MiniCon: A Scalable Algorithm for Answering Queries Using Views

Contact Author: Rachel Pottinger
Department of Computer Science and Engineering
University of Washington
Box 352350 Seattle, WA, 98195
phone: (206)616-3997
fax: (206)543-2969
email: rap@cs.washington.edu

Alon Halevy
Department of Computer Science and Engineering
University of Washington
Seattle, WA, 98195
email: alon@cs.washington.edu

The VLDB Journal manuscript No.
(will be inserted by the editor)

MiniCon: A Scalable Algorithm for Answering Queries Using Views

Rachel Pottinger, Alon Halevy*

The University of Washington, Seattle, WA 98195
e-mail: {rap, alon}@cs.washington.edu

Received: date / Revised version: date

Abstract The problem of answering queries using views is to find efficient methods of answering a query using a set of previously materialized views over the database, rather than accessing the database relations. The problem has received significant attention because of its relevance to a wide variety of data management problems, such as data integration, query optimization, and the maintenance of physical data independence. To date, the performance of proposed algorithms has received very little attention, and in particular, their scale up in the presence of a large number of views is unknown.

We first analyze two previous algorithms, the bucket algorithm and the inverse-rules algorithm, and show their deficiencies. We then describe the MiniCon algorithm, a novel algorithm for finding the maximally-contained rewriting of a conjunctive query using a set of conjunctive views. We present the first experimental study of algorithms for answering queries using views. The study shows that the MiniCon algorithm scales up well and significantly outperforms the previous algorithms. We describe an extension of the MiniCon algorithm to handle comparison predicates, and show its performance experimentally. Finally, we describe how the MiniCon algorithm can be extended to the context of query optimization.

Keywords: materialized views, data integration, query optimization, web and databases.

* Thanks to Daniela Florescu, Marc Friedman, Gösta Grahne, Zack Ives, Ioana Manolescu, Dan Weld, and Steve Wolfman for their comments on earlier drafts of this paper. This research was funded by a Sloan Fellowship, NSF Grants #IIS-9978567 and #IIS-9985114, a NSF Graduate Research Fellowship, a Lucent Technologies GRPW Grant, and gifts from Microsoft Research, NTT and NEC.

1 Introduction

The problem of answering queries using views (a.k.a. rewriting queries using views) has recently received significant attention because of its relevance to a wide variety of data management problems [Hal01]: query optimization [CKPS95,LMSS95,ZCL+00], maintenance of physical data independence [YL87,TSI96,PDST00], data integration [LRO96,DG97b,KW96,LKG99], and data warehouse and web-site design [HRU96,TS97]. 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 $V_1, \ldots, V_n$, and if so, how?

There are two main contexts in which the problem of answering queries using views has been considered. In the first context, where the goal is query optimization or maintenance of physical data independence [YL87, TSI96,CKPS95], we search for an expression that uses the views and is *equivalent* to the original query. Here it is usually assumed that the number of views is on the same order as the size of the schema. The second context is that of data integration, where views describe a set of autonomous heterogenous data sources. A user poses a query in terms of a mediated schema, and the data integration system needs to reformulate the query to refer to the data sources. In a subsequent phase, the queries over the sources are optimized and executed. The reformulation problem can be solved by algorithms for answering queries using views, though in this context, we usually cannot find a rewriting that is equivalent to the user query because of the data sources' limited coverage. Instead, we search for a *maximally-contained rewriting*, which provides the best answer possible, given the available sources. When the query and views are conjunctive (i.e., select-project-join) without comparison predicates, the maximally-contained rewriting is a union of conjunctive queries over the views. In some data integration applications, the number of data sources may be quite large

– for example, data sources may be a set of web sites, a large set of suppliers and consumers in an electronic marketplace, or a set of peers containing fragments of a larger data set in a peer-to-peer environment. Hence, the challenge in this context is to develop an algorithm that scales up in the number of views.

We consider the problem of answering conjunctive queries using a set of conjunctive views in the presence of a large number of views. In general, this problem is NP-Complete because it involves searching through a possibly exponential number of rewritings [LMSS95]. Previous work has mainly considered two algorithms for this purpose. The bucket algorithm, developed as part of the Information Manifold System [LRO96], controls its search by first considering each subgoal in the query in isolation, and creating a bucket that contains only the views that are relevant to that subgoal. The algorithm then creates rewritings by combining one view from every bucket. As we show, the combination step has several deficiencies, and does not scale up well. The inverse-rules algorithm, developed in [Qia96,DG97a], is primarily used in the InfoMaster System [DG97a]. The inverse-rules algorithm considers rewritings for each database relation independent of any particular query. Given a user query, these rewritings are combined appropriately. We show that the rewritings produced by the inverse-rules algorithm need to be further processed in order to be appropriate for query evaluation. Unfortunately, in this additional processing step the algorithm must duplicate much of the work done in the second phase of the bucket algorithm.

Based on the insights into the previous algorithms, we introduce the MiniCon algorithm, which addresses their limitations and scales up to a large number of views. The key idea underlying the MiniCon algorithm is a change of perspective: instead of building rewritings by combining rewritings for each query subgoal or 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. Hence, as we show experimentally, the MiniCon algorithm scales up much better. The specific contributions of the paper are the following:

- We describe the MiniCon algorithm and its properties.
- We present a detailed experimental evaluation and analysis of algorithms for answering queries using views. The experimental results show (1) the MiniCon algorithm significantly outperforms the bucket and inverse-rules algorithms, and (2) the MiniCon algorithm scales up to hundreds of views, thereby showing for the first time that answering queries using views can be efficient on large scale problems. We believe that our experimental evaluation in itself is a significant contribution that fills a void in previous work on this topic.
- We describe an extension of the MiniCon algorithm to handle comparison predicates and experimental results on its performance.
- We describe an extension of the MiniCon algorithm to the context of cost-based query optimization, where the goal is to find the single cheapest plan for the query using the views. In doing so we distinguish the role of two sets of views: those that are needed for the logical correctness of the plan, and those that are only needed to reduce the cost of the plan. We show that different techniques are needed in order to identify each of these sets.

This paper focuses on the problem of answering queries using views for select-project-join queries under set semantics. While such queries are quite common in data integration applications, many applications will need to deal with queries involving grouping and aggregation, semi-structured data, nested structures and integrity constraints. Indeed, the problem of answering queries using views has been considered in these contexts as well [GHQ95, SDJL96, CNS99, GRT99, PV99, CGLV99, DL97, Gry98]. In contrast to these works, our focus is on obtaining a scalable algorithm for answering queries using views and the experimental evaluation of such algorithms. Hence, we begin with the class of select-project-join queries.

The paper is organized as follows. Section 2 formally defines the problem, and Section 3 discusses the limitations of the previous algorithms. Section 4 describes the MiniCon algorithm, and Section 5 presents the experimental evaluation. Section 6 describes an extension of the MiniCon algorithm to comparison predicates. Section 7 describes how to extend the MiniCon algorithm to context of query optimization. Section 8 discusses related work and Section 9 concludes. The proof of the MiniCon algorithm is described in Appendix A.

2 Preliminaries

Queries and views: We consider the problem of answering queries using views for *conjunctive queries* (i.e., select-project-join queries). A *conjunctive query* has the form:

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

where q and $e_1, \ldots, e_n$ are predicate names. The atoms $e_1(\bar{X}_1), \ldots, e_n(\bar{X}_n)$ are the *subgoals* in the body of the query, where $e_1, \ldots, e_n$ refer to database relations. The atom $q(\bar{X})$ is called the *head* of the query, and refers to the answer relation. 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). The variables in $\bar{X}$ are the *distinguished* variables of the query, and all the others are *existential* variables. We denote individual variables by lowercase letters. We use $Vars(Q)$ ($Subgoals(Q)$) to refer to the set of variables (subgoals) in Q, and $Q(D)$ to refer to the result of evaluating the query Q over the database D.

Note that unions can be expressed in this notation by allowing a set of conjunctive queries with the same head predicate. A *view* is a named query. If the query results are stored, we refer to them as a materialized view, and we refer to the result set as the *extension* of the view. In Section 6 we consider queries that contain subgoals with comparison predicates $<$, $\leq$, $\neq$. 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.

Example 1 : Consider the following schema that we use throughout the paper. The relation cites(p1,p2) stores pairs of publication identifiers where p1 cites p2. The relation sameTopic stores pairs of papers that are on the same topic. The unary relations inSIGMOD and inVLDB store ids of papers published in SIGMOD and VLDB respectively. The following query asks for pairs of papers on the same topic that also cite each other. Note that join predicates in this notation are expressed by multiple occurrences of the same variables.

Q(x,y):- sameTopic(x,y), cites(x,y), cites(y,x)

Query containment and equivalence: The concepts of query containment and equivalence enable us to compare between queries and rewritings. We say that a query Q_1 is *contained* in the query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if the answer to Q_1 is a subset of the answer to Q_2 for *any* database instance. We say that Q_1 and Q_2 are *equivalent* if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$, i.e., they produce the same set of tuples for any given database.

Containment mappings provide a necessary and sufficient condition for testing query containment. A mapping τ from $Vars(Q_2)$ to $Vars(Q_1)$ is a containment mapping if (1) τ maps every subgoal in the body of Q_2 to a subgoal in the body of Q_1, and (2) τ maps the head of Q_2 to the head of Q_1. The query Q_2 contains Q_1 if and only if there is a containment mapping from Q_2 to Q_1 [CM77].

Given a partial mapping τ on the variables of a query, we extend it in the obvious manner to apply to sets of variables and to subgoals of the query (when all the variables of the subgoal are in the domain of τ).

Answering queries using views: Given a query Q and a set of view definitions $\mathcal{V} = V_1, \ldots, V_m$, a rewriting of the query using the views is a query expression Q' whose body predicates are either $V_1, \ldots, V_m$ or comparison predicates.

We distinguish between two types of query rewritings: *equivalent rewritings*, that are used in the contexts of query optimization and the maintenance of physical data independence, and *maximally-contained rewritings*, that are used in the context of data integration.

Definition 1 *(Equivalent rewriting) Let Q be a query, and $\mathcal{V} = V_1, \ldots, V_n$ be a set of views, both over the same database schema. The query Q' is an equivalent rewriting of Q using $\mathcal{V}$ if for any database D, the result of evaluating Q' over $V_1(D), \ldots, V_n(D)$ is the same as $Q(D)$.*

Example 2 : Consider the query from Example 1 and the following views. The view V1 stores pairs of papers that cite each other, and V2 stores pairs of papers on the same topic and each of which cites at least one other paper.

Q(x,y):- sameTopic(x,y), cites(x,y), cites(y,x)
V1(a,b):- cites(a,b), cites(b,a)
V2(c,d) :- sameTopic(c,d), cites(c,c1), cites(d,d1)

The following is an equivalent rewriting of Q:

Q'(x,y):- V1(x,y), V2(x,y) .

To check that Q' is an equivalent rewriting, we unfold the view definitions to obtain Q", and show that Q is equivalent to Q" using a containment mapping (in this case it's the identity on x and y and x1 → y, y1 → x).

Q"(x,y):- cites(x,y), cites(y,x), sameTopic(x,y), cites(x,x1)
cites(y,y1)

Data Integration: One of the main uses of algorithms for answering queries using views is in the context of data integration systems that provide their users with a uniform interface to a multitude of data sources [LRO96, KW96, FW97, Ull97, LKG99]. Users pose queries in terms of a *mediated schema*, which is a set of relations designed to capture the salient aspects of the application. The data, however, is stored in the sources. In order to be able to translate users' queries into queries on the data sources, the data integration system needs a description of the contents of the sources. One of the approaches to specifying such descriptions is to describe a data source as a view over the mediated schema, specifying which tuples can be found in the source. For example, in our domain, we may have two data sources, S1 and S2, containing pairs of SIGMOD (respectively VLDB) papers that cite each other. The sources can be described as follows:

S1(a,b):- cites(a,b), cites(b,a), inSIGMOD(a),
inSIGMOD(b)
S2(a,b):- cites(a,b), cites(b,a), inVLDB(a), inVLDB(b)

Given a query Q, the data integration system first needs to reformulate Q to refer to the data sources, i.e., the views. There are two differences between this application of answering queries using views and that considered in the context of query optimization. First, the views here are not assumed to contain *all* the tuples in their definition since the data sources are managed autonomously. For example, the source S1 may not contain all the pairs of SIGMOD papers that cite each other. Second, we cannot always find an equivalent rewriting of the query using the views because there may be no data sources that contain all of the information the query needs. Instead, we consider the problem of finding a maximally-contained rewriting, as illustrated below.

Example 3 : Continuing with our example, assuming we have the data sources described by S1, S2 and V2 and the same query Q, the best rewriting we can generate is:

Q'(x,y):- S1(x,y), V2(x,y)
Q'(x,y):- S2(x,y), V2(x,y)

Note that this rewriting is a union of conjunctive queries, describing multiple ways of obtaining answer to the query from the available sources. The rewriting is not an equivalent rewriting, since it misses any pair of papers that is not both in SIGMOD or both in VLDB, but we don't have data sources to provide us such pairs. Furthermore, since the sources are not guaranteed to have all the tuples in the definition of the view, our rewritings need to consider different views that may have similar definitions. For example, suppose we have the following source S3:

S3(a,b):- cites(a,b), cites(b,a), inSIGMOD(a),
inSIGMOD(b)

The definition of S3 is identical to that of S1, however, because of source incompleteness, it may contain different tuples than S1. Hence, our rewriting will also have to include the following in addition to the other two rewritings.

Q'(x,y):- S3(x,y), V2(x,y)

Maximally-contained rewritings are defined with respect to a particular query language in which we express rewritings. Intuitively, the maximally-contained rewriting is one that provides all the answers possible from a given set of sources. Formally, they are defined as follows.

Definition 2 *(Maximally-contained rewriting) The query Q' is a maximally-contained rewriting of a query Q using the views $\mathcal{V} = V_1, \ldots, V_n$ w.r.t. a query language $\mathcal{L}$ if*

1. *for any database D, and extensions $v_1, \ldots, v_n$ of the views such that $v_i \subseteq V_i(D)$, for $1 \leq i \leq n$, then $Q'(v_1, \ldots, v_n) \subseteq Q(D)$ for all i,*
2. *there is no other query Q_1 in the language $\mathcal{L}$, such that for every database D and extensions $v_1, \ldots, v_n$ as above (1) $Q'(v_1, \ldots, v_n) \subseteq Q_1(v_1, \ldots, v_n)$ and (2) $Q_1(v_1, \ldots, v_n) \subseteq Q(D)$, and there exists at least one database for which (1) is a strict set inclusion.*

Note that in the above definition, Q_1 and Q' need to be in the language $\mathcal{L}$, but Q does not have to.

Given a conjunctive query Q and a set of conjunctive views $\mathcal{V}$, the maximally-contained rewriting of a conjunctive query may be a union of conjunctive queries (we refer to the individual conjunctive queries as *conjunctive rewritings*). Hence, considering Definition 2, if the language $\mathcal{L}$ is less expressive than non-recursive datalog, there may not be a maximally-contained rewriting of the query. When the queries and the views are conjunctive and do not contain comparison predicates, it follows from [LMSS95] that we need only consider conjunctive rewritings Q' that have at most the number of subgoals in the query Q.

The ability to find a maximally-contained rewriting depends in subtle ways on other properties of the problem. It follows from [AD98] that if (1) the query contains comparison subgoals, or (2) the views are assumed to be complete, then there may not be a maximally-contained rewriting if we consider $\mathcal{L}$ to be the language of unions of conjunctive queries or even if we consider datalog with recursion.

Remark 1 : It is important to emphasize at this point that the definitions considered in this section only ensure that the rewriting of the query obtains as many answers as possible from a set of views, which is the main concern in the context of data integration. The bulk of this paper is not concerned with the problem of finding the rewriting that yields the *cheapest* query execution plan over the views, which would be the main concern if our goal was query optimization. In Section 7 we present an extension of the MiniCon algorithm to the context of query optimization, and show how the ideas underlying the MiniCon algorithm apply in that context as well. In addition, we do not consider here the issue of ordering the results from the sources. □

3 Previous Algorithms

The theoretical results on answering queries using views [LMSS95] showed that when there are no comparison predicates in the query, the search for a maximally-contained rewriting can be confined to a finite space: an algorithm needs to consider every possible conjunction of n or fewer view atoms, where n is the number of subgoals in the query. Two previous algorithms, the bucket algorithm and the inverse-rules algorithm, attempted to find more effective methods to produce rewritings that do not require such exhaustive search. In this section we briefly describe these algorithms and point out their limitations. In Section 5 we compare these algorithms to our MiniCon algorithm and show that the MiniCon algorithm significantly outperforms them. We describe the algorithms for queries and views without comparison subgoals.

3.1 The Bucket Algorithm

The bucket algorithm was developed as part of the Information Manifold System [LRO96]. The key 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 a particular subgoal. The bucket algorithm is even more effective in the presence of comparison subgoals because comparison subgoals often enable the bucket algorithm to deem many views as being irrelevant to a query.

We illustrate the bucket algorithm with the following query and views. Note that the query now only asks for a set of papers, rather than pairs of papers.

Q1(x) :- cites(x,y),cites(y,x),sameTopic(x,y)
V4(a) :- cites(a,b), cites(b,a)
V5(c,d) :- sameTopic(c,d)
V6(f,h) :- cites(f,g),cites(g,h),sameTopic(f,g)

In the first step, the bucket algorithm creates a bucket for each subgoal in Q1. The bucket for a subgoal g contains the views that include subgoals to which g can be mapped in a rewriting of the query. If a subgoal g unifies with more than one subgoal in a view V, then the bucket of g will contain multiple occurrences of V. [1] The bucket algorithm would create the following buckets:

cites(x,y)	cites(y,x)	sameTopic(x,y)
V4(x)	V4(x)	V5(x,y)
V6(x,y)	V6(x,y)	V6(x,y)

Note that it is possible to unify the subgoal cites(x,y) in the query with the subgoal cites(b,a) in V4, with the mapping x → b, y → a. However, the algorithm did not include the entry V4(y) in the bucket because it requires that every distinguished variable in the query be mapped to a distinguished variable in the view.

In the second step, the algorithm considers conjunctive query rewritings, each consisting of one conjunct from every bucket. Specifically, for each element of the Cartesian product of the buckets, the algorithm constructs a conjunctive rewriting and checks whether it is contained (or can be made to be contained by adding join predicates) in the query. If so, the rewriting is added to the answer. Hence, the result of the bucket algorithm is a union of conjunctive rewritings.

In our example, the algorithm will try to combine V4 with the other views and fail (as we explain below). Then it will consider the rewritings involving V6, and note that by equating the variables in the head of V6 a contained rewriting is obtained. Finally, the algorithm will also note that V6 and V5 can be combined. Though not originally described as part of the bucket algorithm, it is possible to add an additional simple check that will determine that the resulting rewriting will be redundant (because V5 can be removed). Hence, the only rewriting in this example (which also turns out to be an equivalent rewriting) is:

Q1'(x) :- V6(x,x)

[1] If we have knowledge of functional dependencies in the schema, then it is often possible to recover an attribute that has been projected away, but we do not consider this case here.

The main inefficiency of the bucket algorithm is that it misses some important interactions between view subgoals by considering each subgoal in isolation. As a result, the buckets contain irrelevant views, and hence the second step of the algorithm becomes very expensive. We illustrate this point on our example.

Consider the view V4, and suppose that we decide to use V4 in such a way that the subgoal cites(x,y) is mapped to the subgoal cites(a,b) in the view, as shown below:

```
Q1(x) :-  cites(x,y),cites(y,x), sameTopic(x,y)
              ↓          ↓                  ?
V4(a) :-  cites(a,b)cites( b,a)
```

We can map y to b and be able to satisfy both cites predicates. However, since b does not appear in the head of V4, if we use V4, then we will not be able to apply the join predicate between cites(x,y) and sameTopic(x,y) in the query. Therefore, V4 is not usable for the query, but the bucket algorithm would not discover this.

Furthermore, even if the query did not contain sameTopic(x,y), the bucket algorithm would not realize that if it uses V4, then it has to use it for *both* of the query subgoals. Realizing this would save the algorithm exploring useless combinations in the second phase.

As we explain later, the MiniCon algorithm discovers these interactions in the first phase. In this example, MiniCon will determine that V4 is irrelevant to the query. In the case in which the query does not contain the subgoal sameTopic(x,y), the MiniCon algorithm will discover that the two cite subgoals need to be treated atomically.

3.2 The Inverse-Rules Algorithm

Like the bucket algorithm, the inverse-rules algorithm [Qia96, DG97a] was also developed in the context of a data integration system. 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. Given the views in the previous example, the algorithm would construct the following inverse rules:

R1: cites(a, f1(a)) :- V4(a)
R2: cites(f1(a), a) :- V4(a)
R3: sameTopic(c,d) :- V5(c,d)
R4: cites(f, f2(f,h)) :- V6(f,h)
R5: cites(f2(f,h), h) :- V6(f,h)
R6: sameTopic(f, f2(f,h)) :- V6(f,h)

Consider the rules R1 and R2; intuitively, their meaning is the following. A tuple of the form (p1) in the extension of the view V4 is a witness of two tuples in the relation cites. It is a witness in the sense that it tells that the relation cites contains a tuple of the form (p1, Z), for some value of Z, and that the relation also contains a tuple of the form (Z, p1), 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 Skolem term f1(Z). Note that there may be several values of Z in the database that cause the tuple (p1) to be in the self-join of cites, but all that we know is that there exists at least one such value.

The rewriting of a query Q using the set of views $\mathcal{V}$ is simply the composition of Q and the inverse rules for $\mathcal{V}$. Hence, one of the important advantages of the algorithm is that the inverse rules can be constructed ahead of time in polynomial time, independent of a particular query.

The rewritings produced by the inverse-rules algorithm, as originally described in [DG97a], are not appropriate for query evaluation for two reasons. First, applying the inverse rules to the extension of the views may invert some of the useful computation done to produce the view. Second, we may end up accessing views that are irrelevant to the query. To illustrate the first point, suppose we use the rewriting produced by the inverse-rules algorithm in the case where the view V6 has the extension { (p1, p1), (p2,p2) }.

First, we would apply the inverse rules to the extensions of the views. Applying R4 would yield cites(p1, f2(p1,p1)), cites(p2, f2(p2,p2)), and similarly applying R5 and R6 would yield the following tuples:

cites(p1, f2(p1,p1)),
cites(f2(p1,p1),p1),
cites(f2(p2,p2),p2),
sameTopic(p1,p1),
sameTopic(p2,p2).

Applying the query Q1 to the tuples computed above obtains the answers p1 and p2. However, this computation is highly inefficient. Instead of directly using the tuples of V6 for the answer, the inverse-rules algorithm first computed tuples for the relation cites, and then had to recompute the self-join of cites that was already computed for V6. Furthermore, if the extensions of the views V4 and V5 are not empty, then applying the inverse rules would produce useless tuples as explained in Section 3.1.

Hence, before we can fairly compare the inverse-rules algorithm to the others, we need to further process the rules. Specifically, we need to expand the query with every possible combination of inverse rules. However, expanding the query with the inverse rules turns out to repeat much of the work done in the second phase of the bucket algorithm. In our example, since we have four rules for cites and two rules for sameTopic, we may need to consider 32 such expansions in the worst case.

In the experiments described in Section 5 we consider an extended version of the inverse-rules algorithm that produces a union of conjunctive queries by expanding the definitions of the inverse rules. We expanded the subgoals of the query one at a time, so we could stop an expansion of the query at the moment when we detect that a unification for a subset of the subgoals will not yield a rewriting (thereby optimizing the performance of the inverse-rules algorithm). We show that the inverse-rules algorithm can perform much better than the bucket algorithm, but the MiniCon algorithm scales up significantly better than either algorithm.

Remark 2 It is important to clarify why our study considers the extended version of the inverse-rules algorithm, rather than the original version. It is easy to come up with (real) examples in which the execution of plan generated by the original inverse-rules algorithm would be arbitrarily worse than that of the bucket algorithm or the MiniCon algorithm. Hence, we face the usual tradeoff between spending significant time on optimization, but with much more substantial savings at run-time. An optimizer that would accept the result of the original inverse-rules algorithm would definitely try to optimize the plan by trying to reduce the number of joins it needs to perform. By using the extended version of the inverse-rules algorithm we are putting all three algorithms on equal footing in the sense that one does not need more optimization than the other. Optimizations will still be applied to them, but the same optimizations can be applied to the results of each of the algorithms. □

4 The MiniCon Algorithm

The MiniCon algorithm begins like the bucket algorithm, considering which views contain subgoals that correspond to subgoals in the query. However, once the algorithm finds a partial mapping from a subgoal g in the query to a subgoal g_1 in a view V, it changes perspective and looks at the variables in the query. The algorithm considers the join predicates in the query (which are specified by multiple occurrences of the same variable) and finds the minimal additional set of subgoals that need 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 generalization of buckets. In the second phase, the algorithm combines the MCDs to produce the rewritings. It is important to note that because of the way we construct the MCDs, the MiniCon algorithm does not require containment checks in the second phase, giving it an additional speedup compared to the bucket algorithm. Section 4.1 describes the construction of MCDs, and Section 4.2 describes the combination step. For ease of exposition we describe the MiniCon algorithm for queries and views without constants. The proof of correctness of the MiniCon algorithm can be found in Appendix A.

4.1 Forming the MCDs

We begin by introducing a few terms that are used in the description of the algorithm. Given a mapping τ from $Vars(Q)$ to $Vars(V)$, we say that a view subgoal g_1 *covers* a query subgoal g if $\tau(g) = g_1$.

A MCD is a mapping from a subset of the variables in the query to variables in one of the views. Intuitively, a MCD represents a fragment of a containment mapping from the query to the rewriting of the query. The way in which we construct the MCDs guarantees that these fragments can later be combined seamlessly.

As seen in our example, we need to consider mappings from the query to specializations of the views, where some of the head variables may have been equated (e.g., V6(x,x) instead of V6(x,y) in our example). Hence, every MCD has an associated *head homomorphism.* A head homomorphism h on a view V is a mapping h from $Vars(V)$ to $Vars(V)$ that is the identity on the existential variables, but may equate distinguished variables, i.e., for every distinguished variable x, $h(x)$ is distinguished, and $h(x) = h(h(x))$.

Formally, we define MCDs as follows.

Definition 3 *(MiniCon Descriptions) A MCD C for a query Q over a view V is a tuple of the form $(h_C, V(\bar{Y})_C, \varphi_C, G_C)$ where:*

- *h_C is a head homomorphism on V,*
- *$V(\bar{Y})_C$ is the result of applying h_C to V, i.e., $\bar{Y} = h_C(\bar{A})$, where $\bar{A}$ are the head variables of V,*
- *φ_C is a partial mapping from $Vars(Q)$ to $h_C(Vars(V))$*
- *G_C is a subset of the subgoals in Q which are covered by some subgoal in $h_C(V)$ using the mapping φ_C (note: not all such subgoals are necessarily included in G_C).*

In words, φ_C is a mapping from Q to the specialization of V obtained by the head homomorphism h_C. G_C is a set of subgoals of Q that we cover by the mapping φ_C. Property 1 below specifies the exact conditions we need to consider when we decide which subgoals to include in G_C. Note that $V(\bar{Y})_C$ is uniquely determined by the other elements of a MCD, but is part of a MCD specification for clarity in our subsequent discussions. Furthermore, the algorithm will not consider all the possible MCDs, but only those in which h_C is the least restrictive head homomorphism necessary in order to unify subgoals of the query with subgoals in a view.

The mapping φ_C of a MCD C may map a set of variables in Q to the same variable in $h_C(V)$. In our discussion, we sometimes need to refer to a representative variable of such a set. For each such set of variables in Q we choose a representative variable arbitrarily, except that we choose a distinguished variable whenever possible. For a variable x in Q, $EC_{\varphi_C}(x)$ denotes the representative variable of the set to which x belongs. $EC_{\varphi_C}(x)$ is defined to be the identity on any variable that is not in Q.

procedure **formMCDs**$(Q, \mathcal{V})$
/* Q and $\mathcal{V}$ are conjunctive queries. */
  $\mathcal{C} = \emptyset$.
  For each subgoal $g \in Q$
    For view $V \in \mathcal{V}$ and every subgoal $v \in V$
    Let h be the least restrictive head homomorphism on V
      such that there exists a mapping φ, s.t. $\varphi(g) = h(v)$.
    If h and φ exist, then add to $\mathcal{C}$ any new MCD C
      that can be constructed where:
      (a) φ_C (resp. h_C) is an extension of φ (resp. h),
      (b) G_C is the minimal subset of subgoals of Q such
        that G_C, φ_C and h_C satisfy Property 1, and
      (c) It is not possible to extend φ and h to φ'_C and h'_C
        s.t. (b) is satisfied and G'_C, as defined in (b), is
        a subset of G_C.
  Return $\mathcal{C}$

Fig. 1 First phase of the MiniCon algorithm: Forming MCDs. Note that condition (b) minimizes G_c *given* a choice of h_C and φ_C, and is therefore not redundant with condition (c).

The construction of the MCDs is based on the following observation on the properties of query rewritings. The proof of this property is a corollary of the correctness proof of the MiniCon algorithm.

Property 1 Let C be a MCD for Q over V. Then C can only be used in a non-redundant rewriting of Q if the following conditions hold:

C1. For each head variable x of Q which is in the domain of φ_C, $\varphi_C(x)$ is a head variable in $h_C(V)$.
C2. If $\varphi_C(x)$ is an existential variable in $h_C(V)$, then for every g, subgoal of Q, that includes x (1) all the variables in g are in the domain of φ_C, and (2) $\varphi_C(g) \in h_C(V)$

Clause C1 is the same as in the bucket algorithm. Clause C2 captures the intuition we illustrated in our example, where if a variable x is part of a join predicate which is not enforced by the view, then x must be in the head of the view so the join predicate can be applied by another subgoal in the rewriting. In our example, clause C2 would rule out the use of V4 for query Q1 because the variable b is not in the head of V4, but the join predicate with sameTopic(x,y) has not been applied in V4.

The algorithm for creating the MCDs is shown in Figure 1. Consider the application of the algorithm to our example with the query Q1 and the views V4, V5, and V6. The MCDs that will be created are shown in Figure 2.

We first consider the subgoal cites(x,y) in the query. As discussed above, the algorithm does not create a MCD for V4 because clause C2 of Property 1 would be violated (the property would require that V4 also cover the subgoal sameTopic(x,y) since b is existential in V4). For the same reason, no MCD will be created for V4 even when we consider the other subgoals in the query.

$V(\bar{Y})$	h	φ	G
V5(c,d)	c → c, d → d	x → c, y → d	3
V6(f,f)	f → f, h → f	x → f, y → f	1,2,3

Fig. 2 MCDs formed as part of our example of the MiniCon algorithm.

In a sense, the MiniCon algorithm shifts some of the work done by the combination step of the bucket algorithm to the phase of creating the MCDs. The bucket algorithm will discover that V4 is not usable for the query when combining the buckets. However, the bucket algorithm needs to discover this many times (each time it considers V4 in conjunction with another view), and every time it does so, it uses a containment check, which is much more expensive. Hence, as we show in the next section, with a little more effort spent in the first phase, the overall performance of the MiniCon algorithm outperforms the bucket algorithm and the inverse-rules algorithm.

Another interesting observation is the difference in performance in the presence of repeated occurrences of the same predicate in the views or the query. For the bucket algorithm repeated occurrences lead to larger buckets, and hence more combinations to check in the second phase. For the inverse-rules algorithm, repeated occurrences mean there are more expansions to check in the second phase. In contrast, the MiniCon algorithm can more often rule out the consideration of certain occurrences of a predicate due to violations of Property 1.

Remark 3 (covered subgoals) : When we construct a MCD C, we must determine the set of subgoals of the query G_C that are covered by the MCD. The algorithm includes in G_C only the *minimal* set of subgoals that are necessary in order to satisfy Property 1. To see why this is not an obvious choice, suppose we have the following query and views:

Q1'(x) :- cites(x,y),cites(z,x), inSIGMOD(x)
V7(a) :- cites(a,b), inSIGMOD(a)
V8(c) :- cites(d,c), inSIGMOD(c)

One can also consider including the subgoal inSIGMOD(x) in the set of covered subgoals for the MCD for both V7 and V8, because x is in the domain of their respective variable mappings anyway. However, our algorithm will not include inSIGMOD(x), and will instead create a special MCD for it.

The reason for our choice is that it enables us to focus in the second phase only on rewritings where the MCDs cover *mutually exclusive* sets of subgoals in the query, rather than overlapping subsets. This yields a more efficient second phase. □

4.2 Combining the MCDs

Our method for constructing MCDs pays off in the second phase of the algorithm, where we combine MCDs to build the conjunctive rewritings. In this phase we consider combinations of MCDs, and for each valid combination we create a conjunctive rewriting of the query. The final rewriting is a union of conjunctive queries.

The following property states that the MiniCon algorithm need only consider combinations of MCDs that cover pairwise disjoint subsets of subgoals of the query. The proof of the property follows from the correctness proof of the MiniCon algorithm.

Property 2 Given a query Q, a set of views $\mathcal{V}$, and the set of MCDs $\mathcal{C}$ for Q over the views in $\mathcal{V}$, the only combinations of MCDs that can result in non-redundant rewritings of Q are of the form $C_1, \ldots, C_l$, where

D1. $G_{C_1} \cup \ldots \cup G_{C_l} = Subgoals(Q)$, and
D2. for every $i \neq j$, $G_{C_i} \cap G_{C_j} = \emptyset$.

The fact that we only need to consider sets of MCDs that provide partitions of the subgoals in the query drastically reduces the search space of the algorithm. Furthermore, even though we do not discuss it here, the algorithm can also be extended to output the rewriting in a compact encoding that identifies the common subexpressions of the conjunctive rewritings, and therefore leads to more efficient query evaluation. We note that had we chosen the alternate strategy in Remark 3, clause D2 would not hold.

Given a combination of MCDs that satisfies Property 2, the actual rewriting is constructed as shown in Figure 3.

In the final step of the algorithm we tighten up the rewritings by removing redundant subgoals as follows. Suppose a rewriting Q' includes two atoms A_1 and A_2 of the same view V, whose MCDs were C_1 and C_2, and the following conditions are satisfied: (1) whenever A_1 (resp. A_2) has a variable from Q in position i, then A_2 (resp. A_1) either has the same variable or a variable that does not appear in Q in that position, and (2) the ranges of φ_{C_1} and φ_{C_2} do not overlap on existential variables of V. In this case we can remove one of the two atoms by applying to Q' the homomorphism τ that is (1) the identity on the variables of Q and (2) is the most general unifier of A_1 and A_2. The underlying justification for this optimization is discussed in [LMSS95], and it can also be applied to the bucket algorithm and the inverse-rules algorithm.

We note that even after this step, the rewritings may still contain redundant subgoals. However, removing them involves several tests for query containment; both inverse-rules algorithm and the bucket algorithm require these removal steps as well.

In our example, the algorithm will consider using V5 to cover subgoal 3, but when it realizes that there are no

procedure **combineMCDs**($\mathcal{C}$)
/* $\mathcal{C}$ are MCDs formed by the first step of the algorithm. */
/* Each MCD has the form $(h_C, V(\bar{Y}), \varphi_C, G_C, EC_C)$. */
Given a set of MCDs, $C_1, \ldots, C_n$, we define the function EC on $Vars(Q)$ as follows:
If for $i \neq j$, $EC_{\varphi_i}(x) \neq EC_{\varphi_j}(x)$, define $EC_{\mathcal{C}}(x)$ to be one of them arbitrarily but consistently across all y for which $EC_{\varphi_i}(y) = EC_{\varphi_i}(x)$
Let $Answer = \emptyset$
For every subset $C_1, \ldots, C_n$ of $\mathcal{C}$ such that $G_{C_1} \cup\ G_{C_2} \cup \ldots \cup G_{C_n} = subgoals(Q)$ and for every $i \neq j$, $G_{C_i} \cap G_{C_j} = \emptyset$
Define a mapping Ψ_i on the $\bar{Y}_i$'s as follows:
If there exists a variable $x \in Q$ such that $\varphi_i(x) = y$
$\Psi_i(y) = x$
Else
Ψ_i is a fresh copy of y
Create the conjunctive rewriting
$Q'(EC(\bar{X}))$:- $V_{C_1}(EC(\Psi_1(\bar{Y_{C_1}}))), \ldots,$
$V_{C_n}(EC(\Psi_n(\bar{Y_{C_n}})))$
Add Q' to $Answer$.
Return $Answer$.

Fig. 3 Phase 2: combining the MCDs.

MCDs that cover either subgoal 1 or 2 without covering subgoal 3, it will discard V5. Thus the only rewriting that will be considered is

Q1'(x) :- V6(x,x).

Constants in the query and views: When the query or the view include constants, we make the following modifications to the algorithm. First, the domain and range of φ_C in the MCDs may also include constants. Second, a MCD also records a (possibly empty) set of mappings ψ_C from variables in $Vars(Q)$ to constants.

When the query includes constants, we add the following condition to Property 1:

C3. If a is a constant in Q it must be the case that either (1) $\varphi_C(a)$ is a distinguished variable in $h_C(V)$ or (2) $\varphi_C(a)$ is the constant a.

When the views have constants, we modify Property 1 as follows:

- We relax clause C1: a variable x that appears in the head of the query must either be mapped to a head variable in the view (as before) or be mapped to a constant a. In the latter case, the mapping $x \rightarrow a$ is added to ψ_C.
- If $\varphi_C(x)$ is a constant a, then we add the mapping $x \rightarrow a$ to ψ_C. (Note that condition C2 only applies to existential variables, and therefore if $\varphi_C(x)$ is a constant that appears in the body of V but not in the head, a MCD is still created).

Next, we combine MCDs with some extra care. Two MCDs, C_1 and C_2, both of whom have x in their domain, can be combined only if they (1) either both map x to the same constant, or (2) one (e.g., C_1) maps x to a constant and the other (e.g., C_2) maps x to distinguished variable in the view. Note that if C_2 maps x to an existential variable in the view, then the MiniCon algorithm would never consider combining C_1 and C_2 in the first place, because they would have overlapping G_C sets.

Finally, we modify the definition of EC, such that whenever possible, it chooses a constant rather than a variable. □

The following theorem summarizes the properties of the MiniCon algorithm. Its full proof is given in the appendix.

Theorem 1 *Given a conjunctive query Q and conjunctive views $\mathcal{V}$, both without comparison predicates or constants, the MiniCon algorithm produces the union of conjunctive queries that is a maximally-contained rewriting of Q using $\mathcal{V}$.* □

It should be noted that the worst-case asymptotic running time of the MiniCon algorithm is the same as that of the bucket algorithm and of the inverse-rules algorithm after the modification described in Section 3.2. In all cases, the running time is $O(n\, m\, M)^n$, where n is the number of subgoals in the query, m is the maximal number of subgoals in a view, and M is the number of views.

The next section describes experimental results showing the differences between the three algorithms in practice.

5 Experimental Results

The goal of our experiments was twofold. First, we wanted to compare the performance of the bucket algorithm, the inverse-rules algorithm, and the MiniCon algorithm in different circumstances. Second, we wanted to validate that MiniCon can scale up to large number of views and large queries. Our experiments considered three classes of queries and views: (1) chain queries, (2) star queries and (3) complete queries, all of which are well known in the literature [MGA97].

To facilitate the experiments, we implemented a random query generator which enables us to control the following parameters (1) the number of subgoals in the queries and views, (2) the number of variables per subgoal, (3) the number of distinguished variables, and (4) the degree to which predicate names are duplicated in the queries and views. The results are averaged over multiple runs generated with the same parameters (at least 40, and usually more than 100). All graphs either contain 95% confidence intervals or the intervals were less than twice as thick as the line in the graph and were thus excluded. An important variable to keep in mind throughout the experiments is the number of rewritings that can actually be obtained.

In most experiments we considered queries and views that had the same query shape and size. Our experiments were all run on a dual Pentium II 450 MHz running Windows NT 4.0 with 512MB RAM. All of the algorithms were implemented in Java and compiled to an executable.

5.1 Chain queries

In the context of chain queries we consider several cases. In the first case, shown in Figure 4, only the first and last variables of the query and the view are distinguished. Therefore, in order to be usable, a view has to be identical to the query, and as a result there are very few rewritings. The bucket algorithm performs the worst, because of the number and cost of the query containment checks it needs to perform (it took on the order of 20 seconds for 5 views of size 10 subgoals, and hence we do not even show it on the graph). The inverse-rules algorithm and the MiniCon algorithm scale linearly in the number of views, but the MiniCon algorithm outperforms the inverse-rules algorithm by a factor of about 2 (and this factor is independent of query and view size). In fact, the MiniCon algorithm can handle more than 350 views with 10 subgoals each in less than one second.

The difference in the performance between the inverse-rules algorithm and the MiniCon algorithm in this context and in others is due to the second phases of the algorithms. In this phase, the inverse-rules algorithm is searching for a unification of the subgoals of the query with heads of inverse rules. The MiniCon algorithm is searching for sets of MCDs that cover all the subgoals in the query, but cover pairwise disjoint subsets. Hence, the MiniCon algorithm is searching a much smaller space, because the number of subgoals is smaller than the number of variables in the query. Moreover the MiniCon algorithm is performing better because in the first phase of the algorithm it already removed from consideration views that may not be usable due to violations of Property 1. In contrast, the inverse-rules algorithm must try unifications that include such views and then backtrack. The amount of work that the inverse-rules algorithm will waste depends on the order in which it considers the subgoals in the query when it unifies them with the corresponding inverse rules. If a failure appears late in the ordering, more work is wasted. The important point to note is that the optimal order in which to consider the subgoals depends heavily on the specific views available and is, in general, very hard to find. Hence, it would be hard to extend the inverse-rules algorithm such that its second phase would compare in performance to that of the MiniCon algorithm.

In the second case we consider, shown in Figure 5, the views are shorter than the query (of lengths 2, 3 and 4, while the query has 12 subgoals).

Finally, as shown in Figure 6, we also considered another case in which all the variables in the views are distinguished. In this case, there are many rewritings (often more than 1000), and hence the performance of the algorithms is limited because of the sheer number of rewritings. Since virtually all combinations produce contained rewritings, any complete algorithm is forced to form a possibly exponential number of rewritings; for queries and views with 8 subgoals, the algorithms take on the order of 100 seconds for 5 views. The graph in Figure 6 shows that on average the MiniCon algorithm performs better than the inverse-rules algorithm by anywhere between 10% and 25%. However, in this case the variance in the results is very high, and hence it is hard to draw any general conclusions. (The confidence intervals cannot be shown in the graph without cluttering it.) The reason for the large variance is that some of the queries in the workload have a huge number of rewritings (and hence take much more time), while others have a very small number of rewritings. Other experiments showed that the savings for the MiniCon algorithm over the inverse-rules algorithm, as expected, grew with the number of views and the number of subgoals in the query; this is because the number of combinations that was considered was much higher and thus the smaller search space that the MiniCon algorithm considered was much more evident.

5.2 Star and complete queries

In star queries, there exists a unique subgoal in the query that is joined with every other subgoal, and there are no joins between the other subgoals. In the cases of two distinguished variables in the views or all view variables being distinguished, the performance of the algorithms mirrors the corresponding cases of chain queries. Hence, we omit the details of these experiments. Figure 7 shows the running times of the inverse-rules algorithm and the MiniCon algorithm in the case where the distinguished variables in the views are the ones that do not participate in the joins. In this case, there are relatively few rewritings. We see that the MiniCon algorithm scales up much better than the inverse-rules algorithm. For 20 views with 10 subgoals each, the MiniCon algorithm runs 15 times faster than the inverse-rules algorithm. Here the explanation is that the first phase of the MiniCon algorithm is able to prune many of the irrelevant views, whereas the inverse-rules algorithm discovers that the views are irrelevant only in the second phase, and often it must be discovered multiple times.

An experiment with similar settings but for complete queries is shown in Figure 8. In complete queries every subgoal is joined with every other subgoal in the query. As the figure shows, the MiniCon algorithm outperforms the inverse-rules algorithm by a factor of 2.3 for 20 views, and by a factor of 3 for 50 views, which is less of a speedup than with of star queries. The explanation for this is that there are more joins in the query, and

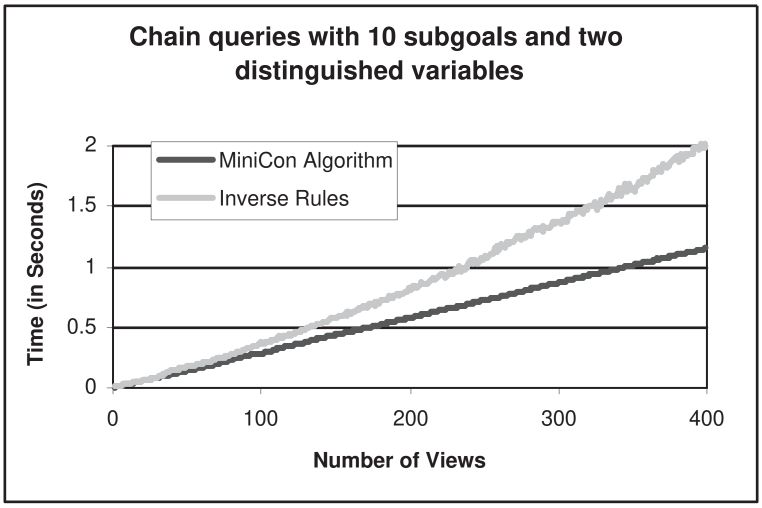

Fig. 4 This graph considers chain queries with two distinguished variables in the views, and shows that the MiniCon algorithm and the inverse-rules algorithms both scale up to hundreds of views. The MiniCon algorithm outperforms the inverse-rules algorithm by a factor of 2.

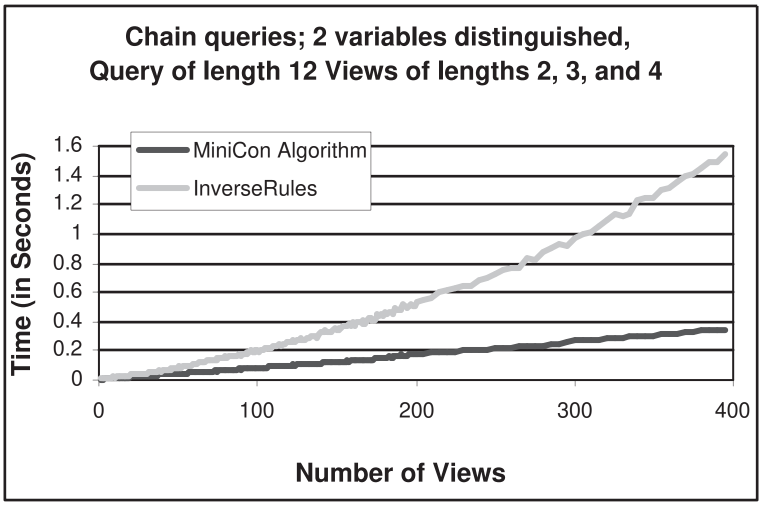

Fig. 5 This graph shows chain queries where the views are of lengths 2, 3 and 4, and the query has 12 subgoals.

thus the inverse-rules algorithm is able to detect useless views earlier in its search because failures to unify occur more frequently. Finally, we also ran some experiments on queries and views that were generated randomly with no specific pattern. The results showed that the MiniCon algorithm still scales up gracefully, but the behavior of the inverse-rules algorithm was too unpredictable (though always worse than the MiniCon algorithm), due to the nature of when the algorithms discover that a rule cannot be unified. Additional experiments are needed in order to draw any conclusion as to how the algorithms perform for completely random queries.

5.3 Summary

In summary, our experiments showed the following points. First, the MiniCon algorithm scales up to large numbers of views and significantly outperforms the other two algorithms. This point is emphasized by Table 1, where we tried to push the MiniCon algorithm to its limits. The table considers the number of subgoals and number of views that the MiniCon algorithm is able to process given 10 seconds. In some cases, the algorithm can handle thousands of views, which is a magnitude that was clearly out of reach of previous algorithms.

Second, the experiments showed that the bucket algorithm performed much worse than the other two algorithms in all cases. More interesting was the comparison between the MiniCon algorithm and the inverse-rules

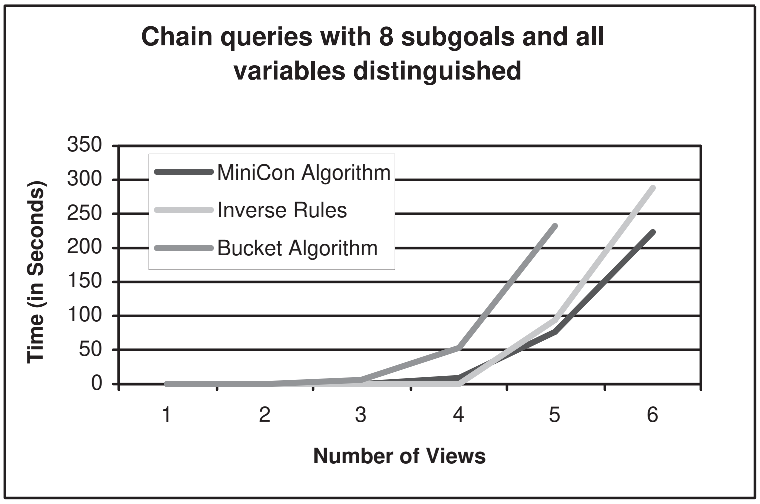

Fig. 6 Chain queries where all variables in the views are distinguished. Note that the containment check required by the bucket algorithm causes it to be roughly twice as slow as either the MiniCon algorithm or inverse-rules algorithm.

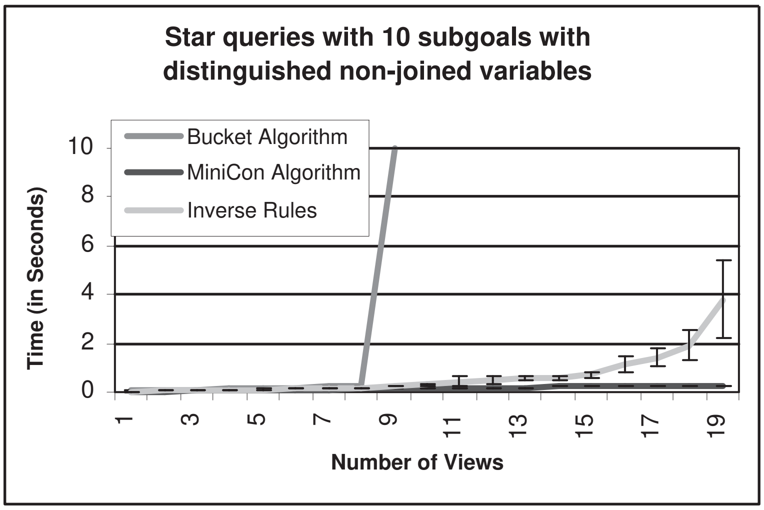

Fig. 7 This figure shows the running times for star queries, where the distinguished variables in the views are those not participating in the joins. The MiniCon algorithm significantly outperforms the inverse-rules algorithm.

Query type	Distinguished	# of subgoals	# of views
Chain	All	3	45
Chain	All	12	3
Chain	Two	5	9225
Chain	Two	99	115
Star	Non Joined	5	12235
Star	Non Joined	99	35
Star	Joined	10	4520
Star	Joined	99	75

Table 1 The number of views that the MiniCon algorithm can process in under 10 seconds in various situations

algorithm. In all cases the MiniCon algorithm outperformed the inverse-rules algorithm, though by differing factors. In particular, the performance of the inverse-rules algorithm was very unpredictable. The problem with the inverse-rules algorithm is that it discovers many of the interactions between the views in its second phase, and the performance in that phase is heavily dependent on the order in which it considers the query subgoals. However, since the optimal order depends heavily on the interaction with the views, a general method for ordering the subgoals in the query is hard to find. Finally, all three algorithms are limited in cases where the number of resulting rewritings is especially large since a complete algorithm must produce a possibly exponential number of rewritings.

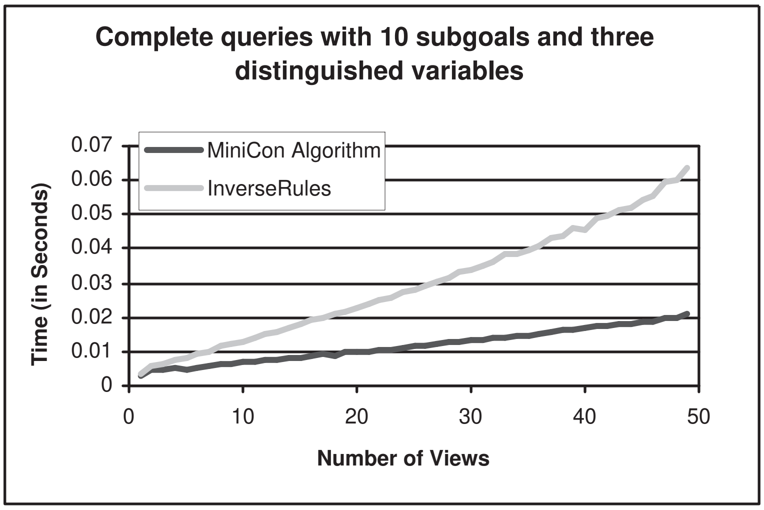

Fig. 8 This figure shows running times for complete queries where three variables are distinguished. As in Figure 7, the MiniCon algorithm significantly outperforms the inverse-rules algorithm.

6 Comparison predicates

The effect of comparison predicates on the problem of answering queries using views is quite subtle. If the views contain comparison predicates but the query does not, then the MiniCon algorithm without any changes still yields the maximally-contained query rewriting. On the other hand, if the query contains comparison predicates, then it follows from [AD98] that there can be no algorithm that returns a maximally-contained rewriting, even if we consider rewritings that are recursive datalog programs (let alone unions of conjunctive queries).

In this section we present an extension to the MiniCon algorithm that would (1) always find only correct rewritings (2) find the maximally-contained rewriting in many of the common cases in which comparison predicates are used, and (3) is guaranteed to produce the maximally-contained rewriting when the query contains only *semi-interval* constraints, i.e., when all the comparison predicates in the query are of the form $x \leq c$ or $x < c$, where x is a variable and c is a constant (or they are all of the form $x \geq c$ or $x > c$). We refer to this algorithm as MiniCon IP. We show experiments demonstrating the scale up of the extended algorithm. Finally, we show an example that provides an intuition for which cases the algorithm will not capture.

In our discussion, we refer to the set of comparison subgoals in a query Q as $I(Q)$. Given a set of variables $\bar{X}$, we denote by $I_{\bar{X}}(Q)$ the subset of the subgoals in $I(Q)$ that includes (1) only variables in $\bar{X}$ or constants and (2) contains at least one existential variable of Q. Intuitively, $I_{\bar{X}}(Q)$ denotes the set of comparison subgoals in the query that *must* be satisfied by the view if $\bar{X}$ is the domain of a MCD. We assume without loss of generality that $I(Q)$ is logically closed, i.e., that if $I(Q) \models g$, then $g \in I(Q)$. We can always compute the logical closure of $I(Q)$ in time that is quadratic in the size of Q [Ull89].

We make three changes to the MiniCon algorithm to handle comparison predicates. First, we only consider MCDs C that satisfy the following conditions:

1. If $x \in Vars(Q)$, $\varphi_C(x)$ is an existential variable in $h_C(V)$ and y appears in the same comparison atom as x, then y must be in the domain of φ_C.
2. If $\bar{X}$ is the set of variables in the domain of the mapping φ_C, then $I(h_C(V)) \models \varphi_C(I_{\bar{X}})$.

The first condition is an extension of Property 1, and the second condition guarantees the comparison subgoals in the view logically entail the relevant comparison subgoals in the query. Note that because of the second condition, the only subgoals in $I_{\bar{X}}(Q)$ that may not be satisfied by V must include only variables that φ_C maps to distinguished variables of V. As a result, such a subgoal can simply be added to the rewriting after the MCDs are combined.

The second change is that we disallow all MCDs that constrain variables to be incompatible with the variables they map in the query. For example, if a query has a subgoal x > 17 and a MCD maps x to a view variable a, and a < 5 is in the view, then we can ignore the MCD.

The third change we make to the MiniCon algorithm is the following: after forming a rewriting Q' by combining a set of MCDs, we add the subgoal $EC(g)$ for any subgoal of $I(Q)$ that is not satisfied by Q'.

Example 4 : Consider a variation on our running example, where the predicate year denotes the year of publication of a paper.

Q2(x) :- inSIGMOD(x), cites(x,y), year(x,r1), year(y,r2),
    r1 ≥ 1990, r2 ≤ 1985
V9(a,s1) :- inSIGMOD(a), cites(a,b), year(a,s1),

year(b,s2), s2 $\leq$ 1983
V10(a,s1) :- inSIGMOD(a), cites(a,b), year(a,s1), year(b,s2), s2 $\leq$ 1987

Our algorithm would first consider V9 with the mapping {x $\rightarrow$ a, y $\rightarrow$ b, r1 $\rightarrow$ s1, r2 $\rightarrow$ s2}. In this case, the subgoal r2 $\leq$ 1985 is satisfied by the view, but r1 $\geq$ 1990 is not. However, since s1 is a distinguished variable in V9, the algorithm can create the rewriting:

Q2'(x) :- V9(x,r1), r1 $\geq$ 1990

When the algorithm considers a similar variable mapping to V10, it will notice that the constraint on r2 is not satisfied, and since it is mapped to an existential variable in V10, no MCD is created.

Example 5 : The following example provides an intuition for which rewritings our extended algorithm will not discover. Consider the following query and view:

Q3(u) :- e(u,v), u $\leq$ v
V11(a) :- e(a,b), e(b,a)

The algorithm will not create any MCD because the subgoal $u \leq v$ in the query is not implied by the view. However, the following is a contained rewriting of $Q3$.

Q3'(u) :- V11(u)

In general, in order to find a containment mapping in the presence of comparison predicates, [Klu88] shows that we must find a mapping for every ordering of the variables. For example, we must consider two different containment mappings, depending on whether $a \leq b$ or $a > b$. In each of these mappings, the subgoal $e(u, v)$ may be mapped to a different subgoal. Our algorithm will only find rewritings in which the target of the mapping for a subgoal in the query is the same for any possible order on the variables.

Figures 9 and 10 show sample experiments that we ran on the extended algorithm in the case of chain queries. In the experiments, we took the identical queries and views and added a number of comparison subgoals of the form $x < c$ or $x > c$ to the queries under consideration by MiniCon IP.

The experiments show that the same trends we saw without comparison predicates appear here as well. In general, the addition of comparison predicates reduces the number of rewritings because more views can be deemed irrelevant. This is illustrated in Figure 10 where all of the variables in the views are distinguished and therefore without comparison predicates there would be many more rewritings. However, since the comparison predicates reduce the number of relevant views, the algorithm with comparison predicates scales up to a larger number of views. In Figure 9, the number of rewritings is very small, but the addition of the overhead to deal with comparison predicates does not appreciably slow the MiniCon algorithm.

7 Cost-based query rewriting

The previous sections considered the problem of answering queries using views for the context of data integration, where the incompleteness of the data sources required that we consider the union of all possible rewritings of the query. In this section we show how the principles underlying the MiniCon algorithm can also be used for answering queries using views in the context of query optimization (as in [TSI96, CKPS95]), and in the process, shed some light on the problem of query optimization with views. The fundamental difference in this context is that we want the *cheapest* rewriting of the query using the views. Since the views are assumed to be complete (i.e., include all the tuples satisfying their definition) and since we are looking for an equivalent rewriting, we can limit ourselves to a single rewriting.

The following example shows how considering cost affects the result of a rewriting algorithm.

Example 6 : Suppose we have the following query and views:

Q4(x,y) :- e1(x,y), e2(y,z), e3(z,x)
V12(a,b,c) :- e1(a,b), e2(b,c)
V13(d,e,f) :- e2(d,e), e3(e,f)
V14(g) :- e1(g,h), e3(i, g)

If the join of e1 and e3 is very selective, the cheapest rewriting of the query may be the following (assuming the subgoals are joined from left to right):

Q4'(x,y) :- V14(x), V12(x,y,z), V13(y,z,x)

Here, the view V14 does not contribute to the *logical* correctness of the query, but only to reducing the cost of the query plan. Note that the MiniCon algorithm would not consider V14(x) because it would not create a MCD for V14, since Property 1 would not be satisfied. □

In general, the problem of answering queries using views in the context of query optimization requires that we consider views for two different roles: the logical correctness of the query, and the reduction in the cost of the rewriting. In fact, it is shown in [CH00] that the optimal query execution plan may include an exponential (in the size of the query and schema) number of views in the second role, while it follows from [LMSS95] that the number of views in the first role is bounded by the number of subgoals in the query.

We proceed in two steps. In Section 7.1 we show how the information captured in MCDs can be used to improve the bottom-up dynamic-programming algorithm used in [TSI96] for query optimization using materialized views. However, the algorithm we describe in Section 7.1 only considers views that contribute to the logical correctness of the rewriting, and therefore may not produce the optimal rewriting. In Section 7.2 we show how we can augment the resulting rewriting with cost-reducing views. Note that the approach we describe in Section 7.2

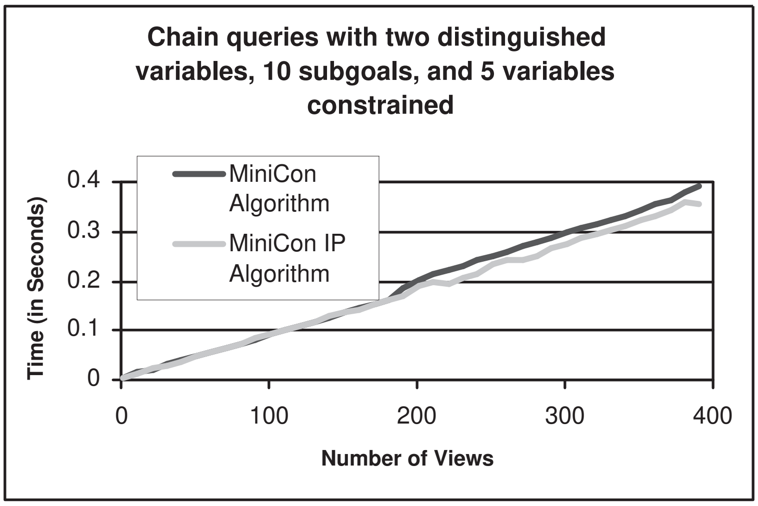

Fig. 9 Experiments with the MiniCon algorithm and comparison predicates. The query and view shapes are the same as in Figure 4. The graph shows that adding comparison predicates does not appreciably slow the MiniCon algorithm, and the additional views that can be pruned cause the algorithm to speed up overall.

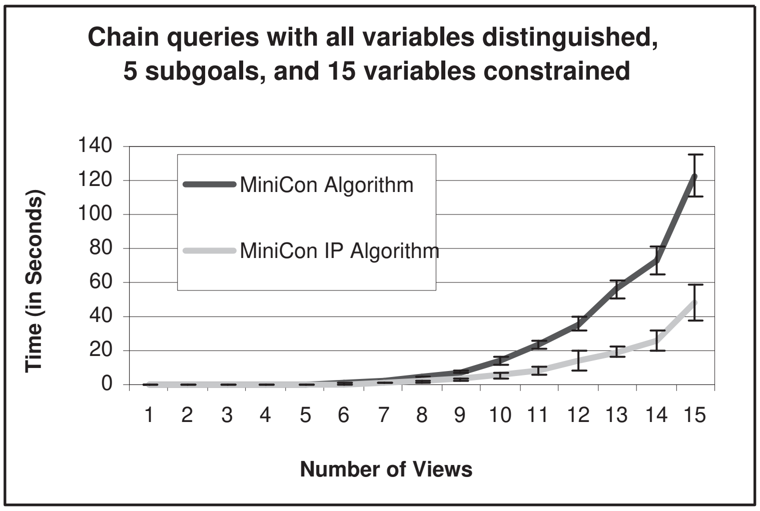

Fig. 10 Running times for the MiniCon algorithm and comparison predicates when all of the variables in the views are distinguished.

is inherently heuristic, and its goal is to avoid the exhaustive enumeration whose cost (according to [CH00]) would be prohibitive.

7.1 Modifying GMAP to consider MCDs

In the context of query optimization, we may have access to the database relations in addition to the views. In order to uniformly treat database relations and views, we assume that for every database relation E we define a view of the form $V_E(\bar{X})$:- $E(\bar{X})$, where $\bar{X}$ is a tuple of distinct variables. In our running example we will assume that we do not have access to the database relations.

We first briefly recall the principles underlying the GMAP algorithm [TSI96], and then describe how we modify it to exploit MCDs. The GMAP algorithm is a modification of System-R style bottom-up dynamic programming, except that the 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.

The GMAP algorithm begins by considering only views that can be used in a rewriting of the query (e.g., pruning views that refer to relations not mentioned in the query or do not apply necessary join predicates). The algorithm distinguishes between *partial query execution plans* of the query and *complete execution plans*, that provide an equivalent rewriting of the query using the views. The enumeration of the possible join trees terminates when there are no more unexplored partial plans.

The GMAP algorithm grows the plans by combining a partial plan (using all join methods) with an additional view. A partial plan P is pruned from further consideration if there is another plan P' such that (1) P' is cheaper than P, and (2) P' contributes the same or more 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 attributes that are needed further up the query tree.

Our algorithm precedes the join enumeration phase by the creation of MCDs, but it considers only a subset of the views that were considered in the data integration context.

In our discussion, we use the following notation to make use of the variable mappings used in procedure **combineMCDs** (Figure 3). Given a set of MCDs, $\mathcal{C} = C_1, \ldots, C_l$, we denote by $VtoQ_{\mathcal{C}}$, the set of atoms $V_{C_1}(EC(\Psi_1(\bar{Y_{C_1}})))$, ..., $V_{C_l}(EC(\Psi_l(\bar{Y_{C_l}})))$, as defined in procedure **combineMCDs**. Note that $VtoQ_{\mathcal{C}}$ effectively creates a set of atoms of the heads of the views in $\mathcal{C}$, such that the atoms use the variables of Q whenever possible. Hence, $VtoQ_{\mathcal{C}}$ makes explicit exactly which join predicates need to be applied between view atoms in the rewriting. So, in our example, if $\mathcal{C}_1$ denotes the set of MCDs created for the views in the rewriting Q4', then $VtoQ_{\mathcal{C}_1}$ is V12(x,y,z), V13(y,z,x), V14(x).

Given a query Q and a set of views, $V_1, \ldots, V_n$, our algorithm proceeds as follows:

1. We prune from further consideration any view V for which there does not exist a variable mapping ψ from the variables of V to the variables of Q, such that for every subgoal $g \in V$, $\psi(g)$ is a subgoal in Q. (Note that this condition is similar to that of a containment mapping [CM77], except that we do not require that ψ map the head of V to the head of Q.) Views that do not satisfy this condition cannot be part of an equivalent rewriting of Q using the views. In our example, if we also had a view defined as:

   V15(m,n) :- e1(m,n), e4(m)

   then we would prune V15 because it cannot be part of an equivalent rewriting of Q (the subgoal e4 cannot be mapped to Q).
2. With the views selected in the first step, we construct the MCDs as described in Section 4.1. In our example we would create MCDs for V12 and V13, but we do not create a MCD for V14 because it does not satisfy Property 1.
3. We now begin the bottom-up construction of *candidate solutions*. A candidate solution is a query execution plan over the views, which may either be a partial or complete plan for the query.[2]
   (a) For the base case, we start with plans that access a single view. Specifically, for every MCD C, we create the atom $VtoQ_{\{C\}}$. We then select the best access path to (the single atom in) $VtoQ_{\{C\}}$. In our example, we create the atoms V12(x,y,z) and V13(y,z,x).
   (b) With every candidate solution P, we associate a subset of the subgoals of the query, denoted by P_G. Intuitively, this set specifies which subgoals in the query are covered by the solution P, and this information is gleaned from the MCDs. In the base case, the set P_G associated with the candidate solution constructed for the view in MCD C is G_C.
   We combine a candidate solution P with a candidate solution (of size 1) P' only if the union of P_G and P'_G contains strictly more subgoals than either P_G or P'_G. Note that using the information in P_G and P'_G enables us to significantly prune the number of candidate solutions we consider compared to the GMAP algorithm. For example, suppose our example also included the view:
   V16(k,l) :- e1(k,l)
   and we had a partial solution, P, that included the single atom V12(x,y,z). Then, we would not combine P with V16(x,y) since V16 does not cover any more subgoals than V12. On the other hand, we would consider adding V13(y,z,x) to P since V13 covers e3, which is not covered by P, and P covers e1 which is not covered by V13.
   Given the views in P (whose corresponding MCDs are $\mathcal{C}$) and the view V in P', whose MCD is C_V, we compute $VtoQ_{\{\mathcal{C} \cup \{C_V\}\}}$. This tells us exactly which join predicates need to be applied between P and P' (specifically, whenever P and P' share a variable, a join predicate needs to be applied). We will try combining P and P' using every possible join method for every join predicate that needs to be applied.
   (c) As in the GMAP algorithm, we distinguish complete solutions, which correspond to equivalent rewritings of the query using the views, and partial solutions which can possibly be extended to complete solutions. Furthermore, as in GMAP, we compare every pair of candidate solutions P and P'. If P is both cheaper than P' and contributes as much or more to the query, then we prune P'. For example, if we had two candidate solutions $P1$, which consists of V12(x,y,z) and the candidate solution $P2$ which consists of V16(x,y), if $P1$ is cheaper than $P2$ we would prune $P1$ because $P1$

[2] We describe the algorithm for the case where we construct only left-linear trees, but the generalization to arbitrary bushy trees is straightforward.

is both cheaper than $P2$ and contributes more than $P1$. However, if $P2$ is cheaper than $P1$, we would prune neither candidate solution because $P1$ contributes more than $P2$.

(d) We terminate when there are no new combinations of partial solutions to be explored.

7.2 Adding cost-reducing views

As stated earlier, the algorithm in the previous section may not produce the cheapest plan because it only considers views that are needed for the logical correctness of the plan, and not cost-reducing views. (Note, however, that the algorithm will always find a plan if one exists even when we don't have access to the database relations). In this section we describe a heuristic approach to augmenting the plan produced in the previous section with cost-reducing views. Informally, we consider each cost-reducing view in turn, and try to place it in the places in the plan where it may have an effect. For example, consider the view V14(x) in our example. This view can only be useful if it is placed before the atom V12(x,y,z) (in order to reduce the number of values of x) or after the atom V12(x,y,z) (to reduce the size of the join with V13(y,z,x). However, V14(x) is useless if placed after V13(y,z,x).

We denote the plan produced by the algorithm in the previous section by P_{mg}. Recall that we are considering left-linear plans in our description. We create *cost-reducing view atoms* as follows:

(a) As in the previous section, we consider only views that can be part of an equivalent rewriting of the query using the views.

(b) We create MCDs for these views, except that we do *not* require the MCDs to satisfy Property 1. Denote the resulting MCDs by $C_1, \ldots, C_k$. In our example we would create MCDs for V12(x,y,z), V13(y,z,x), V14(x).

(c) Let the set of MCDs corresponding to the views in the plan P_{mg} be $\mathcal{C}_{mg}$. For every MCD C_j, $1 \leq j \leq k$, we compute $VtoQ_{\{\mathcal{C}_{mg} \cup \{C_j\}\}}$, and we denote by U_j the atom corresponding to C_j in $VtoQ_{\{\mathcal{C}_{mg} \cup \{C_j\}\}}$ (recall that $VtoQ_{\{\mathcal{C}_{mg} \cup \{C_j\}\}}$ computes an atom for every MCD). We will now try to insert the atoms $U_1, \ldots, U_k$ in the plan P_{mg}.

(d) Note that with every join operation in P_{mg} we can associate a set of variables, specifically, the variables that occur in the subtree of the join operator. The positions in P_{mg} that are *relevant* to the atom U_j are the join operators beginning with the first operator whose variable set includes any of the variables in U_j, and ending with the first join operator that includes all the variables in U_j. For every j, $1 \leq j \leq k$, we proceed as follows. We consider the cheapest plan P'_{mg}, that results from inserting U_j in one of the positions relevant to U_j. If a variable in U_j appears in the left-most leaf of the join tree, then we also consider the plan in which U_j is the left child of the first join operator in the plan. If P'_{mg} is cheaper than P_{mg}, we replace P_{mg} by the plan P'_{mg}.[3]

(e) We continue iterating through the cost-reducing view atoms until no change is made to the resulting plan.

In our example, we would consider placing the atom V14(x) as the first or second left-most leaf of the tree (i.e., either before V12(x,y,z) or immediately after it).

It is important to note that our algorithm may still not obtain the cheapest plan. The main reason is that we're beginning from the plan P_{mg}, and only modifying it locally, while the cheapest plan may actually be an augmentation of a plan that was found to be more expensive than P_{mg} in the cost-based join enumeration. It is possible to consider applying our algorithm to several plans from the cost-based join enumeration, rather than only to the cheapest one. However, in general, obtaining the cheapest plan may involve a prohibitively expensive search.

8 Related Work

Algorithms for rewriting queries using views are surveyed in [Hal01]. Most of the previous work on the problem focused on developing algorithms for the problem, rather that on studying their performance. In addition to the algorithms mentioned previously, algorithms have been developed for conjunctive queries with comparison predicates [YL87], queries and views with grouping and aggregation [GHQ95,SDJL96,CNS99,GRT99], queries over semi-structured data [PV99,CGLV99], and OQL queries [FRV96]. The problem of answering queries using views has been considered for schemas with functional and inclusion dependencies [DL97,Gry98], languages that query both data and schema [Mil98], and disjunctive views [AGK99]. Clearly, each of the above extensions to the basic problem represents an opportunity for a possible extension of the MiniCon algorithm. Two works [AD98,GGM99] examine the complexity of finding all the possible answers from a set of view extensions. They show that if the views are assumed to be complete, then finding the maximal set of answers is NP-hard in the size of the data. Hence, finding a maximally-contained rewriting may not be possible if we consider

[3] For ease of exposition, we chose to describe a relatively conservative condition on the positions in which we can insert a cost-reducing view atom. Several further optimizations are possible the most obvious of which is that we would not insert a cost-reducing view atom in a plan *after* all the joins performed in the view have already been performed in the plan.

query languages with polynomial data complexity. Mitra [Mit99] developed a rewriting algorithm that also captures the intuition of Property 1, and thus would likely lead to better performance than the bucket algorithm and the inverse-rules algorithm. He also considered an optimization similar to our method for removing redundant view subgoals.

Several works discussed extensions to query optimizers that try to make use of materialized views in query processing [TSI96,CKPS95,ALU01,BDD+98,PDST00,ZCL+00]. In some cases, they modified the System-R style join enumeration component [TSI96,CKPS95], and in others they incorporated view rewritings into the rewrite phase of the optimizer [ZCL+00,PDST00]. These works showed that considering the presence of materialized views did not negatively impact the performance of the optimizer. However, in these works the number of views tended to be relatively small. In [ALU01] the authors consider the problem of finding the most efficient rewriting of the query using a set of views, in the context of query optimization. The paper considers three specific cost models, and for each describes an algorithm that produces the cheapest plan. The algorithm we describe in Section 7 is independent of a particular cost model, and can incorporate the models described in [ALU01]. In addition, our algorithm can also handle cost models that consider relation sizes, special orders and specific join implementations, as done in traditional query optimizers. In [PDST00], the authors consider a more general setting where they use a constraint language to describe views, physical structures and standard types of constraints.

A commercial implementation of answering queries using views is described for Oracle 8i in [BDD+98]. Their 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 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.

9 Conclusions

This paper makes two important contributions. First, we present a new algorithm for answering queries using views, and second, we present the first experimental evaluation of such algorithms. We began by analyzing the two existing algorithms, the bucket algorithm and the inverse-rules algorithm, and found that they have significant limitations. We developed the MiniCon algorithm, a novel algorithm for answering queries using views, and showed that it scales gracefully and outperforms both existing algorithms. As a result of our work, we have established that answering queries using views can be done efficiently for large-scale problems. Finally, we described an extension of our algorithm to handle comparison predicates, and showed that the techniques underlying the MiniCon algorithm are also useful for the context of cost-based query optimization using views.

We close by briefly discussing another important extension of the MiniCon algorithm. In data integration applications, where views represent data sources, we often have limited access patterns to the data. For example, if Amazon.com has a relation $Book(title, price)$, we cannot ask for all tuples in the relation. Instead, we need to provide a value for the title in order to get a price. The problem of answering queries using views in this context has been considered in [RSU95,KW96,DL97,LKG99]. In [RSU95] it is shown that when we consider equivalent rewritings, the rewriting may be *longer* than the query. In [DL97] it is shown that if we are looking for the maximally-contained rewriting, it may have to be a recursive datalog program over the views.

The MiniCon algorithm can be adapted in a straightforward fashion to the presence of binding patterns. Specifically, we can follow the same strategy of [DL97], where inverse rules were augmented by *domain rules*. In our case we produce the rewriting by the MiniCon algorithm by first ignoring the binding pattern limitations. Then we add domain rules, and augment the rewriting by adding domain subgoals where necessary.

A Proof of Correctness of the MiniCon algorithm

A.1 Preliminaries

We consider conjunctive queries and views without built-in predicates. We assume the query has the form

$$Q(\bar{X}) : -e_1(\bar{X_1}), \ldots, e_n(\bar{X_n})$$

Without loss of generality we assume that no variable appears in more than one view, and the variables used in the views are disjoint from those in the query. Furthermore, we assume that the heads of the views and the query do not contain multiple occurrences of any variable. We apply variable mappings to tuples and to atoms with the obvious meaning, i.e., $\varphi(\bar{X}) = (\varphi(x_1), \varphi(x_2), \ldots, \varphi(x_n))$ where $\bar{X} = (x_1, \ldots, x_n)$.

Recall that a maximally-contained rewriting is, in general, a union of conjunctive rewritings. A conjunctive rewriting has the form

$Q'(\bar{Y})$:- $V_1(\bar{Y_1}), V_2(\bar{Y_2}), \ldots, \mathcal{V}_k(\bar{Y_k})$

Note that for any $i \neq j$ it is possible that $V_i = V_j$.

Given a conjunctive rewriting Q', the *expansion* of Q', denoted by Q'' is the query in which the view atoms are replaced by their definitions (i.e., they are unfolded). Note that when expanding the view definitions we need to create fresh variables for the existential variables in the views. We assume we have a function $f^i(x)$ that returns the ith fresh copy of a variable x. For a given subgoal $g_i \in Q'$, we denote by $exp(i)$ the set of subgoals in Q'' obtained by expanding the definition of V_i.

Given two head homomorphisms h_1 and h_2 over the variables of a view V, we say that h_2 is more restrictive than h_1 if whenever $h_1(x) = h_1(y)$, then $h_2(x) = h_2(y)$.

Recall that the MiniCon algorithm produces conjunctive rewritings of the form

$$Q'(EC(\bar{X})) \text{ :- } V_{C_1}(EC(\Psi_1(\bar{Y_{C_1}}))), \ldots, V_{C_m}(EC(\Psi_m(\bar{Y_{C_m}})))$$

Where for a variable x in Q, $EC(x)$ denotes the representative variable of the set to which x belongs. EC is defined to be the identity on any variable that is not in Q.

Remark 4 The following property will be used in the soundness proof. Suppose that a subgoal $g \in Q$ is in G_i, i.e., $\varphi_i(g) \in h_i(V_i)$. The expansion Q'' will contain an atom $\tau(g)$, where, for a variable x:

- $\tau(x) = EC(x)$ if $\varphi_i(x)$ is a head variable in $h_i(V_i)$, and
- $\tau(x) = f^i(x)$ otherwise.

A.2 Proof of Soundness

We need to show that every conjunctive rewriting Q' that is obtained by the MiniCon algorithm is contained in Q. To show soundness, we show that there is a containment mapping Υ, from Q to Q''.

We define an intermediate Υ_i for $i = 0, \ldots, k$ by induction as follows. The containment mapping Υ will be defined to be Υ_k.

U1 For all x where $x \in Vars(Q)$ and $EC(x) \in Vars(Q'')$, $\Upsilon_0(x) = EC(x)$.

U2 Υ_i is an extension of Υ_{i-1}, defined as follows: for all x in the $Domain(\varphi_i)$, if $x \notin Domain(\Upsilon_{i-1})$ then $\Upsilon_i(x) = f^i(EC(\varphi_i(x)))$.

Now we show that Υ is a containment mapping.

- Mapping of the head: we need to show that $\Upsilon(\bar{X}) = EC(\bar{X})$. Because of U1, it suffices to show that for every variable in $x \in \bar{X}$, $EC(x)$ appears in Q''. By Property 1, clause C1, we know whenever x is in the domain of φ and is a head variable in Q, φ maps x to a head variable in $h(V)$. By Property 2, clause D1, we know that given a MCD set, all the head variables in Q are in the domain of *some* MCD in the set. From the definition of Ψ_i, we know that $\bar{X}$ is a subset of the union of the ranges of the Ψ_i's, and hence, $EC(x)$ is in Q'' for every $x \in \bar{X}$.
- Mapping of a subgoal g. We need to show that Q'' includes $\Upsilon(g)$. By Remark 4 we know that Q'' includes $\tau(g)$. It suffices to show that $\Upsilon(g) = \tau(g)$, which follows immediately from the definition of Υ.

A.3 Completeness

Let P be a maximally-contained rewriting of Q using $\mathcal{V}$, and let R be the rewriting produced by the MiniCon algorithm. The MiniCon algorithm is complete if $R \sqsupseteq P$. Since both R and P are unions of conjunctive queries, it suffices to show that if p' is a conjunctive rewriting in P, then there exists a conjunctive rewriting r' in R, such that $r' \sqsupseteq p'$ [SY81].

Since p' is part of a maximally-contained rewriting of Q, there exists a containment mapping θ from Q to the expansion p'' of p' [CM77]. We will use θ to show that there exists a set of MCDs that are created by the MiniCon algorithm such that when the MCDs are combined, we obtain a conjunctive rewriting r' that contains p'.

We proceed as follows:

- For each subgoal $g_i \in p'$, we define G_i to be the set of subgoals $g \in Q$, such that $\theta(g) \in exp(i)$ (i.e., G_i includes the set of subgoals in Q that are mapped to the expansion of g_i in p''). Note that for $i \neq j$, the sets G_i and G_j are disjoint.
- We denote by θ_i the restriction of the containment mapping θ to the variables appearing in G_i.
- The mapping θ_i is a mapping from $Vars(G_i)$ to $Vars(exp(g_i))$. However, it can be written as a composition of two mappings, one from $Vars(G_i)$ to $h_i(Vars(V_i))$ (where h_i is a head homomorphism on V_i), and another from $h_i(Vars(V_i))$ to $Vars(exp(g_i))$. Formally, there exists a mapping $\tau_i : Vars(G_i) \rightarrow h_i(Vars(V_i))$ and a renaming α of the variables in $h_i(Vars(V_i))$, such that $\theta_i(x) = \alpha(\tau_i(h_i(x)))$ for every variable $x \in G_i$.

  We choose h_i to be the least restrictive head homomorphism on $Vars(V_i)$ for which τ_i and α exist. Note that since we chose h_i to be the least restrictive head homomorphism, then *any* MCD created by the MiniCon algorithm for V_i would at least as restrictive as τ_i (hence, τ_i depends only on Q and the view V_i, and not on how V_i is used in the rewriting p').
- We show that we now have all the components of a MCD, which we will denote by C_i:
  - h_i is a head homomorphism on $Vars(V_i)$,
  - $h_i(V_i(\bar{A}))$ is the result of applying h_i to the head variables $\bar{A}$ of V_i.
  - τ_i is a partial mapping from $Vars(Q)$ to $h_i(Vars(V_i))$, and
  - G_i is a set of subgoals in Q that are covered by τ_i.

Furthermore, the MCD C_i satisfies the conditions of Property 1 which are enforced by the MiniCon algorithm:

C1. For any head variable x of Q, $\tau_i(x)$ is a head variable of $h_i(V_i)$, because $\theta_i(x)$ is a head variable of p''.

C2. It follows from the fact that θ_i is a restriction of a containment mapping from Q to p'', that if $\tau_i(x)$ is an existential variable in $h_i(V_i)$, then for every subgoal $g_1 \in Q$ that includes x (1) all the variables in g_1 are in the domain of τ_i, and (2) $\tau_i(g_1) \in h_i(V_i)$.

In addition, note that $C_1, \ldots, C_k$ satisfy Property 2, which is the condition that the MiniCon algorithm checks before it combines a set of MCDs:

D1. $G_1 \cup \ldots \cup G_k = Subgoals(Q)$ because θ is a containment mapping from Q to p'', and

D2. for every $i \neq j$, $G_i \cap G_j = \emptyset$ because of the way we constructed the G_i's.

– The only difference between the MCD C_i and a MCD created by the MiniCon algorithm is that τ_i may not be the *minimal* mapping necessary to satisfy Property 1. However, this is easy to fix by simply decomposing the MCD C_i into a set of MCDs that satisfy Property 1 exactly and contain only minimal mappings for τ_i and minimal sets of subgoals in their fourth component. Note that even after decomposing the MCDs, the G_i's are still disjoint subsets of subgoals in Q, and hence Property 2 is still satisfied.

– At this point we have shown that we have a set of MCDs $C_1, \ldots, C_l$, that satisfy Properties 1 and 2. Furthermore, each of the mappings τ_i in the MCDs less restrictive than θ in the following sense: for any variables x, y, if $\tau_i(x) = \tau_i(y)$ then $\theta(x) = \theta(y)$.

As a result, when procedure **combineMCDs** creates the function EC, it will have the property that $EC(x) = EC(y)$ only if $\theta(x) = \theta(y)$. Consequently, the conjunctive rewriting r' that is produced when $C_1, \ldots, C_l$ are combined will have the same property: whenever the same variable appears in two argument positions in r', those two argument positions will have the same variable in p'. Hence, there is a containment mapping from r' to p', and therefore $p' \sqsubseteq r'$.

References

[AD98] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *PODS*, 1998.

[AGK99] Foto Afrati, M. Gergatsoulis, and Th. Kavalieros. Answering queries using materialized views with disjunctions. In *ICDT*, 1999.

[ALU01] Foto N. Afrati, Chen Li, and Jeffrey D. Ullman. Generating efficient plans for queries using views. In *SIGMOD*, 2001.

[BDD+98] Randall Bello, Karl Dias, Alan Downing, James Feenan, Jim Finnerty, William Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in oracle. In *VLDB*, 1998.

[CGLV99] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Rewriting of regular expressions and regular path queries. In *PODS*, 1999.

[CH00] Rada Cirkova and Alon Y. Halevy. On the computational complexity of the view selection problem. Submitted for publication, 2000.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *ICDE*, 1995.

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

[CNS99] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *PODS*, 1999.

[DG97a] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.

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

[DL97] Oliver M. Duschka and Alon Y. Levy. Recursive plans for information gathering. In *IJCAI*, 1997.

[FRV96] Daniela Florescu, Louiqa Raschid, and Patrick Valduriez. A methodology for query reformulation in CIS using semantic knowledge. *Int. Journal of Intelligent & Cooperative Information Systems*, 5(4), 1996.

[FW97] M. Friedman and D. Weld. Efficient execution of information gathering plans. In *IJCAI*, 1997.

[GGM99] Gösta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *ICDT*, 1999.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, 1995.

[GRT99] Stephane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. Querying aggregate data. In *PODS*, 1999.

[Gry98] Jarek Gryz. Query folding with inclusion dependencies. In *ICDE*, 1998.

[Hal01] Alon Y. Halevy. Answering queries using views: A survey. *To appear in the VLDB Journal*, 2001.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

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

[KW96] Chung T. Kwok and Daniel S. Weld. Planning to gather information. In *AAAI*, 1996.

[LKG99] Eric Lambrecht, Subbarao Kambhampati, and Senthil Gnanaprakasam. Optimizing recursive information gathering plans. In *IJCAI*, 1999.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *PODS*, 1995.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

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

[Mil98] R. J. Miller. Using schematically heterogeneous structures. In *SIGMOD*, 1998.

[Mit99] Prasenjit Mitra. An algorithm for answering queries efficiently using views. Stanford University Technical Report, Stanford, CA, USA, 1999.

[PDST00] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A chase too far? In *SIGMOD*, 2000.

[PV99] Yannis Papakonstantinou and Vasilis Vassalos. Query rewriting for semi-structured data. In *SIGMOD*, 1999.

[Qia96] Xiaolei Qian. Query folding. In *ICDE*, 1996.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[SDJL96] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering SQL queries using materialized views. In *VLDB*, 1996.

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

[TS97] Dimitri Theodoratos and Timos Sellis. Data warehouse design. In *VLDB*, 1997.

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

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

[Ull97] Jeffrey D. Ullman. Information integration using logical views. In *ICDT*, Delphi, Greece, 1997.

[YL87] H. Z. Yang and P. A. Larson. Query transformation for PSJ-queries. In *VLDB*, 1987.

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