# 1

# OPTIMIZING QUERIES WITH MATERIALIZED VIEWS

**Surajit Chaudhuri§,**
**Ravi Krishnamurthy†,**
**Spyros Potamianos‡,**
**Kyuseok Shim***

*§Microsoft Research, Redmond, WA*
*†Hewlett-Packard Laboratories, Palo Alto, CA*
*‡Momferatoy 71, Athens, Greece*
**IBM Almaden Research Center, San Jose, CA*

## ABSTRACT

While much work has addressed the problem of maintaining materialized views, the important problem of optimizing queries in the presence of materialized views has not been resolved. In this paper, we analyze the optimization problem 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 used by commercial database management systems.

## 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, dno, sal, age)` and a department relation `Dept(dno, size, loc)`.

**Example 1.1:** Let `Executive(name, dno, sal)` be a materialized view 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 `dno = 419`. If the relation `Emp` has no index on `dno`, 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).

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:

$$L(\mathbf{x}, \mathbf{y}) \rightarrow V(\mathbf{x})$$

where the variables $\mathbf{x}$ correspond to *projection variables* for the view. The variables $\mathbf{y}$ 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.

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., $=$, $<$, $\geq$) only. Thus, the query has no aggregates or group-by clause. We will use the domain-calculus notation [Ull89] to express conjunctive queries. Generalization of our results for queries that are not necessarily conjunctive are also discussed in this 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.

Our notion of equivalence of queries is as in SQL standard [ISO92], i.e., two queries are *equivalent* if they result in the *same bag of tuples* over every database. However, we need to define the notion of equivalence of queries *with respect to a set of rewrite rules*. In the following definition, we say that a database is 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 3.1:** Two queries $Q$ and $Q'$ are *equivalent* with respect to a set of rewrite rule $\mathcal{R}$ if they result in the same bag of tuples over any valid database for $\mathcal{R}$. ∎

We will denote such an equivalence by $Q \equiv_{\mathcal{R}} Q'$. In case $Q$ and $Q'$ are unconditionally equivalent (i.e, equivalent independent of any rewrite rules), we denote that by $Q \equiv Q'$. This problem of generating equivalent queries in the presence of views has been studied before (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.

Intuitively, we expect to generate an equivalent query 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 straight-forward substitution could be *incorrect*.

**Example 3.2:** Consider Example 1.2. The materialized view $Loc\_Emp$ is represented by the following rule:

$$Emp(name, dno, sal, age), Dept(dno, size, loc) \rightarrow Loc\_Emp(name, size, loc)$$

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

$$\begin{aligned} Q(name) \quad :- \quad & Emp(name, dno, sal, age), age < 35 \\ & Dept(dno, size, \mathtt{SF}) \end{aligned}$$

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

$$Q'(name) :- Loc\_Emp(name, size, \mathtt{SF}), age < 35$$

However, clearly, $Q$ and $Q'$ are not equivalent queries. In particular, $Q'$ is unsafe. ∎

Example 3.2 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 3.2 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 3.3:** The presence of the materialized view $Executive$ is represented by the following one-level rule.

$$Emp(name, dno, sal, age), sal > \texttt{200k} \\ \rightarrow Executive(name, dno, sal)$$

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

$$Q(name) : -Emp(name, dno, sal, age), sal > \texttt{220k} \\ Dept(dno, size, loc), size > \texttt{30}$$

Observe that there is no syntactic substitution for the rule for $Executive$, since there is no renaming such that the literal $sal > \texttt{200k}$ 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 $Q$.

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

∎

Example 3.3 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. Example 3.3 illustrates that 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.

$$L(\mathbf{x}, \mathbf{y}), \mathcal{I}(\mathbf{x}) \rightarrow V(\mathbf{x})$$

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

**Example 3.4:** Consider the one-level rewrite rule for $Executive$ in Example 3.3. We note that $\mathcal{I}(name, dno, sal) \equiv sal > \texttt{200k}$ and $L(name, dno, sal, age) \equiv Emp(name, dno, sal, age)$. Since $\mathcal{I}$ depends only on $sal$, we will abbreviate reference to it as $\mathcal{I}(sal)$. ∎

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 3.5:** 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 $V_r$. ∎

Valid renaming is related to and is derived from *containment mapping* [CM77] (cf. [Ull89]). Specifically, only projection variables may map to constants. Also, no two variables in the rule may map to the same variable in the query unless these two variables are both projection variables. Such renaming results in a query expression that corresponds to a selection over the materialized view. Consider the one-level rule $A(x, y), B(y, z) \rightarrow V(x, z)$. If we map both $x$ and $z$ to $r$, and $y$ to $s$, then we obtain the renamed rule $A(r, s), B(s, r) \rightarrow V(r, r)$. Observe that $V(r, r)$ corresponds to a selection over $V$ that equates two columns of the view $V$.

We will show that if valid renaming of body of a one-level rule matches a set of literals in the query, then those literals may be replaced by the materialized view to obtain an equivalent query. Towards that end, we now define the notions of *safe occurrence* and *safe substitution*. In the following definitions, the symbol $\Rightarrow$ stands for logical implication.

**Definition 3.6:** Given a set of one-level rules $\mathcal{R}$, a query $Q$ has a *safe occurrence* of $\mathcal{R}$, if for a rewrite rule $r \in \mathcal{R}$ there is a valid renaming of the rule $r$ with respect to $Q$ such that the renamed rule has the form $L(\mathbf{x}, \mathbf{y}), \mathcal{I}(\mathbf{x}) \rightarrow V(\mathbf{x})$. Furthermore, the following two conditions must hold:
(1) The query $Q$ has the form:

$$Q(\mathbf{u}) \equiv L(\mathbf{x}, \mathbf{y}), \mathcal{I}'(\mathbf{x}), G(\mathbf{v})$$

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

(2) $\mathcal{I}'(\mathbf{x}) \Rightarrow \mathcal{I}(\mathbf{x})$.

The *safe substitution* corresponding to the above safe occurrence is:

$$Q'(\mathbf{u}) \equiv V(\mathbf{x}), \mathcal{I}'(\mathbf{x}), G(\mathbf{v})$$

∎

Condition (1) ensures that there can be no dangling selection condition (unlike Example 3.2) when the view replaces its matching subexpression in the query. Valid renaming plays an important role in the above definition, making it possible to identify not only subexpressions in the query that match the materialized view, but also subexpressions that match selections over the materialized view. Such selections can be equality to a constant ($x = c$) or equality of column values. For example, assume $A(x, y), B(y, z) \rightarrow V(x, z)$ and the given query is $A(i, j), B(j, i)$ then $V(i, i)$ will be a safe substitution, which corresponds to a selection of the latter kind over the materialized view $V$.

Testing condition (2) 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). In reality, we retain only a subset of inequality constraints in $\mathcal{I}'$, i.e., those constraints that are not subsumed by $\mathcal{I}$. It is possible to have safe substitutions even if there is no syntactic match between left-hand side of a rule and the query (as in Example 3.3)

**Example 3.7:** Let us revisit Example 3.3 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 $\mathcal{I}'(sal) \equiv sal > 220\mathtt{k}$ and $L(name, dno, sal, age) \equiv Emp(name, dno, sal, age)$. Furthermore, the following is true:

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

From Example 3.4, we note that Since $\mathcal{I}'(sal) \Rightarrow \mathcal{I}(sal)$. Hence, the condition for safe occurrence is satisfied and we obtain the following equivalent query

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

∎

The following lemma states that queries obtained by safe substitution are equivalent to the original query over any database that stores the materialized view, consistent with its view definition. The lemma is true not only for queries with bag semantics, but also for queries with set semantics (Select Distinct).

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

**Proof:** We sketch the proof that every safe substitution results in an equivalent query. We will use the notation of Definition 3.6. Observe that $\forall \mathbf{x}(L(\mathbf{x}, \mathbf{y}) \wedge \mathcal{I}'(\mathbf{x}) \equiv V(\mathbf{x}) \wedge \mathcal{I}'(\mathbf{x})$. The equivalence is true for bag equivalence as well since $\mathcal{I}'(\mathbf{x})$ acts as a filter and condition (2) holds [CV93]. Since $\mathbf{v}$ and $\mathbf{u}$ in $Q(\mathbf{u})$ are connected to $L(\mathbf{x}, \mathbf{y}) \wedge \mathcal{I}'(\mathbf{x})$ only through $\mathbf{x}$, it follows that $Q(\mathbf{u}) \equiv Q'(\mathbf{u})$.
∎

For queries with set semantics (i.e., Select Distinct), it is not necessary that all equivalent queries are obtained by one or more safe substitutions. For example, if $A(x), B(x) \rightarrow V_1(x)$ and $B(x), C(x) \rightarrow V_2(x)$ are two rewrite rules, then the query $Q(x) :- A(x), B(x), C(x)$ is equivalent to $Q'(x) :- V_1(x), V_2(x)$, but the latter cannot be obtained from $Q(x)$ by a sequence of safe substitutions. For equi-join queries with bag semantics, the converse of Lemma 3.8 is true as well. The proof exploits unique properties of bag equivalence.

**Lemma 3.9:** *Let $Q$ and $Q'$ be conjunctive queries without inequalities. If $Q \equiv_{\mathcal{R}} Q'$ up to isomorphism, then $Q'$ must have been obtained from $Q$ by one or more safe substitutions.*

**Proof:** If there are no occurrences of view symbols in $Q'$, then it follows from [CV93] that $Q'$ and $Q$ are isomorphic. Let us now assume that there is

one view literal $V$ in $Q'$ and the rewrite rule corresponding to $V$ be $l \rightarrow r$. We assume $\sigma$ to be a renaming of the rewrite rule above such that $\sigma(r) = V$ and that all existential variables of $l$ are mapped to variables that do not occur in $Q'$. Let us consider the query $Q''$ that results from replacing the literal $V$ in $Q'$ with the set of literals in $\sigma(l)$. Observe that $Q''$ is a query independent of $V$ and $Q'' \equiv Q$. Therefore, $Q$ and $Q''$ must be isomorphic and let $g$ be the mapping so that $g(Q) = Q''$. It can be seen that $g \circ \sigma$ is a valid renaming of $l \rightarrow r$ that results in the safe substitution $Q'$. The proof extends to the case where there are multiple occurrences of materialized view in $Q''$. ∎

## *Generalizations*

When the given query and the materialized views are arbitrary relational expressions and *not* restricted to be conjunctive queries, it may not be possible to enumerate *all* safe substitutions by any finite computation. The above is a consequence of the undecidability of first-order logic. However, we can extend Definition 3.6 such that generalized safe substitutions for arbitrary SQL queries result in equivalent queries. In the following definition, $L(\mathbf{x}, \mathbf{y})$ and $Q(\mathbf{u})$ are arbitrary SQL expressions.

**Definition 3.10:** Given a set of one-level rules $\mathcal{R}$, a query $Q$ has a *safe occurrence* of $\mathcal{R}$, if for a rewrite rule $r \in \mathcal{R}$ there is a valid renaming of the rule $r$ with respect to $Q$ such that the renamed rule has the form $L(\mathbf{x}, \mathbf{y}) \rightarrow V(\mathbf{x})$. Furthermore, the expression tree for $Q$ has a subexpression $L(\mathbf{x}, \mathbf{y})$ such that it shares only the variables in $\mathbf{x}$ with $\mathbf{u}$ and the rest of the expression for $Q$, i.e., variables among $\mathbf{y}$ do not occur in among $\mathbf{u}$ or the rest of $Q$. Then, a *generalized safe substitution* corresponding to the above safe occurrence is obtained by replacing the subexpression $L(\mathbf{x}, \mathbf{y})$ with $V(\mathbf{x})$. ∎

It can be shown that generalized safe substitutions result in equivalent queries. However, for the rest of the paper, we will continue to focus on conjunctive queries.

We have modeled views using one-level rules where the view is expressed in terms of base tables. Nonrecursive *nested views* can be flattened and are expressed as one-level rules as well. However, we also note that the nested representation of views may be used to identify safe substitutions efficiently. For example, let $V'$ be a nested view defined in terms of another view $V$: $A(x), B(x) \rightarrow V(x)$ and $V(x), C(x) \rightarrow V'(x)$. Observe that if there is no safe

substitution for view $V$ in a query $Q$, the there can be no safe substitution for $V'$ in $Q$ as well. However, since building MapTable is not a bottleneck (see discussion below), such optimization plays a limited role in performance improvement.

## 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 $L(\mathbf{x}, \mathbf{y}) \rightarrow V(\mathbf{x})$ 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 shown in Figure 1. 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.

**Example 3.11:** In addition to the rules for $Executive$ and $Loc\_Emp$, consider the following one-level rewrite rule for $Large\_Dept$

$$Dept(dno, size, loc), size > 30 \rightarrow Large\_Dept(dno, loc)$$

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.

$$Query(name) :- Emp(name, dno, sal, age), size > 30$$
$$Dept(dno, size, \mathtt{SF})$$

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

$$(\{Dept(dno, size, \mathtt{SF}), size > 30\},$$

**Procedure** MakeMapTable($Q, \mathcal{R}$)
**begin**
    Initialize MapTable
    **for** each rewrite rule $r : L \rightarrow V$ in $\mathcal{R}$ **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**

**Figure 1** Algorithm for Creating the MapTable

$$Large\_Dept(dno, \texttt{SF})$$

$$(\{Emp(name, dno, sal, age), Dept(dno, size, \texttt{SF})\}, \\ Loc\_Emp(name, size, \texttt{SF}))$$

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.

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

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

$$(\{Emp(name, dno, sal, age), sal > \texttt{200k}\}, \\ Executive(name, dno, sal))$$

$$(\{Dept(dno, size, loc), size > \texttt{30}\}, \\ Large\_Dept(dno, loc))$$

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*. There is one entry in the literal table for each table variable in SQL. 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 *MakeMapTable* is linear in the number of safe substitutions. The number of safe substitutions depend on the structure of the query. If the query has at most one literal for every table name, there can be at most one safe substitution for every rule. For such queries, a safe substitution can be found in time *linear* in the size of the query. For example, in Example 3.3, the literal *Emp* in the body of the rule can only map to the literal for *Emp* in the body of the query. As in the case of containment mapping, determining safe substitutions can have exponential running time in the size of the query in the worst case. However, our experience with the *Papyrus* project [CS93] at HP Labs indicates that such is rarely the case since queries tend to have few or no repeated predicates (i.e., few or no "self-joins") and are often connected (i.e., no cartisian products). Furthermore, for a given query, only few rules are applicable. Thus, finding safe substitution is a relatively inexpensive step in optimization.

# 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 enumeration and discuss its time complexity. We contrast our enumeration algorithm with known approaches.

## 4.1 Traditional Algorithm

The execution of a query is traditionally represented syntactically as *annotated join trees* 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 subexpression $Q_S$ of $Q$ (referred to as a *subplan*) is constructed *exactly* once and it is stored in the data structure

```
Procedure OptPlan(Q) :
begin
    if existOptimal(Q) then
        return(plantable[Q]);
    bestPlan := a dummy plan with infinite cost
    for each q_i ∈ Q do
        Let S_i = Q − {q_i};
        Temp := OptPlan(S_i);
        p := Plan for Q from Temp and q_i
        if cost(p) < cost(bestPlan) then
            bestPlan := p
    endfor
    plantable[Q] := bestPlan
    return(bestPlan)
end

Procedure ExOptPlan(Q) :
begin
    if existOptimal(Q) then
        return(plantable[Q]);
    bestPlan := a dummy plan with infinite cost
    for each ((D_i , a_i) in MapTable such that D_i ⊆ Q) do
            Let P_i := Q − D_i
            Temp := ExOptPlan(P_i);
            p := Plan for Q from Temp and a_i
            if cost(p) < cost(bestplan) then
                bestPlan := p
    endfor
    plantable[Q] := bestPlan
    return(bestPlan)
end
```

**Figure 2** Traditional and Extended Join Enumeration Algorithm

*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 $O(n2^{n-1})$ (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. The *execution space* over which the optimal plan for the query $Q$ (with respect to a set $\mathcal{R}$ 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 $\mathcal{R}$. 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.

We have presented the enumeration algorithm ExOptPlan in Figure 2. 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. The algorithm ExOptplan 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 2 behave identically 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. Therefore, the following theorem follows:

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

## *Complexity*

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 $O(l2^n)$ (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.

The following theorem establishes the "goodness" of algorithm ExOptPlan, i.e., shows that ExOptPlan avoids generation of redundant plans.

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

**Proof:** Assume that $Q = \{R_1, .., R_n\}$ is the given query over base tables. Observe that every view is equivalent to a join among a subset of relations in $Q$. Therefore, every subquery is equivalent to some subset of relations in $Q$. Hence, it suffices to prove that for every subset of relations on $Q$, a unique plan is stored. As Figure 2 shows, ExOptPlan represents every plan $P_i$ in terms of base relations in $Q$. Thus, a new plan is compared against its stored *bestplan* and the cheaper is retained. Therefore, the theorem follows. ∎

## *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 $O(l^n n2^{n-1})$, which is significantly worse than the upper bound for ExOptPlan, which is $O(l2^n)$. Intuitively, the

shortcoming 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. The following example illustrates this point.

**Example 4.3:** Let us assume that the query $Q$ is represented by the set of literals $\{1,2,3,4\}$ where each integer number represents a literal in the query expression. Let us also assume that the subexpression $\{3,4\}$ can be replaced by the literal $\{5\}$ by application of a one-level rule. Then, we have two equivalent queries: $\{1,2,3,4\}$ and $\{1,2,5\}$. During the optimization, we first build optimal plans for the subqueries in the plan table for $\{1,2,3,4\}$ (e.g., $\{1,2\}$, $\{1,3\}$, $\{1,2,3\}$). Next, we optimize the query $\{1,2,5\}$. During this step, we don't rederive a plan for the entry $\{1,2\}$ since it was generated while optimizing the first query $\{1,2,3,4\}$. However, we do construct the plan for $\{1,5\} \bowtie 2$ as well as $\{1,3,4\} \bowtie 2$ in [CS93, CR94]. The *principle of optimality* tells us that we would have done as well if we constructed only the plan for $cheaper(\{1,5\},\{1,3,4\}) \bowtie 2$. In other words, equivalence of subqueries is not fully exploited. ExOptPlan avoids generating such redundant plans by ensuring the above. ∎

## 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

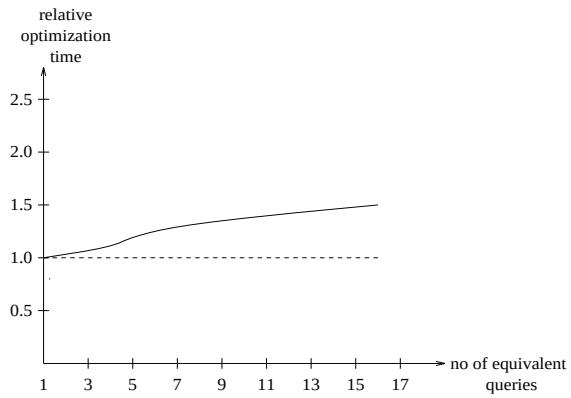


**Figure 3** Relative Cost of Optimization

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, Shi93]. The machine used for the experiments was a DECstation 3100. The queries were tested with a randomly generated relation catalog where relation cardinalities ranged from 1000 to 100000 tuples, and the numbers of unique values in join columns varied from 10% to 100% of the corresponding relation cardinality[1]. Each page of a relation was assumed to contain 32 tuples. Each relation had four attributes, and was clustered on one of them. If a relation was not physically sorted on the clustered attribute, there was a $B^+$-tree or hashing primary index on that attribute. These three alternatives were equally likely. For each of the other attributes, the probability that it had a secondary index was 1/2, and the choice between a $B^+$-tree and hashing secondary index were again uniformly random. As for join methods, we used block nested-loops, merge-scan, and simple and hybrid hash-join [Sha86]. In our experiment, only the cost for number of I/O (page) accesses [IK90, Kan91, CS94] was accounted.

The experimental result is shown in Figure 3. 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

[1] This was the most varied catalog (catalog 'relcat3') that was used in previous experiments [IK90, INSS92, Kan91, Shi93, CS94].

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 OptPlan and ExOptPlan in Figure 2 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 following section).

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). Most commercial optimize multi-block SQL queries by optimizing each block locally. Therefore, we can use ExOptPlan also for multi-block queries.

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 Postgres. 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, YL87, 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


[BC79] P.O. Buneman and E.K. Clemons. Efficiently monitoring relational databases. *ACM TODS*, 4(3):368–382, 1979.

[BLT86] J. A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 61–71, Washington, DC, May 1986.

[CGM90] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, pages 162–207, June 1990.

[CLR90] T. Cormen, C. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[CM77] A. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the 9th Symposium on Theory of Computing*, pages 77–90, New York, August 1977.

[CR94] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In *EDBT 1994*, Cambridge, UK, March 1994.

[CS93] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proceedings of the 19th International VLDB Conference*, pages 529–542, Dublin, Ireland, August 1993.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 20th International VLDB Conference*, Santiago, Chile, Sept 1994.

[CV93] S. Chaudhuri and M.Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of the 13th PODS*, Washington D.C., June 1993.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International VLDB Conference*, pages 577–589, Barcelona, Spain, September 1991.

[Fin82] S. Finkelstein. Common expression analysis in database applications. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 235–245, Orlando, FL, June 1982.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the 1992 ACM-SIGMOD Conference on the Management of Data*, pages 9–18, San Diego, CA, May 1992.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM-SIGMOD*, pages 157–166, Washington D.C., June 1993.

[IK90] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the 1990 ACM-SIGMOD Conference on the Manage ment of Data*, pages 312–321, Atlantic City, NJ, May 1990.

[INSS92] Y. Ioannidis, R. Ng, K. Shim, and T. Sellis. Parametric query optimization. In *Proceedings of the 18th International VLDB Conference*, Vancouver, Canada, August 1992.

[ISO92] ISO. Database language sql. Technical report, Document ISO/IEC 9075:1992, 1992.

[Kan91] Y. Kang. *Randomized Algorithms for Query Optimization.* PhD thesis, University of Wisconsin, Madison, May 1991.

[KP81] S. Koening and R. Paige. A transformation framework for the automatic control of derived data. In *Proceedings of the 7th International VLDB Conference*, pages 306–318, Cannes, France, Sept 1981.

[LY85] P. A. Larson and H. Z. Yang. Computing queries from derived relations. In *Proceedings of the 11th International VLDB Conference*, pages 259–269, Stockholm, August 1985.

[Rou91] N. Roussopoulos. An incremental access method for view cache. *ACM Transactions on Database Systems*, pages 535–563, Sept, 1991.

[SAC+79] 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 *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, pages 23–34, Boston, MA, June 1979.

[Sel88] T. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, pages 175–185, 1988.

[Sha86] L. D. Shapiro. Join processing in database systems with large main memories. *ACM TODS*, 11(3):239–264, September 1986.

[Shi93] K. Shim. *Advanced Query Optimization Techniques for Relational Database Systems*. PhD thesis, University of Maryland, College Park, MD, June 1993. (also available as Technical Report, CS-TR-3086, UMIACS-TR-93-50).

[SI84] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of the ACM SIGMOD Conference on the Management of Data*, pages 240–255, Boston, MA, May 1984.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proceedings of the 1990 ACM-SIGMOD Conference on the Management of Data*, pages 281–290, Atlantic City, NJ, May 1990.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[YL87] H. Z. Yang and P. A. Larson. Query transformation for psj-queries. In *Proceedings of the 13th International VLDB Conference*, pages 245–254, Brighton, August 1987.