
        Registered.quarter=Teaches.quarter and
        Teaches.prof=Advises.prof and
        Registered.quarter ≥ "winter98".
```

```
create view V5 as
select  Registered.student, Teaches.prof, Registered.quarter
from    Registered, Teaches
where   Registered.c-name=Teaches.c-name 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.quarte Since the attributes Teaches.quarter and Registered.quarter are not selected by V_3, the join predicate cannot be applied in the rewriting. View V_4 considers only the courses whose professor advises at least one student. Hence, the view applies an additional condition that does not exist in the query, and cannot be used in an equivalent rewriting unless we allow union and negation in the rewriting language. However, if we have an integrity constraint stating that every professor advises at least one student, then an optimizer should be able to realize that V_4 is usable. Finally, view V_5 applies a stronger predicate than in the query (Registered.quarter > "summer98"), and is therefore usable for a containe rewriting, but not for an equivalent rewriting of the query.

To summarize, the following conditions need to hold in order for a select-project-join view V to be usable in an equivalent rewriting of a query Q. The intuitive conditions below 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 occurrences of tables mentioned in the from clause of V to those mentioned in the from clause of Q, mapping every table name to itself. In the case of bag semantics, ψ must be a 1-1 mapping, whereas for set semantics, ψ can be a many-to-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 (or from the original table if it's available).

Finally, we note that the introduction of bag semantics introduces additional subtleties. In particular, we must ensure that the multiplicity of answers required in the query are not lost in the

views (e.g., by the use of distinct), and are not increased (e.g., by the introduction of additional joins).

5 Incorporating Materialized Views into Query Optimization

This section describes the different approaches to incorporating materialized views into query optimization. The focus of these algorithms is to judiciously decide when to use views to answer a query, and their output is an execution plan for the query. The approaches differ depending on which phase of query optimization was modified to consider materialized views. Section 5.1 describes how the System R-style join enumeration algorithm is modified to consider materialized views [CKPS95, TSI96]. Section 5.2 considers approaches in which materialized views are considered in the query rewrite phase [ZCL+00, DPT99, PDST00] or as a separate add-on to the optimizer [BDD+98]. Section 5.3 discusses some of the issues that arise when rewriting algorithms are extended to consider grouping and aggregation. These extensions are key to incorporating materialized views into decision support applications.

5.1 Modifying the Join Enumeration Algorithm

In this section we consider select-project-join queries and discuss the changes that need to be made to a join enumeration algorithm to incorporate materialized views. To illustrate the changes 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., chooses the best access paths to every table mentioned in the query. In phase n, the algorithm considers plans of size n, by combining pairs of plans obtained in the previous phases.[3] The algorithm terminates after constructing plans that cover all the relations in the query.

Intuitively, the efficiency of System-R stems from the fact that it partitions query execution plans into *equivalence classes*, and only considers a single execution class for every equivalence class. 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 the relations mentioned in the query.

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.

We illustrate the changes to the join enumeration algorithm with an example that includes the following views:

```
create view V1 as
select    student, dept
from      Major
```

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

[3]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 $n-k$.

```
create view V3 as
select    dept, c-number
from      Registered, Major
where     Registered.student=Major.student and c-number≥500.
```

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

```
select    student, dept
from      Registered, Major, Course
where     Registered.student=Major.student and Registered.c-number=Course.c-number and
          c-number≥500.
```

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. These differences formed the basis for the algorithms described in [TSI96, CKPS95].

A. In the first iteration 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.

In our example, 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 some of 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.

B. 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 several changes to the traditional algorithm:

   1. **Termination testing:** the algorithm needs to distinguish *partial query execution plans* of the query from *complete execution plans*. The enumeration of the possible join orders terminates when there are no more unexplored partial plans. In contrast, in the traditional setting the algorithm terminates after considering the equivalence classes that include all the relations in the query.
   2. **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 than 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. **Combining partial plans:** 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 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.

In our example, the algorithm will consider in the second iteration 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. The algorithm will consider the following combinations (in this discussion we ignore the choice of inner versus outer input to the join):

- the join of V1 and V2 on the attribute student: This join produces a partial result to the query. There are two ways to extend this join to complete execution plan. The first is to apply an additional selection on the c-number attribute and a projection on student and dept. The second, which is explored in the subsequent iteration is to join the result with V3. Hence, the algorithm produces one complete execution plan and keeps V1 $\bowtie$ V2 for the subsequent iterations.

  In principle, as explained in (3), the algorithm should also consider joining V1 and V2 on other attributes (e.g., V1.student=V2.c-number), 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 dept) and of V2 with V3 (on c-number): These two joins produce partial solutions to the query, but only if set semantics are considered (otherwise, the resulting rewriting will have multiple occurrences of the Major (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. One of the plans the algorithm will consider is the one in which the result of joining V2 and V3 is then joined with V1. Even though this plan may seem redundant compared to V1 $\bowtie$ V2, it may be cheaper depending on the available indexes on the views, because it enables pruning the (possibly larger) set of students based on the selective course number.

Variations on the above principles 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 goal of the algorithm described in [CKPS95] is to make use of materialized views in query evaluation. They consider select-project-join queries with bag semantics and which may also include arithmetic comparison predicates. Under bag 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 bi-directional 1-1 mapping between the two queries, which maps the join predicates of one query to those of the other [CV93]. Hence, if we ignore the arithmetic comparison operators, a view is usable only if it is isomorphic to a subset of the query. An additional difference between [TSI96] and [CKPS95] is that the latter searches the space of join orderings in a top-down

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.

fashion, compared to the bottom-up fashion in [TSI96]. However, since the algorithms consider different semantics, their search spaces are incomparable. 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.

5.2 Other approaches to incorporating materialized views

In [ZCL+00] the authors describe an approach where the algorithm for answering queries using views is incorporated into the rewrite phase of query optimization (in their case in the optimizer of IBM's DB2 UDB). In that work the algorithm operates on the Query Graph Model (QGM) representation of a query [HFLP89]. The algorithm attempts to match pairs of QGM boxes in the views with those in the query. The algorithm navigates the QGM in a bottom up fashion, starting from the leaf boxes. A match between a box in the query and in the view can be either (1) exact, meaning that the two boxes represent equivalent queries, or (2) may require a *compensation*. A compensation represents a set of additional operations that need to be performed on a box of the view in order to obtain an equivalent result to a box in the query. The algorithm considers a pair of boxes only after the match algorithm has been applied to every possible pair of their children. Therefore, the match (and corresponding compensation) can be determined without looking into the subtrees of their children. The algorithm terminates when it finds a match between the root of the query QGM and QGMs of the available views. The authors show that by considering rewritings at the QGM level, they are able to extend previous algorithms to handle SQL queries and views with multiple blocks, while previous algorithms considered only single block queries. As we point out in the next section, their algorithm also extends previous work to handle more complex types of grouping and aggregation.

In [BDD+98] the authors describe the implementation of a view rewriting algorithm in the Oracle 8i 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, such as functional dependencies and foreign key constraints. 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.

In [DPT99] the authors propose an elegant optimization framework that treats uniformly the use of materialized views, specialized indexes and semantic integrity constraints, all of which are represented as constraints. Their procedure involves two phases. In the first phase, the *chase,* the query is expanded to include any other structure (e.g,. materialized view or access structure) that is relevant to the query, resulting in a *universal* query plan. In the second phase, the *backchase,* the optimizer tries to remove structures (and hence joins) from the universal plan, in order to obtain a plan of minimal cost. The chase procedure is based on a generalization of the standard chase procedure to handle path conjunctive queries [PT99], thereby enabling the algorithm to handle certain forms of object-oriented queries. In [PDST00] the authors describe an implementation of the framework and experiments that prove its feasibility, focusing on methods for speeding up the backchase phase.

5.3 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, and in a sense *partially* projecting it out. If the query requires the same or a coarser grouping than performed in the view, and the aggregated column is either available or can be reconstructed from other attributes, then the view is still usable for 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. To make this example slightly more appealing, we assume the quarter attribute of the relation Teaches is replaced by a year attribute. 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    c-number, year, Max(evaluation) as maxeval, Count(*) as offerings
from      Teaches
```

where c-number $\geq$ 400
groupBy c-number, year.

The following query considers only Ph.D-level courses, 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 c-number $\geq$ 500
groupBy year.

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

select year, sum(offerings), Max(maxeval)
from V
where c-number $\geq$ 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 c-number). Thus, the answer to the query can be obtained by coalescing groups from the view. Second, since the view groups the answers by c-number and thereby loses the multiplicity of each course, we would have ordinarily not been able to use the view to compute the number of course offerings per year. 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, by summing the offerings per course.

Several works considered the problem of answering queries using views in the presence of grouping and aggregation. This first approach involved a set of transformations in the query rewrite phase [GHQ95]. In this approach, the algorithm performs syntactic transformations on the query until it is possible to identify a subexpression of the query that is identical to the view, and hence substitute the view for the subexpression. However, as the authors point out, the purely syntactic nature of this approach is a limiting factor in its applicability.

A more semantic approach is proposed in [SDJL96]. The authors 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 bag 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]. The work described in [ZCL$^+$00] extends the treatment of grouping and aggregation to consider multi-block queries and to multi-dimensional aggregation functions such as cube, rollup and grouping sets [GBLP98].

Several works [CNS99, GRT99, GT00] 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.

An interesting issue that has not received attention to date is extending the notion of maximally-contained rewritings to the presence of grouping and aggregation. In particular, one can imagine a notion of maximally-contained plans in which the answers provide the best possible *bounds* on the aggregated columns.[4]

6 Answering Queries Using Views for Data Integration

The previous section focused on extending query optimizers to accommodate the use of views. They were designed to handle cases where the number of views is relatively small (i.e., comparable to the size of the database schema), and cases where we require an equivalent rewriting of the query. In addition, for the mostpart, 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, the view definitions contain many complex predicates, whose goal is to express fine-grained distinctions between the contents of different data 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. Furthermore, in the context of data integration it is often assumed that the views are not complete, i.e., they may only contain a subset of the tuples satisfying their definition.

In this section we describe 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 [LRO96b] and later studied in [GM99a], the *inverse-rules algorithm* [Qia96, DGL00] which was implemented in the InfoMaster system [DG97b], and the MiniCon algorithm [PL00]. 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 rather a query referring to the view relations.

6.1 Datalog notation

For this and the next section, it is necessary to revert to datalog notation and terminology. Hence, below we provide a brief reminder of datalog notation and of conjunctive queries [Ull89, AHV95].

Conjunctive queries are able to express select-project-join queries 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

[4] I thank an anonymous reviewer for suggesting this problem.

must also appear in an ordinary subgoal. We refer to the subgoals of comparison predicates of a query Q by $C(Q)$.

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
from     Registered, Teaches, Advises
where    Registered.c-number=Teaches.c-number 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) :-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. It is important to recall that if a datalog program is not recursive, then it can be equivalently rewritten as a union of conjunctive queries, though possibly with an exponential blowup in the size of the query. As we see in Section 7.2, certain cases may require rewritings that are recursive datalog queries.

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 $Q(D)$ is the set of tuples computed for the query predicate. Note that since the number of tuples that can be computed for each relation is finite and monotonically increasing, the evaluation is guaranteed to terminate. Finally, we say that a datalog query refers only to views if instead of EDB predicates we have predicates referring to views (but we still allow arithmetic comparison predicates and IDB predicates).

6.2 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 conjunctive queries that may include atoms of arithmetic comparison predicates (hereafter referred to simply as predicates). 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. there is a unifier(see [AHV95] Pg. 293) θ for g and g_1, and

b. after applying $\theta_{h(V)}$ to the query and to the view, where $\theta_{h(V)}$ is the restriction of θ to the head variables of V, the predicates in Q and in V are mutually satisfiable, i.e., $\theta_{h(V)}(C(Q)) \wedge \theta_{h(V)}(C(V))$ is satisfiable.

The bucket contains the atom $\theta(head(V))$. 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 bucket 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,c-number,quarter,title) :- Registered(student,c-number,quarter), Course(c-number,title), c-number≥500, quarter≥Aut98.
V2(student,prof,c-number,quarter) :- Registered(student,c-number,quarter), Teaches(prof,c-number,quarter)
V3(student,c-number) :- Registered(student,c-number,quarter), year ≤ Aut94.
V4(prof,c-number,title,quarter) :- Registered(student,c-number,quarter), Course(c-number,title), Teaches(prof,c-number,quarter), quarter≤Aut97.

Suppose our query is:

q(S,C,P) :- Teaches(P,C,Q), Registered(S,C,Q), Course(C,T), C≥300, Q≥Aut95.

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 Teaches(P,C,Q) includes views V2 and V4, since the following mapping unifies the subgoal in the query with the corresponding Teaches subgoal in the views (thereby satisfying (a) above):

{ P → prof, C → c-number, Q → quarter }.

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,Q) contains the views V1 and V2, since the following mapping unifies the subgoal in the query with the corresponding Registered subgoal in the views:

{ S → student, C → c-number, Q → quarter }.

Teaches(P,C,Q)	Registered(S,C,Q)	Course(C,T)
V2(S',P,C,Q)	V1(S,C,Q,T')	V1(S',C,Q',T)
V4(P,C,T',Q)	V2(S,P',C,Q)	V4(P',C,T,Q')

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,Q) because after applying the unification mapping, the predicates Q $\geq$ *Aut*95 and Q $\leq$ Aut94 are mutually inconsistent.

Next, consider the bucket of the subgoal Course(C,T). The views V1 and V4 will be included in the bucket because of the mapping

{ C $\rightarrow$ c-number, T $\rightarrow$ title }.

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,C,P) :- V2(S',P,C,Q), V1(S,C,Q,T'), V1(S', C, Q', T).

As can be checked, this rewriting can be simplified by equating the variables S and S', and Q and Q', and then removing the third subgoal, resulting with

q'(S,C,P) :- V2(S',P,C,Q), V1(S,C,Q,T').

Another possibility that the bucket algorithm would explore is:

q'(S,C,P) :- V4(P, C, T', Q), V1(S,C,Q,T'), V4(P', C, T, Q').

However, this rewriting would be dismissed because the quarters given in V1 are disjoint from those given in V4. In this case, the views V1 and V4 are relevant to the query in *isolation*, but, if joined, would yield the empty answer.

Finally, the algorithm would also produce the rewriting

q'(S,C,P) :- V2(S,P,C,Q), V4(P, C, T', Q).

The reader should 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. In fact, 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. □

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. The bucket algorithm also extends naturally to cases where the query (but not the views) is a union of conjunctive queries, and to other forms of predicates in the query such as class hierarchies [LRO96a]. 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 [LRO96a].

The main disadvantage of the bucket algorithm is that the Cartesian product of the buckets may still be rather large. Furthermore, 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, though only in the size of the query and the view definition, and hence quite efficient in practice.

6.3 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: (we omit the quarter attribute of Registered for brevity in this example):

V3(dept, c-number) :- Major(student,dept), Registered(student,c-number).

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

Major(f_1(dept,X), dept) :- V3(dept,X)
Registered(f_1(Y, c-number), c-number) :- V3(Y,c-number)

Intuitively, the inverse rules have the following meaning. A tuple of the form (dept,c-number) in the extension of the view V3 is a *witness* of tuples in the relations Major and Registered. The tuple (dept,c-number) is a witness in the sense that it tells us two things:

- the relation Major contains a tuple of the form (Z, dept), for some value of Z.
- the relation Registered contains a tuple of the form (Z, c-number), 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,c-number). Formally, f_1 is a Skolem function (see [ABS99], Pg. 96) and therefore uninterpreted. Note that there may be several values of Z in the database that cause the tuple (dept,c-number) to be in the join of Major 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 $\mathcal{V}$, and
- the query Q.

As shown in [DG97a, DGL00], the inverse-rules algorithm returns the maximally-contained rewriting of Q using $\mathcal{V}$. In fact, the algorithm returns the maximally contained query even if Q is an arbitrary recursive datalog program.

Example 6.2 Suppose a query asks for the departments in which the students of the 444 course are majoring,

q(dept) :- Major(student,dept), Registered(student, 444)

and the view V3 includes the tuples:

{ (CS, 444), (EE, 444), (CS, 333) }.

The inverse rules would compute the following tuples:

Registered: { (f_1(CS,444), CS), (f_1(EE,444), EE), (f_1(CS,333), CS) }
Major: { (f_1(CS,444),444), (f_1(EE,444),444), (f_1(CS,333),333) }

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].

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 contain 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]).

A key advantage of the inverse-rules algorithm is its conceptual simplicity and modularity. As shown in [DGL00], extending the algorithm to handle functional dependencies on the database schema, recursive queries or the existence of access-pattern limitations can be done by adding another set of rules to the inverse rules. Furthermore, the algorithm produces the maximally-contained rewriting in time that is polynomial in the size of the query and the views. Note that the algorithm does not tell us whether the maximally-contained rewriting is equivalent to the original query, which would contradict the fact that the problem of finding an equivalent rewriting is NP-complete [LMSS95] (see Section 7).

On the other hand, using the resulting rewriting produced by the algorithm for actually evaluating queries from the views has a significant drawback, since it insists on recomputing the extensions of the database relations. In our example, evaluating the inverse rules computes tuples for Registered and Major, 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, much 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 unfold the inverse rules and remove redundant subgoals from the unfolded rules. Unfolding the rules turns out to be similar to (but still slightly better than) the second phase of the bucket algorithm, where we consider the Cartesian product of the buckets.

6.4 The MiniCon algorithm

The MiniCon algorithm [PL00] addresses the limitations of the previous algorithms. The key idea underlying the algorithm is a change of perspective: instead of building rewritings by combining

rewritings for each of the query *subgoal* or the database relation, we consider how each of the *variables* in the query can interact with the available views. The result is that the second phase of the MiniCon algorithm needs to consider drastically fewer combinations of views. The following example illustrates the key idea of MiniCon. Consider the query

q(D) :- Major(S, D), Registered(S, 444, Q), Advises(P, S)

and the views:

```
V1(dept) :- Major(student,dept), Registered(student, 444, quarter).
V2(prof, dept, area) :- Advises(prof, student), Prof(name, area)
V3(dept, c-number) :- Major(student,dept), Registered(student, c-number, quarter),
                      Advises(prof, student).
```

The bucket algorithm considers each of the subgoals in the query in isolation, and therefore puts the view V1 into the buckets of Major(student, dept) and Registered(student, 444, quarter). However, a careful analysis reveals that V1 cannot possibly be used in a rewriting of the query. The reason is that since the variable student is not in the head of the view, then in order for V1 to be useful, it must contain the subgoal Advises(prof,student) as well. Otherwise, the join on the variable S in the query cannot be applied using the results of V1.

The MiniCon algorithm starts out like the bucket algorithm, considering which views contain subgoals that correspond to subgoals in the query. However, once the algorithm finds a partial mapping from a subgoal g in the query to a subgoal g_1 in a view V, it changes perspective and looks at the variables in the query. The algorithm considers the join predicates in the query (which are specified by multiple occurrences of the same variable) and finds the minimal additional set of subgoals that *must* to be mapped to subgoals in V, given that g will be mapped to g_1. This set of subgoals and mapping information is called a *MiniCon Description* (MCD), and can viewed as a generalized bucket. Unlike buckets, MCDs are associated with *sets* of subgoals in the query. In the second phase, the MCDs are combined to produce the query rewritings.

In the above example, the algorithm will determine that it cannot create an MCD for V1 because it cannot apply the join predicates in the query. When V2 is considered, the MCD will contain only the subgoal Advises(prof, student). When V3 is considered, the MCD will include all of the query subgoals.

The key advantage of the MiniCon algorithm is that the second phase of the algorithm considers much fewer combinations of MCDs compared to the Cartesian product of the buckets or compared to the number of unfoldings of inverse rules. The work in [PL00] describes a detailed set of experiments that shows that the MiniCon significantly outperforms the inverse rules algorithm, which in turn outperforms the bucket algorithm. The paper shows exactly how these savings are obtained in the second phase of the algorithm. Furthermore, the experiments show that the algorithm scales up to hundreds of views with commonly occurring shapes such as chain, star and complete queries (see [MGA97] for an description of these query shapes).

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* datalog 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 co-NP-hard in the size of the extensions of the views.

7.1 Completeness and complexity of finding 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 query does not contain comparison predicates and has n subgoals, then there exists an equivalent conjunctive 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.[5]

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) we use a sound and complete algorithm for query containment for testing equivalence of rewritings, (2) when combining two subplans, the algorithm considers all possible join predicates on the attributes of the combined subplans, and (3) we consider self-joins of the views. These conditions essentially guarantee that the algorithm searches through all rewritings whose size is bounded by the size of the query. It is important to emphasize that the rewriting of the query that produces the most *efficient* plan for answering the query may have *more* conjuncts that the original query. The bound of [LMSS95] also guarantees that the bucket algorithm is guaranteed to find the maximally-contained rewriting of the query when the query does not contain arithmetic comparison predicates (but the views may), and that we consider unions of conjunctive queries as the language for the rewriting.

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.

In [RSU95] the authors extend the bound on the size of the rewriting to the case where the views contain access-pattern limitations (discussed in detail in Section 8.2). In [CR97] the authors exploit the close connection between the containment and rewriting problems, and show several

[5]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.

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 [DGL00]. 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 access-pattern limitations on the extensions of the views [DL97] (see Section 8.2 for a more detailed discussion), when views contain unions [AGK00] (though even recursion is does not always suffice here), 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:

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

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, DGL00] 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 [EGW97, 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 co-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 co-NP-hard. The following example is the crux of the proof of the co-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 co-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 co-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 that without comparison predicates in the views or the query, when either all the views are complete or all of them may contain incorrect tuples, finding all certain answers can be done in polynomial time in the size of the view extensions. In other cases, the problem is co-NP-hard.

Finally, [MLF00] considers the problem of relative query containment, i.e., whether the set of certain answers of a query Q_1 is always contained in the set of certain answers of a query Q_2. The paper shows that for the conjunctive queries and views with no comparison predicates the problem is Π_2^p-complete, and that the problem is still decidable in the presence of access pattern limitations.

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, DPT99], and views with access pattern limitations [RSU95, KW96, DL97].

8.1 Object Query Language

In [FRV96, Flo96] the authors 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 at least as old as 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. □

As noted in Section 5.2, the algorithm described in [DPT99, PDST00] also considers certain types of queries over object-oriented data.

8.2 Access 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 access-pattern limitations 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 a 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 access-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 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, DGL00], 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 by citation chains 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. In [FW97] and [LKG99] the authors describe additional optimizations to this basic algorithm.

8.3 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, ABS99]. 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 problem of rewriting a regular path query using a set of regular path views, and show that the problem is in 2EXPTIME (and checking whether the rewriting is an equivalent one is in 2EXPSPACE). In [CGLV00a] the authors consider the problem of finding all the certain answers when queries and views are expressed using regular path expressions, and show that the problem is co-NP-complete when data complexity (i.e., size of the view extensions) are considered. In [CGLV00b] the authors extend the results of [CGLV99, CGLV00a] to path expressions that include the inverse operator, allowing both forward and backward traversals in a graph.

The second problem that arises in the context of semi-structured data stems 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. For the most part, considering queries with restructuring remains an open research problem.

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 for this seemingly curious problem is that when a data source has the capability to perform *local* processing, it can be modeled by the (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, LRO96b]. 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

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 (e.g., [BDD+98, ZCL+00]), 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, DFF+99]).

As described in the article, different motivations have led to two strands of work on answering queries using views, one in the context of optimization and the other in the context of data integration. In part, these differences are due to the fact that in the data integration context the algorithms search for a maximally-contained rewriting of the query and assume that the number of views is relatively large. However, as we illustrated, the principles underlying the two strands are similar. Furthermore, interesting challenges arise as we try to bridge the gaps between these bodies of work. The first challenge is to extend the work on query optimization to handle a much larger number of more complex views. The second challenge is to extend data integration algorithms to choose judiciously the best rewritings of the query. This can be done by either trying to order the access to the data sources (as in [FKL97, DL99, NLF99]), or to combine the choice of rewritings with other adaptive aspects of query processing explored in data integration systems (e.g., [UFA98, IFF+99].

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, I believe that the next challenge is to develop algorithms for selecting views to materialize in a data warehouse, web site, or environment in which data is spread over multiple (possibly mobile) devices (e.g., [ILM+00]). Even though there has been work on this problem, the research is still in its infancy. The wealth of techniques developed for answering queries using views will be very useful in this realm.

Acknowledgments

I would like to thank Phil Bernstein, Todd Millstein, Rachel Pottinger and the anonymous reviewers for valuable comments on earlier drafts of this paper. I would like to acknowledge the support of a Sloan Fellowship and NSF Grant #IIS-9978567.

References

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

[ABS99] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web.* Morgan Kaufmann, 1999.

[ACN00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes in Microsoft SQL Server. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[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.

[AGK00] Foto Afrati, M. Gergatsoulis, and Th. Kavalieros. Personal communication, 2000.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Weseley, 1995.

[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.

[CG00] Rada Chirkova and Michael Genesereth. Linearly bounded reformulations of conjunctive databases. In *Proc. of the Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 2000.

[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.

[CGLV00a] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Answering regular path queries using views. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2000.

[CGLV00b] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. View-based query processing for regular path queries with inverse. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 2000.

[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.

[CN98a] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 367–378, 1998.

[CN98b] S. Chaudhuri and V. R. Narasayya. Microsoft index tuning wizard forSQL Server 7.0. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 553–554, 1998.

[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.

[CV92] Surajit Chaudhuri and Moshe Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 55–66, San Diego, CA., 1992.

[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.

[DFF+99] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. In *Proceedings World-Wide Web 8 Conference*, 1999.

[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.

[DFS99] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semi-structured data with STORED. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 431–442, 1999.

[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.

[DGL00] Oliver Duschka, Michael Genesereth, and Alon Levy. Recursive query plans for data integration. *Journal of Logic Programming, special issue on Logic Based Heterogeneous Information Systems*, 43(1):49–73, 2000.

[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.

[DL99] Anhai Doan and Alon Levy. Efficiently ordering query plans for data integration. In *IJCAI Workshop on Intelligent Data Integration*, Stockholm, Sweden, August 1999.

[DPT99] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints and optimization with universal plans. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[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.

[EGW97] Oren Etzioni, Keith Golden, and Daniel S. Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89(1-2):113–148, 1997.

[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.

[FKL97] Daniela Florescu, Daphne Koller, and Alon Levy. Using probabilistic information in data integration. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 216–225, Athens, Greece, 1997.

[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 Raschid, 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.

[GBLP98] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, 1998.

[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 Inderpal Mumick, editors. *Materialized Views: Techniques, Implementations and Applications.* The MIT Press, 1999.

[GM99c] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, pages 453–470, 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.

[GT00] Stephane Grumbach and Leonardo Tininini. On the content of materialzed aggregate views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 2000.

[Gup97] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[HFLP89] Laura Haas, Johann Freytag, Guy Lohman, and Hamid Pirahesh. Extensible query processing in Starburst. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 377–388, 1989.

[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.

[IFF+99] Zachary Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Dan Weld. An adaptive query execution engine for data integration. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 1999.

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

[ILM+00] Zachary Ives, Alon Levy, Jayant Madhavan, Rachel Pottinger, Stefan Saroiu, Igor Tatarinov, Ewa Jaslikowska, and Theodora Yeung. Self-organizing data sharing communities with SAGRES: System demonstration. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 2000.

[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.

[LRO96a] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Query answering algorithms for information agents. In *Proceedings of the National Conference on Artificial Intelligence*, 1996.

[LRO96b] 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.

[MGA97] M.Steinbrunn, G.Moerkotte, and A.Kemper. Heuristic and randomized optimization for the join. *VLDB Journal*, 6(3):191–208, 1997.

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

[MLF00] Todd Millstein, Alon Levy, and Marc Friedman. Query containment for data integration systems. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Dallas, Texas, 2000.

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

[NLF99] Felix Naumann, Ulf Leser, and Johann C. Freytag. Quality-driven integration of heterogeneous information systems. In *25th Conference on Very Large Database Systems (VLDB)*, pages 447–458, 1999.

[PDST00] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A chase too far? In *Proc. of ACM SIGMOD Conf. on Management of Data*, 2000.

[PL00] Rachel Pottinger and Alon Levy. A scalable algorithm for answering queries using views. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[PT99] Lucian Popa and Val Tannen. An equational chase for path conjunctive queries, constraints and views. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, 1999.

[PV99] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semi-structured 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.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 130–141, Seattle, WA, 1998.

[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.

[Wie92] Gio Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, 1992.

[YFIV00] Khaled Yagoub, Daniela Florescu, Valerie Issarny, and Patrick Valduriez. Caching strategies for data-intensive web sites. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000.

[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.

[ZCL+00] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex SQL queries using automatic summary tables. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 2000.

[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.