Using Views to Answer Queries with Contraction and Relaxation Options *†

Sibel Adalı
Department of Computer Science
University of Maryland
sibel@cs.umd.edu

Xiaolei Qian
Computer Science Laboratory
SRI International
qian@csl.sri.com

Abstract

In this paper, we address the problem of deciding if and how a user query can be answered using a set of pre-defined views. The views may correspond to materialized views, cached results or queries that can be posed to external information sources. Even though some views may have overlapping data definitions, they may correspond to data from different information systems. We develop an algorithm that takes as input a conjunctive user query and a set of conjunctive view definitions to generate as output all possible foldings of it, i.e ways of rewriting the query using these views. Our algorithm provides the following options (when both queries and views may contain built-in predicates): (1) foldings that provide the same answers as the original query, i.e. equivalent foldings, (2) foldings that provide a subset of the answers to the original query, i.e. contracted foldings, (3) foldings as in 1 and 2 above for relaxations of the query. Similarly, foldings may use views partially or they may consist of views only. In addition, we explore the completeness properties of our algorithm and prove that it generates all possible foldings for different cases.

1 Introduction

In this paper, we address the problem of deciding if and how a user query can be answered using a set of pre-defined views. This problem, which we will refer to as query folding, arises naturally in many different contexts. The views used in this process may refer to materialized views, cached results or queries that can be posed to external information sources. In any case, views may provide the answers faster than a database or they may be the only way to access some information requested by the user as in the case of non-proprietary data sources. Most of the research in this area [5, 22, 13] has concentrated on generating equivalent queries, i.e. queries that provide the same answers as the user query, but access views instead of the requested tables. It may not always be possible to find such an equivalent way to answer a user query since users are not necessarily aware of the details of the system. Hence, some views that contain relevant information may only be referred to after certain modifications to the user query. There are two types of modifications possible on a query: (1) *Contraction* refers to the process of adding extra conditions to the query, hence making it stronger. The resulting query cannot return more answers than the original query. (2) *Relaxation* refers to removing some conditions from the query, hence making it weaker. The resulting query cannot return fewer answers than the original query. Both of these

†Note to reviewers: This paper contains an appendix containing the formal proofs of the results stated in the paper.

*This work was supported in part by U.S. Department of Defense Advanced Research Projects Agency and U.S. Air Force Rome Laboratory under contract F30602-94-C-0198, in part by the National Science Foundation under grant ECS-94-22688, in part by the Army Research Office under grant DAAH-04-95-10174, and in part by ARPA/Rome Labs contract Nr. F30602-93-C-0241 (Order Nr. A716).

modifications allow the user to access a set of sources that are relevant to his/her query. Contractions were addressed in [14]. Relaxations in the context of query folding have not been addressed yet.

In this paper, we present an algorithm that attacks the query folding problem in its most general form. It generates all possible foldings when both views and queries may or may not contain built-in predicates. The foldings may be complete (using only view predicates) or partial. The foldings may or may not contract the original query or the relaxations of the original query. Finally, the algorithm can be used to check whether one query is contained in the other. All these options are available with small modifications to the algorithm or its input. In addition, we explore the completeness properties of the algorithm and show that it computes all possible foldings for different cases.

The complexity of query folding and query containment have been studied extensively. Query containment is a special case of query folding. To determine whether a query is contained in another query, we could try to determine instead whether the second query can be answered using a view defined by the first query. It is shown to be undecidable for relational queries [1] and Datalog queries [18]. The problem is NP-complete for conjunctive queries [4], unions of conjunctive queries [17], and conjunctive queries under functional dependencies [2, 3]. Query containment is Π_2^p-complete for conjunctive queries with built-in predicates [21], and NP-complete for the subclasses of such queries called the left and right semiinterval queries [11]. The complexity of query containment is also studied for conjunctive queries under functional and inclusion dependencies [9] and more general constraints [7], and under the bag semantics [6, 8]. Query folding is recently shown to be NP-complete for conjunctive queries and views [13], and for conjunctive queries and views with binding patterns [16]. The problem is Π_2^p-complete for conjunctive queries and views with built-in predicates [13].

Our work differs from the previous work in three significant ways. First of all, we put a special emphasis on completeness of our algorithm which is more general than any of the existing algorithms both in its scope and the range of its output. We guarantee generating all possible foldings for different cases. Furthermore, we show the source of differences in the complexity of query foldings due to the existence of built-in predicates as discussed above and discuss necessary extensions to our algorithm for each case. We also show the utility and completeness of our algorithm for checking containment. Folding algorithms for conjunctive queries and views with built-in predicates have been developed in [5, 19, 22]. The algorithms employed all use exhaustive search strategies that are exponential-time in complexity, and do not guarantee to find all foldings. The algorithms in [19, 22] compute only foldings that depend solely on the views. When such complete foldings do not exist, partial foldings that are contained in the original query or depend partially on the views could be very useful in practice, especially in the distributed environment.

The other important difference is the emphasis we put on generating only executable rewritings of the query efficiently (a rewriting is a folding only when it is executable.) Even though it may be possible to match the user query with some view definitions, the resulting rewriting may make it impossible to check some conditions present in the original query. This is referred to as the safety condition in [5]. Most of the algorithms proposed in the past (e.g., [14]) first generate rewritings and then select the executable ones, generally with the help of containment checks which introduces an extra component of complexity to the problem. In this paper, we use a special marking technique which allows us to prune many rewritings that will not lead to foldings (i.e. will not be executable). This technique was first introduced in [15] when no built-in predicates were present. In this paper, we generalize this technique to address novel problems posed by the existence of built-in predicates and the generation of relaxed foldings.

Finally, we address the possibility of relaxing a user query to fit to the available information. This is very important in the presence of many information sources containing related but different pieces of information. Even though the meaning of the user query may change considerably depending on the relaxations being made, it may still be desirable to perform relaxation in many cases since some sources may contain valuable information that cannot be accessed in any other way. The user may have requested too detailed information which is not available due to the differences in data models or for security reasons. We show that our algorithm (with slight modifications) can provide all the information necessary to derive the required relaxations to find foldings that make use of a certain view. The modified algorithm still continues to find all (un-relaxed) foldings. In addition, it generates rewritings marked in a specific way that makes it possible to find relaxed foldings without re-executing the algorithm for the relaxed form of the query.

This work is part of a joint effort between two large scale projects, the HERMES project at the University of Maryland and the Query Mediation Project at the Computer Science Laboratory at SRI International. Currently, there is also research on related issues as a part of the TSIMMIS project at Stanford University. We believe our results can also be applied to the framework of that project as well.

2 Basics

Suppose we have a set of relational views defined according to a common data model. The problem of query folding is then to decide, given a user query, how to find answers for it using these views.

Queries and views are expressible as conjunctive queries. Predicates are divided into four disjoint groups: base predicates, view predicates, query predicates, and built-in predicates (BIPs) $>, \geq, <, \leq, \neq$. An atomic formula containing a base predicate is called a (positive) base literal. Similar for view, query, and built-in literals (BILs). A conjunctive query has the form

$$Q :- p_1, p_2, \ldots, p_n, C_1, \ldots, C_k.$$

where Q is an atomic formula called the head, $p_1, \ldots, p_n$ are base literals collectively called the body, and $C_1, \ldots, C_k$ are BILs collectively called the constraint. The variables that appear in the head are *distinguished*. A view is a conjunctive query whose head is a view literal, and a query is one whose head is a query literal. We assume that all views use different view predicates, hence the view predicate is defined completely by its body.

The value of a conjunctive query in a database is the value of its head obtained by evaluating its body in that database. A conjunctive query Q is contained in another conjunctive query Q', if the value of Q is a subset of the value of Q' in every database.

A *partial rewriting* of query Q using a set of views $\mathcal{V}$ is a conjunctive query Q' whose head is a query literal, such that $Q' \subseteq Q$ and the body of Q' contains one or more view literals defined in $\mathcal{V}$. A *complete rewriting* of Q using $\mathcal{V}$ is a partial rewriting of Q using $\mathcal{V}$ whose body contains view literals and BILs only. Hence a partial rewriting may contain base literals whereas a complete rewriting may not contain any base literals. A partial (complete) rewriting is called a *partial (complete) folding* whenever it can be evaluated. It may not be possible to evaluate a rewriting because it contains dangling variables. Let Q' be a rewriting of Q. We use unfold(Q') to denote the query obtained by replacing every view literal in Q' by its corresponding body.

Let Q_1 and Q_2 be two conjunctive queries. A *containment mapping* h from Q_2 to Q_1 is a mapping of variables in Q_2 to variables in Q_1 such that

1. if P is a base literal in Q_2 then $h(P)$ is a base literal in Q_1,
2. h makes the arguments in the head of Q_2 same as the arguments in the head of Q_1, and
3. if C is a BIL in Q_2, then $h(C)$ is implied by the BILs in Q_1.

It is well-known [20, pp. 889-890] that having a containment mapping from Q_2 to Q_1 is sufficient for $Q_1 \subseteq Q_2$. It is also necessary when Q_2 does not contain BIPs. If Q_2 contains BIPs, then (1) and (2) above are still necessary but (3) is not.

We use the terms *rewriting* and *folding* to mean contractions of the user query. It may happen that a given rewriting (or folding) does not contract and the resulting expression is equivalent to the original query. Note that, this is a more general use of the term rewriting than some previous work. A relaxation of a query can be obtained by removing base literals, distinguished variables and weakening the constraint part. A relaxed folding of a query Q is a folding for a relaxation of Q, hence it may still contract the relaxed query. We will examine this issue in further detail in section 5. We finish this section with an example that illustrates some of the terms defined above.

Example 2.1 *Suppose the user asks for the Prescription for all the patients older than 30 who were diagnosed to have "glaucoma." The query is given below:*

Query(Prescription) :– *Patient(Name, Age, PatientID), Age* $\geq$ *30,*
Diagnosis(PatientID, `glaucoma`*, Pres_Id),*
Prescription(Pres_ID, Physician_Id, Prescription, Note_Id).

We want to use the following view to rewrite this query:

View(Diagnosis, Prescription) :– *Patient(Name, Age, PatientID),*
Diagnosis(PatientID, Diagnosis, Pres_Id),
Prescription(Pres_ID, Physician_Id, Prescription, Note_Id).

The resulting rewriting is given as follows:
Query(Prescription) :– *View(*`glaucoma`*, Prescription), Age* $\geq$ *30.*
Obviously, this is not acceptable since the Age attribute has disappeared in the rewriting and there is no way to run this query in its current form. Hence the above expression is not a folding even if it is a rewriting.

Suppose the view above also had the condition "Age > 40" in its definition. In this case, there is no need to check the condition "Age $\geq$ *30" in the query since all the tuples from the view will definitely satisfy it and the condition can be removed. But, the rewriting using the modified view is not equivalent to the original query, rather it contracts the query resulting in possibly a smaller table as the answer (skipping all the tuples corresponding to people aged between 30 and 39).*

Finally, we observe that we could have relaxed the original query by removing the Age constraint and then the original view could be used to derive a relaxed folding.

3 A Query Rewriting Algorithm

In this section we will describe an algorithm for generating rewritings of a query given a set of view definitions as described in the previous section. The generator can be thought of as a very specific resource locator for views. As we will see in Sections 4 and 5 the output of the generator contains all the information necessary for finding the foldings of the original query (Section 4) and also the relaxed foldings of the query (Section 5).

First, we will pre-process the views to generate rewrite rules that will be used to derive the rewritings of a user query. The skolemization process that will be described in section 3.1 was first introduced in [15]. Here we generalize the same concept for views with non-empty constraints. The skolemization contributes greatly to the efficiency of the algorithm by helping prune many rewritings that will never lead to foldings in much earlier steps. This is due to the careful marking of non-distinguished variables in the views that avoids them from taking arbitrary instantiations. As we will see these marks will be propagated to the queries by the application of rewrite rules.

3.1 Generating Rewrite Rules

Consider a view of the form:

$$V :\!- p_1, \ldots, p_n, C_1, \ldots, C_k.$$

where $p_1, \ldots, p_n$ is the body and $C_1, \ldots, C_k$ is the constraint. Let $X_1, \ldots, X_l$ be the distinguished variables, and $Y_1, \ldots, Y_m$ be the rest of variables. We skolemize the view by substituting for all Y_j a different skolem term of the form $f_j(X_1, \ldots, X_l)$. Let θ be the substitution that skolemizes all such Y, i.e. $\theta = \{f_1(X_1, \ldots, X_l)/Y_1, \ldots, f_m(X_1, \ldots, X_l)/Y_m\}$. Then, above view can be expressed equivalently as:

$$V :\!- (p_1, \ldots, p_n, C_1, \ldots, C_k)\theta.$$

Unless mentioned otherwise, we'll use the skolemized version of views in the rest of the paper. Since all the view predicates are different, the head is completely defined by its body, hence the view implies the formula:

$$(\forall X_1, \ldots, X_l)(V \leftrightarrow (p_1 \wedge \ldots \wedge p_n \wedge C_1 \wedge \ldots \wedge C_k)\theta).$$

From this formula, we derive n *rewrite rules* below.

$$\begin{array}{lcl} p_1\theta & :\!- & V, \texttt{context}(C_1\theta, \ldots, C_k\theta) \\ & \vdots & \\ p_n\theta & :\!- & V, \texttt{context}(C_1\theta, \ldots, C_k\theta). \end{array}$$

Notice that, for a rewrite rule, the head is a base literal, the body contains a view literal, and every variable in the head also appears in the body. In addition to the view literal, we introduced a context predicate in the body of the rewrite rules. We assume that `context`(C) is always true and there is no need to satisfy it. We also observe that unfold(V) implies $C_1\theta, \ldots, C_k\theta$. We will use context for carrying the BILs that are true in the body of a rewriting, implicitly, even if they do not appear in the rewriting explicitly. Hence context can be used to determine if rewritings are not satisfiable since their unfoldings contain unsatisfiable constraints.

The application of these rules to a query may result in a rewriting that is not a folding. The reason for this is that the rewriting may contain skolem terms and these terms cannot be interpreted by the query processor. A rewriting then is a folding when it does not contain any skolem terms except in the context predicate.

Example 3.1 *Suppose we have the following views in the system originally:*

$$\begin{aligned} View1(X1, Y1) &:- P1(X1, Y1), X1 \geq 5, Y1 \leq 5. \\ View2(Y2, Z2) &:- P2(Y2, Z2), Y2 \geq 10. \\ View3(X3, Z3) &:- P1(X3, Y3), P2(Y3, Z3), Y3 \leq 15. \end{aligned}$$

After skolemization, we replace View3 with the view:

$$View3(X3, Z3) \;:-\; P1(X3, f_1(X3, Z3)), P2(f_1(X3, Z3), Z3), f_1(X3, Z3) \leq 15.$$

View1 and View2 remain the same. We generate the following rewrite rules from these views:

$$\begin{aligned} P1(X1, Y1) &:- View1(X1, Y1), \texttt{context}(X1 \geq 5, Y1 \leq 5). \\ P2(Y2, Z2) &:- View2(Y2, Z2), \texttt{context}(Y2 \geq 10). \\ P1(X3, f_1(X3, Z3)) &:- View3(X3, Z3), \texttt{context}(f_1(X3, Z3) \leq 15). \\ P2(f_1(X3, Z3), Z3) &:- View3(X3, Z3), \texttt{context}(f_1(X3, Z3) \leq 15). \end{aligned}$$

We assume that rewrite rules are standardized apart among themselves so that two rewrite rules have no variables in common.

3.2 Application of Rewrite Rules

In this section, we will illustrate the use of the rewrite rules as well as the context predicates they contain. A unique characteristic of query folding is the generation of contractions of the user query. The reason for this is that foldings may use only a part of the definitions of the views. When a view is chosen for a folding, all the conditions in its definition become a part of the folding. These conditions are already satisfied, hence they do not need to be placed in the folding. But they may be needed implicitly since they may help prune foldings that will not produce any tuples. Let us demonstrate this point with an example.

Example 3.2 *Suppose we are given the following query and the rewrite rules given in example 3.1.*

$Query(X, Z) :- P1(X, Y), P2(Y, Z), Y \leq 20, Z \geq 5.$

We first rewrite the above query using the first rewrite rule in example 3.1 to get:

$Query(X1, Z) :- View1(X1, Y1), P2(Y1, Z), Y1 \leq 20, Z \geq 5, \texttt{context}(X1 \geq 5, Y1 \leq 5).$

The first observation we make is that there is no need to satisfy the expression $Y1 \leq 20$ in the body of the literal anymore since all the tuples we get from this rewriting will have Y values less than 5 according to the context, *and hence subsuming the literal $Y1 \leq 20$. We remove this literal from the body and the resulting expression is a rewriting of Query.*

Suppose we want to continue and find a rewriting that makes better use of view literals. We now apply the second rewriting rule in example 3.1 to this rewriting and get the expression (after merging the contents of the context predicates):

$Query(X, Z) :- View1(X, Y), View2(Y, Z), Z \geq 5, \texttt{context}(X \geq 5, Y \leq 5, Y \geq 10).$

By examining the context predicate, we realize that the BILs in the context predicate are unsatisfiable, hence the tuples obtained from View1 and View2 will never have matching Y values and the join will return an empty table. Thus we do not accept this expression as a rewriting.

As stated earlier, context predicate always evaluates to true. But this does not mean that query rewritings may contain arbitrary contexts. We specified exactly what is contained in the rewrite rules and that the contents of the context is implied by the view literal that appears next to it. We assume that there is always one context predicate in a query rewriting. Hence if any rewriting contains more than one context predicate, we can merge them by merging their contents. Suppose $\texttt{context}(C_1, ..., C_{k_1})$ and $\texttt{context}(C'_1, ..., C'_{k_2})$ are two contexts. Then, $\texttt{context}(C_1, ..., C_{k_1}) \wedge \texttt{context}(C'_1, ..., C'_{k_2}) \equiv \texttt{context}(C_1, ..., C_{k_1}, C'_1, ..., C'_{k_2})$. This means that whenever more than one rewrite rule is applied to a query, we simply merge the contexts that they are carrying. From now on we will assume that there is only one context predicate in every rewriting.

We have seen in this example how to use the rewrite rules and their contexts to generate satisfiable rewritings. In the next section we will introduce an algorithm for finding all possible rewritings of a query by applying these rewrite rules exhaustively.

3.3 An Algorithm for Rewriting Generation

We now present an exhaustive algorithm for finding all the possible rewritings of a query by applying the rewriting rules in all possible ways. The generator algorithm is going to construct a set called rewritings(Q) that contains all the possible rewritings of a query Q using a set of rewriting rules $\mathcal{F}$. The algorithm consists of three steps, initialization, label merging, and rewriting construction.

3.3.1 Generating Labels

A *label* is a table consisting of three types of columns:

1. a list of attribute columns,
2. a rewriting column, and
3. a context column.

We denote the attribute columns of a label L by attr(L). An attribute column can be designated as distinguished. A tuple in a label is *proper* if it does not contain skolem terms in distinguished or rewriting columns.

The first step in the algorithm is to compute a label for every base literal in Q. Given base literal p in Q, its label L_p has as attribute columns the list of variables in p, namely var(p). An attribute column is distinguished if it corresponds to a distinguished variable in Q.

For every base literal p in Q, and every folding rule $F \in \mathcal{F}$ of the form head(F) :$-$body(F), $\texttt{context}(F)$ such that head(F) unifies with p with an mgu σ, we construct a tuple t as follows. The contents of the attribute columns of t are var$(p)\sigma$, the rewriting column of t contains body$(F)\sigma$, and the context column of t is $\texttt{context}(F)\sigma$. If t is proper, then we include it in L_p. In case that we are seeking partial foldings of Q that may contain the base literal p, we add a *default* tuple (var$(p), p, _$) to L_p.

Example 3.3 *Below we give the labels of the base literals of the query: $Query(X, Z)$:$-P1(X, Y), P2(Y, Z), Y \leq 20, Z \geq 5$.*

The rewrite rules used to generate these labels were given in example 3.1. Note that the first tuple in every label is the default tuple as explained above. If we did not have this tuple, we would only generate complete foldings, if there exists any.

attribute columns		*rewriting column*	*context column*
X	Y		
X	Y	$P1(X, Y)$	
$X1$	$Y1$	$View1(X1, Y1)$	$\texttt{context}(X1 \geq 5, Y1 \leq 5)$
$X3$	$f_1(X3, Z3)$	$View3(X3, Z3)$	$\texttt{context}(f_1(X3, Z3) \leq 15)$

attribute columns		*rewriting column*	*context column*
Y	Z		
Y	Z	$P2(Y, Z)$	
$Y2$	$Z2$	$View2(Y2, Z2)$	$\texttt{context}(Y2 \geq 10)$
$f_1(X3, Z3)$	$Z3$	$View3(X3, Z3)$	$\texttt{context}(f_1(X3, Z3) \leq 15)$

The purpose of the first step is to locate all views that contain information that may be relevant to the user query where relevance is determined by the existence of a base literal both in the user query and in the body of a view. We see the first use of skolem terms in rewrite rules in this step. Suppose query Q and a view V both contain a base literal P, but P has a distinguished variable X for an attribute A in Q but A holds a non-distinguished variable in V. In this case, V cannot be used to rewrite Q through P since the variable X will disappear in the rewriting. In accordance with this fact, the rewrite rule with P in the head and V in the body will contain a skolem term for A and the label generated using this rule will not be proper.

The initial labels only rewrite one base literal of the query. In the next step, we will explore the possibility of combining the initial labels iteratively to find rewritings for the complete query. The skolem terms will be used again to eliminate the rewritings that cannot lead to foldings.

3.3.2 Label Merging – Unification Join

The second step is to merge the labels of base literals in Q by applying a binary *u-join* operator to labels in arbitrary order. U-join stands for unification-join and the unification is performed on the attribute columns of tuples as explained below.

Let L_p and L'_p be two labels associated with base literals p and p' in Q. Also let $\mathcal{Y} = \text{attr}(L_p) \cap \text{attr}(L'_p)$ be the common attribute columns in L_p and L'_p, and $\mathcal{Z} = \text{attr}(L'_p) - \text{attr}(L_p)$ be the additional attribute columns from L'_p. The *u-join* of L_p and L'_p, denoted as $L_p \overset{u}{\bowtie} L'_p$, is a label whose attribute columns consist of those of both L_p and L'_p given by $(\text{attr}(L_p), \mathcal{Z})$. The tuples in $L_p \overset{u}{\bowtie} L'_p$ are computed as follows:

Let (t, r, c) be a tuple in L_p and (t', r', c') be a tuple in L'_p. We denote with $t[\mathcal{Y}]$ the projection of t on the variables in $\mathcal{Y}$. If $t[\mathcal{Y}]$ unifies with $t'[\mathcal{Y}]$ with an mgu σ, then we place the following tuple in $L_p \overset{u}{\bowtie} L'_p$ if it is proper:

$$((t, t'[\mathcal{Z}])\sigma,\ (r, r')\sigma,\ (c \wedge c')\sigma).$$

Example 3.4 *We now give the label generated by u-joining the two labels given in example 3.3. Note that the columns for the variables X and Z below are distinguished.*

attribute columns			*rewriting column*	*context column*
X	Y	Z		
X	Y	Z	$P1(X, Y), P2(Y, Z)$	
X	Y	$Z2$	$P1(X, Y), View2(Y, Z2)$	$\texttt{context}(Y \geq 10)$
$X1$	$Y1$	Z	$View1(X1, Y1), P2(Y1, Z)$	$\texttt{context}(X1 \geq 5, Y1 \leq 5)$
$X1$	$Y1$	$Z2$	$View1(X1, Y1), View2(Y1, Z2)$	$\texttt{context}(X1 \geq 5, Y1 \leq 5, Y1 \geq 10)$
$X3$	$f_1(X3, Z3)$	$Z3$	$View3(X3, Z3)$	$\texttt{context}(f_1(X3, Z3) \leq 15)$

Note that using unification to combine labels leads to another improvement in the efficiency of query folding algorithms. Let's illustrate this point with a simple example: Suppose we have $Q(X) : -P1(X, Y), P2(X, Y)$. and the view $V(X, Y) : -P1(X, Y), P2(X, Y)$. Without unification, we will end up with the folding $Q(X1) : -V(X1, Y1), V(X2, Y2)$ (since rewrite rules are standardized apart) which is contained in the more efficient folding given by $Q(X) : -V(X, Y)$. This is a significant gain over approaches that use simple cartesian products instead of unification as in [14]. This improvement in efficiency is independent of the gain resulting from the use of skolem terms in rewrite rules.

3.3.3 Rewriting Construction

Let L be the label obtained in the previous step. Notice that L has as attribute columns all the variables in Q. The final step is to construct the set rewritings(Q) from L as follows.

Let $(t, r, \texttt{context}(C))$ be a tuple in L and θ be the substitution that replaces every variable in Q by the corresponding entry in t. If r contains at least one view predicate, and $C \wedge \text{constraint}(Q)\theta$ is satisfiable, then we place the rewriting

$$\text{head}(Q)\theta \quad :- \quad r, C', \texttt{context}(C)$$

in rewritings(Q), where C' is the set of BILs in constraint$(Q)\theta$ that are not implied by C. Notice that skolem terms could only appear in the constraint of a rewriting of Q, since every tuple in label L is proper. If however constraint$(Q)\theta \wedge C$ is inconsistent, then the above rewriting is not placed in rewritings(Q).

The purpose of the simplification described above has a dual purpose. The first one is to remove any unsatisfiable rewritings. The second one is to remove the BILs from rewritings that are already satisfied. This does not only make the execution of the rewritings faster (since their constraints are simplified), but it also enables us to find more foldings since some of the BILs containing skolem terms may disappear from the rewritings as a result of this operation.

One point to note is that it is possible to attach the relevant parts of constraint(Q) to the context column of every tuple in the initialization step. Then this information can be carried over to the final label by successive joins. The advantage of this would be to prune unsatisfiable rewritings due to unsatisfiable BILs at intermediate steps of the algorithm.

Example 3.5 *Finally, we construct the set rewritings(Q) from the label given in example 3.4. Recall that the original query was Query(X, Z) $:- P1(X, Y), P2(Y, Z), Y \leq 20, Z \geq 5$. We will apply the last step in slow motion, we first attach the constraint part of the query to the rewritings in the labels and then observe what happens at the simplification step. Hence before the last simplifications, we obtain the following expressions:*

(1) Query(X,Z) :– P1(X,Y), P2(Y,Z), Y $\leq$ 20, Z $\geq$ 5.
(2) Query(X,Z2) :– P1(X,Y), View2(Y,Z2), Y$\leq$ 20, Z2 $\geq$ 5, `context`*(Y $\geq$ 10)*
(3) Query(X1,Z) :– View1(X1,Y1), P2(Y1,Z), Y1 $\leq$ 20, Z $\geq$ 5, `context`*(X1 $\geq$ 5, Y1 $\leq$ 5)*
(4) Query(X1,Z2) :– View1(X1,Y1), View2(Y1,Z2), Y1 $\leq$ 20,Z2 $\geq$ 5, `context`*(X1 $\geq$ 5, Y1 $\leq$ 5, Y1$\geq$ 10)*
(5) Query(X3,Z3) :– View3(X3,Z3), f_1(X3,Z3) $\leq$ 20, Z3 $\geq$ 5, `context`*(f_1(X3,Z3) $\leq$ 15)*

We notice that the first expression is not a rewriting since it does not contain any view literals. In fact, it is identical to the original query. Expressions 2 and 3 on the other hand, are partial rewritings and expressions 4 and 5 are complete rewritings. If the labels given in example 3.3 had not contained the first (default) tuples we would not have obtained the first three expressions. Rewritings 2 and 3 are not only partial rewritings but are also foldings. We see that the answers we will get from rewriting 2 will only give us those that correspond to Y values between 10 and 20 because of the implications posed by the context predicate. Similarly, rewriting 3 will only give us X values greater than or equal to 5 and Y values less than or equal to 5. Both of them are satisfiable and they are placed in rewritings(Q) after simplification. The only change after simplification in them is that the literal $Y \leq 20$ in (3) is removed since it is implied by the context.

If we look closely to rewriting 4 we see that $Y1 \leq 20, Y1 \leq 5, Y1 \geq 10$ is inconsistent, then this rewriting will not be placed in rewritings(Q) since it is unsatisfiable.

Finally, rewriting 5 contains a BIL with a skolem term. But we see that $f_1(X3, Z3) \leq 20$ is implied by $f_1(X3, Z3) \leq 15$ that appears in the context. Hence, we can remove $f_1(X3, Z3) \leq 20$ and add the rewriting:

Query$(X3, Z3) :- View3(X3, Z3), Z3 \geq 5,$ `context`$(f_1(X3, Z3) \leq 15)$

into rewritings(Q). This way, the simplification step enables us to remove the unsatisfiable rewritings and find additional foldings where the BILs involving skolem terms are already satisfied.

Below, we state the soundness of this algorithm, in the sense that any expression generated by this algorithm is a rewriting, and hence is contained in the query. (The proof of this theorem and the other results stated in the paper can be found in the Appendix.)

Theorem 3.1 If $Q_r \in$ rewritings(Q) then there is a containment mapping from Q to unfold(Q_r).

Note that the above theorem establishes the soundness regardless of whether rewritings obtained at the end of the algorithm are foldings or not (since their constraints might contain skolem terms, which are not evaluatable). Hence anything we get by applying the rewrite rules gives us a query stronger than the original one.

Then this algorithm can be used directly to find the foldings of a query. Whenever a rewriting in rewritings(Q) does not contain any skolem terms, then it is a folding. We will examine in section 4 the completeness properties of this algorithm and discuss the cases when additional operations must be performed to find all the foldings using the contents of rewritings(Q).

4 Collecting Foldings from the Rewritings

In this section, we will examine in detail the completeness properties of our algorithm for finding foldings. We will see if and when we need to perform additional operations to guarantee finding all possible foldings of a query. To this end, we will construct a new set called foldings(Q), which is the set of foldings of query Q, using the rewritings generated by the algorithm of Section 3.3. We have seen that the rewritings in rewritings(Q) may contain skolem terms and hence making them impossible to evaluate. We will investigate in this section how these rewritings may be converted to foldings when possible and how to construct the set foldings(Q).

The construction of foldings(Q) from rewritings(Q) is divided into three stages.

1. We collect foldings that are rewritings (in other words, expressions in rewritings(Q) that are already foldings.) This stage is complete when (1) Q does not contain BIPs but views do, or (2) Q contains BIPs but views do not and containment mappings exist.

2. We collect foldings that are rewritings plus additional constraints. This stage is complete when containment mappings from Q exist.

3. We collect foldings that are unions of rewritings. This stage is complete for testing if a query is a folding of Q.

The three stages are presented in the three subsections that follow.

4.1 Foldings That Are Rewritings

We have seen in the previous sections that the rewritings generated by the algorithm may not be evaluatable because of the skolem terms they contain. Note that these skolem terms could only appear in the constraint of a rewriting, and the constraint of a rewriting is an instantiation of the constraint of the original query. When the query does not contain any BIPs (the views might still do), then by the construction of rewritings(Q) there cannot be any skolem terms in the rewritings except within the context predicate which is not evaluated. Hence, by the soundness of the rewriting generation algorithm, all the rewritings generated by the algorithm are foldings. In other words, foldings(Q) $\equiv$ rewritings(Q). This is stated formally below.

Lemma 4.1 Suppose Q_f is a folding of query Q. If there is a containment mapping from Q to unfold(Q_f), then there is a $Q_r \in$ rewritings(Q) and a substitution θ such that body(Q_r)θ = body(Q_f) and constraint(unfold(Q_r))θ is implied by constraint(unfold(Q_f)).

Theorem 4.2 (Completeness when Queries Contain no Built-In Predicates) If Q_f is a folding of Q and Q contains no built-in predicates, then there is a $Q_r \in$ rewritings(Q) such that Q_r is a folding of Q and $Q_f \subseteq Q_r$.

4.2 Foldings That Are Rewritings Plus Constraints

Recall in the first stage, we placed in foldings(Q) all the rewritings in rewritings(Q) that do not contain any skolem terms. In the second stage, we consider the remaining rewritings in rewritings(Q) and try to remove the BILs with skolem terms by introducing additional constraints. We will place in foldings(Q) all those rewritings that do not contain any skolem terms at the end of this stage. To see how the BILs containing skolem terms can be removed, let us consider the following example:

Example 4.1 *Suppose we have the following query:*

$Q(X,Y) :- E(X,Y),\ P(U,V),\ U \leq V.$

and we rewrite it using the following view:

$V(X,Y) :- E(X,Y),\ P(U,V),\ U \leq X,\ Y \leq V.$

If we apply the rewriting generation algorithm, we get the following rewriting of Q*:*

$Q(X,Y) :- V(X,Y), f_1(X,Y) \leq f_2(X,Y), \texttt{context}(f_1(X,Y) \leq X, Y \leq f_2(X,Y)).$

Since the rewriting still contains skolem terms, this is not a folding. But we observe that the following is a folding of query Q *using view* V*:*

$Q(X,Y) :- V(X,Y), X \leq Y.$

The question still remains whether it is possible to obtain the BIL given by $X \leq Y$ *from the rewriting above. Note that if we had this BIL in the rewriting, then we would be able to get rid of the BIL given by* $f_1(X,Y) \leq f_2(X,Y)$ *since it is implied by the contents of the context predicate together with* $X \leq Y$*. Hence, we need an operator that finds the extra literal(s) to remove BILs with skolem terms, should such literal(s) exist. We will call this operator* subtract*.*

Definition 4.1 (Subtract operation) Let C, C' be two sets of BILs containing skolem terms. Then, subtract(C', C) computes a set of sets of BILs, such that $C^- \in$ subtract(C', C) if:

1. C^- does not contain skolem terms.

2. $C^-, C' \models C$.

3. There does not exist a C'' satisfying the two conditions above such that $C^- \models C''$.

   Hence, the C^- is a weakest such set.

If there does not exist such C^-, then subtract(C', C) returns an empty set.

The intended use of the subtract operator is then clear. If a rewriting Q' in rewritings(Q) has the form $Q :- Body, C, \texttt{context}(C')$ and C contains skolem terms, then for every C^- in subtract(C', C) we add the following folding to foldings(Q): $Q :- Body, C^-, \texttt{context}(C')$.

We first want to point out that the subtract operation is decidable. To compute subtract(C', C), we only need to consider BILs that involve the variables and constants in C' and C. This is exponential in the worst case. Then we need to check if any of these satisfy the second condition and finally eliminate the ones implied by the others. In reality, the subtract operation can be executed more efficiently by making several observations.

Sketch Algorithm for Subtract: Suppose we are solving subtract(C', C). First observation we make is that we only need to consider the relations between those distinguished variables that appear in relation with a skolem term. Similarly, we only need to consider the BILs in the constraint of the body that involve skolem terms, let us call this part of the constraint C with C_{sk} and the rest with C_d. Now we sketch an algorithm for subtract that is completely oriented towards removing C_{sk}. Assume all the constraints we are dealing with are minimal.

First, we will compute the transitive closure C_{tr} of the constraint C_d, C' leaving the skolem terms uninterpreted. Then we try to find the possible partial orders on the distinguished variables, imposed by the BILs (the term partial order here refers to the ordering of a subset of the distinguished variables.) We will examine each BIL in C_{sk} and try to find different possible partial orders needed to remove this literal using the constraint C_{tr}.

As a simple example of this, suppose we have $F_1 \leq F_2$ in C_{sk} where F_1 and F_2 are both skolem terms. Then we find all distinguished variables in C_{tr} of the form either $F_1 \leq X$ and $Y \leq F_2$ and for each such pair we create a partial order $X \leq Y$ for this BIL only. If the above BIL was $F_1 < F_2$, then we had to create the partial order $X < Y$ using the above BILs. Similarly, BILs in C_{tr} containing $\neq$ may contribute to removing BILs of the form $F_1 \neq F_2$.

At the end, we will create a set of partial orders where each is sufficient to remove all the BILs containing skolem terms. These partial orders should not contradict with C_{tr}. We then generate the weakest preconditions C for each partial order and add the constraint C, C_d to the output of subtract (after removing redundant constraints.)

Completeness using Subtract: Below we give two completeness results. The first result states that given a folding Q' of a query Q such that there exists a containment mapping from Q to the unfolding of Q', we can construct it using subtract. Then, the next lemma that follows from this theorem states that, if we had only views that did not contain any BIPs, then rewritings(Q) contains all the foldings Q_f of Q, such that there is a containment mapping from Q to unfold(Q_f). Note that we do not need subtract for this case.

Theorem 4.3 (Completeness when Containment Mappings Exist) If Q_f is a folding of Q and there is a containment mapping from Q to unfold(Q_f), then there is a $Q_r \in$ rewritings(Q) and a Q'_f, where Q'_f has the same head and body as Q_r and constraint$(Q'_f) \in$ subtract$(\texttt{context}(Q_r),$ constraint$(Q_r))$, such that $Q_f \subseteq Q'_f$.

Corollary 4.4 (Completeness when Views Contain No BIPs and Containment Mappings Exist) If views in $\mathcal{V}$ contain no BIPs, Q_f is a folding of Q, and there is a containment mapping from Q to unfold(Q_f), then there is a $Q_r \in$ rewritings(Q) such that Q_r is a folding of Q and $Q_f \subseteq Q_r$.

4.3 Foldings That Are Unions of Rewritings

As we have seen earlier, the results we have presented up to now show that we can find those foldings of a query to which containment mappings from the query exist. Hence the relation $Q_1 \subseteq Q_2$ for these queries can be determined by examining the expressions syntactically. In section 2, we have seen that $Q_1 \subseteq Q_2$ can still hold even if the constraint part of Q_1 is not implied by the constraint part of Q_2 through some containment mapping. We will see in this section, what happens, when the relations $Q_1 \subseteq Q_2$ is due to implicit constraints on the domain of attributes. This corresponds to the third stage of the construction of foldings(Q), i.e. that of taking unions of the rewritings of Q. This is the first *deterministic* approach to our knowledge that deals with this special case. We first illustrate this case with the following example.

Example 4.2 *Suppose we are given the query:*

$$Q(X) :- E(X), R(U,V), U \leq V.$$

and we have the view:

$$V(X) :- E(X), R(U_1,V_1), R(V_1,U_1).$$

At a first glance, it might appear as if it is not possible to fold this query using the given view. However, if we assume a total order on the underlying domain of relation R, then we conclude that for any given U and V, either $U \leq V$ or $V \leq U$ must be true. In the first case, we set $U = U_1$ and $V = V_1$ in the view, then the condition in the query is true. Similarly for the other case, we just set $U = V_1$ and $V = U_1$ and the condition still holds.

If we use the rewriting generation algorithm of Section 3.3, we get two rewritings of this query:

$$\begin{aligned} Q_1(X) &:- & V(X), f_1(X) \leq f_2(X). \\ Q_2(X) &:- & V(X), f_2(X) \leq f_1(X). \end{aligned}$$

Obviously, none of the above qualifies as a folding because of the skolem terms. But, we know that both of these are contained in the original query, i.e. $Q_1 \subseteq Q$ and $Q_2 \subseteq Q$ by the soundness of rewriting generation. Now, suppose we take the union of these two rewritings. We have that $Q_3(X) \equiv Q_1(X) \vee Q_2(X) \subseteq Q$ where Q_3 is given as follows: (after appropriate variable substitutions) :

$$\begin{aligned} Q_3(X) &:- & (V(X) \wedge f_1(X) \leq f_2(X)) \vee (V(X) \wedge f_2(X) \leq f_1(X)). \\ &:- & V(X) \wedge ((f_1(X) \leq f_2(X) \vee f_2(X) \leq f_1(X))). \text{ (by De Morgan's law)} \\ &:- & V(X) \text{ (by the total order on elements of R).} \end{aligned}$$

The resulting query is indeed a folding of Q.

The reason for such a split was illustrated by Levy, Mendelzon, Sagiv and Srivastava in [13]. It was shown that if there is no containment mapping from a query Q to another query Q', then for $Q' \subseteq Q$ to hold the following must be true: there must exist queries $Q'_1, \ldots, Q'_m$ such that (1) $Q'_1, \ldots, Q'_m$ and Q' only differ from each other in their constraint part, (2) there exists a containment mapping from Q to each Q'_i for $1 \leq i \leq m$, (3) Q' is equivalent to $Q'_1 \vee \ldots \vee Q'_m$. In other words, Q'_i contains extra constraints that can be removed through union.

We use the above result and generalize as follows: to find a folding of a query for such a case, we must find a set of rewritings with the same body, and the disjunction of the constraints of these rewritings together with the contents of the context predicate must give us a single conjunction without skolem terms. Alternatively, when the result is a disjunction and/or it contains additional skolem terms that cannot be removed, we must apply a variation of subtract to it to find the corresponding folding.

Definition 4.2 (Product Rewriting Generation) Let $Q_1, \ldots, Q_d$ be rewritings in rewritings(Q) whose bodies are unifiable via mgu θ such that $Q_i\theta$ is of the form Q :$-Body, C_i, \texttt{context}(C_c)$ for $1 \leq i \leq d$. Then the product rewriting of $Q_1, \ldots, Q_d$ is denoted by Q_{pr} and it is given by:

$$Q :\!-Body, \texttt{context}(C_c), ((C_1 \wedge C_c) \vee \ldots \vee (C_d \wedge C_c)).$$

We first observe that if the disjunction $(C_1 \wedge C_c) \vee \ldots \vee (C_d \wedge C_c)$ is reducible to a single conjunction that does not contain any skolem terms then the resulting rewriting is a folding. Similarly, if after simplifying the resulting conjunction by removing all the conditions implied by C_c we do not have any skolem terms, then the result is a folding. This is similar to the simplification in the last step of the algorithm for rewriting generation. Finally, we can use subtract to remove the remaining skolem terms and thus obtain a folding.

If, however, the above disjunction does not reduce to a conjunction, then we must use a variation of the subtract operator that takes not only a single conjunction, but a disjunction of conjunctions that (possibly) contain skolem terms, then finds the weakest constraint that removes the disjunction and the skolem terms according to the same criteria. This new subtract operator then will enable us to find all the possible foldings of a query.

We finish the section on collecting foldings with a final theorem. We observe that the completeness for this case depends on the development of the extended subtract algorithm. The following theorem states a different completeness result, which states that the output of our algorithm can be used to check if a given expression is a folding of a query even if there is no containment mapping for it. Moreover, this check is accomplished in terms of a containment mapping to the output of our algorithm. This complements the results in [13] and provides an algorithm for the case when a query is contained in another without the existence of a containment mapping.

Theorem 4.5 (Completeness for Folding Test) If Q_f is a folding of Q, then there are $Q_{r_1}, \ldots, Q_{r_m} \in$ rewritings(Q) and $\theta = \text{mgu}(\text{body}(Q_{r_1}), \ldots, \text{body}(Q_{r_m}))$ such that $\text{body}(Q_{r_i})\theta = \text{body}(Q_f)$ and $Q_{r_1}\theta \vee \cdots \vee Q_{r_m}\theta$ is implied by constraint(Q_f).

5 Generating Relaxed Foldings

We have given in the previous sections, a sound algorithm for finding the foldings of a query and examined its completeness properties. The foldings generated by the algorithm give queries stronger than the user query. Hence given a query, we can find answers to a part of the query which corresponds to adding extra conditions to the original query. We will now see how the same algorithm may be extended to provide ways of relaxing a query in order to find additional foldings of it. In other words, we want to weaker forms of the original query that can be folded.

The problem of query relaxation becomes critical when some information sources allow access only through pre-defined queries, given by the views in the system. These views may come very close to answering the user query, but fail to provide foldings since the user requested too detailed information. An easy example of this is a query that asks for the name, age and prescription of the people who were treated for a certain medical problem. It may be possible to query an external information source for this query, but the source does not provide the name of the patients for security reasons. Then, an easy relaxation may be to drop the name attribute in the query and obtain a possible folding of it using the same algorithm.

Even though relaxations change the meaning of the user query, relaxed foldings may provide access to very valuable data that cannot be discarded. The decision as to when, how and how much to relax changes from system to system. In this paper, we will only concentrate on generating all possible ways of relaxing the query using our algorithm. The problem of ordering these relaxations and specifying their acceptability is left out of the scope of this paper.

In this section, we make the following slight modification in the rewriting generation algorithm of Section 3.3: Recall that both when creating the initial labels for the query and when merging the labels we required the resulting tuples in the labels to be proper. This obviously allowed us to prune the search by not generating rewritings that would contain skolem terms in the head or the body (there might still be skolem terms in the constraint), since these would never correspond to foldings. Now suppose we change this restriction and the tuples are not required to be proper. Instead, they can contain skolem terms everywhere except for the *view literals* in the rewriting columns. In other words, we apply the rewriting rules generated in Section 3.1 with the only restriction on view literals. Then we compute the rewritings exactly as before. Let us denote the set of all rewritings (including the improper ones) of query Q with all_rewritings(Q). Note first that rewritings(Q) $\subseteq$ all_rewritings(Q). We concentrate mainly on the rewritings with skolem terms in the head or the body, and investigate how to use skolem terms to guide query relaxation.

Before we discuss the different relaxation techniques, we are going to define a specific relaxation for built-in literals. This sort of relaxation will be needed to ensure that the built-in literals do not contain any dangling variables as a result of another relaxation operation.

Definition 5.1 (Strongest Simplification) Suppose C is a conjunction of BILs and $X_1, \ldots, X_n$ are terms (either variables or skolem terms) that appear in C. Then, C' is called the strongest simplification of C with respect to $X_1, \ldots, X_n$ iff C' is the strongest constraint that is implied by C such that C' does not contain any of the terms $X_1, \ldots, X_n$.

As an example, suppose we have the constraint $X \leq Y, Y \leq Z$. Then the strongest simplification of this constraint with respect to the variable Y is given as $X \leq Z$.

The strongest simplification is very easy to compute. Given a conjunction C of BILs, we simply compute all the possible inequalities C^+ implied by C using the algorithm given in [20, pp. 892]. Then, we remove from C^+ all those BILs that contain one of the terms that we want to get rid of. The remaining conjunction is then the strongest simplification of C.

5.1 Dropping Attributes

This process refers to dropping some of the distinguished variables in the query. We motivate this with the example given earlier in this section.

Example 5.1 *Suppose we are given the following query that asks for the name, age and prescription for all the patients who where treated for cataract.*

Query(Name, Age, Prescription) :– *Patient(Name, Age, PatientID),*
Diagnosis(PatientID, `cataract`*, Pres_Id),*
Prescription(Pres_ID, Physician_Id, Prescription, Note_Id).

This query could be executed in the local database where it is possible to see the names of the patients. Suppose we are accessing the another database which does not allow us to see the names of the patients for security reasons. The following is the view that can be executed on that database:

View(Age, Diagnosis, Prescription) :– *Patient(Name, Age, PatientID),*
Diagnosis(PatientID, Diagnosis, Pres_Id),
Prescription(Pres_ID, Physician_Id, Prescription, Note_Id).

There is no folding of the query using this view. However if we remove the attribute Name from the head of the query, we get the following folding using our algorithm for the relaxed query: (here the symbol "_" is used for dropped attributes)

Query(_, Age, Prescription) :– *View(Age,* `cataract`*, Prescription).*

Corollary 5.1 (Dropping Attributes) Let Q' be a rewriting in all_rewritings(Q) and the substitution θ be a containment mapping from Q to unfold(Q'). Suppose that Q' does not contain any skolem terms in the body but it contains skolem terms for the distinguished variables $X_1, \ldots, X_k$ in the head of Q.

We drop the attributes $X_1, \ldots, X_k$ from Q and $(X_1, \ldots, X_k)\theta$ from Q'. Furthermore, we replace the constraint of Q (and of Q' respectively) with its strongest simplification with respect to the terms $X_1, \ldots, X_k$ (and w.r.t. $(X_1, \ldots, X_k)\theta$ in Q')

Then the relaxed Q' is a rewriting for the relaxed Q.

Example 5.2 *Let us go back to the query given in example 5.1. By applying the less restricted rewriting generation algorithm, we will get the following rewriting in all_rewritings(Q):*

Query(f(Age, `cataract`*, Prescription), Age, Prescription)* :– *View(Age,* `cataract`*, Prescription).*
Since the body does not contain any skolem terms and only the first distinguished variable is substituted with a skolem term, we can obtain a folding by removing this attribute as we had seen in example 5.1.

The reason such a situation may arise is that we use the rewrite rules to match against the base predicates. It is possible for some of the distinguished variables in the base predicates not to appear in the view definition. Hence, these variables will be assigned to skolem terms in the rewriting. This is the simplest example of our gain by running the less restricted algorithm. Corollary 5.1 above follows from the soundness of our algorithm as given in theorem 3.1. The theorem is independent of the existence of skolem terms in the rewritings.

5.2 Dropping Base Literals

In this section we investigate the possibility of removing some of the base literals from the query and obtaining a folding for the relaxed query. Let us consider the following example:

Example 5.3 *Suppose we want to find the foldings of the following query that asks for the allergy symptoms observed in people who were prescribed a certain drug say* `xd_2001`*, and the physician who prescribed this drug.*

Query(Allergy_Text, Physician_Name) :– Prescription(Pres_Id, Physician_Id, `xd_2001`*, Note_Id),*
Notes(Note_Id, Allergy_Text),
Physician(Physician_Id, Physician_Name, Clinic).

Again for security reasons, suppose the information about the name of the physician is only visible to the local users and this database can only be queried by external users via the following view:

View(Prescription, Allergy_Text) :– Prescription(Pres_Id, Physician_Id, Prescription, Note_Id),
Notes(Note_Id, Allergy_Text).

If we drop the last base literal from the query that enabled us to find the names of the physicians, then we can obtain a folding of this revised query given as follows:

Query(Allergy_Text, _) :– View(`xd_2001`*, Allergy_Text).*

Note that we dropped the attribute Physician_Name from the head of the query since there is no way to instantiate this variable using the current folding. As a result, now we can get a larger set of allergy symptoms for the given drug.

Corollary 5.2 (Dropping Base Literals) Let Q' be a (partial) rewriting in all_rewritings(Q) and the substitution θ be a containment mapping from Q to unfold(Q'). Suppose Q' contains skolem terms in some base literals but does not contain any skolem terms in view literals or in the head.

We drop from Q' all the base literals $p\theta$ that contain a skolem term and from Q the corresponding literal p. Now, let $X_1, \ldots, X_n$ be the variables in the relaxed Q that no longer appear in any of the base literals. We drop from the head of Q (and Q' respectively) any of $X_1, \ldots, X_n$ (and of $(X_1, \ldots, X_n)\theta$.) Finally, we replace the constraint of Q (and Q' respectively) with its strongest simplification with respect to the terms $X_1, \ldots, X_k$ (and w.r.t. $(X_1, \ldots, X_k)\theta$ in Q').

Then, the relaxed Q' is a rewriting for the relaxed Q.

Example 5.4 *Let us go back to the query in example 5.3. If we fold the query with the view given in the same example using the less restricted rewriting generation algorithm, we get the following rewriting in all_rewritings*(Q)*:*

Query(Allergy_Text, Physician_Name) :– View(`xd_2001`*, Allergy_Text)*
Physician(f(`xd_2001`*, Allergy_Text), Physician_Name, Clinic).*

Since the base literal Physician contains a skolem term, we decide to drop this literal and the distinguished variable Physician_Name since it does not occur anywhere except in the base literal Physician that is being dropped. The resulting folding is same as the one given in example 5.3.

5.3 Relaxing Built-In Literals

Finally, we consider the relaxation of the query by weakening the BILs in it. Again we will see that it is very easy to formulate the possible relaxations of queries by simply analyzing the rewritings obtained by our algorithm. For this case however we do not need to look at the rewritings in all_rewritings(Q), since rewritings in rewritings(Q) already contain skolem terms in BILs. The corollaries in this section apply to the rewritings in all_rewritings(Q) without any modifications. First let us consider the following example.

Example 5.5 *Let us consider the following query that asks for the drugs administered to all the patients who are younger than 25 and were treated for a certain medical problem, say "vitreous deficiency" that is not very common among this age group.*

Query(Prescription) :– *Patient(Name, Age, PatientID), Age* $\leq$ *25,*
Diagnosis(PatientID, `vitreous_deficiency`*, Pres_Id),*
Prescription(Pres_ID, Physician_Id, Prescription, Note_Id).

Suppose that we can query results of a survey. The results do not contain any specific age information, but we know that the survey was conducted on people younger than 30. This implicit information may be coded explicitly to give the following view on the survey information:

View(Diagnosis, Prescription) :– *Patient(Name, Age, PatientID), Age* $\leq$ *30,*
Diagnosis(PatientID, Diagnosis, Pres_Id),
Prescription(Pres_ID, Physician_Id, Prescription, Note_Id).

Obviously this view cannot be used for a folding unless the condition Age $\leq$ *25 in Query is relaxed to Age* $\leq$ *30. This may be an acceptable relaxation from the user's point of view. After this relaxation we have the following folding for the relaxed query:*

Query(Prescription) :– *View(*`vitreous_deficiency`*, Prescription).*

Corollary 5.3 (Relaxing BILs) Let Q' be a (partial) rewriting in all_rewritings(Q) and the substitution θ be a containment mapping from Q to unfold(Q'). Suppose Q' only contains skolem terms in BILs.

Now, let C be the strongest constraint implied by constraint(Q) (i.e. constraint(Q) $\models C$) such that the BILs containing skolem terms in (the transitive closure of) $C\theta$ are implied by the context predicate of Q'.

Then, head(Q') :–body(Q'), $C\theta$ is a folding of head(Q) :–body(Q), C.

Example 5.6 *Let us go back to the query in example 5.5. We have the following rewriting in rewritings(Query).*

Query(Prescription) :– *View(*`vitreous_deficiency`*, Prescription),*
f(`vitreous deficiency`*, Prescription)* $\leq$ *25,*
`context`*(f(*`vitreous_deficiency, Prescription`*)* $\leq$ *30).*

We use corollary 5.3 and find the strongest constraint that is implied by the context predicate is f(vitreous_deficiency, Prescription) $\leq$ *30. We hence relax the corresponding constraint in the query, i.e. replace Age* ≤ 25 *with Age* $\leq$ 30*, and the rewriting of this relaxed query with the same view literal has the constraint f(vitreous_deficiency, Prescription)* $\leq$ *30 and this can be removed since it is implied by the context. The resulting rewriting is then a folding.*

5.4 Combining Different Relaxation Techniques

We will discuss in this section how to combine the three different relaxation techniques. Note that these techniques all refer to a different part of the rewriting, namely to the distinguished variables, the base literals and the BILs. When more than one relaxation is needed for a folding, it is very important to choose the right order to apply them. For example dropping of base literals may result in dropping of attributes and BILs that do not contain skolem terms. Similarly dropping an attribute may result in removing a BIL with a skolem terms (that involves the dropped attribute.) We close this section with a general relaxation method that combines all the three relaxations discussed earlier.

Corollary 5.4 (Relaxing Rewritings) Let Q' be a (partial) rewriting in all_rewritings(Q) that contains skolem terms and the substitution θ be a containment mapping from Q to unfold(Q'). Then a relaxation of Q for which Q' can be relaxed to a rewriting without skolem terms is found as follows:

1. Drop all the base predicates $p\theta$ that contain a skolem term from Q' and corresponding p from Q. Clean-up both Q and Q'.
2. Drop all distinguished variables $X_1, \ldots, X_k$ from the relaxed Q and $(X_1, \ldots, X_k)\theta$ from the relaxed Q' where θ substitutes each X_i with a skolem term. Clean-up both Q and Q'.
3. Let C be the strongest constraint implied by the constraint of the relaxed Q such that the BILs containing skolem terms in (the transitive closure of) $C\theta$ are implied by the context predicate of the relaxed Q'. Then, replace the constraint of the relaxed Q with C and the constraint of the relaxed Q' with $C\theta$.
4. Finally, remove from the (transitive closure of the) constraint of the relaxed Q' all the BILs that are implied by `context`(Q').

The clean-up phase of a query/rewriting removes all the dangling terms (i.e. terms that do not appear in any view or base literal) from the head by removing them and from the constraint by finding its strongest simplification with respect to those terms.

6 Conclusions

In this paper, we have developed an algorithm for generating possible ways of answering a user query using views. The algorithm takes as input a conjunctive query and a set of conjunctive view definitions and finds all possible ways of folding the query using the views. The output of this algorithm contains all the possible foldings of a query when the query does not contain any built-in predicates. When the query contains built-in predicates, then extra processing of the output is needed to find all the foldings. The subtract operator performed on the output is sufficient to find all the foldings that are contained in the original query via a containment mapping. In cases when there is no such containment mapping, then the union operator must be performed together with a revised version of subtract to get all the foldings. The set of rewritings together with the union operator (without subtract) is complete for checking if an expression is a folding, even when there is no containment mapping for this folding. These foldings are contractions of the query in the sense that they answer the query completely but they may provide only a subset of the answers to it.

In addition to this, the output of the slightly modified version of the generator algorithm can be used to find the relaxed foldings a user query. The relaxations allow the query to access different views (sources) that provide answer to only a part of the query. Hence, the query can be modified by dropping projection attributes, base predicates and relaxing/removing the built-in predicates in the query.

This allows the development of flexible systems in which the user does not need to be aware of all the differences between the information services provided by different sources. The system provides immediate help in trying to match the user query to the available sources. There are many issues related to query optimization and query relaxation in such an environment that need to be addressed. We observe that *all* foldings generated by the algorithm may need to be executed to find all possible answers to a user query. Hence, removing redundant calls to views from the final set of foldings is a challenging problem. Similarly, providing access to sources by relaxation must be done in accordance with the user's query and preferences. There are many problems that need to be addressed to solve this problem as well.

Query folding has obvious applications in centralized databases. For example, databases often maintain materialized views. A query then can be answered by accessing views instead of base relations if the query can be folded using the views [5]. In multiple query answering, the result of a query can be used to at least partially answer another query if the second query can be folded using the first one. Query folding is even more important in a distributed environment. In client-server applications, views and queries might be cached at the client site. Client queries can be answered more efficiently if they can be folded using the cached data [10]. In the situation of a disconnected network, a query can still be answered at least partially if it can be folded using views and queries maintained at available sites.

In a federated environment containing multiple heterogeneous, autonomous, and legacy data sources, a data source might be capable of answering only limited kinds of queries [12]. We can envision a data source being described using a set of statements of the form[1]

$$q_1, \ldots, q_m \quad :- \quad p_1, \ldots, p_n$$

where $q_1, \ldots, q_m$ is a conjunctive query in the federated schema, and $p_1, \ldots, p_n$ is a conjunctive query in a data source. This statement says that a federated query $q_1, \ldots, q_m$ can be answered by a source query $p_1, \ldots, p_n$. From this statement, we can define a view in the federated database

$$r \quad :- \quad q_1, \ldots, q_m.$$

The job of the federated query processor is to determine if and how federated queries can be answered using these views, or equivalently, to determine if and how federated queries can be folded using these views.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems*, 4(4):435–454, December 1979.

[1] Notice that these statements are more general than the site descriptions in [12].

[3] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalence of relational expressions. *SIAM Journal of Computing*, 8(2):218–246, 1979.

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

[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 190–200, 1995.

[6] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of the Twelfth ACM Symposium on Principles of Database Systems*, pages 59–70, 1993.

[7] G. Dong and J. Su. Conjunctive query containment with respect to views and constraints. *Information Processing Letters*, 1995.

[8] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Transactions on Database Systems*, 20(3):288–324, September 1995.

[9] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and System Sciences*, 28(1):167–189, February 1984.

[10] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, 1994.

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

[12] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 1995.

[13] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM Symposium on Principles of Database Systems*, pages 95–104, 1995.

[14] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Query processing in the information manifold. Technical report, AT&T Bell Laboratories, 1996.

[15] X. Qian. Query folding. In *Proceedings of the Twelfth International Conference on Data Enginnering*, 1996.

[16] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *Proceedings of the Fourteenth ACM Symposium on Principles of Database Systems*, pages 105–112, 1995.

[17] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, October 1980.

[18] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 237–249, 1987.

[19] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 367–378. Morgan Kaufmann, 1994.

[20] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 2. Computer Science Press, 1989.

[21] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, pages 331–345, 1992.

[22] H. Z. Yang and P.-Å. Larson. Query transformation for PSJ-queries. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 245–254. Morgan Kaufmann, 1987.

Appendix A: Proofs of Results

Theorem 3.1 If $Q_r \in \text{rewritings}(Q)$ then there is a containment mapping from Q to $\text{unfold}(Q_r)$.

Proof Let the body of Q be $p_1, \ldots, p_n$. Since $Q_r \in \text{rewritings}(Q)$, there is a tuple in the u-join of labels associated with base literals in Q of the form $(t, \text{body}(Q_r), \texttt{context}(C))$, and there are no skolem terms in $\text{head}(Q_r)$ or $\text{body}(Q_r)$. Therefore there are rewrite rules $F_1, \ldots, F_n$ and a substitution θ such that

1. $\text{head}(Q_r) = \text{head}(Q)\theta$,
2. $\text{body}(Q_r) = (\text{body}(F_1), \ldots, \text{body}(F_n))\theta$,
3. $C \wedge \text{constraint}(Q_r)$ implies $\text{constraint}(Q)\theta$, and
4. $\text{body}(Q)\theta = (\text{head}(F_1), \ldots, \text{head}(F_n))\theta$.

From the above we conclude that

1. $\text{head}(Q)\theta = \text{head}(\text{unfold}(Q_r))$ since $\text{head}(\text{unfold}(Q_r)) = \text{head}(Q_r)$,
2. $\text{body}(Q)\theta \subseteq \text{unfold}(\text{body}(Q_r)) = \text{body}(\text{unfold}(Q_r))$ since $\text{head}(F_i)\theta \in \text{unfold}(\text{body}(F_i)\theta)$ for $1 \leq i \leq n$, and
3. $\text{constraint}(Q)\theta$ is implied by $\text{constraint}(\text{unfold}(Q_r))$ since $C \wedge \text{constraint}(Q_r)$ is equivalent to $\text{constraint}(\text{unfold}(Q_r))$.

Hence θ is a containment mapping from Q to $\text{unfold}(Q_r)$. □

Lemma 4.1 Suppose Q_f is a folding of query Q. If there is a containment mapping from Q to $\text{unfold}(Q_f)$, then there is a $Q_r \in \text{rewritings}(Q)$ and a substitution θ such that $\text{body}(Q_r)\theta = \text{body}(Q_f)$ and $\text{constraint}(\text{unfold}(Q_r))\theta$ is implied by $\text{constraint}(\text{unfold}(Q_f))$.

Proof Let the body of Q be $p_1, \ldots, p_n$, and θ' be a containment mapping from Q to $\text{unfold}(Q_f)$. Since $p_i\theta' \in \text{body}(\text{unfold}(Q_f))$, we assume without loss of generality that $\text{body}(Q_f)$ is a list of view literals $v_1, \ldots, v_n$ where $p_i\theta' \in \text{unfold}(v_i)$.

Since $\text{body}(Q_f)$ does not contain skolem terms, there are rewrite rules $F_1, \ldots, F_n \in \mathcal{F}$ and a substitution θ'' such that $v_i = \text{body}(F_i)\theta''$ and $p_i\theta' = \text{head}(F_i)\theta''$. Therefore $\sigma = \text{mgu}(p_i, \text{head}(F_i))$ exists, and there is a $Q_r \in \text{rewritings}(Q)$ where $\text{body}(Q_r)$ is $(\text{body}(F_1), \ldots, \text{body}(F_n))\sigma$ and $\text{head}(Q_r) = \text{head}(Q)\sigma$.

Hence there is a substitution θ such that $\text{head}(Q)\theta' = \text{head}(Q)\sigma\theta$ and $\text{body}(F_i)\theta'' = \text{body}(F_i)\sigma\theta$. In other words, $\text{head}(Q_r)\theta = \text{head}(Q_f)$ and $\text{body}(Q_r)\theta = \text{body}(Q_f)$.

Since θ' is a containment mapping from Q to $\text{unfold}(Q_f)$, $\text{constraint}(Q)\theta'$ is implied by $\text{constraint}(\text{unfold}(Q_f))$. Since $\text{constraint}(Q_r) = \text{constraint}(Q)\sigma$, $\text{constraint}(Q_r)\theta = \text{constraint}(Q)\theta'$ and hence $\text{constraint}(Q_r)\theta$ is implied by $\text{constraint}(\text{unfold}(Q_f))$. Since $\text{body}(Q_r)\theta = \text{body}(Q_f)$, $\text{constraint}(\text{unfold}(Q_r)\theta)$ is implied by $\text{constraint}(\text{unfold}(Q_f))$. □

Theorem 4.2 (Completeness when Queries Contain no Built-In Predicates) If Q_f is a folding of Q and Q contains no built-in predicates, then there is a $Q_r \in \text{rewritings}(Q)$ such that Q_r is a folding of Q and $Q_f \subseteq Q_r$.

Proof Since Q does not contain BIPs, there is a containment mapping from Q to unfold(Q_f) [13]. According to Lemma 4.1, there is a $Q_r \in$ rewritings(Q) and a substitution θ such that body$(Q_r)\theta =$ body(Q_f) and constraint(unfold(Q_r))θ is implied by constraint(unfold(Q_f)), meaning that $Q_f \subseteq Q_r$. Again since Q does not contain BIPs, neither does Q_r. Thus Q_r is a folding of Q. □

Theorem 4.3 (Completeness when Containment Mappings Exist) If Q_f is a folding of Q and there is a containment mapping from Q to unfold(Q_f), then there is a $Q_r \in$ rewritings(Q) and a Q'_f, where Q'_f has the same head and body as Q_r and constraint$(Q'_f) \in$ subtract(`context`(Q_r), constraint(Q_r)), such that $Q_f \subseteq Q'_f$.
Proof According to Lemma 4.1, there is a $Q_r \in$ rewritings(Q) and a substitution θ such that body$(Q_r)\theta =$ body(Q_f) and constraint(unfold(Q_r))θ is implied by constraint(unfold(Q_f)). Let C be `context`(Q_r), then $C\theta$ is the same as `context`(Q_f). Notice that constraint(unfold(Q_r)) $= C \wedge$constraint(Q_r), and constraint(unfold(Q_f)) $=$ $C\theta \wedge$ constraint(Q_f). Hence constraint$(Q_r)\theta$ is implied by $C\theta \wedge$ constraint(Q_f). Since constraint(Q_f) contains no skolem terms, there is a $C' \in$ subtract(C, constraint(Q_r)) such that constraint(Q_f) implies $C'\theta$. We may choose Q'_f to be head(Q_r) :–body$(Q_f), C'$. Obviously $Q_f \subseteq Q'_f$ since θ is a containment mapping from Q'_f to Q_f. □

Theorem 4.5 (Completeness for Folding Test) If Q_f is a folding of Q, then there are $Q_{r_1}, \ldots, Q_{r_m} \in$ rewritings(Q) and $\theta =$ mgu(body$(Q_{r_1}), \ldots,$ body(Q_{r_m})) such that body$(Q_{r_i})\theta =$ body(Q_f) and $Q_{r_1}\theta \vee \cdots \vee Q_{r_m}\theta$ is implied by constraint(Q_f).
Proof Since Q_f is a folding of Q, Levy et. al. [13] showed that there are rewritings $Q_{f_1}, \ldots, Q_{f_m}$ of Q and containment mappings from Q to unfold(Q_{f_i}), such that

1. head$(Q_f) =$ head(Q_{f_i}),
2. body$(Q_f) =$ body(Q_{f_i}), and
3. constraint(unfold(Q_f)) is equivalent to $\bigvee_i$ constraint(unfold(Q_{f_i})).

According to Lemma 4.1, there are $Q_{r_1}, \ldots, Q_{r_m} \in$ rewritings(Q) and substitutions θ_i such that body$(Q_{r_i})\theta_i =$ body(Q_{f_i}). Hence there is a substitution θ such that

1. head$(Q_{r_i})\theta$ and head(Q_{f_i}) have the same arguments,
2. body$(Q_{r_i})\theta =$ body(Q_f), and
3. constraint(unfold(Q_{r_i}))θ is implied by constraint(unfold(Q_{f_i})).

Therefore $\bigvee_i$ constraint(unfold(Q_{r_i}))θ is implied by constraint(unfold(Q_f)). □