Optimizing Queries with Materialized Views

Surajit Chaudhuri Ravi Krishnamurthy Spyros Potamianos* Kyuseok Shim†
Hewlett-Packard Laboratories, Palo Alto
Email: lastname@hpl.hp.com

Abstract

While much work has addressed the problem of maintaining materialized views, the important question of optimizing queries in the presence of materialized views has not been resolved. In this paper, we analyze the optimization question and provide a comprehensive and efficient solution. Our solution has the desirable property that it is a simple generalization of the traditional query optimization algorithm.

1 Introduction

The idea of using materialized views for the benefit of improved query processing has been proposed in the literature more than a decade ago. In this context, problems such as definition of views, composition of views, maintenance of views [BC79, KP81, SI84, BLT86, CW91, Rou91, GMS93] have been researched but one topic has been conspicuous by its absence. This concerns the problem of the judicious use of materialized views in answering a query.

It may seem that materialized views should be used to evaluate a query whenever they are applicable. In fact, blind applications of materialized views may result in significantly worse plans compared to alternative plans that do not use any materialized views. Whether the use of materialized views will result in a better or a worse plan depends on the query and the statistical properties of the database. Since queries are often generated using tools and since the statistical property of databases are time-varying, it should be the responsibility of the optimizer to consider the alternative execution plans and to make a cost-based decision whether or not to use materialized views to answer a given query on a given database. Such enumeration of the possible alternatives by the optimizer must be syntax independent and efficient. By syntax independent, we mean that the set of alternatives enumerated by the optimizer (and hence the choice of the optimal execution plan) should not depend on whether or not the query explicitly references materialized views. Thus, the optimizer must be capable of considering the alternatives implied by materialized views. In particular, a materialized view may need to be considered even if the view is not directly applicable (i.e., there is no subexpression in the query that syntactically matches the view). Also, more than one materialized views may be relevant for the given query. In such cases, the optimizer must avoid incorrect alternatives where mutually exclusive views are used together while considering use of mutually compatible views.

The following examples illustrate the issues in optimizing queries with materialized views. The first example emphasizes the importance of syntax independence and also shows that sometimes use of materialized views may result in worse plans. The second example illustrates the subtleties in syntax independent enumeration discussed above. The examples use a database containing an employee relation Emp(name, duo, sal, age) and a department relation Dept(dno, size, loc).

Example 1.1: Let there be a materialized view Executive(name, duo, sal) that contains all employees whose salary is greater than 200k. Consider the query that asks for employees (and their department number) whose salary is greater than 200k and who are in the department with duo = 0419. If the relation Emp has no index on duo, then it is better to access the materialized view Executive even though the user presents a query which does not refer to the materialized view Executive. This example illustrates that the use of a materialized view can be beneficial even if a query does not refer to the materialized view explicitly. On the other hand, it may be possible to obtain a cheaper plan by not using a materialized view even if the query does reference the view explicitly. Consider
the query that asks for all executives in dno = 419. This query explicitly refers to the materialized view Executive. However, if there is an index on the dno attribute of the relation Emp, then it may be better to expand the view definition in order to use the index on dno attribute of the relation Emp. Thus, the choice between a materialized view and a view expansion must be cost-based and syntax-independent. 

Example 1.2: The purpose of this example is to illustrate the nature of enumeration of alternatives that arise when materialized views are present. Consider the query which asks for employees who earn more than 220k. Although the materialized view for Executive does not syntactically match any subexpression of the query, it could still be used to answer the above query by retaining the selection condition on salary. Next, we illustrate a case of mutually compatible use of materialized views. Consider the query that asks for employees who earn more than 200k and who have been working in departments of size > 30 employees. If there is a materialized view Large.Dept(dno, loc) containing all departments (with their location) where number of employees exceed 30, then the latter may be used along with Executive to answer the query. Finally, there are cases where uses of two materialized views are incompatible. Assume that a materialized view Loc.Emp(name, size, loc) is maintained that records for each employee the location of her work. If the query asks for all employees who work in large departments located in San Francisco, then each of Loc.Emp and Large.Dept materialized views help generate alternative executions. But, uses of these two materialized views are mutually exclusive, i.e., they cannot be used together to answer a query.

The presence of materialized views and the requirement of syntax-independent optimization has the effect of increasing the space of alternative executions available to the optimizer since the latter must consider use and non-use of the materialized views. Since the query optimization algorithms take time exponential in the size of the queries, we must also ensure that the above enumeration of alternatives is done efficiently so as to minimize the increase in optimization time. Furthermore, we must also recognize the reality that for our proposal to be practical and immediately useful, it is imperative that our proposal be a generalization of the widely accepted optimization algorithm [SAC+79].

In this paper, we show how syntax-independent enumeration of alternative executions can be done efficiently. Our proposal constitutes a simple extension to the cost-based dynamic programming algorithm of [SAC+79] and ensures the optimality of the chosen plan over the extended execution space. The simplicity of our extension makes our solution practically acceptable. Yet, our approach proves to be significantly better than any simple-minded solution to the problem that may be adopted (See Section 4).

For the rest of the paper, we assume that the query as well as the materialized views are conjunctive queries, i.e., the Select-Project-Join expressions such that the Where clause consists of a conjunction of simple predicates (e.g., =, <, ≥) only. Thus, the query has no aggregates or group-by clause. We will use the domain-calculus notation [Ul89] to express conjunctive queries. However, our results extend to any Select-Project-Join queries and extensions beyond the above class are also considered in the full version of the paper [CKPS94].

The rest of the paper is organized as follows. We begin with an overview of our approach. In Section 3, we show how the equivalent queries may be formulated from the given query and the materialized views. In Section 4, we present the algorithm for join enumeration. We also contrast the efficiency of our algorithm with the existing approaches and present an experimental study. In Section 5, we discuss further generalizations of our approach. Section 6 mentions related work.

2 Overview of Our Approach

In traditional query processing systems, references to views in a query are expanded by their definitions, resulting in a query that has only base tables. Relational systems that support views can do such unfolding. However, the presence of materialized views provide the opportunity to fold one or more of the subexpressions in the query into references to materialized views, thus generating additional alternatives to the unfolded query. Therefore, we must convey to the optimizer the information that enables it to fold the subexpressions corresponding to the materialized views.

For every materialized view \( V \), we will define a one-level rule as follows. The left-hand side of the one-level rule is a conjunctive query (body of the view definition) \( L \) and the right-hand side of the rule is a single literal (name of the view). We represent the rule as:

\[ L(x, y) \rightarrow V(x) \]

where the variables \( x \) correspond to projection vari-
ables for the view. The variables \( y \) are variables in the body of the view definition that do not occur among projection variables. We call these rules one-level rules since a literal that occurs in the right side of any of the rules (view-name) does not occur in any left-hand side since the left-hand side may have references to only base tables. Thus, given a set of views that are conjunctive queries, we can generate the corresponding set of one level rules from the SQL view definitions.

Our approach to optimization in the presence of materialized views has three main steps. First, the query is translated in the canonical unfolded form, as is done in today’s relational systems that support views. Second, for the given query, using the one-level rules, we identify possible ways in which one or more materialized views may be used to generate alternative formulations of the query. These two steps together ensure syntax independence. Finally, an efficient join enumeration algorithm, that retrofits the System R style join enumeration algorithm [SAC+79], is used to ensure that the costs of alternative formulations are determined and the execution plan with the least cost is selected.

Since the first step is routinely done in many commercial relational systems, in the rest of the paper, we will focus only on the second and the third steps. In the next two sections, we discuss each of these steps:

- Encode in a data-structure (MapTable) the information about queries equivalent to the given one (Section 3).
- Generalize the traditional join enumeration algorithm so that it takes into account the additional execution space implied by the equivalent queries (Section 4). This is the heart of the paper.

### 3 Equivalent Queries: Generation of MapTable

In this section, we discuss how we can use one-level rewrite rules to derive queries that are equivalent to the given query in the presence of materialized views. This problem of generating equivalent queries in the presence of views has been studied before [YL87] (also see Section 6). In addition to a simplified exposition of the problem for conjunctive queries, the novelty here is in generating an implicit representation of equivalent queries in such a way that the join enumeration phase can exploit it. Our notion of equivalence of queries is as in SQL standard [ISO92]. Thus, we assume that two queries are equivalent if they result in the same bag of tuples over every database.

Intuitively, an equivalent query is generated by identifying a subexpression in the query that corresponds to the left-hand side of one-level rewrite rule. The subexpression is then replaced by the literal in the right-hand side of the rule (i.e., the view name). However, it turns out, that a straightforward substitution could be incorrect.

**Example 3.1:** Consider Example 1.2. The materialized view Loc.Emp is represented by the following rule:

\[
\text{Emp}(name, dno, sal, age), \text{Dept}(dno, size, loc) \rightarrow \text{Loc.Emp}(name, size, loc)
\]

Consider the following query that obtains all employees of age less than 35 who work in San Francisco (SF).

\[
\text{Q(name)} : - \text{Emp}(name, dno, sal, age), age < 35 \text{Dept}(dno, size, SF)
\]

Observe that it is possible to obtain the query \( Q' \) through a naive substitution using the rewrite rule for Loc.Emp.

\[
Q'(name) : -\text{Loc.Emp}(name, size, SF), age < 35
\]

However, clearly, \( Q \) and \( Q' \) are not equivalent queries. In particular, \( Q' \) is unsafe.

Example 3.1 makes the point that a syntactic substitution of the body of a materialized view need not result in an equivalent query. The crux of the problem in Example 3.1 is that the naive approach of replacing a matching subexpression resulted in a query with a “dangling” selection condition that refers to a variable in the subexpression that has been replaced. Besides the fact that a straightforward substitution may be incorrect, additional substitutions may be applicable as seen in the following example.

**Example 3.2:** The presence of the materialized view Executive is represented by the following one-level rule.

\[
\text{Emp}(name, dno, sal, age), sal > 200k \rightarrow \text{Executive}(name, dno, sal)
\]

Consider the following query which asks for employees who work in a department of size at least 30 and who earn more than 220k.

\[
\text{Q(name)} : -\text{Emp}(name, dno, sal, age), sal > 220k \text{Dept}(dno, size, loc), size > 30
\]
Observe that there is no syntactic substitution for the rule for Executive, since there is no renaming such that the literal \( \text{sal} > 200k \) in the one-level rule for the materialized view Executive maps to a literal in \( Q \). However, the following query \( Q' \) is clearly equivalent to \( Q \).

\[
Q'( \text{name} ) : \neg \text{Executive}( \text{name}, \text{dno}, \text{sal} ), \text{sal} > 220k, \\
\text{Dept}( \text{dno}, \text{size}, \text{loc} ), \text{size} > 30
\]

Example 3.2 illustrates a case where although there is no subexpression that syntactically matches the body of the view definition, the materialized view can be applied. This example is specially significant since it illustrates that to be able to use materialized views, we may have to reason with implication (subsumption) between sets of inequality (and may be arithmetic) constraints.

In Section 3.1, we define safe substitution that identifies equivalent queries that result due to applications of one-level rules. In Section 3.2, we explain how safe substitutions are used to construct MapTable, that implicitly stores the queries that are equivalent to the given query. This MapTable is subsequently used in the join enumeration step.

3.1 Safe Substitution

Every safe substitution identifies a subexpression in the given query that may be substituted by a materialized view to generate an equivalent query. As Example 3.2 tells us, presence of inequality constraints needs to be considered in identifying equivalent queries. Accordingly, we adopt the following somewhat more detailed representation of one-level rules that recognizes existence of inequality constraints.

\[
L( x, y ), I( x ) \Rightarrow V( x )
\]

where \( I( x ) \) represents a conjunction of inequality (and may be arithmetic) constraints that involve only the projection variables \( x \) of the rule. However, \( L( x, y ) \) may contain variables \( y \) that are not projection variables.

Example 3.3: Consider the one-level rewrite rule for Executive in Example 3.2. We note that \( I( \text{name}, \text{dno}, \text{sal} ) \equiv \text{sal} > 200k \) and \( L( \text{name}, \text{dno}, \text{sal}, \text{age} ) \equiv \text{Emp}( \text{name}, \text{dno}, \text{sal}, \text{age} ) \). Since \( I \) depends only on \( \text{sal} \), we will abbreviate reference to it as \( I( \text{sal} ) \).

The task of finding a suitable subexpression for substitution begins with renaming of variables in a rule to identify occurrences of the left-hand side of the rule in the query. Let \( r \) be a rule with variables \( V_r \) and \( Q \) be a query with variables \( V_Q \) and constants \( C_Q \).

Definition 3.4: A valid renaming \( \sigma \) of \( r \) with respect to a query \( Q \) is a symbol mapping from \( V_r \) to \( V_Q \) subject to the following two constraints: (a) If \( v \in V_r \) is a projection variable, then \( \sigma( v ) \in V_Q \cup C_Q \). (b) If \( v \in V_r \) is not a projection variable, then \( \sigma( v ) \in V_Q \) and \( \sigma( v ) \neq \sigma( v' ) \) where \( v' \) is any other variable in \( V_r \).

Thus, only projection variables may map to constants. Moreover, no two variables in the rule may map to the same variable in the query unless these two variables are both projection variables. The reader will notice the similarity between valid renaming and the well-known containment mapping [CM77] (cf. [Ull89]). We can now define a notion of safe occurrence and the corresponding safe substitution. In the following definition, the symbol \( \Rightarrow \) stands for logical implication.

Definition 3.5: Given a set of one-level rules \( R \), a query \( Q \) has a safe occurrence of \( R \), if for a rewrite rule \( r \in R \) there is a valid renaming of the rule \( r \) with respect to \( Q \) such that the renamed rule has the form \( L( x, y ), I( x ) \Rightarrow V( x ) \). Furthermore, the following two conditions must hold:

1. The query \( Q \) has the form:
\[
Q( u ) \equiv L( x, y ), I'( x ), G( v )
\]

where each of \( x, y, u \) and \( v \) is a set of variables. These sets may share variables except that \( y \) must be disjoint from \( x, u \) and \( v \).

2. \( I'( x ) \Rightarrow I( x ) \).

The safe substitution corresponding to the above safe occurrence is:

\[
Q'( u ) \equiv V( x ), I'( x ), G( v )
\]

Testing condition (2) above entails checking implication between two sets of inequality constraints. Many efficient algorithms have been proposed that can test such implication (See [Ull89] for an algorithm). Finally, note that in reality, we retain only a subset of inequality constraints in \( I' \), i.e., those constraints that are not subsumed by \( I \).
Observe that the definition of safe occurrence ensures that there could be no dangling selection condition (unlike Example 3.1) when the view replaces its matching subexpression in the query. On the other hand, the above definition does allow for a safe substitution to be possible even if there is no syntactic match between left-hand side of a rule and the query (as in Example 3.2).

Example 3.6: Let us revisit Example 3.2 which illustrated the need for reasoning with inequality constraints. In this example, there is a safe occurrence of the one-level rule for `Executive`. This is true since $I'(\text{sal}) \equiv \text{sal} > 220k$ and $L(\text{name}, \text{dno}, \text{sal}, \text{age}) \equiv \text{Emp}(\text{name}, \text{dno}, \text{sal}, \text{age})$. Furthermore, the following is true:

$$G(\text{dno}, \text{size}, \text{loc}) \equiv \text{Dept}(\text{dno}, \text{size}, \text{loc}), \text{size} > 30$$

From Example 3.3, we note that Since $I'(\text{sal}) \Rightarrow I(\text{sal})$. Hence, the condition for safe occurrence is satisfied and we obtain the following equivalent query

$$Q'(\text{name}) : \neg \text{Executive}(\text{name}, \text{dno}, \text{sal}), \text{sal} > 220k, \text{Dept}(\text{dno}, \text{size}, \text{loc}), \text{size} > 30$$

Queries obtained by safe substitution are equivalent to the original query in any database that stores the materialized view, consistent with its view definition. Formally, this leads to a notion of equivalence of queries with respect to a set of rewrite rules. In the following definition, we say that a database is a a valid database with respect to a set of rules if the left-hand side and the right-hand sides of each rule returns the same bag of tuples over that database.

**Definition 3.7:** Two queries $Q$ and $Q'$ are equivalent with respect to a set of rewrite rules $R$ if they result in the same bag of tuples over any valid database for $R$.

We will denote such an equivalence by $Q \equiv_R Q'$. In case $Q$ and $Q'$ are unconditionally equivalent (i.e., equivalent independent of any rewrite rules), we denote that by $Q \equiv Q'$.

**Lemma 3.8:** If $Q'$ is obtained from $Q$ by a sequence of safe substitutions with respect to a set of rewrite rules $R$, then $Q$ and $Q'$ are equivalent with respect to $R$.

The lemma confirms the soundness of safe substitution and is applicable for bag as well as set semantics.

For equi-join queries with bag semantics the converse of the theorem is true as well. The latter proof exploits unique properties of bag equivalence that are not necessarily true of queries with set semantics. In the full paper, we generalize the above results and also consider the important case of set equivalence [CKPS94], as well as queries that are not necessarily conjunctive queries.

### 3.2 Representation and Enumeration of Safe Substitutions

Intuitively, each safe substitution results in a new query, equivalent to the given one. We encode the equivalent queries by storing the information about safe substitutions in the `MapTable` data structure.

From the definition of safe substitution, it follows that every safe substitution of a query $Q$ with respect to a rule $L(x,y) \rightarrow V(x)$ corresponds to a renaming $\sigma$ for the rule. Therefore, we can encode the information about a safe substitution by the doublet $[\sigma(L), \sigma(V)]$. The first component in the doublet is called the `deletelist` and the second component in the doublet is called the `addliteral`. The `deletelist` denotes the subexpression in the query that is replaced due to the safe substitution $\sigma$ and the `addliteral` denotes the literal that replaces `deletelist`. Since $L$ may have more than one literals, the `deletelist` is a set of literals. However, `addliteral` is a single literal. The algorithm to construct the MapTable for a given query is given below. The last `for` loop iterates over all literals in the query. Its purpose is best explained in the context of the join enumeration algorithm, described in the next section.

**Procedure MakeMapTable(Q, R)**

```
begin
  Initialize MapTable
  for each rewrite rule $r : L \rightarrow V$ in $R$ do
    for each safe substitution $\sigma$ from $r$ to $Q$ do
      MapTable := MapTable $\cup$ $[\sigma(L), \sigma(V)]$
    endfor
  endfor
  for each literal $q \in Q$ do
    MapTable := MapTable $\cup$ $[\{q\}, q]$
  endfor
end
```

**Example 3.9:** In addition to the rules for `Executive` and `Loc_Emp`, consider the following one-level rewrite rule for `Large_Dept`

```
\text{Dept}(\text{dno}, \text{size}, \text{loc}), \text{size} > 30 \rightarrow \text{Large_Dept}(\text{dno}, \text{loc})
```
We illustrate the enumeration of safe substitutions using the above three materialized views.

(i) Consider the following query which asks for employees who work at a department in SF.

\[ \text{Query}(\text{name}) : -\text{Emp}(\text{name}, \text{dno}, \text{sal}, \text{age}), \text{size} > 30 \]
\[ \text{Dept}(\text{dno}, \text{size}, \text{SF}) \]

It can be seen that the MapTable will have the following two doublets.

\[ \{ \text{Dept}(\text{dno}, \text{size}, \text{SF}), \text{size} > 30 \}, \]
\[ \text{Large_Dept}(\text{dno}, \text{SF}) \]
\[ \{ \text{Emp}(\text{name}, \text{dno}, \text{sal}, \text{age}), \text{Dept}(\text{dno}, \text{size}, \text{SF}) \}, \]
\[ \text{Loc_Emp}(\text{name}, \text{size}, \text{SF}) \]

Observe that the doublets correspond to materialized views that are mutually exclusive.

(ii) Consider the query to find employees who earn more than 200k and work in departments with more than 30 employees.

\[ \text{Q'(name)} : -\text{Emp}(\text{name}, \text{dno}, \text{sal}, \text{age}), \text{sal} > 200\text{k}, \]
\[ \text{Dept}(\text{dno}, \text{size}, \text{loc}), \text{size} > 30 \]

It can be seen that the MapTable will have the following two doublets which correspond to applications of mutually compatible materialized views.

\[ \{ \text{Emp}(\text{name}, \text{dno}, \text{sal}, \text{age}), \text{sal} > 200\text{k} \}, \]
\[ \text{Executive}(\text{name}, \text{dno}, \text{sal}) \]
\[ \{ \text{Dept}(\text{dno}, \text{size}, \text{loc}), \text{size} > 30 \}, \]
\[ \text{Large_Dept}(\text{dno}, \text{loc}) \]

Notice that these two doublets implicit represent three alternatives to the given query.

In our implementation of the optimizer, literals of the query are stored in a literal-table and are referenced by unique literal-ids. For example, a query with four literals may be represented as \{1, 2, 3, 4\}. A MapTable entry would be of the form \{\{1, 2\}, 7\} which indicates that the occurrence of the literal that replaces the subexpression corresponding to \{1, 2\} is stored in position 7 of the literal table. For efficiency of access, MapTable is indexed by literal-ids that occur in the deletelist.

The running time of the algorithm \text{MakeMapTable} is linear in the number of safe substitutions. However, it may seem that in order to determine all safe substitutions, exhaustive renaming of the variables in the rule will be required. Such is rarely the case [CKPS94]. Rather, in the likely event where the query has at most one literal for every table name, there can be at most one safe substitution for every rule. We note that such a safe substitution can be found in time linear in the size of the query. For example, in Example 3.2, we know that the literal \text{Emp} in the body of the rule must map to the literal for \text{Emp} in the body of the query. Furthermore, our experience with the optimizer indicates that for a given query, only few rules are applicable. Therefore, for a given query, the size of MapTable is not a dominant factor and finding safe mappings is typically an inexpensive step in optimization [CKPS94].

4 Join Enumeration

In the previous section, we have seen how the information about equivalent queries is implicitly stored in MapTable. The equivalent queries provide the optimizer with an extended execution space since the optimizer can pick a plan from the union of execution spaces of these equivalent queries. Therefore, the challenge is to extend the traditional join enumeration algorithm such that optimality over the extended execution space is ensured.

An obvious solution is to invoke the traditional optimizer repeatedly for each equivalent query. Indeed, this technique was adopted in [CGM90]. Unfortunately, the above approach leads to rederivation of many shared subplans among the equivalent queries, thus leading to significant inefficiency in optimization (See Section 4.3). In contrast, our approach guarantees that no subplan is rederived. We show that while the worst case complexity of other enumeration algorithms could be an exponential function of the number of safe substitutions, our algorithm takes time only linear in the number of safe substitutions. Thus, not only is the enumeration algorithm a simple extension of the traditional approach, it also is an efficient algorithm.

In the first part of this section, we will review the traditional join enumeration algorithm which is widely used in relational systems. Next, we propose our extension to the existing algorithm to enumerate the expanded execution space. We present a result that shows that the algorithm achieves complete enumera-
4.1 Traditional Algorithm

The execution of a query is traditionally represented syntactically as annotated join trees [GHK92] where the internal node is a join operation and each leaf node is a base table. The annotations provide details such as selection conditions, choice of access paths and join algorithms. The set of all annotated join trees for a query that are considered by the optimizer, is traditionally called the execution space of the query. Like many relational optimizers, we will restrict the execution space for each alternative to be its left-deep trees only. Note that in such a case, every execution is a total ordering of joins.

The optimality of a plan is with respect to a cost model. So far as the cost model is concerned, we assume that the cost model assigns a real number to any given plan in the execution space and satisfies the principle of optimality [CLR90].

In this part, we briefly explain the join enumeration algorithm OptPlan (See Figure 2), which is a simplification and abstraction of the algorithm in [SAC+79] (cf. [GHK92]). Let us assume that the query is a join among \( n \) literals where \( n > 2 \). The optimal plan for join of \( n \) relations can be obtained by enumerating \( n \) choices for the last relation to join and for each choice joining the chosen relation with the optimal plan for the remaining \((n-1)\) relations. The optimal plan is the least expensive plan of these \( n \) plans, so constructed. We omitted the details of the actual join methods and other annotations of the actual execution since they are not germane to our discussion here. Note that every subset \( S \) of the above set of all \( n \) relations in a query corresponds to a unique subquery (say, \( Q_S \)). Thus, the optimal plan for every subquery \( Q_S \) of \( Q \) (referred to as a subplan) is constructed exactly once and it is stored in the plantable. All subsequent calls to OptPlan for the same \( Q_S \) looks up the cost of the optimal plan from the table. Since looking up the plantable helps avoid repeated recomputation of the optimal plan, the complexity of the algorithm is \( O(n^{2n-1}) \) (instead of \( O(n!) \)).

4.2 Extended Algorithm

In this section, we discuss the optimization algorithm in the presence of equivalent queries (implicitly) represented in the MapTable. We have presented the algorithm ExOptPlan in Figure 1. The test \( D_i \subseteq Q \) tests whether all the literals that occur in \( D_i \) (a deletelist) also occurs in \( Q \).

As explained in Section 4.1, in the traditional algorithm, complete enumeration of the search space is achieved by repeating the following step for each literal \( q_i \) in the query \( Q \). We construct a plan for the rest of the literals in the query, i.e., the optimal plan for \( Q - q_i \). Putting together the optimal plan for \( Q - q_i \) with \( q_i \) results in the optimal plan for \( Q \) subject to the restriction that \( q_i \) is the last literal being joined.

Our extension follows the above technique for enumeration closely. Recall that MapTable contains the doublets \( \{(q_i), q_i\} \) for each literal \( q_i \) that is in the query \( Q \). It can be seen immediately that if these are the only doublets stored in MapTable, the algorithms OptPlan and ExOptPlan in Figure 1 behave identi-
cally since $P_i$ in ExOptPlan will be no different from $S_i$ in OptPlan. Let us now consider any other doublet that corresponds to a safe substitution. A key observation is that we can ensure exhaustive enumeration if for each such safe substitution $(D_i, a_i)$, we consider all plans where $a_i$ is the last literal to be joined. However, in the unfolded query, there is no occurrence of the materialized view $a_i$. Therefore, instead of constructing the plan $(Q - a_i) \bowtie a_i$, we must construct the optimal plan for $(Q - D_i) \bowtie a_i$, since $D_i$ is the set of literals (i.e., subexpression) in the unfolded query which when replaced by $a_i$ results in an equivalent query. As can be seen, our algorithm does precisely the above.

Correctness and Complexity

The execution space over which the optimal plan for the query $Q$ (with respect to a set $R$ of one-level rewrite rules) is being sought is the set of all left-deep trees over the queries that are obtained from $Q$ by safe substitution with respect to $R$. The optimization problem is to pick an optimal plan from the above execution space with respect to a cost model that respects the principle of optimality.

**Theorem 4.1:** The algorithm ExOptPlan produces the optimal plan with respect to a given MapTable.

Observe that a step which we need to perform efficiently in ExOptPlan($Q$) is to check if deletelist $\subseteq Q$. In order to do so, we use bit maskings to represent the literals in deletelist and the subquery $Q$. Then, the subset relationship can be checked with bit-wise logical operators in $O(1)$ in most cases.

In the absence of any equivalent queries, the time complexity of ExOptPlan is no different from OptPlan, the traditional join enumeration algorithm used by commercial optimizers. Therefore, the interesting complexity question is the dependence of the time complexity of ExOptPlan on the number of safe substitutions in MapTable (say, $l$) for the query $Q$. It can be shown that the time complexity of ExOptPlan is bounded by $O(l^n)$ (when computed in a generous matter). In contrast, the time complexity for OptPlan is $O(n2^{n-1})$. Thus, the worst case complexity degrades by at most $2l/n$. As argued in Section 3.2, the number of safe substitutions ($l$) is likely to be small and so the relative increase in optimization time is very modest.

We can complement the discussion of worst-case complexity with an analysis of subplans derived by the algorithm ExOptPlan. Such a measure makes it possible to identify any possible sources of redundant work done by the optimizer. We have proved the following property of ExOptPlan, which says that ExOptPlan avoids generation of redundant plans. A detailed discussion is given in [CKPS94].

**Theorem 4.2:** For every set of equivalent subqueries of the given query and with respect to a given MapTable, ExOptPlan stores a unique optimal subplan.

Comparison to Other Approaches

The simplest alternative to ExOptPlan is to invoke the optimizer for each equivalent query. This approach turns out to be very inefficient. Let us assume that a query is of size $n$ and the MapTable has $l$ entries. Then, the worst case time complexity of the simple approach is $O(l^n2^{n-1})$, which is significantly worse than the upper bound for ExOptPlan, which is $O(l^n)$. Intuitively, the folly of the naive approach is that no subplans are reused and all shared plans are rederived. The need for sharing plans for the subqueries was observed in [CS93, CR94]. In that approach, optimal plans of the shared subqueries are maintained and reused. However, while this approach maintains a unique optimal subplan for each shared subquery, it does not maintain a unique optimal plan for each set of equivalent subqueries [CKPS94].

4.3 An Experimental Study

Our complexity analysis shows that the increase in optimization cost is modest compared to traditional optimization. To strengthen our confidence, we used our implementation of the optimizer for experimenting, which seems to point to the computational efficiency of our algorithm as well.

Our optimization algorithm was executed on ten queries consisting of seven relations and six equality
joins. Among all relations participating in the query, 50% of relations were chosen and an attribute of each selected relation was assigned to have selection predicate with equality predicate. These attributes for selection condition were chosen among those who did not participate in join predicates. For each query, six views were generated as the same as joins in the query and projection attributes of views were selected so that the materialized views can be used to generate equivalent queries. We tested each query varying the number of materialized views available ranging from 0 to 6. Note that due to the presence of indexes, the decision of using (and selecting) materialized views had to be based on cost estimates. We have used an experimental framework similar to that in [IK90, INSS92, Kan91, Shi50]. The details are in [CKPS94].

The experimental result is shown in Figure 2. The cost of optimization is normalized with respect to the cost of optimizing a single query, as in the traditional optimizer. The effect of saving redundant work by our enumeration algorithm has resulted in a rather slow growth in optimization cost. In particular, for the case where there are 16 equivalent queries, the additional optimization cost on the average was less than 50%.

5 Discussion

In the introduction, we stated that we would like the optimization to be syntax independent and efficient. Let us revisit those desiderata to see whether our optimization algorithm ensures that these requirements are satisfied.

Observe that syntax independent optimization is achieved because we unfold all the queries in terms of base tables to provide a canonical representation of the query. Opportunities for using materialized views are then discovered by enumerating safe substitutions using one-level rules. Furthermore, our enumeration algorithm is also capable of deciding amongst the use of multiple materialized views when their uses are mutually exclusive (i.e., use of one view excludes the use of another).

Example 5.1: Let us assume that the query Q is \{1, 2, 3, 4, 5, 6\}. Let the entries in the MapTable be the following three doublets:

\[(\{1, 2\}, 7), (\{2, 3\}, 8), (\{4, 5\}, 9)\]

Observe that because of the enumeration strategy in ExOptPlan, any candidate plan P that uses the literal 7 (an occurrence of a materialized view), ensures that in the (unfolded) query corresponding to the rest of the plan, subexpression \{1, 2\} is absent, since they occurred in the deletelist of 7 in the MapTable. Therefore, the rest of the plan P can not have an occurrence of the other "overlapping" materialized view 8 since the subexpression for 2 will be missing. On the other hand, the cost of a plan which uses both the views 7 and 9 will be considered. For simplicity, consider a plan where the literal 7 occurs as the last literal to be joined. The remainder of the plan (i.e., excluding the literal 7) represents the subexpression \{3, 4, 5, 6\}. Therefore, while optimizing recursively, the plan for \{3, 6, 9\} will be considered.

Last but not the least, our objective was to ensure that the extensions to the optimizer are simple. A comparison of Figures 2 and Figures 3 confirms that this goal has been met.

Generalizations

The enumeration algorithm ExOptPlan is robust in that it is completely independent of the algorithm used to generate MapTable. This would make it possible to pick an algorithm for generating equivalent queries using other algorithms [Fin82, YL87, CR94] (see discussion in the related work). We have also studied several important extensions to the algorithm to efficiently handle nested views and to avoid applications of one-level rules where unnecessary. These are described in the full paper [CKPS94].

The optimization algorithm presented in this paper extends to the case where the query and the materialized views are single block Select-Project-Join queries (i.e., not necessarily conjunctive queries). For arbitrary SQL queries, the extensions are more complex. For the case where the query and the views are arbitrary first-order queries, enumeration cannot be complete since we have shown in the full paper that given a set of relational views S and two relational queries Q & Q', it is undecidable whether Q \equiv_S Q'.

Finally, note that our algorithm can be used to exploit cached results. Caching results of a query to speed up query processing has been suggested in advanced database management systems such as PostgreSQL. We observe that the cached results of queries differ from materialized views in that there may not be any degree of permanence to the cache. For optimizing an interactive query, it may be profitable to exploit the results that are currently cached. However, as in the case of materialized views, such choices need to be cost-based. Therefore, we maintain a system table
which records the queries (in unfolded form) that are cached and their corresponding one-level rules. This table is updated by the cache manager to reflect the contents of the current cache.

6 Related Work

To the best of our knowledge, no work has previously been done on extending the dynamic-programming based join enumeration algorithm to optimize queries in a cost-based fashion when the database contains one or more materialized views. For example, although Postgres [SJGP90] provides the ability to implement a view either through materialization or by view expansion, the choice between the approaches has to be predetermined. Thus, the optimizer can not explore both the options depending on the query and cost estimations.

The task of generating equivalent queries based on existing query fragments or semantic knowledge has been studied in several different contexts [Fin82, LY85, Sel88, CGM90, CS93, CR94]. However, all these techniques generate equivalent queries explicitly. In contrast, much of our efficiency in optimization stems from the implicit encoding of the set of equivalent queries in MapTable and a join enumeration algorithm that exploits the encoding.

7 Summary

We have presented a comprehensive approach to solving the problem of optimization in the presence of materialized views. Our solution is not only efficient but is also syntax independent and cost-based. Every materialized view corresponds to a one-level rule. The set of equivalent queries due to applications of the above rules are encoded compactly in the MapTable data structure. This data structure is used by the enumeration algorithm to efficiently enumerate the the space of additional execution alternatives generated due to one-level rules (i.e., due to presence of materialized views). Our proposal requires few extensions to the traditional optimization algorithm that is used by commercial systems. Our approach also extends to architectures where results of queries are cached.

References


