IMPROVING VIEW SELECTION IN QUERY REWRITING USING DOMAIN SEMANTICS

Qingyuan Bai¹, Jun Hong², and Michael F. McTear¹
¹School of Computing and Mathematics, University of Ulster, Newtownabbey, Co.Antrim, BT37 0QB, UK
²School of Computer Science, Queen’s University of Belfast, Belfast, BT7 1NN, UK

Keywords: Data integration, query rewriting using views, domain semantics, view selection

Abstract: Query rewriting using views is an important issue in data integration. Several algorithms have been proposed, such as the bucket algorithm, the inverse rules algorithm, the SVB algorithm, and the MiniCon algorithm. These algorithms can be divided into two categories. The algorithms of the first category are based on use of buckets while the ones of the second category are based on use of inverse rules. The bucket-based algorithms have not considered the effects of integrity constraints, such as domain semantics, functional and inclusion dependencies. As a result, they might miss query rewritings or generate redundant query rewritings in the presence of these constraints. A bucket-based algorithm consists of two steps. The first step is called view selection that selects views relevant to a given query and puts the views into the corresponding buckets. The second step is to generate all the possible query rewritings by combining a view from each bucket. In this paper, we consider an improvement of view selection in the bucket-based algorithms using domain semantics. We use the resolution method to generate a pseudo residue for each view given a set of domain semantics. Given a query, the pseudo residue of each view is compared with it and any conflict that exists can be found. As a result, irrelevant views can be removed even before a bucket-based algorithm is used.

1 INTRODUCTION

In a data integration system, data sources are autonomous, distributed, and heterogeneous. Usually, a logical virtual mediated schema is used to make queries and describe the contents of the data source. The actual data is however stored in the data sources. To answer a user query, we need to reformulate it into new queries over the data source schemas in order to get access to the data sources. This process is called query rewriting.

In general, there are two main approaches to query rewriting, i.e., Global As View (GAV) and Local As View (LAV). As stated in (Levy, 2001), the LAV approach is more suitable for a data integration system in a dynamic environment. Hence, we will focus on the LAV approach. Query rewriting using views on the LAV approach is closely related to the problem of answering queries using views, which has recently received considerable attention (Levy, 2001).

So far, there have been a number of rewriting algorithms presented. These algorithms can be divided into two categories, bucket-based algorithms and inverse rule-based algorithms. They can generate all the query rewritings in the absence of integrity constraints. However, if there are any integrity constraints in a mediated schema, a bucket-based algorithm might miss query rewritings or generate redundant query rewritings, because it does not consider the effects of these integrity constraints.

In the context of traditional databases, integrity constraints are the rules enforced on a database schema. From integrity constraints, some relationships among the relations in a database schema can be inferred. As mentioned previously, in a data integration system, data sources are described in terms of a mediated schema. Therefore, if there are integrity constraints in the mediated schema, some special relationships among the data sources can be found, which are useful in query processing.

There are three most ubiquitous types of integrity constraints enforced on a database schema: domain semantics, functional dependencies, and inclusion dependencies (DSs, FDs and INDs for short respectively). There have been some inverse rule-based rewriting algorithms that address the problem...
of query rewriting using views in the presence of functional and/or inclusion dependencies (Duschka et al. 2000, Gryz, 98, 99). A logic-based approach for the problem of query rewriting using views (Grant and Minker, 2002) is presented, where there are functional and inclusion dependencies in a mediated schema and a resolution method is used to generate all the possible query rewritings. However, there has been no bucket-based algorithm to address the problem of query rewriting in the presence of domain semantics. This paper will address this issue.

The following two examples show that the MiniCon algorithm, which is a bucket-based algorithm and the best one among the existing algorithms (Pottinger and Levy, 2000), generates a redundant rewriting or misses a query rewriting because domain semantics has not been considered.

**Example 1 (Adapted from (Mitra, 2001)).** Suppose that there are five data sources as follows:

\[ V_1(Seller) \] carCars, sellsCars

\[ V_2(Car, S_2) \] carCars, sellsCars

\[ V_3(Car) \] carCars, dealersCars

\[ V_4(Dealername, State) \] dealersCars

\[ V_5(Union) \] memberCars, CA's Dealer Union

The logical predicates, \( car(carType) \), \( dealer(D_3) \), located(D_3, “CA”), \( sells(D_3, Car_1) \), \( dealer(D_4) \), located(D_4, State_4) are defined in a mediated schema.

Assume that there exists domain semantics:

\( (x \neq “TESCO”) \rightarrow (x, y, z) \) represents the ith subgoal of Q.

We can substitute the comparison, \( 1000 < S_ID < 2000 \) with \( “CS” \), \( 2000 < S_ID < 3000 \) with \( “EN” \), \( 3000 < S_ID < 4000 \) with \( “BI” \). These rules tell us that S IDs in a particular department are restricted to a specific range.

A query is to ask for students’ names from the Computer Science department, i.e.,

\[ Q'S Name \] Student(S_ID, SName, Gender, Dept, RegDate).

The MiniCon algorithm fails to form any MCD for the given query over either \( V_1 \) or \( V_2 \), because comparisons are not consistent between either Q and \( V_1 \) or \( V_2 \). However, we can see that there exists a query rewriting as follows:

\[ Q'(SName) \rightarrow V_1(SName).

The reason is that we can substitute the comparison, \( Dept = “CS” \), in Q with equivalent comparisons, \( 1000 < S_ID < 2000 \) as shown in domain semantics (1). Then using the MiniCon algorithm, we can get the above rewriting for the query.

The rest of the paper is organized as follows. In the next section, we have a brief overview of the related work. In Section 3, preliminaries of query rewriting using views and semantics query optimization are given. In Section 4, we discuss query rewriting using views in the presence of semantic constraints. In Section 5, we conclude the paper.

## 2 RELATED WORK

As stated in Section 1, there are two categories of query rewriting algorithms, i.e., bucket-based rewriting algorithms and inverse rule-based rewriting algorithms. The key idea underlying the inverse rule-based algorithms 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 a set of views V is simply the composition of Q and the inverse rules for V using the transformation method (Duschka et al., 2000), the unification-join method (Qian, 1996), or the resolution method (Grant and Minker, 2002).

A bucket-based algorithm consists of two stages. The first stage is called view selection that selects the views relevant to a given query and puts the views into the corresponding buckets. The second stage is to generate all the possible query rewritings by combining a view from each bucket. View selection is based on containment mapping from the given query to each view. There are three representative algorithms.

**Bucket algorithm (Levy et al., 1996a, 1996b):**

Given a query, a bucket is first created for each subgoal of the query. A view is put in the bucket if it can be unified with the subgoal in the query. Next, candidate query plans are generated by combining a view from each of the buckets. These plans are then verified using containment tests.

**SVB algorithm (Mitra, 2001):**

A non-distinguished variable that appears in more than one subgoal of a query is called a shared variable. In the SVB algorithm, 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, containing a shared variable, to Q in some subgoals in a view. Once all the buckets are created, the algorithm generates rewritings by combining views from buckets which contain disjoint sets of subgoals of Q.

**MiniCon algorithm (Pottinger and Levy, 2000):**

In the first phase of the MiniCon algorithm, a MiniCon Description (MCD for short) for a query Q over a view V is formed to contain a set of subgoals in Q and the mapping information. In fact, a MCD takes the role of a bucket in the bucket algorithm and the SVB algorithm. The MCDs and the minimum MCDs in the MiniCon algorithm correspond to the single-subgoal buckets and the shared-variable buckets in the SVB algorithm respectively. In the second phase, the MiniCon algorithm combines the MCDs to generate query rewritings.

In summary, view selection is not needed in inverse rule-based rewriting algorithm, but it needs to be done in the first stage of bucket-based algorithms. As shown in the previous section, the problems of missing query rewritings and generating redundant query rewritings in bucket-based algorithms might occur if domain semantics is not taken into account.

3 PRILIMINARIES

3.1 Domain Semantics and Residues

**Domain Semantics**

In the context of databases, integrity constraints are in the forms of three main practical types of constraints, i.e., domain semantics, functional dependencies, and inclusion dependencies. The functional and inclusion dependencies are mainly used for the design of database schemas, e.g., normalization of schemas, data duplication. Domain semantics is relevant to the knowledge of a specific application domain. In this paper, we only consider the effects of domain semantics in query rewriting. In some cases, a query may even be answered without accessing a database if sufficient knowledge is contained in the domain semantics.

Domain semantics is represented in this paper using the following types of rules:

- **D<sub>1</sub> (Equivalence Proposition):** CQ<sub>1</sub> ↔ CQ<sub>2</sub>.
- **D<sub>4</sub> (Dependency rule):** R.CQ<sub>1</sub> ↔ S.CQ<sub>2</sub>.
- **D<sub>3</sub> (Production rule):** CQ<sub>1</sub> ↔ R(X), S(Y), CQ<sub>2</sub>.

where, CQ<sub>i</sub> (i=1,2) refers to the comparison expressions whose variables appear in some relations in a mediated schema.

D<sub>1</sub> means that two expressions are equivalent. D<sub>2</sub> means that if CQ<sub>2</sub> holds, CQ<sub>1</sub> should be satisfied in a database schema. The variables in CQ<sub>1</sub> and CQ<sub>2</sub> are in either relation R or relations R and S. The right-hand side of D<sub>1</sub> is a conjunction of two or more relations. CQ<sub>1</sub> in D<sub>3</sub>, i=2,3, can be null.

**Residues**

Residues are used in semantic query optimization (Chakravarthy et al., 1990) to eliminate redundant joins in a given query. In this paper, we exploit residues to remove the views irrelevant to a query.

The notion of residues, associated with the concept of subsumption, is used for semantic query optimization in the presence of integrity constraints. A clause C<sub>1</sub> subsumes a clause C<sub>2</sub> if there is a substitution θ such that C<sub>1</sub>θ is a sub-clause of C<sub>2</sub>. The refutation tree is used to test subsumption between C<sub>1</sub> and C<sub>2</sub>. C<sub>1</sub> subsumes C<sub>2</sub> if and only if there is a refutation tree that ends with the null clause.

In general, a null clause can not be obtained because integrity constraints rarely subsume relations. But integrity constraints may partially...
subsume a relation, leaving a fragment at the bottom of the refutation tree. Such a fragment is called a residue representing an interaction between a relation and an integrity constraint.

**Definition 1.** An integrity constraint IC partially subsumes an atom A, if and only if IC does not subsume ¬A, but a sub-clause of IC+ (expansion of IC) subsumes ¬A. Let C be the clause at the bottom of a refutation tree. Then (C')01 (C is a result of reducing C) is a residue of IC and A.

3.2 Query Containment and Query Rewriting Using Views

Queries and views

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

\[ Q(\vec{x}) : -R_1(\vec{x}_1), ..., -R_k(\vec{x}_k), C \]

where \( R_i(\vec{x}_i) \) are the subgoals referred to database relations, \( C_0 \) is a comparison expression. \( Q(\vec{x}) \) is the head of the query. The tuples \( \vec{x}, \vec{x}_1, ..., \vec{x}_k \) contain either variables or constants. We require that the query be safe, i.e., \( \vec{x}, \vec{x}_1, ..., \vec{x}_k \) contain either variables or constants.

**Query containment and equivalence**

The concepts of query containment and equivalence enable us to make a comparison between queries and rewritings. We say that a query \( Q_1 \) is contained in another query \( Q_2 \), 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. Containment mappings provide a necessary and sufficient condition for testing query containment. A mapping \( \phi \) from \( \text{Vars}(Q_2) \) to \( \text{Vars}(Q_1) \) is a containment mapping if

1. \( \phi \) maps every subgoal in the body of \( Q_2 \) to a subgoal in the body of \( Q_1 \), and
2. \( \phi \) 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 \( Q_1 \). The query \( Q_1 \) is equivalent to \( Q_2 \) if and only if \( Q_1 \equiv Q_2 \) and \( Q_2 \equiv Q_1 \).

**Answering queries using views**

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

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.

**Definition 2** (Maximally-contained rewriting): \( Q' \) is a maximally-contained rewriting of a query \( Q \) using views \( V \) with respect to a query language \( L \) if (1) for any database \( D \), \( Q' \not\subseteq Q \), and (2) there is no other query rewriting \( Q'' \) in the language \( L \), such that for every above database \( D \), \( Q'' \not\subseteq Q' \) and \( Q' \not\subseteq Q'' \).

4 VIEW SELECTION IN QUERY REWRITING USING DOMAIN SEMANTICS

Assume that there are a set of views \( V_i \) (i=1,2,...,n) and a set of domain semantics in the three types of rules as described in Section 3.1. As an equivalence proposition, \( D_i \) will be added to the relevant views without losing any information. \( D_2 \) and \( D_3 \) enforce constraints on certain views. As stated later, we will resolute each of \( D_2 \) and \( D_3 \) with every corresponding view \( V_i \) to get a so called pseudo residue \( PR_i \) for \( V_i \). A pseudo residue is a fragment at the bottom of the refutation tree, representing an interaction between a view and a \( D_2 \) or \( D_3 \). \( PR_i \) takes a role of the residue, but whether it can play a role in view selection also depends on a given query. According to (Chakravarthy et al., 1990), each relation has a residue for each integrity constraint, which results in several SCAs (semantically constrained axioms) in a view. However, in this paper, each \( D_2 \) or \( D_3 \) is viewed as a whole and there is only one pseudo residue for each \( D_2 \) or \( D_3 \) over a view.
4.1 Computing a Pseudo Residue

Computing a pseudo residue for $V_i$, ($i=1,...,n$) is based on the resolution method as follows:

**Case 1.** For each $D_1$ (Equivalence Proposition): $CQ_1 \leftrightarrow CQ_2$, where the variables of both $CQ_1$ and $CQ_2$ are in a relation $R$, we check whether view $V$ contains $R$. If not, then the constraint can not be enforced on $V_i$. Otherwise we check whether $CQ_1$ (or $CQ_2$) appears in $V_i$. If so, a pseudo residue is $CQ_2$ (or $CQ_1$), denoted by $ER_i=CQ_2$ (or $ER_i=CQ_1$).

**Case 2.** For each $D_2$ (Dependency rule): $R.CQ_1 \leftrightarrow S.CQ_2$, we check whether view $V$ contains $R$ and $S$. If not, then the constraint can not be enforced on $V_i$. Otherwise, we check whether $CQ_2$ is consistent with $V_i$. If so, $PR_i=CQ_1$.

**Case 3.** For each $D_3$ (Production rule): $CQ_1 \leftrightarrow R(X),S(Y),CQ_2$, we check whether view $V_i$ contains the join of relations $R$ and $S$. If not, then the constraint can not be enforced in $V_i$. Otherwise, we check whether $CQ_2$ is consistent with $V_i$. If so, $PR_i=CQ_1$.

We use the resolution method to get the $PR_i$ for each $V_i$. We construct a linear refutation tree with the body of $V_i$ as the root, using at each step an element of the right side of $D_3$ in resolution. If the tree ends with empty or the subgoals only in $V_i$, then $PR_i=CQ_1$. Figure 1 shows the process.

Figure 1: Computing a pseudo residue of $V_i$

We show the process of resolution between a view and a $D_3$ using the following example.

**Example 3.** Continuing with Example 1, we can get the pseudo residues for each view as follows:

<table>
<thead>
<tr>
<th>View</th>
<th>$V_1$</th>
<th>$V_2$</th>
<th>$V_3$</th>
<th>$V_4$</th>
<th>$V_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PR_i$</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

For simplicity, we show only the process of computing the pseudo residue of $V_3$. In this example, $CQ_1=\{\text{located}.x \neq \text{"TESCO"}\}$ and $CQ_2=\{\text{located.y = "CA"}\}$.
Thus, \( PR_3 = \{ \text{located}.x \neq \text{"TESCO"} \} \). For each of the other views \( V_i \) (\( i=1,2,4,5 \)), a linear refutation tree would not end with such subgoals that appear only in the views, then the pseudo residues of the views are null, denoted by \( \{ \} \).

### 4.2 View Selection in Query Rewriting Using Views

As mentioned in Section 2, the first step in a bucket-based algorithm is to select the views relevant to a given query. For a given query \( Q \), using domain semantics we may remove the irrelevant views or pick up certain relevant views before using any bucket-based rewriting algorithm. As a result, the soundness and completeness of query rewriting algorithms can be improved.

Given a query \( Q \), for each \( V_i \) (\( i=1,2,\ldots,n \)), if \( PR_i \) of \( V_i \) is not compatible with \( CQ \) in \( Q \), then the view \( V_i \) is irrelevant to \( Q \) and should not be considered when rewriting the query. Note that each pseudo residue is in the form of a comparison expression and so are both \( CQ \) in \( Q \) and ones in \( V_i \), we need only to check the consistency between two comparison expressions. Two comparison expressions, \( CQ_1 \) and \( CQ_2 \), are comparable if their variables are in the same relations. \( CQ_1 \) is consistent with \( CQ_2 \) if they are equivalent or their conjunct, \( CQ_1 \land CQ_2 \), is not always false for any values of the variables involved. In the following algorithm, all comparison expressions are comparable. Otherwise, the corresponding process would have stopped.

Our algorithm consists of two steps. In the first step, some irrelevant views, in the presence of domain semantics, with respect to \( Q \) are removed by comparing \( CQ \) in \( Q \) with the pseudo residues of views. In the second step, other irrelevant views, in terms of unification, with respect to \( Q \) are removed by unifying \( Q \) with the definitions of views, which is used in any previous bucket-based rewriting algorithm.

**Algorithm:** View Selection in Query Rewriting Using Domain Semantics

**Input:** A given query \( Q \), a set of views \( V_i \) (\( i=1,2,\ldots,n \)) associated with a set of pseudo residues \( PR_i \) (\( i=1,2,\ldots,n \)).

**Output:** A set of buckets or MCDs containing views \( V_j \) (\( j=1,2,\ldots,t \leq n \)) which are relevant to \( Q \) both in the presence of domain semantics and in terms of unification.

**Methods:**

**Step 1:** Removing the irrelevant views with respect to \( Q \) by comparing \( CQ \) with pseudo residues.

For each \( V_i \) (\( i=1,2,\ldots,n \)) associated with a pseudo residue \( PR_i \), we proceed according to the following cases:

- **Case 1:** The pseudo residue is \( ER_i \): We check whether \( CQ \) in \( Q \) is consistent with \( CQ_1 \) or \( CQ_2 \) in \( D_i \). If not, the view \( V_i \) is irrelevant to \( Q \), i.e., \( \mathcal{V} = \mathcal{V} - \{ V_i \} \). Otherwise, \( ER_i \) is added into the definition of view \( V_i \).

- **Case 2:** The pseudo residue is \( PR_i \): We check whether \( CQ \) in \( Q \) is consistent with \( PR_i \) of \( V_i \). If not, the view \( V_i \) is irrelevant to \( Q \), i.e., \( \mathcal{V} = \mathcal{V} - \{ V_i \} \).

**Step 2:** Selecting relevant views with respect to \( Q \) by unifying \( Q \) with the definitions of views.

For each view \( V_i \) in \( \mathcal{V} \), a set of the buckets is built according to the SVB algorithm or a set of the MCDs is formed according to the MiniCon algorithm. This procedure is based on unification from \( Q \) to views. The views in the buckets or the MCDs are relevant to \( Q \) in the presence of domain semantics and in terms of unification.

**End.**

**Example 4.** Continuing with Example 3, the pseudo residue of \( V_3 \) is \( PR_3 = \{ \text{located}.x \neq \text{"TESCO"} \} \). Note that \( x \) is the first argument in relation \text{located}, resulting in the pseudo residue of \( V_3 \) is \( D \neq \text{"TESCO"} \). It conflicts with the comparison in \( Q \). Hence, the view \( V_3 \) is irrelevant to the given query \( Q \) in the presence of domain semantics. For the rest of views, we use the MiniCon algorithm to form the MCDs as follows:

<table>
<thead>
<tr>
<th>( \mathcal{V} )</th>
<th>( \mathcal{G} )</th>
<th>( \mathcal{V} )</th>
<th>( \mathcal{G} )</th>
<th>( \mathcal{V} )</th>
<th>( \mathcal{G} )</th>
<th>( \mathcal{V} )</th>
<th>( \mathcal{G} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V_2 )</td>
<td>( g_1 )</td>
<td>( V_2 )</td>
<td>( g_4 )</td>
<td>( V_4 )</td>
<td>( g_2 )</td>
<td>( V_4 )</td>
<td>( g_3 )</td>
</tr>
</tbody>
</table>

There are two views relevant to \( Q \) in terms of unification. However, there is only one query rewriting formed by combining MCDs so that all subgoals of \( Q \) can be covered.

\( Q(X) \land \neg V_2(X,D), V_4(D, \text{"CA"}), D = \text{"TESCO"} \).
5 CONCLUSIONS

In previous bucket-based rewriting algorithms, for a given query Q, view selection is done by unifying Q with the definition of views. In other words, the meaning of “relevant to Q” is in terms of unification. We found that there is another explanation about “relevant to Q”, i.e., in the presence of domain semantics. That is, we can remove the irrelevant views which could not be found in any bucket-based algorithm. Also, in some cases, we can avoid the problem of missing relevant views, which occurs in bucket-based algorithms.

In this paper, we have aimed to solve the problems of missing query rewritings and redundant query rewritings in bucket-based rewriting algorithms so that we can improve the soundness and completeness of these algorithms. In the presence of domain semantics in a mediated schema, we first compute the pseudo residue for each constraint over the views using the resolution method. In fact, what we have done is to transfer the integrity constraints over the relations of the mediated schema into a rule over a view. As a result, for a given query, we can determine which view is irrelevant to the query, in the presence of domain semantics, by making a comparison between the pseudo residue of a view and the comparison expression of the query. The pseudo residues can be calculated in advanced, which means that the total increased computation in Step 1 in our algorithm is only in polynomial size of $|D|^*|V|$, where $|D|$ and $|V|$ are the number of domain semantics in a mediated schema and of the views respectively. This process is useful for query rewriting, which has been shown by examples in Section 1.

REFERENCES


