An Algorithm for Answering Queries Efficiently Using Views

Prasenjit Mitra
Infolab, Stanford University
Stanford, CA, 94305, U.S.A.
mitra@db.stanford.edu

September, 1999

Abstract Algorithms for answering queries using *views* have been used in query planning to answer queries posed to knowledge bases, databases, and information systems. However, these algorithms do not scale well when the number of views increases. Three known algorithms, the bucket algorithm, the inverse-rules algorithm and an algorithm suggested by Leser have been used to reformulate queries before generating their answers. The bucket algorithm, predominantly used to reformulate queries, generates a candidate rewriting to a query using views, and checks that the rewriting is contained in the original query. An exponential conjunctive-query-containment-test needs to be performed to check each candidate rewriting. Using a few extra buckets (shared-variable-buckets), we show how we can avoid the conjunctive-query-containment-test. This paper presents a scalable query rewriting algorithm - the shared-variable-bucket(SVB) algorithm. Experimental results demonstrate its superiority over other known algorithms.
Keywords: database, views, query rewriting, optimization

1 Introduction

With the advent of the world-wide web, a large number of information sources are available to users. A number of web-sites use forms or other interfaces that provide views of the underlying information. Traditionally, it was left to the end-user to integrate information from the various sources manually. Today, information agents can find *rewritings* [1] of user queries using views of each information source. Mediated systems like Information Manifold [2], Infomaster [3] and Tsimmis [4] spend a nontrivial amount of time in query rewriting especially if the number of sources is large. We present an algorithm that enables such a system to rewrite queries efficiently, reduces the response time to users, improves its scalability and makes it feasible for a system to handle a large number of information sources.

Views and user queries are expressed as conjunctions of abstract, global predicates [5]. Answering queries from the data available in the information sources requires rewriting the queries as the conjunction of the available views. Computing the answer to the query then involves computing a join of the data from the relevant views. Computing such joins is costly. Therefore the query needs to be rewritten using a *locally minimal* set of views, i.e., if any view is removed from the solution, it will no longer satisfy the query [1].

Several algorithms have been developed to solve the problem of answering queries using views. Notable among them are the *bucket algorithm* [6] first used in the Information Manifold system [2], the inverse-rules algorithm [7], and an algorithm proposed by Leser [8].

The bucket algorithm attempts to rewrite a query using views instead of the logical schema predicates. Views that have subgoals that can be unified with query subgoals are put in the buckets corresponding to the query subgoals. Candidate solutions are constructed by selecting views from each bucket and conjoining them. The validity of a candidate solution is verified using an conjunctive-query-containment test.

The Leser algorithm constructs *partial containment mappings* [8] i.e., mappings from the variables in a query subgoal to the variables in a view subgoal, while unifying the two subgoals. Views are put into buckets corresponding to the query subgoals upon unification. After a solution is generated by selecting a view from

each bucket, the partial containment mappings are checked to determine whether they are *c-compatible* [8], in order to validate the compatibility of constants in the partial mappings. Leser's algorithm, though faster than the bucket algorithm, cannot guarantee the soundness of all generated solutions, and is thus limited in its use.

The other known algorithm, the inverse rules algorithm, is too expensive to use in a practical information integration system because of the complexity of constructing solutions from the inverse rules.

The bucket algorithm has two avoidable sources of complexity. First, even when the number of sound solutions is small, the bucket algorithm may generate a large number of candidate solutions and then reject them. The second source of complexity is the exponential conjunctive-query-containment test that is used to validate each candidate solution. This paper highlights an algorithm that avoids the two drawbacks of the bucket algorithm. While constructing buckets, it considers the equality constraints introduced by *shared variables*, that is, variables that occur across multiple subgoals, and constructs special buckets, shared-variable-buckets, in order to handle such constraints. By using a few extra buckets, we ensure that all generated solutions are sound solutions and thus avoid the exponential conjunctive-query-containment test.

The rest of the paper is organized as follows. Section 2 discusses the preliminaries. Section 3 details the Shared-Variable Bucket Algorithm. Section 4 contains a brief analysis of its performance. Section 5 discusses variations of the algorithm. Section 6 outlines the related work and we conclude in section 7 by noting the contributions of this paper.

2 Preliminaries

Views and queries are represented using the logical rule notation described in [9]. A rule has the form

$$q(\overline{X}) :- r_1(\overline{X}_1), \ldots, r_n(\overline{X}_n) \tag{1}$$

where q, and $r_1, \ldots, r_n$ are predicate names and $\overline{X}, \overline{X}_1, \ldots, \overline{X}_n$ are either variables or constants. A *subgoal* $r_1(\overline{X}_1)$ represents a *logical global predicate*. The *head* of the rule is $q(\overline{X})$. The *body* of the rule is the conjunction of subgoals $r_1(\overline{X}_1), ..., r_n(\overline{X}_n)$. For the purpose of this work, all rules will be assumed to be *safe*, i.e., if any variable occurs in the head of a rule, it must also occur in the rule's body.

Example 1 We use a running example to illustrate our algorithm.

```
Query:       Q(X) :- car(X),dealer(D),located(D,"CA"),sells(D,X)

CarSellers1:  V1( Seller ):-car(Car1), sells(Seller,Car1)
CarSellers2:  V2( Car2, S2 ) :- car(Car2), sells(S2,Car2)
CACars:       V3( Car3 ):- dealer(D3), located(D3, "CA"), sells(D3, Car3)
CarDealers:   V4( D4, State ):- dealer(D4), located(D4,State)
CADealerUnion:V5( Union ):- member(D5, Union),dealer(D5),located(D5,"CA")
```

The logical predicates are *car*, *dealer*, *located* and *sells*. Queries and views are defined as conjuncts of these predicates. The query asks for all cars sold by dealers in California. We want to rewrite the query using the available views. There are five views of various information sources, e.g., the view $V2$, lists tuples of cars and their sellers. As is the case with information sources on the world-wide web, a view is not exhaustive, i.e., the view $V2$ is not guaranteed to contain information about all cars that were sold.

In the rest of the discussion we use the following terms as defined in [1].

A *rewriting* of a query Q is a conjunctive query that produces the answers that are also produced by Q for any given database. For example, $Q(X) :- V2(X), V3(X)$ is a rewriting of the query $Q(X)$.

A *distinguished variable* is a variable that occurs in the head of a rule and a *non-distinguished variable* is a variable that occurs in the body of a rule but not in the head. A non-distinguished variable is existentially quantified in the rule. For example, $Union$ is a distinguished variable while $D5$ is non-distinguished in view $V5$.

A subgoal $g_1(\overline{X})$ *covers* another subgoal $g_2(\overline{Y})$ if and only if the following conditions hold: (1) if the argument at position i in $g_2(\overline{Y})$ is a distinguished variable or a constant, then $g_1(\overline{X})$ also has that variable or the constant at the same position and (2) if the arguments at positions i and j are equal in $g_2(\overline{Y})$, then so

car(X)	dealer(D)	located(D,'CA')	sells(D,X)
V1(S1')	V3(X)	V3(X)	V1(S2')
V2(D1',X)	V4(D2', 'CA')	V4(D3', 'CA')	V2(D4', X)
	V5(U1')	V5(U2')	V3(X)

Table 1: The buckets

are the arguments at positions i and j in $g_1(\overline{X})$. For example, $located(D5, "CA")$ covers $located(D, "CA")$ while $car(Car1)$ does not does not cover the query subgoal $car(X)$.

2.1 Containment of Conjunctive Queries

Conjunctive queries and their containment was first studied in [10]. A variable in argument position i in one subgoal *maps* to a variable in the same argument position of another subgoal provided they have the same predicate. A conjunctive query Q_1 is said to be *contained* in another conjunctive query Q_2 if and only if all the tuples obtained as the output of the query Q_1 on a database are also obtained as the output of the query Q_2 on the same database. Alternatively, Q_1 is contained in Q_2 if and only if there exists a *containment mapping* from Q_2 to Q_1, i.e., a mapping exists from the variables from Q_2 to the variables in Q_1 such that each subgoal in the query Q_2 has been mapped to one in Q_1 and the head of query Q_2 maps to the head of Q_1. Determining containment of conjunctive queries is a NP-complete problem.

A rewriting Q' of a query Q using a set of views V is a *maximally contained rewriting*, if and only if (1) $Q'(V(D))$, the tuples obtained by evaluating Q' using the set of views V on any database D is the subset of $Q(D)$, i.e., the tuples obtained by evaluating Q on the same database D, and (2) there exists no other query Q_1 such that $Q'(V(D))$ is a subset of $Q_1(V(D))$, which in turn is a subset of $Q(D)$, for any database D, and there exists a database D_1 such that $Q'(V(D_1))$ is a strict subset of $Q_1(V(D_1))$. In this paper, we discuss an algorithm to obtain a maximally contained rewriting (as opposed to an equivalent rewriting - one where the tuples produced by the rewriting using the views on any database are exactly the same as those produced by the query on the database) of a conjunctive query using views.

2.2 The Bucket Algorithm

The bucket algorithm proceeds in two stages. Initially, a bucket is created for each query subgoal. A view is put in the bucket corresponding to a query subgoal if that subgoal can be unified with a subgoal in the view definition. In the second stage of the algorithm, candidate solutions are generated by picking one conjunct (view) from each bucket. The candidate solutions are then verified using containment tests to see that they are indeed contained in the user query. If the view contains several subgoals that can be unified with a subgoal from the query, the view occurs several times in the bucket of that subgoal.

Table 1 lists all the buckets generated by the bucket algorithm for the example shown above. The algorithm includes the views $V1$ and $V2$ in the bucket for the query subgoal $sells(D, X)$ using the mappings $\{D \Rightarrow Seller, X \Rightarrow Car1\}$ and $\{D \Rightarrow S2, X \Rightarrow Car2\}$ from the query subgoal to the subgoals $sells(S2, Car2)$ and $sells(Seller, Car1)$ respectively.

In the next stage of the algorithm, candidate solutions are generated by selecting one view from each bucket and rewriting the query as their conjunction. The number of candidate solutions that the bucket algorithm generates by performing a cartesian product of the views in the buckets is (2 x 3 x 3 x 3) =54. Each solution is then checked for containment in the original query. In the next section, we show how we can prune out views from the buckets such that only sound solutions are generated.

3 The Shared-Variable Bucket Algorithm

Like in the bucket algorithm, the task of query rewriting is accomplished in two steps:

- Bucket Construction
- Solution Generation

Also like the bucket algorithm we create buckets for each query subgoal. However, a view is only put in the bucket after a more stringent test. We also introduce a small number of shared-variable buckets, if required, to account for the equality constraints imposed by shared variables (variables occurring in multiple subgoals).

- The "trick" that allows us to avoid a containment test is that in addition to buckets for individual subgoals, we create buckets associated with shared variables. Each bucket contains only views that cover all the subgoals in which the shared variables, representing the bucket, appear.

In the solution generation stage, initially, a set of buckets is chosen such that each subgoal is represented by one and only one bucket in the set. From each bucket a view is then selected. Consequently, the solution to the query is expressed as a conjunctive query whose body is the conjunct of the selected views.

3.1 Definitions

In the description of the algorithm below we use the following terms and symbols. Q is a conjunctive query and R is a rewriting for the query Q. Qv_{ij} is the j^{th} variable in the i^{th} subgoal g_i in the query. V is a view that can be used to construct a solution and Vv_{ij} is the i^{th} variable in the j^{th} subgoal v_j in the view. We also use the term Qv_i to refer to the i^{th} variable in a query subgoal g of interest and Vv_i to refer to the i^{th} variable in a view subgoal v.

A *shared variable* is a variable that occurs in more than one subgoal in the body of a conjunctive query, e.g., in the example shown above, D is a shared variable in the query $Q(X)$.

The *expansion* $E(V(\overline{X}))$ *of a view* $V((\overline{Y}))$ is the body of the view V with the distinguished variables $(\overline{Y})$ replaced by $(\overline{X})$. For example, $E(V2(Mycar, Aseller))$ is $car(Mycar), sells(Aseller, Mycar)$.

The *expansion* $E(R)$ *of a rewriting* R is a conjunctive query where each view $V(\overline{X})$ occurring in R has been replaced by $E(V(\overline{X}))$. For example, the expansion of the rewriting $Q(X)$ is: $E(Q(X)) :: Q(X) : -car(X), sells(S2, X), dealer(D3), located(D3, `CA'), sells(D, X)$ where $Q(X) : -V2(X), V3(X)$. Note that the expansion is obtained by simply expanding the individual views in the rewriting.

We use the notation $\overline{X}[V_1|V_2]$ to indicate that the variable V_1 in the set of variables $\overline{X}$ has been replaced by the variable V_2.

Apart from views covering subgoals, we also need views to cover shared variables before they can be put in a shared variable bucket.

Definition 1 *A view* $V(\overline{X})$ covers *a query variable* Qv *iff there exists a mapping* μ *from the query* Q *to* V *such that* $\forall i, j, k, l$, *if* $Qv_{ij} = Qv_{kl} = Qv$, *then* $\mu(Qv_{ij}) = \mu(Qv_{kl}) = Vv$ *where* Vv *is a variable in* $E(V(\overline{X}))$.

If a view covers a query variable, then it ensures that a mapping exists from the query variable to a unique variable in the expansion of any generated solution that uses the view. In our example, the view $V3(X)$ covers the shared variable D in the query since the individual occurrences of D in the subgoals $dealer(D)$, $located(D, `CA')$ and $sells(D, X)$ all map to the same view variable $D3$ in $E(V3(X))$. In a mapping from the query to $V4$ and $V5$ the occurrences of $D$ in $dealer(D)$ and $located(D, `CA')$ map to $D4$ and $D5$ respectively. Both views do not contain the subgoal $sells(D, X)$ and therefore the third occurrence of D in the query is not mapped to any variable. Thus views $V4(D4, S)$ and $V5(U)$ do not cover D.

Definition 2 *A view* $V(\overline{X})$, covers *a subgoal* $q(\overline{Y})$ *using a subgoal* $v(\overline{Z})$ *if and only if the subgoal* $v(\overline{Z})$ *is in the expansion* $E(V(\overline{X}))$ *and* $v(\overline{Z})$ *covers the subgoal* $q(\overline{Y})$.

A view covers a query subgoal if it has a subgoal in its body that covers the query subgoal. In our running example, the view $V3(X)$ covers the query subgoal $sells(D, X)$ using the subgoal $sells(D3, Car3)$.

3.2 Bucket Construction

We now illustrate how we create buckets. Views export the values of only the distinguished variables. The shared-variable bucket algorithm, puts a view in a bucket only if no distinguished query variables maps to an non-distinguished view variable.

All variables in the definitions of the query and views are initially renamed so that no two conjunctive queries share the same variable. Unique naming avoids the imposition of any unintended equality constraints.

We insert views into two types of buckets. A bucket is said to *represent* a subgoal if all views in the bucket cover the subgoal. A single subgoal bucket(SSB) represents a single subgoal and a shared variable bucket(SVB) represents multiple subgoals in which the shared variables occur in.

3.2.1 Renaming of Distinguished View Variables

Before a view is put in a bucket, its head variables are renamed while unifying a view subgoal with a query subgoal. In our running example, we can unify the last query subgoal $sells(D, X)$ with the view subgoal $sells(S2, Car2)$ in view $V2$ using the mapping $\{D \Rightarrow S2, X \Rightarrow Car2\}$. We use the inverse mapping to construct the view head $V2(X, D)$, before putting it in the bucket for the last query subgoal. Now, consider the first query subgoal $car(X)$. While considering $V2$ for the bucket corresponding to this subgoal, we see that it can be unified with the view subgoal $car(Car2)$ in view $V2$ using the mapping $\{X \Rightarrow Car2\}$. Using the inverse mapping, we find that the head variable $Car2$ in view $V2$ can be replaced by the query variable X, but the inverse mapping does not restrict what we can use to replace the head variable $S2$ with. In such cases, we create a unique variable $D1'$ and construct the view head $V2(X, D1')$ to be put in the bucket corresponding to the first query subgoal. If the "free" variables are not uniquely renamed, there might be an accidental equality between such variables that imposes unnecessary constraints in the rewriting.

The detailed algorithm that renames the view heads is given below. In the algorithm listed below, we consider constants to be distinguished.

Algorithm 1 *Renaming of Distinguished View Variables*
For each distinguished variable Vv:
Let $Qv_1, \ldots, Qv_n$ be query variables that map to a distinguished variable Vv in view V and $i, j \in (1, \ldots, n)$. If n=0, then rename Vv uniquely and continue.
Else

- *If $\exists i : Qv_i$ is distinguished then if:*
  1. *$\forall k, l$: if Qv_k, Qv_l are constants, then $Qv_k = Qv_l$.*
  2. *and if Qv_j is not distinguished then V covers Qv_j.*

  Then rename V_v to Qv_i, and $\forall j \neq i$: If Qv_j is distinguished, addpredicate$(Qv_j = Qv_i)$.
  Else return error;
- *Else*

  If V covers all Qv_i, rename V_v to Qv_i.
  Elseif V covers all Qv_i except one (say Qv_j), rename V_v to Qv_j.
  Else return error;

If only one query variable maps to a distinguished view variable, the view variable is replaced by the query variable in the head of the view. If multiple query variables map to a distinguished view variable, we have to choose one of those. This choice is made based on a priority - a variable (or constant) having higher priority is chosen. Usually, the priority order is: constants, distinguished variables, shared non-distinguished variables and non-shared non-distinguished variables.

If multiple distinguished query variables map to the same view variable we replace the view variable with any one of them and introduce an equality predicate between the distinguished query variables in the solution.

If a distinguished query variable and an non-distinguished query variable map to the same view variable, we replace the view variable with the distinguished query variable only if the view covers the non-distinguished query variable. If the view does not cover the non-distinguished query variable, we do not insert the view into the bucket.

If multiple non-distinguished query variables map to a view variable, we check to see if the view covers at least all but one the non-distinguished query variable. We then replace the view variable with the non-distinguished query variable that the view does not cover. If the view covers all non-distinguished query variables that map to the view variable, then the view variable is replaced by any of the query variables that map to it.

The function *addpredicate* creates or appends an equality constraint to a list of existing equality constraints among query variables. These equality constraints must be added to the solution in which the view is used.

In the next two subsections we give the necessary conditions a view must satisfy before it is inserted in any of the buckets. It is important to remember that the first condition any view needs to satisfy before it can be put in any bucket is that there must exist an appropriate renaming of its head variables - i.e., the algorithm stated above must not throw an error.

3.2.2 Single-Subgoal Buckets

We now present the conditions a view must satisfy before it can be inserted in a single-subgoal bucket.

Property 1 *$V(\overline{X})$ can be inserted into a bucket representing a single subgoal $q(\overline{Y})$ only if there exists a subgoal $v(\overline{Z})$ in $E(V(\overline{X}))$ such that:*

1. *$V(\overline{X})$ covers $q(\overline{Y})$ using $v(\overline{Z})$, and*
2. *If Qv is a non-distinguished shared query variable in $\overline{Y}$, it does not map to a non-distinguished view variable Z in $\overline{Z}$.*

In our running example, the view $V4$ cannot be used to cover the subgoal *dealer* since the non-distinguished query variable D, which maps to the non-distinguished view variable $D4$, is shared. $V4$ contains information about a dealer and the state it is located in but does not contain any information about whether the dealer sells a car. If there is a solution R, where $V4$ covers *dealer*, some other view, say V' must cover the predicate *sells*. In the expansion of the solution $E(R)$, D will map to two variables, $D4$ and at least another variable, say D' present in the expansion of the view V'. $D4$ is unique to view $V4$ and is not equal to D'. Therefore, such a rewriting R does not assure that D will map to a unique view variable. Thus, R is not contained in the query Q and is not a sound rewriting.

$V2$ and $V3$ cover the subgoals $sells(D, X)$ and $V2$ covers the subgoal $car(X)$.

The novelty of our algorithm is the introduction of the second condition which a view does not necessarily need to satisfy before being inserted in a bucket in the bucket algorithm. The stringent check enforced in our algorithm ensures that we avoid generating unsound candidate solutions and obviate the necessity of the exponential conjunctive-query-containment-test.

The algorithm needs to ensure that in the mapping from the query to its rewriting, all query variables map to a unique variable in the expansion of the rewriting. We need to ensure that the non-distinguished shared query variables that do not appear in the rewriting, map to a unique variable in the expansion of the rewriting. This is ensured by the use of shared-variable buckets.

3.2.3 Shared-Variable Buckets

The algorithm, initially, tries to put a view in a single-subgoal bucket. However, if one or more non-distinguished shared query variables map to non-distinguished view variables, the second condition in the check for single-subgoal buckets is not satisfied. We, then, check to see if the view can be inserted in a shared-variable bucket corresponding to the shared query variables. Views that cover all subgoals that the shared variables occur in are inserted into shared-variable buckets.

Property 2 *$V(\overline{X})$ is inserted in a shared variable bucket representing the shared variables $\overline{Y}$ only if for each subgoal $q(\overline{Z})$ that any shared variable $Y \in \overline{Y}$ occurs in*

1. *There exists a subgoal $v(\overline{U})$ in $E(V(\overline{X}))$ that covers $q(\overline{Y})$ and*

car(X)	dealer(D)	located(D,'CA')	sells(D,X)	SVB(D):g2,g3,g4
V2(X,D1')	V4(D2', 'CA')	V4(D3', 'CA')	V2(X,D4')	V3(X)

Table 2: The buckets

2. *for each non-distinguished shared query variable Qv, in $q(\overline{Z})$ that maps to a non-distinguished variable X in $E(V(\overline{X}))$, $V(\overline{X})$ covers Qv.*

We check to see if view $V3$ covers *dealer*. The query variable D maps to an non-distinguished view variable $D5$. This forces us to check if $V3$ covers $D5$. $V3$ indeed does cover $D5$ and any tuple $V3(X)$ guarantees the existence of a dealer in California who sells the car X. The mapping of D to $D5$ leads us to create a bucket corresponding to D that represents the set of subgoals ($dealer(D), located(D, 'CA')$, and $sells(D, X)$).

$V4$ is inserted in the bucket representing the subgoal $dealer(D)$ and $located(D, 'CA')$. The buckets generated by our algorithm is illustrated in Table 2. The last column lists the shared variable bucket corresponding to the shared variable D and represents the second, third and fourth subgoals of the query (the subgoals that D occurs in).

To illustrate the general rule, let us consider the another example

Example 2 `Q(X) :- conn(X,Y,Z),city(Y),dirConn(Z,A), city(A)`

```
V1(E):- conn(E,B,C),city(B), dirConn(C,D)
V2(E):- conn(E,B,C),city(B), dirConn(C,D), city(D)
```

The query wants all X, such that X is connected to Z via a city Y and Z is directly connected to a city A. Initially, we consider the view $V1$ for insertion into the single-subgoal bucket corresponding to the query subgoal $conn(X, Y.Z)$. Since Y and Z are shared non-distinguished query variables that map to non-distinguished view variables, B and C, respectively, we need the view to cover Y and Z. So we attempt to cover the second subgoal, $city(Y)$, using $V1$ and we find a mapping to $city(B)$. Similarly, $dirConn(Z, A)$ in the query maps to $dirConn(C, D)$ in view $V1$. However, now again the shared non-distinguished variable A maps to a non-distinguished view variable, thus we require that $V1$ has to cover A. A occurs in the query subgoal $city(D)$. There exists no mapping from Q to V where A is covered, since if we try to cover the last query subgoal with the second view subgoal, A maps to two view variables D and B, which is not acceptable. Therefore, $V1$ cannot be used to cover any query subgoal.

It is easy to see that the view $V2$ can be inserted into the shared-variable bucket for the subgoals no. 1,2,3,4 of the query corresponding to the shared-variables Y, Z, and A.

A shared-variable bucket covers multiple query subgoals. If two different sets of shared variables occur in the same set of subgoals, we do not create two shared-variable buckets but use the same shared-variable bucket. Since for the purposes of the solution generation stage of the algorithm, we only look at what subgoals the buckets cover, we do not need to create two different buckets for the different sets of shared variables if they occur in the same set of query subgoals.

The shared-variable bucket algorithm and avoids the exponential conjunctive-query-containment-test by spending

3.2.4 Minimality of Buckets

While constructing buckets, the algorithm tries to include as small a set of subgoals as it can without violating the conditions under which a view can be put in a shared-variable bucket. To understand why we construct such minimal shared-variable buckets let us consider the following example:

Example 3 `Q(X,Y) :- red(X), green(Y)`

```
V(X,Y) :- rec(X), green(Y), path(X,Y)
```

That is, we want all tuples (X,Y) such that X is red and Y is green. Now, we want to generate a maximally-contained rewriting of Q. We create the buckets:

```
Bucket        red(X)          green(Y)
              V(X, E1)        V(E2, Y)
```

and the rewriting will be:

$$R1 : Q(X, Y) : -V(X, E1), V(E2, Y). \tag{2}$$

At first sight this might seem wasteful. Why can we not generate a "minimized" rewriting:

$$R2 : Q(X, Y) : -V(X, Y) \tag{3}$$

Instead of generating buckets for minimal units, we could generate a bucket for the maximal unit, i.e., one bucket for both subgoals and since V covers both subgoals, we have V(X,Y) in the bucket and are done with.

In order to see why R1 is a "better" solution than R2, consider a database red(1), red(2), green(3), green(4), path(1,3), path(2,4). R1 will generate the tuples (1,3),(1,4),(2,3),(2,4) whereas R2 will only generate (1,3),(2,4). The query asks for tuples such that the first point is red and the second point is green and the view gives only tuples that not only satisfy the color conditions but also have a path within them. From the view results, R1 extracts the points and their color information and generates all combinations of points that satisfy the query, whereas R2 relies only on the restrictive view and does not generate all the combinations.

3.2.5 Adding Built-in Predicates

Often, queries have some built-in predicates, like the arithmetic order predicate $(\leq, <, =)$, that impose certain constraints on the variables. Let a query have a built-in predicate p_q. Let p_v be the conjunction of built-in view predicates and the equality predicates listed in *addpredicate*(in the algorithm above) for a view $V(\overline{X})$. Let $\overline{Q}v$ represent the query variables in p_q that also occur in a query subgoal q and Vv represent a view variable in $V(\overline{X})$ that any query variable Qv in $\overline{Q}v$ maps to. Before $V(\overline{X})$ is inserted in a bucket representing q, it needs to be checked that p_v does not violate p_q. If p_v does not imply p_q, then to any rewriting with a subgoal $V(\overline{X})$, we add the predicae p_q. If p_v implies p_q, then we do not need to add anything to the solution, since $V(\overline{X})$, which implies p_v (which iin turn implies p_q), already occurs in the solution.

3.3 Solution Generation

All possible sets of buckets are generated such that each query subgoal has exactly one bucket representing it in a set. One view is selected from each bucket in the set of buckets and the conjunction of the selected views is used to generate a solution. All possible combinations of views selected from each bucket are used to generate all solutions related to one set of buckets. By iterating over all possible sets of buckets, all sound rewritings of a query are generated.

The equality constraints between two query variables indicated in *addpredicate* can be added to the solution if built-in predicates are allowed. To generate a solution without the additional predicates listed in *addpredicate*, we rename the variables that are listed to be equal to make them the same.

The candidate solutions generated by the process mentioned above are all sound solutions and there is no need for performing a conjunctive query containment test. In comparison to the bucket algorithm that generated 54 candidate solutions, we generate only the two valid solutions (after the minimization step detailed in the next subsection) $Q(X) : -V2(X, D), V3(X)$ and $Q(X) : -V2(X, D), V4(D, `CA')$.

Theorem 1 *Given a conjunctive query Q and a set of views without built-in predicates,* $(V_1, \ldots, V_n)$*, any solution S generated by the Shared Variable Bucket algorithm is a sound rewriting of Q using the views.*

Theorem 2 *Given a conjunctive query Q and a set of views without built-in predicates,* $(V_1, \ldots, V_n)$*, the Shared Variable Bucket algorithm generates a complete set of maximally contained rewritings of Q using the views.*

Proofs of the soundness and completeness theorems for the algorithm are included in the appendix.

3.4 Minimization of Solutions

A rewriting of a conjunctive query whose conjuncts are views is obtained after the second step of the algorithm. A join of the views present in the rewriting is needed to answer the query. For views occuring multiple times in the rewriting, a self-join needs to be perf ormed. Some of these self-joins may be redundant and can be eliminated thereby minimizi ng the rewriting. Chandra and Merlin [10] have showed that the minimization of conjunctive queries is an NP-complete problem and that a given query can be minimized by query folding, that is, by eliminating redundant subgoals from the query.

Given a rewriting R that we seek to minimize, we can exhaustively construct all possible rewritings R' that can be obtained by eliminating conjuncts from R. Then, using the conjunctive-query-containment-test we test whether R' is equivalent to R, that is, whether R is contained in R' and R' is contained in R. Instead of trying out all possible folded rewritings for equivalence, we can prune the set of folded rewritings using certain heuristics [?]. For example, subgoals that occur only once in the rewriting are essential subgoals and any R' obtained by eliminating the essential subgoals from R are obviously not equivalent to R. However, since the problem is NP-complete, the worst-case performance remains exponential with respect to the size of the rewriting R. Finally, in the absence of other sophisticated cost models, we choose the equivalent rewriting with the smallest number of predicates as the minimal query rewriting.

4 Performance Analysis

The introduction of shared variable buckets implies that our algorithm uses more buckets than the bucket algorithm. Let the number of subgoals in the query be N_g and the number of non-distinguished shared query variables be N_v. Since each bucket corresponds to a unique set of query subgoals and non-distinguished shared query variables, an upper bound on the number of shared variable buckets is given by $min(2^{N_g} - N_g - 1, 2^{N_v} - 1)$. In practice, however, the number of shared-variable buckets (not all non-distinguished shared query variables trigger the creation of a new shared-variable bucket), and the cost associated with creating each is rather small.

Since the problem is NP-complete, the generation of a solution is in the worst case (when each view covers each and every query subgoal we have an exponential number of rewritings) exponential with respect to the size of the input. However, preliminary results of our experiments show the superiority of our algorithm with respect to the bucket or inverse rules algorithm.

Addition of built-in predicates, however, implies that the algorithm, like a modification of the bucket algorithm, needs to check that the conjunction of the built-in predicates of the views does not violate the constraints indicated by the built-in query predicates and this might invalidate a few solutions. However, even in this case, the number of candidate solutions generated by our algorithm is much less than that in the bucket algorithm.

5 Variations of the Algorithm

A version of the algorithm using only single-subgoal buckets can be designed that will also eliminate the conjunctive-query-conjunctive-test. Instead of checking that a non-distinguished shared query variable Qv is covered by a view, we save the partial mapping of Qv to the view variable that it maps to and include the view in the bucket if the view covers the subgoal. Since we have not checked whether the view covers Qv, we cannot guarantee that each generated solution is sound. After the generation of a candidate solution, we check to see if all the partial mappings of Qv obtained from the views that cover the subgoals in which Qv occurs are v-compatible [8] (i.e., Qv maps to a unique variable). This approach is similar, in principle, to the algorithm discussed in [8] but the added check eliminates the unsound solutions generated by Leser.

A hybrid approach is to use limited number of shared variable buckets. Initially, we keep creating SVBs until we hit the limit. Consequently, we switch to the algorithm described above and start saving the partial containment mappings. If a solution is generated with partial containment mappings we check their compatibility and reject unsound solutions.

6 Related Work

Algorithms for rewriting queries have been recently used to satisfy the various needs of several information integration systems [2], [3] and [11]. Query rewriting has also been studied for its use in query optimization [12], [13]. Other approaches to query rewriting is using query folding [14] and the inverse-rules algorithm [7]. The inverse-rules algorithm works for recursive queries but the second stage of the algorithm in which it puts together the inverse rules is almost as expensive as the bucket algorithm's exponential conjunctive-query-containment-test. An algorithm by Pottinger and Levy [15], based on a similary property as that used for shared-variable buckets has been propoesd and is less expensive than the bucket algorithm. Rewriting queries utliizing views with specified binding patterns is considered in [16]. Levy et al. [17] illustrate how an infinite set of views can be used to answer queries. The complexity of answering queries using materialized views is discussed in [18]. Finally, the problem of answering queries in description logics and its complexities has been addressed in [19], [20].

7 Conclusion

In the current work, the shared-variable bucket algorithm has been presented that introduces the concept of shared-variable buckets and uses it to generate only sound candidate solutions thereby eliminating the need for an exponential conjunctive query containment test. Our algorithm considerably speeds up the query reformulation stage of an information integration system. This enables the construction of scalable integration systems that can handle large amounts of information. In today's world, with huge amounts of information obtainable from the World-Wide Web, an efficient query rewriting algorithm is necessary - our algorithm fills this void.

Acknowledgements

I would like to acknowledge Professor Jeffrey D. Ullman for introducing me to this problem and encouraging me and for his numerous comments while reviewing the draft of this paper.

References

[1] A.Y. Levy, A. Mendelzon, D. Srivastava, and Y. Sagiv. Answering queries using views. In *Proc. of Symposium on Principles of Database Systems, San Jose, CA*, pages 163–173, 1995.

[2] The information manifold, http://portal.research.bell-labs.com/orgs/ssr/people/levy/paper-abstracts.html#iga.

[3] Information integration using infomaster, http://infomaster.stanford.edu/infomaster-info.html.

[4] The stanford-ibm manager of multiple information sources, http://www-db.stanford.edu/tsimmis/.

[5] J.D. Ullman. Information integration using logical views. In *Proc. of the International Conference on Database Theory, Delphi, Greece*, 1997.

[6] A.Y. Levy, A. Rajaraman, and J.O. Ordille. Query-answering algorithms for information agents. In *Proc. of the 13th National Conference on Artificial Intelligence, AAAI-96*, 1996.

[7] O.M. Duschka and M.R. Genesereth. Answering recursive queries using views. In *Proc. of the 16th ACM Symposium on Principles of Database Systems, Tucson, AZ*, pages 109–116, 1997.

[8] Ulf Leser. Combining heterogenous data sources through query correspondence assertions. In *Proc. of Workshop on Web Information and Data Management (WIDM98)*, 1998.

[9] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vols. I and II.* Computer Science Press, New York, 1988.

[10] A.K. Chandra and Merlin P.K. Optimal implementation of conjunctive queries in relational databases. In *Proc. of the 9th Annual ACM Symp on Theory of Computing*, pages 77–90, 1977.

[11] F. Naumann, U. Leser, and J.C. Freytag. Quality-driven integration of heterogenous information sources. In *Proceedings of Internation Conference on VLDB '99*, pages 447–458, 1999.

[12] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th ICDE, March 6-10*, pages 190–200, 1995.

[13] Yang H.Z. and Larson P.A. Query transformation for psj-queries. In *Proc. of the 13th International VLDB Conference*, pages 254–254, 1987.

[14] X. Qian. Query folding. In *Proc. of the Twelveth International Conference on Data Engineering, New Orleans, LA*, pages 48–55, 1996.

[15] R. Pottinger and A. Levy. A scalable algorithm for answering queries using views. In *Proceedings of VLDB 2000*, 2000.

[16] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns. In *Proc. of 14th Symp. on Principles of Database Systems*, pages 105–112, 1995.

[17] A. Y. Levy, A. Rajaraman, and J.D. Ullman. Answering queries using limited external processors. In *Proc. of the 15th Symp. on Principles of Database Systems*, pages 227–237, 1996.

[18] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *Proc. of the 17th ACM Symp. on Principles of Database Systems*, pages 254–263, 1998.

[19] C. Beeri, A. Y. Levy, and M. Rousset. Rewriting queries using views in description logics. In *Proc. of the 16th ACM Symposium on Principles of Database Systems*, pages 99–108, 1997.

[20] D. Calvanese, G. De Giacomo, and M. Lenzerini. Answering queries using views in description logics. In *Proceedings of the 6th International Workshop on KRDB*, 1999.

8 Appendix

Theorem 3 *Given a conjunctive query Q and a set of views without built-in predicates, $(V_1, \ldots, V_n)$, any solution S generated by the Shared Variable Bucket algorithm is a sound rewriting of Q using the views.*

Outline of Proof: To prove that the solution S is a sound rewriting, I construct a mapping μ from Q to $E(S)$ as follows: For all i, $\mu(Qv_i) = Qv_i'$ where Qv_i is a distinguished variable in the query at position i and Qv_i' is the distinguished variable in the same position in S. The proof proceeds by considering the distinguished and non-distinguished variables and constructing the containment mapping.

Case (1): Let Qv_i be a distinguished variable. If Qv_i maps to a view variable Vv then Vv must be distinguished, which is ensured when we check that the view V covers the query subgoal in which Qv_i occurs.

We consider the two sub-cases:

(a): Let no other Qv_j map to Vv. Then we replace Vv with Qv_i in the head of the view V before inserting it into the bucket of the query subgoal it covers. Thus for all occurrences of Qv_i, $\mu(Qv_i) = Qv_i$.

(b): Let Qv_i and Qv_j both map to the same distinguished view variable Vv. Vv is replaced by Qv_i or Qv_j and a predicate indicating that they are equal is included before the view is put in a bucket. When a solution is being generated all occurrences of both Qv_i and Qv_j are replaced by a unique Qv'.

Thus any distinguished Qv_i maps to a unique variable in the solution.

Case (2): Now, let Qv_j be a non-distinguished variable. The algorithm ensures that if Qv_j is a shared variable and maps to a non-distinguished shared variable, Qv_j must be covered by the same view and all occurrences of Qv_j must map to Vv (ensured by the bucket construction). This ensures that Qv_j maps to a unique variable in $E(S)$.

If Qv_j does not map to a non-distinguished view variable, let all occurrences of Q_j map to distinguished variables $Vv_1, \ldots, Vv_n$ and no other query variable Q_i maps to any of Vv_i, then we simply replace each distinguished variable Vv_i by Q_j. Thus in the mapping from the query to the solution Q_j maps to itself i.e., $\mu(Q_j) = Q_j$.

Now consider the case when along with Q_j some Q_i also maps to a distinguished view variable Vv_i. Once again, we choose any of Q_i or Q_j and include equality predicates that ensure that Q_i and Q_j are replaced by a unique variable while generating the solution. Therefore, Q_j maps to a unique variable in the solution.

Since all query variables map to a unique view variable in the solution, there exists a containment mapping from the query to $E(S)$ and $E(S)$ is contained in the query. Thus S is a sound rewriting of Q.

Theorem 4 *Given a conjunctive query Q and a set of views without built-in predicates, $(V_1, \ldots, V_n)$, the Shared Variable Bucket algorithm generates a complete set of maximally-contained rewritings of Q using the views.*

Outline of Proof: We need to show that all possible rewritings of Q are generated. Let S' be a sound maximally-contained rewriting of Q that the Shared-Variable-Bucket algorithm did not generate and S' is not contained in any solution generated by our algorithm. Let S be the minimized conjunctive query equivalent to S'. There are two cases:

(1) Let us consider the case where there is at least one view head $V(\overline{X})$ in S that is not in any bucket the algorithm generates. Clearly, $V(\overline{X})$ is not a redundant subgoal, since S is a minimized version of S'. Each view in a minimized maximally-contained rewriting of a query covers at least one query subgoal. Thus $E(V(\overline{X}))$ contains a subgoal g' that a query subgoal g maps to in the mapping from Q to $E(S)$.

However, since $V(\overline{X})$ does not occur in any bucket generated, it must have been rejected by one of the tests (single subgoal bucket test or shared-variable-bucket test), we perform before putting a view head into a bucket. Let the view $V\overline{X})$ be rejected by the first test i.e., it does not cover g. Thus, there is no possible mapping from the query subgoal to a subgoal in the view that maps query variables to unique variables in the solution. However, if there exists no unique mapping betwen the query variables and the view variables, the view cannot occur in any valid solution - which is a contradiction to our assumption above. Therefore, a view $V\overline{X})$ cannot fail the first test and yet appear in a valid solution.

Let the view $V(\overline{X})$ be rejected by the second test. Let a shared non-distinguished query variable Qv occur in g. If Qv maps to an non-distinguished view variable, the view must cover the query variable for it to be included in a bucket for g. Since it was not inserted into the bucket corresponding to g, the view must not be covering the query variable, i.e., either Qv maps to two different non-distinguished view variables or Qv occurs in a subgoal not covered by $V(\overline{X})$. In the former case, no mapping can exist from Q to S with g mapping to a g' in $V(\overline{X})$ since Qv maps to two different view variables. In the latter case, there must be some view V' that covers the occurrence of Qv outside subgoals covered by V. Thus Qv maps to a non-distinguished view variable in $V(\overline{X})$ and another variable in V'. Since view variables are named uniquely in different views, Qv does not map to a unique variable in the mapping from Q to $E(S)$. Therefore, S is not a valid solution.

(2) We now consider the case where a subgoal in the solution S occurs in at least one bucket generated by our algorithm. In the solution generation step, we generate all possible combinations of buckets and views from the buckets. The only restriction we impose is that the buckets must represent subgoals that do not overlap. Therefore, if the algorithm did not generate a solution, it must be because of this restriction - since otherwise we exhaustively generate all possible combinations of view heads from the buckets.

We now consider the various ways we can relax the restriction and prove by contradiction that the restriction should not be relaxed.

Let us consider the case where we select two subgoals from the same single-subgoal bucket to construct S. Since both the subgoals cover the same query subgoal, one of the subgoals could be eliminated and still S would be a valid solution. This contradicts with the earlier assumption that S is a minimized version of S' which is a maximally-contained rewriting of the query.

Now let us construct S by selecting two subgoals from buckets which overlap in the subgoals they represent.

Let one bucket be a single-subgoal bucket and another a shared-variable bucket. Clearly, the single-subgoal bucket can be eliminated from S and still S is contained in the query because the subgoal that the view from the single-subgoal bucket covers is already covered by the view from the shared-subgoal bucket. This contradicts with the fact that S is minimized version of S' which is maximally contained in the query.

Now let us consider the solution S obtained by selecting two view heads from buckets both of which represent multiple subgoals and the set of subgoals they represent overlap. If both sets are exactly the same, once again either view-head can be eliminated and still S should remain a solution, which contradicts our initial assumption that S' is maximally-contained.

Now if both the sets of subgoals are not exactly the same, consider a query subgoal g that is common to both. There must be at least two shared variables, say Qv and Qv' such that Qv is shared between g and the first set of subgoals and Qv' is shared between g and the second set of subgoals. If we try to construct a mapping from the query to the expansion of the solution and map g to the subgoal in the expansion of the first view, Qv' maps to a variable in the expansion of the first view. Furthermore, the other occurrences of Qv' in subgoals not covered by the first view map to a different view variable since view variables across views are uniquely named. Thus, g cannot map to the subgoal in the first view. Similary, since Qv will map to two different view variables, g cannot map to the subgoal in the second view either. Furthermore, if g is covered by a third view, the soundness theorem indicates that since it shares non-distinguished variables with other subgoals, all of them need to be covered by one view in order for the solution to be sound. Clearly, g can not map to any subgoal in the first or second or any other view expansions and yet guarantee the soundness of the generated solution. Thus there exists no mapping from the query to the solution. Therefore, S and in turn S' is not a maximally-contained rewriting of the query.

Therefore, if there is any rewriting S of Q that is not generated by the SVB algorithm, it is either not a maximimally-contained rewriting of Q or is contained in at least one of the solutions generated by the algorithm.