A Scalable Algorithm for Answering Queries Using Views[*]

Rachel Pottinger
University of Washington
rap@cs.washington.edu

Alon Levy
University of Washington
alon@cs.washington.edu

Abstract

The problem of answering queries using views is to find efficient methods of answering a query using a set of previously materialized views over the database, rather than accessing the database relations. The problem has 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. Finally, we describe an extension of the MiniCon algorithm to handle comparison predicates, and show its performance experimentally.

[*]Thanks to Daniela Florescu, Marc Friedman, 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 Grant #IIS-9978567, a NSF Graduate Research Fellowship, and a Lucent Technologies GRPW Grant

Proceedings of the 26th VLDB Conference,
Cairo, Egypt, 2000.

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 [20]: query optimization [6, 21, 36], maintenance of physical data independence [35, 33, 27], data integration [22, 9, 18, 19], and data warehouse and web-site design [16, 32]. 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 [35, 33, 6], 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. The key 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 [21]. Previous work has mainly considered two algorithms for this purpose.

The bucket algorithm, developed as part of the Information Manifold System [22], 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 primarily in the InfoMaster System [29, 8], 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, (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.

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 [15, 31, 7, 13, 26, 4, 10, 14]. 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 presents the problem formally, 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 discusses related work and Section 8 concludes.

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:

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

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 2.1 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 $Q_2 \sqsubseteq Q_1$.

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 [5].

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). A conjunctive query is said to be *redundant* if it is possible to remove some of its subgoals and obtain an equivalent query.

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 2.1 (**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 2.2 Consider the query from Example 2.1 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 the identity mapping except for x1 $\rightarrow$ y, y1 $\rightarrow$ 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 [22, 18, 12, 19]. 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 2.3 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 w.r.t. 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 2.2 (**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 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 subset. □

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*). When the queries and the views are conjunctive and do not contain comparison predicates, it follows from [21] that we need only consider conjunctive rewritings Q' that have at most the number of subgoals in the query Q.

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. We are not considering here the problem of finding the rewriting that yields the *cheapest* query execution plan over the views, which would be the main concern when using algorithms for answering queries using views for query optimization and maintenance of physical data independence. In the concluding section we revisit this issue. 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 [21] 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 less 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 [22]. 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.

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. 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 $\rightarrow$ b, y $\rightarrow$ 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, 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) 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)

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)
```

However, the variable b does not appear in the head of V4, and therefore, 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 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 [29, 8] 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 [8], 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 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.

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 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. The proof of correctness of the algorithm is omitted for lack of space, but is described in the full version of this paper [28].

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 4.1 (**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 $h_C(V)$ and 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.

The construction of the MCDs is based on the following observation on the properties of query rewritings.

```
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 $\varphi$, s.t. $\varphi(g) = h(v)$.
        If $h$ and $\varphi$ exist, then add to $\mathcal{C}$ any new MCD $C$
            that can be constructed where:
            (a) $\varphi_C$ (resp. $h_C$) is an extension of $\varphi$ (resp. $h$),
            (b) $G_C$ is the minimal subset of subgoals of $Q$ such that
                $G_C$, $\varphi_C$ and $h_C$ satisfy Property 1, and
            (c) it is not possible to extend $\varphi$ and $h$ to an MCD that
                covers fewer subgoals than $G_C$.
    Return $\mathcal{C}$
```

Figure 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).

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

Figure 2: MCDs formed as part of our example

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 Same-Topic(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.

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.

Remark 2 *(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 MCD 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:

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 2, clause D2 would not hold.

```
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 $i \neq j$, $G_{C_i} \cap G_{C_j} = \emptyset$ then
            Define a mapping $\Psi_i$ 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
                $\Psi_i$ 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 $Answer$.
```

Figure 3: Phase 2: combining the MCDs.

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 [21], 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.

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

The following theorem summarizes the properties of the MiniCon algorithm:

Theorem 4.1 *Given a conjunctive query Q and conjunctive views $\mathcal{V}$, both without comparison predicates, the MiniCon algorithm produces the union of conjunctive queries that is the 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 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 [25].

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). 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(a), 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 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 5 (and this factor is independent of query and view size). In fact, the MiniCon algorithm can handle more than 1000 views with 10 subgoals each in less than one second.

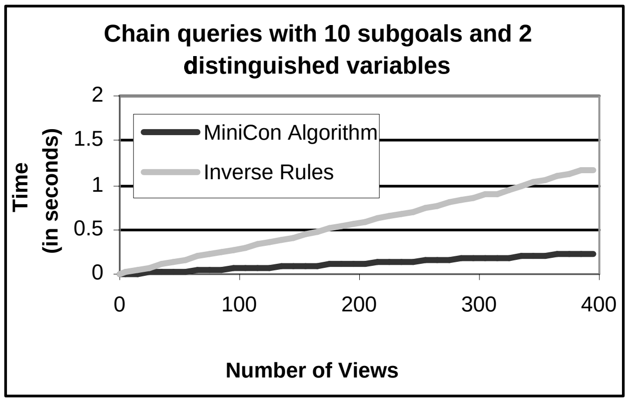
The difference in the performance between the inverse-rules algorithm and the MiniCon algorithm in this context and in others is due to the second phases of the algorithms. In this phase, the inverse-rules algorithm is searching for a unification of the subgoals of the query with heads of inverse rules. 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 4(b), the views are shorter than the query (of lengths 2, 3 and 4, while the query has 12 subgoals). In this case the MiniCon algorithm stills scales linearly while the inverse-rules algorithm grows faster. For example, for 90 views, the MiniCon algorithm runs 3 times faster than the inverse-rules algorithm, and for 180 views it runs 6 times faster.

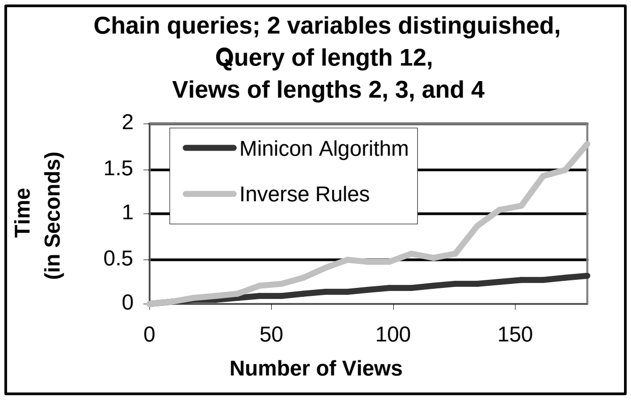
Finally, though not shown here, we also considered another case in which all the variables in the views are distinguished. In this case, there are many rewritings (often more than 1000), and hence the performance of the algorithms is limited because of the sheer number of rewritings. Since virtually all combinations produce contained rewritings, any complete algorithm is forced to form an exponential number of rewritings. The MiniCon algorithm still performs better than the inverse-rules algorithm by anywhere between 10% better and a factor of 2, but with queries and views with 5 subgoals, the algorithms take on the order of 10 seconds for 10 views. It should be emphasized that the difference in performance between the MiniCon algorithm and the inverse-rules algorithm in this case is only due to the smaller search space being considered by the MiniCon algorithm.

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 5(a) 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



(a)



(b)

Figure 4: Experimental results for chain queries. The graph on the left considers 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 5. In the right graph, the views are of lengths 2, 3 and 4, and the query has 12 subgoals. In this case the MiniCon algorithm still scales linearly, while the inverse-rules algorithm does not.

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 20 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 5(b). 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 4 for 20 views, and by a factor of 6 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 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 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.

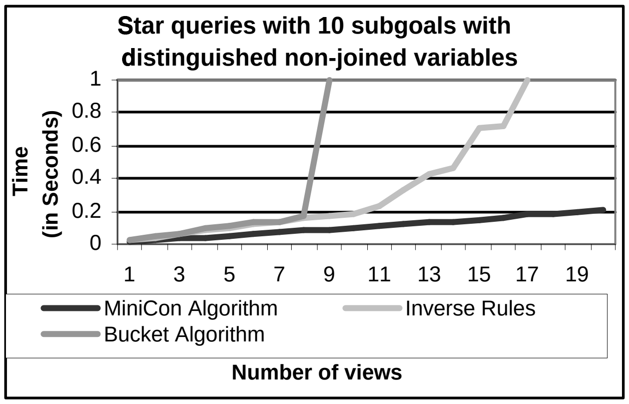
Query type	Distinguished	# of subgoals	# of views
Chain	All	3	45
Chain	All	12	3
Chain	Two	5	9225
Chain	Two	99	115
Star	Non Joined	5	12235
Star	Non Joined	99	35
Star	Joined	10	4520
Star	Joined	99	75

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

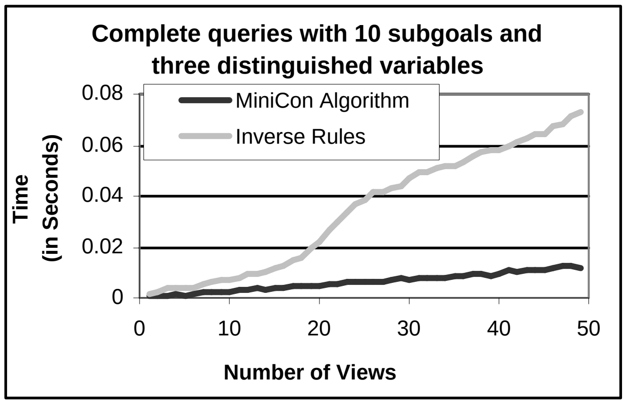
Second, the experiments showed that the bucket algorithm performed much worse than the other two algorithms in all cases. More interesting was the comparison between the MiniCon algorithm and the inverse-rules algorithm. In all cases the MiniCon algorithm outperformed the inverse-rules algorithm, though 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 an exponential number of rewritings.

6 Comparison predicates

The effect of comparison predicates on the problem of answering queries using views is quite subtle. If the views



(a)



(b)

Figure 5: The left figure shows the running times for star queries, where the distinguished variables in the views are those not participating in the joins. The graph on the right shows running times for complete queries with a similar setting. In both cases the MiniCon algorithm significantly outperforms the inverse-rules algorithm.

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 [1] 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 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 [34].

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 6.1 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
V7(a,s1) :- inSIGMOD(a), cites(a,b), year(a,s1),
            year(b,s2), s2 ≤ 1983
V8(a,s1) :- inSIGMOD(a), cites(a,b), year(a,s1),
            year(b,s2), s2 ≤ 1987
```

Our algorithm would first consider V7 with the mapping {x → a, y → b, r1 → s1, r2 → s2}. In this case, the subgoal r2 ≤ 1985 is satisfied by the view, but r1 ≥ 1990 is not. However, since s1 is a distinguished variable in V7, the algorithm can create the rewriting:

```
Q2'(x) :- V7(x,r1), r1 ≥ 1990
```

When the algorithm considers a similar variable mapping to V8, it will notice that the constraint on r2 is not satisfied, and since it is mapped to an existential variable in V8, no MCD is created. □

Example 6.2 The following example provides an intuition for which rewritings our extended algorithm will not discover. Consider the following query and view:

Q(u) :- e(u,v), u $\leq$ v
V1(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 Q.

Q'(u) :- V1(u)

In order to show that the query contains the rewriting, we need to consider two different containment mappings, depending on whether $a \leq b$ or $a > b$ [17]. In each of these mappings, the subgoal $e(u, v)$ is 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. □

Figure 6 shows sample experiments that we ran on the extended algorithm in the case of chain queries. In the experiments, we added to the queries and views a number of comparison subgoals of the form $x < c$ or $x > c$.

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 6(b) 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 6(a), the number of rewritings is very small, and the extra overhead of processing the comparison predicates causes slow down of a factor of 4. This factor can be decreased with further optimizations of our comparison predicate code that we did not explore.

7 Related Work

Algorithms for rewriting queries using views are surveyed in [20]. 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 [35], queries and views with grouping and aggregation [15, 31, 7, 13], OQL queries [11], and queries over semi-structured data [26, 4]. The problem of answering queries using views has been considered for schemas with functional and inclusion dependencies [10, 14], languages that query both data and schema [23], and disjunctive views [2]. Clearly, each of the above extensions to the basic problem represents an opportunity for a possible extension of the MiniCon algorithm. Mitra [24] 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.

Several works discussed extensions to query optimizers that try to make use of materialized views in query processing [33, 6, 3, 27, 36]. In some cases, they modified the System-R style join enumeration component [33, 6], and in others they incorporated view rewritings into the rewrite phase of the optimizer [36, 27]. 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 [27], the authors consider a more general setting where they use a constraint language to describe views, physical structures and standard types of constraints. Unlike our algorithm, the above works are designed to produce a *single* conjunctive rewriting that is equivalent to the query and has the least cost, whereas we search for the maximally-contained rewriting.

8 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. As we show in the extended version of the paper, the MiniCon algorithm can also be extended in a straightforward fashion to deal with access-pattern limitations [30] in the same spirit of [10].

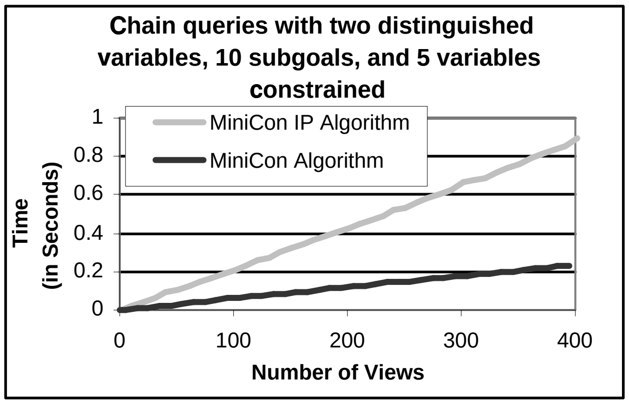
An interesting direction of future research is to extend the MiniCon algorithm to the context of using materialized views for query optimization and to consider bag semantics. In this context, we are interested in the *cheapest* rewriting of the query. Conceivably, it is possible as in [33, 6] to modify the second phase of the MiniCon algorithm such that it combines the MCDs in a bottom-up dynamic programming style, and hence saves only the cheapest rewriting. However, for the algorithm to guarantee finding the cheapest rewriting, we now need to consider rewritings that contain logically redundant subgoals.

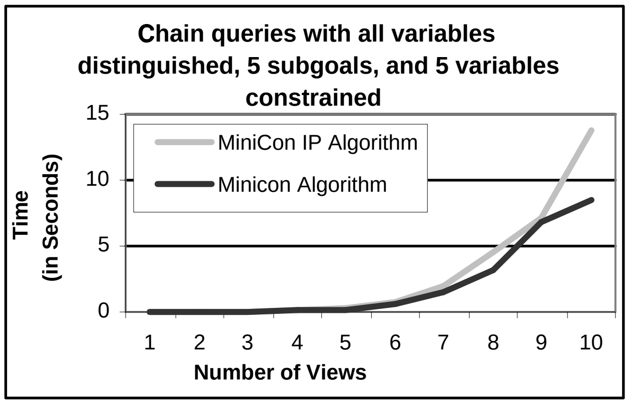
Example 8.1 Suppose we have the following query and views:

Q(x,y) :- e1(x,z), e2(z,y)
V1(x,y) :- e1(x,y)
V2(z,y) :- e2(z,y)
V3(x) :- e1(x,z), e2(z,y)

If the join of e1 and e2 is very selective, the rewriting of the query that will yield the cheapest query execution plan may be:

q'(x,y) :- v3(x), v1(x,y), v2(z,y).





(a) (b)

Figure 6: Experiments with the MiniCon algorithm and comparison predicates. The left graph shows the same case as in Figure 4(a) and shows that adding comparison predicates only slows down the running time by a factor of 4. The graph on the right shows the running times when all of the variables in the views are distinguished.

However, the MiniCon algorithm would not create a MCD that includes V3, because it would violate Property 1, and would hence miss this rewriting. □

Hence, to extend the MiniCon algorithm to this context we need to establish a bound on the size of rewritings that need to be considered, and to relax the definition of MCDs.

References

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

[2] F. Afrati, M. Gergatsoulis, and T. Kavalieros. Answering queries using materialized views with disjunctions. In *ICDT*, 1999.

[3] R. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *VLDB*, 1998.

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

[5] A. Chandra and P. 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.

[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, 1995.

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

[8] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.

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

[10] O. M. Duschka and A. Y. Levy. Recursive plans for information gathering. In *IJCAI*, 1997.

[11] D. Florescu, L. Raschid, and P. Valduriez. A methodology for query reformulation in cis using semantic knowledge. *Int. Journal of Intelligent & Cooperative Information Systems, special issue on Formal Methods in Cooperative Information Systems*, 5(4), 1996.

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

[13] S. Grumbach, M. Rafanelli, and L. Tininini. Querying aggregate data. In *PODS*, 1999.

[14] J. Gryz. Query folding with inclusion dependencies. In *ICDE*, 1998.

[15] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *VLDB*, 1995.

[16] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.

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

[18] C. T. Kwok and D. S. Weld. Planning to gather information. In *AAAI*, 1996.

[19] E. Lambrecht, S. Kambhampati, and S. Gnanaprakasam. Optimizing recursive information gathering plans. In *IJCAI*, pages 1204–1210, 1999.

[20] A. Y. Levy. Answering queries using views: A survey, 2000. Manuscript available from `www.cs.washington.edu/homes/alon/views.ps`.

[21] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.

[22] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

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

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

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

[26] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD*, 1999.

[27] L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far? In *SIGMOD*, 2000.

[28] R. Pottinger and A. Levy. A scalable algorithm for answering queries using views. Technical Report UW-CSE-2000-06-01, University of Washington Department of Computer Science and Engineering, 2000.

[29] X. Qian. Query folding. In *ICDE*, pages 48–55, New Orleans, LA, 1996.

[30] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, 1995.

[31] D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering SQL queries using materialized views. In *VLDB*, 1996.

[32] D. Theodoratos and T. Sellis. Data warehouse design. In *VLDB*, 1997.

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

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

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

[36] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *SIGMOD*, 2000.