Generating Efficient Plans for Queries Using Views

Foto N. Afrati[*]
Computer Science Division
National Technical University of Athens
157 73 Athens, Greece
afrati@cs.ece.ntua.gr

Chen Li[†]
Computer Science Department
Stanford University
Stanford, CA 94305
chenli@cs.stanford.edu

Jeffrey D. Ullman
Computer Science Department
Stanford University
Stanford, CA 94305
ullman@cs.stanford.edu

ABSTRACT

We study the problem of generating efficient, equivalent rewritings using views to compute the answer to a query. We take the closed-world assumption, in which views are materialized from base relations, rather than views describing sources in terms of abstract predicates, as is common when the open-world assumption is used. In the closed-world model, there can be an infinite number of different rewritings that compute the same answer, yet have quite different performance. Query optimizers take a logical plan (a rewriting of the query) as an input, and generate efficient physical plans to compute the answer. Thus our goal is to generate a small subset of the possible logical plans without missing an optimal physical plan.

We first consider a cost model that counts the number of subgoals in a physical plan, and show a search space that is guaranteed to include an optimal rewriting, if the query has a rewriting in terms of the views. We also develop an efficient algorithm for finding rewritings with the minimum number of subgoals. We then consider a cost model that counts the sizes of intermediate relations of a physical plan, without dropping any attributes, and give a search space for finding optimal rewritings. Our final cost model allows attributes to be dropped in intermediate relations. We show that, by careful variable renaming, it is possible to do better than the standard "supplementary relation" approach, by dropping attributes that the latter approach would retain. Experiments show that our algorithm of generating optimal rewritings has good efficiency and scalability.

[*]Work partly done while this author was visiting Stanford University. Work of this author partly supported by Graduate Program MPLA funded under the Community Support Framework of EU.

[†]Research partially supported by the Stanford Graduate Fellowship and the National Science Foundation under grant #IRI-9631952.

1. INTRODUCTION

The problem of using materialized views to answer queries [16] has recently received considerable attention because of its relevance to many data-management applications, such as information integration [3, 7, 13, 14, 17, 26], data warehousing [24], web-site designs [10], and query optimization [6]. The problem can be stated as follows: given a query on a database schema and a set of views over the same schema, can we answer the query using only the answers to the views?

In this paper we study the problem of how to generate *efficient* equivalent rewritings using views to compute the answer to a query; that is, how to generate *logical plans* (i.e., equivalent rewritings) using views for a query such that the logical plans are efficient to evaluate. We take the *closed-world assumption* [1], in which views are materialized from base relations, rather than views describing sources in terms of abstract predicates, as is common when the open-world assumption is used [1, 17]. In the closed-world model, there can be an infinite number of rewritings using views that compute the same answer to a query, yet they have quite different performance.

We focus on the step of generating rewritings for a query, without specifying in detail how each rewriting is evaluated in a physical plan. Each rewriting is passed as a logical plan to an *optimizer*, which translates the rewriting to a *physical plan*, i.e., an execution plan. Each physical plan accesses the stored ("materialized") views, and applies a sequence of relational operators to compute the answer to the original query. The task of the optimizer is to search in the space of all physical plans for an optimal one. Traditional optimizers such as the System-R optimizer [22] search in the space of left-deep-join trees of a logical plan for an optimal physical plan, which specifies the execution detail such as join ordering, and evaluation of a join (e.g., hash join, merge join).

Our goal is to generate rewritings for a query that are guaranteed to produce an optimal physical plan, if the query has a rewriting. In other words, we want to make sure that at least one rewriting generated by our algorithm can be translated by the optimizer into an optimal physical plan. A rewriting is called *optimal* if it has a physical plan that has the lowest cost among all physical plans of all rewritings of the original query under certain cost model. Thus the step of generating optimal rewritings should be *cost-based.*

The following example illustrates several issues in generating optimal rewritings using views for a query:[1] (1) There can be an infinite number of rewritings for a query. (2)

[1]We will refer to this example as the "car-loc-part example"

Traditional query-containment techniques [5] cannot find a rewriting with the minimum number of joins. (3) Adding more view relations to a rewriting could make the rewriting more efficient to evaluate.

EXAMPLE 1.1. *Suppose we have three base relations:*

- `car(Make,Dealer)`. *A tuple* `car(m,d)` *means that dealer* `d` *sells cars of make* `m`.
- `loc(Dealer,City)`. *A tuple* `loc(d,c)` *means that dealer* `d` *has a branch in the city* `c`.
- `part(Store,Make,City)`. *A tuple* `part(s,m,c)` *means that store* `s` *in city* `c` *sells parts for cars of make* `m`.

A user submits the following query Q:

$$q_1(S, C) \text{ :- } car(M, anderson), loc(anderson, C), part(S, M, C)$$

that asks for cities and stores that sell parts for car makes in the anderson *branch in this city. Assume that we have the following materialized views on the base relations (for simplicity, "anderson" is abbreviated as "a"):*

V_1: $v_1(M, D, C)$:- $car(M, D), loc(D, C)$
V_2: $v_2(S, M, C)$:- $part(S, M, C)$
V_3: $v_3(S)$:- $car(M, a), loc(a, C), part(S, M, C)$
V_4: $v_4(M, D, C, S)$:- $car(M, D), loc(D, C), part(S, M, C)$
V_5: $v_5(M, D, C)$:- $car(M, D), loc(D, C)$

In the closed-world model, these five views are computed from the three base relations. In particular, views V_1 and V_5 have the same definition, thus their view relations always have the same tuples for any base relations. Under the open-world assumption, however, we would only know that V_1 and V_5 contain only tuples in $car(M, D), loc(D, C)$; either or both could even be empty. Suppose we do not have access to the base relations, and can answer the query only using the answers to the views. The following are some rewritings for the query using the views. Notice that there is an infinite number of rewritings for the query, since each rewriting P has an infinite number of rewritings that are equivalent to P as queries [25].

P_1: $q_1(S, C)$:- $v_1(M, a, C_1), v_1(M_1, a, C), v_2(S, M, C)$
P_2: $q_1(S, C)$:- $v_1(M, a, C), v_2(S, M, C)$
P_3: $q_1(S, C)$:- $v_3(S), v_1(M, a, C), v_2(S, M, C)$
P_4: $q_1(S, C)$:- $v_4(M, a, C, S)$
P_5: $q_1(S, C)$:- $v_1(M, a, C_1), v_5(M_1, a, C), v_2(S, M, C)$

We can show that all of these rewritings compute the answer to the query Q. However, some of them may lack an efficient physical plan. For instance, compared to rewriting P_2, rewriting P_1 needs one more access to the view relation V_1 and one more join operation. In addition, we cannot easily minimize P_1 to generate P_2 using traditional query-containment techniques [5], since neither of the first two subgoals of P_1 is redundant. Furthermore, although P_3 uses one more view V_3 than P_2, the former can still produce a more efficient execution plan if the view relation V_3 is very selective. That is, if there are few stores that sell parts for cars that dealer anderson *sells, and are located in the same city as* anderson, *then view V_3 can be used as a filtering relation. Rewriting P_4 could be an optimal rewriting, since it requires only one access to view V_4.* □

throughout the paper.

In general, given a query and a set of views, the following questions arise:

1. In what space should we search for optimal rewritings?
2. How do we find optimal rewritings efficiently?
3. How does an optimizer generate an efficient physical plan from a logical plan by considering the view definitions?

1.1 Our solution

In this paper we answer these questions by considering several cost models. We define search spaces for finding optimal rewritings, and develop efficient algorithms for finding optimal rewritings in each search space. The following are the main contributions of the paper:

1. We first consider a simple cost model M_1 that counts only the number of subgoals in a physical plan. We analyze the internal relationship of all rewritings for a query, and show a search space for finding optimal rewritings under this cost model (Section 3).
2. We develop an efficient algorithm called CoreCover for finding optimal rewritings in the above search space under M_1 (Section 4).
3. We then study a more complicated cost model M_2 that considers the sizes of view relations and intermediate relations [11] in a physical plan of a rewriting. We also show a search space for finding optimal rewritings under M_2, and develop an algorithm for finding them in this space (Section 5).
4. Finally we study a cost model M_3 that allows attributes to be dropped in intermediate relations. We show that, by careful variable renaming, it is possible to do better than the standard "supplementary relation" approach [4], by dropping attributes that the latter approach would retain (Section 6).

Experiments show that the CoreCover algorithm of generating optimal rewritings has good efficiency and scalability (Section 7).

1.2 Related work

The problem of finding whether there exists an equivalent rewriting for a query using views was studied in [16]. Recently, several algorithms have been developed for finding rewritings of queries using views. Algorithms most closely related to our approach include the bucket algorithm [12, 17], the inverse-rule algorithm [9, 21, 2], the MiniCon algorithm [20], and the Shared-Variable-Bucket algorithm [19]. (See [15] for a survey.) These algorithms aim at generating *contained rewritings* for a query that compute a subset of the answer to the query, while we want to find *equivalent rewritings* that compute the same answer to a query. Another difference is that they have no optimization considerations since under the open-world assumption, different equivalent rewritings for a query can produce different answers over the same view instance. Under the closed-world assumption, however, two equivalent rewritings produce the same answer for any instance of the view database. In this case, choosing rewritings with more efficient physical plans is an interesting issue.

Our algorithms for generating optimal rewritings share some observations with the MiniCon algorithm. In addition, as we will see in Section 4, since we want to generate equivalent rewritings rather than contained rewritings, this different goal helps us develop more efficient algorithms by considering a containment mapping from the expansion of an equivalent rewriting to the query. The detailed comparison is in Section 4.3.

Other related work includes [6] and [27], which consider generating efficient plans using materialized views by replacing subgoals in a query with view literals. In these works, they aim at improving the performance of query evaluation by replacing some of the base relations in the query by materialized views. Our optimization considerations differ from theirs in that (1) we take a two-step approach by separating the rewriting generator and the optimizer, (2) we explore optimization issues that relate to other parameters such as selectivity of the view relations. In this perspective, the introduction of more literals can make a rewriting more efficient, as shown by the rewritings P_2 and P_3 in the car-loc-part example.

2. PRELIMINARIES

In this section, we review some concepts about answering queries using views. We also introduce some notions that are used throughout the paper.

2.1 Answering queries using views

We consider the problem of answering queries using views for conjunctive queries (i.e., select-project-join queries) in the form:

$$h(\bar{X}) \text{ :- } g_1(\bar{X}_1), \ldots, g_k(\bar{X}_k)$$

In each subgoal $g_i(\bar{X}_i)$, predicate g_i is a *base relation*, and every argument in the subgoal is either a variable or a constant. We consider views defined on the base relations by safe conjunctive queries, i.e., every variable in a query's head appears in the body. A variable is called *distinguished* if it appears in the head. We take the closed-world assumption [1], since the views are computed from existing database relations. We shall use names beginning with lower-case letters for constants and relations, and names beginning with upper-case letters for variables. We use $V, V_1, \ldots, V_m$ to denote *views* that are defined by conjunctive queries on the base relations.

DEFINITION 2.1. (**query containment and equivalence**) *A query Q_1 is* contained *in a query Q_2, denoted $Q_1 \sqsubseteq Q_2$, if for any database D of the base relations, the answer computed by Q_1 is a subset of the answer by Q_2, i.e., $Q_1(D) \subseteq Q_2(D)$. The two queries are* equivalent*, denoted $Q_1 \equiv Q_2$, if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.* □

Chandra and Merlin [5] show that a conjunctive query Q_1 is contained in another conjunctive query Q_2 if and only if there is *containment mapping* from Q_2 to Q_1. The containment mapping maps the head and all the subgoals in Q_2 to Q_1. It maps each variable to either a variable or a constant, and maps each constant to the same constant.

DEFINITION 2.2. (**expansion of a query using views**) *The* expansion *of a query P on a set of views $\mathcal{V}$, denoted P^{exp}, is obtained from P by replacing all the views in P with their corresponding base relations. Existentially quantified variables in a view are replaced by fresh variables in P^{exp}.* □

DEFINITION 2.3. (**equivalent rewritings**) *Given a query Q and a set of views $\mathcal{V}$, a query P is an* equivalent rewriting *of query Q using $\mathcal{V}$, if P uses only the views in $\mathcal{V}$, and P^{exp} is equivalent to Q, i.e., $P^{exp} \equiv Q$.* □

In our car-loc-part example, both

$$\begin{array}{lll} P_1: & q_1(S,C) & \text{:- } v_1(M,a,C_1), v_1(M_1,a,C), v_2(S,M,C) \\ P_2: & q_1(S,C) & \text{:- } v_1(M,a,C), v_2(S,M,C) \end{array}$$

are two equivalent rewritings for the query

$$Q: \; q_1(S,C) \text{ :- } car(M,a), loc(a,C), part(S,M,C)$$

because their expansions

$$\begin{array}{lll} P_1^{exp}: & q_1(S,C) & \text{:- } car(M,a), loc(a,C_1), \\ & & \quad car(M_1,a), loc(a,C), part(S,M,C) \\ P_2^{exp}: & q_1(S,C) & \text{:- } car(M,a), loc(a,C), part(S,M,C) \end{array}$$

can be shown to be equivalent to Q. This example also shows that two equivalent rewritings of the same query might not be equivalent as queries. That is, although $P_1^{exp} \equiv P_2^{exp}$, it is not true that $P_1 \equiv P_2$. Notice that the test for $P_1 \equiv P_2$ involves containment mappings in *views*, while the test for $P_1^{exp} \equiv P_2^{exp}$ involves containment mappings in *base relations*. We say that two rewritings P_1 and P_2 are *equivalent as queries* if $P_1 \equiv P_2$. Whereas, we say that two rewritings P_1 and P_2 are *equivalent as expansions* if $P_1^{exp} \equiv P_2^{exp}$.

In the rest of this paper, unless otherwise specified, the term "rewriting" means an "equivalent rewriting" of a query using views.

2.2 Efficiency of rewritings

Let P be a rewriting of a query Q using views $\mathcal{V}$. We define three cost models, as shown in Table 1. For each of them, we define a *physical plan* for P and a *cost measure* on this physical plan.

Cost model	Physical plan	Cost measure
M_1	a set of subgoals	number of subgoals: n
M_2	a list of subgoals	$\sum_{i=1}^{n}(size(g_i) + size(IR_i))$
M_3	a list of subgoals annotated with projected attributes	$\sum_{i=1}^{n}(size(g_i) + size(GSR_i))$

Table 1: Three cost models

Under cost model M_1, a physical plan of P is a set of the view subgoals in P, and the cost measure is the number of subgoals in P. That is, the cost of a physical plan F is:

$$cost_{M_1}(F) = number\ of\ subgoals\ in\ F$$

The main motivation of cost model M_1 is to minimize the number of join operations, which tend to be expensive in practice, when a rewriting is evaluated.

Under cost model M_2, a physical plan F of rewriting P is a list $g_1, \ldots, g_n$ of the view subgoals in P. The views corresponding to these subgoals are joined in the order listed. After joining the first i subgoals in the list, the *intermediate relation* IR_i is the join result with all attributes retained [11]. The cost measure for F under M_2 is the sum of the sizes of the views joined, plus the sizes of the intermediate relations computed during the multiway join. More formally, The cost measure of F under M_2 is:

$$cost_{M_2}(F) = \sum_{i=1}^{n}(size(g_i) + size(IR_i))$$

where $size(g_i)$ is the size of the relation for the subgoal g_i, and $size(IR_i)$ is the size of the intermediate relation IR_i. The motivation of cost model M_2 is that, as shown in [11], the time of executing a physical plan is usually determined by the number of disk IO's, which is a function of the sizes of those relations used in the plan.

Cost model M_3 is motivated by the supplementary-relation approach [4], whose main idea is to drop attributes during the evaluation of a sequence of subgoals. Under M_3, a physical plan of rewriting P is a list $g_1^{\bar{X}_1}, \ldots, g_n^{\bar{X}_n}$ of the view subgoals in P, with each subgoal g_i annotated with a set $\bar{X}_i$ of *nonrelevant* attributes. All the attributes in $\bar{X}_i$ can be dropped after the first i subgoals are processed, while still being able to compute the answer to the original query after the evaluation terminates. The *generalized supplementary relation* ("GSR" for short) after the first i subgoals are processed, denoted GSR_i, is the intermediate relation IR_i with the attributes in $\bar{X}_i$ dropped.

The cost measure for M_3 is the sum of the sizes of the views joined, plus the sizes of the generalized supplementary relations computed during the multiway join. More formally, for a physical plan $F = g_1^{\bar{X}_1}, \ldots, g_n^{\bar{X}_n}$, its cost under M_3 is:

$$cost_{M_3}(F) = \sum_{i=1}^{n}(size(g_i) + size(GSR_i))$$

where $size(g_i)$ is the size of the relation for the subgoal g_i, and $size(GSR_i)$ is the size of the generalized supplementary relation GSR_i.

Notice that a special case of cost model M_3 is when the nonrelevant attributes in $\bar{X}_i$ are defined as the attributes in the join that are not used in either the query's head, or any subsequent subgoals after subgoal g_i. Then we get the supplementary relation as defined in the literature [4, 25]. However, as we will see in Section 6, by careful variable renaming, it is possible to drop more attributes than the traditional supplementary-relation approach.

DEFINITION 2.4. (**efficiency of rewritings**) *Under a cost model M, a rewriting P_1 of a query Q is* more efficient *than another rewriting P_2 of Q if the cost of an optimal physical plan of P_1 under cost model M is less than the cost of an optimal physical plan of P_2. A rewriting P is an* optimal rewriting *if it has a physical plan with the lowest cost in all the physical plans of rewritings of Q under M.* □

3. COST MODEL M_1: NUMBER OF VIEW SUBGOALS

In this section we study how to find optimal rewritings under cost model M_1, i.e., rewritings with the minimum number of view subgoals. We first show, given a rewriting, how to minimize its view subgoals. However, this minimization step might miss optimal rewritings if it uses only traditional query-containment techniques. Then we analyze the internal structure of all rewritings of a query, and give a space that is guaranteed to include a rewriting with the minimum number of subgoals, if the query has a rewriting.

3.1 Minimizing view subgoals in a rewriting

Suppose we are given a rewriting P of a query Q using views $\mathcal{V}$. The first step to take is to find the minimal equivalent query of P (not P^{exp}) by removing its redundant subgoals. Let P_m be this minimal equivalent. However, even for the minimal rewriting P_m, we might still be able to remove some of its view subgoals while retaining its equivalence to Q, because we are really interested in rewritings *after* expansion of the views. For instance, P_3 in the car-loc-part example is a minimal rewriting, but we can still remove its subgoal $v_3(S)$ and obtain rewriting P_2 with fewer subgoals. Notice that P_2 and P_3 are not equivalent as queries, although they both compute the same answer to the query. Thus in the second minimization step, we keep removing subgoals from the minimal rewriting P_m, until we get a *locally-minimal rewriting* ("LMR" for short), denoted P_{LMR}. That is, P_{LMR} is a rewriting from which we cannot remove any subgoals and still retain equivalence to the query Q. For instance, the rewritings P_1 and P_2 are two LMRs of the query. The rewriting P_3 is a minimal rewriting, but not an LMR.

For the obtained rewriting P_{LMR}, we cannot remove further subgoals while retaining its equivalence to the query Q. For instance, neither of the first two subgoals in the rewriting P_1 can be removed and still retain its equivalence to the query Q. However, as we will see shortly, we can still reduce the number of view subgoals in an LMR by proper variable renaming. In addition, our goal is to find *globally-minimal rewritings* ("GMR" for short), i.e., rewritings with the minimum number of subgoals. For this goal we analyze the structure of all rewritings of a query.

3.2 Structure of rewritings

Consider the two LMRs P_1 and P_2 in the car-loc-part example. Notice that rewriting P_2 is properly contained in P_1 as queries, while P_2 has fewer subgoals than P_1. Surprisingly, we can generalize this relationship between containment of two LMRs and their numbers of subgoals as follows.

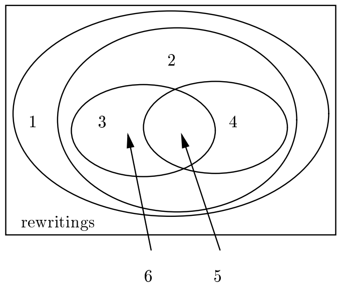
LEMMA 3.1. *Let P_1 and P_2 be two LMRs of a query Q. If $P_1 \sqsubseteq P_2$ as queries, then the number of subgoals in P_1 is not greater than the number of subgoals in P_2.* □

PROOF. Since $P_1 \sqsubseteq P_2$, there is a containment mapping μ from P_2 to P_1. Suppose that the number of subgoals in P_1 is greater than the number of subgoals in P_2. Then at least one subgoal of P_1 is not used in μ. Consider the expansions P_1^{exp} and P_2^{exp} of P_1 and P_2, respectively. The mapping μ implies a mapping from P_2^{exp} to P_1^{exp}. This mapping, together with a containment mapping from Q to P_2^{exp}, implies a mapping from Q to P_1^{exp}. The latter leads to a rewriting that uses only a proper subset of the subgoals in P_1, contradicting the fact that P_1 is an LMR. □

We say an LMR is a *containment-minimal rewriting* ("CMR" for short) if there is no other LMR that is properly contained in this rewriting as queries. For instance, the rewriting P_2 in the car-loc-part example is a CMR, while rewriting P_1 is not. However, a GMR might not be a CMR, as shown by the following query, views, and rewritings:

Query:	Q:	$q(X)$	:- $e(X,X)$
Views:	V:	$v(A,B)$	:- $e(A,A), e(A,B)$
Rewritings:	P_1:	$q(X)$	:- $v(X,B)$
	P_2:	$q(X)$	:- $v(X,X)$

The rewriting P_1 is a GMR, but it is not a CMR, since there is another rewriting P_2 (also a GMR) that is properly contained in P_1. We will give a space that is guaranteed to include a GMR of a query, if the query has a rewriting.



1. Minimal rewritings.
2. Locally minimal rewritings.
3. Containment minimal rewritings.
4. Globally minimal rewritings.
5. Region 3 ∩ region 4.
6. Region 3 - region 4.

Figure 1: Relationship of rewritings of a query.

The relationship of all different rewritings of a query Q is shown in Figure 1. It can be summarized as follows:

1. A minimal rewriting P does not include any redundant subgoals as a query.

2. A locally-minimal rewriting (LMR) is a minimal rewriting whose subgoals cannot be dropped and still retain equivalence to the query. As we will see shortly, all LMRs form a partial order in terms of their number of subgoals and containment relationship.

3. A containment-minimal rewriting (CMR) P is a locally minimal rewriting with no other locally minimal rewritings properly contained in P as queries.

4. A globally-minimal rewriting (GMR) is a rewriting with the minimum number of subgoals. A globally-minimal rewriting is also locally minimal. The subtlety here is that by Lemma 3.1, each GMR P has at least one CMR contained in P with the same number of subgoals. Thus, for each GMR in region 6 in Figure 1, there exists a GMR in region 5 that has the same number of subgoals. Therefore, we can just limit our search space to all CMRs for finding GMRs.

More formally, the following two propositions are corollaries of Lemma 3.1.

PROPOSITION 3.1. *Each GMR P has at least one CMR that i) is contained in P and ii) has the same number of subgoals as P.* □

PROPOSITION 3.2. *The set of CMRs contains at least one GMR.* □

EXAMPLE 3.1. *Consider the following query Q, view V, and three rewritings P_1, P_2, and P_3.*

Q:	$q(X,Y,Z)$	:- $e_1(X,c), e_2(Y,c), e_3(Z,c)$
V:	$v(X,Y,Z,W)$	:- $e_1(X,W), e_2(Y,W), e_3(Z,W)$
P_1:	$q(X,Y,Z)$	:- $v(X,Y,Z,c)$
P_2:	$q(X,Y,Z)$	:- $v(X,Y,Z_1,c), v(X_1,Y_1,Z,c)$
P_3:	$q(X,Y,Z)$	:- $v(X,Y_1,Z_1,c), v(X_2,Y,Z_2,c),$ $v(X_3,Y_3,Z,c)$

LMR P_1 is properly contained in LMR P_2 as queries, which is properly contained in LMR P_3 as queries. Rewriting P_1 is containment minimal. We can generalize this example to m base relations $e_1, e_2, \ldots, e_m$ in the query, and get a partial order of LMRs that is a chain of length m. □

Since containment mapping is transitive, all the locally-minimal rewritings of a query form a partial order in terms of their containment relationships. The bottom elements in this partial order are the CMRs. In addition, by Lemma 3.1, the containment relationship between two LMRs also implies that the contained rewriting has no more subgoals than the containing rewriting. Figure 2(a) shows the partial order of the four LMRs (P_1, P_2, P_4, and P_5) in the car-loc-part example. Figure 2(b) shows the partial order of the rewritings in Example 3.1. Each edge in the figure represents a proper containment relationship: the upper rewriting properly contains the lower rewriting.

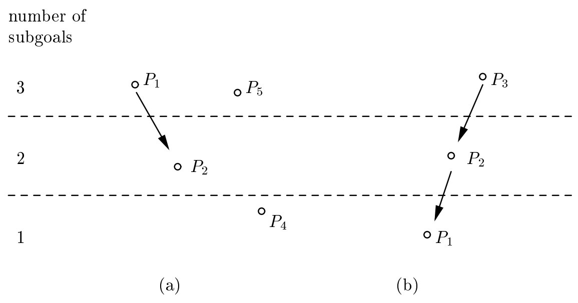


Figure 2: Partial order of locally-minimal rewritings of a query.

3.3 A space including globally-minimal rewritings

The conclusion of the previous subsection is that we can search in the space of CMRs for a GMR, if the query has a rewriting. Now we define a search space in a more constructive way. We need first define several notations. Given a query Q, a *canonical database* D_Q of Q is obtained by turning each subgoal into a fact by replacing each variable in the body by a distinct constant, and treating the resulting subgoals as the only tuples in D_Q. Let $\mathcal{V}(D_Q)$ be the result of applying the view definitions $\mathcal{V}$ on database D_Q. For each tuple in $\mathcal{V}(D_Q)$, we restore each introduced constant back to the original variable of Q, and obtain a *view tuple* of the query given the views. Let $\mathcal{T}(Q,\mathcal{V})$ denote the set of all view tuples *after* the replacement.

In our car-loc-part example, a canonical database for the query Q is:

$$D_Q = \{car(m,a), loc(a,c), part(s,m,c)\}$$

where the variables M, C, and C are replaced by new distinct constants m, c, and s, respectively. By applying the five view definitions $\mathcal{V}$ on D_Q, we compute $\mathcal{V}(D_Q)$:

$$\{v_1(m,a,c), v_2(s,m,c), v_3(s), v_4(m,a,c,s), v_5(m,a,c)\}$$

Thus the set of view tuples $\mathcal{T}(Q,\mathcal{V})$ is

$$\{v_1(M,a,C), v_2(S,M,C), v_3(S), v_4(M,a,C,S), v_5(M,a,C)\}$$

The following lemma, which is a rephrasing of a result in [16], helps us restrict the search space for finding globally-minimal rewritings for a query.

LEMMA 3.2. *For any rewriting P*

$$q(\bar{X}) \text{ :- } p_1(\bar{Y}_1), \ldots, p_k(\bar{Y}_k)$$

of a query Q using views $\mathcal{V}$, there is a rewriting P' of Q such that P' is in the form:

$$q(\bar{X}) \text{ :- } p_1(\bar{Y}_1'), \ldots, p_k(\bar{Y}_k')$$

In addition, each $p_i(\bar{Y}_i')$ is a view tuple in $\mathcal{T}(Q, \mathcal{V})$, and $P' \sqsubseteq P$. □

The main idea of the proof is to consider a containment mapping μ from P^{exp} to Q, and replace each variable X in P by its target variable $\mu(X)$ in Q. For instance, let us see how to transform P_1 in the car-loc-part example to the LMR P_2 that uses the view tuples only. Consider the mapping μ from P_1^{exp} to Q: $\{M_1 \rightarrow M, M \rightarrow M, a \rightarrow a, C_1 \rightarrow C, C \rightarrow C, S \rightarrow S\}$. Under μ, we transform P_1 to:

$$P_1' : q_1(M, C) \text{ :- } v_1(M, a, C), v_1(M, a, C), v_2(S, M, C)$$

After removing one duplicate subgoal from P_1', we have the rewriting P_2.

In Section 3.2, we showed that the set of CMRs contains a GMR. Below we define a search space for GMRs in a more constructive fashion. The following lemma shows that CMRs are contained in a set of rewritings defined constructively, hence we can regard this set as a search space for optimal rewritings under cost model M_1.

LEMMA 3.3. *All LMRs of a query using views that use only view tuples of the query include all CMRs of the query.*[2] □

An immediate consequence is the following theorem that defines a restricted space for searching globally-minimal rewritings of a query.

THEOREM 3.1. *By searching in the space of all LMRs of a query that use only view tuples in $\mathcal{T}(Q, \mathcal{V})$, we guarantee to find a globally-minimal rewriting, if the query has a rewriting.* □

Theorem 3.1 suggests a naive algorithm that finds a globally-minimal rewriting of a query Q using views $\mathcal{V}$ as follows. We compute all the view tuples for the query. We start checking combinations of view tuples. We first check all combinations containing one view tuple, then all combinations containing two view tuples, and so on. Each combination could be a rewriting P. We test whether there is a containment mapping from Q to P^{exp}. (By the construction of the view tuples, there is always a containment mapping from P^{exp} to Q.). If there is, then P is a GMR. It is known [16] that if there is a rewriting for the query, then there is one with at most n subgoals, where n is the number of subgoals in the query. Thus we stop after having considered all combinations of up to n view tuples.

[2]We assume two rewritings are the same if the only difference between them is variable renamings.

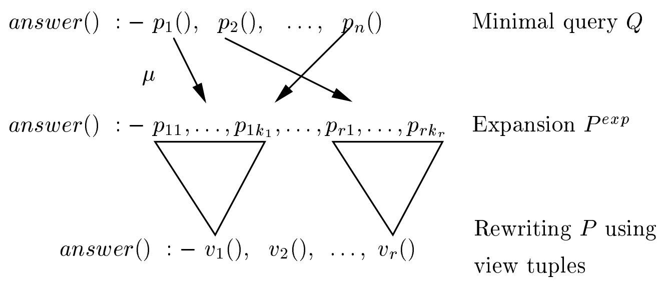
4. AN ALGORITHM FOR FINDING GLOBALLY-MINIMAL REWRITINGS

In this section we develop an efficient algorithm, called CoreCover, for finding optimal rewritings of a query under the cost model M_1, i.e., globally-minimal rewritings. The algorithm searches in the space of rewritings using view tuples for GMRs of the query. Intuitively, the algorithm considers each view tuple to see what query subgoals can be covered by this view tuple. The set of query subgoals covered by the view tuple is called *tuple-core*. The algorithm then uses the *minimum* number of view tuples to cover all query subgoals, and each cover yields a GMR of the query.

4.1 Tuple-core: query subgoals covered by a view tuple

The algorithm CoreCover first finds the set of query subgoals that can be "covered" by a view tuple, called *tuple-core*. Before giving the definition of tuple-core, we show a nice property of rewritings using view tuples for a minimal query. Note that for the rewritings we consider in this section, we may think as follows: All the variables of rewriting P (recall that P is generated out of view tuples) are also variables of Q, i.e., $Var(P) \subseteq Var(Q)$.

LEMMA 4.1. *For a minimal query Q and a set of views $\mathcal{V}$, let P be a rewriting of Q that uses only view tuples in $\mathcal{T}(Q, \mathcal{V})$. There is a containment mapping μ from Q to P^{exp}, such that (1) μ is a one-to-one mapping, i.e., different arguments in Q are mapped to different arguments in P^{exp}; (2) For all arguments in Q that appear in P, they are mapped by μ as is the identity mapping on arguments, i.e., $\mu(X) = X$ for all $X \in Var(P)$.*[3] □



$answer()$:- $p_1()$, $p_2()$, …, $p_n()$
Minimal query Q
μ
$answer()$:- $p_{11}, \ldots, p_{1k_1}, \ldots, p_{r1}, \ldots, p_{rk_r}$
Expansion P^{exp}
$answer()$:- $v_1()$, $v_2()$, …, $v_r()$
Rewriting P using view tuples

Figure 3: A containment mapping from Q to P^{exp}.

Figure 3 shows the intuition of the lemma. For instance, the rewriting P_2 in the car-loc-part example uses view tuples only. We have a containment mapping from the query Q to P_2^{exp}: $\{M \rightarrow M, a \rightarrow a, C \rightarrow C, S \rightarrow S\}$. This containment mapping maps the arguments $\{M, a, C, S\}$ in Q that appear in P_2 on themselves.

In general, there can be different containment mappings from a minimal query to the expansion of a rewriting using view tuples. By Lemma 4.1, it turns out that we can just focus on a containment mapping that has the two properties in the lemma, and decide what query subgoals are covered by the *expansion* of each view tuple under this containment

[3]The definition of rewriting P guarantees a containment mapping from Q to P^{exp}, but this containment mapping might not have the two properties.

mapping. The expansion of a view tuple t_v, denoted t_v^{exp}, is obtained by replacing t_v by the base relations in this view definition. Existentially quantified variables in the definition are replaced by fresh variables in t_v^{exp}. Clearly this expansion t_v^{exp} will appear in the expansion of any rewriting using t_v.

DEFINITION 4.1. (**tuple-core**) *Let t_v be a view tuple of view v for a minimal query Q. A* tuple-core *of t_v is a* maximal *collection G of subgoals in the query Q, such that there is a containment mapping μ from G to the expansion t_v^{exp} of t_v, and μ has the following properties:*

1. *μ is a one-to-one mapping, and it maps the arguments in G that appear in t_v as is the identity mapping on arguments.*
2. *Each distinguished variable X in G is mapped to a distinguished variable in t_v^{exp} (moreover, by Property (1), $\mu(X) = X$).*
3. *If a nondistinguished variable X in G is mapped under μ to an existential variable in t_v's expansion, then G includes all subgoals in Q that use this variable X.*

□

The purpose of these properties is to make sure when we construct a rewriting using view tuples whose tuple-cores cover all query subgoals, the containment mappings of these core-tuples can be combined seamlessly to form a containment mapping from the query to the rewriting's expansion. In particular, Property (1) is based on Lemma 4.1. Properties (2) and (3), which are satisfied by any containment mapping from the query to a rewriting expansion, are also used in the MiniCon algorithm. A view tuple can have an empty tuple-core. As expected:

LEMMA 4.2. *A view tuple for a minimal query has a unique tuple-core.* □

The unique tuple-core of a view tuple t_v is denoted by $\mathcal{C}(v_t)$.

EXAMPLE 4.1. *For an example, consider the following query and views:*

$$\begin{array}{lll} \textit{Query } Q\text{:} & q(X,Y) & \text{:- } a(X,Z), a(Z,Z), b(Z,Y) \\ \textit{Views } V_1\text{:} & v_1(A,B) & \text{:- } a(A,B), a(B,B) \\ V_2\text{:} & v_2(C,D) & \text{:- } a(C,E), b(C,D) \end{array}$$

A canonical database D_Q of the query includes $a(x,z)$, $a(z,z)$, and $b(z,y)$. By applying the view definitions on D_Q, we have $\mathcal{V}(D_Q) = \{v_1(x,z), v_1(z,z), v_2(z,y)\}$. Thus the set of view tuples is $\mathcal{T}(Q,\mathcal{V}) = \{v_1(X,Z), v_1(Z,Z), v_2(Z,Y)\}$. The table shows the tuple-cores for the three view tuples.

view tuple t_v	expansion t_v^{exp}	tuple-core $\mathcal{C}(t_v)$	mapping μ from $\mathcal{C}(t_v)$ to t_v^{exp}
$v_1(X,Z)$	$a(X,Z), a(Z,Z)$	$a(X,Z), a(Z,Z)$	$X \rightarrow X, Z \rightarrow Z$
$v_1(Z,Z)$	$a(Z,Z), a(Z,Z)$	$a(Z,Z)$	$Z \rightarrow Z$
$v_2(Z,Y)$	$a(Z,E), b(Z,Y)$	$b(Z,Y)$	$Z \rightarrow Z, Y \rightarrow Y$

Table 2: Tuple-cores for the three view tuples in Example 4.1.

By using the three tuple-cores, the only minimum cover of the query subgoals is the union of the tuple-cores of $v_1(X,Z)$ and $v_2(Z,Y)$, which yields the following GMR of the query:

$$q(X,Y) \text{ :- } v_1(X,Z), v_2(Z,Y)$$

□

For another example, let us derive the tuple-cores of the five view tuples in the car-loc-part example (we omit the details that they *are* view tuples as trivial in this example). The tuple-cores for $v_1(M,a,C)$, $v_2(S,M,C)$, $v_4(M,a,C,S)$ and $v_5(M,a,C)$ are identical to the body of the corresponding rules, with variable D replaced by constant a. View tuple $v_3(S)$, though, has an empty tuple-core, since the only possible mapping from a collection of subgoals of Q to $v_3(S)^{exp}$ that satisfies property (3) of Definition 4.1, is: $M \rightarrow M_3, a \rightarrow a, C \rightarrow C_3, S \rightarrow S$. (To avoid confusion, in the definition of v_3, we replace variable M by variable M_3, and variable C by variable C_3.) However, this mapping does not satisfy property (2), since it maps a distinguished variable C in Q to a nondistinguished variable C_3 in $v_3(S)^{exp}$.

4.2 Using tuple-cores to cover query subgoals

The second step of CoreCover finds a minimum number of view tuples to cover query subgoals. This problem can be modeled as a classic *set-covering problem* [8]. Notice by the construction of the tuple-cores, a containment-mapping check is not needed in this step. This step is based on the following theorem:

THEOREM 4.1. *For a minimal query Q and a set of views $\mathcal{V}$, let P be a query that has the head of Q and uses only view tuples in $\mathcal{T}(Q,\mathcal{V})$ in its body. P is a rewriting of Q if and only if the union of the tuple-cores of its view tuples includes all the query subgoals in Q.* □

COROLLARY 4.1. *For a minimal query Q and a set of views $\mathcal{V}$, each GMR of Q using view tuples in $\mathcal{T}(Q,\mathcal{V})$ corresponds to a minimum cover of the query subgoals using the tuple-cores of the view tuples.* □

For instance, consider the tuple cores of the view tuples in car-loc-part example. The minimum cover of the query subgoals is to use the tuple core of view tuple $v_4(M,a,C,S)$, which yields the GMR P_4 of the query. Figure 4 summarizes the CoreCover algorithm.

The complexity of the algorithm CoreCover is exponential, since the problem of finding whether there exists a rewriting is $\mathcal{NP}$-hard [16]. The running time of the algorithm, though, depends mostly on the number of view tuples produced in the second step. Since this number tends to be small in practice, the algorithm performs efficiently in the later steps, as shown by our experimental results in Section 7.

4.3 Comparison with the MiniCon algorithm

CoreCover and MiniCon [20] share the same observation of the Properties (2) and (3) in Definition 4.1, which should be satisfied by any mapping from query subgoals to a view subgoal that can be used in a rewriting. Since we want to find equivalent rewritings, rather than contained rewritings, the different goal gives us the chance to develop a more efficient algorithm. In particular, given the fact that there is a containment mapping from the expansion of an equivalent rewriting to the query, CoreCover limits the search space for useful view literals by applying the view definitions on the canonical database of the query. In other words, this containment mapping helps CoreCover not to consider *all* possible head homomorphisms on the views, which could be a huge set.

```
Algorithm CoreCover: Find rewritings with minimum
    number of subgoals.
Input:  • $Q$: A conjunctive query.
        • $\mathcal{V}$: A set of conjunctive views.
Output: A set of rewritings using view tuples with
        minimum number of subgoals.
Method:
  (1) Minimize $Q$ by removing its redundant subgoals.
      Let $Q_m$ be the minimal equivalent.
  (2) Construct a canonical database $D_{Q_m}$ for $Q_m$.
      Compute the view tuples $\mathcal{T}(Q_m, \mathcal{V})$ by applying
      the view definitions $\mathcal{V}_m$ on the database.
  (3) For each view tuple $t \in \mathcal{T}(Q_m, \mathcal{V})$, compute its
      tuple-core $\mathcal{C}(t)$.
  (4) Use the nonempty tuple-cores to cover the query
      subgoals in $Q_m$ with minimum number of tuple-cores.
      For each cover, construct a rewriting by combining
      the corresponding view tuples.
```

Figure 4: The algorithm CoreCover.

Another advantage that the new goal gives CoreCover is that, each tuple-core of a view tuple includes the *maximal* subset of query subgoals that satisfy the three properties in Definition 4.1. Correspondingly, the "MCD" concept used in MiniCon includes a *minimal* subset of query subgoals. The reason MCD finds a minimal subset of query subgoals is that it tries to find maximally-contained rewritings, and each MCD should be as relaxing as possible, so that all MCDs can be combined. In our case, since we are finding equivalent rewritings, we are more aggressive to cover as many query subgoals as possible using a view tuple. As a consequence, in the last step of CoreCover, the tuple-cores of a set of view tuples that form a rewriting can overlap. That is, a query subgoal can be covered by two tuple-cores. In the second step of MiniCon, the MCDs that form a contained rewriting do not overlap.

Since MiniCon does not aim at generating efficient rewritings, it may produce some rewritings with redundant subgoals, as shown by the following example.

EXAMPLE 4.2. *Consider the following query Q and views $V_1, \dots, V_{k-1}$:*

$$
\begin{array}{lll}
Q\text{:} & q(X,Y) & \text{:- } a_1(X,Z_1), b_1(Z_1,Y), \\
 & & \vdots \\
 & & a_k(X,Z_k), b_k(Z_k,Y). \\
V\text{:} & v(X,Y) & \text{:- } \textit{same as above} \\
V_1\text{:} & v_1(X,Y) & \text{:- } a_1(X,Z_1), b_1(Z_1,Y) \\
 & & \vdots \\
V_{k-1}\text{:} & v_{k-1}(X,Y) & \text{:- } a_{k-1}(X,Z_{k-1}), b_{k-1}(Z_{k-1},Y)
\end{array}
$$

For view V, algorithm CoreCover computes only one view tuple $V(X,Y)$, whose tuple-core includes all *the $2k$ subgoals in Q. In addition, CoreCover also computes a view tuple $v_i(X,Y)$ for each of the rest $k-1$ views. Thus CoreCover creates only one rewriting P with the minimum number of subgoals:*

$$P: \; q(X,Y) \text{ :- } v(X,Y)$$

Correspondingly, for view V, MiniCon generates k different MCDs, each MCD covering two query subgoals: $a_i(X,Z_i)$ and $b_i(Z_i,Y)$. In addition, MiniCon also produces an MCD for each of the rest $k-1$ views. Thus it produces rewritings with redundant subgoals. Notice that the minimization step described in [20] after running the MiniCon algorithm still cannot generate this rewriting P. □

5. COST MODEL M_2: COUNTING SIZES OF RELATIONS

In this section we study cost model M_2 that considers sizes of view relations and intermediate relations in a physical plan. We show that the space of all minimal rewritings that use view tuples is guaranteed to include an optimal rewriting of a query under M_2, if the query has a rewriting.

5.1 A search space for optimal rewritings under M_2

The following lemma helps us find a search space for optimal rewritings under M_2.

LEMMA 5.1. *Under cost model M_2, for any rewriting P of a query Q using views $\mathcal{V}$, there is a minimal rewriting P' that uses only view tuples in $\mathcal{T}(Q,\mathcal{V})$, such that P' is at least as efficient as P.* □

Under cost model M_2, plan P_2 in the car-loc-part example is at least as efficient as plan P_1, since there is a containment mapping from P_1 to P_2, such that all the subgoals of P_2' are images under the mapping.

THEOREM 5.1. *For a query Q and a set of views $\mathcal{V}$, the space of minimal writings using view tuples in $\mathcal{T}(Q,\mathcal{V})$ is guaranteed to include an optimal rewriting under cost model M_2, if the query has a rewriting.* □

By Theorem 4.1 in Section 4, we can modify the algorithm CoreCover to get another algorithm CoreCover* that finds all minimal rewritings using view tuples for a query. The only difference between these two algorithms is that in the last step, CoreCover finds all *minimum* sets of view-tuples whose tuple-cores cover query subgoals, while CoreCover* considers *all* sets of view-tuples to cover the query subgoals. The view tuples that have an empty tuple-core are also used by CoreCover*. By Theorem 5.1, these minimal rewritings guarantee to include an optimal rewriting under cost model M_2, if the query has a rewriting.

As shown by the minimal rewriting P_3 in the car-loc-part example, subgoal $v_3(S)$ can be used to improve the efficiency of the plan, although it does not cover any query subgoal. In general, some view subgoals in a minimal rewriting may be removed without changing the equivalence to the original query, but these view subgoals can serve as *filtering subgoals* to reduce the sizes of intermediate relations. The optimizer can do a cost-based analysis, and decide whether adding some filtering subgoals to a rewriting can make the rewriting more efficient.

5.2 Concise representation of minimal rewritings

In the case where there are many views that can be used to answer a query, the number of view tuples could be large. For instance, consider the case where we have n views that are exactly the same as the query. Then there can be n view tuples, and each has a tuple-core that includes all the query subgoals. Then there can be $2^n - 1$ minimal rewritings of the query.

We propose the following solution to the problem. First, we partition all views into equivalence classes, such that all the views in each class are equivalent as queries. When we run the CoreCover algorithm, we only select a view from each class as a representative. Second, after the view tuples are computed, we also partition these view tuples into equivalence classes, such that all the view tuples in each class have the same tuple-core, i.e., they cover the same set of query subgoals.

Our solution has several advantages. (1) There is a small number of groups of rewritings, with each group having specific properties that might facilitate a more efficient algorithm for the optimizer. (2) The number of view tuples that need to be considered by CoreCover to cover the query subgoals is bounded by the number of query subgoals, thus it becomes *independent from the number of views.* (3) The optimizer can find efficient physical plans by considering the "representative rewritings," and then decide whether each rewriting can become more efficient by adding view tuples as filtering subgoals. The optimizer uses the information about the sizes of relations and selectivity of joins to make this decision. (4) The optimizer can replace a view tuple in a rewriting with another view tuple in the same equivalence view-tuple class, and yet get a new rewriting to the query. Our experiments in Section 7 will show that this solution helps the CoreCover algorithm achieve good performance.

5.3 Generalization of cost model M_2

The key reason that cost model M_2 allows us to restrict the search space in minimal rewritings using view tuples is that M_2 has what we called the property of *containment monotonicity*. That is, a cost model M is *containment monotonic* if for any two rewritings P_1 and P_2, if the following two conditions

1. there is a containment mapping from P_1 to P_2;
2. all subgoals in P_2 are images under the mapping;

can imply $cost_M(P_2) \leq cost_M(P_1)$. Theorem 5.1 can be generalized to any cost model that is containment monotonic.

6. COST MODEL M_3: DROPPING NON-RELEVANT ATTRIBUTES

Cost model M_3 improves M_2 by considering the fact that after computing an intermediate relation in a physical plan, some attributes can be dropped. In this section, we first give an example to show that if the optimizer uses the traditional supplementary-relation approach to decide what attributes to drop, the rewritings using view tuples might not yield an optimal physical plan under M_3. Then we propose a heuristic that can be taken by the optimizer to drop more attributes without changing the final answer of the evaluation, thus producing a more efficient physical plan.

6.1 Dropping attributes using the supplementary-relation approach

Recall that in cost model M_3, a physical plan F of a rewriting P is a list $g_1^{\bar{X}_1}, \ldots, g_n^{\bar{X}_n}$ of the subgoals in P, with each subgoal g_i annotated with a set of attributes $\bar{X}_i$ that can be dropped after subgoal g_i is processed in the sequence. Given a rewriting P, the optimizer considers all possible orderings of the subgoals, and decides the dropping strategy for each ordering. By taking the supplementary-relation approach, for an order of subgoals $g_1, \ldots, g_n$, after subgoal g_i is processed, the optimizer drops the nonrelevant arguments that are not used in subsequent subgoals or in the head of P. The corresponding supplementary relation SR_i is the $SR_{i-1} \bowtie g_i$ with the nonrelevant arguments dropped.

The following example shows that by taking this approach, the optimizer might miss an optimal physical plan under cost model M_3, if the rewriting generator passes to it only rewritings using view tuples.

EXAMPLE 6.1. *Consider the following query, views, and rewritings:*

Query:	Q:	$q(A)$	:- $r(A,A), t(A,B), s(B,B)$
Views:	V_1:	$v_1(A,B)$	:- $r(A,A), s(B,B)$
	V_2:	$v_2(A,B)$	:- $t(A,B), s(B,B)$
Rewritings:	P_1:	$q(A)$	:- $v_1(A,B), v_2(A,C)$
	P_2:	$q(A)$	:- $v_1(A,B), v_2(A,B)$

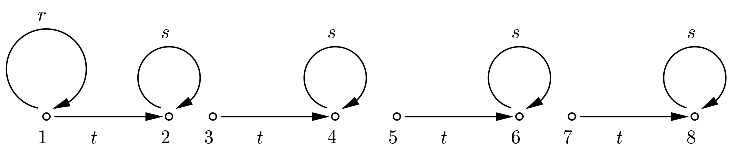


Figure 5: Base relations.

Rewriting P_2 is the only minimal rewriting of Q using the two view tuples $v_1(A,B)$ and $v_2(A,B)$, while rewriting P_1 uses a fresh variable C in its second subgoal. Consider the database shown in Figure 5. The three base relations (r, s, and t) and two view relations (v_1 and v_2) are:

r	s	t	v_1	v_2
$\langle 1,1\rangle$	$\langle 2,2\rangle$	$\langle 1,2\rangle$	$\langle 1,2\rangle$	$\langle 1,2\rangle$
	$\langle 4,4\rangle$	$\langle 3,4\rangle$	$\langle 1,4\rangle$	$\langle 3,4\rangle$
	$\langle 6,6\rangle$	$\langle 5,6\rangle$	$\langle 1,6\rangle$	$\langle 5,6\rangle$
	$\langle 8,8\rangle$	$\langle 7,8\rangle$	$\langle 1,8\rangle$	$\langle 7,8\rangle$

By taking the supplementary-relation approach, the physical plans of P_1 are more efficient than those of P_2. To see why, consider an order $O_2 = [v_1(A,B), v_2(A,B)]$ of subgoals in P_2, and a corresponding order $O_1 = [v_1(A,B), v_2(A,C)]$ of P_1. Order O_2 yields a physical plan

$$F_2 = [v_1(A,B)^{\{\}}, v_2(A,B)^{\{B\}}]$$

In particular, its first supplementary relation needs to keep attributes A and B, since both will be used later. This supplementary relation includes all the four tuples in v_1. Order O_1 yields a physical plan

$$F_1 = [v_1(A,B)^{\{B\}}, v_2(A,C)^{\{C\}}]$$

Its first supplementary relation does not keep attribute B, since B is not used by the second subgoal or the head. This supplementary relation has only one tuple $\langle 1\rangle$. The rest costs of F_1 and F_2 are the same. Thus, $cost_{M_3}(F_1) < cost_{M_3}(F_2)$. If we reverse the two subgoals in the two orderings, the new physical plan of P_1 is still more efficient than that of P_2. □

A minimal rewriting using view tuples may fail to generate an optimal physical plan under M_3 because the variables in the rewriting are made as restrictive as possible by only using the variables in the query. Then view literals in a rewriting might be removed while obtaining the equivalence to the query. However, if the optimizer takes the supplementary-relation approach to decide what attributes to drop, these restrictive variables might not be dropped, since some may be used later in a sequence of subgoals.

The reason that P_1 is more efficient than P_2 is that a physical plan of P_1 has the freedom to drop the second argument after processing its first subgoal. However, P_2 needs to keep the argument, since this argument will be used later in the second subgoal to do a comparison. Now we show that if the optimizer can be "smarter" by using the information about the query and views, it can do better than the supplementary-relation approach.

6.2 A heuristic for an optimizer to drop attributes

We give a heuristic that helps the optimizer drop more attributes than the supplementary-relation approach. Intuitively, given a rewriting P of a query Q, the optimizer considers all orderings of the subgoals in P. For each ordering $O = g_1, \ldots, g_n$, it considers what attributes can be dropped after subgoal g_i is processed without changing the final result of the computation.

For a variable Y that appears in the intermediate relation IR_i, let us consider in what case we can drop Y without changing the result of the computation. As in the supplementary-relation approach, if Y does not appear in subsequent subgoals or the head, it can be dropped. However, even if Y appears in a subsequent subgoal, it might still be dropped, as shown by the variable B in rewriting P_2 in Example 6.1. Notice:

> Dropping Y will *not* change the result of the computation if and only if, should we rename Y in $g_1, \ldots, g_i$ with a fresh variable, the corresponding new query P' is still an equivalent rewriting of Q.

Therefore, for each variable Y that appears in $g_1, \ldots, g_i$, the optimizer adds Y to the annotation X_i (i.e., the set of attributes that can be dropped) if one of the following conditions is satisfied:

- If Y does not appear in subsequent subgoals or the head of P (as in the supplementary-relation approach);
- If Y appears in a subsequent subgoal, but after replacing the Y instances in $g_1, \ldots, g_i$ with a fresh variable Y', the new query P' using views is still an equivalent rewriting of the original query Q. (This equivalence is done by testing the equivalence between Q and the expansion of P'.)

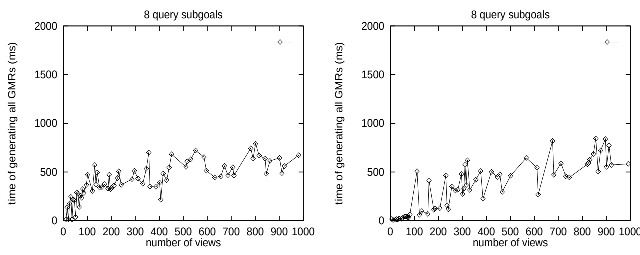
In the second case, dropping a variable Y that appears in a subsequent subgoal $g_k(\ldots, Y, \ldots)$ means we might remove an equality comparison between GSR_{k-1} and $g_k(\ldots, Y, \ldots)$, which could increase the size of GSR_k. Thus the optimizer needs to make the tradeoff between dropping Y and removing this comparison by using the information about the sizes of view relations and generalized supplementary relations.

7. EXPERIMENTAL RESULTS

We did experiments to study the search spaces for optimal rewritings under cost models M_1 and M_2, and evaluate the performance of the CoreCover algorithm. We studied different shapes of queries, such as chain queries, star queries, and randomly generated queries [23]. We implemented a query generator that takes as input parameters such as: (1) number of base relations; (2) number of attributes in a base relation; (3) number of views; (4) number of subgoals in a view; (5) number of subgoals in a query; (6) shape of queries and views. In the experiments, queries and views were set to have the same parameters, except that they might have different number of subgoals. For the same number of views, we ran 40 queries and computed their average measures. The algorithm CoreCover was implemented in Java. The experiments were run on a dual-processor Sun Ultra 2 workstation, running SunOS 5.6 with 256 MB memory.

7.1 Star queries

We first considered star queries. Each query had 8 subgoals, and each view randomly had 1, 2, or 3 subgoals. We ignored queries that did not have rewritings. Figure 6 (a) shows the running time for CoreCover to get all globally-minimal rewritings (GMRs) as the number of views increased, if all variables were distinguished. As the number of views increased, the time of finding all GMRs did not increase steadily. Instead, the time was bound in the range from 0ms to 1 second. On the average, it took CoreCover about 500ms to generate all GMRs for a query. Even if there were 1000 views, the time was still less than 1 second. Figure 6 (b) shows the running time for CoreCover to generate GMRs if one variable was distinguished.



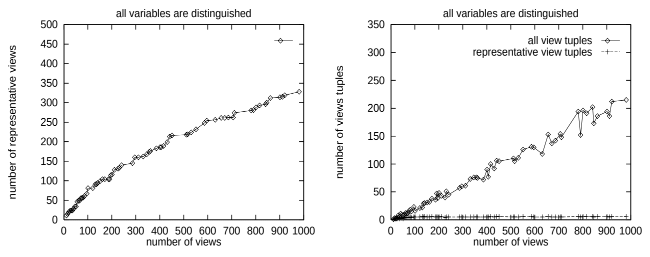
(a) All variables are distinguished

(b) 1 variable is nondistinguished

Figure 6: Time for CoreCover to generate all GMRs for star queries.

The reason CoreCover has good efficiency and scalability is that we can group views and view tuples into equivalence classes, respectively. From each equivalence class of views, we selected only one representative that was equivalent as queries to other views in the class. (See [18] for more work on similar considerations.). Similarly, from each equivalence class of view tuples, we also selected one representative view tuple that had the same tuple-core as others. Therefore, the number of representative view tuples depends on the number of query subgoals only, and it is independent from the number of views. Notice that the running time includes the time of grouping views and view tuples into equivalence classes.

Although in the early stage of CoreCover, we paid extra cost to test view equivalence by testing query containments, this extra cost paid off later when the number of views was more than 100.



(a) Views (b) View tuples

Figure 7: Number of equivalence classes for star queries.

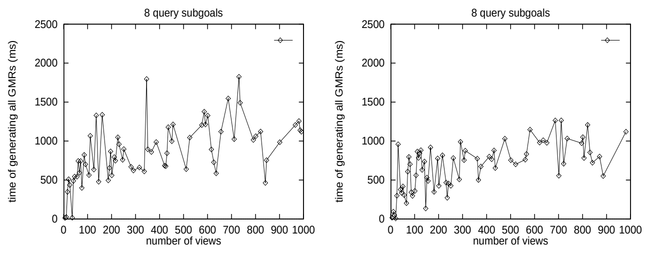
For instance, consider the case where all variables were distinguished. Figure 7 (a) shows that as the number of views increased, the number of view equivalence classes also increased, but with a decreasing slope. When there were 1000 views, there were only about 350 equivalent view classes. Figure 7 (b) shows that the number of equivalence classes of view tuples was almost a constant (less than 10) as the number of views increased, while the number of view tuples increased to more than 200.

7.2 Chain queries

We then considered chain queries, and had the similar observation. Each query had 8 subgoals, and each view had 1, 2, and 3 subgoals randomly. All relations were binary. If we only kept the head and tail variables of the chain as the head arguments of the query and views, then there are very few rewritings generated. Thus, we ran our experiments by first considering all variables as distinguished, and then let a few variables be nondistinguished. For the views that had only one subgoal, both variables were still distinguished. We ignored queries that did not have rewritings. Figure 8 (a) shows the running time of CoreCover if all variables were distinguished, and Figure 8 (b) shows the running time if one variable was nondistinguished.

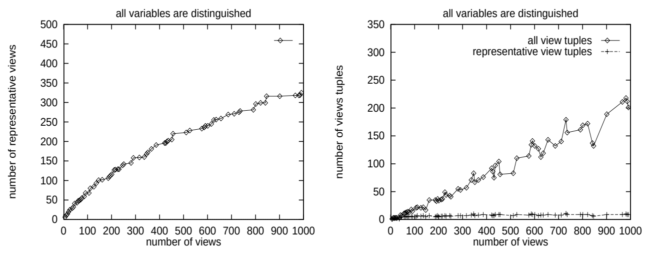
Again, the CoreCover algorithm showed good efficiency and scalability. For instance, in the case where all variables were distinguished, it took the algorithm less than 2 seconds to generate all GMRs for a query when there were 1000 views. In the case where one variable was distinguished, it took the algorithm less than 1.4 seconds to generate all GMRs for a query when there were 1000 views. To illustrate the reason, Figure 9 (a) shows that as the number of views increased, the number of equivalence view classes increased with a decreasing slope. Figure 9 (b) shows that as the number of views increased, the number of representative view tuples was almost a constant.

In summary, our experiments illustrated two points. (1) The CoreCover algorithm has good efficiency and scalability. (2) By grouping views and view tuples into equivalence classes respectively, we can reduce the number of views and view tuples used in the algorithm, thus the algorithm can perform efficiently.



(a) All variables are distinguished (b) 1 variable is nondistinguished

Figure 8: Time of generating all GMRs of chain queries.



(a) Views (b) View tuples

Figure 9: Number of equivalence classes for chain queries.

8. CONCLUSION AND DISCUSSION

In this paper, we studied the problem of generating efficient rewritings using views to answer a query. That is, how to generate a search space of rewritings that is guaranteed to include a rewriting with an optimal physical plan. We studied three cost models. Under the first cost model M_1 that considers the number of subgoals in a plan, we gave a search space for optimal rewritings for a query. We analyzed the internal relationship of all rewritings of a query using views, and developed an efficient algorithm, CoreCover, for finding rewritings with the minimum number of subgoals.

We then considered a cost model M_2 that counts the sizes of relations in a physical plan. We also gave a search space for finding optimal rewritings under M_2. Surprisingly, we need to consider the fact that introduction of more view subgoals might make a rewriting more efficient. Finally, we considered a cost model M_3 that allows some nonrelevant attributes to be dropped during the evaluation of a plan without changing the result of the computation. We proposed a heuristic for an optimizer to drop more attributes than the traditional supplementary-relation approach. Experiments showed that the CoreCover algorithm has good efficiency and scalability. Among other subtleties, this good result is also due to the fact that the algorithm (i) considers only a small number of *relevant* view tuples for the rewritings, and (ii) uses a concise representation of these view tuples.

Currently we are investigating how to develop an improved optimizer to optimize rewritings using the information of the query and views. The heuristic for M_3 is an example. We are also extending our work to other cases, such as the case where the query and views have built-in predicates, and the case where we want to find maximally-contained rewritings of the query. In both cases, it is known that a rewriting of a query can be a union of conjunctive queries. Thus the challenge is how to evaluate the performance of a union of conjunctive queries, as shown by the following example borrowed from [16]. Consider the query and views:

$$\begin{array}{lll} \text{Query } Q\text{:} & q(X,Y,U,W) & \text{:- } p(X,Y), r(U,W), r(W,U) \\ \text{Views:} & v_1(A,B,C,D) & \text{:- } p(A,B), r(C,D), C \leq D \\ & v_2(E,F) & \text{:- } r(E,F) \end{array}$$

The following rewriting P_1 of Q using the two views is a union of two conjunctive queries, and it uses only the variables in Q:

$$\begin{array}{lll} P_1\text{:} & q(X,Y,U,W) & \text{:- } v_1(X,Y,U,W), v_2(W,U) \\ & q(X,Y,U,W) & \text{:- } v_1(X,Y,W,U), v_2(U,W) \end{array}$$

However, the following rewriting P_2 has only one conjunctive query, and it uses new variables C and D not in the query.

$$P_2: \; q(X,Y,U,W) \text{ :- } v_1(X,Y,C,D), v_2(U,W), v_2(W,U)$$

Notice that P_2 uses fewer conjunctive queries than P_1. However, this fact does not imply that P_2 is always more efficient than P_1, since P_2 uses three view subgoals, while each conjunctive query in P_1 uses only two view subgoals. Currently we are investigating how to compare the efficiency of two unions of conjunctive queries, and find efficient rewritings under certain cost models.

9. REFERENCES

[1] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.

[2] F. N. Afrati, M. Gergatsoulis, and T. G. Kavalieros. Answering queries using materialized views with disjunctions. In *ICDT*, pages 435–452, 1999.

[3] R. J. Bayardo Jr. et al. Infosleuth: Semantic integration of information in open and dynamic environments (experience paper). In *SIGMOD*, pages 195–206, 1997.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS*, pages 269–283, 1987.

[5] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. *STOC*, pages 77–90, 1977.

[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[7] S. S. Chawathe et al. The TSIMMIS project: Integration of heterogeneous information sources. *IPSJ*, pages 7–18, 1994.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Presss, 1990.

[9] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *PODS*, pages 109–116, 1997.

[10] D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of run-time management of data intensive web-sites. In *Proc. of VLDB*, pages 627–638, 1999.

[11] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, 2000.

[12] G. Grahne and A. O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *ICDT*, pages 332–347, 1999.

[13] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of VLDB*, pages 276–285, 1997.

[14] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution engine for data integration. In *SIGMOD*, pages 299–310, 1999.

[15] A. Levy. Answering queries using views: A survey. *Technical report, Computer Science Dept., Washington Univ.*, 2000.

[16] A. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, pages 95–104, 1995.

[17] A. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, pages 251–262, 1996.

[18] C. Li, M. Bawa, and J. D. Ullman. Minimizing view sets without losing query-answering power. In *ICDT*, pages 99–113, 2001.

[19] P. Mitra. An algorithm for answering queries efficiently using views. In *Proceedings of the Australasian Database Conference*, 2001.

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

[21] X. Qian. Query folding. In *ICDE*, pages 48–55, 1996.

[22] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

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

[24] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Proc. of VLDB*, 1997.

[25] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes II: The New Technologies.* Computer Science Press, New York, 1989.

[26] J. D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40, 1997.

[27] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *SIGMOD*, pages 105–116, 2000.