Query Rewriting in the Presence of Functional Dependencies with Lossless Join Decomposition

Qingyuan Bai Jun Hong M. F. McTear
School of Computing and Mathematics
University of Ulster at Jordanstown
Newtownabbey, Co. Antrim BT37 0QB, UK
{q.bai, j.hong, m.mctear}@ulster.ac.uk

Abstract: Query rewriting is an essential issue in data integration systems over the Web and has received considerable attention. Many rewriting algorithms, e.g., the bucket algorithm, the inverse rules algorithm, the SVB algorithm and the MiniCon algorithm, have been proposed to address this issue in the absence of functional dependencies. These algorithms can be divided into two categories: bucket-based algorithms and inverse rule-based algorithms. All the bucket-based algorithms may sometimes miss query rewritings in the presence of functional dependencies with/without lossless join decomposition because they do not consider the effects of functional dependencies. However, some efforts have been made in developing inverse rule-based algorithms to solve this problem. In this paper, we propose an algorithm for query rewriting using views in the presence of functional dependencies with lossless join decomposition, which solves the problem of missing query rewritings in the MiniCon algorithm. We analyse the violation situation of the unification process in the MiniCon algorithm and describe our approach to handling such violation. Our algorithm extends the MiniCon algorithm to cope with query rewriting using views in the presence of functional dependencies with lossless join decomposition.

1. Introduction

Query rewriting is an essential issue in data integration systems over the Web and has received considerable attention. There are two main approaches for query rewriting, i.e., Global As View and Local As View (LAV for short). In the LAV approach, data sources are described by views that are defined over a mediated schema, and user queries are also posed in terms of the mediated schema. Thus, it is needed to reformulate user queries into new queries over data sources schemas in order to answer user queries. The LAV approach for query rewriting is closely related to the problem of answering queries using views, which is relevant to a wide variety of database applications [8].

In data integration, views describe a set of autonomous heterogeneous data sources. Thus, in this context we usually try to find maximally-contained rewritings that provide the best possible answers for a given query. Many algorithms, for example, the bucket algorithm [9],[10], the inverse rules algorithm [13],[2], the Shared Variables Bucket (SVB for short) algorithm [11], and the MiniCon algorithm [12], have been proposed. These algorithms can be divided into two categories: bucket-based algorithms and inverse rule-based algorithms. All these algorithms can generate the
maximally-contained query rewriting in the absence of functional dependencies. Because they do not consider the effects of functional dependencies, they may sometimes miss query rewritings in the presence of functional dependencies with/without lossless join decomposition. Some efforts have been made in developing inverse rule-based algorithms to address this issue. In the context of database systems, there are usually some functional dependencies in the schema. In the context of data integration, each data source is represented by views which are defined over a mediated schema. There are often functional dependencies in the mediated schema.

Now let us first have a look at the following example that motivates us to develop the algorithm for query rewriting in the presence of functional dependencies with lossless join decomposition.

**Example 1** Suppose that a mediated schema has a relation \(W(X,Y,Z)\) and a query is made as follows:

\[ Q(X,Y,Z) :- W(X,Y,Z) \]

There are two available data sources denoted by \(V_1\) and \(V_2\).

\[ V_1(X,Y) :- W(X,Y,Z) \]
\[ V_2(X,Z) :- W(X,Y,Z) \]

Assume that there are functional dependencies in relation \(W(X,Y,Z)\):

\[ X \rightarrow Y, X \rightarrow Z. \]

Previous bucket-based algorithms that are based on unification between a subgoal of a given query and a subgoal of some views cannot generate any rewritings for \(Q\), because there is no containment mapping between \(Q\) and either \(V_1\) or \(V_2\) such that all the distinguished variables of \(Q\) can be mapped to the distinguished variables of \(V_1\) or \(V_2\).

Note that due to the functional dependencies in \(W\), \(W\) can be decomposed losslessly as follows (The symbol \(\bowtie\) represents a natural join throughout this paper):

\[ W(X,Y,Z) = W_1(X,Y) \bowtie W_2(X,Z). \]

Hence, the query and the views can be redefined as follows:

\[ Q(X,Y,Z) :- W_1(X,Y), W_2(X,Z) \]
\[ V_1(X,Y) :- W_1(X,Y), W_2(X,Z_i) \]
\[ V_2(X,Z) :- W_1(X,Y_2), W_2(X,Z) \]

Now we can apply any of the previous bucket-based rewriting algorithms to generate all possible query rewritings for \(Q\). For instance, using the MiniCon algorithm, we can get the MCDs as follows (The symbol \(\rightarrow_m\) means the mapping throughout this paper):

<table>
<thead>
<tr>
<th>View</th>
<th>(\rightarrow_m)</th>
<th>(\phi)</th>
<th>(G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(V_1(X,Y))</td>
<td>X (\rightarrow_m) X, Y (\rightarrow_m) Y</td>
<td>X (\rightarrow_m) X, Y (\rightarrow_m) Y</td>
<td>(g_1)</td>
</tr>
<tr>
<td>(V_2(X,Z))</td>
<td>X (\rightarrow_m) X, Z (\rightarrow_m) Z</td>
<td>X (\rightarrow_m) X, Z (\rightarrow_m) Z</td>
<td>(g_2)</td>
</tr>
</tbody>
</table>

Table 1: The MCDs for \(Q\) over each view (\(g_i\) means the \(i\)th subgoal of \(Q\))

We can then get the following rewriting of \(Q\) by combining views from the MCDs such that each subgoal of \(Q\) is covered by some of those views:

\[ Q'(X,Y,Z) :- V_1(X,Y), V_2(X,Z) \]

In [12], experimental results show that the MiniCon algorithm which is based on use of buckets is more efficient than any other rewriting algorithms, including the inverse rule algorithm. Thus, in
this paper, we have focused on an extension of the MiniCon algorithm for query rewriting in the presence of functional dependencies with lossless join decomposition in a mediated schema.

2. Preliminaries

2.1 Query Containment and Answering Queries Using Views

We consider the problem of answering conjunctive queries using views. A conjunctive query has the form:

\[ Q(X) : - R_1(X_1), \ldots, R_n(X_n) \]

where \( R_1(X_1), \ldots, R_n(X_n) \) are the subgoals referred to database relations. \( Q(X) \) is the head of the query. The tuples \( X, X_1, \ldots, X_n \) contain either variables or constants. We require that the query be safe, i.e., \( X \subseteq X_1 \cup \ldots \cup X_n \). The variables in \( X \) are the distinguished variables, and others are existential variables. A view is a named query. If the query results are stored, we refer to them as a materialized view.

A query \( Q_1 \) is contained in the \( Q_2 \), denoted by \( Q_1 \subseteq Q_2 \), if the answers to \( Q_1 \) are a subset of the answers to \( Q_2 \) for any database instance. A mapping \( \varphi \) from the variables of \( Q_2 \) to the variables of \( Q_1 \) is a containment mapping if (1) \( \varphi \) maps every subgoal in the body of \( Q_2 \) to a subgoal in the body of \( Q_1 \), and (2) \( \varphi \) 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 \( Q_1 \).

Given a query \( Q \) and a set of view definitions \( V_1, \ldots, V_m \), a rewriting of \( Q \) using views is a query expression \( Q' \) whose body predicates are only from \( V_1, \ldots, V_m \).

Note that the views are not assumed to contain all the tuples in their definitions since the data sources are managed autonomously. Moreover, we cannot always find an equivalent rewriting of the query using the views because data sources may not contain all the answers to the query. Instead, we consider the problem of finding maximally-contained rewritings as follows.

**Definition 1 (Maximally-contained Rewriting)** \( Q' \) is a maximally-contained rewriting of a query \( Q \) using views \( V_1, \ldots, V_n \) with respect to a query language \( L \) if

1. \( Q' \) is a rewriting of \( Q \) such that \( Q' \subseteq Q \),
2. there is no other rewriting \( Q_1 \) of \( Q \) such that \( Q_1 \subseteq Q \) and \( Q' \subseteq Q_1 \).

2.2 The MiniCon Algorithm [12]

Since our algorithm extends the MiniCon algorithm in the presence of functional dependencies, we first have a brief look at it. In the MiniCon algorithm, MiniCon Descriptions (MCDs for short), containing the information about the unification between a given query and a view, take the roles of buckets in the bucket algorithm. The formal definition of MCD is given in [12], referred to Example 1 as the above. The MiniCon algorithm consists of two steps: forming the MCDs and combining the MCDs.
Algorithm (Forming the Minimum MCDs)
For each subgoal $R$ of $Q$ and each subgoal $R'$ of view $V$, find a least restrictive head homomorphism $h$ on $V$ such that there exists a mapping $\phi$ such that $\phi(R)=h(R')$. If $h$ and $\phi$ exist, then extend the domain of $\phi$ to variables in a minimum set $G$ of subgoals of $Q$ such that,
(a) each head variable in $G$ is mapped to a head variable in $h(V)$;
(b) if an existential variable $x$ in $G$ is mapped to an existential variable of $h(V)$, then (i) each subgoal of $Q$ involving $x$ is in $G$; (ii) all variables in the comparisons $B$ of $Q$ that involve $x$ are in the domain of $\phi$ and comparisons of $Q$ and $h(V)$ are consistent.

Algorithm (Generating Rewritings)
The rewriting is generated by combinations of the minimum MCDs $(h_i, V_i, \phi_i, G_i), i=1,...,k$, in the following way:

$$Q'(X) :- V_1(X_1),...,V_k(X_k)$$

where, for $i\neq j$, $G_i\cap G_j=\emptyset$, and $G_1\cup G_2\cup...\cup G_k=$ subgoals($Q$).

2.3 Lossless Join Decomposition

An instance of a relation $R$ satisfies the functional dependency $A_1,..., A_n \rightarrow B$ if for every two tuples $t$ and $u$ in $R$ with $t.A_i=u.A_i$ for $i=1,...,n$, also $t.B=u.B$. Usually, each database schema has a set of functional dependencies which would be considered in database schema design. Lossless join decomposition means that no information of the original relation $R$ is lost when $R$ is decomposed into two sub-relations $R_1$ and $R_2$. A decomposition of a relation $R(U)$ into two relations $R_1(X)$ and $R_2(Y)$ with respect to a set of functional dependencies $\Sigma$ is lossless join decomposition if for every instance $r$ of $R$, $r = \pi_X(r) \Join \pi_Y(r)$, where $U$, $X$, and $Y$ are a set of attributes in $R$, $R_1$, and $R_2$ respectively, and $U=X\cup Y$, $\pi_X$ is projection operation on $X$. It is well known that a decomposition of a relation $R$ into $R_1$ and $R_2$ is lossless join decomposition if one of the following conditions holds:

(1) $R_1 \cap R_2 \rightarrow R_1$ or (2) $R_1 \cap R_2 \rightarrow R_2$

In the presence of functional dependencies $\Sigma$, we redefine the notion of query containment to be query containment relative to $\Sigma$. Query $Q_1$ is contained in query $Q_2$ relative to $\Sigma$, denoted $Q_1 \subseteq_{\Sigma} Q_2$, if for each database $D$ satisfying $\Sigma$, $Q_1(D) \subseteq Q_2(D)$. Maximal query containment relative to $\Sigma$ can be defined accordingly.

3. Query Rewriting in the Presence of Functional Dependencies with Lossless Join Decomposition

Now we consider the problem of query rewriting in the presence of functional dependencies. In the context of databases, database schemas often have some integrity constraints, such as functional dependencies, and/or inclusion dependencies. Thus, dealing with integrity constraints has practical significance. In this section, we consider query rewriting in the presence of functional dependencies with lossless join decomposition.
We first analyse the unification procedure of the MiniCon algorithm that plays a key role in forming a MCD. To make sure that unification is successful, the following conditions should be satisfied:

(C1) a distinguished variable of Q should be mapped to a distinguished variable of a view;
(C2) an existential variable of Q could be mapped to a distinguished variable of a view;
(C3) an existential variable \( B_Q \) of Q could be mapped to an existential \( B_V \) variable of a view. If it is a shared variable (an existential variable that appears at least twice in the body of Q) of Q, then a set of subgoals of Q containing \( B_Q \) should contain a set of subgoals of the view containing \( B_V \).

The distinguished variables in a view represent the outputs of the view and can be constrained if needed, while the existential variables in a view can not be posed on any constraints. When either (C1) or (C3) is violated, the MiniCon algorithm would fail to form a MCD for a subgoal R of Q over a view V. In some cases, we can eliminate the violation of (C1) by making use of lossless join decomposition. Specifically, suppose that R(A,...) is a subgoal of Q, A is a distinguished variable, but would be mapped into an existential variable of some view V using the MiniCon algorithm. In this case, in the point of view of the MiniCon algorithm, V cannot be used further. In other words, V makes no contribution to answering Q. However, we find that if R can be losslessly decomposed into some sub-relations, we can change Q and V into the equivalent formalisms by replacing R with those new views. As a result, some query rewritings, which cannot be generated by the MiniCon algorithm, can be generated. As for the violation of (C3), it cannot be eliminated in this way because it is due to the mismatching of subgoals between Q and V, not the mismatching of variables. The lossless join decomposition does not help.

Therefore, we try to extend the MiniCon algorithm in such a context, i.e., there is a subgoal R in the bodies of both Q and some views, some of the distinguished variables of Q appearing in R would be mapped to non-distinguished variables of views, and there are a set of functional dependencies \( \Sigma \) so that R can be losslessly decomposed. From now on, we use X, \( X_i \), Y, et al, to represent a set of attributes (variables) in a relation, A, B, C to represent single attribute (variable) in a relation. Our algorithm is described as follows.

Suppose that Q has the following form:
Q(X, Z):- ...,R(X, Y), ...

First, for each subgoal, e.g., R, of Q we check each view whether the MCDs can be formed. If so, those MCDs will be used to generate possible query rewritings. Otherwise, we check whether (C1) is violated. If so, we put such a view into a set of views \( V_R=\{V_j(Y_j),j=1,2,...,m\} \). After this procedure, there may be some groups of views, \( V_R, V_S, \) etc. Now we consider query rewriting from those views that could not be used in the MiniCon algorithm.

If there are a set of functional dependencies \( \Sigma \) on R, R could be losslessly decomposed in polynomial time [1]. Suppose that R is losslessly decomposed to \( \{R_1(X_1),...,R_k(X_k)\} \). For each \( R_i(X_i), i=1,...,k \), we check whether there exists a view \( V_j(Y_j) \in V_R, j=1,2,...,m \), such that
\[
DV(Y_j) \supseteq DV(X_i)
\]
(*)

where, DV(X) represents the distinguished variables in X. If so, the violation of (C1) is eliminated with respect to the distinguished variables in R. Then, the same procedure is applied to the rest of
the groups, e.g., V_S. After this step, we can replace the original relations with sub-relations obtained from lossless join decomposition, and then carry out unification in the same way as the first step in the MiniCon algorithm. By the definition of lossless join decomposition, equivalence between the original query and the new form of query, as well as the original views and the new forms of views, are guaranteed. This follows from the following theorem which can be easily proved.

**Theorem 1**: Let a view (or query) be defined as follows:
\[ V(X, Y) : R(X, Y, Z), \]
where, X, Y and Z are a set of attributes of R. Without loss of generality, we assume that a relation R is losslessly decomposed into two relations \( R_1(X, Y) \) and \( R_2(X, Z) \), i.e.,
\[ R(X, Y, Z) = R_1(X, Y) \Join R_2(X, Z). \]
Then the subgoal \( R_2(X, Z) \) is redundant, i.e., \( V(X, Y) \) is equivalent to the following view (or query):
\[ V'(X, Y) : R_1(X, Y). \]

In fact, the unification between new query and new views obtained by replacing R with its sub-relations can be simplified. Suppose that for each \( R_i(X_i), i=1,...,k \), there exists a view \( V_j(Y_j) \in V_R \), such that \( DV(Y_j) \subseteq DV(X_i) \). We can create a bucket for \( R_i \) over \( V_j \) as follows:

<table>
<thead>
<tr>
<th>View</th>
<th>G (subgoal covered by view)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_j )</td>
<td>( G_j = R_i )</td>
</tr>
</tbody>
</table>

**Table 2**: The buckets for such a view that satisfies (*)

If there are a set of buckets such that:
1. \( G_i \cap G_j = \emptyset \), \( i \neq j \), and
2. \( G_1 \cup ... \cup G_m = R_1 \cup ... \cup R_k \), \( 1 \leq m \leq m \),
then, \( V_1 \Join ... \Join V_m \) is contained in \( R \). In other words, if \( Q \) contains only a relation \( R \), then \( V_1 \Join ... \Join V_m \) is a rewriting of \( Q \).

**Algorithm**: Query Rewriting in the Presence of Functional Dependencies with Lossless Join Decomposition

**Input**: A set of the relations in a mediated schema, \( R, S, ..., \) and a set of functional dependencies \( \Sigma \);
A set of the views \( \{V_1, V_2, ..., V_n\} \); A conjunctive query \( Q \);

**Output**: \( Q \)'s, the maximally-contained rewritings of \( Q \) relative to functional dependencies \( \Sigma \).

**Method**:

**Step 1**: For each subgoal of \( Q \), check each view whether the MCDs can be formed. If so, those MCDs will be used to generate maximally-contained rewritings [12]. Otherwise, check whether (C1) is violated. If so, the corresponding view is put into \( V_R, V_S, \) etc.

**Step 2**: For each subgoal of \( Q \), check whether it can be losslessly decomposed. If so, we losslessly decompose it into some sub-relations, and then check whether it is covered by the views in \( V_R, V_S, \) etc, as stated above.

**Step 3**: Generate maximally-contained rewritings in the same way as the MiniCon algorithm, i.e., combining the resulting views in step 2 with the views in the MCDs obtained in setp 1. These rewritings would be neglected by the MiniCon algorithm without consideration of functional dependencies.
Computational Complexity of the Algorithm:

In the MiniCon algorithm, for each subgoal of a query Q, each view will be unified for forming the MCDs. Thus, the computation in step 1 of our algorithm is the same as the MiniCon algorithm when unifying each view for each subgoal of Q. The computation of lossless join decomposition in step 2 of our algorithm is in polynomial time. The computation of checking condition (*) is also in polynomial time. In step 3, the computation of generating rewritings by combining the views in V_R, V_S, etc, with the views in the MCDs is the same as the MiniCon algorithm. Thus, compared with the MiniCon algorithm, the total increased computation in our algorithm is polynomial time. In [12], it is shown that the MiniCon algorithm can scale up to hundreds of views. Hence, our algorithm is expected to have the same scale-up.

Correctness of the Algorithm:

It follows from the MiniCon algorithm that the obtained rewritings in step 1 are maximally-contained rewritings of Q. By the definition of the lossless join decomposition and theorem 1, the original query and views are equivalent to the new query and those new views obtained by replacing a relation R with its sub-relations, respectively. After step 2, the violation of (C_1) is eliminated, which means there exists a containment mapping between the new query and those new views. Hence, the results of step 3 in our algorithm are the maximally-contained rewritings of Q.

We end this section with the following example that shows how our algorithm works in detail.

Example 2: Suppose that a mediated schema contains two relations R and S as follows:
R(X, Y, Z, K, L, M), S(A, B). There is a set of functional dependencies over R:
X → YZ, K → XM.
The query is as follows:
Q(X, Y, Z, M):- R(X, Y, Z, K, L, M), S(Z, C)
Suppose that there are six views as follows:
V_1(X, Y, Z, K):- R(X, Y, Z, K, L, M)
V_2(X, K, M):- R(X, Y, Z, K, L, M)
V_3(X, Y, Z, L, M):- R(X, Y, Z, K, L, M)
V_4(K, L):- R(X, Y, Z, K, L, M)
V_5(X, Y, L):- R(X, Y, Z, K, L, M)
V_6(Z, C):- S(Z, C)

Step 1: Using the MiniCon algorithm, we get the following MCDs:

<table>
<thead>
<tr>
<th>View</th>
<th>h(homomorphism)</th>
<th>φ(mapping)</th>
<th>G(subgoals of Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_3(X,Y,Z,L,M)</td>
<td>X→_m X, Y→_m Y, Z→_m Z, L→_m L, M→_m M</td>
<td>X→_m X, Y→_m Y, Z→_m Z, K→_m K, L→_m L, M→_m M</td>
<td>R</td>
</tr>
<tr>
<td>V_6(Z, C)</td>
<td>Z→_m Z, C→_m C</td>
<td>Z→_m Z, C→_m C</td>
<td>S</td>
</tr>
</tbody>
</table>

Table 3: The MCDs for Q over each view

We generate one maximally-contained query rewriting as follows:
Q’ (X, Y, Z, M):- V_3(X, Y, Z, L, M), V_6(Z, C).
The violation of (C₁) occurs. We get that \( V_R = \{ V_1(X, Y, Z, K), V_2(X, K, M), V_4(K, L), V_5(X, Y, L) \} \).

**Step 2:** Due to functional dependencies, we can losslessly decompose \( R \) into three sub-relations as follows:
\[
R(X, Y, Z, K, L, M) = \{ R_1(X, Y, Z), R_2(X, K, M), R_3(K, L) \}.
\]
We create the following buckets for \( R_i, i=1,2,3 \), over views in \( V_R \) as follows:

<table>
<thead>
<tr>
<th>View</th>
<th>( G_i = R_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_1(X, Y, Z, K) )</td>
<td>( G_1 = R_1 )</td>
</tr>
<tr>
<td>( V_2(X, K, M) )</td>
<td>( G_2 = R_2 )</td>
</tr>
<tr>
<td>( V_4(K, L) )</td>
<td>( G_3 = R_3 )</td>
</tr>
</tbody>
</table>

**Table 4:** The buckets for such a view that satisfies (*)

**Step 3:** Because \( G_1 \cup G_2 \cup G_3 = R_1 \cup R_2 \cup R_3 \), it means that \( V_1(X, Y, Z, K) \) \( V_2(X, K, M) \) \( V_4(K, L) \) is contained in \( R \). By combining these views with \( V_6(Z, C) \), we generate another maximally-contained rewriting as follows:

\[
Q''(X, Y, Z, M) :\text{if } V_1(X, Y, Z, K), V_2(X, K, M), V_4(K, L), V_6(Z, C).
\]

### 4. Related Work

As stated before, we can divide all the previous algorithms into two categories as below. Now we give a brief look at each category of algorithms.

#### 4.1 The Algorithms Based on Use of Buckets

The bucket algorithm [9],[10] proceeds in two stages. Initially, a bucket is created for each subgoal of a query. A view is put in the bucket if it can be unified with a subgoal in the query. Next, candidate query plans are generated by picking one view from each bucket and then verified using containment tests.

A non-distinguished variable that appears in more than one subgoal of a query is called a shared variable. In the SVB algorithm [11], given a query \( Q \), two types of buckets are created. The first type of buckets, the single-subgoal buckets are built in the same way as the bucket algorithm. The second type of buckets, the shared-variable buckets are created by checking the containment mapping from a set of subgoals in \( Q \) containing a shared variable to some subgoals in a view. Once all the buckets are created, the algorithm constructs rewritings by combining views from buckets which contain disjoint sets of subgoals of \( Q \).

The MiniCon algorithm [12] proceeds in principle in the same way as the SVB algorithm. Both algorithms focus on the roles of distinguished variables and shared variables in a query. The details of the MiniCon algorithm are given in Section 2. In [12], the MiniCon algorithm is shown to be able to scale up to hundreds of views.
4.2 The Algorithm Based on Use of Inverse rules

The key idea underlying the inverse rules algorithm [13][2] is to first construct a set of rules called inverse rules that invert the view definitions, and then replace existential variables in the view definitions with Skolem functions in the heads of the inverse rules. The rewriting of a query $Q$ using the set of views $V$ is simply the composition of $Q$ and the inverse rules for $V$ by the transformation method in [2] or the u-join method in [13].

A key advantage of the inverse rules algorithm is its conceptual simplicity. The inverse rules can be constructed in advance in polynomial time, independent of a particular query. However, even if the computational cost of constructing rewriting is polynomial, the rewriting contains rules that may cause accessing irrelevant views. The problem of eliminating irrelevant rules has exponential time complexity. Another drawback is that evaluating the inverse rules over the source extension may invert some useful computation done to produce the views. Hence, much of the computational advantage of exploiting the materialized view is lost due to recomputing the extensions of the database relations.

Some rewriting algorithms based on use of inverse rules have been presented in the presence of functional and/or inclusion dependencies [4][5][6][7]. Due to the limitation mentioned above, we are interested in rewriting algorithm based on use of buckets in the presence of functional dependencies.

5. Conclusions

In this paper, we have considered the problem of query rewriting in data integration by using the approach of answering query using views. The algorithms for this problem can be divided into two groups, bucket-based algorithms and inverse rules-based algorithms. The algorithms based on inverse rules have considered the problem of query rewriting in the presence of functional dependencies and/or inclusion dependencies in a mediated schema. However, there has been no bucket-based algorithm for this problem. In this paper, we consider query rewriting in the presence of functional dependencies with lossless join decomposition in a mediated schema. We first analyse the unification process in the MiniCon algorithm, find out the reasons why unification fails in the MiniCon algorithm, and then present a bucket-based algorithm to overcome the deficiency of the MiniCon algorithm. The problem of missing query rewritings which occurs in the previous bucket-based algorithms in the presence of functional dependencies with lossless join decomposition is avoided. Our main contribution is the extension of the MiniCon algorithm to query rewriting in this context.

References: