The VLDB Journal 10: 270–294 (2001) / Digital Object Identifier (DOI) 10.1007/s007780100054

Answering queries using views: A survey

A.Y. Halevy

Department of Computer Science and Engineering, University of Washington, Seattle, WA, 98195; E-mail: alon@cs.washington.edu

Edited by M. Carey. Received: 1 August 1999 / Accepted: 23 March 2001
Published online: 6 September 2001 –

Abstract. The problem of answering queries using views is to find efficient methods of answering a query using a set of previously defined 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 over a mediated schema. 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.

Keywords: Materialized views – Data integration – Query optimization – Survey – Date warehousing – Web-site management

1 Introduction

The problem of answering queries using views (also known as 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 tuples in the answer of 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. 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.

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

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, the 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 at the physical level as a set of 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 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 necessary that rewriting of the query using the views be an *equivalent* rewriting in order for the query execution plan to be correct. It is important to note that some of the views included in the query plan may not contribute to the logical correctness of the plan, but only to reducing the plan's cost.

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 expression, 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. A contained query rewriting provides a subset of the answer to the query, but perhaps not the entire answer. In addition, the works on data integration distinguished between the case in which the individual views are complete (i.e., contain all the tuples in their definition) and the case where they may be incomplete (as is common when modeling autonomous data sources). 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, Sect. 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 Sect. 8 discusses several extensions to the algorithms in Sects. 5 and 6 to accommodate queries over object-oriented databases and queries with access-pattern limitations. Finally, Sect. 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, Gup97b, YKL97, GM99c, CG00, CHS01] 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 classifying the different works on the topic into a taxonomy.

We use the following familiar university schema in our examples throughout the paper. We assume that professors, 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 for query optimization. If part of the computation needed to answer a query

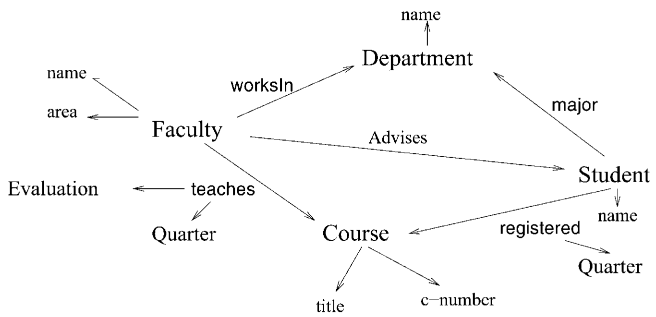

Fig. 1. An entity/relationship diagram for the university domain. Note that quarter is an attribute of the relationships registered and teaches

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 and course titles 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 of 400 and above, and Ph.D-level courses numbers of 500 and above):

```
select  Registered.student, Course.title
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  Registered.student, Course.title, Course.c-number,
        Registered.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  Graduate.student, Graduate.title
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 (the courses that actually account for most of the activity going on in a typical 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 works on answering queries using views were inspired by the goal of maintaining physical data independence in relational and object-oriented databases [YL87,TSI96,Flo96]. One of the principles underlying modern database systems is the separation between the logical view of the data (e.g., as tables with their named attributes) and the physical view of the data (i.e., how it is 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,TIHW01], and in data integration. In fact, the STORED System [DFS99] stores XML documents in a relational database, and uses views to describe the mapping from XML into relations in the database. 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 Fig. 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 index, 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 stores a set of pairs containing students and the departments in which they major, and these pairs are indexed by a B^+-tree on attribute Student.name. The GMAP G2 stores an index from the names

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

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 Department
    where Student registered Course and
    Student major Department.
```

Fig. 2. GMAPs for the university domain

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, Department
where    Student registered Course and
         Student major Department 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 Department. 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, Lev00] 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 source 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 lists Ph.D-level courses being taught at the University of Washington (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 (not just in databases) being offered at UW. Given that only these two sources are available, the data integration system cannot find *all* tuples in the answer to the query. Instead, the system can attempt to find the maximal set of tuples in the answer that are 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 Sect. 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 A taxonomy of the field

As illustrated by the examples, there are several dimensions along which we can classify the treatments of the problem of answering queries using views. In this section we describe a taxonomy for classifying the different works on this problem, and highlight the main differences between the problem treatments. Figure 3 shows the taxonomy and some of the representative works belonging to each of its classes.

The most significant distinction between the different works is whether their goal is data integration or whether it is query optimization and maintenance of physical data independence. The key difference between these two classes of works is the output of the algorithm for answering queries using views. In the former case, given a query Q, and a set of views $\mathcal{V}$, the goal of the algorithm is to produce an expression Q' that references the views and is either equivalent to or contained in Q. In the latter case, the algorithm must go further and produce a (hopefully optimal) query execution plan for answering Q using the views (and possibly the database relations). Here the rewriting must be an equivalent to Q in order to ensure the correctness of the plan.

The similarity between these two bodies of work is that they are concerned with the core issue of whether a rewriting of a query is equivalent to or contained in the query. However, while logical correctness suffices for the data integration context, it does not in the query optimization context where we also need to find the *cheapest* plan using the views. The complication arises because the optimization algorithms need to consider views that do not contribute to the *logical* correctness of the rewriting, but do reduce the cost of the resulting plan. Hence, while the reasoning underlying the algorithms in the data integration context is mostly logical, in the query optimization case it is both logical and cost-based. On the other hand, an aspect stressed in the data integration context is the importance of dealing with a large number of views, which correspond to data sources. In the context of query optimization it is generally assumed (not always!) that the number of views is roughly comparable to the size of the schema.

The works on query optimization can be classified into System-R style optimizers and transformational optimizers. The initial works incorporated views into System-R style join enumeration, while later works that attempt to deal with a more extended subset of SQL realized that the power of rewriting rules is required in order to incorporate views.

The main line of work on data integration attempted to develop algorithms for answering queries using views that scale up to a large number of views[2]. A second line of work started considering different properties of the data sources. For example, it was shown that if data sources are assumed to be complete (i.e., they include all the tuples that satisfy their definition), then the problem of answering queries using views becomes computationally harder. Intuitively, the reason for the added complexity is that when sources are complete, we can also infer negative information as a result of a query to the source. This led to asking the following more basic question: given a query Q, a set of views $\mathcal{V}$ *and* their extensions, what is the complexity of finding the maximal set of tuples in the answer to Q from $\mathcal{V}$.[3] This work established an interesting connection between the problem of answering queries using views and query answering in conditional tables [IL84]. In these works, a major factor affecting the complexity of the problem is whether the view extensions are assumed to be complete or not (when they are complete, the complexity is higher). Note that in the context of query optimization, the views are always assumed to be complete.

[2] Strictly speaking, the motivation for the work of [YL87] was the maintenance of physical data independence, but their algorithm has more similarities with the data integration algorithms.

[3] Some authors refer to the distinction between the two problems as the *rewriting* problem versus the *query answering* problem.

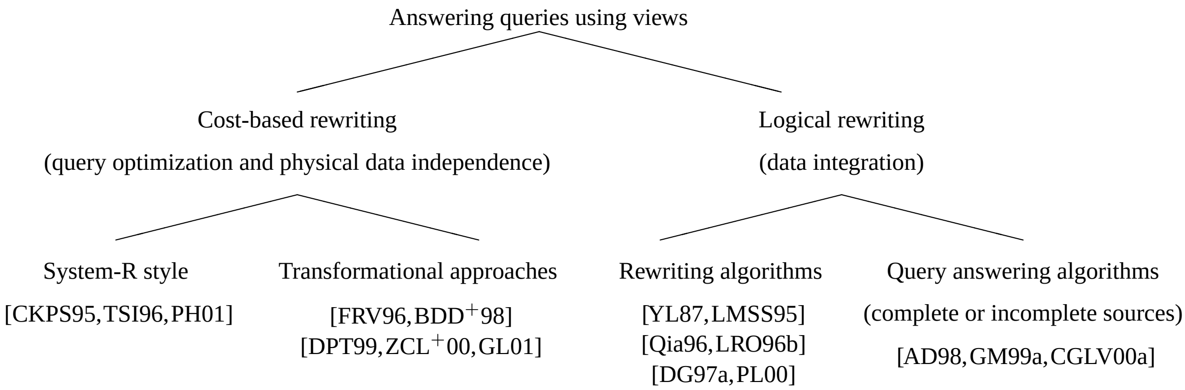

Fig. 3. A taxonomy of work on answering queries using views. The main distinction is between works on query optimization and maintenance of physical data independence and works considering logical rewritings, mostly in the context of data integration. The works on query optimization have considered both System-R style algorithms and transformation-based algorithms. The works on data integration considered algorithms that scale to a large number of views, and the question of finding all the answers to the query, given the view extensions

Table 1. Extensions to query and view languages

Extension	Relevant works
Grouping and aggregation	[GHQ95,SDJL96,CNS99,GRT99,ZCL+00,GT00] (Sect. 5.3)
Bag semantics	[CKPS95,ZCL+00] (Sect. 5.3)
OQL	[FRV96,DPT99] (Sect. 8.1)
Multi-block queries	[ZCL+00] (Sect. 5.2)
Integrity constraints	[DL97,Gry98,ZCL+00,DPT99] (Sect. 7.2)
Access-pattern limitations	[RSU95,KW96,DL97] (Sect. 8.2)
Unions in the views	[AGK99,Dus98] (Sect. 8.3)
Queries over semi-structured data	[CGLV99,PV99] (Sect. 8.3)
Hierarchies in Description Logics	[BLR97,CGL99] (Sect. 8.3)
Languages for querying schema	[Mil98] (Sect. 8.3)

A separate dimension for classifying the different works is the specific language used for expressing views and queries. Much of the early work on the problem focused on select-project-join queries, but, as shown in Table 1, many extensions have been considered as well. The works on query optimization have considered extensions of interest to SQL engines, such as grouping and aggregation and the presence of certain integrity constraints on the database relations. For obvious reasons, these works have also considered the implications of bag semantics on the rewriting problem. The data integration works have considered extensions such as access-pattern limitation to the views, recursive queries, path expressions in the queries, and integrity constraints expressed in description logics.

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 their rewritings, 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 (referred to as the set of certain answers).

The bulk of our discussion will focus on the class of select-project-join queries on relational databases. A view is a named query. It is said to be materialized if its results are stored in the database. A database instance is an assignment of an extension (i.e., a set of tuples) to each of the relations in the database.

We assume the reader is familiar with the 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 Sect. 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 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 database instances 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], recursive queries [Shm93, Sag88, LS93, CV92, CV94], and queries with bag semantics [CV93].

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 $V_1, \ldots, V_m$.[4] In SQL, a query refers only to the views if all the relations mentioned in the from clauses 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 Sect. 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 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. Unlike the case of equivalent rewritings, the maximally-contained rewriting may differ depending on the query language we consider for the rewriting. Hence, the following definition depends on a particular query language:

Definition 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 $\mathcal{V}$ with respect to $\mathcal{L}$ 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 $Q' \sqsubseteq Q_1 \sqsubseteq Q$ and Q_1 is not equivalent to Q'.*

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

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 Sect. 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 with respect to 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 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 a is a certain answer *to the query Q under the closed-world assumption given $v_1, \ldots, v_m$ if $a \in Q(D)$ for all database instances D such that $V_i(D) = v_i$ for every i, $1 \leq i \leq m$.*

The tuple a is a certain answer *to the query Q under the open-world assumption given $v_1, \ldots, v_m$ if $a \in Q(D)$ for all database instances 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 database instance. Hence, given the extensions 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 instances that are consistent with the given extensions of the views. Section 7.3 considers the complexity of finding certain answers.

Example 1. As a very simple example, consider a database schema $R(A, B)$ that includes a single relation with two attributes. Suppose the view V_1 is defined to be the projection of R on A, while V_2 is defined to be the projection of R on B, and suppose that our query Q is to retrieve all of the relation R.

Suppose we are given that the extension of V_1 includes the single tuple (c_1), and that the extension of V_2 includes the single tuple (c_2),

Under the closed-world assumption, we can infer that the tuple (c_1, c_2) *must* be in the relation R, and hence it is a certain answer to Q. However, under the open-world assumption, since V_1 and V_2 are not necessarily complete, the tuple (c_1, c_2) need not be in R. For example, R may contain the tuples (c_1, d) and (e, c_2) for some constants d and e. Hence, (c_1, c_2) is not a certain answer to Q. □

4 When is a view usable for a query?

The common theme across all of the works on answering queries using views is that they all have to deal with the fundamental question of when a view is usable to answer a query. Hence, 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, and in what ways a view may be useful. 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. Hence, we can use V_1 to answer the query by joining it with the relation Advises. Furthermore, V_1 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, V_1 applies a predicate Registered.quarter > "winter97" which is weaker than the predicate Registered.quarter ≥ "winter98" in the query. However, since V_1 selects the attribute Registered.quarter, the stronger predicate can be applied as part of 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 Fig. 4 illustrate how minor modifications to V_1 change their usability in answering the query. The view V_2 is similar to V_1, except that it does not select the attribute Teaches.prof, which is needed for the join with the relation Advises and in the select clause of the query. Hence, to use V_2 in the rewriting, we would need to join V_2 with the Teaches relation again (in addition to a join with Advises). Still, if the join of the relations Registered and Teaches is very selective, then employing V_2 may actually result in a more efficient query execution plan.

The view V_3 does not apply the necessary equi-join predicate between Registered.quarter and Teaches.quarter. Since the attributes Teaches.quarter and Registered.quarter are not selected by V_3, the join predicate cannot be applied in the rewriting, and therefore there is little to gain by using V_3. The view V_4 considers only the professors who have at least one area of research. 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 has at least one area of research, 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 > "winter99"), and is therefore usable for a contained 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. The output of the algorithm is an execution plan for the query. The approaches differ depending on which

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

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

```
create view V5 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 > "winter99".
```

Fig. 4. Examples of unusable views

phase of query optimization was modified to consider materialized views. Section 5.1 describes algorithms based on System R-style optimization, where materialized views are considered during the join enumeration phase [CKPS95,TSI96]. Section 5.2 describes works based on transformational optimizers [ZCL+00,DPT99,PDST00,GL01]. There, the key idea is that replacing a query subexpression by a view is yet another transformation employed by the optimizer. 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 System-R style optimization

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 (Note that if the algorithm is considering only left-deep plans, it will try to combine plans of size $n-1$ with plans of size 1. Otherwise, it will consider combining plans of size k with plans of size $n-k$.) 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 plan 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   Registered.student, Registered.c-number
from     Registered, Course
where    Registered.c-number=Course.c-number and
         Course.title LIKE '%theory%'.
```

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

Suppose the query below asks for all of the students attending Ph.D level classes with 'theory' in their title, and the departments in which the students are majoring.

```
select   Registered.student, Major.dept
from     Registered, Major, Course
where    Registered.student=Major.student and
         Registered.c-number=Course.c-number and
         Course.c-number≥500 and
         Course.title LIKE '%theory%'.
```

We now describe the additional issues that the optimizer needs to consider in the presence of materialized views. Figure 5 shows a side-by-side comparison of the steps of a traditional optimizer vs. one that exploits materialized views. The algorithm described below is a slight modification of the GMAP algorithm [TSI96]. The algorithm described in [CKPS95] uses the same principles, but, as we explain later, with several differences.

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 Sect. 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 and selection predicates in the query.

B. Since the query execution plans involve joins over views, rather than joins 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. In our context, 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, it may not be obvious a priori which join predicate will yield a correct rewriting of the query, since we are joining views rather than database relations directly. Hence, the enumeration algorithm needs to consider several alternative join predicates. Fortunately, in practice, the number of join predicates that need to be considered can be significantly pruned using meta-data about the schema. For example, there is no point in trying to join a string attribute with a numeric one. Furthermore, in some cases we can use knowledge of integrity constraints and the structure of the query to reduce the number of join predicates we consider. 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 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 bullet 3 above, 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 Sect. 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 and 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 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 examine the cost of considering materialized views in query optimization.

Conventional optimizer

Iteration 1

a) find all possible access paths.

b) Compare their cost and keep the least expensive.

c) If the query has one relation, stop.

Iteration 2

For each query join:

a) Consider joining the relevant access paths found in the previous iteration using all possible join methods.

b) Compare the cost of the resulting join plans and keep the least expensive.

c) If the query has only 2 relations, stop.

Iteration 3

. . .

Optimizer using views

Iteration 1

a1) Find all views that are *relevant* to the query.

a2) Distinguish between partial and complete solutions to the query.

b) Compare all pairs of views. If one has neither greater contribution nor a lower cost than the other, prune it.

c) If there are no partial solutions, stop.

Iteration 2

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) If any newly generated solution is either not relevant to the query, or dominated by another, prune it.

c) If there are no partial solutions, stop.

Iteration 3

. . .

Fig. 5. A comparison of a traditional query optimizer with one that exploits materialized views

5.2 Transformational and other approaches to view rewriting

In this section we describe several works that incorporate view rewriting as transformations. The common theme in these works is that replacing some part of a query with a view is considered as another transformation available to the optimizer. This approach is necessary when: (1) the entire optimizer is transformational (e.g, in [GL01]); and (2) in the logical rewriting phase of a System-R style optimizer that is considering more complex SQL queries (as in [ZCL$^+$00]).

In [GL01] the authors describe an algorithm for rewriting queries using views that is implemented in the transformational optimizer of Microsoft SQL Server. In the algorithm, view matching is added as another transformation rule in the optimizer. The transformation rule is invoked on select-project-join-group-by (SPJG) expressions, and it attempts to replace the SPJG expression by a single view. The authors describe in detail the conditions under which a sub-query is replaced by a view. The key novelty in this work is the *filter-tree*, a clever index structure that makes it possible to efficiently filter the set of views that are relevant to a particular SPJG expression. The index is composed of several sub-indexes, each of which is built on a particular property of the views (e.g., the set of tables in the view, the set of output columns, grouping columns). The sub-indexes are combined in a hierarchical fashion into the filter tree, where each level in the tree further partitions the views according to another property. The authors describe a set of experiments that shows that their algorithm adds relatively little to the optimization time, even in the presence of 1,000 views.

In [ZCL$^+$00] the authors describe how view rewriting is incorporated into the query rewrite phase of the IBM DB2 UDB optimizer. Their algorithm operates on the Query Graph Model (QGM) representation of a query [HFLP89], which decomposes the query into multiple QGM *boxes*, each corresponding to a select-project-join block. 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 view and some box in the QGM of the query. 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 [DPT99] the authors use a transformational approach to uniformly incorporate the use materialized views, specialized indexes and semantic integrity constraints. All of these are represented as constraints. Their procedure involves two phases, each involving a different set of transformations. 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 *back-chase*, 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 back-chase phase.

In [BDD$^+$98] the authors describe a limited use of transformation rules to incorporate view rewriting algorithm into 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 Sect. 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.

Finally, in [ALU01] the authors consider using views for query optimization from a different angle. They consider the problem of finding the rewriting of the query with minimal cost under three specific cost models: (1) minimizing the number of views in the rewriting (hence the number of joins); (2) reducing the size of the intermediate relations computed during the rewriting; and (3) reducing the size of intermediate relations while also dropping irrelevant attributes as the computation proceeds. The techniques underlying the CoreCover algorithm described in [ALU01] are closer in spirit to those used in the MiniCon Algorithm [PL00] described in Sect. 6.4.

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 (and hence, there are likely to be several offerings of the same course during an academic year). 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 ≥ 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 ≥ 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 ≥ 500
groupBy   year.
```

There are a couple of points to note about the rewriting. First, even though the view performed an aggregation on the attribute evaluation, 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. One approach considered 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, roll-up, 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.[5]

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 most part, 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 Sect. 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. The maximally-contained rewriting will often involve a union of several queries over the sources. 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,PH01]. 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 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, c-number, quarter),
                   Teaches(prof, c-number, 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, c-number, prof, 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 (extensional database) predicates that refer to the database relations from the IDB (intensional database) predicates that refer to intermediate computed 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

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

see in Sect. 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, starting with empty extensions for the IDB relations. 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 we explain in Sect. 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, to decide whether the view V should be in the bucket of a subgoal g, we consider each of the subgoals g_1 in V and do the following:

A. Check whether there is a unifier θ for g and g_1, i.e., θ is a variable mapping such that $\theta(g) = \theta(g_1)$. If there is no unifier, we proceed to the next subgoal.
B. Given the unifier θ, we check whether the view and the query would be compatible after the unifier is applied. Hence, we apply $\theta_{h(V)}$ to the query and to the view, where $\theta_{h(V)}$ is the same as θ but its domain does not include the existential variables in V (since only the head variables of V are part of a rewriting). Then we check two conditions: (1) that 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; and (2) that θ treats the head variables occurring in g correctly, i.e., if X is a head variable that appears in position i of the subgoal g, then the variable appearing in position i of g_1 must be a head variable of V.

If the above conditions are satisfied, then we insert the atom $\theta(head(V))$ into the bucket of g. 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.

Table 2. Contents of the buckets. The primed variables are those that are not in the domain of the unifying mapping

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')

In the second step, the bucket algorithm finds a set of *conjunctive query rewritings*, each of them being a conjunctive query that includes one conjunct from every bucket. Each of these conjunctive query rewritings represents one way of obtaining part of the answer to Q from the views. The result of the bucket algorithm is defined to be the union of the conjunctive query rewritings (since each of the rewritings may contribute different tuples). Given a conjunction, constructed from a single element from every bucket, it is a conjunctive query rewriting if either: (1) the conjunction is contained in the query Q; or (2) it is possible to add atoms of comparison predicates such that the resulting conjunction is contained in Q.

Example 2. Consider 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), quarter ≤ 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 2. 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 condition (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 }.

The view V3 is not included in the bucket of Registered(S,C,Q) because after applying the unification mapping, the predicates Q ≥ Aut95 and Q ≤ Aut94 are mutually inconsistent. The view V4 is not included in the bucket of Registered(S,C,Q) because the variable student is not in the head of V4, while S is in the head of the query.

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 → c-number, T → 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 when they are considered 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).

Hence, the result of the bucket algorithm is the union of two conjunctive queries, one obtains answers by joining V1 and V2, and the other by joining V2 and V4. The reader should note that in this example, as often 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 definition 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,[6] 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 $\mathcal{V}$.
- 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 3. Suppose a query asks for the departments in which the students of the 444 course are majoring,

[6] For conjunctive queries with no comparison predicates, query containment is in NP because we only need to guess a containment mapping. Here, however, we need to guess a containment mapping for every possible ordering on the variables in containing query.

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 exploit 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 Sect. 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 (see [PL00] for an experimental analysis).

6.4 The MiniCon algorithm

The MiniCon algorithm [PL00, PH01] 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 *subgoals* 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 useful 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 variable 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 be 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 many 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 demonstrates 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 a description of these query shapes). The work in [PH01] also explains how to exploit the key ideas of the the MiniCon algroithm to the context of query optimization with materialized views, where the cost of the query plan if the primary concern.

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 that 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 Sect. 7.1.

Completeness of a rewriting algorithm is characterized with respect to 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 with respect to 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 Sect. 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 Sect. 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 an equivalent rewriting of a query using a set of views is in NP, because it suffices to guess a rewriting and check its correctness.[7]

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 Sect. 8.2). In [CR97] the authors exploit the close connection between the containment and rewriting problems, and show several polynomial-time cases of the rewriting problems, corresponding to analogous cases for the problem of query containment.

7.2 The need for recursive rewritings

As noted earlier, in cases where we cannot find an equivalent rewriting of the query using a set of views, we consider the problem of finding maximally-contained rewritings. Our hope is that when we apply the maximally-contained rewriting to the extensions of the views, we will obtain the set of *all* certain answers to the query (Definition 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: when we exploit the presence of functional dependencies on the database relations or when there are access-pattern limitations on the extensions of the

[7] 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.

views [DL97] (see Sect. 8.2 for a more detailed discussion), when views contain unions [Afr00] (though even recursion 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 4. 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 attribute Airline of the schedule relation):

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

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

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

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

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

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

In [DL97,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

A different perspective on the problem of answering queries using views is the following. Given a set of materialized views and the corresponding view definitions, we obtain some *incomplete* information about the contents of the database. More specifically, the views define a set of *possible* underlying databases $\mathcal{D}$. Given a query Q over the database and a tuple t, there are a few possibilities: (1) t would be an answer to Q for every database in $\mathcal{D}$; (2) t is an answer to Q for some database in $\mathcal{D}$; or (3) t is not an answer to Q for any database in $\mathcal{D}$. The notion of certain answers, (see Definition 4) formalizes case (1).

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.

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 with respect to 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. 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 in the data integration context). 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 (in the size of the view extensions!) 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 5. 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], providing a formalization to the intuition starting out this section.

The work in [GM99a] also considers the case where the views may either be incomplete, complete, or contain tuples that don't satisfy the view definition (referred to as *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. The work in [MM01] considers the query answering problem in cases where we may have bounds on the soundness and/or completeness of the views.

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 a 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. Here we have the class Universities, with subclass France.Universities and the class City. The first two classes have the attributes students, PhDstudents (a sub-attribute of students), professors and adjuncts.

Example 6. 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. Below we use the notation of OQL. Note that the select clause of OQL defines the record structure of the result. In addition, note the use of path expressions – for example, y in x.students means that the variable y will be bound to each of the students of the object to which x will be bound.

```
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 Sect. 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 and schema language: In [AGK99, Dus98] the authors consider the rewriting problem when the views may contain unions. The consideration 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 problem 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. In [MRP99] the authors show that when the schema contains a single universal relation, answering queries using views and several related operations can be done more efficiently.

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+99,CRF00].

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) is 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 that 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. As a simple example, consider a data source that stores a set of documents, and can answer queries of the form:

q(doc) :- document(doc), contains(doc, w1), . . . ,
 contains(doc,wn)

where we can have any number of occurrences of the contains subgoal, each with a different word.

To answer queries using such sources, one need not only choose which sources to query, but we must also choose 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 makes it possible 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 inter-schema relationships [CL93]. For that reason, description logics have been used in several data integration systems [AKS96,LRO96a]. 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,CRF00]).

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 configuration, 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.

In this survey we considered the problem of using materialized views when they are available. I believe that the next challenge is *selecting* which views to materialize in the first place. The problem of view selection also has a surprising number of potential applications, such as query optimization, data warehousing, web-site design, content distribution networks, peer-to-peer computing and ubiquitous computing. Even though there has been work on this problem (e.g., [CHS01,ACN00,Gup97a,CG00,GM99c, TS97,YKL97,BPT97,GHRU97,HRU96,GHI+01]), the research is still in its infancy. The wealth of techniques developed for answering queries using views will be very useful in this realm.

Acknowledgements. I would like to thank Phil Bernstein, Mike Carey, Anhai Doan, Todd Millstein, Rachel Pottinger, Arnie Rosenthal, Igor Tatarinov and the anonymous reviewers for valuable comments on earlier drafts of this paper. I would like to acknowledge the support of a Sloan Fellowship, NSF Grants #IIS-9978567 and #IIS-9985114, and gifts from Microsoft Corporation and from NEC, Japan.

References

[Abi97] Abiteboul S. Querying semi-structured data. In: Proc. of ICDT. pp 1–18, Delphi, Greece, 1997

[ABS99] Abiteboul S., Buneman P., Suciu D. Data on the Web. Morgan Kaufmann, San Francisco, 1999

[ACN00] Agrawal S., Chaudhuri S., Narasayya V. Automated selection of materialized views and indexes in Microsoft SQL Server. In: Proc. of VLDB. pp 496–505, Cairo, Egypt, 2000

[ACPS96] Adali S., Candan K., Papakonstantinou Y., Subrahmanian V.S. Query caching and optimization in distributed mediator systems. In: Proc. of SIGMOD. pp 137–148, Montreal, Canada, 1996

[AD98] Abiteboul S., Duschka O. Complexity of answering queries using materialized views. In: Proc. of PODS. pp 254–263, Seattle, Wash., USA, 1998

[Afr00] Afrati F. Personal communication, 2000

[AGK99] Afrati F., Gergatsoulis M., Kavalieros T. Answering queries using materialized views with disjunctions. In: Proc. of ICDT. pp 435–452, 1999

[AHV95] Abiteboul S., Hull R., Vianu V. Foundations of databases. Addison Wesley, Reading, Mass., USA, 1995

[AKS96] Arens Y., Knoblock C.A., Shen W.M. Query reformulation for dynamic information integration. Int. J. Intell. Coop. Inf. Syst. (6)2/3:99–130, 1996

[ALU01] Afrati F., Li C., Ullman J. Generating efficient plans for queries using views. In: Proc. of SIGMOD. pp 319–330, 2001

[AQM+97] Abiteboul S., Quass D., McHugh J., Widom J., Wiener J. The Lorel query language for semistructured data. Int. J. Digital Libr. 1(1):68–88, 1997

[BDD+98] Bello R., Dias K., Downing A., Feenan J., Finnerty J., Norcott W., Sun H., Witkowski A., Ziauddin M. Materialized views in Oracle. In: Proc. of VLDB. pp 659–664, 1998

[BDHS96] Buneman P., Davidson S., Hillebrand G., Suciu D. A query language and optimization techniques for unstructured data. In: Proc. of SIGMOD. pp 505–516, Montreal, Canada, 1996

[BLR97] Beeri C., Levy A.Y., Rousset M.C. Rewriting queries using views in description logics. In: Proc. of PODS. pp 99–108, Tucson, Ariz., USA, 1997

[Bor95] Borgida A. Description logics in data management. IEEE Trans. Knowl. Data Eng. 7(5):671–682, 1995

[BPT97] Baralis E., Paraboschi S., Teniente E. Materialized views selection in a multidimensional database. In: Proc. of VLDB. pp 156–165, 1997

[Bun97] Buneman P. Semistructured data. In: Proc. of PODS. pp 117–121, Tucson, Ariz., USA, 1997

[CG00] Chirkova R., Genesereth M. Linearly bounded reformulations of conjunctive databases. In: Proc. of DOOD. pp 987–1001, 2000

[CGL99] Calvanese D., De Giacomo G., Lenzerini M. Answering queries using views in description logics. In: Working notes of the KRDB Workshop. 1999

[CGLV99] Calvanese D., De Giacomo G., Lenzerini M., Vardi M. Rewriting of regular expressions and regular path queries. In: Proc. of PODS. pp 194–204, 1999

[CGLV00a] Calvanese D., De Giacomo G., Lenzerini M., Vardi M. Answering regular path queries using views. In: Proc. of ICDE. pp 389–398, 2000

[CGLV00b] Calvanese D., De Giacomo G., Lenzerini M., Vardi M. View-based query processing for regular path queries with inverse. In: Proc. of PODS. pp 58–66, 2000

[CHS01] Chirkova R., Halevy A., Suciu D. A formal perspective on the view selection problem. In: Proc. of VLDB. 2001

[CKPS95] Chaudhuri S., Krishnamurthy R., Potamianos S., Shim K. Optimizing queries with materialized views. In: Proc. of ICDE. pp 190–200, Taipei, Taiwan, 1995

[CL93] Catarci T., Lenzerini M. Representing and using interschema knowledge in cooperative information systems. J. Intell. Coop. Inf. Syst.. pp 55–62, 1993

[CM77] Chandra A.K., Merlin P.M. Optimal implementation of conjunctive queries in relational databases. In: Proc. Ninth Annual ACM Symposium on Theory of Computing. pp 77–90, 1977

[CN98a] Chaudhuri S., Narasayya V.R. Autoadmin 'what-if' index analysis utility. In: Proc. of SIGMOD. pp 367–378, 1998

[CN98b] Chaudhuri S., Narasayya V.R. Microsoft index tuning wizard forSQL Server 7.0. In: Proc. of SIGMOD. pp 553–554, 1998

[CNS99] Cohen V.R., Nutt W., Serebrenik A. Rewriting aggregate queries using views. In: Proc. of PODS. pp 155–166, 1999

[CR94] Chen C., Roussopoulos N. Implementation and performance evaluation of the ADMS query optimizer. In: Proc. of EDBT. pp 323–336, March 1994

[CR97] Chekuri C., Rajaraman A. Conjunctive query containment revisited. In: Proc. of ICDT. pp 56–70, Delphi, Greece, 1997

[CRF00] Chamberlin D.D., Robie J., Florescu D. Quilt: an XML query language for heterogeneous data sources. In: WebDB (Informal Proceedings) 2000. pp 53–62, 2000

[CV92] Chaudhuri S., Vardi M. On the equivalence of recursive and nonrecursive datalog programs. In: Proc. of PODS. pp 55–66, San Diego, Calif., USA, 1992

[CV93] Chaudhuri S., Vardi M. Optimizing real conjunctive queries. In: Proc. of PODS. pp 59–70, Washington D.C., USA, 1993

[CV94] Chaudhuri S., Vardi M. On the complexity of equivalence between recursive and nonrecursive datalog programs. In: Proc. of PODS. pp 55–66, Minneapolis, Minn., USA, 1994

[DFF+99] Deutsch A., Fernandez M., Florescu D., Levy A., Suciu D. A query language for XML. In: Proc. World-Wide Web 8 Conference. pp 1155–1169, 1999

[DFJ+96] Dar S., Franklin M.J., Jonsson B., Srivastava D., Tan M. Semantic data caching and replacement. In: Proc. of VLDB. pp 330–341, 1996

[DFS99] Deutsch A., Fernandez M., Suciu D. Storing semistructured data with STORED. In: Proc. of SIGMOD. pp 431–442, 1999

[DG97a] Duschka O.M., Genesereth M.R. Answering recursive queries using views. In: Proc. of PODS. pp 109–116, Tucson, Ariz., USA, 1997

[DG97b] Duschka O.M., Genesereth M.R. Query planning in infomaster. In: Proc. ACM Symposium on Applied Computing. pp 109–111, San Jose, Calif., USA, 1997

[DGL00] Duschka O., Genesereth M., Levy A. Recursive query plans for data integration. In: Special issue on Logic Based Heterogeneous Information Systems. J. Logic Program. 43(1):49–73, 2000

[DL97] Duschka O.M. Levy A.Y. Recursive plans for information gathering. In: Proc. 15th International Joint Conference on Artificial Intelligence. pp 778–784, 1997

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

[DPT99] Deutsch A., Popa L., Tannen V. Physical data independence, constraints and optimization with universal plans. In: Proc. of VLDB. pp 459–470, 1999

[Dus97] Duschka O. Query optimization using local completeness. In: Proc. AAAI 14th National Conference on Artificial Intelligence. pp 249–255, 1997

[Dus98] Duschka O.M. Query planning and optimization in information integration. PhD thesis, Stanford University, Stanford, Calif., USA, 1998

[EGW97] Etzioni O., Golden K., Weld D.S. Sound and efficient closed-world reasoning for planning. Artif. Intell. 89(1-2):113–148, 1997

[FFLS97] Fernandez M., Florescu D., Levy A., Suciu D. A query language for a web-site management system. SIGMOD Record 26(3):4–11, 1997

[FK99] Florescu D., Kossmann D. Storing and querying XML data using an rdbms. IEEE Data Eng. Bull. 22(3):27–34, 1999

[FKL97] Florescu D., Koller D., Levy A. Using probabilistic information in data integration. In: Proc. of VLDB. pp 216–225, Athens, Greece, 1997

[FLM98] Florescu D., Levy A., Mendelzon A. Database techniques for the world-wide web: a survey. SIGMOD Record 27(3):59–74, 1998

[Flo96] Florescu D.D. Search spaces for object-oriented query optimization. PhD thesis, Univerisity of Paris VI. France, 1996

[FLSY99] Florescu D., Levy A., Suciu D., Yagoub K. Optimization of run-time management of data intensive web sites. In: Proc. of VLDB. pp 627–638, 1999

[FRV96] Florescu D., Raschid L., Valduriez P. Answering queries using OQL view expressions. In: Workshop on Materialized Views, in cooperation with ACM SIGMOD. Montreal, Canada, 1996

[FW97] Friedman M., Weld D. Efficient execution of information gathering plans. In: Proc. International Joint Conference on Artificial Intelligence, Nagoya, Japan. pp 785–791, 1997

[GBLP98] Gray J., Bosworth A., Layman A., Pirahesh H. Data cube: a relational aggregation operator generalizing group-by, cross-tab and sub-totals. In: Proc. of ICDE. pp 152–159, 1998

[GHI+01] Gribble S., Halevy A., Ives Z., Rodrig M., Suciu D. What can databases do for peer-to-peer? In: ACM SIGMOD WebDB Workshop 2001. 2001

[GHQ95] Gupta A., Harinarayan V., Quass D. Aggregate-query processing in data warehousing environments. In: Proc. of VLDB. pp 358–369, 1995

[GHRU97] Gupta H., Harinarayan V., Rajaraman A., Ullman J.D. Index selection for OLAP. In: Proc. of ICDE. pp 208–219, 1997

[GL01] Goldstein J., Larson P.A. Optimizing queries using materialized views: a practical, scalable solution. In: Proc. of SIGMOD. pp 331–342, 2001

[GM99a] Grahne G., Mendelzon A.O. Tableau techniques for querying information sources through global schemas. In: Proc. of ICDT. pp 332–347, 1999

[GM99b] Gupta A., Mumick I., (eds.) Materialized views: techniques, implementations and applications. MIT, Cambridge, Mass., USA, 1999

[GM99c] Gupta H., Mumick I.S. Selection of views to materialize under a maintenance cost constraint. In: Proc. of ICDT. pp 453–470, 1999

[GMPQ+97] Garcia-Molina H., Papakonstantinou Y., Quass D., Rajaraman A., Sagiv Y., Ullman J., Widom J. The TSIMMIS project: Integration of heterogeneous information sources. J. Intell. Inf. Syst. 8(2):117–132, 1997

[GRT99] Grumbach S., Rafanelli M., Tininini L. Querying aggregate data. In: Proc. of PODS. pp 174–184, 1999

[Gry98] Gryz J. Query folding with inclusion dependencies. In: Proc. of ICDE. pp 126–133, Orlando, Fla., USA, 1998

[GT00] Grumbach S., Tininini L. On the content of materialzed aggregate views. In: Proc. of PODS. 2000

[Gup97a] Gupta H. Selection of views to materialize in a data warehouse. In: Proc. of ICDT. pp 98–112, 1997

[Gup97b] Gupta H. Selection of views to materialize in a data warehouse. In: Proc. of ICDT. pp 98–112, Delphi, Greece, 1997

[HFLP89] Haas L., Freytag J., Lohman G., Pirahesh H. Extensible query processing in Starburst. In: Proc. of SIGMOD. pp 377–388, 1989

[HRU96] Harinarayan V., Rajaraman A., Ullman J.D. Implementing data cubes efficiently. In: Proc. of SIGMOD. pp 205–216, 1996

[IFF+99] Ives Z., Florescu D., Friedman M., Levy A., Weld D. An adaptive query execution engine for data integration. In: Proc. of SIGMOD. pp 299–310, 1999

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

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

[Klu88] Klug A. On conjunctive queries containing inequalities. J. ACM. pp 35(1):146–160, 1988

[KMT98] Kolaitis P., Martin D., Thakur M. On the complexity of the containment problem for conjunctive queries with built-in predicates. In: Proc. of PODS. pp 197–204, Seattle, Wash., USA, 1998

[KW96] Kwok C.T., Weld D.S. Planning to gather information. In: Proc. AAAI 13th National Conference on Artificial Intelligence. pp 32–39, 1996

[Lev96] Levy A.Y. Obtaining complete answers from incomplete databases. In: Proc. of VLDB. pp 402–412, Bombay, India, 1996

[Lev00] Levy A.Y. Logic-based techniques in data integration. In: MinkerJ (ed.) Logic-based artificial intelligence. Kluwer Academic, Dordrecht, 2000, pp 575–595

[LFS97] Levy A.Y., Fikes R.E., Sagiv S. Speeding up inferences using relevance reasoning: a formalism and algorithms. Artif. Intell. 97(1–2), 1997

[LKG99] Lambrecht E., Kambhampati S., Gnanaprakasam S. Optimizing recursive information gathering plans. In: Proc. 16th International Joint Conference on Artificial Intelligence. pp 1204–1211, 1999

[LMSS95] Levy A.Y., Mendelzon A.O., Sagiv Y., Srivastava D. Answering queries using views. In: Proc. of PODS. pp 95–104, San Jose, Calif, USA, 1995

[LRO96a] Levy A.Y., Rajaraman A., Ordille J.J. Query answering algorithms for information agents. In: Proc. National Conference on Artificial Intelligence. pp 40–47, 1996

[LRO96b] Levy A.Y., Rajaraman A., Ordille J.J. Querying heterogeneous information sources using source descriptions. In: Proc. of VLDB. pp 251–262, Bombay, India, 1996

[LRU96] Levy A.Y., Rajaraman A., Ullman J.D. Answering queries using limited external processors. In: Proc. of PODS. pp 227–237, Montreal, Canada, 1996

[LS93] Levy A.Y., Sagiv Y. Queries independent of updates. In: Proc. of VLDB. pp 171–181, Dublin, Ireland, 1993

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

[Mil98] Miller R.J. Using schematically heterogeneous structures. In: Proc. of SIGMOD. pp 189–200, Seattle, Wash., USA, 1998

[MLF00] Millstein T., Levy A., Friedman M. Query containment for data integration systems. In: Proc. of PODS. pp 67–75, Dallas, Tex., USA, 2000

[MM01] Mendelzon A., Mihaila G. Querying partially sound and complete data sources. In: Proc. of PODS. pp 162–170, 2001

[MMS79] Maier D., Mendelzon A., Sagiv Y. Testing implications of data dependencies. ACM Trans. Database Syst. 4(4):455–469, 1979

[MRP99] Minock M., Rusinkiewicz M., Perry B. The identification of missing information resources through the query difference operator. In: Proc. 4th IFCIS International Conference on Cooperative Information Systems (CoopIS 99). September 1999

[NLF99] Naumann F., Leser U., Freytag J.C. Quality-driven integration of heterogeneous information systems. In: 25th Conference on Very Large Database Systems (VLDB). pp 447–458, 1999

[PDST00] Popa L., Deutsch A., Sahuguet A., Tannen V. A chase too far? In: Proc. of SIGMOD. pp 273–284, 2000

[PH01] Pottinger R., Halevy A. Minicon: a scalable algorithm for answering queries using views. VLDB J. 10(2): 182–198, 2001

[PL00] Pottinger R., Levy A. A scalable algorithm for answering queries using views. In: Proc. of VLDB. pp 484–495, Cairo, Egypt, 2000

[PT99] Popa L., Tannen V. An equational chase for path conjunctive queries, constraints and views. In: Proc. of ICDT. 1999

[PV99] Papakonstantinou Y., Vassalos V. Query rewriting for semi-structured data. In: Proc. of SIGMOD. pp 455–466, 1999

[Qia96] Qian X. Query folding. In: Proc. of ICDE. pp 48–55, New Orleans, La., USA, 1996

[RSU95] Rajaraman A., Sagiv Y., Ullman J.D. Answering queries using templates with binding patterns. In: Proc. of PODS. pp 105–112, San Jose, Calif., USA, 1995

[SAC+79] Selinger P., Astrahan M., Chamberlin D., Lorie R., Price T. Access path selection in relational database systems. In: Proc. of SIGMOD. pp 23–34, Boston, Mass., USA, 1979

[Sag88] Sagiv Y. Optimizing datalog programs. In: Minker J (ed) Foundations of deductive databases and logic programming. pp 659–698. Morgan Kaufmann, Los Altos, Calif., USA, 1988

[SDJL96] Srivastava D., Dar S., Jagadish H.V., Levy A.Y. Answering SQL queries using materialized views. In: Proc. of VLDB. Bombay, India, 1996

[SGT+99] Shanmugasundaram J., Gang H., Tufte K., Zhang C., DeWitt D.J., Naughton J. Relational databases for querying XML documents: limitations and opportunities. In: Proc. of VLDB. pp 302–314, 1999

[Shm93] Shmueli O. Equivalence of datalog queries is undecidable. J. Logic Program. 15:231–241, 1993

[SR92] Srivastava D., Ramakrishnan R. Pushing constraint selections. In: Proc. of PODS. pp 301–315, San Diego, Calif., USA, 1992

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

[TIHW01] Tatarinov I., Ives Z., Halevy A., Weld D. Updating XML. In: Proc. of SIGMOD. pp 413–424, 2001

[TS97] Theodoratos D., Sellis T. Data warehouse configuration. In: Proc. of VLDB. pp 126–135, Athens, Greece, 1997

[TSI94] Tsatalos O.G., Solomon M.H., Ioannidis Y.E. The GMAP: a versatile tool for physical data independence. In: Proc. of VLDB. pp 367–378, Santiago, Chile, 1994

[TSI96] Tsatalos O.G., Solomon M.H., Ioannidis Y.E. The GMAP: a versatile tool for physical data independence. VLDB J. 5(2):101–118, 1996

[UFA98] Urhan T., Franklin M.J., Amsaleg L. Cost-based query scrambling for initial delays. In: Proc. of SIGMOD. pp 130–141, Seattle, Wash., USA, 1998

[Ull89] Ullman J.D. Principles of database and knowledge-base systems, vols. I, II. Computer Science, Rockville, Md., USA, 1989

[Ull97] Ullman J.D. Information integration using logical views. In: Proc. of ICDT. pp 19–40, Delphi, Greece, 1997

[Val87] Valduriez P. Join indices. ACM Trans. Database Syst. 12(2):218–246, 1987

[VP97] Vassalos V., Papakonstantinou Y. Describing and using query capabilities of heterogeneous sources. In: Proc. of VLDB. pp 256–265, Athens, Greece, 1997

[Wie92] Wiederhold G. Mediators in the architecture of future information systems. IEEE Comput. pp 38–49, 1992

[YFIV00] Yagoub K., Florescu D., Issarny V., Valduriez P. Caching strategies for data-intensive web sites. In: Proc. of VLDB. pp 188–199, Cairo, Egypt, 2000

[YKL97] Yang J., Karlapalem K., Li Q. Algorithms for materialized view design in data warehousing environment. In: Proc. of VLDB. pp 136–145, Athens, Greece, 1997

[YL87] Yang H.Z., Larson P.A. Query transformation for PSJ-queries. In: Proc. of VLDB. pp 245–254, Brighton, UK, 1987

[ZCL+00] Zaharioudakis M., Cochrane R., Lapis G., Pirahesh H., Urata M. Answering complex SQL queries using automatic summary tables. In: Proc. of SIGMOD. pp 105–116, 2000

[ZO93] Zhang X., Ozsoyoglu M.Z. On efficient reasoning with implication constraints. In: Proc. of DOOD. pp 236–252, 1993